Instituto Profesional AIEP Spa.
Dirección Nacional de Educación Continua
Programa Talento Digital 2021

Curso desarrollo de aplicaciones móviles Android Trainee

Módulo 4: Desarrollo de aplicaciones móviles Android Kotlin (4 unidades)

MANUAL DEL PARTICIPANTE

Temario de unidades

Unidad 1

Introducción a Kotlin

1.1 Kotlin

Competencias:

- Aprender qué es Kotlin y su relación con Jetbrains

Un Lenguaje llamado Kotlin

Muchos desarrolladores de software piensan que Kotlin es un lenguaje muy nuevo, esto debido a que se popularizó el año 2017 cuando fue anunciado por Google como un lenguaje más para desarrollar aplicaciones Android, junto con Java y C++.

Pero Kotlin no nace el 2017, este lenguaje fue anunciado el año 2011 por los desarrolladores de JetBrains como un proyecto que corre sobre la máquina virtual de Java y que puede interoperar con código Java.

El proyecto empezó el año 2010 debido a una necesidad del equipo de IntelliJ IDEA, este IDE está desarrollado en Java, por lo que necesitaban algo mejor que Java y que fuera igual de rápido, existía un posible candidato SCALA, pero se descarto debido a que era mucho para lo que ellos necesitaban, por lo que al no existir nada que solucione sus problemas con Java, crearon su propio lenguaje de programación.

Hoy en día Kotlin es un lenguaje que permite trabajar sobre la JVM, Java Script y Kotlin Native.

Hoy ya Kotlin es uno de los lenguajes más queridos por los desarrolladores según StackOverFlow y Google ya anunció este año a Kotlin-first como el lenguaje más utilizado para desarrollar aplicaciones en Android.

Instalación

Competencias:

- Aprender a instalar Kotlin

Introducción

Es muy importante recordar que Kotlin trabaja sobre la máquina virtual de Java, por lo que es un requisito que en nuestro computador tengamos instalado el JDK de Java.

Requisitos para Instalar IntelliJ

Instalando IntelliJ IDEA

Existen 2 versiones de IntelliJ que podemos utilizar, esta la versión “Community” que es gratuita y la versión “Ultimate” que es pagada para las empresas y se puede obtener una licencia para estudiantes o profesores.

Para Mac

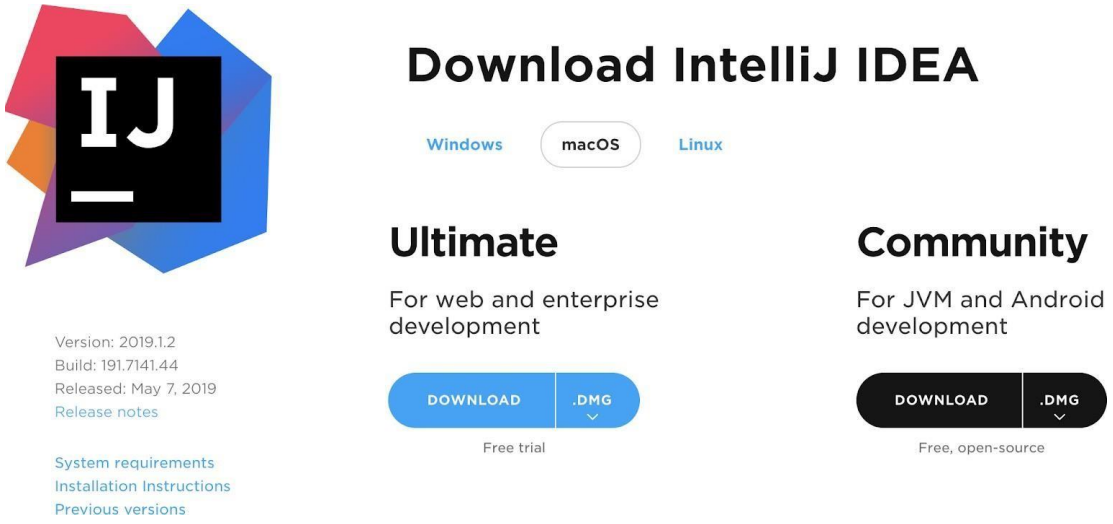


Imagen 1. Descargar IntelliJ IDEA para Mac.

Instalando en Mac

- **Paso 1:** Arrastramos el icono de “IntelliJ IDEA CE” hacía la carpeta de Applications.

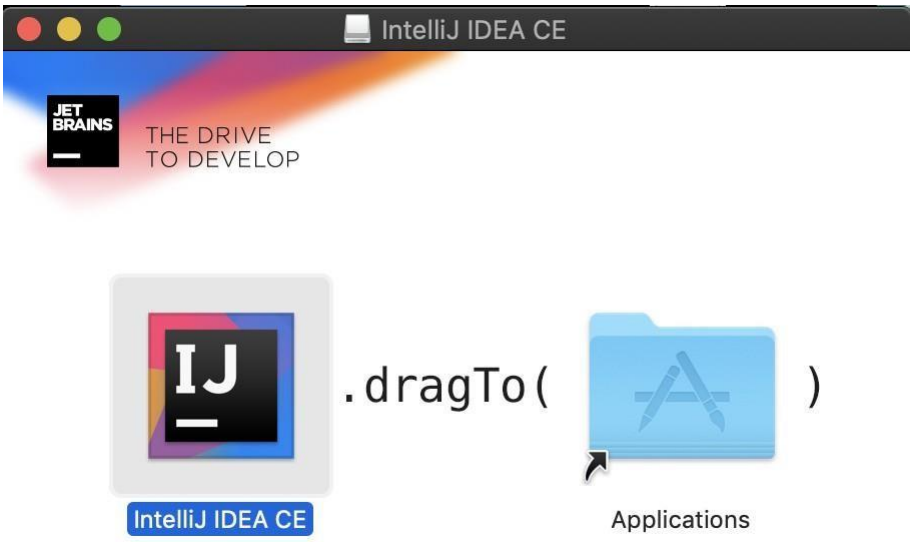


Imagen 2. Instalar IntelliJ IDEA en Mac - Paso 1.

- **Paso 2:** Abrimos la aplicación.

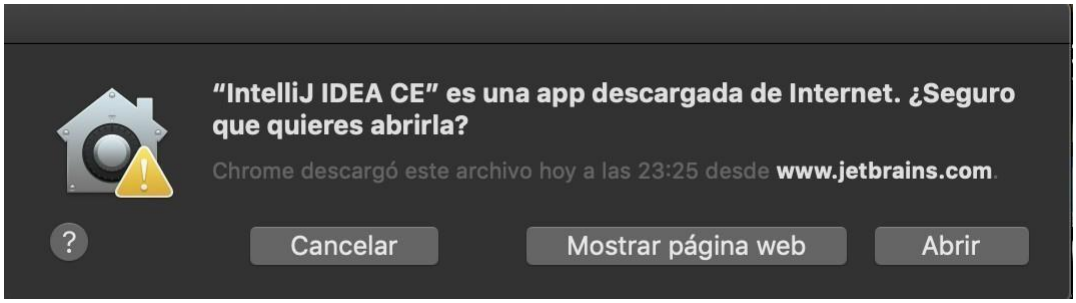


Imagen 3. Instalar IntelliJ IDEA en Mac - Paso 2.

- **Paso 3:** Presionamos el botón “OK” por defecto.

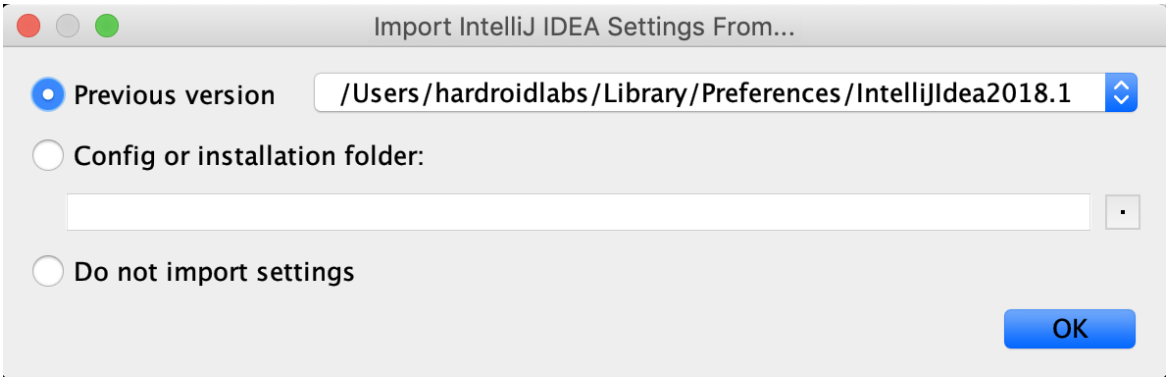


Imagen 4. Instalar IntelliJ IDEA en Mac - Paso 3.

- **Paso 4:** Presionamos el botón “Create New Project” y listo.

Imagen 5. Instalar IntelliJ IDEA en Mac - Paso 4.

Para Windows

Imagen 6. Descargar IntelliJ IDEA para Windows.

Instalando en Windows

- **Paso 1:** Abrimos el instalador de IntelliJ y presionamos el botón “next”.

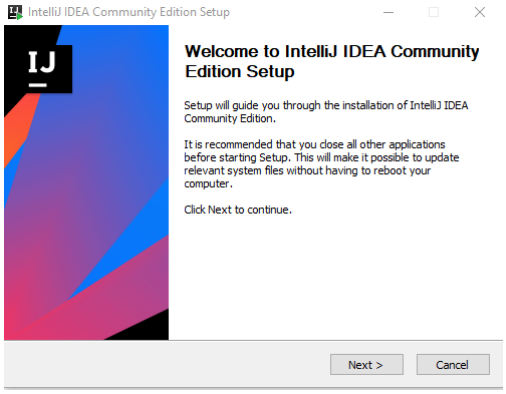


Imagen 7. Instalar IntelliJ IDEA en Windows - Paso 1.

- **Paso 2:** Seleccionamos la carpeta donde se instalará nuestro IDE, recomiendo dejar la ruta por defecto y presionamos el botón “Next >”.

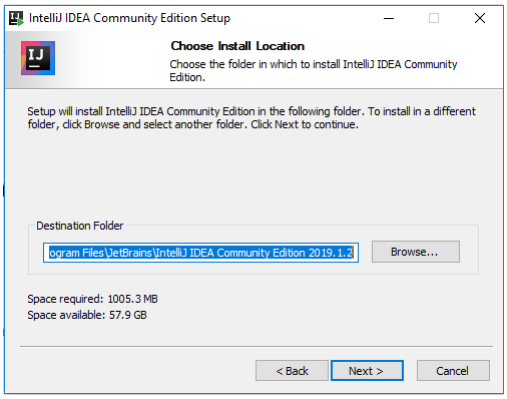


Imagen 8. Instalar IntelliJ IDEA en Windows - Paso 2.

- **Paso 3:** Seleccionamos a criterio las opciones de instalación, recomiendo seleccionar los archivos ".java", "groovy" y ".kt" y presionamos el botón "Next >".

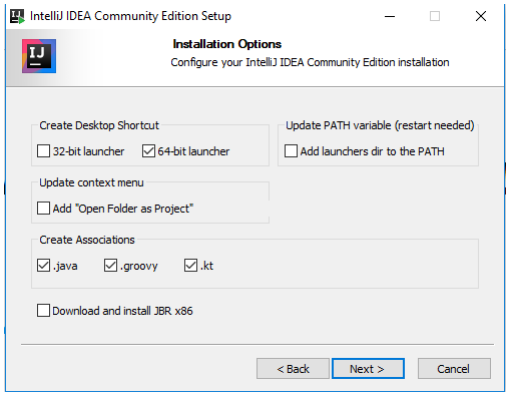


Imagen 9. Instalar IntelliJ IDEA en Windows - Paso 3.

- **Paso 4:** Presionamos el botón "Install".

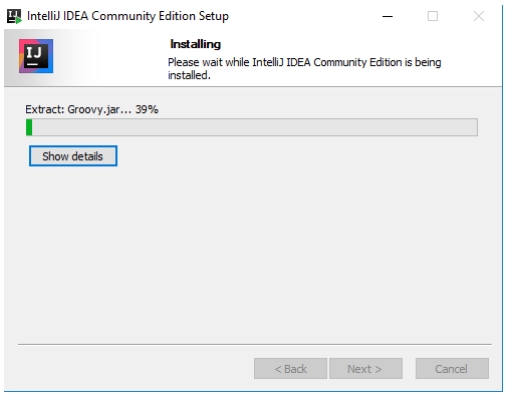


Imagen 10. Instalar IntelliJ IDEA en Windows - Paso 4.

- **Paso 5:** Presionamos finalizar, podemos seleccionar la opción "Run IntelliJ" y nuestro IDE se abrirá.

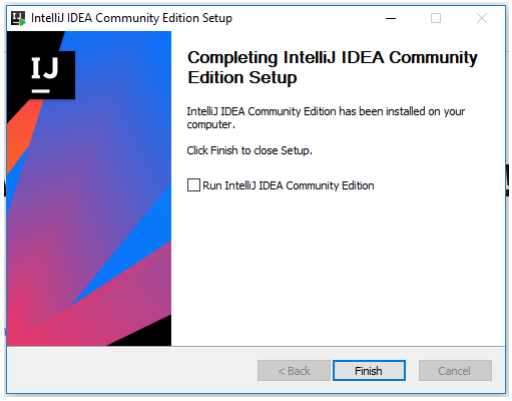


Imagen 11. Instalar IntelliJ IDEA en Windows - Paso 5.

- **Paso 6:** Seteamos las configuraciones presionando el botón “OK”.

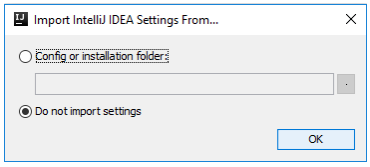


Imagen 12. Instalar IntelliJ IDEA en Windows - Paso 6.

- **Paso 7:** Aceptamos las condiciones de privacidad de IntelliJ y presionamos el botón “Continue”.

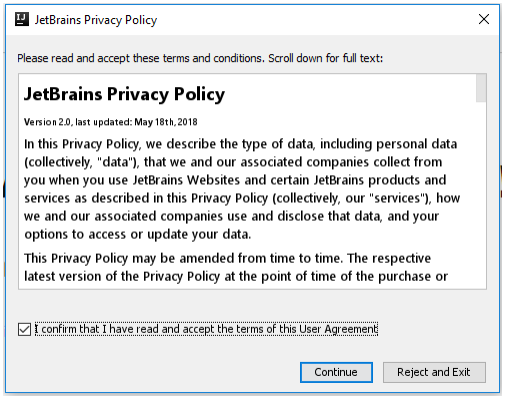


Imagen 13. Instalar IntelliJ IDEA en Windows - Paso 7.

- **Paso 8:** Seleccionamos el tema que más te guste, en mi caso, “Darcula”.

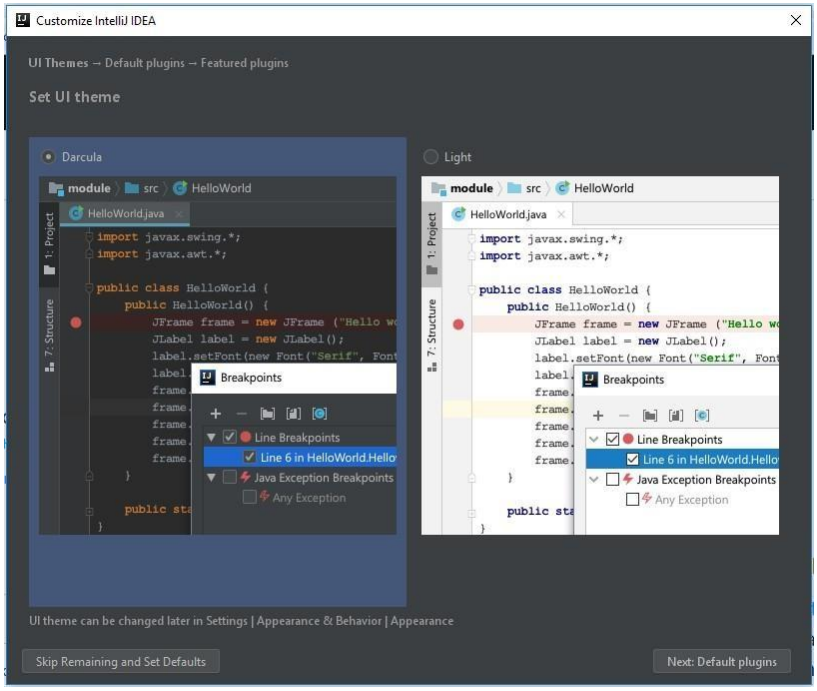


Imagen 14. Instalar IntelliJ IDEA en Windows - Paso 8.

Ejercicio: “Hola Mundo en Kotlin”

Ahora que tenemos instalado nuestro IntelliJ IDEA, ya podemos crear nuestro primer proyecto en Kotlin, vamos a hacer un “Hola Mundo!”

- **Paso 1:** Abrimos nuestro IDE y presionamos el botón “Create New Project” .

Imagen 15. Crear nuevo proyecto

- **Paso 2:** Hacemos click en: Java -> Kotlin ->JVM y presionamos el botón “Next”.

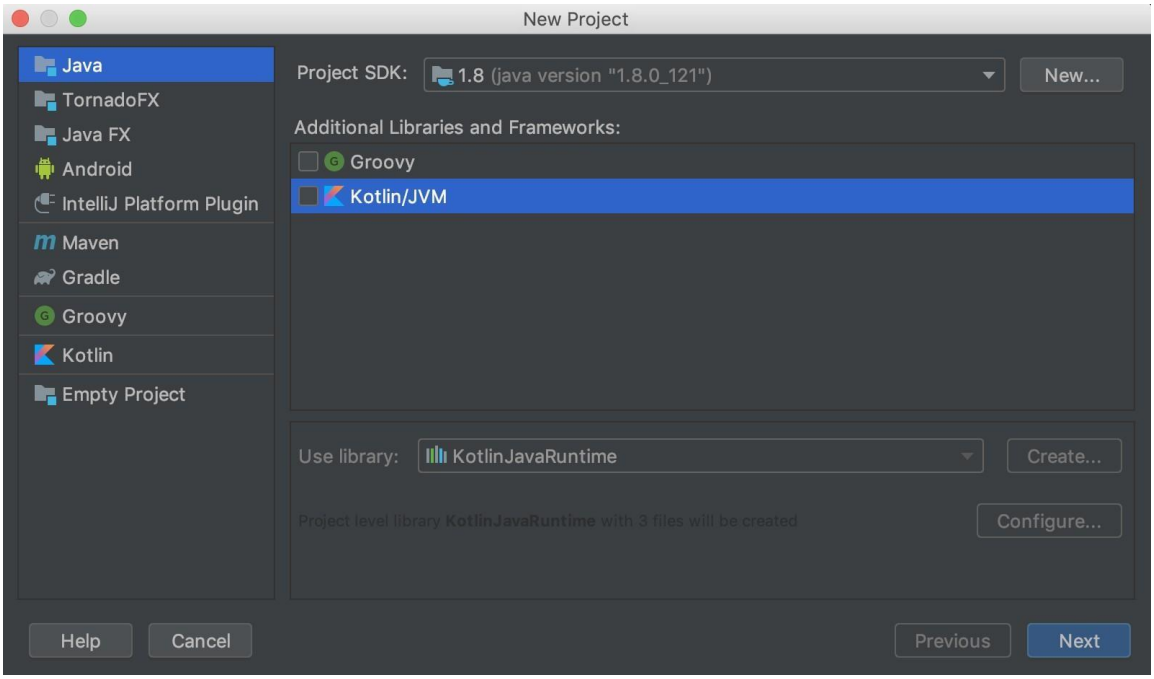


Imagen 16. Seleccionar librerías y frameworks

- **Paso 3:** Presionamos el botón “Next” .

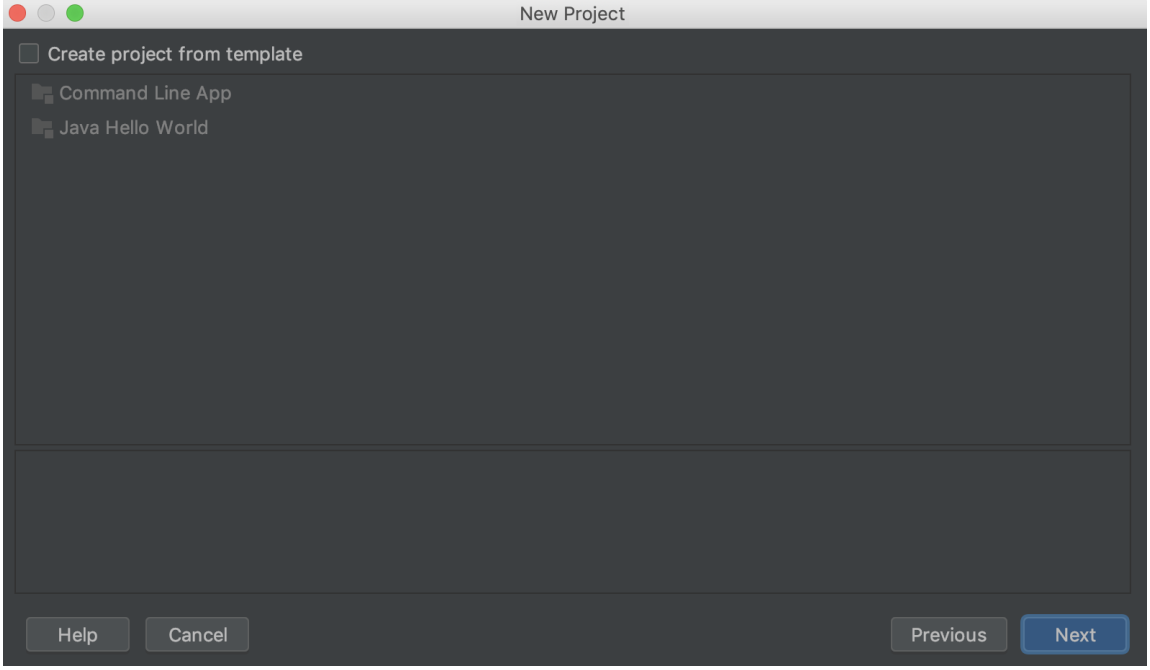


Imagen 17. Crear proyecto desde plantilla

- **Paso 4:** Agregamos el nombre del proyecto, nombre del módulo y presionamos el botón “Finish”.

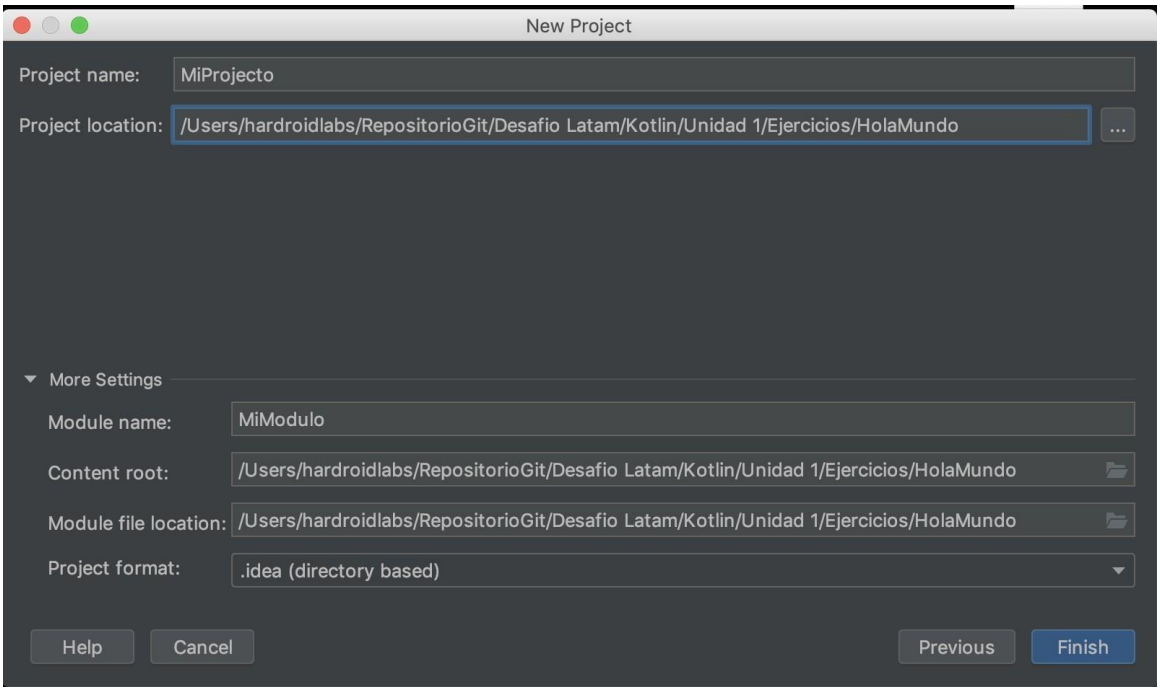


Imagen 18. Agregar nombre al proyecto y módulo

Project name: Este será el nombre de nuestro proyecto, se creará una carpeta con este nombre.

Project location: Esta es la ruta donde se crea nuestro proyecto.

Module name: Este corresponde al nombre del módulo, un proyecto Kotlin puede contener muchos módulos, por lo que en este caso se crea un proyecto con un módulo llamado “MiModulo”

- **Paso 5:** Botón derecho en la carpeta “**src**” y seleccionamos “New/Kotlin File/Class”.

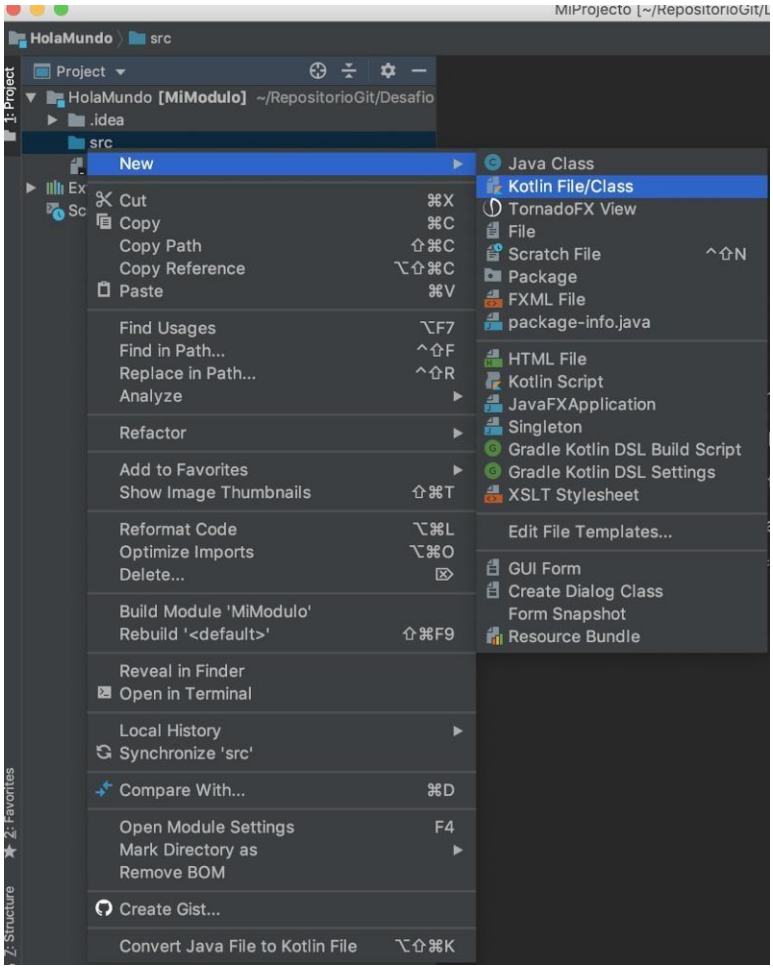


Imagen 19. Agregar nuevo archivo/clase Kotlin

- **Paso 6:** Escribimos el nombre del archivo Kotlin y presionamos el botón “OK”.

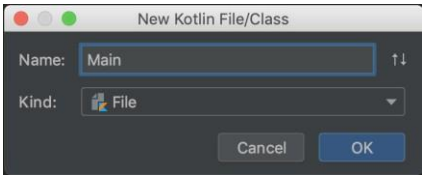


Imagen 20. Agregar nombre al nuevo archivo/clase

- **Paso 7:** Abrimos el archivo “Main.kt” y escribimos la siguiente función “main”. Esta función es equivalente al método main de Java “public static void main”.

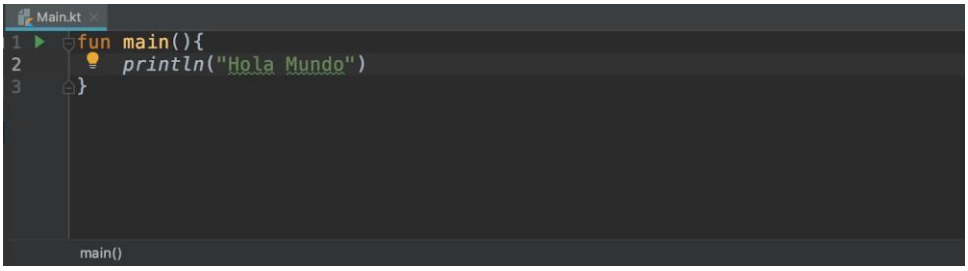


Imagen 21. Archivo `main.kt`

- **Paso 8:** Ejecutamos el método “main” haciendo clic en el botón Play o ejecutamos la acción que se indica en nuestro IDE.

Imagen 22. Ejecutar acción

- **Paso 9:** Vemos el saludo en la consola de nuestro IDE.

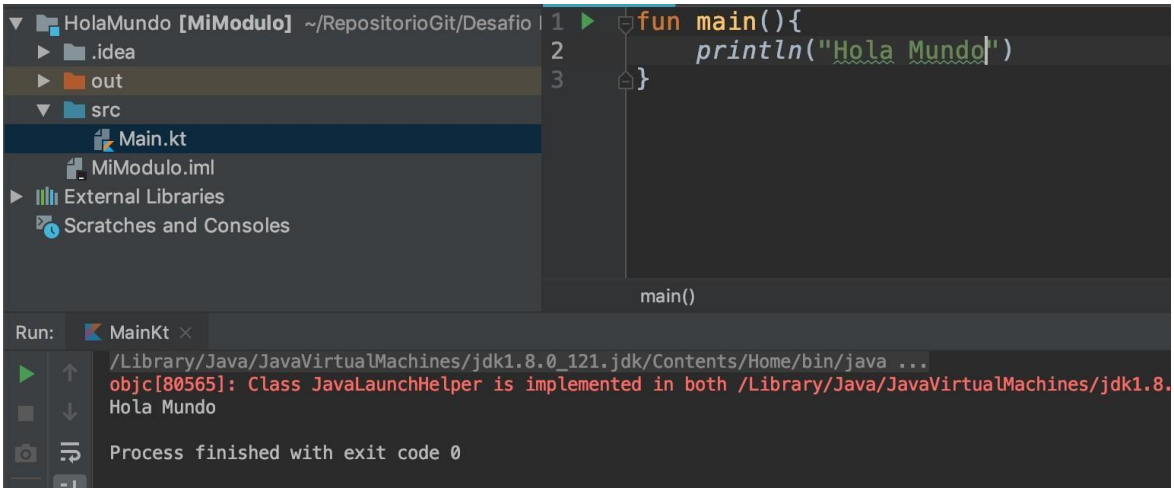


Imagen 23. Ejecutar código

Variables

Competencias:

- Conocer las variables en Kotlin
- Declarar variables en Kotlin
- Declarar variables por inferencia en Kotlin
- Aprender formato de caracteres en Kotlin

Introducción

Todo el código necesita de variables, en este capítulo aprenderemos cómo funcionan las variables en Kotlin y como se deben declarar para ser utilizadas, aprenderemos sobre el uso de variables por inferencia en Kotlin y cómo usar una variable mutable o inmutable.

Variables

Cuándo pienses en variables imaginate que estas son vasos, existen distintos tipos de vasos de distinta forma y distinto color, pero todas tienen algo en común, un vaso contiene algo.

Declarar variables en Kotlin

Para poder declarar una variable en Kotlin, necesitamos crear antes una función, para los ejemplos utilizaremos la función “**main**”, que profundizaremos en el capítulo de funciones.Además de crear una función, necesitamos saber 3 cosas:

1. Tipo de variable (“var” o “val”)
2. Nombre de la variable
3. Tipo de valor o clase(Numérico, Cadena de Caracteres, Lógico, etc.)

```
fun main() {
  val numero : Int
  var variable : TipoVariable
}
```

Variables por inferencia en Kotlin

En Kotlin podemos omitir la declaración del tipo de valor, y esto lo hacemos inicializando la variable con un valor determinado.

```
fun main() {
  val nombre = “Goro Daimon”
  val edad = 22
}
```

Acá podemos ver como omitimos el tipo de valor en la declaración de la variable “*nombre*” y “*edad*”.

En Kotlin al asignarle un valor a la variable, el compilador determina automáticamente el tipo de valor que estas variables tendrán, para este ejemplo la variable *“nombre”* es String y la variable *“edad”* es Int.

Diferencias entre un val y var

En Kotlin existen dos tipos de variables, las variables mutables y las variables inmutables.

Las variables inmutables se declaran como "*val*" y contienen un valor que "**No**" puede ser modificado.**Las variables mutables** se declaran como "*var*" y estas variables "**Si**" pueden ser modificadas.

```
fun main() {
  val numeroFijo = 10
  var numeroVariable = 10
  numeroFijo = 12
}
```

En este código podemos ver la declaración de dos variables, estas dos variables son de tipo entero (Int), pero solo la variable "númeroVariable" puede ser modificado su valor.

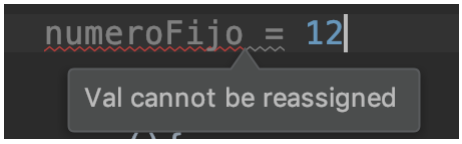


Imagen 24. Declaración de variables

En esta imagen podemos ver el error que nos da nuestro IDE al tratar de modificar una variable declarada como "val".

String Template en Kotlin

Una de las mejoras importantes que trae Kotlin como lenguaje de programación, es la facilidad de trabajar con variables String, a esto se les conoce como "**String Template**".

Cuando estemos utilizando una variable String podemos utilizar el signo "**$**" para concatenar valores.

```
fun main() {
  val x = "valor string"
  val template = "Concatenando **** $x **** valores"
  println(template)
}
```

Imagen 25. Código para concatenar valores

Esto nos facilita mucho el concatenar distintos valores String, dejando un código fácil de entender.

```
Concatenando **** valor string **** valores

Process finished with exit code 0
```

Imagen 26. Resultado de concatenar valores

También podemos llamar a una función del lenguaje o de una variable utilizando el signo " **$**" seguido del uso de llaves "**{ }**". Dentro de estas llaves podemos hacer la llamada a la función que necesitemos.

```
fun main() {
  val x = "valor string"
  val template = "Tiene ${x.count()} caracteres"
  println(template)
}
```

```
Tiene 12 caracteres

Process finished with exit code 0
```

Imagen 27. Llamada a función

Ejercicio1 : “Probando las Variables”

1. Crear una función main
2. Dentro de la función main declarar una variables mutable y una variable inmutable de forma inteligente.
3. Modifica el valor de la variable mutable.
4. Imprime el valor por pantalla de las dos variables.

Método main

```
fun main() {
  var a = 10
  val e = "Goku"
  a = 20
  println("Variable mutable: $a")
  println("Variable inmutable: $e")
}
```

```
Variable mutable: 20
Variable inmutable: Goku

Process finished with exit code 0
```

Imagen 28. Resultado de la ejecución del método main - Ejercicio 1.

Ejercicio 2: “Probando las Variables Numéricas”

1. Crear una función main
2. Dentro de la función main declarar tres variables numéricas con los valores 10, 33, 66.
3. Declara una variable resultado que guarde la suma de las 3 variables.
4. Imprime la variable resultado
5. Modificar la variable con el valor 10, por el valor 55.
6. Imprimir nuevamente variable resultado
7. Declarar una variable promedio, que calcule el promedio de las 3 variables
8. Imprimir promedio

Método main

```
fun main() {
  var a = 10
  val b = 33
  val c = 66
  var resultado = a + b + c
  println("$a + $b + $c = $resultado")
  a = 55
  resultado = a + b + c
  println("$a + $b + $c = $resultado")
  val promedio = resultado/3
  print("Promedio : $promedio")
}
```

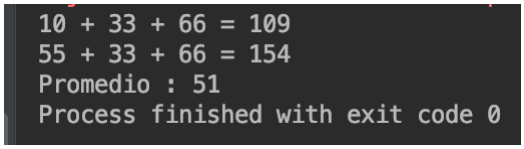


Imagen 29. Resulltado de la ejecución del método main - Ejercicio 2.

Variables Básicas en Kotlin

Competencias:

- Variables enteras.
- Variables de coma flotante.
- Variables booleanas.
- Variables String

Introducción

Conoceremos las variables Básicas que existen en Kotlin y para qué sirve cada una de estos tipos de datos.

Variables enteras

En Kotlin existen 4 tipos de variables básicas para trabajar con números, y estos son:Byte, Short, Int y Long y el uso de estas variables dependerá del número que nosotros queramos almacenar en nuestra variable.

Tipo	Tamaño	Rango de valor
Byte	8 bits	-128 a 127
Short	16 bits	-32768 a 32767
Int	32 bits	-2147483648 a 2147483647
Long	64 bits	-9223372036854775808 a 9223372036854775807

Nota: Para que los alumnos no tengan que saber estos valores de memoria, recuerde que en cada clase Byte, Short, Int y Long existe una constante llamada `MAX_VALUE` y `MIN_VALUE`.

```
fun main() {
  val byteMaximo = Byte.MAX_VALUE
  val  byteMinimo  =  Byte.MIN_VALUE
  val shortMinimo = Short.MIN_VALUE
  val shortMinimo = Short.MIN_VALUE
  val intMinimo = Int.MIN_VALUE
  val intMinimo = Int.MIN_VALUE
  val longMinimo = Long.MIN_VALUE
  val longMinimo = Long.MIN_VALUE
}
```

Variables de coma flotante

En Kotlin existen 2 tipos de variables básicas para trabajar con valores de coma flotante y estos son:

Float y Double, uso de estas variables dependerá del número que nosotros queramos almacenar en nuestra variable.

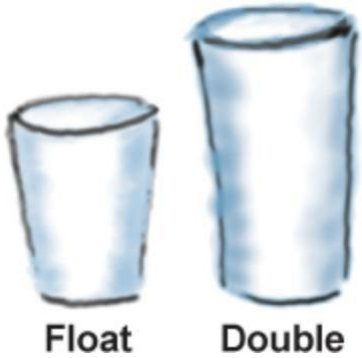


Imagen 30. Tipos de variables básicas

Para el caso de las variables Float, al momento de asignar valor a la variable, esta debe venir acompañada con una letra “f” o “F”:

```
fun main() {
  val x = 19.5f
  var y : Double = 20.5
}
```

Variables booleanas

Las variables booleanas almacenan sólo dos valores, “verdadero” o “falso”

```
fun main() {
  val x = true
  val y : Boolean
  y = false
}
```

Cadenas de caracteres

En Kotlin existen dos tipos básicos para trabajar con cadenas de caracteres.En el caso de que se necesite almacenar un carácter único utilizamos el tipo Char.En el caso de que se necesite almacenar una cadena de caracteres, utilizamos String.

```
fun main() {
  val x = 'a'
  val y = "Cadena de caracteres"
  var vocal : Char
  val nombre : String
  vocal = 'u'
  nombre = "Chizuru Kagura"
}
```

En este ejemplo podemos ver cómo al igual que en Java, los objetos String se utilizan con comilla doble “Esto es un String” y los objetos Char se utilizan con comilla simple ‘A’

Ejercicio 3: “Probando las Variables Básicas”

1. Crear una función main
2. Dentro de la función main declarar una variable de tipo Int, String, Char y Float
3. Imprime el valor por pantalla de todas variables.

Solución: Método main

```
fun main() {
  val entera = 10
  val cadena = "Goku"
  val vocal = 'U'
  val flotante = 10F
  println("Variable Int: $entera")
  println("Variable String: $cadena")
  println("Variable Char: $vocal")
  println("Variable Float: $flotante")
}
```

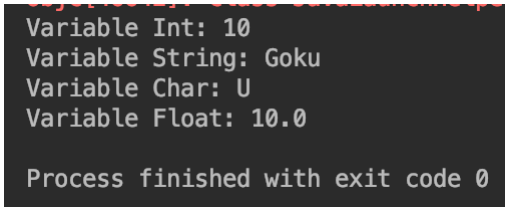


Imagen 31. Resultado de la ejecución con el método main - Ejercicio 3

Tipos de funciones

Competencias:

- Tipos de funciones
- Métodos Java en relación a las Funciones Kotlin
- Funciones en Kotlin
- Conocer el uso de Void, void y Unit en Kotlin
- Funciones con parámetros

Introducción

Hasta ahora hemos trabajado con la función “main”, en este capítulo aprenderemos a separar nuestro código en funciones Kotlin, veremos los tipos de funciones que tiene Kotlin y cómo podemos utilizar nuestras funciones en distintas partes de nuestro proyecto.

Tipos de Funciones

Para poder comprender mejor los tipos de funciones en Kotlin, comparemos las funciones de Kotlin con las funciones de Java.

En Java, a las funciones se llaman métodos, y estos se clasifican en dos grupos, métodos sin tipo de retorno y métodos con tipo de retorno.

Método sin tipo de retorno

```
public void imprimirNombreCompleto(){
   System.out.println("Sanji  Vinsmoke");
}
```

Son todos los métodos declarados como “void”, y no necesitan un return dentro de sus llaves { }.

Métodos con tipo de retorno

```
public String getNombreCompleto(){
   return "Monkey D. Luffy";
}
```

Son todos los métodos que declaran una clase en antes de especificar su nombre y devuelve un objeto de la clase declarada. Estos métodos tienen la obligación de llevar un return entre sus llaves { }.

En el ejemplo anterior clase declarada es “**String**” y el valor del objeto retornado es “**Monkey D. Luffy**”.

Funciones en Kotlin

En Kotlin, solo existen los métodos con tipo de retorno y a estos métodos se les llama funciones.

Y estos se declaran de la siguiente forma:

```
fun getNombreCompleto(): String{
  return "Monkey D. Luffy"
}
```

- “fun”: Indica que es una función Kotlin.
- “getNombreCompleto”: es el nombre de la función.
- “ ( ) ”: Indica los parámetros que tiene la función, en este caso es vacío o sin parámetro.“
- **:** String” : indica el tipo de retorno que tendrá la función, en este caso un “**String**”.
- “ return “**Monkey D. Luffy**” “ : Es el objeto retornado por la función.

Repasando métodos void de Java

Ya se mencionó que Kotlin solo trabaja con funciones que retornan valores, entonces: ¿Qué pasa con los métodos void que tiene Java y existen en Kotlin ?

Para responder a esta pregunta, entendamos bien lo que es un método void.“void” es una palabra reservada del lenguaje Java, y lo utilizamos para indicar que un método no retorna valor.“Void” con mayúscula, es el nombre de una clase Java que se encuentra en el package “java.lang”, es un wrapper que transforma un valor primitivo “void” a un objeto Java.

El uso de la clase “Void” no es muy popular, esta clase existe desde que Java soporto el uso de objetos genéricos “Generics”. Todo valor primitivo Java tiene un equivalente en una clase Java, por eso existe la clase Integer, Boolean, Double, Float, etc.

Funciones void en Kotlin

En Kotlin no existen la palabra reservada “void”, sólo existe la clase “Void”, pero si le indicamos a una función que retornará un objeto tipo “Void” nos llevaremos una sorpresa.

```
fun imprimirNombreCompleto(): Void {
    println("Sanji Vinsmoke")
}
Remove explicitly specified return type of enclosing function 'imprimirNombreCompleto'
```

Imagen 32. Función void

Me dice que remueva explícitamente el “Void” especificado como tipo de retorno.

Entonces… ¿Cómo declaro una función Void en Kotlin?

R) con la clase “Unit”

La clase Unit es el equivalente en Kotlin a la variable primitiva “void” de Java

```
fun imprimirNombreCompleto(): Unit {
  println("Sanji Vinsmoke")
}
```

Si declaramos una función que retorna un tipo “Unit”, automáticamente Kotlin detecta que esa función no retorna nada (como los void en Java).

De hecho si declaramos explícitamente que nuestra función retorna un tipo “Unit”, el mismo lenguaje te sugiere que no lo agregues. Ya que todas las funciones en Kotlin que no tengan declarado el tipo de retorno, automáticamente se declara un tipo de retorno “Unit” y lo podemos veren la siguiente imagen.

```
fun imprimirNombreCompleto(): Unit{
}
Redundant 'Unit' return type less... (⌘F1)
Inspection info: This inspection reports a redundant Unit return type which can be omitted.
```

Imagen 33. Retorno "Unit"

Así se declara una función “void” en Kotlin y ya sabemos que en realidad devuelve un “Unit”

```
fun imprimirNombreCompleto(){
    println("Sanji Vinsmoke")
}
```

Imagen 34. Declarar función void en Kotlin que devuelve un "Unit"

Funciones con parámetros

La forma de declarar los parámetros en una función Kotlin es indicando el nombre de la variable, seguido por “ : “ donde indicamos el tipo de dato de esta variable.

En caso de querer agregar más de un parámetro, este se indica por medio de una “ , ”Ej:

```
fun imprimirNombreCompleto(nombres: String, apellidos: String, edad: Int){
  println("Nombres: $nombres")
  println("Apellidos: $apellidos")
  print("Edad: $edad")
}
```

Función Main en Kotlin

Una de las funciones más importantes a nivel de programación es el método main.

El método main es tan importante, que si desarrollamos una aplicación y no creamos el método main, no podemos ejecutar nuestra aplicación.

En Kotlin a los métodos les llamamos funciones, y la función main no es la excepción.

A diferencia de Java, la función main de Kotlin es mucho más sencilla de declarar

Ejemplo:

```
fun main(){
  println("Hola Mundo, soy la función Main y no soy estática.")
}
```

Otro tema importante a destacar, es que no necesitamos que nuestra función main se encuentre dentro de una clase, es decir, solo creando un archivo Kotlin ya es suficiente para poder declarar nuestra función main.

Imagen 35. Función main dentro del archivo `.kt`

Ejercicio 4: “Probando las funciones”

1. Crear una función llamada getNombre que devuelva un String con tu nombre.
2. Crea una función llamada “imprimirNombreCompleto” que reciba los siguientes parámetros “nombres”, “apellidos”, edad.
3. Crear una función llamada main() que llame a estas dos funciones.

Solución

```
fun main(){
    println("Nombre: ${getNombre()}")
    imprimirNombreCompleto( nombre: "Roger",  apellido: "Gol D.",  edad: 53)
}
```

Imagen 36. Código método main - Ejercicio 4

```
Nombre: Sanji Vinsmoke
Nombre: Roger
Apellido: Gol D.
Edad: 53
Process finished with exit code 0
```

Imagen 37. Resultado de la ejecución del método main - Ejercicio 4

Clases en Kotlin

Competencias:

Clases en KotlinData class en Kotlin

Contexto

Uno de los pilares básicos de la programación orientada a objetos es el uso de clases y objetos.En este capítulo veremos cómo funcionan las clases en Kotlin y lo fácil que es crear una clase POJO en Kotlin con las “Data Class”.

Clases en Kotlin

Tarde o temprano necesitaremos utilizar algo más que las clases básicas de Kotlin (Int, String, Boolean, etc.) , ahí es donde entran nuestras clases.Las clases son plantillas que nos permiten crear nuestros propios objetos, definiendo nosotros las propiedades y funciones.

En Kotlin y Java existen la mismas clases por lo que no profundizaremos sobre este contenido en este capítulo.

Nuestra primera clase en Kotlin :

```
class Persona{ }
```

Lo primero que extrañamos en nuestra clase, es el uso de la palabra reservada “public”, en Kotlin todas las clases son “public” por defecto, por lo tanto declarar una clase como public es redundante en Kotlin.

Imagen 1. Palabra reservada Public.

Data Class en Kotlin

Kotlin también puede trabajar con clases tipo P.O.J.O. (Plain Old Java Object) de forma muchos mas eficiente que en otros lenguajes, cuando necesites crear una clase con atributos, get y set. Data class es la solución a este problema.

P.O.J.O en Java

```
public class Persona {
  private String nombre;
  private String rut;
  private int edad;

  public Persona(String nombre, String rut, int edad) {
     this.nombre = nombre;
     this.rut = rut;
     this.edad = edad;
  }

  public String getNombre() {
     return nombre;
  }

  public void setNombre(String nombre) {
     this.nombre = nombre;
  }

  public String getRut() {
     return rut;
  }

  public void setRut(String rut) {
     this.rut = rut;
  }

  public int getEdad() {
     return edad;
  }

  public void setEdad(int edad) {
     this.edad = edad;
  }
}
```

Esta misma clase en Kotlin se puede desarrollar con un Data Class

```
data class Persona(val nombre: String, val rut: String, val edad: Int)
```

Main.kt

```
fun main(){
   val persona = Persona("Juan", "1-9", 39)
   println("Nombre Persona = ${persona.nombre}")
   println("Rut Persona = ${persona.rut}")
   println("Edad Persona = ${persona.edad}")
}
```

Solución

Imagen 2. Solución Data Class

Open Class en Kotlin

En Kotlin por defecto las clases son finales, es decir, no se puede heredar de dicha clase. Para poder utilizar la Herencia en Kotlin lo hacemos utilizando los ":" seguido del nombre de la clase padre y la llamada al constructor de esta clase.

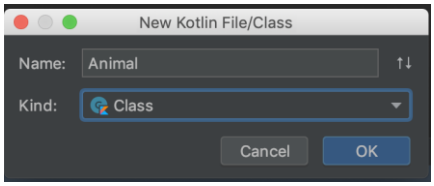


Imagen 3. Crear clase animal.

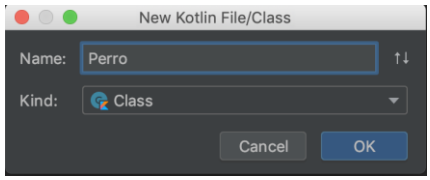


Imagen 4. Crear clase perro.

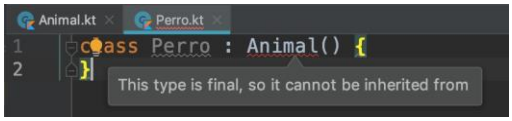


Imagen 5. Clase perro.

Acá vemos como nuestra clase Animal por defecto es una clase “final”

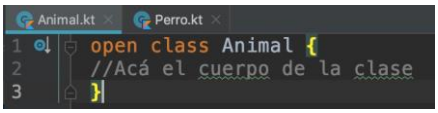


Imagen 6. Clase final.

Al declarar nuestra clase como open, esta se puede heredar en otra clase:

Imagen 7. Heredar desde otra clase.

Ya no tenemos el error de que nuestra clase es “final”.

Sealed Class en Kotlin

Las “Sealed Class” o clases selladas, se utilizan para definir una jerarquía de clases restringidas, esto con el fin de utilizar valores predeterminados.Funciona de forma similar a las enumeraciones, con la ventaja de que las clases selladas son mucho más flexibles que las enumeraciones y lo veremos en el siguiente ejemplo:

Creamos una enumeración llamada TipoDeMensaje.kt

```
enum class TipoDeMensaje(val tipo : String) {
  ALERTA("Esto es una alerta"),
  TOAST("Esto es una tostada"),
  DIALOGO("Esto es un cuadro de diálogo"),
  VENTANA("Esto es una ventana")
}

fun main(){
  val tipoDeMensaje = TipoDeMensaje.ALERTA
  println(tipoDeMensaje.tipo)
}
```

El mismo ejemplo, lo podemos hacer con clases selladasCreamos una clase sealed llamada TipoDeMensaje.kt

```
sealed class TipoDeMensaje
class MensajeAlerta(val mensaje: String) : TipoDeMensaje()
class MensajeToast(val mensaje: String) : TipoDeMensaje()
class MensajeDialog(val mensaje: String) : TipoDeMensaje()
class MensajeVentana(val mensaje: String) : TipoDeMensaje()

fun main() {
  var alerta = MensajeAlerta("Esto es una alerta")
  val toast = MensajeToast("Esto es una tostada")
  val dialog = MensajeDialog("Esto es un cuadro de diálogo")
  val ventana = MensajeVentana("Esto es una ventana")
  println(alerta.mensaje)
  println(toast.mensaje)
  println(dialog.mensaje)
  println(ventana.mensaje)
}
```

Acá queda en evidencia la ventaja de las clases selladas ya que si quisiéramos agregar un nuevo tipo de mensaje, no necesitamos modificar la clase principal, lo único que debemos hacer es crear una nueva clase que herede de nuestra clase sellada.

Ejercicio: “Probando las clases en Kotlin”

1. Crear una clase POJO en Kotlin llamada JuegoDeArcade
2. Definir las siguientes propiedades: nombre, cantidad de jugadores, tipo de juego.
3. Crear funcion main
4. Declarar una variable de tipo “JuegoDeArcade” y asignarle valor
5. Imprime el valor por pantalla de la variable creada.

JuegoDeArcade.kt

```
data class JuegoDeArcade(val nombre : String, val cantidadJugadores : Int, val tipoJuego: String)
```

Método main

```
fun main() {
  val juegoDeArcade = JuegoDeArcade("Kof98", 2, "Peleas")
  println("Nombre Juego: ${juegoDeArcade.nombre}")
  println("Nro de Jugadores: ${juegoDeArcade.cantidadJugadores}")
  println("Tipo de Juego: ${juegoDeArcade.tipoJuego}")

}
```

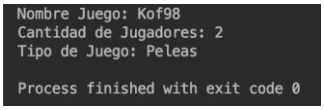


Imagen 8. Resultado de la ejecución del método main.

Constructores en Kotlin

Competencias:

- Constructores en Kotlin
- Constructor sin parámetros
- Constructor con parámetros
- Sobrecarga de constructores

Contexto

El uso de los constructores es esencial para la programación orientada a objetos, sin constructores nuestras clases no pueden ser instanciadas.En este capítulo veremos qué es un constructor, como se declaran un constructor con y sin parámetros y aprenderemos cómo trabajar con sobrecarga de constructores en una clase Kotlin.

Constructores en Kotlin

Kotlin, al igual que Java necesita crear constructores en sus clases para que estas puedan crear objetos, es tan importante el constructor en una clase, que el mismo lenguaje se encarga de crear un constructor sin parámetros en caso de que nosotros no lo creamos en nuestra clase.

Constructor sin parámetros

El constructor sin parámetros, es el constructor por defecto que tiene una clase, como ya mencionamos, si nosotros creamos una clase Kotlin, podremos llamar al constructor de esa clase por medio de la siguiente instrucción:

```
class Perro {
}

fun main(){
   val perroSalchicha = Perro()
}
```

A diferencia de Java, en Kotlin no utilizamos la palabra reservada “new” para llamar al constructor.En Kotlin para llamar al constructor se hace por medio del signo de igualdad “=” seguido del nombre del constructor que nosotros queramos llamar.En este caso se llama al constructor sin parámetros que crea por defecto Kotlin “= Perro( )”.

Constructor con parámetros

Cuando nosotros declaramos un constructor con parámetros, esto se debe hacer de la siguiente manera:

```
class Perro(val nombreParam: String, val edadParam: Int) {
  val nombre = nombreParam
  val edad = edadParam
}

fun main(){
  val perroSalchicha = Perro("Toky", 2)
  print("El nombre del perro es : ${perroSalchicha.nombre} y la edad es: ${perroSalchicha.edad}.")
}
```

En este caso para llamar al constructor con parámetros, llamamos al nombre de la clase,. seguido de los parámetros de su constructor. “= Perro(“Toky”, 2)”.

Sobrecarga de Constructores

Cuando necesitemos una clase con más de un constructor a esto se le llama sobrecarga de constructores y esto se hace de la siguiente manera:

```
class Perro {
  val nombre: String
  val edad: Int
  constructor() {
      nombre = "Sin Nombre"
      edad = 0
  }
  constructor(nombreParam: String, edadParam: Int){
     nombre = nombreParam
     edad = edadParam
  }
}

fun main() {
  val perroSalchicha = Perro()
  val perroBulldog = Perro("Firulais", 15)
  println("El nombre del perro es : ${perroSalchicha.nombre} y la edad es: ${perroSalchicha.edad}.")
  println("El nombre del perro es : ${perroBulldog.nombre} y la edad es: ${perroBulldog.edad}.")
}
```

```
El nombre del perro es : Sin Nombre y la edad es: 0.
El nombre del perro es : Firulais y la edad es: 15.

Process finished with exit code 0
```

Imagen 9. Salida del programa.

Ejercicio: “Probando los constructores en Kotlin”

1. Crear una clase llamada EquipoDeportivo
2. Definir las siguientes propiedades: nombre y deporte.
3. Crear constructor con parámetros con las dos propiedades
4. Crear constructor sin parámetros y asigna por defecto los valores “Sin nombre” para el nombre del equipo y “Fútbol” para el deporte.
5. Crear la función main.
6. Declarar dos variables de tipo “EquipoDeportivo”
7. Asignar valor a la primera variable llamando al constructor sin parámetros.
8. Asignar valor a la segunda variable llamando al constructor con parámetros.
9. Imprime el valor por pantalla de las 2 variables creadas.

EquipoDeportivo.kt

```
class EquipoDeportivo {
  val nombre: String
  val deporte: String
  constructor() {
      nombre = "Sin Nombre"
      deporte = "Fútbol"
  }
  constructor(nombreParam: String, deporteParam: String){
     nombre = nombreParam
     deporte = deporteParam
  }
}
```

Método main

```
fun main() {
  val equipo = EquipoDeportivo()
  val equipo2 = EquipoDeportivo("Los Ferros", "Fútbol Americano")
  println("Nombre Equipo: ${equipo.nombre}")
  println("Deporte: ${equipo.deporte}")
  println("Nombre Equipo: ${equipo2.nombre}")
  println("Deporte: ${equipo2.deporte}")
}
```

```
Nombre Equipo: Sin Nombre
Deporte: Fútbol
Nombre Equipo: Los Ferros
Deporte: Fútbol Americano

Process finished with exit code 0
```

Imagen 10. Resultado de la ejecución del método main.

Funciones del lenguaje

Competencias:

- Funciones en archivos Kotlin.
- Funciones de extensión en Kotlin

Introducción

En Kotlin además de crear funciones dentro de una clase, podemos crear funciones fuera de una clase e incluso podemos crear funciones de extensión para clases donde no tengamos acceso al código fuente.

Funciones en archivos Kotlin

En Kotlin al momento de crear una clase también podemos crear archivos, estos archivos Kotlin tienen la misma extensión que una clase ".kt".En estos archivos podemos crear nuestras funciones kotlin sin la necesidad de crear una clase, hacer esto en Java es imposible, ya que todas las funciones deben estar dentro de una clase.
Botón derecho en carpeta "src" -> "New" -> "Kotlin File/Class".

Imagen 11. Crear nueva clase/archivo.

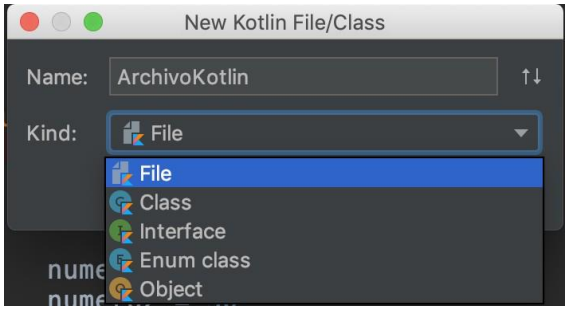


Imagen 12. Crear archivo Kotlin.

ArchivoKotlin.kt

```
fun saludo(){
  print("Hola")
}
fun despedida(){
  print("Chao")
}
```

En este ejemplo podemos ver un archivo Kotlin que contiene 2 funciones, estas funciones pueden ser llamadas desde cualquier parte de nuestro proyecto

Main.kt

```
fun main(){
  saludo() //
  despedida()
}
```

En este ejemplo podemos ver cómo se utilizan 2 funciones sin la necesidad de crear ninguna clase, estos métodos son propios del lenguaje Kotlin y podemos ver una representación de cómo funcionan por medio del IDE IntelliJ.

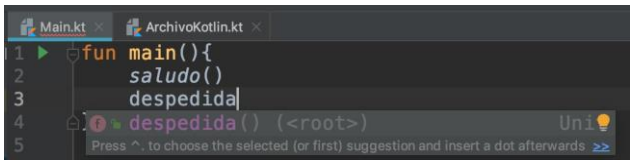


Imagen 13. Utilizar dos funciones sin una clase.

Los métodos declarados en archivos de tipo Kotlin se consideran como métodos () y gracias a esto podemos utilizarlos en cualquier clase.

Funciones de extensión Kotlin

Las funciones de extensión permiten al desarrollador agregar una función a una clase específica sin la necesidad de modificar el código fuente de la clase, estas funciones se crean de la misma manera que las funciones de los archivos Kotlin, pero se le debe indicar la clase donde queramos extender de estas funciones.

Ejemplo: Botón derecho en carpeta “src” -> “New” -> “Kotlin File/Class”.

Imagen 14. Crear clase/archivo.

Acá creamos un archivo Kotlin:

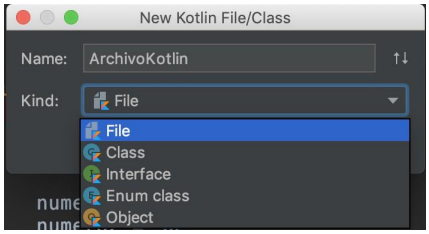


Imagen 15. Crear archivo Kotlin.

ArchivoKotlin.kt

```
fun String.saludo(){
   print("Hola")
}
fun String.despedida(){
   print("Chao")
}
```

En este ejemplo podemos ver cómo declaramos una función llamada “String.saludo” cuando al nombre de nuestra función le anteponemos el nombre de una clase (en este caso la clase String) significa que cualquier objeto de la clase String puede utilizar nuestra función “saludo” y “despedida”

Main.kt

```
fun main(){
    val variable = "Hola" //declaramos una variable String
    variable.saludo() //Utilizamos nuestra función de extención "saludo"
    variable.despedida() //Utilizamos nuestra función de extención "despedida"
}
```

Imagen 16. `main.kt`

En este ejemplo podemos ver cómo se utilizan 2 funciones en una variable de tipo String sin la necesidad de modificar la clase String, este tipo de función es propios del lenguaje Kotlin y podemos ver una representación de cómo funcionan por medio del IDE IntelliJ.

```
fun main(){
    val variable = "Hola"
    variable.saludo()
    variable.despedida()
}
```

Imagen 17. Funciones en una variable string.

Las funciones de extensión declarados en Archivos Kotlin se consideran como métodos () para variables String y gracias a esto podemos utilizarlos en cualquier objeto que sea de este tipo.

Ejercicio: “Probando las funciones del lenguaje”

1. Crear un archivo Kotlin y agregar una función de extensión de la clase String llamada “encriptarValor”
2. Agregar la lógica que permita a la función “encriptarValor” cambiar los siguientes valores:

a	**4**
e	3
i	1
o	0
u	T_T

3. Crear la función “main” y agregar una variable de tipo String
4. Agregar un valor a la variable
5. Ejecutar la funcion “encriptarValor” de nuestra variable y agregar esta llamada en un método print.

Solución:

ArchivoKotlin.kt

```
fun String.encriptarValor(): String {
  return this.replace("a","4")
      .replace("e","3")
      .replace("i", "1")
      .replace("o", "0")
      .replace("u", "T_T")
}
```

Main.kt

```
fun main(){
  val variable = "Paralelepipedou"
  println("Valor normal: $variable")
  println("Valor encriptado: ${variable.encriptarValor()}")
}
```

```
Valor normal: Paralelepipedou
Valor encriptado: P4r4l3l3p1p3d0T_T

Process finished with exit code 0
```

Imagen 18. Solución Ejercicio.

Android y Kotlin

1.2 Iniciando con Kotlin – Integrando Kotlin a un Proyecto en Android

Competencias:

- Integrar Kotlin en Android Studio.
- Aprender a crear clases Kotlin.
- Convertir Clases Java a Kotlin

Motivación

Kotlin es un lenguaje relativamente nuevo que contiene muchas diferencias respecto a Java, diferencias que vienen de los lenguajes más modernos como Python o Swift. A pesar de todo esto, Kotlin fue pensado para funcionar en conjunto con Java y esto ha sido parte fundamental para hacer que se convierta en el lenguaje oficial de Android, ya que tiene muy buena compatibilidad con Java.

Por lo tanto si queremos aprender a desarrollar en Android, será necesario entender cómo se integra con Android Studio, ya que éste es el IDE oficial de Android. Nuestro objetivo ahora será descubrir como Android Studio nos puede facilitar el desarrollo Android usando kotlin, descubrir cómo iniciar un proyecto, como crear archivos kotlin y finalmente entender qué herramientas nos puede entregar este IDE para facilitar nuestra migración de Java a Kotlin.

Integrar Kotlin en Android Studio

Para comenzar a trabajar con Kotlin en Android, debemos partir usando Android Studio, que es el IDE oficial de google para desarrollar en Android, este puede ser descargado desde la página oficial de Google para Windows, Linux o Mac, debe ser si la versión 3 o superior, ya que desde ahí en adelante se le hizo soporte oficial a Kotlin. Si tenemos un proyecto antiguo que no tiene soporte para kotlin también se le puede agregar, pero eso es algo que no profundizaremos ahora.

Para comenzar un proyecto con Kotlin, comenzamos un nuevo proyecto en Android con Android Studio, pero en el primer paso que nos aparece, donde seteamos el nombre de la app, es necesario marcar la opción Include Kotlin Support, como se destaca en la siguiente imagen.

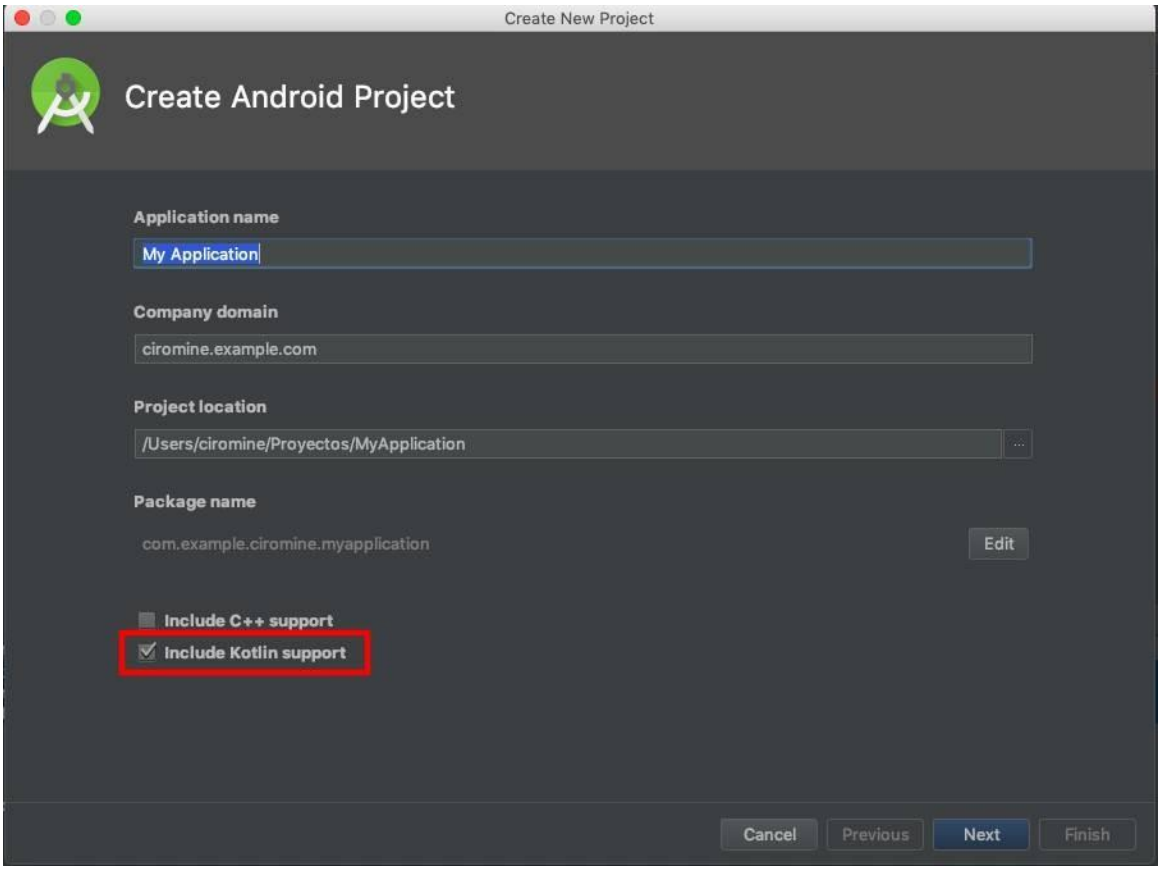


Imagen 1. Crear proyecto Android.

Ya con esta opción seleccionada, nuestro nuevo proyecto partirá con nuestra clases creadas en con la extensión .**kt**, que es el formato usado por Kotlin para sus archivos. Pero sigamos creando nuestro nuevo proyecto, al Application name le pondremos DesafioLatam y ahora haremos click en Next.

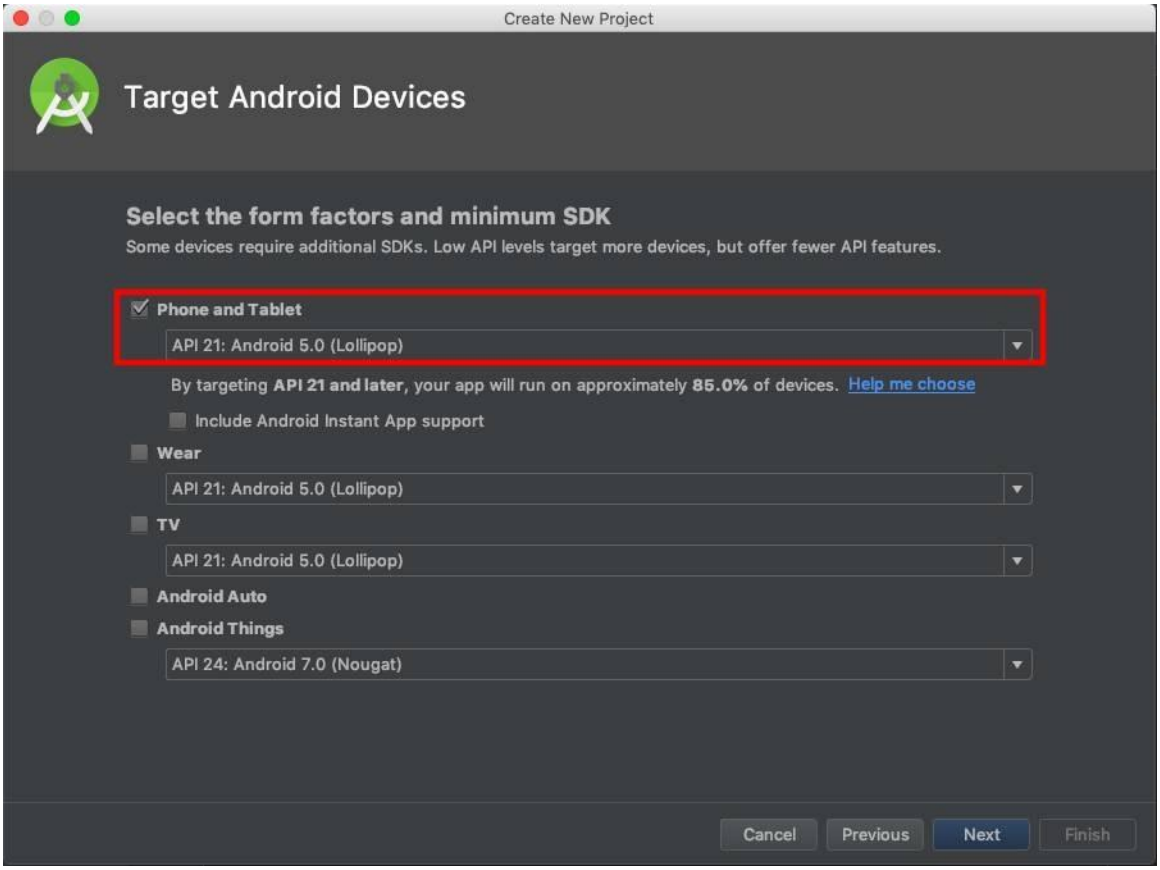


Imagen 2. Soporte del proyecto.

Acá vemos que nos indica que queremos que soporte nuestro proyecto, si va a ser para Teléfono y Tablet, o también tendrá compatibilidad con Wear (relojes por ejemplo), TV, Android Auto o android Things. En este caso solo haremos compatible con teléfono y seleccionaremos el API 21 Android 5.0 (Lollipop), ¿qué significa esto? Significa que nuestro proyecto soportará el api de desarrollo como mínimo hasta **Android 5.0**. O sea que la mínima versión que soportaremos será Android 5.0, ya que con esto como el mismo IDE lo indica soportamos el 85% de los dispositivos actuales del mercado. Ahora podemos seguir con el botón Next.

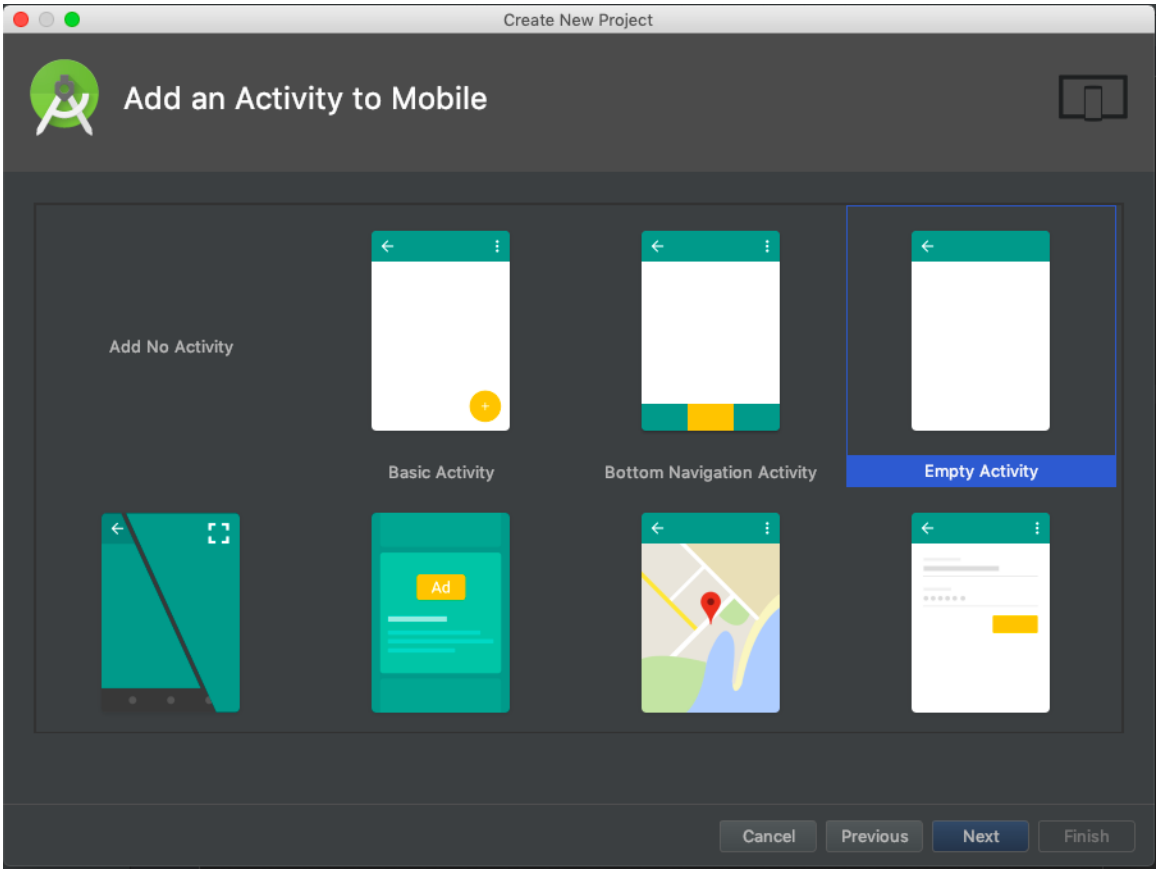


Imagen 3. Agregar actividad al proyecto.

Escogemos la opción Empty Activity, el cual traerá una Activity vacía con un textview al medio que contiene una frase de “Hello World”. Luego podemos hacer click en Next.

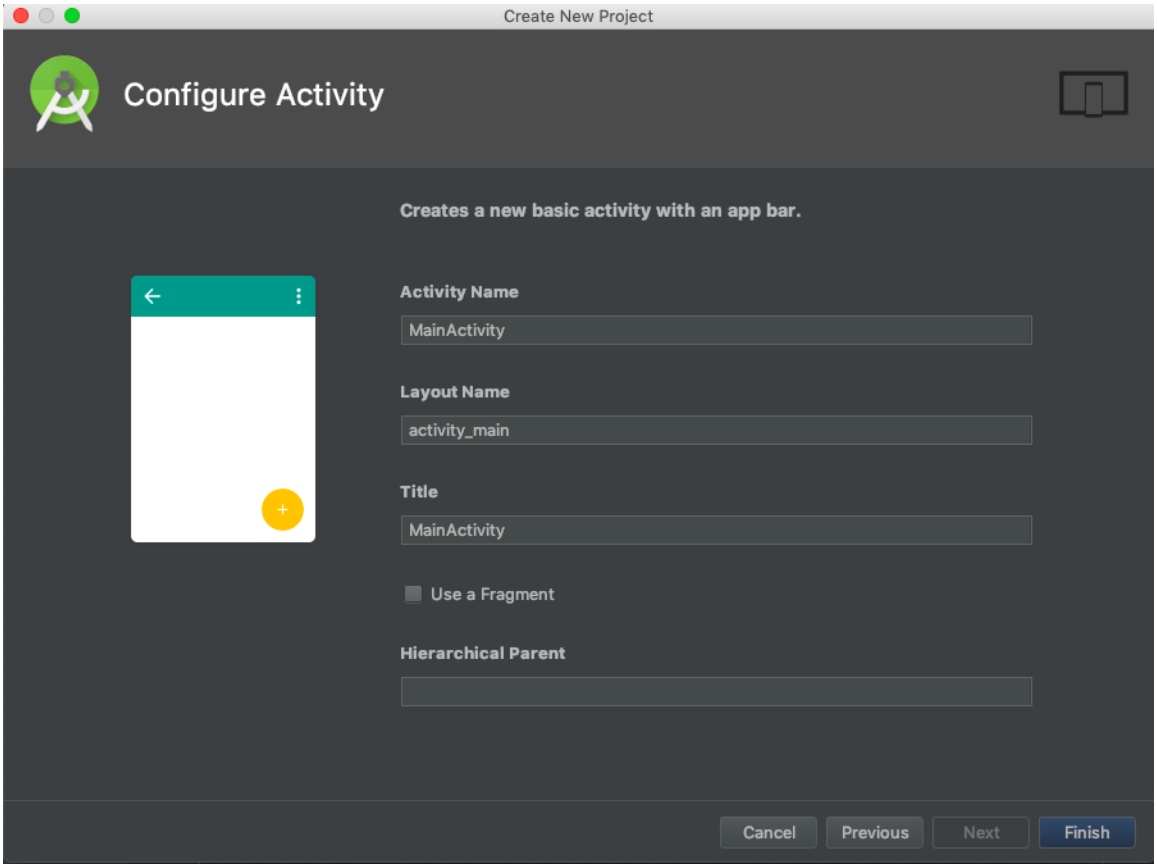


Imagen 4. Proyecto nuevo.

Acá en el último paso le podemos cambiar el nombre a esa activity, al layout y el título, en este caso nosotro, no cambiaremos nada de esto, ya podemos apretar el botón Finish y comenzar a codear. Nuestra Activity en Kotlin debería verse de la siguiente forma.

```
class MainActivity : AppCompatActivity() {

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

  }
}
```

Si nos fijamos es como cualquier clase en Java, que no tiene nada de particular, solamente tiene la sintaxis de kotlin.

Ahora veremos cómo crear una clase kotlin desde Android Studio.

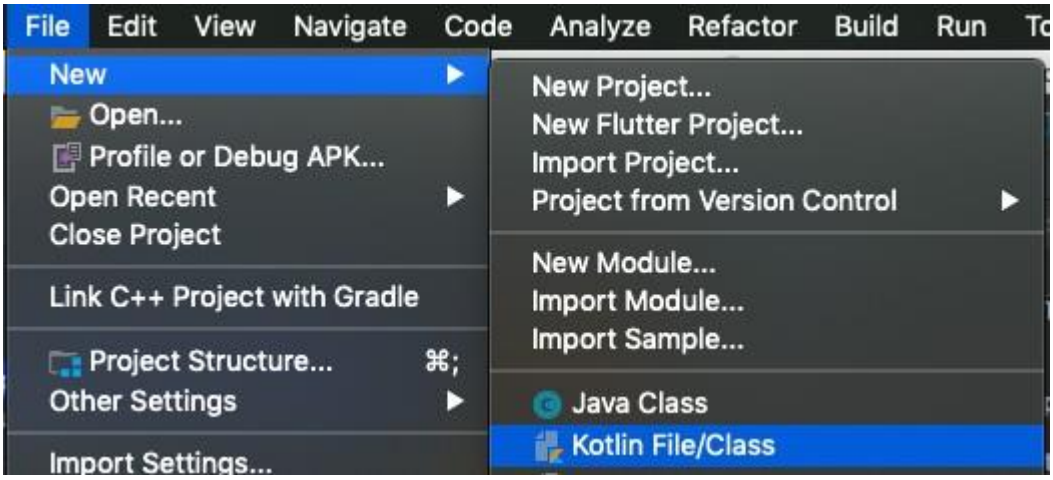


Imagen 5. Crear una clase Kotlin desde Android Studio.

Debemos hacer click en File/New/Kotlin file Class.

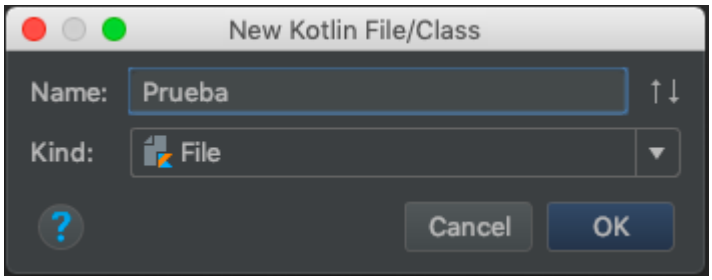


Imagen 6. Crear archivo llamado Prueba.

Luego le seteamos un nombre a nuestro archivo y listo, crearemos nuestro archivo kotlin para crear otra Activity o una clase con código para lo que queramos realmente. A diferencia de las clases de Java, los archivos de kotlin vienen vacíos, por lo que es necesario nosotros escribir todo el código, por ejemplo, el archivo recién creado quedaría así.

```
package  com.example.ciromine.desafiolatam
```

Ahora si quisiéramos que la clase Prueba tuviera por ejemplo un método Suma que retornará un int, deberíamos hacer algo así.

```
package com.example.ciromine.desafiolatam

class Prueba {

  fun suma(): Int {
      return 1+1
  }
}
```

Así podemos generar más clases de Kotlin y podremos seguir generando código para realizar todas las funciones e ideas que nosotros queramos.

Convertir Clases Java a Kotlin

Aprovechemos de usar otra funcionalidad muy útil de Android Studio que nos servirá para ir transformando clases Java a Kotlin de manera más sencilla. Para esto supongamos que tenemos la misma clase anterior Calculadora

```
public class Calculadora {

  public Calculadora() {}

  public int suma(int a, int b) {
     return  a + b;
  }
}
```

Ahora si nos vamos a la siguiente opción en el Menú.

Imagen 7. Opciones del menú - Convertir archivo Java a Kotlin

Una vez seleccionada esta opción, el mismo IDE hará la conversión automática de la clase Java a Kotlin, el resultado sería el siguiente.

```
class Calculadora {

  fun suma(a: Int, b: Int): Int {
      return a + b
  }
}
```

De esta manera, tenemos una herramienta que nos puede ayudar a convertir nuestro código Java en Kotlin, hay que destacar que a veces la conversión no resulta perfecta y quedan algunos pequeños errores de sintaxis, que uno mismo debe corregir, aunque eso solo ocurre con clases más complejas.

Kotlin y Java

Competencias:

- Usando la Interoperabilidad entre Android y Kotlin

Contexto

Kotlin siempre fue pensado como un lenguaje cien por ciento compatible con Java y esto permite que sean interoperables, de manera que podremos trabajar con ambos lenguajes a la vez, algo que nos permitirá ir migrando lentamente nuestro código. Literalmente podemos ir migrando clase a clase y evitar tener que generar una gran migración de código obligadamente al cambiarnos a Kotlin.

Otra ventaja de esto es: si nos imaginamos un proyecto real, el cual ya tenemos escrito en Java y comenzamos a migrar a kotlin, pero tenemos ciertas libraries que son antiguas pero no podemos prescindir de ellas, no tendríamos problema, ya que aunque nuestro código esté usando kotlin, podría comunicarse sin problemas con cualquier library aunque este hecha en Java.

Ahora veremos bien, como es el tema de la interoperabilidad con kotlin y como nos puede simplificar la vida en nuestro desarrollo en Android.

Interoperabilidad Java y Kotlin

¿Qué es la interoperabilidad?

El Instituto de Ingenieros Eléctricos y Electrónicos (IEEE) define interoperabilidad como la habilidad de dos o más sistemas o componentes para intercambiar información y utilizar la información intercambiada.

En este caso cuando nos referimos interoperabilidad, nos referimos a la forma en que las clases Java y Kotlin pueden comunicarse, como si fueran un solo lenguaje de programación.

Kotlin fue concebido como un lenguaje compatible con Java, esto le permite ser interoperable, o sea, que podemos tener clases definidas en kotlin y estás puedes ser llamadas o utilizadas desde clases hechas en java y viceversa.

A continuación veremos un ejemplo de interoperabilidad entre ambos lenguajes. Crearemos una app simple con un layout, para que podamos ver nuestro resultado en ejecución, por lo tanto crearemos una app simple donde llamemos a un método de una clase java.

Primero crearemos la vista, supongamos que creamos el ejemplo básico de un Activity en blanco con el siguiente layout, que contiene solo un textview, pero que se le agregó un **layout_constraintVertical_bias="0.04000002"** para que no esté centrado en la vista, si no este verticalmente más pegado hacia la parte superior de la pantalla.

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  tools:context="com.example.ciromine.desafiolatam.MainActivity">

  <TextView
    android:id="@+id/textview"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Hello World!"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintVertical_bias="0.04000002"  />

</android.support.constraint.ConstraintLayout>
```

Es un layout simple con un textview, ahora tendremos el siguiente código en la activity.

```
class MainActivity : AppCompatActivity() {

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
  }
}
```

Y finalmente tendremos una clase en Java llamada Calculadora, que por temas de simplicidad para el ejemplo solo tendrá un método suma que retornada la suma de 2 enteros.

```
public class Calculadora {

  public Calculadora() {}

  public int suma(int a, int b) {
    return  a + b;
  }
}
```

Ahora para mostrar la interoperabilidad de Kotlin y Java, llamaremos a esta clase en Java desde nuestra activity en kotlin, y modificaremos el Textview.

```
class MainActivity : AppCompatActivity() {

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    val textview = findViewById<TextView>(R.id.textview) as TextView

    val calculadora = Calculadora()

    textview.text = "Tu resultado es: " + calculadora.suma(1,1)
  }
}
```

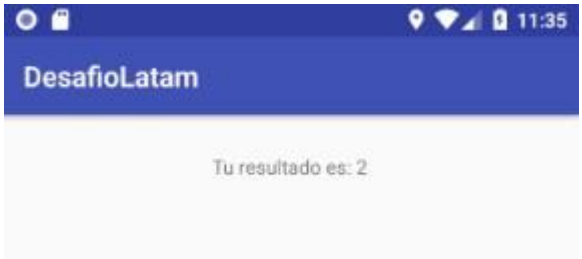


Imagen 8. Resultado al correr el proyecto.

De esta manera, hemos demostrado en un ejemplo como funciona la interoperabilidad que hay entre estos 2 lenguajes. Esta posibilidad nos da una herramienta muy poderosa, nos puede permitir trabajar con bibliotecas antiguas escritas en Java o incluso, poder ir migrando nuestro código a kotlin de a poco.

¿Por qué se usa textview.text y no textview.setText() que es el clásico método de Textview para setear un valor?

La razón de esto, es debido a que kotlin, para facilitar el código, adaptó el método setText(“”), por este nuevo textview.text = “”. De esta manera tenemos un código más simple.

Ventajas de usar Kotlin

Competencias:

- Utilizar sentencia When
- Usar Null Safety
- Manejar los operadores del null safety.

Contexto

Kotlin al ser un lenguaje moderno, tiene ciertas ventajas que vienen por defecto y que nos ayudarán a acelerar nuestro desarrollo de código y como siempre, a simplificar nuestro código. Así que en el siguiente capítulo profundizaremos cómo funcionan estas herramientas y cómo las podemos usar en nuestro camino del desarrollo de código.

Por ejemplo, en kotlin cuando creamos objetos POJO no es necesario definir todos los setters y getters de nuestras variables definidas, basta definir los parámetros en el constructor y kotlin, asume los métodos setters y getters automáticamente, de manera que no es necesario definir nada, veamos un ejemplo en código, para explicar más claro esto.

```
public class Cerveza {

  String nombre;
  String tipo;
  float grados;

  public Cerveza(String nombre, String tipo, float grados) {
     this.nombre = nombre;
     this.tipo = tipo;
     this.grados = grados;
  }

  public String getNombre() {
     return nombre;
  }

  public void setNombre(String nombre) {
     this.nombre = nombre;
  }

  public String getTipo() {
     return tipo;
  }

  public void setTipo(String tipo) {
     this.tipo = tipo;
  }
```

```
    public float getGrados() {
        return grados;
    }

    public void setGrados(float grados) {
        this.grados = grados;
    }
}
```

Este sería un POJO de ejemplo para una clase Cerveza, ahora veamos como Kotlin, reduce el código.

```
class Cerveza(var nombre: String, var tipo: String, var grados: Float)
```

Como podemos observar, Kotlin tiene muchas mejoras, que nos simplifican el desarrollo y esta son parte de las ventajas que profundizaremos en el siguiente capítulo.

When

When es una sentencia que nos ayuda a controlar el flujo dentro de nuestra aplicación, cuando tenemos un caso en el que tenemos que usar muchos if y else seguidos, puede volverse difícil de leer y generase un código engorroso. Para estos casos, podemos usar when.

La sentencia **when**, tiene la siguiente estructura:

```
when (valor){
//serie de condiciones
}
```

Ahora veamos, de donde nace el when, veamos una sentencia un poco compleja con **if** y **else**.

```
if (numero == 0) {
    println("numero invalido")
} else if (numero == 1 || numero == 2) {
    println("numero muy bajo")
} else if (numero == 3) {
    println("numero correcto")
} else if (numero == 4) {
    println("numero alto, pero aceptable")
} else {
    println("numero muy alto")
}
```

Acá vemos un caso engorroso donde tenemos muchas condiciones posibles para una variable llamada numero, entonces ahora veamos como la sentencia when nos puede ayudar a simplificar el código.

```
when(numero) {
    0 -> println("numero invalido")
    1, 2 -> println("numero muy bajo")
    3 -> println("numero correcto")
    4 -> println("numero alto, pero aceptable")
    else -> println("numero muy alto")
}
```

Como podemos ver, con la función when se ve un código mucho más legible y simple, ponemos al inicio nuestra variable y luego podemos solo la condición que evaluamos y después de -> ponemos las acciones a hacer en esa condición. Además el resultado fuera más complejo que hacer un println como en este ejemplo, se pone entre **llaves { }** y podemos realizar toda la lógica que nosotros queramos dentro.

Ahora como dijimos antes, when nos permite muchas maneras de evaluar y muchas condiciones y variaciones posibles, las cuales veremos a continuación algunos ejemplos.

```
var result = when(number) {
    0 -> "numero invalido"
    1, 2 -> "numero muy bajo"
    3 -> "numero correcto"
    4 -> "numero alto, pero aceptable"
    else -> "numero muy alto"
}
// it prints when returned "Number too low"
println("Retorno \"$result\"")
```

Podemos hacer que se retorne el valor y asignarlo en una variable, para después hacer el println con la variable, claramente podríamos retornar otro tipo de variable y hacer otra cosa que no sea un println. Ahora veamos otro ejemplo sin números

```
val check = true

val result = when(check) {
    true -> println("it's true")
    false -> println("it's false")
}
```

Como podemos ver, también podríamos evaluar fácilmente variables booleanas de manera muy simple, otro ejemplo posible.

```
var result = when(number) {
    0 -> "numero invalido"
    1, 2 -> "numero muy bajo"
    3 -> "numero correcto"
    in 4..10 -> "numero alto, pero aceptable"
    else -> "numero muy alto"
}
```

Ahora nos basamos en el primer ejemplo que vimos, pero esta vez, agregamos en vez de una evaluación simple, un rango, como vemos donde dice **in 4..10**, quiere decir que cualquier número que esté entre la brecha de 4 y 10, entrará en esa condición.

Ahora veamos qué más podemos hacer con when.

```
when (x) {
    is Int -> print("es un Entero")
    is String -> print("Es un String")
    is IntArray -> print("Es un arreglo de Enteros")
}
```

Podemos evaluar el tipo de la variable y tomar una decisión respecto de la que sea, o sea si miramos la primera condición dice **is Int** o sea, que evalúa si la variable **x**, es del tipo Int. La segunda condición ve si es del tipo **String** y la última si es un **IntArray o arreglo de int**.

Null Safety

¿Que es un un error de null?

Null o nulo (en español) es literalmente un valor vacío, cuando iniciamos una variable en Java como por ejemplo un String y hacemos **String a;** pero no le asignamos un valor, por defecto esta variable **a** se le asigna un valor nulo, que sería equivalente a un valor vacío. Por eso cuando una variable es nula y tratamos de hacer alguna operación sobre ella, podemos obtener un nullPointerException, ya que esa variable no tiene un valor definido y por ende no podríamos calcular el largo del String por ejemplo.

Kotlin es un lenguaje de programación que evita el uso de **null**, esto se debe a que así se mejora la calidad y seguridad del software, porque los errores con variables que son nulas, ocurren en tiempo de ejecución y hacen que la aplicación se caiga completamente.

Es por esto que kotlin, tiene una manera segura para trabajar con nulls, comenzaremos a explicar ahora en qué consiste.

Si nosotros intentamos de hacer lo siguiente, tendremos un problema de sintaxis, o sea que el IDE nos mostraría el siguiente código en rojo

```
//esto estaría malo ya que en Kotlin no podemos decir que una variable es igual a null
var palabra: String = "abc"
palabra = null
//Y esto también

var palabra: String = null //esto también estaría malo
```

Para poder hacer algo similar a esto tendríamos que hacer lo siguiente.

```
//esta es la manera correcta, ya que usamos el operador ? que nos permite asignar la variable igual a
null
var palabra: String? = "abc"
palabra = null

//Y esta también

var palabra: String? = null //esta es la manera correcta, ya que usamos el operador ?
```

La clave está en el operador **?** que nos está indicando que este campo puede ser null, pero es porque nosotros estamos forzando esa situación, de esta manera somos conscientes de que esa variable es null bajo nuestro propio riesgo.

Ahora que tenemos nuestra variable que podría ser null, debemos saber trabajar con ella para evitar errores al correr nuestro código, por ejemplo, si quisiéramos obtener el largo de ese String posiblemente null

```
var palabra: String? = null
palabra?.length
```

Tenemos que usar nuestro operador **?**, si no, Kotlin no nos lanzará un error de sintaxis, ya que así uno prevé que puede ser null y solo en caso que no lo sea, se podrá llamar a **length**. Veamos un caso un poco más complejo.

```
class Perro {

  var raza: Raza? = null
 .
 .
 .
}

class Raza {

  var nombre: String? = null
 .
 .
 .
}
```

En este caso tenemos clases que están unidas, un Perro tiene raza y a la vez la Raza tiene un nombre y todos esas variables fueron definidas como posibles null. Entonces si quisiéramos obtener el nombre de la raza de un perro, deberíamos hacer lo siguiente.

```
var perro = Perro()
perro?.raza?.nombre
```

Como podemos ver, en este caso donde hay una serie de relaciones entre clases con otra clases y parámetros, tendríamos que usar el operador **?** en cada nivel a medida que accedemos a la información. Ahora supongamos que en vez de usar **?** usamos otro operador

```
perro.raza!!.nombre
```

El operador **!!** lo que hace es **asegurar** de que la variable nunca va a ser **null**, o sea le decimos al compilador, que nosotros estamos cien por ciento seguros que esta variable no será null, en caso de que lo sea, obtendremos un **nullpointerException** y esto provocará que la app fallé mientras se ejecuta, por lo que hay que usarlo de manera muy responsable.

Este operador aunque no lo parezca es muy útil e importante, ya que dado que Kotlin es un lenguaje que no acepta variables nulas por defecto, hay veces que si definimos una variable con eloperador **?** indicando que podría ser null y después necesitamos agregar esa variable como parámetro de algún método, el IDE nos pedirá usar el operador **!!**, ya que necesita que aseguremosque no será null.

Veamos el siguiente código de ejemplo.

```
var lista: ArrayList<String>? = null
.
.
.
adapter.setData(lista)
```

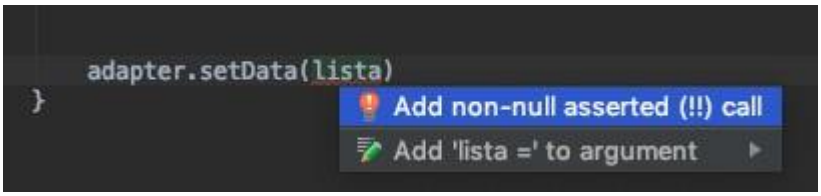


Imagen 9. Error de sintaxis.

Como vemos, el mismo IDE nos recomienda agregar el operador **!!**, ya que eso le indica que esta lista no será null, ya que nosotros nos estamos asegurando de ello, esta es una de las formas en que kotlin nos hace responsables sobre el código.

```
adapter.setData(lista!!)
```

Imagen 10. Agregar operador `!!`.

Este es un detalle importante que tenemos que tener en cuenta cuando desarrollemos en Android usando Kotlin.

¿Qué pasaría si necesitamos poner algún valor null dentro de una lista?

La verdad en Kotlin se ha pensando en todo, veamos un ejemplo.

```
val listaDeRazas: List<String?> = listOf("bulldog", "labrador", null, "pitbull")
```

Usando el mismo operador **?**, podemos definir que nuestra lista contendrá elementos del tipo String, pero soporta null, maravillosamente simple.

Con esto ya vimos funciona el null safety en kotlin y cómo debemos darle uso a los operadores para hacer un correcto desarrollo de código y evitar errores.

Android y Kotlin

Utilidades de Kotlin

Competencias:

- Concatenar Strings
- Utilizar onClickListener
- Utilizar Funciones lambda
- Utilizar View Binding

Introducción

Kotlin contiene muchas utilidades, que nos ayudarán a hacer un código más simple y legible, ahora nuestro objetivo será aprender herramientas que tiene kotlin y nos ayudarán a mejorar nuestro desarrollo de software en Android.

Lo que veremos en este capítulo, nos ayudarán mucho a simplificar nuestro desarrollo en Android, aprenderemos cómo mejorar la concatenación de strings con una alternativa que nos da kotlin, además de aprender una manera para trabajar con los listener de manera muy simplificada a lo que estamos acostumbrados en Java y finalmente aprenderemos a trabajar de manera más simple con nuestro elementos visuales definidos en nuestro layout.

Concatenar Strings

Aprovechemos de usar una de las ventajas de Kotlin, para concatenar Strings, usaremos la misma clase MainActivity que usamos anteriormente, pero le haremos un pequeño cambio

```
class MainActivity : AppCompatActivity() {

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    val textview = findViewById<TextView>(R.id.textview) as TextView

    val calculadora = Calculadora()

    val a = 1
    val b = 2

    textview.text = "Tu resultado "+a+" + "+b+" es: "+calculadora.suma(a, b)
  }
}
```

Ahora haremos el siguiente cambio en la concatenación de String que vamos a setear al Textview para simplificar lo que se hacía normalmente en Java.

```
class MainActivity : AppCompatActivity() {

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    val textview = findViewById<TextView>(R.id.textview) as TextView

    val calculadora = Calculadora()

    val a = 1
    val b = 2

    textview.text = "Tu resultado $a + $b es: ${calculadora.suma(a, b)}"
  }
}
```

Si nos fijamos, en vez de agregar un símbolo **+**, para concatenar el String con una variable, , en Kotlin podemos hacer lo siguiente, que es dejar todo entre comillas y lo que sea variables de código, lo encerramos entre las llaves **${}** si es un método (como cuando llamamos a calculadora.suma()), si no, le anteponemos un signo **$** solamente (como el caso de b). ay

Esto nos da una manera más simple para manejar la concatenación de Strings con variables, muy usada en otros lenguajes de programación como Ruby, a diferencia del antiguo uso del símbolo **+**, que varias veces se volvía bastante engorroso.

Listeners

Cuando en Android queremos darle acción al tocar un elemento usamos el Listener Onclick, el cual es un manera bastante verbosa (con mucho código) de darle acción a un botón, pero en Kotlin tenemos una alternativa que nos permite usar OnClickListener pero de manera mucho más legible.

Veamos un ejemplo a continuación, lo primero que haremos es agregar un botón al layout que teníamos antes.

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:app="http://schemas.android.com/apk/res-auto"
xmlns:tools="http://schemas.android.com/tools"
android:layout_width="match_parent"
android:layout_height="match_parent"
tools:context="com.example.ciromine.desafiolatam.MainActivity">

  <TextView
    android:id="@+id/textview"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Hello World!"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintVertical_bias="0.04000002"  />

  <Button
    android:id="@+id/button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginEnd="8dp"
    android:layout_marginStart="8dp"
    android:layout_marginTop="8dp"
    android:text="Cambiar Texto"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintHorizontal_bias="0.521"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/textview"  />

</android.support.constraint.ConstraintLayout>
```

Es un simple botón que no hace nada, ahora agregaremos el botón en la activity y le daremos una acción con el OnclickListener, que será que cambiará el texto de Textview cuando se apriete el botón.

```
class MainActivity : AppCompatActivity() {

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    val textview = findViewById<TextView>(R.id.textview) as TextView

    val button = findViewById<Button>(R.id.button) as Button

    button.setOnClickListener(object : View.OnClickListener {
      override fun onClick(v: View) {
        textview.text = "Texto cambiado"
      }
    })
  }
}
```

Acá podemos ver cómo asociamos el botón desde el layout con el findViewById, pero además usamos onClickListener para cambiar el Textview una vez que se haga click en el botón, esta es como se dijo antes la manera tradicional de hacer esto, ahora haremos uso de lambda para dejar el código más simple, algo maravilloso que nos permite Kotlin.

```
override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  setContentView(R.layout.activity_main)

  val textview = findViewById<TextView>(R.id.textview) as TextView

  val button = findViewById<Button>(R.id.button) as Button

  button.setOnClickListener { textview.text = "Texto cambiado" }
}
```

Listo, de esta manera kotlin nos permite usar una función lambda lo que simplifica el código, no hace falta llamar a View ni sobrescribir el método onClick, simplemente usamos la expresión setOnClickListener y con llaves y podemos definir todas las funciones que queramos que sucedan.

Imagen 1. Resultado.

Esto puede ser aplicado, a cualquier Listener de los que tenemos en Android como por ejemplo **OnLongClickListener**, **OnFocusChangeListener**, **OnTouchListener**, etc.

Función Lambda

Las funciones lambdas las podríamos definir como simplificada de pasar funcionalidad como parámetro sin la necesidad de crear funciones anónimas, esto nos permite hacer funciones complejas, pero con menos código, lo que hace más simple su lectura.

¿Qué es una función anónima?

Una función anónima es una función que se define, pero no se le asigna un nombre, o sea algo como así.

```
func () {
 println('Esta función no tiene un nombre')
}
```

Este tipo de funciones, se usan mucho en lenguajes tipo javascript. Y son una manera excelente de reducir la cantidad de código necesaria para ejecutar ciertas tareas que son repetitivas y con mucho código en el desarrollo Android, como los listeners y los callbacks.

Una función lambda tiene la siguiente estructura.

```
{ x: Int, y: Int -> x + y }
```

Las reglas de las éstas funciones son las siguiente:

- La expresión lambda debe estar delimitada por llaves.
- Si la expresión contiene cualquier parámetro, debes declararlo antes del símbolo **->**.
- Si estás trabajando con múltiples parámetros, debes separarlos con comas.
- El cuerpo de la función va luego del signo **->**.

Las funciones lambda te permiten definir funciones anónimas y luego pasar estas funciones inmediatamente como una expresión. Entonces que no necesitas escribir la especificación de la función en una clase abstracta o interfaz. Como fue en el ejemplo del onClickListener donde primero teníamos el siguiente código:

```
button.setOnClickListener(object : View.OnClickListener {
        override fun onClick(v: View) {
            textview.text = "Texto cambiado"
        }
    })
```

Y gracias a la función lambda, logramos simplificarlo a esto:

```
button.setOnClickListener { view -> textview.text = "Texto cambiado" }
```

Sin embargo, dado que la variable view no se está usando kotlin, nos permite simplificar esta función aún más.

```
button.setOnClickListener { textview.text = "Texto cambiado" }
```

como podemos ver, en este caso este método setOnClickListener {} de kotlin es una función lambda que nos solamente ejecutar toda la funcionalidad, en vez de tener que definir todos los métodos y vistas y recién ahí poder generar las acciones que nosotros queríamos.

En Kotlin podemos crear nuestras propias funciones lambdas, pero es algo en lo que no profundizaremos ahora.

View Binding

Kotlin tiene una de las herramientas más útiles que podrían existir para el desarrollo en Android, las Kotlin Android Extensions, que son una library que contiene herramientas muy útiles para Android, a continuación veremos cómo integrarlas al proyecto, en caso de que no esté integrado, aunque en la versión actual de Android Studio ya debería venir por defecto agregadas en los proyectos Android.

Agregando Kotlin Android Extensions

Lo primero que debemos hacer es agregar al archivo app/build.gradle lo siguiente

```
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'
apply plugin: 'kotlin-android-extensions'
```

Usando Kotlin Android Extensions

Esto generalmente se pone al principio del archivo. Ya con este pequeño cambio tenemos habilitadas las extensiones en Android, ahora veamos un ejemplo de cómo usarlas.

Supongamos tenemos el siguiente layout, que tiene 2 textview, uno arriba del otro y un botón debajo de los textview.

```xml
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:app="http://schemas.android.com/apk/res-auto"
xmlns:tools="http://schemas.android.com/tools"
android:layout_width="match_parent"
android:layout_height="match_parent"
tools:context="com.example.ciromine.desafiolatam.MainActivity">

  <TextView
    android:id="@+id/textview"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Hello World!"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintVertical_bias="0.04000002"  />

  <Button
    android:id="@+id/button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginEnd="8dp"
    android:layout_marginStart="8dp"
    android:layout_marginTop="8dp"
    android:text="Boton"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintHorizontal_bias="0.521"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/textview2"  />

  <TextView
    android:id="@+id/textview2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginEnd="8dp"
    android:layout_marginStart="8dp"
    android:layout_marginTop="8dp"
    android:text="TextView"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/textview"  />

</android.support.constraint.ConstraintLayout>
```

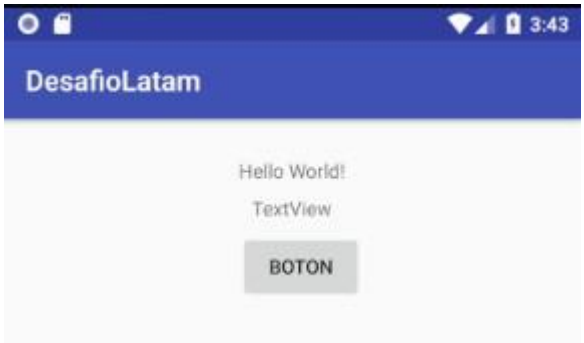


Imagen 2. Resultado

Ahora nuestro código en java sería como en el ejemplo anterior.

```
override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  setContentView(R.layout.activity_main)

  val textview = findViewById<TextView>(R.id.textview) as TextView
  val textview2 = findViewById<TextView>(R.id.textview2) as TextView

  val button = findViewById<Button>(R.id.button) as Button

  button.setOnClickListener {
     textview.text = "Texto cambiado"
     textview2.text = "Texto 2 cambiado también"
  }
}
```

Ahora con si usamos las Kotlin Android Extensions, **no tendremos que usar más el findViewById**, o sea sería así.

```
override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  setContentView(R.layout.activity_main)

  button.setOnClickListener {
     textview.text = "Texto cambiado"
     textview2.text = "Texto 2 cambiado también"
  }
}
```

Como podemos ver, se simplifica mucho el código ¿no? Solo hay que llamar a los componentes con el mismo nombre que les pusimos de ID en el xml. Si nos fijamos en los IDs del xml

```
<TextView
  android:id="@+id/textview"

<Button
  android:id="@+id/button"

<TextView
  android:id="@+id/textview2"
```

Son los mismos nombres de las variables en el código java. Algo importante a destacar es que para que esto funcione, hay que **agregar un import**, que sería.

```
import kotlinx.android.synthetic.main.activity_main.*
```

Si nos fijamos, al final del import está llamando a activity_main que es el nombre del archivo xml, donde tenemos el layout.

De todas formas una manera simple de tener el import, es sencillamente usando el IDE a nuestro favor, si escribimos el nombre de la variable y sobre ella usamos el shortcut del teclado para importar (que en **Mac** es **Opción + Ente**r y en **Windows y Linux** es, **Alt + Intro**), el mismo IDE nos indica el import correcto, como se ve en las siguientes imágenes.

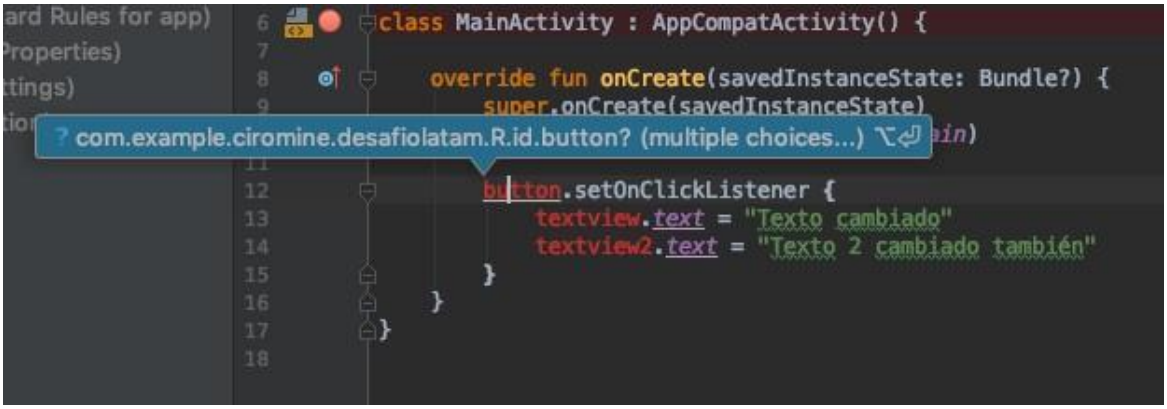


Imagen 3. Nombre de la variable.

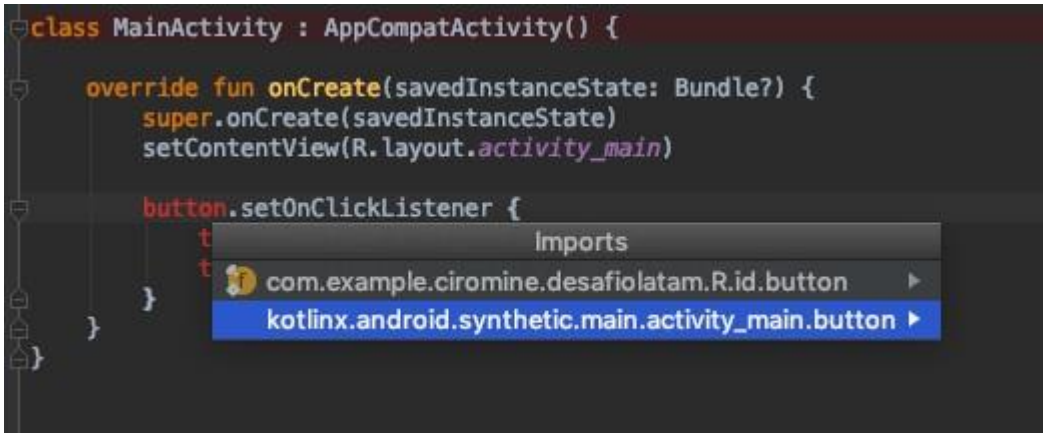


Imagen 4. Importar.

Con esta herramienta no tendremos problema para agregar el import correcto.

Imagen 5. Resultado.

Profundizando collections en Kotlin

Competencias:

- Usar Filters, Listas y Maps.
- Utilizar funciones de ordenamiento.
- Utilizar el patrón delegate.

Contexto

Kotlin nos provee varias funcionalidades para trabajar con colecciones, o sea con listas, maps, y otras estructuras de datos. Ya que generalmente siempre que uno trabaja con estos componentes, nos encontraremos con problemas y limitaciones como ordenar o filtrar el contenido de éstas, para poder hacer un uso correcto de la información con la que trabajaremos mientras programamos.

En el siguiente capítulo veremos varias alternativas que tenemos para trabajar con estas estructuras de datos, para así no tener problemas al trabajar con listas o maps. Esto nos ayudará muchísimo para poder mostrar datos de forma más precisa y con menos código que en Java.

Filters

Los Filters son una herramienta muy poderosa que nos da Kotlin para trabajar con ciertas estructuras de datos, como Lists o Maps. A continuación veremos ejemplos para trabajar con Filters y ver cómo nos pueden ayudar para simplificar nuestro trabajo y código.

Cabe destacar que no todos los filtros funcionan para List o Maps, iremos usándolos según podamos.

List

Las listas son una estructura de datos en la cual tenemos un grupo de elemento consecutivos, que pueden ser de varios tipos de variables. Además de esto las listas tienen una serie de funciones como insertar elementos, eliminar, calcular su largo, entre muchos otros que nos ayudan a trabajar con los elementos de manera más óptima.

Para generar listas en kotlin, podemos usar el siguiente método que nos generará una lista con los valores que ingresemos.

```
listOf("one", "two", "three", "four")
```

Con este método podemos generar una lista, que en este caso contendría 4 strings, a la vez podemos llenarla de otras variables como int, boolean, float, etc. Como en el siguiente ejemplo.

```
listOf(null, 1, "two", 3.0, "four")
```

Las listas son usadas en Android por ejemplo cuando queremos usar un Spinner o un RecyclerView.

Maps

Los maps, son una estructura de datos, que nos permite tener elemento almacenados con “llave” y “valor”, en otros lenguajes se le conoce como diccionario y la gracia de estos elementos, que al tener una llave para cada elemento, es más rápido y simple acceder al valor que necesitamos.

Para generar maps en kotlin tenemos la siguiente función por ejemplo.

```
mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key11" to 11)
```

Nos generará un map con las respectivas llaves y valores. Podemos agregar al igual que el listOf distintos tipos de variables.

```
mapOf("key1" to null, "key2" to 2.0, "key3" to "algo", "key11" to 11)
```

Con esto ya podemos comenzar a ver los distintos tipos de filtrados.

Filtrando por predicado

El filtrado por predicado es es sencillamente una función que nos devuelve otra lista, ya filtrada según la condición que nosotros requerimos, veamos el siguiente ejemplo.

En este caso el predicado es una condición que se evaluará, como cuando hacemos un if y según ese evaluación tendremos un resultado con la lista filtrada.

```
val numeros = listOf("one", "two", "three", "four")
val largoMayorA3 = numbers.filter { it.length > 3 }
```

Primero expliquemos un poco el código anterior, **listOf** nos generará la lista que ingreamos. En este caso **largoMayorA3** contiene una lista con lo elementos que tengan un largo mayor a 3 caracteres. En este caso serían **three** y **four**, o sea tiene 2 elementos. Veamos esto en el código corriendo para comprobar que lo que estamos diciendo es real, volvamos a nuestra app que el botón y el textview que usamos anteriormente

```
override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  setContentView(R.layout.activity_main)

  val inicial = findViewById<TextView>(R.id.textview) as TextView
  val resultado = findViewById<TextView>(R.id.textview2) as TextView

  val button = findViewById<Button>(R.id.button) as Button

  val numeros = listOf("one", "two", "three", "four")
  val largoMayorA3 = numeros.filter { it.length > 3 }

  button.setOnClickListener {
     inicial.text = numeros.toString()
     resultado.text = largoMayorA3.toString()
  }
}
```

Una vez que levantemos la app y apretemos el botón vamos a ver la lista original en el primer Textview y en el segundo Textview la lista después del filter.

Imagen 6. Resultado.

Ahora veamos el mismo ejemplo pero para Maps.

```
val numeros = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key11" to 11)
val filtro = numeros.filter { (key, value) -> key.endsWith("1") && value > 10}
```

Como vemos acá tenemos un map con key y value, donde agregamos un filtro con dos condiciones, una para la llave que tiene que terminar con el subString **“1”** y el valor de esa llave sea mayor a 10, en este caso el único que cumple esa condición es el último elemento con **key = “key11”** y **valor = 11**.

Hagamos lo mismo que la vez pasada para comprobar nuestro código.

```
override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  setContentView(R.layout.activity_main)

  val inicial = findViewById<TextView>(R.id.textview) as TextView
  val resultado = findViewById<TextView>(R.id.textview2) as TextView

  val button = findViewById<Button>(R.id.button) as Button

  val numeros = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key11" to 11)
  val filtro = numeros.filter { (key, value) -> key.endsWith("1") && value > 10}

  button.setOnClickListener {
     inicial.text = numeros.toString()
     resultado.text = filtro.toString()
  }
}
```

Imagen 7. Resultado.

Ahora veamos otro ejemplo, filterIndexed.

```
val numeros = listOf("one", "two", "three", "four")

val filteredIdx = numeros.filterIndexed { index, s -> (index != 0) && (s.length < 5)  }
```

En este caso filterIndexed recibe 2 parámetros el índice que queramos filtrar y la condición que queramos usar para filtrar, en este caso se va a filtrar por que el índice sea distinto de cero, o sea **queda eliminado automáticamente el primer elemento** y luego una condición que el largo del contenido no sea mayor a 5, por lo que queda **eliminado el elemento que contiene la palabra "three"**.

```
override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  setContentView(R.layout.activity_main)

  val inicial = findViewById<TextView>(R.id.textview) as TextView
  val resultado = findViewById<TextView>(R.id.textview2) as TextView

  val button = findViewById<Button>(R.id.button) as Button

  val numeros = listOf("one", "two", "three", "four")
  val filtro = numeros.filterIndexed { index, s -> (index != 0) && (s.length < 5) }

  button.setOnClickListener {
     inicial.text = numeros.toString()
     resultado.text = filtro.toString()
  }
}
```

Imagen 8. Resultado.

Este filtro no está disponible para Maps. Ahora veamos otro ejemplo con filterNot.

```
val numbers = listOf("one", "two", "three", "four")

val filteredNot = numbers.filterNot { it.length <= 3 }
```

Este filtro nos permite crear condiciones negadas, en este caso, lo aplicaremos cuando el contenido de la posición de la lista **NO tenga un largo menor o igual a 3**. O sea solo quedarían los elementos **“three” y “four”** en la lista después de aplicar el filtro.

```
override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  setContentView(R.layout.activity_main)

  val inicial = findViewById<TextView>(R.id.textview) as TextView
  val resultado = findViewById<TextView>(R.id.textview2) as TextView

  val button = findViewById<Button>(R.id.button) as Button

  val numeros = listOf("one", "two", "three", "four")
  val filtro = numeros.filterNot { it.length <= 3 }

  button.setOnClickListener {
     inicial.text = numeros.toString()
     resultado.text = filtro.toString()
  }
}
```

Imagen 9. Resultado.

Este filtro si está disponible para Maps y se usa de la misma manera que filter normal, pero asumimos que es la condición negada. Ahora veamos otro ejemplo con filterIsInstance.

```
val numeros = listOf(null, 1, "two", 3.0, "four")
var result = ""
numeros.filterIsInstance<String>().forEach {
  result += it.toUpperCase()
}
```

Como podemos ver acá podemos hacer uso de **filterIsInstance**, esto hará que se filtre según el tipo de Objeto que necesitemos hacer el filtro, de esta manera el ejemplo anterior, quedaría después de aplica el filtro solos los campos **"two"** y **"four"**. Veamos el ejemplo.

```
override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  setContentView(R.layout.activity_main)

  val inicial = findViewById<TextView>(R.id.textview) as TextView
  val resultado = findViewById<TextView>(R.id.textview2) as TextView

  val button = findViewById<Button>(R.id.button) as Button

  val numeros = listOf(null, 1, "two", 3.0, "four")
  val filtro = numeros.filterIsInstance<String>()

  button.setOnClickListener {
     inicial.text = numeros.toString()
     resultado.text = filtro.toString()
  }
}
```

Imagen 10. Resultado.

Este filtro no aplica para Maps. Ahora veamos otro filtro filterNotNull.

```
val numeros = listOf(null, "one", "two", null)
val listaFiltrada = numeros.filterNotNull()
```

FilterNotNull, nos saca los elementos que tengamos null dentro de nuestra lista, algo que puede ser de mucha utilidad cuando uno desarrolla. En este caso los únicos elementos que quedarían después de aplicar el filtro serian **“one” y “two”**.

```
override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  setContentView(R.layout.activity_main)

  val inicial = findViewById<TextView>(R.id.textview) as TextView
  val resultado = findViewById<TextView>(R.id.textview2) as TextView

  val button = findViewById<Button>(R.id.button) as Button

  val numeros = listOf(null, "one", "two", null)
  val filtro = numeros.filterNotNull()

  button.setOnClickListener {
     inicial.text = numeros.toString()
     resultado.text = filtro.toString()
  }
}
```

Imagen 11. Resultado.

Partitioning

Otra función que tenemos disponible es Partitioning, que nos permite, dividir en una lista en 2 según un criterio, veamos un ejemplo.

```
val numbers = listOf("one", "two", "three", "four")
val (entranEnElRango, resto) = numbers.partition { it.length > 3 }
```

Aca podemos ver, como usamos el método partition, generamos una condición que si el elemento tiene un largo mayor a 3, pasa a guardarse en la lista **entranEnElRango** y lo que no cumplea la condición, pasa a **resto**. O sea en cada lista quedan con 2 elementos.

```
override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  setContentView(R.layout.activity_main)

  val inicial = findViewById<TextView>(R.id.textview) as TextView
  val resultado = findViewById<TextView>(R.id.textview2) as TextView

  val button = findViewById<Button>(R.id.button) as Button

  val numeros = listOf("one", "two", "three", "four")
  val (entranEnElRango, resto) = numeros.partition { it.length > 3 }

  button.setOnClickListener {
     inicial.text = numeros.toString()
     resultado.text = "${entranEnElRango.toString()} - ${resto.toString()}"
  }
}
```

Imagen 12. Resultado.

Como podemos ver partition es una manera muy simple de dividir listas en base a condiciones.

Prueba de predicados

Tenemos unos métodos que nos sirven para hacer pruebas sobre las condiciones que usemos.

- **any**: retorna true si al menos uno de los elementos cumple la condición.
- **none**: retorna true si ningún elemento cumple la condición.
- **all**: retorna true si todos lo elementos cumplen la condición.

Veamos una prueba de cada uno.

Any

```
override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  setContentView(R.layout.activity_main)

  val textview = findViewById<TextView>(R.id.textview) as TextView

  val button = findViewById<Button>(R.id.button) as Button

  val numbers = listOf("one", "two", "three", "four")

  button.setOnClickListener { textview.text = numbers.any { it.endsWith("e") }.toString() }
}
```

En este caso tenemos la lista que contiene los string con los nombres de los números del 1 al 4 en inglés y corremos la prueba **any**, cuando hacemos click en el botón, en esta prueba, estamos evaluando que al menos uno de los elementos termina con la letra **"e"**, lo que obviamente es **true**, ya que tenemos el caso del **one** y el **three**.

Imagen 13. Resultado.

Como podemos ver se cumple la condición. Ahora revisemos un ejemplo de none.

None

```
override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  setContentView(R.layout.activity_main)

  val textview = findViewById<TextView>(R.id.textview) as TextView

  val button = findViewById<Button>(R.id.button) as Button

  val numbers = listOf("one", "two", "three", "four")

  button.setOnClickListener { textview.text = numbers.none { it.endsWith("a") }.toString() }
}
```

En este caso usaremos el mismo ejemplo anterior pero cambiando la prueba, usaremos **none** para ver que **ningún elemento de la lista termina con "a"** y si revisamos los elementos, esto debería cumplirse sin problemas y retornar **true**.

Imagen 14. Resultado.

Como vemos nuevamente se cumplió la condición señalada, ahora veamos un caso de all.

All

```
override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  setContentView(R.layout.activity_main)

  val textview = findViewById<TextView>(R.id.textview) as TextView

  val button = findViewById<Button>(R.id.button) as Button

  val numbers = listOf("one", "two", "three", "four")

  button.setOnClickListener { textview.text = numbers.all { it.endsWith("e") }.toString() }
}
```

Ahora haremos la siguiente evaluación, si es que todos los elementos de la lista terminan con **“e”**, lo cual obviamente no se cumpley debería retornar **false**.

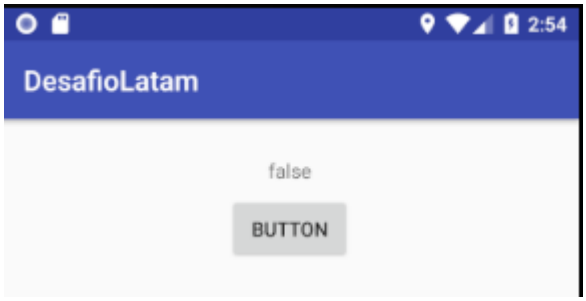


Imagen 15. Resultado.

Acá hicimos una revisión bien completa sobre los filters y como estos nos pueden ayudar en el desarrollo. Cabe destacar que estas pruebas podemos aplicarlas también con los maps, revisemos el caso pero solo con un ejemplo.

```
override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  setContentView(R.layout.activity_main)

  val textview = findViewById<TextView>(R.id.textview) as TextView

  val button = findViewById<Button>(R.id.button) as Button

  val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key11" to 11)

  button.setOnClickListener { textview.text = numbersMap.any { (key, value) -> key.endsWith("1") &&
value > 10 }.toString() }
}
```

en este caso usaremos el mismo map del primer ejemplo de filter y le pondremos la misma condición que teníamos esa vez, pero usando **any**. En este caso, dado que alguna de las key termina con **"1"** y algún value es **mayor a 10**, debería ser **true**.

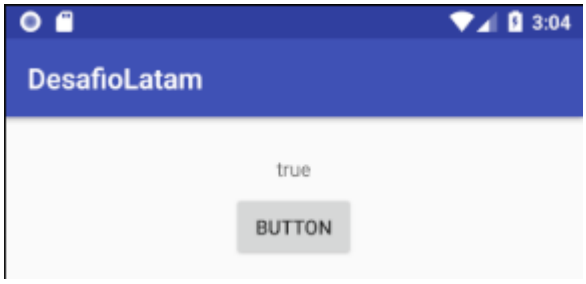


Imagen 16. Resultado.

Ya con esto hicimos una gran revisión de los filters y como estos nos pueden ayudar a nuestro trabajo.

Sorting

Ahora revisaremos cómo funciona el ordenamiento en Kotlin y las distintas formas que podemos utilizar para simplificar nuestro trabajo.

```
override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  setContentView(R.layout.activity_main)

  val inicial = findViewById<TextView>(R.id.textview) as TextView
  val resultado = findViewById<TextView>(R.id.textview2) as TextView

  val button = findViewById<Button>(R.id.button) as Button

  val lista = mutableListOf(7, 8, 1, 5, 3, 2, 4)

  button.setOnClickListener {
     inicial.text = lista.toString()
     lista.sort()
     resultado.text = lista.toString()
  }
}
```

Ahora vemos cómo generamos una lista mutable de números desordenados.

¿Qué quiere decir que sea una lista mutable?

El hecho de que sea mutable, quiere decir que le podemos agregar y quitar elementos.
En este caso, aplicamos el método sort() que nos ordenará automáticamente la lista de menor a mayor.

Imagen 17. Resultado.

Y funciona perfectamente y nos ayudará a hacer ordenamiento rápido, ahora veamos otro caso.

Veamos el mismo ejemplo pero con un arreglo normal, para que veamos un caso más típico sin mutableList.

```
override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  setContentView(R.layout.activity_main)

  val inicial = findViewById<TextView>(R.id.textview) as TextView
  val resultado = findViewById<TextView>(R.id.textview2) as TextView

  val button = findViewById<Button>(R.id.button) as Button

  val lista = arrayOf(5, 7, 21, 1, 2)

  button.setOnClickListener {
    inicial.text = lista.contentToString()
    lista.sort()
    resultado.text = lista.contentToString()
  }
}
```

como podemos ver se usa igual el método sort sin problemas y el resultado es el mismo.

Imagen 18. Resultado.

SortBy

Imaginemos tenemos la siguiente lista.

```
val lista = mutableListOf(1 to "z", 2 to "y", 7 to "x", 6 to "t", 5 to "m", 6 to "a")
```

Esto daría una lista de este estilo:

```
[(1, z), (2, y), (7, x), (6, t), (5, m), (6, a)]
```

Entonces en casos como este podemos usar SortBy lo que nos permitirá ordenar según qué elemento dentro de cada posición, o sea podemos ordenar todos los elementos según el primero o el segundo de cada paréntesis. Veamos un ejemplo ordenando toda la lista según las letras.

```
override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  setContentView(R.layout.activity_main)

  val inicial = findViewById<TextView>(R.id.textview) as TextView
  val resultado = findViewById<TextView>(R.id.textview2) as TextView

  val button = findViewById<Button>(R.id.button) as Button

  val lista = mutableListOf(1 to "z", 2 to "y", 7 to "x", 6 to "t", 5 to "m", 6 to "a")

  button.setOnClickListener {
     inicial.text = lista.toString()
     lista.sortBy { it.second }
     resultado.text = lista.toString()
  }
}
```

Como vemos usamos **sortBy{ it.second }**, o sea que estamos ordenando por el segundo parámetro dentro del elemento (o paréntesis), veamos el resultado.

Imagen 19. Resultado.

Como podemos ver en el resultado queda primero el **(6, a)** y al ultimo **(1, z)**, esto debido a que se ordena según las letras, ahora veamos el mismo ejemplo pero en base a los números.

```
override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  setContentView(R.layout.activity_main)

  val inicial = findViewById<TextView>(R.id.textview) as TextView
  val resultado = findViewById<TextView>(R.id.textview2) as TextView

  val button = findViewById<Button>(R.id.button) as Button

  val lista = mutableListOf(1 to "z", 2 to "y", 7 to "x", 6 to "t", 5 to "m", 6 to "a")

  button.setOnClickListener {
     inicial.text = lista.toString()
     lista.sortBy { it.first }
     resultado.text = lista.toString()
  }
}
```

Como podemos ver el código es el mismo, pero usamos **sortBy { it.first }**.

Imagen 20. Resultado.

Se aprecia como ahora la lista parte con **(1, z)** y termina con **(7, x)**, ya que ahora el orden se basa en los números.

Reverse

Si necesitamos sencillamente revertir el orden actual, tenemos un método llamado reverse, el cual revisaremos a continuación.

```
override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  setContentView(R.layout.activity_main)

  val inicial = findViewById<TextView>(R.id.textview) as TextView
  val resultado = findViewById<TextView>(R.id.textview2) as TextView

  val button = findViewById<Button>(R.id.button) as Button

  val lista = mutableListOf(1 to "z", 2 to "y", 7 to "x", 6 to "t", 5 to "m", 6 to "a")

  button.setOnClickListener {
     inicial.text = lista.toString()
     lista.reverse()
     resultado.text = lista.toString()
  }
}
```

Es muy sencillo al igual que **sort()**, llamamos al método **reverse()** y este nos entregará el ls lista ordenada de manera inversa.

Imagen 21. Resultado.

Patrón Delegate

El patrón delegate es una patrón de diseño que se usa en la programación orientada a objetos, que se usa generalmente cuando los lenguajes no soportan la herencia múltiple.

¿Qué es la herencia múltiple?

Herencia múltiple hace referencia a la característica de los lenguajes de programación orientada a objetos en la que una clase puede heredar comportamientos y características de más de una superclase. Esto contrasta con la herencia simple, donde una clase sólo puede heredar de una superclase.

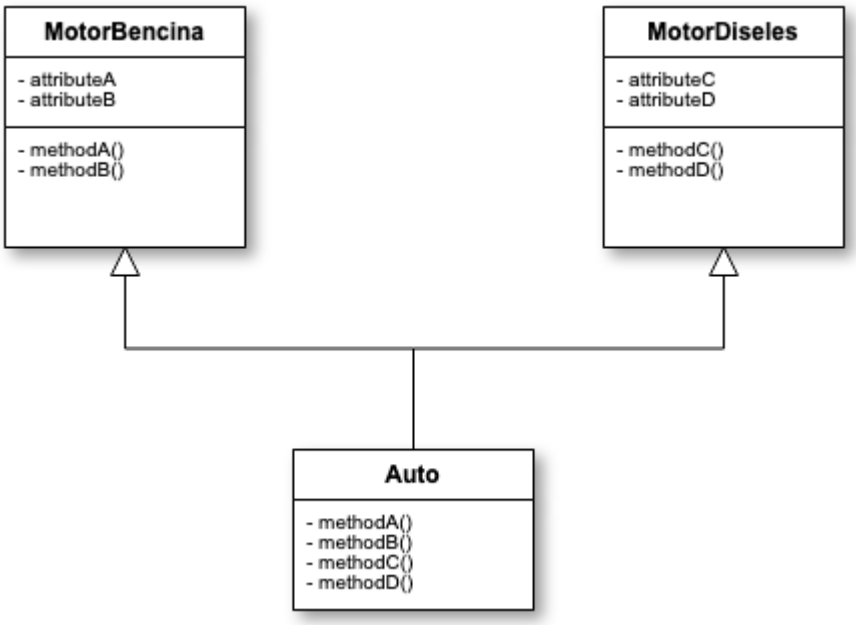


Imagen 22. Herencia múltiple.

O sea para poner un ejemplo, supongamos tenemos una clase Auto y quisiéramos que esta heredara de las clases MotorBencina y MotorDiesel. En este caso estaríamos tratando se usar herencia múltiple y no podríamos en lenguajes como Java. Una solución para esto es el uso de éste patrón.

Kotlin tiene funciones que nos permiten trabajan usando este patrón de manera más simple. A continuación veremos un ejemplo explicando este patrón usando Kotlin.

Supongamos que tenemos la siguiente interfaz **Mamifero**.

```
interface Mamifero {

   fun nombre(): String
}
```

Esta tendrá una función nombre que retorna un String. Ahora implementemos esta interfaz en 2 clases que representarán animales.

```
class Gato : Mamifero {

   override fun nombre() = "Gato"
}
```

```
class Lince : Mamifero {

   override fun nombre() = "Lince"
}
```

Como podemos ver tenemos 2 clases que implementan la interfaz **Mamifero** y cada una sobrescribe el método nombre y le pone su propio valor. Ahora Crearemos otra clase llamada felinos, que hará uso del patrón delegate.

```
class Felinos(private val delegate: Mamifero) : Mamifero by delegate {

   override fun nombre() = "Soy mamifero y felino, mi nombre es: ${delegate.nombre()}"
}
```

Creamos la clase **Felinos**, que en este ejemplo, hemos indicado que encapsulará un objeto delegado de tipo **Mamifero** y también puede usar la funcionalidad de la implementación del **Mamifero**, que puede ser **Gato** o **Lince** en este caso.

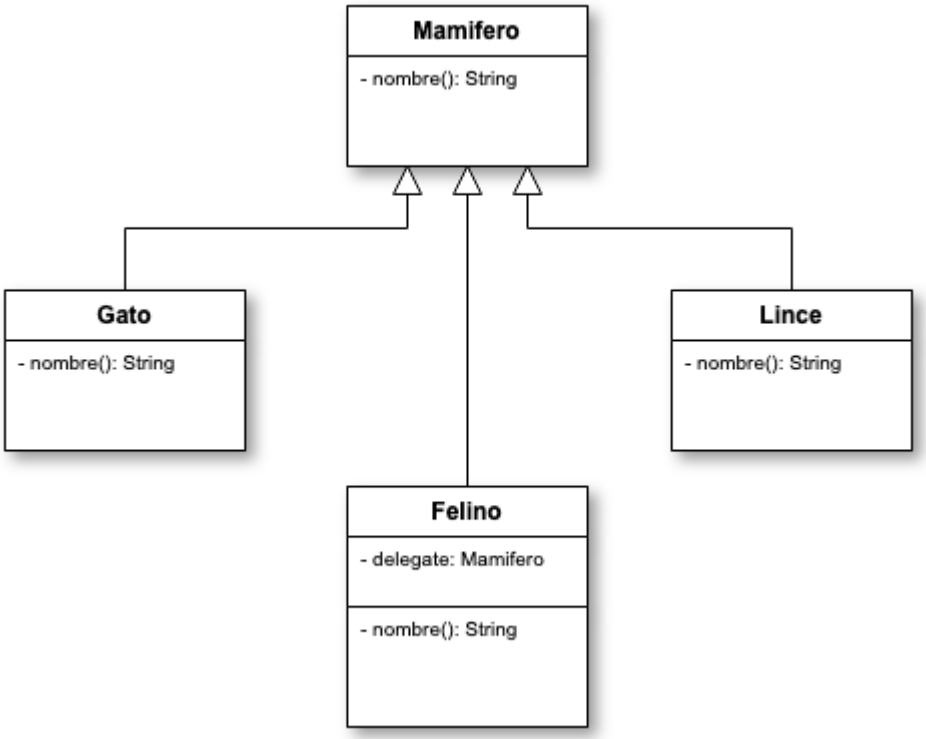


Imagen 23. Ejemplo.

Veamos como usar esto y como se vería si lo ejecutamos.

```
override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  setContentView(R.layout.activity_main)

  val mamifero1 = Felinos(Gato())
  textview.text = mamifero1.nombre()

  val mamifero2 = Felinos(Lince())
  textview2.text = mamifero2.nombre()
}
```

Como podemos ver, creamos una variable llamada **mamifero1** que usa la clase **Felinos**, pero pasándole el valor de **Gato**. Y lo mismo, pero para la clase **Lince** en otra variable llamada **mamifero2**, veamos la ejecución.

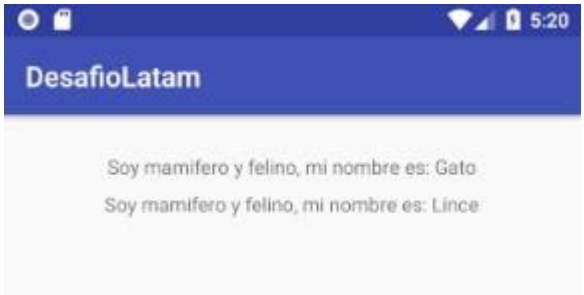


Imagen 24. Resultado.

Cómo hemos podido ver acá, un ejemplo simple de como Kotlin, nos permite usar el patrón delegate e implementar la clase **Felinos**, usando 2 clases como **Gato** o **Lince**. Podríamos crear por ejemplo otra clase **Paquidermos** y esta que llame a otros tipos de animales como **Elefante** y **Rinoceronte**, y todas a la vez todas vendrían de **Mamifero** y así podríamos ir agrandando el árbol de clases de nuestro ejemplo y ayudarnos con el patrón delegate, para implementarlo de manera simple, como se pudo ver.

Unidad 2

2.1 Threads

Síncrono y Asíncrono

Competencias:

- Conocer los conceptos de tareas y procesos síncronos
- Conocer los conceptos de tareas y procesos asíncronos
- Comprender modalidad background de un proceso
- Comprender modalidad main thread de un proceso

Introducción

En este capítulo estudiaremos los conceptos de desarrollo de software asociados a la creación de tareas, procesos y acciones síncronas y asíncronas. También revisaremos los conceptos de un proceso ejecutado en background y de un proceso ejecutado en el main thread.

Los conceptos síncrono y asíncrono son muy simples de entender a nivel general como procesos de comunicación, y están absolutamente presentes en todos los sistemas informáticos de la actualidad, por lo cual, es importante estar claro en su significado y aplicación.

Mover numerosas o largas tareas desde el hilo principal, para que no interfieran con el renderizado suave y la rápida respuesta a la entrada del usuario, es la razón más importante para adoptar subprocesos en una aplicación.

Tareas, procesos síncronos

En primer lugar, aclaremos que un proceso puede ser un sinónimo de una tarea, la diferencia estaría en que un proceso ocasionalmente requiere de más pasos para su cumplimiento que el de una tarea, entonces podemos entender ambos conceptos como similares, solo los separaría su dimensión en el trabajo que ejecutan, por lo tanto, de aquí en adelante, usaremos ambos términos para referirnos a la ejecución de un trabajo.

Un proceso síncrono en la programación de sistemas, se refiere a la conexión o dependencia sobre otra tarea o proceso, es decir, cuando se ejecuta un proceso síncrono se espera a que termine dicha ejecución antes de continuar con otro proceso.

```
SYNCHRONOUS
 |-------A--------|
                |-------B-------|
```

La representación gráfica anterior, muestra un proceso A que se ejecuta y sólo cuando este finaliza, comienza la ejecución de un proceso B.

Si declaramos constantes numéricas que indique el tiempo de una tarea, por ejemplo:

```
const val TIME_TO_COOK_IN_MINUTES: Int = 3;
const val TIME_TO_EAT_IN_MINUTES: Int = 2;
```

Un método que muestre los segundos que se toman dichas tareas expresadas en minutos:

```
fun getSecondsSpendByTask(TIME_OF_TASK: Int, tarea: String){
  println(tarea)
  for(minutes in 0..TIME_OF_TASK) {
     var segundosTranscurridos = minutes * 60;
     println("$segundosTranscurridos segundos")
     threadRealTime()
  }
}
```

Otro método que haga la simulación del tiempo real con un thread, y que utilizamos en nuestra función anterior “getSecondsSpendByTask” de la siguiente forma:

```
fun threadRealTime(){
  try{
     Thread.sleep(60000)//60 segundos equivalentes a 1 minuto
  }catch (ie: InterruptedException){
     ie.printStackTrace()
  }
}
```

Nota: el método sleep() de la clase Thread recibe milisegundos como parámetros.

Y por último, un método que represente el ejemplo real, de una persona que necesita cocinar para comer, destacando el proceso como síncrono, ya que en esta situación necesitamos que se cocine, y una vez lista la comida, se coma, contando con nuestras limitaciones, en este caso el que una sola persona hace ambas labores.

```
fun totalSynchronousTasksTimes(){
  getSecondsSpendByTask(TIME_TO_COOK_IN_MINUTES, "COCINANDO")
  getSecondsSpendByTask(TIME_TO_EAT_IN_MINUTES, "COMIENDO")
}
```

Al invocar este último método iniciaremos un proceso síncrono que ejecuta dos tareas, la primera cocinar y la segunda comer, demostrando que solo si terminamos de cocinar se podrá comenzar a comer. Para probar los métodos anteriores podemos crear el método main, dentro de una clase Utils que los contenga de la siguiente forma:

```
fun main(){
  val utils = Utils()
  try {
    utils.totalSynchronousTasksTimes()
  }catch (e: Exception){e.printStackTrace()}
}
```

Nuestra salida por consola en el terminal sería la siguiente:

```
COCINANDO
0 segundos
60 segundos
120 segundos
COMIENDO
0 segundos
60 segundos
120 segundos

Process finished with exit code 0
```

Nota: el terminal está dentro de android studio al igual que el logcat, de hecho en la parte inferior izquierda de tu android studio podrás escoger entre la vista del logcat y terminal.

Los procesos síncronos no deben ser considerados siempre como un problema, ya que las computadoras son super rápidas en su ejecución: sin embargo, si tu proceso es robusto, por ejemplo, recorre listas o mapas de datos grandes, hacer tu proceso síncrono no es la opción ya que tu sistema quedará en suspenso (bloqueado) hasta completar la tarea.

Tareas, procesos asíncronos

Una tarea o proceso asíncrono, en el modelo de programación de sistemas es aquel que se inicia, permitiendo continuar con otros procesos, para posteriormente recibir una respuesta; en otras palabras, cuando se ejecuta un proceso asíncrono, podemos iniciar otro proceso sin esperar a que el primero sea completado.

```
ASYNCHRONOUS
 |-------A-------|
     |-------B-------|
```

Imaginemos que estamos en nuestra cocina, tenemos un plato en la mesa con un pan relleno de palta y deseamos iniciar un proceso asíncrono de comer, que nos permita hacer otras tareas, como por ejemplo, poner a cocer unas papas para el almuerzo, y echar a lavar nuestros pantalones en la lavadora. Estos tres procesos se pueden ejecutar sin esperar a que alguno de ellos termine, es decir, tomamos el pan, le aplicamos un mordisco, tomamos 4 papas y las colocamos en una olla de agua, aplicamos otro mordisco al pan, vamos por la ropa y la colocamos dentro de la lavadora, aplicamos otro mordisco al pan, ponemos la olla con las papas sobre el fuego, aplicamos otro mordisco al pan, echamos jabón en la lavadora… y así sucesivamente.

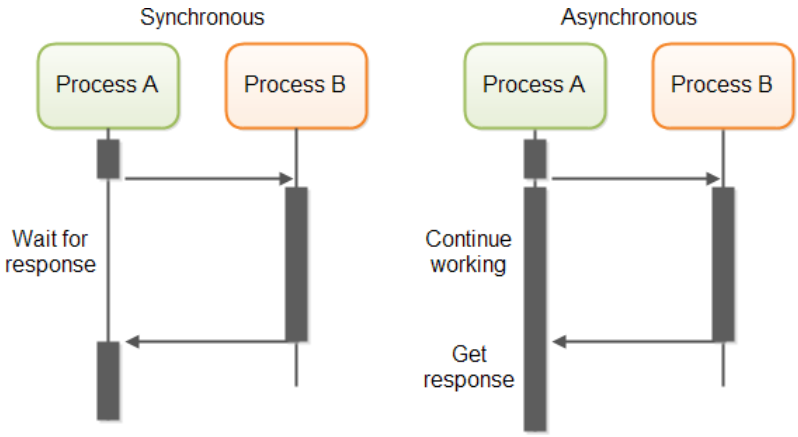


Imagen 1. Comportamiento síncrono y asíncrono.

Para lograr constituir procesos asíncronos en android, disponemos de algunas técnicas para su correcto funcionamiento, las principales serían las siguientes:

- Threading
- Callbacks
- Promises
- Reactive Extensions (observable)
- Coroutines

Hagamos el ensayo de comer, cocinar y lavar creando procesos asíncronos independientes con un breve ejemplo, creamos una clase “UtilsAsync” donde copiaremos los siguientes métodos de nuestro ejemplo de tareas síncronas anterior:

```
fun getSecondsSpendByTask(TIME_OF_TASK: Int, tarea: String) {
  for (minutes in 0..TIME_OF_TASK) {
      println(tarea)
      var segundosTranscurridos = minutes * 60;
      println("$segundosTranscurridos segundos")
      threadRealTime()
  }
}

fun threadRealTime() {
  try {
      Thread.sleep(6000)
  } catch (ie: InterruptedException) {
      ie.printStackTrace()
  }
}
```

Destacamos el println, dado que para este ejemplo es mejor moverlo dentro del ciclo for del método getSecondsSpendByTask().

Creamos las siguientes variables constantes:

```
const val TIME_TO_COOK_POTATOES_IN_MINUTES: Int = 3;
const val TIME_TO_EAT_BREAD_IN_MINUTES: Int = 2;
const val TIME_TO_WASH_CLOTHES_IN_MINUTES: Int = 4;
```

Y finalmente creamos un método que nos permita una ejecución independiente de las tres tareas a través de threads de la siguiente forma:

```
fun doAsyncTasks() {

  Thread(Runnable  {
      getSecondsSpendByTask(TIME_TO_EAT_BREAD_IN_MINUTES, "COMIENDO")
  }).start()

  Thread(Runnable {
      getSecondsSpendByTask(TIME_TO_COOK_POTATOES_IN_MINUTES, "COCINANDO PAPAS")
  }).start()

  Thread(Runnable {
      getSecondsSpendByTask(TIME_TO_WASH_CLOTHES_IN_MINUTES, "LAVANDO ROPA")
  }).start()
}
```

Para probar el ensayo, copiamos nuestro método main del ejemplo anterior, indicando en esta ocasión nuestra nueva clase de la siguiente forma:

```
fun main(){
  val utilsAsync = UtilsAsync()
  try {
     utilsAsync.doAsyncTasks()
  }catch (e: Exception){e.printStackTrace()}
}
```

El resultado por consola sería el siguiente:

```
COCINANDO PAPAS
0 segundos
LAVANDO ROPA
0 segundos
COMIENDO
0 segundos
LAVANDO ROPA
COMIENDO
60 segundos
COCINANDO PAPAS
60 segundos
60 segundos
COCINANDO PAPAS
COMIENDO
120 segundos
120 segundos
LAVANDO ROPA
120 segundos
COCINANDO PAPAS
180 segundos
LAVANDO ROPA
180 segundos
LAVANDO ROPA
```

Procesos en background

Cada aplicación de android tiene un hilo principal (Thread UI) que se encarga de manejar la interfaz de usuario, coordinar las interacciones del usuario y recibir eventos del ciclo de vida de la aplicación. Si hay demasiado trabajo en este hilo principal, la aplicación tiende a bloquearse o ponerse lenta, lo que conlleva a una mala experiencia de usuario, que terminará en el abandono de tu aplicación por medio de las críticas.

Todas las operaciones de larga duración deben realizarse en un subproceso de fondo separado, lo que llamamos “background process”. En general, cualquier cosa que tome más de unos pocos milisegundos debe delegarse en un subproceso de fondo. Es posible que deba realizar algunas de estas tareas mientras el usuario interactúa activamente con la aplicación.

Las aplicaciones también pueden requerir que se ejecuten algunas tareas, incluso cuando el usuario no está utilizando la aplicación de forma activa, como sincronizar periódicamente con un servidor de fondo o buscar periódicamente contenido nuevo dentro de una aplicación. Las aplicaciones también pueden requerir que los servicios se ejecuten inmediatamente hasta su finalización, incluso después de que el usuario haya terminado de interactuar con la aplicación.

Desafíos de los procesos background

Las tareas en segundo plano consumen los recursos limitados de un dispositivo, como la RAM y la batería. Esto puede resultar en una mala experiencia para el usuario si no se maneja correctamente.

Para maximizar la batería y aplicar un buen comportamiento de la aplicación, Android restringe el trabajo en segundo plano cuando la aplicación (o una notificación de servicio en primer plano) no es visible para el usuario.

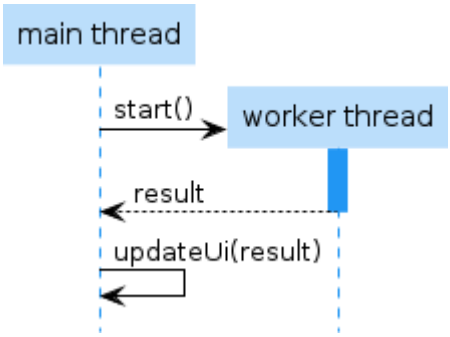


Imagen 2. Interacción hilo principal y secundario.

Procesos en main thread (hilo UI)

Cuando el usuario inicia su aplicación, Android crea un nuevo proceso de Linux junto con un hilo de ejecución. Este hilo principal, también conocido como el hilo UI, es responsable de todo lo que sucede en la pantalla. Comprender cómo funciona, nos ayuda a diseñar nuestras aplicaciones de tal manera que el rendimiento sea el mejor posible.

Mientras visualizamos una animación o una actualización de la pantalla en nuestra aplicación, el sistema intenta ejecutar un bloque de trabajo (responsable de dibujar la pantalla), con el fin de renderizar sin problemas a 60 fps (cuadros por segundo). Para que el sistema alcance este objetivo, la jerarquía de UI/Vista debe actualizarse en el hilo principal.

Cuando la cola de mensajes del hilo principal contiene tareas que son demasiado numerosas o demasiado largas para que el hilo principal complete la actualización lo suficientemente rápido, la aplicación se ve forzada a mover este trabajo a un hilo de trabajo.

Si el hilo principal no puede terminar de ejecutar bloques de trabajo en el tiempo que realiza la renderización, podemos observar bloqueos, retrasos o falta de respuesta de la interfaz de usuario a las acciones que allí ejecutamos. Si el subproceso principal se bloquea durante aproximadamente cinco segundos, el sistema mostrará que la aplicación no responde (ANR), que permite al usuario cerrar la aplicación directamente.

Mover numerosas o largas tareas desde el hilo principal, para que no interfieran con el renderizado suave y la rápida respuesta a la entrada del usuario, es la razón más importante para adoptar subprocesos en una aplicación.

Concurrencia y Paralelismo

Competencias:

- Conocer los conceptos de la concurrencia en la programación de sistemas.
- Conocer los conceptos de paralelismo en la concurrencia.

Introducción

En este capítulo profundizaremos conceptos sobre la concurrencia y el paralelismo de los procesos en el desarrollo de software para tener una mayor visión al momento de programar múltiples procesos en nuestras aplicaciones, la necesidad de utilizar de la forma más óptima los recursos de los que dispongamos en el hardware de los dispositivos, además de comprender cómo trabaja la concurrencia bajo el concepto de paralelismo.

La comprensión o mejor entendimiento de los conceptos en este capítulo, supondrán un gran avance para desarrollar los conceptos de esta unidad, además de obtener un mayor aprendizaje y preparación del alumno como futuro programador en su carrera profesional.

Concurrencia

La concurrencia es básicamente la capacidad de ejecutar múltiples tareas o procesos al mismo tiempo, pero no necesariamente simultáneos. En una aplicación concurrente tres procesos pueden iniciarse y completarse en períodos de tiempo distintos.

En principio, un dispositivo que dispone de un único microprocesador solamente puede ejecutar un proceso al mismo tiempo. No es posible ejecutar otro proceso hasta que ha finalizado el anterior.

Desde otro punto de vista, al ejecutarse tres tareas concurrentes, la capacidad de que finalicen rápidamente radica en el procesador (CPU) del dispositivo, es decir, sabemos que los computadores han ido evolucionando tecnológicamente y con ello su capacidad durante el tiempo. Expongamos la diferencia con los siguientes ejemplos:

- Un dispositivo con un procesador (CPU) single core, realizará estos procesos concurrentes a través de la técnica denominada “context-switching” para modelar el uso de los recursos de manera simultánea traspasando esfuerzos de un proceso a otro y disminuyendo el consumo de otros procesos propios del sistema operativo, buscando velocidad y capacidad para completar el trabajo.
- Un dispositivo con un procesador (CPU) multi core, puede realizar estos mismos tres procesos de manera paralela, ocupando distintos cores para cada proceso, resultando muchas veces en ejecuciones simultáneas paralelas que llegan o se acercan mucho a un mismo tiempo de finalización.

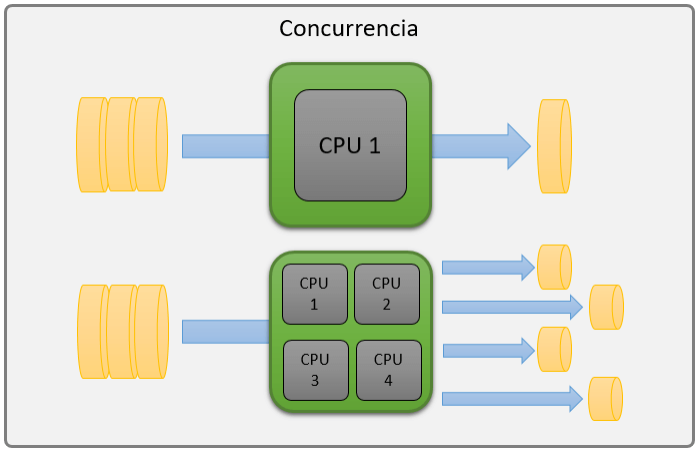


Imagen 3. Concurrencia single-core y multi-core.

Context Switching

Un context-switching consiste en la ejecución de una rutina perteneciente al núcleo del sistema operativo multitarea de una computadora, cuyo propósito es parar la ejecución de un hilo o proceso para dar paso a la ejecución de otro distinto.

El término se origina en la informática aunque es muy similar a la aplicación de las tareas mentales realizadas por humanos. Después de todo, la mente humana es similar a una procesador de computadora en muchos aspectos, y se compara con el context switching en términos de que como humanos, solo podemos realizar dos tareas complejas dándole foco a una de ellas e intercambiando el enfoque para avanzar con ambas.

Es de esta manera que un procesador single-core ejecuta tareas multiproceso, asignando temporalmente el estado “en espera” a la ejecución de un subproceso dado, mientras inicia o retoma la ejecución de otro proceso.

Paralelismo

El paralelismo es un tipo de concurrencia que se distingue principalmente por la capacidad que tiene el CPU para trabajar más de un proceso al mismo tiempo. Un procesador puede trabajar con una cantidad de procesos igual a su cantidad de cores, es decir, si un procesador tiene un solo core, entonces solo podrá ejecutar un proceso a la vez, por otro parte, si tenemos un procesador QUAD-CORE (4), entonces podremos ejecutar hasta cuatro procesos al mismo tiempo.

Como consumidores de smartphones que somos, entendemos entonces la importancia de las capacidades que nos brinda un equipo con estas características multi-core, asegurando alternabilidad en el trabajo del procesador pudiendo pasar de un core a otro cuando alguno falle, o ejecutando sistemas muy potentes y versátiles usando sus distintos cores en modo concurrente, entre otros beneficios.

Desde el punto de vista del desarrollo de las aplicaciones asociando los conceptos de los procesadores multi-core, estamos en la capacidad de ejecutar procesos o tareas simultáneamente, como descargar archivos, descargar imágenes, insertar datos en una base de datos, etc…

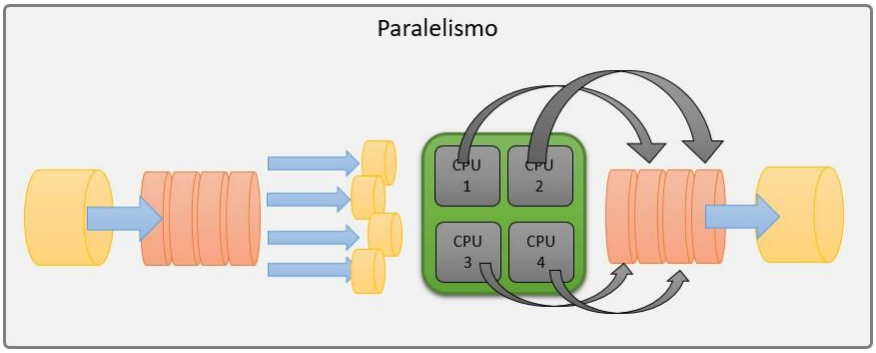


Imagen 4. Paralelismo representado por un procesador quad-core.

Threads

Competencias:

- Conocer qué son los threads en la programación de software.
- Conocer el Thread UI en el desarrollo android.
- Aplicar conocimientos en ejecución de un programa sobre el thread UI de android.

Introducción

En este capítulo aprenderemos qué son los threads (hilos) y para qué sirven en el desarrollo de nuestros sistemas. De igual forma, profundizaremos en los conceptos del thread UI en android, aplicando un ejercicio que nos permita conocer más sobre su funcionamiento.

Es muy importante conocer y aprender a trabajar con threads, ya que forman parte activa de nuestra vida laboral como programadores, y cada día se hace más común la ejecución de tareas en procesos separados y concurrentes, sobretodo en empresas consolidadas con sistemas robustos y de una cantidad importante de usuarios.

Procesos

Un proceso es una instancia de un programa en ejecución. Un programa puede tener múltiples procesos. Un proceso generalmente comienza con un solo subproceso, en android lo conocemos como main thread UI, es decir, un subproceso primario, sin embargo, en el camino del proceso principal se pueden crear múltiples subprocesos.

Threads

Los hilos (threads o subproceso) son una secuencia de ejecución de código que puede ejecutarse independientemente uno del otro. **Es la unidad de tareas más pequeña que puede ejecutar un sistema operativo**. Un programa puede ser de un solo subproceso o de subprocesos múltiples. Los threads o hilos son la manera más tradicional dentro del mundo de la programación para evitar el bloqueo de aplicaciones.

Es fundamental para la capacidad de respuesta de la aplicación a través de su main thread IU, que no sea bloqueado por subprocesos largos y complejos. Si tenemos que realizar operaciones que no son inmediatas, tenemos que asegurarnos de hacerlo en subprocesos separados (subprocesos en segundo plano o background processes).

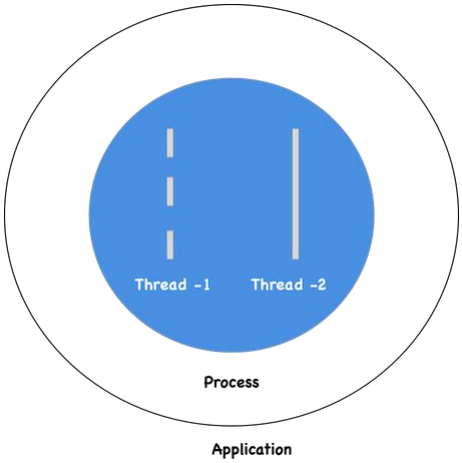


Imagen 5. Distribución de procesos y threads en una aplicación.

Por ejemplo, el código siguiente muestra un gestor que descarga una imagen desde la web en un subproceso separado a través de thread, y la asigna a una vista de tipo ImageView:

```
public void onClick(View v) {
  new Thread(new Runnable() {
    public void run() {
      Bitmap b = loadImageFromNetwork("http://example.com/image.png");
      mImageView.setImageBitmap(b);
    }
  }).start();
}
```

El ejemplo anterior, está correctamente diseñado para evitar sobrecargar el main thread; sin embargo, tenemos un detalle, android no aconseja interactuar con los objetos de vista desde un subproceso, como es el caso de asignar el bitmap al objeto mImageView. Para solucionarlo, utilizamos el método View.post(Runnable) de la siguiente forma:

```
public void onClick(View v) {
    new Thread(new Runnable() {
        public void run() {
            final Bitmap bitmap =
                    loadImageFromNetwork("http://example.com/image.png");
            mImageView.post(new Runnable() {
                public void run() {
                    mImageView.setImageBitmap(bitmap);
                }
            });
        }
    }).start();
}
```

Ahora esta implementación es segura para la main UI, ya que la operación de la red se realiza desde un thread separado mientras que el objeto de vista ImageView se manipula desde el subproceso de la IU.

No obstante, a medida que la complejidad de la operación se incrementa, este tipo de código puede volverse complicado y difícil de mantener. Para manejar interacciones más complejas con un thread, podemos considerar usar el uso de AsyncTask en estos casos, como lo explica el capítulo “AsyncTask” de esta unidad.

Main Thread Android

Al desarrollar aplicaciones de Android, siempre debemos ser conscientes de nuestro Main Thread.

El hilo principal está ocupado lidiando con cosas cotidianas, como dibujar nuestra interfaz de usuario, responder a las interacciones del usuario y, en general, de forma predeterminada, ejecutar en su mayoría el código que escribimos.

Un buen desarrollador sabe que necesita descargar tareas pesadas para un subproceso de trabajo para evitar obstruir el subproceso principal y permitir una experiencia de usuario más fluida y evitar el bloqueo de la aplicación con el mensaje “no responde la aplicación”. (error ANR, Application Not Responding).

Cuando llega el momento de actualizar la interfaz de usuario, debemos volver al hilo principal, ya que solo este puede tocar y actualizar la interfaz de usuario de la aplicación.

Una forma común de lograr esto es llamar al método runOnUiThread() de la Actividad:

```
runOnUiThread(new Runnable() {
    void run() {
        // Tu código...
    }
});
```

Con el código anterior podemos indicar las acciones a realizar sobre el hilo principal y las actualizaciones de las vistas desde la respuesta o el fin del proceso secundario o de segundo plano.

Threads síncronos y asíncronos

Repasemos los conceptos de una tarea síncrona y asíncrona con el enfoque de los hilos para comprender su funcionamiento:

- Synchronous single thread:

```
1 thread ->  |<---A---->||<----B---------->||<------C ---- >|
```

- Synchronous multi thread:

```
thread A -> |<---A --- >|
                \
thread B_________>  ->|<___B_________>|
                            \
thread C_______________________________>  ->|<_____C____>|
```

- Asynchronous single thread:

```
         A-Start ------------------------------------------ A-End
           | B-Start ----------------------------------------|--- B-End
           |     |       C-Start -------------------- C-End      |       |
           |     |        |                            |         |       |
           V     V        V                            V         V       V
1 thread->|<-A-|<--B---|<-C-|-A-|-C-|--A--|-B-|--C-->|---A---->|--B-->|
```

- Asynchronous multi thread:

```
thread A ->     |<---A--- >|
thread B ----->     |<----B--------- >|
thread C --------->     |<------C-------- >|
```

Ejercicio 1:

Obtención de datos desde el api de la nasa para ser utilizados sus datos a lo largo de la unidad.

1. A lo largo de esta unidad utilizaremos imágenes de la nasa, por lo tanto la primera acción que realizaremos es ingresar a la página http://api.nasa.gov en la cuál crearemos nuestra llave propia para consumir un método que retorna la url de la imagen del día. Ver imágenes 6 y 7.

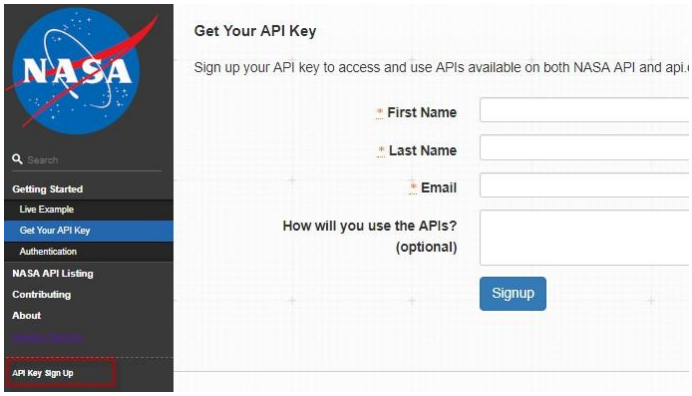


Imagen 6. Acción de solicitud de Api key Nasa.

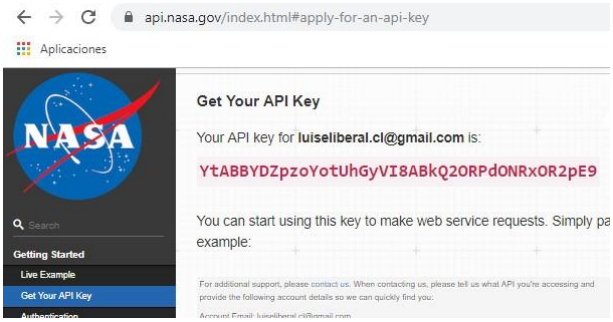


Imagen 7. Respuesta y obtención de api key.

2. Recordando ingresar en la url del método el api key que obtuvimos, a continuación utilizaremos el siguiente método para obtener la imagen del día (campo url):

https://api.nasa.gov/planetary/apod?api_key=&date=2019-08-22

Imagen 8. Respuesta api imagen del día nasa.

Ejercicio 2:

Construir una aplicación que al iniciarse descargue una misma imagen en ocho(8) oportunidades a través de hilos simultáneos utilizando la clase ThreadPoolExecutor aprovechando la máxima capacidad posible del teléfono a través de sus múltiples cores. Dependiendo del teléfono podremos visualizar dos hilos, cuatro hilos hasta ocho hilos simultáneos en el Logcat. Se deberá mostrar un progressbar que indique el progreso de la descarga y también el inicio y fin de los hilos en los logs de la aplicación.

1. Creamos un proyecto kotlin de nombre “Threads”, indicando un api mínimo número 19.
2. En nuestro archivo de layout auto-generado “activity_main.xml” crearemos dos vistas nuevas, un TextView para visualizar el contador de descargas y un ProgressBar para mostrar el avance del proceso, de la siguiente forma:

```
<TextView
    android:id="@+id/downloadsCount"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    android:layout_marginTop="50dp"
/>

<ProgressBar
    android:id="@+id/progressBar"
    style="?android:attr/progressBarStyleHorizontal"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toBottomOf="@id/downloadsCount"  />
```

3. A continuación crearemos una clase auxiliar para el proceso del hilo de descarga llamada “DownloadThread” implementado la clase Runnable, la cual ejecutará los métodos run(), sendMessage() de la clase Handler y getBitmap(), este último utilizando java.net.URL para abrir el stream de la imagen y capturarla. La clase DownloadThread sería de la siguiente forma:

```
package cl.desafiolatam.threads

import android.graphics.Bitmap
import android.graphics.BitmapFactory
import android.os.Handler
import android.util.Log
import java.net.URL

class DownloadThread(private val threadNo: Int, private val imageUrl: String, private val handler:
Handler) : Runnable {

    override fun run() {
        Log.i(TAG, "Iniciando Thread : $threadNo")
```

```
            getBitmap(imageUrl)
            sendMessage(threadNo, "Thread completado")

            Log.i(TAG, "Thread completado $threadNo")
        }

        fun sendMessage(what: Int, msg: String) {
            val message = handler.obtainMessage(what, msg)
            message.sendToTarget()
        }

        private fun getBitmap(url: String): Bitmap? {
            var bitmap: Bitmap? = null
            try {
                //Download Image
                val input = URL(url).openStream()
                // Decode Bitmap
                bitmap = BitmapFactory.decodeStream(input)
            } catch (e: Exception) {
                e.printStackTrace()
            }

            return bitmap
        }

        companion object {
            private val TAG = "DownloadThread"
        }
    }
```

4. Teniendo el layout requerido y la clase auxiliar DownloadThread, nos quedá solo adaptar nuestra actividad la “Main Activity” para iniciar estos hilos y recibir sus respuestas. Comenzaremos por crear las variables de clase que requerimos:

```
var downloadsCount: TextView? = null
var currentCount = 0
var progressBar: ProgressBar? = null
var urlTestImg = "https://apod.nasa.gov/apod/image/1908/NGC4945_HaLRGB.jpg"
var numberTimesImgWillDownload = 8
```

Nota: El valor de la variable urlTestImg corresponde a una imagen obtenida desde la consulta del api de la nasa con nuestras token.

5. En nuestro método onCreate() inicializamos las vistas:

```
downloadsCount = findViewById(R.id.downloadsCount) as TextView
progressBar = findViewById(R.id.progressBar) as ProgressBar
```

6. A continuación, en nuestro método onCreate(), creamos la instancia del ThreadPoolExecutor que utilizará la clase auxiliar DownloadThread para el procesamiento de las solicitudes, en esta ocasión consultaremos el número de cores del dispositivo para pasar parámetros al ThreadPool:

```
val numberOfCoresInPhone = Runtime.getRuntime().availableProcessors()

val threadPoolExecutor = ThreadPoolExecutor(
  numberOfCoresInPhone * 2,
  numberOfCoresInPhone * 2,
  60L,
  TimeUnit.SECONDS,
  LinkedBlockingQueue()
)

for (x in 0 until numberTimesImgWillDownload) {
  threadPoolExecutor.execute(DownloadThread(x, urlTestImg, Handler(this)))
}

downloadsCount!!.text = "Comenzando las descargas..."
```

7. Necesitamos obtener la respuesta de los avances, por ello es necesario implementar la interfaz Handler.Callback en nuestra actividad de la siguiente forma:

```
class MainActivity : AppCompatActivity(), Handler.Callback {
```

8. Se nos solicitará implementar los métodos de la interfaz, dónde incrementaremos nuestro contador de descargas, asignaremos el texto del mensaje al Textview y haremos que la progressBar muestre la finalización de las descargas:

```
override fun handleMessage(msg: Message?): Boolean {

    currentCount++
    val per = currentCount / numberTimesImgWillDownload * 100
    progressBar!!.progress = per
    if (per < 100)
      downloadsCount!!.text = "Descargas [" + currentCount + "/" +
numberTimesImgWillDownload + "]"
    else
      downloadsCount!!.text = "Todas las imagenes han sido descargadas"
    return true
}
```

9. Nuestra aplicación al iniciarse se mostrará como en las siguientes imágenes:

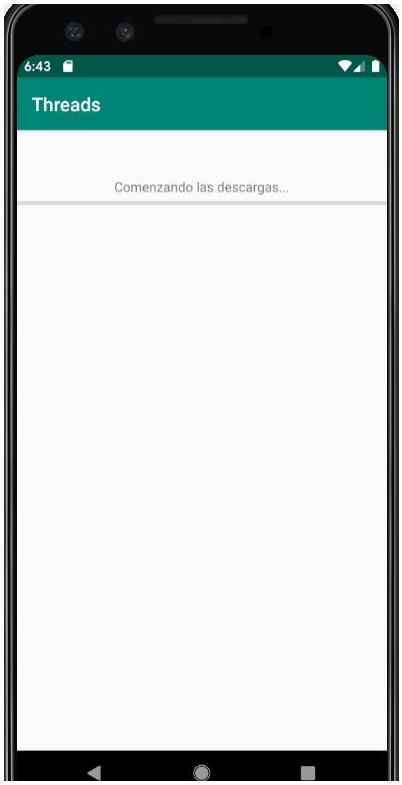


Imagen 9. Inicio de las descargas.

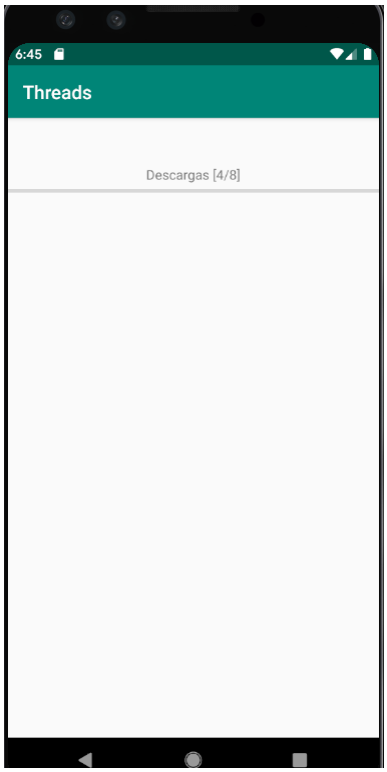


Imagen 10. Actualización descargas.

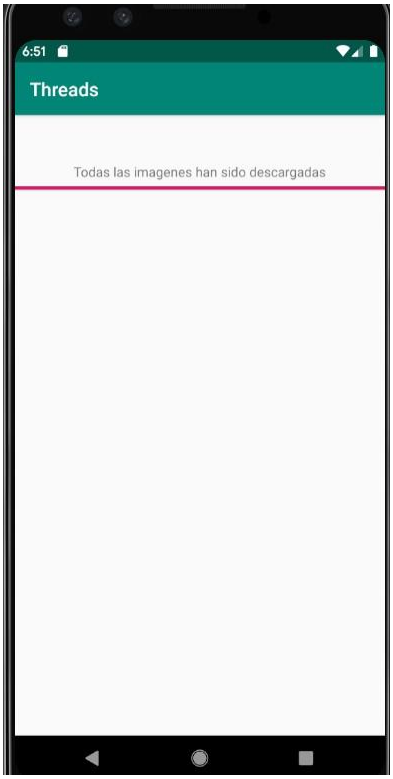


Imagen 11. Finalización de descargas.

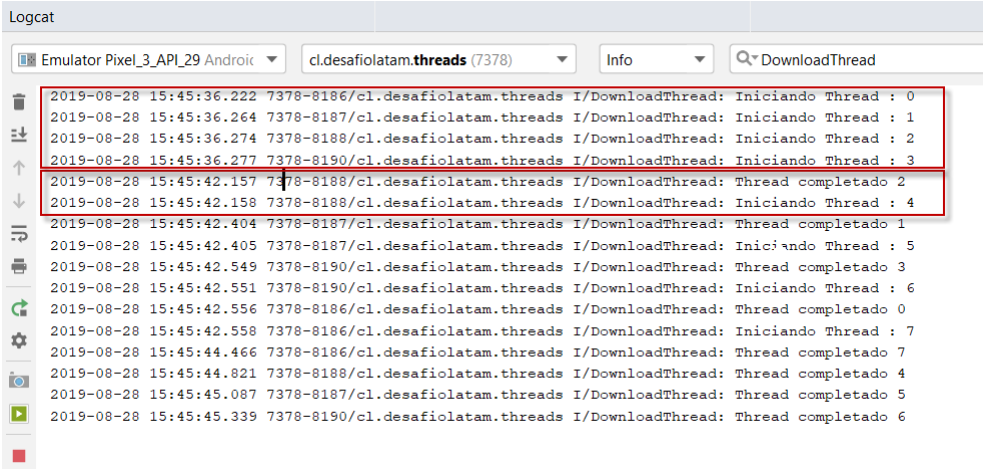


Imagen 12. Logs de ejecución de hilos.

Services

Competencias:

- Conocer los conceptos de desarrollo de procesos a través de servicios
- Crear un servicio en una aplicación android.

Introducción

En este capítulo revisaremos los conceptos de creación de procesos a través del componente Service de android. Este componente nos permite crear procesos en threads background, es decir en hilos que se ejecutan en segundo plano, no en el principal, permitiendo que dicho proceso se ejecute de una manera óptima y no se sature el hilo principal de la interfaz de usuario (Thread UI).

La importancia de desarrollar servicios radica en la optimización y performance de nuestras aplicaciones para que sean más eficientes. Es muy buena práctica construir servicios cuando realmente se requiera, en el mercado de las aplicaciones, existen muchos servicios funcionando en segundo plano con tareas que son parte de otras aplicaciones o del mismo sistema.

En este capítulo utilizaremos un servicio propio del sistema android llamado “DownloadManagerService” para demostrar cómo funcionan los servicios en una aplicación.

Service

Un servicio es un componente de aplicación que puede realizar operaciones de larga ejecución en segundo plano y que no proporciona una interfaz de usuario, es decir, se ejecuta detrás de todo proceso visual de nuestra pantalla. Un componente de la aplicación, como por ejemplo una Actividad, puede iniciar un servicio y continuará ejecutándose en segundo plano aunque el usuario cambie a otra aplicación.

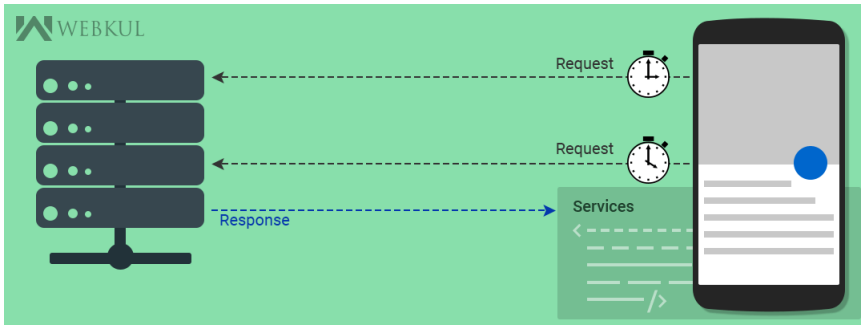


Imagen 12. Representación request/service.

Cuando requerimos descargar grandes cantidades de datos desde la web, se recomienda explícitamente no utilizar implementaciones de clientes HTTP como retrofit o volley por ejemplo, en cambio para este tipo de tareas android recomienda el uso de la clase Service llamada “Download Manager”.

Un componente puede enlazarse con un servicio para interactuar y realizar una comunicación entre procesos (IPC), por ejemplo, un servicio puede manejar transacciones de red, reproducir música, subir, bajar, leer, cerrar, archivos o interactuar con un proveedor de contenido (content provider), todo en segundo plano.

Download Manager Service

El Download Manager es un servicio del sistema que maneja descargas HTTP de larga duración. Los clientes pueden solicitar que se descargue un URI a un archivo de destino particular. El Download Manager realizará la descarga en segundo plano, se ocupará de las interacciones HTTP y volverá a intentar las descargas después de fallas o en los cambios de conectividad y reinicios del sistema.

Las aplicaciones que solicitan descargas a través de esta API deben registrar un receptor de transmisión para ACTION_NOTIFICATION_CLICKED y manejar adecuadamente cuando el usuario hace clic en una descarga en ejecución en una notificación o desde la IU de descargas.

Iniciar un Service

Un servicio está "iniciado" cuando un componente de aplicación (como una actividad) lo inicia llamando a startService(). Una vez iniciado, un servicio puede ejecutarse en segundo plano de manera indefinida, incluso si se destruye el componente que lo inició. Por lo general, un servicio iniciado realiza una sola operación y no devuelve un resultado al emisor. Por ejemplo, puede descargar o cargar un archivo a través de la red. Cuando la operación está terminada, el servicio debe detenerse por sí mismo.

Enlazar a un Service

Un servicio es de “de enlace” cuando un componente de la aplicación se vincula a él llamando al método bindService(). Un servicio de enlace ofrece una interfaz cliente-servidor que permite que los componentes interactúen con el servicio, envien solicitudes, obtengan resultados e incluso lo hagan en distintos procesos con la comunicación entre procesos (IPC). Un servicio de enlace se ejecuta solamente mientras otro componente de aplicación está enlazado con él. Se pueden enlazar varios componentes con el servicio a la vez, pero cuando todos ellos se desenlazan, el servicio se destruye.

Utilizar un Service o un SubProceso

Un servicio es simplemente un componente que puede ejecutarse en segundo plano, incluso cuando el usuario no está interactuando con tu aplicación. Por lo tanto, debes crear un servicio solo si eso es lo que necesitas.

Si requerimos realizar un trabajo fuera del subproceso principal, pero solo mientras el usuario interactúa con la aplicación, es oportuno crear un subproceso, en lugar de un servicio, por ejemplo, si se desea reproducir música, pero sólo mientras se ejecute la actividad, podemos crear un subproceso en el método onCreate(), comenzar a ejecutarlo en onStart() y luego detenerlo en onStop().

Es importante considerar utilizar AsyncTask o HandlerThread, en lugar de la clase Thread tradicional.

Si utilizamos un servicio, este se ejecuta en el subproceso principal de tu aplicación de forma predeterminada, por lo que debes crear un subproceso nuevo dentro del servicio si este realiza operaciones intensivas o de bloqueo.

Declaración de un servicio en el manifiesto

Al igual que las actividades (y otros componentes), debes declarar todos los servicios en el archivo de manifiesto de tu aplicación.

Para declarar tu servicio, agrega un elemento como campo secundario del elemento de .

```
<manifest ... >
  ...
  <application ... >
      <service android:name=".ExampleService" />
      ...
  </application>
</manifest>
```

Para garantizar que tu aplicación sea segura, utiliza siempre una intent explícita cuando inicia o enlaza su Service y no declares filtros de intents para el servicio.

Extensión de la clase Service

Dado que la mayoría de los servicios iniciados no necesitan manejar múltiples solicitudes simultáneamente (lo que, en realidad, puede ser un escenario peligroso de subprocesos múltiples), es mejor implementar un servicio utilizando la clase IntentService.

La clase IntentService realiza las siguientes tareas:

- Crea un subproceso de trabajo predeterminado que ejecute todas las intents enviadas a onStartCommand(), independientemente del subproceso principal de tu aplicación.
- Crea una cola de trabajo que pase una intent a la vez a tu implementación onHandleIntent(), para no preocuparnos por varios subprocesos.
- Detiene el servicio una vez que se han manejado todas las solicitudes de inicio, para no tener que llamar al método stopSelf().
- Proporciona la implementación predeterminada de onBind() que devuelve NULL.
- Proporciona una implementación predeterminada de onStartCommand() que envía el intent a la cola de trabajo y luego a la implementación del método onHandleIntent().

Según lo anterior, lo único que debemos hacer es implementar el método onHandleIntent() para hacer el trabajo proporcionado por el cliente; sin embargo, se debe crear un pequeño constructor para el servicio.A continuación, un ejemplo de implementación de IntentService:

```
public class HelloIntentService extends IntentService {

  /**
   * A constructor is required, and must call the super IntentService(String)
   * constructor with a name for the worker thread.
   */
  public HelloIntentService() {
      super("HelloIntentService");
  }

  /**
   * The IntentService calls this method from the default worker thread with
   * the intent that started the service. When this method returns, IntentService
   * stops the service, as appropriate.
   */
  @Override
  protected void onHandleIntent(Intent intent) {
      // Normally we would do some work here, like download a file.
      // For our sample, we just sleep for 5 seconds.
      try {
          Thread.sleep(5000);
      } catch (InterruptedException e) {
          // Restore interrupt status.
          Thread.currentThread().interrupt();
      }
  }
}
```

Implementación de callbacks del ciclo de vida

Al igual que una actividad, un servicio tiene métodos callback del ciclo de vida que tú puedes implementar para controlar los cambios en el estado del servicio y realizar trabajos en los momentos adecuados.

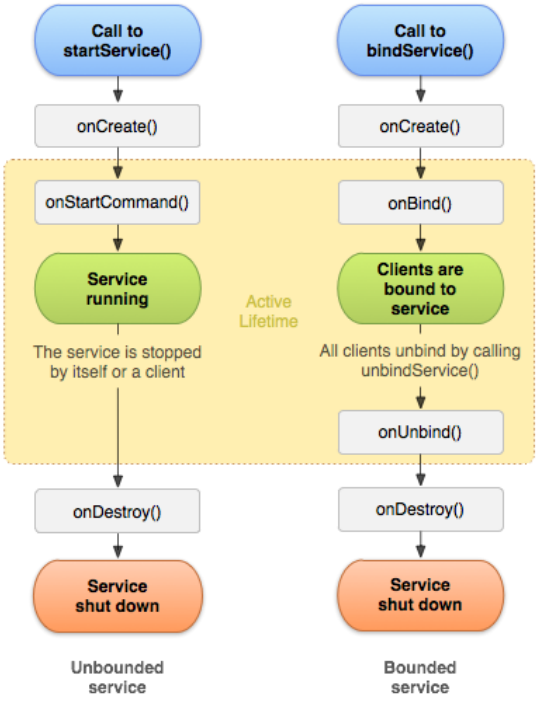


Imagen 13. Ciclo de vida Service.

Ejercicio 3:

Crear un proyecto de nombre “ServiceDownloader” que utilice el api de la nasa para descargar la imágen del día en formato HD, a través de un proceso background utilizando la clase Download Manager de android.

1. Creamos dos vistas una TextView y otra Button en nuestro archivo de layout activity_main.xml de la siguiente manera:

```
<TextView
    android:id="@+id/title"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Servicio de descarga"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent"/>


<Button
    android:layout_width="wrap_content"
    android:id="@+id/btnDownload"
    android:text="INICIAR"
    android:layout_centerInParent="true"
    android:layout_height="wrap_content"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/title"
/>
```

2. En nuestra actividad principal creamos una variable para identificar nuestra descarga de tipo Long:

```
private var downloadId : Long? = null
```

3. Creamos una variable de clase de sólo lectura para el manejo de la respuesta del servicio a través de un componente BroadcastReceiver de la siguiente manera:

```
 //MANEJO DE RESPUESTA DE SERVICIO CON NOTIFICACION BROADCAST
private val onDownloadComplete = object : BroadcastReceiver() {
  override fun onReceive(context: Context, intent: Intent) {
    //Fetching el id de download recibido
     val id = intent.getLongExtra(DownloadManager.EXTRA_DOWNLOAD_ID, -1)
    //Verificamos si el id recibido es el correspondiente a nuestra descarga
    if (downloadId == id) {
      Toast.makeText(this@MainActivity, "Descarga completada", Toast.LENGTH_SHORT).show()
    }
  }
}
```

4. Creamos un método para gestionar la descarga e invocar a la clase DownloadManager, donde indicaremos donde se guardará la imagen una vez se haya descargado en nuestro dispositivo, la url de la imagen en HD desde nuestra api. Este método lo llamaremos initDownload:

```
private fun initDownload() {
  val file = File(getExternalFilesDir(null), "nasaImagenDelDia")

  val request = DownloadManager.Request(Uri.parse("https://apod.nasa.gov/apod/image/1908/M61-
HST-ESO-L.jpg"))
    .setTitle("Imagen del día de la Nasa")//Titulo de la notificación de la descarga
    .setDescription("Descargando")//Descripción
    .setNotificationVisibility(DownloadManager.Request.VISIBILITY_VISIBLE)//Visibilidad de la
notificación de descarga
    .setDestinationUri(Uri.fromFile(file))//Uri del destino del archivo
    .setAllowedOverMetered(true)
  val downloadManager = getSystemService(Context.DOWNLOAD_SERVICE) as DownloadManager
  downloadId = downloadManager.enqueue(request)//Colocamos el request en la cola del download
manager
}
```

5. En nuestro método de actividad onCreate(), instanciamos el botón de la vista para indicar que su acción será ejecutar el método initDownload(), y también registramos la petición de recibir un “onDownloadComplete” para el BroadcastReceiver, esto de la siguiente manera:

```
val btnIniciar: Button = findViewById(R.id.btnDownload)
//INICIALIZAMOS EL REGISTRO DEL ESCUCHA ESPERA BROADCAST PARA LA RESPUESTA DEL
SERVICIO
registerReceiver(onDownloadComplete,
IntentFilter(DownloadManager.ACTION_DOWNLOAD_COMPLETE))
//ACCION DEL BOTON PARA INICIAR LA DESCARGA
btnIniciar.setOnClickListener {initDownload()}
```

6. Finalmente debemos eliminar el registro de la escucha del broadcast cuando la actividad sea eliminada, de la siguiente manera:

```
public override fun onDestroy() {
  super.onDestroy()
  //ELIMINAMOS EL ESCUCHA ESPERA DEL BROADCAST
  unregisterReceiver(onDownloadComplete)
}
```

7. Por último, no se nos olvide agregar los permisos a nuestro archivo manifest:

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

8. Es importante que ejecuten este proyecto con un dispositivo android conectado a android studio por medio de USB para utilizar correctamente de las funcionalidades que nos proporciona el sistema para este tipo de tarea. El resultado de nuestra aplicación de manera secuencial sería como lo muestran las siguientes imágenes:

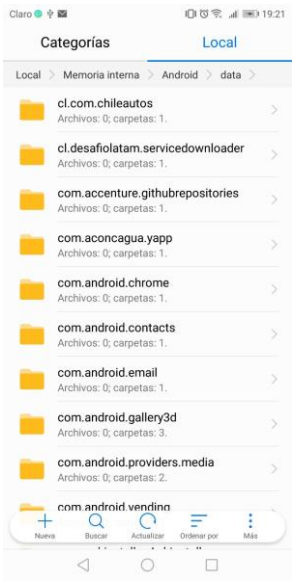


Imagen 14. Grupo de imagenes - Ejercicio 3.

Alarm Manager

Competencias:

- Conocer sobre la configuración de tareas con AlarmManager
- Crear una tarea que se ejecute con AlarmManager

Introducción

En este capítulo profundizaremos los conceptos asociados a las tareas programadas, las cuales se utilizan para ejecutar un proceso dada una fecha y hora determinada para su inicio. Las tareas programadas son bastante comunes en los sistemas de las empresas, ya que suele ocurrir que exista la necesidad de ejecutar un proceso en la noche según la información recopilada durante el día, y a su vez, se requiera disponer de la información actualizada a primera hora del siguiente día de trabajo.

En android tenemos un componente llamado AlarmManager para facilitarnos la programación de este tipo de tareas en nuestras aplicaciones móviles.

Alarm Manager

Esta clase proporciona acceso a los servicios de alarma del sistema operativo android. Estos servicios permiten programar tareas para que se ejecuten en algún momento del futuro. Cuando una alarma se desactiva, el Intent que se registró en su momento por un broadcast receiver, inicia automáticamente la aplicación contenedora de la tarea, si está no se está ejecutando.

El AlarmManager mantiene un bloqueo de activación de la CPU siempre que el método onReceive() del receptor de alarma se esté ejecutando. Esto garantiza que el teléfono no se suspenderá hasta que haya terminado de manejar la tarea programada. Una vez que se recibe la notificación de finalización del proceso en el método onReceive(), Alarm Manager libera este bloqueo de activación. Esto significa que, en algunos casos, el teléfono se dormirá tan pronto como se complete su método onReceive().

Si el receiver de alarma llamó al método Context.startService(), es posible que el teléfono se suspenda antes de que se inicie el servicio solicitado. Para evitar esto, se crea un BroadcastReceiver y un Service con políticas de bloqueo de activación propias, para garantizar que el teléfono continúe funcionando hasta que el servicio esté disponible.

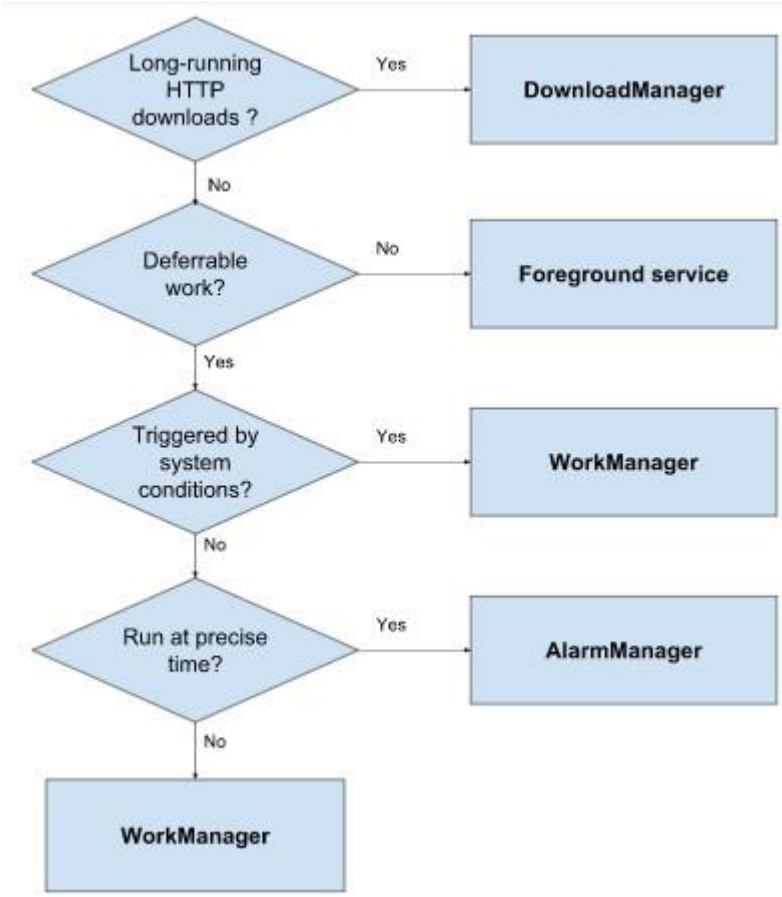


Imagen 15. Diagrama tareas y componentes según el criterio.

Características del AlarmManager

- Te permite disparar Intents en tiempos programados o intervalos de tiempo programados.
- Puedes usar AlarmManager junto con el componente BroadCastReceiver para iniciar servicios o ejecutar otras operaciones.
- Funcionan fuera de la aplicación, lo que te permite activar acciones o eventos fuera de tu aplicación sin necesidad de que esté funcionando.
- AlarmManager ayuda a minimizar los requerimientos de recursos en nuestras aplicaciones, sin necesidad de tener procesos background contínuamente escuchando o corriendo.

Repetir una alarma

Las alarmas repetidas son una buena opción para programar eventos regulares o búsquedas de datos. Una alarma repetitiva tiene las siguientes características:

- Un AlarmType.
- Un tiempo de activación, si el tiempo de activación que especifica está en el pasado, la alarma se activa de inmediato.
- El intervalo de la alarma, por ejemplo, una vez al día, cada hora, cada 5 minutos.
- Un Pending Intent que se dispara cuando se activa la alarma. Cuando configuramos una segunda alarma que usa el mismo PendingIntent, esta reemplaza la alarma original o inicial.

Tiempos y Tipos de Alarma

Una de las primeras consideraciones al usar una alarma repetitiva es cuál debería ser su tipo.

Hay dos tipos generales de reloj para alarmas: "tiempo real transcurrido" y "reloj en tiempo real" (RTC). El tiempo real transcurrido usa el "tiempo desde el inicio del sistema" como referencia, y el reloj en tiempo real usa el tiempo UTC (reloj de pared), esto significa que el tiempo real transcurrido es adecuado para configurar una alarma basada en el paso del tiempo (por ejemplo, una alarma que se dispara cada 30 segundos) ya que no se ve afectada por la zona horaria / localidad. El tipo de reloj en tiempo real es más adecuado para alarmas que dependen de la ubicación actual.

Ambos tipos tienen una versión de "activación", que dice activar la CPU del dispositivo si la pantalla está apagada. Esto asegura que la alarma se disparará a la hora programada. Esto es útil si la aplicación depende del tiempo, por ejemplo, si tiene una ventana limitada para realizar una operación en particular. Si no utilizamos la versión de activación del tipo de alarma, todas las alarmas que se repiten se activarán la próxima vez que el dispositivo esté activado.

Si necesitamos que la alarma se active en un intervalo de tiempo particular (por ejemplo, cada media hora), usamos uno de los tipos de tiempo real transcurrido. En general, esta es la mejor opción.

Si necesitamos que la alarma suene a una hora particular del día, podemos elegir uno de los tipos de reloj basados en tiempo real.

Tipos de Alarma

- **ELAPSED_REALTIME**: activa el PendingIntent en función del tiempo transcurrido desde que se inició el dispositivo, pero no activa el dispositivo. El tiempo transcurrido incluye cualquier momento durante el cual el dispositivo estuvo dormido.

- **ELAPSED_REALTIME_WAKEUP**: despierta el dispositivo y dispara el PendingIntent después de que haya transcurrido el tiempo especificado desde el inicio del dispositivo.

```
// Hopefully your alarm will have a lower frequency than this!
alarmMgr?.setInexactRepeating(
    AlarmManager.ELAPSED_REALTIME_WAKEUP,
    SystemClock.elapsedRealtime() + AlarmManager.INTERVAL_HALF_HOUR,
    AlarmManager.INTERVAL_HALF_HOUR,
    alarmIntent
)
```

- **RTC**: activa el PendingIntent a la hora especificada pero no activa el dispositivo.

- **RTC_WAKEUP**: despierta el dispositivo para disparar el PendingIntent a la hora especificada. Ejemplo:

```
// Set the alarm to start at approximately 2:00 p.m.
val calendar: Calendar = Calendar.getInstance().apply {
    timeInMillis = System.currentTimeMillis()
    set(Calendar.HOUR_OF_DAY, 14)
}

// With setInexactRepeating(), you have to use one of the AlarmManager interval
// constants--in this case, AlarmManager.INTERVAL_DAY.
alarmMgr?.setInexactRepeating(
        AlarmManager.RTC_WAKEUP,
        calendar.timeInMillis,
        AlarmManager.INTERVAL_DAY,
        alarmIntent
)
```

Ejercicio 4:

Crear un proyecto de nombre “AlarmManagerNotification” que muestre un título centrado con el mensaje “Desafío Latam Notification Example” en su actividad principal y ejecute una notificación a los 10 segundos de iniciarse la aplicación utilizando las clases AlarmManager, AlarmReceiver, BroadCastReceiver, Notification, NotificationManager, IntentService.

1. Adaptamos nuestro layout activity_main.xml con el mensaje requerido:

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="DesafioLatam Notification Example"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    android:textAlignment="center"
    android:textSize="22dp"
    android:padding="20dp"
    android:textStyle="bold"
    />
```

2. Creamos un servicio de notificación llamado “NotificationService” extendiendo de la clase IntentService, para el funcionamiento de la tarea de notificar cuando la alarma haya llegado a su tiempo de 10 segundos, el código de la clase sería el siguiente:

```
package cl.desafiolatam.alarmmanagernotification

import android.annotation.SuppressLint
import android.app.*
import android.content.Context
import android.content.Intent
import android.graphics.BitmapFactory
import android.graphics.Color
import android.media.RingtoneManager
import android.os.Build
import android.support.v4.app.NotificationCompat
import java.util.*

class NotificationService : IntentService("NotificationService") {
  private lateinit var mNotification: Notification
  private val mNotificationId: Int = 1000

  @SuppressLint("NewApi")
  private fun createChannel() {


    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {

      // Create the NotificationChannel, but only on API 26+ because
      // the NotificationChannel class is new and not in the support library

      val context = this.applicationContext
```

```
        val notificationManager = context.getSystemService(Context.NOTIFICATION_SERVICE) as
NotificationManager

        val importance = NotificationManager.IMPORTANCE_HIGH
        val notificationChannel = NotificationChannel(CHANNEL_ID, CHANNEL_NAME, importance)
        notificationChannel.enableVibration(true)
        notificationChannel.setShowBadge(true)
        notificationChannel.enableLights(true)
        notificationChannel.lightColor = Color.parseColor("#e8334a")
        notificationChannel.description = "Este es un ejemplo del canal de notificación con
AlarmManager para estudiantes de Desafío Latam"
        notificationChannel.lockscreenVisibility = Notification.VISIBILITY_PUBLIC
        notificationManager.createNotificationChannel(notificationChannel)
    }

  }

  companion object {

    const val CHANNEL_ID = "cl.desafiolatam.alarmmanagernotification.CHANNEL_ID"
    const val CHANNEL_NAME = "Notificación de Prueba"
  }


  @SuppressLint("PrivateResource")
  override fun onHandleIntent(intent: Intent?) {

    //Create Channel
    createChannel()


    var timestamp: Long = 0
    if (intent != null && intent.extras != null) {
        timestamp = intent.extras!!.getLong("timestamp")
    }

    if (timestamp > 0) {

        val context = this.applicationContext
        val notificationManager: NotificationManager =
getSystemService(Context.NOTIFICATION_SERVICE) as NotificationManager
        val notifyIntent = Intent(this, ResultActivity::class.java)

        val title = "Notificación de Prueba"
        val message = "Has recibido una notificación de prueba. Esta notificación te llevará a los
detalles."

        notifyIntent.putExtra("title", title)
        notifyIntent.putExtra("message", message)
        notifyIntent.putExtra("notification", true)

        notifyIntent.flags = Intent.FLAG_ACTIVITY_NEW_TASK

        val calendar = Calendar.getInstance()
        calendar.timeInMillis = timestamp
```

```
        val pendingIntent = PendingIntent.getActivity(context, 0, notifyIntent,
PendingIntent.FLAG_UPDATE_CURRENT)
        val res = this.resources
        val uri = RingtoneManager.getDefaultUri(RingtoneManager.TYPE_NOTIFICATION)

        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) mNotification =
Notification.Builder(this, CHANNEL_ID)
            // Set the intent that will fire when the user taps the notification
            .setContentIntent(pendingIntent)
            .setSmallIcon(R.drawable.notification_bg)
            .setLargeIcon(BitmapFactory.decodeResource(res, R.mipmap.ic_launcher))
            .setAutoCancel(true)
            .setContentTitle(title)
            .setStyle(Notification.BigTextStyle()
                .bigText(message))
            .setContentText(message).build() else {

            mNotification = NotificationCompat.Builder(context, CHANNEL_ID)
                // Set the intent that will fire when the user taps the notification
                .setContentIntent(pendingIntent)
                .setSmallIcon(R.drawable.notification_bg)
                .setLargeIcon(BitmapFactory.decodeResource(res, R.mipmap.ic_launcher))
                .setAutoCancel(true)
                .setContentTitle(title)
                .setSound(uri)
                .setContentText(message).build()

        }

        // mNotificationId is a unique int for each notification that you must define
        notificationManager.notify(mNotificationId, mNotification)
    }
  }
}
```

3. A continuación crearemos una clase llamada “AlarmReceiver” que implemente el BroadCastReceiver, donde se haga instancia del servicio “NotificationService” y sea iniciado dentro del método onReceive(), de la siguiente manera:

```
package cl.desafiolatam.alarmmanagernotification

import android.content.BroadcastReceiver
import android.content.Context
import android.content.Intent

class AlarmReceiver : BroadcastReceiver() {

  override fun onReceive(context: Context, intent: Intent) {

    val service = Intent(context, NotificationService::class.java)
    service.putExtra("reason", intent.getStringExtra("reason"))
    service.putExtra("timestamp", intent.getLongExtra("timestamp", 0))

    context.startService(service)
  }

}
```

5. Es necesario que agreguemos unamos nuestro AlarmReceiver junto a una instancia de AlarmManager a través de un PendingIntent que se encargará de disparar la petición broadcast una vez cumplido el tiempo de la alarma, para ello, crearemos una nueva clase llamada “NotificationUtils” con el siguiente código:

```
package cl.desafiolatam.alarmmanagernotification

import android.app.Activity
import android.app.AlarmManager
import android.app.PendingIntent
import android.content.Intent
import java.util.*

class NotificationUtils {

  fun setNotification(timeInMilliSeconds: Long, activity: Activity) {
    //------------ alarm settings start ---------------- //
    if (timeInMilliSeconds > 0) {


      val alarmManager = activity.getSystemService(Activity.ALARM_SERVICE) as AlarmManager
      val alarmIntent = Intent(activity.applicationContext, AlarmReceiver::class.java) // AlarmReceiver1
= broadcast receiver

      alarmIntent.putExtra("reason", "notification")
      alarmIntent.putExtra("timestamp", timeInMilliSeconds)


      val calendar = Calendar.getInstance()
      calendar.timeInMillis = timeInMilliSeconds
```

```
        val pendingIntent = PendingIntent.getBroadcast(activity, 0, alarmIntent,
PendingIntent.FLAG_CANCEL_CURRENT)
        alarmManager.set(AlarmManager.RTC_WAKEUP, calendar.timeInMillis, pendingIntent)

    }
    //------------ end of alarm settings ----------------- //
  }
}
```

6. A continuación, crearemos una nueva actividad para emular una pantalla de detalle una vez se haga click sobre la notificación, la llamaremos “ResultActivity”, para ello crearemos su archivo de diseño y su archivo de actividad de la siguiente forma:

- activity_result.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
                                    android:layout_width="match_parent"
                                    android:layout_height="match_parent"
                                    xmlns:tools="http://schemas.android.com/tools"
                                    xmlns:app="http://schemas.android.com/apk/res-auto"
                                    tools:context=".ResultActivity">

<TextView
android:id="@+id/txtTitleView"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:textStyle="bold"
android:textSize="22dp"
android:padding="10dp"
app:layout_constraintBottom_toTopOf="@+id/txtMsgView"
app:layout_constraintEnd_toEndOf="parent"
app:layout_constraintStart_toStartOf="parent" />

<TextView
android:id="@+id/txtMsgView"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="Esperando por la notificación..."
android:textAlignment="center"
app:layout_constraintBottom_toBottomOf="parent"
app:layout_constraintEnd_toEndOf="parent"
app:layout_constraintStart_toStartOf="parent"
app:layout_constraintTop_toTopOf="parent"
android:padding="20dp"/>

</android.support.constraint.ConstraintLayout>
```

- ResultActivity.kt:

```
package cl.desafiolatam.alarmmanagernotification

import android.os.Bundle
import android.support.v7.app.AppCompatActivity
import kotlinx.android.synthetic.main.activity_result.*

class ResultActivity : AppCompatActivity() {

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_result)

    if (intent.getBooleanExtra("notification", false)) {
      txtTitleView.text = intent.getStringExtra("title")
      txtMsgView.text = intent.getStringExtra("message")

    }
  }
}
```

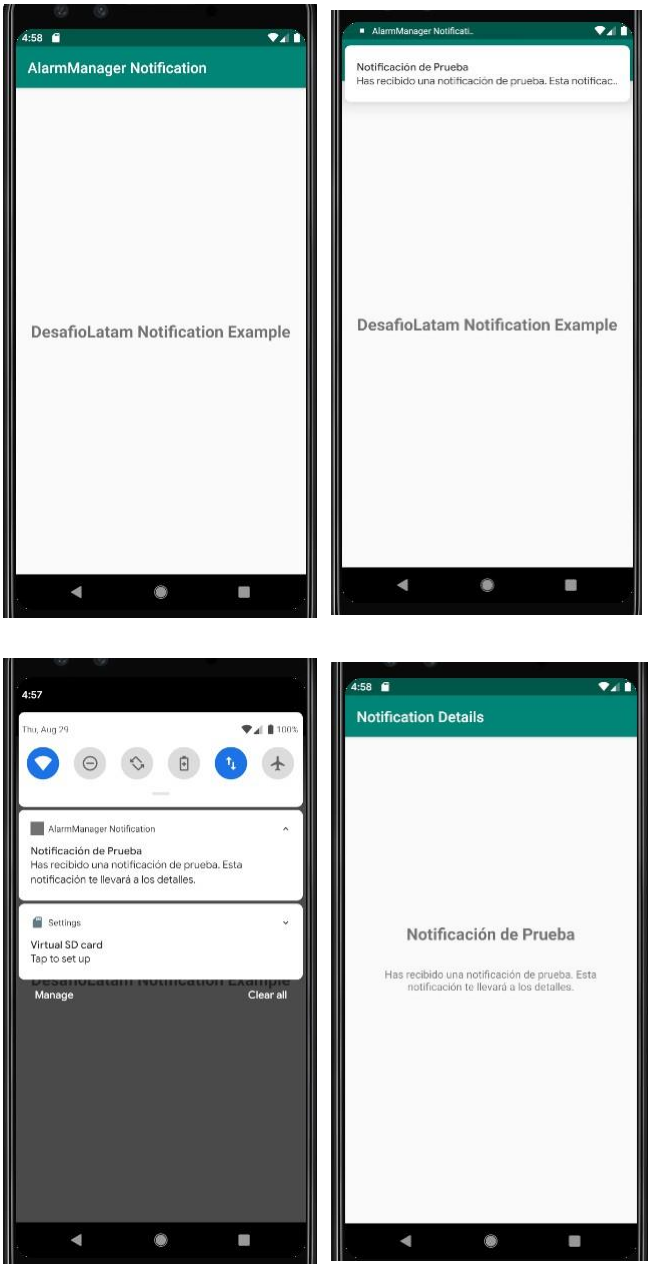
7. Necesitamos registrar la nueva actividad, el servicio y el broadcast receiver en nuestro archivo manfiest, antes del cierre del tag `</application>`, de la siguiente manera:

```
<receiver
    android:name=".AlarmReceiver"
    android:enabled="true" />
<service
    android:name=".NotificationService"
    android:enabled="true" />
<activity
    android:name=".ResultActivity"
    android:excludeFromRecents="true"
    android:label="Notification Details"
    android:launchMode="singleTask"
    android:taskAffinity="" />
```

8. Por último, editamos nuestra actividad principal “MainActivity” para indicar el tiempo de ejecución de la alarma a través de la clase NotificationUtils ya creada, de la siguiente manera:

```
package cl.desafiolatam.alarmmanagernotification

import android.support.v7.app.AppCompatActivity
import android.os.Bundle
import java.util.*

class MainActivity : AppCompatActivity() {

  private val mNotificationTime = Calendar.getInstance().timeInMillis + 10000//La alarma se ejecutará
10 segundos desde el momento en que se inicie la app
  private var mNotified = false

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    if (!mNotified) {
      NotificationUtils().setNotification(mNotificationTime, this@MainActivity)
    }
  }
}
```

9. Ejecutando nuestra aplicación podremos visualizar un mensaje de inicio y la llegada de una notificación pasados los 10 segundos, que a su vez, al hacer click sobre esta, nos llevará a una pantalla con mensajes de detalle como lo demuestran las siguientes imágenes:



Imágen 16. Grupo de imágenes - Ejercicio 4.

Work Manager

Competencias:

- Conocer la agenda de tareas con WorkManager en android.
- Implementar WorkManager en un proyecto android.

Introducción

En este capítulo estudiaremos la creación de tareas programables en android con la biblioteca WorkManager, la cual garantiza la ejecución óptima de los procesos agendados en segundo plano. WorkManager es una de las librerías más modernas para crear tareas desarrollada por android.

Las tareas programables son parte esencial de la actividad laboral, como desarrolladores en ocasiones nos solicitarán ejecutar algún tipo de operación repetitiva y a determinado horario, es en este tipo de ocasiones donde nos será muy útil aprender a utilizar WorkManager.

WorkManager

WorkManager es uno de los componentes de arquitectura de Android y parte de Android Jetpack, una nueva y obstinada forma de construir aplicaciones modernas de Android.

WorkManager es una biblioteca de Android que ejecuta trabajos en segundo plano diferibles cuando se cumplen las restricciones del trabajo .Está destinado a tareas que requieren garantizar que el sistema los ejecutará incluso si la aplicación sales.

En otras palabras, WorkManager proporciona una API amigable con la batería que encapsula años de evolución de las restricciones de comportamiento en segundo plano de Android. ¡Esto es crítico para las aplicaciones de Android que necesitan ejecutar tareas en segundo plano.

Métodos WorkManager

enqueue: después de que la tarea ha sido definida por la clase Worker y sabemos también el tipo de tarea (periódica o no), podemos programarla con el método enqueue().

cancel: si decidimos cancelar toda la tarea, podríamos usar el método cancelAllWork() o el método cancelWorkById(id).

```
WorkRequest request = new OneTimeWorkRequest.Builder(FooWorker.class).build();
 workManager.enqueue(request);
 workManager.cancelWorkById(request.getId());
```

Métodos WorkManager

- **enqueue**: después de que la tarea ha sido definida por la clase Worker y sabemos también el tipo de tarea (periódica o no), podemos programarla con el método enqueue().

- **cancel**: si decidimos cancelar toda la tarea, podríamos usar el método cancelAllWork() o el método cancelWorkById(id).

```
WorkRequest request = new OneTimeWorkRequest.Builder(FooWorker.class).build();
 workManager.enqueue(request);
 workManager.cancelWorkById(request.getId());
```


- **beginWith**: permite separar los trabajos o tareas y priorizarlos.

```
WorkRequest request1 = new OneTimeWorkRequest.Builder(FooWorker.class).build();
 WorkRequest request2 = new OneTimeWorkRequest.Builder(BarWorker.class).build();
 WorkRequest request3 = new OneTimeWorkRequest.Builder(BazWorker.class).build();
 workManager.beginWith(request1, request2).then(request3).enqueue();
```

- **beginUniqueWork**: este método nos permite comenzar cadenas de tareas únicas para situaciones en las que solo deseamos que una cadena con un nombre determinado esté activa a la vez. Por ejemplo, es posible que necesitemos que solo una operación de sincronización esté activa.

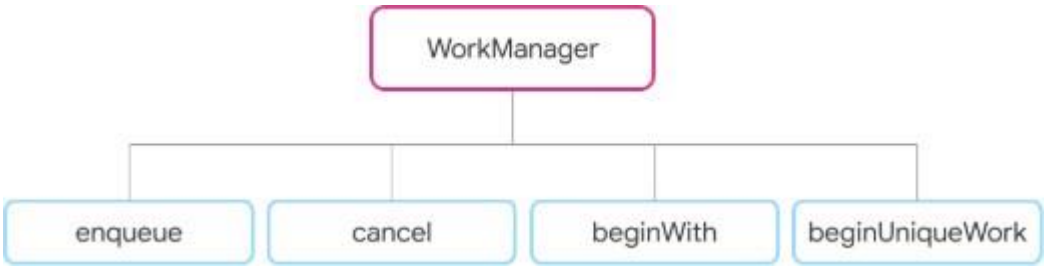


Imagen 17. Métodos del WorkManager.

Evaluar el uso de WorkManager

WorkManager ejecuta la tarea en segundo plano si el proceso cumple varias restricciones, independientemente de si el proceso de la aplicación está activo o no.

WorkManager está destinado a tareas que requieren una garantía de que el sistema las ejecutará, incluso si la aplicación se cierra, no está destinado a trabajos/tareas en segundo plano que requieran ejecución inmediata o requieran ejecución en un momento exacto. Si necesitamos que una tarea se ejecute a una hora exacta (como un reloj despertador o un recordatorio de evento), es mejor que utilicemos el AlarmManager. Para tareas que deben ejecutarse de inmediato pero es de larga ejecución, a menudo deberá asegurarse de que el trabajo se ejecute en primer plano; ya sea limitando la ejecución al primer plano o utilizando un Servicio de primer plano.

WorkManager está diseñado para tareas que son diferibles, es decir, que no requieren ejecutarse de inmediato y que se deben ejecutar de manera confiable incluso si la aplicación se cierra o el dispositivo se reinicia. Por ejemplo:

- Envío de registros o análisis a servicios de backend.
- Sincronizar periódicamente los datos de la aplicación con un servidor.

WorkManager no está diseñado para el trabajo en segundo plano en proceso que se puede finalizar de forma segura si el proceso de la aplicación desaparece o para tareas que requieren una ejecución inmediata.

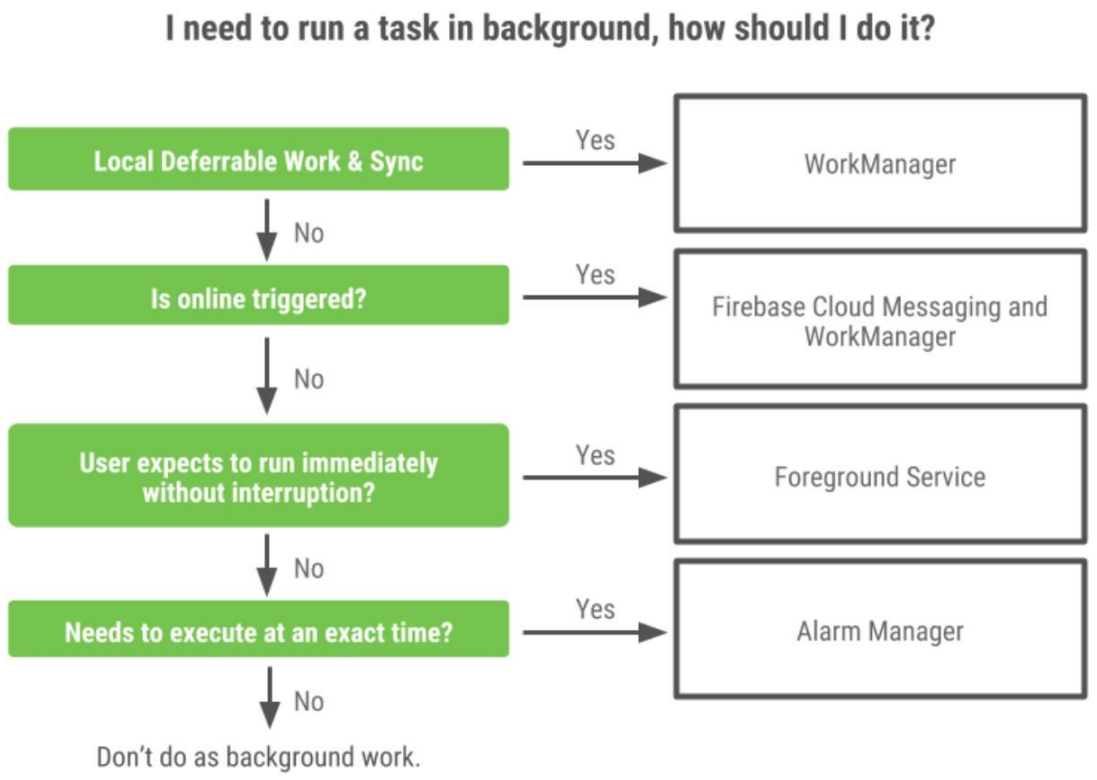


Imagen 18. Cuándo utilizar WorkManager

Hay muchas situaciones diferentes en las que necesita ejecutar una tarea en segundo plano y, por lo tanto, diferentes soluciones para ejecutar el trabajo en segundo plano.

Beneficios de utilizar WorkManager

- Maneja la compatibilidad con diferentes versiones del sistema operativo
- Sigue las mejores prácticas de salud del sistema
- Admite tareas asincrónicas únicas y periódicas.
- Admite tareas encadenadas con entrada / salida
- Le permite establecer restricciones cuando se ejecuta la tarea
- Garantiza la ejecución de tareas, incluso si la aplicación o el dispositivo se reinicia.

Ejercicio 1:

Crear una aplicación que realice un trabajo en segundo plano utilizando la librería WorkManager. La tarea a realizar será contar del 1 al 10, haciendo esperar un segundo al sistema por cada número (cuenta real). La app ha de informar por medio de un mensaje Toast cuando se inicia la tarea y cuando se termina, mostrando también por consola o log la ejecución de la tarea.

1. Crear un proyecto android llamado “WorkManager”, seleccionando blank activity y api mínima 19.
2. Agregamos las librerías necesarias para trabajar con WorkManager en nuestro archivo build.gradle app.

```
implementation "android.arch.work:work-runtime-ktx:1.0.0-beta01"
androidTestImplementation  "android.arch.work:work-testing:1.0.0-beta01"
```

1. Editar el layout principal “activity_main” para agregar un título con textView denominado “Tarea con WorkManager”, un botón que indique “Ejecutar Tarea”. El código es el siguiente:

```
<?xml **version**="1.0" encoding="utf-8"?>
<android.support.**constraint**.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:**app**="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

  <TextView android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:text="TAREA CON WORKMANAGER"
      **app**:layout_constraintBottom_toTopOf="@id/button_start"
      **app**:layout_constraintLeft_toLeftOf="parent"
      **app**:layout_constraintRight_toRightOf="parent"
      **app**:layout_constraintTop_toTopOf="parent"
      android:textAlignment="center"
      android:textSize="20dp"
      android:textStyle="bold"
  />
  <Button android:id="@+id/button_start"
     android:layout_width="match_parent"
     android:layout_height="wrap_content"
     **app**:layout_constraintBottom_toTopOf="parent"
     **app**:layout_constraintLeft_toLeftOf="parent"
     **app**:layout_constraintRight_toRightOf="parent"
     **app**:layout_constraintTop_toTopOf="parent"
     android:text="EJECUTAR TAREA"
     android:backgroundTint="#63c6b3"
     android:textColor="#FFFFFF"
  />

</android.support.**constraint**.ConstraintLayout>
```

1. Crear un Worker, a través de una nueva clase denominada “TareaWorkerOne” donde utilizaremos la librería androidx.work y sobreescribiremos el método doWork() para realizar la cuenta del 1 al 10, según ha sido indicado, de la siguiente manera:

```
import android.content.Context
import android.util.Log
import androidx.work.Worker
import androidx.work.WorkerParameters
import java.util.concurrent.TimeUnit.SECONDS

class TareaWorkerOne(
  context: Context,
  workerParams: WorkerParameters
) : Worker(context, workerParams) {
  override fun doWork(): Result {
    //TAREA QUE CUENTA DEL 1 AL 10 ESPERANDO UN SEGUNDO
    for (i in 1..10) {
      SECONDS.sleep(1)
      Log.d("TareaWorkerOne", "progreso: $i")
    }
    return Result.success()
  }
}
```

Editamos nuestra actividad principal para asociar el nuevo Worker, su inicialización y observación, para lo cual crearemos un nuevo método llamado startAndObserveJob() que debe ser ejecutado al presionar el botón de la vista.

```
package cl.desafiolatam.workmanager

import android.arch.lifecycle.Observer
import android.support.v7.app.AppCompatActivity
import android.os.Bundle
import android.util.Log
import android.widget.Toast
import androidx.work.*
import kotlinx.android.synthetic.main.activity_main.*
import androidx.work.WorkManager

class MainActivity : AppCompatActivity() {

   override fun onCreate(savedInstanceState: Bundle?) {
      super.onCreate(savedInstanceState)
      setContentView(R.layout.activity_main)
      button_start.setOnClickListener { startAndObserveJob() }
   }

   private fun startAndObserveJob() {
      // CONFIGURACION DEL WORKREQUEST
      val constraints = Constraints.Builder()
         .setRequiredNetworkType(NetworkType.CONNECTED)
         .build()
```

```
        val inputData = Data.Builder()
            .putString("example_key", "example_value")
            .build()
        val myWork = OneTimeWorkRequest.Builder(TareaWorkerOne::class.java)
            .setConstraints(constraints)
            .setInputData(inputData)
            .build()

        val workId = myWork.id
        WorkManager.getInstance().apply {
            // enqueue Work
            enqueue(myWork)
            // observe work status
            getWorkInfoByIdLiveData(workId)
                .observe(this@MainActivity, Observer { status ->
                    val isFinished = status?.state?.isFinished
                    if(isFinished == true) {
                        Toast.makeText(
                            applicationContext,
                            "TAREA/JOB: $workId ESTADO COMPLETADO: $isFinished",
                            Toast.LENGTH_LONG
                        ).show()
                    }
                        Log.d(TAG, "TAREA/JOB $workId; ¿Ha terminado?: $isFinished")
                })
        }

        Toast.makeText(this, "TAREA/JOB: $workId ENQUEUED", Toast.LENGTH_LONG).show()
    }
}

private const val TAG = "MainActivity"
```

Al ejecutar nuestra aplicación observamos que al presionar el botón “Ejecutar Tarea” se nos muestra el id del trabajo por pantalla y su estado iniciado o concluido. Por consola visualizamos la cuenta del 1 al 10 realizada por el trabajo en segundo plano.

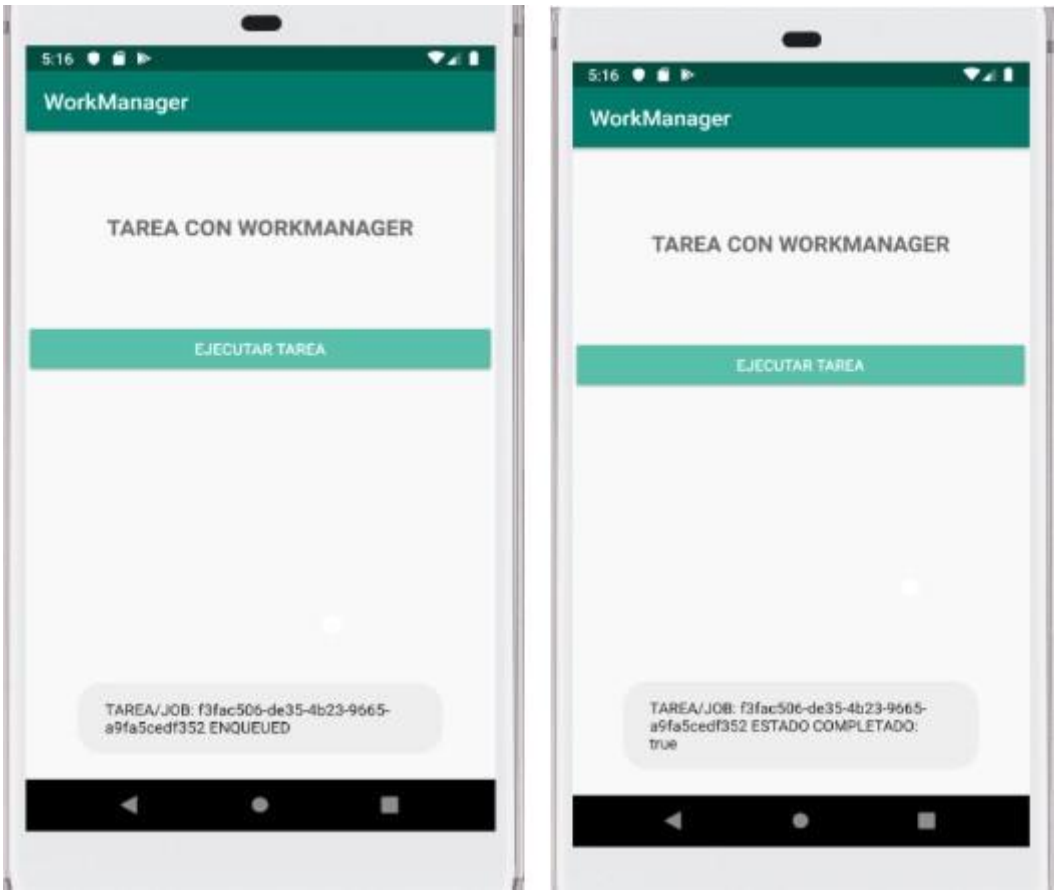


Imagen 19. Ejecutar tarea

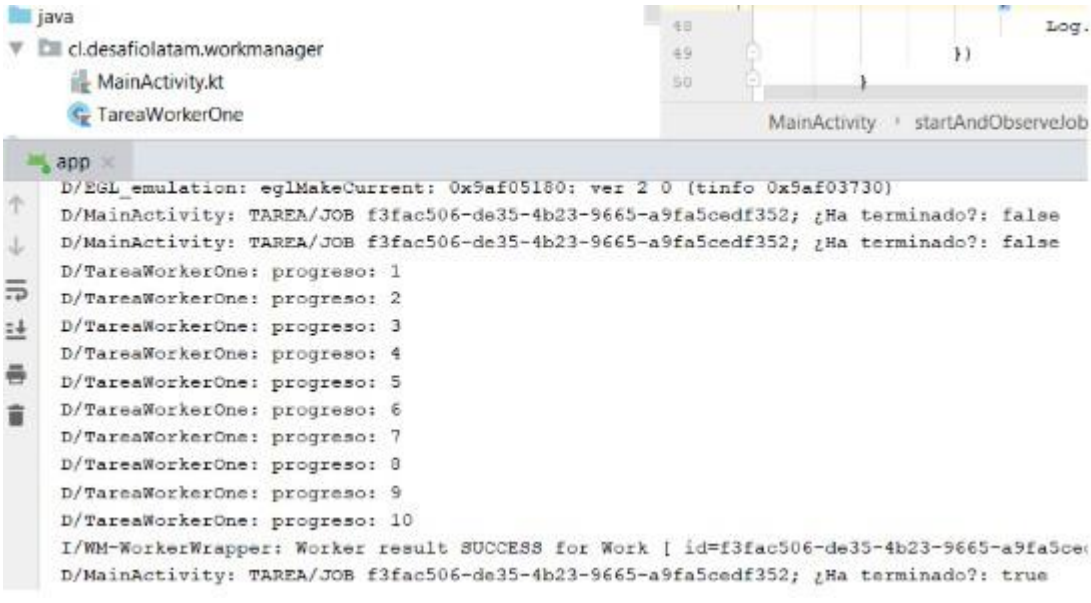


Imagen 20. Visualización en consola.

2.2 Coroutines

AsyncTask

Competencias:

- Conocer la clase AsyncTask de android
- Implementar tareas con la clase AsyncTask de android en un proyecto

Introducción

En este capítulo estudiaremos la clase AsyncTask de android para el desarrollo de tareas asíncronas que permiten el envío de una petición, continuando con el flujo normal del sistema, para posteriormente recibir la respuesta a dicha petición y ser utilizada, por ejemplo, en forma de datos dentro de nuestra aplicación.

La clase AsyncTask es ampliamente utilizada en muchas aplicaciones android desarrolladas en la actualidad, además de ser una clase con un continuo soporte y desarrollo por parte del equipo de android.

AsyncTask realiza operaciones en el subproceso en segundo plano (background thread) y se actualizará en el subproceso principal (thread UI). Asynctask nos ayuda a establecer comunicación entre hilos secundarios y principales (subThreads y mainThread/threadUI).

La clase AsyncTask

AsyncTask permite el uso correcto y fácil del main thread UI de android. Esta clase nos ayuda a realizar operaciones en segundo plano y publicar resultados en el subproceso de la UI (interfaz de usuario) sin tener que manipular subprocesos, controladores o servicios.

AsyncTask está diseñado para ser una clase de ayuda en todo Thread y Handler, y no constituye un framework genérico para threads. Las AsyncTasks deben usarse idealmente para operaciones cortas, de unos pocos segundos.

Si se desea mantener los subprocesos en ejecución durante largos períodos de tiempo, es recomendable utilizar las diversas herramientas del API proporcionadas por android, como lo son los servicios, la clase Executor, ThreadPoolExecutor y, entre otras.

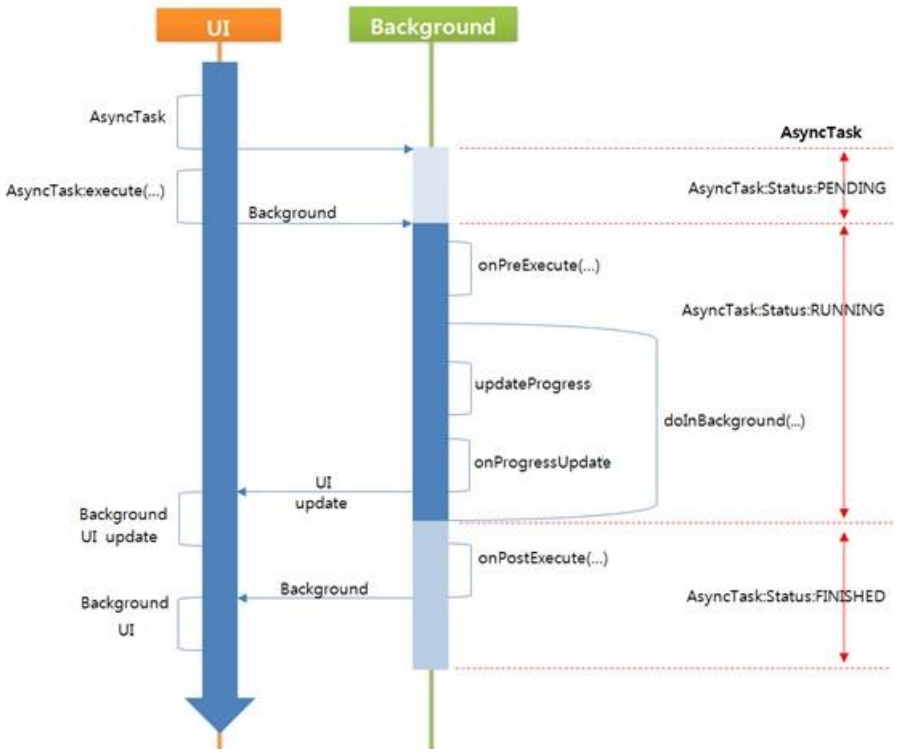


Imagen 1. Diagrama proceso AsyncTask UI Thread y Background Thread.

Una AsyncTask se define mediante tareas que se ejecuta en un subproceso en segundo plano, y cuyo resultado se publica en el subproceso de la interfaz de usuario. Un AsyncTask define tres métodos genéricos:

- doInBackground(), para todo aquel código que se ejecutará en segundo plano.
- onProgressUpdate(), que recibe toda actualización del progreso del método doInBackground.
- onPostExecute(), que lo utilizamos para actualizar la interfaz de usuario (UI) una vez el proceso en segundo plano se haya completado.

Una clase auxiliar AsyncTask se vería de la siguiente forma:

```
private class DownloadFilesTask extends AsyncTask<URL, Integer, Long> {
protected Long doInBackground(URL... urls) {
// code that will run in the background
return ;
}

protected void onProgressUpdate(Integer... progress) {
// receive progress updates from doInBackground
}

protected void onPostExecute(Long result) {
// update the UI after background processes completes
}
}
```

Iniciar un AsyncTask

Una clase AsyncTask se ejecutaría desde el mainThread a través de una actividad o fragmento de la siguiente forma:

```
new DownloadFilesTask().execute(url1, url2, url3);
```

Cancelar un AsyncTask

Una tarea puede cancelarse en cualquier momento invocando el método cancel(). Invocar este método hará que las siguientes llamadas (calls) invoquen al método isCancelled(), y este retornará el valor booleano true.

Una vez invocado el método cancel() la aplicación en vez de ejecutar el método onPostExecute() de AsyncTask ejecutará este método isCancelled(). Para asegurarnos de cancelar la tarea lo más rápido posible es oportuno validar el retorno de este método isCancelled() durante la ejecución del método doInBackGround().

Indicaciones para el uso correcto de AsyncTask

- La clase AsyncTask debe cargarse en el hilo de la interfaz de usuario (UI Thread). Esto se hace automáticamente a partir de la versión android JELLY_BEAN.
- La instancia de la tarea debe crearse en el subproceso de la interfaz de usuario (UI Thread).
- El método execute, debe invocarse en el hilo de la interfaz de usuario (UI Thread).
- No se debe invocar los métodos de la clase AsyncTask por separado o manualmente doInBackGround, PostExecute, etc.).
- La tarea se puede ejecutar solo una vez. Si se intentará ejecutar por una segunda vez una excepción debe ser enviada.

El método onPreExecute de AsyncTask

Este método se puede invocar en el subproceso de la interfaz de usuario antes de ejecutar la tarea (UI Thread). Este paso se usa normalmente para configurar la tarea, por ejemplo, iniciando una barra de progreso en la interfaz de usuario.

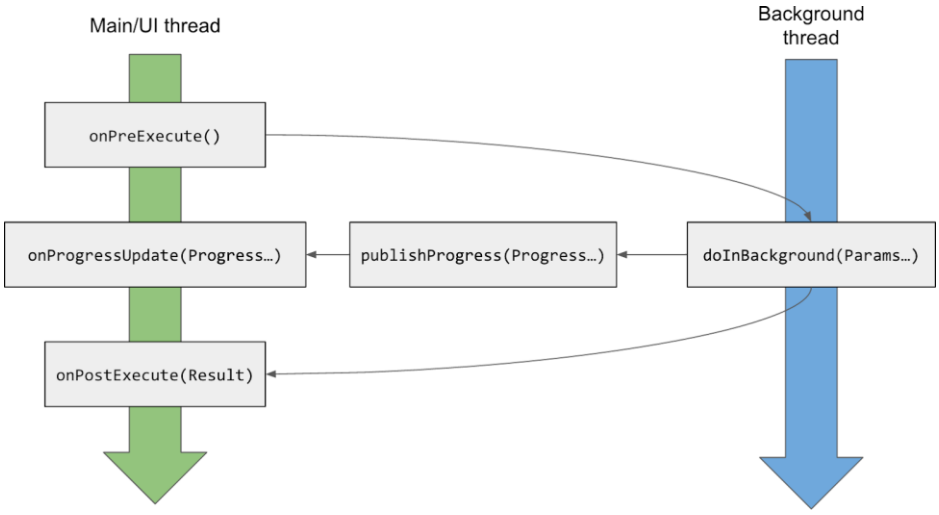


Imagen 2. Diagrama simple AsyncTask Threads.

Ejercicio 5:

Crear un proyecto llamado “AsyncTask” donde se descargue al momento de iniciarse la aplicación, la imagen del día de la nasa en formato HD y sea instanciada la imagen en una vista tipo ImageView. También se requiere el uso de una progressbar que se muestre en el lugar que ocuparála imagen una vez se haya descargado.

1. Editamos nuestro archivo de diseño “activity_main.xml” para incluir tres vistas, un textview, un progressbar y un imageview de la siguiente forma:

```
<TextView
    android:id="@+id/title"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="AsyncTask Download Image Example"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    android:layout_marginTop="100dp"
    android:textSize="20dp"
    android:textAlignment="center"/>

<ProgressBar android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/progressBar"
        style="?android:attr/progressBarStyleLarge"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        android:visibility="gone"
/>

<ImageView android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:id="@+id/imgOfDay"
       app:layout_constraintLeft_toLeftOf="parent"
       app:layout_constraintRight_toRightOf="parent"
       app:layout_constraintBottom_toBottomOf="parent"
       app:layout_constraintTop_toTopOf="parent"
       android:contentDescription="Nasa image of day"
/>
```

2. En nuestro método onCreate() de la “MainActivity”, iniciamos nuestros objetos de vista de la siguiente manera:

```
val imgOfDay = findViewById<ImageView>(R.id.imgOfDay)
val progressBar = findViewById<ProgressBar>(R.id.progressBar)
```

3. Dentro de nuestra actividad, crearemos una subclase privada llamada DownloadTask, que reciba como parámetros en su constructor las variables imgOfDat y progressBar, además de recibir la url de la imagen del día dentro de sus parámetros propios AsyncTask, y declarando el tipo de retorno como un bitmap para ser manipulado en el método postExecute(). Por otra parte, en nuestro método doInBackground debemos hacer visible la progressBar y programar la descarga de la imagen haciendo uso de java.net.URL y BitmapFactory, como hemos visto en ejemplos anteriores. El código es de la siguiente manera:

```
private class DownloadImageTask(val imgOfDay: ImageView, val progressBar: ProgressBar) :
AsyncTask<String, Void, Bitmap>() {

  override fun doInBackground(vararg urls: String): Bitmap? {
    val urldisplay = urls[0]
    var bmp: Bitmap? = null
    try {
      progressBar.visibility = View.VISIBLE

      val inputStream = java.net.URL(urldisplay).openStream()
      bmp = BitmapFactory.decodeStream(inputStream)
    } catch (e: Exception) {
      Log.e("Error", e.message)
      e.printStackTrace()
    }

    return bmp
  }

  override fun onPostExecute(result: Bitmap?) {
    try {
      progressBar.visibility = View.GONE
      imgOfDay.setImageBitmap(result)
    }catch (e: java.lang.Exception){e.printStackTrace()}
  }
```

4. Por último, al final de nuestro método onCreate() debemos declarar el llamado a esta nueva clase DownloadTask de la siguiente manera:

```
DownloadImageTask(imgOfDay, progressBar).execute("https://apod.nasa.gov/apod/image/1908/M61-HST-ESO-S1024.jpg")
```

5. Para finalizar no se nos escape agregar los permisos necesarios a nuestro archivo manifest:

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

6. Al ejecutar la aplicación observaremos el inicio del trabajo y su fin, como lo muestran las siguientes imágenes:

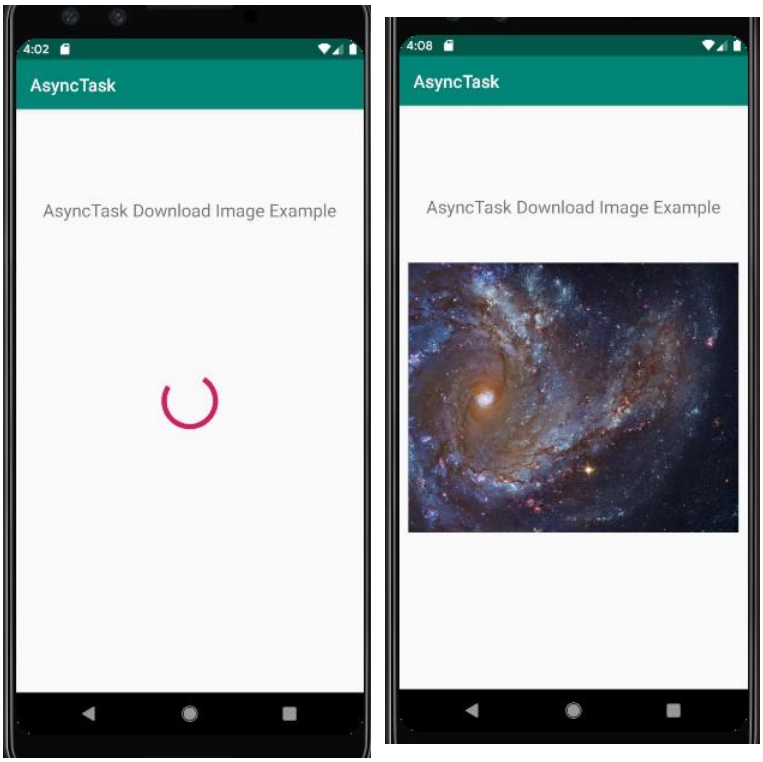


Imagen 3. AsyncTask Download.

Promises y Coroutines

Competencias:

- Conocer el patrón moderno de desarrollo de promesas para tareas asíncronas.
- Aprender sobre las coroutines y sus conceptos modernos en android.
- Entender la asociación entre promesas y coroutines
- Implementar promesas y coroutines en un proyecto android.

Introducción

En este capítulo estudiaremos el concepto moderno de las promesas y coroutines, su evolución y utilización para la ejecución de tareas y procesos asíncronos.

Tanto las promesas como las coroutines se pueden interpretar como una forma de escribir código asíncrono de manera moderna y eficiente, ambas surgen de la evolución de antiguos procesos y formas de realizar procesos asíncronos.

Es importante aprender a trabajar con promesas y coroutines en android para en primer lugar ahorrarnos grandes dolores de cabeza por errores de agrupación de llamados ya superados, y también para escribir un código más limpio, eficiente y fácil de leer.

Promises

Pensemos en una promesa como un objeto que espera a que finalice una acción asincrónica, luego llama a una segunda función. Podemos programar esa segunda función llamando al método .then() y pasando la nueva función. Cuando finaliza esta última función asincrónica, da su resultado a la promesa y la promesa lo da a la siguiente función (como parámetro).

Cada llamada al método .then() espera la promesa anterior y ejecuta la siguiente función, luego convierte el resultado en una promesa si es necesario. Esto nos permite encadenar sin problemas llamadas síncronas y asíncronas. Simplifica tanto el código, que la mayoría de las nuevas especificaciones devuelven promesas de sus métodos asincrónicos.

Las promesas ofrecen una mejor manera de trabajar con el código y llamados asíncronos, estando presentes en el mercado por un tiempo considerable a través de distintas librerías.

Las promesas en principio surgen como iniciativa para reemplazar los callbacks, motivado a sus fallas o agrupaciones interminables (callback hell) que no ayudaban en mucho a la lectura del código del sistema, y a su vez creaban fallas difíciles de seguir o debuggear. Las promesas han ido evolucionando con el tiempo dando origen a las nuevas versiones de async/await de ES2017 (EcmaScript).

CallBack vs Promises

El siguiente es un ejemplo de un método que valida si un usuario es muy joven según su edad utilizando callbacks:

```
fun isUserTooYoung(id, callback) {
  openDatabase(fun(db) {
  getCollection(db, 'users', fun(col) {
      find(col, {'id': id}, fun(result) {
        result.filter(fun(user) {
        callback(user.age < cutoffAge);
        });
      });
    });
  });
}
```

De este código podemos apreciar en primera instancia la cantidad de llaves de cierre al final, lo que fácilmente terminaría en un error de compilación por la falta de alguna de estas, sin imaginar la tortura que sería agregar una condición más a esta función.

Manteniendo la misma validación anterior de si un usuario es muy jóven para acceder, veamos cómo quedaría el código utilizando las promesas:

```
fun isUserTooYoung(id): Db {
  return openDatabase() // returns a promise
  .then(fun(db: Database) {return getCollection(db, 'users');})
  .then(fun(col: Collection) {return find(col, {'id': id});})
  .then(fun(user: User) {return user.age < cutoffAge;});
}
```

A primeras, se entiende que es un código más legible y tiene tres acciones subsecuentes.

Promises.All

A menudo queremos ejecutar acciones sólo después de que una colección de operaciones asincrónicas se haya completado con éxito. El objeto Promise.all devuelve una promesa que se resuelve si todas las promesas pasadas se resuelven. Si alguna de las promesas aprobadas rechaza, Promise.all rechaza con la razón de la primera promesa que rechazó. Esto es muy útil para garantizar que se complete un grupo de acciones asincrónicas antes de continuar con otro paso.

En el ejemplo a continuación, promise1 y promise2 devuelven promesas. Queremos que ambas se carguen antes de continuar. Ambas promesas pasan como parámetros al objeto Promise.all. Si cualquiera de las solicitudes es rechazada, Promise.all rechaza toda la acción con el resultado de la promesa rechazada. Si se cumplen ambas solicitudes, Promise.all resuelve con los valores de ambas promesas (como una lista).

```
var promise1 = getJSON('/users.json');
var promise2 = getJSON('/articles.json');

Promise.all([promise1, promise2]) // Array of promises to complete
.then(fun(results) {
  println('all data has loaded');
})
.catch(fun(error) {
  println('one or more requests have failed: ' + error);
});
```

Coroutines

Una corrutina es un patrón de diseño de concurrencia que puede usarse en Android para simplificar el código que se ejecuta de forma asincrónica. Se agregaron corutinas a Kotlin en la versión 1.3 y se basan en conceptos establecidos de otros idiomas.

Las corrutinas podrían interpretarse como la última versión, la más avanzada y moderna de los callbacks y promesas para android, ya que en realidad las corrutinas, son en mucho el patrón async/await de javascript que lidera este avance en operaciones asíncronas.

En Android, las rutinas ayudan a resolver dos problemas principales:

- Administrar tareas de larga duración que de otro modo podrían bloquear el hilo principal y hacer que su aplicación se congele.
- Proporcionar seguridad principal, o llamadas con seguridad a las operaciones de red o disco desde el hilo principal.

Coroutines en tareas extensas

En Android, cada aplicación tiene un hilo principal que maneja la interfaz de usuario y gestiona las interacciones del usuario. Si una aplicación está asignando demasiado trabajo al hilo principal, la aplicación puede congelarse o detenerse. Las solicitudes de datos a través de grandes apis, el análisis de objetos JSON extensos, la lectura o la escritura desde una base de datos, o incluso la iteración de listas grandes, pueden hacer que la aplicación se ejecute lo suficientemente lento como para provocar que nuestra pantalla responda lentamente a eventos táctiles. Estas operaciones de larga duración deben ejecutarse fuera del hilo principal.

El siguiente ejemplo muestra la implementación de rutinas simples para una tarea hipotética de larga duración:

```
suspend fun fetchDocs() {                          // Dispatchers.Main
    val result = get("https://developer.android.com") // Dispatchers.IO for `get`
    show(result)                                   // Dispatchers.Main
}

suspend fun get(url: String) = withContext(Dispatchers.IO) { /* ... */ }
```

Las corrutinas se basan en funciones regulares agregando dos operaciones para manejar tareas de larga duración. Además de invoke(o call) y return, las corrutinas agregan suspend y resume:

- **suspend**: hace pausa en la ejecución de la rutina actual, guardando todas las variables locales.
- **resume:** continúa la ejecución de una corrutina suspendida desde el lugar donde fue suspendida.

En el ejemplo anterior, el método get() todavía se ejecuta en el hilo principal, pero suspende la rutina antes de que comience la solicitud de red.

Cuando finaliza la solicitud de red, get() reanuda la rutina suspendida en lugar de utilizar una devolución de llamada para notificar al hilo principal.

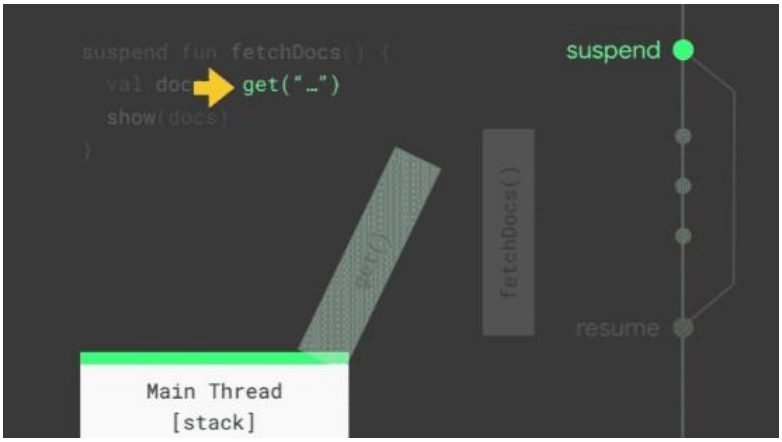


Imagen 4. Funcionamiento suspend resume en coroutines.

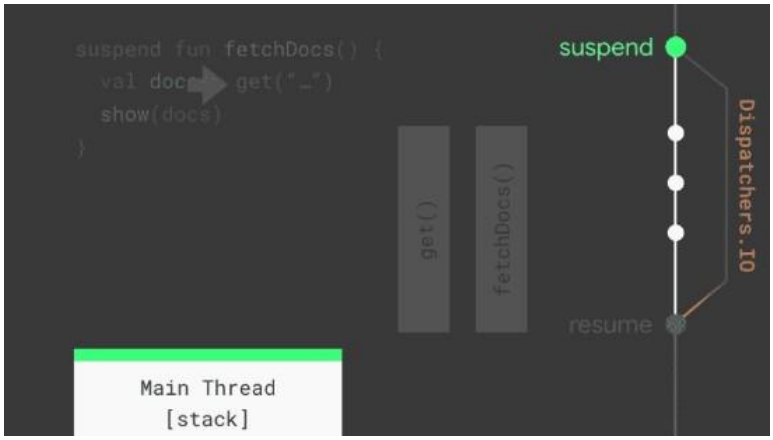


Imagen 5. Funcionamiento Dispatcher.IO resume en coroutines.

Imagen 6. Funcionamiento resume en coroutines.

Seguridad y Coroutines

Las corrutinas de Kotlin usan despachadores para determinar qué hilos se usan para la ejecución de la rutina. Para ejecutar el código fuera del hilo principal, puede indicarle a las corrutinas de Kotlin que realicen el trabajo en un despachador predeterminado. En Kotlin, todas las corrutinas deben ejecutarse en un despachador, incluso cuando se ejecutan en el hilo principal. Las corrutinas pueden suspenderse, y el despachador es responsable de reanudarlas.

Para especificar dónde deben ejecutarse las corrutinas, Kotlin proporciona tres despachadores que puede usar:

- **Dispatchers.Main**: este despachador se usa para ejecutar una rutina en el hilo principal de Android. Este debe usarse solo para interactuar con la interfaz de usuario y realizar un trabajo rápido.
- **Dispatchers.IO**: este despachador está optimizado para realizar operaciones de disco o de red fuera del hilo principal.
- **Dispatchers.Default**: este despachador está optimizado para realizar un trabajo intensivo de CPU fuera del hilo principal. Un ejemplo podría ser ordenar una lista y analizar un objeto JSON.

```
suspend fun fetchDocs() {                    // Dispatchers.Main
    val result = get("developer.android.com") // Dispatchers.Main
    show(result)                             // Dispatchers.Main
}

suspend fun get(url: String) =               // Dispatchers.Main
    withContext(Dispatchers.IO) {            // Dispatchers.IO (main-safety block)
        /* perform network IO here */        // Dispatchers.IO (main-safety block)
    }                                        // Dispatchers.Main
}
```

Iniciando Coroutines

Podemos comenzar las corrutinas de una de dos maneras:

- **launch**: comienza una nueva corrutina y no devuelve el resultado al objeto que llama. Cualquier trabajo que se considere iniciarse y no esperar por su respuesta es ideal indicar la palabra launch.
- **async**: inicia una nueva rutina y le permite devolver un resultado con una función de suspensión llamada await.

Por lo general, ejecutamos launch para iniciar una nueva corrutina de una función regular, ya que una función regular no puede llamar await. Para usar async debemos hacerlo sólo cuando se esté dentro de otra corrutina o cuando se esté dentro de una función de suspensión.

```
fun onDocsNeeded() {
    viewModelScope.launch {  // Dispatchers.Main
        fetchDocs()          // Dispatchers.Main (suspend function call)
    }
}
```

Uso de Coroutines en Android

Para utilizar las corrutinas en nuestros proyectos android es necesario implementar en el gradle de la app las librerías que nos permiten tener acceso a estas. Por ejemplo el siguiente código de versión 1.0.1:

```
implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-android:1.0.1'
```

Ejercicio 6:

Crear un proyecto llamado “Coroutines” que realice la descarga de la imagen del día de la nasa y la muestre en un imageview, utilizando java.net.URL. La actividad debe implementar CoroutineScope y tener un método que aplique launch{}, que a su vez llame otro método que ejecuta la descarga utilizando una referencia a Dispatchers.IO.

1. Agregamos la librería de coroutines en nuestro archivo gradle de la carpeta app.

```
implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-android:1.0.1'
```

2. Copiamos las vistas creadas en el ejercicio del capítulo “AsyncTask” en nuestro archivo layout activity_main.xml de la siguiente manera:

```
<TextView
    android:id="@+id/title"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Coroutine Download Image Example"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    android:layout_marginTop="100dp"
    android:textSize="20dp"
    android:textAlignment="center"/>

<ProgressBar android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/progressBar"
        style="?android:attr/progressBarStyleLarge"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        android:visibility="gone"
/>

<ImageView android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:id="@+id/imgOfDay"
       app:layout_constraintLeft_toLeftOf="parent"
       app:layout_constraintRight_toRightOf="parent"
       app:layout_constraintBottom_toBottomOf="parent"
       app:layout_constraintTop_toTopOf="parent"
       android:contentDescription="Nasa image of day"
/>
```

3. Extendemos nuestra clase actividad de la clase CoroutineScope, reescribiendo la variable coroutineContext de la siguiente manera:

```
class MainActivity : AppCompatActivity(), CoroutineScope {

  private val job = Job()
  override val coroutineContext = Dispatchers.Main + job
```

4. Creamos una variable de clase final para el string de nuestra imagen:

```
private val urlPictureOfTheDay: String = "https://apod.nasa.gov/apod/image/1908/M61-HST-ESO-S1024.jpg"
```

5. Creamos un método llamado “downloadImageBlocking” que ejecute la acción de descargar la imagen desde el api de la nasa y retorne un bitmap:

```
private fun downloadImageBlocking(): Bitmap? {
  val bmp: Bitmap
  try {
    val inputStream = URL(urlPictureOfTheDay).openStream()
    bmp = BitmapFactory.decodeStream(inputStream)
    return bmp

  }catch (e: Exception){
    e.printStackTrace()
  }
  return null
}
```

6. A continuación crearemos un método llamado “downloadImage” en donde aplicaremos el launch de la corrutina, habilitamos y deshabilitamos la progressBar, llamamos un subproceso Dispatchers.IO invocando el método suspend downloadImageBlocking(), para finalmente asignar el bitmap a nuestra vista imageview.

```
private fun downloadImage(){
  launch {
    progressBar.visibility = View.VISIBLE
    val bitmap: Bitmap? = withContext(Dispatchers.IO) { downloadImageBlocking() }
    if(bitmap != null) {
      imgOfDay.setImageBitmap(bitmap)
    }
    progressBar.visibility = View.GONE
  }
}
```

7. Finalmente sobreescribimos nuestro método de ciclo de vida onStart() para iniciar la corrutina:

```
override fun onStart() {
   super.onStart()
   downloadImage()
}
```

8. No olvidemos los permisos en el manifest:

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

9. Nuestra aplicación al ejecutarla tendrá el resultado del ejercicio del capítulo AsyncTask, pero en esta ocasión el código ha sido reescrito utilizando coroutines para entender su funcionamiento y diferencia.

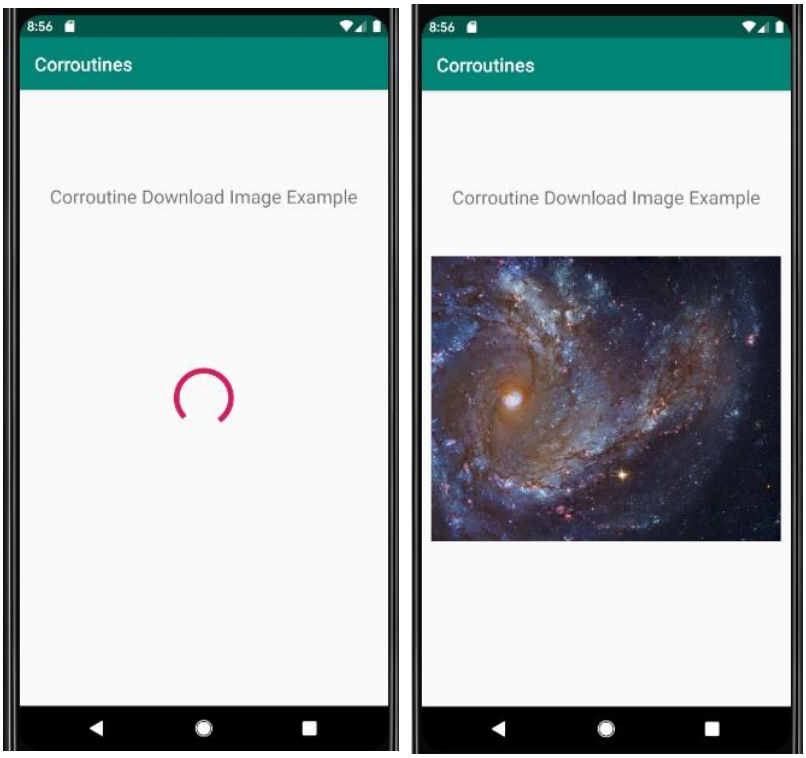


Imagen 7. Resultado del ejercicio.

2.3 Persistencia de datos en Android

Persistencia de datos en Android

Competencias:

- Entender las diferentes opciones para almacenar datos en Android.
- Decidir cuál es la mejor opción para otorgar persistencia de datos en las Aplicaciones.
- Saber qué opciones ofrece Android para compartir información entre aplicaciones.

Introducción

El sistema operativo Android nos ofrece diferentes opciones para almacenar los datos que utiliza nuestra aplicación. Las opciones disponibles se utilizan en diferentes casos de uso, normalmente las opciones por defecto de una aplicación es más que suficiente para la mayoría de los casos.Las opciones por defecto de Android para almacenar información o datos en una aplicación son SharedPreferences o el Sandbox de la aplicación.

Qué opción de almacenamiento tomamos depende directamente de las necesidades de nuestra aplicación, sean estas necesidades de seguridad, espacio, disponibilidad, etc. Aprenderemos sobre las diferentes opciones que nos ofrece Android, y de qué manera podemos elegir dichas opciones.

Persistencia de datos en Android

En Android, una aplicación cuenta con un ecosistema que le permite almacenar datos que son utilizados por nuestra aplicación. Cada aplicación cuenta con un Sandbox al cual sólo tiene acceso la aplicación, y el sistema operativo en modo root. Este Sandbox es el contenedor de ejecución seguro de cada aplicación, donde se almacena el código fuente, los assets, archivos de configuración, imágenes, etc. No confundir contenedor de ejecución de la aplicación con ART, que es Android Run Time. ART es donde se ejecuta el código en Android, pero el contenedor de cada aplicación es el Sandbox. Nadie tiene acceso a este contenedor o sandbox, el cual se ejecuta con el mínimo de privilegios necesarios en el dispositivo. Este concepto de Sandbox es gracias a linux, que crea un id para cada proceso que corre en el sistema operativo, con el mínimo de privilegios y sin intervención externa.

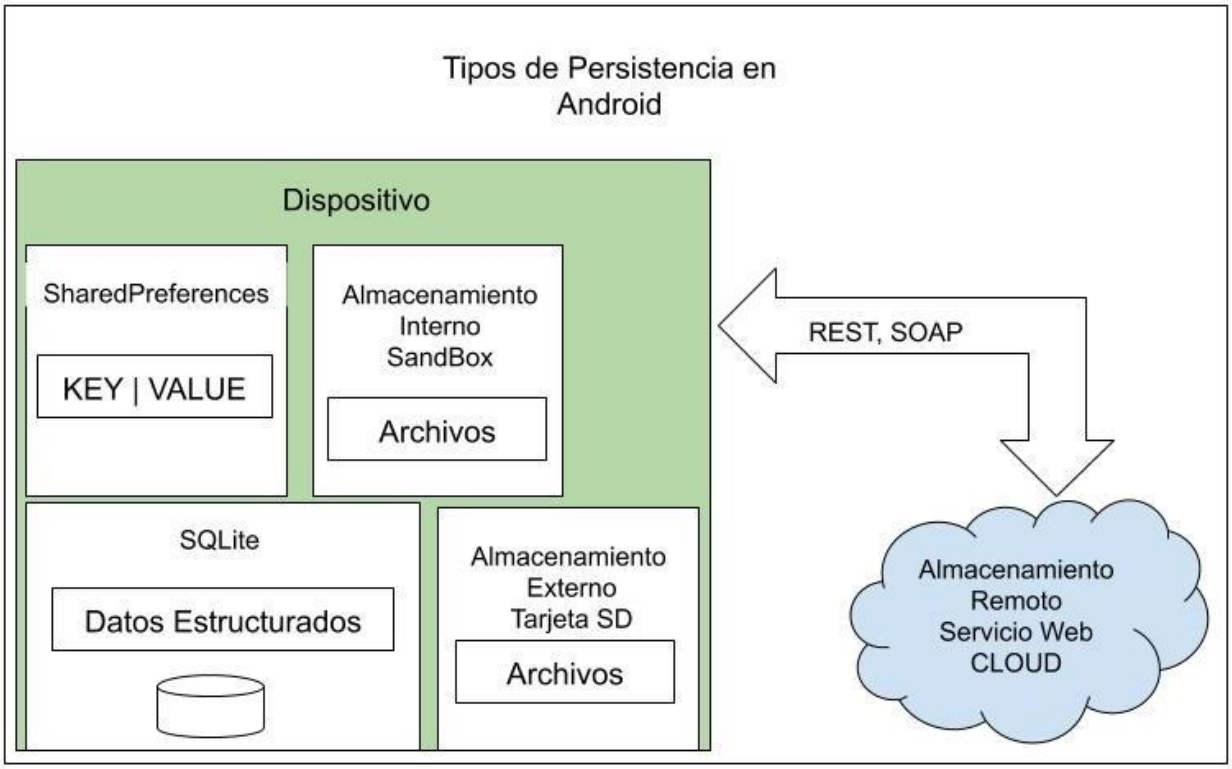


Imagen 1. Diagrama con los tipos de persistencia en Android.

Las diferentes opciones de persistencia de datos son las siguientes:

1. **SharedPreferences**: Nos permite almacenar datos primitivos en formato clave-valor. Este se profundizará más adelante.
2. **Almacenamiento Interno**: Nos permite almacenar información en la memoria interna del dispositivo. La información se puede almacenar en archivos dentro del Sandbox de la aplicación. Sólo la aplicación tiene acceso a esos archivos, a menos que el dispositivo se encuentre rooteado.
3. **Almacenamiento externo**: Los dispositivos Android permiten conectar memoria externa, normalmente en formato de memoria SD. Esta memoria es extraíble o se puede montar en un computador, por lo cual no se asegura su disponibilidad, y se recomienda almacenar información que no sea crítica para el funcionamiento de la aplicación. Se pueden almacenar archivos, los cuales son de acceso público para todos los usuarios. Cualquiera de estos usuarios puede eliminar o modificar dichos archivos. Este tipo de almacenamiento no tiene seguridad.
4. **Base de Datos SQLite**: Almacenamiento de datos, estructurado, privado. Esta implementación de SQL está optimizada para dispositivos móviles, es la opción por defecto de base de datos en Android. Desde los inicios se ha podido crear bases de dato en SQLite. La librería Room de Googlenos permite interactuar con SQLite de manera más simple, con menos errores y menos boilerplate code, facilitando el uso y el desarrollo. Es decir, Room crea una capa de abstracción que nos permite interactuar directamente con la base de datos en SQLite, pero con la familiaridad de trabajarcon objetos en vez de tablas y queries SQL. Room será visto en profundidad más adelante.
5. **Almacenamiento remoto**: Podemos almacenar información de modo remoto, ya sea a través de un servicio web, normalmente REST, o a través de un servicio tipo Cloud.

Datos y Seguridad

¿Cómo podemos decidir qué tipo de almacenamiento nos conviene utilizar?. Para decidir esto debemos tener claro:

- ¿Qué tipo de información o datos estamos almacenando?.
- ¿Los datos son públicos o privados?
- ¿Qué disponibilidad debe tener esa información o datos?
- ¿Quién tiene acceso a esa información o datos ?

Estas interrogantes son claves para decidir qué tipo de almacenamiento utilizaremos. Debemos considerar no sólo el tamaño de los datos, también es necesario considerar la privacidad y disponibilidadde los mismos. Veamos algunos casos de ejemplo.

Almacenamiento de fotografías de la Cámara

Por ejemplo la aplicación de la cámara necesita almacenar archivos de imagen de gran tamaño, en este caso podemos utilizar el almacenamiento interno o externo. Las imágenes que se toman con la cámara del dispositivo, con la aplicación por defecto de la cámara, son de acceso público y se encuentran disponibles en la galería del dispositivo, o los archivos multimedia del mismo. Estos mismos archivos son utilizados por aplicaciones para ser publicadas en internet, estas aplicaciones deben pedir permiso para acceder a las imágenes, pero nada impide que esos mismos archivos sean extraídos en un computador personal, incluso eliminados de la memoria del dispositivo, sea interna o externa.

Preferencias de una aplicación

Consideremos el caso donde nuestra aplicación permite a un usuario configurar a su gusto algunas cosas, por ejemplo el tamaño de la letra o los colores de ciertos componentes. Como esta configuración es local, no representa problemas de privacidad, y no tiene mucha complejidad, podemos utilizar SharedPreferences para almacenar un conjunto de clave-valor y mantener los datos de la configuración entre instancias de nuestra aplicación.Si estos datos se llegan a perder, nuestra aplicación no sufre ninguna consecuencia severa, salvo pequeños inconvenientes para el usuario.

Aplicación Bancaria

Imaginemos una aplicación bancaria, en la cual tenemos acceso a diferentes servicios, por ejemplo revisar nuestra cartola mensual en formato pdf. Esta cartola contiene información sensible para el usuario, y debe ser almacenada con la mayor seguridad posible, por lo que en este caso es necesario utilizar el almacenamiento interno, en el sandbox de la aplicación, para que nadie, salvo el usuario en nuestra aplicación, pueda ver el archivo descargado. Medidas adicionales son bienvenidas en casos de extrema sensibilidad de información, algunos son: claves de acceso, encriptación o cifrado de losarchivos.

Aplicación con modo Online/Offline

Lo más común actualmente es contar con aplicaciones que utilizan Internet para obtener información, pero muchas veces se ven afectadas por la intermitencia de las conexiones de los teléfonos celulares. Enestos casos podemos desarrollar una versión Offline, normalmente con Caché o data almacenada localmente, que podemos mostrar al usuario mientras no tenemos conexión. Este caché, normalmente, se almacena en una base de datos local, y se va renovando con la última información, a medida que vamos cargando desde el servicio Online. Hay aplicaciones que para mejorar la experiencia del usuario, almacenan grandes cantidades de caché, lo cual termina resintiendo el desempeño del dispositivo por falta de espacio. Tener cuidado al implementar este tipo de almacenamiento en una aplicación.

Compartiendo datos en Android

Como ya sabemos, los datos o información, en Android, sólo están disponibles para la aplicación. En el caso que necesitemos compartir información entre aplicaciones contamos con las siguientes opciones:

1. **Compartir a través de SharedPreferences:** Podemos compartir nuestro archivo de SharedPreferences con otras aplicaciones a través del modo global.
2. **ContentProvider**: los proveedores de contenido nos permiten acceder a datos de otra aplicación. Estos datos son compartidos para que sean usados por otras aplicaciones, incluso pueden ser modificados si el ContentProvider lo permite. Se pueden compartir imágenes, archivos, o datos de una base de datos. Si queremos compartir información desde una aplicación A a una aplicación B, la aplicación A debe implementar el ContentProvider. La aplicación B debe utilizar el ContentProvider implementado por A y pedir los permisos necesarios. Por ejemplo WhatsApp debe utilizar el ContentProvider de la aplicación de Contactos de el sistema, así obtiene los contactos necesarios para empezar a enviar mensajes o llamar por teléfono.

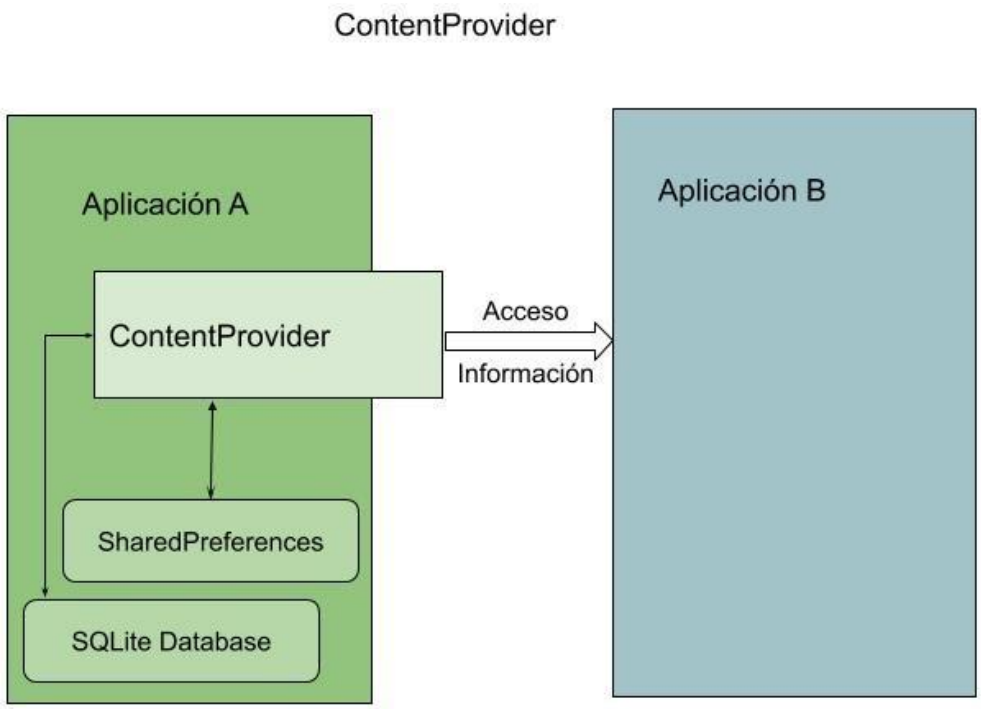


Imagen 2. Diagrama de un proveedor de contenidos que nos permite accedes a datos de otra aplicaci´ón.

SharedPreferences

Competencias:

- Añadir Sharedpreferences a una aplicación
- Utilizar buenas prácticas en el uso de SharedPreferences
- Compartir información entre aplicaciones a través de SharedPreferences

Motivación

A continuación nos adentraremos en una de las opciones más simples, y utilizadas, para almacenar datos en Android. Comprenderemos que tipo de almacenamiento tenemos disponible, que podemos almacenar, buenas y malas prácticas con respecto al uso de SharedPreferences. Aprenderemos a crear un Archivo de Preferencias, los diferentes modos en que podemos crear el mismo, y cómo hacer que ese archivo esté disponible para otras aplicaciones.

SharedPreferences, o archivo de preferencias

SharedPreferences es la forma más simple de almacenar información en Android. Nos permite almacenar pequeñas cantidades de información, en un formato de clave-valor, en un archivo en nuestro dispositivo. Este archivo se puede crear con diferentes configuraciones que limitan el acceso de terceros a la información almacenada en este archivo. El archivo es directamente manejado por Android, todos los componentes de la aplicación tienen acceso al archivo, pero no está disponible para otras aplicaciones. El archivo de sharedpreferences se inicia de manera recomendada en modo privado, indicando que sólo los componentes de la aplicación dueña del archivo tienen acceso al mismo. Existe un modo público para este archivo, pero no está recomendado, y se han deprecado sus opciones desde hace varias versiones de Android, más información sobre esto en la sección donde explicamos como crear un archivo de preferencias.

SharedPreferences está recomendado sólo para pequeñas cantidades de información, si la información que se debe almacenar es más compleja que unas cuantas primitivas, por ejemplo un objeto, se recomienda utilizar una base de datos SQLite.

Los datos almacenados en SharedPreferences son privados para la aplicación. Los datos persisten más allá de una sesión de la aplicación, los datos almacenados en el archivo se mantienen incluso si la aplicación es cerrada explícitamente. Se Utiliza normalmente para almacenar las preferencias del usuario, su nickname o su puntaje en un juego, nada muy complejo o delicado.

Utilizando SharedPreferences en una aplicación

Para utilizar SharedPreferences en una aplicación, necesitamos de alguna clase que extienda Context, para los ejemplos y explicaciones de este contenido asumimos la existencia de una aplicación ficticia que cuenta con una clase MainActivity, que nos permite obtener el Context. Más adelante construiremos una aplicación de ejemplo, paso por paso, donde utilizaremos todo lo que veremos a continuación, pero primero debemos tener claro cómo crear un archivo, como utilizarlo y las operaciones que nos permite. Además, debemos entender lo siguiente:

No debemos confundir SharedPreferences con Preference:

- **SharedPreferences** es un conjunto de APIs que nos permite almacenar datos primitivos (String, Int, Boolean ,etc), en formato clave-valor, estos datos persisten entre instancias de la aplicación.
- **Preference** es un conjunto de APIs que nos permiten crear una interfaz de usuario para una o varias pantallas de preferencias, por ejemplo la configuración de idioma o notificaciones push de la aplicación. Preference ocupa SharedPreferences para almacenar y persistir estos datos que son seleccionados en la pantalla de preferencias.

Creando un archivo de SharedPreferences en nuestra aplicación ficticia

Normalmente necesitamos sólo un archivo por aplicación, el nombre está asociado con el paquete de la misma. De esta forma es más fácil asociar el archivo con la aplicación. Esto no es una restricción y podemos utilizar cualquier nombre, pero está recomendado que utilicemos el nombre del paquete de la aplicación.

Para crear un archivo de preferencias debemos tener acceso a un Context, esto significa que debemos tener acceso a una Actividad o algún otro componente de Android que nos permita obtener el objeto Context. Por ejemplo en nuestra Actividad en una clase llamada MainActivity agregaremos una variable que mantendrá disponible nuestro archivo de preferencias. Para hacer esto debemos agregar el siguiente código a esta clase de ejemplo que contiene la actividad:

```
class MainActivity : AppCompatActivity() {
  lateinit var sharedPreferences: SharedPreferences
  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    val fileName = "cl.desafiolatam.persistenciaandroid"
    sharedPreferences = getSharedPreferences(fileName, Context.MODE_PRIVATE)
  }
}
```

Como vemos, nuestra MainActivity extiende AppCompatActivity que nos permite obtener un Context. Además, contiene una variable lateinit var llamada sharedPreferences, de tipo SharedPreferences. Esta variable tiene un modificador lateinit, que nos permite definir una propiedad de la clase, que no será iniciada inmediatamente en el constructor de la misma.

```
class MainActivity : AppCompatActivity() {
  lateinit var sharedPreferences: SharedPreferences
...
```

Todas las variables tipo var en kotlin deben ser inicializadas en el constructor de la clase, pero cuando esto no resulta conveniente, como en este caso que inicializamos la variable en el método onCreate, podemos utilizar el modificador lateinit para inicializar la variable más adelante en nuestro código. Si queremos acceder a esta variable antes de que sea inicializada lo más recomendable es hacer lo siguiente:

```
if (this::sharedPreferences.isInitialized) {
  //hacer la operación de forma segura
}
```

De esta forma accedemos de manera segura a nuestra variable, en este caso usamos la referencia al nivel donde está declarada la variable `this::` , luego ocupamos el nombre de la variable y la propiedad isInitialized para saber si ya fue inicializada.

Para crear el archivo hacemos uso del método Context.getSharedPreferences. Este método es parte de un objeto instanciado que extiende Context en Android. En este caso nuestra MainActivity es una instancia de un objeto que extiende Context, por lo que podemos hacer referencia directa al método:

```
val fileName = "cl.desafiolatam.persistenciaandroid"
//Modo recomendado, privado
sharedPreferences = getSharedPreferences(
fileName,
Context.MODE_PRIVATE)
//otros modossharedPreferences = getSharedPreferences(
fileName,
Context.MODE_WORLD_READABLE)sharedPreferences = getSharedPreferences(
           fileName,
Context.MODE_WORLD_WRITEABLE)sharedPreferences = getSharedPreferences(
           fileName,
Context.MODE_MULTI_PROCESS)
```

El método recibe dos parámetros, las opciones para estos parámetros son las siguientes:

- **name**: El nombre del archivo de preferencia. Si el archivo no existe, se creará cuando se utilice un editor, SharedPreferences.edit(), y se haga commit a los cambios, Editor.commit(). Es decir, el sistema creará nuestro archivo, y después lo mantendrá, junto con sus datos almacenados, hasta que la aplicación sea eliminada del dispositivo o el usuario borre explícitamente las preferencias de la aplicación, al eliminar los datos de la aplicación desde el manager de aplicaciones del dispositivo.
- **mode**: El modo en el cual operará el archivo, en este caso utilizamos el modo privado pero tenemos las siguientes opciones:
    - **MODE_PRIVATE**: El archivo puede sólo ser accedido por la aplicación, o todas las aplicaciones que comparten el mismo ID. Este es el modo por defecto, y recomendado para lacreación y operación de archivos de preferencias.
    - **MODE_WORLD_READABLE**: El archivo puede ser leído por todas las aplicaciones, esto modo no es recomendado y fue deprecado en la API 17, tiene problemas de seguridad.
    - **MODE_WORLD_WRITEABLE**: El archivo puede ser modificado por todas las aplicaciones, no recomendado, con problemas de seguridad y deprecado en la API 17.
    - **MODE_MULTI_PROCESS**: El archivo en el disco es revisado por posibles modificaciones, aunque ya se encuentre cargado en el proceso actual de la aplicación. Este modo es deseado cuando tenemos una aplicación con múltiples procesos, los cuales escriben en el mismo archivo simultáneamente. Este modo es ocupado principalmente cuando se necesita comunicar datos entre procesos, pero existen alternativas mucho mejores. Este modo fue deprecado en la API 23, debido a problemas de desempeño, consistencia entre procesos y la poca fiabilidad de funcionamiento en varias versiones de Android. Las aplicaciones deben utilizar métodos más fiables para comunicar información entre procesos, lo recomendado es utilizar un ContentProvider.

SharedPreferences.Editor

SharedPreferences es una Interfaz, que nos permite interactuar con el archivo de preferencias que creamos. Para poder almacenar datos en este archivo debemos utilizar un editor. Para esto debemos hacer lo siguiente:

```
sharedPreferences = getSharedPreferences(fileName, Context.MODE_PRIVATE)
sharedPreferences.edit().putString("TestString","Hola").commit()sharedPreferenc
es.edit().putString("TestString2","Hola 2").apply()
```

Al ocupar edit(), SharedPreferences nos retorna una interfaz SharedPreferences.Editor, esta interfaz nos permite ejecutar diferentes métodos para almacenar distintos datos. El método edit no se puede utilizar por sí solo, por lo cual debemos agregar alguno de los siguientes métodos, disponibles a través de la interfaz Editor:

- **apply**: Este método nos permite aplicar los cambios que hicimos sobre el archivo de preferencias que se encuentra en memoria. Hay que tener cuidado con este método ya que si bien ejecuta de forma atómica la operación de modificación, si hay dos editores modificando al mismo tiempo, el último que aplica apply termina sobreescribiendo los datos. Este método guarda en el disco de manera asíncrona, aunque modifica inmediatamente el archivo en memoria. Además, si se da el caso que más de un editor está haciendo un commit(), y hay algún apply que no ha terminado, el commit() bloqueará todo hasta que terminen todos los commit, incluido el que está bloqueando. Estemétodo puede reemplazar de manera segura cualquier operación commit(), si estamos ignorando elresultado que retorna la operación. Este método retorna void. No se señala si la operación falló o fueexitosa. Recomendado si estamos ejecutando los cambios en el Thread principal del sistema.
- **clear**: este método elimina todos los valores que se encuentran en el archivo de preferencia. Después que hagamos commit() no quedará nada en el archivo de preferencias, salvo que hayamos salvado algo en el editor que ocupamos para ejecutar clear. Luego de que se ejecuta el método sobre el archivo de preferencias, este clear se ejecuta primero que cualquier otra operación, sin importar si clear se llamó primero o después. Este método retorna un SharedPreferences.Editor
- **commit**: Nos permite almacenar los datos de manera inmediata en el archivo de preferencias, al igual que con apply debemos considerar que si dos editores hacen commit, el último sobreescribirá los datos. Este método se ejecuta de modo síncrono al disco, por lo que no se recomienda su ejecución en el Thread principal de la aplicación. Todos los commit pueden ser reemplazados por apply de manera segura. Commit retorna un boolean verdadero si los valores fueron almacenados de forma correcta.
- **putBoolean**: método que permite almacenar un valor boolean en las preferencias, recibe una clave tipo String, que es el nombre de la preferencia, y el valor de la misma, en este caso un true o false. Si la llave no existe creará el valor en el archivo, si existe la reemplaza. Este método retorna un editor que es ocupado para hacer commit o apply.
- **putFloat**: método que permite almacenar un valor del tipo Float en las preferencias, recibe una clave tipo String y un valor de tipo Float. Se crea la clave de no existir, si existe se reemplaza. Este método retorna un editor que es ocupado para hacer commit o apply.

- **putInt**: método que permite almacenar un valor de tipo Int en las preferencias. Recibe una clave tipo String y un valor de tipo Int. Se crea si no existe, si existe se reemplaza. Este método retorna un editor que es ocupado para hacer commit o apply.
- **putLong**: método que permite almacenar un valor de tipo Long. Recibe una clave tipo String y un valor tipo Long. Se crea si no existe, si existe se reemplaza. Este método retorna un editor que es ocupado para hacer commit o apply.
- **putString**: método que permite almacenar un valor de tipo String. Recibe una clave tipo String y un valor tipo String. Se crea si no existe, si existe se reemplaza. Este método retorna un editor que es ocupado para hacer commit o apply.
- **putStringSet**: método que permite almacenar un Set de Strings o Set en las preferencias. Recibe una clave tipo String y un Set de String para almacenar. Si no existe se crea, si existe se reemplaza. Este método retorna un editor que es ocupado para hacer commit o apply.
- **remove**: método que nos permite eliminar claves específicas, se ejecuta cuando se aplica commit al archivo de preferencias. El método se ejecuta primero que cualquier otro método independiente del orden en que se ejecutó en el código.

A continuación podemos ver algunos ejemplos del uso de un Editor de SharedPreferences, recordando que los utilizaremos en nuestra ejemplo paso por paso.

```
sharedPreferences.edit().putString("TestString",  "Hola").apply()
sharedPreferences.edit().putBoolean("TestBoolean",  true).apply()
sharedPreferences.edit().putFloat("TestFloat", 1.0f).apply()
sharedPreferences.edit().putInt("TestInt", 1).apply()
sharedPreferences.edit().putLong("TestLong", 1L).apply()
val setString = setOf("one", "two", "three", "four")
sharedPreferences.edit().putStringSet("TestStringSet",  setString).apply()
```

Métodos de SharedPreferences

La interfaz nos permite no sólo editar, con la interfaz Editor, también nos permite leer y realizar otras operaciones sobre el archivo de preferencias. Las otras operaciones que podemos realizar son las siguientes:

- **contains**: método que recibe una clave y nos devuelve un boolean indicando si esta clave está presente o no en el archivo de preferencias. Sólo nos dice si el archivo contiene la clave, no el valor.
- **edit**: es el método que nos devuelve el Editor que describimos en el apartado anterior.
- **getAll**: este método nos entrega un Map con todas las claves y valores presentes en el archivo de preferencias. Podemos iterar sobre este Map y recorrer todas la claves del archivo, pero debemos saber específicamente qué tipo de valor está relacionado con cada clave, para poder operar correctamente con ese valor.
- **getBoolean**: nos permite leer un boolean desde el archivo de preferencias. Recibe dos parámetros una clave y un valor por defecto. Con la clave se hace una búsqueda en el archivo de preferencias, si no existe la clave se crea y se devuelve el valor por defecto que le pasamos como segundo parámetro. Si la clave existe, se devuelve el valor boolean de esa clave y se ignora el valor por defecto que entregamos como parámetro. Si la clave existe, pero no es de tipo boolean, el método entrega una ClassCastException.
- getFloat: nos permite leer un Float desde el archivo de preferencias. Recibe dos parámetros una clave y un valor por defecto. Con la clave se hace una búsqueda en el archivo de preferencias, si no existe la clave se crea y se devuelve el valor por defecto que le pasamos como segundo parámetro.Si la clave existe, se devuelve el valor float de esa clave y se ignora el valor por defecto que entregamos como parámetro. Si la clave existe, pero no es de tipo Float, el método entrega una ClassCastException.
- **getInt**: nos permite leer un Int desde el archivo de preferencias. Recibe dos parámetros una clave y un valor por defecto. Con la clave se hace una búsqueda en el archivo de preferencias, si no existe la clave se crea y se devuelve el valor por defecto que le pasamos como segundo parámetro. Si la clave existe, se devuelve el valor int de esa clave y se ignora el valor por defecto que entregamos como parámetro. Si la clave existe, pero no es de tipo Int, el método entrega unaClassCastException.
- **getLong**: nos permite leer un Long desde el archivo de preferencias. Recibe dos parámetros una clave y un valor por defecto. Con la clave se hace una búsqueda en el archivo de preferencias, si no existe la clave se crea y se devuelve el valor por defecto que le pasamos como segundo parámetro. Si la clave existe, se devuelve el valor long de esa clave y se ignora el valor por defecto que entregamos como parámetro. Si la clave existe, pero no es de tipo Long, el método entrega una ClassCastException.

- **getString**: nos permite leer un String desde el archivo de preferencias. Recibe dos parámetros una clave y un valor por defecto. Con la clave se hace una búsqueda en el archivo de preferencias, si no existe la clave se crea y se devuelve el valor por defecto que le pasamos como segundo parámetro. Si la clave existe, se devuelve el valor String de esa clave y se ignora el valor por defecto que entregamos como parámetro. Si la clave existe, pero no es de tipo Int, el método entrega una ClassCastException.
- **getStringSet**: nos permite leer un Set de Strings desde el archivo de preferencias. Recibe dos parámetros una clave y un valor por defecto. Con la clave se hace un búsqueda en el archivo de preferencias, si no existe la clave se crea y se devuelve el valor por defecto que le pasamos como segundo parámetro. Si la clave existe, se devuelve el Set con los valores de esa clave y se ignora el valor por defecto que entregamos como parámetro. Si la clave existe, pero no es de tipo Set, el método entrega una ClassCastException.

Algunos ejemplos de estos métodos

```
//para saber si contiene cierta clave
sharedPreferences.contains("TestString")
//si quiero todos los valores y claves contenidos
//pasa de ser getAll() a all gracias a kotlin
sharedPreferences.all
//obtener un boolean
sharedPreferences.getBoolean("TestBoolean", false)
//obtener un int
sharedPreferences.getInt("TestInt", -1)
//obtener un long
sharedPreferences.getLong("TestLong", -1L)
//obtener un Float
sharedPreferences.getFloat("TestFloat", -1.0f)
//obtener un String
sharedPreferences.getString("TestString", "NotFound")
//obtener un Set<String>
sharedPreferences.getStringSet("TestStringSet", setOf())
```

Además SharedPreferences cuenta con una interfaz listener que nos permite ser notificados cuando una propiedad cambia en el archivo de preferencias.

SharedPreferences.OnSharedPreferencesChangeListener

Esta interfaz nos permite escuchar los cambios que ocurren en un archivo de preferencias. Para que esto funcione, debemos implementar la Interfaz en nuestra clase que va a escuchar, para este caso digamos que nuestra MainActivity será la clase que escuchar y debe implementar la interfaz. El código queda de la siguiente manera:

```
class MainActivity : AppCompatActivity(),
SharedPreferences.OnSharedPreferenceChangeListener {
   override fun onSharedPreferenceChanged(
          changedSharedPreferences: SharedPreferences?,
          changedKey: String?) {

    if (changedKey.equals("UserNickName", true)) {
           val newNickName =  changedSharedPreferences?
                              .getString(changedKey, "")
           displayText.text = newNickName
       }
   }

   private lateinit var sharedPreferences: SharedPreferences
   private lateinit var displayText:TextView
...
```

Lo primero que hacemos es implementar la interfaz en la firma de nuestra Activity class, separando la herencia de AppCompatActivity(). Recordando que podemos heredar de sólo una clase, pero podemos implementar todas las interfaces que necesitemos.

```
class MainActivity : AppCompatActivity(),
SharedPreferences.OnSharedPreferenceChangeListener
```

Luego implementamos el método que se define en la interfaz, el método onSharedPreferencesChanged. Este método tiene dos parámetros, que se envían cuando el archivo de SharedPreferences es modificado. El primer parámetro es el archivo de SharedPreferences, el segundo es la clave que cambió.En este caso estamos esperando el cambio de la clave “UserNickName”, para desplegar ese nuevo nickName en un TextView, que se llama displayText, una variable definida en la Actividad.

```
override fun onSharedPreferenceChanged(
     changedSharedPreferences: SharedPreferences?,
     changedKey: String?) {

  if (changedKey.equals("UserNickName", true)) {
      val newNickName = changedSharedPreferences?
               .getString(changedKey, "")
      displayText.text = newNickName
    }
  }
```

Para registrar, y desregistrar, un listener en un archivo de SharedPreferences ocupamos dos métodos que están disponibles en SharedPreferences:

- **registerOnSharedPreferenceChangeListener**: nos permite registrar un listener en un archivo de preferencias. Este listener será notificado cuando ocurra un cambio en dicho archivo. Este listener, para ser notificado, debe no ser reciclado en alguna de las pasadas del recolector de basura del sistema. SharedPreferences guarda siempre una referencia débil de los listeners, para evitar memory leaks con referencias circulares. Una referencia circular ocurre cuando A refiere a B y B refiere a A, en ese caso, cuando el recolector de basura quiere reciclar, A y B siempre tienen una referencia. Cuando un objeto siempre tiene una referencias, esa memoria no se libera, por lo que se produce un memory leak si ese objeto está sin utilizar, pero no se puede reciclar.Por lo que se recomienda que el listener sea siempre el que tiene la referencia a SharedPreferences, esta referencia debe ser fuerte, normalmente una referencia a this es fuerte. El método de registro recibe como parámetro un OnSharedPreferenceChangeListener listener, normalmente una clase que implementa la interfaz, o una implementación de la interfaz como propiedad de otra clase.
- **unregisterOnSharedPreferenceChangeListener**: este método nos permite desregistrar un listener del archivo de preferencias. Recibe como parámetro el listener que se quiere remover.

Podemos registrar un listener de la siguiente forma

```
sharedPreferences.registerOnSharedPreferenceChangeListener(this)
```

En este caso estamos registrando nuestra MainActivity, que implementa la interfaz OnSharedPreferenceChangeListener. Esta actividad será notificada cuando cambie sharedPreferences, y cambiará el texto en displayText, cuando la clave que cambió sea igual a UserNickName.

```
override fun onSharedPreferenceChanged(
     changedSharedPreferences: SharedPreferences?,
     changedKey: String?) {

  if (changedKey.equals("UserNickName", true)) {
      val newNickName = changedSharedPreferences?
               .getString(changedKey, "")
      displayText.text = newNickName
    }
  }
```

También podemos definir una variable que sea parte de la Actividad, en este caso se puede implementar como un lambda, debido a que es una interfaz con un sólo método, un ejemplo de cómo podemos hacer esto es:

```
val listener = SharedPreferences
    .OnSharedPreferenceChangeListener { preferences, key ->
     if (key.equals("TestString", true)) {
        displayText.text = preferences.getString(key, "")
     }
}
sharedPreferences.registerOnSharedPreferenceChangeListener(listener)
```

En este caso definimos una val listener, que es asignada a una clase anónima con `object:SharedPreferences.OnSharedPreferenceChangeListener`. Esta clase define el método `onSharedPreferenceChanged`. Finalmente ocupamos la variable para registrarla como listener de sharedPreferences.

Debemos tener una referencia a este listener para poder desregistrar el mismo cuando ya no sea necesario escuchar los cambios de ese archivo de preferencias. Recordar que es importante desregistrarel listener para evitar posibles referencias circulares que pueden generar memory leaks. Para desregistrar, ocupamos el método de SharedPreferences unregisterOnSharedPreferenceChangeListener,pasando la referencia del listener que queremos desregistrar. El registro y desregistro normalmente se hacen en onResume y onPause respectivamente

```
sharedPreferences.unregisterOnSharedPreferenceChangeListener(this)
sharedPreferences.unregisterOnSharedPreferenceChangeListener(listener)
```

Aplicación de Ejemplo

Desarrollaremos un pequeño ejemplo para entender el uso de SharedPreferences en un aplicación real. En este ejemplo crearemos una aplicación que nos permite almacenar datos entre diferentes corridas. La aplicación que haremos es la siguiente:

- Partimos de una aplicación vacía desde crear un nuevo proyecto en Android Studio
- Nuestra aplicación contará con una pantalla que contiene los siguiente
    - Un input para ingresar un número
    - Un input para ingresar un Texto
    - Un switch para seleccionar
    - Un input para ingresar un número decimal
    - Un botón para salvar todo y uno para borrar todo
    - Textos para desplegar los diferentes tipos guardados

En esta aplicación usaremos todo lo aprendido hasta ahora.

Crear proyecto inicial

Este es el paso más simple, desde la pantalla de Bienvenida de Android Studio utilizamos la opción Start a new Android Studio, y seguimos los paso para crear un Empty activity.

La secuencia se muestra en la imágenes siguientes, los datos utilizados para el proyecto son los siguiente:

- Name: EjemploSharedPreferences
- Package name: cl.desafiolatam.ejemplosharedpreferences
- Languaje: Kotlin
- Minimum API level: 23, Marshmallow

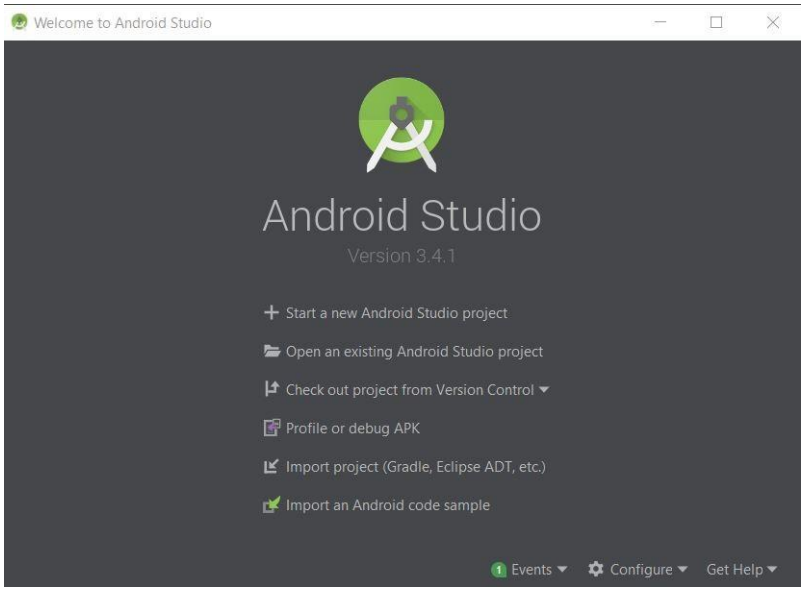


Imagen 3. Pantalla de bienvenida.

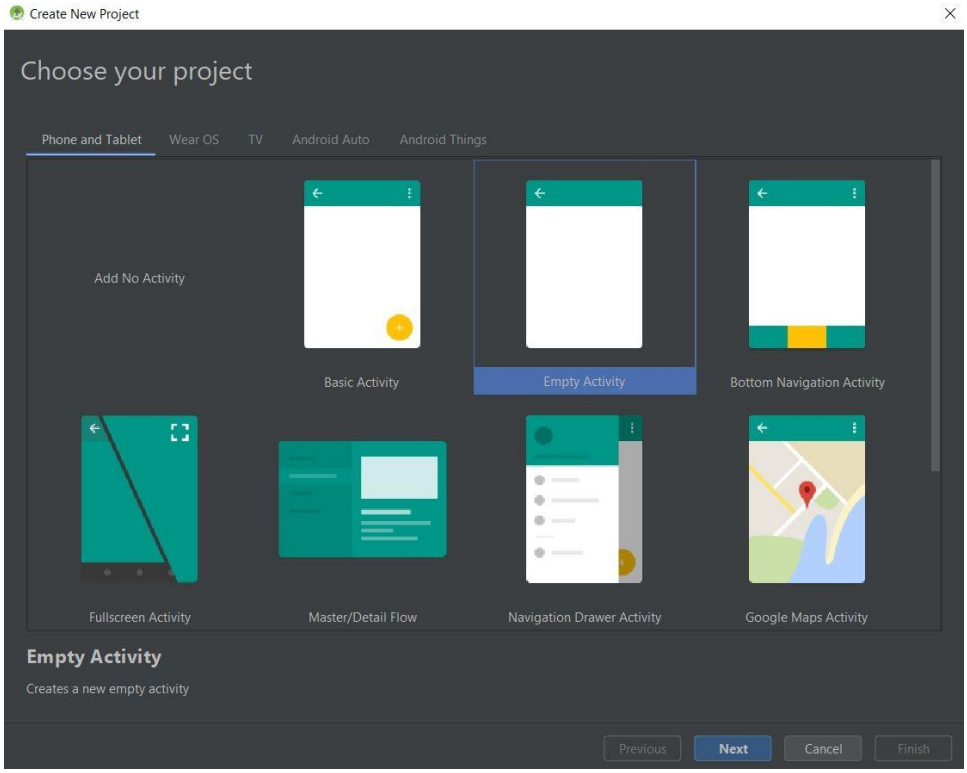


Imagen 4. Pantalla de selección de EmptyActivity.

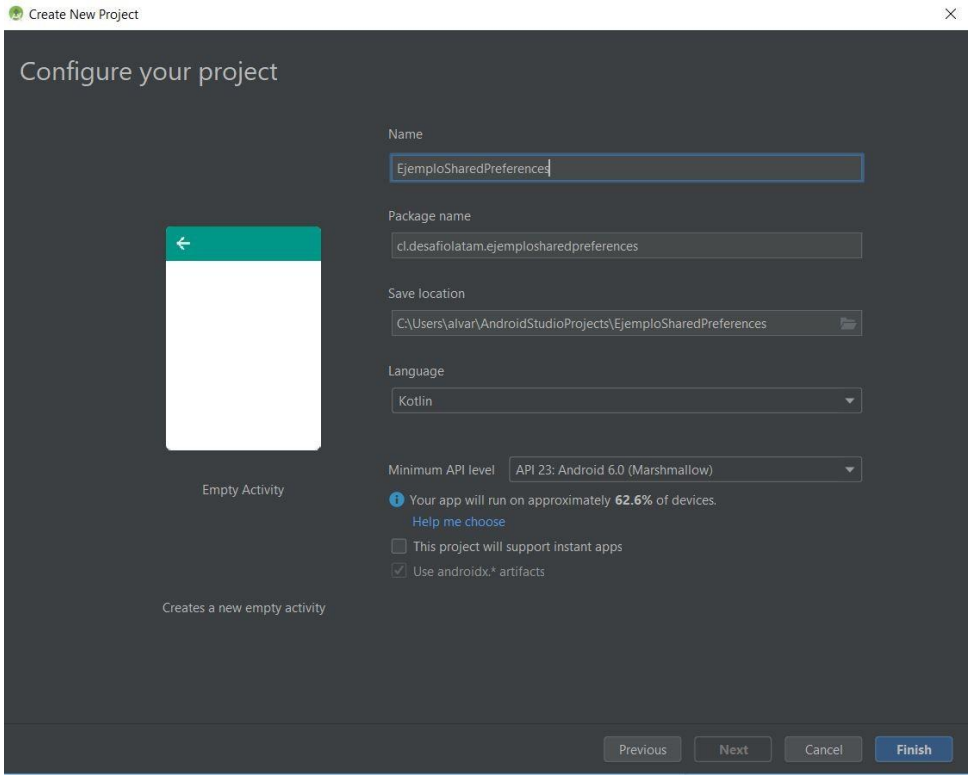


Imagen 5. Configurando el proyecto

Después de presionar Finish, nuestro proyecto inicial está listo. No necesitamos configurar nada más en este proyecto, pongamos manos a la obra.

Pantalla de la aplicación

Para tener de referencia implementaremos una pantalla para esta aplicación con los componentes que podemos ver en la imagen

Imagen 6. Ejemplo de pantalla a implementar.

Para no perder mucho tiempo en esta pantalla, debemos implementar rápidamente el siguiente código XML.

```xml
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
       xmlns:android="http://schemas.android.com/apk/res/android"
       xmlns:tools="http://schemas.android.com/tools"
       xmlns:app="http://schemas.android.com/apk/res-auto"
       android:layout_width="match_parent"
       android:layout_height="match_parent"
       tools:context=".MainActivity">
   <com.google.android.material.textfield.TextInputLayout
           android:layout_width="0dp"
           android:layout_height="wrap_content"
           android:layout_marginTop="32dp"
           app:layout_constraintTop_toTopOf="parent"
           app:layout_constraintEnd_toEndOf="parent"
           android:layout_marginEnd="8dp"
           app:layout_constraintStart_toStartOf="parent"
           android:layout_marginStart="8dp"
           android:id="@+id/numero_container">

       <com.google.android.material.textfield.TextInputEditText
               android:layout_width="match_parent"
               android:layout_height="wrap_content"
               android:hint="Ingresa un número entero"
               android:id="@+id/numero_input"
               android:inputType="number"/>
   </com.google.android.material.textfield.TextInputLayout>
   <TextView
           android:text="Almacenado"
           android:layout_width="wrap_content"
           android:layout_height="wrap_content"
           android:id="@+id/num_ent_title"
           app:layout_constraintTop_toBottomOf="@+id/numero_container"
           app:layout_constraintStart_toStartOf="parent"
           android:layout_marginStart="8dp"
           android:layout_marginTop="16dp"
           android:textSize="18sp"/>
   <TextView
           android:layout_width="0dp"
           android:layout_height="wrap_content"
           android:id="@+id/num_ent_valor"
           app:layout_constraintStart_toEndOf="@+id/num_ent_title"
           android:layout_marginStart="8dp"
```

```
        app:layout_constraintEnd_toEndOf="parent"
        android:layout_marginEnd="8dp"
        app:layout_constraintTop_toBottomOf="@+id/numero_container"
        android:layout_marginTop="16dp"
        android:textSize="18sp"/>
<com.google.android.material.textfield.TextInputLayout
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginTop="16dp"
        app:layout_constraintTop_toBottomOf="@+id/num_ent_valor"
        app:layout_constraintStart_toStartOf="parent"
        android:layout_marginStart="8dp"
        android:layout_marginEnd="8dp"
        app:layout_constraintEnd_toEndOf="parent"
        android:id="@+id/texto_container">

    <com.google.android.material.textfield.TextInputEditText
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:hint="Ingresa un texto"
            android:id="@+id/texto_input"
            android:inputType="text"/>
</com.google.android.material.textfield.TextInputLayout>
<TextView
        android:text="Almacenado"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/text_title"
        app:layout_constraintTop_toBottomOf="@+id/texto_container"
        app:layout_constraintStart_toStartOf="parent"
        android:layout_marginStart="8dp"
        android:layout_marginTop="16dp"
        android:textSize="18sp"/>
<TextView
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:id="@+id/text_valor"
        app:layout_constraintEnd_toEndOf="parent"
        android:layout_marginEnd="8dp"
        app:layout_constraintTop_toBottomOf="@+id/texto_container"
        app:layout_constraintStart_toEndOf="@+id/text_title"
        android:layout_marginStart="8dp"
        android:layout_marginTop="16dp"
        android:textSize="18sp"/>
<Switch
        android:text="Seleccionado"
```

```
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:id="@+id/switch_input"
        app:layout_constraintEnd_toEndOf="parent"
        android:layout_marginEnd="16dp"
        app:layout_constraintStart_toStartOf="parent"
        android:layout_marginStart="8dp"
        app:layout_constraintTop_toBottomOf="@+id/text_valor"
        android:layout_marginTop="16dp"
        android:textSize="18sp"/>
<com.google.android.material.textfield.TextInputLayout
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginTop="16dp"
        app:layout_constraintTop_toBottomOf="@+id/switch_input"
        android:layout_marginEnd="8dp"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        android:layout_marginStart="8dp"
        android:id="@+id/num_dec_container">

    <com.google.android.material.textfield.TextInputEditText
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:hint="Ingresa un número decimal"
            android:id="@+id/num_dec_input"
            android:inputType="number|numberDecimal"/>
</com.google.android.material.textfield.TextInputLayout>
<TextView
        android:text="Almacenado"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/num_dec_title"
        android:layout_marginTop="16dp"
        app:layout_constraintTop_toBottomOf="@+id/num_dec_container"
        app:layout_constraintStart_toStartOf="parent"
        android:layout_marginStart="8dp"
        android:textSize="18sp"/>
<TextView
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:id="@+id/num_dec_valor"
        app:layout_constraintEnd_toEndOf="parent"
        android:layout_marginEnd="8dp"
        android:layout_marginTop="16dp"
        app:layout_constraintTop_toBottomOf="@+id/num_dec_container"
```

```
            app:layout_constraintStart_toEndOf="@+id/num_dec_title"
            android:layout_marginStart="8dp"
            android:textSize="18sp"/>
    <Button
            android:text="Guardar"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:id="@+id/guardar"
            android:layout_marginTop="32dp"
            app:layout_constraintTop_toBottomOf="@+id/num_dec_valor"
            app:layout_constraintEnd_toEndOf="parent"
            android:layout_marginEnd="32dp"/>
    <Button
            android:text="Borrar"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:id="@+id/borrar"
            android:layout_marginTop="32dp"
            app:layout_constraintTop_toBottomOf="@+id/num_dec_valor"
            app:layout_constraintStart_toStartOf="parent"
            android:layout_marginStart="32dp"
    />
</androidx.constraintlayout.widget.ConstraintLayout>
```

Utilizamos un ConstraintLayout para mantener todos nuestros controles ordenados, con un id, que se utilizará en el código para referenciarlos. Notar que los inputs definen sus teclados para evitar problemas de valores, number para el número entero, decimal Number para el número decimal. Esto se hace agregando la propiedad android:inputType="number" y android:inputType="number|numberDecimal" a los TextInputEditText correspondientes. Ahora veremos que es lo necesario para utilizar SharedPreferences en esta aplicación.

Actividad Principal, variables y métodos

En nuestra aplicación abrimos el archivo de MainActivity.kt. Podemos ver que en su estado inicial sólo contamos con el esqueleto y un método onCreate donde se inicializa la actividad y se configura su vista en xml. El código debería verse de la siguiente manera

```
class MainActivity : AppCompatActivity() {
override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }
}
```

Lo primero que debemos hacer es crear una variable para mantener nuestro archivo referenciado durante todo el ciclo de esta aplicación, en este caso nos basta con utilizar una propiedad de tipo var en MainActivity. Agreguemos el siguiente código, recordado que esta variable debe ser lateinit para ser iniciada en onCreate, el código debe ser agregado antes de la función onCreate.

```
class MainActivity : AppCompatActivity() {
lateinit var sharedPreferences: SharedPreferences
override fun onCreate(savedInstanceState: Bundle?) {
```

También agregamos dentro del método onCreate, después de setContentView, la inicialización de sharedPreferences, con el nombre del paquete y en modo privado:

```
override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val fileName = "cl.desafiolatam.ejemplosharedpreferences"
sharedPreferences = getSharedPreferences(fileName, Context.MODE_PRIVATE)
}
```

Hemos creado nuestro archivo de manera correcta, y en el modo correcto. Ahora necesitamos referenciar todos nuestros controles y los nombres de las claves que utilizaremos para almacenar y leer los datos desde el archivo, en este caso usaremos las siguientes variables que agregamos después de nuestra var de sharedpreferences y antes de onCreate. Además agregamos un companion object para definirnuestras claves, usamos el siguiente código:

```
class MainActivity : AppCompatActivity() {
companion object {
   private const val baseClave = "EjemploSP"
   const val claveNumEnt = baseClave + "NumEnt"
   const val claveTexto = baseClave + "Texto"
   const val claveSwitch = baseClave + "Switch"
   const val claveNumDec = baseClave + "NumDec"
}
lateinit var sharedPreferences: SharedPreferences
lateinit var numEntInput:TextInputEditText
lateinit var numEntValue:TextView
lateinit var textoInput:TextInputEditText
lateinit var textoValue:TextView
lateinit var switch:Switch
lateinit var numDecInput:TextInputEditText
lateinit var numDecValue:TextView
lateinit var borrar:Button
lateinit var guardar:Button
```

Recordemos que un companion object en Kotlin es lo mismo que definir variables estáticas en java, y estas variables son compartidas por todas las instancias de la clase que las implementa.

Para inicializar nuestros widgets variables escribimos un método, en el cuerpo de Main Activity, que llamaremos después de inicializar el archivo de sharedPreferences en el método onCreate, el método que utilizamos para inicializar los widgets o controles de la pantalla es el siguiente:

```
private fun initViews() {
   numEntInput = findViewById(R.id.numero_input)
   numEntValue = findViewById(R.id.num_ent_valor)
   textoInput = findViewById(R.id.texto_input)
   textoValue = findViewById(R.id.text_valor)
   switch = findViewById(R.id.switch_input)
   numDecInput = findViewById(R.id.num_dec_input)
   numDecValue = findViewById(R.id.num_dec_valor)
   borrar = findViewById(R.id.borrar)
   guardar = findViewById(R.id.guardar)
```

```
    borrar.setOnClickListener(object: View.OnClickListener {
        override fun onClick(p0: View?) {
            borrar()
        }
    })

    guardar.setOnClickListener(object: View.OnClickListener {
        override fun onClick(p0: View?) {
            guardar()
        }
    })
}
```

En este método también definimos los listener necesarios para que el botón borrar y guardar funcionen. En el método borrar() y guardar() utilizamos SharedPreferences para almacenar y eliminar lo que uno agrega como usuario al archivo que hemos creado. Además creamos un tercer método que nos permite leer los datos almacenados desde el archivo de SharedPreferences y actualizar los controles necesarios para mostrar esos datos al usuario. Veamos estos métodos y cómo utilizan SharedPreferences.

Método: leerDesdeSharedPreferences()

Este método se utiliza para leer los diferentes datos almacenados en el archivo de sharedpreferences que creamos y cargar los datos iniciales en la aplicación. La primera vez que corremos la aplicación, tendremos sólo valores por defecto, pero en las siguientes corridas se mostrarán los datos almacenados por el usuario al presionar guarda. Definimos el método de la siguiente manera:

```
private fun leerDesdeSharedPreferences() {
    numEntValue.text = sharedPreferences.getInt(claveNumEnt, 0).toString()
    textoValue.text = sharedPreferences.getString(claveTexto, "")
    val checked = sharedPreferences.getBoolean(claveSwitch, false)
    switch.isChecked = checked
    switch.isSelected = checked
    numDecValue.text = sharedPreferences.getFloat(claveNumDec,0.0f).toString()
}
```

En este método, que llamamos en onCreate después de inicializar el archivo en sharedPreferences, hacemos lo siguiente:

- En el TextView numEntValue mostraremos el valor almacenado para el número entero que podemos ingresar en la aplicación. Como vemos en el código utilizamos el método getInt, con la clave claveNumEnt definida en nuestro companion object, y tenemos un valor por defecto 0, para mostrar el texto en un TextView debemos convertir a string el valor, usamos el método toString para esto. Mientras el usuario no guarde un valor, el valor mostrado será 0.
- En el TextView textoValue mostramos el texto almacenado por el usuario. Utilizamos getString para leer el valor, con la clave claveTexto y un valor por defecto vacío. Mientras el usuario no almacene algo nuevo, el texto mostrado será vacío.
- En el Switch del mismo nombre mostramos si el usuario ha seleccionado o no este switch. El valor de esto lo almacenamos como un boolean, por lo que para obtener el valor almacenado utilizamos getBoolean, con la clave claveSwitch y un valor por defecto false. Notar que se debe setear el valor de isChecked y isSelected para que el switch muestre el estado correcto.
- En el TextView numDecValue mostramos el valor del número decimal almacenado, para obtener el valor utilizamos getFloat, con la clave claveNumDec y un valor por defecto de 0.0

En este método utilizamos lo que nos ofrece SharedPreferences para leer el archivo, ahora veamos cómo almacenar y eliminar datos.

Método guarda()

En el método guardar ocupamos el editor para almacenar los diferentes datos que se pueden ingresar en la aplicación, el código de este método es el siguiente:

```
private fun guardar() {
   if (numEntInput.text!!.isNotEmpty()) {
       val numEnt = numEntInput.text.toString().toInt()
       sharedPreferences.edit().putInt(claveNumEnt, numEnt).apply()
   }
   if (textoInput.text!!.isNotEmpty()) {
 sharedPreferences.edit()
  .putString(claveTexto, textoInput.text.toString())
   .apply()
   }
   val checked = switch.isChecked
   sharedPreferences.edit().putBoolean(claveSwitch, checked).apply()
   if (numDecInput.text!!.isNotEmpty()) {
       val numDec = numDecInput.text.toString().toFloat()
       sharedPreferences.edit().putFloat(claveNumDec, numDec).apply()
   }
}
```

En este método utilizamos edit() y almacenamos diferentes tipos de datos de la siguiente manera:

- Lo primero es almacenar el número entero que el usuario ingresa en numEntInput. Cómo tenemos un texto, no podemos llegar y almacenar el valor, lo primero es ver si no está vacío. Si no es vacío, convertimos el valor a string y luego int, almacenando el mismo en numEnt. Luego lo almacenamos con edit.putInt(claveNumEnt, numEnt).apply(), de esta forma nos aseguramos de que la aplicación no se caiga y que no se almacenen datos vacíos.
- Para almacenar el texto en textoInput, también vemos si no es vacío. Si pasa esta prueba podemos almacenar el valor desde el input, convirtiéndolo con toString. Para almacenar utilizamos putString(claveTexto,...).apply().
- Para el switch almacenaremos un boolean, rescatamos el valor de isChecked en val checked y luego utilizamos edit().putBoolean(claveSwitch, checked).apply() para almacenar el valor en elarchivo. Nunca es vacío así que no necesitamos chequear este valor.
- Para el número decimal, chequeamos que no esté vacío. Utilizamos numDec para almacenar el valor no vacío convertido, y lo guardamos usando edit().putFloat(claveNumDec, numDec).apply().

Ahora veamos cómo borrar estos valores del archivo.

Método: borra()

En este método utilizamos edit().remove para eliminar los datos guardados, definimos este método de la siguiente forma:

```
private fun borrar() {
    sharedPreferences.edit().remove(claveNumEnt).apply()
    sharedPreferences.edit().remove(claveTexto).apply()
    sharedPreferences.edit().remove(claveSwitch).apply()
    sharedPreferences.edit().remove(claveNumDec).apply()
}
```

Este método es bastante simple, sólo debemos notar que no hay problemas en borrar claves que no existe, si el archivo no las encuentra no hace nada y el método apply se ejecuta correctamente.

Para finalizar nuestro ejemplo debemos modificar el código en el método onCreate de la siguiente forma:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    val fileName = "cl.desafiolatam.ejemplosharedpreferences"
    sharedPreferences = getSharedPreferences(fileName, Context.MODE_PRIVATE)
    initViews()
    leerDesdeSharedPreferences()
}
```

Notar el orden de los métodos initViews y leerDesdeSharedPreferences, si no utilizamos ese orden la aplicación se caerá por null pointers en las variables que referencian a los controles en pantalla. Luego de modificar el método onCreate, y hemos agregado todo el código a MainActivity, presionamos run app, el botón play en Android Studio. Si todo está bien debemos ver la siguiente pantalla:

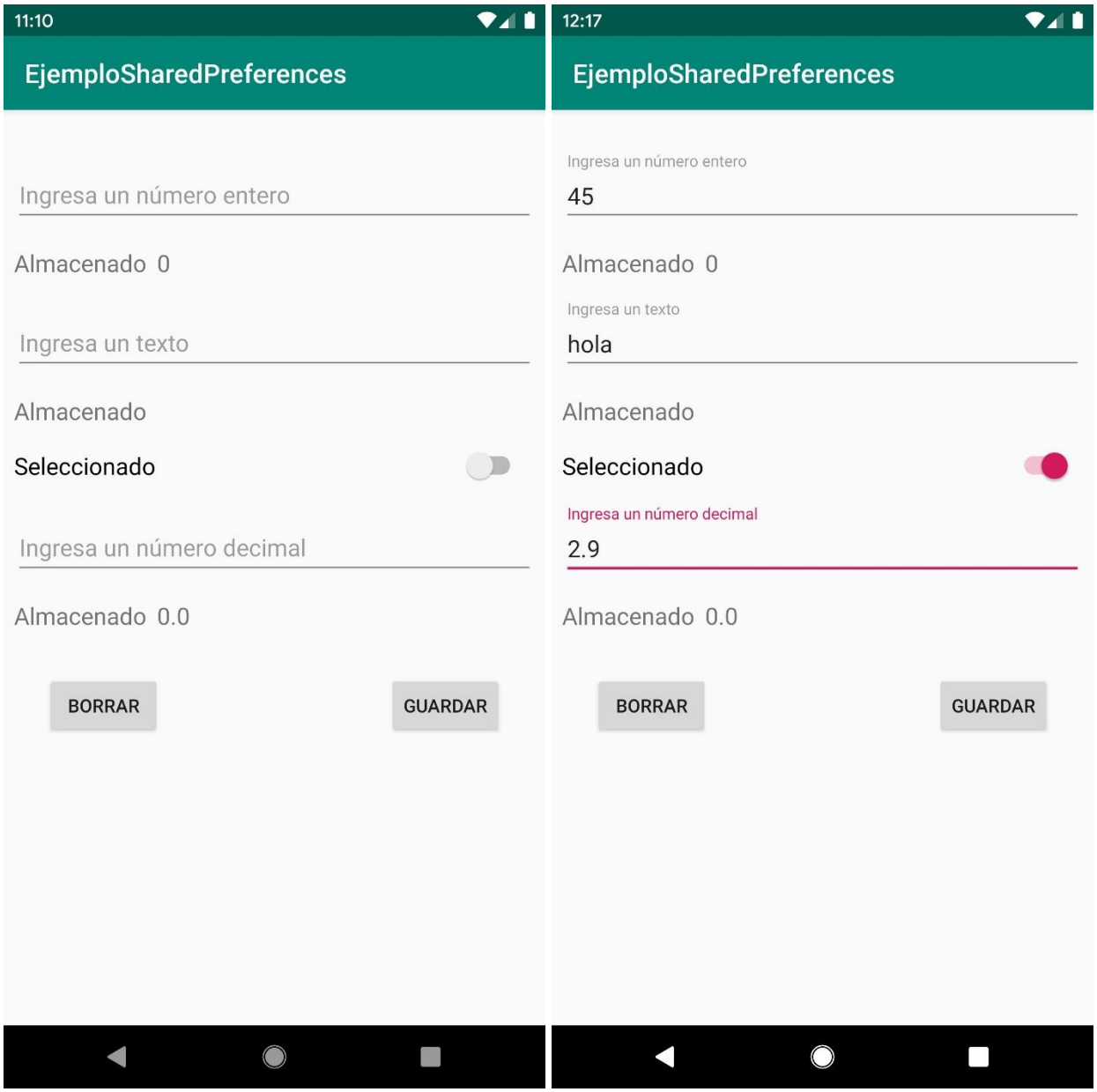


Imagen 7 y 8. Pantallas al ejecutar la aplicación.

Después de iniciar la aplicación podemos agregar algunos datos, para luego presionar guardar. Después de guardar debemos cerrar la aplicación. Al abrirla nuevamente veremos los datos cargados en los distintos controles como vemos en la imagen a continuación:

Imagen 9. Datos cargados posterior al haber cerrado la aplicación.

Así finalizamos el ejemplo. A continuación veremos Room.

2.4 Android Jetpack y Room

Room

Competencias:

- Entender que es un ORM, y cómo interactúa con una base de datos
- Definir que es Room, y qué papel juega en el paquete de Architecture Components
- Utilizando Room para construir una base de datos, tablas y DAO para persistir datos en una aplicación.
- Describir las ventajas de utilizar Room sobre SQL en una aplicación.

Motivación

Una de las necesidades más recurrentes de una aplicación móvil es el almacenamiento de datos estructurados. La opción más común para manejar datos estructurados en Android es SQLite. El principal problema con SQLite es la generación de consultas. Lamentablemente no podemos saber si una de estas consultas presenta algún error hasta que la ocupemos en nuestro código, es decir hasta que corramos la aplicación en un dispositivo y utilicemos la consulta. Estos errores son conocidos como Run Time Error, lo cuales sólo se presentan en tiempo de ejecución. Cuando utilizamos SQLite directamente debemos manejar estos errores considerando los distintos escenarios, o simplemente hacer prueba y error para corregir, lo que hace el proceso de desarrollo lento y tedioso. Para evitar estos errores, y simplificar la interacción con SQLite, Google crea Room, una librería que permite generar una capa de interacción con la base de datos, que además identifica los errores en tiempo de compilación, evitando los errores en tiempo de ejecución. Room es parte de Jetpack. Veremos toda la teoría detrás de Room, con un ejemplo donde construiremos las bases de un aplicación y posteriormente completamos la aplicación y veremos ejemplos más avanzados del uso de Room en Android.

¿Por qué Room?

Room nos permite crear una capa de abstracción sobre SQLite para facilitar el acceso a los datos, y la interacción con dicha Base de Datos. Una aplicación que maneja grandes cantidades de data estructurada se benefician directamente con la implementación de room.

Para entender bien los beneficios veamos un ejemplo teórico de cómo podríamos modelar una libreta de contactos.

Libreta de Contactos

Necesito crear una aplicación que maneje contactos telefónicos, normalmente esto lo podemos modelar en una aplicación orientada a objetos con una clase que represente a la Entidad Persona, esta clase será la base de los datos que maneja nuestra aplicación. Nuestros contactos tendrán las siguientes características:

- Nombre y Apellido
- Número telefónico
- Dirección, donde almacenaremos los siguientes datos:
    - Nombre calle
    - Número de la calle
    - Comuna
    - Región
- Correo electrónico

Las clases necesarias para este modelo de datos se pueden definir de la siguiente manera en Kotlin:

```
class Person(var name:String,
      var surname:String,
      var phonenumber:String,
      var address: Address,
      var email:String)
```

```
class Address(var streetName:String,
       var streetNumber:Int,
       var district: String,
       var region:String)
```

Tenemos dos clases, una representa a una persona, que contiene un conjunto de datos, además de una segunda clase que representa una dirección. Podemos discutir cómo modelar esto de diferentes manera, pero por ahora esta será nuestra definición.

La pregunta es: ¿cómo almacenamos esto en nuestro dispositivo?.

Respuesta: SQLite

Debemos pensar cómo son las tablas que necesitamos, cuáles son las id primarias, cómo se modela, conecta e interpreta la información entre las tablas y los objetos que definimos anteriormente, etc. Lo anterior es bastante trabajo y genera una gran cantidad de código. Necesitamos crear todo el código que crea la base de datos, las migraciones, las tablas, un DAO para interactuar con la base de datos, etc. A todo este código necesario, pero que genera mucho trabajo extra para un beneficio pequeño, se le conoce como boilerplate code.

Normalmente este boilerplate code, es bastante trabajo y genera muchos problemas de mantenimiento, cada vez que hacemos un cambio necesitamos actualizar todas las clases involucradas, lo cual puede generar más de algún error.Normalmente para evitar tener que escribir todo este código, se puede utilizar una herramienta. En este caso un ORM.

Room como ORM

Un Object Relational Mapper, ORM, es una herramienta que genera todo este código intermedio, boilerplate code, que une o pega nuestra aplicación con su base de datos.

Los ORM no son invención de Android, son ocupados hace mucho tiempo, y tenemos a nuestra disposición muchas opciones. La cantidad de opciones generó mucha fragmentación y limitaciones. Además, las aplicaciones tienen cada vez ciclos de vida más complejos, lo que hace difícil tomar una decisión de cuál elegir. Si elegimos de mala manera, y la arquitectura de nuestra aplicación no es la adecuada, cambiar de ORM puede ser un dolor de cabeza mayor.

Como respuesta a la fragmentación, errores comunes, y como parte de su cruzada por mejores aplicaciones con mejor arquitectura, Google crea Room. Room es parte de Jetpack, y se recomienda como uno de los componentes de una arquitectura por capas tipo MVVM o MVP. Room es parte de la capa de datos, que se recomienda implementar como un repositorio. Conceptualmente podemos entender Room como el mediador entre la Base de datos SQLite y la capa de acceso de datos de la aplicación, pudiendo esta ser implementada como repositorio u otro tipo de patrón arquitectónico.

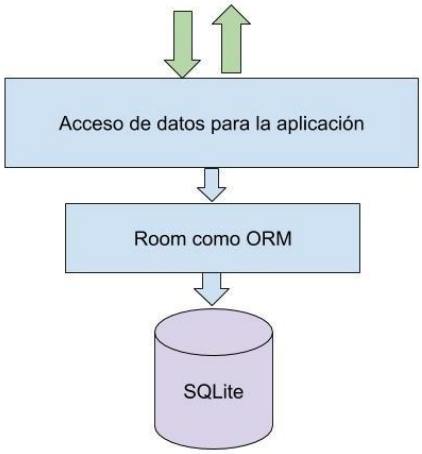


Imagen 1. Room actúa como intermediario entre SQLite y la capa de acceso a datos

Componentes de Room

Para utilizar Room debemos generar sus tres componentes principales:

- **Entity**: Representa un data model, o modelo de datos, que se mapea directamente a las tablas de la base de datos.
- **DAO**: Data Access Object, este objeto nos permite interactuar con la base de datos. Normalmente la interacción es del tipo CRUD (Create, Read, Update and Delete)
- **Database**: Un contenedor que mantiene una referencia y actúa como único punto de acceso a la base de datos.

Para explicar cómo funcionan, y qué papel juegan, cada uno de estos componentes en Room, definamos un problema a resolver. Necesitamos almacenar en un dispositivo un conjunto de recomendaciones, en una especie de libro de apuntes o notas. Pensemos en la libreta que tenía el capitán américa en Winter Soldier, veamos una imagen:

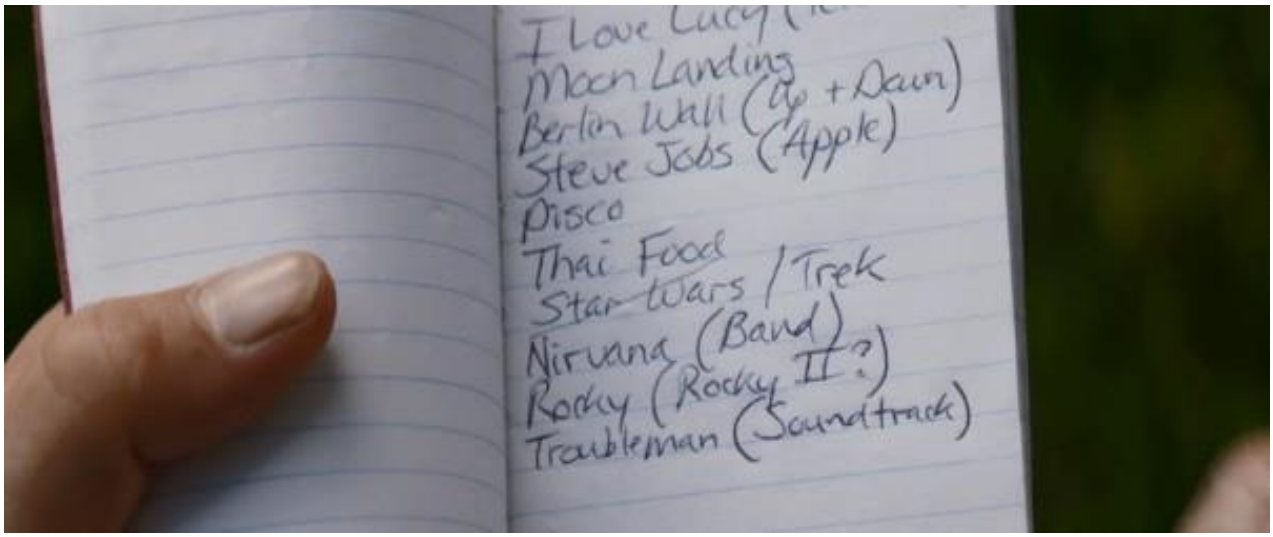


Imagen 2. Libreta.

Esta aplicación ficticia nos permitirá agregar información, o recomendaciones para ponernos al día, y guardarlas persistentemente en nuestro dispositivo. La aplicación tiene una pantalla donde vemos la lista, con un botón que nos permite agregar información. Un ejemplo de la pantalla en la siguiente imagen:

Imagen 3. Hola Mundo.

El objetivo de este ejercicio teórico es crear un aplicación, siguiendo los diferentes pasos del documento, pero en este capítulo veremos toda la parte teórica de Room, sus anotaciones y componentes. En el siguiente capítulo completamos la aplicación aquí descrita. Utilizaremos este ejemplo para definir los diferentes componentes de Room, cómo interactúan y que necesitamos en el código para cada uno. Vamos a partir configurando Room en un proyecto, para este paso se puede utilizar cualquier proyecto inicial, para crear un proyecto inicial puede seguir los pasos descritos en el capítulo dos para la aplicación de ejemplo de SharedPreferences. Veamos cómo se agrega Room a un proyecto.

Configuración de Room en un proyecto Android

Para tener acceso a la librería y todas sus herramientas, debemos configurar las dependencias necesarias en nuestro Gradle file del módulo. Cómo estamos usando Kotlin, debemos cerciorarnos de lassiguientes cosas:

En el gradle principal del proyecto debemos agregar las dependencias de kotlin

```
buildscript {
   ext.kotlin_version = '1.3.40' //versión de kotlin definida como variable
   repositories {
       google()
       jcenter()

   }
   dependencies {
       classpath 'com.android.tools.build:gradle:3.4.1'
       classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
   }
}
```

En el gradle file del módulo debemos tener las siguientes configuraciones:

- En el tope del archivo debemos agregar los siguientes plugin

```
apply plugin: 'kotlin-android'
apply plugin: 'kotlin-android-extensions'
apply plugin: 'kotlin-kapt'
```

- Y en la parte de dependencias debemos agregar lo siguiente

```
def room_version = "2.1.0"
implementation "androidx.room:room-runtime:$room_version"
kapt "androidx.room:room-compiler:$room_version"
```

Notar que como estamos usando Kotlin, debemos utilizar kapt para la dependencia de room, en vez de annotationProcessor que es lo normal en java. Con estas configuraciones agregadas a nuestros gradle files, estamos listos para utilizar Room en nuestro proyecto. Comencemos por definir la primera parte de nuestro puzzle, definamos nuestro Entity.

Entity

Es el primer componente que debemos definir para nuestra aplicación. Un Entity, es un objeto que nos ayuda a mantener y modelar datos dentro de nuestra aplicación. Estos datos pueden estar almacenados o ser manejados en memoria. Un Entity modela con un objeto el tipo de datos que luego almacenaremos en la base de datos. Además, a este “modelo” le agregamos información para describir cómo construir la tabla en la base de datos. Para nuestro ejemplo definimos un set de datos y una tabla con columnas, considerando lo siguiente:

- Almacenaremos un texto con la recomendación
- Almacenaremos un id para mantener ordenados nuestros datos, este id será numérico y auto incrementado por la base de datos.
- Nuestra base de datos sólo tendrá una tabla, con dos columnas, por lo que hemos definido anteriormente.

Resolver esto en SQLite no es trivial, pero gracias a las anotaciones y la simplicidad de Room, podemos implementar un Entity que cumple con todos estos requerimientos de la siguiente manera:

```
@Entity(tableName = "recommendations_list")
data class RecommendationEntity(@ColumnInfo(name = "id")
                @PrimaryKey(autoGenerate = true)
                val id:Long = 0,
                @ColumnInfo(name = "recommendation_text")
                val recommendation:String)
```

Varias cosas debemos entender de este código. Primero, la clase es una data class, algo particular de Kotlin y que no es parte de Java. Este tipo de clases nos permiten definir modelos o POJOS de datos estructurados, también conocidos como objetos de datos. Estas clases son utilizadas como contenedores de información, y nos permiten mover la misma a través de los diferentes componentes de la aplicación, cuando los objetos transportan la información se conocen como DTO o Data Transfer Object.

Nuestro Entity está definido en una data class que contiene la información que nuestra aplicación necesita y almacenará en la base de datos. Entrando en el detalle de esta clase, tenemos los siguientes componentes:

- **Entity**: es una anotación de Room. Nos permite definir que la data class anotada es un Entity. Dentro de la anotación definimos que Entity almacenará la información en una tabla llamada “recommendations_list”. Está tabla tiene dos columnas. Para señalar que es una columna ocupamos la anotación que sigue. Cabe destacar que Entity puede inferir los nombres de las columnas, ocupando los nombres de las variables en la clase, pero cuando utilizamos camelCase, o deseamos un nombre específico, podemos definir el mismo dentro de la anotación

- **ColumnInfo.ColumnInfo**: Esta anotación nos permite definir que la variable anotada es una columna de la tabla antes nombrada. Dentro de la anotación definimos el nombre de la columna en la tabla. En este caso tenemos dos variables anotadas como columnas:
    - id: es el identificador de la fila en la tabla, definimos un valor inicial de 0 para el id.
    - recommendation_text: la columna que contiene texto de la recomendación.
- **PrimaryKey**: Esta anotación nos permite definir la clave primaria de cada fila en nuestra tabla. En este caso hemos definido id como clave primaria. También definimos que se autoincrementa dentro de la anotación.

De esta forma hemos definido todo lo que necesitamos en nuestra Entity. Notemos lo simple que ha resultado crear una tabla, con sus columnas, y los ids autoincrementales. Sigamos definiendo los otros dos componentes que nos faltan para completar este puzzle.

DAO

Para que nuestra aplicación ficticia pueda interactuar con la base de datos necesitamos un objeto que nos de acceso, y actúe como intermediario entre las tablas y los objetos. Este objeto, o clase, se llama DAO, o DataBase Access Object.

Este objeto se implementa como una interfaz, que luego es creado por Room, con el patrón Fachada, se explicará más adelante. Un DAO nos permite ejecutar diferentes interacciones contra nuestra base de datos. Normalmente nos permite realizar las operaciones CRUD sobre nuestra base de datos. Podemos Crear, escribir, actualizar y borrar una fila/tabla en la base de datos. También nos permite consultar por los datos almacenados, generar otras consultas, etc.

Para nuestro ejemplo necesitamos almacenar recomendaciones, insertar en la base de datos las que ingresamos manualmente. Además, debemos poder leer todas las recomendaciones almacenadas en la tabla. Para realizar estas operaciones en Room necesitamos un DAO. Implementar esto con Room es bastante sencillo. Para generar un DAO debemos utilizar una serie de anotaciones, veamos como quedaría un DAO en nuestro ejemplo:

```
@Dao
interface RecommendationsDAO {
 @Query("SELECT * FROM recommendations_list")
 fun getAllRecommendations(): List<RecommendationEntity>

 @Insert
 fun insertRecommendations(var recommendations:List<RecommendationEntity>)
}
```

Lo primero que se debe notar es que DAO es una interfaz y no una clase. Esto se debe a que Room ocupa un interesante patrón llamado Fachada. Este patrón nos permite definir la interfaz y dejar que la librería construya todo el código necesario para hacer funcionar esa interfaz, en este caso un DAO. Este comportamiento de Interfaz/Fachada es usado en muchas librerías populares, como Dagger o Retrofit, por lo cual Google decidió aprovechar la familiaridad que tiene la comunidad con el patrón y lo utilizó para construir Room.

Volvamos al código.

Comenzaremos mirando las anotaciones de la clase, la primera que vemos sobre la definición de la Interfaz es la anotación `@Dao`. Esta anotación le dice a Room que esta interfaz es un DAO a una base de datos. Además, le señala que debe implementar los métodos definidos en ella.

La segunda anotación que vemos en este código es `@Query`, la cual nos permite definir una query SQL que se ejecuta en la base de datos. Aquí podemos definir operaciones tan complejas como deseemos. La principal ventaja de esta anotación por sobre SQL puro es: las queries SQL en Room son revisadas en tiempo de compilación. Al realizar la revisión en la compilación, podemos saber de manera inmediata donde nos estamos equivocando, antes de correr la aplicación, o si nos falta algún parámetro en la consulta. Room construye la base de datos y cuando compila sabe de manera inmediata si la operación, o query, compila correctamente contra la base de datos definida.

Esto nos ahorra muchos dolores de cabeza, con SQL uno corre la aplicación y ejecutar la query en Runtime, esperando que estuviera correcta. Si la query falla, sólo lo sabremos después de correr la aplicación, con Room lo sabemos antes de correr la aplicación.

Dentro de la anotación Query en la primera función de este DAO definimos una consulta, en este caso la consulta es un *SELECT * FROM recommendations_list,* que nos permite obtener todas las filas almacenadas en la tabla recommendations list. Con esta lista podremos desplegar todas las recomendaciones al usuario.Este método no recibe parámetros, después de todo es un select *, nos retorna una lista de entidades RecommendationEntity, la cual puede ser utilizada directamente para alimentar la lista que implementaremos en el siguiente capítulo con un RecyclerView y un Adaptador. Por una decisión de diseño, no queremos exponer un Entity a la lista, por lo que definiremos otra clase para transportar esos datos, más sobre esto en el próximo capítulo.

Además Room cuando no pueda mapear directamente la data a una clase despliega un warning CURSOR_MISMATCH, y hace un mapeo parcial de la data. Es decir, incluso cuando no funciona 100%, Room hace su mejor esfuerzo para cumplir con nuestro requerimiento. En el siguiente capítulo aprenderemos a lidiar con un problema común en Room relacionado a los tipos de datos.

Siguiendo en el código, la segunda función es para insertar datos a la base de datos, en este caso debemos utilizar la anotación @Insert. Esta anotación le dice a Room que este método inserta data en la base de datos. Los parámetros de este tipo de métodos pueden ser sólo clases anotadas con `@Entity` o colecciones de dichas clases, en este caso tenemos una List de entities. Cómo Room conoce la base de datos, generará todo el SQL necesario para realizar este Insert en la base de datos.Sólo nos falta nuestro tercer componente para completar el puzzle, veamos cómo definir un Database.

Database

Ya definimos dos de nuestros tres componentes, tenemos un DAO y un Entity, necesitamos algo que las conecte. El componente que une a los otros dos es un Database. En este caso no necesitamos definir mucho, pero como referencia es interesante saber que podemos tener más de un DAO asociado a un database, lo cual nos permite granularizar más nuestro código y ordenar de manera más consciente. Para definir un database en Room necesitamos usar una anotación que es clave, además de extender una Clase de Room, veamos el código:

```
@Database(entities = [(RecommendationEntity::class)], version = 1)
abstract class RecommendationsDatabase: RoomDatabase() {
  abstract fun getRecommendationsDAO(): RecommendationsDAO
}
```

La anotación `@Database`, nos permite decirle a Room que esta clase es una base de datos. Dentro de la anotación definimos un arreglo de entities asociadas a esta base de datos. En la anotación también definimos qué versión de la base de datos estamos ocupando en esta base de datos, este n{umero está asociado a las diferentes migraciones que debemos hacer cuando modificamos nuestra base de datos, agregando entities o modificando las mismas. Veremos un ejemplo de migración en el próximo capítulo.

La clase y los métodos que definimos deben ser tipo abstract. nuestra clase debe heredar de RoomDatabase, de esta forma Room sabe que este es la clase que nos permite acceder a los DAOs disponibles en el database.

Cada método que se define retorna un DAO, en este caso sólo uno, por lo que definimos sólo un método.

Ya tenemos todo lo necesario para utilizar Room en nuestra aplicación, Antes de concluir este capítulo, veremos otras anotaciones importantes en Room. Estas anotaciones son las más comunes, pero no son todas las que uno encuentra en Room. Además, veremos un ejemplo simple de como migrar una base de datos con Room.

Otras Anotaciones y Migraciones en Room

Lo que hemos explorado hasta ahora es un set acotado de funcionalidades de Room. En esta sección revisaremos otras anotaciones, además del proceso de migración, que se vuelve útil cuando comenzamos a agregar más entidades y filas a la base de datos.

Primero veremos las anotaciones más comunes, el resto de las anotaciones pueden ser revisadas en la guía oficial de google disponible en https://developer.android.com/reference/android/arch/persistence/room/package-summary. En esta guía no sólo se encuentran la documentación, también tenemos ejemplos, lamentablemente la mayoría de ellos están en Java, esperemos que en un futuro cercano Google actualice todo a Kotlin.

@Update

Muchas veces no necesitamos agregar nuevos elementos a nuestra tabla, queremos actualizar lo que ya está en ella, para eso existe la anotación update. El siguiente método actualiza una entidad Address que recibe como parámetro. Si la entidad existe en la base de datos, se realiza la actualización de todos sus campos, salvo el la clave primaria. Si la entidad no existe, no se hace nada, la función se puede definir de la siguiente forma:

```
@Update
fun updateAddress(Address address)
```

Un problema típico con esta anotación es cuando una entidad tiene muchos campos y necesitamos actualizar uno sólo. En ese caso es más conveniente usar `@Query` con un Update en SQL, que una función con la anotación Update.

@Delete

Esta anotación nos permite borrar de la base de datos lo que recibe como parámetro, los parámetros siempre son entidades o colecciones de entidades. Una implementación de delete puede ser la siguente:

```
@Dao
interface UserDao {
     @Delete
     fun deleteUsers(vararg users:User)
     @Delete
     fun deleteWithFriends(user:User, friends:List<User> )
 }
```

Un problema que podemos tener con esta anotación es que no podemos borrar toda la tabla a menos que tengamos todas las entidades para pasar como parámetro. En el caso que queramos hacer un borrado total, se recomienda hacer una query SQL con un método anotado con `@SQL`.

@ForeignKey

Esta anotación nos permite utilizar una clave foránea desde otra entidad, por ejemplo si tenemos una entidad Persona y una entidad Mascota, sabemos que la mascota tiene dueño, podemos definir la relación entre mascota y persona de la siguiente forma:

```
@Entity
data class Person(
        @PrimaryKey(autoGenerate = true)
        val id: Long,
        val name: String = "",
        val sureName: String = "",
        val phone: String = ""
)

@Entity(foreignKeys = arrayOf(ForeignKey(entity = Person::class,
        parentColumns = arrayOf("id"),
        childColumns = arrayOf("ownerId"),
        onDelete = ForeignKey.CASCADE)))
data class Pet(
        @PrimaryKey(autoGenerate = true)
        val id: Long,
        val ownerId: Long,
        val name: String
)
```

Un Pet pertenece a una persona, el Entity Pet contiene una propiedad ownerId, que será asignada de la siguiente manera´en la anotación `@Entity`:

- Utilizamos foreignKeys para definir un arreglo de claves foráneas, en este caso definimos sólo una clave foránea.

- En el arreglo definimos ForeignKey con los siguientes parámetros
    - **entity**: la entidad de la cual viene la clave foránea, en este caso Person.
    - **parentColumn**: la columna que contiene el valor de la clave foránea en la entidad de origen, en este caso es el id de Person.
    - **childColumn**: la columna que almacena el valor de la clave foránea en la entidad de destino, en este caso ownerId.
    - **onDelete**: aquí definimos el comportamiento de la base de datos con respecto a la entidad de destino, cuando eliminamos la entidad de origen. Al definir ForeignKey.CASCADE la base de datos borrará en cascada las entidades, es decir si borramos al dueño se borran las mascotas. También podemos definir el comportamiento para onUpdate, que define qué se hace en caso de una actualización.

      Hay que tener cuidado con esta anotación, al igual que cuando se ocupa SQL. El comportamiento al eliminar puede ser desastroso para nuestra base de datos.Esta anotación nos permite generar relaciones entre entidades, al igual que Embedded y Relation.

@Embedded

Muchas veces tenemos objetos que contienen otros objetos, esto es la norma en OOP, pero no necesitamos almacenar ese objeto en la base de datos como entidad. Y cuando recuperamos esta información también queremos tener nuevamente ese objeto disponible para ocuparlo, como objeto. Para estos casos es útil la anotación `@Embeded`.

También es importante hacer notar que esta es una de las anotaciones que nos permite hacer relaciones en la base de datos, en este caso puede ser 1-1 o 1 - N, dependiendo cuantas direcciones tenga una persona.

Recordando nuestro caso de una lista de contactos, donde teníamos una persona y una dirección, podemos embeber la dirección como objeto dentro de persona, la anotamos con `@Embedded` y dejamos que Room maneje la referencia y el almacenamiento, veamos como queda esto en código:

```
@Entity
data class Person(
        ...// other fields
        @Embedded
        val address: Address
)

data class Address(val postcode: String,
                   val streetName: String,
                   val streetNumber:Long,
                   val city: String)
)
```

En este caso el objeto dirección se almacenará como campos que son parte de Person, no se crea una Entity Address. Pero cuando se rescate desde la base de datos un Person, se mapeará el objeto Address a un Objeto independiente, manteniendo la intención original del diseño, muy conveniente y útil para manejar datos en nuestra aplicación.

@Relation

Esta anotación también nos permite generar relaciones entre entidades, 1-1, 1-N, etc. Al utilizar esta anotación en un entity, Room restaura automáticamente todas las entidades involucradas, sin la necesidad de utilizar consultas adicionales, ahorrandonos tiempo y código.

Cosas importantes que debemos saber de Relation:

- Esta anotación se puede utilizar sólo en campos o propiedades que sean: Set o List. Esta anotación no se puede utilizar para propiedades que nos sean una lista o un set de datos
- Cada entidad involucrada debe tener declarada una clave primaria.
- Las claves primarias definidas en objetos embebidos no son consideradas primarias en el objeto que lo contiene.

Para entender mejor cómo funciona esta anotación veamos el siguiente ejemplo:

- Queremos definir la relación entre una Persona y todas sus mascotas.
- Podemos definir esta relación en una nueva Entity, llamada PersonAndAllHisPets.
- Esta entity mantiene la relación de un Person con la lista de todos sus Pets. Al hacer una consulta sobre esta nueva entidad en un DAO, podemos obtener una lista de personas, con todas sus mascotas en una lista.

Para hacer esto debemos utilizar la anotación @Relation, definir la nueva entidad y DAO. En el ejemplo la nueva entidad contiene un Person embebido, además tiene una lista de pets, veamos el código:

```
@Entity
class Person(
    @PrimaryKey
     val id:Int,
     // otros campos, esta entidad es la dueña de las mascotas)

@Entity
class Pet (
     @ PrimaryKey
     val id:Int,
     val userId:Int,
     val name:String,
     // otros campos, esta entidad pertenece a un dueño Person
)
```

```
@Entity //define la relación entre un Person y sus Pets
class PersonAndAllHisPets(
     @Embedded
     val person:Person,
     @Relation(parentColumn = “id”, entityColumn = “userId”)
     val pets:List<Pet>
 )

@Dao //nos permite consultar por todas las personas y sus mascotas
 interface PersonPetDao {
     @Query("SELECT * FROM person_table WHERE id=:personId")
     fun  loadPersoAndPets(personId:Int):List<PersonAndAllHisPets>
 }
```

Como podemos ver en el ejemplo, hemos creado un nuevo Entity, PersonAndAllHisPets. Lo interesante es que este nuevo entity nos permite rescatar directamente una lista mascotas asociadas a una persona en particular. En la anotación @Relation definimos la columna del dueño o padre que se relaciona con la columna de la entidad de destino, en este caso Pet. Esta relación nos permite generar el DAO con la función `loadPersonsAndPets()`, gracias a este dao podemos obtener todos las mascotas de la persona con el id = personId. Esto es realmente poderoso si pensamos lo simple que es generar esta nueva entidad y nuevo dao, para generar una consulta que normalmente se hace con un join en SQL. Podemos modelar todo tipo de relaciones con esta entidad, siempre y cuando tengamos una lista o set asociado, recordado la restricción para su uso.

@Ignore

Esta anotación nos permite definir campos en un objeto que no necesitamos guardar en la base de datos, por ejemplo un bitmap que representa un avatar o foto de identificación. Lo ocupamos de la siguiente manera

```
@Ignore
val avatar:Bitmap
```

Hay muchas más anotaciones disponibles para diferentes operaciones, pero estas son las más habituales y comunes al utilizar Room. Revisar la documentación oficial para otra anotaciones y propiedades https://developer.android.com/reference/android/arch/persistence/room/package-summary.

Migración de una base de datos con Room

A medida que agregamos nuevas funciones y componentes a una aplicación se nos presenta el siguiente problema:

Agregué nuevas entidades y campos a mi base de datos por lo que necesito migrar el esquema a una versión más nueva. Para realizar esto debemos escribir clases de migración, para decirle explícitamente a Room que hacer con cada versión de nuestro esquema.

La migración permite que los usuario actuales de la aplicación no pierdan sus datos cuando actualicen a la nueva versión. Antes de que migremos una base de datos necesitamos una versión del esquema de la base de datos exportada en nuestra aplicación. Para hacer esto necesitamos agregar el siguiente código a nuestro gradle file del módulo, en la configuración por defecto.

```
defaultConfig {
   applicationId "cl.desafiolatam.roomexample"
   minSdkVersion 23
   targetSdkVersion 29
   versionCode 1
   versionName "1.0"
   testInstrumentationRunner  "androidx.test.runner.AndroidJUnitRunner"
   javaCompileOptions { //esto exporta el esquema
       annotationProcessorOptions {
           arguments = ["room.schemaLocation":
                                "$projectDir/schemas".toString()]
       }
   }
}
```

Ya tenemos nuestro esquema, para migrar una base de datos Room necesita clases que extienden la clase migración. Estas clases debe tener una versión de inicio y una versión final. Para poder ejecutar las migraciones necesarias, en el orden correspondiente. Las clases deben ejecutarse en orden consecutivo, de esta forma se lleva la base de datos a su versión más nueva sin perder los datos almacenados.

Migration es el nombre de la clase que Room ofrece para generar una migración, normalmente estas clases migración se implementan como clases estáticas. Para generar una migración en Kotlin debemos hacer lo siguiente:

```
val MIGRATION_1_2 = object : Migration(1, 2) {
    override fun migrate(database: SupportSQLiteDatabase) {
        database.execSQL("CREATE TABLE `Fruit` (`id` INTEGER, `name` TEXT, "
                         + "PRIMARY KEY(`id`))")
    }
}
val MIGRATION_2_3 = object : Migration(2, 3) {
    override fun migrate(database: SupportSQLiteDatabase) {
        database.execSQL("ALTER TABLE Book ADD COLUMN pub_year INTEGER")
    }
}
Room.databaseBuilder(applicationContext, MyDb::class.java, "database-name")
        .addMigrations(MIGRATION_1_2, MIGRATION_2_3).build()
```

En este ejemplo se definen dos migraciones, la primera migración lleva la base de datos desde la versión 1 a la versión 2, inicializando un objeto de la clase Migration con los parámetros 1 y 2.

```
 val MIGRATION_1_2 = object : Migration(1, 2)
```

dentro del cuerpo de esta clase se debe sobreescribir la función migrate, que recibe como parámetro un database

```
override fun migrate(database: SupportSQLiteDatabase)
```

Este database, que recibe la función migrate, debe ejecutar el SQL necesario para modificar el esquema actual y llevarlo al nuevo esquema, en este caso se ejecuta un Create table:

```
database.execSQL("CREATE TABLE `Fruit` (`id` INTEGER, `name` TEXT, " +
"PRIMARY KEY(`id`))")
```

En este método podemos ejecutar todas las queries y sentencias SQL que sean necesarias para actualizar la base de datos a la versión de destino.

Finamente todas las migraciones se ejecutan al construir la base de datos con el databaseBuilder de Room. La operación en este caso es la siguiente:

```
Room.databaseBuilder(applicationContext,
MyDb::class.java,
"database-name")
.addMigrations(MIGRATION_1_2, MIGRATION_2_3).build()
```

Así aseguramos la completa compatibilidad de nuestra nueva base de datos con las versiones más antiguas, y el usuario no pierde datos al actualizar la aplicación.

Creando una aplicación con Android Room

Competencias

- Desarrollar una aplicación completa en Android con Room, creando todo lo necesario para que funcione.
- Entender cómo se convierten los datos cuando debemos lidiar con campos que no son compatibles con SQL, por ejemplo un Date.
- Entender por qué es necesario ejecutar las operaciones con Room en un Thread distinto al principal en Android.

Motivación

Ya conocemos la teoría de que necesitamos y cómo funciona Room. En este capítulo completamos una aplicación simple, que nos permite ver en acción la librería. Además, veremos como solucionar algunos problemas que se pueden presentar al utilizar Room, pondremos especial énfasis en TypeConverter y cómo convertir de un tipo a otro, y devuelta.

Continuando con la aplicación del capítulo anterior

En este capítulo completamos la aplicación que empezamos a realizar en el capítulo anterior. En este capítulo crearemos rápidamente le proyecto con lo necesario para desplegar la lista, agregar tareas, y completar las mismas.

Creación del Proyecto

Seguiremos los mismos pasos que hicimos en el capítulo 2 para crear el proyecto de SharedPreferences, pero esta vez seleccionamos Basic Activity como proyecto inicial. Luego de que se ejecuten todos las sincronizaciones, configuramos todo lo descrito en el capítulo anterior. Incluimos la librería al proyecto, los plugins de kotlin, y el código para exportar el esquema. Sincronizamos gradle y estamos listos para continuar.

Creamos un paquete en el código con el nombre room, y creamos las clases Entity, DAO y Database que describimos en el capítulo anterior. Se puede copiar y pegar directamente el código. El resultado final de estos pasos se debe ver así en el proyecto:

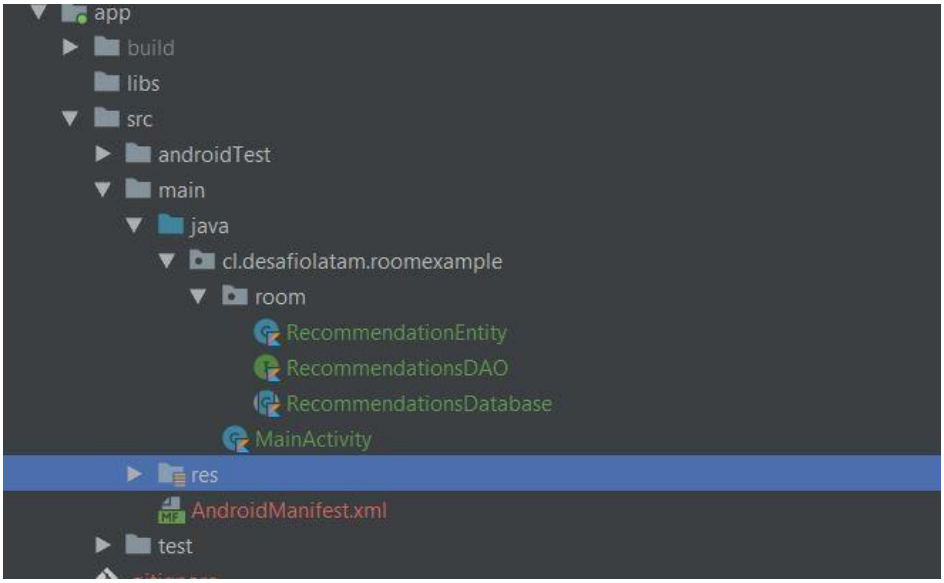


Imagen 4. Creamos los paquetes para Entity, DAO y Database.

Necesitamos configurar algunas otras cosas para poder utilizar completamente Room en el proyecto. Debemos crear una clase Application.

Clase Application y como crear una instancia de nuestra Base de Datos

Para utilizar la base de datos debemos crear una instancia de la misma, en la aplicación. Para crear la instancia debemos considerar que este proceso consume muchos recursos, por lo cual se recomienda utilizar un Singleton. Un Singleton nos permite tener una instancia global que se comparte a través de la aplicación, de esa forma no tenemos que crear una nueva instancia cada vez que necesitamos usarla. Esta instancia Singleton debe ser creada, o inicializada, en una clase que extienda Application, o alguna de sus variantes en Android. En este caso utilizaremos Application para crear lo que necesitamos para inicializar nuestra base de datos. La clase la creamos en el paquete principal del proyecto, y agregamos el siguiente código:

```
class RoomApplication: Application() {
   companion object {
       var recommendationsDatabase:RecommendationsDatabase? = null
   }

   override fun onCreate() {
       super.onCreate()
       recommendationsDatabase = Room
           .databaseBuilder(this,
               RecommendationsDatabase::class.java,
               "recommendations-master-db").build()
   }
}
```

Esta clase extiende o hereda de Application, que representa la clase aplicación en Android. Si nos falta alguna referencia podemos presionar alt + Enter en windows o option + Enter en Mac, para que Android Studio agregue la dependencia faltante.Debemos definir una instancia única, un Singleton de la base de datos. Para hacer esto en Kotlin debemos utilizar un companion object. Este companion object es un objeto que será compartido por todas las posibles instancias de esta clase, en este caso hay sólo una. Todo lo que se encuentra dentro de un companion object es compartido entre las diferentes instancias y puede ser accedido como los métodos o propiedades estáticas en Java. En este caso definimos nuestra variable database, y la inicializamos en el método onCreate de la clase. La variable database es de tipo RecommendationsDatabase?, el modificador ? significa que puede ser null, por eso la inicializamos null, para luego inicializarla una sóla vez. El método onCreate es llamado sólo cuando la clase Application es creada, esto ocurre una sóla vez al correr nuestra Aplicación. Tener una variable dentro de un companion object es el equivalente a declarar una variable static en Java.

Al crear la base de datos utilizamos el databaseBuilder de Room con tres parámetros:

- un Context, en este caso usamos this para pasar la instancia de Application.
- la clase que representa y está anotada con @Database, en este caso RecommendationsDatabase::class.java.
- un nombre en formato String, este es el nombre del archivo que será almacenado en la aplicación con nuestra base de datos y los datos almacenados en ella.

  Nos falta sólo un paso para terminar con la configuración, debemos agregar la clase aplicación al manifest.

Agregando Application al Manifest

Para configurar la aplicación que hemos creado en el manifest, lo hacemos con la siguiente modificación:

- agregamos la línea android:name, dentro del tag application, como se ve a continuación.

```
<application
       android:name=".RoomApplication"
       android:allowBackup="true"
       android:icon="@mipmap/ic_launcher"
       android:label="@string/app_name"
       android:roundIcon="@mipmap/ic_launcher_round"
       android:supportsRtl="true"
       android:theme="@style/AppTheme">
```

Luego de esto estamos listos para construir esta aplicación. Veremos una pequeña explicación para entender que necesitamos a nivel de Interfaz de Usuario.

Aplicación Libreta de Recomendaciones

Para utilizar la base de datos que definimos anteriormente, construimos una sencilla aplicación que permite almacenar recomendaciones, que son desplegadas en una lista y almacenadas en una base de datos. En este diagrama podemos observar como está diseñada la interacción de los distintos componentes de la aplicación.

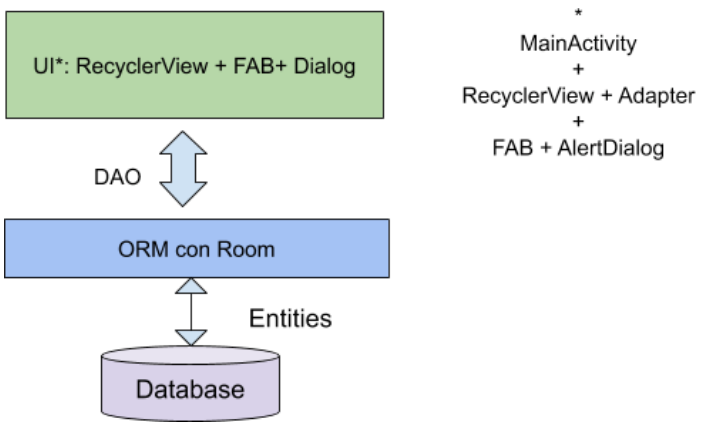


Imagen 5. Integración de los distintos componenentes en la aplicación.

Los componentes principales de la aplicación son:

- **ORM**: definido en los componentes de Room, y que ya hemos construido
- **MainActivity**: la pantalla principal que tiene una lista implementada con un RecyclerView más un Adaptador. Se crea también una dataclass que contiene los datos obtenidos de la base de dato. Este contenedor específico es utilizado con el adaptador del recyclerview. La idea es no exponer a capas superiores las entities, esto es una buena práctica. Por ejemplo si decidimos cambiar las entities, pero no cambiamos la información que se muestra, sólo debemos cambiar la función o método que mapea la información de las entities a los dataclass de pantalla, evitando tener que cambiar mucho código, facilitando la extensión y mantención de la aplicación.
- Cuatro **layouts**, en xml, que representan:
    - el AlertDialog que se utiliza para ingresar recomendaciones de a una.
    - Un layout que contiene la pantalla principal, que a su vez contiene otro layout.
    - Un layout con el RecyclerView que representa la lista que desplegamos, además de contener el FAB
    - Un layout que representa una fila de la lista

Comenzaremos construyendo los xml que necesitamos.

Creación de XML de la aplicación

El XML que representa la pantalla principal es el siguiente:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.coordinatorlayout.widget.CoordinatorLayout
       xmlns:android="http://schemas.android.com/apk/res/android"
       xmlns:app="http://schemas.android.com/apk/res-auto"
       xmlns:tools="http://schemas.android.com/tools"
       android:layout_width="match_parent"
       android:layout_height="match_parent"
       tools:context=".MainActivity">
   <include layout="@layout/content_main"/>
   <com.google.android.material.floatingactionbutton.FloatingActionButton
           android:id="@+id/fab"
           android:layout_width="wrap_content"
           android:layout_height="wrap_content"
           android:layout_gravity="bottom|end"
           android:layout_margin="@dimen/fab_margin"
           app:srcCompat="@drawable/add_plus_white"/>
</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

Podemos ver en este xml:

- Un CoordinatorLayout, que coordina las interacciones en la pantalla, este fue seleccionado por defecto, y se utiliza para generar animaciones en la interfaz. Además de coordinar otras interacciones, no necesitamos saber mucho más sobre este componente.
- Tenemos un include, que nos permite reutilizar otros xml, al incluirlos como parte de este. El include en este caso agrega content_main, que es la pieza que contiene el recyclerview que mostrará la lista.
- Finalmente tenemos un FloatingActionButton que será el nos permite agregar recomendaciones a nuestra lista.
- Además, definimos un drawable que representa el símbolo + en el botón fab.

El drawable se define así

```
<?xml version="1.0" encoding="utf-8"?>
<vector  xmlns:android="http://schemas.android.com/apk/res/android"
                android:width="24dp"
                android:height="24dp"
                android:viewportWidth="24.0"
                android:viewportHeight="24.0">
<path android:fillColor="#FFFFFFFF"
      android:pathData="M19,13h-6v6h-2v-6H5v-2h6V5h2v6h6v2z"/>
</vector>
```

El segundo Xml que debemos definir es content_main, lo definimos de la siguiente forma:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
       xmlns:android="http://schemas.android.com/apk/res/android"
       xmlns:tools="http://schemas.android.com/tools"
       xmlns:app="http://schemas.android.com/apk/res-auto"
       android:layout_width="match_parent"
       android:layout_height="match_parent"
       app:layout_behavior="@string/appbar_scrolling_view_behavior"
       tools:showIn="@layout/activity_main"
       tools:context=".MainActivity">
   <androidx.recyclerview.widget.RecyclerView
           android:layout_width="match_parent"
           android:layout_height="match_parent"
           app:layout_constraintTop_toTopOf="parent"
           app:layout_constraintEnd_toEndOf="parent"
           app:layout_constraintBottom_toBottomOf="parent"
           app:layout_constraintStart_toStartOf="parent"
           android:id="@+id/recommendations_list"/>
</androidx.constraintlayout.widget.ConstraintLayout>
```

Lo más importante de este xml es el RecyclerView que nos permitirá mostrar la lista de recomendaciones.

Además debemos crear un xml que represente en pantalla el texto, para esto creamos un xml llamado recommendation_item, este xml queda de la siguiente manera

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
       xmlns:android="http://schemas.android.com/apk/res/android"
       xmlns:app="http://schemas.android.com/apk/res-auto"
       android:orientation="vertical"
       android:layout_width="match_parent"
       android:layout_height="wrap_content">
   <TextView
           android:layout_width="0dp"
           android:layout_height="wrap_content"
           android:id="@+id/recommendationBox"
           app:layout_constraintEnd_toEndOf="parent"
           android:layout_marginEnd="8dp"
           app:layout_constraintStart_toStartOf="parent"
           android:layout_marginStart="8dp"
           android:layout_marginTop="8dp"
           app:layout_constraintTop_toTopOf="parent"
           android:textSize="18sp"
           app:layout_constraintBottom_toBottomOf="parent"
           android:layout_marginBottom="8dp"/>
</androidx.constraintlayout.widget.ConstraintLayout>
```

Para agregar las diferentes recomendaciones utilizaremos un AlertDialog. Debemos crear un xml custom para este caso. El xml que utilizaremos es el siguiente, add_recommendation_layout.xml:

```xml
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
       xmlns:android="http://schemas.android.com/apk/res/android"
       xmlns:app="http://schemas.android.com/apk/res-auto"
       android:orientation="vertical"
       android:layout_width="match_parent"
       android:layout_height="wrap_content">

   <com.google.android.material.textfield.TextInputLayout
           android:layout_width="0dp"
           android:layout_height="wrap_content"
           android:layout_marginTop="8dp"
           app:layout_constraintTop_toTopOf="parent"
           android:layout_marginBottom="8dp"
           app:layout_constraintBottom_toBottomOf="parent"
           app:layout_constraintStart_toStartOf="parent"
           android:layout_marginStart="8dp"
           app:layout_constraintEnd_toEndOf="parent"
           android:layout_marginEnd="8dp"
           android:id="@+id/recommendation_input_layout">

       <com.google.android.material.textfield.TextInputEditText
               android:layout_width="match_parent"
               android:layout_height="wrap_content"
               android:hint="@string/agrega_hint"
               android:id="@+id/recommendation_input"/>
   </com.google.android.material.textfield.TextInputLayout>
</androidx.constraintlayout.widget.ConstraintLayout>
```

con este xml completamos todo lo necesario para mostrar y agregar recomendaciones a nivel visual, veamos ahora cómo pegamos la base de datos y la interfaz.

Creando el código para unir la interfaz con los datos

Como dijimos anteriormente, ya tenemos toda la base de nuestra base de datos y UI, ahora debemos construir el código para unir las dos capas, creamos un nuevo paquete llamado list, para mantener el orden. Lo primero que construiremos en este nuevo paquete es un contenedor de los datos que rescatamos de la base de datos. Esto es opcional, pero como no queremos exponer los Entities, creamos este pequeño data class llamado RecommendationDataView, el código a continuación:

```
data class RecommendationDataView(
var id:Long,
var text:String)
```

Para usar un recyclerview debemos construir dos componentes claves, su descripción a continuación:

- Un Adapter: que nos permite crear las vistas del RecyclerView, mantiene una lista de los datos y genera las vistas de cada fila.
- Un ViewHolder, que es el contenedor visual de los datos. Este viewholder representa visualmente una fila de la lista y está asociado directamente al adaptador.

La forma en que RecyclerView implementa las listas es eficiente y práctica, reduciendo el uso de memoria y facilitando nuestro trabajo. Lo primero que definiremos es el ViewHolder. El código a continuación:

```
class RecommendationViewHolder(view: View): RecyclerView.ViewHolder(view) {
val recommendationCheck = view.findViewById<TextView>
(R.id.recommendation_text)
}
```

El adaptador se define los métodos por defecto que debe definir cada clase que extiende RecyclerView.Adapter. Además, se agrega un método para actualizar la data de manera general, reemplazando la data presente en el adaptador y notificar que el set de datos ha cambiado, para refrescar la lista en pantalla, el código que define este adaptador es el siguiente:

```
class RecommendationsAdapter(
   private var recommendationsList: MutableList<RecommendationDataView>,
   var context: Context
): RecyclerView.Adapter<RecommendationViewHolder>() {

   override fun onCreateViewHolder(parent: ViewGroup, viewType:Int)
           :RecommendationViewHolder {
       return RecommendationViewHolder(
           LayoutInflater.from(context)
               .inflate(R.layout.recommendation_item, parent, false))
   }
   override fun getItemCount(): Int = recommendationsList.size

   override fun onBindViewHolder(holder: RecommendationViewHolder, position:
Int) {
       val data = recommendationsList[position]
       holder.recommendationText.text = data.text
   }

   fun updateData(items:List<RecommendationDataView>) {
//hace update de la data cuando el usuario agrega nuevas recomendaciones
       recommendationsList.clear()
       recommendationsList.addAll(items)
       notifyDataSetChanged()
   }
}
```

Para finalizar, la aplicación despliega su MainActivity, en la cual se definen las interacciones y las diferentes variables. La Actividad es muy grande para incluirla completamente en este documento, pero se incluyen los métodos principalesLo primero es definir MainActivity como una clase que extiende AppCompatActivity, y sus propiedades:

```
class MainActivity : AppCompatActivity() {
   private lateinit var list: RecyclerView
   private lateinit var adapter:RecommendationsAdapter
   private lateinit var dataBase:RecommendationsDatabase
   private lateinit var dao: RecommendationsDAO
   private lateinit var addButton:FloatingActionButton
```

En el método onCreate de la actividad inicializamos todas las propiedades de esta Actividad, el método onCreate queda definido de la siguiente manera, todo lo creado antes de setUpViews se crea automáticamente al crear el main activity. El modificador !! es para asegurar que la variable no es nunca null en este momento de la ejecución de la aplicación:

```
override fun onCreate(savedInstanceState: Bundle?) {
   super.onCreate(savedInstanceState)
   setContentView(R.layout.activity_main)
   setSupportActionBar(toolbar)
   setUpViews()
   dataBase = RoomApplication.recommendationsDatabase!!
   dao = dataBase.getRecommendationsDAO()
   setUpAddButton()
}
```

El método setUpViews está definido de la siguiente forma, tener claro que mutableListOf() crea una lista modificable vacía, es como hacer new ArrayList() en Java:

```
private fun setUpViews() {
   list = recommendations_list
   list.layoutManager = LinearLayoutManager(this)
   addButton = fab
   adapter = RecommendationsAdapter( mutableListOf(), this)
   list.adapter = adapter //el adapter de la derecha es la propiedad
//de la actividad, el de la izquierda es la propiedad del adapter
}
```

El método setUpAddButton está definido de la siguiente manera:

```
private fun setUpAddButton() {
   addButton.setOnClickListener {
   val dialogView = layoutInflater
                    .inflate(R.layout.add_recommendation_layout, null)
   val recommendationText = dialogView.recommendation_input
   val dialogBuilder = AlertDialog
           .Builder(this)
           .setTitle("Agrega una recomendación")
           .setView(dialogView)
           .setNegativeButton("Cerrar") {
                   dialog:DialogInterface, _: Int -> dialog.dismiss()}
      .setPositiveButton("Agregar") {dialog:DialogInterface, _: Int ->
      if (recommendationText.text?.isNotEmpty()!!) {
        AsyncTask.execute {
    dao.insertRecommendations(createEntity(recommendationText.text.toString()))
    val newItems = createEntityListFromDatabase(dao.getAllRecommendations())
    runOnUiThread {
                    adapter.updateData(newItems)
                    dialog.dismiss()
                    }
                  }
               }
           }
       dialogBuilder.create().show()
   }
}
```

Acá se define la interacción que nos permite agregar recomendaciones a la base de datos. El addButton despliega un AlertDialog cuando es presionado.

Dentro del lambda que define el click listener del botón pasan varias cosas:

- Creamos la vista con el xml definido, add_recommendation_layout
- Creamos un TextInput para obtener el texto ingresado por el usuario
- Creamos el AlertDialog con un builder
    - Agregamos el título
    - Agregamos la vista que creamos
    - Agregamos la acción del botón negativo
    - Agregamos la acción del botón positivo, este botón ejecuta un AsycnTask en background. Los AsyncTask se deben ejecutar en un thread diferente al de la interfaz. En Android está prohibido ejecutar cálculos o tareas pesadas en el Thread principal, este thread maneja la interfaz. Si ejecutamos tareas en este thread, bloqueamos la interfaz de usuario y eso es un pecado mortal en Android. El AsyncTask ejecuta las siguientes operaciones
        - Inserta la nueva recomendación en la base de datos con el Dao
        - Rescata todos los elementos de la base de datos con el Dao
        - Finalmente vuelve al thread de UI para ejecutar un update de los datos en el adaptador y cerrar el dialog
- La última línea crea y muestra el AlertDialog cuando el addButton es presionado.

Tenemos dos pequeñas clases que nos ayudan a mapear los datos desde la entidad a el dataclass que se usa para desplegar información en la pantalla, y otra para crear la entidad a almacenar en la base de datos. Ambas funciones se ven a continuación:

```
private fun createEntity(text:String):RecommendationEntity {
   return RecommendationEntity(recommendation = text, isCheck = false)
}

private fun createEntityListFromDatabase(entities:List<RecommendationEntity>):
MutableList<RecommendationDataView> {
   val dataList = mutableListOf<RecommendationDataView>()
   if (entities.isNotEmpty()) {
       for (entity in entities) {
           val dataView = RecommendationDataView(
                               entity.id,
                               entity.recommendation,
                               entity.isCheck)
           dataList.add(dataView)
       }
   }
   return dataList
}
```

En la función createEntity creamos un Entity con parámetros nombrados, así le decimos explícitamente qué parámetro estamos ocupando. Uniendo todos estos componentes nuestra aplicación funciona correctamente, persistiendo los datos entre instancias y desplegando todo lo que se ha agregado en la lista principal. Algunas imágenes de la aplicación

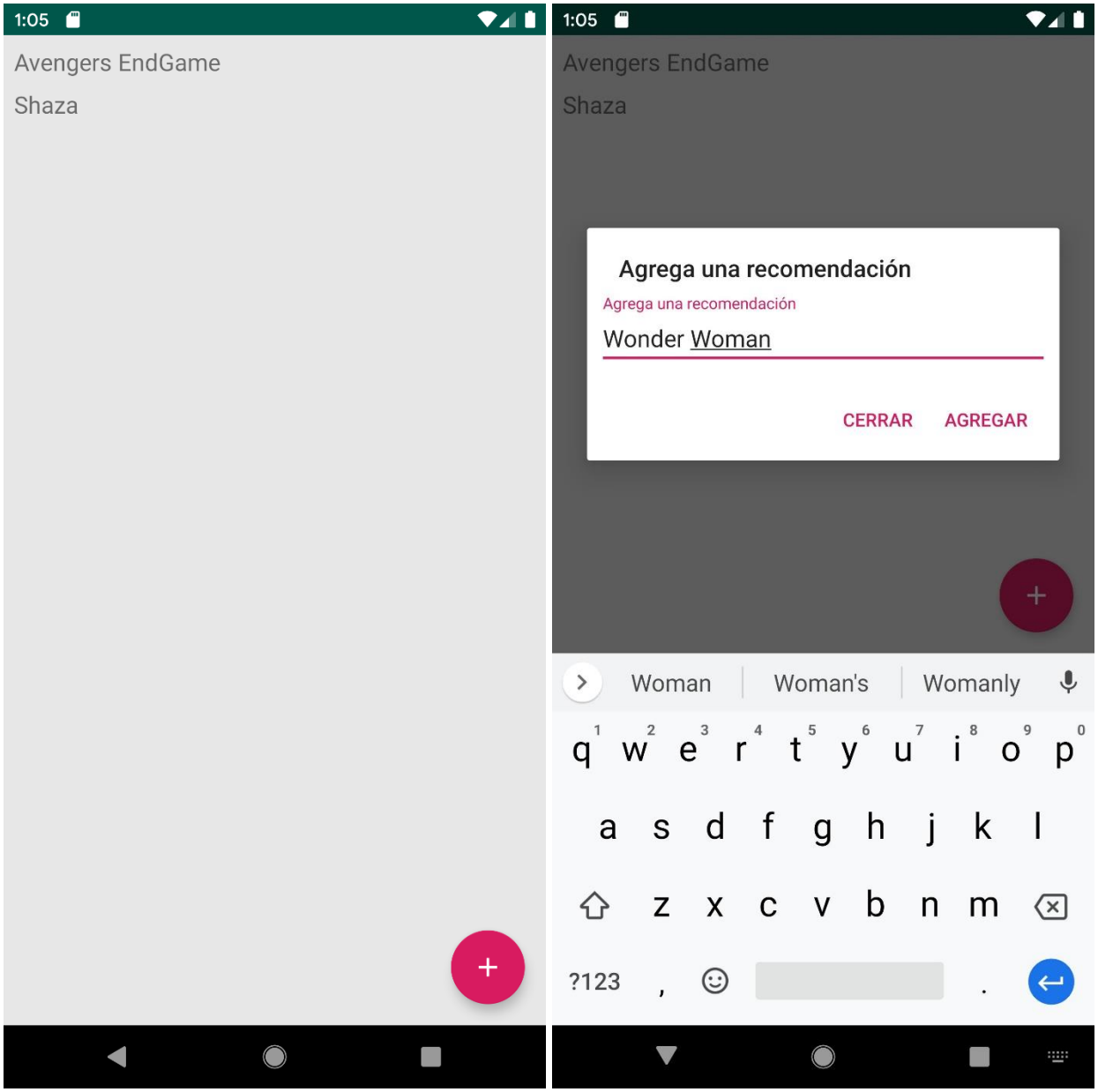


Imagen 6. Imágenes de la aplicación.

Imagen 7

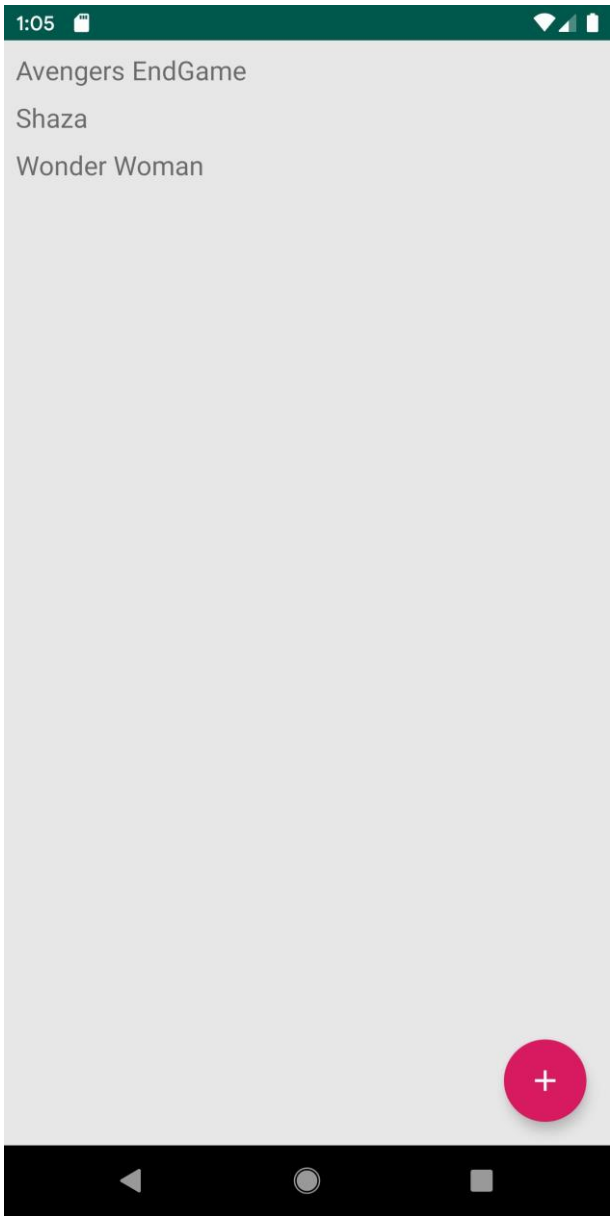


Imagen 8. Imágenes de la aplicación.

Nuestra aplicación funciona, pero qué ocurre si quiero agregar la fecha en que agregué las recomendaciones y mostrarlas. Esto nos crea dos problemas que resolveremos a continuación.

Manejo de datos no compatibles con SQL, TypeConverters

Queremos agregar la fecha en que se agrega cada recomendación y mostrarla en pantalla. Esto debería ser simple, ¿cierto?. Dos problemas ocurren cuando queremos hacer esto, y ya hemos corrido nuestra aplicación.

- El primer problema es que nuestro esquema ya existe, y si queremos agregar un campo a la tabla, necesitamos hacer una migración. Ya veremos como se soluciona ese problema.
- El segundo problema es que SQLite no puede almacenar Date, u otro objeto complejo, no es un tipo de dato compatible. Entonces, ¿cómo solucionamos este problema?, con un TypeConverter.

TypeConverter

Room nos permite almacenar tipos de datos primitivos como enteros, decimales, longs, texto, etc. Para poder guardar tipos de datos más complejos, debemos realizar una transformación de ese dato cuando lo almacenamos, y cuando lo rescatamos de la base de datos. Para este propósito tenemos disponible la anotación @TypeConverter. Esta anotación nos permite señalar que hemos definido una función que transformará de un tipo a otro, esto se hace ejecutando el código en la función.

En nuestro caso queremos agregar la fecha en que se agrega cada recomendación para lo cual haremos la siguiente modificación a nuestro Entity:

```
@Entity(tableName = "recommendations_list")
data class RecommendationEntity(@ColumnInfo(name = "id")
                                @PrimaryKey(autoGenerate = true)
                                val id:Long = 0,
                                @ColumnInfo(name = "recommendation_text")
                                val recommendation:String,
                                @ColumnInfo(name = "created_at")
                                val createdAt: Date?)
```

Agregamos la propiedad createAt, de tipo Date. Para que Room pueda persistir este nuevo campo debemos crear un TypeConverter. Para hacer esto debemos crear una clase que contenga las funciones para convertir los tipos de datos. Convertiremos el tipo Date en un UnitTimeStamp, esto se representa con un Long. Desde este Long crearemos un date cuando necesitemos hacer la conversión contraria. Definimos una clase llamada RecommendationsConverter, el código para esta clase es el siguiente:

```
class RecommendationsConverter {
   @TypeConverter
   fun fechaDesdeTimestampLong(value: Long?): Date? {
       return value?.let { Date(it) }
   }

   @TypeConverter
   fun longTimeStampDesdeDate(date: Date?): Long? {
       return date?.time
   }
}
```

Hemos creado dos funciones, estas funciones hacen lo siguiente:

- **fechaDesdeTimeStampLong**: crea un Date desde el Long que tenemos almacenado en la base de datos. Esta función ocupa let para crear el objeto Date, siempre y cuando exista el Long. De esta forma no debemos preocuparnos de posibles NullPointers u otra excepción.
- **longTimeStampDesdeDate**: crea un Long desde un Date, que se almacena en la base de datos. En este caso esto es simple debido a que Date de java tiene un método time que nos devuelve la representación en Long de la fecha en cuestión.Para que esta clase sea utilizada por nuestra base de datos debemos configurar la clase que convierte data en nuestra clase que contiene la base de datos. Para hacer esto, debemos agregar la anotación TypeConverte a la base de datos de la siguiente manera:

```
@Database(entities = [(RecommendationEntity::class)], version = 1)
@TypeConverters(RecommendationsConverter::class)
abstract class RecommendationsDatabase: RoomDatabase() {
   abstract fun getRecommendationsDAO(): RecommendationsDAO
}
```

Al agregar la anotación @TypeConverters, le decimos a este database que todos sus componentes pueden ocupar esta clase TypeConverter. Podemos definir un alcance menor a quién puede ocupar el TypeConverter de la siguiente forma:

- Si le agregamos la anotación a un Dao, todas las funciones de ese Dao pueden ocupar la clase.
- Si le agregamos la anotación a un método del Dao, sólo ese método puede ocupar la clase.
- Si le agregamos la anotación a un Entity, todos los campos de ese entity pueden ocupar la clase.
- Si le agregamos la anotación a un campo del Entity, sólo ese campo puede ocupar la clase.

Con esto ya hemos solucionado nuestro primer problema, podemos convertir nuestro nuevo campo para almacenarlo en la base de datos, y lo podemos convertir devuelta cuando lo necesitamos. Veamos cómo resolver nuestro problema de migración.

Migrando la base de datos

Necesitamos migrar nuestra base de datos para que acepte este nuevo campo. Debemos implementar la clase que conocimos en el capítulo anterior, Migration. Debemos agregar una nueva columna a nuestra tabla. En este caso utilizaremos el siguiente código, que se encuentra definido en un archivo llamado RecommendationsMigration.kt:

```
val MIGRATION_1_2:Migration = object : Migration(1,2) {
   override fun migrate(database: SupportSQLiteDatabase) {
       database.execSQL("ALTER TABLE recommendations_list " +
               "ADD COLUMN created_at INTEGER NOT NULL")
   }
}
```

Primero, notar que no es necesario definir una clase para definir nuestra clase Migration, lo podemos hacer en un Archivo .kt, esta es una de las ventajas de Kotlin.

Segundo, para definir la clase anónima utilizamos object:Migration(1,2). Al hacer esto le estamos diciendo a Room que esta clase Migration lleva la base de datos de la versión 1 a la 2.

Tercero, ejecutamos una sola query SQL, en este caso un Alter table, con el que agregamos a la tabla recommendations_list una nueva columna llamada created_at de tipo INTEGER, ya que SQLite no maneja LONGs. También le decimos a la base de datos que esta nueva columna es NOT NULL. Con estologramos agregar lo que necesitamos para migrar la base de datos a la versión 2.

Debemos actualizar la versión en la base de datos por lo que modificamos el código de la siguiente manera:

```
@Database(entities = [(RecommendationEntity::class)], version = 2, exportSchema
=   true)
@TypeConverters(RecommendationsConverter::class)
abstract class RecommendationsDatabase: RoomDatabase() {
   abstract fun getRecommendationsDAO(): RecommendationsDAO
}
```

Cambiamos la versión a 2, y agregamos el exportSchema a la anotación, de esta forma nuestro esquema aparecerá en la carpeta schemas de nuestro proyecto.

Para utilizar esta clase Migration la debemos agregar cuando inicializamos la base de datos. Recordemos que la inicialización de la base de datos se hace el método onCreate de la clase que extiende de Application. Debemos modificar el código de la siguiente manera:

```
override fun onCreate() {
   super.onCreate()
   recommendationsDatabase = Room
       .databaseBuilder(this,
           RecommendationsDatabase::class.java,
           "recommendations-master-db")
       .addMigrations(MIGRATION_1_2) // agregamos esta nueva operación para
migrar
       .build()
}
```

Al agregar la operación addMigrations le decimos a Room que actualice la base de datos al inicializarla. Con esto hemos migrado correctamente nuestra base de datos. Agreguemos lo que nos falta para poder mostrar la fecha de creación en Pantalla.

Modificando nuestro Adapter, ViewHolder y layout

Necesitamos hacer algunas modificaciones para mostrar la fecha en nuestra lista, lo primero que modificaremos es recommendation_item. Agregamos un nuevo textview con id creation_date. El layout, queda de la siguiente forma:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
       xmlns:android="http://schemas.android.com/apk/res/android"
       xmlns:app="http://schemas.android.com/apk/res-auto"
       android:orientation="vertical"
       android:layout_width="match_parent"
       android:layout_height="wrap_content">
   <TextView
           android:layout_width="0dp"
           android:layout_height="wrap_content"
           android:id="@+id/recommendation_text"
           app:layout_constraintEnd_toEndOf="parent"
           android:layout_marginEnd="8dp"
           app:layout_constraintStart_toStartOf="parent"
           android:layout_marginStart="8dp"
           android:layout_marginTop="8dp"
           app:layout_constraintTop_toTopOf="parent"
           android:textSize="18sp"
   />
   <TextView
           android:layout_width="0dp"
           android:layout_height="wrap_content"
           android:id="@+id/creation_date"
           android:layout_marginTop="10dp"
           app:layout_constraintTop_toBottomOf="@+id/recommendation_text"
           app:layout_constraintStart_toStartOf="parent"
           android:layout_marginStart="8dp"
           app:layout_constraintEnd_toEndOf="parent"
           android:layout_marginBottom="8dp"
           app:layout_constraintBottom_toBottomOf="parent"/>
</androidx.constraintlayout.widget.ConstraintLayout>
```

La siguiente modificación que necesitamos hacer es en nuestro ViewHolder. Agregamos el textview nuevo que mostrará la fecha, para agregarlo modificamos nuestro ViewHolder de la siguiente manera:

```
class RecommendationViewHolder(view: View): RecyclerView.ViewHolder(view) {
   val recommendationText: TextView =
view.findViewById(R.id.recommendation_text)
   val dateCreation: TextView = view.findViewById(R.id.creation_date)
}
```

Debemos agregar la nueva propiedad a nuestro DataView, lo hacemos de la siguiente manera:

```
data class RecommendationDataView(
var id:Long,
var text:String,
var creationDate: Date
)
```

Modificamos nuestro adapter agregando lo necesario para mostrar la fecha en el método onBindViewHolder, el código queda de la siguiente manera:

```
override fun onBindViewHolder(holder: RecommendationViewHolder, position: Int)
{
   val data = recommendationsList[position]
   holder.recommendationText.text = data.text
   //Formato para parsear date de creación
 val dateFormater = SimpleDateFormat("EEE, d MMM yyyy HH:mm",
Locale.getDefault())
   val dateString = dateFormater.format(data.creationDate)
   holder.dateCreation.text = dateString
}
```

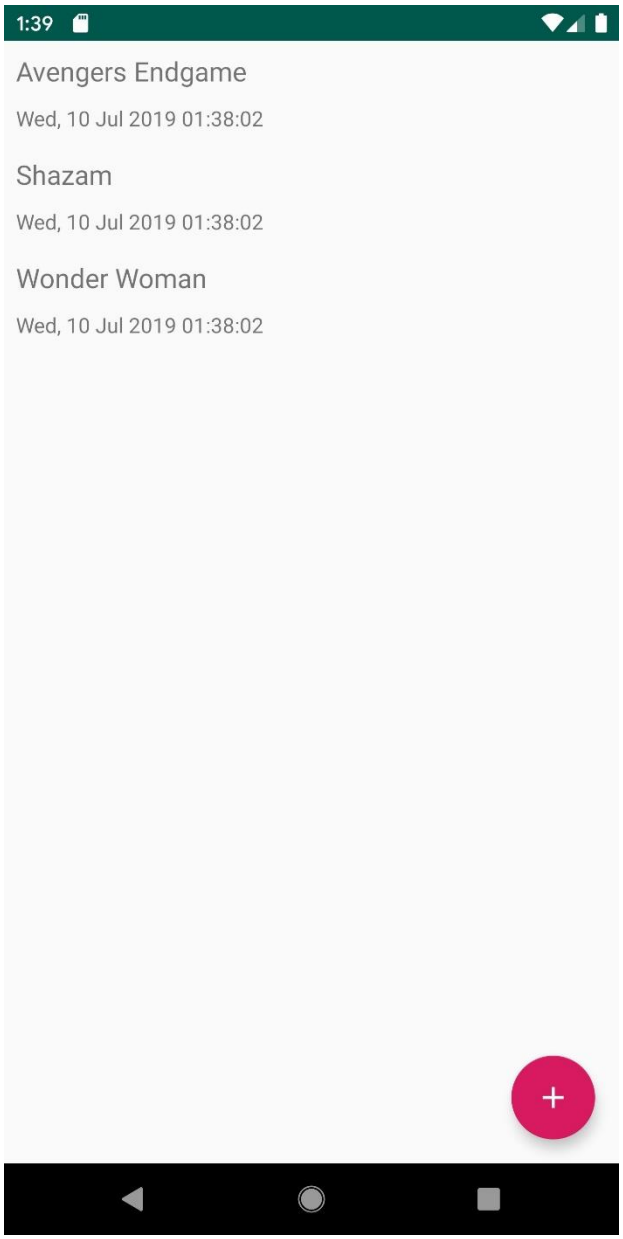
Sólo nos queda modificar dos métodos de MainActivity, los métodos que debemos modificar son createEntity, agregando createdAt con un Date que será la fecha y hora actual. El código queda de la siguiente forma:

```
private fun createEntity(text:String):List<RecommendationEntity> {
   val listEntities = mutableListOf<RecommendationEntity>()
   listEntities.add(RecommendationEntity(recommendation = text, createdAt =
Date()))
   return listEntities
}
```

También modificamos createEntityListFromDatabase, debemos agregar un método que maneje la fecha de creación, sobretodo en las primeras recomendaciones, ellas tienen created_at = null, ya que la columna no existía cuando fueron creadas. Las nuevas entradas tendrán la fecha de creación gracias a la modificación que hicimos en createEntity. El método createEntityListFromDatabase queda de la siguiente forma:

```
private fun createEntityListFromDatabase(entities:List<RecommendationEntity>):
MutableList<RecommendationDataView> {
   val dataList = mutableListOf<RecommendationDataView>()
   if (entities.isNotEmpty()) {
       for (entity in entities) {
           val date = entity.createdAt ?: Date()
           val dataView = RecommendationDataView(
               entity.id,
               entity.recommendation,
               date)
           dataList.add(dataView)
       }
   }
   return dataList
}
```

Después de estas modificaciones estamos listos para correr la aplicación. Si implementamos todo correctamente, la migración será sin problemas y veremos las siguientes pantallas:



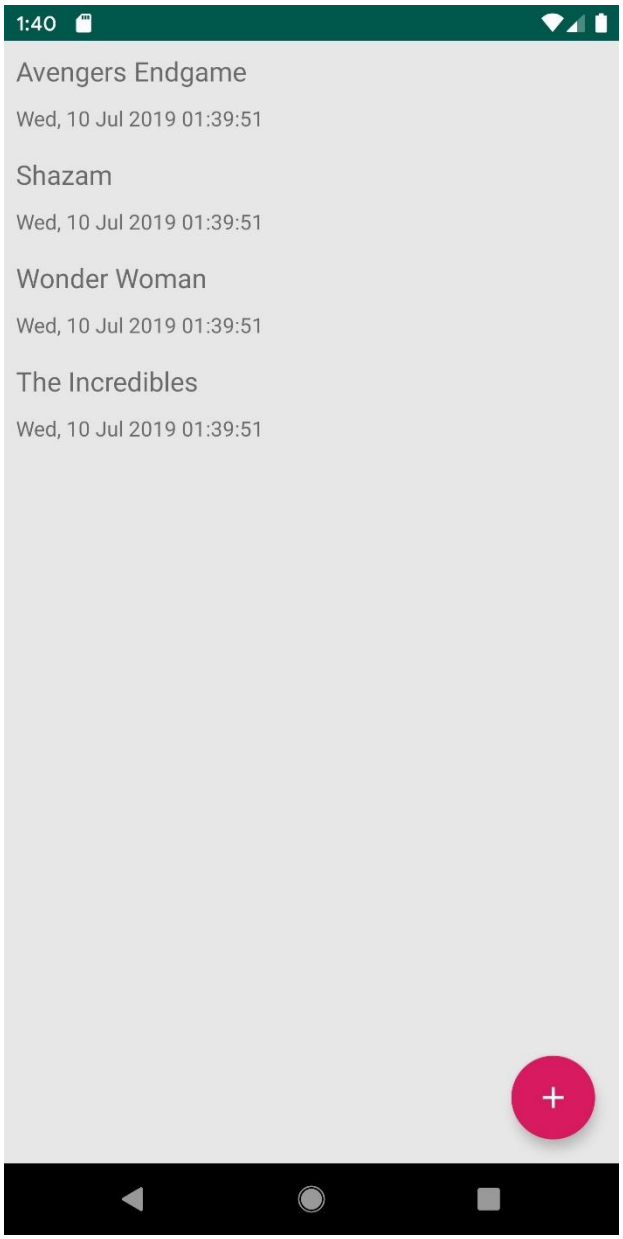


Imagen 9 y 10. Pantallas de la aplicación.

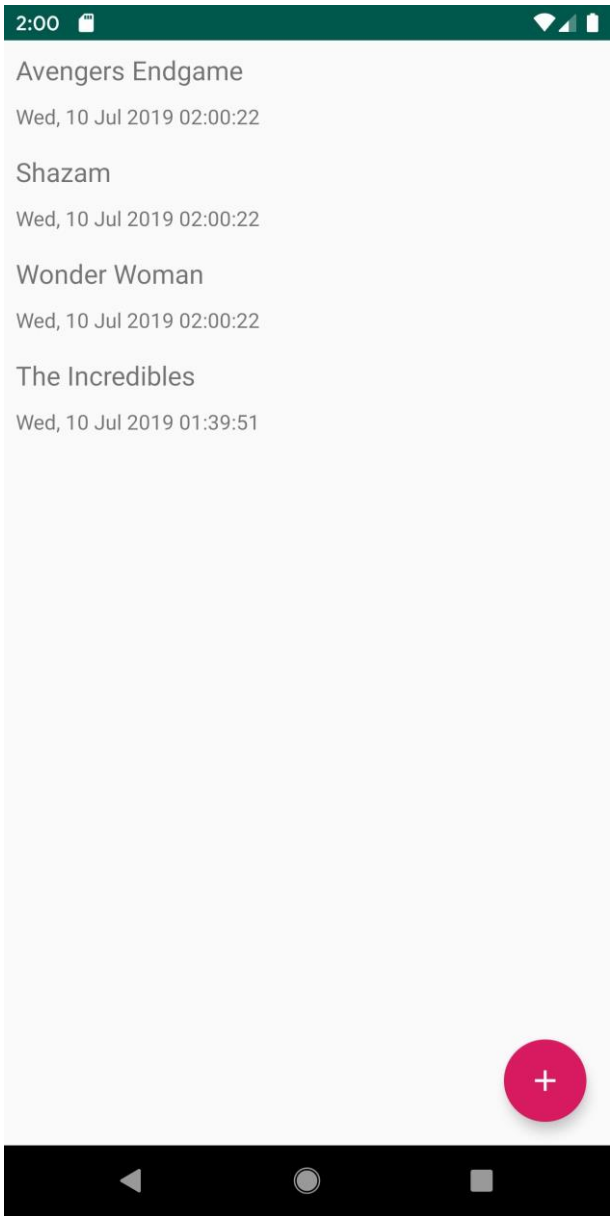


Imagen 11. Pantallas de la aplicación.

Cómo vemos las primeras recomendaciones van a mostrar siempre la última fecha, ya que su fecha es null. Pero las recomendaciones que ingresamos después de crear la nueva columna, veremos la fecha real de cuando se guardaron.Con esto concluimos el capítulo de Room. Ahora tenemos un capítulo donde veremos algunas de las opciones que existen aparte de Room en Android.

Realm y ObjectBox

Competencias:

- Conocer opciones a SQLite y Room para almacenar datos en Android
- Entender las principales ventajas y desventajas de Realm y ObjectBox
- Entender qué opción es la mejor para almacenar datos en una aplicación, dependiendo las necesidades de dicha aplicación.

Motivación

Conocer dos nuevas opciones para tener una capa de persistencia en nuestra aplicación. Si bien Android nos ofrece una librería oficial, muchas veces hay soluciones que son más fáciles de implementar, ocupan menos espacio o tienen mejor desempeño. Veremos dos de esas opciones, la madura Realm y la novata ObjectBox. Este capítulo no es una guía completa de ambas opciones, está más bien orientado a presentar y describir las ventajas, mostrar algunos ejemplos e incentivar la curiosidad para su uso.

Realm

Comenzaremos hablando de Realm. Se define como una base de datos no relacional o NoSQL. Este tipo de base de datos están diseñadas para tener un gran desempeño al leer, las consultas son casi inmediatas. Realm está específicamente diseñada para las plataformas móviles, originalmente estaba diseñada sólo para iOS, pero dado su éxito la versión de Android se puso a disposición de la comunidad en poco tiempo.

¿Cuáles son las ventajas y desventajas de Realm?. Veamos la respuesta:

Ventajas y Desventajas Realm

Las **ventajas** son las siguientes:

- Simplicidad: A diferencia de una base de datos SQL, SQLite para ser exactos, el código para realizar la mayoría de las operaciones en Realm es corto y conciso. Debemos lidiar con objetos y árboles de objetos, no tablas ni filas.
- Velocidad: como todo el sistema trabaja en base a objetos, de manera no relacional, las operaciones CRUD son muy rápidas. Además, la librería completa está escrita en c++, asegurando un desempeño a toda prueba.
- Live Objects: Realm no copia datos, todos los accesos a los datos son cargados de forma lazy, generando referencias, sin copiar los datos. Esto, en la práctica, significa que cada vez que accedemos a un objeto obtenemos su última versión.
- Documentación de primer nivel, una gran comunidad: todo lo que rodea a Realm es muy completo, una gran comunidad, buena tecnología y buenísima documentación.
- Soporte para Json: Realm incluye soporte para Json, esto es muy útil cuando estamos utilizando servicios REST o utilizando archivos Json para configuraciones u otros casos en la aplicación. Normalmente las base de dato NoSQL construyen documentos para almacenar la data, algunas ocupan json para lograr eso. Nos ahorra la necesidad de crear data classes para el parseo de datos de servicios web o archivos.
- Seguridad: Se puede utilizar encriptación para la base de datos y los datos.
- Programación Reactiva: Los cambios en los objetos de datos pueden ser “observados” y de esa forma mantener la UI actualizada, con cada cambio. Compatible con RxJava, la librería de programación reactiva más popular en Android.
- Adapters para la UI de Android: Realm trae adaptadores listos para ser utilizados con la UI de Android, por ejemplo en una lista de sugerencias.

Las principales **desventajas** de Realm, que si bien no son muchas pueden resultar relevantes dependiendo el caso, son:

- No autoincrementa valores: por lo que debemos definir funciones para hacer el trabajo de incrementar por ejemplo ids de los objetos.
- Model classes limitados: si bien podemos definir clases que modelan los datos, estos no se pueden personalizar mucho, normalmente se puede modificar los getters y setters, pero nada más.
- No es multi-hilo: Android es un sistema multi-hilo, la interfaz corre en un hilo principal, y no se debe bloquear nunca. Las entidades, model classes, no pueden pasar de un thread a otro en Realm. Por ejemplo si estamos modificando un dato fuera del hilo que corre la interfaz esa modificación no se puede pasar al hilo principal. Para poder leer datos modificados fuera del hilo principal, debemos hacer una consulta en el hilo principal, después de terminar de modificar los datos.
- Problemas con el tamaño: Las base de datos pueden ser de gran tamaño, debido a cómo está construída en Realm.
- Se necesita un binario para cada arquitectura soportada, recordemos que el código está en c++. Esto implica un APK universal más grande con cada arquitectura que agreguemos. Normalmente se soportan 4: armeabi-v7a, arm64-v8a, x86 y x86_64.
- Puede tener problemas de performance, dependiendo que tan compleja y grande sea la base de datos.

Después de ver las ventajas y desventajas de Realm, veremos algunos ejemplos. Para más información revisar https://realm.io/docs/java/latest, no hay documentación en Kotlin por ahora en la documentación oficial, pero está 100% soportado.

Ejemplos de Realm

Así podemos definir un modelo:

```
open class Person(
@PrimaryKey var id: Long = 0,
var firstName: String = "",
var lastName: String = "",
var age: Int= 0,
@Ignore var campoCalculado: Long= 0L) : RealmObject() {}
```

Definimos un objeto, como una clase, anotamos la clave primaria, y extendemos de RealmObject. Con esto tenemos todo lo necesario para operar. Notar que también podemos anotar que campos se ignoran. En Kotlin debemos poner valores por defecto obligatoriamente, Realm no tiene valores por defecto para los tipos.

Para inicializar la base de datos simplemente hacemos lo siguiente:

```
class ExampleApplication : Application() {

    override fun onCreate() {
        super.onCreate()
        //al igual que Room sólo inicializamos una vez cuando
        //parte la aplicación
        Realm.init(this)
    }
}
```

La inicialización se debe hacer una sola vez, después de esto podemos utilizar la base de datos llamando la misma de la siguiente manera:

```
realm = Realm.getDefaultInstance()
```

Teniendo la referencia a la base de datos podemos ejecutar transacciones y queries, por ejemplo si quiero buscar a una persona con cierta edad en mi base de datos de personas podemos hacer lo siguiente:

```
val results = realm.where<Person>().equalTo("age", 55).findAll()
```

En este caso results es una lista de todas las coincidencias de esa búsqueda, ya que usamos findAll(). muchos más métodos están disponibles, todos bien explicados y con ejemplos en la documentación oficial. Para buscar por un campo ocupamos el método where, al igual que SQL en un select where. Le decimos que el tipo de objeto que vamos a buscar es de tipo Person y que necesitamos que nos encuentre a todos los que tienen el campo “age” igual a 55. Podemos ocupar otros comparadores para números, como por ejemplo:

- between, busca en un rango, incluidos los extremos
- lessThan: buscando el menor que el número definido

Hay más métodos disponibles en la documentación.

Cuando queremos escribir a la base de datos debemos ejecutar una transacción, para ejecutar transacciones lo debemos hacer en un bloque de código dentro de la función executeTransaction de la siguiente manera:

```
// buscamos a la primera persona de apellido bond, le
  //cambiaremos el apellido y la edad
  val person = realm.where<Person>()
                  .equalTo(“lastName”,”Bond”)
                  .findFirst()!!
  realm.executeTransaction { _ ->
          person.lastName = "DoubleOSeven"
          person.age = 50
      }
 //Hemos actualizado a la James Bond, ahora James DoubleOSeven
```

Las transacciones se pueden ejecutar en un thread distinto al de la UI, que es el que se ocupa cuando hacemos un executeTransaction, para hacer una ejecución en el background utilizamos executeTransactionAsync y Realm se encargará de ejecutar la transacción en un thread diferente, para no bloquear la UI.

Podemos ejecutar transacciones más complejas, pero como vemos en los ejemplos siempre utilizamos métodos de realm y las propiedades de los objetos. Esto simplifica mucho el trabajo y también nos permite utilizar la base de datos como si estuviéramos usando objetos.

Por ejemplo si queremos encontrar a todas las personas entre 25 y 35, que tienen apellido Perez o Soto, ordenados alfabéticamente, podemos hacer lo siguiente:

```
val results = realm.where<Person>()
            .between("age", 25, 35)
           .equalsTo("lastName", "Perez")
        .or()
        .equalsTo("lastName", "Soto")
         .sort(Person::age.name, Sort.DESCENDING)
            .findAll()
```

Realm es una tremenda herramienta, revisemos ObjectBox

ObjectBox

Una nueva opción que aparece hace muy poco, y que se presenta como una base de datos hecha para la IoT. Con esto en mente se diseñó para que fuera muy rápida. Sus principales ventajas son

- Rápida: diseñada para tener una performance sobresaliente, ganando en casi todos los benchmarks contra Room y Realm
- API basada en Objetos: ObjectBox no es un ORM, está construida en base a objetos 100%, no tiene columnas ni filas, sólo objetos, ni sql.
- QueryBuilder: Permite preguntar por objetos, hacer queries 100% basadas en objetos con verificación en tiempo de compilación, como lo hace Room con las queries de SQL.
- No es necesario migrar la base de datos, ObjectBox lo hace por uno, agrega automáticamente todos los cambios.

Parece no tener muchas desventajas a primera vista, salvo lo nueva que es, pero al parecer no tiene desventajas que la hagan un problema. Quizás en el futuro y con más uso, se encontrarán más desventajas.

Ejemplos de OpenBox

Para definir un Entity hacemos lo siguiente, los id se anotan con la anotación @Id

```
@Entity
data class Note(
@Id var id: Long = 0,
val text: String? )
```

Similar a lo que ya hemos visto en Room y Realm.

Para construir entidades ObjectBox ocupa una de estas dos opciones en Kotlin:

- El constructor por defecto, es decir el que tiene todos los argumentos asignables. Siempre prefiere este constructor.
- Un constructor sin todos los argumentos, esto lo podemos construir con un constructor custom o agregando valores por defecto a los argumentos del constructor por defecto. Este se elige después que falló llamar al constructor con todos los argumentos.

Por ejemplo esta clase permite ocupar la segunda opción

```
@Entity
data class Note(
        @Id var id: Long = 0,
        var text: String? = null,
        var comment: String? = null,
        var date: Date? = null
)
```

Para inicializar la base de datos hacemos uso de la clase MyObjectBox, de la siguiente manera, nuevamente esto se recomienda hacer una vez y en la clase Application de nuestra aplicación.

```
class ExampleApplication : Application() {

    lateinit var boxStore: BoxStore
    override fun onCreate() {
      super.onCreate()
      boxStore = MyObjectBox
                 .builder().androidContext(this).build()
    }
}
```

Hemos creado nuestra base de datos, queda referenciada en boxStore, para interactuar con los otros objetos o entidades debemos crear un box. Eso lo podemos hacer de la siguiente manera:

```
var notesBox = ExampleApplication.boxStore.boxFor()
```

Con este box podemos ejecutar Queries, por ejemplo ordenar todas las notas del sistema alfabéticamente

```
var notesQuery = notesBox.query {
order(Note_.text)
}
```

Para obtener todas las notas del sistema

```
val notes = notesQuery.find()
```

Para crear una nueva nota y agregarla a nuestra base de datos

```
val noteText = "texto de la nota"
val df = DateFormat.getDateTimeInstance(DateFormat.MEDIUM, DateFormat.MEDIUM)
val comment = "Agregada el " + df.format(Date())
//creamos la nota igual que como creamos un objeto
val note = Note(text = noteText, comment = comment, date = Date())
// para agregar la nueva nota a la base de dato hacemos put
notesBox.put(note)
```

Como se puede ver la simplicidad es la regla en esta herramienta, entonces la pregunta que nos debemos hacer ahora es:

¿Cuál ocupo en mi proyectos?

Room vs Realm vs ObjectBox, ¿Cúal es la mejor para mi proyecto?

Esta pregunta no tiene una respuesta simple ni obvia, porque va a depender de muchos factores. Si bien uno puede tener preferencias por razones personales, hay que aplicar una mezcla entre facilidad de uso, rendimiento y mantenimiento. Hay que tener claro que las herramientas de terceros pueden quedarse sin soporte, tener errores que nos son corregidos, etc. Al final depender de un tercero me facilita el trabajo, pero no me asegura fiabilidad todo el tiempo.

Es normal encontrar errores, incluso de Google, que no tienen solución oficial, y se termina “hackeando” la solución, o se busca un camino alternativo.Veamos algunos criterios para decidir, revisaremos algunos test realizados por terceros, para tener un punto de partida en esta discusión. Las comparativas completas se referencian en la bibliografía.

Desempeño y Tamaño

Si es por velocidad pura de operaciones CRUD, podemos mirar estos gráficos de una comparativa de AndroidProDev, donde el eje Y muestra el tiempo en milisegundos y el eje X muestra la cantidad de elementos en la base de datos, de 10000 a 50000. Los tests se corrieron 10 veces para cada librería, por cada tramo 10k, 20k, 30k, 40k y 50k. En los gráficos se muestran los promedios. El código de los tests se encuentra disponible en https://github.com/mac229/Databases/blob/master/app/src/androidTest/java/com/maciejkozlowski/databases/DatabasesTest.kt

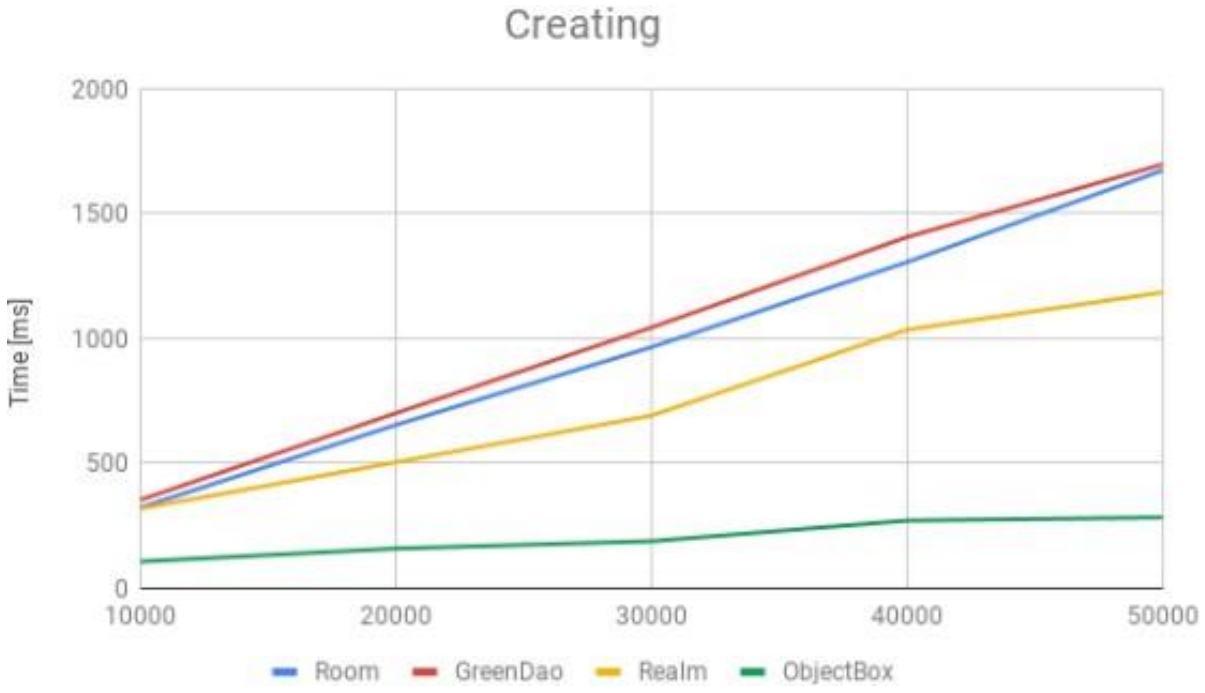


Imagen 12. Elementos creados en las bases de datos en el tiempo.

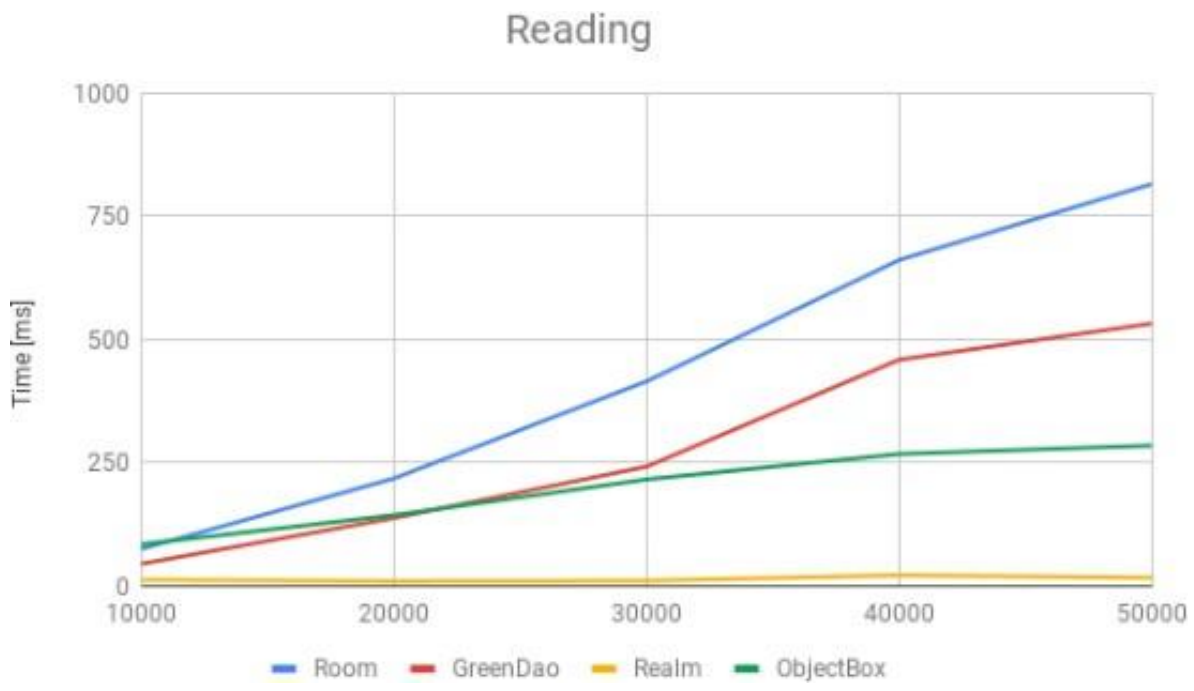


Imagen 13. Elementos leido en las bases de datos en el tiempo.

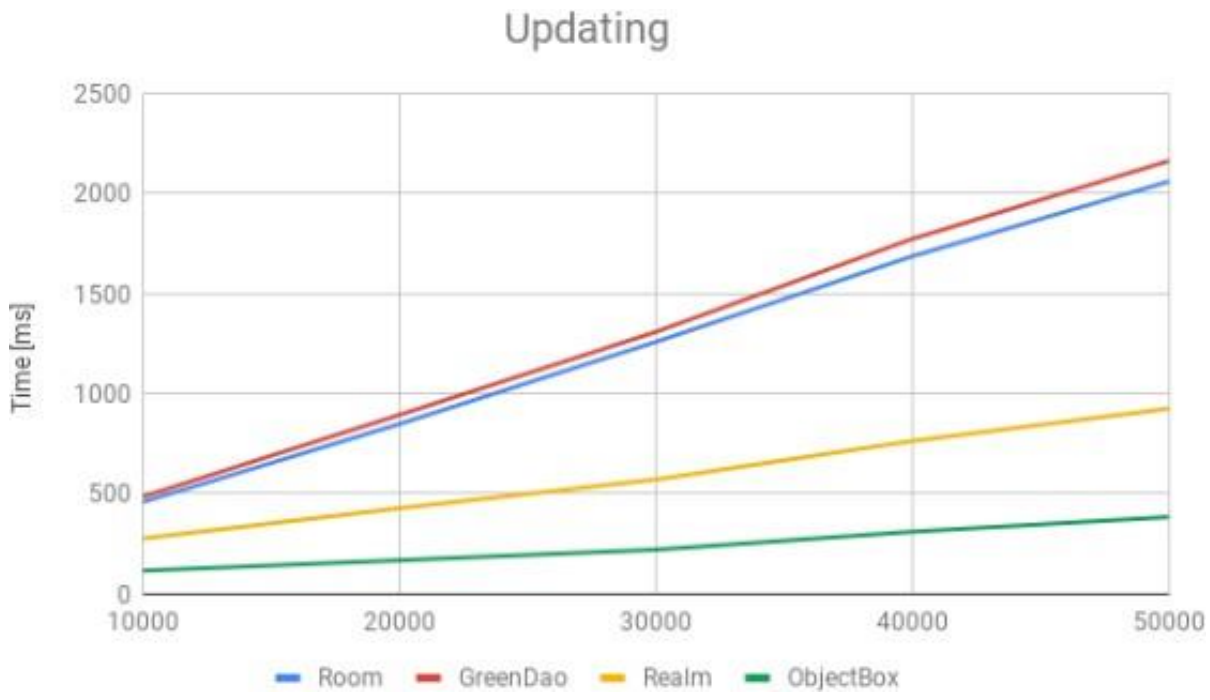


Imagen 14. Elementos actualizados en las bases de datos en el tiempo.

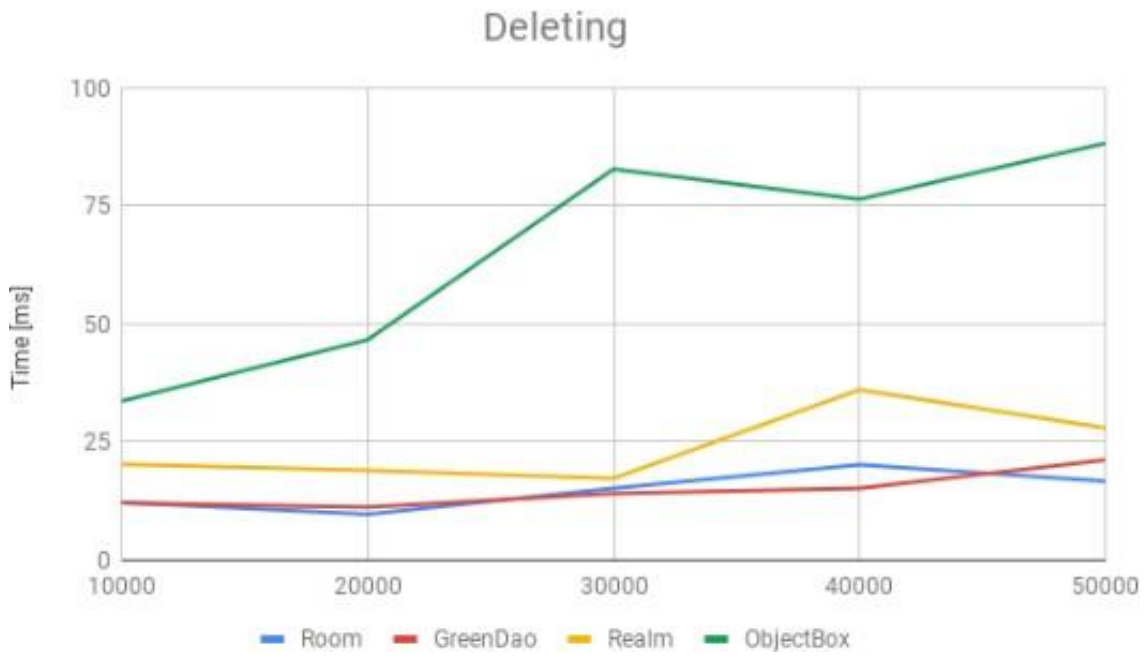


Imagen 15. Elementos eliminados en las bases de datos en el tiempo.

En estos gráficos aparece una cuarta opción, en rojo llamada GreenDao, que es de los mismos que crearon ObjectBox, la gente de GreenRobot, un gran aporte a Android en general con tremendas librerías. GreenDao, ORM SQL, no es parte de nuestra comparativa por lo que la ignoramos.

Como vemos ObjectBox tiene un tremendo desempeño en casi todas las categorías, salvo borrar donde se dispara su tiempo. Realm tiene un desempeño notable en la lectura, donde el tiempo es prácticamente constante. Room tiene un desempeño casi lineal con respecto a la cantidad de data en la base de datos, y se destaca en el borrado de datos. Todo esto hace mucho sentido con el tipo de implementación de las librerías:

- ObjectBox está orientada a desempeño en las operaciones comunes Crear, Leer y Actualizar.
- Realm está orientada a ser rápida sobretodo en lectura y actualizaciones, recordando la simplicidad de sus queries y transacciones. El generar las relaciones de objetos, en NoSQL se generan “documentos”, es caro, pero leer y actualizar es muy veloz.
- Room está diseñada con SQL en mente, trabaja sobre SQLite, por lo que se explica muy bien su desempeño lineal en todas las operaciones. Borrar data en SQL es muy veloz, mucho más que en NoSQL.

En cuanto a tamaño podemos mirar el siguiente gráfico, que nos muestra cual el resultado final de cada base de datos, y aquí también se ve la relación entre el diseño de la librería, su velocidad y que podemos esperar en casos extremos:

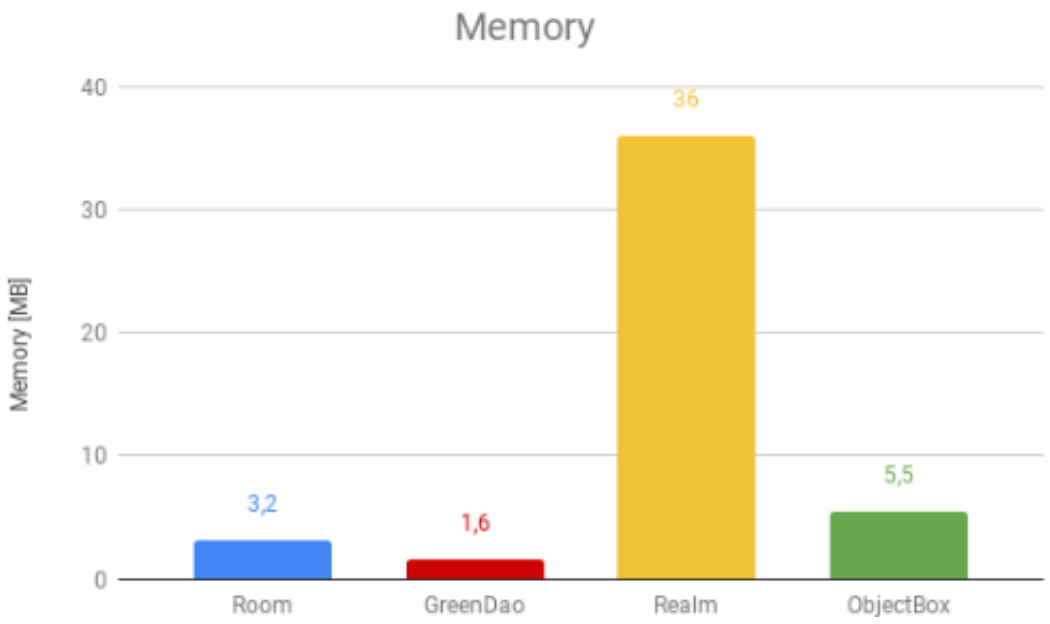


Imagen 16. Uso de memoria de una base de datos.

Hace sentido que Realm se la que ocupa más espacio, su tiempo de lectura era constante, independiente del tamaño de los datos, eso implica que la Base de Datos NoSQL está almacenando todos lasrelaciones y los resultados posibles de sus entidades, esto también se ve reflejado en ObjectBox, pero su implementación de objetos le da un mucho mejor resultado que Realm. Pero ObjectBox es casi el doble que la base de datos de Room, que se puede dar el lujo de depender de sus keys foráneas para reconstruir datos, por lo que no debe guardar ninguna relación extra, como sí lo tienen que hacer las NoSQL.

Entonces, ¿cuál me conviene?

La respuesta no es sencilla, pero depende directamente de las necesidades del projecto. Si necesitamos desempeño por sobre cualquier otra variable, tenemos Realm y ObjectBox. Si tenemos limitaciones de espacio, cosa habitual en los dispositivos móviles, ObjectBox es una buena opción, también lo es Room. Normalmente las base de datos SQL se utilizan para almacenar data que debe ser confiable y recuperable, no importando la lentitud del acceso. Cuando necesitamos velocidad, tenemos data que no cambia muy seguido, es media estática, y podemos perderla sin problemas, se recomienda NoSQL. Casoemblema de NoSQL, una memoria caché, que es volátil y podemos perder.Lo importante es tener claro que requerimientos tiene la aplicación, que dispositivos debemos soportar y si es necesario una librería compleja para resolver el problema.

LiveData and UI

Patrón Observador y Ciclos de vida de una actividad y fragmento

Competencias:

- Conocer qué es el patrón observador.
- Comprender el ciclo de vida de una actividad o fragmento.
- Construir un proyecto que registre los ciclos de vida de una actividad.
- Conocer qué es LifeCycle awareness.

Introducción

Antes de comenzar a explicar y utilizar LiveData, debemos recordar un patrón de diseño llamado Observador. Recordemos que los patrones de diseño son referencias para la construcción de software y podemos encontrar mucha información y literatura acerca de cómo nos ayudan y cuando utilizarlos, por ejemplo el famoso libro *Patrones de diseño: Elementos reutilizables en programación orientada a objetos*.

Este libro ha sido una gran referencia para el diseño y utilización de patrones de diseño en ingeniería de software, es recomendable es tenerlo como referencia de consulta cada vez que estamos utilizando un patrón o definiendo una solución para nuestro problema.

Tener un entendimiento de cómo funcionan las herramientas a este nivel abstracto, nos permitirá que conozcamos más en profundidad el porqué de las cosas. Y más importante aún las bases con las cuales están construidas herramientas que nos facilitaran la vida como desarrolladores.

En el caso de esta unidad, conocer cómo funcionan internamente LiveData o clases y componentes que son Life Cycle awareness, nos permitirán explicar cómo nuestras aplicaciones pueden realizar tareas complejas bajo un nuevo punto de entendimiento.

El patrón de diseño Observador.

El Patrón observador está calificado como un patrón de comportamiento, este tipo de patrón se encarga principalmente en la comunicación entre clases y objetos en un programa. El patrón observer está recomendado para cuando necesitamos reaccionar a los cambios que ocurren a un objeto, pero sin la necesidad de que estemos constantemente preguntando por el estado de ese objeto. El objeto se encarga de avisar cuando ocurre un cambio en su estado, notificando a todos los que están observando esos cambios.

Un problema que resuelve el patrón observador: Interfaces gráficas

A medida que avanzó la computación, diferentes cambios llevaron a pensar cómo diseñar nuestros programas de manera más eficiente e inteligente. Cuando aparecen las interfaces gráficas, se nos presenta el problema de que los **programas monolíticos** no escalan de manera correcta para atender este nuevo componente. Debido a que la interfaz podría cambiar por diferentes motivos, no había forma de obtener estos cambios, sin estar constantemente preguntando si los valores habían cambiado. El comportamiento ideal que debería tener el programa en este tipo de casos es:

1. La interfaz reacciona a los cambios, no pregunta por ellos
2. El objeto que cambia debe avisar a los interesados cada vez que cambia
3. Cada vez que los interesados son notificados de los cambios, se procede a actualizar la interfaz que esperaba esos cambios.
4. La relación entre un Sujeto y sus observadores es 1 - N (uno a muchos).

Para obtener este comportamiento se define en el patrón observador los siguientes elementos:

- **Subject**: Es considerado el que mantiene la información, los datos o la lógica de negocios del programa. Es la fuente de la “verdad”.
- **Observer**: Es un interesado en los datos, y quiere ser notificado cuando estos cambien. Es el que reacciona ante el evento gatillado
- **Register/Attach**: Los observers se registran en la lista de observadores del Subject, para ser notificados cuando ocurra un cambio.

- **Event**: Los eventos gatillan cambios en el Subject, todos los observers serán notificados de esos cambios.
- **Notify**: Dependiendo de qué tipo de implementación se haya realizado del patrón el Subject puede empujar la nueva información al observer, o el observer puede rescatar la nueva información desde el Subject, este proceso dependiendo del tipo de implementación será transparente para nosotros.
- **Update**: Los observers actualizan sus estados de manera independiente y sin verse afectado por otros observers.

Podemos simplificar el patrón observador de manera muy abstracta indicando que tendremos un tema de interés (Subject). El cual puede ser un dato en la BBDD, o un cambio de estado de algún elemento. Y luego quien esté interesado en escuchar o saber de los cambios que puedan producirse, deberásuscribirse para convertirse en Observadores(Observers).

Teniendo todo esto en cuenta, podemos observar el siguiente diagrama para ver una representación gráfica abstracta del patrón:

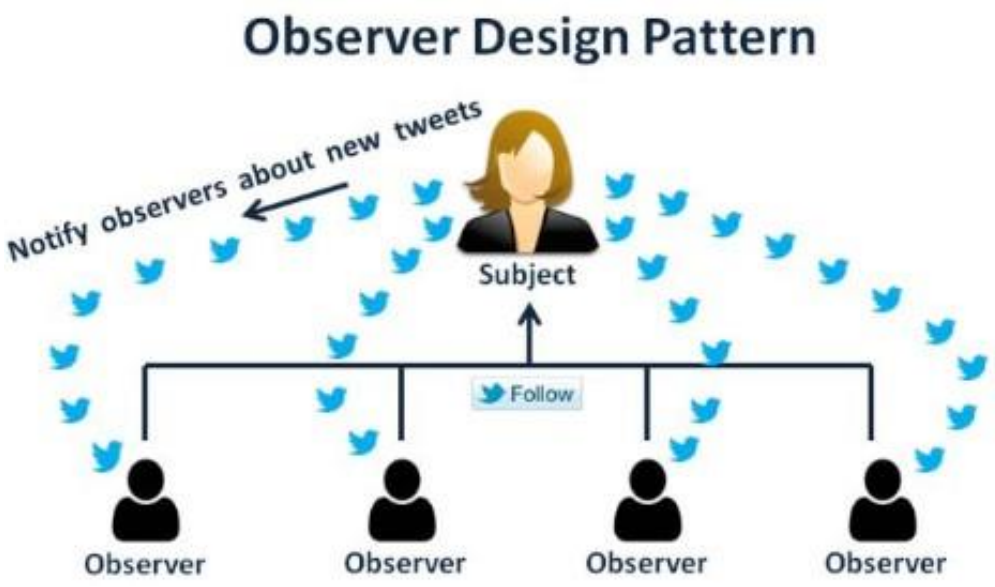


Imagen 1. Patrón observador.

Cómo podemos ver Twitter es un muy buen ejemplo abstracto de cómo funciona el patrón observador. Tenemos un Subject, que twittea su estado, el cual es visto por todos sus Observers, seguidores, los cuales son notificados de ese nuevo twit.

A nivel más técnico la implementación, normalmente, definimos dos interfaces una para el Subject y otra para el Observer, luego los objetos concretos implementan estas interfaces. En el siguiente diagrama podemos ver esta implementación más técnica:

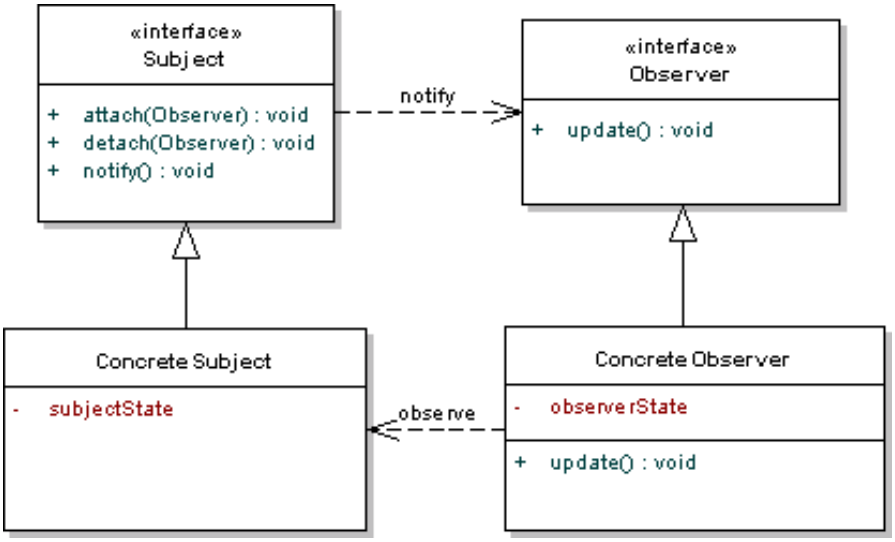


Imagen 2. Diagrama del comportamiento de Twitter.

Ahora que sabemos algunos detalles de cómo funciona este patrón pasaremos a explicar otro concepto muy importante para el desarrollo de aplicaciones móviles Android.

Ciclo de vida de una Actividad y Fragmento.

Sabemos que una actividad es un punto de partida en una aplicación Android, también es asociada con una representación de una vista cuando se asocia a un Layout. También podemos afirmar que una actividad tiene distintos estados y siempre está asociada a un ciclo de vida el cual llama a diferentes métodos cuando va cambiado de uno a otro, nosotros incluso podemos aprovechar estos métodos para ejecutar código que beneficie las funciones de nuestra aplicación.

Este ciclo de vida está definido en el framework de Android desde sus inicios. Este ciclo cuenta con 6 métodos principales, que son llamados a medida que la Actividad avanza en su ciclo de vida, los métodos son:

- **onCreate**: Este método se llama cuando la actividad es creada por primera vez. En este método debemos unir la interfaz, el layout xml a la actividad. Debemos configurar todas las variables iniciales, valores, y otras cosas que queremos que el usuario vea cuando la aplicación es desplegada. Debemos evitar realizar tareas grandes en este método, ya que entre más tiempo lleveesa tarea, más se demora en aparecer en pantalla la interfaz de la actividad.

- **onStart**: Este método se llama cuando la actividad se encuentra en estado started, ya es visible para el usuario, pero este no puede interactuar con ella. Normalmente en este método podemos gatillar tareas asíncronas, que no interfieren con el ciclo de la aplicación, y pueden cargar información cuando terminen sin bloquear la interfaz de usuario.
- **onResume**: Este método nos señala que la aplicación se encuentra en estado resumed, lo que básicamente significa que la actividad es visible, está ocupando la pantalla y el usuario puede interactuar con ella. En este estado es cuando normalmente se ejecutan las diferentes funciones interactivas de la aplicación, por ejemplo poder hacer like a fotos o escribir un correo.
- **onPaused**: este método se llama cuando la aplicación pierde el foco principal y entra en estado paused. Básicamente el usuario está dejando la actividad actual. Esto ocurre cuando gatillamos otra actividad, aparece un dialog o recibimos una llamada de teléfono.
- **onStop**: este método es llamado cuando la actividad entró en estado stopped. Esto significa que ya no es visible para el usuario, y que podría ser destruida por el sistema. La actividad ya no se encuentra en la pantalla del dispositivo.
- **onDestroy**: este método es el último que se llama en el ciclo de vida de una actividad. Este método se llama justo antes que la actividad sea terminada por el sistema operativo.

Este es a grandes rasgos el ciclo de vida de una Actividad, si bien estos métodos son los principales, debe ser claro que se cuenta con muchos más métodos y otras consideraciones cuando trabajamos con actividades en Android.

También es importante entender que los Fragments también poseen un ciclo de vida que está altamente ligado al ciclo de vida de la actividad que lo soporta.

Los estados y métodos más relevantes en el ciclo de vida de los fragmentos son los siguientes:

- onAttach():
- onCreate():
- onCreateView():
- onPause():

Existen otros, pero por el alcance de esta unidad basta con mencionar los principales.

En el siguiente diagrama podemos ver visualmente cual es la secuencia del ciclo de vida de una actividad y fragmento, también cómo interactúan los diferentes métodos.

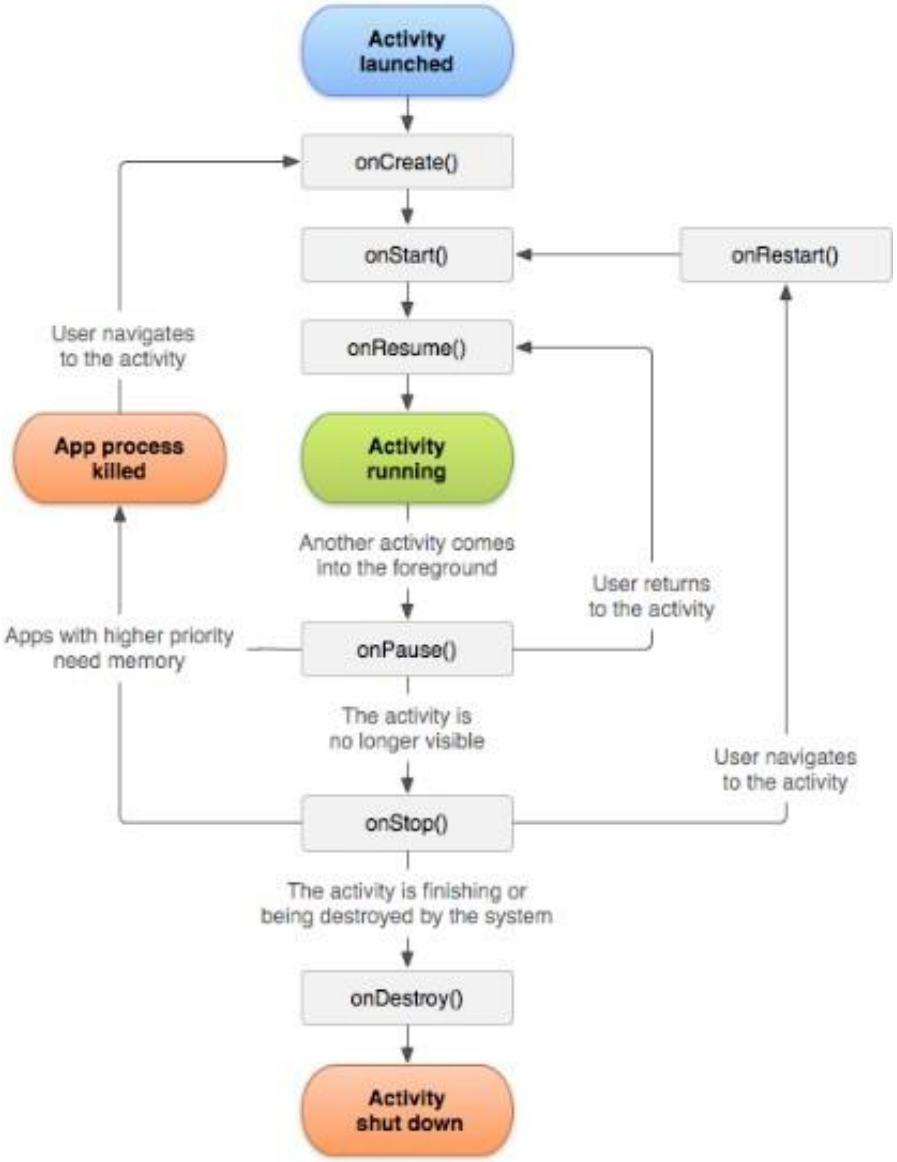


Imagen 3. Ciclo de vida de Actividad

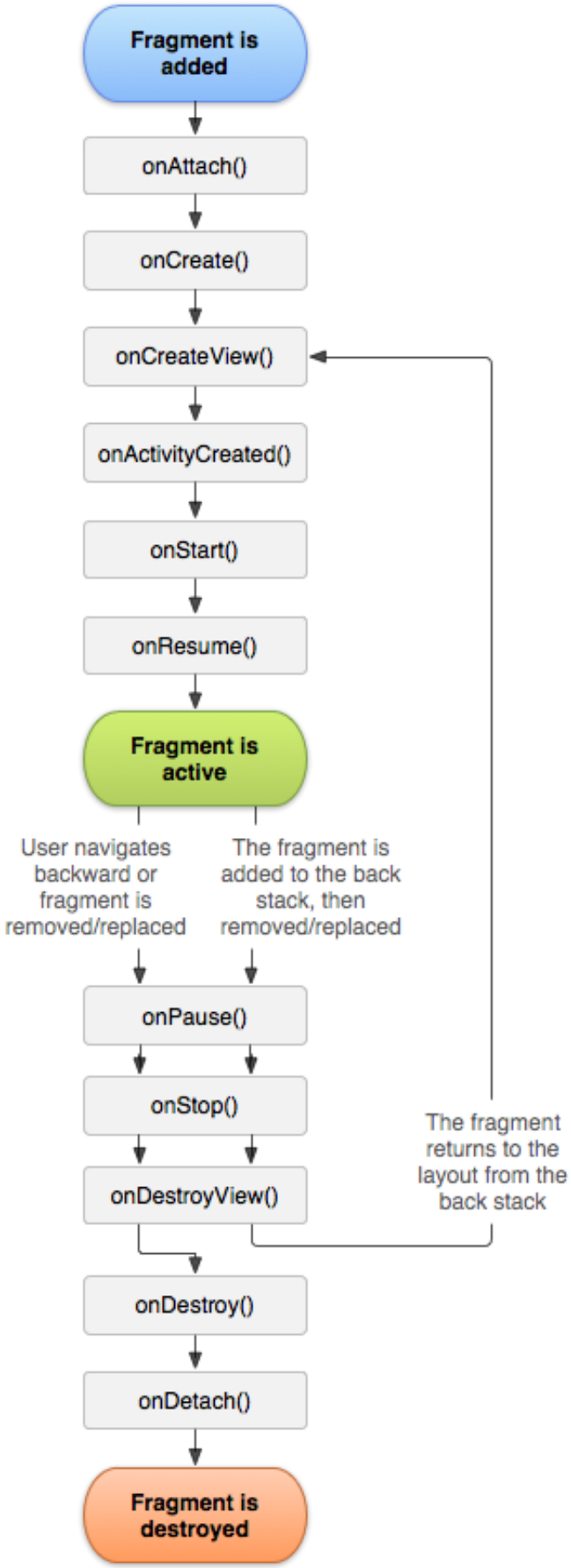


Imagen 4. Ciclo de vide de un Fragmento.

Manejo Manual de los ciclos de Vida

Como desarrolladores de aplicaciones Android debemos estar preocupados de que nuestras interfaces gráficas o UI mantengan sus datos lo más actualizado posibles o que se adapten a los cambios del ciclo de vida de la actividad o fragmento que se está mostrando, de forma que no pierdan información cuando ocurre un cambio en la configuración, el cual puede ocurrir simplemente rotando el dispositivo.

Existen varias alternativas para solventar estos problemas como por ejemplo:

- Manejar los datos en los componentes de UI (Objeto Dios)
- Usar Listeners
- Usar alguna alternativa de “eventBus”.

Sin embargo muchas de estas alternativas, nos dan un trabajo mayor ya que tenemos que estar preocupados de manejar manualmente el ciclo de vida y los posibles cambios que puedan ocurrir, además de añadir en algunos casos librerías de terceros.

LifeCicle-Aware components

Gracias a los nuevos componentes de arquitectura ya no tenemos que preocuparnos del manejo de los ciclos de vida como hace algunos años, debido a que elementos como LifeCycle classes se crearon para que los componentes puedan manejar el ciclo de vida en el cual están inmersos y sobrevivir a los cambios de configuración sin tener que realizar tareas extras por nuestra parte.

Estamos mencionando estos componentes conscientes de su ciclo de vida para que quede claro como LiveData y ViewModel manejan estos cambios sin necesidad de que participemos o escribamos código para ello.

Lifecycle y LifeCycleOwner

Como mencionamos, Android cuenta con componentes que son *Lifecycle aware*, lo cual significa que estos son capaces de responder o reaccionar a cambios en el ciclo de vida de otros componentes como Actividades o Fragmentos.

Ahora explicaremos los elementos que están presentes en estos componentes y que permiten que estos tengan esa clase de comportamiento.

Lifecycle

Es una clase que mantiene información acerca del estado del ciclo de vida de un componente, una actividad o un fragmento. Esta clase ocupa dos enumeraciones principales para mantener un registro de estado del ciclo de vida del componente asociado a la clase.

- Event: Los eventos del ciclo de vida que son despachados desde el framework de Android y la clase Lifecycle. Estos eventos mapean los callbacks del ciclo de vida de las actividades y los fragments.
- State: El estado actual del componente que es trackeado por el objeto Lifecycle.

En el siguiente diagrama podemos ver una representación gráfica del ciclo de vida completo de un componente y los diferentes estados del mismo. Observa como los estados en el lifecycle se producen inmediatamente después de los métodos de la Actividad hasta onResume, pero luego se ejecutan de forma anterior a los métodos desde onPause en adelante hasta el final del ciclo de vida de la actividad.

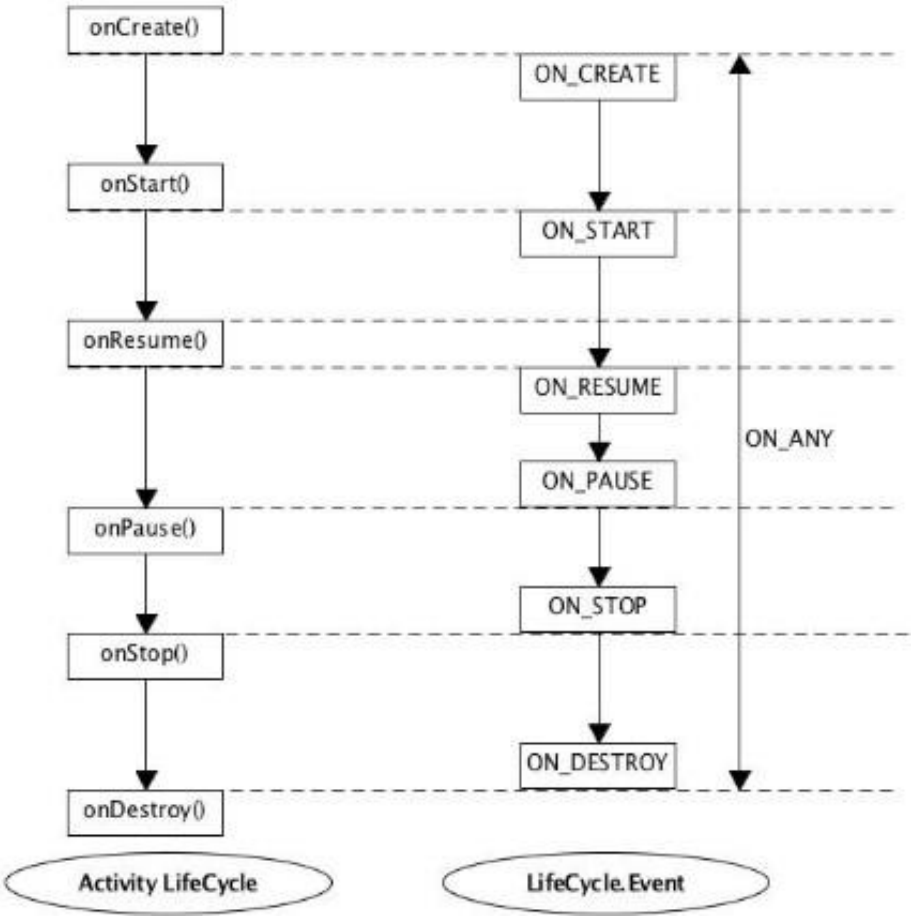


Imagen 5. Diagrama de flujo del ciclo de vida de una actividad.

Las clases que implementen Lifecycle pueden observar los estados de un LifecycleOwner y reaccionar a esos cambios.

LifecycleOwner y LifecycleObserver

Un Lifecycle owner es cualquier componente que implementa la interfaz LifecycleOwner, esta interfaz indica que tiene un ciclo de vida en Android. Los Fragmentos y las Actividades implementan esta interfaz desde la biblioteca de soporte 26.1.0.

Se pueden crear LifecycleOwners custom, implementando la interfaz LifecycleRegistry que le permite a el componente custom manejar múltiples observers.

Un LifecycleObserver es un componente que observa los diferentes estados asociados a un LifecycleOwner, reaccionando a los diferentes cambios.

Componentes Lifecycle, LifecycleOwner, LifecycleObserver, LifecycleRegistry

Veremos en acción estas clases en el siguiente capítulo, por ahora veamos sus componentes principales.

- **LifeCycle**:
  - **Lifecycle.Event**: Enum que enumera los distintos estados del ciclo de vida ON_CREATE, ON_START, ON_RESUME, ON_PAUSED, ON_STOP, ON_DESTROY y un valor especial que sirve para comparar con cualquier evento ON_ANY.
  - **Lifecycle.State**: enum que enumera los diferentes estados que tiene el LifecycleOwner asociado. En este caso tenemos lo siguiente
    - CREATED: estado después de onCreate y justo antes de onStop
    - DESTROYED: Estado final, justo antes de onDestroy, no se despachan más eventos después de este estado.
    - INITIALIZED: Es el estado cuando la actividad ha sido construida, pero no se ha llamado a su método onCreate.
    - RESUMED: estado después del llamado a onResume
    - : estado después de onStart y justo antes de onPause
    - Tiene 3 métodos asociados: isAtLeast, valueOf y values. El primero nos dice si el estado actual es al menos el valor que le pasamos como parámetro.
  - **addObserver**: método que recibe un LifecycleObserver y que será notificado cuando se produzca algún cambio de estado en el ciclo de vida.
  - **getCurrentState**: nos entrega el estado actual del ciclo de vida del componente asociado.
  - **removeObserver**: remueve el observador de la lista de observers. Recibe un LifecycleObserver.
- **LifecycleOwner**: es una interfaz que tiene un sólo método, este método, getLifecycle(), nos entrega el Lifecycle del LifecycleOwner. De esta forma clases lifecycle aware pueden observar los cambios en ese componente.

- **LifecycleObserver**: interfaz que marca una clase como Observador del ciclo de vida de otro componente. Esta interfaz ya no se utiliza, había que implementar un DefaultLifecycleObserver, actualmente se anotan los método con la siguiente anotación:
- **OnLifecycleEvent**: esta anotación nos dice que evento está observando el método anotado. Por ejemplo @OnLifecycleEvent(Lifecycle.Event.ON_CREATE) nos señala que el método anotado reaccionará al evento ON_CREATE.
- **LifecycleRegistry**: Esta clase es una implementación de Lifecycle que permite manejar múltiples observadores. Se utiliza para implementar los custom LifecycleOwners, cuenta con los siguientes componentes:
    - **Constructor** que recibe un LifecycleOwner que provee el ciclo de vida
    - **addObserver**: para agregar observers, recibe un LifecycleObserver.
    - **getCurrentState**: retorna el estado actual del ciclo de vida
    - **getObserverCount**: retorna el conteo de observadores de este registro.
    - **handleLifecycleEvent**: establece el estado actual del ciclo de vida, recibe un Lifecycle.Event. Notifica a los observers.
    - **markState**: recibe un Lifecycle.State, se mueve a ese estado y despacha todos los eventos necesarios a los observadores.
    - **removeObserver**: recibe un LifecycleObserver, lo remueve de la lista de observadores de este componente.

A continuación realizaremos un pequeño proyecto de ejemplo en el cual registramos estos estados en el LogCat de android studio, esto nos permitirá tener un mayor entendimiento de los diferentes estados del ciclo de vida de una actividad.

Proyecto Registro de estados

A continuación realizaremos un proyecto de ejemplo el cual será llamado EjemploLifecycle.

1. **Crea un nuevo proyecto en Android Studio.**

   a. Utiliza Kotlin como lenguaje.

   b. Crea una actividad vacía.

   c. utiliza una Api mínima 26.

2. **Configurando los componentes en el proyecto**

Para este caso la configuración es mínima, tendremos que agregar sólo lo siguiente a nuestro archivo gradle de dependencias del módulo app:

```
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'
apply plugin: 'kotlin-android-extensions'
apply plugin: 'kotlin-kapt'

...
dependencies {
   implementation fileTree(dir: 'libs', include: ['*.jar'])
   implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"
   implementation 'androidx.appcompat:appcompat:1.0.2'
   implementation 'androidx.core:core-ktx:1.0.2'
   implementation 'androidx.constraintlayout:constraintlayout:1.1.3'
   def lifecycle_version = "2.0.0"
   implementation "androidx.lifecycle:lifecycle-livedata:$lifecycle_version"
   kapt "androidx.lifecycle:lifecycle-compiler:$lifecycle_version"
   testImplementation 'junit:junit:4.12'
   androidTestImplementation 'androidx.test:runner:1.2.0'
   androidTestImplementation 'androidx.test.espresso:espresso-core:3.2.0'
}
```

Lo único que debemos agregar nosotros es la variable lifecycle_version y la dependencia de Livedata y el procesador de anotaciones con kapt, lo demás fue agregado por Android Studio cuando creó el proyecto base. Recordar también agregar el plugin kapt cómo se ve en el código adjunto.

3. **Registrando en un log los estados de un lifecycleOwner**

   a. Crea una clase llamada EventLogger

   b. Añade el código que hemos puesto a tu disposición.

Esta clase nos permitirá loguear los distintos eventos de un LifecycleOwner.

La clase EventLogger se verá de la siguiente manera:

```
class EventLogger {
   companion object {
       private const val TAG = "EventLogs"
   }

   fun eventLog(event: Lifecycle.Event) {
       when (event) {
           Lifecycle.Event.ON_CREATE -> Log.d(TAG, "OnCreate")
           Lifecycle.Event.ON_START -> Log.d(TAG, "OnStart")
           Lifecycle.Event.ON_RESUME -> Log.d(TAG, "OnResume")
           Lifecycle.Event.ON_PAUSE -> Log.i(TAG, "onPause")
           Lifecycle.Event.ON_STOP -> Log.i(TAG, "onStop")
           Lifecycle.Event.ON_DESTROY -> Log.i(TAG, "onDestroy")
           /**
            * ON_ANY no es un ciclo de vida, lo dejaremos por si ocurre algún
evento no cubierto
            * con las opciones anteriores.
            */
           Lifecycle.Event.ON_ANY -> Log.i(TAG, "onAny")

       }
   }
}
```

Cómo vemos la clase es simple, tiene un companion object que es la forma en que declaramos componentes estáticos en las clases de Kotlin, en este caso es el TAG que nos permite mantener un registro del Log.

También cuenta con un constructor vacío, denotado por la ausencia de paréntesis al lado del nombre de la clase.

Dentro de la clase tenemos un método principal llamado EventLog, que recibe como parámetro un Lifecycle.Event e imprime el método que fue llamado con ese evento. Dentro del método tenemos un when, para crear una iteración.

Cuando hacemos que el when maneje el evento, de tipo enum Lifecycle.Event, nos recomendará que sea exhaustivo, es decir, debemos ocupar todos los posibles valores del enum y hacer una acción asociada. No necesitamos un valor default como en un Switch.

4. **Creando una clase LifeCycle aware.**

A continuación crearemos una clase lifecycle aware que nos permita ir mostrando en la consola los distintos estados de una actividad.

Nuestra clase debe verse así:

```
class LifeCycleCustomObservation(
   private val lifeCycle: Lifecycle,
   private val eventLogger: EventLogger
) : LifecycleObserver {

   @OnLifecycleEvent(Lifecycle.Event.ON_CREATE)
   fun logOnCreate() {
       eventLogger.eventLog(Lifecycle.Event.ON_CREATE)
   }

   @OnLifecycleEvent(Lifecycle.Event.ON_START)
   fun logStart() {
       if (lifeCycle.currentState.isAtLeast(Lifecycle.State.STARTED)) {
           eventLogger.eventLog(Lifecycle.Event.ON_START)
       }
   }

   @OnLifecycleEvent(Lifecycle.Event.ON_RESUME)
   fun logResume() {
       eventLogger.eventLog(Lifecycle.Event.ON_RESUME)
   }

   @OnLifecycleEvent(Lifecycle.Event.ON_PAUSE)
   fun logPause() {
       eventLogger.eventLog(Lifecycle.Event.ON_PAUSE)
   }

   @OnLifecycleEvent(Lifecycle.Event.ON_STOP)
   fun logStop() {
       eventLogger.eventLog(Lifecycle.Event.ON_STOP)
   }
```

```
    @OnLifecycleEvent(Lifecycle.Event.ON_DESTROY)
    fun logDestroy() {
        if (lifeCycle.currentState.isAtLeast(Lifecycle.State.DESTROYED)) {
            eventLogger.eventLog(Lifecycle.Event.ON_DESTROY)
        }
    }
}
```

En esta clase podemos ver varias cosas importantes, partamos por su constructor y la interfaz que implementa.

Vemos que su constructor recibe 2 parámetros, un lifecycle que nos servirá para entender el estado en que se encuentra el LifecycleOwner, y un logger que nos permite escribir en la consola los eventos a medida que van ocurriendo.

Además esta clase implementa una interfaz LifecycleObserver, que le dice a nuestra aplicación, que esta clase puede observar cambios en el ciclo de vida de un componente.

Todos los métodos de la clase se encuentran anotados con @OnLifecycleEvent, esta anotación nos permite ser notificados de un evento en el ciclo de vida del owner, y según lo que definamos en los paréntesis de la anotación, se activará el método correspondiente.

Debemos notar que los eventos onStar y onDestroy, requieren el chequeo del estado del ciclo de vida, lifeCycle, que recibimos en el constructor. ¿Por qué ocurre esto? recordemos que los estados de un ciclo de vida no son lo mismo que los eventos de el lifeCycle observer, los estados son valores que nos indican en qué estado se encuentra el ciclo de vida, los eventos nos avisan cuando se ha llamado un cierto método del ciclo de vida.

Entonces para el evento ON_START el ciclo de vida debe estar en por lo menos el estado STARTED, esto lo sabemos usando el método isAtLeast del Enum State de Lifecycle. Lo mismo ocurre para el evento ON_DESTROY.

5. **Añadiendo el observador a la Actividad**

Finalmente añadiremos en la actividad el observador para loguear sus diferentes estados en la consola. La actividad principal queda de la siguiente manera:

```
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        setSupportActionBar(toolbar)

        val observer = LifeCycleCustomObservation(lifecycle, EventLogger())
        lifecycle.addObserver(observer)
    }
}
```

La actividad crea una variable del tipo observer, entregando dos parámetros: el lifecycle de sí misma y un EventLogger().

Finalmente agregamos ese observador a los observadores del ciclo de vida de la actividad.

6. **Ejecutando la Aplicación.**

Ahora ejecutamos la aplicación, si interactuamos con ella, presionando el botón para volver al escritorio, o la cerramos con el task manager. Veremos que diferentes Logs se registraran en el LogCat de Android Studio.

Veamos las siguientes imágenes para tener idea de cómo va esta secuencia:

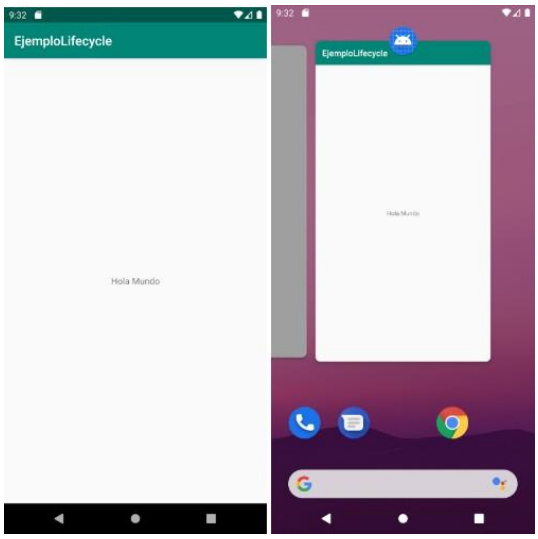


Imagen 6. Cerrando la aplicación.

Finalmente en Logcat podemos ver el los logs generados al ejecutar y cerrar la aplicación, usando el TAG “EventLogs” para encontrar más rápido y desplegar sólo los logs que queremos ver.

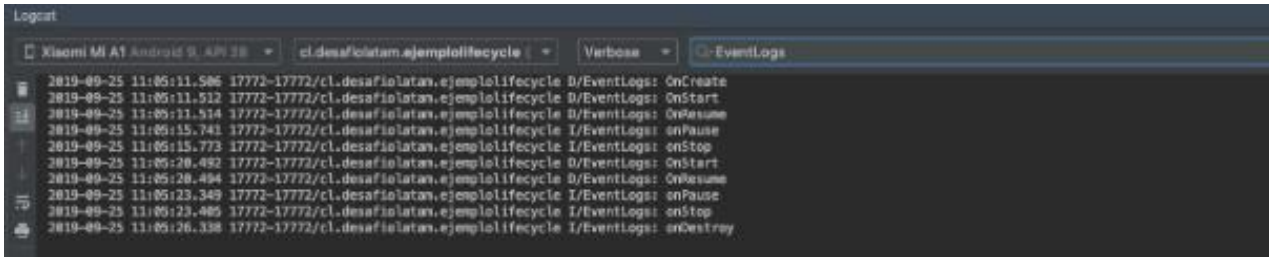


Imagen 7. Logcat.

Como puedes observar, a medida que ocurren los distintos cambios de configuración o simples pasos en los ciclos de vida, nuestra aplicación registrará estos cambios dejando un registro en el LogCat de Android Studio.

LiveData y ViewModel

Competencias:

- Entender que es LiveData
- Entender que es ViewModel
- Conocer los beneficios de usar Live Data
- Observar objetos LiveData
- Conocer MutableLiveData
- Utilizar setValue y postValue

Introducción

Ahora que sabemos lo que es un patrón observador y también sabemos que existen diferentes ciclos de vida en los elementos de nuestra UI, elementos que hace algún tiempo deberíamos manejarlos manualmente, lo cual se vio facilitado gracias a componentes de arquitectura que son lifecycle awareness (conscientes del ciclo de vida donde están funcionando), podemos comenzar a conocer LiveData y ViewModel.

La principal motivación para comprender qué es y cómo podemos utilizar estos elementos debe ser que nos permitirán mantener nuestras interfaces de usuarios actualizadas con los datos más frescos y siempre estarán escuchando cualquier cambio que se produzca en estos, sin tener que manejar los ciclosde vida de forma manual.

Para esta unidad utilizaremos conocimientos adquiridos en unidades anteriores como el uso de Room para proporcionar una persistencia de datos y a través de LiveData y el uso de ViewModels alimentaremos nuestra UI.

LiveData

Según la documentación oficial LiveData es una clase contenedora de datos observables. Además indica que LiveData al contrario de otros observables es consciente de los ciclos de vida, lo que quiere decir que respeta los ciclos de vida de otros componentes como por ejemplo Actividades, Fragmentos o servicios.

Si bien LiveData no es la única implementación del patrón observador en cuanto a contenedores de datos en Android, es la única implementación que es sensible a estos cambios en el ciclo de vida. Esta capacidad de entender el ciclo de vida de los observadores nos entrega una tremenda ventaja, LiveData **sólo notifica a los componentes que se encuentran en estado activo**.

LiveData considera que un observador está activo si su ciclo de vida se encuentra en STARTED o RESUMED. Sólo los objetos que cumplan con ese requisito serán notificados de los cambios. Los observadores inactivos no son notificados de dichos cambios. Esto es de mucha importancia ya que este comportamiento predeterminado ayuda mucho a que podamos en actividades o fragmentos observar datos sin tener que preocuparnos por leaks o pérdida de datos, debido a que automáticamente estos serán descartados de la suscripción de observación cuando su ciclo de vida sea destruido.

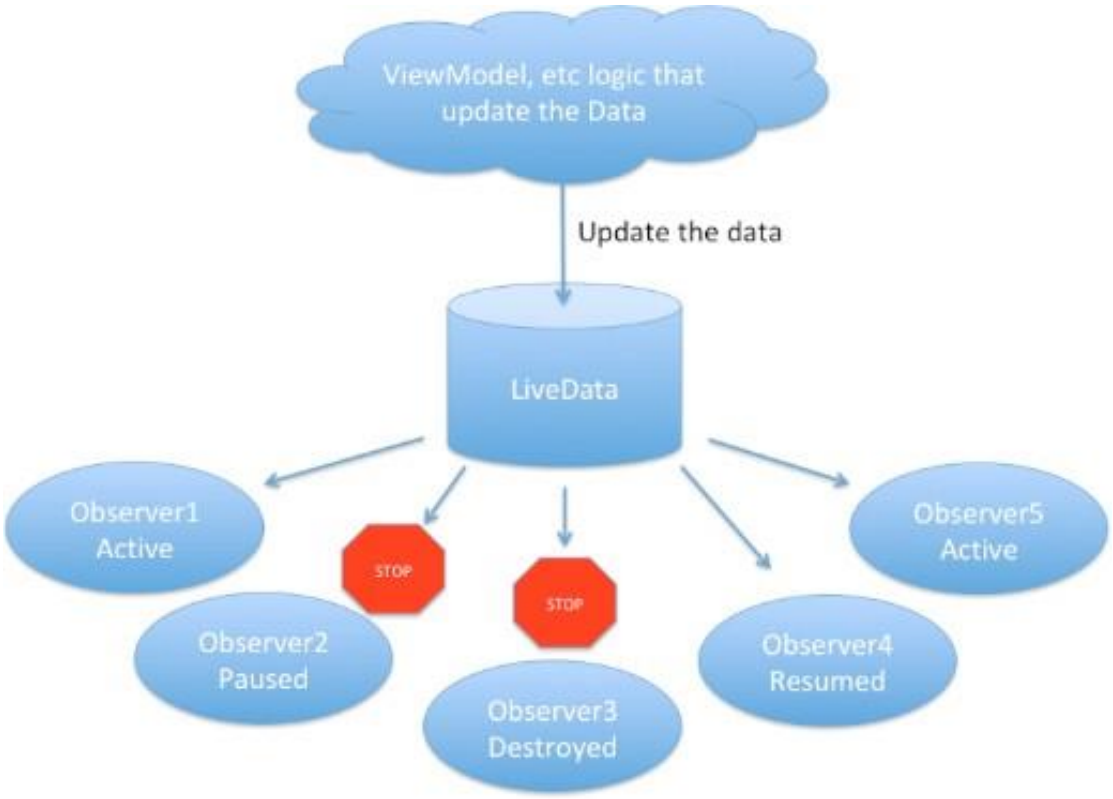


Imagen 8. Diagrama de LiveData.

Las ventajas de usar LiveData son las siguientes:

- **Asegura que la Interfaz de Usuario, UI, esté siempre actualizada**: Al seguir el patrón observador LiveData notifica los cambios a sus observadores, en este caso la interfaz gráfica. El observador será el encargado de actualizar la interfaz gráfica cuando ocurran cambios.
- **No más memory leaks**: Esto es una de las mejores ventajas, ya no debemos preocuparnos de limpiar los observadores para evitar memory leaks cuando están referenciados, pero no se usan más. Recordando que un objeto referenciado, pero que no se usa, no puede ser reciclado por el garbage collector.
- **No más caídas debido a actividades que son detenidas:** Si el ciclo de vida del observador está inactivo, los eventos no son notificados a ese observador, el cual puede ser null y generar un NullPointerException, muy común en otras implementaciones del patrón observer, cómo RxJava, si no se limpian los observadores.
- **No hay necesidad de manejar los ciclos de vida a mano**: Los componentes gráficos sólo deben observar los datos relevantes, no es necesario inscribir y reinscribir los observadores al pausar y resumir los ciclos de vida.
- **Datos siempre actualizados**: si un ciclo de vida pasa a estar inactivo, recibirá el último valor de los datos cuando vuelva a estar activo. Por ejemplo si tenemos una actividad en segundo plano, cuando vuelva a ser mostrada en primer plano, será actualizada con el último valor de los datos disponibles.
- **Manejo de cambios de configuración:** si ocurre algún cambio de configuración en el dispositivo, se rota la pantalla por ejemplo, la actividad o fragment afectado por ese cambio, recibirá inmediatamente el valor más actual de los datos.

Utilizar LiveData objects

Existen tres pasos para utilizar objetos de LiveData:

1. Crea una instancia de LiveData para mantener algún tipo de dato, regularmente esto se realizará al interior de una clase ViewModel.
2. Crea un objeto Observador que defina cuando se ejecuta un método onChange(), el cual controla que ocurre con el objeto LiveData cuando se produce un cambio. Regularmente el objeto observador lo crearemos en algún elemento de UI, ya sea actividad o fragmento.
3. Luego debes unir al objeto observador con el de LiveData usando el método **observe()**, Esto realiza la subscripción del objeto observador al objeto Livedata para que los cambios puedan ser notificados. Regularmente el objeto observador será atado a una actividad o fragmento.

En otras palabras, cuando se actualizan los datos almacenados en un objeto **LiveData**, esto gatilla a todos los observadores registrados durante el tiempo que el **LifeCycleOwner** esté en estado activo.

De esta forma **LiveData** mantiene a los elementos de UI automáticamente recibiendo actualizaciones.

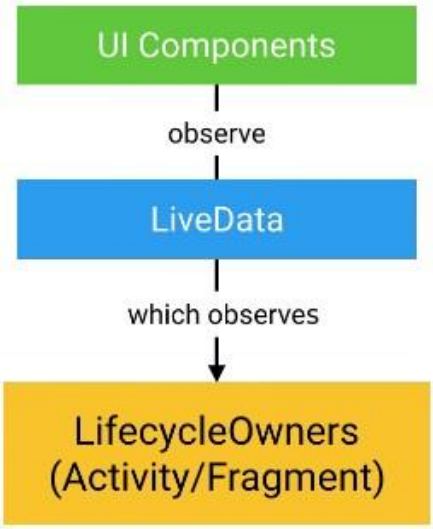


Imagen 9. LiveData mantiene los elementos de UI actualizados.

Crear objetos LiveData

Como LiveData es un envoltorio de datos, lo podemos utilizar con cualquier tipo de objeto por ejemplo colecciones, listas etc. Los objetos LiveData regularmente son almacenados al interior de un objeto ViewModel y se puede acceder a ellos a través de métodos de getter.

```
class NameViewModel : ViewModel(){
  //Create a LiveData with a String
  val currentName: MutableLiveData<String> by lazy{
    MutableLiveData<String>
  }
  // Rest of the ViewModel...
}
```

Más adelante realizaremos este procedimiento.

Observar objetos Livedata

Según la documentación oficial el método más idóneo para comenzar a observar un objeto es en el método **onCreate()**. Los motivos son los siguientes.

1. Para asegurarse que el sistema no realice llamadas redundantes desde un actividad o fragmento al momento de ejecutarse el método **onResume()**
2. Para asegurar que la actividad fragmento obtenga la data y la pueda mostrar tan pronto como esté activa. Es decir tan pronto como los componentes de la app están en STARTED, puedan recibir la actualización de los datos.

Generalmente, LiveData envía actualizaciones cuando los datos cambian y solo a los observadores activos. El siguiente código muestra cómo comenzar a observar un objeto **LiveData**.

```
class NameActivity : AppCompatActivity() {

    private lateinit var model: NameViewModel

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        // Other code to setup the activity...

        // Get the ViewModel.
        model = ViewModelProviders.of(this).get(NameViewModel::class.java)


        // Create the observer which updates the UI.
        val nameObserver = Observer<String> { newName ->
            // Update the UI, in this case, a TextView.
            nameTextView.text = newName
        }

        // Observe the LiveData, passing in this activity as the LifecycleOwner and the observer.
        model.currentName.observe(this, nameObserver)
    }
}
```

Imagen 10. Comenzado a observar un objeto LiveData.

Actualizando objetos LiveData (Update)

LiveData no tiene métodos públicos para actualizar los datos almacenados. Para esto, la clase MutableLiveData tiene a nuestra disposición los métodos setValue(T) y postValue(T), por lo tantodebemos usarla si necesitamos cambiar los datos almacenados en un objeto LiveData.

Usualmente MutableLiveData es usado en el ViewModel y el ViewModel solo expone objetos inmutables LiveData a los observadores. Para esto se utiliza la buena práctica de exponer estos objetos con un getter hacia la vista, esto ayudará a limitar la modificación de ese objeto. Esto ocurre de esta forma porque LiveData no tiene implementado los métodos setValue y postValue. Entonces si necesitáramos que se modifiquen, es mejor añadir una función que lo actualice. por ejemplo:

```
public LiveData<User> getUser(){
    return mUserInfoLiveData;
}
public void updateUser(User userUpdated){
    mUserInfoLiveData.postValue(userUpdated);
}
```

La documentación oficial indica que después que tengas establecida la relación con el observador, puedes actualizar los valores de un objeto LiveData.

```
button.setOnClickListener {
    val anotherName = "John Doe"
    model.currentName.setValue(anotherName)
}
```

Imagen 11. Actualizar valores de un objeto LiveData.

Llamar al método setValue(T) del ejemplo resultará en que los observadores llamaran al método onChange() pasando el nuevo nombre(“jhon Doe”).

La documentación también indica que si necesitamos realizar la actualización desde el Main Thread se utiliza setValue(T) y si lo haces desde un background thread, lo realizaremos con postValue(). Ambos métodos provocarán que los observadores actualicen la UI.

LiveData y Room

Afortunadamente la librería de persistencia de datos Room soporta consultas observables, las cuales pueden retornar objetos de LiveData. Estas consultas se escriben directamente en los DAO(Database access object).

Room generará todo el código necesario para actualizar el objeto LiveData cuando la base de datos sea actualizada. Estás consultas funcionan de forma asíncrona en un background thread cuando sean necesarias, esto permite que nuestra UI se mantenga sincronizado con los cambios en la base de datos.

En el siguiente capítulo realizaremos un ejemplo de esto.

ViewModel

ViewModel también es un nuevo elemento presente en los componentes de arquitectura, se define como una clase llamada ViewModel, la cual es la responsable de preparar la data para ser mostrada en la UI o vista, esta clase está optimizada para respetar los ciclos de vida. Esto quiere decir que no importa que se produzca alguna rotación del dispositivo o cambio de configuración, los datos estarán disponibles para la actividad o fragmento objetivo.

En la práctica utilizaremos ViewModel en conjunto con LiveData, será en este elemento donde se almacenan esos datos para ser mostrados en la actividad o fragmento que utilizaremos como UI.

ViewModel también cuenta con un ciclo de vida, lo podemos ver en la siguiente Imagen.

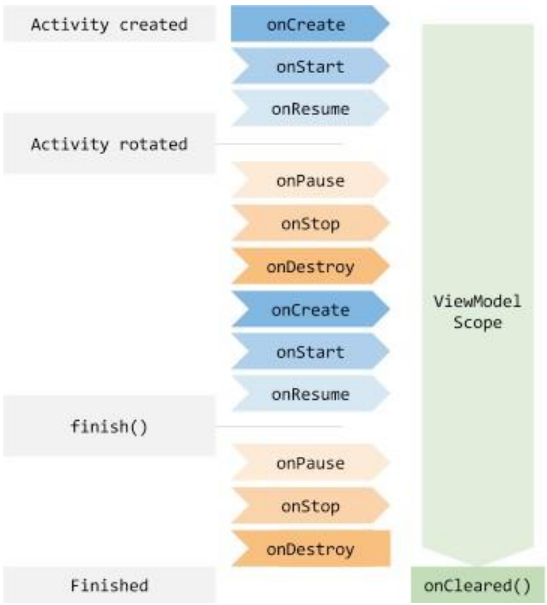


Imagen 12. Ciclo de vida.

Usualmente llamaremos al ViewModel la primera vez que el sistema llama al objeto de la actividad en el método onCreate(); El sistema podrá llamar a este método muchas veces si se produce algún cambio de configuración. Pero nuestro ViewModel ya existe hasta que la actividad sea finalizada y destruida.

En los siguientes capítulos veremos cómo implementar estos elementos en algún proyecto, mezclando diferentes componentes de arquitectura y aprovechando las características que hemos visto en este capítulo.

Pasando Datos entre Fragmentos con ViewModel

En el desarrollo de aplicaciones, existen varias formas de pasar datos entre Fragmentos la más usual es a través de una Interface. Gracias a la creación de **ViewModel** existe una nueva forma de hacerlo.

Esta forma nos trae todas las ventajas como permitirnos no tener que traer data múltiples veces si existe un cambio de configuración o rotación del dispositivo. Y esto ocurre gracias a que el viewModel está asociado al ciclo de vida de la actividad.

Los pasos son simples, necesitamos crear un ViewModel con el scope de la actividad de los dos fragmentos, luego inicializar el objeto ViewModel y pasar los valores a través de un objeto LiveData.

Después de esto el otro fragmento puede observar el objeto Livedata y obtener los valores para ser mostrados en la UI

LiveData y ViewModel en un proyecto.

Competencias

- Crear un proyecto usando el patrón de diseño MVVM.
- Utilizar LiveData en un Proyecto.
- Utilizar ViewModel para el manejo de los datos.
- Utilizar Patrón repositorio en un proyecto.
- Utilizar LiveData con Room.

Introducción

En este capítulo comenzaremos a converger nuestros conocimientos, para ello crearemos un proyecto en el cual añadiremos los componentes de arquitectura que venimos repasando hasta esta unidad.

Nos centraremos en crear la estructura para seguir un patrón de arquitectura recomendado, utilizando Room para persistencia de datos en conjunto con LiveData para para alimentar nuestra UI.

El ejemplo será simple, pero demostrará todo lo que necesitamos saber de estos componentes. La idea es que todos los elementos que hemos estado estudiando trabajen juntos y tengamos una idea de cómo las aplicaciones modernas son construidas.

Durante esta unidad nos centraremos en los componente base y en la siguiente la conectaremos con la UI.

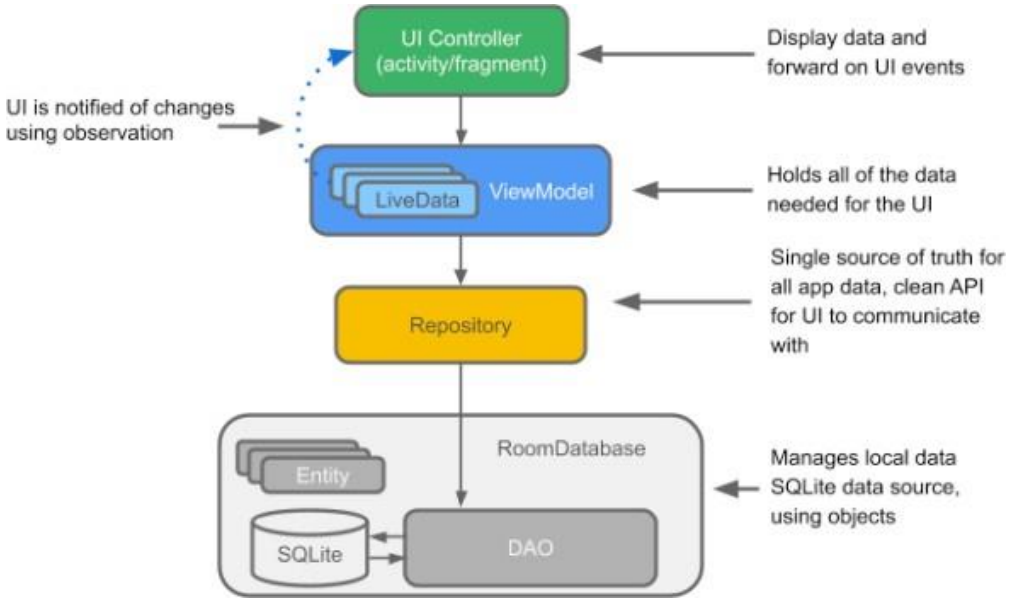


Imagen 13. Esquema del proyecto.

Este es el esquema que nuestra aplicación tendrá, como podemos observar es el mismo que propone google como alternativa a otros patrones de diseño.

A modo de resumen los participantes serán:

Nuestras Entidades describirán las tablas de la aplicación manejadas por Room.

- **SQLite database** se refiere a la persistencia de datos en el dispositivo y para facilitarnos la vida será también manejada con Room.
- **DAO** Data access object serán las encargadas de ser el puente entre nuestra aplicación y las consultas a esta base de datos, afortunadamente Room se encargará de esto.
- **Repository** Esta clase se encargará de manejar las diferentes fuentes de datos que puede tener nuestra aplicación, en el caso particular de nuestra aplicación solo manejara los datos desde la persistencia local. Pero más adelante se pueden añadir otras fuentes.
- **ViewModel** Actuará como el centro de comunicación entre el repositorio es decir los datos y nuestra UI. Nuestras UI ya no tienen que preocuparse por el origen de los datos, además la instancia del ViewModel sobrevive a los cambios de configuración.
- **LiveData** Será la clase que contenga los datos que podemos observar, Siempre tendrá la última versión de los datos y va a notificar a los observadores cuando se produzca algún cambio, además LiveData también es consciente de los ciclos de vida de la aplicación.

Los componentes de nuestra UI solo observan estos datos y no tendrán que preocuparse de otros detalles.

Si quieres saber más acerca de guias para la arquitectura de aplicaciones puedes verlo en el siguiente enlace. https://developer.android.com/jetpack/docs/guide

¿Que vamos a construir?

Respecto al alcance de esta unidad, vamos a construir una aplicación que tenga una interfaz grafica que pueda ser actualizada con datos que nosotros generemos.

La aplicación será un contador de tragos y pedidos, con ella podremos mantener un control de cuanto hemos consumido en un pub o restaurante.

- Los productos o Ítems serán almacenados en la base de datos del dispositivo.
- Mostraremos nuestro consumo a través de un RecyclerView en la MainActivity.
- Para crear un nuevo consumo deberemos presionar el botón flotante para se abra una segunda actividad para la creación.
- En esta Pantalla modificaremos los precios y la cantidad, veremos el total en tiempo real gracias a LiveData.
- Podremos borrar todos los consumos desde un botón en la actividad principal

Imágenes de referencia

Estas son algunas pantallas de la aplicación, puedes mejorar la UI como tu quieras, solo ten cuidado de utilizar los mismos campos que mencionaremos en el código.

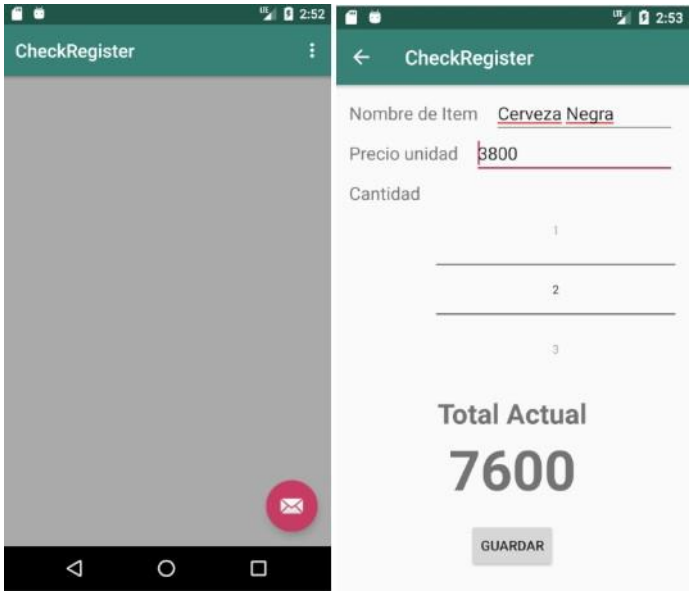


Imagen 14. Imágenes de referencia para la creación de la aplicación.

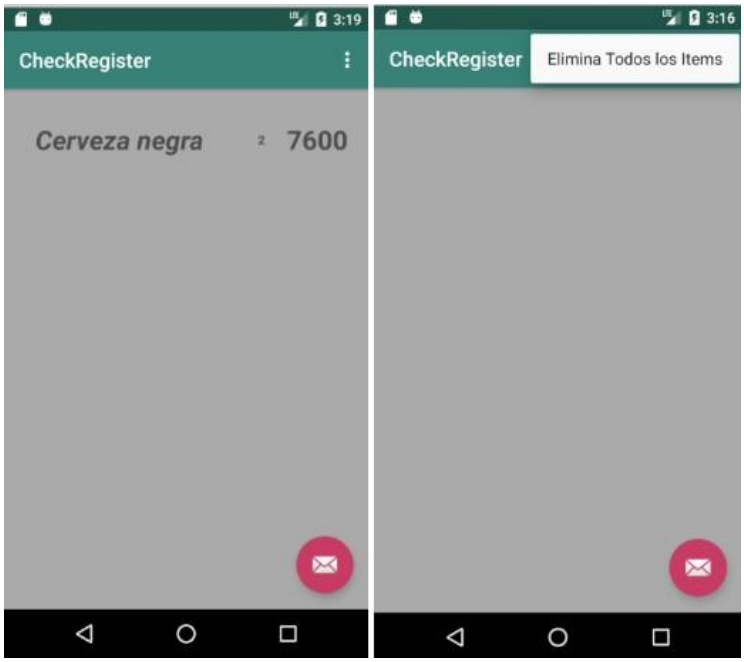


Imagen 16. Imágenes de referencia para la creación de la aplicación.

La aplicación utilizará LiveData para mostrar el listado de Items creados y también utilizará LiveData para guardar cambios en Room cambiar las vistas en base a estos.

Diagrama de la aplicación.

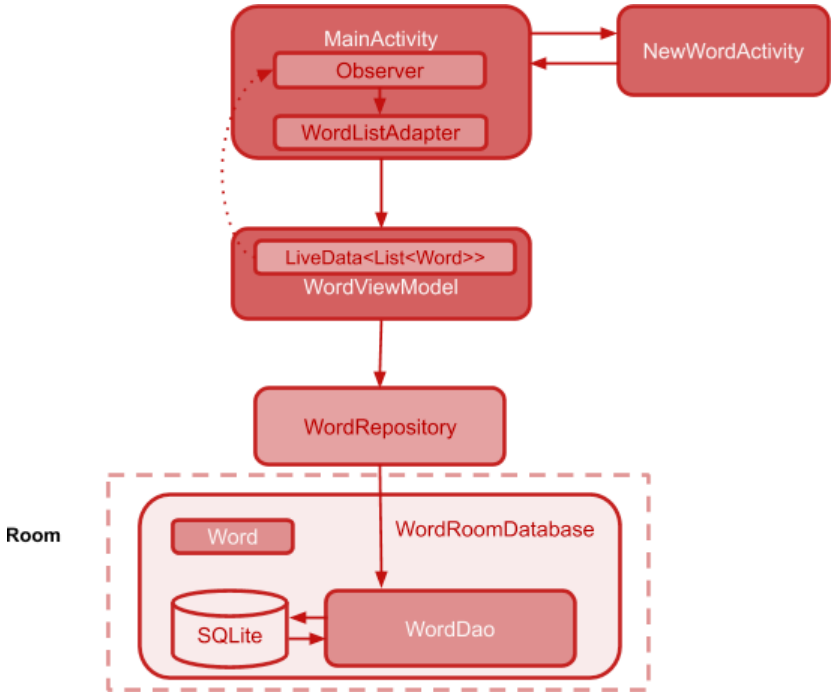


Imagen 17. Diagrama de la aplicación.

La imagen 17 obtenida de los CodeLabs oficiales de Google, resume un poco el funcionamiento de nuestra aplicación.

Como el alcance de esta unidad está centrado en el aprendizaje de LiveData, MVVM y pasar datos a la UI. Algunos componentes de la app no serán explicados con detalles porque se vieron en unidades anteriores, de igual forma todo el código estará a tu disposición.

Creando el proyecto CheckRegister

1. Creando El proyecto

a. Crea un proyecto en Android Studio.

b. Escoge una Basic Activity.

c. El lenguaje debe ser Kotlin.

d. Como Api mínima usaremos 23.

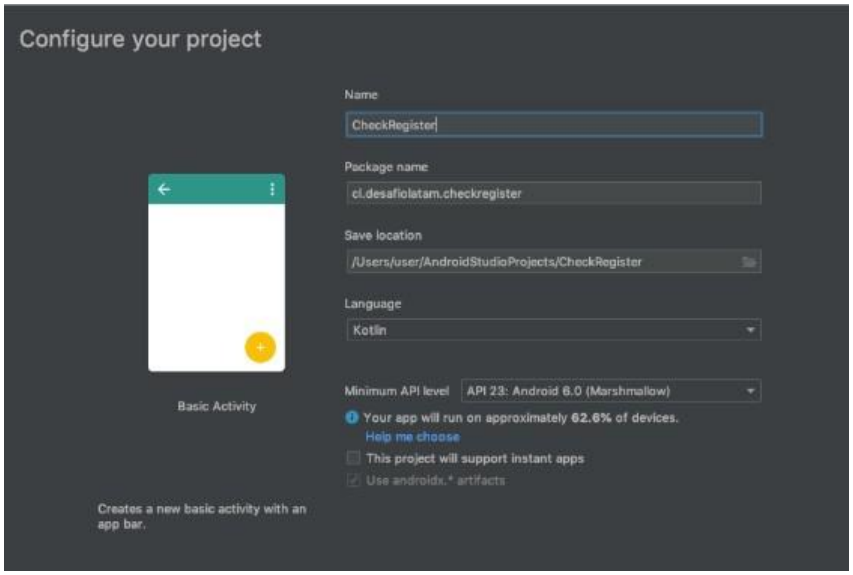


Imagen 18. Creando un nuevo proyecto.

2. Archivos Gradle y dependencias

a. Primero vamos añadir el siguiente plugin en nuestro archivo build.gradle (Module: app)

```
apply plugin: 'kotlin-kapt'
```

b. A continuación añadiremos al interior del bloque de Android lo siguiente, esto disminuirá los warning de la librería y también utilizaremos DataBinding.

```
android {

    // other configuration
   packagingOptions {
   exclude 'META-INF/atomicfu.kotlin_module'
}

dataBinding {
   enabled = true
}

}
```

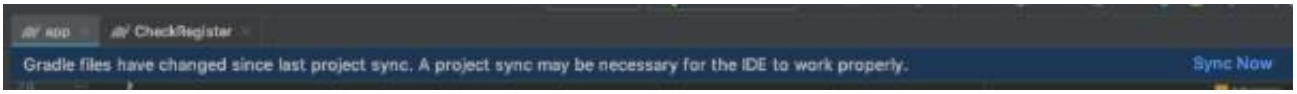
c. Ahora vamos añadir las dependencias.

```
// Room components
implementation "androidx.room:room-runtime:$rootProject.roomVersion"
implementation "androidx.room:room-ktx:$rootProject.roomVersion"
kapt "androidx.room:room-compiler:$rootProject.roomVersion"
androidTestImplementation "androidx.room:room-testing:$rootProject.roomVersion"

// Lifecycle components
implementation "androidx.lifecycle:lifecycle-
extensions:$rootProject.archLifecycleVersion"
kapt "androidx.lifecycle:lifecycle-compiler:$rootProject.archLifecycleVersion"
androidTestImplementation "androidx.arch.core:core-
testing:$rootProject.androidxArchVersion"

// ViewModel Kotlin support
implementation "androidx.lifecycle:lifecycle-viewmodel-
ktx:$rootProject.archLifecycleVersion"
```

Como puedes observar no estamos indicando un número específico en cada versión de los componentes, solo una referencia. por ejemplo “rootProject.roomVersion”, Lo dejamos de esta forma porque en nuestro archivo build.gradle (Project: CheckRegister) indicaremos las versiones de cada componente.

```
ext {
    roomVersion = '2.2.0-rc01'
    archLifecycleVersion = '2.2.0-alpha05'
    androidxArchVersion = '2.0.0'
    coroutines = '1.2.0'
}
```

Estas son las versiones de las librerías actualizadas hasta la publicación de esta lectura. No Olvides realizar la sincronización de Gradle.



3. Creación de entidades y DAO

a. Ahora vamos a crear nuestra DataClass Check la cual será la entidad que represente los datos en la base de datos.

```
@Entity(tableName = "check_items")
data class Check(
    @PrimaryKey @NonNull val name: String,
    val singlePrice: String = "",
    val quantity: Int = 0,
    val totalItem: String = "0"
)
```

Como puedes observar hemos añadido los atributos o variables name para asignar un nombre y como clave primaria, singlePrice que será el precio unitario del ítem registrado, quantity para la cantidad del ítem y totalItem que almacenará el total del ítem que estemos guardando.

b. A continuación añadiremos el DAO a nuestro proyecto, Crea un nuevo Kotlin File de nombre CheckDao y añade este código.

```
@Dao
interface CheckDao {

   @Insert(onConflict = OnConflictStrategy.REPLACE)
    fun insertCheckItem(check: Check)

   @Insert
   suspend fun updateCheckItem(check: Check)

   @Delete
   fun clearAllCheckItems(vararg creature: Check)

   @Query("SELECT * FROM check_items ORDER BY name ASC")
   fun getAllCheckItems(): LiveData<List<Check>>
}
```

Como puedes observar este Dao es igual a los que has construido anteriormente, solo tiene funciones o métodos que nos permiten insertar un nuevo elemento, hacer update, borrar todos los elementos y obtener todos los elementos.

Debes haber notado que el método getAllCheckItems( ) incorpora LiveData antes del listado de check, añadir LiveData en este método hará que Room se encarga internamente de actualizar la información desde la base de datos al objeto LiveData, cuando la base de datos cambie.

Recordemos que, cuando los datos sean cambiados, necesitamos que se actualice nuestra UI, esto quiere decir que necesitamos observar estos datos, y cuando cambien podamos reaccionar.

Más adelante en este ejemplo veremos los cambios que se producen a través de un observador en nuestro MainActivity.

4. Añadiendo Room al proyecto.

a. Debemos añadir la clase de configuración de Room. Crea un nuevo Kotlin File llamado CheckRoomDataBase y añade el siguiente código:

```
// Annotates class to be a Room Database with a table (entity) of the Check
class
@Database(entities = [Check::class], version = 1)
abstract class CheckRoomDataBase : RoomDatabase() {

   abstract fun checkDao(): CheckDao

   companion object {
       // Singleton prevents multiple instances of database opening at the
       // same time.
       @Volatile
       private var INSTANCE: CheckRoomDataBase? = null

       fun getDatabase(context: Context): CheckRoomDataBase {
           val tempInstance = INSTANCE
           if (tempInstance != null) {
               return tempInstance
           }
           synchronized(this) {
               val instance = Room.databaseBuilder(
                   context.applicationContext,
                   CheckRoomDataBase::class.java,
                   "check_database"
               ).build()
               INSTANCE = instance
               return instance
           }
       }
   }
}
```

No entraremos en detalle en esté código en particular, es solo la implementación de Room como una clase abstracta heredando desde RoomDatabase y los métodos correspondientes para inicializar un Singleton o instancia única e la base de datos.

También crearemos un archivo de Kotlin que será el punto de partida de nuestra aplicación, en el se inicializará la BBDD.

```
class CheckApplication : Application() {

   companion object {
       lateinit var database: CheckRoomDataBase
   }

   override fun onCreate() {
       super.onCreate()
       database = Room.databaseBuilder(this, CheckRoomDataBase::class.java,
"check_database")
           .build()
   }
}
```

Se llamará CheckApplication, también debemos indicarlo en nuestro Manifest.

```
<application
       android:name=".app.CheckApplication"
       android:allowBackup="true"
       android:icon="@mipmap/ic_launcher"
       android:label="@string/app_name"
       android:roundIcon="@mipmap/ic_launcher_round"
       android:supportsRtl="true"
       android:theme="@style/AppTheme">
       <activity
           android:name=".ui.MainActivity"
           android:label="@string/app_name"
           android:theme="@style/AppTheme.NoActionBar">
           <intent-filter>
               <action android:name="android.intent.action.MAIN" />
               <category android:name="android.intent.category.LAUNCHER" />
           </intent-filter>
       </activity>
   </application>
```

5. Creando el Repositorio

El repositorio será una clase abstracta que tendrá acceso a diferentes fuentes de datos. Esta clase repositorio no forma parte de los componentes de arquitectura, es solo sugerencia de buena práctica para mejorar la separación de código y la arquitectura de la aplicación.

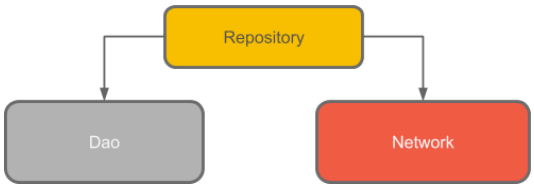


Imagen 20. Diagrama de implementación simple de un repositorio.

La imagen 2 representa una implementación simple de un repositorio, en nuestro caso solo nos conectaremos al DAO, ya que nuestro ejemplo no tiene acceso a la red o a un servidor para obtener datos, si ese fuera el caso, se utilizará el mismo repositorio.

El uso de un repositorio permite manejar las consultas a múltiples Backends o servicios, El ejemplo más recurrente es que el repositorio maneja la lógica de actualizar los datos desde un servidor o usar los datos en la base de datos local.

Integremos una clase pública CheckRepository y añade el siguiente código:

```
// Declares the DAO as a private property in the constructor. Pass in the DAO
// instead of the whole database, because you only need access to the DAO
class CheckRepository {

   private val checkDao: CheckDao = CheckApplication.database.checkDao()
   // Room executes all queries on a separate thread.
   // Observed LiveData will notify the observer when the data has changed.
   val allCheckItems: LiveData<List<Check>> = checkDao.getAllCheckItems()

   // this funtion are using AsyncTask for not block the UI
   fun insertCheckItem(checkItem: Check) {
       InsertAsyncTask(checkDao).execute(checkItem)
   }

   fun deleteAllCheckItems() {
       val creatureArray = allCheckItems.value?.toTypedArray()
       if (creatureArray != null) {
           DeleteAsyncTask(checkDao).execute(*creatureArray)
       }
   }
```

```
    private class InsertAsyncTask internal constructor(private val dao:
CheckDao) :
        AsyncTask<Check, Void, Void>() {
        override fun doInBackground(vararg params: Check): Void? {
            dao.insertCheckItem(params[0])
            return null
        }
    }

    private class DeleteAsyncTask internal constructor(private val dao:
CheckDao) :
        AsyncTask<Check, Void, Void>() {
        override fun doInBackground(vararg params: Check): Void? {
            dao.clearAllCheckItems(*params)
            return null
        }
    }



}
```

A grandes rasgos estamos pasando CheckDao a nuestra clase repositorio. De esta forma solo accederemos a los métodos que escriben o leen la base de datos, sin tener que exponer la base de datos completa al repositorio.

El listado de elementos viene inicializado al obtener el objeto LiveData que proviene desde Room, más específicamente del método getAllCheckItems. Room ejecutará esta petición en un hilo diferente al de UI y nos traerá los datos que han cambiado al main Thread.

Las operaciones de Insertar un nuevo elemento y Borrar se realizarán a través de AsyncTask, de esa forma no bloquearemos la interfaz de usuario cuando se ejecuten.

6. Creando el ViewModel

A continuación vamos a crear nuestro primer ViewModel, este se encargará de pasar la data a nuestra vista y sobrevivir a cualquier cambio de configuración. En otras palabras el ViewModel funcionara como un centro de comunicación o puente entre el repositorio y la UI, además incluso podemos utilizarlo para compartir datos entre fragmentos.

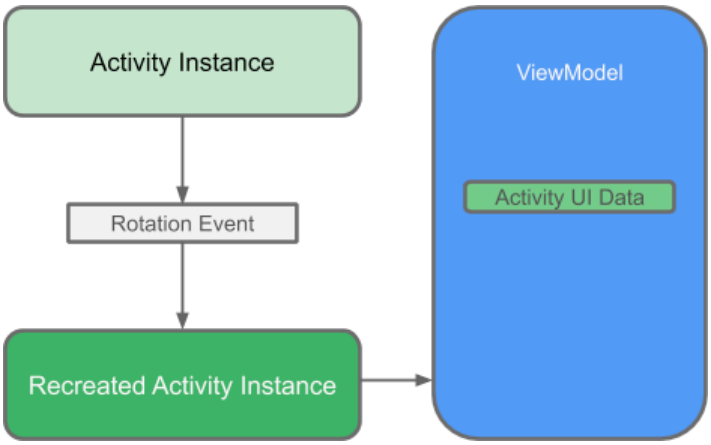


Imagen 21. Diagrama relacionando el ViewModel.

Nuestro Viewmodel contendrá los datos de la aplicación que se mostrarán en la UI, separando estos datos de nuestras actividades o fragmentos nos permitirá seguir el principio de “single responsibility” es decir de una sola responsabilidad. Ya que las vistas solo se encargan de dibujar estos datos en la pantalla mientras el ViewModel se tomará el control de procesar estos datos.

a. Implementando el ViewModel

Crea un Kotlin file llamado CheckViewModel y añade el siguiente código:

```
// Class extends AndroidViewModel and requires application as a parameter.
class CheckViewModel(application: Application) : AndroidViewModel(application)
{

   // The ViewModel maintains a reference to the repository to get data.
   private val repository: CheckRepository
   // LiveData gives us updated checkItem when they change.
   val AllCheckItems: LiveData<List<Check>>

   // Gets reference to CheckDao from CheckRoomDatabase to construct
   // the correct CheckRepository.
   init {
       val checkDao = CheckRoomDataBase.getDatabase(application).checkDao()
       repository = CheckRepository()
       AllCheckItems = repository.allCheckItems
   }
```

```
    fun insertCheckItem(checkItem: Check) {
        repository.insertCheckItem(checkItem)
    }

    fun deleteAllCheckItem() {
        repository.deleteAllCheckItems()
    }
}
```

En este código creamos la clase llamada CheckViewModel la cual tiene como parámetro Application y hereda AndroidViewModel.

Tenemos una variable privada que referencia nuestro repositorio, también añadimos una variable de LiveData para mantener un caché de el listado de Items de la cuenta.

También creamos un bloque init el cual hace referencia a nuestro CheckDao desde CheckRoomDatabase. En este mismo bloque inicializamos AllCheckItems LiveData usando el repositorio.

Luego tenemos las funciones que realizarán las tareas de insertar un nuevo item o borrarlos todos. Si necesitaramos realizar otra tarea que necesito acceso a la BBDD, este es el archivo.

7. Lógica de negocio

Ahora vamos a crear la clase que se encargará de manejar los cálculos que se realizarán a los ítem que vamos a ir añadiendo. El ejemplo es bastante básico pero nos permitirá tener una visión de cómo realizar gestiones más completas siguiente una estructura.

```
class CheckGeneratorTotal {

    fun generateCheckTotal(
                            name: String = "",
                            singlePrice: String = "",
                            quantity: Int = 0) : Check {
        val total = quantity * singlePrice.toInt()
        return Check(name, singlePrice, quantity, total.toString())
    }
}
```

La lógica es ir multiplicando por la cantidad el valor unitario para obtener el total del item añadido.

8. Organización de package del proyecto

a. Organizaremos un poco el proyecto

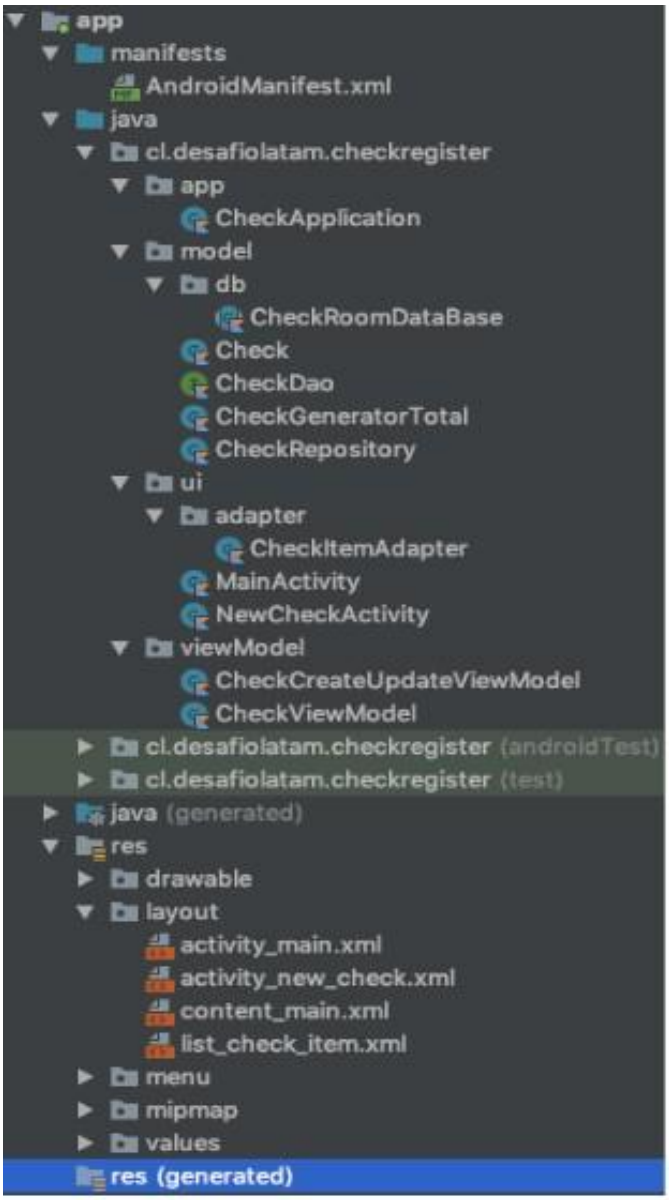


Imagen 22. Estructura del proyecto.

Creamos cuatro package principales al interior de nuestro proyecto, “app”, “model”, “ui” y “viewModel”. Refactorizamos moviendo las clases que hemos creado a sus respectivas carpetas.

En la imagen hay archivos que aún no has creado, tranquilo en los siguientes pasos se crearán en su totalidad.

9. Creando la Interfaz Gráfica.

A continuación crearemos los elementos que ya conocemos, Parte de la UI , layouts de ítems para el recyclerView y también el adapter que mostrará todos los items.list_check_item.xml

```
?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:app="http://schemas.android.com/apk/res-auto"
   android:layout_width="match_parent"
   android:layout_height="100dp"
   android:orientation="vertical">

   <TextView
       android:id="@+id/title_item"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:layout_marginStart="32dp"
       android:text="title"
       android:textSize="26sp"
       android:textStyle="italic|bold"
       app:layout_constraintBottom_toBottomOf="parent"
       app:layout_constraintStart_toStartOf="parent"
       app:layout_constraintTop_toTopOf="parent" />

   <TextView
       android:id="@+id/quantity_item"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:layout_marginStart="48dp"
       android:text="Cantidad"
       android:textSize="12sp"
       android:textStyle="bold"
       app:layout_constraintBottom_toBottomOf="parent"
      app:layout_constraintEnd_toStartOf="@+id/total_item"
       app:layout_constraintStart_toEndOf="@+id/title_item"
       app:layout_constraintTop_toTopOf="parent" />
```

```
    <TextView
        android:id="@+id/total_item"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="16dp"
        android:layout_marginEnd="16dp"
        android:text="Total"
        android:textSize="26sp"
        android:textStyle="bold"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toEndOf="@+id/quantity_item"
        app:layout_constraintTop_toTopOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

Tambien vamos a añadir el recyclerView en activity_main.xml

```
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:app="http://schemas.android.com/apk/res-auto"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   app:layout_behavior="@string/appbar_scrolling_view_behavior"
   tools:context=".ui.MainActivity"
   tools:showIn="@layout/activity_main">

   <androidx.recyclerview.widget.RecyclerView
       android:id="@+id/check_item_recyclerview"
       android:layout_width="match_parent"
       android:layout_height="match_parent"
       android:background="@android:color/darker_gray"
       tools:listitem="@layout/list_check_item" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Este será nuestro Adaptador para mostrar los elementos. CheckItemAdapter.kt

```
class CheckItemAdapter(
   private val checkItems: MutableList<Check>
) : RecyclerView.Adapter<CheckItemAdapter.CheckItemViewHolder>() {

   private var checkItem = emptyList<Check>() // Cached copy of checkItem
   inner class CheckItemViewHolder(itemView: View) :
RecyclerView.ViewHolder(itemView) {
       val checkItemView: TextView = itemView.findViewById(R.id.title_item)
       val checkQuantityItem: TextView =
itemView.findViewById(R.id.quantity_item)
       val checkTotalItem: TextView = itemView.findViewById(R.id.total_item)
   }
   override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
CheckItemViewHolder {
       val view =
LayoutInflater.from(parent.context).inflate(R.layout.list_check_item, parent,
false)
       return CheckItemViewHolder(view)
   }
   override fun onBindViewHolder(holder: CheckItemViewHolder, position: Int) {
       val current = checkItems[position]
       holder.checkItemView.text = current.name
       holder.checkQuantityItem.text = current.quantity.toString()
       holder.checkTotalItem.text = current.totalItem
   }
   internal fun updateCheckItems(checkItems: List<Check>) {
       this.checkItems.clear()
       this.checkItems.addAll(checkItems)
       notifyDataSetChanged()
   }
   override fun getItemCount() = checkItems.size
}
```

10. Añadiendo los observadores en Main Activity

Ahora vamos a terminar la primera parte de esta aplicación añadiendo los observadores de LiveData en la MainActivity.

```
private lateinit var viewModel: CheckViewModel
   private val adapter = CheckItemAdapter(mutableListOf())

   override fun onCreate(savedInstanceState: Bundle?) {
       super.onCreate(savedInstanceState)
       setContentView(R.layout.activity_main)
       setSupportActionBar(toolbar)
       viewModel = ViewModelProviders.of(this).get(CheckViewModel::class.java)

       val recyclerView = findViewById<RecyclerView>
(R.id.check_item_recyclerview)
       recyclerView.adapter = adapter
       recyclerView.layoutManager = LinearLayoutManager(this)


       fab.setOnClickListener { view ->
           startActivity(Intent(this, NewCheckActivity::class.java))
       }

       viewModel.AllCheckItems.observe(this, Observer { checkItem ->
           checkItem?.let {
               adapter.updateCheckItems(it)
           }
       })
   }
   override fun onCreateOptionsMenu(menu: Menu): Boolean {
       menuInflater.inflate(R.menu.menu_main, menu)
       return true
   }
   override fun onOptionsItemSelected(item: MenuItem): Boolean {

       return when (item.itemId) {
           R.id.action_settings -> {
               viewModel.deleteAllCheckItem()
               true
           }
           else -> super.onOptionsItemSelected(item)
       }
   }
}
```

Como puedes observar, estamos observando los datos que están presentes en el ViewModel, que a su vez son obtenidos desde el repositorio.

Si ejecutamos nuestra aplicación aún no podremos ver los cambios, ya que nuestra base de datos está vacía, En el próximo capítulo, vamos añadir una nueva actividad y también a travès de una combinación entre LiveData, ViewModel y databinding, veremos cambios en tiempo real en nuestra Interfaz de usuario, además cuando hayamos creado nuestros items, de forma automática se actualiza el recyclerView con los datos más nuevos.

Actualizando datos en la UI

Competencias

- Observar LiveData en el proyecto.
- Construir una segunda actividad y ViewModel
- Actualizar la UI del proyecto con LiveData

Introducción

Previamente, logramos dejar nuestra aplicación lista para mostrar datos que provengan directamente desde la BBDD a la UI a través de Livedata. Estos datos que van directo al adapter, no pueden ser modificados, ya que la clase LiveData es una instancia abstracta que no tiene métodos para ser modificable.

Afortunadamente, se creó una clase llamada MutableLiveData, es a través de esta que podemosmodificar los valores provenientes de estos objetos gracias a métodos públicos que nos facilitaran las tareas.

La idea es que nuestra aplicación contenga una nueva actividad para ingresar Ítems, a medida que se llenan los datos, podremos saber el cálculo de la cantidad de dinero que saldrá nuestros consumo.

A continuación terminaremos la aplicación que te puede servir de base para otros proyectos.

1. **Añadir una nueva actividad al Proyecto**

Primero vamos a crear una nueva actividad, para eso a través de android studio añade una segunda actividad al proyecto. Utiliza el template de “Empty Activity”.

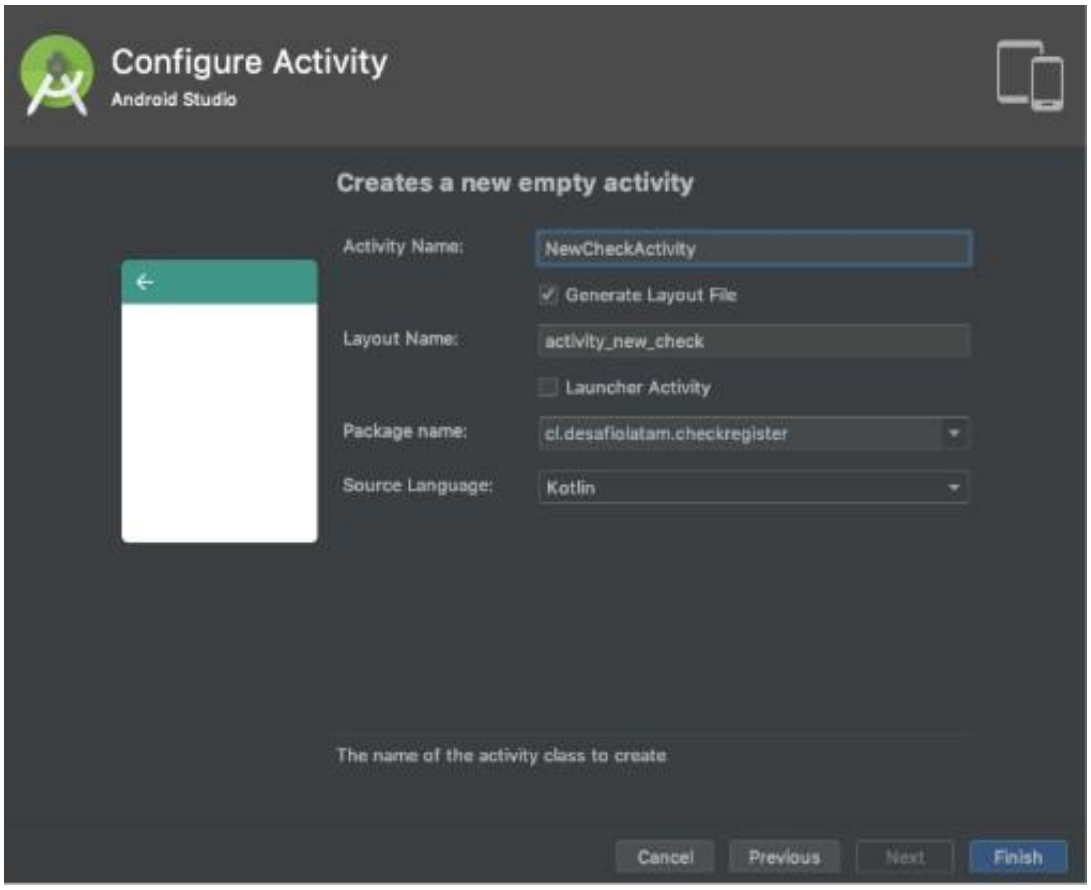


Imagen 1.

Revisa que la actividad nueva se haya registrado correctamente en el archivo de Manifest.xml y que su parent activity sea MainActivity.

```
<manifest  xmlns:android="http://schemas.android.com/apk/res/android"
   package="cl.desafiolatam.checkregister">

   <application
       android:name=".app.CheckApplication"
       android:allowBackup="true"
       android:icon="@mipmap/ic_launcher"
       android:label="@string/app_name"
       android:roundIcon="@mipmap/ic_launcher_round"
       android:supportsRtl="true"
       android:theme="@style/AppTheme">
```

```
        <activity
            android:name=".ui.MainActivity"
            android:label="@string/app_name"
            android:theme="@style/AppTheme.NoActionBar">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:name=".ui.NewCheckActivity"
            android:parentActivityName=".ui.MainActivity">
        </activity>
    </application>
</manifest>
```

2. Modificar el Layout que viene por defecto.

Ahora vamos a crear una interfaz gráfica para que los usuarios de la aplicación puedan ingresar el consumo. Modifica el archivo activity_new_check.xml.

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

    <data>
        <variable
            name="viewModel"
            type="cl.desafiolatam.checkregister.viewModel.
                   CheckCreateUpdateViewModel" />
    </data>
    <androidx.constraintlayout.widget.ConstraintLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:context=".ui.NewCheckActivity">

        <TextView
            android:id="@+id/nameLabel"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_margin="16dp"
            android:text="Nombre de Item"
            android:textSize="18sp"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toTopOf="parent" />
```

```
<EditText
    android:id="@+id/nameEditText"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_margin="16dp"
    android:hint="Ingresa el nombre del Item"
    android:inputType="text"
    android:text="@={viewModel.name}"
    app:layout_constraintBaseline_toBaselineOf="@+id/nameLabel"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toEndOf="@+id/nameLabel" />

<TextView
    android:id="@+id/unitPriceLabel"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_margin="16dp"
    android:text="Precio unidad"
    android:textSize="18sp"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/nameLabel"  />

<EditText
    android:id="@+id/unitPriceEditText"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_margin="16dp"
    android:inputType="number"
    android:text="@={viewModel.singlePrice}"
    app:layout_constraintBaseline_toBaselineOf="@+id/unitPriceLabel"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toEndOf="@+id/unitPriceLabel" />

<TextView
    android:id="@+id/quantityLabel"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_margin="16dp"
    android:text="Cantidad"
    android:textSize="18sp"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/unitPriceLabel"  />

<NumberPicker
    android:id="@+id/quantitySpinner"
```

```
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_margin="16dp"
            app:layout_constraintBaseline_toBaselineOf="@+id/quantityLabel"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toEndOf="@+id/quantityLabel"/>

        <TextView
            android:id="@+id/hitPointsLabel"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_margin="16dp"
            android:text="Total Actual"
            android:textSize="28sp"
            android:textStyle="bold"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toBottomOf="@+id/quantitySpinner"
            tools:text="Total actual" />

        <TextView
            android:id="@+id/total_item"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@={``+ viewModel.total}"
            android:textSize="56sp"
            android:textStyle="bold"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toBottomOf="@+id/hitPointsLabel"  />

        <Button
            android:id="@+id/saveButton"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_margin="16dp"
            android:onClick="@{() -> viewModel.saveCheck()}"
            android:text="Guardar"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toBottomOf="@+id/total_item"  />
    </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

Estamos utilizando un “layout” porque usaremos DataBinding para comunicar directamente nuestra vista con nuestro ViewModel.

El preview de este archivo debería verse así.

Imagen 2. Preview de la aplicación.

La idea es que el usuario pueda ingresar el nombre de un Ítem, un valor inicial y luego seleccionar a través de un Spinner la cantidad de su consumo.

3. Agregar un Nuevo ViewModel

En el capítulo anterior utilizamos un ViewModel para mostrar los datos provenientes de la BBDD en nuestra vista, también a través de él podemos insertar y eliminar los elementos. Como Buena práctica, Para que nuestro proyecto sea más modular, escalable, vamos a generar un nuevo ViewModel para manejar los datos de la vista de la nueva actividad y añadir un nuevo ítem con sus respectivos atributos.

```
 class CheckCreateUpdateViewModel(
   private val generator: CheckGeneratorTotal = CheckGeneratorTotal(),
   private val checkRepository: CheckRepository = CheckRepository()
) : ViewModel() {

   private val checkItemLiveData = MutableLiveData<Check>()
   private val saveLiveData = MutableLiveData<Boolean>()
   fun getCheckLiveData(): LiveData<Check> = checkItemLiveData
   fun getSaveLiveData(): LiveData<Boolean> = saveLiveData

   var name = ObservableField<String>("")
   var singlePrice = ObservableField<String>()
   var total = 0
   var quantity = 0
   lateinit var checkItem: Check

   fun updateCheckItem() {
//        val attributes = CreatureAttributes(intelligence, strength,
endurance)
       checkItem = generator.generateCheckTotal(
           name.get() ?: "",
           singlePrice.get() ?: "",
           quantity
       )
       checkItemLiveData.postValue(checkItem)
   }

   fun quantitySelected(quantitySelected: Int) {
       quantity = quantitySelected
       updateCheckItem()
   }

   fun saveCheck() {
       checkRepository.insertCheckItem(checkItem)
       saveLiveData.postValue(true)
   }
}
```

Para que esta vez podamos modificar los datos provenientes de liveData, vamos a utilizar las clases MutableLiveData, las cuales nos permitiran a través de sus métodos públicos, poder añadir datos a los objeto LiveData. Observa también, que hemos creado métodos para hacer Update a los cálculos de los items(updateCheckItem) y también para guardar (saveCheck). Recuerda que las vistas están unidas por DataBinding.

Solo nos queda Unir este ViewModel a la vista correspondiente.

4. **Observando unidireccionalmente con LiveData en la actividad NewCheckActivity.**

```
 class NewCheckActivity : AppCompatActivity() {
   private lateinit var viewModel: CheckCreateUpdateViewModel
   lateinit var binding: ActivityNewCheckBinding

   override fun onCreate(savedInstanceState: Bundle?) {
       super.onCreate(savedInstanceState)
       binding = DataBindingUtil.setContentView(this,
           R.layout.activity_new_check
       )
       viewModel =
ViewModelProviders.of(this).get(CheckCreateUpdateViewModel::class.java)
       binding.viewModel = viewModel
       observableIntVariable()
       configureLiveDataObservers()
   }

   private fun observableIntVariable() {
       quantitySpinner.maxValue = 50
       quantitySpinner.minValue = 0
       quantitySpinner.wrapSelectorWheel = true
       quantitySpinner.setOnValueChangedListener { numberPicker, i, i2 ->
           viewModel.quantitySelected(numberPicker.value)
       }
   }

   private fun configureLiveDataObservers() {
       viewModel.getCheckLiveData().observe(this, Observer { checkItem ->
           checkItem?.let {
               total_item.text = it.totalItem
               nameEditText.setText(checkItem.name)
               unitPriceEditText.setText(checkItem.singlePrice)
           }
       })

       viewModel.getSaveLiveData().observe(this, Observer { saved ->
           saved?.let {
```

```
                if (it) {
                    Toast.makeText(this, "Guardado Exitosamente",
Toast.LENGTH_SHORT).show()
                    finish()
                } else {
                    Toast.makeText(this, "Error al guardar",
Toast.LENGTH_SHORT).show()
                }
            }
        })
    }
}
```

El método observableIntVariable controla el numberPicker y cada vez que se cambia su valor realiza una llamada al método del viewModel que pasa el valor para realizar el cálculo.

Si ejecutamos nuestra aplicación podremos añadir Items, calcular su precio en tiempo real, guardar estos datos en la persistencia del teléfono, y todos lo estamos realizando con un patrón de diseño robusto y además potenciado por los patrones de arquitectura.

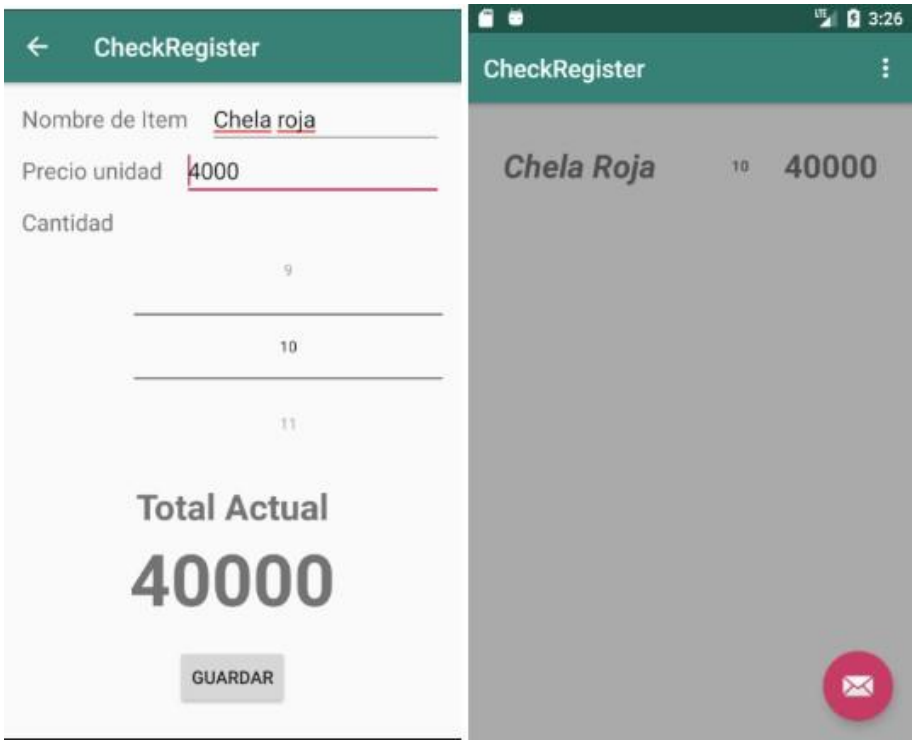


Imagen 3. Agregando items.

Con esto concluimos la unidad de LiveData y UI. Para seguir profundizando estos temas, te recomendamos ver los ejemplos CodeLabs de google para profundizar tus conocimientos.

Unidad 3

API REST y acceso a recursos remotos

3.1 API REST

Cliente - Servidor

Competencias:

- Identificar que es un cliente.
- Identificar que es un servidor.
- Comprender la integración entre un cliente y un servidor.

Introducción

Internet ha traído un cambio revolucionario en el ámbito de las tecnologías, interconectando a el mundo entero. En este contexto, las tecnologías de comunicación en internet siguen una arquitectura y una estructura específica para conectarse entre sí. La más popular es la arquitectura cliente-servidor, la cual profundizaremos en este capítulo.

Es fundamental entender estos conceptos básicos de programación moderna, dado que hoy por hoy la gran mayoría de los sistemas y aplicaciones que se construyen siguen un standard de arquitectura basada en el concepto cliente-servidor.

Arquitectura cliente-servidor

La arquitectura cliente-servidor se denomina estructura de computación de red porque cada solicitud y sus servicios asociados se distribuyen a través de una red privada o pública en internet.

En la arquitectura cliente-servidor, cuando la computadora cliente envía una solicitud de datos al servidor a través de Internet, el servidor acepta la solicitud, la procesa y entrega los paquetes de datos solicitados al cliente. Una característica especial es que la computadora servidor tiene el potencial de administrar numerosos clientes al mismo tiempo. Además, un solo cliente puede conectarse a numerosos servidores, donde cada servidor proporciona un conjunto diferente de servicios a ese cliente específico.

Imagen 1. Arquitectura clientes-servidor.

Las computadoras cliente proporcionan una interfaz para permitir que un usuario de la computadora solicite servicios del servidor y muestre los resultados que el servidor devuelve. Los servidores esperan que lleguen las solicitudes de los clientes y luego responden a ellos. Idealmente, un servidor proporciona una interfaz transparente estandarizada a los clientes para que los clientes no necesiten conocer los detalles del sistema (es decir, el hardware y el software) que proporciona el servicio.

Los clientes a menudo se encuentran en estaciones de trabajo o en computadoras personales, mientras que los servidores se encuentran en otras partes de la red, generalmente son las máquinas más potentes. Este modelo de computación es especialmente efectivo cuando los clientes y el servidor tienen tareas distintas que realizan rutinariamente.

Imagen 2. Representación de modelo de comunicación Request - Response.

En el procesamiento de datos de un hospital, por ejemplo, una computadora cliente puede ejecutar un programa de aplicación para ingresar información del paciente mientras la computadora del servidor ejecuta otro programa que administra la base de datos en la que la información se almacena permanentemente. Muchos clientes pueden acceder a la información del servidor simultáneamente y, al mismo tiempo, una computadora cliente puede realizar otras tareas, como enviar un correo electrónico.

Cliente

En una arquitectura cliente-servidor de una red computacional el cliente se puede definir como un procesador remoto que recibe y envía peticiones de un servidor. Cada teléfono móvil android desde su ubicación en el planeta es un cliente para cada una de las empresas o servicios que prestan diferentes aplicaciones desde sus servidores principales.

Imagen 3. Dispositivos cliente frecuentes.

Principales Funciones de un cliente

- Iniciar el request o solicitud.
- Esperar y recibir respuestas.
- Poder conectarse con un servidor o múltiples servidores al mismo tiempo.
- Interactuar con los usuarios finales usando una Graphical User Interface (GUI)

Servidor

Un servidor es un computador de red, programa o dispositivo que procesa solicitudes de un cliente. En la internet hoy por hoy un servidor web es una computadora o una red de computadoras que utiliza el protocolo HTTP para enviar datos y archivos a la computadora de un cliente cuando el este último lo solicita.

En el caso de una red de área local, por ejemplo, un servidor de impresoras administra una o más impresoras e imprime los archivos que le envían los equipos cliente. Los servidores de red (que administran el tráfico de red) y los servidores de archivos (que almacenan y recuperan archivos para clientes) son dos ejemplos mas de servidores.

Imagen 4. Google datacenter (servidores).

Principales Funciones de un Servidor

- Esperar por los requerimientos de un cliente.
- Recibido un requerimiento, procesar, preparar y enviar las respuestas.
- Aceptar múltiples conexiones de una largo número de clientes.
- Normalmente, no interactuar directamente con los usuarios finales.

Cliente-Servidor en la actualidad

Entendiendo los conceptos básicos anteriores, entremos en el escenario de la tecnología actual con otro ejemplo, el requerimiento de los datos meteorológicos de hoy para nuestra ciudad.

Antes de la existencia de las computadoras, las personas recibían estas noticias meteorológicas a través del diario(periódico) o lo hacían escuchando la radio o televisión transmitiendo noticias del clima.

En el mundo moderno con computadoras y servidores, los usuarios, que buscan el informe meteorológico a través de sus dispositivos móviles enviando peticiones a través de direcciones web (URL) que el servidor provee para la ejecución de un servicio o tarea específico, proporcionando información actualizada del clima.

Por lo tanto, los clientes y servidores son dos computadoras diferentes en diferentes partes del mundo que están conectadas a través de Internet, por lo tanto, como usuarios, podemos bajar una aplicación de consulta del clima y navegar por el informe meteorológico, gracias a los servicios expuestos desde los servidores de la aplicación.

Algunos factores adicionales entran en acción con tales tecnologías de recopilación de datos en línea. Un periódico o radio usa su idioma local para darle el informe meteorológico y otra información; sin embargo, para la arquitectura cliente-servidor en la web, se deben considerar factores específicos:

- Un conjunto específico de idiomas junto con un estándar de comunicación, exclusivamente un protocolo para la interacción de dos sistemas. Los más populares son HTTP y HTTPS (Protocolo seguro de transferencia de hipertexto).
- Mecanismo y protocolo para solicitar los aspectos requeridos del servidor. Eso podría estar en cualquier estructura de datos formateados. Los formatos más implementados y populares se realizan en XML y JSON.
- Respuesta del servidor enviando en una estructura de datos formateados (generalmente XML o JSON).

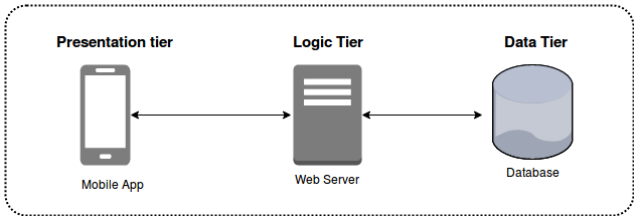


Imagen 5. Representación arquitectura moderna tradicional.

API Rest

Competencias:

- Conocer qué es un api rest.
- Comprender cómo interactúa un api rest con las aplicaciones android.
- Diferenciar los métodos para consumir e interactuar con una API Rest.

Introducción

En este capítulo estudiaremos los conceptos asociados a un api REST y aprenderemos a trabajar con este estilo de arquitectura distribuida muy popular hoy por hoy. REST es una tecnología que utiliza un conjunto de protocolos y estándares para intercambiar datos entre aplicaciones.

Al consumir servicios web REST desde nuestras aplicaciones somos capaces de desarrollar sistemas móviles multi-usuario, masivos y de largo alcance, entendiendo que desde el lugar en que se encuentre un cliente android, este se puede conectar con servicios centralizados expuestos con la tecnología rest.

En la actualidad casi no existen proyectos o aplicaciones que no disponga de una API REST para la creación de servicios profesionales. WhatsApp, YouTube, Twitter, Facebook y cientos de miles de empresas generan negocio gracias a REST y las APIs REST, sin esto, el crecimiento en horizontal sería prácticamente imposible y REST lo facilita gracias a que es el estándar más lógico, eficiente y habitual en la creación de APIs para servicios de Internet.

Conocer cómo funcionan estos servicios proporcionará al alumno herramientas para trabajar en cualquier empresa que ofrezca servicios rest, o necesite que las aplicaciones se conecten a un tercero.

Api REST

Una api REST simplifica el desarrollo al admitir métodos HTTP estándar, manejo de errores y otras convenciones RESTful. Un api REST expone recursos a través de una o distintas direcciones web (URL) utilizando arquitectura basada en HTTP, exponiendo servicios web para el procesamiento de distintas tareas y operaciones.
REST es el acrónimo de Representational State Transfer, un estilo arquitectónico para sistemas hipermedia distribuidos que fue presentado por primera vez por Roy Fielding en el año 2000.

Al igual que cualquier otro estilo arquitectónico, REST también tiene sus propias seis(6) restricciones guía que deben cumplirse si una interfaz necesita ser referida como una interfaz RESTful.

REST le pide a los desarrolladores que usen métodos HTTP explícitamente y de manera consistente con la definición del protocolo. Este principio de diseño REST básico establece un mapeo uno a uno entre las operaciones de creación, lectura, actualización y eliminación (CRUD) y los métodos HTTP.

Principios REST

1. **Cliente-servidor:** al separar las preocupaciones de la interfaz de usuario de las preocupaciones de almacenamiento de datos, mejoramos la portabilidad de la interfaz de usuario en múltiples plataformas y mejoramos la escalabilidad al simplificar los componentes del servidor.
2. **Sin estado (Stateless):** cada solicitud del cliente al servidor debe contener toda la información necesaria para comprender la solicitud y no puede aprovechar ningún contexto almacenado en el servidor. Por lo tanto, el estado de la sesión se mantiene completamente en el cliente.
3. **Caché:** las restricciones de caché requieren que los datos dentro de una respuesta a una solicitud se etiqueten implícita o explícitamente como almacenables o no almacenables. Si una respuesta es almacenable en caché, se le otorga a un caché de cliente el derecho de reutilizar esos datos de respuesta para solicitudes equivalentes posteriores.
4. **Interfaz uniforme:** al aplicar el principio de generalidad de ingeniería de software a la interfaz del componente, se simplifica la arquitectura general del sistema y se mejora la visibilidad de las interacciones. Para obtener una interfaz uniforme, se necesitan múltiples restricciones arquitectónicas para guiar el comportamiento de los componentes. REST está definido por cuatro restricciones de interfaz:
    - Identificación de recursos.
    - Manipulación de recursos a través de representaciones.
    - Mensajes autodescriptivos.
    - Hipermedia como motor del estado de la aplicación.
5. **Sistema en capas:** el estilo de sistema en capas permite que una arquitectura se base en capas jerárquicas al restringir el comportamiento de los componentes de modo que cada componente no pueda ver más allá de la capa inmediata con la que están interactuando.
6. **Código bajo demanda (opcional):** REST permite ampliar la funcionalidad del cliente descargando y ejecutando código en forma de scripts. Esto simplifica a los clientes al reducir la cantidad de funciones que se deben implementar previamente.

REST - CRUD

REST significa REpresentational State Transfer y describe recursos (en nuestro caso, URL) en los que podemos realizar acciones.

CRUD significa Crear, Leer, Actualizar, Eliminar, son las acciones que realizamos. Aunque, REST y CRUD son los mejores amigos, los dos pueden funcionar bien por sí mismos. De hecho, cada vez que programamos un sistema que permite agregar, editar y eliminar elementos de la base de datos, y una interfaz que permite que viajen los datos de estos elementos; hemos estado trabajando con CRUD sin saberlo.

3.1 HTTP Request y Threads

HTTP Response Codes

Si intentamos acceder a una página que no existe, el servidor te devolverá la página 404 No encontrada o error 404, si arruinamos nuestro código rest del servidor, obtendremos un error 500 de sistemas. Estos son códigos de respuesta que el servidor envía para que su navegador sepa lo que está sucediendo.

Tipos de Respuesta

- Entre 200-299 significa que la solicitud fue exitosa.
- 300-399 significa que la solicitud estuvo bien, pero debe hacer otra cosa.
- 400-499 es un error dónde no se encuentra algún objeto o recurso, o no se ha invocado correctamente.
- 500-599 es un error realmente malo.

Código 200: Obtendrá de una solicitud GET, PUT o DELETE. Significa que la solicitud se verificó y se tomaron las medidas apropiadas.

Código 201 Creado: Le notifica que el comando POST creó correctamente el objeto publicado.

Código 404 No encontrado: Significa que no se encontró el recurso de solicitud. Puede obtener esto de GET, PUT o DELETE.

Código 406 no aceptable: El verbo no está permitido en los recursos que solicitó (Más en esta próxima parte)

Código 500 Error interno de servidor: Algo sucedió terriblemente malo en el código del sistema

Operaciones REST

Para comprender la forma en que funciona la arquitectura REST es fundamental conocer los métodos HTTP, por medio de los cuales le indicamos al servidor la forma en que debe de tratar una determinada petición, dicho esto, una misma URL puede ser tratada de forma diferente por el servidor.

HTTP define una gran cantidad de métodos que son utilizados en diferentes circunstancias. Los más relevantes para la construcción de servicios REST son los siguientes:

- **GET:** Es utilizado únicamente para consultar información al servidor, muy parecidos a realizar un SELECT a la base de datos. No soporta el envío del payload.
- **POST:** Es utilizado para solicitar la creación de un nuevo registro, es decir, algo que no existía previamente, es decir, es equivalente a realizar un INSERT en la base de datos. Soporta el envío del payload.
- **PUT:** Se utiliza para actualizar por completo un registro existente, es decir, es parecido a realizar un UPDATE a la base de datos. Soporta el envío del payload.
- **PATCH:** Este método es similar al método PUT, pues permite actualizar un registro existente, sin embargo, este se utiliza cuando actualizar solo un fragmento del registro y no en su totalidad, es equivalente a realizar un UPDATE a la base de datos. Soporta el envío del payload.
- **DELETE:** Este método se utiliza para eliminar un registro existente, es similar a DELETE a la base de datos. No soporta el envío del payload.
- **HEAD:** Este método se utilizar para obtener información sobre un determinado recurso sin retornar el registro. Este método se utiliza a menudo para probar la validez de los enlaces de hipertexto, la accesibilidad y las modificaciones recientes.

HTTP VERBS (Métodos de solicitud)

Cualquier desarrollador web que haya tenido que lidiar con datos de formularios conocerá los métodos GET y POST. El primero enviará datos al servidor a través de una cadena de consulta que se parece a esto “direccion?Key2=value2&key2=value2” y el segundo envía los datos a través de encabezados HTTP. Lo que quizás no nos hemos dado cuenta es que cada vez que carga una página (que no es una forma de tipo POST) está realiza una solicitud de tipo GET. Entonces, cuando hacemos clic en el enlace de nuestra preferencia, estaremos haciendo una solicitud GET.

Hay otros dos verbos de los que quizás no hayas oído hablar porque los navegadores no los admiten: PUT y DELETE. La acción de eliminar es obvia, si le indica al servidor que elimine algo. La instrucción PUT es un poco más compleja, es lo mismo que la acción POST desde el punto de vista que enviar datos a través de encabezados, pero está diseñada para modificar o actualizar datos o archivos.

Ejercicio 1:

Descargar e instalar postman y utilizarlo para el consumo de métodos REST del api público fake JsonPlacesHolder.

Postman es una plataforma de colaboración para el desarrollo ágil de APIs. Las características de Postman simplifican cada paso de la creación de una API y agilizan la colaboración para que podamos crear más y mejores APIs.

1. Descargar e instalar postman desde la siguiente url: https://www.getpostman.com/downloads/
2. Ir a la web de JsonPlaceHolder https://jsonplaceholder.typicode.com, para utilizar los métodos REST de su api con data de prueba. Utilizaremos los datos de posts para este ejemplo como lo muestra la siguiente imagen:

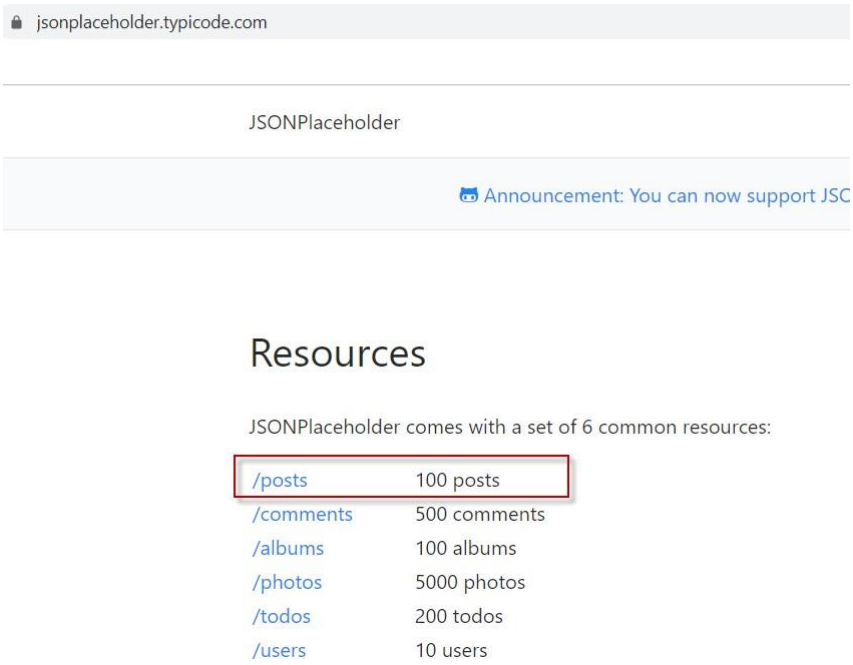


Imagen 6. JSONPlaceholder.

3. Abrir el programa y presionar nuevo, seguido de request como lo muestra la siguiente imagen:

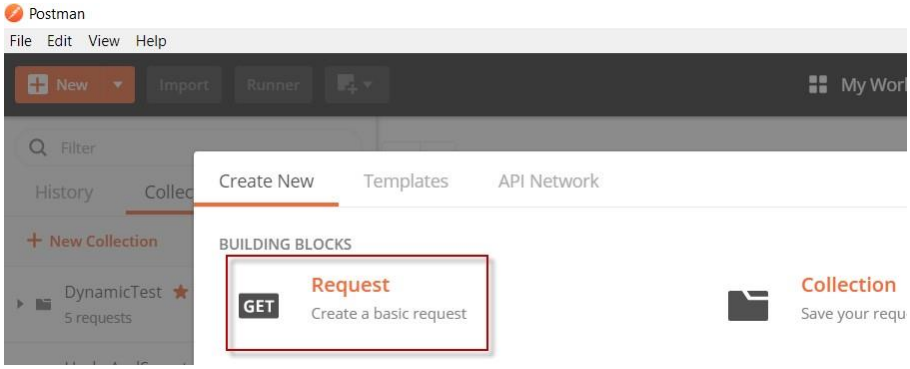


Imagen 7. Request.

4. Acto seguido, indicar un nombre a la operación de consultar posts y crear la colección Desafío Latam como se muestra en la siguiente imagen:

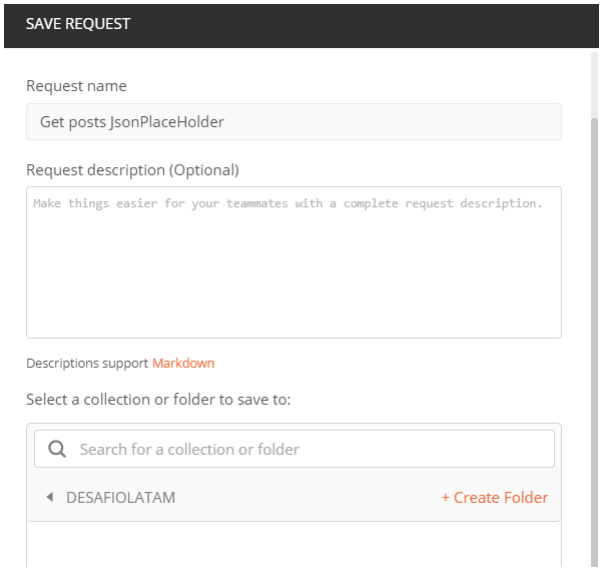


Imagen 8. Operación a consultar.

5. Copiando la url de la consulta de posts de la página tendremos una vista de detalle y rápido acceso a la operación en Postman que se visualizará de la siguiente manera:

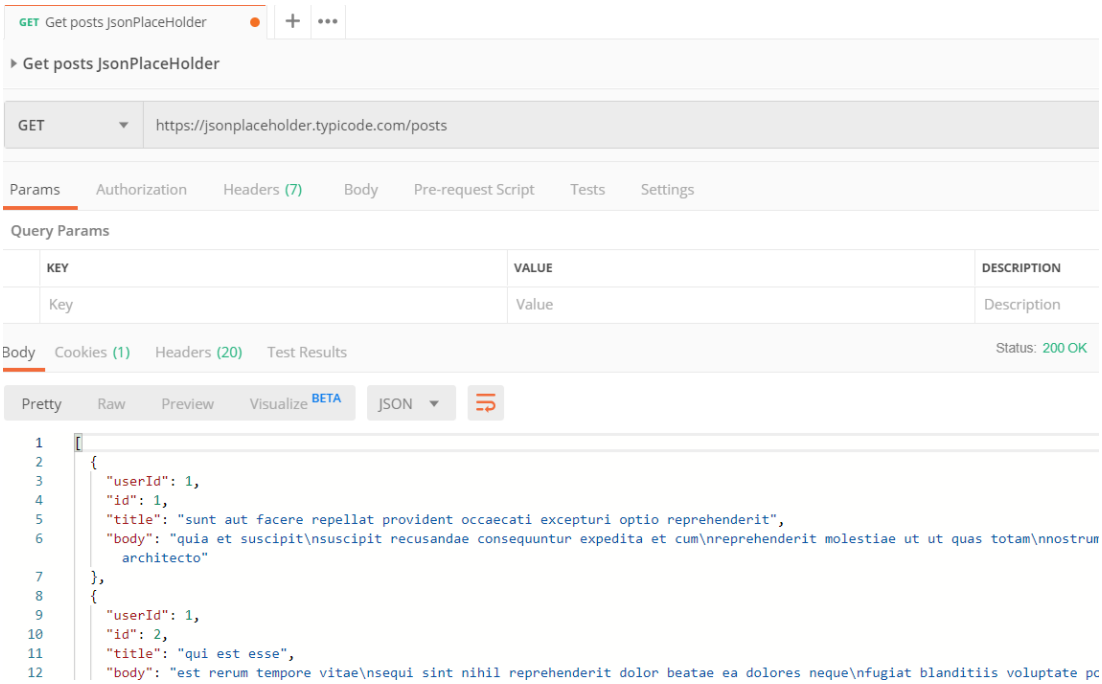


Imagen 9. Detalle de la consulta.

REST en Android

Mientras desarrollamos aplicaciones android, regularmente nos encontramos trabajando con un back-end a través de las api REST. Estas api pueden ser desarrolladas por un equipo interno o por terceros.

A continuación comenzaremos los ejercicios interactuando desde nuestro entorno de desarrollo de aplicaciones android studio (IDE) con el api público de JsonPlaceHolder ya que dispone de diferentes métodos que nos serán útiles a lo largo de esta unidad.

Ejercicio 2:

Consultar el método get posts del api JsonPlaceHolder desde android studio en un proyecto llamado “RestApi”, con la utilización de los métodos nativos de HTTPUrlConnection, para visualizar por consola del IDE los resultados una vez inicie la aplicación android.

1. Crear un proyecto android llamado “RestApi” con las opciones minSdk 19, blank activity y lenguaje kotlin.

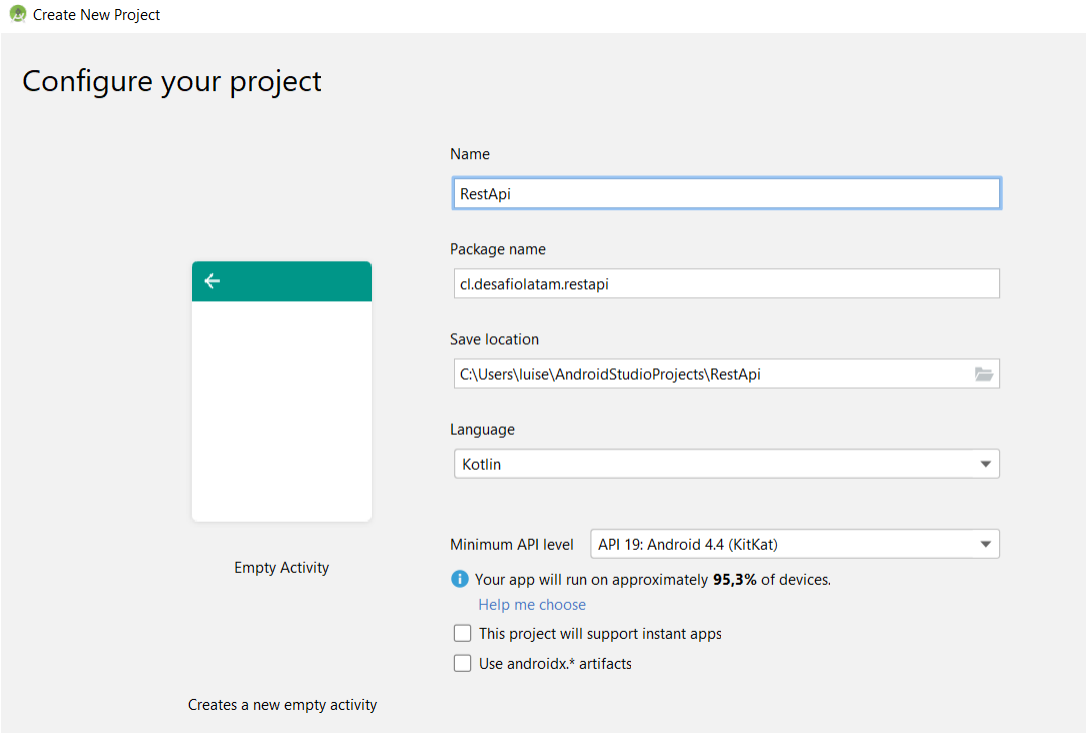


Imagen 10. Crear proyecto.

2. A continuación en el package cl.desafiolatam.restapi, creamos una nueva clase llamada TEST.kt, en la cual desarrollaremos un método llamado connectApi() con una variable string con la url del get Posts de la api JsonPlaceHolder y una segunda variable de inicialización de HTTPUrlConnection, de la siguiente forma:

```
fun connectApi() {
  try {
      val url =
          URL("https://jsonplaceholder.typicode.com/posts")
      val conn = url.openConnection() as HttpURLConnection
} catch (e: MalformedURLException) {
      e.printStackTrace()
  } catch (e: IOException) {
      e.printStackTrace()
  }
}
```

3. A continuación desarrollamos el llamado indicando el tipo de método, validando el código de respuesta y leyendo la respuesta con la clase BufferedReader de la siguiente forma:

```
conn.requestMethod = "GET"

if (conn.responseCode != 200) {
  throw RuntimeException("Failed : HTTP error code : " + conn.responseCode)
}

val br = BufferedReader(
  InputStreamReader(
      conn.inputStream
  )
)

println("Output from Server .... ")
while (br.readLine() != null) {
  println(br.readLine())
}

conn.disconnect()
```

4. Por último, creamos el método main de prueba fuera de la clase, haciendo referencia al método connectApi() que creamos anteriormente, de la siguiente forma:

```
fun main() {
  val test = TEST()
  test.connectApi()
}
```

5. Al ejecutar la clase TEST.kt, el resultado se visualiza como lo muestra la siguiente imagen:

```
cl.desafiolatam.restapi.TESTKt
"C:\Program Files\Android\Android Studio\jre\bin\java.exe" ...
Output from Server ....
  {
    "id": 1,
    "body": "quia et suscipit\nsuscipit recusandae consequuntur expedita et
  {
    "id": 2,
    "body": "est rerum tempore vitae\nsequi sint nihil reprehenderit dolor
  {
    "id": 3,
```

Imagen 11. Ejecutar `.TEST.kt`.

De este ejercicio, es de notar que no hemos realizado el llamado al api desde los métodos del ciclo de vida de una actividad por una razón fundamental, es necesario seguir las políticas de android para el consumo de métodos HTTP fuera del hilo principal (main thread) y de manera asíncrona. Por lo tanto, comenzamos a visualizar con este ejemplo porque se utilizan librerías como Retrofit para realizar estas tareas, y la razón no es otra que ahorrarnos mucho código de creación de métodos asíncronos e hilos en segundo plano, además de código httpclient, y posibles errores al consumir los datos externos. En los próximos capítulos profundizaremos estos conceptos.

Ventajas de usar REST

- **Separación entre el cliente y el servidor:** el protocolo REST separa totalmente la interfaz de usuario del servidor y el almacenamiento de datos. Eso tiene algunas ventajas cuando se hacen desarrollos. Por ejemplo, mejora la portabilidad de la interfaz a otro tipo de plataformas, aumenta la escalabilidad de los proyectos y permite que los distintos componentes de los desarrollos se puedan evolucionar de forma independiente.
- **Visibilidad, fiabilidad y escalabilidad.** La separación entre cliente y servidor tiene una ventaja evidente y es que cualquier equipo de desarrollo puede escalar el producto sin excesivos problemas. Se puede migrar a otros servidores o realizar todo tipo de cambios en la base de datos, siempre y cuando los datos de cada una de las peticiones se envíen de forma correcta. Esta separación facilita tener en servidores distintos el front y el back y eso convierte a las aplicaciones en productos más flexibles a la hora de trabajar.
- **La API REST siempre es independiente del tipo de plataformas o lenguajes:** la API REST siempre se adapta al tipo de sintaxis o plataformas con las que se estén trabajando, lo que ofrece una gran libertad a la hora de cambiar o probar nuevos entornos dentro del desarrollo. Con una API REST se pueden tener servidores PHP, Java, Python o Node.js. Lo único que es indispensable es que las respuestas a las peticiones se hagan siempre en el lenguaje de intercambio de información usado, normalmente XML o JSON.

3.3 Buenas Practicas de Autenticación en request HTTP

HTTP Client con Retrofit

Competencias:

- Conocer los conceptos del cliente http retrofit.
- Implementar Retrofit en una aplicación Android.
- Conocer alternativas a Retrofit.

Introducción

En este capítulo utilizaremos retrofit, el cual es una librería que encapsula los métodos de httpCLient. Es necesario aprender a implementar retrofit para la comunicación con apis externas a nuestras aplicaciones móviles, por su amplio uso y características que lo han posicionado como casi un estándar en la industria.

Retrofit es ampliamente utilizada en la comunidad de desarrollo android, convirtiéndose prácticamente en un estándar de la industria, razón que nos compromete estudiar y conocer sus métodos e implementación en nuestros proyectos.

Realizaremos ejercicios con distintas operaciones de creación, actualización, consulta y eliminación de datos.

Retrofit

Es una librería Http client para específicamente realizar peticiones http tipo rest para android y java, permitiendo el envío y recepción de respuestas de datos entre nuestras aplicaciones y distintas plataformas de software a través de internet. Retrofit simplifica la codificación para lograr esta comunicación, gestionando y transformando nuestros datos para ser traspasados a nuestras clases pojo (plain old java object).

Para realizar una petición con Retrofit se necesitará lo siguiente:

- **Una clase Retrofit:** donde se cree la instancia a la librería de retrofit para utilizar sus métodos base, definiendo aquí nuestra url base que nuestra aplicación usará para todas las peticiones http.
- **Una interfaz de operaciones:** en la cual definamos todas nuestras peticiones con sus respectivos endpoints y datos de entrada o salida.
- **Clases de modelo de datos o pojo:** para mapear los datos que llegan desde el servidor traspasando estos a objetos e instancias automáticamente para ser usados en nuestra aplicación.

GSON Converter

Habitualmente, las peticiones y respuestas son intercambiadas en formato JSON, razón por la cual se debe utilizar un convertidor junto con retrofit para lidiar con la serialización de datos en este formato. Retrofit puede deserializar por defecto respuestas de tipo OkHttp, pero admite el uso de convertidores como gson.

Gson es una biblioteca de java que nos permite convertir objetos java a json y viceversa de manera automática, sin mayores configuraciones.

Ejercicio 3:

Agregar dependencias retrofit y gson al proyecto “RestApi”.

1. Abrir el archivo build.gradle a nivel de nuestra carpeta app e incluir las dependencias de retrofit y gson de la siguiente forma:

```
implementation 'com.squareup.retrofit2:retrofit:2.6.0'
implementation  'com.squareup.retrofit2:converter-gson:2.6.0'
```

2. Sincronizar nuestro archivo gradle.

Características de Retrofit como cliente HTTP

- **Declaración de API:** las anotaciones sobre los métodos de interfaz y sus parámetros indican cómo se maneja una solicitud.
- **Método request (solicitud):** cada método debe tener una anotación HTTP que proporcione el método de solicitud y la URL relativa. Hay cinco anotaciones incorporadas: GET, POST, PUT, DELETE, y HEAD. La URL relativa del recurso se especifica en la anotación. Ejemplo:

```
@GET("posts/list")
```

También puedes especificar parámetros en la url de la consulta, de la siguiente manera:

```
@GET("posts/list?sort=desc")
```

- **Manipulación de url:** una url de solicitud se puede actualizar dinámicamente utilizando bloques y parámetros de reemplazo en el método. Un bloque de reemplazo es una cadena alfanumérica rodeada por { and }. Un parámetro correspondiente debe ser agregado con la anotación @Path usando la misma cadena. Ejemplo:

```
@GET("group/{id}/posts")
Call<List<Post>> groupList(@Path("id") int groupId);
```

Parámetros de consulta también se pueden agregar.

```
@GET("group/{id}/posts")
Call<List<Post>> groupList(@Path("id") int groupId, @Query("sort") String sort);
```

Para combinaciones complejas de parámetros de consulta se puede utilizar un “Map”.

```
@GET("group/{id}/posts")
Call<List<Post>> groupList(@Path("id") int groupId, @QueryMap Map<String, String> options);
```

- **Request Body:** se puede especificar un objeto para usarlo como un cuerpo de solicitud HTTP con la anotación @Body.

```
@POST("posts/new")
Call<Post> createPost(@Body Post post);
```

- **Form Encode, multipart:** los métodos también se pueden declarar para enviar datos codificados por formularios y multiparte. Los datos codificados en forma se envían cuando @FormUrlEncodedestán presentes en el método. Cada par clave-valor se anota con @Fieldel nombre y el objeto que proporciona el valor.

```
@FormUrlEncoded
@POST("post/edit")
Call<Post> updatePost(@Field("first_name") String first, @Field("last_name") String last);
```

- **Manipulación de cabecera (header):** podemos establecer encabezados estáticos para un método utilizando la anotación @Headersanotación.

```
@Headers("Cache-Control: max-age=640000")
@GET("widget/list")
Call<List<Widget>> widgetList();


@Headers({
    "Accept: application/vnd.github.v3.full+json",
    "User-Agent: Retrofit-Sample-App"
})
@GET("users/{username}")
Call<User> getUser(@Path("username") String username);
```

Ejercicio 4:

Crear una aplicación que se conecte a un api pública que disponga de varios métodos HTTP para utilizar servicios REST. En este caso utilizaremos la api pública de JsonPlaceHolder.com la cuál nos brinda distintos métodos HTTP Rest, con data de prueba que nos ayuda a emular este tipo de desarrollos con comodidad.

Nota: A lo largo de la unidad estaremos trabajando sobre este ejercicio, ampliando sus funciones y operaciones.

1. Revisar los métodos expuestos en la página de la api pública siguiente: https://jsonplaceholder.typicode.com/

De los recursos del api anterior, nos vamos a concentrar en utilizar el referido a los posts. https://jsonplaceholder.typicode.com/posts

Imagen 12. Respuesta del api para retornar posts.

2. Abrir el proyecto del capítulo 2 “RestApi” y continuar con el desarrollo del mismo, agregando el archivo pojo “Post.kt”, con los mismos datos que se muestran en cada objeto de respuesta de post del api en un nuevo package “cl.desafiolatam.restapi.pojo”, esto para ser utilizado como clase de datos para ser instancia de la respuesta desde retrofit, por ejemplo, de la siguiente forma:

```
package cl.desafiolatam.restapi.pojo

import com.google.gson.annotations.Expose
import com.google.gson.annotations.SerializedName

class Post {

  @SerializedName("userId")
  @Expose
  var userId: Int? = null
  @SerializedName("id")
  @Expose
  var id: Int? = null
  @SerializedName("title")
  @Expose
```

```
    var title: String? = null
    @SerializedName("body")
    @Expose
    var body: String? = null

  }
```

Existen alternativas prácticas para crear estos archivos pojo a partir de un json, como por ejemplo un convertidor online llamado “jsonschema2pojo.org”.

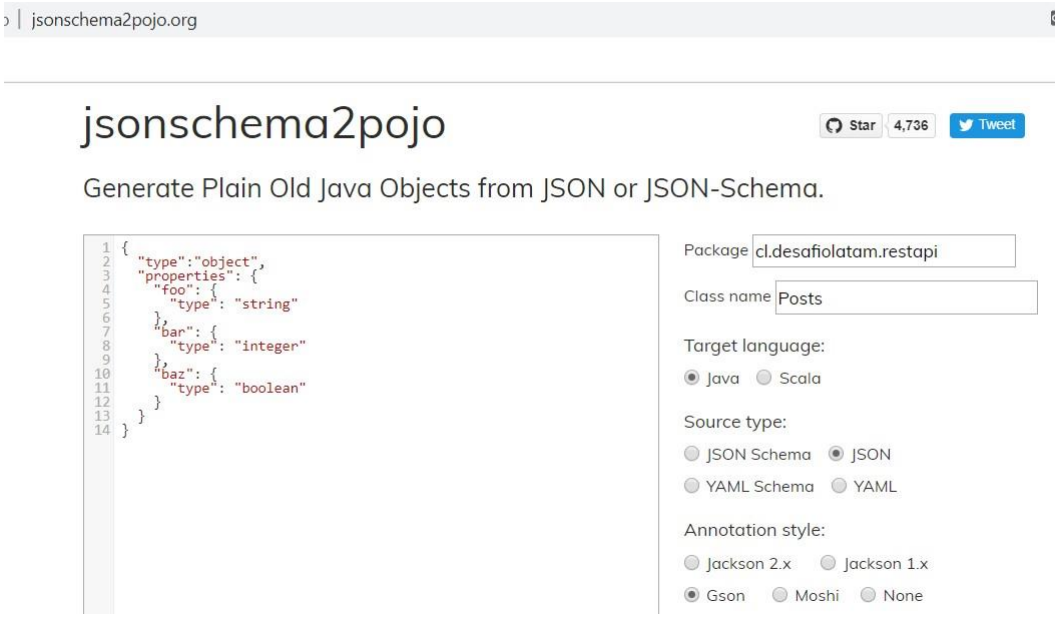


Imagen 13. Convertidor online Json - Pojo.

3. Creamos una nueva interface llamada “Api.kt” en el package cl.desafiolatam.restapi.utils, donde declararemos el método GET para obtener los datos de los posts existentes en el api, señalando la url relativa del método GET, en este caso “/posts” , de la siguiente forma:

```
package cl.desafiolatam.restapi.utils

import cl.desafiolatam.restapi.pojo.Post
import retrofit2.Call
import retrofit2.http.*

import java.util.ArrayList

interface Api {
  @GET("/posts")
  fun getAllPosts(): Call<ArrayList<Post>>
```

4. Creamos un nuevo archivo kotlin llamado RetrofitClient.kt dentro del package cl.desafiolatam.restapi.utils, que servirá de cliente base para retornar una instancia retrofit a la sección del código de nuestra app que lo requiera, con el código siguiente:

```
package cl.desafiolatam.restapi.utils

import retrofit2.Retrofit
import retrofit2.converter.gson.GsonConverterFactory

class RetrofitClient {

    companion object {
        private const val BASE_URL = "https://jsonplaceholder.typicode.com"

        fun retrofitInstance(): Api {

            val retrofit = Retrofit.Builder()
                .baseUrl(BASE_URL)
                .addConverterFactory(GsonConverterFactory.create())
                .build()

            return retrofit.create(Api::class.java)
        }
    }
}
```

5. Para mostrar la lista de posts a partir de los datos del api en forma de cards, como lo indica el estándar de las aplicaciones, es necesario que implementemos las librerías RecyclerView y CardView en nuestro archivo gradle de la app:

```
implementation 'com.android.support:cardview-v7:28.0.0'
implementation  'com.android.support:recyclerview-v7:28.0.0'
```

6. A continuación editamos el archivo principal del layout de la actividad “activity_main.xml” para agregar la vista del RecyclerView que nos ayudará a crear la lista de elementos en la pantalla (UI) a partir de los datos retornados por el método GET del api. El archivo de estilo quedaría de la siguiente forma añadiendo un coordinator layout y un recyclerview:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:id="@+id/coordinatorLayout"
    android:layout_height="match_parent">

        <android.support.v7.widget.RecyclerView
            android:id="@+id/recyclerView"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
```

```
            android:orientation="vertical"
app:layoutManager="android.support.v7.widget.LinearLayoutManager" />

</android.support.design.widget.CoordinatorLayout>
```

7. A continuación crearemos un nuevo archivo layout en la carpeta res para dar un diseño de carta (card) a los datos de la api. Creamos el archivo “posts_list.xml” con el único propósito de mostrar el título de cada ítem de la respuesta usando la clase CardView, de la siguiente forma:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v7.widget.CardView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:card_view="http://schemas.android.com/apk/res-auto"
    android:id="@+id/cardView"
    android:layout_width="match_parent"
    android:layout_height="64dp"
    android:layout_margin="8dp"
    card_view:cardCornerRadius="0dp"
    card_view:cardElevation="2dp">


  <RelativeLayout
      android:id="@+id/relativeLayout"
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      android:paddingBottom="8dp"
      android:paddingLeft="8dp"
      android:paddingRight="8dp">


    <TextView
        android:id="@+id/txtTitle"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerVertical="true"
        android:text="Item 1"
android:textAppearance="@style/TextAppearance.Compat.Notification.Title"  />
  </RelativeLayout>

</android.support.v7.widget.CardView>
```

8. Para trabajar con el recyclerview necesitamos crear un adaptador que le indique a la vista cuales son los elementos a mostrar, su posición, cuántos son en total, permitiendo una inyección sobre el elemento de vista “recyclerview” declarado en el archivo activity_main.xml. El código de la clase “CustomAdapter” es el siguiente:

```
package cl.desafiolatam.restapi.adapters

import android.support.v7.widget.RecyclerView
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.TextView
import cl.desafiolatam.restapi.R
import cl.desafiolatam.restapi.pojo.Post

import java.util.ArrayList

class CustomAdapter(val data: ArrayList<Post>) :
RecyclerView.Adapter<CustomAdapter.CustomViewHolder>() {

  inner class CustomViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
    val mTitle: TextView = itemView.findViewById(R.id.txtTitle)
  }

  override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): CustomViewHolder {
    val itemView = LayoutInflater.from(parent.context).inflate(R.layout.posts_list, parent, false)
    return CustomViewHolder(itemView)
  }

  override fun onBindViewHolder(holder: CustomViewHolder, position: Int) {

    holder.mTitle.text = data[position].title
  }

  override fun getItemCount(): Int {
    return data.size
  }
```

9. Por último, editamos nuestra actividad principal la “MainActivty.kt”, incluyendo un nuevo método llamado “loadDataApi()”, el cual cumplirá la función de hacer el llamado al método rest GET del api , utilizando retrofit. Adicionalmente, crearemos las instancias necesarias para activar el recyclerView a partir de la respuesta no nula del api de datos, como lo muestra el código siguiente:

```
package cl.desafiolatam.restapi
import android.os.Bundle
import android.support.design.widget.CoordinatorLayout
import android.support.v7.app.AppCompatActivity
import android.support.v7.widget.RecyclerView
import android.util.Log
import android.widget.Toast
import cl.desafiolatam.restapi.adapters.CustomAdapter
import cl.desafiolatam.restapi.pojo.Post
import cl.desafiolatam.restapi.utils.RetrofitClient
import retrofit2.Call
import retrofit2.Callback
import retrofit2.Response
import java.lang.Exception
import java.util.ArrayList

class MainActivity : AppCompatActivity(){
  internal lateinit var recyclerView: RecyclerView
  internal lateinit var mAdapter: CustomAdapter
  internal lateinit var coordinatorLayout: CoordinatorLayout

    override fun onCreate(savedInstanceState: Bundle?) {
     super.onCreate(savedInstanceState)
     setContentView(R.layout.activity_main)
     recyclerView = findViewById(R.id.recyclerView)
     coordinatorLayout = findViewById(R.id.coordinatorLayout)
     loadApiData()
  }

  private fun loadApiData(){
     val service = RetrofitClient.retrofitInstance()
     val call = service.getAllPosts()
     //Async
     call.enqueue(object : Callback<ArrayList<Post>> {
        override fun onResponse(call: Call<ArrayList<Post>>, response: Response<ArrayList<Post>>) {

           val postsFromApi = response.body()
           if (postsFromApi != null) {
              mAdapter = CustomAdapter(postsFromApi)
              recyclerView.adapter = mAdapter
        }
      }
        override fun onFailure(call: Call<ArrayList<Post>>, t: Throwable) {
           Log.d("MAIN", "Error: "+t)
           Toast.makeText(
              applicationContext,
              "Error: no pudimos recuperar los posts desde el api",
              Toast.LENGTH_SHORT
           ).show()
        }
     })
```

```
    }
  }
```

10. Iniciamos la aplicación para ver el resultado del código desarrollado anteriormente, la cual se debe ver como lo demuestran las siguientes imagen:

Imagen 14. Lista de datos consultados al api JsonPlaceHolder a través del método Http GET, modo Rest con Retrofit, y visualizados con la ayuda de las librerías recyclerview y cardview de android.

Ejercicio 5:

Continuando con el ejercicio anterior, profundizaremos la funcionalidad de nuestra app “Rest Api”, agregando en esta oportunidad la opción de eliminar un post de nuestra vista y emulando su eliminación en el api invocando el método http DELETE como lo indica la página del api, además realizaremos esta tarea aprendiendo a trabajar con una técnica denominada “SwipeToDelete”, de la misma forma que gmail envía a archivo los correos de tu bandeja.

Imagen 15. Invocar método DELETE.

Nota: Recuerden no se altera la base de datos de un api público, solo emulamos el método DELETE a través del HTTP expuesto).

1. Abrimos nuestro archivo CustomAdapter.kt, para agregar un método que maneje la eliminación de un objeto de la lista de posts, al final de la clase de la siguiente manera:

```
fun removeItem(position: Int) {
  data.removeAt(position)
  notifyItemRemoved(position)
}
```

2. A continuación, creamos un método adicional en la misma clase “CustomAdapter.kt”, para agregar la funcionalidad de deshacer la eliminación del post, con el código siguiente al final de la clase:

```
fun restoreItem(item: Post, position: Int) {
  data.add(position, item)
  notifyItemInserted(position)
}
```

3. Verificamos si tenemos la librería **“com.android.support:design:28.0.0”** instalada en nuestro archivo gradle de la app, de no ser así agregamos el siguiente código y sincronizamos:

```
implementation 'com.android.support:design:28.0.0'
```

4. Creamos una clase “UIHelper.kt” en el package cl.desafiolatam.restapi.utils, de ayuda para el trato del “swipe”, colores, icono de basura, en fin, gestos aplicados a la interfaz de usuario (UI), utilizando el paquete “android.graphics”. El código es el siguiente:

```
package cl.desafiolatam.restapi.utils

import android.content.Context
import android.graphics.Canvas
import android.graphics.Color
import android.graphics.Paint
import android.graphics.PorterDuff
import android.graphics.PorterDuffXfermode
import android.graphics.drawable.ColorDrawable
import android.graphics.drawable.Drawable
import android.support.v4.content.ContextCompat
import android.support.v7.widget.RecyclerView
import android.support.v7.widget.helper.ItemTouchHelper

abstract class UIHelper internal constructor(mContext: Context,
                                    private val mBackground: ColorDrawable = ColorDrawable(),
                                    private val backgroundColor: Int = Color.parseColor("#b80f0a"),
                                    private val mClearPaint: Paint = Paint()
) : ItemTouchHelper.Callback() {
  private val deleteDrawable: Drawable?
  private val intrinsicWidth: Int
  private val intrinsicHeight: Int


  init {
     mClearPaint.xfermode = PorterDuffXfermode(PorterDuff.Mode.CLEAR)
     deleteDrawable = ContextCompat.getDrawable(mContext, android.R.drawable.ic_menu_delete)
     intrinsicWidth = deleteDrawable!!.intrinsicWidth
     intrinsicHeight = deleteDrawable.intrinsicHeight
  }


  override fun getMovementFlags(recyclerView: RecyclerView, viewHolder: RecyclerView.ViewHolder):
Int {
     return ItemTouchHelper.Callback.makeMovementFlags(0, ItemTouchHelper.LEFT)
  }

  override fun onMove(
     recyclerView: RecyclerView,
     viewHolder: RecyclerView.ViewHolder,
     viewHolder1: RecyclerView.ViewHolder
  ): Boolean {
     return false
  }

  override fun onChildDraw(
     c: Canvas,
     recyclerView: RecyclerView,
     viewHolder: RecyclerView.ViewHolder,
     dX: Float,
     dY: Float,
     actionState: Int,
     isCurrentlyActive: Boolean
```

```
    ) {
        super.onChildDraw(c, recyclerView, viewHolder, dX, dY, actionState, isCurrentlyActive)

        val itemView = viewHolder.itemView
        val itemHeight = itemView.height

        val isCancelled = dX == 0f && !isCurrentlyActive

        if (isCancelled) {
            clearCanvas(
                c,
                itemView.right + dX,
                itemView.top.toFloat(),
                itemView.right.toFloat(),
                itemView.bottom.toFloat()
            )
            super.onChildDraw(c, recyclerView, viewHolder, dX, dY, actionState, isCurrentlyActive)
            return
        }

        mBackground.color = backgroundColor
        mBackground.setBounds(itemView.right + dX.toInt(), itemView.top, itemView.right,
itemView.bottom)
        mBackground.draw(c)

        val deleteIconTop = itemView.top + (itemHeight - intrinsicHeight) / 2
        val deleteIconMargin = (itemHeight - intrinsicHeight) / 2
        val deleteIconLeft = itemView.right - deleteIconMargin - intrinsicWidth
        val deleteIconRight = itemView.right - deleteIconMargin
        val deleteIconBottom = deleteIconTop + intrinsicHeight
        deleteDrawable!!.setBounds(deleteIconLeft, deleteIconTop, deleteIconRight, deleteIconBottom)
        deleteDrawable.draw(c)

        super.onChildDraw(c, recyclerView, viewHolder, dX, dY, actionState, isCurrentlyActive)
    }

    private fun clearCanvas(c: Canvas, left: Float?, top: Float?, right: Float?, bottom: Float?) {
        c.drawRect(left!!, top!!, right!!, bottom!!, mClearPaint)

    }

    override fun getSwipeThreshold(viewHolder: RecyclerView.ViewHolder): Float {
        return 0.7f
    }
}
```

5. Agregamos un nuevo método Rest, para invocar el método de la api de JsonPlaceHolder que elimina los posts, en nuestra interfaz “Api.kt”, de la siguiente manera:

```
@DELETE("/posts/{postId}")
fun deletePost(@Path("postId") postId: Int?): Call<Void>
```

{postId} es un parámetro que recibe la anotación y se envía a través del método deletePost, donde @Path hace la conexión de la variable hacia la anotación, que en este caso es de tipo número (Int).

6. Con todos estos elementos creados, es hora de editar nuestra actividad principal “MainActivity.kt” para crear un método que llame la acción a través de retrofit para invocar el api rest con el método DELETE; para esto es necesario crear el siguiente método al final de la clase de actividad:

```
private fun deletePostFromApi(postId: Int?){
  val service = RetrofitClient.retrofitInstance()
  val call = service.deletePost(postId)
  call.enqueue(object : Callback<Void> {
     override fun onResponse(call: Call<Void>, response: Response<Void>) {

        if(response.code() == 200) {
           Toast.makeText(
              applicationContext,
              "Se ha eliminado correctamente el post en el api. Código respuesta de servidor: "+
response.code(), Toast.LENGTH_LONG).show()

           Toast.makeText(applicationContext, "Recuerda, nuestra api sólo facilita métodos http públicos
para REST. Esta operación no elimina datos reales.",
           Toast.LENGTH_LONG
           ).show()

        }
     }

     override fun onFailure(call: Call<Void>, t: Throwable) {
        // handle failure
        Toast.makeText(applicationContext, "Ocurrió un error borrando el post en el api. "+t,
Toast.LENGTH_LONG).show()
     }
  })
}
```

Nota: Recordemos que esta api nos devuelve un código desde su servicio rest para pruebas, no existe una eliminación de los datos reales en la base de datos de esta página por el solo hecho de ser pública.

7. El método anterior será invocado desde un nuevo método que ejecutará las acciones de modificación de la UI con la clase creada anteriormente llamada “UIHelper.kt”, esté método llamado “enableSwipeToDeleteAndUndo()” cumplirá tres funciones:
    - Ejecutar la eliminación de un post deslizando el dedo desde la derecha hacia la izquierda de la pantalla.
    - Enviar la solicitud de eliminación tipo rest DELETE al servidor del api de JsonPlaceHolder y procesar un código de respuesta, el cual se mostrará a través de un mensaje Toast en la pantalla.
    - Mostrar un widget tipo SnackBar con la opción de deshacer la acción de borrar para restaurar el post que se ha eliminado. Esta barra deberá aparecer al final de la pantalla y durará entre 12 y 15 segundos, lo suficiente para permanecer luego de los mensajes Toast.

Al final de nuestra MainActivity.kt, creamos el nuevo método y su código es el siguiente:

```
private fun enableSwipeToDeleteAndUndo() {
  val swipeToDeleteCallback = object : UIHelper(this) {
    override fun onSwiped(viewHolder: RecyclerView.ViewHolder, i: Int) {


      val position = viewHolder.adapterPosition
      val item = mAdapter.data.get(position)

      //ELIMINAMOS DEL SERVIDOR EL POST (FAKE)
      deletePostFromApi(item.id)
      mAdapter.removeItem(position)


      val snackbar = Snackbar
        .make(coordinatorLayout, "El post ha sido eliminado de la lista", Snackbar.LENGTH_LONG)
      snackbar.setAction("DESHACER") {
        mAdapter.restoreItem(item, position)
        recyclerView.scrollToPosition(position)
      }

      snackbar.setActionTextColor(Color.YELLOW)
      snackbar.setDuration(12000)
      snackbar.show()

    }
  }

  val itemTouchhelper = ItemTouchHelper(swipeToDeleteCallback)
  itemTouchhelper.attachToRecyclerView(recyclerView)
}
```

8. Por último, agregamos al método ya existente “loadApiData()” de la misma clase de actividad la instancia al método enableSwipeToDeleteAnUndo(), dentro de su sub método onResponse(), justo después de la asignación del adaptador a la variable recyclerview, como se muestra a continuación:

```
if (postsFromApi != null) {
    mAdapter = CustomAdapter(postsFromApi)
    recyclerView.adapter = mAdapter

    enableSwipeToDeleteAndUndo()
}
```

Esto permitirá la funcionalidad de eliminar los posts en la pantalla para cada uno de los elementos de la lista cargados desde el api JsonPlaceHolder con el método GET posts.

9. Ejecutamos nuestra aplicación y la funcionalidad se visualiza como lo muestran las siguientes imágenes:

Imagen 16. Acción Swipe To Delete.

Imagen 17. Mensaje respuesta del api a través del servicio rest /posts/{id} , método http DELETE, y barra de acción “deshacer” para recuperar el post eliminado en la UI.

Volley HTTP Client

Volley es una biblioteca HTTP-Client al igual que Retrofit, que hace que la creación de redes para aplicaciones de Android sea más fácil y, lo que es más importante, más rápida.

Volley está disponible en GitHub.Volley ofrece los siguientes beneficios:

- Programación automática de solicitudes de red.
- Múltiples conexiones de red concurrentes.
- Disco transparente y memoria caché de respuesta con coherencia de caché HTTP estándar .
- Soporte para solicitud de priorización.
- Solicitud de cancelación API. Puede cancelar una sola solicitud o puede establecer bloques o ámbitos de solicitudes para cancelar.
- Facilidad de personalización, por ejemplo, para reintentos y retrocesos.
- Ordenación sólida que facilita el llenado correcto de la interfaz de usuario con los datos obtenidos de forma asincrónica de la red.
- Herramientas de depuración y rastreo.

Volley se destaca en las operaciones de tipo RPC que se utilizan para completar una interfaz de usuario, como buscar una página de resultados de búsqueda como datos estructurados.

Se integra fácilmente con cualquier protocolo y viene de fábrica con soporte para cadenas sin procesar, imágenes y JSON. Al proporcionar soporte integrado para las funciones que necesita, Volley lo libera de escribir código repetitivo y le permite concentrarse en la lógica específica de su aplicación.

Volley no es adecuado para grandes operaciones de descarga o transmisión, ya que Volley guarda todas las respuestas en la memoria durante el análisis. Para operaciones de descarga grandes, considere usar una alternativa como DownloadManager.

La forma más fácil de agregar Volley a su proyecto es agregar la siguiente dependencia al archivo build.gradle de su aplicación:

```
dependencies { ...    implementation 'com.android.volley: volley: 1.1.1' }
```

También puede clonar el repositorio de Volley y configurarlo como un proyecto de biblioteca. Git clone el repositorio escribiendo lo siguiente en la línea de comando:

```
git clone https://github.com/google/volley
```

Un ejemplo para devolver una respuesta con Volley como cadena string y asignarla a un textview en una actividad sería el siguiente:

```
// function for network call
  fun getUsers() {
    // Instantiate the RequestQueue.
    val queue = Volley.newRequestQueue(this)
    val url: String = "https://api.github.com/search/users?q=eyehunt"

    // Request a string response from the provided URL.
    val stringReq = StringRequest(Request.Method.GET, url,
        Response.Listener<String> { response ->

          var strResp = response.toString()
          val jsonObj: JSONObject = JSONObject(strResp)
          val jsonArray: JSONArray = jsonObj.getJSONArray("items")
          var str_user: String = ""
          for (i in 0 until jsonArray.length()) {
            var jsonInner: JSONObject = jsonArray.getJSONObject(i)
            str_user = str_user + "\n" + jsonInner.get("login")
          }
          textView!!.text = "response : $str_user "
        },
        Response.ErrorListener { textView!!.text = "That didn't work!" })
    queue.add(stringReq)
  }
```

Importando las siguientes clases a la actividad android:

```
import com.android.volley.Request
import  com.android.volley.Response
import com.android.volley.toolbox.StringRequest
import com.android.volley.toolbox.Volley
```

Intercambio de datos con JSON

Competencias:

- Conocer el formato de intercambio de datos JSON.
- Comprender la estructura que conforma un JSON.
- Identificar la cabecera de una consulta a una api rest.
- Identificar el cuerpo de una consulta a una api rest.

Introducción

En este capítulo estudiaremos los conceptos asociados a JSON (Javascript Object Notation) como la notación más ligera, eficiente y rápida de toda la web en la actualidad.

JSON ha ido desplazando desde hace algunos años las antiguas formas de intercambio de datos que por ejemplo se hacían utilizando XML, llegando a tal punto que las bases de datos modernas han comenzado a evolucionar hacia el almacenamiento de datos JSON en su forma compilada BSON, como lo son por ejemplo, MongoDB y Oracle 12c+.

Actualmente el estándar de comunicación es a través de Json, Comprender la sintaxis y cómo se ordenan los datos en este formato será de vital importancia para nuestro desarrollador como creadores de apps.

JSON

JSON (JavaScript Object Notation) es un formato ligero de intercambio de datos. Es fácil para los humanos leer y escribir. Es fácil para las máquinas analizar y generar. Se basa en un subconjunto del lenguaje de programación denominados JavaScript Standard ECMA-262.

JSON es un formato de texto que es completamente independiente del lenguaje pero utiliza convenciones que son familiares para los programadores de la familia de lenguajes C, incluido Kotlin, C, C ++, C#, Java, JavaScript, Perl, Python y muchos otros. Estas propiedades hacen que JSON sea un lenguaje ideal para el intercambio de datos.

Estructura de un JSON

JSON se basa en dos estructuras:

- Una colección de pares de nombre / valor, en distintos idiomas y lenguajes de programación; esto es conocido como un objeto, registro, estructura, diccionario, tabla hash, lista con clave o arreglo asociativo.
- Una lista ordenada de valores. En la mayoría de los lenguajes de programación, esto se implementa como arreglos, vectores, listas o secuencias.

Estas son estructuras universales; virtualmente todos los lenguajes de programación las soportan de una forma u otra. Es razonable que un formato de intercambio de datos que es independiente del lenguaje de programación se base en estas estructuras.

En JSON, se presentan de estas formas:

Un objeto es un conjunto desordenado de pares de nombre / valor. Un objeto comienza con {llave izquierda y termina con} llave derecha. Cada nombre es seguido por: dos puntos y los pares de nombre / valor están separados por una coma.

Ejemplo:

```
{
   "glossary": {
      "title": "example glossary",
      "GlossDiv": {
         "title": "S",
         "GlossList": {
            "GlossEntry": {
               "ID": "SGML",
               "SortAs": "SGML",
               "GlossTerm": "Standard Generalized Markup Language",
               "Acronym": "SGML",
               "Abbrev": "ISO 8879:1986",
               "GlossDef": {
                  "para": "A meta-markup language, used to create markup languages such as DocBook.",
                  "GlossSeeAlso": ["GML", "XML"]
               },
               "GlossSee": "markup"
            }
         }
      }
   }
}
```

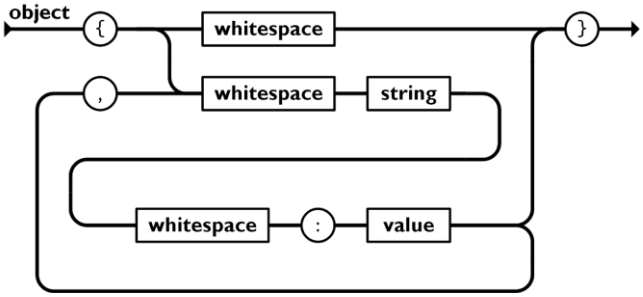


Imagen 18. Representación gráfica de la estructura de un arreglo JSON.

Ejemplo de array json:

```
[
{ "id": 1, "nombre": "Luis Liberal" },
{ "id": 2, "nombre": "Pamela Martín" }
]
```

Un valor puede ser una cadena entre comillas dobles, o un número, o verdadero o falso o nulo, o un objeto o una matriz. Estas estructuras pueden estar anidadas.

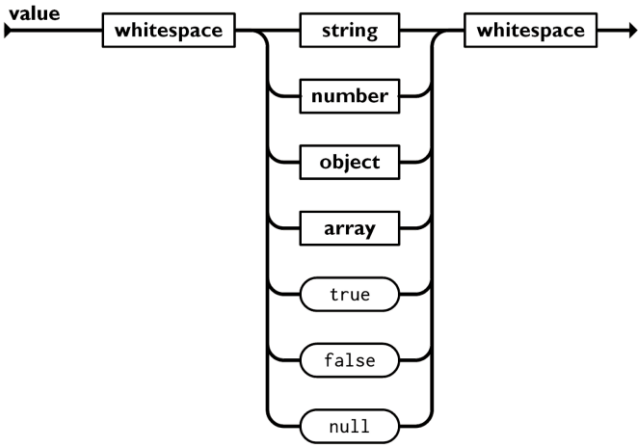


Imagen 19. Representación gráfica de la estructura que tienen los valores en los objetos json.

Una cadena es una secuencia de cero o más caracteres Unicode, entre comillas dobles, usando escapes de barra invertida. Un carácter se representa como una cadena de caracteres única. Una cadena es muy parecida a una cadena C o Java.

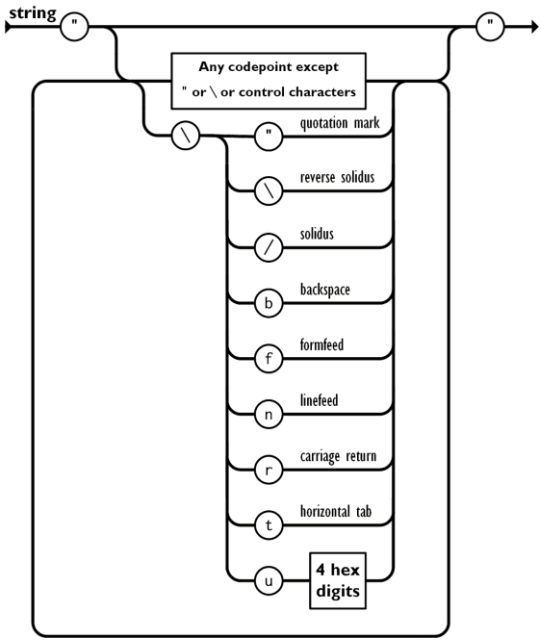


Imagen 20. Representación gráfica de secuencia de tipos de valores en un String o cadena.

Un número es muy parecido a un número C o Java, excepto que los formatos octal y hexadecimal no se usan.

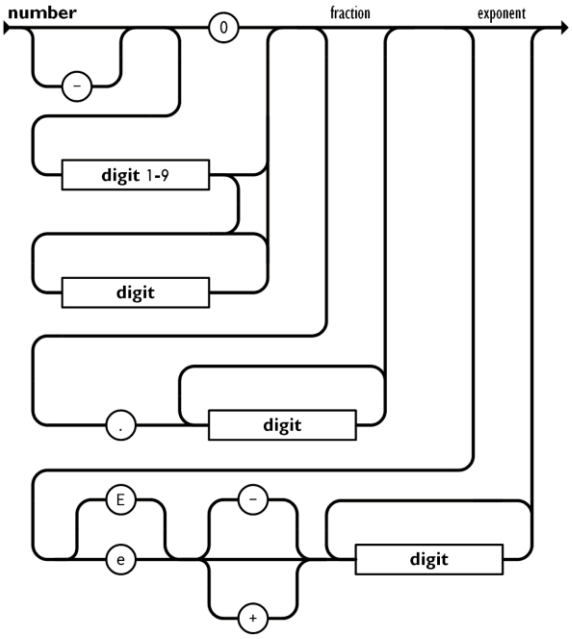


Imagen 21. Representación gráfica tipos de valores numéricos en JSON.

Se puede insertar un espacio en blanco entre cualquier par de tokens. Exceptuando algunos detalles de codificación, que describe completamente el lenguaje.

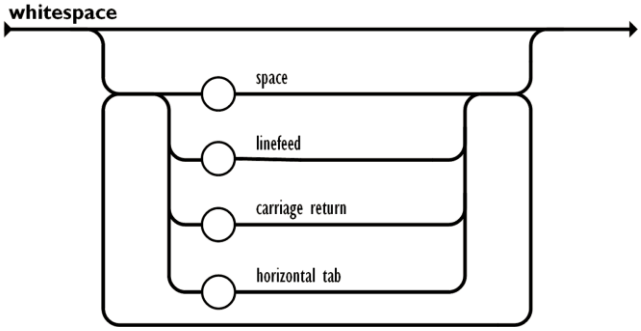


Imagen 22. Representación gráfica del trato de espacios en blanco en JSON.

JSON - REST HEADER (Cabecera/Encabezado)

Los headers y parámetros REST contienen una gran cantidad de información que puede ayudarlo a localizar problemas cuando los encuentre. Los encabezados HTTP son una parte importante de la solicitud y respuesta de la API, ya que representan los metadatos asociados con la solicitud y la respuesta de la API. Los encabezados llevan información para:

- Request and Response Body (Cuerpos de datos en solicitudes y respuestas)
- Request Authorization (Solicitar autorización)
- Caché de respuesta
- Cookies de respuesta

Además de las categorías anteriores, los encabezados HTTP también contienen otras informaciones sobre los tipos de conexión HTTP, servidores proxy, etc. La mayoría de estos encabezados son para la administración de conexiones entre cliente, servidor y servidores proxy; no requieren validación explícita a través de las pruebas.

Los encabezados se clasifican principalmente como encabezados de solicitud y encabezados de respuesta. Los encabezados de solicitud se configuran cuando enviamos la solicitud para probar una API y debemos establecer la aserción en los encabezados de respuesta para garantizar que se devuelvan los encabezados correctos.

Los encabezados con los que nos encontraremos más durante el trabajo con un API son los siguientes:

- **Authorization**: lleva credenciales que contienen la información de autenticación del cliente para el recurso que se solicita.
- **WWW-Authenticate**: el servidor lo envía si necesita una forma de autenticación antes de que pueda responder con el recurso real que se solicita. A menudo se envía junto con un código de respuesta de 401, que significa "no autorizado".
- **Accept-Charset**: este es un encabezado que se establece con la solicitud y le dice al servidor qué conjuntos de caracteres son aceptables para el cliente.
- **Content-Type**: indica el tipo de medio (texto / html o texto / JSON) de la respuesta enviada al cliente por el servidor, esto ayudará al cliente a procesar el cuerpo de la respuesta correctamente.
- **Cache-Control**: esta es la política de caché definida por el servidor para esta respuesta, el cliente puede almacenar una respuesta en caché y volver a usar hasta el momento definido por el encabezado de Control de caché.

RETROFIT - JSON HEADERS

En retrofit para asignar parámetros a nuestros métodos http de las interfaces, basta con agregar la anotación @Headers sobre la anotación del tipo de método http, ya sea get, post, put, delete.

Ejemplo:

```
@Headers("Content-Type: application/json; charset=UTF-8")
```

JSON - REST BODY (Cuerpo)

El cuerpo de un objeto json, no requiere mayor explicación de la que ya hemos dado, el cuerpo json (body) es ese objeto json que puede anexarse a una solicitud (request) o recibirse en la respuesta del servicio. Un ejemplo de un cuerpo json sería el siguiente:

```
{
  "customers":
  {
    "firstName": "Joe",
    "lastName": "Bloggs",
    "fullAddress":
    {
        "streetAddress": "21 2nd Street",
        "city": "New York",
        "state": "NY",
        "postalCode": 10021
    }
  }
}
```

RETROFIT - JSON BODY

En kotlin utilizando retrofit podemos disponer de una anotación @Body que se utiliza para asignar un objeto tipo pojo (clase que contiene instancias de los datos similares a los que contiene el objeto json) a una respuesta o solicitud. Por ejemplo, anteponiendo los siguientes ejercicios, así sería una asignación de un método Post en la interfaz de retrofit para enviar un nuevo post al servidor:

```
@Headers("Content-Type: application/json; charset=UTF-8")
@POST("/posts")
fun createNewPost(@Body post: Post): Call<Post>
```

Dónde “Post” es nuestra clase pojo que contiene los objetos del json body.

Ejercicio 6:

Crear un post nuevo usando Postman con los datos indicados en la página del api de JsonPlaceHolder, enviando cabeceras y cuerpos (headers, body) como se indican en su guía: https://jsonplaceholder.typicode.com/guide.html

1. Abrir Postman y asignar la url del método POST para la creación de un nuevo post.

```
https://jsonplaceholder.typicode.com/posts
```

2. Asignar el cuerpo como se indica en la guía de la siguiente manera:

Imagen 23. Asignar el cuerpo.

3. Asignar los parámetros de cabecera Content-Type y Charset de la siguiente forma:

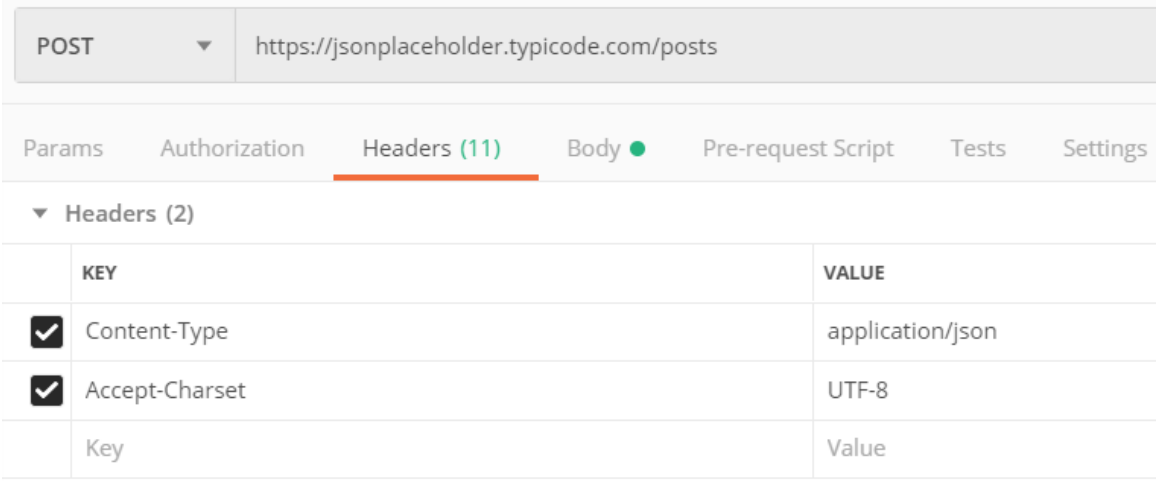


Imagen 24. Asignar parámetros.

4. Visualizar la respuesta con código 201 del servidor, donde se indica su aceptación y retorna un id de post nuevo (estático ya que no lo crea realmente en base de datos por ser pública).

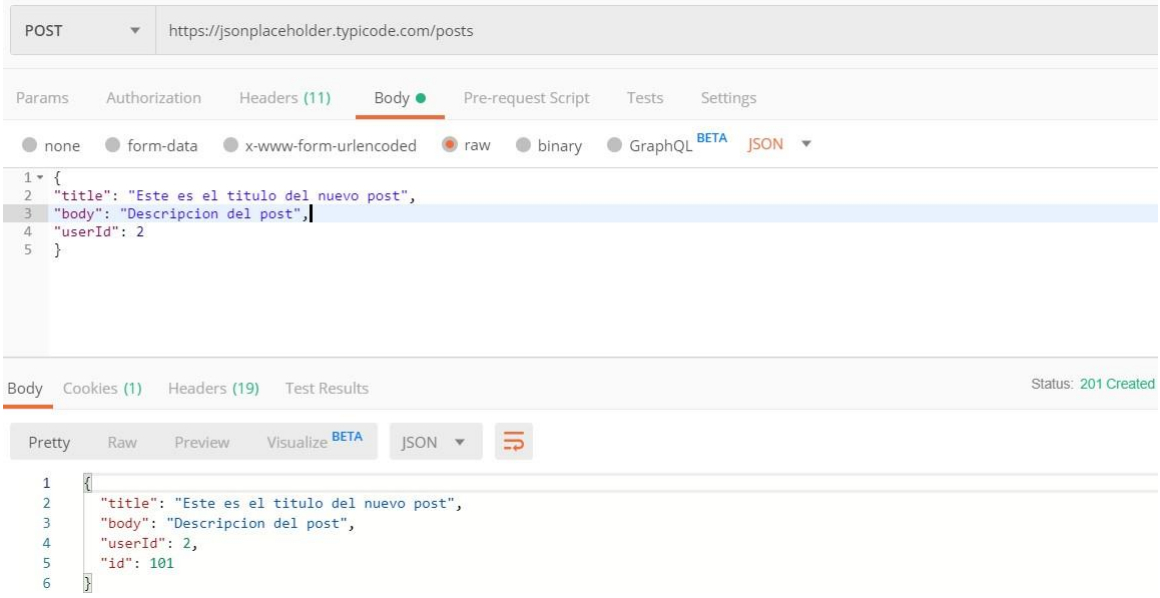


Imagen 25. Visualizar respuesta.

API REST y acceso a recursos remotos

Restfull/Restless y REST Post

Competencias:

- Conocer qué son los servicios RestFull
- Conocer qué son los servicios Restless
- Implementar Retrofit para consumir una API en una aplicación Android.

Introducción

En este capítulo abordaremos las características que componen a un sistema REstfull, diferenciando y aclarando otras variantes para tener conceptos más claros, respecto a estos términos utilizados en la industria, de esta manera desarrollaremos mejores aplicaciones basadas en arquitectura REST.

Recordando que REST incluye la forma en que los estados de los recursos se dirigen y transfieren datos y archivos a través de HTTP, profundizaremos en la utilización del método POST para continuar con la aplicación “RestApi” de los ejercicios y crear en esta ocasión nuevos post.

Adicionalmente, revisaremos los conceptos Restfull y Restless de la programación de servicios informáticos REST.

Restfull

La palabra Restfull hace referencia a la disponibilidad de servicios web u otros sistemas, que ofrecen recursos basados en el protocolo HTTP, por lo que para su acceso se usan los métodos GET, POST, etc., es decir, Restfull se puede entender como un grupo de servicios web o programas basados en la arquitectura HTTP que convoca REST.

Estos servicios web o sistemas ofrecen recursos en diversos tipos de formatos, según se requiera, son accesibles mediante direcciones web (url) y no se necesitan frameworks para acceder a ellos, por eso se les consideran ligeros.

Restless

Restless es cualquier sistema que no sea Restfull, es decir, basta con que el sistema no cumpla con alguna de las características de un sistema Restfull.

Aunque parezca asombroso, la mayoría de los sistemas son RESTless dado que no cumplen con una característica elemental algo desconocida en su concepto, el Hypermedia as the Engine of Application State **(**HATEOAS), el cual es un componente de la arquitectura de aplicaciones REST que lo distingue de otras arquitecturas de red.

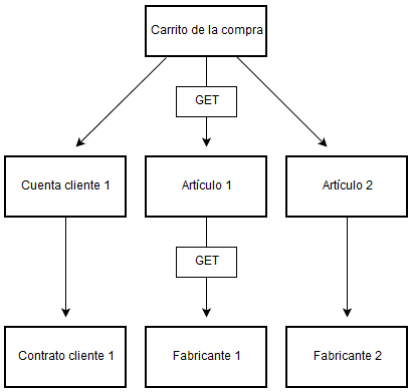


Imagen 1. Representación esquemática de la aplicación de HATEOAS en una tienda online.

En un servicio REST que siga el principio HATEOAS tal y como lo define REST, el usuario solo ha de conocer el URI inicial para poder interactuar con la oferta.

Características de un sistema RestFull

Para la conexión entre el servidor y el cliente, REST define estas cuatro características:

- **Identificación inequívoca de todos los recursos**: todos los recursos han de poder identificarse con un URI (Unique Resource Identifier).
- **Interacción con los recursos por medio de representaciones**: si un cliente necesita un recurso, el servidor le envía una representación (p. ej., HTML, JSON o XML) para que el cliente pueda modificar o borrar el recurso original.
- **Mensajes explícitos**: cada mensaje intercambiado por el servidor y el cliente ha de contener todos los datos necesarios para entenderse.
- **HATEOAS**: este principio también integra una API REST. Esta estructura basada en hipermedia facilita a los clientes el acceso a la aplicación, puesto que de este modo no necesitan saber nada más de la interfaz para poder acceder y navegar por ella.

HATEOAS y REST: el paradigma hipermedia

HATEOAS es en definitiva, una de las propiedades más elementales según las definiciones de las API REST y como tal, imprescindible en cualquier servicio REST, dada la encapsulación que profesa en las acciones a realizar de cara al cliente final.

El sistema REST, ha de basarse siempre en una estructura cliente-servidor que permita la comunicación sin estado entre el servidor y los clientes, lo que significa que todas las peticiones de los clientes al servidor se tratan de forma independiente a peticiones anteriores. El servicio debe estar estructurado en capas y utilizar las ventajas del caching HTTP (almacenamiento en caché) para que la utilización del servicio sea lo más sencilla posible para el cliente que realiza la petición. Por último, ha de tener una interfaz unitaria.

Ejercicio 7:

Implementar el método POST del api JsonPlaceHolder para crear nuevos posts desde nuestro sistema “RestApi”, dónde necesitaremos incluir un FloatingButton en la página de lista principal, cuya acción sea desplegar un Dialog o Modal que muestre tres campos para ingresar datos (los mismos que requiere el api) y un botón “ENVIAR” de acción.

1. Abrimos nuestro proyecto “RestApi”, donde crearemos un nuevo método POST en el archivo interfaz “Api.kt” de la siguiente forma:

```
@Headers("Content-Type: application/json; charset=UTF-8")
@POST("/posts")
fun createNewPost(@Body post: Post): Call<Post>
```

2. En nuestro archivo de layout activity_main.xml, agregamos una nueva vista de tipo FloatingButton entre el RecyclerView y la etiqueta de cierre del CoordinatorLayout.

```
<android.support.design.widget.FloatingActionButton
    android:id="@+id/fab"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@drawable/add"
    android:layout_margin="30dp"
    android:layout_gravity="end|bottom"
    app:backgroundTint="#8ccff8"
/>
```

3. En nuestra actividad principal necesitaremos crear un nuevo método que realice la comunicación con retrofit y el api enviando los datos solicitados, para esto crearemos un nuevo método llamado addNewPostOnApi(), de la siguiente manera:

```
fun addNewPostOnApi(post: Post){
    val service = RetrofitClient.retrofitInstance()
    val call = service.createNewPost(post)
    //Async
    call.enqueue(object : Callback<Post> {
        override fun onResponse(call: Call<Post>, response: Response<Post>) {

            val responseCode = response.code()
            if(responseCode == 200 || responseCode == 201){
                Toast.makeText(applicationContext,
                    "Post creado exitosamente. Recuerda esto solo es una respuesta desde el servidor del api
público, tus datos no serán almacenados. ",Toast.LENGTH_LONG).show()
                Toast.makeText(applicationContext,
                    "Título: " +response.body()!!.title+ ", " +
                        "Descripción: "+response.body()!!.body+", " +
                        "UserId: "+response.body()!!.userId, Toast.LENGTH_LONG).show()
            }else{
                Toast.makeText(applicationContext, "Error al crear post. Código: "+responseCode,
Toast.LENGTH_LONG).show()
            }
        }

        override fun onFailure(call: Call<Post>, t: Throwable) {
            Log.d("MAIN", "Error: "+t)
            Toast.makeText(
                applicationContext,
                "Error: no pudimos recuperar los posts desde el api",
                Toast.LENGTH_SHORT
            ).show()
        }
    })
}
```

4. Necesitamos crear un nuevo archivo de layout llamado “new_post_dialog.xml”, que sirva de vista de ingreso de datos para crear el nuevo post, incluyendo tres vistas EditText y una Button. El código es el siguiente:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
                                    android:layout_width="match_parent"
                                    android:layout_height="wrap_content"
                                    xmlns:app="http://schemas.android.com/apk/res-auto"
android:padding="20dp">

  <EditText
        android:id="@+id/titleJsonParam"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        android:textSize="16dp"
        android:hint="Ingresa un título..."
  />

  <EditText
        android:layout_marginTop="20dp"
           android:id="@+id/bodyJsonParam"
           android:layout_width="match_parent"
           android:layout_height="wrap_content"
           app:layout_constraintTop_toBottomOf="@id/titleJsonParam"
           app:layout_constraintLeft_toLeftOf="parent"
           app:layout_constraintRight_toRightOf="parent"
           android:textSize="16dp"
           android:hint="Ingresa una descripción..."
  />

  <EditText
        android:layout_marginTop="20dp"
        android:id="@+id/userIdJsonParam"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        app:layout_constraintTop_toBottomOf="@id/bodyJsonParam"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        android:textSize="16dp"
        android:hint="Ingresa tu usuarioId..."
        android:inputType="number"/>

     <Button
           android:layout_marginTop="20dp"
           android:id="@+id/btnSend"
           android:layout_width="match_parent"
           android:layout_height="50dp"
           android:background="@android:color/white"
           android:clickable="true"
           android:text="ENVIAR"
           android:textColor="#FFFFFF"
           android:textStyle="bold"
```

```
            android:backgroundTint="#1e88e5"
            app:layout_constraintTop_toBottomOf="@id/userIdJsonParam"
            app:layout_constraintLeft_toLeftOf="parent"
            app:layout_constraintRight_toRightOf="parent"
            />

</android.support.constraint.ConstraintLayout>
```

5. Teniendo ya el método que ejecuta la acción de crear contra el servidor api, es necesario llamar este método desde algún sitio, para ello vamos a crear un nuevo método que se llamará “showDialog()”, el cual mostrará un modal para permitirnos el ingreso de los datos deltítulo, descripción y userId necesarios para crear el post en el api de JsonPlaceHolder.

```
private fun showDialog(){

    val dialog = Dialog(this)
    dialog.requestWindowFeature(Window.FEATURE_NO_TITLE)
    dialog.setCancelable(true)
    dialog.setContentView(R.layout.new_post_dialog)
    val title = dialog .findViewById(R.id.titleJsonParam) as TextView
    val body = dialog .findViewById(R.id.bodyJsonParam) as TextView
    val userId = dialog .findViewById(R.id.userIdJsonParam) as TextView
    val btnSend = dialog .findViewById(R.id.btnSend) as Button

    btnSend.setOnClickListener {
      val post = Post()

      try {
        post.title = title.text.toString()
        post.body = body.text.toString()
        val userIdStr = userId.text.toString()
        post.userId = userIdStr.toInt()
        addNewPostOnApi(post)
      }catch (e: Exception){
        Log.d("MAIN", "Error al crear post",e)
      }
      dialog .dismiss()
    }
    dialog .show()

  }
```

6. Por último en nuestro método de ciclo de vida de actividad onCreate(), agregaremos antes del llamado al método loadApiData(), la inicialización, instancia y declaración del botón Float de la siguiente forma:

```
val mFab = findViewById<FloatingActionButton>(R.id.fab)
mFab.setOnClickListener {
  showDialog()
}
```

7. Al iniciar nuestra app el resultado de las acciones y vistas creadas sería como lo demuestran las siguientes imágenes:

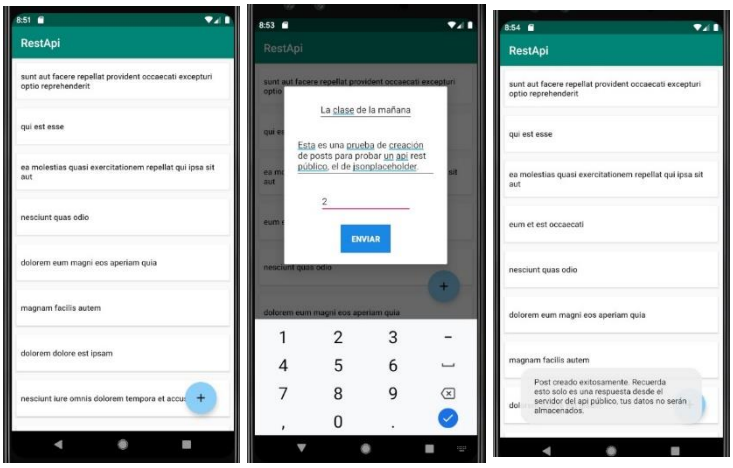


Imagen 2. Grupo de imágenes - Acción POST create post JsonPlaceHolder.

Ejercicio 8:

Ejercicio bonus de la unidad, agregar la funcionalidad swipeToRefresh a la actividad principal para refrescar los datos cargados del api.

1. En nuestro archivo de layout activity_main.xml, declararemos una nueva vista tipo widget que encierre a la vista RecyclerView ya existente entre sus etiquetas de la siguiente manera:

```
<android.support.v4.widget.SwipeRefreshLayout
    android:id="@+id/swipeRefreshLayout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:layout_behavior="android.support.design.widget.AppBarLayout$ScrollingViewBehavior">

  <android.support.v7.widget.RecyclerView
      android:id="@+id/recyclerView"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:orientation="vertical"
      app:layoutManager="android.support.v7.widget.LinearLayoutManager"  />

</android.support.v4.widget.SwipeRefreshLayout>
```

2. En nuestro archivo de la actividad principal “MainActivity.kt” , vamos a implementar la clase SwipeRefreshLayout de esta forma:

```
class MainActivity : AppCompatActivity(), SwipeRefreshLayout.OnRefreshListener {
```

3. Requerimos implementar el método de la clase, onRefresh(), y lo haremos invocando al método loadApiData() del que ya disponemos. El código es el siguiente:

```
override fun onRefresh() {
  //WE CHECK INTERNET CONNECTION
  try{
     loadApiData()
  }catch(e: Exception){
     Toast.makeText(this, "Ocurrió un error, verifica tu conexión a internet",
Toast.LENGTH_LONG).show()
  }
}
```

4. Necesitamos declarar la variable de clase instancia de SwipeRefreshLayout así:

```
private lateinit var swipeRefreshLayout : SwipeRefreshLayout
```

5. A continuación, en nuestro método del ciclo de vida onCreate(), agregamos las instancia del objeto de la vista y definimos el listener en el contexto de nuestra actividad, justo después de la declaración del layout a través del método setContentView, de esta manera:

```
setContentView(R.layout.activity_main)

swipeRefreshLayout = findViewById(R.id.swipeRefreshLayout)
swipeRefreshLayout.setOnRefreshListener(this)
```

6. Por último dentro de nuestro método loadApiData(), justo al final indicamos que la acción del swipe to refresh ya no debe continuar:

```
swipeRefreshLayout.isRefreshing = false
```

7. Hemos finalizado, la funcionalidad se visualizará como una rueda de carga o spin en la parte superior de la pantalla de manera centrada, y al deslizar nuestro dedo desde arriba hacia abajo, volveremos a cargar los posts que nos retorna el api y se refrescan los datos en la pantalla; por ejemplo, podemos hacer el ejercicio de eliminar algunos posts (que sabemos no se eliminan realmente en una base de datos por ser api público) y refrescamos la data para observar cómo vuelven los elementos anteriormente eliminados a su posición.

Imagen 3. Acción SwipeRefreshLayout - Recarga de posts.

Imágenes con Picasso

Competencias:

- Conocer los conceptos asociados al uso de la librería Picasso.
- Implementar Picasso en aplicaciones android.
- Realizar caching de imágenes con Picasso.
- Conocer librerías alternativas para el manejo de imágenes.

Introducción

En este capítulo estudiaremos cómo se realiza la carga y descarga de imágenes utilizando la librería más común y popular de android “Picasso”. El éxito de esta librería se debe a su gran eficiencia en el manejo de los recursos gráficos y en su rendimiento, siendo la de mejor manejo de la memoria del dispositivo.

Es esencial aprender a utilizar Picasso, ya que el trabajo con imágenes en las aplicaciones móviles son recurrentes, por ejemplo en listas, detalles, iconos, entre otras vistas, siendo imprescindible que sepamos no sobrecargar el sistema y practicar los mejores códigos para estas implementaciones.

Picasso

Las imágenes agregan un contexto muy necesario y un toque visual a las aplicaciones de Android que hace que destaquen nuestros sistemas. Picasso permite la carga de imágenes sin problemas y de una manera muy sencilla, que en ocasiones, bastará con una línea de código.

Picasso fue creada con el propósito de mostrar imágenes de una manera rápida y sencilla simplificando el proceso que en años anteriores se tenía que hacer; además Picasso se encarga de la solicitud HTTP, el almacenamiento en caché de imágenes, y hasta depurar, por lo tanto, nos hará la vida más fácil a la hora de trabajar con imágenes en android.

Imagen 4. Logo Picasso.

Funcionalidades de Picasso

- **Resource Loading**: la más común es la descarga de recursos, assets, archivos (files), content providers son todos soportados como fuentes de imágenes..

```
Picasso.get().load(R.drawable.landing_screen).into(imageView1);
Picasso.get().load("file:///android_asset/DvpvklR.png").into(imageView2);
Picasso.get().load(new File(...)).into(imageView3);
```

- **Adapter Downloads:** la reutilización del adaptador se detecta automáticamente y se cancela la descarga anterior.

```
@Override public void getView(int position, View convertView, ViewGroup parent) {
  SquaredImageView view = (SquaredImageView) convertView;
  if (view == null) {
    view = new SquaredImageView(context);
  }
  String url = getItem(position);

  Picasso.get().load(url).into(view);
}
```

- **Image Transformations**: transforme las imágenes para adaptarse mejor a los diseños y para reducir el tamaño de la memoria.

```
Picasso.get()
  .load(url)
  .resize(50, 50)
  .centerCrop()
  .into(imageView)
```

- **Custom Image Transformations**: podemos especificar transformaciones con efectos más avanzados.

```
public class CropSquareTransformation implements Transformation {
  @Override public Bitmap transform(Bitmap source) {
    int size = Math.min(source.getWidth(), source.getHeight());
    int x = (source.getWidth() - size) / 2;
    int y = (source.getHeight() - size) / 2;
    Bitmap result = Bitmap.createBitmap(source, x, y, size, size);
    if (result != source) {
      source.recycle();
    }
    return result;
  }

  @Override public String key() { return "square()"; }
}
```

- **Place Holders**: Picasso soporta la descarga de placeholders y errores como opcionales.

```
Picasso.get()
    .load(url)
    .placeholder(R.drawable.user_placeholder)
    .error(R.drawable.user_placeholder_error)
    .into(imageView);
```

Caché de imágenes con Picasso

Como hemos indicado, dependemos mucho de las imágenes en nuestras aplicaciones. En algunos casos las imágenes suelen repetirse, como por ejemplo una imagen de perfil, la cual sería ideal guardar de alguna forma para no tener que recargar dicha imagen cada vez que se ingresa a la aplicación usando el tráfico de datos, que termina siendo costoso en servidores y alojamientos. Ejemplos de este trabajo de caché de imágenes está en las redes sociales más populares como lo son Facebook e Instagram.

Básicamente de esto se trata cuando hablamos de caché de imágenes. Librerías como Picasso se encargan de almacenarlas y recuperarlas de la memoria volátil, memoria interna o descargarla nuevamente si lo anterior no funciona, también es posible eliminar las imágenes cuando no se utilizan más.

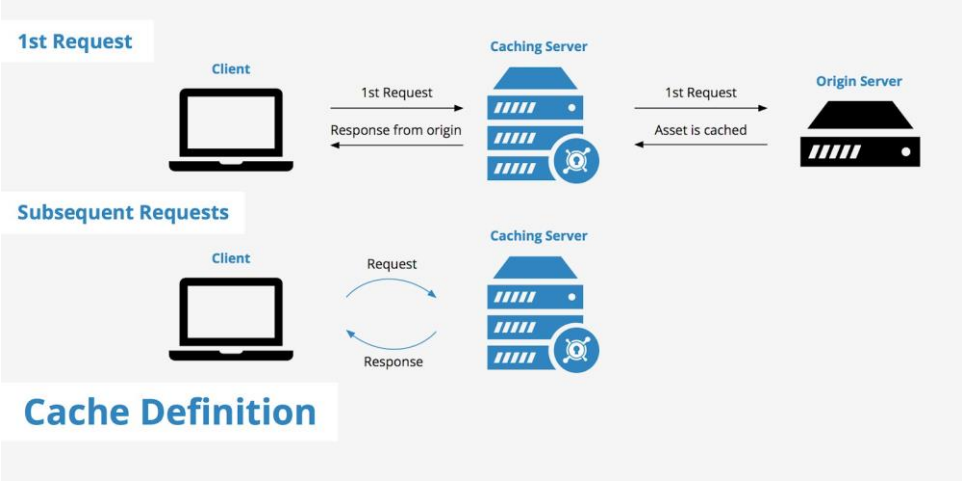


Imagen 5. Representación de funcionalidad de caché en arquitectura de red.

Ejercicio 8:

En nuestra aplicación “RestApi”, agregar una imagen descriptiva única para cada post. en el listado del ejercicio anterior, utilizando además las técnicas de caché de imagen, todo con Picasso.

1. Implementamos la librería de Picasso en nuestro archivo gradle app, con el siguiente código:

```
implementation ('com.squareup.picasso:picasso:2.71828') { exclude group: 'com.android.support'
exclude module: ['exifinterface', 'support-annotations'] }
```

Nota: se hace la exclusión de estos módulos de anotación dado que dan un error con nuestra librería de soporte appcompat-v7:28.0.0. Con el exclude evitamos el error. Esto puede ser resuelto con el tiempo y que hay que estar atento a los cambios en la documentación de la librería.

2. Usaremos la siguiente dirección web de una imágen para ser cargada con Picasso y asociarla a cada post de nuestra lista: https://cdn.pixabay.com/photo/2014/03/25/16/54/envelope-297570_960_720.png

3. Abrimos nuestro archivo layout “posts_list.xml” para agregar una vista de tipo ImageView, antes del TextView existente.

```
<ImageView
    android:id="@+id/imgPost"
    android:layout_width="60dp"
    android:layout_height="40dp"
    android:layout_marginTop="10dp"
 android:layout_marginRight="10dp"/>
```

4. Como la imágen compartirá la línea de la carta con el texto del título cambiaremos el RelativeLayout existente por un LinearLayout con orientación horizontal.

```
<LinearLayout
    android:orientation="horizontal"
    android:id="@+id/relativeLayout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="8dp"
    android:paddingLeft="8dp"
    android:paddingRight="8dp">
```

5. A continuación, modificaremos nuestra clase “CustomAdapter” para agregar la carga de la imagen con Picasso, en primer lugar creamos las instancia de la vista en la inner class “CustomViewHolder” ya existente, de esta forma:

```
inner class CustomViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
   val mTitle: TextView = itemView.findViewById(R.id.txtTitle)
   val mImgPost: ImageView = itemView.findViewById(R.id.imgPost)
}
```

6. Modificamos el método onBindViewHolder() para agregar el contenido de esta vista ImageView con Picasso de la siguiente manera:

```
override fun onBindViewHolder(holder: CustomViewHolder, position: Int) {

   holder.mTitle.text = data[position].title
   Picasso.get().load("https://cdn.pixabay.com/photo/2014/03/25/16/54/envelope-
297570_960_720.png").into(holder.mImgPost)
  }
```

7. Finalmente, debajo de la última línea de código agregado para la carga de Picasso, vamos a indicar un parámetro muy útil que nos hará saber si la imagen fue cargada a través de la red, de la memoria del dispositivo o del almacenamiento del dispositivo, con un color en la parte superior izquierda de la imagen que se vería de la siguiente forma:

Imagen 6. Representación de colores de origen de imágenes.

El código a agregar es el siguiente:

```
Picasso.get().setIndicatorsEnabled(true)
```

8. Como nos hemos dado cuenta, Picasso maneja casi automáticamente el caché de nuestras imágenes, decidiendo cómo manejarlas para mejorar el rendimiento, este es uno de los grandes potenciales de esta librería y el porqué es tan popular. Al correr la aplicación, la visualizamos de la siguiente manera:

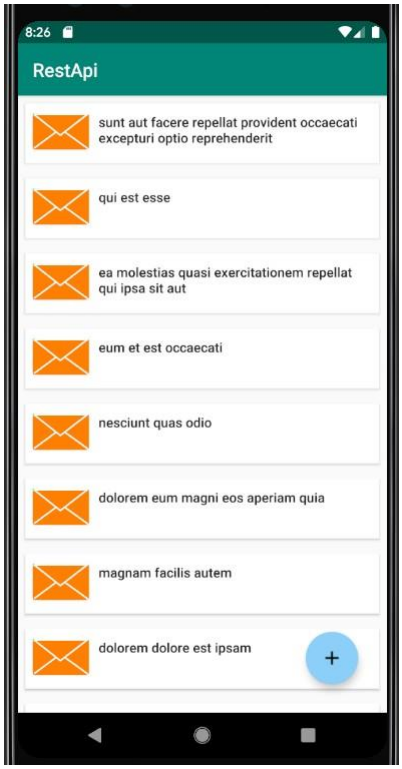


Imagen 7. Picasso load image en lista de posts con caché de memoria (verde).

Glide

Glide es un marco de carga de imágenes y gestión de medios de código abierto (open source) rápido y eficiente para Android, que ofrece una alternativa a Picasso. Glide envuelve la decodificación de medios, el almacenamiento en caché de memoria y disco, además de ofrecer la agrupación de recursos en una interfaz simple y fácil de usar.

Imagen 8. Logo Glide.

Glide admite la obtención (fetching) , decodificación (decode) y visualización de imágenes fijas de video (video stills), imágenes y GIF animados. Glide incluye una API flexible que permite a los desarrolladores conectarse a casi cualquier pila de red (Network stack). De forma predeterminada, Glide utiliza una pila basada en HttpUrlConnection personalizada, pero también incluye bibliotecas de utilidades conectadas al proyecto Volley de Google o la biblioteca OkHttp de Square.

El enfoque principal de Glide es hacer que el desplazamiento de cualquier tipo de lista de imágenes sea lo más suave y rápido posible, pero Glide también es efectivo para casi cualquier caso en el que necesite buscar, cambiar el tamaño y mostrar una imagen remota.

Implementar Glide en un proyecto android

Para agregar las librerías de Glide, lo hacemos de la siguiente forma en nuestro archivo gradle de app:

```
dependencies {
  implementation 'com.github.bumptech.glide:glide:4.10.0'
  annotationProcessor  'com.github.bumptech.glide:compiler:4.10.0'
}
```

Por último para implementar glide en nuestras aplicaciones, podemos seguir el siguiente ejemplo en una actividad:

```
// For a simple view:
@Override public void onCreate(Bundle savedInstanceState) {
  ...
  ImageView imageView = (ImageView) findViewById(R.id.my_image_view);

  Glide.with(this).load("http://goo.gl/gEgYUd").into(imageView);
}

// For a simple image list:
@Override public View getView(int position, View recycled, ViewGroup container) {
  final ImageView myImageView;
  if (recycled == null) {
    myImageView = (ImageView) inflater.inflate(R.layout.my_image_view, container, false);
  } else {
    myImageView = (ImageView) recycled;
  }

  String url = myUrls.get(position);

  Glide
    .with(myFragment)
    .load(url)
    .centerCrop()
    .placeholder(R.drawable.loading_spinner)
    .into(myImageView);

  return myImageView;
}
```

Fresco

Fresco es una biblioteca poderosa para mostrar imágenes en Android. Descarga y almacena en caché imágenes remotas de manera eficiente en la memoria, utilizando una región especial de memoria no recolectada en Android llamada ashmem.

Imagen 9. Logo Fresco.

Al trabajar con Fresco, es útil estar familiarizado con los siguientes términos:

- **ImagePipeline**: responsable de obtener la imagen. Se obtiene de la red, un archivo local, un proveedor de contenido o un recurso local. Mantiene un caché de imágenes comprimidas en el almacenamiento local, y un segundo caché de imágenes descomprimidas en la memoria.
- **Drawee**: los drawees se ocupan de representar imágenes en la pantalla y se componen de tres partes:
    - **DraweeView**: la vista que muestra la imagen. Se extiende desde ImageView, pero solo por conveniencia. La mayoría de las veces usaremos SimpleDraweeView en el código.
    - **DraweeHierarchy:** Fresco ofrece mucha personalización, podemos agregar una imagen de marcador de posición, una imagen de reintento, una imagen de falla, una imagen de fondo, etc. La jerarquía es la que hace un seguimiento de todos estos elementos dibujables y cuándo deben mostrarse.
    - **DraweeController**: esta es la clase responsable de tratar con el cargador de imágenes. Fresco nos permite personalizar el cargador de imágenes si no deseamos utilizar el ImagePipeline.

Implementar Fresco en un proyecto android

Para implementar la librería Fresco de Facebook en nuestro proyecto, es necesario agreguemos su dependencia en el archivo gradle app, de la siguiente manera:

```
dependencies {
 implementation  'com.facebook.fresco:fresco:1.13.0'
}
```

Inicializando Fresco:

```
Fresco.initialize(this)
```

SimpleDraweeView en reemplazo de ImageView en tu archivo layout:

```
<com.facebook.drawee.view.SimpleDraweeView
   android:id="@+id/posterImage"
   android:layout_width="match_parent"
   android:layout_height="500dp"
          ---------------
   fresco:placeholderImage="@drawable/ic_broken_image"  />
```

Para cargar imágenes el código es el siguiente:

```
posterImage : SimpleDraweeView = findViewById(R.id.posterImage)
imgURI : Uri = Uri.parse("
https://hogaraldia.cl/fit/c/240/240/1*SF2VIRFshYt2etl6OhNm_Q.png")
posterImage.setImageURI(imgURI)
```

3.4 Testing con Kotlin

Test Unitario en Android con Kotlin

Por supuesto, Kotlin también nos permite hacer test unitarios sobre el código Java de forma muy sencilla, y muy parecida a lo que estamos acostumbrados en Java.

Hay alguna pequeña cosa que se complica cuando utilizamos librerías como Mockito, pero veremos algunos trucos para que sea más sencillo.

Aunque el tema de lo que es un test unitario siempre genera controversia, no voy a entrar aquí en detalles sobre lo que es o no es un test unitario.

Para nuestro ejemplo en particular, es suficiente pensar que un test unitario es aquel test que no necesita un dispositivo para ejecutarse. El IDE será capaz de ejecutarlos y mostrar un resultado, identificando cuáles se ejecutaron y cuáles fallaron.

Configura Gradle

Necesitas añadir jUnit a tus dependencias. Es posible que cuando creaste el proyecto ya se incluyera por defecto. También vamos a añadir Mockito, pues lo utilizaremos después:

```
testCompile "junit:junit:4.12"
testCompile "org.mockito:mockito-core:1.10.19"
```

Crea tu primer test

En la carpeta app/src/test (créala si no existe), puedes crear una nueva clase llama MyTest, que tenga la siguiente forma:

```
class MyTest {

    @Test
    fun testsWork() {
        assertTrue(true)
    }
}
```

Como ves, es muy similar a lo que estás acostumbrado en Java, con la nomenclatura típica de Kotlin.

Cómo usar Mockito

Mockito en Kotlin se puede usar como cualquier otra librería, si bien es cierto que te vas a encontrar con ciertas complicaciones que vas a necesitar solventar.

Aquí tienes un ejemplo extraído del libro:

```
@Test
fun emptyDatabaseReturnsServerValue() {
    val db =
Mockito.mock(ForecastDataSource::class.java)
    val server =
Mockito.mock(ForecastDataSource::class.java)

`when`(server.requestForecastByZipCode(any(Long::
class.java), any(Long::class.java)))
            .then { ForecastList(0, "city",
"country", listOf()) }

    val provider = ForecastProvider(listOf(db,
server))
    assertNotNull(provider.requestByZipCode(0,
0))
}
```

Como ves, todo es muy similar. Puedes crear tus mocks y utilizarlos sin problemas a lo largo del código. Aunque aquí no lo estoy haciendo, también podrías usar MockitoJUnitRunner y anotaciones.

La palabra when es una palabra reservada en Kotlin, y por eso es necesario usarla con comillas invertidas, o incluso podrías renombrar el import y darle el nombre que quieras:

```
import org.mockito.Mockito.`when` as _when
```

El problema surge cuando se intentan mockear tipos que no permiten valores nulos. Mockito por defecto lo que hace es dar valores nulos a los objetos que mockea, con lo que tarde o temprando surgirán problemas.

Un pequeño truco es usar alguna librería como mockito-kotlin, que en vez de usar nulos, dará valores por defecto específicos a cada tipo, solventando ese problema. Además, da otras funciones que aprovechan la potencia de Kotlin para hacer las cosas más sencillas.

Otro problema es que, por defecto, todas las clases y funciones en Kotlin son cerradas, lo que quiere decir que no se pueden extender. Esto es un problema para Mockito, pues no es capaz de mockearlas.

La única solución hoy en día es crear interfaces para todo, o hacer open todo lo que queramos mockear. Pero esto va a dejar de ser un problema pronto, pues Mockito 2 permite mockear objetos finales, y aunque aún no tenemos una versión final (actualmente es Release Candidate), ya es perfectamente usable.

Unidad 4

Publicación de aplicaciones y analítica

4.1 Release
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 Gradley 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*.

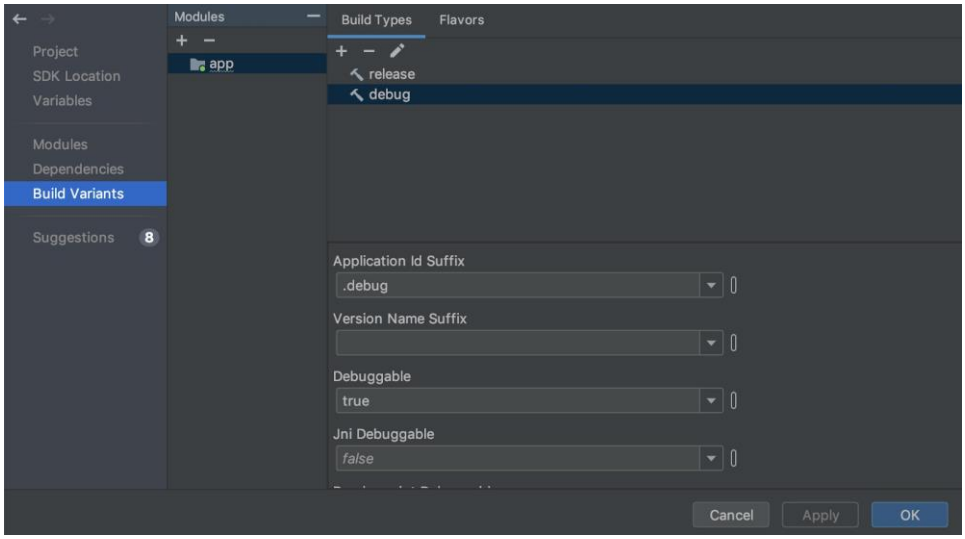


Imagen 2. Propiedades del tipo de configuración.

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 son 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

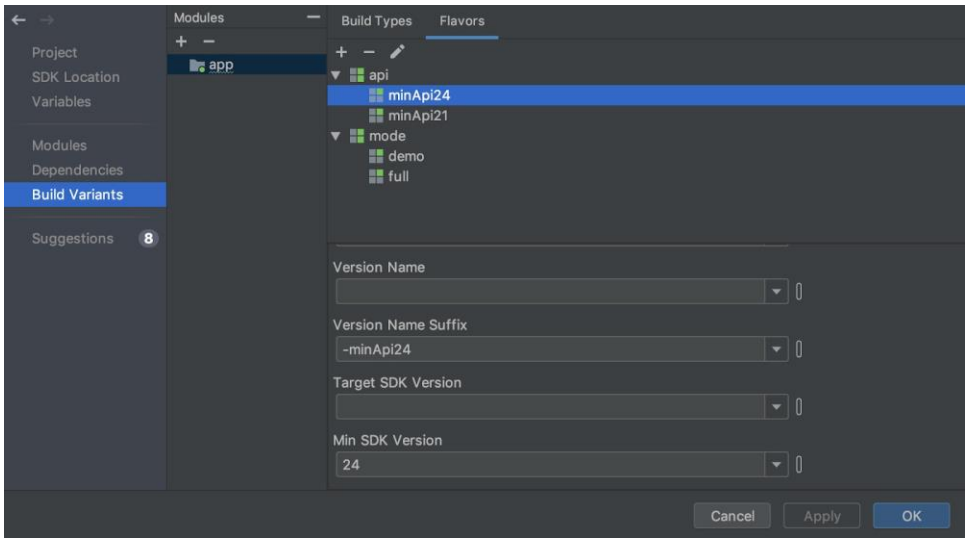


Imagen 3. Administración de configuraciones.

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

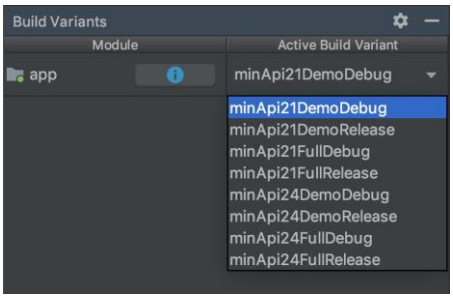


Imagen 4. Identificar variables de compilación.

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 24Flavor para versión mínima de API 26
- Sincronizar el proyecto
- Verificar que las Build Variants generadas correspondan a las indicadas

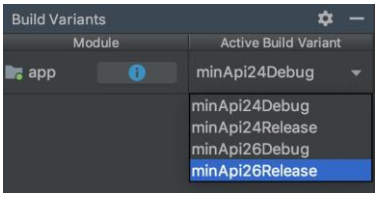


Imagen 5. Ejercicio.

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 sobreescribe 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* requiere 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

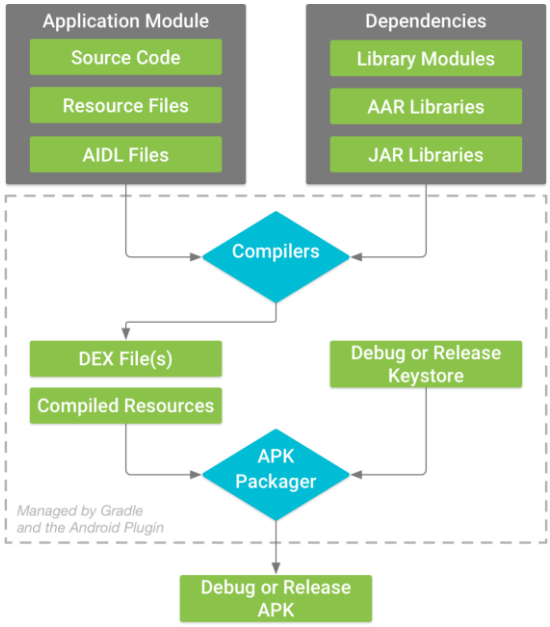


Imagen 6. Proceso de compilación.

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 deaplicaciones o en otras secciones. Además, los servicios de publicación muestran ese ícono en susbúsquedas

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 ambientesUna forma de manejar esto es inyectando variables al manifesto.

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 sobreescrito 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 sobreescrito 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 compañíaEl 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

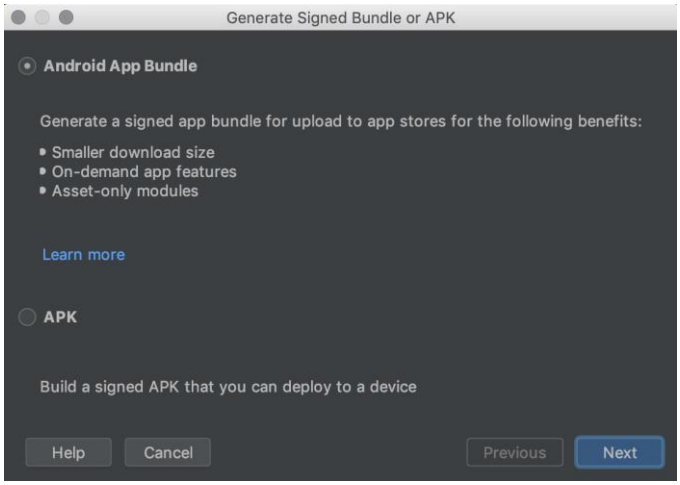


Imagen 11. Generate Signed Bundle or APK.

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

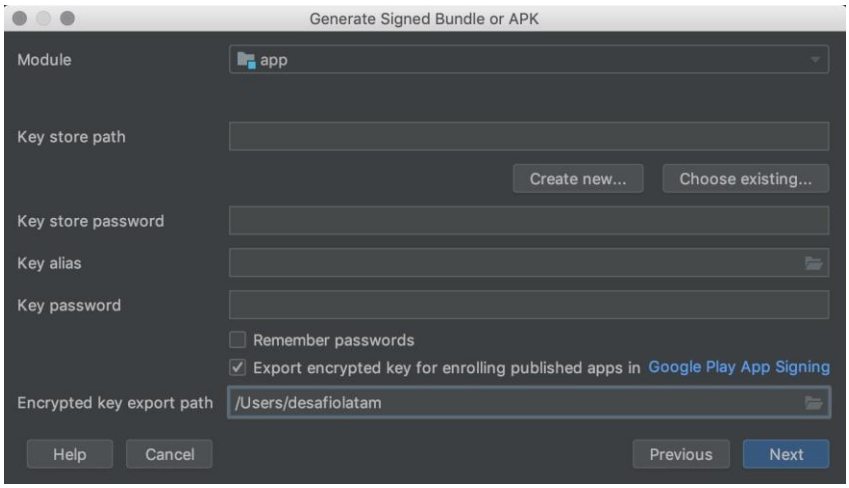


Imagen 12. Crear certificado.

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

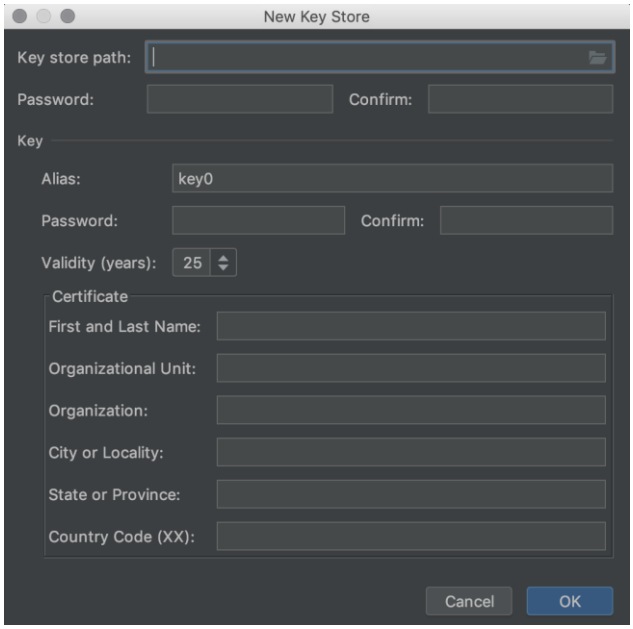


Imagen 13. Configuración de nuevo certificado.

- 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

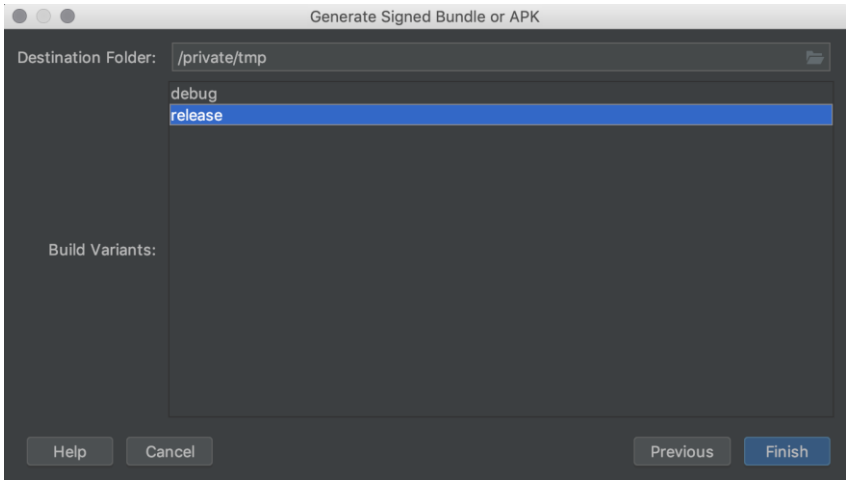


Imagen 14. Seleccionar Buold Variant.

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

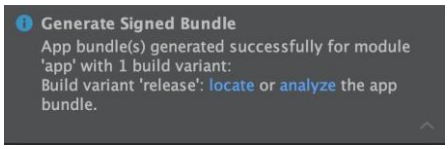


Imagen 15. Creación del Bundle.

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.

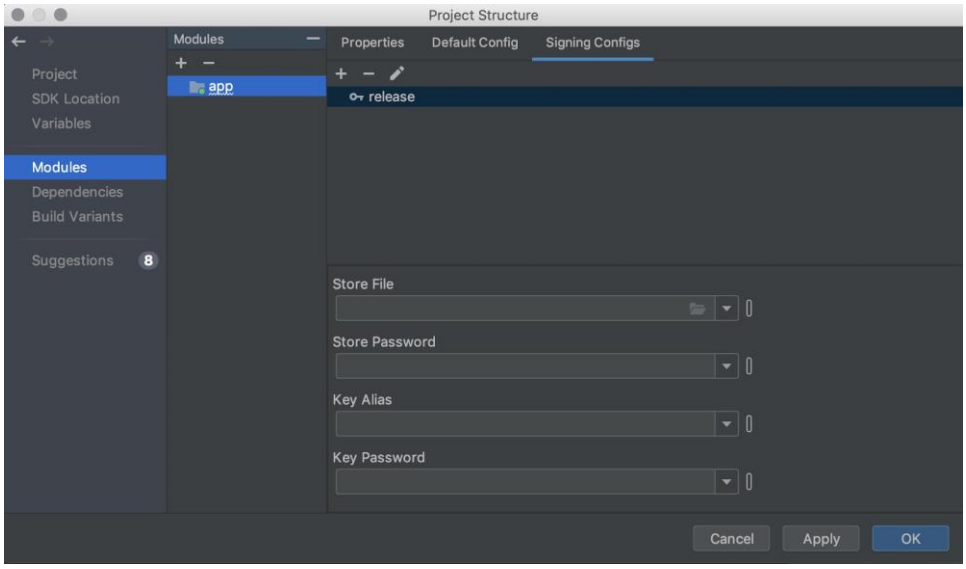


Imagen 16. Generar automáticamente una release app.

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

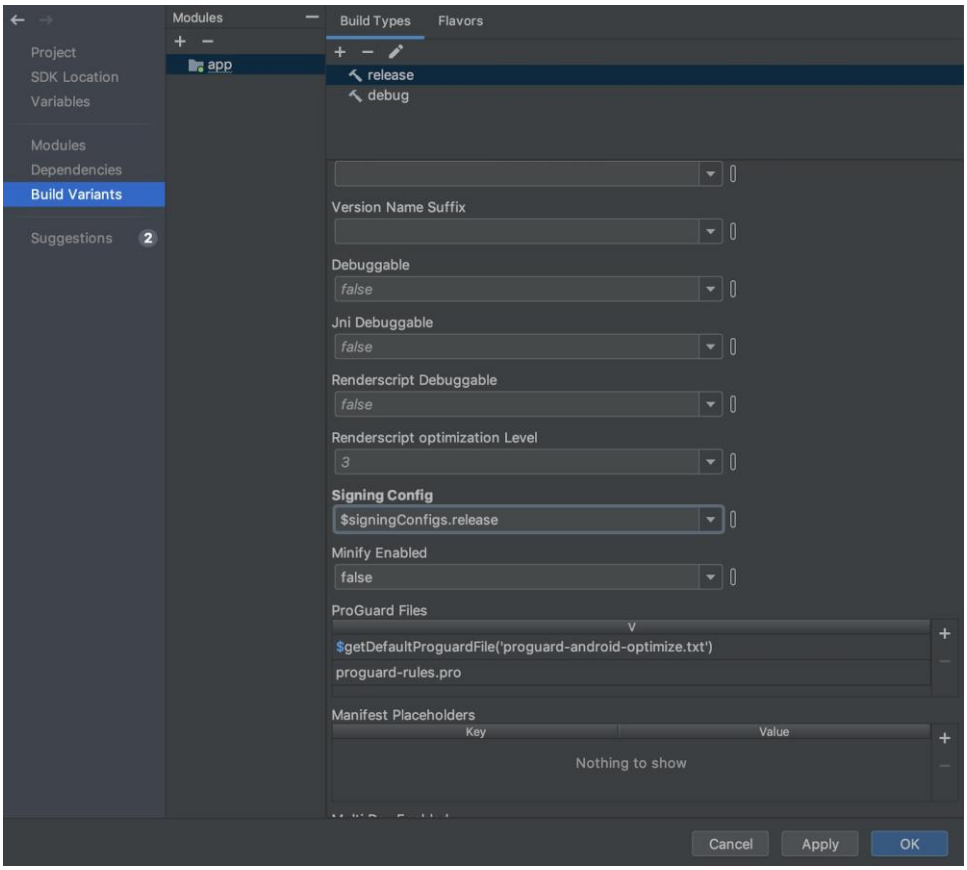


Imagen 17. Editar el Build Type.

4.2 Distribución de la Aplicación

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

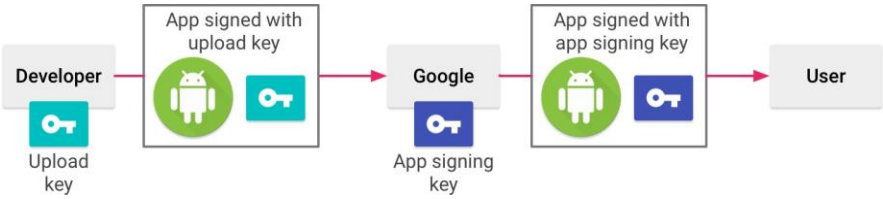


Imagen 18. Firma de apps en Google Play.

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.

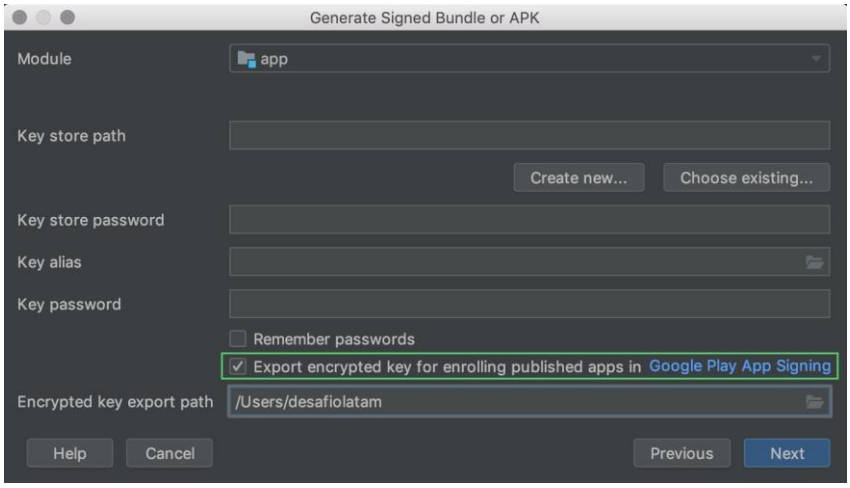


Imagen 19. Exportar claves encriptadas.

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 -> SeguridadEstas instalaciones se bloquean hasta que el usuario lo permita activando explícitamente **fuentes desconocidas** en Ajustes > Seguridad.

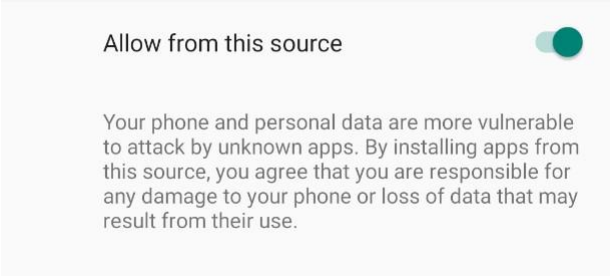


Imagen 20. Activar fuente desconocida.

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

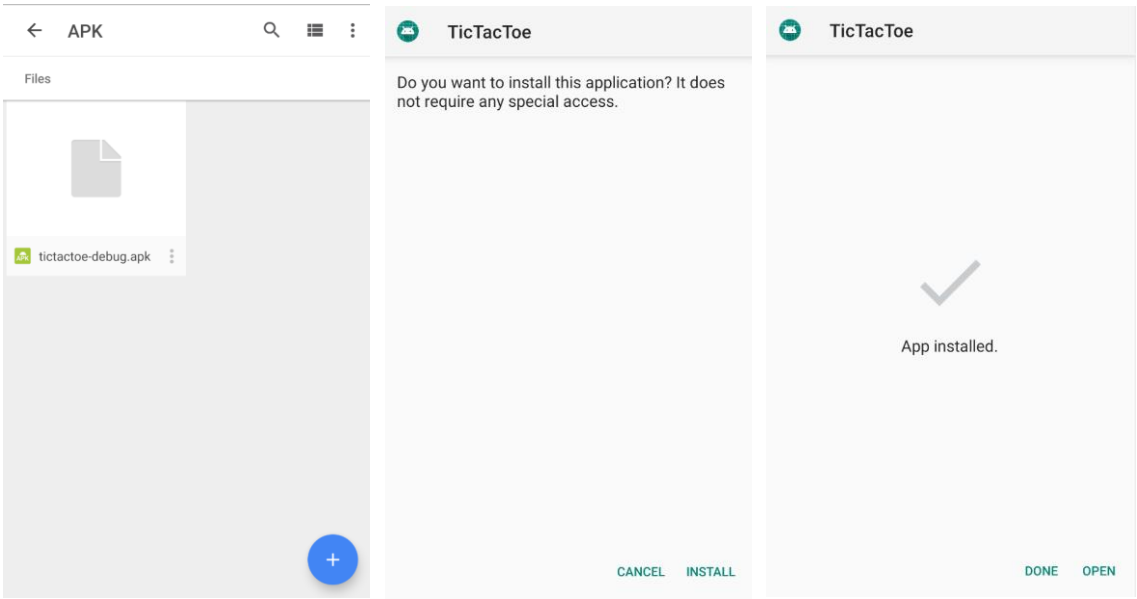


Imagen 21. Grupo de imágenes - Distribuir usando Google Drive.

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

Subir app a Google Play Store

Competencias

- Conocer la información obligatoria para publicar en Google Play.
- Conocer los requisitos de la ficha de Google Play.
- Manejar las distintas pistas de publicación.
- Conocer el flujo de lanzamiento de una app.

Introducción

En este capítulo vamos a revisar la forma de llenar una ficha de publicación en google Play. Es de vital importancia que conozcamos los detalles de esta tarea para que nuestras aplicaciones puedan estar disponibles para nuestros usuarios en el menor tiempo posible

Este capítulo describe el flujo para subir la app a la tienda de aplicaciones usando las pistas de lanzamiento que mejoran la calidad de al app entregando mejoras en forma paulatina a los usuarios

Crear la aplicación en la consola

Se debe crear una aplicación en la consola que tiene la información asociada a la app y el instalador que debe utilizar. Para esto se deben seguir los siguiente pasos:

- Ir a la Play Console
- Crear una aplicación usando el botón CREA UNA APLICACIÓN
- Rellenar el título y el idioma predeterminado para luego crear la app

Imagen 1. Crear aplicación en consola.

La redirección es hacia la configuración de la app, donde en la parte izquierda se indica los componentes obligatorios que deben ser completados antes de realizar el lanzamiento

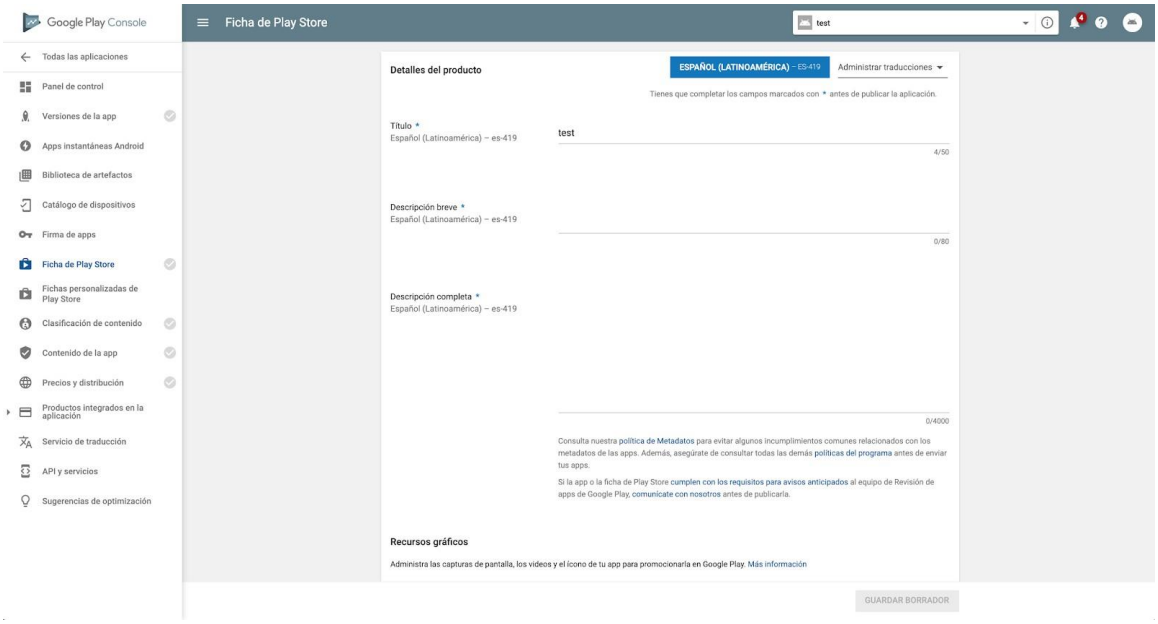


Imagen 2. Configuración de la app.

Para poder hacer el lanzamiento de la app se deben completar las partes obligatorias indicadas con un símbolo check desactivado

Clasificación de contenido

En la tienda se clasifican las apps en distintas categorías por público objetivo, contenido de la app y si es de pago o gratis. Al crear una nueva app se debe entregar información para determinar los criterios de clasificación.

Al seleccionar desde el menú *Contenido de la app* se puede realizar la categorización contestando una serie de preguntas de contenido y precio

Imagen 3. Clasificación de contenido.

Contenido de la app

La clasificación del tipo de app se calcula en forma automática luego de contestar una serie de preguntas orientadas a verificar si cumple con las políticas del programa para desarrolladores que deben ser respetadas para que la app pueda ser publicada.

Entre los tópicos consultados se encuentra el uso de la violencia, miedo y humor crudo

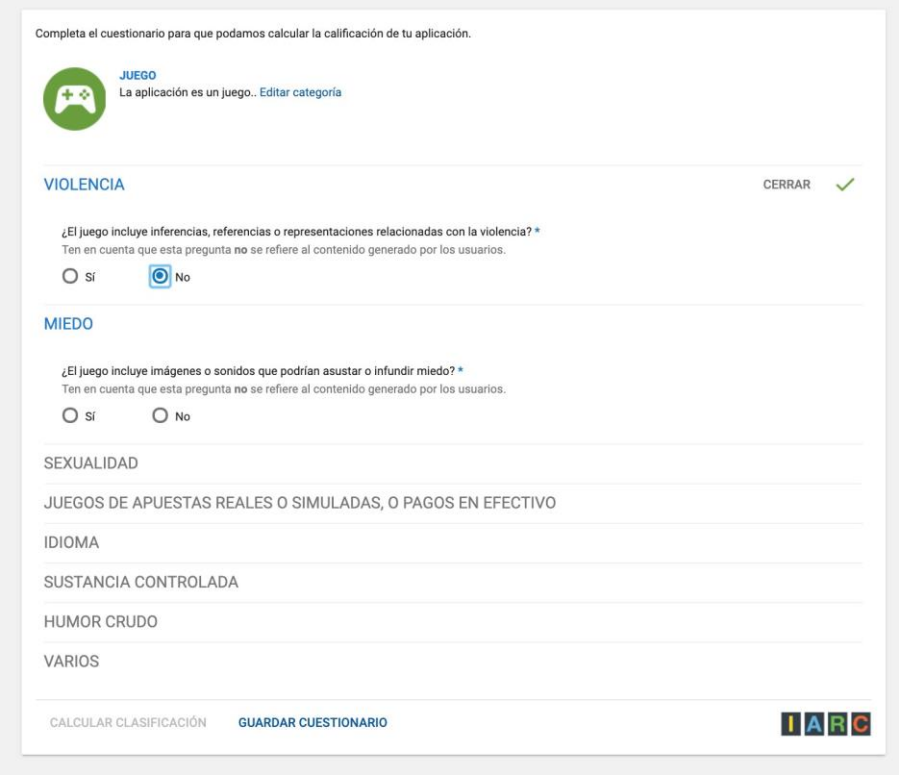


Imagen 4. Ejemplo de tópico consultado.

Estas políticas se van actualizando constantemente y las aplicaciones deben actualizarse junto con ellas. Generalmente existen notificaciones a los desarrolladores con las actualizaciones mayores de las políticas, y siempre quedan disponibles en la documentación oficial los cambios realizados.

Precios y distribución

En la siguiente parte se indica si es una app de pago o gratis además de configurar los países en los que estará disponible la app.

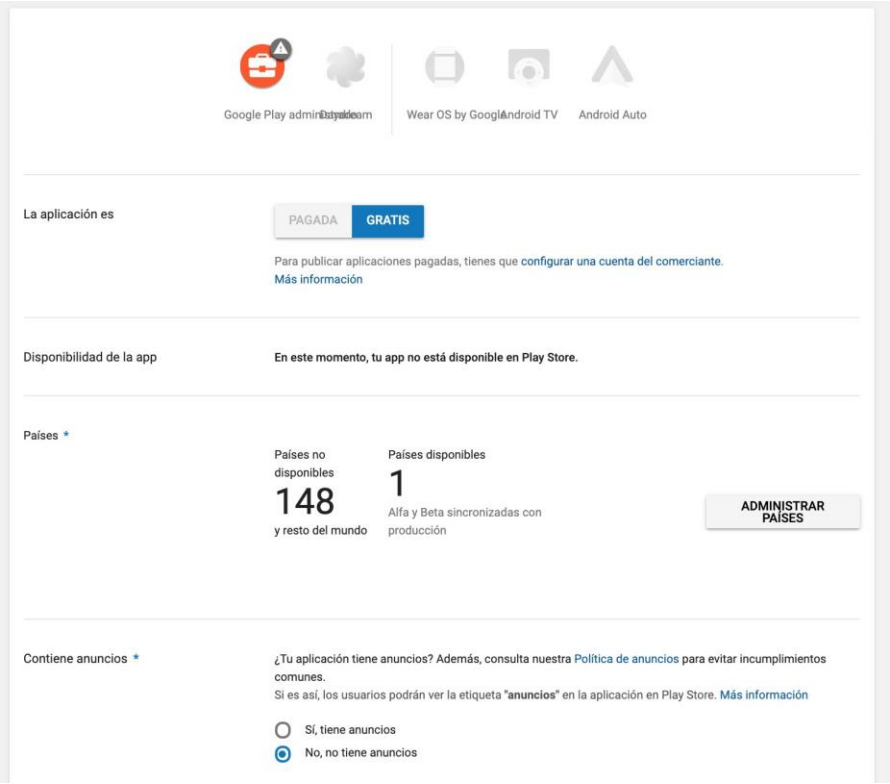


Imagen 5. Precios y distribución.

Ficha de Play Store

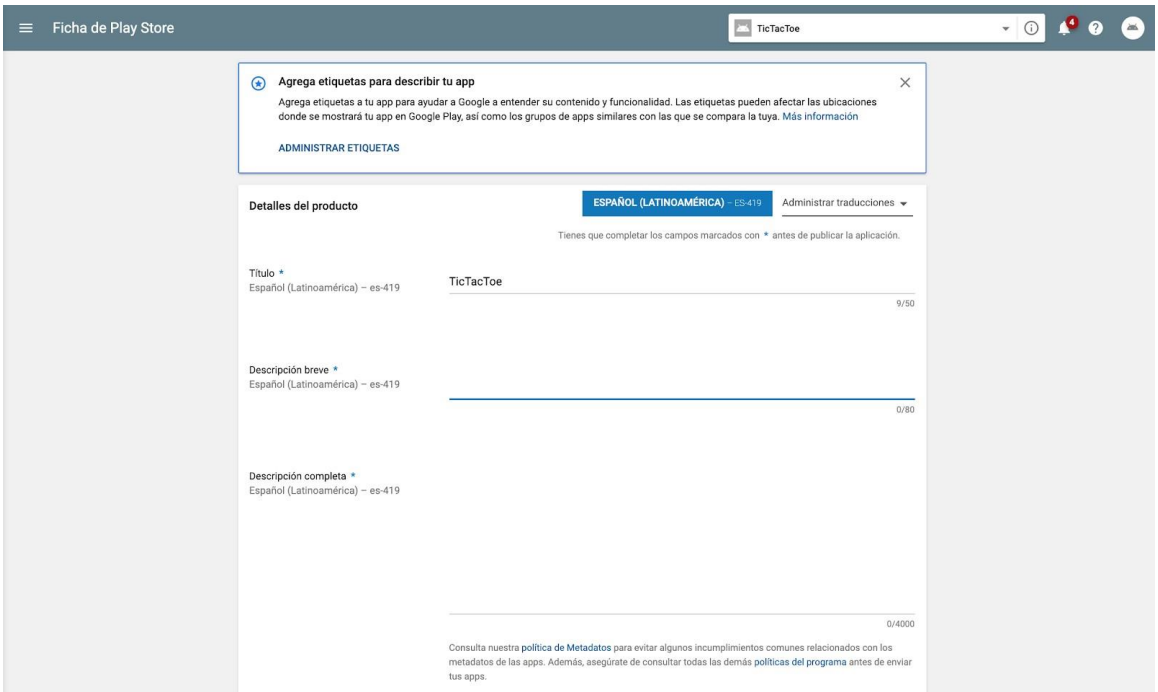


Imagen 6. Ficha de Play Store.

La ficha de Play Store para el detalle del producto recopila información para ser desplegada en la tienda de aplicaciones y está separada en secciones

Detalle del producto

La mayoría de los detalles del producto son obligatorios y corresponden a la información básica que se muestra en la tienda de aplicaciones

- Título
- Descripción breve
- Descripción completa

Recursos Gráficos

Los recursos gráficos serán mostrados al usuario dentro de la tienda de aplicaciones y corresponden a íconos, capturas de pantalla y videos promocionales. La documentación oficial ofrece guías actualizadas sobre formatos, restricciones y configuraciones, que quedan disponibles para todas las pistas

- Ícono de alta resolución
- Capturas de pantalla
- Gráfico de la función
- Video promocional

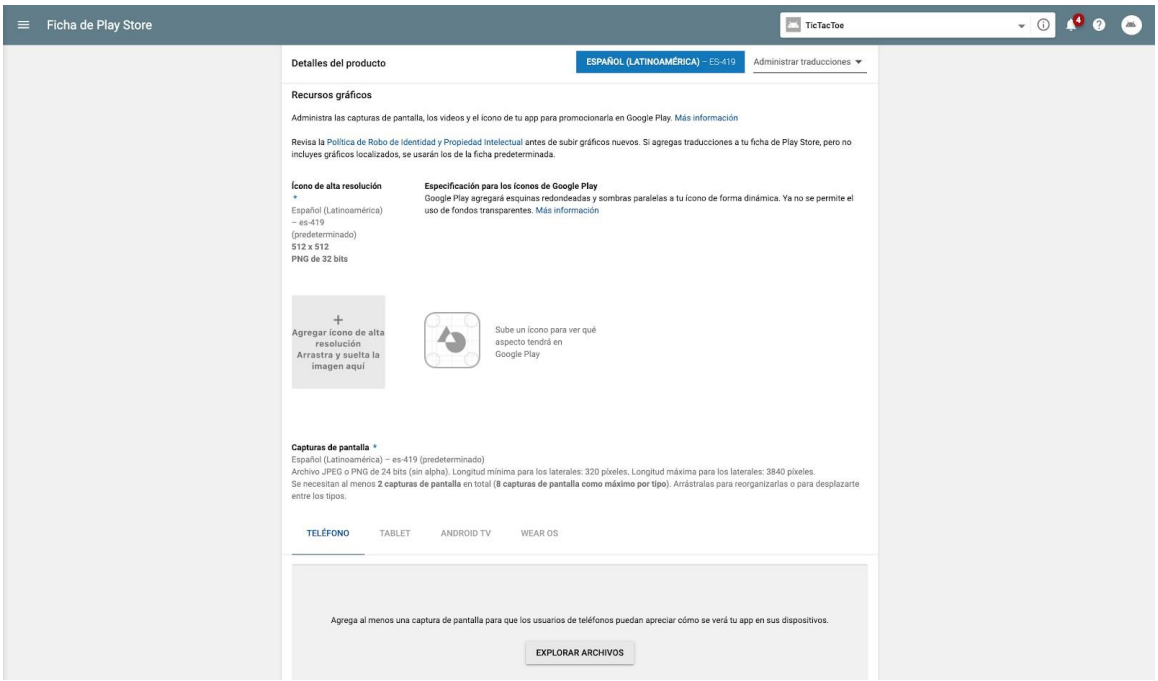


Imagen 7. Recursos gráficos de Play Store.

Categorización

La categorización permite colocar a la app en una categoría específica y administrar las etiquetas que describen el contenido y la funcionalidad de la appEl tipo de aplicación indica si es un juego o una aplicación, en tanto las categorías disponibles corresponden a Arte y diseño, Educación, Productividad entre muchas otras

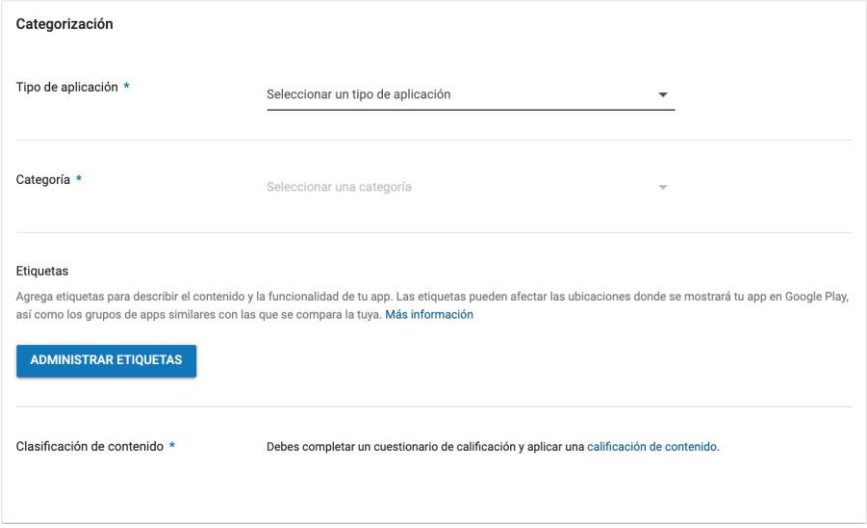


Imagen 8. Categorización.

Las etiquetas describen el contenido y la funcionalidad de la app. Se manejan hasta 5 etiquetas de una lista amplia disponible

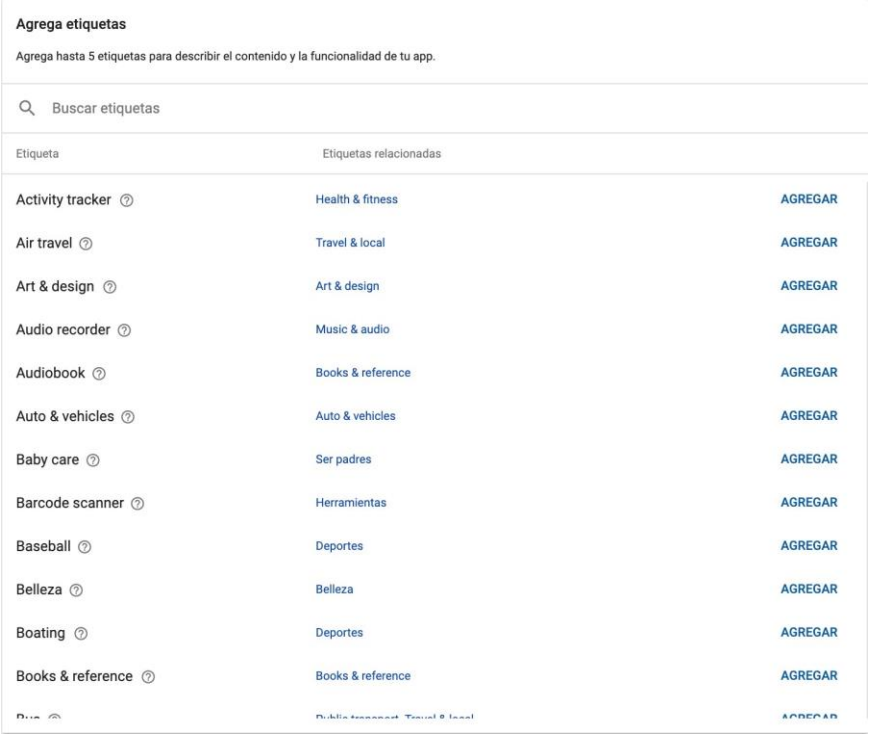


Imagen 9. Agregar etiquetas.

Con esta última información, la ficha de Google Play está completa

Versiones de la app

En la consola de Google Play se encuentra en la parte izquierda el menú con la opción de Versiones de la app.

Aquí se permite hacer el lanzamiento de la app de forma ordenada y escalonada

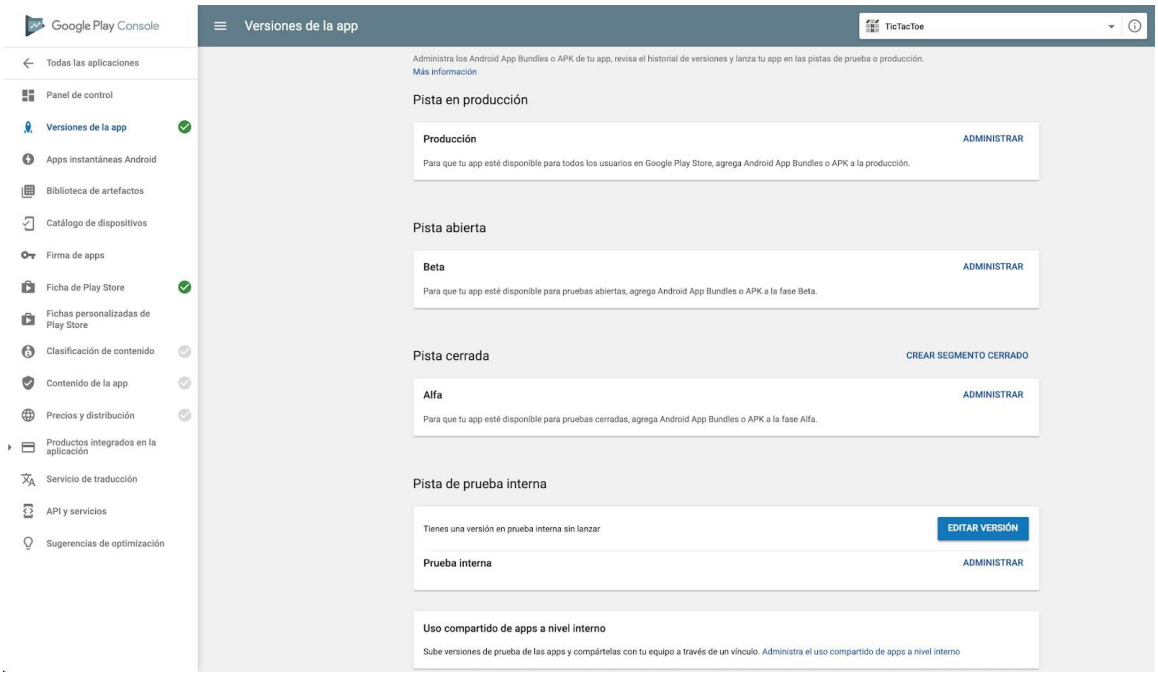


Imagen 10. Versiones de la app.

Para que las apps en las distintas pistas estén disponibles para los verificadores, deben tener un **código de versión más alto** que el de la versión de producción.

Pista de prueba interna

La versión de prueba interna se usa para distribuir rápidamente la app para **pruebas internas** y **controles de calidad** usando los testers internos (se puede tener hasta 100 *testers*)Entre otras características, se encuentran:

- **Rapidez**: Distribuir apps es mucho más rápido desde la pista de prueba interna que desde las pistas abiertas o cerradas. Al publicar un APK o app bundle actualizado en la pista de prueba interna, estará disponible para los verificadores en pocos minutos.
- **Flexibilidad**: Las pruebas internas se pueden ajustar para admitir distintas etapas de prueba. Esto incluye pruebas internas, verificaciones de calidad y depuraciones posteriores al lanzamiento.
- **Seguridad**: Con la pista de prueba interna, tu app de prueba se distribuye a los usuarios a través de Play Store.

Siguiendo con el flujo de TicTacToe, se puede administrar la pista de prueba interna e **iniciar el lanzamiento a prueba interna**.

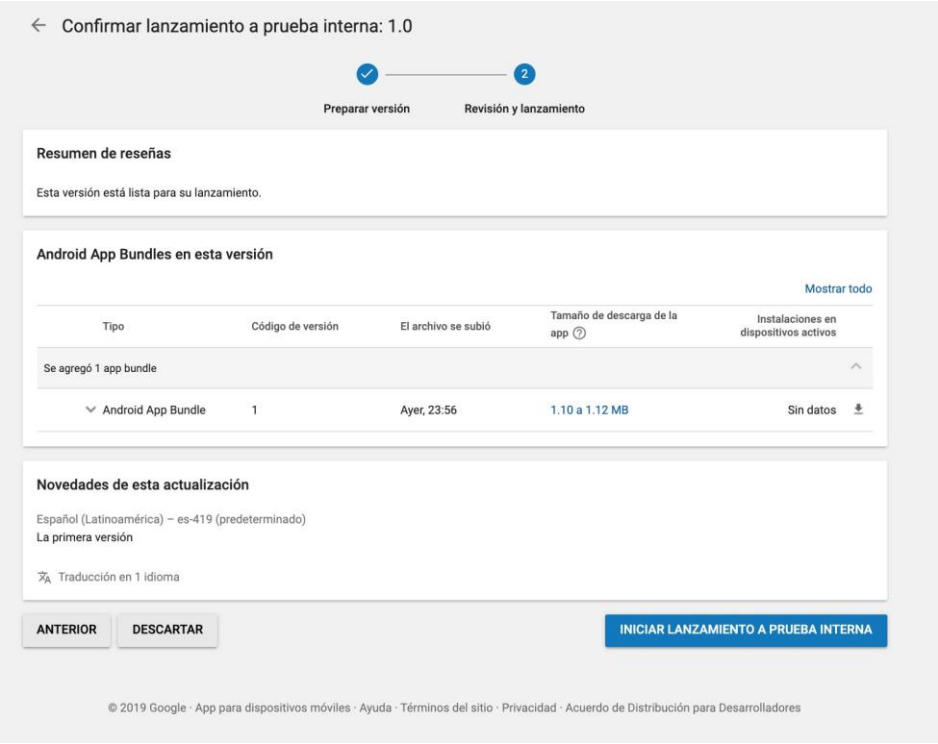


Imagen 11. Iniciar lanzamiento.

Cuando se inicial el lanzamiento, la app pasa a revisión por parte de Google.

En el caso de algunas cuentas de desarrollador, el tiempo para revisar la app es mayor para revisar la app en detalle a fin de proteger a los usuarios. Esto podría generar tiempos de revisión de hasta 7 días (o más en casos excepcionales).

Cuando el estado de publicación de la app cambia a Publicada, la prueba está lista y se puede ver en el home de la consola el nuevo estado

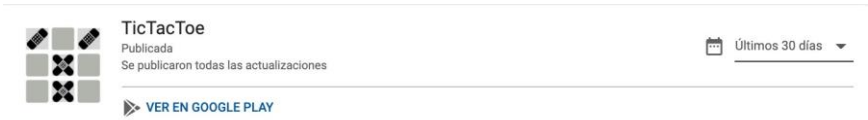


Imagen 12. App publicada.

Administrador de verificadores

La administración de verificadores o *testers* permite inscribir a una lista de usuarios que pueden acceder a la app de prueba interna usando la **URL para participar**

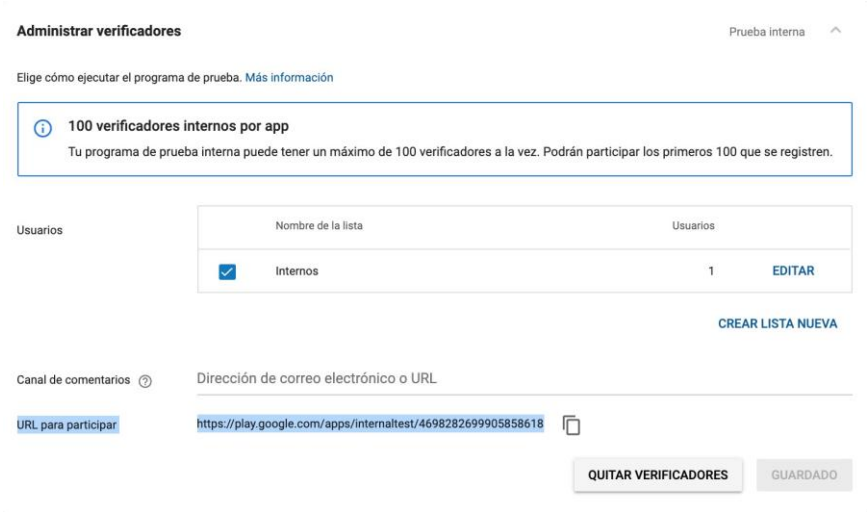


Imagen 13. Administrar verificadores.

Pista cerrada (Alpha)

Los grupos de prueba cerrados deben ser pequeños y probar las versiones más inestables y experimentales de tu app y se puede ejecutar una prueba cerrada con direcciones de correo electrónico individuales

Para esto se debe lanzar a ALFA (Alpha) la app previamente en la pista de prueba interna

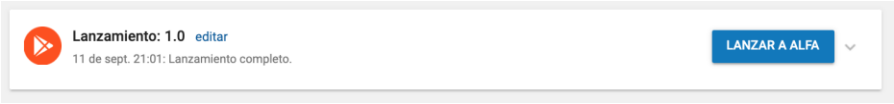


Imagen 14. Lanzamiento a ALFA.

que inicia el borrador para el lanzamiento a alpha

Imagen 15. Inicialización de borrador para lanzamiento.

En el primer paso se prepara la versión que indica la promoción desde la prueba internaLuego, en el paso indica que existen advertencias que deben ser solucionadas antes de hacer el lanzamiento

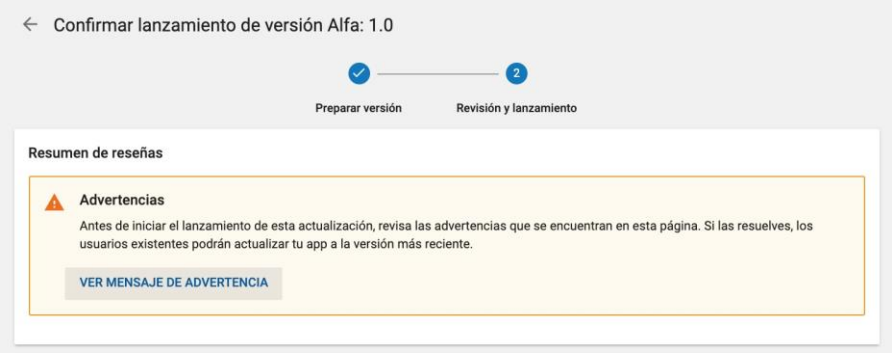


Imagen 16. Confirmar lanzamiento.

En este caso particular de la app TicTacToe no se ha configurado los verificadores, como indica al presionar VER MENSAJE DE ADVERTENCIA

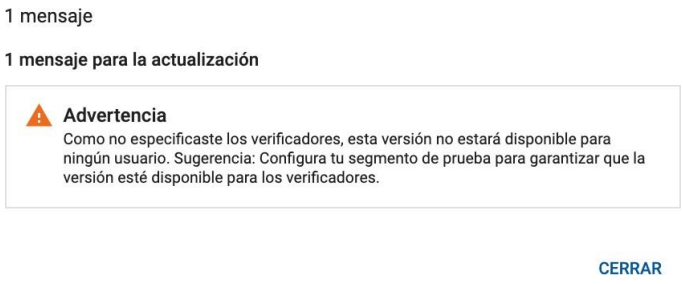


Imagen 17. Visualización de mensaje de advertencia.

No existe un segmento de prueba cerrado configurado por lo que la app no será distribuida a algún usuario

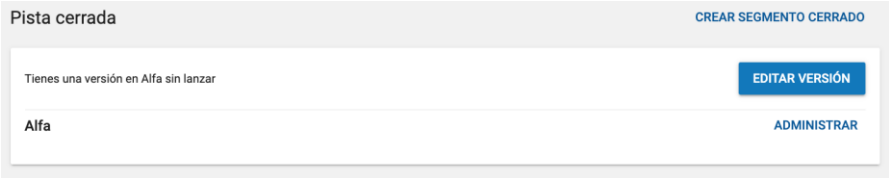


Imagen 18. Pista cerrada.

Luego de crear el segmento, la app aún no sido lanzada. Al editar la versión, la advertencia ha sido solucionada y permite iniciar el lanzamiento de versión alfa

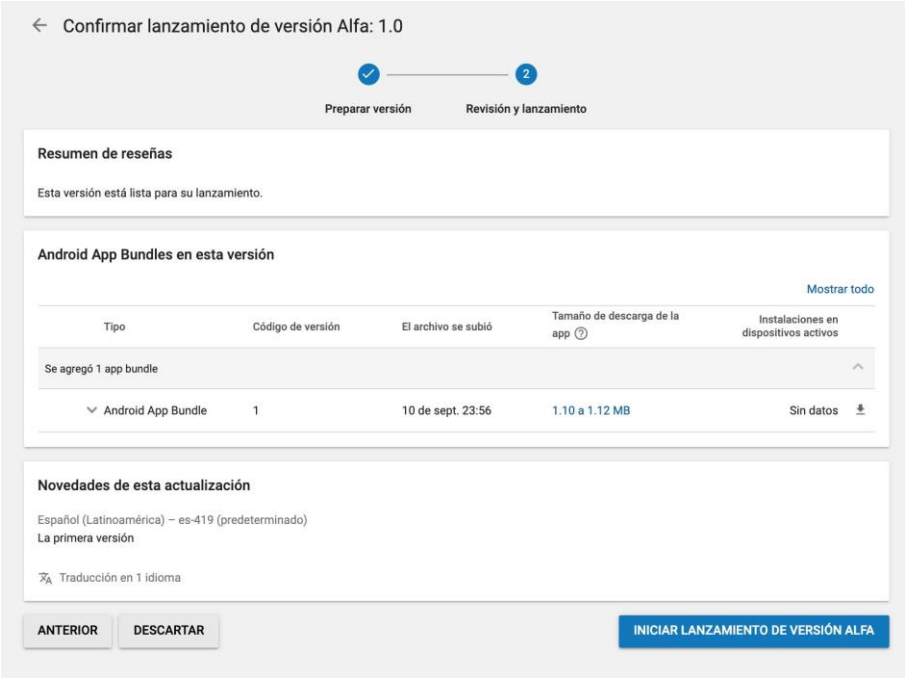


Imagen 19. Advertencia solucionada.

Pista abierta (Beta)

Después de haber probado una versión cerrada, el lanzamiento abierto puede incluir una gama más amplia de usuarios para pruebas, antes de que la app entre en producción.La prueba abierta facilita un *link* que se puede compartir directamente en un foro, a través de una cuenta de correo colectiva o algún otro medio, con la finalidad de conseguir un *beta testing* masivo.

Pista en producción

La versión de producción está disponible para **todos los usuarios** en los países que fueron configurados y es accesible en la tienda de aplicaciones

La publicación en producción supone una espera de al menos 2 hrs para que sea efectiva, por lo que la **actualización no es inmediata**

Flujo de release

El flujo propuesto permite promover la app desde una prueba interna hasta la puesta en producción para probar nuevas funcionalidades y **evaluar** tanto el **desempeño** como la **estabilidad** previa a la puesta en producción

Imagen 20. Pistas de publicación.

Pistas separadas por cantidad de usuarios a los que afecta

Ventajas:

- Disminuye el riesgo de afectar a los usuarios finales con bugs que pueden ser descubiertos en las etapas iniciales
- Probar funcionalidades con grupos de *early users* y comprobar hipótesis de comportamiento
- Administrar a quiénes y cuándo se debe actualizar la app

Cuidados:

- Considerar en la planificación de lanzamiento los tiempos de publicación por cada pista

Google Play

Es una plataforma de distribución digital de aplicaciones móviles para los dispositivos con sistema operativo Android, así como una tienda en línea desarrollada y operada por Google. Esta plataforma permite a los usuarios navegar y descargar aplicaciones (desarrolladas mediante Android SDK), juegos, música, libros, revistas y películas. También se pueden adquirir dispositivos móviles como ordenadores Chromebook, teléfonos inteligentes Nexus, Google Chromecast, entre otros productos de la empresa.

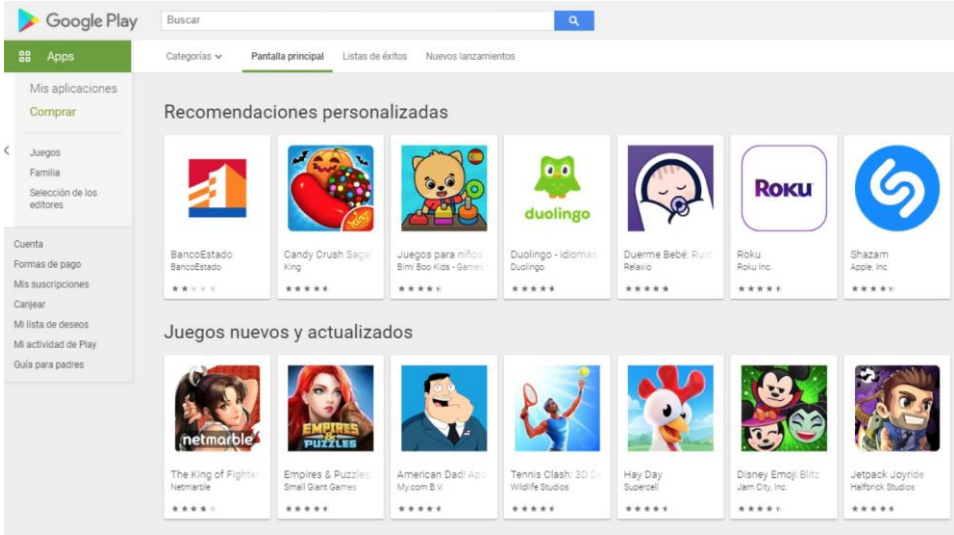


Imagen 21. TIenda Google Play.

Las aplicaciones se encuentran disponibles de forma gratuita o pagadas. Pueden ser descargadas directamente desde un dispositivo con Android a través de la aplicación móvil Play Store.

En Google Play se encuentran disponibles tanto aplicaciones gratuitas como de pago, pudiendo ser instaladas desde el dispositivo o desde la página web de Google Play. Los usuarios también pueden instalar aplicaciones a través de el sitio web del desarrollador o mediante tiendas de distribución digital alternativas. Las aplicaciones pueden ser actualizadas automáticamente si el usuario así lo establece, o pueden hacerlo manualmente una a una.

Google Play Console

La Google Play Console, es una aplicación web de gestión de software asociado a la tienda Google Play. En la console, podrás subir tus propias tus aplicaciones enlazadas a una cuenta personal de desarrollador, administrar, crear productos, consultar reportes y muchísimas más opciones.

Ejercicio 1: Introducción a la consola de desarrolladores de google play

Nuestro primer paso será dirigirnos a la dirección web de la consola de desarrolladores: https://developer.android.com/distribute/console?hl=es , donde seleccionaremos “Acceder”.

Imagen 22. Google Play Console.

Crear Cuenta de Desarrollador

Para publicar apps de Android en Google Play, deberás crear una cuenta de desarrollador de Google Play, siempre y cuando seas mayor de 18 años de edad; esta acción se realiza al acceder como lo indica la imagen de ejercicio 1.

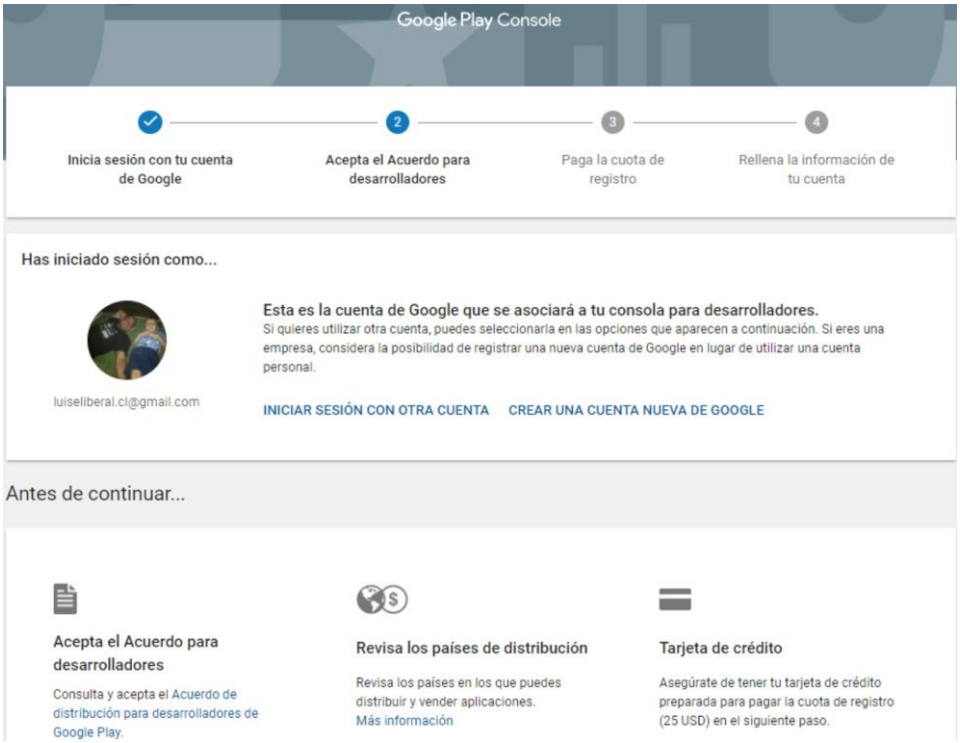


Imagen 23. Crear cuenta de desarrollador.

A continuación los pasos necesarios para concretar la cuenta:

- Regístrate para obtener una cuenta de desarrollador de Google Play.
    - Regístrate con tu Cuenta de Google para obtener una cuenta de desarrollador.
    - Cuando tengas una cuenta de desarrollador, podrás usar Play Console para publicar y administrar tus apps.
- Acepta el Acuerdo de Distribución para Desarrolladores.
    - Durante el proceso de registro, deberás revisar y aceptar el Acuerdo de Distribución para Desarrolladores de Google Play.
- Paga la tarifa de registro.Existe una tarifa de registro única de USD 25 que puedes pagar con las siguientes tarjetas de débito o crédito: MasterCard, Visa, American Express (para Latinoamérica).

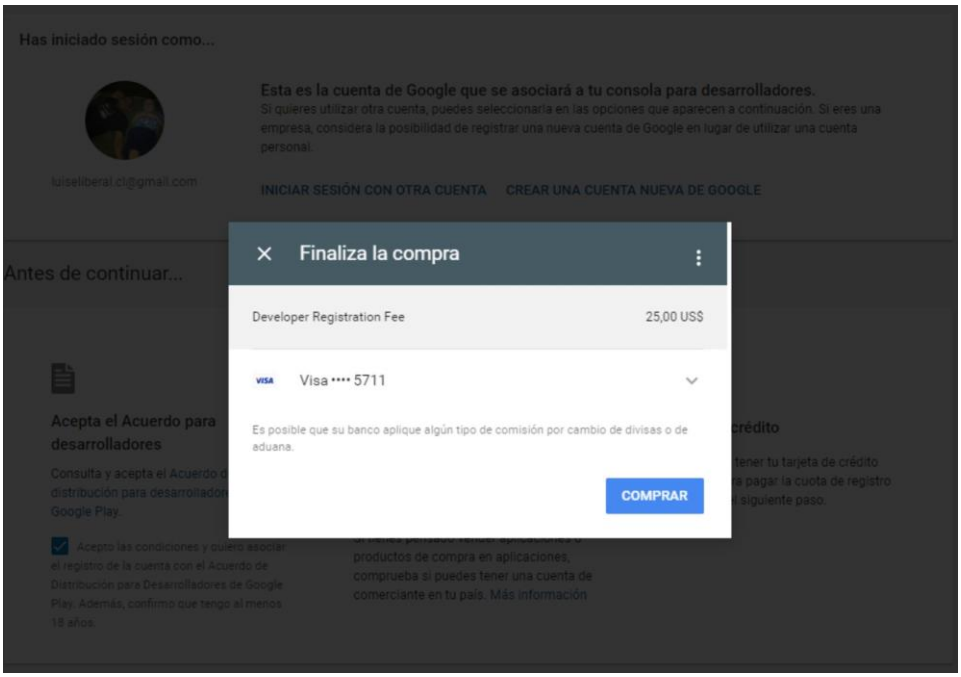


Imagen 24. Finalizar compra.

- Completa los datos de la cuenta.
    - Escribe los detalles de la cuenta. Los clientes pueden ver tu "Nombre del desarrollador" en Google Play. Puedes agregar más información de la cuenta después de crearla.

Cómo usar Play Console

Después de crear y/o seleccionar una app, encontrarás estas páginas en cada categoría:

- **Android vitals**: Información general, Fallas y ANR, Archivos de desfuscación.
- **Herramientas de desarrollo**: Servicios y API.
- **Administración de lanzamientos**: Panel de control de la versión, Versiones de la app, Apps instantáneas Android, Biblioteca de artefactos, Explorador de paquetes de aplicación, Catálogo de dispositivos, Firma de apps, Reporte previo al lanzamiento.
- **Presencia en Google Play Store**: Ficha de Play Store, Experimentos en la Play Store, Precios y distribución, Clasificación de contenido, Productos integrados en la aplicación, Ofertas de aplicaciones pagadas, Servicio de traducción.
- **Adquisición de usuarios**: Informes de adquisición, Campañas de Google Ads, Promociones, Sugerencias de optimización.
- **Informes financieros**: Información general, Ingresos, Compradores, Conversiones, Suscripciones.
- **Comentarios de usuarios**: Calificaciones, Análisis de reseñas, Reseñas, Comentarios de usuarios Beta.

Secciones de la Consola de Google Play

Revisemos en detalle las secciones más importantes de la consola Google Play.

Home

Es la pantalla de inicio donde se muestran tus aplicaciones o crear una, servicios ofrecidos por google, gestión de pedidos realizados a tus aplicaciones, descarga de informes configurados y alertas.

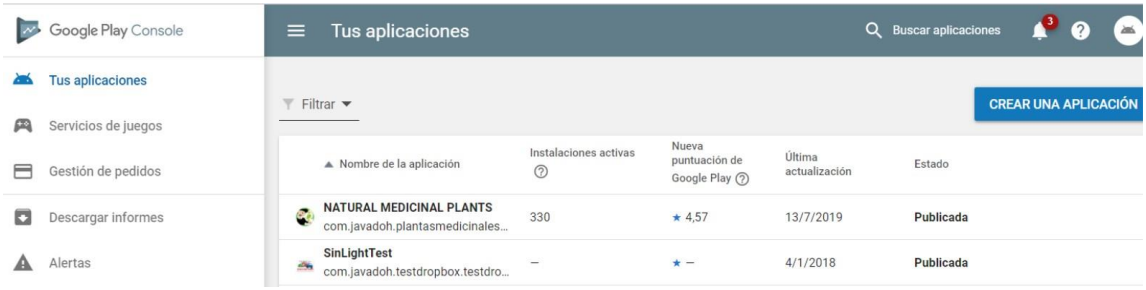


Imagen 25. Home Google Play Console

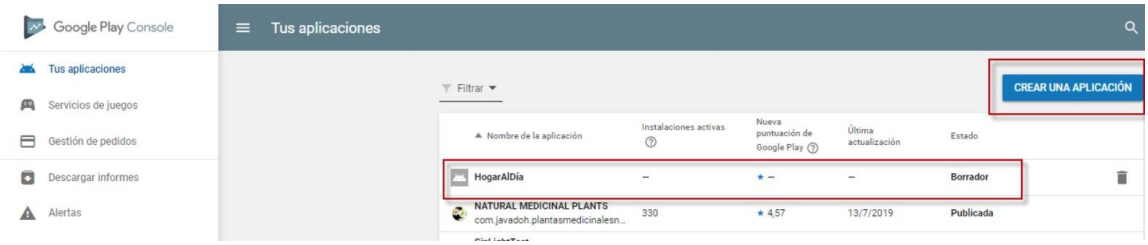


Imagen 26. Home - Crear App - Como se ve Home

App-PanelControl

Al ingresar a alguna de tus aplicaciones la primera opción de tu menú será “Panel de Control” donde podrás acceder a estadísticas relevantes y acciones asociadas de la misma relevancia, como lo son los nuevos usuarios adquiridos, los perdidos, ingresos, errores, puntuación media de la app (valoración), detalles por países, etc...

Imagen 27. Home / Panel de Control / Resultados

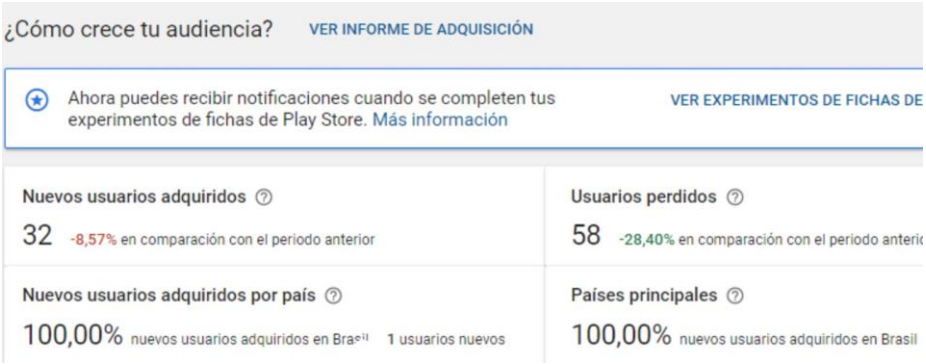


Imagen 28. Home / Panel de Control / Audiencia

Home-App-Estadísticas

En esta sección obtendrás un mayor número de gráficos y detalles de todo tipo referido al comportamiento de tu aplicación a lo largo del tiempo.

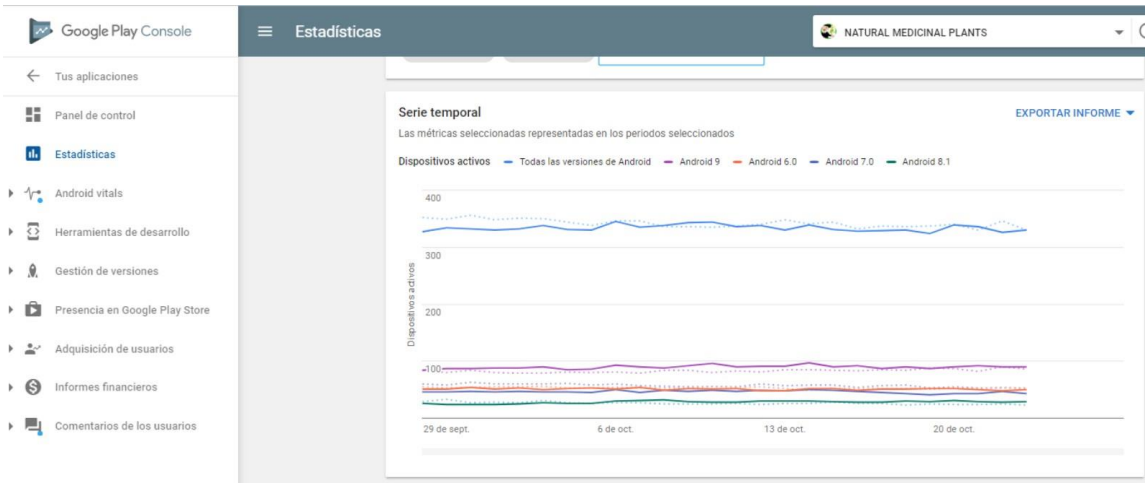


Imagen 29. Home / Estadísticas

Home-App-AndroidVitals

Esta sección informa sobre el estado de “salud” de la aplicación, en cuanto a errores, bloqueos, tamaño de la aplicación, archivos de compresión.

Los errores presentados por los usuarios de tu aplicación de distintos niveles, destacando los errores graves de tipo ANR (Application Not Responding), que indica un error donde la aplicación se ha quedado detenida o congelada sin poder el usuario interactuar; y también nos muestra información sobre los errores de bloqueo que impiden acceder a una funcionalidad específica de nuestra app.

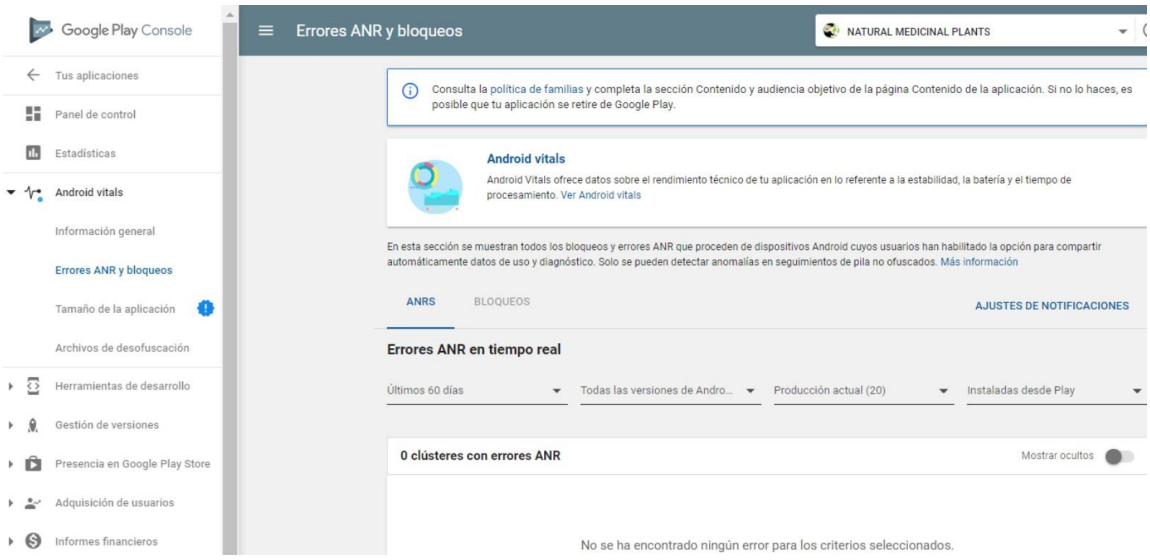


Imagen 30. Home - Android Vitals - Errores ANR

En ocasiones los errores suelen aparecer sólo en algunos modelos de smartphones o versiones de sistema operativo android, razón que hace importante probar con detalle y con tiempo entre distintas versiones y modelos, esto para considerarse siempre en los cálculos de tiempo a la hora de planificar proyectos.

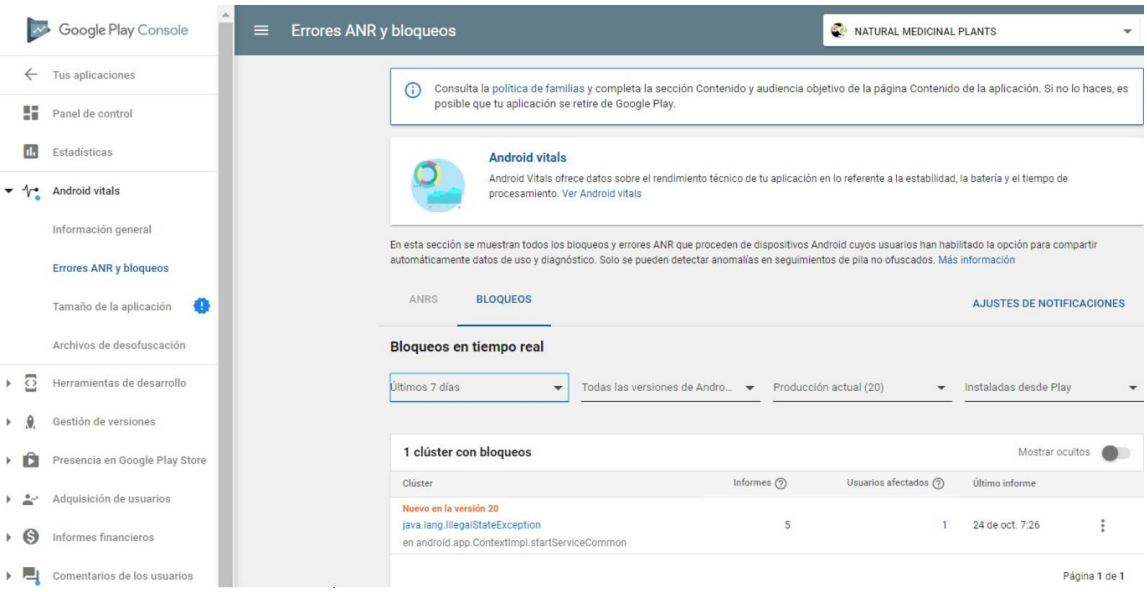


Imagen 31. Home - Android Vitals - Bloqueos

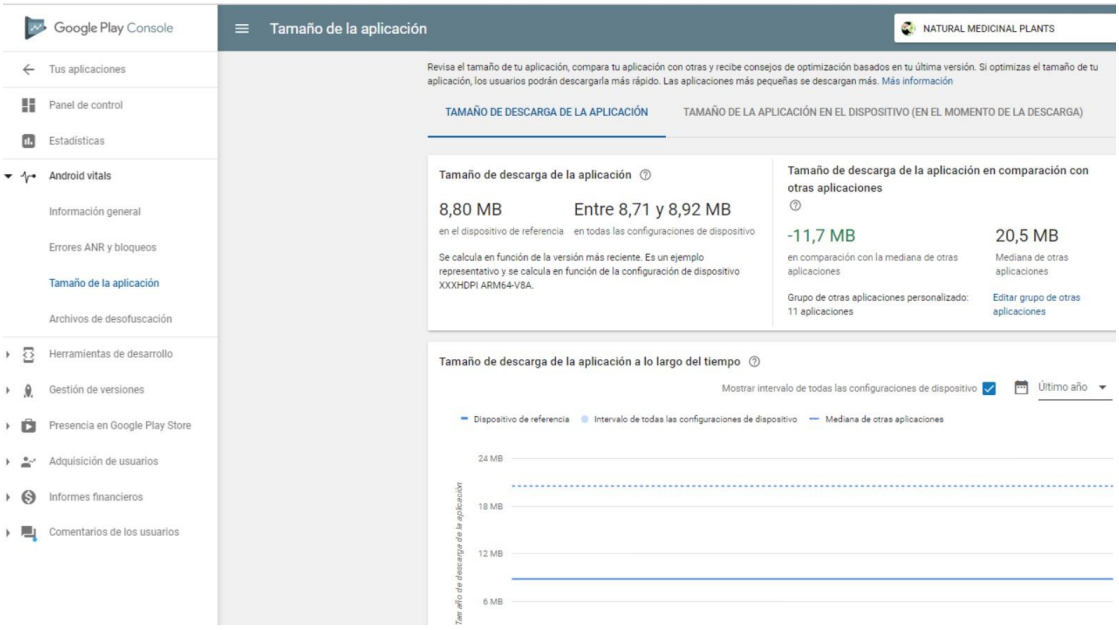


Imagen 32. Home - Android Vitals - Tamaño App

Home-App-Herramientas Desarrollo

Esta sección facilita la administración de equipos que trabajan sobre una misma app, donde los super usuarios, podrán gestionar equipos de pruebas, roles y certificados necesarios para el correcto funcionamiento de la app en la tienda.

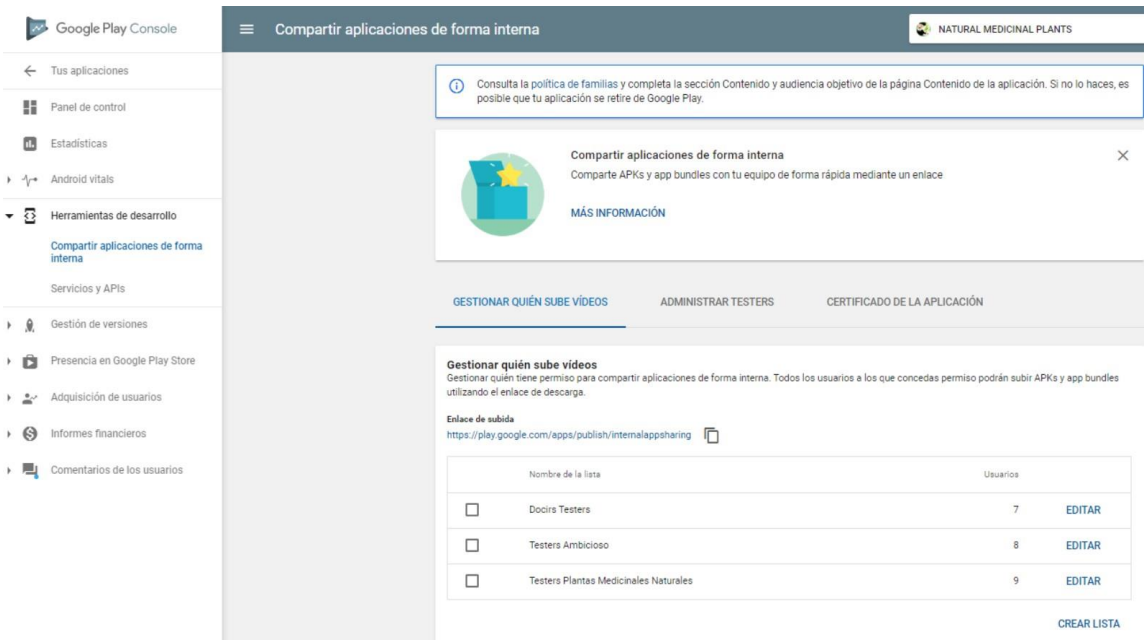


Imagen 33. Home - Herramientas de Desarrollo / Compartir

Los servicios y Apis ofrecidos en esta sección son propios de Google y conllevan a la creación de conexiones entre la aplicación y páginas web de tu dominio, además de tener la posibilidad de adquirir servicios potentes para algunos tipos de apps, como por ejemplo lo son los juegos, donde google te facilita servicios para minimizar la complejidad de creación de este tipo de sistemas.

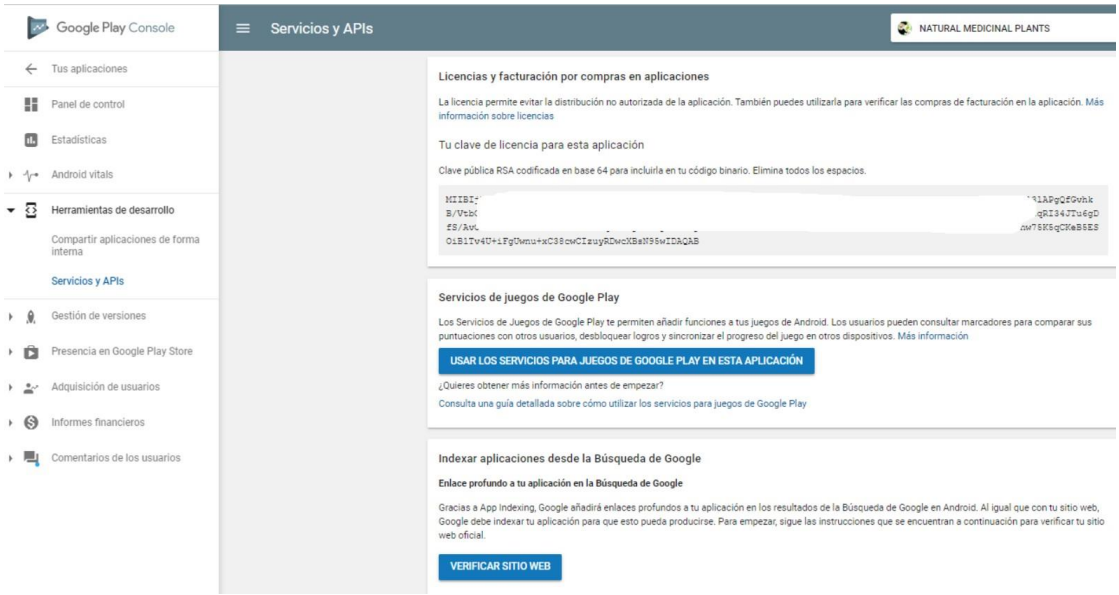


Imagen 34. Home - Herramientas de Desarrollo / Servicios y Apis

Home-App-Gestión de Versiones

Esta sección es de las más concurridas ya que en esta es donde se administran las subidas de las nuevas versiones de tu aplicación, y se traspasan entre diferentes ambientes, además de configurarlos. La siguiente imagen es del panel principal de la gestión de versiones, donde se muestra la información relevante.

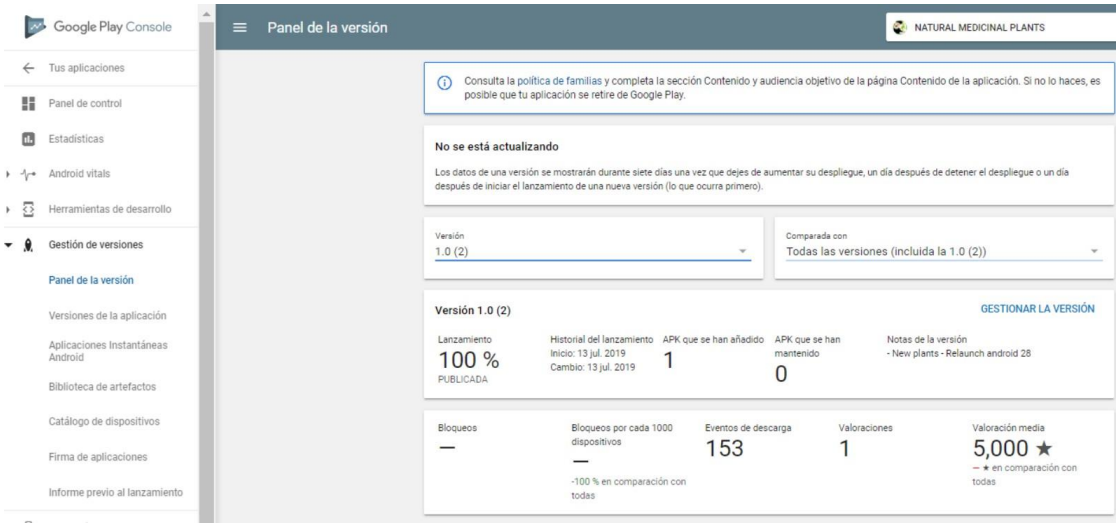


Imagen 35. Panel principal de la gestión de versiones.

Versiones de la Aplicación: esta sección subcategoría es donde podrás gestionar los distintos ambientes disponibles en la google play console, como lo son, ambiente de pruebas internas, beta, alpha y producción.

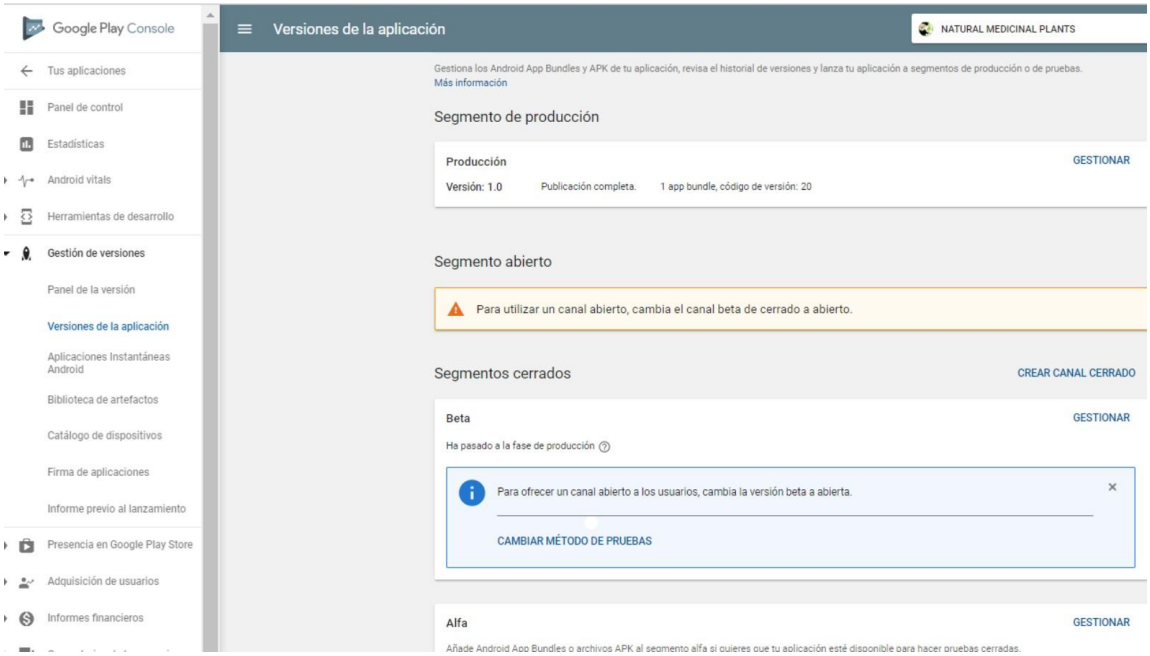


Imagen 36. Home - App - Gestión de versiones / Versiones de la App

Se observa el último número de versión y en qué ambiente se encuentra instalado, además de los grupos de prueba, entre otras funcionalidades.

- **Firma de aplicaciones**: es muy importante dentro de esta sección de Gestión de Aplicaciones, ya que hoy en día, se aconseja para crear App Bundles (las apk modernas) que sugiere android para optimizar y disminuir el tamaño de tus apps.

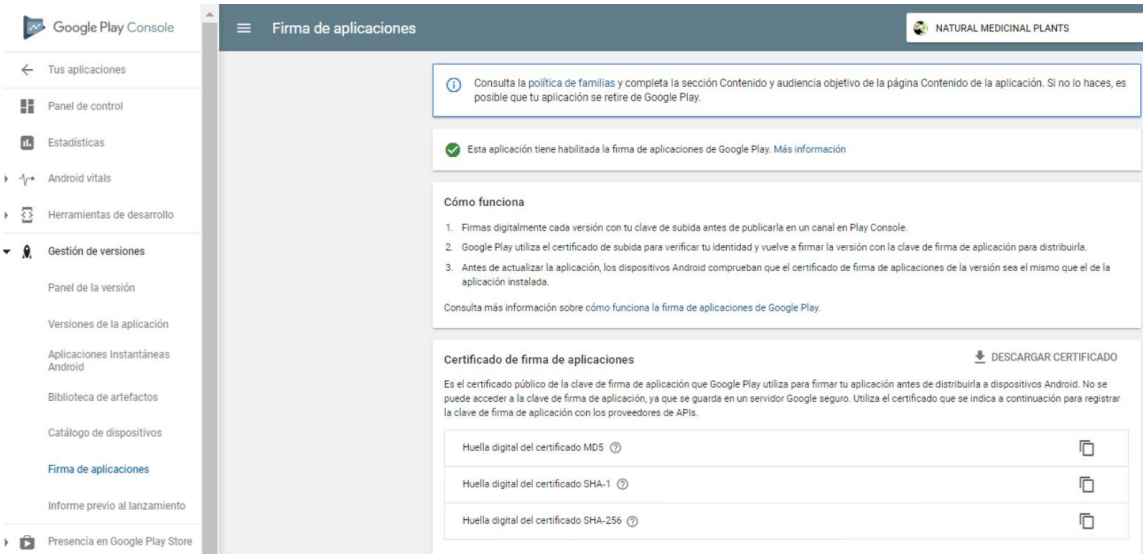


Imagen 37. Home - App - Gestión de Aplicaciones / Firma de Apps

Informe Previo al Lanzamiento: esta subcategoría te indica un historial de informes que se crean antes durante la intención de subida de una app, donde se señalan los errores, advertencias, problemas menores, las pruebas automáticas completas y fechas de entrega, todo esto clasificado por número de versión, es decir, cada vez que subes algún cambio.

Imagen 38. Home - App - gestión de Aplicaciones / Informe Previo al Lanzamiento

Home-App-Presencia en la PlayStore

Ficha de la PlayStore: esta sección permite la parametrización del título de la aplicación, descripción, tags, meta-data, subida de imágenes como la del icono, pantallazos de la aplicación, videos promocionales, banners promocionales, todo aquello que vemos de una aplicación en la google play antes de descargarla, además de tener una plantilla por cada idioma de nuestra aplicación.

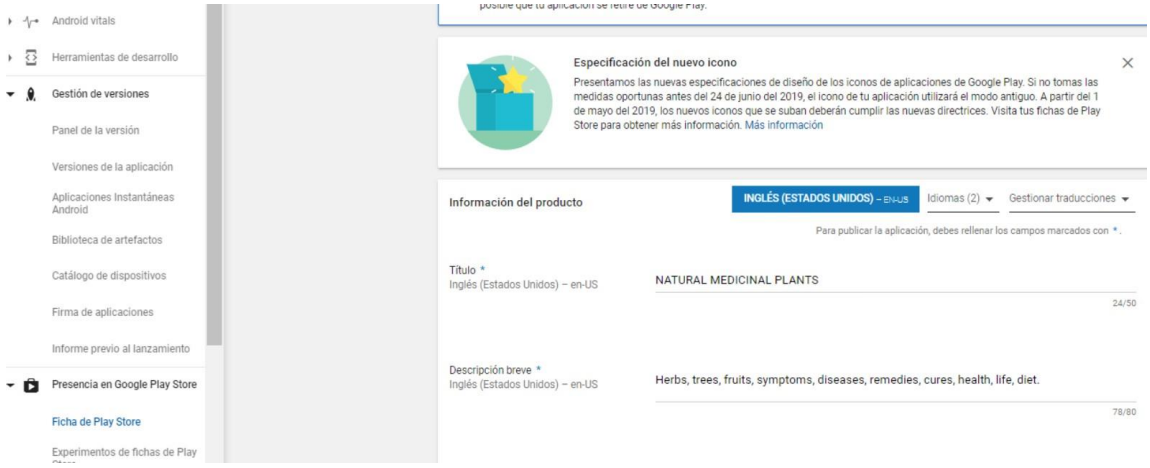


Imagen 39. Home - App - Presencia en la Play Store / Ficha

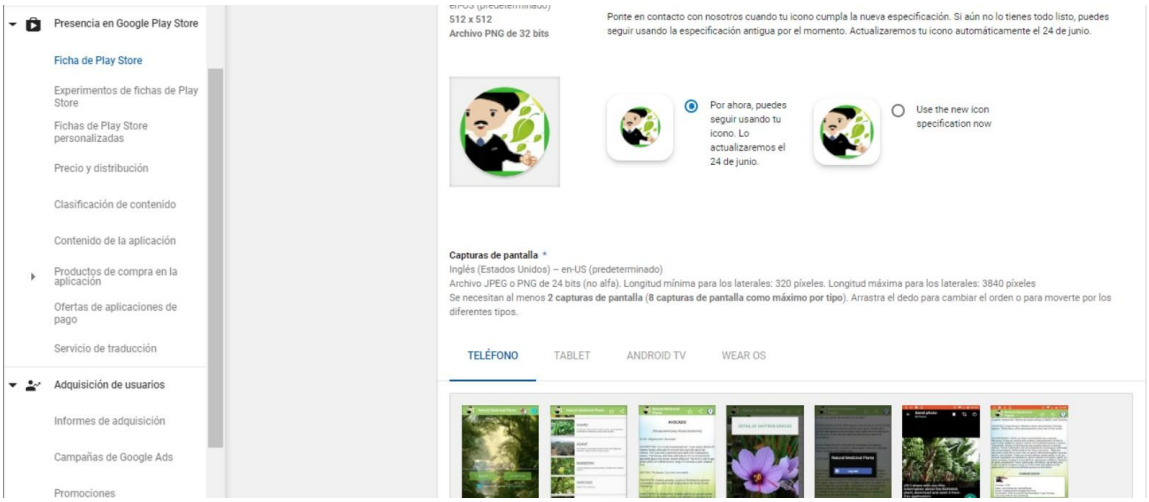


Imagen 40. Home - App - Presencia en la Play Store / Ficha Imágenes

Precio y distribución: esta subcategoría permite configurar si tu aplicación será de descarga gratuita o paga, en qué países estará disponible, y tipo de clasificación, entre otras funcionalidades.

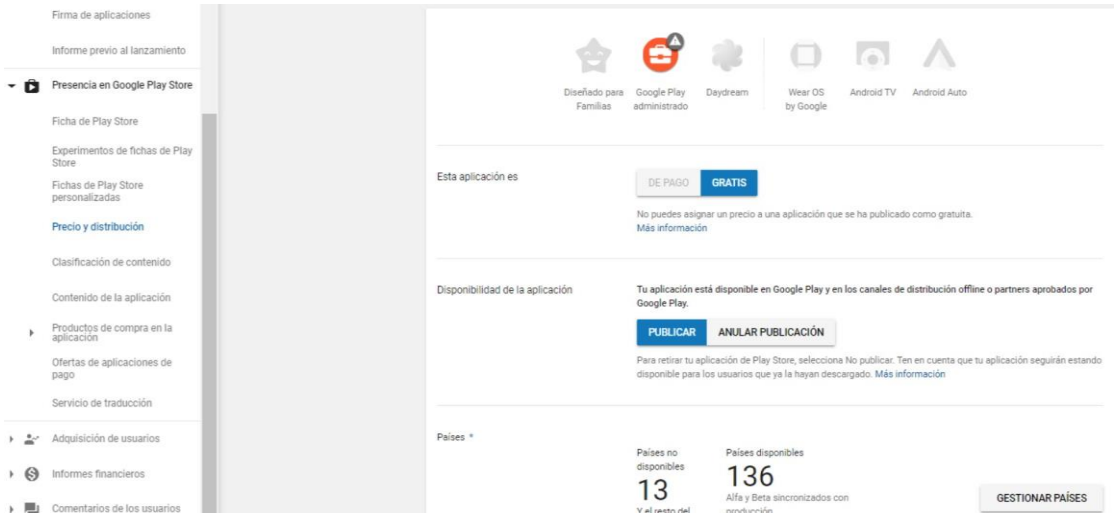


Imagen 41. Home - App - Presencia en la Play Store / Precios y Distribución

Clasificación del contenido: es importante declarar que tipo de contenido posee tu aplicación en esta subcategoría, ya que google estudia tu declaración y según el resultado del estudio la permite o la anula. Referido a tipo de edad, tipo de contenido, lenguaje, contenido sexual entre otros...

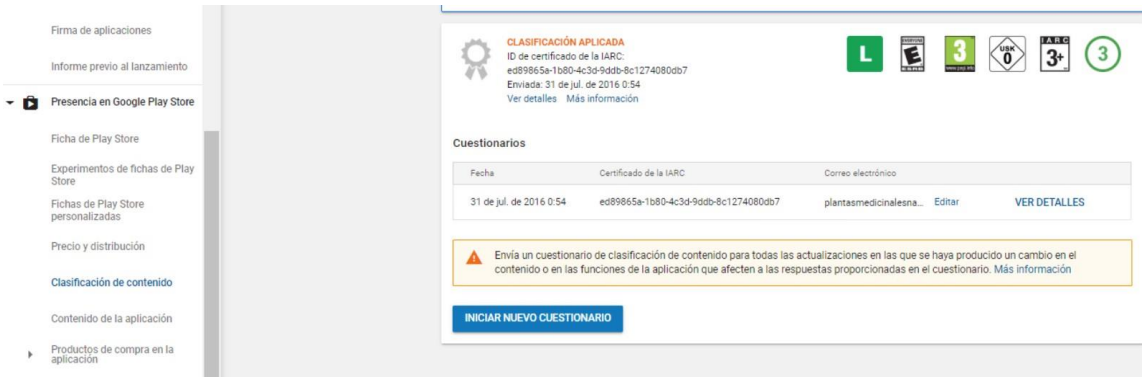


Imagen 42. Home - App - Presencia en la Play Store, / Clasificación contenido

Home-App-Adquisición de Usuarios

Esta sección muestra las campañas publicitarias creadas con Adwords, cómo van evolucionando, informes y promociones que has negociado con google.

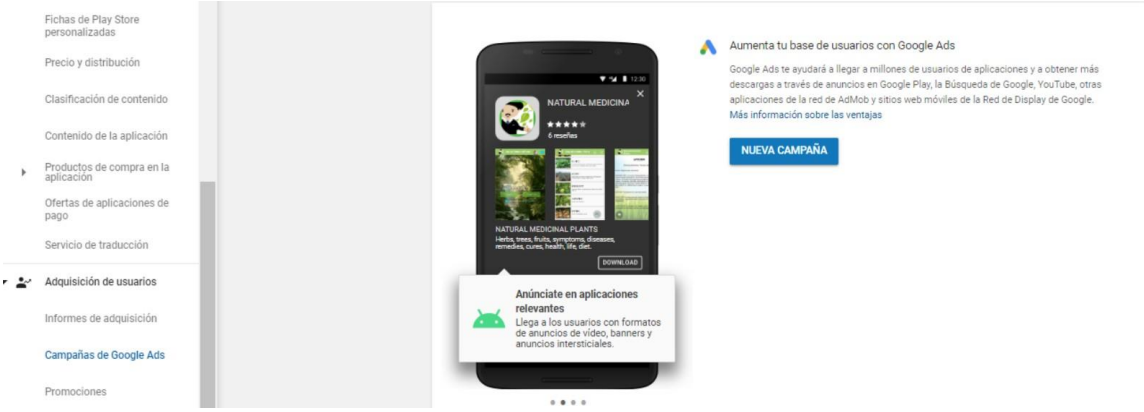


Imagen 43. App - Adquisición de Usuarios - Campañas

Home-App-Comentarios de los Usuarios

Esta sección informa sobre las valoraciones o puntuaciones adquiridas por tu app en la google play y los comentarios de tus usuarios.

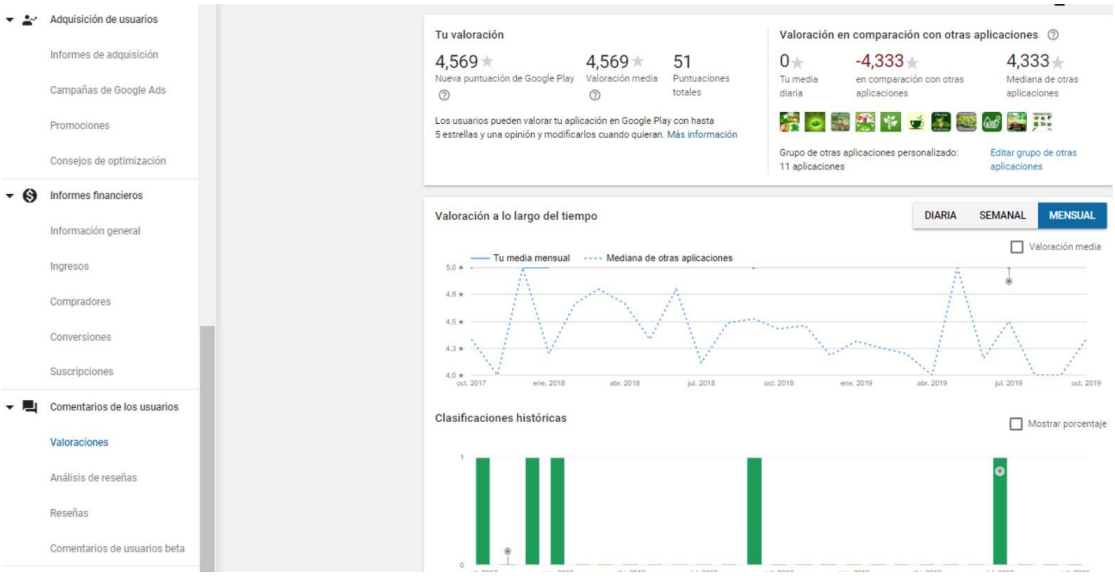


Imagen 44. Home - App - Comentarios de los Usuarios - Valoraciones

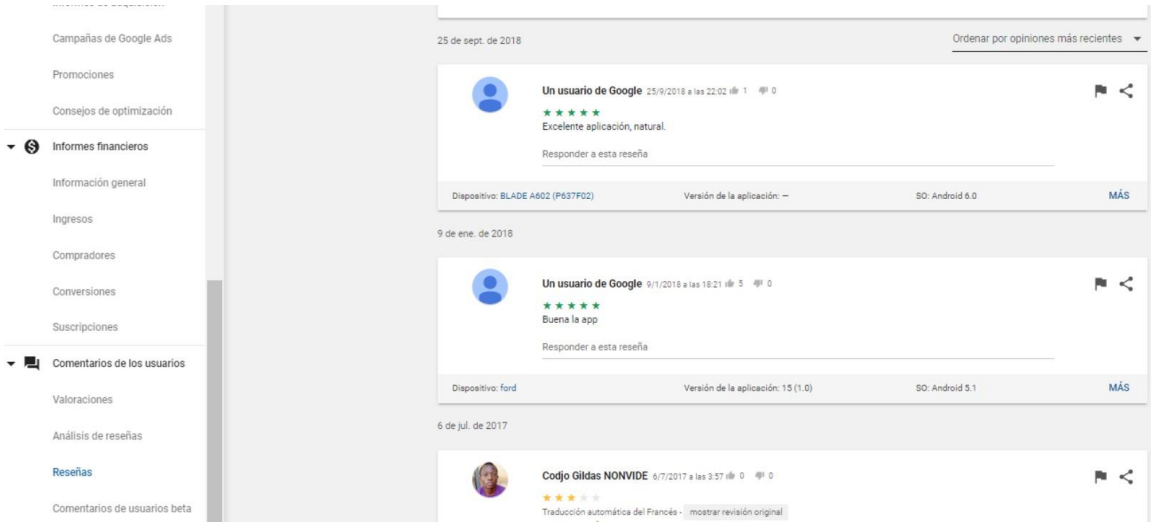


Imagen 45. Home - App - Comentarios de los Usuarios - Reseñas

Pruebas de Desarrollo en la Consola

- **Prueba interna**: te permite distribuir tu app de forma rápida para realizar pruebas internas y verificaciones de control de calidad.
- **Pruebas Abiertas**: te permiten realizar una verificación con un grupo más amplio y publicar la versión de prueba de tu app en Google Play. Si realizas una prueba abierta, cualquier usuario podrá unirse al programa y enviarte comentarios privados. Antes de elegir esta opción, asegúrate de que tu app y su ficha de Play Store estén listas para tener visibilidad en Google Play.

  Nota: Ya no es posible crear pruebas Alfa abiertas ni Beta cerradas. Seguirás teniendo acceso a aquellas que ya estén activas.
- **Pruebas Cerradas:** Te permite realizar pruebas de las versiones previas al lanzamiento de tu app con un grupo de verificadores más extenso. Cuando hayas realizado pruebas con un grupo reducido de empleados o usuarios de confianza, podrás expandir la prueba a una versión abierta. En la página Versiones de la app, encontrarás un segmento Alfa disponible como prueba cerrada inicial. Si es necesario, también puedes crear segmentos cerrados adicionales y colocarles nombres. Si quieres probar una app que ya publicaste anteriormente, solo los usuarios del grupo de prueba recibirán una actualización de la versión cerrada.

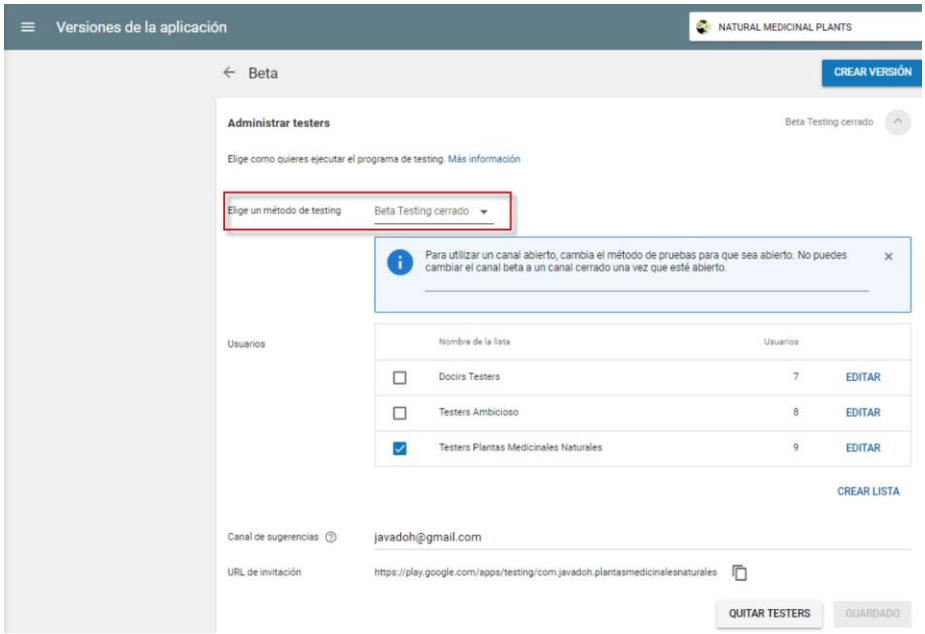


Imagen 46. Pruebas cerradas en ambiente Beta de Google Play Console.

Imagen 47. Pruebas usuarios Lista de participantes Gestión de Ambientes.

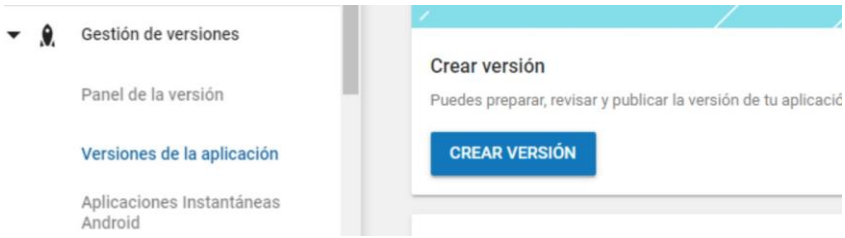


Imagen 48. Crear versión.

Hay cuatro tipos de ambientes de lanzamiento que admite la Consola de Google Play:

- **Pruebas Internas**: este ambiente es relativamente nuevo y se utiliza principalmente para que los desarrolles prueben incluso durante la construcción de la aplicación. Este ambiente pretende facilitar la colaboración entre desarrolladores.
- **Alfa**: en alfa debemos crear una versión para realizar rápidamente las primeras pruebas de la aplicación. Este canal le permite distribuir una acción a un pequeño conjunto de usuarios sin pasar por una revisión completa de Google. Los usuarios recibirán advertencias de que la acción no se ha sometido a una revisión de Google para el cumplimiento de la política. Una vez que hayamos probado con un grupo más pequeño de usuarios de confianza, podemos ampliar las pruebas a una versión beta cerrada.

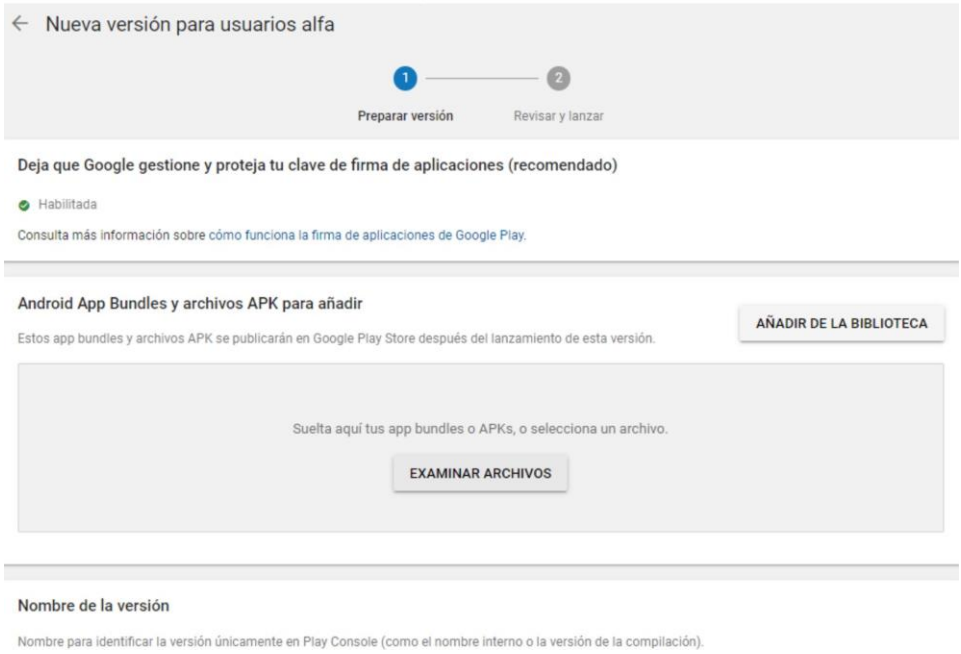


Imagen 49. Subida de archivo apk o bundle a ambiente alpha.

- **Beta**: este canal o ambiente, permite distribuir una acción a un conjunto limitado de usuarios después de pasar una revisión completa de Google. Debe usar este canal para dar acceso a los usuarios fuera de su organización a las primeras versiones de la acción. Se informará a los usuarios que la acción es una versión preliminar. Dado que la versión beta cerrada ha pasado la revisión de Google, puede optar por hacerla pública en cualquier momento sin otra revisión por parte de Google.
- **Producción**: este es el ambiente definitivo que disponibiliza tu aplicación para todo tipo de usuarios, en grupos, pueblos, ciudades y países. Para realizar el lanzamiento en producción es necesario que pasemos por todos los ambientes de prueba para asegurarnos de que los errores e inconvenientes hayan sido minimizados, asegurando la supervivencia de nuestra aplicación en la tienda de google play.

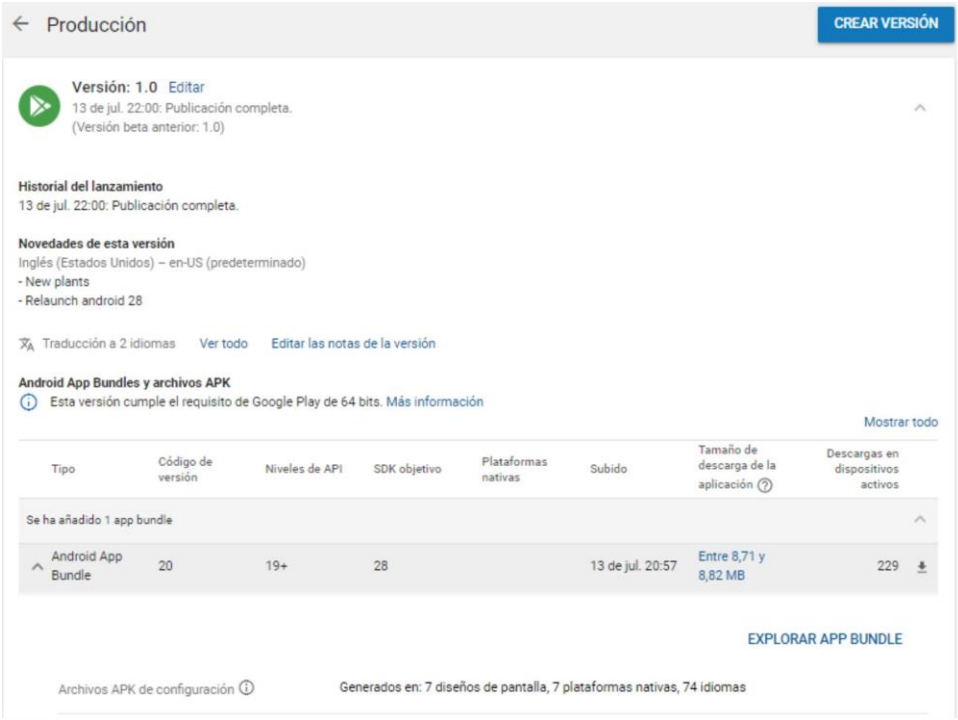


Imagen 50. Aplicación en producción desde gestión de ambientes.

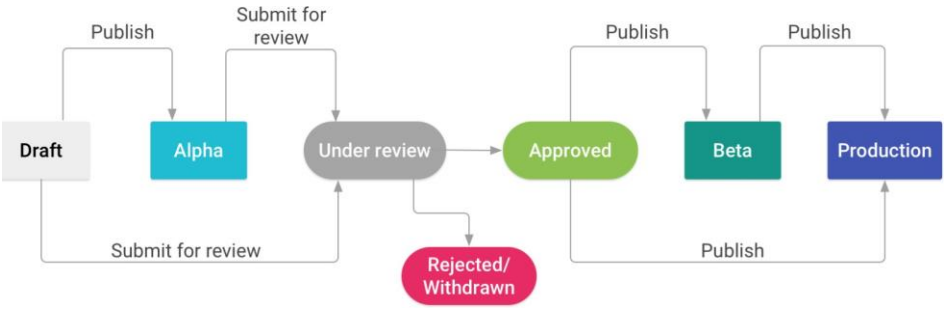


Imagen 51. Diagra,a - Ciclo de vida de una acción de instalación de aplicación en entornos o ambientes de lñanzamiento alpha y beta.

4.2 Crash tracker

Analítica usando Crashlytics

Competencias

- Agregar Crashlytics a un proyecto Firebase.
- Habilitar Google Analytics en un proyecto Firebase.
- Verificar el funcionamiento forzando una falla.
- Conocer el panel de Crashlytics.
- Inicializar Crashlytics en un proyecto.
- Desactivar Crashlytics en modo debug.

Introducción

La estabilidad en una app es un factor relevante a la hora de conquistar a los usuarios. Las apps con bugs hacen usuarios infelices que pueden desde dar una mal review hasta desinstalar la app. Cuando un usuario descontento desinstala una app, es muy complicado que la pueda volver a instalar.

Dependiendo de la app y la cantidad de usuarios los *bugs* pueden convertirse en un dolor de cabeza, ocupando tiempo, personas y recursos en revisar, priorizar y solucionar los errores.

Firebase Crashlytics ayuda en la automatización de estas tareas recolectando y organizando los errores ocurridos, con una interfaz que ayuda a priorizar y que permite disminuir la cantidad de errores, para mantener a los usuarios contentos.

Anteriormente, Crashlytics era una herramienta autónoma que pertenecía a Fabric.io y ha sido recientemente (2019) integrada a Google Firebase. El presente capítulo es sobre **Firebase Crashlytics**.

Requisitos

Para instalar Firebase Crashlytics se debe tener como requisito una cuenta para usar la consola de Firebase y habilitar Google Analytics para el proyecto

Crear un proyecto con Firebase

Primero se debe tener un proyecto Firebase. Para crearlo se accede a la página oficial de la consola de Firebase

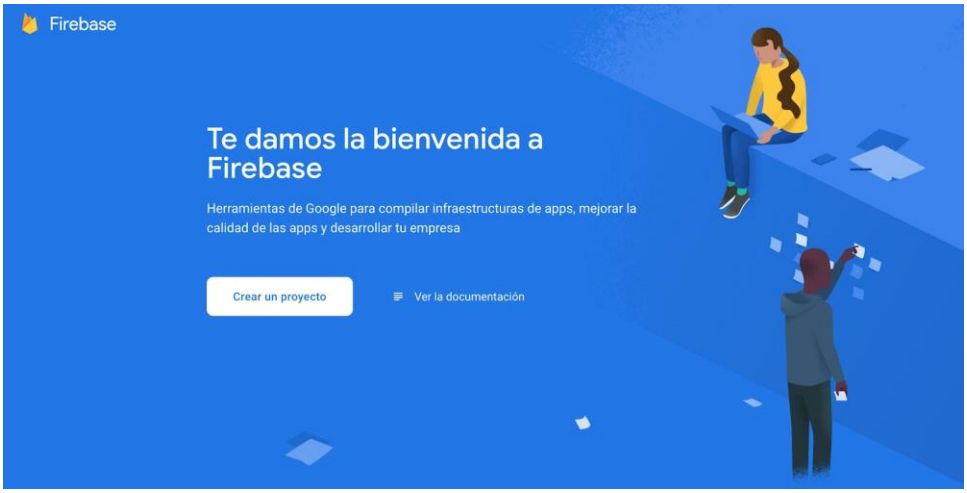


Imagen 52. Crear proyecto con Firebase.

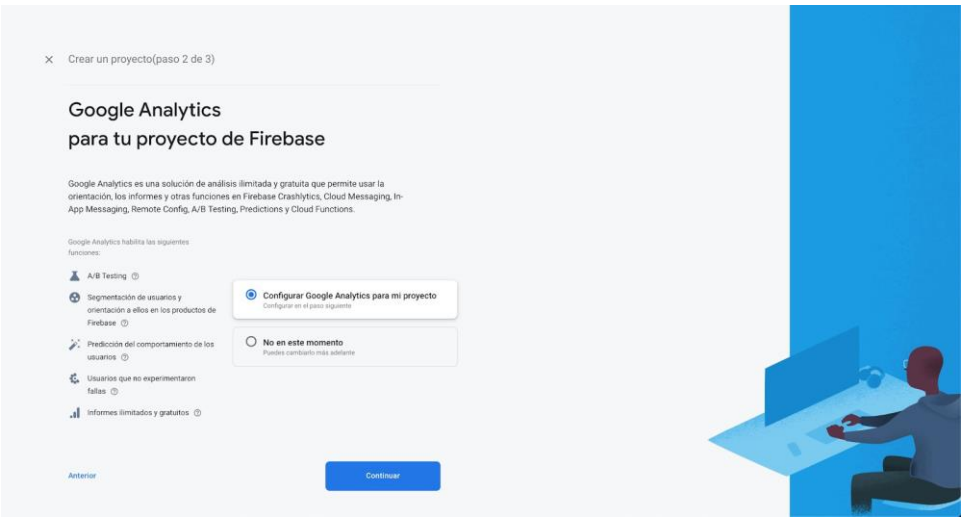


Imagen 53. Asignar nombre de proyecto.

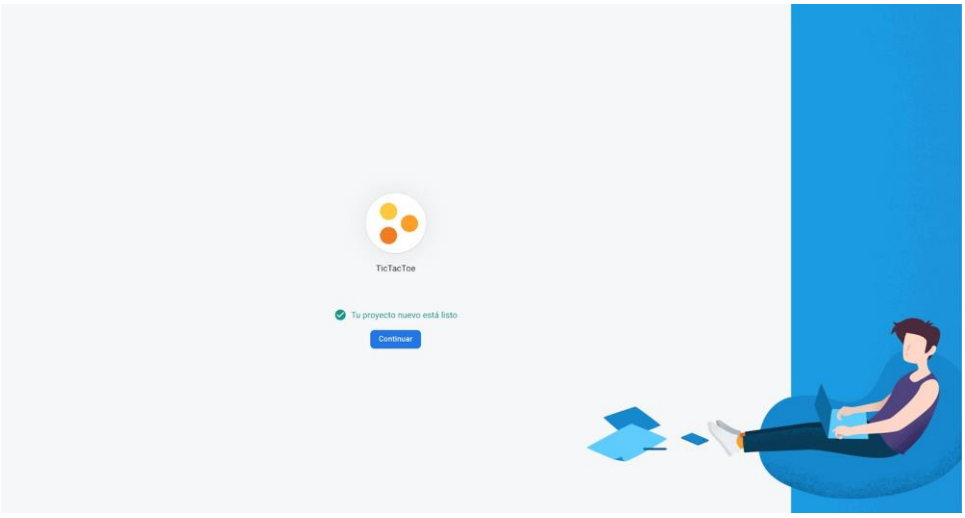


Imagen 54. Proyecto creado listo.

Para luego ser redirigido al home de Firebase.

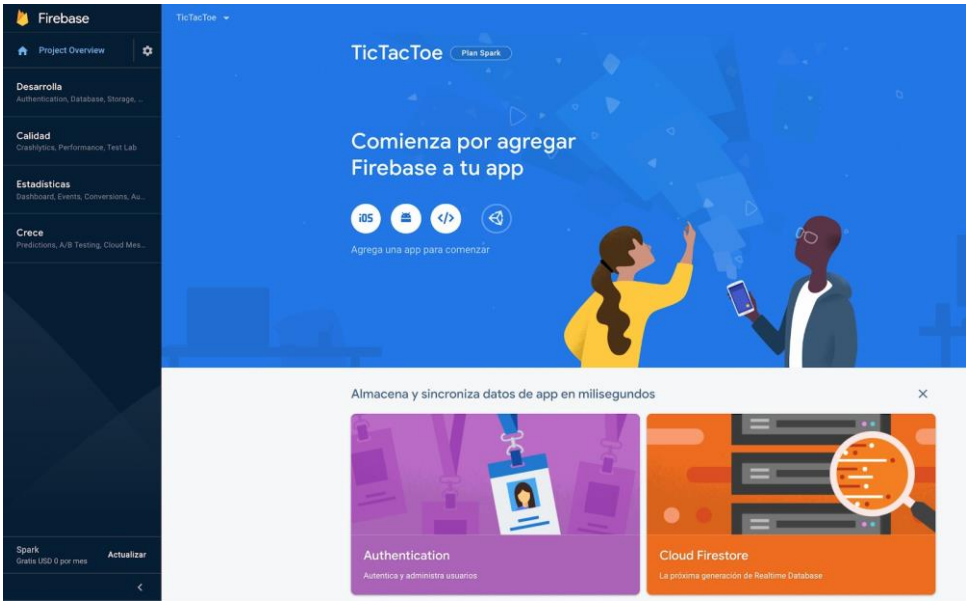


Imagen 55. Dashboard de Firebase.

Habilitar Google Analytics

Google Analytics también ofrece información importante al desarrollador, pero con un propósito de verificar la cantidad e instalaciones, información de que pantalla pasan más los usuarios, entre otros. Es una información complementaria a la que podemos tener usando Crashlytics

Para poder utilizar Crashlytics es necesario configurar Google Analytics en la consola de Firebase. Para esto, desde el menú izquierdo, en el apartado Calidad > Crashlytics se agrega un nuevo proyecto.

Imagen 56. Menú.

La configuración de Google Analytics pide aceptar condiciones para protección de datos, mediciones de datos en general y las condiciones de Google Analytics. Una vez confirmado que se está de acuerdo con los documentos, la habilitación es bastante simple

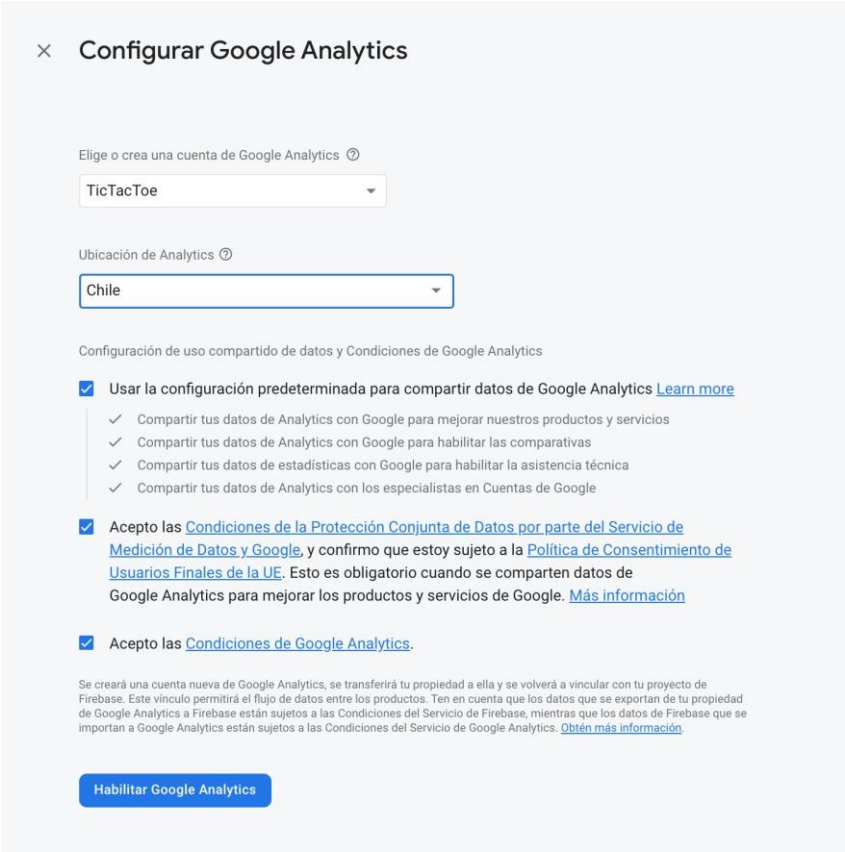


Imagen 57. Configuración de Google Analytics.

Configurar Crashlytics

Desde el home de firebase es accesible Crashlytics para su configuración

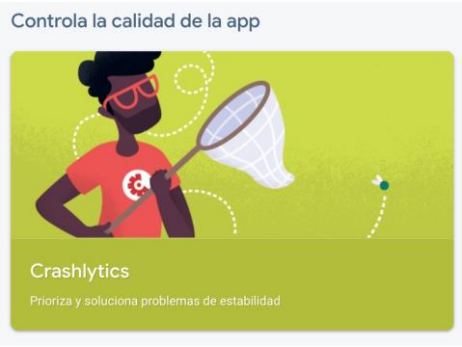


Imagen 58. Panel de Crashlytics.

Al seleccionar la opción se debe iniciar la configuración de Crashlytics

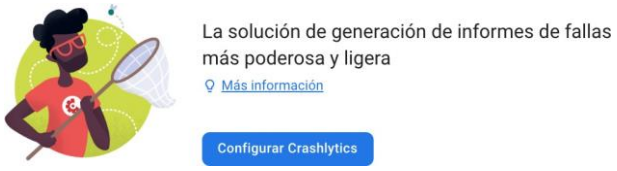


Imagen 59. Configuración de Crashlytics.

A continuación se debe ejecutar 3 pasos

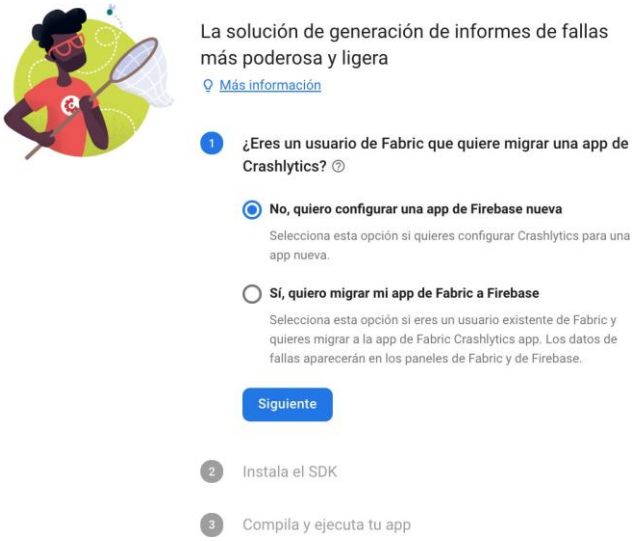


Imagen 60. Pasos de la configuración de Crashlytics.

Dado que es una herramienta que está siendo migrada actualmente hacia Firebase, se puede elegir crear una nueva configuración o migrar una app

Instalar SDK de Crashlytics

Se debe modificar a el script build.gradle a nivel de proyecto para agregar el plugin de Crashlytics

```
buildscript {
    repositories {
        // ...
        // Add the following repositories:
        maven {
            url 'https://maven.fabric.io/public'
        }
    }

    dependencies {
        // ...

        // Check for v3.1.2 or higher
        classpath 'com.google.gms:google-services:4.3.2'  // Google Services plugin

        // Add dependency
        classpath 'io.fabric.tools:gradle:1.31.0'  // Crashlytics plugin


    }
}
```

Además, a nivel de módulo se debe aplicar el plugin y agregar la dependencia a Crashlytics

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

dependencies {
    // ...

    // (Recommended) Add Analytics
    implementation 'com.google.firebase:firebase-analytics:17.2.0'

    // Add dependency
    implementation 'com.crashlytics.sdk.android:crashlytics:2.10.1'
}
```

Después de sincronizar el proyecto queda listo para compilar y ejecutar

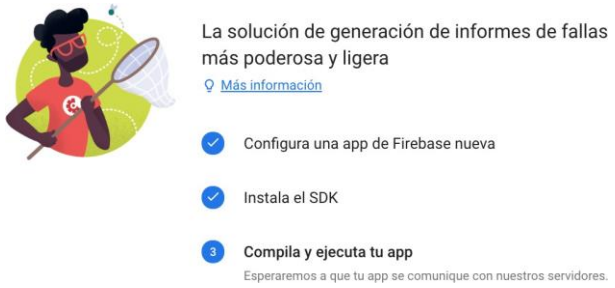


Imagen 61. Proyecto sincronizado listo para compilar y ejecutar.

Con este paso se completa la inclusión y configuración de Crashlytics en la app. Para finalizar el proceso, basta con desinstalar la app y volver a instalarla con estas modificaciones para que se comunique con el servidor y quede habilitado Firebase Crashlytics

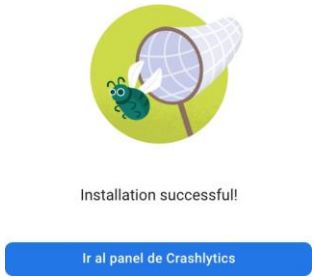


Imagen 62. Habilitación de Firebase Crashlytics.

Panel de Crashlytics

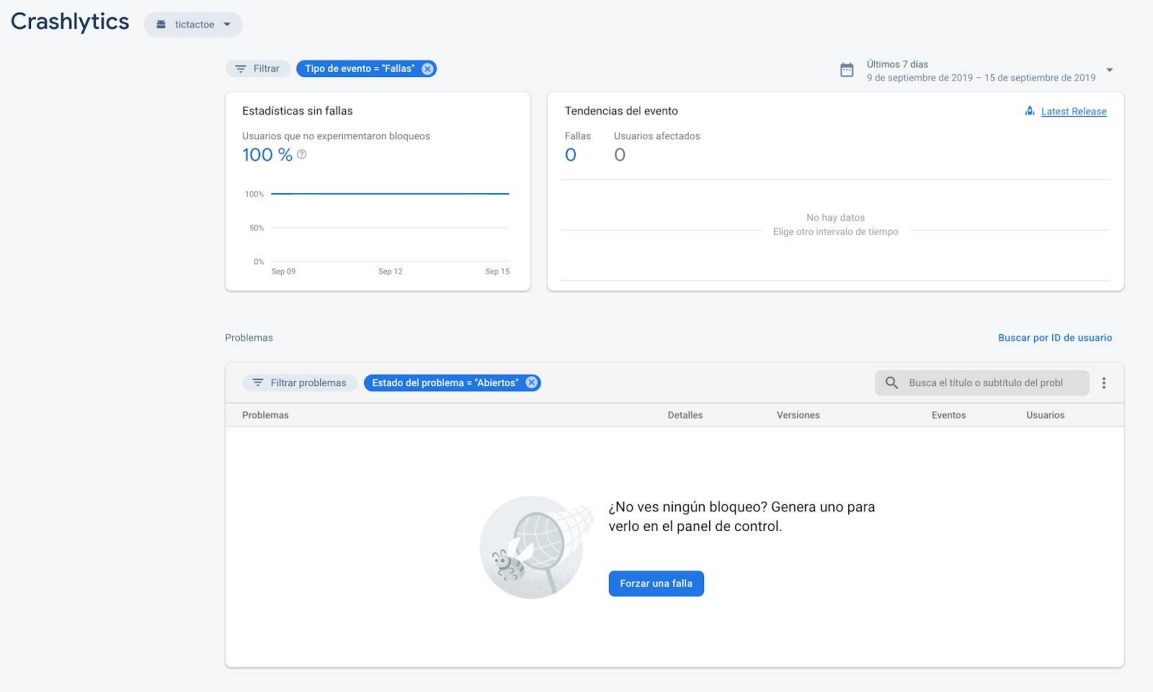


Imagen 63. Paneld e Crashlytics.

El panel de Crashlytics está distribuido en snippet para visualizar las estadísticas de falla y los problemas que se han generado en la app

Forzar una falla

Para forzar una falla de prueba, basta con agregar la siguiente línea

```
Crashlytics.getInstance().crash();
```

Lo que genera un excepción de prueba que se refleja en el panel de Crashlytics

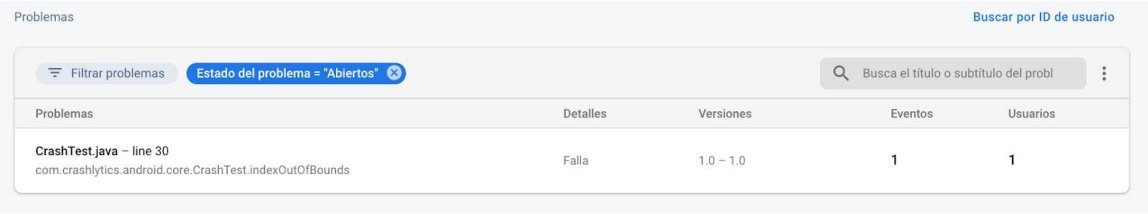


Imagen 64. Excepción de prueba.

Los problemas ocurridos en la app son listados y se pueden analizar en detalle entrando a cada uno si se selecciona de la lista

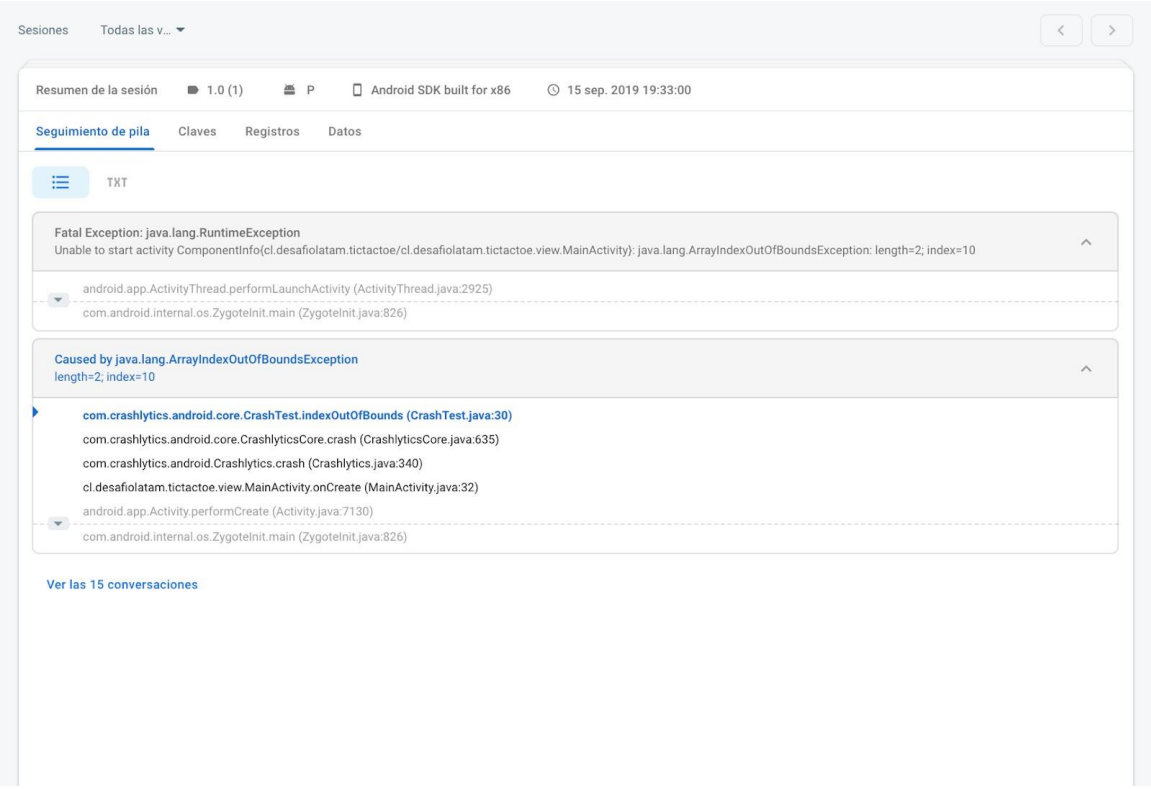


Imagen 65. Listado de problemas de la app.

El detalle incluye dónde fue la excepción y los datos asociados a su causa de forma ordenada y permite ver en detalle el error

Inicializar Crashlytics

La inicialización de Crashlytics debe hacerse al comienzo de la ejecución de la app de modo que quede disponible lo antes posible para recopilar los problemasSe realiza llamando de la siguiente forma al *builder* de Crashlytics

```
private void setupCrashlytics() {
    Crashlytics crashlyticsKit = new Crashlytics.Builder()
            .core(new CrashlyticsCore.Builder().build())
            .build();

    Fabric.with(this, crashlyticsKit);
}
```

Personalizar los reportes

Para entregar más información dentro de los reportes, Crashlytics entrega 4 mecanismos de logging custom keys, custom logs, identificación de usuarios, y manejo de excepciones no fatales.

Claves personalizadas (custom keys)

Asocian pares clave/valor arbitrarios que aportan estados específicos de la app cuando ocurre un problema. Existen 5 métodos para asignar el valor a una clave

```
Crashlytics.setString(key, "foo" /* string value */);

Crashlytics.setBool(key, true /* boolean value */);

Crashlytics.setDouble(key, 1.0 /* double value */);

Crashlytics.setFloat(key, 1.0f /* float value */);

Crashlytics.setInt(key, 1 /* int value */);
```

Se pueden entregar hasta 64 pares que entreguen información extra sobre el estado de la app, por ejemplo

```
Crashlytics.setInt("current_level", 3);
Crashlytics.setString("last_UI_action", "logged_in");
```

Logs personalizados (custom logs)

Para mejorar el contexto sobre los eventos que llevan a que la aplicación falle, se puede utilizar logs en la app que se muestran luego en la consola de Firebase.El método **Crashlytics.log(...)** permite agregar información al reporte de falla y también se puede usar para imprimir en el log de la app

- Para agregar al reporte e imprimir usando Log.println(...)

```
Crashlytics.log(Log.DEBUG, "tag", "message");
```

- Solo para el reporte

```
Crashlytics.log("message");
```

Identificación de usuarios

Para diagnosticar un problema, a menudo ayuda identificar al usuario que tuvo una falla determinada. Crashlytics incluye una forma de identificar a los usuarios en forma anónima dentro de los reportes. Para agregar estos identificadores, a cada usuario se le debe asignar un identificador numérico, un token u otro valor

```
Crashlytics.setUserIdentifier("user123456789");
```

Excepciones no fatales

Una excepción fatal hace que la aplicación deje de funcionar, sin embargo, no todas las excepciones son fatales pero aún así requieren de atención. Además de los reportes auto-generados, se pueden mostrar las excepciones no fatales. Para esto se agrega en el manejo de excepciones el método `Crashlytics.logException`

```
try {
    methodThatThrows();
} catch (Exception e) {
    Crashlytics.logException(e);
    // handle your exception here
}
```

Todas las excepciones aparecen en la consola de Firebase como non-fatal. El resumen del problema contiene la misma información que se tiene de los otros reportes incluyendo la versión de Android y la información del dispositivo

El procesamiento de las excepciones se hace en un hilo dedicado para minimizar el impacto en el rendimiento de la app. Por otro lado, para reducir el tráfico de red de los usuarios, las excepciones escritas se juntan y envían la próxima vez que la aplicación se inicia.

Desactivar Crashlytics en debug

Durante el desarrollo de la app no es necesario tener activo Crashlytics dado que los errores y excepciones se analizan con las herramientas entregadas por Android Studio y el SDK de Android, por lo que una buena práctica es deshabilitar la herramienta Para no ejecutar el informe de Crashlytics y acelerar la compilación de debug se debe inhabilitar de la siguiente maneraEn el script build.gradle del módulo se debe agregar la opción a **ext.enableCrashlytics = false**

```
android {
  // ...
  buildTypes {
     debug {
        ext.enableCrashlytics = false
     }

     release {
       ...
     }
  }
}
```

y la inicialización de Crashlytics, generalmente en la actividad principal, debe habilitar/desahabilitar la construcción usando la variable de configuración según el tipo usando *BuildConfig.DEBUG*

```
Crashlytics crashlyticsKit = new Crashlytics.Builder()
    .core(new  CrashlyticsCore.Builder().disabled(BuildConfig.DEBUG).build())
    .build();

 Fabric.with(this, crashlyticsKit);
```

Desactivar incremento en el ID de compilación

Si aún así se desea usar Crashlytics en las compilaciones de depuración, de todos modos se puede acelerar las compilaciones incrementales evitando que Crashlytics actualice recursos de la app con su ID de compilación único durante cada compilación. Para evitar que Crashlytics actualice todo el tiempo su ID de compilación, se debe agregar lo siguiente al script build.gradle del módulo

```
android {
    ...
    buildTypes {
      debug {
        ext.alwaysUpdateBuildId = false
      }
  }
```

Documentación de Firebase Crashlytics

El menú de la parte izquierda entrega algunas opciones extras. A continuación se revisan las más relevantes

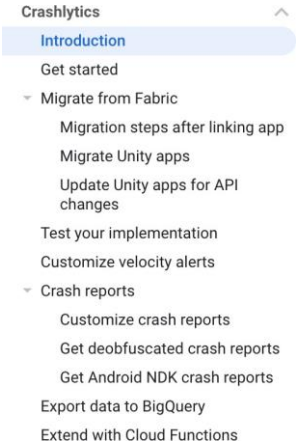


Imagen 66. Documentación de Firebase Crashlytics.

- **Export Crashlytics Data to BigQuery**: BigQuery es una herramienta de Google para almacenamiento rápido de datos y es de gran escalabilidad, además de proveer aprendizaje automático usando estos dato. Crashlytics permite una fácil integración con esta herramienta. Para ver cómo exportar los datos a BigQuery está la documentación oficial.
- **Customize velocity alerts**: Para Crashlytics, un issue es un conjunto de excepciones similares. Las velocity alerts son gatilladas cuándo un issue pasa el nivel mínimo definido calculado como un porcentaje de la sesiones.
- **Get deobfuscated crash reports**: Crashlytics en forma automática desofusca el código para tener reportes humanable legibles, aunque hay algunas configuraciones muy propias de los proyectos que hacen que falle.
- **Extend with Cloud Functions**: Se puede agregar una acción como respuesta a nuevos eventos de Crashlytics, como por ejemplo, notificar a un canal de slack, enviar correos o crear un caso en Jira.

Buenas prácticas

Criterios de retención de usuario

Existen algunos puntos comunes que afectan a la sensación del usuario cuando utiliza una app. Considerarlos ayuda a disminuir las desinstalaciones y a tener una mejor app

- Tamaño de la app: El espacio de almacenamiento es en muchos casos algo primordial. Mientras más espacio utilice la app más probable es que sea desinstalada
- Intermitencias de servicio: Cuando la app no tiene una comunicación estable con los servidores entrega una experiencia pobre y predispone la desinstalación de la app por parte de los usuarios
- Excesivos permisos: La declaración de los permisos requeridos por la app debe ser acorde a sus funcionalidades. Por ejemplo, una app para revisar los precios que pida permisos para ver las llamadas telefónicas hace que los usuarios eviten instalarla. El consejo es mantener sólo los permisos necesarios para el buen funcionamiento
- Excesiva publicidad: La publicidad es una forma de monetización extendida en el mundo de las app. Sin embargo, muchas veces tienden a ser demasiado invasivas y hacen que la interacción del usuario con la app sea poco fluída.
- Excesiva notificaciones: Las notificaciones *Firebase Cloud Messages* deben ser utilizadas para mostrar información importante y no abusar en la cantidad de mensajes enviados

Ficha de Play Store

Una ficha de Play Store atractiva atrae más usuarios. Tener un ícono original y llamativo, buenas capturas de pantalla y un video llamativo que incluya las funcionalidades principales entregan información visual que le sirva al usuario, además de acompañarlo de un título y descripción sugerente que incluya las palabras que los usuarios podrían buscar

Algunas prácticas recomendadas son:

- **Ícono**: Crea un ícono de selector original y relevante. Considera trabajar con un diseñador gráfico profesional o usar las herramientas de Android Asset Studio.
- **Gráfico de funciones**: Asegúrate de que el gráfico de funciones refleje qué tiene de especial tu app. Evita usar esta área para publicar capturas de pantalla adicionales o texto de promociones temporales. En cambio, aprovecha la oportunidad de reforzar la marca de tu app y mostrar lo que la hace genial.
- **Opiniones**: Responde a las opiniones para abordar inquietudes, descubrir problemas comunes y demostrarles a las personas que visitan la ficha de Play Store de tu app que prestas atención a sus comentarios.
- **Promoción de lanzamientos**: Crea una página personalizada para promover tu próximo lanzamiento y alentar a los usuarios para que se inscriban en el registro previo.

Compatibilidad con 64 bits

“A partir del 1 de agosto de 2019, las apps que publiques en Google Play tendrán que ser compatibles con arquitecturas de 64 bits. Las CPU de 64 bits brindan a los usuarios experiencias más enriquecidas y rápidas. Si agregas una versión de 64 bits a tu app, mejorarás su rendimiento, favorecerás la innovación y podrás ofrecerla para dispositivos que cuenten con hardware de 64 bits únicamente.”

Tamaño de pantalla

Diseñar tu aplicación para tablets puede ayudarte a llegar a más clientes en una mayor variedad de dispositivos. Las aplicaciones que se diseñan para tablets aparecen destacadas en la lista de aplicaciones diseñadas para tablets de Play Store, teniendo mayor visibilidad de cara a los usuarios.

Tiendas alternativas a Google Play Store

Para los usuarios de Android instalar aplicaciones es una tarea sencilla usando Google Play, pero no es la única alternativa. Existen otras tiendas de aplicaciones desde donde se puede descargar de manera legal millones de APK que por restricciones de la tienda oficial no pueden ser instalados en los dispositivos

- Amazon Appstore: Se estructura de manera similar que la tienda de Google y ofrece gratis una app de pago por día. Además, las apps quedas asociadas a la cuenta y pueden ser instaladas en los distintos dispositivos con la cuenta de Amazon configurada
- APPGallery: Es la tienda de aplicaciones de Huawei desde donde se pueden descargar juegos, apps y contenido exclusivo para los usuarios de APPGallery
- APKPure: Está disponible como una app al igual que Play Store y reúne una gran parte de aplicaciones gratuitas con control versiones e instalación directa
- UpToDown: Es un servicio popular para descargar aplicaciones para distintos sistemas operativos, entre los cuales, la categoría de Android tiene una amplia gama de aplicaciones. La tienda tiene una app que se instala en el dispositivo y tiene un funcionamiento similar a Google Play, informando de las actualizaciones disponibles de cada app.
- APKMirror: No es una tienda como tal, si no que un repositorio de APK donde se puede acceder en forma rápida y segura a las últimas actualización de las app. La idea es acceder a cualquier aplicación Android que no esté disponible en Play Store, ya sea por alguna restricción de los desarrolladores o por que no se ha subido aún a la tienda. El equipo de APKMirror revisa los APKs antes de subirlo, asegurando que están libres de virus.
- XDA Labs: Ofrece un repositorio de apps al igual que APKMirror. Tiene apps que han sido modificadas para ciertos equipos, con garantías en la seguridad de los usuarios y los teléfonos. Además, es parte del grupo de desarrolladores de la comunidad de Android (XDA-Developers) y es posible acceder a los foros directamente
- Malavida: Es una tienda de aplicaciones desarrollada en España, con una amplia gama de aplicaciones agrupadas en categorías. Los APKs son originales y revisados con antivirus para asegurar que son confiables.