

## Creación de flujos de pantalla de usuario

En este capítulo, aprenderá cómo interactúa el sistema Android con su aplicación a través del ciclo de vida de Android, cómo se le notifican los cambios en el estado de su aplicación y cómo puede usar el ciclo de vida de Android para responder a estos cambios.

Comenzará a desarrollar la interfaz de usuario con Jetpack Compose y utilizará algunos de los grupos de diseño y elementos componibles principales para lograrlo.

Aprenderás a crear recorridos de usuario en tu aplicación y a compartir datos entre pantallas. Aprenderás diferentes técnicas para lograr estos objetivos, de modo que puedas usarlas en tus propias aplicaciones y reconocerlas cuando las veas en otras.

También aprenderá cómo funcionan las actividades y los modos de inicio y cómo guardar y restaurar el estado de su actividad, usar registros para informar sobre el flujo de la aplicación y compartir datos entre pantallas.

Al finalizar este capítulo, habrás aprendido los fundamentos del trabajo con actividades y la creación de flujos de pantalla para el usuario. Se te mostrará cómo usar Jetpack Compose para crear una interfaz de usuario (UI) y también habrás aprendido a guardar el estado y gestionar los cambios realizados por la interacción del usuario con tus aplicaciones.

Cubriremos los siguientes temas en el capítulo:

- El ciclo de vida de la actividad
- Guardar y restaurar el estado de la actividad
- Interacción de la actividad con intenciones
- Intenciones, tareas y modos de lanzamiento

## Requisitos técnicos

El código completo de todos los ejercicios y actividades de este capítulo está disponible en GitHub en <https://packt.link/IQOP7>.

## El ciclo de vida de la actividad

En [el Capítulo 1](#), "Creando tu primera aplicación", usamos este `onCreate(savedInstanceState: Bundle?)` método para mostrar la interfaz de usuario componible en nuestra pantalla. Ahora, exploraremos con más detalle cómo el sistema Android interactúa con nuestra aplicación para lograrlo. Al iniciarse una actividad, esta pasa por una serie de pasos para inicializarse, desde la preparación para su visualización hasta su visualización parcial y, finalmente, su visibilidad completa.

También hay pasos que corresponden a la ocultación, el almacenamiento en segundo plano y la posterior destrucción de su aplicación. EstoEl proceso se denomina **ciclo de vida de la actividad**. Para cada uno de estos pasos, hay una **devolución de llamada** que su actividad puede usar para realizar acciones tales como crear y cambiar la pantalla, guardar datos cuando su aplicación se haya puesto en segundo plano y luego restaurar esos datos después de que su aplicación vuelva al primer plano.

Estas devoluciones de llamada se realizan en la actividad principal, y usted decide si necesita implementarlas en su propia actividad para realizar la acción correspondiente. Cada una de estas funciones de devolución de llamada tiene la `override` palabra clave. `override` En Kotlin, la palabra clave significa que esta función proporciona la implementación de una interfaz o un método abstracto; o, en el caso de su actividad, que es una subclase, proporciona la implementación que sobrescribirá a su padre.

Ahora que ya sabes cómo funciona el ciclo de vida de la actividad en general, veamos con más detalle los principios principales de devoluciones de llamadas con las que trabajarán en orden, desde la creación de una actividad hasta la destrucción de la actividad:

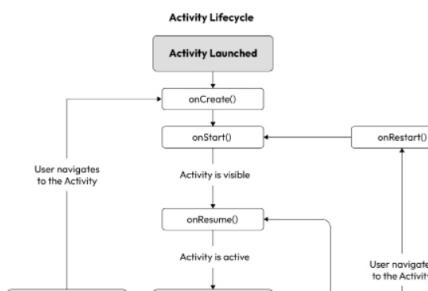
- `override fun onCreate(savedInstanceState: Bundle?)` Esta es la devolución de llamada que se usa con mayor frecuencia en actividades que representan una pantalla completa. En esta etapa, una vez finalizado el método, aún no se muestra al usuario, aunque aparecerá así si no se implementan otras devoluciones de llamada. Normalmente, la interfaz de usuario de la actividad se configura aquí llamando al `setContent{...}` método y realizando la inicialización necesaria.

Este método solo se llama una vez en su ciclo de vida, a menos que la actividad se vuelva a crear. Esto ocurre de forma predeterminada para algunas acciones (como girar el teléfono de vertical a horizontal). El `savedInstanceState` parámetro del `Bundle?` tipo ( `?` significa que el tipo puede ser `null` ) en su forma más simple es un mapa de pares clave-valor optimizado para guardar y restaurar datos.

Lo será `null` si es la primera vez que se ejecuta una actividad después de que se haya iniciado la aplicación, si se crea una actividad por primera vez o si se vuelve a crear una actividad sin guardar ningún estado.

- `override fun onRestart()` Cuando una actividad se reinicia, se llama a este método inmediatamente antes de `onStart()`. Es importante tener clara la diferencia entre reiniciar una actividad y recrearla. Cuando una actividad se pone en segundo plano al pulsar el botón **Inicio**, al volver a primer plano, `onRestart()` se llamará a este método. Recrear una actividad ocurre cuando se produce un cambio de configuración, como la rotación del dispositivo. La actividad finaliza y se vuelve a crear; en ese caso, `onRestart()` no se llamará a este método.
- `override fun onStart()`: Esta es la primera devolución de llamada que se realiza cuando una actividad pasa del segundo plano al primer plano.
- `override fun onRestoreInstanceState(savedInstanceState: Bundle?)`: Si el estado se ha guardado usando `onSaveInstanceState(outState: Bundle?)`, este es el método que el sistema llama después de `onStart()`, donde puede recuperar el `Bundle` estado en lugar de restaurarlo usando `onCreate(savedInstanceState: Bundle?)`.
- `override fun onResume()`: Esta devolución de llamada se ejecuta como la etapa final de la creación de una actividad por primera vez, y también cuando la aplicación ha quedado en segundo plano. Y luego se pone en primer plano. Al finalizar esta devolución de llamada, la actividad está lista para usarse y recibir eventos de usuario.
- `override fun onSaveInstanceState(outState: Bundle?)`: Si desea guardar el estado de una actividad, esta función puede hacerlo automáticamente. Agregue pares clave-valor mediante una de las funciones de conveniencia, según el tipo de dato. Los datos estarán disponibles si su actividad se recrea en [nombre del usuario] `onCreate(savedInstanceState: Bundle?)` y [nombre del usuario] `onRestoreInstanceState(savedInstanceState: Bundle?)`.
- `override fun onPause()`: Esta función se llama cuando una actividad comienza a pasar a segundo plano o cuando otro diálogo o actividad pasa al primer plano.
- `override fun onStop()`: Esta función se llama cuando una actividad está oculta, ya sea porque está en segundo plano o porque se está iniciando otra actividad sobre ella.
- `override fun onDestroy()`: Esto es llamado por el sistema para matar una actividad cuando los recursos del sistema son bajos, cuando `finish()` se llama explícitamente en la actividad o, más comúnmente, cuando el usuario mata una actividad deslizando hacia arriba para cerrar la aplicación desde el menú **Descripción general**.

El flujo de las devoluciones de llamadas/eventos se ilustran en el siguiente diagrama:



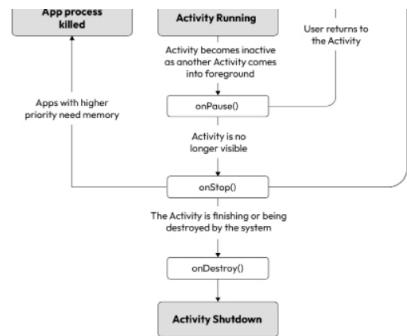


Figura 2.1 – Ciclo de vida de la actividad

Ahora que tuPara comprender qué hacen estas devoluciones de llamadas de ciclo de vida comunes, implementémolas para ver cuándo se llaman.

### Ejercicio 2.01 – Registro de devoluciones de llamadas de actividad

Crea una aplicación llamada `Activity Callbacks` con unActividad vacía. El objetivo de este ejercicio es registrar las devoluciones de llamadas de actividad y el orden en que ocurren para operaciones comunes:

1. Para verificar el orden de las devoluciones de llamada, agreguemos una declaración de registro al final de cada una. Abra `MainActivity` y prepare la actividad para el registro agregando `import android.util.Log` a las `import` declaraciones. Luego, agregue una constante a la clase para identificar la actividad. En Kotlin, las constantes se identifican con la `const` palabra clave y pueden declararse en el nivel superior (fuera de la clase) o en un objeto dentro de la clase.

Las constantes de nivel superior se suelen usar si se requiere que sean públicas. Para las constantes privadas, Kotlin ofrece una forma práctica de añadir funcionalidad estática a las clases mediante la declaración de un `companion` objeto. Añada lo siguiente `onCreate(savedInstanceState: Bundle?)`:

```

companion object {
    private const val TAG = "MainActivity"
}

```

2. Luego, agregue una declaración de registro al final de `onCreate`:

```

Log.d(TAG, "onCreate")

```

Nuestra actividad ahora debería tener el siguiente código:

```

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            ActivityCallbacksTheme {
                Scaffold(
                    modifier = Modifier.fillMaxSize()
                ) {
                    innerPadding ->
                    Greeting(
                        name = "Android",
                        modifier = Modifier
                            .padding(innerPadding)
                    )
                }
            }
        }
        Log.d(TAG, "onCreate")
    }
    companion object {
        private const val TAG = "MainActivity"
    }
}

```

La `d` instrucción en el ejemplo anteriorLa se refiere a *la depuración*. Se pueden usar seis niveles de registro diferentes para generar informe-

mación de mensajes, de menor a mayor importancia: *verbose*, *debug*, *info*, *warn*, *error* y *wtf* what a terrible fault (este último nivel de registro *resalta* una excepción que nunca debería ocurrir). *d i w e wtf*

```
Log.v(TAG, "verbose message")
Log.d(TAG, "debug message")
Log.i(TAG, "info message")
Log.w(TAG, "warning message")
Log.e(TAG, "error message")
Log.wtf(TAG, "what a terrible failure message")
```

3. Ahora veamos Cómo se muestran los registros en Android Studio. Abra la ventana **Logcat**. Puede acceder a ella haciendo clic en el ícono de **Logcat** en la esquina inferior izquierda de la pantalla.



Figura 2.2 – Ícono de Logcat

4. También puede acceder a Logcat yendo a Ver | Ventanas de herramientas | **Logcat**.
5. Ejecute la aplicación en el dispositivo virtual y examine la salida de la ventana **Logcat**. Debería ver la declaración de registro que agregó con el formato que se muestra en la Figura 2.3 :

```
2025-05-05 17:39:55.647 12378-12378 GraphicsEnvironment com.example.activitycallbacks V com.example.activitycallbacks is n
2025-05-05 17:39:55.647 12378-12378 GraphicsEnvironment com.example.activitycallbacks V ANGLE allowlist from config:
2025-05-05 17:39:55.647 12378-12378 GraphicsEnvironment com.example.activitycallbacks V com.example.activitycallbacks is n
2025-05-05 17:39:55.648 12378-12378 GraphicsEnvironment com.example.activitycallbacks V Neither updateable production drive
2025-05-05 17:39:55.648 12378-12378 GraphicsEnvironment com.example.activitycallbacks D Compat change id reported: 2796466
2025-05-05 17:39:55.672 12378-12462 DisplayManager com.example.activitycallbacks I Choreographer implicitly registered
2025-05-05 17:39:55.675 12378-12378 tivitycallbacks com.example.activitycallbacks I AssetManager<2>(0xb4000004/f4905c3fb)
2025-05-05 17:39:55.583 12378-12378 ashmem com.example.activitycallbacks E Pinning is deprecated since Android 11. It will be removed in a future release.
2025-05-05 17:39:55.583 12378-12378 CompatChangeReporter com.example.activitycallbacks D Compat change id reported: 3095784
2025-05-05 17:39:55.587 12378-12378 DesktopModeFlags com.example.activitycallbacks D Toggle override initialized to: DV
2025-05-05 17:39:55.615 12378-12378 MainActivity com.example.activitycallbacks D onCreate
2025-05-05 17:39:55.616 12378-12462 EGL_emulation com.example.activitycallbacks I Opening libGLESv1_CM_emulation.so
2025-05-05 17:39:55.616 12378-12462 EGL_emulation com.example.activitycallbacks I Opening libGLESv2_emulation.so
```

Figura 2.3 – Salida del registro en Logcat

6. Las declaraciones de registro pueden ser bastante difíciles de interpretar a primera vista, así que analicemos la siguiente declaración en sus partes separadas:

```
2025-05-05 17:39:55.615 12378-12378 MainActivity com.exam
```

7. Examinemos los elementos de la declaración de registro en detalle:

Campos	Valores
Fecha	2025-05-05
Tiempo	17:39:55.615
Identificador de proceso e identificador de hilo (el ID del proceso de su aplicación y el ID del hilo actual)	12378-12378
Nombre de la clase	MainActivity
Nombre del paquete	com.example.activitycallbacks
Nivel de registro	D (for Debug)
Mensaje de registro	onCreate

Tabla 2.1 – Tabla que explica una declaración de registro

De forma predeterminada, en el filtro de registro (el cuadro de texto sobre la ventana de registro), aparece `package:mine`, que corresponde a los registros de su aplicación. Puede examinar el Visualice los diferentes niveles de registro de todos los procesos del dispositivo cambiando el filtro de registro de `package:mine` a

`level:debug` u otras opciones en el menú desplegable. Si seleccionas `level:verbose`, como su nombre indica, verá una gran cantidad de resultados.

8. Lo bueno de la `tag` opción de la declaración de registro es que le permite filtrar las declaraciones de registro que se informan en la ventana **Logcat** de Android Studio escribiendo `tag` seguido del texto de la etiqueta, `tag>MainActivity` como se muestra en la *Figura 2.4*:

Figura 2.4 – Filtrado de declaraciones de registro por nombre de etiqueta

Si está depurando un problema en su actividad, puede escribir el nombre de la etiqueta y agregar registros a su actividad para ver la secuencia de las declaraciones de registro. Esto es lo que hará a continuación: implementar devoluciones de llamada de la actividad principal y agregar una declaración de registro a cada una para ver cuándo se ejecutan.

9. Coloque el cursor en una nueva línea después de la llave de cierre de la `onCreate(savedInstanceState: Bundle?)` función y luego agregue la `onRestart()` devolución de llamada con una declaración de registro. Asegúrese de llamar a través de [to] `super.onRestart()` para que la funcionalidad existente de la devolución de llamada de actividad funcione correctamente.

```
override fun onRestart() {  
    super.onRestart()  
    Log.d(TAG, "onRestart")  
}
```

En Android Studio, puedes empezar a escribir el nombre de una función y aparecerán opciones de autocompletado con sugerencias de funciones para anular. Como alternativa, si vas al menú superior y seleccionas **Código | Generar | Anular métodos**, puedes seleccionar los métodos que deseas anular.

10. Haga esto para todas las siguientes funciones de devolución de llamada:

```
onCreate(savedInstanceState: Bundle?)  
onRestart()  
onStart()  
onRestoreInstanceState(savedInstanceState: Bundle)  
onResume()  
onPause()  
onStop()  
onSaveInstanceState(outState: Bundle)  
onDestroy()
```

11. Ejecute la aplicación y, una vez que se haya cargado, debería ver las siguientes declaraciones de registro (esta es una versión abreviada):

```
MainActivity onCreate  
MainActivity onStart  
MainActivity onResume
```

12. Presione el botón redondo **Inicio** en el centro de los controles de navegación en el emuladorVentana sobre el dispositivo virtual y el fondo de la aplicación. Debería ver la siguiente salida de Logcat:

```
MainActivity onPause  
MainActivity onStop  
MainActivity onSaveInstanceState
```

13. Ahora, vuelve a poner la aplicación en primer plano pulsando el botón cuadrado de vista general en los controles del emulador y seleccionándola. Deberías ver lo siguiente:

```
MainActivity onRestart  
MainActivity onStart  
MainActivity onResume
```

`onSaveInstanceState(Bundle)` se llama a la función. Esto se debe a que la actividad no se destruyó ni se volvió a crear.

14. Vuelve a pulsar el botón cuadrado de vista general y desliza la imagen de la aplicación hacia arriba para finalizar la actividad. Este es el resultado:

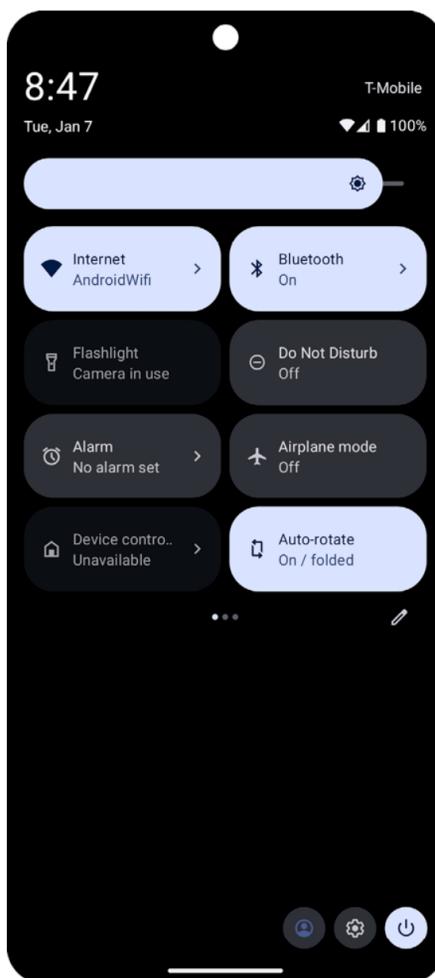
```
MainActivity onPause  
MainActivity onStop  
MainActivity onSaveInstanceState  
MainActivity onDestroy
```

15. Lanza tuVuelva a la aplicación y luego gire el dispositivo usando los íconos de rotación sobre el emulador, como se muestra en *la Figura 2.5*:



*Figura 2.5 – Barra de herramientas del emulador con botones de rotación*

16. Es posible que el teléfono no gire y que la pantalla esté de lado. Si esto ocurre, deslice hacia abajo la barra de estado en la parte superior del dispositivo virtual, busque el botón con un ícono rectangular con flechas llamado "**Rotación automática**" y selecciónelo.



*Figura 2.6 – Barra de configuración rápida con los botones Wi-Fi y Rotación automática seleccionados*

Deberías ver las siguientes devoluciones de llamadas:

```
MainActivity: onPause  
MainActivity: onStop  
MainActivity: onSaveInstanceState  
MainActivity: onDestroy  
MainActivity: onCreate  
MainActivity: onStart
```

```
MainActivity: onRestoreInstanceState  
MainActivity: onResume
```

ConfiguraciónLos cambios, como girar el teléfono, recrean la actividad por defecto. Puedes optar por no gestionar ciertos cambios de configuración en la aplicación, lo que impedirá que se recreé la actividad.

17. Para no recrear la actividad para la rotación, agrégala `android:configChanges="orientation|screenSize|screenLayout"` al `MainActivity` archivo `AndroidManifest.xml`. Inicia la aplicación y gira el teléfono. Estas son las únicas devoluciones de llamada `MainActivity` que verás:

```
D/MainActivity: onCreate  
D/MainActivity: onStart  
D/MainActivity: onResume
```

Los valores " `orientation` y" `screenSize` tienen la misma función para diferentes niveles de la API de Android a la hora de detectar cambios en la orientación de la pantalla. Este `screenLayout` valor detecta otros cambios de diseño que podrían ocurrir en teléfonos plegables.

Estos son algunos de los cambios de configuración que puede gestionar usted mismo (otro común es `keyboardHidden` reaccionar a los cambios en el acceso al teclado). El sistema seguirá notificando estos cambios a la aplicación mediante la siguiente devolución de llamada:

```
override fun onConfigurationChanged(  
    newConfig: Configuration  
) {  
    super.onConfigurationChanged(newConfig)  
    Log.d(TAG, "onConfigurationChanged")  
}
```

Si agrega esta función de devolución de llamada a `MainActivity` (con `import android.content.res.Configuration`) y ha agregado `android:configChanges="orientation|screenSize|screenLayout"` a `MainActivity` en el manifiesto, verá que se llama al rotar. No se recomienda no reiniciar la actividad, ya que el sistema no aplicará recursos alternativos automáticamente. Por lo tanto, rotar un dispositivo de vertical a horizontal no aplicará un diseño horizontal adecuado.

En este ejercicio tienesAprendimos sobre las devoluciones de llamadas de actividad principal y cómo se ejecutan cuando un usuario realiza operaciones comunes con su aplicación a través de la interacción del sistema con `MainActivity`.

En la siguiente sección, cubriremos cómo guardar el estado y restaurarlo, y además veremos más ejemplos de cómo funciona el ciclo de vida de la actividad.

## Guardar y restaurar el estado de la actividad

En esta sección, explorarás cómo se guarda y restaura tu actividad. El estado. Como aprendiste en el *Ejercicio 2.01 (registro de devoluciones de llamadas de actividad , configuración)* Los cambios, como girar el teléfono, hacen que se recreé una actividad.

En estos casos, es importante preservar el estado de la actividad y luego restaurarlo. En los dos siguientes ejercicios, trabajará con un ejemplo para garantizar que los datos se restauren mediante una técnica consolidada para guardar y restaurar el estado de la actividad.

### Ejercicio 2.02 – guardar y restaurar el estado

En este ejercicio, vas a crear una aplicación sencilla que genera un número aleatorio:

1. Crea una aplicación llamada `Save and Restore` con una actividad vacía.
2. Abra el `strings.xml` archivo (ubicado en `app | res | values | strings.xml`) en la vista **de Android** de la ventana **del Proyecto app** o en `src | main | res | values | strings.xml` en la vista **del Proyecto**.
3. Agregue las siguientes cadenas que necesitará para su aplicación:

```
<string name="generate_random_number">
```

```
    Generate Random Number
</string>
<string name="random_number_message">
    Random Number: %d
</string>
```

4. Agregue una propiedad debajo del `class MainActivity : ComponentActivity()` encabezado para establecer el estado del número aleatorio y las importaciones necesarias que necesitará en el ejercicio:

```
import androidx.compose.material3.Button
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.setValue
import androidx.compose.ui.Alignment
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.ui.res.stringResource
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.sp
import kotlin.random.Random
private var randomNumber by mutableStateOf(0)
```

5. Agregue una función a la clase para generar un número aleatorio de 0 a 1000:

```
private fun generateRandomNumber(): Int {
    return Random.nextInt(0, 1000)
}
```

6. ReemplazarEl Greeting componible con código para generary mostrar un número aleatorio:

```
Column(
    verticalArrangement = Arrangement.spacedBy(10.dp),
    horizontalAlignment = Alignment.CenterHorizontally,
    modifier = Modifier
        .padding(innerPadding)
        .fillMaxWidth(),
) {
    Button(
        onClick = {
            randomNumber = generateRandomNumber()
        }
    )
    Text(
        stringResource(
            id = R.string.generate_random_number
        ),
        fontSize = 18.sp
    )
    Text(
        stringResource(
            id = R.string.random_number_message,
            randomNumber
        ),
        fontSize = 18.sp
    )
}
```

La cadena utilizada para mostrar el número aleatorio utiliza un recurso de cadena formateada. Este es unAlternativa de Android al uso de las plantillas de cadena de Kotlin paraVisualizar variables dentro de bloques de texto. La referencia en el `strings.xml` archivo es la siguiente:

```
<string name="random_number_message">
    Random Number: %d
</string>
```

La `%d` instancia es un marcador de posición para un número que se reemplaza cuando se inicializa la cadena pasando el valor que se va a sustituir:

```
stringResource(
    id = R.string.random_number_message,
    randomNumber
```

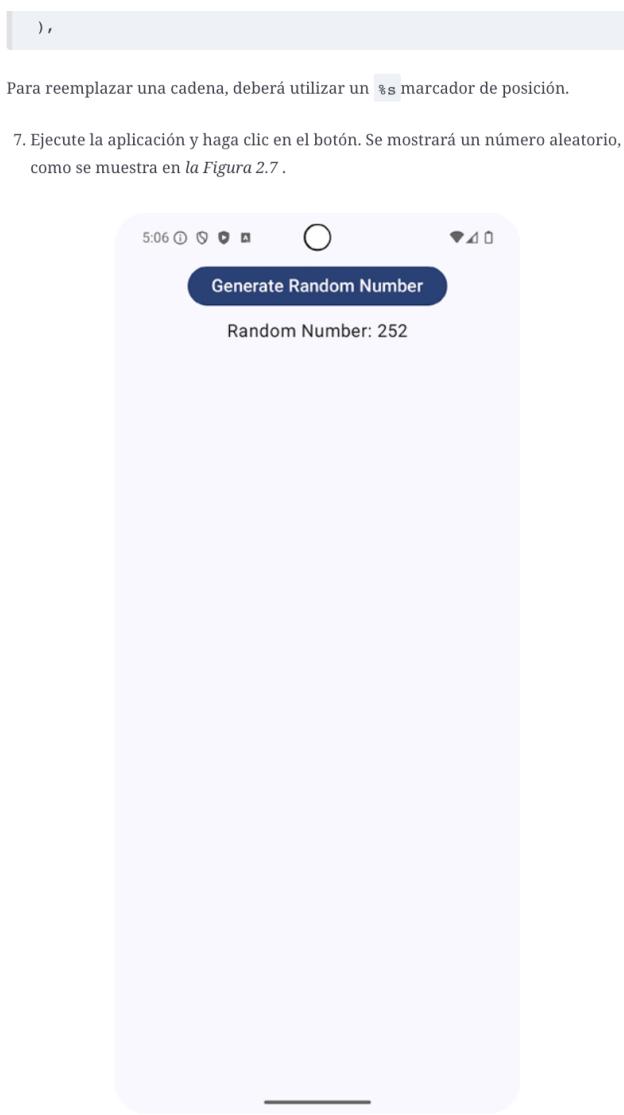


Figura 2.7 – Visualización de la pantalla inicial del generador de números aleatorios

Te darás cuenta que si giras el dispositivo, el número se reinicia a 0. Esto se puede conservar mediante una combinación de `onSaveInstanceState(outState: Bundle)` con `onCreate(savedInstanceState: Bundle?)` o `onRestoreInstanceState(savedInstanceState: Bundle)`.

8. Agregue `onSaveInstanceState(outState: Bundle)` un objeto complementario con una constante:

```
override fun onSaveInstanceState(outState: Bundle) {
    super.onSaveInstanceState(outState)
    outState.putInt(RANDOM_NUMBER, randomNumber)
}
companion object {
    private const val RANDOM_NUMBER = "RANDOM_NUMBER"
}
```

El `Bundle` objeto almacena pares clave-valor. Aquí, se establece el valor del número aleatorio con la clave de la `RANDOM_NUMBER` constante.

9. Añade el recuperación del valor de la constante clavada del `Bundle` objeto sobre el `setContent{...}` bloque:

```
if (savedInstanceState != null) {
    randomNumber =
        savedInstanceState.getInt(RANDOM_NUMBER, 0)
}
```

Con esto en su lugar, si se establecieron claves en `Bundle` cuando `onSaveInstanceState(outState: Bundle)` la actividad estaba finalizando anteriormente, entonces `Bundle` no estarán `null` en `onCreate(savedInstanceState: Bundle)`.

`onCreate(Bundle savedInstanceState: Bundle)`) y los valores se pueden recuperar usando las claves establecidas cuando se vuelve a crear la actividad.

En nuestro caso, usamos la misma `RANDOM_NUMBER` constante para obtener el valor y actualizar el `randomNumber` estado, lo que provoca que el `Text` componente que muestra el `randomNumber` estado se redibuje o recomponga. El segundo argumento de `0` (cero) es el valor predeterminado que se devuelve si no se encuentra la clave. Dispone de dos devoluciones de llamada para recuperar el estado una vez establecido. Si se realiza mucha inicialización en `onCreate(savedInstanceState: Bundle)`, podría ser mejor usar `onRestoreInstanceState(savedInstanceState: Bundle)`. De esta manera, queda claro qué estado se está recreando. Sin embargo, podría ser preferible usar `, onCreate(savedInstanceState: Bundle)` como se hace aquí, si la configuración requerida es mínima.

10. Vuelve a ejecutar la aplicación y gira el dispositivo. La actividad se reiniciará y el número aleatorio se conservará.

Puedes encontrar el código para el ejercicio completo en  
<https://packt.link/LhIjA>.

### Ejercicio 2.03 – guardar y restaurar el estado en Compose

Usar un `Bundle` objeto para establecer pares clave/valor que se pueden guardar y restaurar con devoluciones de llamadas de actividades una técnica bien establecida para mantenerEstado actualizado. Sin embargo, requiere mucho código repetitivo para su configuración. El kit de herramientas de interfaz de usuario de Jetpack Compose presenta técnicas más sencillas para guardar y restaurar el estado mediante `rememberSaveable`. En [el Capítulo 1, "Creando tu primera aplicación"](#), usaste la `remember` función para recordar el estado al actualizarse y recomponerse. Esta función se utiliza para conservar el estado cuando los datos son transitorios y no es necesario restaurarlos al cambiar la configuración. La `rememberSaveable` función difiere de `remember` ya que también puede guardar y restaurar el estado al cambiar la configuración. En el siguiente ejercicio, la usaremos para mantener el estado de los datos de entrada del formulario al cambiar la configuración de la rotación del dispositivo.

Continuaremos usando el [Ejercicio 1.05 \(agregar elementos de UI interactivos para mostrar un saludo personalizado al usuario\)](#), del capítulo anterior, en este ejercicio:

1. Abra el proyecto e introduzca los valores de nombre y apellido. Haga clic en el botón "Intro" y gire el dispositivo. La pantalla quedará en blanco y no se mostrarán valores, como se muestra en la [Figura 2.8](#).

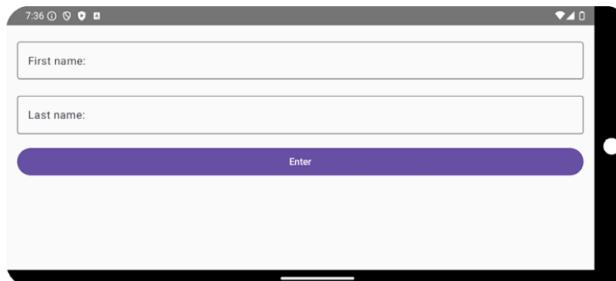


Figura 2.8 – Vista horizontal de un formulario simple con una pantalla vacía

2. Agregarla `rememberSaveable` importación a la lista de importaciones:

```
import androidx.compose.runtime.saveable.rememberSaveable
```

3. Actualice las propiedades `firstName`, `lastName`, y `fullName` para utilizar `rememberSaveable`:

```
var firstName by rememberSaveable { mutableStateOf("") }
var lastName by rememberSaveable { mutableStateOf("") }
var fullName by rememberSaveable { mutableStateOf("") }
```

4. Abra la aplicación de nuevo, complete el formulario y gire el dispositivo. Los

resultados deberían ser similares a los de la Figura 2.9 :

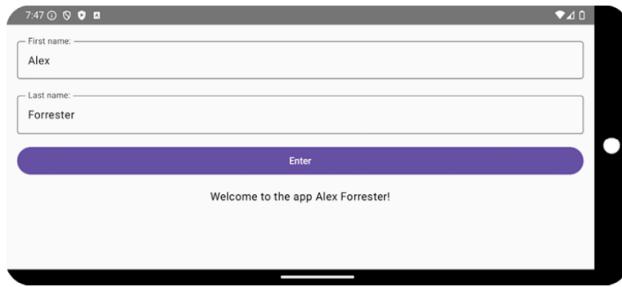


Figura 2.9 – Vista horizontal de un formulario simple con pantalla vacía

El estado se conservó al reiniciar la actividad. Esta única actualización para usar `rememberSaveable` la `MutableState` propiedad ha mantenido el estado correcto. Esto supone una mejora significativa con respecto al esfuerzo requerido para usar el `Bundle` objeto en el *Ejercicio 2.02 (guardar y restaurar el estado)*. De hecho, `rememberSaveable` utiliza el `Bundle` objeto internamente, pero tiene la ventaja de integrarlo con la gestión de estado de Compose. Estas funciones permiten guardar y restaurar datos simples. Android El framework también proporciona `ViewModel`, un componente de la arquitectura de Android que tiene en cuenta el ciclo de vida. Los mecanismos para guardar y restaurar este estado (con `ViewModel`) son gestionados por el framework, por lo que no es necesario gestionarlo explícitamente como se hizo en el ejercicio con el `Bundle` objeto. Aprenderá a usar este componente en [el Capítulo 11, Componentes de la arquitectura de Android](#).

El código completo para este ejercicio se puede encontrar aquí:

<https://packt.link/QKQzu>

En la siguiente sección, agregará otra actividad a una aplicación y navegará entre las dos actividades.

## Interacción de la actividad con intenciones

Una **intención** en Android es un mecanismo de comunicación entre componentes. Dentro de tu propia... aplicación, a vecesQuiere que otra actividad comience cuando ocurre alguna acción en la actividad actual. Especificar exactamenteLa actividad que se iniciará se denomina **intención explícita**. En otras ocasiones, querrá acceder a un componente del sistema, como la cámara. Como no puede acceder a estos componentes directamente, deberá enviar... Una intención que el sistema resuelve para abrir la cámara. Estas se denominan **intenciones implícitas**. Se debe configurar un filtro de intención para que se registre y responda a estos eventos. Consulte el archivo del ejercicio anterior y verá un ejemplo de dos filtros de intención configurados dentro del elemento XML de : `AndroidManifest.xml`

1 <intent-filter> MainActivity

```
<intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category
        android:name="android.intent.category. LAUNCHER"
    />
</intent-filter>
```

La `<action android:name="android.intent.action.MAIN" />` línea indica que este es el punto de entrada principal a la aplicación. Según la categoría establecida, determina qué actividad se inicia primero al iniciar la aplicación. El otro filtro de intención especificado es `<category android:name="android.intent.category. LAUNCHER" />`, que define que la aplicación debe aparecer en el lanzador. Al combinarse, ambos filtros de intención definen que, al iniciar la aplicación desde el lanzador, `MainActivity` se inicie.

En el próximo ejercicio, verá cómo funcionan las intenciones explícitas para navegar por su aplicación.

## Ejercicio 2.04 – Introducción a las intenciones

El objetivo de este ejercicio es crear una aplicación sencilla que utilice intenciones para mostrar texto al usuario en función de su entrada:

1. Crea un nuevo proyecto en Android Studio llamado `Intents Introductio-`

n "Actividad vacía" y selecciona una. Una vez configurado el proyecto, ve a la barra de herramientas y selecciona **"Archivo | Nuevo | Actividad | Galería | Actividad vacía"**. Llámalo `WelcomeActivity` y deja los demás valores predeterminados como están. Se añadirá al `AndroidManifest.xml` archivo y estará listo para usar. El problema ahora que lo has añadido `WelcomeActivity` es saber cómo hacer algo con él. La `MainActivity` actividad comienza al iniciar la aplicación, pero necesitas una forma de iniciarla `WelcomeActivity` y, opcionalmente, pasarle datos, que es cuando se usan las intenciones.

2. Para trabajar con este ejemplo, agregue las siguientes cadenas al `strings.xml` archivo:

```
<string name="header_text">
    Please enter your name and then we'll get started!
</string>
<string name="welcome_text">
    Hello %s, we hope you enjoy using the app!
</string>
<string name="full_name_label">
    Enter your full name:
</string>
<string name="submit_button_text">SUBMIT</string>
```

3. Abre `MainActivity` y crea una `@Composable` función vacía llamada `MainScreen` y otra `@Composable` Función llamada `MainScreenPreview` con una `@Preview` anotación para obtener una vista previa de la pantalla que creará:

```
@Composable
private fun MainScreen() {}
@Preview
@Composable
private fun MainScreenPreview() {
    MainScreen()
}
```

4. Añade una constante llamada `FULL_NAME_KEY` en un objeto complementario. (Un objeto complementario es un objeto especial que pertenece a la propia clase, no a instancias de ella. Es comparable a los métodos y campos estáticos de Java). Usarás la `FULL_NAME_KEY` clave para establecer y recuperar el nombre del usuario. La `MainActivity` clase debería verse así:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            IntentsIntroductionTheme {
                MainScreen()
            }
        }
    }
    companion object {
        const val FULL_NAME_KEY = "FULL_NAME_KEY"
    }
}
```

5. Añade elImportaciones que necesitas para el ejercicio:

```
import android.content.Intent
import android.widget.Toast
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.material3.Button
import androidx.compose.material3.OutlinedTextField
import androidx.compose.runtime.getValue
import androidx.compose.runtime.runtime
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.setValue
import androidx.compose.ui.platform.LocalContext
import androidx.compose.ui.res.stringResource
import androidx.compose.ui.text.TextStyle
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.sp
import com.example.intentsintroduction.MainActivity.Compa
.FULL_NAME_KEY
```

6. A continuación, aéregue el siguiente código al `MainScreen` componible, que

agrega un `OutlinedTextField` campo y un `Button` componente dentro de un `Column` componente:

```
var fullName by remember { mutableStateOf("") }
val context = LocalContext.current
val welcomeIntent =
    Intent(context, WelcomeActivity::class.java)
Scaffold(
    modifier = Modifier.fillMaxSize()
) { innerPadding -> Column(
    verticalArrangement =
        Arrangement.spacedBy(16.dp),
    modifier = Modifier
        .padding(innerPadding)
        .padding(16.dp)
        .fillMaxWidth()
) {
    OutlinedTextField(
        value = fullName,
        onValueChange = { fullName = it },
        label = {
            Text(
                fontSize = 18.sp,
                text = stringResource(
                    id = R.string.full_name_label
                )
            )
        },
        textStyle = TextStyle(fontSize = 20.sp),
        modifier = Modifier
            .fillMaxWidth()
    )
    Button(
        onClick = ({
            if (fullName.isNotEmpty()) {
                welcomeIntent.putExtra(FULL_NAME_KEY,
                    context.startActivity(welcomeIntent)
            } else {
                Toast.makeText(
                    context, context.getString(R.string.error_empty_name),
                    Toast.LENGTH_LONG
                ).show()
            }
        }),
        modifier = Modifier
            .fillMaxWidth()
    ) {
        Text(
            text = stringResource(
                R.string.submit_button_text
            )
        )
    }
}
}
```

7. ElLa aplicación, cuando se ejecuta, se ve como se muestra en la Figura 2.10 :

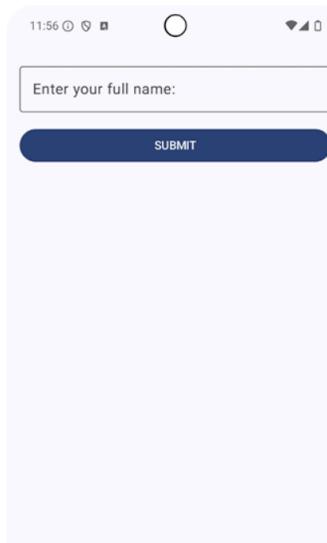


Figura 2.10 – La pantalla de la aplicación después de agregar un campo Outlined-TextField con el nombre completo y un botón ENVIAR

8. Examinemos las propiedades en `MainScreen` y el `onClick` argumento:

```
var fullName by remember { mutableStateOf("") }
val context = LocalContext.current
val welcomeIntent =
    Intent(context, WelcomeActivity::class.java)
onClick = ({
    if (fullName.isNotEmpty()) {
        welcomeIntent.putExtra(FULL_NAME_KEY, fullName)
        context.startActivity(welcomeIntent)
    } else {
        Toast.makeText(
            context, context.getString(R.string.full_name),
            Toast.LENGTH_LONG
        ).show()
    }
}),
```

Se utiliza un contenedor de estado llamado `fullName` para establecer el valor de `OutlinedTextField`, y se accede a un contexto en la `Composable` función con `LocalContext.current`; esto se utiliza para crear una intención explícita de iniciar `WelcomeActivity`. El `fullName` contenedor de estado se verifica para garantizar que el usuario haya completado `Toast`. Esto es; se mostrará un mensaje emergente si está en blanco. Cada intención puede contener un `Bundle` objeto, que es un objeto que puede almacenar pares clave-valor. El proceso de agregar y recuperar datos del `Bundle` objeto utiliza el término "extra", que significa **datos extendidos**, se define como la `FULL_NAME_KEY` clave con el `fullName` valor. El último paso es usar la intención para iniciar `WelcomeActivity`.

9. Ahora, ejecute la aplicación, ingrese su nombre y presione **ENVIAR**, y su pantalla se cargará como se muestra en la *Figura 2.11*:

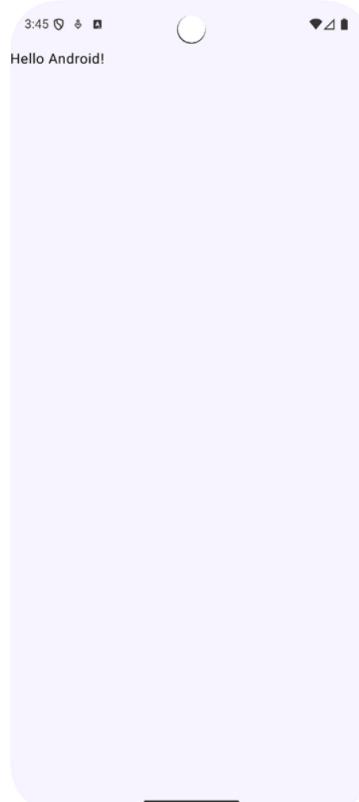


Figura 2.11 – Pantalla en blanco que se muestra cuando no se procesan los datos adicionales de la intención

Bueno, eso no es muy impresionante. Has añadido la lógica para enviar el nombre del usuario, pero no para mostrarlo.

10. A continuación, agregue las importaciones requeridas del *Paso 5* a `WelcomeActivity`
11. A continuación, agregue un `WelcomeScreen` componible con el siguiente contenido y reemplace la `Greeting` función llamando a la `welcomeScreen` función:

```
IntentsIntroductionTheme {  
    WelcomeScreen(intent)  
}  
@Composable private fun WelcomeScreen(intent: Intent) {  
    Scaffold(  
        modifier = Modifier.fillMaxSize()  
    ) { innerPadding -> Box( contentAlignment = Alignment  
        modifier = Modifier  
            .padding(innerPadding)  
            .fillMaxSize()  
    ) {  
        val fullName =  
            intent.getStringExtra(  
                FULL_NAME_KEY  
            ) ?: ""  
        val welcomeText =  
            stringResource(  
                R.string.welcome_text,  
                fullName  
            )  
        Text(  
            textAlign = TextAlign.Center,  
            text = welcomeText,  
            fontSize = 20.sp,  
            fontWeight = FontWeight.Bold,  
            modifier = Modifier.padding(12.dp)  
        )  
    }  
}
```

12. Recuperamos el Intención usada para iniciar `WelcomeActivity` y colocarla en [nombre del `WelcomeScreen` archivo]. Luego, recuperamos el valor de cadena que se pasó desde la intención de la `FULL_NAME_KEY` cadena. Después, formateamos la `<string name="welcome_text">Hello %s, we hope you enjoy using the app!</string>` cadena de recursos obteniendo la cadena de los recursos y pasando el `fullName` valor obtenido desde la intención. Finalmente, esto se establece como el texto de [nombre del archivo `TextView`].
13. Ejecute la aplicación nuevamente y se mostrará un saludo simple, como se muestra en la Figura 2.12 :

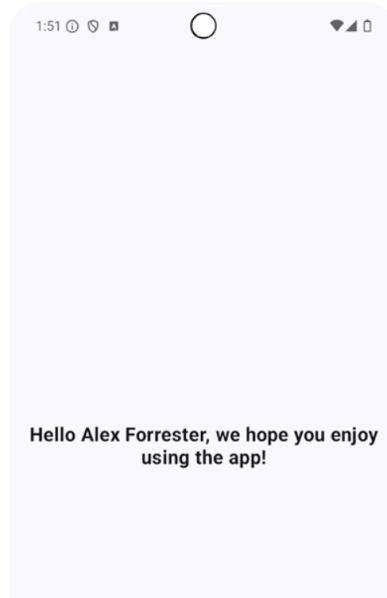




Figura 2.12 – Mensaje de bienvenida al usuario mostrado

Este ejercicio, aunque relativamente simple en términos de diseño e interacción del usuario, demuestra algunos de los conceptos básicosPrincipios de las intenciones. Puedes usarlas para añadir navegación y crear flujos de usuario entre secciones de tu aplicación. En la siguiente sección, verás cómo usar las intenciones para iniciar una actividad y obtener un resultado.

### Ejercicio 2.05 – Recuperar un resultado de una actividad

Para algunos flujos de usuario, solo iniciará unaActividad con el único propósito de obtener un resultado. Este patrón se usa a menudo para solicitar permiso para usar una función específica, abriendo un cuadro de diálogo que pregunta si el usuario da permiso para acceder a los contactos, el calendario, etc., y luego informando el resultado (sí o no) a la actividad que la llama. En este ejercicio, le pedirá al usuario que elija su color favorito del arcoíris y, una vez elegido, mostrará el resultado en la actividad que la llama.

	A partir de ahora, las importaciones de los ejercicios no aparecerán en la lista. Android Studio permite agregar importaciones automáticas habilitando <b>Editor   General   Importación automática   Kotlin   Agregar importaciones inequívocas al instante</b> . También puedes activar importaciones manualmente colocando el cursor sobre el nombre de la clase o función y seleccionando <i>Opción + Intro</i> en Mac y <i>Alt + Intro</i> en Windows/Linux para agregar la importación automáticamente. Si tienes algún problema, puedes consultar el código fuente completo de los ejercicios en el repositorio de GitHub.
	Dos importaciones problemáticas que no se importan automáticamente son las de la <code>by</code> palabra clave. Agregue estas dos importaciones para solucionar este problema:

- `import androidx.compose.runtime.getValue`
- `import androidx.compose.runtime.setValue`

1. Cree un nuevo proyecto `Activity Results` con una actividad vacía y agre-gue las siguientes cadenas al `strings.xml` archivo:

```
<string name="header_text_main">
    Please click the button below to choose your favorite
    color of the rainbow!
</string>
<string name="header_text_picker">Rainbow Colors</string>
<string name="footer_text_picker">
    Click the button above which is your favorite color
    of the rainbow.
</string>
<string name="color_chosen_message">%s is your favorite c
</string>
<string name="submit_button_text">CHOOSE COLOR</string>
<string name="red">RED</string>
<string name="orange">ORANGE</string>
<string name="yellow">YELLOW</string>
<string name="green">GREEN</string>
<string name="blue">BLUE</string>
<string name="indigo">INDIGO</string>
<string name="violet">VIOLET</string>
<string name="title_activity_color_picker">
ColorPickerActivity</string>
```

2. Crear unaNueva actividad llamada `ColorPickerActivity` siguiendo los pa-sos del inicio del *Ejercicio 2.04: una introducción a las intenciones*.
3. Regrese `MainActivity` y agregue un `Text` elemento componible que muestre un color de fondo y el nombre del color de fondo, y un `AppCompat` elemento

el color de fondo y el mensaje de color de fondo, y un `review` componible:

```
@Composable fun TextWithBackgroundColor(backgroundColor: Color, colorMessage: String) {
    Row(
        verticalAlignment = Alignment.CenterVertically,
        modifier = Modifier
            .padding(horizontal = 20.dp)
            .height(50.dp)
            .background(backgroundColor)
            .fillMaxWidth()
    ) {
        Text(
            text = colorMessage,
            fontSize = 22.sp,
            color = Color.White,
            textAlign = TextAlign.Center,
            modifier = Modifier
                .background(backgroundColor)
                .fillMaxWidth()
        )
    }
}
@Preview @Composable fun TextWithBackgroundColorPreview() {
    TextWithBackgroundColor(Color(0xFF00FF00),
        "Chosen color appears here")
}
```

Esto será muestra el panel del color que el usuario ha elegido como su color favorito del arco iris.

4. A continuación, agregue un `MainScreen` elemento componible que agregue un botón para navegar hacia `ColorPickerActivity` y `TextWithBackgroundColor` para la visualización del color elegido:

```
@Composable
fun MainScreen(
    backgroundColor: Color,
    colorMessage: String,
    context: Context,
    startForResult: ActivityResultLauncher<Intent>
) {
    Scaffold(
        modifier = Modifier.fillMaxSize()
    ) { innerPadding ->
        Column(
            verticalArrangement = Arrangement.Top,
            modifier = Modifier
                .fillMaxSize()
                .padding(innerPadding),
        ) {
            Text(
                stringResource(
                    id = R.string.header_text_main
                ),
                fontSize = 18.sp,
                textAlign = TextAlign.Center,
                modifier = Modifier.padding(16.dp)
            )
            Button(
                onClick = {
                    val intent = Intent(
                        context,
                        ColorPickerActivity::class.java
                    )
                    startForResult.launch(intent)
                },
                modifier = Modifier
                    .fillMaxWidth()
                    .padding(20.dp)
            ) {
                Text(
                    text = stringResource(
                        id =
                            R.string.submit_button_text
                    )
                )
            }
            TextWithBackgroundColor(
                backgroundColor, colorMessage
            )
        }
    }
}
```

```
        }
    }
}
```

5. Agregue las siguientes constantes en un objeto complementario: `RAINBOW_COLOR_NAME` para establecer el nombre del color del arco iris en la intención, `RAINBOW_COLOR` para establecer el valor hexadecimal del color y `TRANSPARENT` para establecer un valor predeterminado:

```
companion object {
    const val RAINBOW_COLOR_NAME = "RAINBOW_COLOR_NAME"
    const val RAINBOW_COLOR = "RAINBOW_COLOR"
    const val TRANSPARENT = 0x00FFFFFFL
}
```

6. Agregar `rainbowColor`, `colorName`, y `colorMessage` propiedades debajo del encabezado de clase:

```
private var rainbowColor by
    mutableStateOf(Color(TRANSPARENT))
private var colorName by mutableStateOf("")
private var colorMessage by mutableStateOf("")
```

Estas propiedades son el estado que rige la recomposición de `[número] MainScreen`. Cuando una de estas propiedades cambia, `[número] MainScreen` se recompondrá.

7. Agregue otra propiedad, `startForResult`, que es la parte final de preparación de la actividad para recibir el resultado de `ColorPickerActivity`:

```
private val startForResult =
    registerForActivityResult(
        ActivityResultContracts.StartActivityForResult()
    ) { activityResult ->
    val data = activityResult.data
    rainbowColor = Color(
        data?.getLongExtra(
            RAINBOW_COLOR,
            TRANSPARENT
        ) ?: TRANSPARENT)
    colorName =
        data?.getStringExtra(RAINBOW_COLOR_NAME) ?: ""
    colorMessage = getString(
        R.string.color_chosen_message,
        colorName
    )
    data?.getStringExtra(RAINBOW_COLOR_NAME) ?: ""
}
```

8. Finalmente, actualice `onCreate` para utilizar el `MainScreen` componente componible:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    enableEdgeToEdge()
    setContent {
        val context = LocalContext.current
        ActivityResultsTheme {
            MainScreen(
                rainbowColor,
                colorMessage,
                context,
                startForResult
            )
        }
    }
}
```

La `startForResult` propiedad se utiliza para iniciar la nueva actividad y devolver un resultado. La `activityResult` propiedad devuelta `ColorPickerActivity` contiene la intención en la `activityResult.data` propiedad.

Puede consultar los datos de la intención para obtener los valores esperados. En este ejercicio, queremos obtener el nombre del color de fondo (`colorName`) y su valor hexadecimal (`rainbowColor`) para poder mostrarlo. El `?`  operador comproueba si el valor está definido `null` (es decir, no está definido en la intención) y,

de ser así, el operador Elvis ( `?:` ) define el valor predeterminado. Además, `colorMessage` utiliza el formato de cadena para definir un mensaje, reemplazando el marcador de posición en el valor del recurso con el nombre del color. Si alguno de los datos ha cambiado de valor, `MainScreen`, que depende de él, se recompondrá y el color seleccionado se establecerá como fondo del `Text` campo.

9. Ahora, veamos qué `ColorPickerActivity` hace. Primero, crea un `RainbowColor` elemento componible con una vista previa que agrega un botón con el nombre del color y establece el valor de fondo del color:

```
@Composable
fun RainbowColor(color: Long, colorName: String, onButton
    Button(
        onClick = { onButtonClick(color, colorName) },
        colors = ButtonDefaults.buttonColors(
            containerColor = Color(color), // Background
            contentColor = Color.Black
        ),
        modifier = Modifier
            .padding(horizontal = 20.dp)
            .fillMaxWidth()
    )
{
    Text(
        text = colorName,
        color = Color.White,
        fontSize = 22.sp,
        textAlign = TextAlign.Center,
        fontWeight = FontWeight.Bold
    )
}
@Preview
@Composable
fun RainbowColorPreview() {
    RainbowColor(0xFF00FF00, "GREEN") { color, name -> }
```

10. El `RainbowColor` componente tiene `color`, `colorName` y una `onButton-`  
`click` lambda, que toma los `color` parámetros `colorName` y y los pasa a la  
`onButtonClick` función. Esta es la primera vez que usamos una función de  
orden superior, que puede pasarse a otra función o devolverse desde ella. Po-  
dríamos lograr la misma funcionalidad haciendo referencia directa a la fun-  
ción por su nombre en el `Rainbow` componente, pero esto hace que el compo-  
nente se vincule a otra función y lo hace menos reutilizable, ya que el estado  
debe pasarse al componente. El patrón consiste en elevar el estado que controla la recomposición a un nivel superior en la jerarquía de componentes y luego  
pasarlo hacia abajo como La gestión de parámetros se denomina *elevación de  
estado* y facilita la reutilización de los elementos componibles, ya que no tie-  
nen estado o dependen menos de él. Además, separa el estado de la lógica de la  
interfaz de usuario para facilitar su prueba. Estas y otras ventajas del enfoque  
en patrones...sobre la **separación de preocupaciones (SoC)** entre el estado y  
los componentes de la interfaz de usuario.

11. A continuación, agregue los colores del arco iris en un objeto complementario  
en `ColorPickerActivity`:

```
companion object {
    const val RED = 0xFFFF0000L
    const val ORANGE = 0xFFFFA500L
    const val YELLOW = 0xFFFFEE00L
    const val GREEN = 0xFF00FF00L
    const val BLUE = 0xFF0000FFL
    const val INDIGO = 0xFF4B0082L
    const val VIOLET = 0xFF8A2BE2L
}
```

12. Crea una `setRainbowColor` función, que se pasará `RainbowColor` a través  
del `onButtonClick` parámetro y establecerá la intención con valores de color  
nuevamente a `MainActivity`:

```
private fun setRainbowColor(color: Long, colorName: String
Intent().let { pickedColorIntent ->
    pickedColorIntent.putExtra(RAINBOW_COLOR_NAME, co
    pickedColorIntent.putExtra(RAINBOW_COLOR, color)
    setResult(RESULT_OK, pickedColorIntent)
    finish()
}}
```

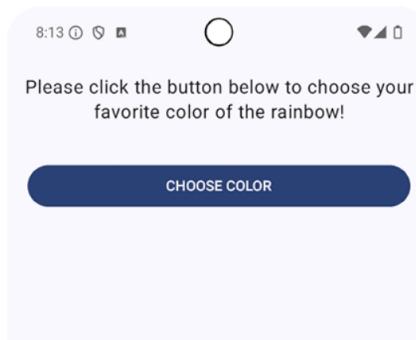
13. El paso final es crear un `ColorPickerScreen` elemento componible en la `ColorPickerActivity` clase que complete el color de fondo de los `RainbowColor` botones y pase una `clickHandler` propiedad que se ejecute al hacer clic en el botón para establecer el valor del color.ser devuelto a `MainActivity` en `setRainbowColor`:

```
@Composable private fun ColorPickerScreen() {
    Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding
        Column(
            verticalArrangement = Arrangement.Top,
            modifier = Modifier
                .fillMaxSize()
                .padding(innerPadding),
        ) {
            val clickHandler = { color: Long, colorName: String ->
                setRainbowColor(color, colorName)
            }
            Text(
                stringResource(id = R.string.header_text),
                textAlign = TextAlign.Center,
                fontSize = 24.sp,
                modifier = Modifier
                    .padding(20.dp)
                    .fillMaxWidth()
            )
            RainbowColor(RED, getString(R.string.red), clickHandler)
            RainbowColor(ORANGE, getString(R.string.orange), clickHandler)
            RainbowColor(YELLOW, getString(R.string.yellow), clickHandler)
            RainbowColor(GREEN, getString(R.string.green), clickHandler)
            RainbowColor(BLUE, getString(R.string.blue), clickHandler)
            RainbowColor(INDIGO, getString(R.string.indigo), clickHandler)
            RainbowColor(VIOLET, getString(R.string.violet), clickHandler)
            Text(
                stringResource(id = R.string.footer_text),
                textAlign = TextAlign.Center,
                fontSize = 18.sp,
                modifier = Modifier
                    .padding(12.dp)
                    .fillMaxWidth()
            )
        }
    }
}
```

14. Luego, actualice `onCreate` para añadiendo un `ColorPickerScreen()` método:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    enableEdgeToEdge()
    setContent {
        ActivityResultsTheme {
            ColorPickerScreen()
        }
    }
}
```

15. Ejecutar ely verla en acción. El diseño debería verse como se muestra en la siguiente captura de pantalla:



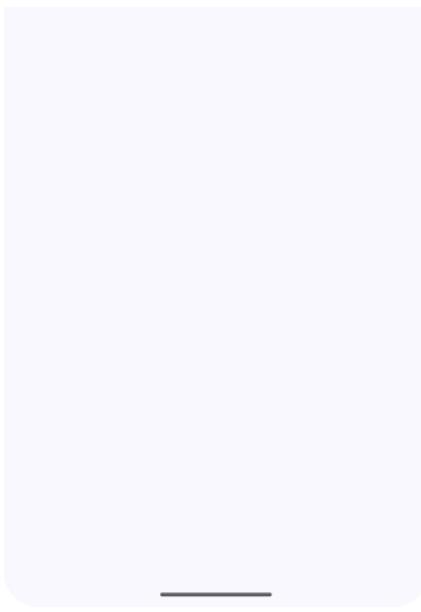


Figura 2.13 – Pantalla inicial de colores del arco iris

16. Presione el botón ELEGIR COLOR , como se muestra en la Figura 2.13 :

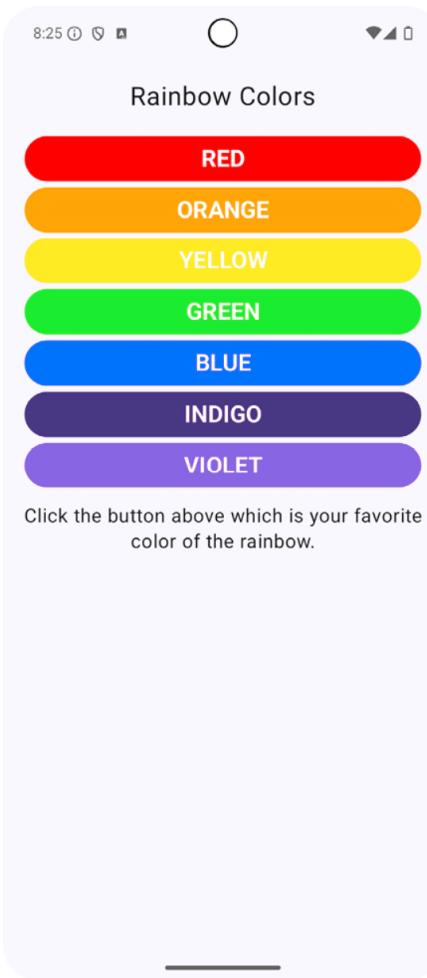


Figura 2.14 – La pantalla de selección de colores del arco iris

17. Si ustedElija AZUL , se mostrará una pantalla con este color, como se muestra en la Figura 2.15 :



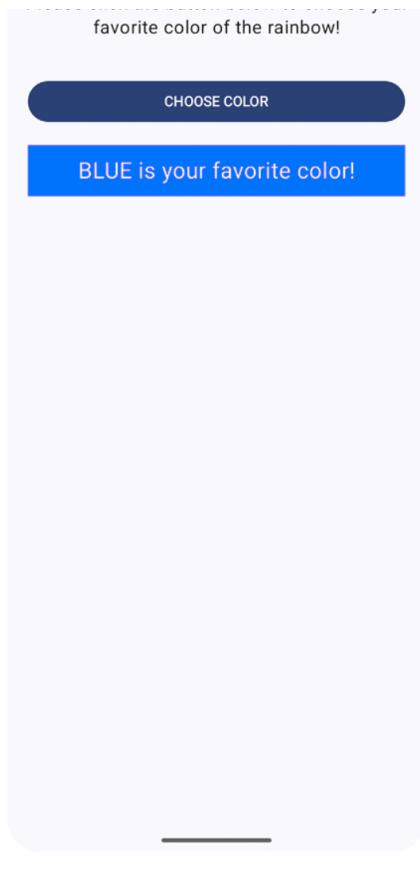


Figura 2.15 – La aplicación que muestra el color seleccionado

El código fuente para este ejercicio se puede encontrar aquí:

<https://packt.link/Et7nn>.

Este ejercicio te presentó otra forma de crear flujos de usuario con `registerForActivityResult`. Esto puede ser muy útil para realizar una tarea específica donde necesitas un resultado antes de continuar con el flujo del usuario a través de la aplicación. A continuación, explorarás los modos de lanzamiento y cómo impactan en el flujo de los recorridos de usuario al crear aplicaciones.

## Intenciones, tareas y modos de lanzamiento

Hasta ahora, has estado usando el comportamiento estándar para crear actividades y pasar de una actividad a otra. Al abrir la aplicación desde el...lanzador con el comportamiento predeterminado, crea su propia tarea y cada actividad que crea se agrega a una pila de actividades, por lo que cuando abre tres actividades una tras otra como parte del recorrido de su usuario, al presionar el botón Atrás tres vecesmoverá al usuario hacia atrás a través de las pantallas/actividades anteriores y luego regresará a la pantalla de inicio del dispositivo, mientras mantiene la aplicación abierta.

El modo de lanzamiento para este tipo de actividad se llama `Standard`; es el predeterminado y no necesita especificarse en el elemento de actividad de `AndroidManifest.xml`. Incluso si se inicia la misma actividad tres veces seguidas, habrá tres instancias de la misma actividad.

Para algunas aplicaciones, es posible que quieras cambiar este comportamiento para que se use la misma instancia. El modo de inicioEsto puede ser útil `singleTop`. Si una `singleTop` actividad es la más reciente, al `singleTop` iniciarse, se utiliza la misma actividad.

Alláson tres otros modos de lanzamiento a tener en cuenta, llamados `singleTask`, `singleInstance` y `singleInstancePerTask`. Estos no son para uso general y solo se usa en escenarios especiales. La documentación detallada de todos los modos de lanzamiento se puede consultar aquí: <https://packt.link/UCY7x>.

Explorarás las diferencias en el comportamiento de los modos de lanzamiento `Standard` y `singleTop` en el próximo ejercicio.

## Ejercicio 2.06 – Configuración del modo de lanzamiento de una actividad

Este ejercicio ilustra el comportamiento de dos de los modos de lanzamiento más comunes. Descarga el código desde aquí: <https://packt.link/MiUDz>. ¡Comencemos!

1. Ábrelo `AndroidManifest.xml` y examínalo. En la `activity` sección del archivo, se han añadido tres actividades:

```
<activity
    android:name=".StandardActivity"
    android:exported="false"
    android:label="@string/title_activity_standard"
    android:launchMode="standard"
    android:theme="@style/Theme.LaunchModes" /> <activity
    android:name=".SingleTopActivity"      android:exported="true"
    android:label="@string/title_activity_single_top"
    android:launchMode="singleTop"
    android:theme="@style/Theme.LaunchModes" /> <activity
    android:name=".MainActivity"         android:exported="true"
    android:label="@string/app_name"
    android:theme="@style/Theme.LaunchModes">
    <intent-filter>
        <action android:name="android.intent.action.MAIN"
            <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

La `android:launchMode` instancia es lo que distingue fundamentalmente entre `SingleTopActivity` y `StandardActivity`.

2. Ejecutar la aplicación, y verá que la **Actividad principal** tiene dos botones para iniciar la actividad estándar y la actividad superior única en la Figura 2.16:

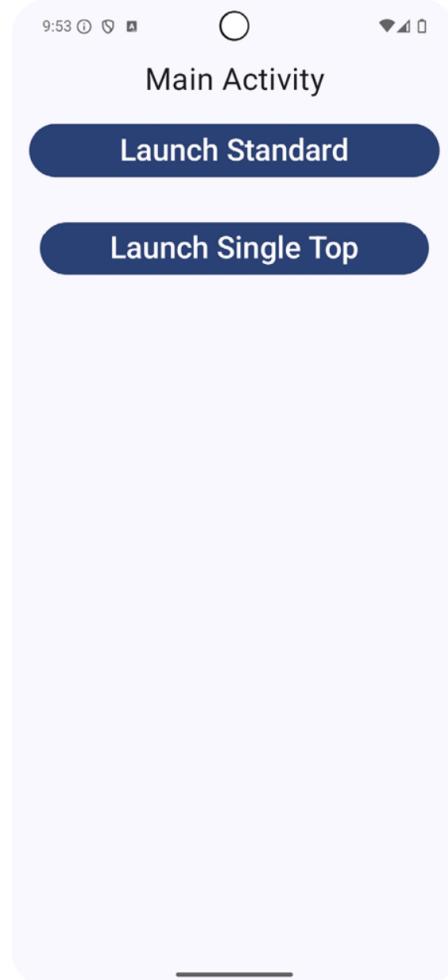
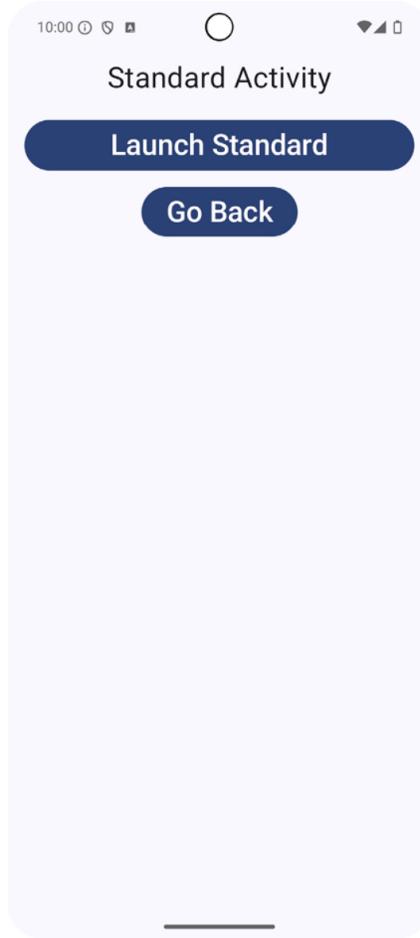


Figura 2.16 – Aplicación que muestra botones para iniciar actividades con diferentes modos de lanzamiento.

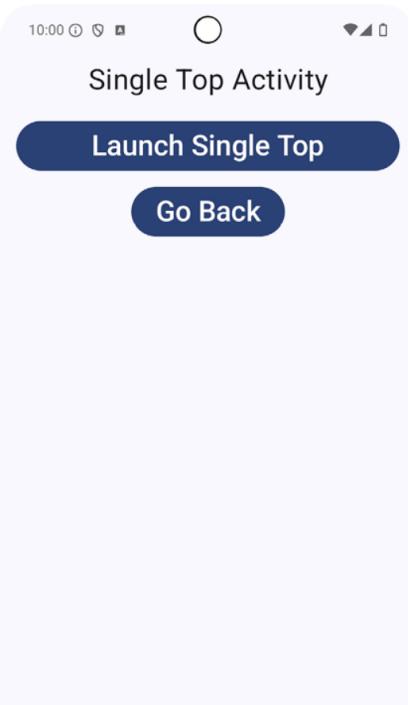
*tes modos de inicio*

3. Hacer clicel botón **Iniciar estándar** y siempre se cargará una nueva actividad:



*Figura 2.17 – Aplicación que muestra una actividad estándar*

4. Haciendo clicAtrás te llevará de regreso a través de la cantidad de actividades que has iniciado antes de regresar a `MainActivity`.
5. Haga clic en el botón **Iniciar Single Top** y siempre se cargará la misma actividad:



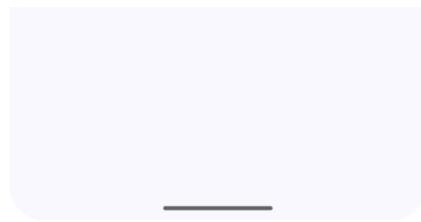


Figura 2.18 – Aplicación que muestra una única actividad principal

Si una sola actividad principal se encuentra en la parte superior de la pila de actividades, no se iniciará una nueva actividad. Esto puede ser útil si no necesitas pasar ningún dato a la actividad como extras en una intención, ya que no tienes que iniciar otra actividad que aumente el uso de memoria de la aplicación.

### Actividad 2.01 y/o Actividad 2.02 – creación de un formulario de inicio de sesión

El objetivo de esteLa actividad consiste en crear un formulario de inicio de sesión con campos de nombre de usuario y contraseña. Una vez enviados los valores de estos campos, se comparan con los valores predefinidos y se muestra un mensaje de bienvenida si coinciden, o un mensaje de error si no, y se redirige al usuario al formulario de inicio de sesión. Los pasos necesarios son los siguientes:

1. Crea un formulario con nombres de usuario y contraseñas `TextField` componibles y un `login` botón.
2. Agregue un `ClickListener` elemento componible al botón para reaccionar ante un evento de pulsación de botón.
3. Validar que los campos del formulario estén completos.
4. Verifique los campos de nombre de usuario y contraseña enviados con los valores codificados.
5. Muestra un mensaje de bienvenida con el nombre de usuario si la verificación es exitosa y oculta el formulario.
6. Muestra un mensaje de error si la verificación no es exitosa y redirige al usuario nuevamente al formulario.

Hay algunasPosibles maneras de intentar completar esta tarea mediante actividades. Aquí tienes dos ideas de enfoques que podrías adoptar:

- Utilice una `standard` actividad para pasar un nombre de usuario y una contraseña a otra actividad y validar las credenciales utilizadas en *la solución 2.01*.
- Úselo `registerForActivityResult` para realizar la validación en otra actividad y luego devolver el resultado utilizado en *la solución 2.02*.



Las soluciones (2.01 y 2.02) de esta actividad se pueden encontrar en <https://packt.link/IQQPZ>.

## Resumen

En este capítulo, cubrimos gran parte de la base de cómo nuestra aplicación interacciona con el framework de Android, desde las devoluciones de llamadas del ciclo de vida de las actividades hasta la retención del estado en nuestras actividades, la navegación entre pantallas y cómo las intenciones y los modos de lanzamiento lo hacen posible, todo ello en el contexto del framework de interfaz de usuario de Jetpack Compose. Estos son conceptos fundamentales que debes comprender para avanzar a temas más avanzados.

En el próximo capítulo, se le presentarán los componentes principales de la interfaz de usuario y los grupos de diseño de Jetpack Compose, y cómo encajan en la arquitectura de su aplicación.

Desbloquea los beneficios exclusivos de este libro ahora

Escanee este código QR o vaya a [packtpub.com/unlock](https://packtpub.com/unlock), luego busque este libro por nombre.

Nota : Tenga a mano la factura de compra antes de



comenzar.



Capítulo anterior

< [Creando tu primera aplicación](#)

Siguiente capítulo

[Desarrollo de la interfaz de us...](#) >



Search collapsed