

Tema 4.2 - Navegación

Descargar estos apuntes [pdf](#) o [html](#)

Índice

- [Introducción](#)
- ▼ [Conceptos Básicos](#)
 - [Dependencias necesarias](#)
 - [NavController](#)
 - [NavHost](#)
- ▼ [Ejemplo básico de navegación](#)
 - [Implementación mínima](#)
 - [Refactorizando con buenas prácticas](#)
- ▼ [Pasando datos entre pantallas](#)
 - [Estrategias para pasar datos entre pantallas](#)
 - [Paso de argumentos de navegación](#)
- ▼ [Integrando Hilt + ViewModel + Animaciones](#)
 - [Cambiando las transiciones entre pantallas](#)
 - [Integrando los ViewModels en la navegación](#)
- [Integrado navegación y NavigationBar](#)

Introducción

- **Navegación con Jetpack Compose**

- Documentación oficial general tanto con XML como Compose UI: [Principles of navigation](#)
- Documentación oficial compose: [Navigation with Compose](#)
- Codelab Básico Oficial de Google: [Navigation with Compose](#)
- Codelab Avanzado Oficial de Google: [Navigation with Compose](#)
- Mejores prácticas de uso de la Navegación en Jetpack Compose (Vídeo Inglés) [Android Developers](#)
- Navegación en Jetpack Compose (Lista de Reproducción): [Stevdza-San](#)
- Compartir datos entre pantallas (Vídeo Inglés): [Philipp Lackner](#)
- Navegación en Jetpack Compose (Vídeo Castellano): [Brais Moure](#)

La necesidad de navegar entre pantallas es una de las características más importantes de cualquier aplicación. Android Jetpack Compose proporciona una biblioteca de navegación que le permite navegar entre pantallas, pasar datos, administrar el historial de navegación y mucho más.

Esta biblioteca, **inicialmente** se implementó para las aplicaciones de Android basadas en vistas XML (Activities y Fragments), incluso dispone de una herramienta visual para diseñar la navegación de la aplicación.

Ahora, la biblioteca de navegación también se puede usar con aplicaciones de **Android Jetpack Compose**. La biblioteca de navegación de Compose proporciona una API declarativa para definir la navegación, y una API de navegación para navegar entre pantallas. Pero sin embargo no dispone de una herramienta visual para diseñar la navegación de la aplicación.

Como la mayoría de clases y el sistema de navegación es común a las aplicaciones basadas en vistas XML y a las basadas en Compose, **la documentación oficial** de la biblioteca de navegación **es común para ambas**. Por lo que en la documentación oficial, en la mayoría de ejemplos, se muestra la implementación con vistas XML y no con Compose. Esto puede llevarnos a confusión y a pensar que la biblioteca de navegación no es compatible con Compose, pero no es así, simplemente hay que saber interpretar los ejemplos y adaptarlos a Compose. De todas formas, tienes una entrada específica para Compose en este [enlace](#)

Desde sus inicios, la biblioteca de navegación para Compose ha tenido críticas por parte de la comunidad de desarrolladores, debido a su complejidad, falta de documentación y constantes cambios en su API. Es por eso que han surgido *multitud* de **librerías de terceros** que en principio

facilitan la implementación evitando excesivo código "*bolierplate*" o incluso algunas son multiplataforma (Android, iOS, Desktop y Web)

Algunas de estas librerías son: [Compose Destinations](#), [Appyx](#), [Voyager](#), [Reimagined](#), [Decompose](#).

Al final muchas de estas librerías son una capa de abstracción sobre la biblioteca de navegación oficial, por lo que es recomendable conocer la biblioteca oficial. Este [vídeo en inglés](#), Philipp Lackner nos hace una reflexión sobre este tema y el porqué conocer la biblioteca oficial de Google y en todo caso hacernos nosotros mismos nuestra propia abstracción de navegación.

Incluso utilizando la biblioteca oficial, hay muchas formas de implementar la navegación y podrán ver en los diferentes tutoriales y vídeos que hay en la red, que cada sigue diferentes patrones de diseño.

👉 **Importante:** Nosotros en este tema vamos a seguir las recomendaciones de uso y mejores prácticas de implementación que en la actualidad recomienda Google y que pueden encontrar en el siguiente [enlace](#) que **recomendamos que visites y leas** tras ver el tema.

Conceptos Básicos

Dependencias necesarias

Para poder utilizar la biblioteca de navegación en Compose, debemos añadir las siguientes dependencias en el fichero **build.gradle.kts** del módulo **app**:

```
implementation("androidx.navigation:navigation-compose:2.7.6")
```

📌 **Nota:** Recuerda que la versión puede cambiar en el futuro, por lo que es recomendable que consultes la última versión en el [enlace](#). También debes tener en cuenta que un cambio en el primer número de la versión, puede implicar cambios en la API que describiremos a continuación.

NavController

El `NavController` es el **encargado de gestionar la navegación entre destinos**. El controlador **ofrece métodos para navegar** entre destinos, manejar enlaces profundos, **administrar la pila de retroceso** y más.

Para crear un `NavController` en Jetpack Compose haremos:

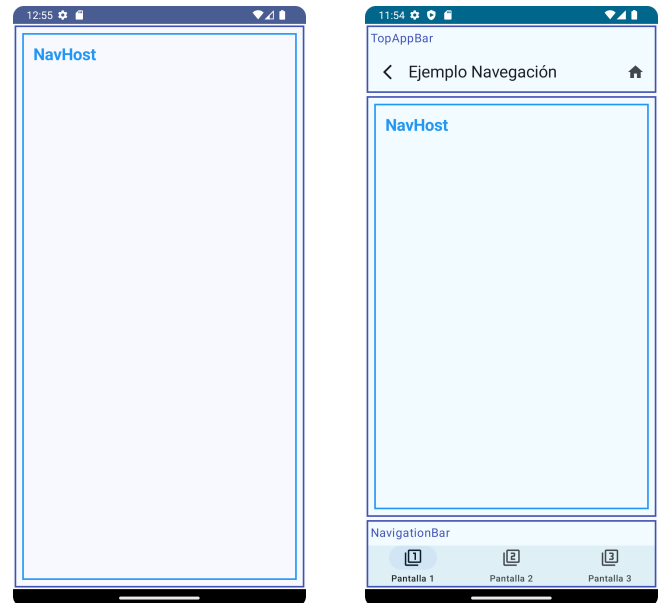
```
val navController = rememberNavController()
```

Puesto que `rememberNavController` es una función `@Composable` NO puede ser creado en un `ViewModel`. Así pues, si quieres que la mayoría de tus componentes puedan acceder a la navegación, debes crear el objeto `NavController` en el componente más alto de la jerarquía a de composición esto es junto al `NavHost` del que hablaremos a continuación.

NavHost

Para poder navegar entre pantallas, debemos tener **un contenedor donde se muestren las pantallas**. En las **aplicaciones basadas compose** este contenedor **es un 'composable'** llamado `NavHost`. Al ser un contenedor, puede ocupar todo la pantalla o solo una parte de ella como se muestra en las imágenes a la derecha. Dentro del mismo.

`NavHost` recibirá un `NavController` y irá definiendo los destinos de navegación a través de un `NavGraph` que o recibe al definirse o creará y gestionará el propio `NavHost` de forma interna si no se le pasa. El `NavGraph` es básicamente por tanto, una colección de destinos recuperables.



Ejemplo básico de navegación

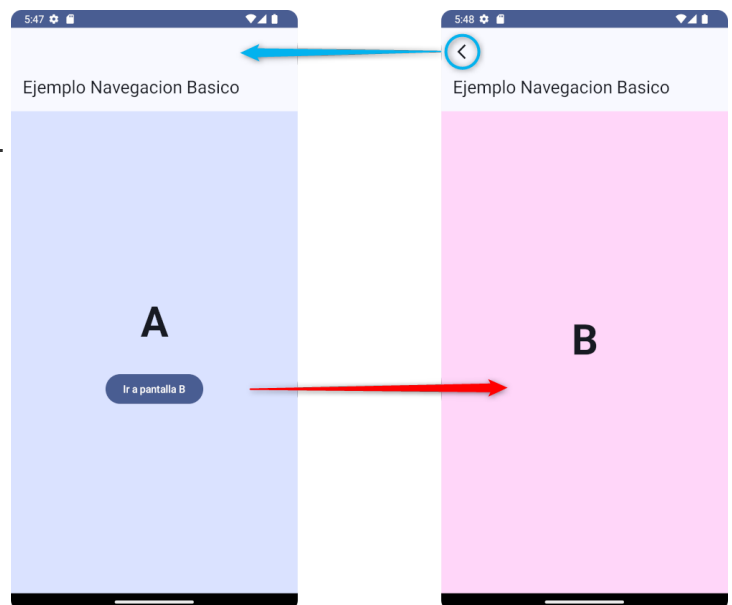
Implementación mínima

Vamos a aplicar los conceptos anteriores a un ejemplo básico de navegación entre dos pantallas **A** y **B**.

Tal y como se muestra en la imagen de ejemplo. La pantalla **A** será la pantalla principal y tendrá un `Scaffold` con un `TopAppBar` y un botón que nos permitirá navegar a la pantalla **B**. La pantalla **B** tendrá un `Scaffold` con un `TopAppBar` y en ella tendrá un `IconButton` de vuelta a la pantalla anterior que solo puede ser pantalla **A**.

Descarga proyecto de **ejemplo 1**:

[EjemploNavegacionBasico1.zip](#)



Los pasos serían los siguientes:

1. Primero definiremos las pantallas en el paquete `ui.features`

Crearemos el fichero `PantallaAScreen.kt` que puedes descargar y cuya función encargada de emitir la pantalla en Compose es la siguiente:

```
@Composable
fun PantallaAScreen(onNavigatePantallaB: () -> Unit)
```

Crearemos el fichero `PantallaBScreen.kt` que puedes descargar y cuya función encargada de emitir la pantalla en Compose es la siguiente:

```
@Composable
fun PantallaBScreen(onNavegarAtras: () -> Unit)
```

👉 **Importante:** Como ves en ambos casos pasamos callbacks que serán llamados cuando se pulse el botón de navegación y así evitamos dependencias con el `NavController` y tendremos centralizadas todas las acciones de navegación en el `NavHost`.

2. Decidimos en que punto vamos atender nuestro **NavHost** y como ya hemos comentado tendremos varias opciones. Para este ejemplo lo normal es que esté lo más alto en la jerarquía de composición, por lo que lo pondremos en el **setContent** de la **MainActivity** y es en este punto donde crearemos al **NavController**. Descargar [MainActivity.kt](#)

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            EjemploNavegacionTheme {
                EjemploNavegacionTheme {
                    Surface(modifier = Modifier.fillMaxSize()) {
8                        val navController = rememberNavController()
9                        NavHostEjemploBasicoInicial(navController = navController)
                    }
                }
            }
        }
    }
}
```

3. Para ello hemos definido la función composable **NavHostEjemploBasico** que recibe una instancia del **NavController**.

```
@Composable
private fun NavHostEjemploBasicoInicial(navController: NavController) {
    NavHost(
        navController = navController,
        startDestination = "pantalla_A"
    ) {
        composable(route = "pantalla_A") { backStackEntry ->
            PantallaAScreen {
                navController.navigate("pantalla_B")
            }
        }
        composable(route = "pantalla_B") { backStackEntry ->
            PantallaBScreen {
                navController.navigateUp()
            }
        }
    }
}
```

Vamos a comentar más detalladamente el código del punto anterior:

```
@Composable
private fun NavHostEjemploBasicoInicial(navController: NavHostController) {
    NavHost(
        navController = navController, // Pasamos el NavController
        startDestination = "pantalla_A" // Indicamos el destino inicial
    ) {
        // En este contexto estamos definiendo un objeto NavGraphBuilder
        // encargado de crear el grafo de navegación.
        // Por lo que solo podemos usar métodos de esta clase.

        ...
    }
}
```

```
@Composable
private fun NavHostEjemploBasicoInicial(navController: NavHostController) {
    NavHost(navController = navController, startDestination = "pantalla_A") {
        ...
        // composable es uno de los métodos de NavGraphBuilder y crea un o
        // destino de navegación en el grafo.
        // Como parámetro obligatorio recibe la ruta del destino que será
        // una cadena de texto similar a una URL de consumo de un API REST.
        // Más adelante comentaremos otros parámetros opcionales.
        composable(route = "pantalla_A") { backStackEntry ->

            // Aquí irá el composable con la pantalla o Screen que se
            // emite en el NavHost en esta ruta del grafo.
            PantallaAScreen {

                // Callback que será llamado cuando se pulse el botón de navegación
                // de ir a pantalla B. Si la ruta especificada no existe en el grafo
                // obtendremos una excepción en tiempo de ejecución.
                navController.navigate("pantalla_B")

            }
            ...
        }
    }
}
```

Por último, al definir el callback de navegación en la pantalla **B** hemos usado el método `navigateUp` que nos permite volver a la pantalla anterior, que en este caso es la pantalla **A**.

Refactorizando con buenas prácticas

En este punto ya tenemos un ejemplo básico de navegación, pero en una aplicación real donde los destinos de navegación pueden ser muchos, podremos pasar parámetros entre pantallas, definir transiciones, usaremos ViewModel para acceder a la lógica de negocio, etc. El código se puede volver muy complejo y poco modularizado. Esto hará que se fácil cometer errores. Además consecuentemente también será difícil de mantener o ampliar, testear y depurar.

Por para solucionar esto, como comentamos en la introducción del tema, vamos a seguir las recomendaciones de uso y mejores prácticas de implementación que en la actualidad recomienda Google y que pueden encontrar en el siguiente [enlace](#). Por lo que vamos a refactorizar el código siguiendo los siguientes pasos:

1. Crearemos un paquete `ui.navigation` donde definiremos los destinos de y el grafo de navegación.
2. Por cada destino de navegación crearemos un fichero `<PantallaDestino>Route.kt` donde definiremos la gestion de la **ruta o rutas** que nos lleven a esa pantalla de destino.

Crearemos primero el fichero `PantallaARoute.kt`:

```
const val PantallaARoute = "pantalla_A"

// Definimos un método de extensión de NavGraphBuilder para poder
// usarlo en el contexto de nuestro NavHost
fun NavGraphBuilder.pantallaAScreen(
    onNavigatePantallaB: () -> Unit
) {
    // En el definimos el destino de navegación sin usar como ruta
    // un literal de cadena. En su lugar usamos la constante.
    composable(
        route = "$PantallaARoute"
    ) { backStackEntry ->
        PantallaAScreen(onNavigatePantallaB)
    }
}
```

Aunque aquí hemos usado una constante de cadena es muy común el uso de enumerados, interfaces o clases selladas para definir las rutas. Por ejemplo:

```
sealed class Rutas(val ruta: String) {
    object PantallaA : Ruta("pantalla_A")
}

// Posteriormente haremos
composable(route = Rutas.PantallaA.ruta)
```


🚩 **Nota:** Nosotros vamos a utilizar constantes de cadena por ser los que recomienda Google en el enlace mencionado.

Ahora crearemos el fichero **PantallaBRoute.kt** donde haremos lo mismo que antes pero también definiremos **un método de extensión de NavController denominado navigateToPantallaB** para poder navegar a esta pantalla desde cualquier punto de la aplicación. Crearíamos más métodos de extensión de este tipo si hubiéramos definido más rutas que naveguen a esta pantalla.

```
const val PantallaBRoute = "pantalla_B"

fun NavController.navigateToPantallaB(navOptions: NavOptions? = null) {
    val ruta = "$PantallaBRoute"
    Log.d("Navegacion", "Navegando a $ruta")
    this.navigate(ruta, navOptions)
}

fun NavGraphBuilder.pantallaBScreen(
    onNavegarAtras: () -> Unit
) {
    composable(
        route = "$PantallaBRoute"
    ) { backStackEntry ->
        PantallaBScreen(onNavegarAtras)
    }
}
```

👉 **Importante:** Si tuviéramos que pasar parámetros a la pantalla destino, tendríamos seguridad de tipos pues, el método extensor de **NavController** recibiría estos parámetros fuertemente tipados y los compondría en la ruta de destino.

3. Por último crearemos el fichero **NavHostEjemploBasico.kt** donde definiremos el *composable* con el **NavHost** usando los métodos de extensión que hemos definido en los ficheros anteriores y que usaremos en el método **setContent** de la **MainActivity**.

```
@Composable
fun NavHostEjemploBasico(navController: NavHostController) {
    NavHost(
        navController = navController,
        startDestination = PantallaARoute
    ) {
        pantallaAScreen { navController.navigateToPantallaB() }
        pantallaBScreen { navController.navigateUp() }
    }
}
```

Como vemos el código queda mucho más limpio y modularizado. Además, no usamos literales de cadena y tendremos una metodología de definición.

Pasando datos entre pantallas

- **Pasando datos entre pantallas**

- Documentación oficial general: [Pass data between destinations](#)
- Documentación oficial compose: [Navigate with arguments](#)
- Vídeo Tutorial (Inglés): [Stevdza-San](#)

En el ejemplo anterior hemos visto como navegar entre pantallas, pero en una aplicación real, es muy probable que necesitemos pasar datos entre pantallas.

Estrategias para pasar datos entre pantallas

👉 **Importante:** En el siguiente [vídeo tutorial](#) de **Philipp Lackner** se explica las diferentes estrategias de cómo pasar datos entre pantallas y las ventajas e inconvenientes de cada una de ellas. Es interesante que le eches un vistazo antes y después de ver este tema.

Der entre las estrategias posibles vamos a destacar las siguientes:

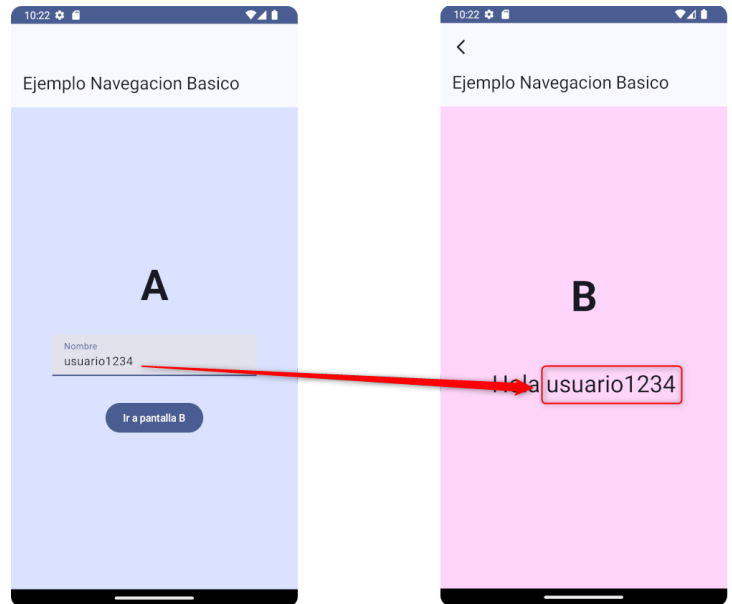
1. **Pasar argumentos de navegación:** La más adecuadas si queremos evitar acoplamiento entre pantallas y de esta manera usar una misma pantalla en diferentes contextos. La librería de navegación de Jetpack Compose para Android nos permite pasar argumentos de navegación entre pantallas.
Estos argumentos **se pasan en la ruta de navegación** soporta el paso de **argumentos de tipo primitivo** y objetos parcelables o serializables.
Lo más común es pasar un argumento de tipo primitivo que sea la **clave primaria** de un objeto que queremos recuperar en el ViewModel de la pantalla destino. Por ejemplo, si queremos recuperar un objeto de tipo `Usuario` en la pantalla destino, pasaremos como argumento de navegación el `id` del usuario y en la pantalla destino recuperaremos el objeto de nuestros repositorios a partir de ese `id`.
2. **Compartir un ViewModel:** Básicamente consiste en tener el dato en un ViewModel que sea compartido por ambas pantallas. Para ello debemos asegurarnos que le `ViewModelOwner` sobreviva a ambas pantallas como por ejemplo el `ViewModelOwner` de la `MainActivity`.
Aunque también hay otras estrategias como usar un `ViewModel` donde el `ViewModelOwner` sea un `NavGraph`.
3. **Compartir una dependencia con estado:** Básicamente consiste en usar Hilt para definir una clase que se pueda **inyectar** en los ViewModels que necesiten compartir datos y esté marcada como `@Singleton`. De esta manera tendremos una única instancia de la clase y por tanto los datos que almacene serán compartidos por todos los ViewModels que la inyecten. Las propiedades de dicha clase deben ser `MutableState` para que los cambios sean observables por los ViewModels que la inyecten.

Paso de argumentos de navegación

Veamos el proceso de paso de argumentos de navegación entre pantallas completando el ejemplo anterior donde navegábamos entre las pantallas **A** y **B**.

Vamos a realizar una **paso simple de datos** entre pantallas. Como puedes ver en la imagen de ejemplo, en la pantalla **A** tendremos un **TextField** donde el usuario introducirá su nombre y en la pantalla **B** mostraremos un **Text** con el nombre introducido.

Descarga proyecto de **ejemplo 2**:
[EjemploNavegacionBasico2.zip](#)



Vamos los pasos lógicos que debemos seguir:

1. Modificamos **PantallaBScreen.kt** para gestionar el argumento de navegación:
PantallaBScreen recibe ahora un parámetro de tipo **String?** que si es distinto de **null** mostrará el nombre un el **Text** justo debajo del rótulo **B**

```
@Composable
fun PantallaBScreen(
  nombre: String? = null,
  onNavegarAtras: () -> Unit)
```

2. Modificamos **PantallaAScreen.kt** para enviar el argumento de navegación:

Fíjate que **onNavigatePantallaB** ahora recibe un parámetro de tipo **String** que será el estado con el nombre introducido por el usuario.

```
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun PantallaAScreen(
    4 onNavigatePantallaB: (String) -> Unit
) {
    val comportamientoAnteScroll = TopAppBarDefaults.pinnedScrollBehavior()
    7 var nombreState by remember { mutableStateOf("anónimo") }
    8 val onNombreChange: (String) -> Unit = { nombreState = it }
    Scaffold(
        content = { innerPadding -> ContenidoPantalla(...) },
    )
}

@Composable
private fun ContenidoPantalla(
    16 nombreState: String,
    onNombreChange: (String) -> Unit,
    18 onNavigatePantallaB: (String) -> Unit,
    modifier: Modifier = Modifier) {
    ...
    Button(onClick = {
    22 onNavigatePantallaB(nombreState)
    }) {
        Text(text = "Ir a pantalla B")
    }
    ...
}
```

3. Modificamos **PantallaBRoute.kt** para definir una nueva entada en nuestro **NavGraphBuilder** con el parámetro de entrada y modificar **navigateToPantallaB**. Veamos el código comentado.

```
const val PantallaBRoute = "pantalla_B"
    2 // Definimos una constante privada para el nombre del parámetro de entrada
private const val NombreParam = "nombre"

fun NavController.navigateToPantallaB(
    6 // Pasamos el parámetro tipado al método de extensión de navegación
    nombre: String,
    navOptions: NavOptions? = null
) {
    10 // Componemos la ruta de navegación con el parámetro de entrada
    val ruta = "$PantallaBRoute/$nombre"
    Log.d("Navegacion", "Navegando a $ruta")
    this.navigate(ruta, navOptions)
}
```

```

fun NavGraphBuilder.pantallaBScreen(
    onNavegarAtras: () -> Unit
) {
    4 // Dejamos la ruta de navegación sin el parámetro de entrada
    composable(
        route = "$PantallaBRoute"
    ) { backStackEntry ->
        PantallaBScreen(
            onNavegarAtras = onNavegarAtras
        )
    }

    13 // Añadimos una nueva ruta de navegación con el parámetro de entrada
    composable(
        15 // fíjate que para indicar que es un parámetro de entrada lo ponemos
        16 // entre llaves como en las rutas de consumo de un API REST
        route = "$PantallaBRoute/{$NombreParam}",
        18 // Indicamos el tipo o tipos, si hubiera más de uno, de los parámetros
        19 // de entrada
        arguments = listOf(
            navArgument(NombreParam) { type = NavType.StringType }
        )
    ) { backStackEntry ->
        24 // Al recuperar el parámetro de entrada, debemos indicar el tipo.
        25 // Al ser arguments nullable, el tipo será String?
        val nombre :String? = backStackEntry.arguments?
            .getString(NombreParam, "anonimo")

        PantallaBScreen(
            29 // Finalmente pasamos el parámetro de entrada a la pantalla
            nombre = nombre,
            onNavegarAtras = onNavegarAtras
        )
    }
}

```

4. Modificamos `NavHostEjemploBasico.kt` para usar el nuevo método de navegación con parámetro en el callback de navegación que pasamos a la pantalla **A**.

```

@Composable
fun NavHostEjemploBasico(navController: NavHostController) {
    NavHost(...) {
        pantallaAScreen(
            onNavigatePantallaB = { nombre ->
                6 navController.navigateToPantallaB(nombre)
            }
        )
        ...
    }
}

```

Integrando Hilt + ViewModel + Animaciones

Si nos fijamos en el ejemplo anterior, al volver de la pantalla **B** a la pantalla **A** el texto introducido en el `TextField` se pierde. Esto es debido a que la pantalla **A** se vuelve a crear y por tanto el estado del `TextField` se reinicia.

Para evitar esto vamos a introducir un `ViewModel` en para cada una de las pantallas. **La forma más simple de hacerlo** es usando `Hilt`. Además, como comentamos también podría ayudarnos a compartir datos entre pantallas. Recuerda que debemos tener la dependencia de `hilt-navigation-compose` en el fichero `build.gradle.kts` del módulo app:

```
implementation("androidx.hilt:hilt-navigation-compose:1.1.0")
```

Otro aspecto que vamos a mejorar es la transición entre pantallas durante la navegación. Para ello, Google ha incorporado en la biblioteca de navegación de Jetpack Compose, un nuevo sistema de animaciones que nos permite definir animaciones de entrada y salida de las pantallas. Recuerda que debes tener añadida la siguiente librería de animaciones en el fichero `build.gradle.kts` del módulo app:

```
implementation("androidx.navigation:navigation-compose:2.7.6")
```

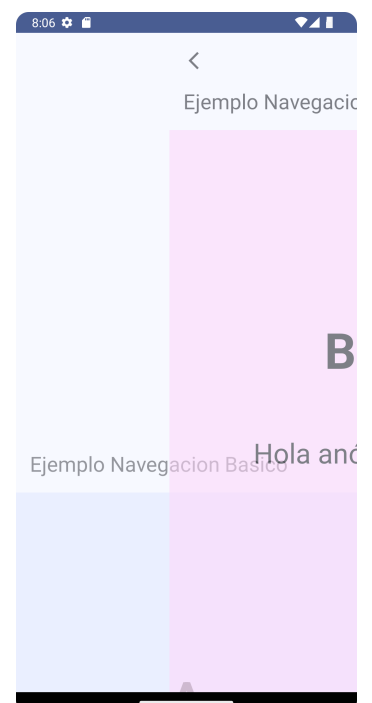
Descarga proyecto de **ejemplo 3:** [EjemploNavegacionBasico3.zip](#)

Cambiando las transiciones entre pantallas

Vamos a continuar con nuestro ejemplo anterior, pero vamos a cambiar las transiciones entre pantallas. Para ello, lo más sencillo aplicar la mismas transiciones a todas nuestras navegaciones y para ello nos hemos inventado la transición en la imagen de ejemplo puedes ver como la pantalla **B** tiene una transición de entrada de derecha a izquierda con un fade-in y la pantalla **A** una transición de salida con de arriba a abajo con un fade-out.

Cuando hagamos un volver atrás con un `popBackStack` las direcciones de las transiciones se invertirán.

Puedes encontrar más información sobre las transiciones de navegación en la [documentación de Accompanist](#). Aunque como se indica en ella ya está incluida en la biblioteca de navegación de Jetpack Compose a partir de la versión `2.7.0`



Podemos crear una **biblioteca de funciones de extensión de**

AnimatedContentTransitionScope<NavBackStackEntry> para definir las transiciones de navegación. Para ello siguiendo la documentación de Accompanist hemos definido las siguientes. Podríamos, por ejemplo, llevarlas a algún tipo de fichero en el paquete **utilities** de nuestro proyecto.

```
fun AnimatedContentTransitionScope<NavBackStackEntry>
    .entradaDesdeDerecha(duracion: Int) =
    fadeIn(animationSpec = tween(duracion)) +
    slideIntoContainer(
        AnimatedContentTransitionScope.SlideDirection.Left,
        tween(duracion)
    )
fun AnimatedContentTransitionScope<NavBackStackEntry>
    .salidaAlaDerecha(duracion: Int) =
    fadeOut(animationSpec = tween(duracion)) +
    slideOutOfContainer(
        AnimatedContentTransitionScope.SlideDirection.Right,
        tween(duracion)
    )
fun AnimatedContentTransitionScope<NavBackStackEntry>
    .salidaHaciaAbajo(duracion: Int) =
    fadeOut(animationSpec = tween(duracion)) +
    slideOutOfContainer(
        AnimatedContentTransitionScope.SlideDirection.Down,
        tween(duracion)
    )
fun AnimatedContentTransitionScope<NavBackStackEntry>
    .entradaHaciaArriba(duracion: Int) =
    fadeIn(animationSpec = tween(duracion)) +
    slideIntoContainer(
        AnimatedContentTransitionScope.SlideDirection.Up,
        tween(duracion)
    )
```

Posteriormente, para conseguir el efecto descrito, simplemente tendremos que usarlas en los parámetros de las transiciones de navegación en el **NavHost** . Por ejemplo:

```
val duracionAnimacion = 1100
NavHost(
    navController = navController,
    startDestination = PantallaARoute,
    5 enterTransition = { entradaDesdeDerecha(duracionAnimacion) },
    exitTransition = { salidaHaciaAbajo(duracionAnimacion) },
    popEnterTransition = { entradaHaciaArriba(duracionAnimacion) },
    8 popExitTransition = { salidaAlaDerecha(duracionAnimacion) }
)
```

Integrando los ViewModels en la navegación

Como sabemos, normalmente nuestras pantallas van a tener toda los 'UIStates', lógica y acceso a datos en un **ViewModel**. Veamos los pasos para hacerlo en nuestro ejemplo:

1. Integraremos **Hilt** en nuestro proyecto como vimos en temas anteriores. No hace falta definir, de momento, el paquete **di** pues no vamos a preparar ningún módulo.
2. Definiremos el VM de la pantalla **A** en **PantallaAViewModel.kt**. En el ascendemos los estados utilizados en la misma. En nuestro caso el nombre introducido y el callback llamado desde el **TextField**.

```
@HiltViewModel
class PantallaAViewModel @Inject constructor() : ViewModel() {
    var nombreState by mutableStateOf("anónimo")
    private set
    fun onNombreChange(nombre: String) {
        nombreState = nombre
    }
}
```

3. Definiremos el VM de la pantalla **B** en **PantallaBViewModel.kt**. En el ascendemos los estados utilizados en la misma. En nuestro caso el estado con el nombre que vamos a visualizar en el **Text** de la pantalla **B**.

```
@HiltViewModel
class PantallaBViewModel @Inject constructor() : ViewModel() {
    var nombreState: String = ""
}
```

Fíjate que **NO hemos hecho el private set** a la propiedad. De esta manera podremos modificar el estado antes de navegar y no pasar parámetro o pasar el parámetro y al recibirlo 'setearlo' en el VM de **B**. Nosotros vamos a optar por la segunda opción.

4. En **PantallaARoute.kt** vamos a instanciar el VM de **A** y pasar los estados que

PantallaAScreen habrá 'elevado' a su VM.

```
fun NavGraphBuilder.pantallaAScreen(
    onNavigatePantallaB: (String) -> Unit
) {
    composable(
        route = "$PantallaARoute"
    ) { backStackEntry ->
        7      val vmPantallaA = hiltViewModel<PantallaAViewModel>()
              PantallaAScreen(
        9          nombreState = vmPantallaA.nombreState,
        10         onNombreChange = vmPantallaA::onNombreChange,
              onNavigatePantallaB = onNavigatePantallaB
            )
        }
    }
}
```

👉 **Importante:** Fíjate que hemos creado el justo antes de emitir la pantalla. En este punto Hilt hará que el **ViewModelOwner** es el **NavGraph** al que pertenezca el destino de navegación. Puesto que el **NavGraph** pertenece al **NavHost** y este no cambia al navegar, el VM no será destruido.

En este caso, equivale a tener un **ViewModel** con **ViewModelOwner** a la **MainActivity** por si hubiéramos creado el VM en esta y lo hubiéramos ido pasando por parámetro por ej...

```
...
Surface(modifier = Modifier.fillMaxSize()) {
    val navController = rememberNavController()
    4    val vmPantallaA = hiltViewModel<PantallaAViewModel>()
        NavHostEjemploBasico(
            navController = navController,
    7    vmPantallaA = vmPantallaA
        )
    }
    ...
```

Solo si usamos **grafos de navegación anidados** y el VM está asociado a un grafo anidado. El VM se destruirá al salir del grafo anidado en la navegación. Nosotros en este curso no vamos a tratar este tema por simplificar, pero puedes ver en el enlace anterior como se hace.

5. En **PantallaBRoute.kt** vamos a instanciar el VM de **B**, 'setear' el nombre recibido como parámetro en el mismo y pasar este a **PantallaBScreen** que lo habrá 'elevado'.

```
...
composable(
    route = "$PantallaBRoute/{$NombreParam}",
    arguments = listOf(
        navArgument(NombreParam) {
            type = NavType.StringType
        }
    )
) { backStackEntry ->
    val nombre :String? = backStackEntry.arguments?.getString(
        NombreParam, "anonimo"
    )
13    val vmPantallaB = hiltViewModel<PantallaBViewModel>()
14    vmPantallaB.nombreState = nombre ?: "anonimo"
    PantallaBScreen(
16        nombre = vmPantallaB.nombreState,
        onNavegarAtras = onNavegarAtras
    )
}
...
```

Otra opción para este último caso **sugerida en la documentación de Jetpack Compose**.

```
@HiltViewModel
class PantallaBViewModel @Inject constructor(
    savedStateHandle: SavedStateHandle
) : ViewModel() {
    var nombreState: String = checkNotNull(savedStateHandle[NombreParam])
    private set
}
```

Ahora ya no tenemos un **setter público**, el `nombreState` se recupera del objeto inyectado `SavedStateHandle` y además, queda establecido nada más crear el VM y nos permitiría recuperar algún tipo de objeto complejo del repositorio si fuese su clave primaria.

`savedStateHandle` devuelve ya un State.

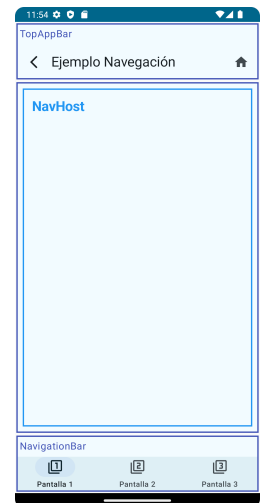
Al hacer esto **PantallaBRoute.kt** quedaría así...

```
composable(
    route = "$PantallaBRoute/{${NombreParam}}",
    arguments = listOf(
        navArgument(NombreParam) {
            type = NavType.StringType
        }
    )
) { backStackEntry ->
    val vmPantallaB = hiltViewModel<PantallaBViewModel>()
    PantallaBScreen(
        nombre = vmPantallaB.nombreState,
        onNavegarAtras = onNavegarAtras
    )
}
```

Fíjate que ya no hace falta hacer `backStackEntry.arguments?.getString` para recuperar el parámetro y '*setearlo*' en el VM.

Integrado navegación y NavigationBar

En el tema anterior ya vimos como definir diferentes componentes gráficos en la jerarquía superior de la pantalla, definidos en Material3 para navegar de una pantalla a otra. Algunos de estos serían: **NavigationBar** , **NavigationDrawer** , **PrimaryTabs** , etc. En este apartado vamos a ver el caso del **NavigationBar** . Para ello, vamos a realizar una aplicación con el esquema mostrado a la derecha.



👉 **Importante:** Si nos fijamos en el mismo, el Scaffold no cambia y será el contenido del mismo el que contenga nuestro **NavHost** que intercambiará diferentes '*pantallas*' que en este caso hemos denominado **Pantalla1**, **Pantalla2** y **Pantalla3**.

Descarga proyecto de **Ejemplo NavigationBar**: [EjemploNavConNavigationBar.zip](#)


Puedes ver un esquema de funcionamiento en la siguiente imagen...



Para simplificar la implementación de la propuesta, no vamos a usar Hilt, ni ViewModel y voy a pasar como parámetro de navegación un entero con el número de la pantalla de donde vengo. Por tanto, siguiendo los pasos descritos a lo largo del tema ...

1. Definiremos los composables con las pantallas **Pantalla1**, **Pantalla2** y **Pantalla3** en el paquete **ui.features**. En nuestro caso, será un único componente pantalla denominado **PantallaScreen.kt** y que tendrá el siguiente interfaz.

```
fun PantallaScreen(  
    pantalla: Int,  
    pantallaDeDondeVengo: Int? = null,  
    onNavigatePantallaAnterior: () -> Unit  
)
```

Si el parámetro **pantallaDeDondeVengo** está a null significa que estoy en la primera pantalla. En caso contrario, se mostrará un tento con la pantalla de donde vengo y un botón de navegación para volver a ella que tendrá el mismo efecto que pulsar el icono de la **TopAppBar**  tal y como se veía en la imagen anterior.

2. En el paquete `ui.navigation` definiremos nuestro componente `NavHost`. Pero antes, definiremos un fuente con la especificación de cada uno de los destinos y cómo navegar a ellos. Por ejemplo, vamos a ver la definición de `Pantalla1Route.kt` ya que `Pantalla2Route.kt` y `Pantalla3Route.kt` serán análogos.

La única diferencia es que en `Pantalla1Route.kt` definiremos **dos rutas** de navegación. **Una sin parámetros de entrada para que pueda ser la pantalla inicial de nuestro `NavHost`** y otra con un parámetro de entrada de tipo entero que será el número de la pantalla de donde vengo.

```
const val Pantalla1Route = "pantalla1"
private const val Pantalla1Parameter1Name = "pantalla_de_donde_vengo"

fun NavController.navigateToPantalla1(
    pantallaAnterior: Int,
    navOptions: NavOptions? = null
) {
    this.navigate("$Pantalla1Route/$pantallaAnterior", navOptions)
}

fun NavGraphBuilder.pantalla1Screen(
    onNavigatePantallaAnterior: () -> Unit
) {
    composable(
        route = "$Pantalla1Route"
    ) { backStackEntry ->
        PantallaScreen(
            pantalla = 1,
            onNavigatePantallaAnterior = onNavigatePantallaAnterior
        )
    }
    composable(
        route = "$Pantalla1Route/{$Pantalla1Parameter1Name}",
        arguments = listOf(
            navArgument(Pantalla1Parameter1Name) {
                type = NavType.IntType
            }
        )
    ) { backStackEntry ->
        val pantallaAnterior :Int? = backStackEntry.arguments?.getInt(
            Pantalla1Parameter1Name, -1
        )
        PantallaScreen(
            pantalla = 1,
            pantallaDeDondeVengo = pantallaAnterior,
            onNavigatePantallaAnterior = onNavigatePantallaAnterior
        )
    }
}
```

3. Definidas los destinos o rutas, ya podemos definir nuestro **NavHost** en **EjemploNavHost.kt** . Este recibirá el **NavHostController** y lo usará en el el callback que pasamos para volver a la pantalla anterior.

```
@Composable
fun EjemploNavHost(navController: NavHostController) {
    NavHost(
        navController = navController,
        startDestination = Pantalla1Route
    ) {
        val onNavigatePantallaAnterior: () -> Unit = {
            navController.navigateUp()
        }
        pantalla1Screen(onNavigatePantallaAnterior = onNavigatePantallaAnterior)
        pantalla2Screen(onNavigatePantallaAnterior = onNavigatePantallaAnterior)
        pantalla3Screen(onNavigatePantallaAnterior = onNavigatePantallaAnterior)
    }
}
```

4. Implementaremos **Scaffold** principal con una **TopAppBar** y la **NavigationBar** en el fuente **EjemploNavDentroDeUnScaffold.kt** tal y como vimos en el tema anterior. Este será emitido desde el **setContent** de la **MainActivity** aunque también podremos tener un **@Preview** y su contenido principal será el **NavHost** .

Nuestro **NavigationBar** solo recibe el estado con el índice de la pantalla actual y el callback para navegar a la pantalla seleccionada.

```
@Composable
fun NavigationBarEnEjemploNav(
    iOpcionNevagacionSeleccionada: Int = 0,
    onNavigateToScreen: (Int) -> Unit
) {
    val titlesAndIcons = remember { ... }
    NavigationBar {
        titlesAndIcons.forEachIndexed { index, (title, icon) ->
            NavigationBarItem(
                icon = { Icon(icon, contentDescription = title) },
                label = { Text(title) },
                selected = iOpcionNevagacionSeleccionada == index,
                onClick = { onNavigateToScreen(index) }
            )
        }
    }
}
```

El contenido principal lo metemos dentro de un Box para poder aplicar el padding recibido desde el Scaffold.

```
@Composable
fun ContenidoPrincipalEnEjemploNav(
    navController: NavHostController,
    modifier: Modifier = Modifier
) {
    Box(modifier = modifier) { EjemploNavHost(navController)}
}
```

Definimos una función de utilidad privada que a partir de la ruta actual del **NavController** nos devuelve el índice de la pantalla actual.

```
private fun iOpcionNevagacionSeleccionadaAPartirDeRuta(ruta: String?): Int {
    return when (ruta?.substringBefore("/")) {
        Pantalla1Route -> 0
        Pantalla2Route -> 1
        Pantalla3Route -> 2
        else -> 0
    }
}
```

Vamos por último a definir la función **EjemploNavDentroDeUnScaffold** que como hemos comentado será llamada desde la MainActivity. Veamos sus diferentes partes:

👉 **Importante:** Cuando pulsemos el botón de volver atrás. **¿Cómo se actualiza el estado de la NavigationBar ?**. Fíjate que obtenemos un **State** con **navController.currentBackStackEntryAsState()** que se actualizará cada vez que se produzca un cambio en el destino de navegación. De él, obtenemos un estado derivado (**derivedStateOf**) con el índice de nuestro **NavigationBar** utilizando la función de utilidad que hemos definido anteriormente y a la que le pasamos la ruta actual con **entradaEnPilaDeNavegacionActuasState.value?.destination?.route**.

```
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun EjemploNavDentroDeUnScaffold() {
    val comportamientoAnteScroll
        = TopAppBarDefaults.exitUntilCollapsedScrollBehavior()
    val navController = rememberNavController()
    7 val entradaEnPilaDeNavegacionActuasState
        = navController.currentBackStackEntryAsState()
    var iOpcionNevagacionSeleccionada = derivedStateOf {
        iOpcionNevagacionSeleccionadaAPartirDeRuta(
            12 entradaEnPilaDeNavegacionActuasState.value?.destination?.route
        )
    }
}
```


Definimos ya el componente **Scaffold** y le pasamos al **AppBar** :

1. El **callback** para navegar atrás.
2. El **callback** para deshacer toda la navegación.

👉 **Importante:** para entender la llamada a **popBackStack** es muy importante que leas la [documentación oficial de Android](#) y que por su 'complejidad' vamos a dejar fuera del Tema.

```
Scaffold(  
    modifier = Modifier.nestedScroll(  
        comportamientoAnteScroll.nestedScrollConnection),  
    topBar = {  
        TopAppBarEnEjemploNav(  
            comportamientoAnteScroll = comportamientoAnteScroll,  
            onNavegarAtras = {  
                navController.navigateUp()  
            },  
            onDeshacerNavegacion = {  
                navController.popBackStack(  
                    navController.graph.startDestinationRoute!!, false  
                )  
            })  
        },  
    ),  
)
```

Nuestro **NavigationBar** recibirá del índice seleccionado a partir del estado que actualizaba la navegación. De tal manera que sólo con navegar este estado cambiará. Además, le pasamos los callbacks para navegar a la pantalla seleccionada utilizando las funciones de extensión de **NavController** que hemos definido anteriormente en cada uno de los destinos.

```
bottomBar = {  
    NavigationBarEnEjemploNav(  
        iOpcionNevagacionSeleccionada = iOpcionNevagacionSeleccionada.value,  
        onNavigateToScreen = { nuevo ->  
            when (nuevo) {  
                0 -> navController.navigateToPantalla1(  
                    pantallaAnterior = iOpcionNevagacionSeleccionada.value + 1  
                )  
                1 -> navController.navigateToPantalla2(  
                    pantallaAnterior = iOpcionNevagacionSeleccionada.value + 1  
                )  
                2 -> navController.navigateToPantalla3(  
                    pantallaAnterior = iOpcionNevagacionSeleccionada.value + 1  
                )  
            }  
        }  
    )  
},  
)
```

Por último, emitiremos el componente que definía el **NavHost** en el contenido principal del **Scaffold** .

```
        content = { innerPadding ->
            ContenidoPrincipalEnEjemploNav(
                navController = navController,
                modifier = Modifier.padding(innerPadding)
            )
        }
    )
}
```