

# Tema 3.8 - Intents y Contracts

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

## Índice

- [Introducción](#)
- ▼ [Tipos de Intents](#)
  - ▼ [Intents Explícitos](#)
    - [Abriendo otra aplicación desde nuestra aplicación](#)
  - ▼ [Intents Implícitos](#)
    - [Enviar un correo electrónico escogiendo la aplicación](#)
    - [Abrir una localización en Google Maps](#)
    - [Obtener el resultado de un Intent registrando un '\*contrato\*'](#)
    - [Registrando contratos dentro de un `@Composable`](#)
- ▼ [Gestion de Permisos](#)
  - [Gestionar intent implícito de llamada telefónica](#)
  - [Gestionar intent implícito para hacer un foto](#)
  - [Gestionar registro hacer foto con TakePicture](#)
  - [Gestionar registro para obtener una imagen de la galería](#)
- [Conclusión](#)
-  [Caso de estudio: Obtener el teléfono de un contacto](#)

# Introducción

- **Intents**
  - Documentación oficial: [Intents](#)
  - Documentación oficial: [Interactuando con otras Apps](#)
  - Vídeo Intents: [Philipp Lackner \(Inglés\)](#)
- **Permissions**
  - Documentación oficial: [Permisos](#)
  - Vídeo Permisos 1: [Android Developers](#)
  - Vídeo Permisos 2: [Android Developers](#)
  - Vídeo Permisos: [Philipp Lackner \(Inglés\)](#)
- **Permissions in Compose con Accompanist Library** (Experimental)
  - Vídeo: [Martin Kiperszmid \(Castellano\)](#)
  - Vídeo: [Stevdza-San \(Inglés\)](#)
  - Documentación librería: [Accompanist](#)

Una **Intent** es un **objeto de mensajería** que puedes usar para solicitar una acción de otro componente de una app. Si bien las intents facilitan la comunicación entre componentes de varias formas, existen tres casos de uso principales:

1. Iniciar una actividad de la propia app o de otra app.
2. Iniciar un servicio.
3. Transmitir una emisión o broadcast.

Nosotros en este tema vamos a centrarnos en el primer caso de uso que es, **Iniciar una actividad**. Como se puede deducir es una funcionalidad del API de Android que nos permite iniciar una actividad desde otra actividad incluso si la actividad pertenece a otra aplicación diferente como por ejemplo hacer una llamada telefónica, enviar un correo electrónico, abrir una página web, etc.

## Resumen

Por expresarlo con otras palabras, podemos decir que una **Intent** es un objeto que proporciona en su interior una descripción de la operación que queremos 'intentar' ya sea de forma implícita o explícita y además podemos pasársela datos que se usarán en la operación. Cuidado puede parecernos que es la intent la que lanza una actividad pero no es así, contiene solo la descripción de la operación y los datos necesarios para realizarla.

 **Dercarga:** El proyecto con el código de los ejemplos de este tema en el siguiente [enlace](#)



# Tipos de Intents

## Intents Explícitos

Especifican qué aplicación las administrará, ya sea incluyendo el **nombre del paquete de la app de destino** o el nombre de clase del componente completamente calificado. Normalmente, el usuario usa una intent explícita para iniciar un componente en su propia aplicación porque conoce el nombre de clase de la actividad o el servicio que desea iniciar. Por ejemplo, puedes utilizarla para iniciar una actividad nueva en respuesta a una acción del usuario o iniciar un servicio para descargar un archivo en segundo plano.

### Abriendo otra aplicación desde nuestra aplicación

Para abrir otra aplicación desde nuestra aplicación, debemos crear un objeto **Intent** y especificar el **nombre del paquete de la aplicación de destino**. Por ejemplo, si queremos abrir la aplicación de **Chrome** desde nuestra aplicación, debemos crear un objeto **Intent** y especificar el nombre del paquete de la aplicación de destino que en este caso es `com.android.chrome`. Para ello usaremos el método `setPackage()` del objeto **Intent**. Una vez creado el objeto **Intent** lo pasaremos como parámetro al método `startActivity()`. Como el método `startActivity()` es un método de la clase **Context**, necesitamos un objeto de esta clase para poder llamar al método. Para ello usaremos el `LocalContext.current` que nos proporciona **Jetpack Compose**. El código sería el siguiente...

```

// IntentExplicito.kt

fun Context.openChrome() {
    // Creamos un Intent con la acción ACTION_MAIN
    // que es abrir una actividad principal de la aplicación Chrome.
    Intent(Intent.ACTION_MAIN).also {
        it.`package` = "com.android.chrome"
        // Lanza ActivityNotFoundException si no está instalada la aplicación.
        startActivity(it)
    }
}

@Composable
fun IntentOpenChrome() {
    val context = LocalContext.current
    Box(modifier = Modifier.fillMaxSize(),
        contentAlignment = androidx.compose.ui.Alignment.Center) {
        Button(onClick = { context.openChrome() }) {
            Text(text = "Open Chrome")
        }
    }
}

```

## Saber el nombre del paquete de la app de destino

Para saber el nombre del paquete de la app de destino, podemos usar la herramienta **adb.exe** ([Android Debug Bridge](#)) instalada junto al SDK de Android. Para ello debemos abrir una consola de comandos e ir a la carpeta `platform-tools` donde esté instalado el SDK para nuestro usuario. (Ej: `C:\Users\alumno\AppData\Local\Android\Sdk\platform-tools>`). Otras opción es añadir esta carpeta al **Path** de nuestro usuario para que podamos ejecutar los comandos desde cualquier ubicación, incluso desde la ventana del terminal de AndroidStudio.

Una vez tenemos acceso a la herramienta **adb.exe** seguiremos estos pasos:

1. Arrancaremos la máquina virtual de nuestro emulador de Android **desde el mismo Android Studio** o desde un terminal si sabemos el nombre de la máquina. Por ejemplo, si nuestro emulador se llama **Pixel\_3a\_API\_33** ejecutaremos el comando:

```
C:\Users\alumno\AppData\Local\Android\Sdk\tools\emulator.exe -avd Pixel_3a_API_33
```

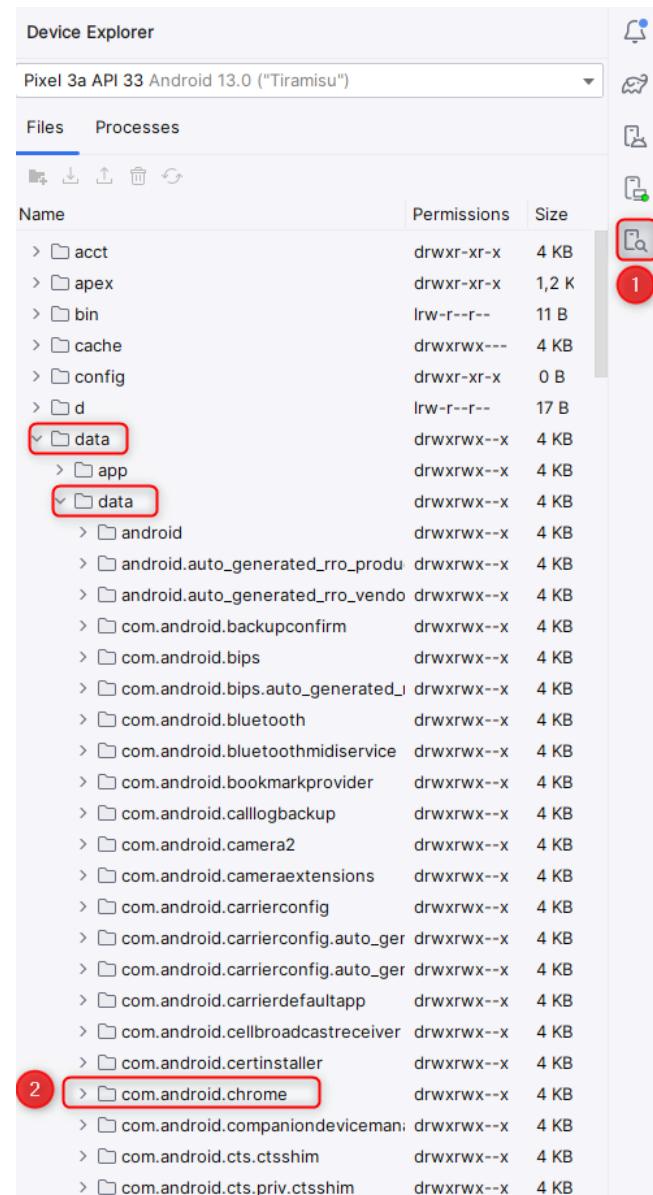
2. **Con el emulador arrancado**, ejecutaremos el comando `adb shell` para acceder a la consola del emulador. Por ejemplo:

```
C:\Users\alumno\AppData\Local\Android\Sdk\platform-tools>adb shell
emu64x:/ $
```

3. Ahora ejecutaremos el comando `pm list packages` para listar todos los paquetes de aplicaciones instaladas en el emulador y si queremos buscar una aplicación en concreto podemos usar el comando `grep`. Por ejemplo, si queremos saber el nombre del paquete de la app de **Chrome**, ejecutaremos el comando:

```
emu64x:/ $ pm list packages | grep chrome
package:com.android.chrome
emu64x:/ $
```

Otra forma más sencilla es a través de **Android Studio**. Una vez arrancado el emulador, abriremos la ventana **Device File Explorer** situado a la derecha como se muestra en la imagen de ejemplo y navegaremos hasta la carpeta `data/data` donde se encuentran todas las aplicaciones instaladas en el emulador. Ahora solo tenemos que buscar la carpeta de la aplicación que nos interese que se corresponde con el nombre del paquete de la aplicación.



# Intents Implícitos

Son los que más interesantes para nosotros. Puesto que no siempre sabemos que aplicaciones vamos a tener instaladas.

No nombran el componente específico, pero en cambio declaran una acción general para realizar, lo cual permite que un componente de otra aplicación la maneje. Por ejemplo, si quieras enviar un correo electrónico no necesitas saber qué aplicación de correo electrónico tiene el usuario instalada, simplemente debes enviar una intent implícita para que cualquier aplicación de correo electrónico pueda responderla. Es más, puedes preguntarle al sistema si existe una actividad que pueda responder a tu intent antes de iniciarla y **ofrecerle la posibilidad al usuario de elegir qué aplicación usar**.

## Enviar un correo electrónico escogiendo la aplicación

Veamos un ejemplo de un **Intent implícito** para solicitar al sistema enviar un correo electrónico con un texto plano. Además, al Intent le pasaremos una serie de parámetros como el asunto, el texto del correo y los destinatarios. Para ello usaremos el método `putExtra()` del objeto **Intent**. El código sería el siguiente...

**Aunque no es necesario**, podemos indicar a Android que tipo de intents vamos a usar en nuestra aplicación al final de Manifest de la aplicación con la etiqueta `<queries>` .

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">

    ...

    <queries>
        <intent>
            <action android:name="android.intent.action.SEND" />
            <data android:mimeType="text/plain" />
        </intent>
    </queries>
</manifest>
```

```

// IntentsImplícitos.kt

fun Context.sendMail(
    correos: Array<String>,
    asunto: String,
    texto: String,
    forzarElección: Boolean = false
) {
    val intent = Intent(Intent.ACTION_SEND).apply {
        type = "text/plain"
        // Añadimos los datos del correo.
        putExtra(Intent.EXTRA_EMAIL, correos)
        putExtra(Intent.EXTRA_SUBJECT, asunto)
        putExtra(Intent.EXTRA_TEXT, texto)
    }
    val chooser = if (forzarElección) {
        val title: String = resources.getString(R.string.enviar_correo)
        Intent.createChooser(intent, title)
    }
    else null
}

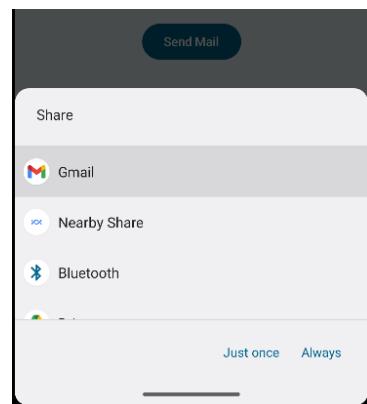
if (intent.resolveActivity(packageManager) != null) {
    startActivity(chooser ?: intent)
}
}

@Composable
fun IntentSendMail() {
    val context = LocalContext.current
    Box(modifier = Modifier.fillMaxSize(),
        contentAlignment = androidx.compose.ui.Alignment.Center) {
        Button(onClick = { context.sendMail(
            correos = arrayOf("correo@alu.edu.gva.es"),
            asunto = "Asunto del correo",
            texto = "Texto del correo"
        ) }) {
            Text(text = "Send Mail")
        }
    }
}

```

Al usar `ACTION_SEND` con esos parámetros, `Android` nos ofrecerá varias opciones para enviar el correo. Si seleccionamos `Gmail` + `Always` la siguiente vez que queramos enviar un correo, **no nos preguntará**.

Realmente el sistema **Android busca el componente apropiado** para iniciar comparando el contenido de la intent con los **filtros de intents** declarados en el archivo de manifiesto de otras aplicaciones en el dispositivo. Si la intent coincide con un filtro de intents, el sistema inicia ese componente y le entrega el objeto Intent. **Si varios filtros de intents son compatibles**, el sistema muestra un cuadro de diálogo para que el usuario pueda elegir la aplicación que se debe usar.



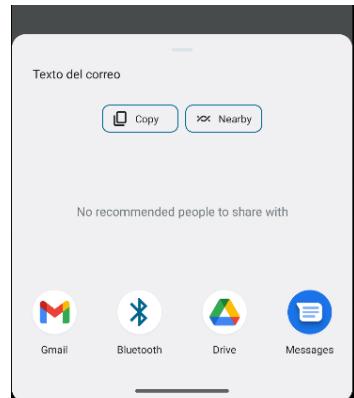
En nuestro caso, la App debe tener una actividad que acepte los parámetros que le pasamos y tener un filtro similar al siguiente:

```
<activity android:name=".ClaseQueDefineLaActividad">
    <android:exported="true"> <!-- Necesario para que otras apps puedan usarla -->
    <intent-filter>
        <action android:name="android.intent.action.SEND" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:mimeType="text/plain" />
    </intent-filter>
</activity>
```

Puedes consultar la [documentación oficial](#) para más información.

 **Nota:** Si no declaras ningún filtro de intent para una actividad, esta solo se puede iniciar con un **intent explícito**.

Si queremos que el sistema nos pregunte siempre que aplicación queremos usar para enviar el correo independientemente de si la hemos predeterminado o no. Deberemos usar el método `createChooser()` del objeto `Intent` como se ve en el código de ejemplo.



## Abrir una localización en Google Maps

Veamos algunos ejemplos más de **Intents implícitos** extraídos de la documentación oficial [aquí](#) o [aquí](#) ...

En este caso vamos a visualizar una geolocalización y la aplicación de `Maps` tiene un filtro que acepta este tipo de intents. Fíjate que en este caso hemos controlado la excepción `ActivityNotFoundException` que se lanza si no tenemos instalada la aplicación de `Maps` mostrando un mensaje temporal.

```
fun Context.buscaEnMaps(lugar: String) {
    val intent = Intent(Intent.ACTION_VIEW).apply {
        data = Uri.parse("geo:0,0?q=$lugar")
    }
    try {
        startActivity(intent)
    } catch (e: ActivityNotFoundException) {
        Toast.makeText(this, "No se puede abrir Maps", Toast.LENGTH_SHORT).show()
    }
}

@Composable
fun IntentBuscaEnMaps() {
    val context = LocalContext.current
    Button(onClick = {
        context.buscaEnMaps("I.E.S Doctor Balmis, +Alicante")
    }) {
        Text(text = "Ver Balmis en Maps")
    }
}
```

## Obtener el resultado de un Intent registrando un 'contrato'

Iniciar otra actividad, ya sea dentro de tu app o desde otra, no tiene por qué ser una operación unidireccional, también puedes iniciar una actividad y recibir un resultado. Por ejemplo, tu app puede iniciar una app de cámara y **recibir la foto tomada como resultado**. También puedes iniciar la app de Contactos para que el usuario **seleccione un contacto y, luego, recibir los detalles correspondientes** como resultado.

Deberemos tener en cuenta que:

- El resultado no es inmediato y **deberemos definir algún tipo de callback para recibirllo**.
- La gestión de ese callback deberá ser **asíncrona** y no bloquear la UI.
- Este callback debe estar ligado al ciclo de vida de la actividad que lo recibe. Esto es, si lanzamos un Intent con respuesta desde una actividad, el callback no se debe gestionar si esta ha finalizado.

Para gestionar todo esto Android ha definido una [serie de clases](#).

Para simplificar, vamos a realizar este proceso creando un **Intent** lanzar otra **Activity** de nuestra propia aplicación de ejemplo **que recibirá un texto y además nos devolverá otro intent con el texto introducido**.

 **Nota:** Si el intent lo hiciéramos sobre **una actividad de otra aplicación** el proceso de comunicación será el mismo de nuestro ejemplo.

Para ello, nos situamos sobre el paquete `views` seleccionamos botón derecho y crear una nueva clase que represente una actividad llamada `ActivityQueProduceUnTexto`. Recuerda que para usar compose debe heredar de la clase `ComponentActivity`. El código sería el siguiente...

```

class ActivityQueProduceUnTexto : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            EjemplosIntentsTheme {
                Surface(
                    modifier = Modifier.fillMaxSize(),
                    color = MaterialTheme.colorScheme.background
                ) {
                }
            }
        }
    }
}

```

Además, en el **manifest.xml** vamos a añadir la actividad que acabamos de crear. El código sería el siguiente, dentro de la etiqueta `<application>` bajo la etiqueta `<activity>` que define nuestro **MainActivity** ...

 **Nota:** Fíjate que hemos definido el recurso cadena `title_activity_que_produce_un_texto`. Haz `Ctrl+.` para definirlo con el texto "**ActivityQueProduceUnTexto**"

```

<activity
    android:name=".ui.views.ActivityQueProduceUnTexto"
    android:exported="true"
    android:label="@string/title_activity_que_produce_un_texto"
    android:theme="@style/Theme.EjemplosIntents">
    <intent-filter>
        <action android:name="android.intent.action.SEND" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:mimeType="text/plain" />
    </intent-filter>
</activity>

```

 **Importante:** Fíjate además, que hemos añadido el atributo `android:exported="true"` . Esto es necesario para que **otras aplicaciones** puedan usar esta actividad y además hemos expuesto a Android que podemos recibir intents de tipo `SEND` con datos de tipo `text/plain` para que nos ofrezca a otras Apps en ese caso.

Podemos ahora definir la siguiente UI con Compose para **ActivityQueProduceUnTexto** en la cual mostramos el dato de texto recibido desde el **MainActivity** y además tenemos un campo de texto para que el usuario introduzca un texto y lo devuelva al **MainActivity** a través del manejador `onClickDevolver` . El código puede ser el siguiente...

```

@Composable
fun PideTexto(
    textoRecibidoPorLlamador: String = "Sin llamador",
    onClickDevolver: (String) -> Unit = {}
) {
    var texto by rememberSaveable { mutableStateOf("") }
    Column(modifier = Modifier.fillMaxSize(),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Center) {
        Text(
            text = textoRecibidoPorLlamador,
            modifier = Modifier
                .padding(10.dp)
                .fillMaxWidth(),
            style = MaterialTheme.typography.titleLarge,
            textAlign = TextAlign.Center
        )
        Spacer(modifier = Modifier.height(16.dp))
        OutlinedTextField(
            label = { Text(text = "Texto a devolver") },
            value = texto,
            onValueChange = { texto = it }
        )
        Spacer(modifier = Modifier.height(16.dp))
        Button(onClick = { onClickDevolver(texto) }) {
            Text(text = "Devolver texto")
        }
    }
}

```

Veamos ahora cómo quedaría el código definitivo de **ActivityQueProduceUnTexto**.

```

class ActivityQueProduceUnTexto : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        // Recuperamos el texto que nos ha pasado el llamador y que está en
        // la propiedad intent de la actividad que contiene los datos
        // que se han pasado desde el llamador.
        // Los asignamos en un estado usado por la interfaz en la composición.
        val textoRecibidoPorLlamador by mutableStateOf(
            intent.getStringExtra("TEXTO") ?: ""
        )

        val onClickDevolver: (String) -> Unit = { texto ->
            // Para devolver datos al llamador, los empaquetamos en un
            // Intent le añadimos los datos y lo pasamos como parámetro
            // al método setResult, que crea un objeto ActivityResult que
            // es lo que espera recibir el llamador en su 'contrato'.
            Intent().also {intento ->
                intento.putExtra("TEXTODEVUELTO", texto)
                setResult(RESULT_OK, intento)
            }
            // Finalizamos la actividad tras poner los datos de resultado en el intent
            finish()
        }

        setContent {
            EjemplosIntentsTheme {
                Surface(
                    modifier = Modifier.fillMaxSize(),
                    color = MaterialTheme.colorScheme.background
                ) {
                    PideTexto(
                        textoRecibidoPorLlamador = textoRecibidoPorLlamador,
                        onClickDevolver = onClickDevolver
                    )
                }
            }
        }
    }
}

```

Veamos ahora cómo sería el proceso de llamar a `ActivityQueProduceUnTexto` con un Intent explícito con un texto y recibir el resultado en otro intent envuelto en un `ActivityResult`.

En primer lugar definimos una interfaz `@Composable` en `MainActivity` (fuera de la clase) que nos permita lanzar la actividad y gestionar su respuesta denominado

`InterfaceParaLanzarActivityQueProduceUnTexto`

```
@Composable
fun InterfaceParaLanzarActivityQueProduceUnTexto(
    textoDevueltoPorIntent: String = "No has llamado aún",
    onClickLanzar: () -> Unit = {}
) {
    Column(modifier = Modifier.fillMaxSize(),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Center) {
        Button(onClick = onClickLanzar) {
            Text(text = "Llama a ActivityQueProduceUnTexto")
        }
        Text(
            text = textoDevueltoPorIntent,
            modifier = Modifier
                .padding(10.dp)
                .fillMaxWidth(),
            style = MaterialTheme.typography.titleLarge,
            textAlign = TextAlign.Center
        )
    }
}
```

Ahora usaremos el método `registerForActivityResult()` de la clase `ComponentActivity` para registrar un `ActivityResultLauncher` que se encargará de gestionar la respuesta del intent. El código sería el siguiente:

```

class MainActivity : ComponentActivity() {
    private var textoDevueltoPorIntent by mutableStateOf("No has llamado aún")
    // Definimos un lanzador con el contrato ActivityResultContracts.StartActivityForResult
    // que recibirá (ENTRADA) el intent explícito sobre la actividad secundaria
    // y devolverá (SALIDA) un objeto ActivityResult que contiene el intent de respuesta
    private val launcherActivityQueProduceUnTexto: ActivityResultLauncher<Intent> =
        registerForActivityResult(ActivityResultContracts.StartActivityForResult())
    { result ->
        if (result.resultCode == Activity.RESULT_OK) {
            textoDevueltoPorIntent = result.data?.getStringExtra("TEXTODEVUELTO")
            ?: "Nada retornado"
        }
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            EjemplosIntentsTheme {
                Surface(modifier = Modifier.fillMaxSize()) {
                    val context = LocalContext.current
                    InterfaceParaLanzarActivityQueProduceUnTexto(
                        textoDevueltoPorIntent = textoDevueltoPorIntent,
                        onClickLanzar = {
                            // En lanzado con el contrato encargado de gestionar una
                            // respuesta ActivityResult. Para ello crea el
                            // Intent explícito pasándole un texto como parámetro
                            // indicándole por quien es llamada.
                            // El contexto se puede sacar de LocalContext.current y también
                            // claururando el propio contexto de la actividad.
                            launcherActivityQueProduceUnTexto.launch(
                                Intent(context, ActivityQueProduceUnTexto::class.java)
                                    .apply {
                                        putExtra("TEXTO", "Te llamo desde MainActivity")
                                    }
                            )
                        }
                    )
                }
            }
        }
    }
}

```

Vemos con más detalle el código de la llamada ...

```

private val launcherActivityQueProduceUnTexto: ActivityResultLauncher<Intent> =
    registerForActivityResult(ActivityResultContracts.StartActivityForResult())
{ result ->
    if (result.resultCode == Activity.RESULT_OK) {
        textoDevueltoPorIntent = result.data?.getStringExtra("TEXTODEVUELTO")
        ?: "Nada retornado"
    }
}

```

Vamos a separar la respuesta de la llamada porque después de acabar la actividad se destruirá.

- **Paso 1:** Crearemos un objeto `ActivityResultLauncher` mediante el método

`registerForActivityResult()` de la clase `ComponentActivity`. Este método recibe dos parámetros:

```

public final <I, O> ActivityResultLauncher<I> registerForActivityResult(
    @NonNull ActivityResultContract<I, O> contract,
    @NonNull ActivityResultCallback<O> callback) {
    return registerForActivityResult(contract, mActivityResultRegistry, callback);
}

```

Donde ...

- `ActivityResultContract` es un 'contrato' que especifica que una actividad se puede llamar con una entrada de tipo I y producir una salida de tipo O. Aunque se pueden crear contratos personalizados, la API proporciona contratos predeterminados para acciones de intent básicas, como tomar una foto, solicitar permisos, etc.

Por ejemplo, `ActivityResultContracts.StartActivityForResult()` es una contrato que especifica que una actividad se puede llamar con una **entrada de tipo Intent** y producir una **salida de tipo ActivityResult**.

- `ActivityResultCallback` es un *callback* al que le llega un objeto de tipo O, para nuestro contrato un `ActivityResult` y hace una determinada acción sin retornar o '*producir*' nada.

`result : ActivityResult -> void`



### Aviso

Los `registerForActivityResult` deben hacerse en el `onCreate` de la actividad.

- **Paso 2:** Crearemos un Intent con los datos y el nombre de la Activity como siempre y después con el objeto `launcherActivityQueProduceUnTexto` lanzaremos en Intent según lo establecido en el contrato del lanzador. El código sería el siguiente:

```
launcherActivityQueProduceUnTexto.launch(  
    Intent(context, ActivityQueProduceUnTexto::class.java).apply {  
        putExtra("TEXTO", "Te llamo desde MainActivity")  
    }  
)
```

## Registrando contratos dentro de un `@Composable`

La API de `rememberLauncherForActivityResult` es análoga a `registerForActivityResult` pero permite registrar un contrato dentro de un `@Composable` y guardarlo como un estado que sobrevive a las recomposiciones. Veamos el siguiente ejemplo extrado de la [documentación oficial](#) ...

```
@Composable  
fun GetContentExample() {  
    var imageUri by remember { mutableStateOf<Uri?>(null) }  
    val launcher = rememberLauncherForActivityResult(GetContent()) { uri: Uri? ->  
        imageUri = uri  
    }  
    Column {  
        Button(onClick = { launcher.launch("image\*") }) {  
            Text(text = "Load Image")  
        }  
        Image(  
            painter = rememberImagePainter(imageUri),  
            contentDescription = "My Image"  
        )  
    }  
}
```

`GetContent()` es un contrato que especifica que se puede llamar a una actividad de selección de imágenes con una entrada de tipo `String`, que es la ruta de donde escogerlas y producir una salida de tipo `Uri` que es la ruta a ella.

# Gestión de Permisos

Para realizar lanzar algunos intents sobre actividades y servicios del sistema necesitaremos de ciertos permisos. Por ejemplo, para abrir la cámara, para leer la agenda de contactos, etc.

La forma de gestionar los permisos ha cambiado en las últimas versiones de Android. En versiones anteriores a **Android 6.0 (API nivel 23)**, los usuarios concedían todos los permisos solicitados en el manifiesto por una app en el momento de la instalación.

Ahora en Android, el **sistema gestiona los permisos de las apps mientras se ejecutan** y los usuarios pueden revocar cualquier permiso en cualquier momento.

El esquema para definir permisos es:

1. Definiremos los requerimientos de permisos en el `AndroidManifest.xml` de la aplicación.
2. Registraremos un contrato de solicitud de permisos

`ActivityResultContracts.RequestPermission()` con el método `registerForActivityResult` o `rememberLauncherForActivityResult` de la clase `ComponentActivity`.

El manejador del resultado de los permisos creará el intent para el servicio o activity y se pueden dar dos casos:

- i. Que el servicio no retorne nada por ejemplo una llamada telefónica.

En este caso, desde el manejador si el permiso ha sido concedido lanzaremos el intent de llamada con el método `startActivity()` como ya hemos visto.

- ii. Que el servicio retorne un resultado por ejemplo una foto.

En este caso, desde el manejador si el permiso ha sido concedido lanzaremos el `launch` de del registro de un contrato que se encargará de gestionar la respuesta con el resultado.

# Gestionar intent implícito de llamada telefónica

1. Añadimos en el `AndroidManifest.xml` el permiso `CALL_PHONE`. Estas etiquetas irán dentro de la etiqueta `<manifest>`:

```
<uses-feature  
    android:name="android.hardware.telephony"  
    android:required="true" />  
<uses-permission android:name="android.permission.CALL_PHONE"/>
```

2. Por ejemplo, si queremos que el método de registro sea `@Composable` haremos.

```
@Composable  
fun registroLlamarPorTelefonoIntent(  
    telefono: String  
) : ManagedActivityResultLauncher<String, Boolean> {  
    val context = LocalContext.current  
    return rememberLauncherForActivityResult(  
        ActivityResultContracts.RequestPermission()  
) { success ->  
        if (success) {  
            Intent(Intent.ACTION_CALL).also {  
                callIntent ->  
                callIntent.data = Uri.parse("tel:$telefono")  
                context.startActivity(callIntent)  
            }  
        }  
    }  
}
```

Posteriormente podemos hacer ...

```
@Composable  
fun IntentLlamadaConPermisos(telefono: String) {  
    val launcherTelefono = registroLlamarPorTelefonoIntent(telefono)  
    Button(onClick = {  
        launcherTelefono.launch(android.Manifest.permission.CALL_PHONE)  
    }) {  
        Text(text = "Llamar al $telefono")  
    }  
}
```

# Gestionar intent implícito para hacer un foto

1. Añadimos en el `AndroidManifest.xml` el permiso `CAMERA`. Estas etiquetas irán dentro de la etiqueta `<manifest>`:

```
<uses-feature  
    android:name="android.hardware.camera.any"  
    android:required="true" />  
<uses-permission android:name="android.permission.CAMERA"/>
```

2. Por ejemplo, si queremos que el método de registro sea `@Composable` haremos.

```
@Composable  
fun registroHacerFotoConIntent(  
    onFotoCambiada: (ImageBitmap) -> Unit  
) : ManagedActivityResultLauncher<String, Boolean> {  
    val cameraLauncher =  
        rememberLauncherForActivityResult(ActivityResultContracts  
            .StartActivityForResult()) { result ->  
            if (result.resultCode == Activity.RESULT_OK) {  
                val androidBitmap = result.data?.extras?.get("data") as Bitmap  
                onFotoCambiada(androidBitmap!!.  
                    asImageBitmap())  
            }  
        }  
  
    return rememberLauncherForActivityResult(  
        ActivityResultContracts.RequestPermission()  
) { success ->  
        if (success) {  
            val cameraIntent = Intent(MediaStore.ACTION_IMAGE_CAPTURE_SECURE)  
            cameraLauncher.launch(cameraIntent)  
        }  
    }  
}
```

Posteriormente podemos hacer ...

```
@Composable
fun IntentFotoConPermisos(
    onFotoCambiada: (ImageBitmap) -> Unit
) {
    val launcherHacerFoto = registroHacerFotoConIntent(onFotoCambiada)
    Button(onClick = {
        launcherHacerFoto.launch(android.Manifest.permission.CAMERA)
    }) {
        Text(text = "Hacer foto")
    }
}
```

# Gestionar registro hacer foto con TakePicture

Esta forma de hacer fotos nos permitirá guardarla foto en una caché local como JPG y lo que recibirá el callback será la ruta de la foto.

1. Además de añadir los permisos de acceso a la cámara como hemos hecho antes. Añadimos en el `AndroidManifest.xml` un proveedor de ficheros para que la aplicación pueda acceder a la caché de la aplicación. Estas etiquetas irán dentro de la etiqueta `<application>`:

```
<application>
    ...
    <provider
        android:name="androidx.core.content.FileProvider"
        android:authorities="${applicationId}.provider"
        android:exported="false"
        android:grantUriPermissions="true">
        <meta-data
            android:name="android.support.FILE_PROVIDER_PATHS"
            android:resource="@xml/path_provider" />
    </provider>
</application>
```

2. Definiremos el recurso XML `path_provider.xml` en `res/xml`. Donde definimos que las imágenes se guardarán en la carpeta `cache` de la aplicación situada en `/sdcard/Android/data/<package_name>/cache`. En esta ubicación encontraremos los JPG que se vayan generando.

```
<?xml version="1.0" encoding="utf-8"?>
<paths>
    <external-cache-path
        name="my_images"
        path="/" />
</paths>
```

3. Ya solo nos queda definir el contrato `ActivityResultContracts.TakePicture()` cuyo callback, a través del `FileProvider` definido nos permitirá obtener la ruta de la foto en la caché.

```

private fun Context.toImageBitmap(uri: Uri): ImageBitmap {
    val contextResolver = this.contentResolver
    val source = ImageDecoder.createSource(contextResolver, uri)
    return ImageDecoder.decodeBitmap(source).asImageBitmap()
}

@Composable
fun registroHacerFotoConTakePicture(
    onFotoCambiada: (ImageBitmap) -> Unit
): ManagedActivityResultLauncher<String, Boolean> {

    val context = LocalContext.current
    val ficheroTemporal = File.createTempFile(
        "JPEG_${SimpleDateFormat("yyyyMMdd_HHmmss").format(Date())}_",
        ".jpg",
        context.getExternalCacheDir
    )
    val uri = FileProvider.getUriForFile(
        context,
        "${context.packageName}.provider",
        ficheroTemporal
    )
    val cameraLauncher =
        rememberLauncherForActivityResult(
            ActivityResultContracts.TakePicture()) { success ->
            if (success) {
                onFotoCambiada(context.toImageBitmap(uri))
            }
        }
    return rememberLauncherForActivityResult(
        ActivityResultContracts.RequestPermission()
    ) { success ->
        if (success) {
            cameraLauncher.launch(uri)
        }
    }
}

```

# Gestionar registro para obtener una imagen de la galería

En el punto Registrando contratos dentro de un `@Composable` ya hemos visto un ejemplo de como hacerlo. Veamos como concretarlo para seguir el mismo esquema que hemos seguido en los ejemplos anteriores.

1. Definimos el contrato `ActivityResultContracts.GetContent()` que especifica que se puede llamar a una actividad de selección de imágenes con una entrada de tipo `String` que es la ruta de donde escogerlas y producir una salida de tipo `Uri` que es ruta a ella.

```
@Composable
fun registroSelectorDeImagenesConGetContent(
    onFotoCambiada: (ImageBitmap) -> Unit
): ManagedActivityResultLauncher<String, Uri?> {
    val context = LocalContext.current
    return rememberLauncherForActivityResult(
        ActivityResultContracts.GetContent()) { uri ->
        uri?.let {
            onFotoCambiada(context.toImageBitmap(uri))
        }
    }
}
```

2. Posteriormente podemos hacer ...

```
@Composable
fun IntentImagenGaleria(
    onFotoCambiada: (ImageBitmap) -> Unit
) {
    val launcherFotoGaleria = registroSelectorDeImagenesConGetContent(onFotoCambiada)
    Button(onClick = {
        launcherFotoGaleria.launch("image/*")
    }) {
        Text(text = "Foto galería")
    }
}
```

# Conclusión

Como has podido observar gestionar permisos y lanzar intents es un proceso que puede ser complejo y que requiere de un conocimiento profundo de la API de Android. Por ello, en ocasiones es recomendable utilizar librerías de terceros que nos faciliten el trabajo. Por ejemplo dispones de la librería **accompanist** que ofrece compatibilidad con Compose para gestionar permisos y lanzar intents.

Nosotros en el módulo hemos creado una librería que puedes usar para gestionar los permisos y lanzar intents de una forma más sencilla. Añadiendo algunos los más comunes vistos en el tema o usado en los ejercicios.

Para usarlos **debes añadir o comprobar si tienes añadidas** la dependencia de la librería:

En `libs.version.toml`:

```
pmdm-ies-balmis-utilities  
= { group = "com.github.pmdmiesbalmis", name = "utilities", version.ref = "pmdmIesBalmisVersion" }
```

En `build.gradle.kts` del módulo app:

```
dependencies { implementation(libs.pmdm.ies.balmis.utilities) }
```

En el fuente donde vayas a usar los métodos de la librería debes añadir el import:

```
import com.github.pmdmiesbalmis.utilities.device.*
```

# Caso de estudio: Obtener el teléfono de un contacto

Partiendo de la de las utilidades de la librería `com.github.pmdmiesbalmis.utilities.device` vamos a añadir un caso más que nos permita **obtener el teléfono de un contacto** de la agenda del dispositivo.

 **Descarga:** Si no consigues hacerlo, puedes descargar el código de este caso de estudio en el siguiente enlace: [intents\\_permisos\\_caso\\_estudio](#)

Para ello vamos a adaptar el tutorial de la documentación oficial:  
[Seleccionar datos de un contacto específico](#) para hacerlo.

**Paso 1:** Asegúrate de tener una cuenta de pruebas en el dispositivo y haber añadido al menos un contacto con un teléfono.

**Paso 2:** Puesto que vamos a usar la agenda de contactos, debemos añadir el permiso `READ_CONTACTS` en el `AndroidManifest.xml`. Estas etiquetas irán dentro de la etiqueta `<manifest>`:

```
<uses-permission android:name="android.permission.READ_CONTACTS"/>
```

**Paso 3:** Analiza el código de `registroSelectorTelefonoContacto` en la librería `com.github.pmdmiesbalmis.utilities.device` que nos permitirá obtener el teléfono de un contacto de la agenda. El código sería el siguiente:

```

@Composable
fun registroSelectorTelefonoContacto(
    onSeleccionNumeroContacto: (String) -> Unit
): ManagedActivityResultLauncher<String, Boolean> {

    val contexto = LocalContext.current
    val registroObtenerTelefono = rememberLauncherForActivityResult(
        contract = ActivityResultContracts.StartActivityForResult(),
        onResult = { result ->
            if (result.resultCode == Activity.RESULT_OK) {
                val contactUri: Uri? = result.data?.data
                val projection = arrayOf(ContactsContract.CommonDataKinds.Phone.NUMBER)
                if (contactUri != null) {
                    contexto.contentResolver
                        .query(contactUri, projection, null, null, null)
                        .use { cursor ->
                            if (cursor != null && cursor.moveToFirst()) {
                                val numberIndex =
                                    cursor.getColumnIndex(ContactsContract.CommonDataKinds
                                        .Phone.NUMBER)
                                val number =
                                    if (numberIndex >= 0)
                                        cursor.getString(numberIndex)
                                    else "NO NUMBER"
                                onSeleccionNumeroContacto(number)
                            }
                        }
                }
            }
        }
    )

    return rememberLauncherForActivityResult(
        ActivityResultContracts.RequestPermission()
    ) { success ->
        if (success) {
            val intent = Intent(Intent.ACTION_PICK).apply {
                type = ContactsContract.CommonDataKinds.Phone.CONTENT_TYPE
            }
            registroObtenerTelefono.launch(intent)
        }
    }
}

```

La función composable `registroSelectorTelefonoContacto` recibe un callback que se ejecutará cuando se seleccione un contacto de la agenda y recibirá como parámetro una cadena con el teléfono el mismo.

Para ello, registramos y devolveremos el contrato de permisos

`ActivityResultContracts.RequestPermission()` que nos permitirá lanzar un intent de selección de contactos `Intent.ACTION_PICK` y que nos devolverá un `Uri` con el contacto seleccionado. Para ello le pasamos el tipo de MIME `CommonDataKinds.Phone.CONTENT_TYPE` que nos devolverá un contacto con un **teléfono**.

### Nota

Como se indica en la documentación, tenemos otros tipos de MIME para obtener contactos con un **correo electrónico** como `CommonDataKinds.Email.CONTENT_TYPE` o una **dirección postal** como `CommonDataKinds.StructuredPostal.CONTENT_TYPE`.

**Paso 4:** Por último, podemos usar la función composable `registroSelectorTelefonoContacto` para obtener el teléfono de un contacto. En el **mismo nivel que definimos el estado del teléfono**, definiremos el registro del contrato y la llamada al mismo. El código sería el siguiente:

```

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            EjemploIntentsTheme {
                var telefono by rememberSaveable { mutableStateOf("")}
                val registroSeleccionContacto = registroSelectorTelefonoContacto {
                    telefono = it
                }
                Surface(
                    modifier = Modifier.fillMaxSize(),
                    color = MaterialTheme.colorScheme.background
                ) {
                    PruebaIntentsScreen(telefono) {
                        registroSeleccionContacto.launch(
                            android.Manifest.permission.READ_CONTACTS
                        )
                    }
                }
            }
        }
    }
}

@Composable
fun PruebaIntentsScreen(
    telefono: String,
    onSeleccionContacto: () -> Unit
) {
    Column(
        modifier = Modifier.fillMaxSize(),
        verticalArrangement = Arrangement.Center,
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
        Button(onClick = onSeleccionContacto) { Text("Seleccionar contacto") }
        Text(telefono)
    }
}

```