

Publicación de aplicaciones y analítica (Parte I)

Gradle

Competencias

- Entender el proceso de construcción usando Gradle.
- Usar Gradle para compilar un APK.
- Entender y manipular los tipos de compilación.
- Entender y manipular los tipos de producto.

Introducción

La construcción de los proyectos Android se realiza ejecutando un *script* de Gradle. El nombre de este archivo es por convención **build.gradle**.

El formato para escribir los *build scripts* es un *Domain-Specific Language* (DSL) basado en Groovy que es un lenguaje para la JVM. Sin embargo, dado que está basado en la JVM es posible escribir tareas y plugins propios en algún otro lenguaje compatible como Kotlin o Java

El enfoque declarativo aplicado con el DSL otorga ventajas, al ser más simple logra que sea más fácil de aprender a diferencia de otros enfoques basados en procedimientos como *Ant* o *Maven*.

Al terminar esta unidad el alumno entenderá la importancia del archivo `build.gradle` y cómo manipularlo para lograr construir APKs con distintas características según los distintos requerimientos

¿Qué es Gradle?

Gradle es una herramienta de automatización de construcción (Build automation) diseñada para ser flexible en la construcción de cualquier tipo de *software*.

Gradle favorece el uso de convenciones por sobre configuraciones y generalmente entrega valores por defecto para ajustes y propiedades

Sus características principales tiene relación principalmente con la **velocidad y flexibilidad en la construcción**. Además, otras características son:

- **Alto rendimiento:** Gradle evita el trabajo innecesario ejecutando sólo las tareas necesarias que han cambiado su entrada o salida. Además se puede activar un caché para la construcción utilizando la salida de construcciones previas y ejecuta tareas en paralelo.
- **Corre sobre JVM:** Gradle se ejecuta sobre la JVM y se debe tener instalado un JDK para que funcione. No tiene dependencias externas, por lo que cualquier proyecto Java o Android (incluso con Kotlin) puede ocupar Gradle. Además es fácil de correr en multiplataforma.
- **Convenciones:** Gradle toma parte de la filosofía de Maven y hacer que los tipos comunes de proyectos, como los proyectos Java, sean fáciles de construir mediante la implementación de convenciones. Se puede aplicar los complementos apropiados y tener instrucciones de compilación acotadas para muchos proyectos.
- **Extensibilidad:** Más aún, las tareas comunes definidas por convención pueden ser extendidas y personalizadas entregando más flexibilidad a la construcción.
- **Soporte para IDE:** Varios IDEs permiten importar los archivos Gradle para compilar y permiten interactuar con ellos de manera sencilla. Entre estos, se encuentran Android Studio, IntelliJ IDEA, Eclipse, NetBeans y Visual Studio.

Ciclo de vida de la construcción

La construcción usando Gradle está dividida en 3 etapas en las cuales se inicializa, configura y ejecuta tareas específicas.

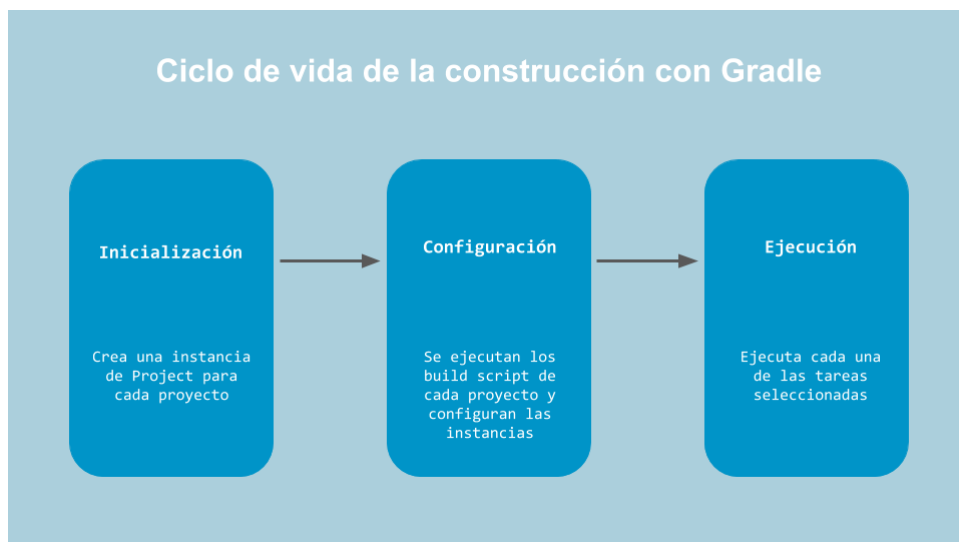


Imagen 1. Ciclo de vida de la construcción con Gradle.

Inicialización

Gradle soporta construcciones para uno o varios proyectos. Durante la fase de inicialización determina qué proyectos van a tomar parte de la construcción y crea una instancia de la clase `Project` para cada uno de estos proyectos

Archivo de configuración

Además del *script* de construcción, Gradle utiliza un archivo de configuración, que tiene el nombre por default de `s`. Este archivo indica si es una construcción de un proyecto único o se conforma por varios proyectos, y además indica cuales son los proyectos que componen la construcción

En el caso de un proyecto único, *settings.gradle* solo hace referencia al nombre de un proyecto

```
include 'app'
```

En el caso de multi-proyecto, el archivo indica cada uno de ellos

```
include 'project1', 'project2:child', 'project3:child1'
```

Dependiendo de las funcionalidades y la estructura del código se puede necesitar más de un proyecto, por ejemplo para agrupar código para un módulo específico que interactúe con código común dentro de la app.

Configuración y Ejecución

Para un proyecto único, el flujo después de la fase de inicialización es simple. El *script* de construcción es ejecutado contra la instancia de `Project` previamente creada. Luego Gradle busca por las tareas con nombre igual a los pasados como argumento. Si las tareas con esos nombres existe, entonces son ejecutadas en forma independiente en mismo orden entregado.

Para ejecutar multi-proyectos el flujo es más complejo y está por fuera del alcance de la unidad, dado que para Android se genera 1 solo proyecto. Para un análisis completo de la construcción y ejecución de multi-proyectos, la documentación oficial de Gradle le dedica una parte especialmente

Android Gradle plugin

El sistema de construcción que utiliza Android Studio está basado en Gradle el *Android Gradle Plugin* entrega funcionalidades que son específicas para la construcción de Android. Tanto el plugin como Gradle pueden correr en forma independiente de Android Studio

Archivo build.gradle

El archivo **build.gradle** es el utilizado para compilar y empaquetar los proyectos, que es el nombre asignado por convención.

En un proyecto Android hay 2 archivos build.gradle que son creados por defecto para cada nuevo proyecto.

1. El build.gradle de la raíz del proyecto es para agregar las configuraciones comunes a todos los módulos o subproyectos
2. El build.gradle dentro de cada subproyecto o módulo, que tiene la configuración particular

En un proyecto Android no se debe manipular el archivo en la raíz del proyecto, modificando exclusivamente el script de cada módulo, añadiendo o quitando elementos de los archivos gradle.

Actualización de Android Gradle plugin

El *plugin* es actualizado en sincronía con Android Studio. Luego de actualizar Android Studio se muestra un mensaje para actualizar el plugin en forma automática pero opcional.

Por ejemplo, para configurar el plugin de Gradle en una versión específica, se debe modificar el archivo build.gradle de la raíz del proyecto. En este caso, se especifica la versión 3.5.0 del plugin (no de Gradle)

```
// Top-level build file where you can add configuration options common to all sub-projects/modules.
```

```
buildscript {  
    repositories {  
        google()  
    }  
    dependencies {  
        classpath 'com.android.tools.build:gradle:3.5.0'  
        // NOTE: Do not place your application dependencies here; they belong  
        // in the individual module build.gradle files  
    }  
}  
...  
}
```

Desde la versión 4.1 de Gradle incluye el soporte para el repositorio de Google Maven usando el método **google()**. Para las versiones anteriores hay que incluir el repositorio para poder utilizarlo.

Precaución: NO se debe utilizar los números de dependencia en forma dinámica, por ejemplo, com.android.tools.build:gradle:3.+Este tipo de actualización automática dificulta resolver los conflictos entre versiones.

Configuración personalizadas

Usando el plugin para Android Studio y Gradle, se puede personalizar los siguientes aspectos de la compilación

Tipos de compilación (Build Types)

Cada **variante de compilación** representa una versión diferente de la app que se puede compilar y surgen del uso de conjuntos de reglas específicas de Gradle que combinan configuración, códigos y recursos configurados.

El archivo *build.gradle* del módulo contiene las variantes de compilación. Al crear un nuevo proyecto, el plugin de Gradle crea automáticamente los tipos de compilación para depuración o lanzamiento. Para depuración crea una clave de depuración que permite depurar la app en dispositivos.

```
android {
    compileSdkVersion 29
    defaultConfig { ... }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'
        }
    }
}
```

En el archivo *build.gradle* no se hace referencia explícita a la variable de compilación de depuración, Android Studio maneja la configuración con **debuggable true**.

Lo que explícitamente sería de la siguiente manera

```
android {
    compileSdkVersion 29
    defaultConfig { ... }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'
        }

        debug {
            debuggable true
        }
    }
}
```

Existen más propiedades para configurar los tipos de compilación, que se puede revisar en las *referencias DSL del tipo de compilación*. Usando Android Studio se pueden configurar seleccionando desde el menú *File > Project Structure*.

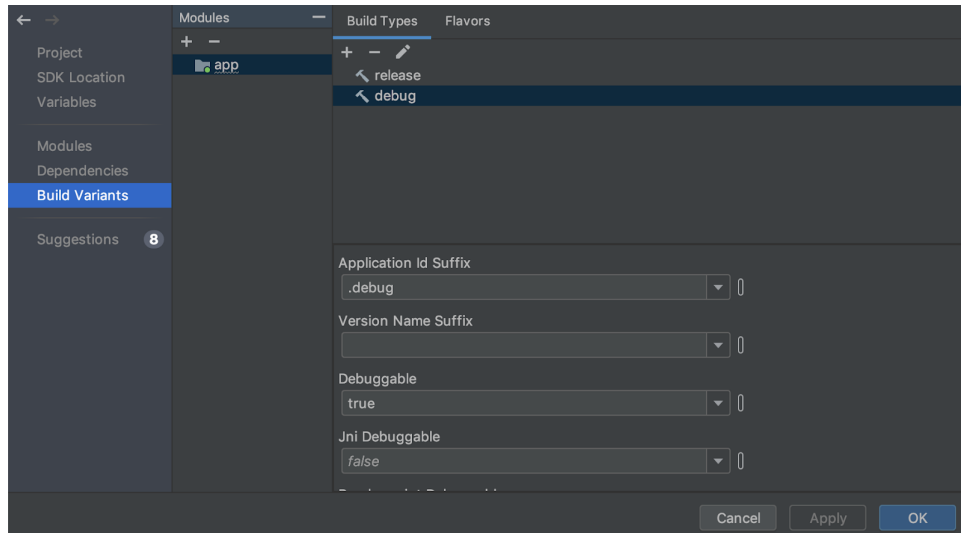


Imagen 2. Propiedades del tipo de configuración.

Tipos de producto (Flavors)

La compilación de tipos de productos es similar a las variantes de compilación y representa varias versiones de la misma app, como por ejemplo, las versiones pagadas y gratuitas. Se puede personalizar los tipos de producto para utilizar código y recursos diferentes y al mismo tiempo reutilizar partes comunes a las distintas versiones. Los tipos de productos son **opcionales** y deben ser definidos en el archivo *build.gradle*

Un *build.gradle* que tenga dos tipos de producto: una versión **demo** y una versión **full** es de la siguiente manera

```
android {
    ...
    defaultConfig {...}
    buildTypes {
        debug {...}
        release {...}
    }

    flavorDimensions "version"
    productFlavors {
        demo {
            dimension "version"
            applicationIdSuffix ".demo"
            versionNameSuffix "-demo"
        }
        full {
            dimension "version"
            applicationIdSuffix ".full"
            versionNameSuffix "-full"
        }
    }
}
```

flavorDimensions permite combinar configuraciones de varios tipos de producto, por ejemplo, para crear productos del tipo demo y full mezcladas con niveles de API mínima segmentadas. Se define una o más dimensiones y se indica a qué dimensión corresponde cada flavor.

```
android {  
    ...  
    buildTypes {  
        debug {...}  
        release {...}  
    }  
  
    flavorDimensions "api", "mode"  
    productFlavors {  
        demo {  
            dimension "mode"  
            applicationIdSuffix ".demo"  
            versionNameSuffix "-demo"  
        }  
  
        full {  
            dimension "mode"  
            applicationIdSuffix ".full"  
            versionNameSuffix "-full"  
        }  
  
        minApi24 {  
            dimension "api"  
            minSdkVersion 24  
            versionNameSuffix "-minApi24"  
            ...  
        }  
  
        minApi21 {  
            dimension "api"  
            minSdkVersion 21  
            versionNameSuffix "-minApi21"  
            ...  
        }  
    }  
    ...  
}
```

Si no se agrega ningún flavorDimension, Gradle no puede construir el proyecto porque debe pertenecer al menos a una dimensión. El error reportado es de este estilo

ERROR: All flavors must now belong to a named flavor dimension.
Learn more at <https://d.android.com/r/tools/flavorDimensions-missing-error-message.html>

Las configuraciones también pueden ser administradas usando la interfaz visual proporcionada por Android Studio seleccionando desde el menú File -> Project Structure

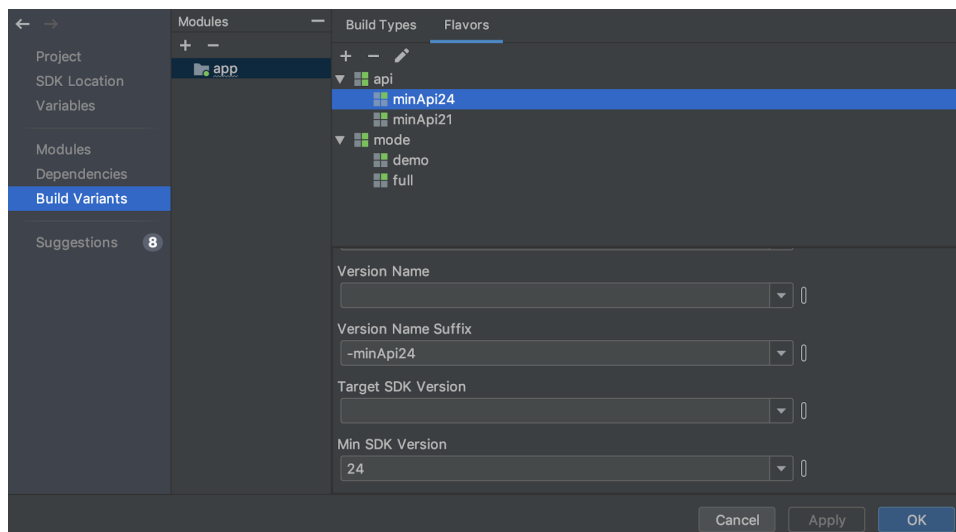


Imagen 3. Administración de configuraciones.

Con los flavors y dimensiones definidas, se hace un producto cruzado entre las dimensiones, dejando excluido todos los productos dentro de la misma dimensión, generando así las variables de compilación

Variables de compilación (Build Variants)

Una variable de compilación es el producto cruzado entre los tipos de compilación y tipo de producto, y es la configuración que se utiliza para compilar el APK. Dado lo anterior, las variables de compilación no son configuradas directamente si no que mediante la configuración de los tipos de compilación y producto.

Seleccionando la opción Build -> Select Build Variant, es posible identificar las variables de compilación generadas a partir de la configuración de ambos tipos

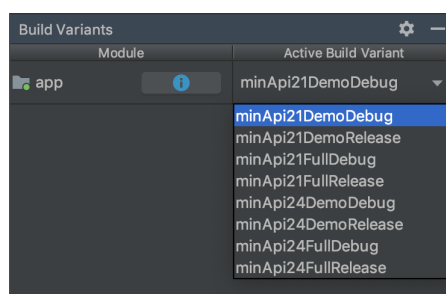


Imagen 4. Identificar variables de compilación.

De esta forma se pueden tener APK optimizados para cada plataforma para distintas, por ejemplo, para versiones de API o manejar versiones de una app con modelo *freemium*.

Ejercicio

Construir el APK de desarrollo y producción para un proyecto con versiones mínimas de API 24 y 26. Además, el nombre del APK de desarrollo debe tener el sufijo **-dev** usando la documentación oficial.

Instrucciones:

- Crear un proyecto usando el template de Actividad vacía
- Editar el archivo build.gradle a nivel de módulo para agregar: Build Type para debug
- Asignar el sufijo -dev al nombre de la versión usando la documentación oficial de BuildType
- Verificar Build Type para release
- Agregar la variable *flavorDimensions* para usar en los flavor
- Flavor para versión mínima de API 24 Flavor para versión mínima de API 26
- Sincronizar el proyecto
- Verificar que las Build Variants generadas correspondan a las indicadas

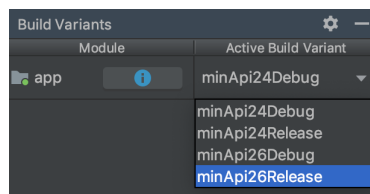


Imagen 5. Ejercicio.

Solución:

1. Crear un proyecto usando el template de Actividad vacía
2. El archivo build.gradle queda:

```
android {  
  
    defaultConfig { ... }  
    buildTypes {  
        release {  
            minifyEnabled false  
            proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'  
        }  
  
        debug {  
            debuggable true  
            versionNameSuffix "-debug"  
        }  
    }  
  
    flavorDimensions "api"  
    productFlavors {  
        minApi24 {  
            dimension "api"  
            minSdkVersion 24  
        }  
  
        minApi26 {  
            dimension "api"  
            minSdkVersion 26  
        }  
    }  
    ...  
}
```

3. Sincronizar el proyecto para generar las build variantsVerificar que las Build
4. Variants generadas correspondan a las indicadas

Dependencias

El sistema de compilación administra las dependencias locales y remotas en forma automática. Basta con definir las dependencias en el build.gradle para que sean sincronizadas desde los repositorios indicados. Las dependencias son agregadas al archivo build.gradle como una tarea independiente usando **dependencies**

```
apply plugin: 'com.android.application'

android { ... }
dependencies {
    // Dependencia a módulo local llamado mylibrary
    implementation project(":mylibrary")

    // Dependencia de binarios alojados localmente en el directorio libs
    implementation fileTree(dir: 'libs', include: ['*.jar'])

    // Dependencia a un binario remoto
    implementation 'cl.desafiolatam.android:cool-app:4.2'
}
```

Al definir la dependencia a un binario remoto de esta forma, es una abreviación de:

```
implementation group: 'cl.desafiolatam.android', name: 'cool-app', version: '4.2'
```

Esto significa que agrega una dependencia a la versión **4.2** de la aplicación **cool-app** dentro del grupo del espacio de nombres de **cl.desafiolatam.android**

Firmas

El sistema de compilación permite definir configuraciones de firma para firmar los APK automáticamente durante el proceso de compilación. Para la versión de depuración, el proceso de compilación firma el APK con una clave y un certificado predeterminados, evitando la solicitud de contraseña durante la compilación. Para la versión de lanzamiento, se debe definir explícitamente una configuración de firma y credenciales.

ProGuard

Durante la compilación se puede especificar un conjunto de reglas *ProGuard* diferentes para cada variable de compilación. Proguard es una herramienta de línea de comandos que encoge, optimiza y ofusca código

Con cada compilación, ProGuard crea los siguientes archivos:

- **dump.txt**: Describe la estructura interna de todos los archivos de clase del APK.
- **mapping.txt**: Proporciona una traducción entre la clase original y la oculta, el método y los nombres de campos. Se sobrescribe cada vez que se construye el proyecto
- **seeds.txt**: Indica las clases y los miembros que no se ocultaron.
- **usage.txt**: Indica el código que se quitó del APK.

Estos archivos se guardan en `<module-name>/build/outputs/mapping/release/`.

El propósito de ofuscar el código es reducir su tamaño acortando los nombres de clases, métodos y variables utilizados en la app usando R8. Un ejemplo de código ofuscado:

```
androidx.appcompat.app.ActionBarDrawerToggle$DelegateProvider -> a.a.a.b:  
androidx.appcompat.app.AlertController -> androidx.appcompat.app.AlertController:  
    android.content.Context mContext -> a  
    int mListItemLayout -> O  
    int mViewSpacingRight -> I  
    android.widget.Button mButtonNeutral -> w  
    int mMultiChoiceItemLayout -> M  
    boolean mShowTitle -> P  
    int mViewSpacingLeft -> j  
    int mButtonPanelSideLayout -> K
```

Donde se reemplaza **androidx.appcompat.app.ActionBarDrawerToggle\$DelegateProvider** con la abreviación **a.a.a.b** disminuyendo el tamaño de las líneas y de la app

La ofuscación no elimina código de la app, pero puede ahorrar espacio significativo al disminuir el tamaño de los archivos DEX que contienen las abundantes clases, métodos y campos. Sin embargo, la ofuscación renombra diferentes partes del código, así como algunas tareas y hace que inspeccionar el *stack trace* requiera herramientas adicionales. La documentación oficial explica cómo *decodificar un stack trace ofuscado*.

Compatibilidad con varios APK

El sistema de compilación permite compilar automáticamente diferentes APK que contengan solo el código y los recursos necesarios para una determinada densidad de pantalla, personalizando el instalador y disminuyendo su tamaño

Proceso de compilación

En el proceso de compilación intervienen herramientas y procesos que convierten el proyecto en un APK

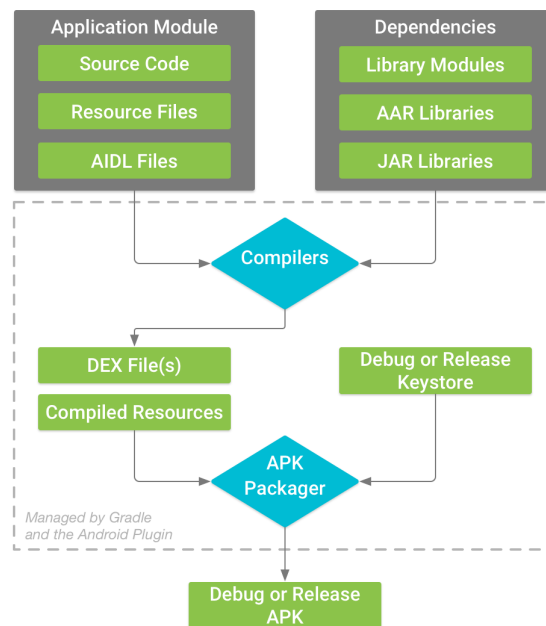


Imagen 6. Proceso de compilación.

Los pasos que sigue el proceso son los siguientes:

1. Los compiladores convierten:
 - El código fuente en archivos DEX que incluyen el código en bytes que se ejecuta en el dispositivo
 - Los recursos en recursos compilados
2. El empaquetador combina los archivos DEX y los recursos compilados en un solo APK
3. Para que este APK pueda ser instalado y utilizado debe ser previamente firmado para 1 de estos propósitos:
 - Versión de depuración, para hacer pruebas
 - Versión de producción o lanzamiento
4. Antes de generar el APK final, el empaquetador usa *zipalign* para optimizar la utilización de memoria al ejecutarse en el dispositivo

El resultado de este proceso es un APK de depuración o producción listo para ser instalado en los dispositivos

Por último, en la documentación oficial hay una página con algunos consejos (*tips*) para utilizar Gradle que complementan lo aprendido, al igual que la documentación oficial.

Preparando la publicación de tu app

Competencias

- Preparar la app para lanzamiento.
- Reconocer materiales y recursos necesarios para armar la app.
- Configurar una app en modo release.
- Firmar una app en modo release.
- Versionar una app para distribuirse la tienda de aplicaciones.
- Conocer otros canales de distribución.

Introducción

Luego de diseñar, codificar y probar una app se llega al paso final que es ponerla a disposición de los usuarios.

Para publicar la app se siguen 2 pasos:

1. Preparar la app de lanzamiento, compilando una versión de lanzamiento que los usuarios puedan descargar e instalar en sus dispositivos Android
2. Realizar el lanzamiento de la app para los usuarios, donde se publicita, vende y distribuye la versión de lanzamiento para los usuarios



Imagen 7. Pasos para publicar una app release.

De las opciones disponibles para el lanzamiento de la app, utilizar la tienda oficial de Android es la más utilizada, están disponibles Amazon Appstore y APPGallery de Huawei entre otras como tiendas alternativas (y complementarias) para hacer el lanzamiento masivo a los usuarios. Si se quiere entregar a un grupo de usuarios más reducido es posible también hacer el lanzamiento compartiendo directamente el APK

Este capítulo está enfocado en cómo preparar y lanzar (*release*) un app para los usuarios, abordando el proceso y requerimientos, explorando las distintas alternativas.

Preparación de la app de lanzamiento

En esta etapa se construye y prueba la app antes de hacer el lanzamiento.



Imagen 8. Tareas previas al lanzamiento release.

Reunir materiales y recursos

La app utiliza varios elementos secundarios y cómo mínimo se debe proporcionar la clave criptográfica y el ícono de la app.

Clave criptográfica

Cada app para lanzamiento debe ser firmada digitalmente con un certificado que pertenece al desarrollador de la app, manejado con una clave privada. Luego, el sistema puede identificar el autor de una app y establecer sistemas de confianza con otras apps.

Íconos de la app

Cada app debe contar un ícono de aplicación y que cumpla con las *pautas definidas para íconos*. El ícono ayuda a los usuarios a identificar la app en la pantalla de inicio, en el launcher de aplicaciones o en otras secciones. Además, los servicios de publicación muestran ese ícono en sus búsquedas

Licencia

Es una buena práctica incluir un contrato de licencia para el usuario final (EULA) puede ayudar a proteger al desarrollador, organización y propiedad intelectual. Algunos ejemplos de estas políticas de privacidad las podemos ver en Facebook o Instagram o podemos usar alguna *política estándar*.

Configurar release app

Hay ciertas recomendaciones, que si bien son opcionales optimizan la app final

Elegir un buen nombre de paquete

El nombre del paquete es usado como identificador en la app y se usa para toda la vida útil de la app sin poder cambiarlo, por lo que es importante elegir un buen nombre de paquete

Este nombre es definido en el manifiesto usando el atributo *package*.

Desactivar el registro y la depuración

Es importante desactivar el registro e inhabilitar la depuración antes de construir el APK de release para lograr una app más fluida.

Para inhabilitar la depuración se debe asignar la variable *android:debuggable* en *false* en el manifiesto. Por defecto, la construcción de Release maneja este valor en false

Se deben eliminar las llamadas a *Log* del código para eliminar el registro. Se puede extender la clase *Log* para activar/desactivar logs o utilizar una biblioteca como *Timber* que tiene este comportamiento incorporado

Limpiar el proyecto

Eliminar todos los archivos que no sean utilizados por la app en los directorios *res*, *lib*, *assets* para empaquetar solamente lo necesario por la app. El proyecto puede ser analizado por Android Studio usando *Analyze > Inspect Code*, que indica posibles mejoras al código, entre esas, encontrar los archivos no referenciados

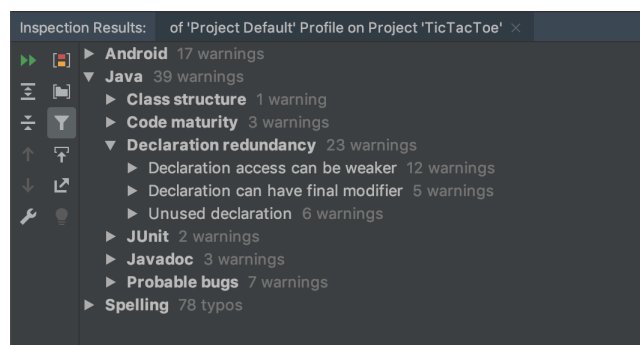


Imagen 9. Eliminar archivos.

Actualizar el manifiesto

Se debe comprobar que la configuración del manifiesto y los elementos de compilación sean los correctos.

Elemento

La aplicación debe declarar exclusivamente los permisos mínimos necesarios para su funcionamiento, evitando incluir permisos que no serán utilizados.

Atributos `android:icon` y `android:label`

Estos valores se encuentran en el elemento `<application>` y son utilizados en la app como los valores por defecto a utilizarse en toda la app.

Atributos `android:versionCode` y `android:versionName`

Estos valores son del elemento `<manifest>` y son utilizados en el control de versiones de la app por parte de las tiendas de aplicaciones.

Atributos `android:minSdkVersion` y `android:targetSdkVersion`

Se encuentran en el elemento `<uses-sdk>` para indicar las versiones adecuadas para la app.

Actualizar las rutas a los servidores

Si la app accede a servidores remotos hay que verificar que las URLs utilizadas por la app correspondan a las de producción y no otros ambientes. Una forma de manejar esto es inyectando variables al manifiesto.

Compilar la release app

Con la configuración terminada, se puede compilar un APK de release, firmado y optimizado. El JDK incluye las herramientas para firmar el archivo APK: Keytool para administrar claves y certificados y *Jarsigner* para firmar y verificar la firma e integridad de los JAR firmados.

Android Studio permite construir el APK firmado usando Gradle, que también puede ser ocupado para automatizar el proceso completo de compilación.

Probar release app

Algunas pruebas básicas por hacer:

- Probar la app en al menos un dispositivo físico. Si bien los emuladores sirven mucho para el desarrollo, la real fluidez la veremos en un dispositivo físico
- Probar la app en al menos una pantalla de teléfono y una pantalla de tablet para verificar que los elementos visuales tengan el tamaño correcto
- Verificar el rendimiento de la app y la utilización de la batería sean aceptables

En la documentación oficial se entregan otros consejos sobre qué aspectos se deben probar.

Al terminar las pruebas y verificar la app de release tiene el comportamiento esperado, la app está lista para ser lanzada

Versionamiento de la app

El versionamiento de la app es un componente fundamental de la estrategia de actualización y mantenimiento de la app, principalmente:

- Las tiendas de aplicaciones utilizan la versión para determinar la compatibilidad y la actualización de la app
- Otras apps pueden realizar consultas al sistema respecto de la versión de la app para determinar compatibilidad y verificar las dependencias

La información de la versión es definida en el archivo build.gradle y luego reemplazada en el manifiesto durante la compilación. No se debe hacer directamente en el manifiesto porque va a ser sobrescrito y se anulará la configuración

La versión se configura con 2 atributos: `versionCode` y `versionName`

versionCode

Es un valor entero que se usa como número interno de versión, siendo el número más grande la versión mas nueva. Normalmente el `versionCode` comienza con valor 1 y se incrementa para cada versión subida a las tiendas de aplicaciones. Su valor máximo es 2100000000

El número es interno y no se le muestra al usuario, para eso se utiliza `versionName`

versionName

Es un texto usado exclusivamente como el número de versión que se le muestra al usuario. Al ser un string se puede referenciar a la versión usando versionamiento semántico `<major>.<minor>.<patch>` o cualquier otro descriptor de la versión

Por ejemplo, para el siguiente build.gradle

```
android {
    def version = "1.1"
    defaultConfig {
        ...
        versionCode 1
        versionName $version
    }

    productFlavors {
        demo {
            ...
            versionName "demo-$version"
        }
    }

    full { ... }
}
...
```

Se define un `versionName` por defecto en `defaultConfig`, que luego es sobrescrito en el flavor demo, usando `demo-$version` que se convierte en `demo-1.1` durante la compilación

Nivel de API

El nivel de API requerido por una app puede ser especificado en el archivo build.gradle. Al especificar el nivel mínimo requerido se asegura que la app se puede instalar exclusivamente en dispositivos que ejecuten una versión compatible

Hay 2 configuraciones asociadas al nivel de API

- **minSdkVersion:** versión mínima de la plataforma Android en la cual la app podrá ejecutarse, especificada por el identificador de nivel de API de la plataforma.
- **targetSdkVersion:** elemento que especifica el nivel de API con el cual la app está diseñada para funcionar. En algunos casos, esto permite que la app use elementos del manifiesto o comportamientos definidos en el nivel de API de destino, en vez de limitarse a usar únicamente aquellos definidos por el nivel mínimo de API.

Firma de la app

Android exige que todos los APK sean firmados digitalmente con un certificado antes de poder instalarse

Certificados y keystores

Un certificado de clave pública, también conocido como certificado digital o un certificado de identidad, contiene la clave pública de un par de claves públicas y privadas, y otros metadatos que identifican al propietario de la clave como el nombre, ubicación y compañía. El propietario del certificado guarda la clave privada correspondiente.

Al firmar un APK, la herramienta de firma adjunta el certificado de clave pública. El certificado de clave pública sirve como una "huella digital" que asocia de manera exclusiva el APK con la clave privada correspondiente. Esto permite a Android verificar que cualquier actualización futura a esa app sea auténtica y provenga del autor original. La clave que se usa para la creación de este certificado se llama **clave de firma de apps**.

Un **keystore** es un campo binario que contiene una o más claves privadas. Por esto es importante conservar la clave privada, ya que las apps deben ser firmadas con la misma clave y certificado durante su vida útil para que los usuarios puedan instalar versiones nuevas como actualizaciones de la misma app.



La pérdida de la clave privada de firma significa que la app no puede ser actualizada.

Imagen 10. Pérdida de la clave primaria.

Si la clave privada de firma es obtenida por personas ajenas al desarrollo de la app, se puede agregar código malicioso y subir una actualización. Por lo que hay que tener necesario tomar las precauciones necesarias en su administración

Firmar la release app

1. Seleccionar desde el menú **Build -> Generate Signed Bundle or APK**
2. En el cuadro de diálogo **Generate Signed Bundle or APK** se debe elegir una Android App Bundle o un APK

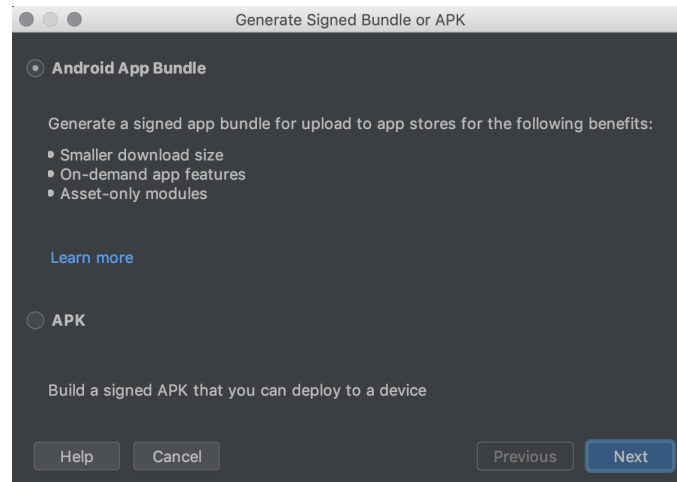


Imagen 11. Generate Signed Bundle or APK.

3. Si no se cuenta con un certificado, se puede crear uno seleccionando debajo del campo **Key store path**, la opción **Create new**.

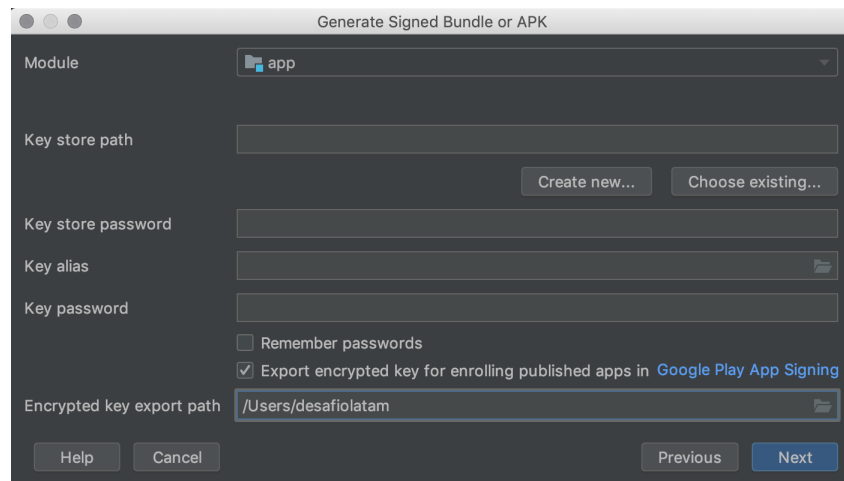


Imagen 12. Crear certificado.

4. En el diálogo **New Key Store** se debe proporcionar la información para identificar al dueño

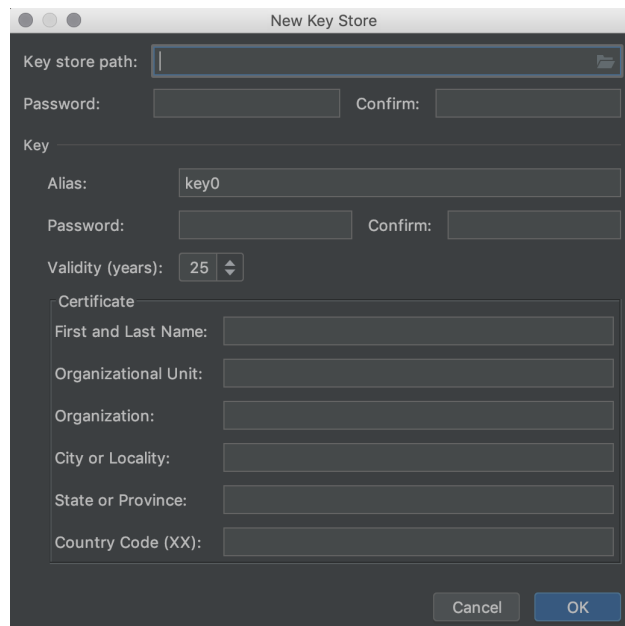


Imagen 13. Configuración de nuevo certificado.

- Key store path: ubicación en la que se va a crear la Key Store
- Password: la contraseña de la Key Store
- Alias: Nombre de identificación para la key (clave)
- Password: Contraseña segura para la key que debe ser distinta a la de la Key Store. Si bien las contraseñas pueden ser iguales, se recomienda que sean distintas
- Validity: Cantidad de años que dura la key que debe ser al menos 25 años
- Certificate: Datos personales que no se muestran en la app pero se incluyen como parte del APK
- Country Code: corresponde al código de 2 letras definidos en la *ISO 3166* como código de país. En el caso de Chile es **cl**

5. Luego de crear la Key Store, la siguiente parte es seleccionar la Build Variant adecuada. En este caso **release**, aunque es posible seleccionar más de una Build Variant

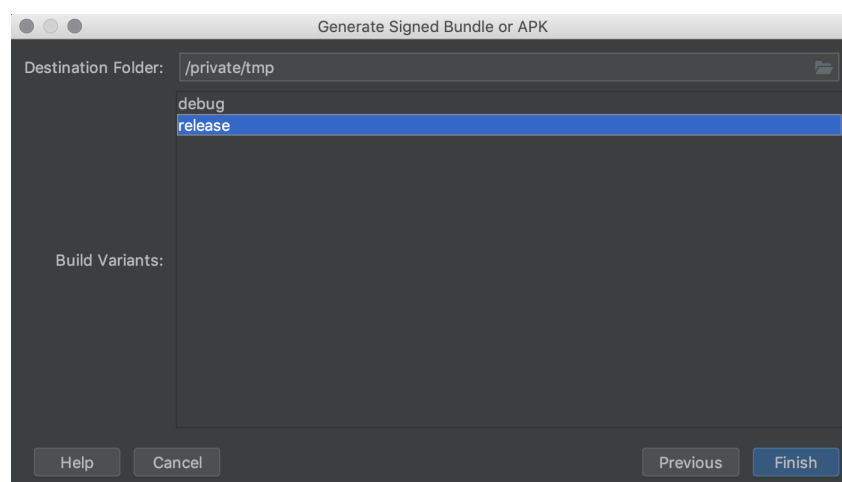


Imagen 14. Seleccionar Buold Variant.

6. Al finalizar el proceso, se ejecuta la construcción con Gradle que al finalizar muestra un mensaje indicando la creación del Bundle

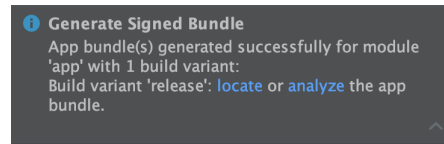


Imagen 15. Creación del Bundle.

Generar automáticamente una release app

Con Android Studio es posible generar automáticamente una release app seleccionando desde el menú **File > Project Structure** y agregar una nueva **Signing Configs** al módulo entregando la Key Store, Key Alias y las contraseñas correspondientes.

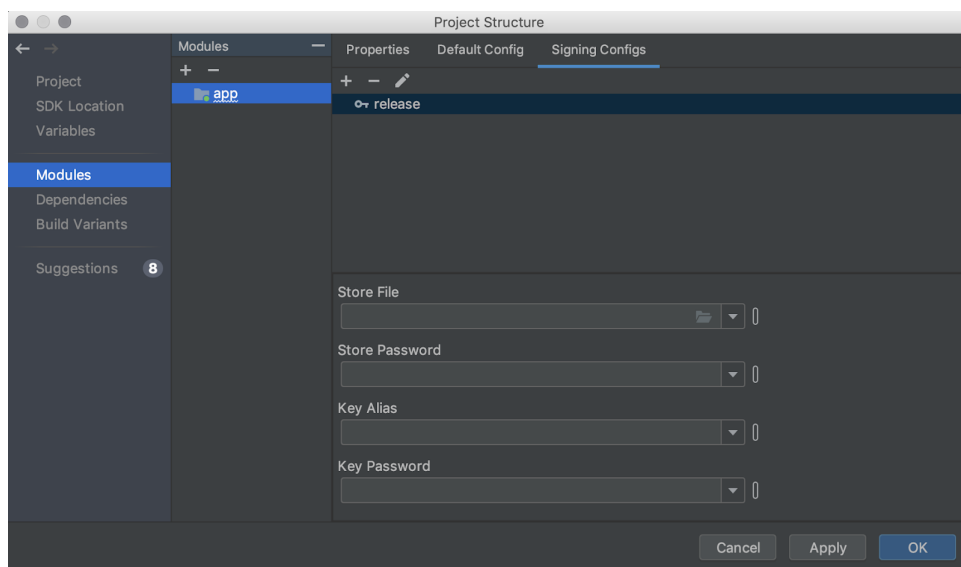


Imagen 16. Generar automáticamente una release app.

Tanto la Keystore como las contraseñas deben ser administradas con sentido común pensando en la seguridad y **no deben ser compartidas ni agregadas al control de versiones**.

Luego, en la pestaña Build Variants se debe editar el Build Type de release indicando en el campo Signing Config la configuración recién creada

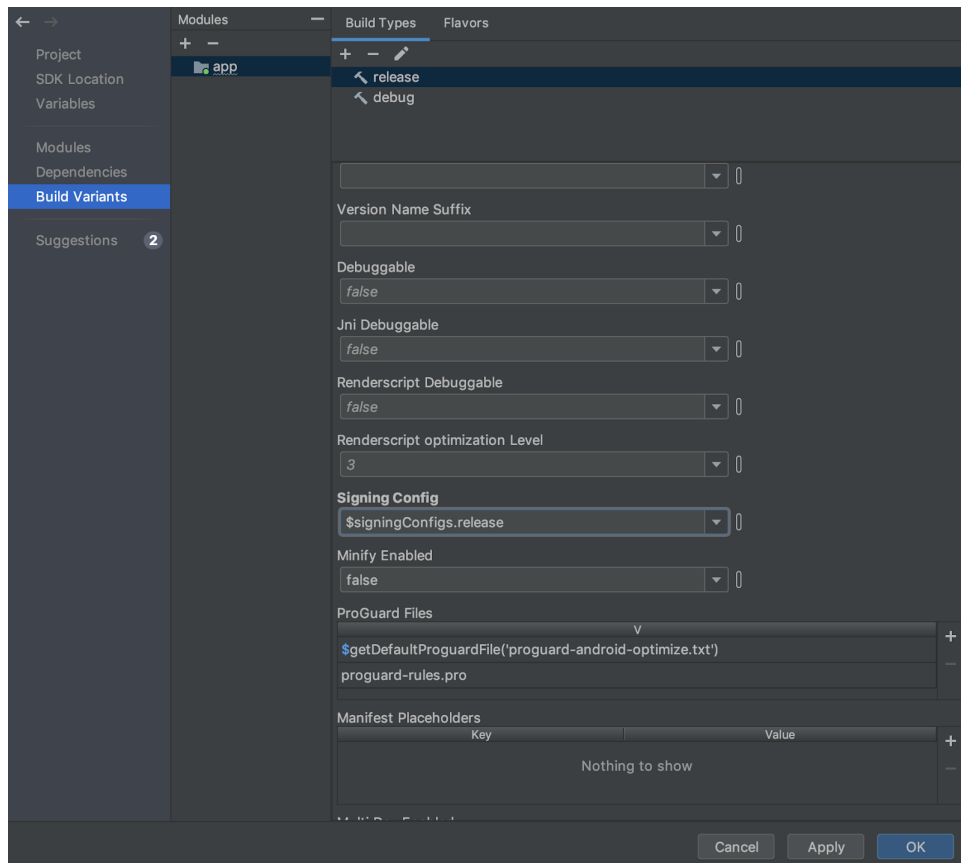


Imagen 17. Editar el Build Type.

Firma de apps de Google Play

Con la firma de apps de Google Play, Google gestiona y protege la clave de firma de la app y la utiliza para firmar los APK. Para usar la firma de apps de Google Play se utilizan 2 claves:

- La firma de apps
- La firma de subida

Al usar la firma de apps de Google Play se encripta y exporta la clave de firma de apps usando Play Encrypt Private Key (PEPK) de Google Play, para subirla a la infraestructura de Google. Al momento de la publicación, la app es firmada con la **clave de subida** y subida a Google Play. Luego, usando el **certificado de subida** para verificar la identidad del autor se firma el APK con la **clave de firma** de apps para su distribución

Si la clave de subida está comprometida o se pierde es posible anularla y crear una nueva. Como la clave de firma de apps está en los servidores de Google se seguirá usando y la app puede seguir siendo actualizada

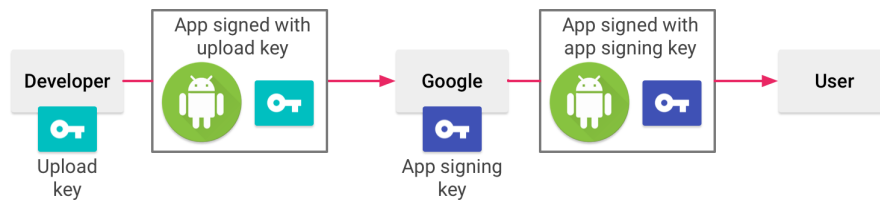


Imagen 18. Firma de apps en Google Play.

Para utilizar esta opción, al generar un nuevo signed bundle de release se debe seleccionar la opción de exportar las claves encriptadas para usar Google Play App Signing.

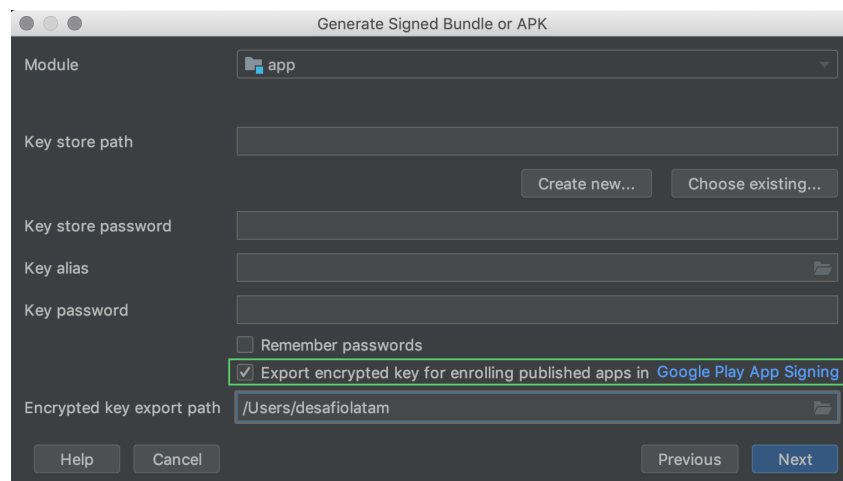


Imagen 19. Exportar claves encriptadas.

Otras formas de distribución para apps

Android brinda protección a los usuarios contra descargas e instalaciones inadvertidas de apps desde otras ubicaciones que no sean Google Play (confiable).

Para poder usar otras formas de distribución distintas a la tienda oficial es necesario configurar el dispositivo Android para que permita la instalación desde fuentes desconocidas.

Activar fuentes desconocidas

Se debe activar la opción para permitir fuentes desconocidas desde el menú de Ajustes -> Seguridad. Estas instalaciones se bloquean hasta que el usuario lo permita activando explícitamente **fuentes desconocidas** en Ajustes > Seguridad.

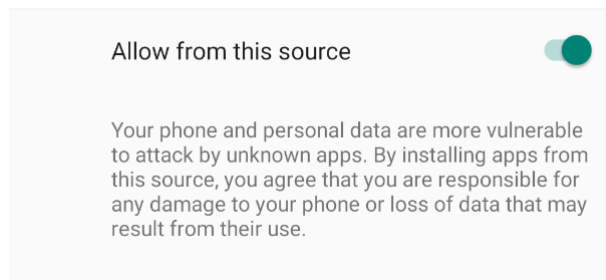


Imagen 20. Activar fuente desconocida.

Distribuir usando Google Drive

Por razones de seguridad ya no es posible enviar archivos de extensión APK por correo usando Gmail, sin embargo, cuando el APK está listo, se puede compartir desde Google Drive. Al seleccionar desde el dispositivo Android el APK compartido será instalado

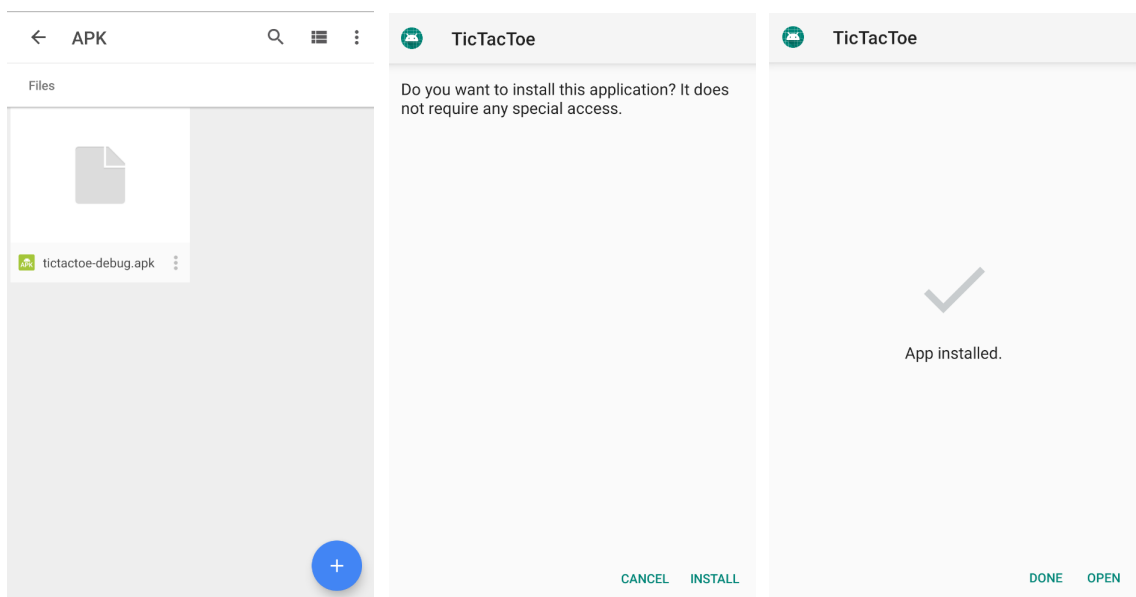


Imagen 21. Grupo de imágenes - Distribuir usando Google Drive.

Distribuir usando un servidor

Al igual que distribuirlo usando Google Drive, el APK se puede alojar en un servidor para que los usuarios puedan acceder desde un dispositivo Android para la instalación con la misma pantalla de confirmación y postinstalación que al instalar de Google Drive