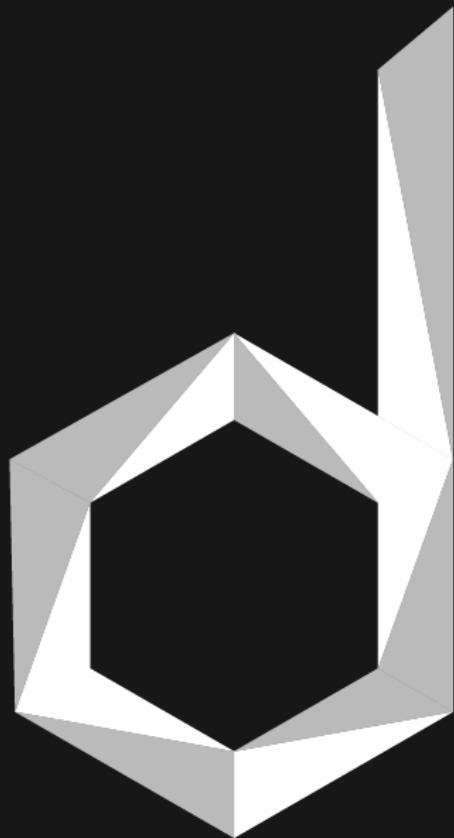


INGENIERÍA MECATRÓNICA



DI\_CERO

DIEGO CERVANTES RODRÍGUEZ

DESARROLLO MÓVIL - ANDROID

FIREBASE Y FIRESTORE

Bases de Datos:  
Firebase y Firestore

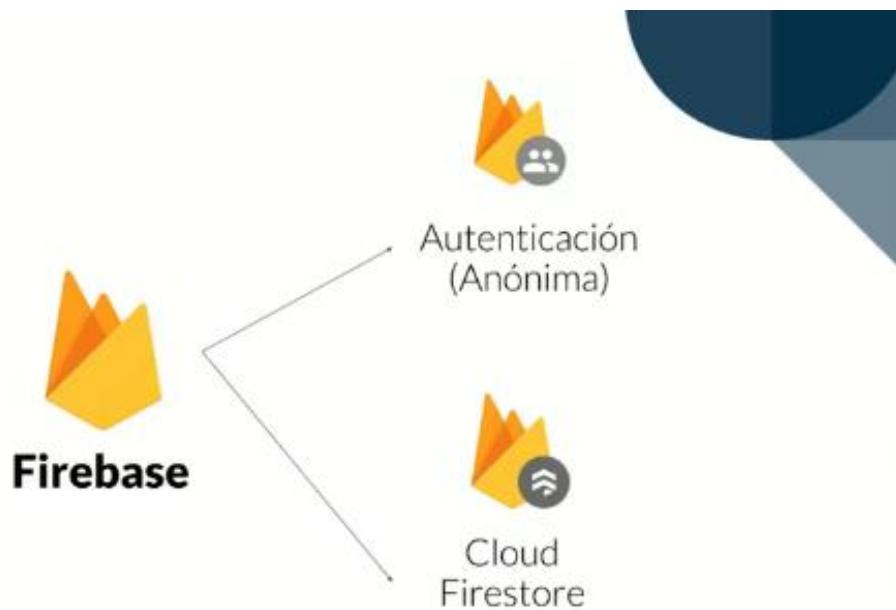
## Contenido

Kotlin - Firebase .....	2
Servicios de Firebase:.....	3
Configuración e instalación Cloud Firestore .....	4
Configuración del proyecto Android:.....	5
Creación de recursos: app/java/res .....	18
Creación de la base de datos en Firebase.....	19
Subir datos a la base de datos FireCloud:.....	21
Extracción de datos de la base de datos FireCloud: .....	25
Referencias: .....	33

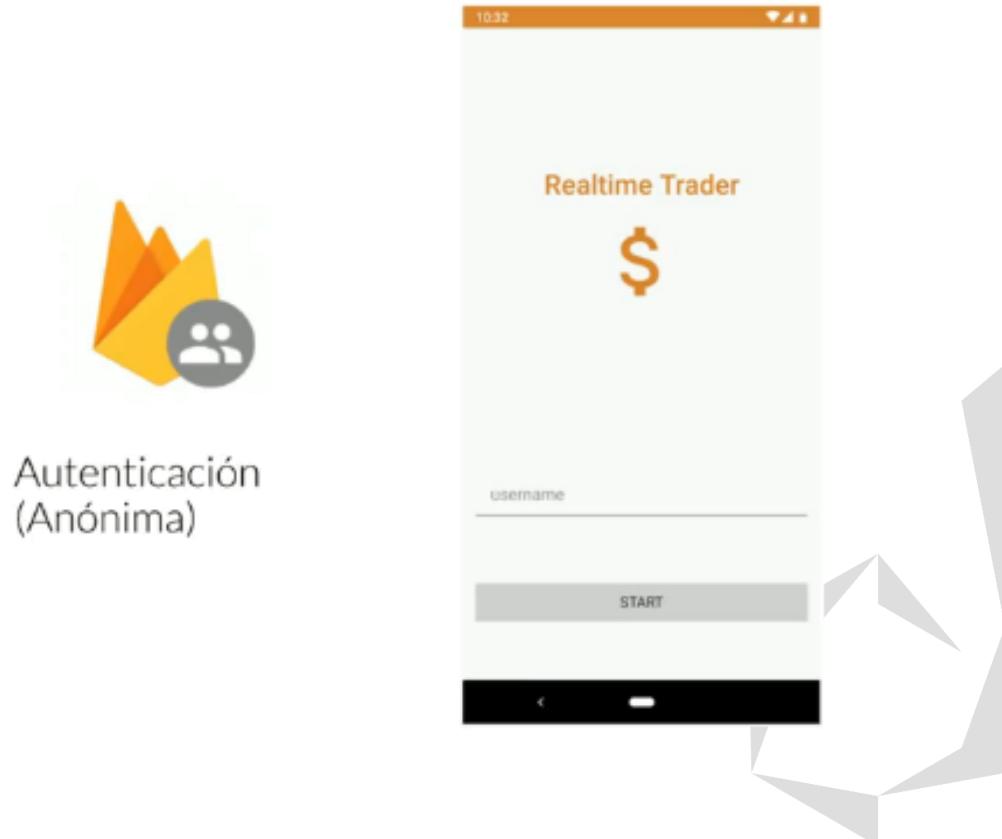


# Kotlin - Firebase

Firebase es una herramienta hecha para desarrolladores móviles Android, la cual ofrece servicios que ofrecen el módulo de Autenticación para realizar registros de usuarios y el módulo de Cloud Firestore para almacenamiento de datos.



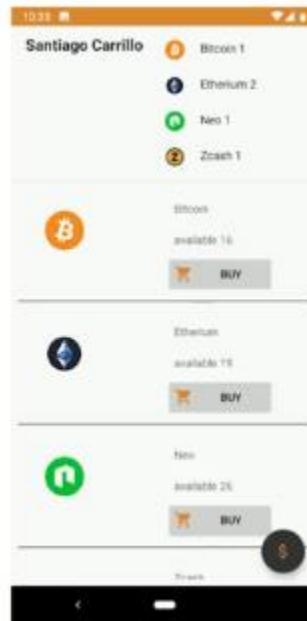
El módulo de autenticación permite crear cuentas temporales, las cuales pueden acceder a los datos protegidos con reglas de seguridad.



El módulo de Cloud Firestore proporciona una base de datos en tiempo real sincronizada, no relacional y flexible a escalar.



Cloud  
Firestore



Firebase tiene grandes alcances, por ejemplo, puede crear hasta una aplicación de Criptomonedas que actualice la información de cada moneda virtual.

Dentro de FireBase se pueden realizar varias acciones, que son las siguientes:

ML	<b>Machine Learning Kit</b> Machine Learning para apps móviles	User icon	<b>Autenticación</b> Autenticación simple y segura
Wi-Fi icon	<b>Cloud Firestore</b> Almacena y sincroniza datos a escala global	Globe icon	<b>Hosting</b> Ofrece recursos Web con velocidad y seguridad
Function icon	<b>Cloud Function</b> Ejecute código en el backend sin gestionar servidores	Cloud icon	<b>Cloud Storage</b> Almacena y accede archivos en la escala de Google
		Database icon	<b>Realtime Database</b> Almacena y sincroniza datos en milisegundos

## Servicios de Firebase:

- **Firestore:** Es una base de datos NoSQL que nos permite almacenar y sincronizar en tiempo real los datos de nuestra aplicación 🤝📝.
- **La Autenticación:** Nos permite crear, autenticar y administrar los usuarios de nuestras aplicaciones con email y password, login y signup con redes sociales, correos electrónicos de recuperación de contraseña, entre muchas otras cosas ✘☒ ✕ ✓.
- **Cloud Storage:** Sirve para almacenar archivos estáticos 🙄.

- **Cloud Messaging:** Sirve para enviar notificaciones .
- **Firebase Hosting (Firestore):** Permite desplegar nuestras aplicaciones y Cloud Functions para despliegues sin programar el código backend de la aplicación.

#### Medición de rendimiento KPI (Key Performance Indicator):

	<b>Crashlytics</b> Prioriza y repara fallos con reportes de errores en tiempo real
	<b>Monitor de Desempeño</b> Recibe datos del desempeño de tus Apps
	<b>Test Lab</b> Prueba tus aplicaciones en dispositivos de Google
	<b>In-App Messaging</b> Conéctate con tus usuarios por medio de mensajes de contexto
	<b>Cloud Messaging</b> Envía mensajes dirigidos y notificaciones.
	<b>Google Analytics</b> Obtén analítica de tu app gratis e ilimitada
	<b>Configuración Remota</b> Modifica tu App sin necesidad de despliegue de nueva versiones
	<b>Predicciones</b> Segmentación inteligente de usuarios basada en comportamiento predictivo.
	<b>Links Dinámicos</b> Maneja el crecimiento usando deep links
	<b>A/B Testing</b> Optimiza la experiencia de tu App a través de experimentación.
	<b>App Indexing</b> Maneja el tráfico de búsqueda de tu App móvil

## Configuración e instalación Cloud Firestore

Cloud Firestore es una base de datos en tiempo real, es un servicio de Firebase y está creado para las personas que no tienen mucho conocimiento de backend, ya que no es necesario tener mucho conocimiento del tema para poder conectar la aplicación Android a la base de datos, además de

proporcionar KPIs de rendimiento y registro de usuarios, haciendo que el proyecto pueda ser escalable con mayor facilidad.

## Cloud Firestore

La base de datos en tiempo real.

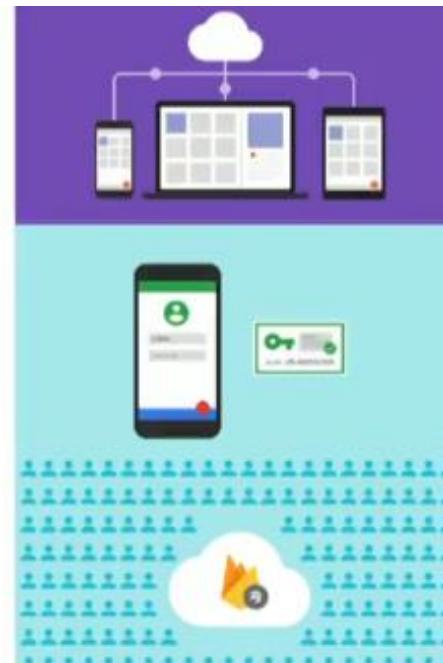


## Configuración del proyecto Android:

Ya que se haya creado la interfaz de usuario de la aplicación [en la carpeta app/res/layout](#) con el lenguaje de programación XML se podrán crear los archivos de funcionalidad con el lenguaje Kotlin [en la carpeta app/java/com.nombreDominio.nombreProyecto](#), estos son los que estarán en contacto con FireCloud para subir y bajar datos de la base de datos.

## Cloud Firestore

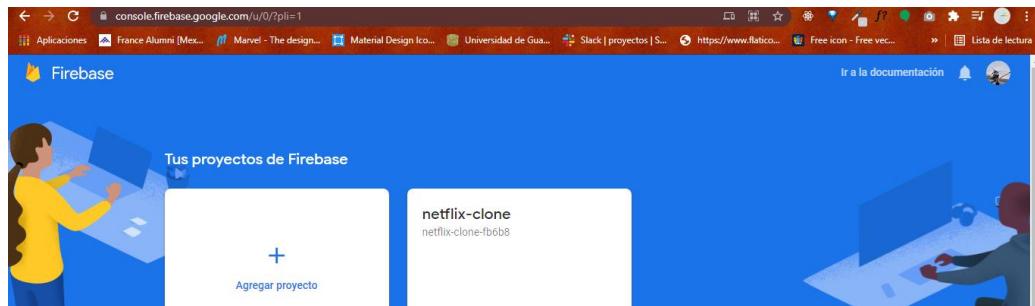
Con características especiales.



Para que se pueda conectar la aplicación Android con una base de datos de Firestore se debe realizar la configuración del proyecto Android primero desde la consola en línea de Firebase que se puede encontrar en el siguiente link:

<https://console.firebaseio.google.com/u/0/?pli=1>

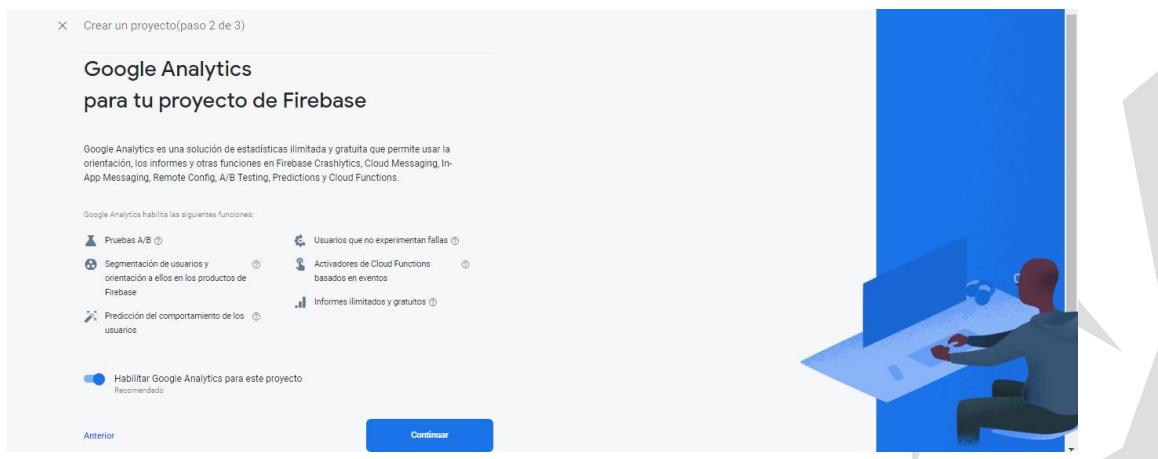
Para ello primero se debe iniciar sesión con una cuenta de Gmail, llegando finalmente a una pantalla como la que se muestra:



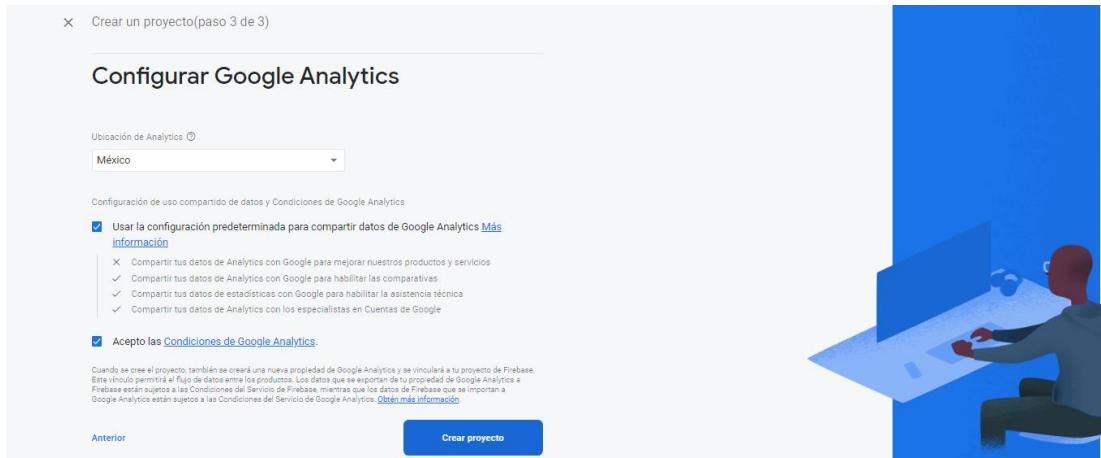
Ya habiendo llegado a esta ventana se debe dar clic en el signo de + Agregar proyecto, llegando a la siguiente pantalla donde se coloca el nombre del nuevo proyecto, en el nombre no se debe colocar espacios ni caracteres especiales:



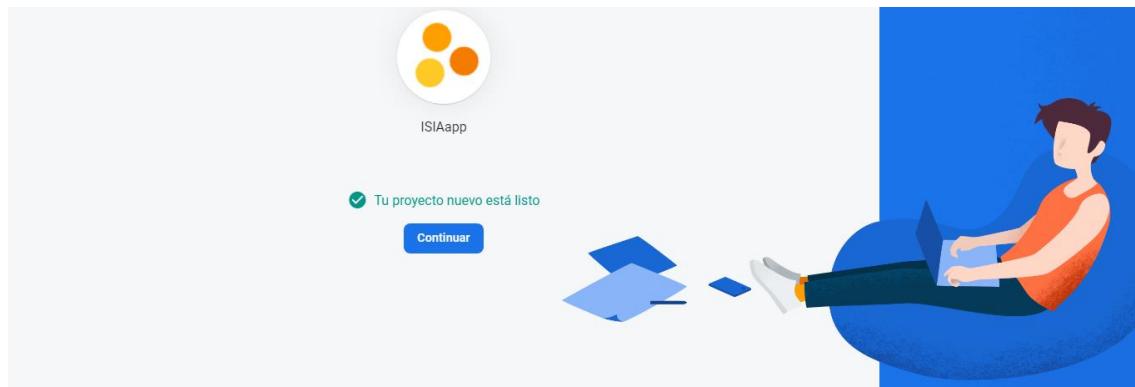
Al dar clic en el botón de Continuar pide habilitar algunas características de Google Analytics y se deja tal cual como está, dando clic en el botón de Continuar:



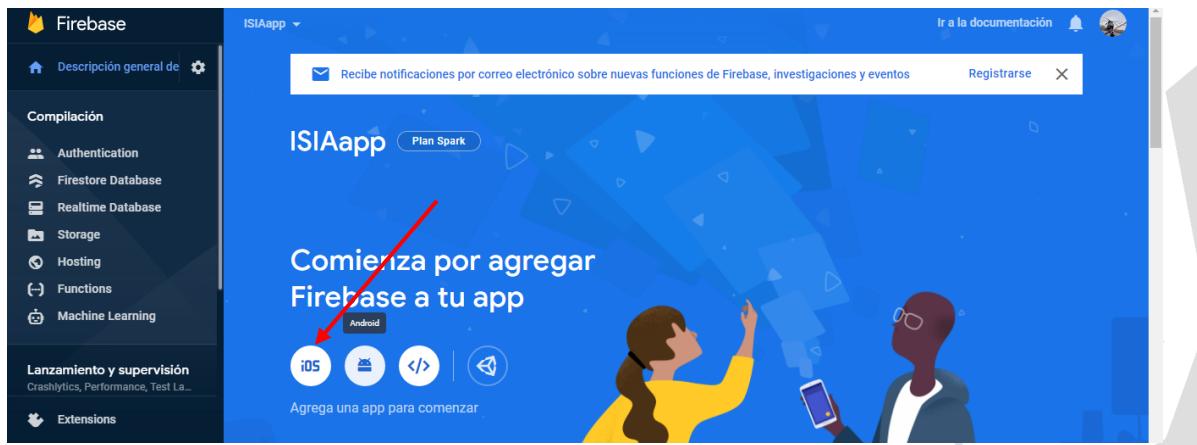
Después se indica el país desde donde se está creando el proyecto y se aceptan las condiciones de Google Analytics para finalmente dar clic en el botón de Crear proyecto:



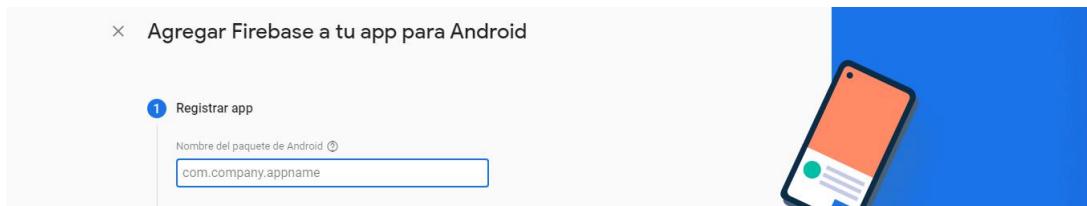
Con esto se creará el proyecto y ya estará configurado desde el lado de Firebase, ya que haya terminado de cargar simplemente se deberá dar clic en el botón de Continuar para que podamos pasar a enlazar el proyecto con la aplicación móvil:



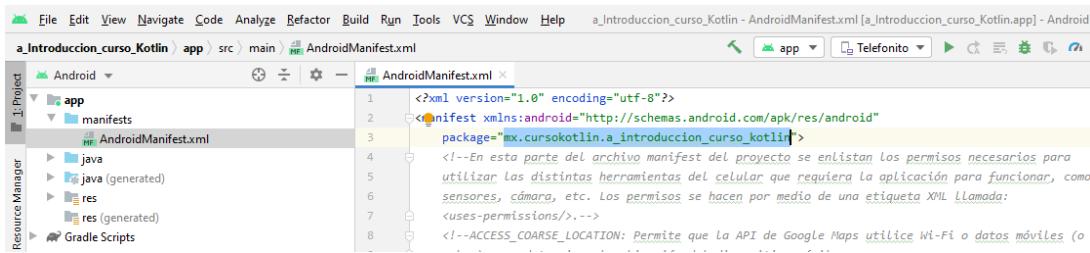
La siguiente pantalla es la interfaz de usuario de Firebase, donde se muestran todas las herramientas con las que se cuenta, en específico lo que se hará en este momento es simplemente conectar Firebase con un proyecto, este puede ser conectado a un proyecto con iOS, Android o un proyecto web, en esta ocasión se conectará con un proyecto Android:



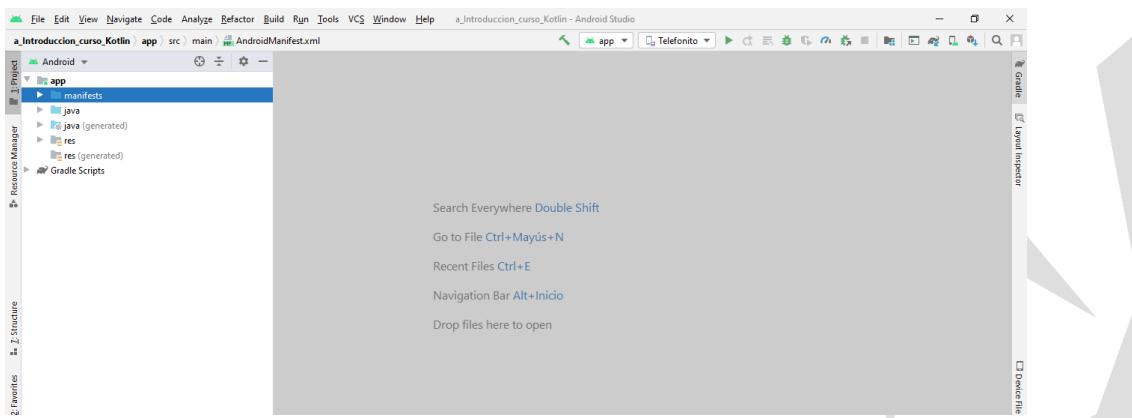
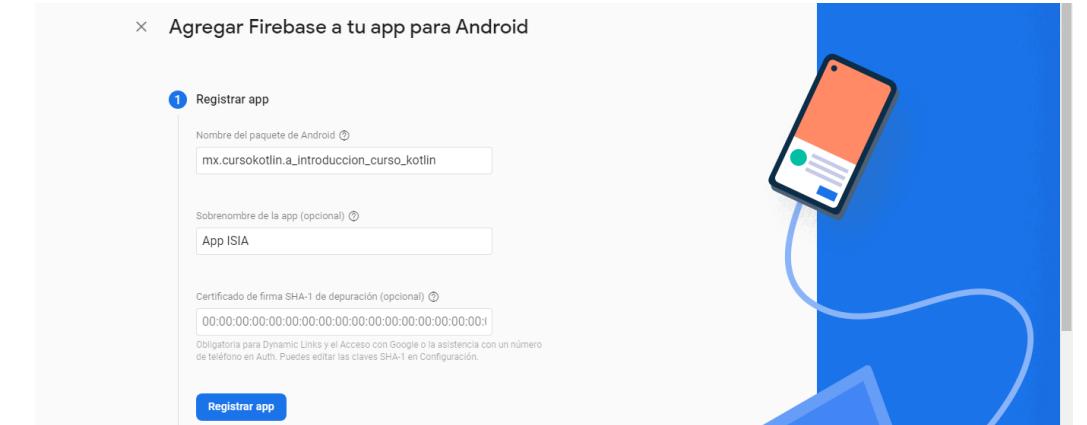
Y después pedirá el nombre del paquete o dominio de la aplicación Android a la cual se quiere conectar la base de datos.



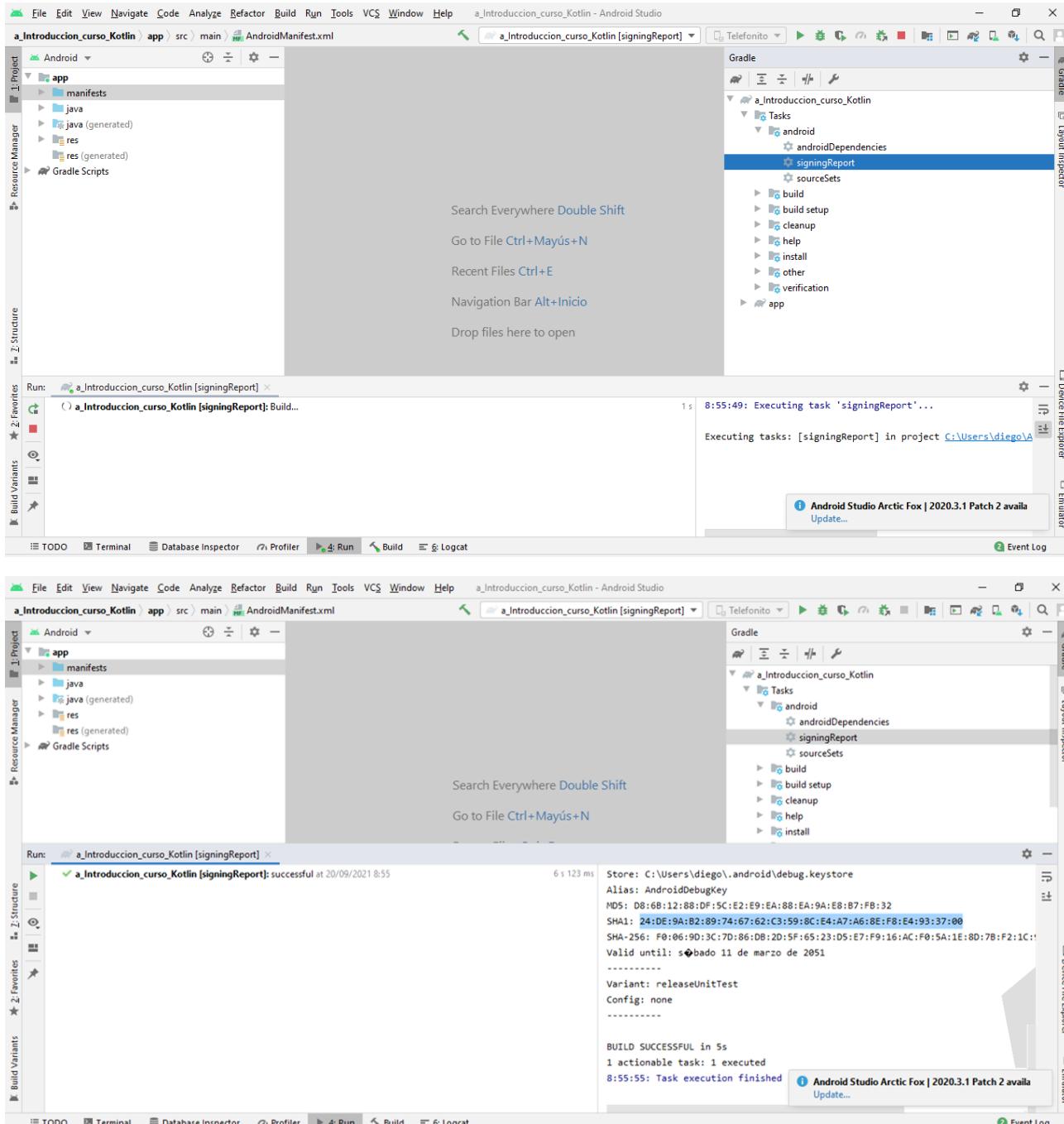
Para acceder al nombre del dominio o paquete debemos ingresar al archivo **app/manifests/AndroidManifest.xml** y el nombre del paquete se encontrará en el atributo package de la etiqueta manifest:



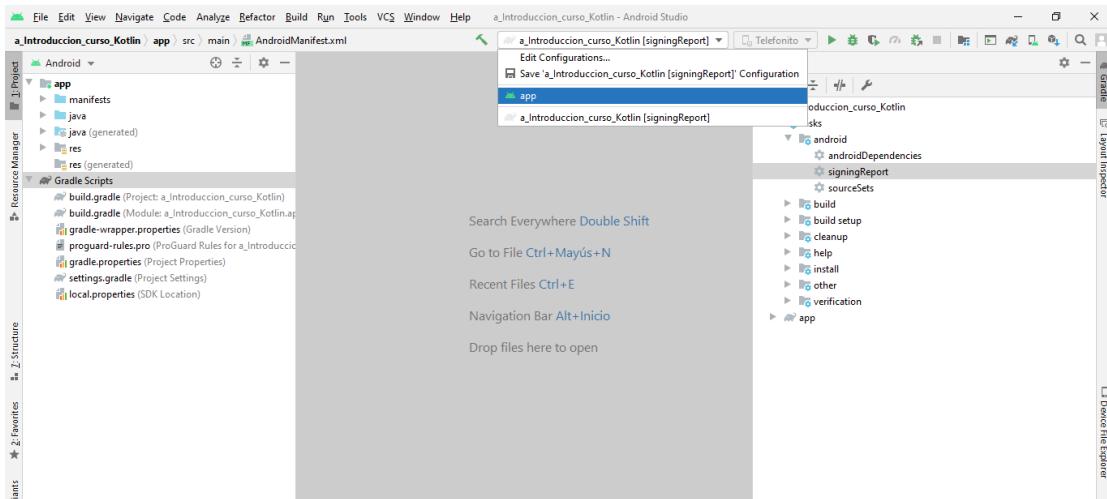
Después se nos pide un apodo para la app que es opcional ponerlo o no y finalmente pide el certificado de firma de depuración SHA, este se puede obtener de la pestaña lateral derecha que dice Gradle perteneciente al proyecto de Android Studio.



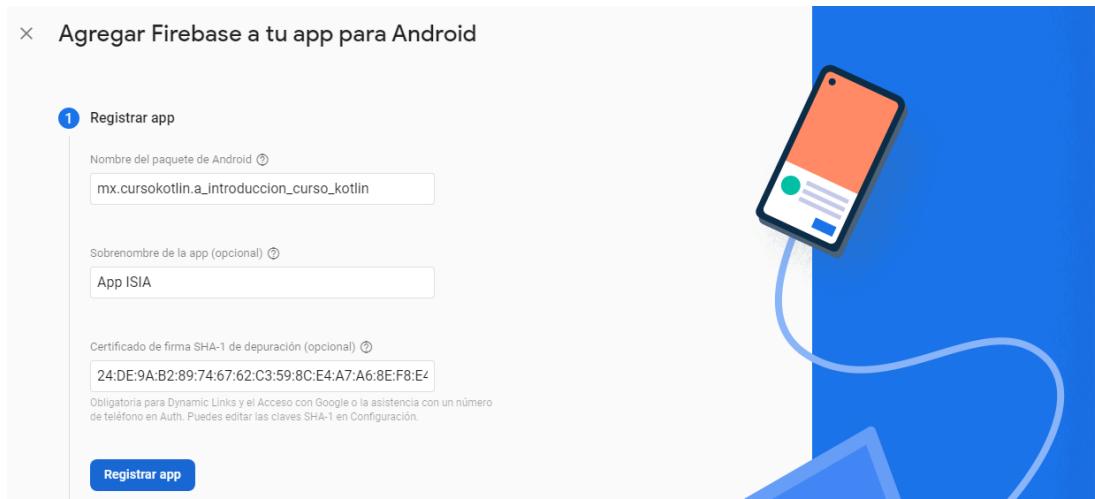
Ya dentro de la pestaña de Gradle de Android Studio debemos entrar a la carpeta **com.nombreDominio.nombreProyecto/Tasks/android/signingReport** y al dar doble clic en esta parte se ejecutará en la consola inferior derecha un programa que devolverá el código SHA1 que es el que necesitamos:



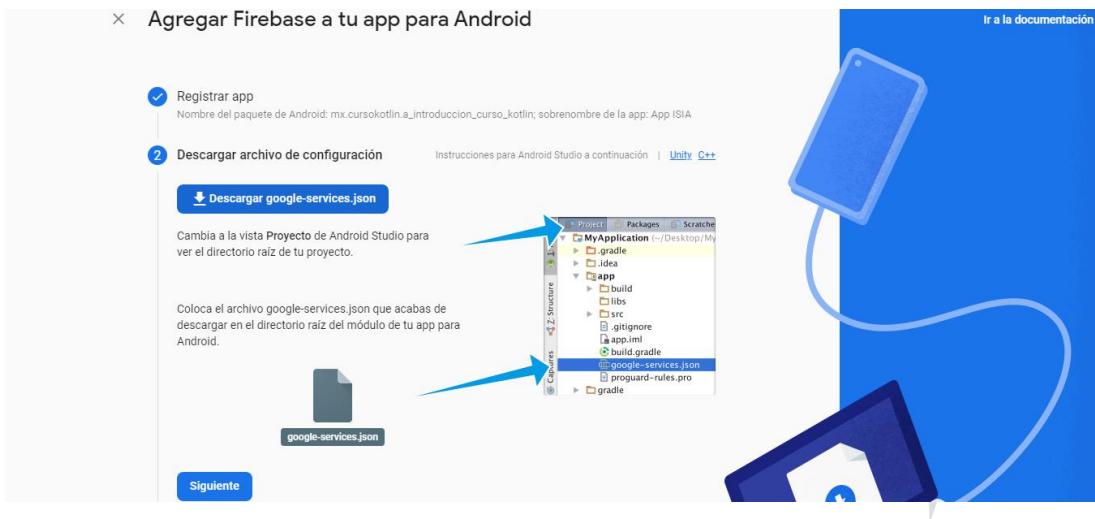
Al ejecutar la instrucción que devuelve el certificado de seguridad se cambiará la configuración del proyecto, para que pueda seguir ejecutando la app debo regresarl a que tenga la opción de app:



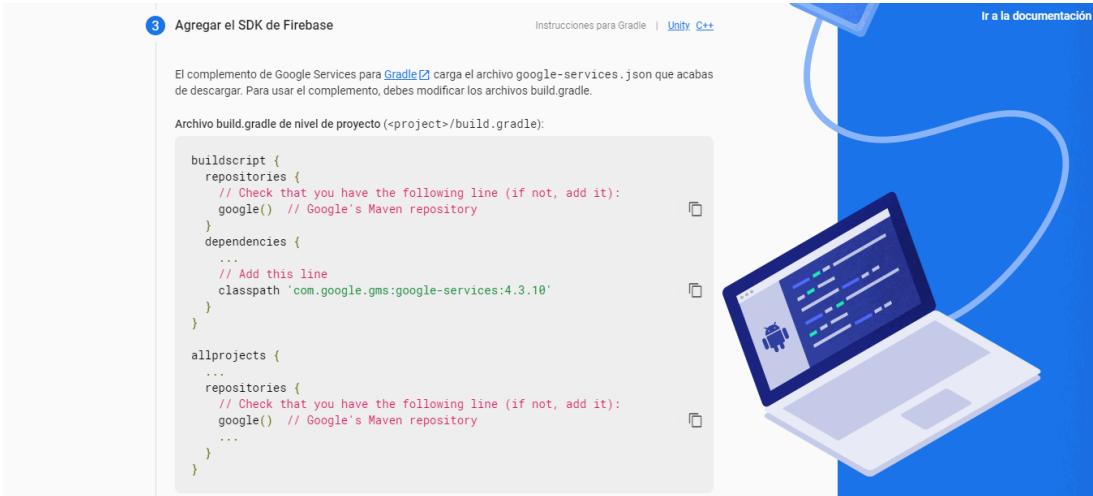
Con esto ya se podrá dar clic en el botón de Registrar app.



Ya que se haga registrado la app, se nos devolverá un archivo json que se debe descargar y agregar a la siguiente carpeta del proyecto Android que se encuentra en el ordenador:  
**C:\Users\nombre\_usuario\AndroidStudioProjects\ nombreProyecto\app**



Ya que se haya descargado el archivo google-services.json podremos dar clic en el botón de Siguiente, en donde se indican las dependencias que se deben agregar al archivo **app/Gradle Scripts/build.gradle(Project: NombreProyecto.app)** del proyecto en donde se debe agregar un classpath:



En específico se debe contar con el repositorio de google(), en este caso ya lo tenía.

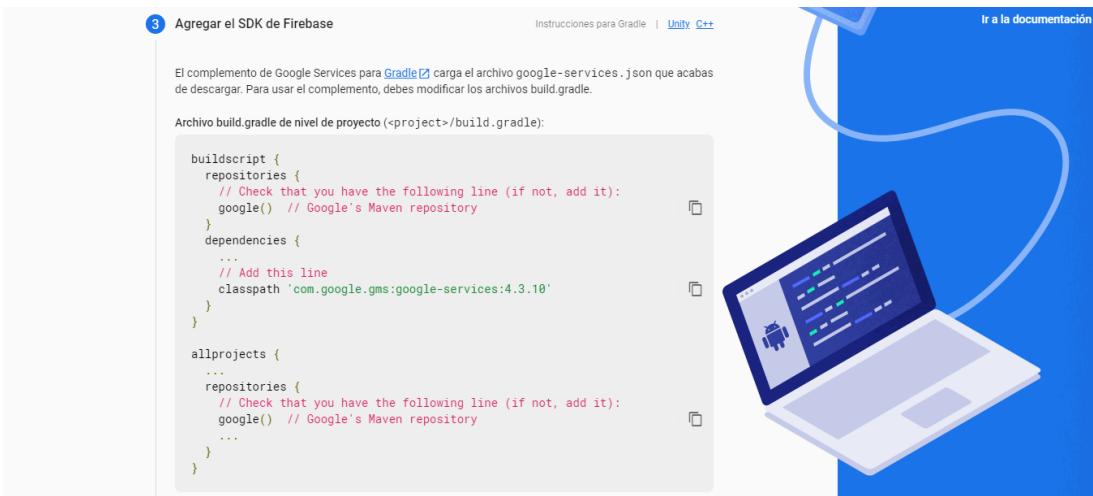
```

buildscript {
    /*Dentro del archivo build.gradle (Project: NombreProyecto.app) se indica la versión de Kotlin
    que se está utilizando en el proyecto, si se quisiera actualizar la versión de Kotlin nos
    debemos introducir a:
    Tools → Kotlin → Configure Kotlin Plugin Updates → Check again → Install
    Y si hay una versión nueva, el cambio se verá reflejado en esta parte del código.*/
    ext.kotlin_version = "1.4.32"
}

repositories {
    google()
    jcenter()
}

```

Y en las dependencias es donde se debe agregar el classpath que viene en las instrucciones de la página de firebase: classpath '**com.google.gms:google-services:4.3.10**'



```

Y si hay una versión nueva, el cambio se verá reflejado en esta parte del código.*
ext.kotlin_version = "1.4.32"
repositories {
    google()
    jcenter()
}
dependencies {
    /*Librerías incluidas por default*/
    /*En esta parte del código se indica la versión de gradle que se está utilizando.*/
    classpath "com.android.tools.build:gradle:4.1.0"
    classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
    /*Librerías añadidas*/
    /*Android Jetpack: Son un conjunto de herramientas que sirven para añadir funcionalidad a los layouts de la aplicación Android, entre otras cosas. Se necesita agregar dos dependencias adicionales al archivo build.gradle(Module: NombreProyecto.app) para implementar Jetpack.*/
    classpath 'androidx.navigation:navigation-safe-args-gradle-plugin:2.1.0'
    /*Firebase: Cloud Firebase es una herramienta hecha para desarrolladores móviles Android, la cual ofrece servicios que ofrecen el módulo de Autenticación y el módulo de Cloud Firestore para bases de datos.*/
    classpath 'com.google.gms:google-services:4.3.10'
}
}

```

Además en los repositorios también debe estar agregado el repositorio de google(), en este caso ya está agregado.

```

allprojects {
    repositories {
        google()
        jcenter()
    }
}

task clean(type: Delete) {
    delete rootProject.buildDir
}

```

Luego se deben agregar las dependencias indicadas en la página de Firebase al archivo **app/Gradle Scripts/ build.gradle(Module: NombreProyecto.app)** del proyecto, en esta parte se debe indicar el lenguaje de programación que se está usando ya sea Java o Kotlin:

Java  Kotlin

Archivo build.gradle de nivel de app (<project>/<app-module>/build.gradle):

```

apply plugin: 'com.android.application'
// Add this line
apply plugin: 'com.google.gms.google-services'

dependencies {
    // Import the Firebase BoM
    implementation platform('com.google.firebase:firebase-bom:28.4.1')

    // Add the dependency for the Firebase SDK for Google Analytics
    // When using the BoM, don't specify versions in Firebase dependencies
    implementation 'com.google.firebase:firebase-analytics-ktx'

    // Add the dependencies for any other desired Firebase products
    // https://firebase.google.com/docs/android/setup#available-libraries
}

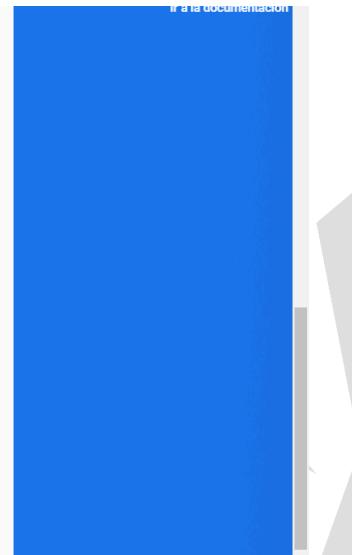
```

Si usas la BoM de Firebase para Android, tu app siempre utilizará versiones compatibles de la biblioteca de Firebase.  
[Más información](#)

Por último, presiona "Sincronizar ahora" en la barra que aparece en el entorno IDE:

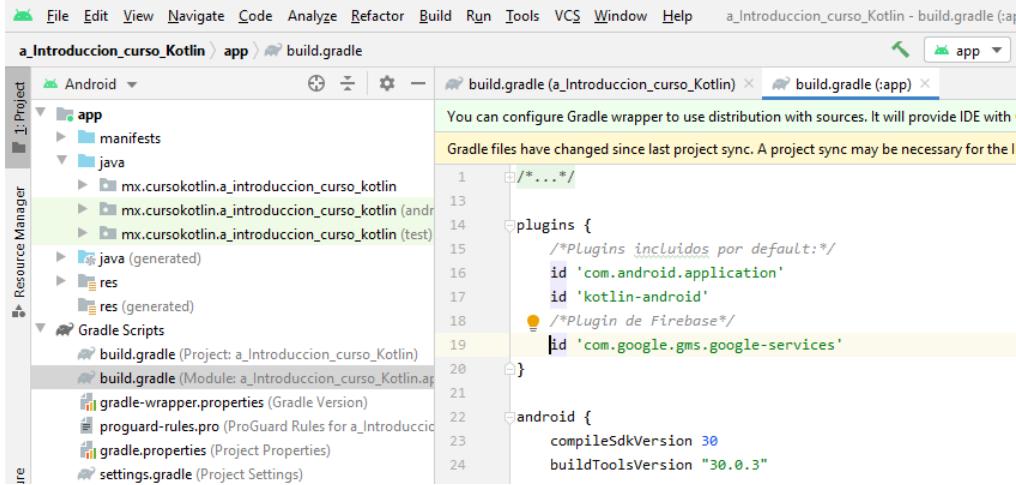
Gradle files have changed since last sync [Sync now](#)

Anterior [Siguiente](#)



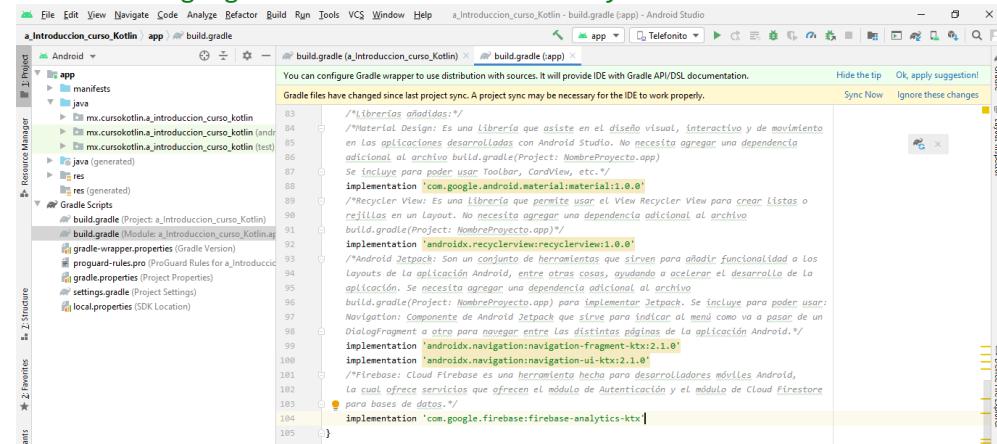
Primero debemos añadir los plugins indicados por medio de la misma nomenclatura que está en el proyecto, que es distinta a la mencionada en la documentación de Firebase:

```
id 'com.google.gms.google-services'
```

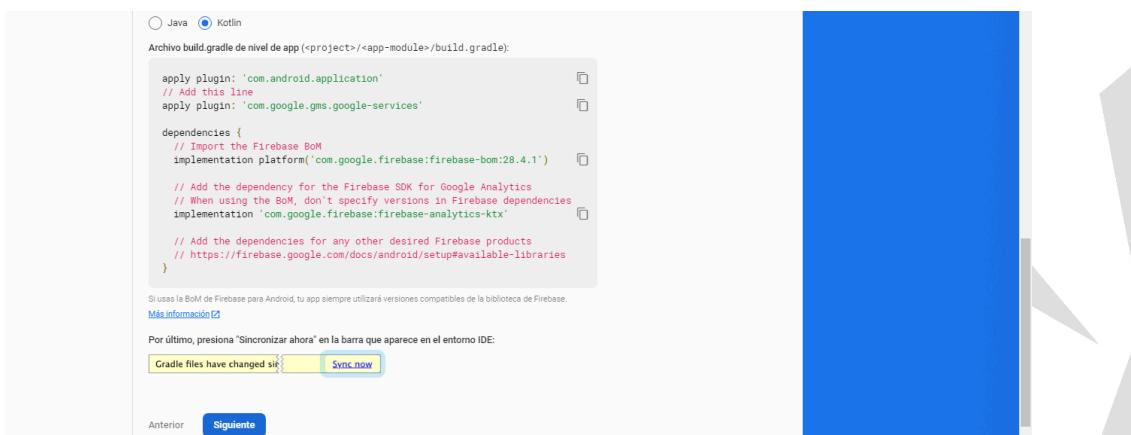


Luego debemos añadir una nueva dependencia al archivo y no se debe dar clic en Sync Now:

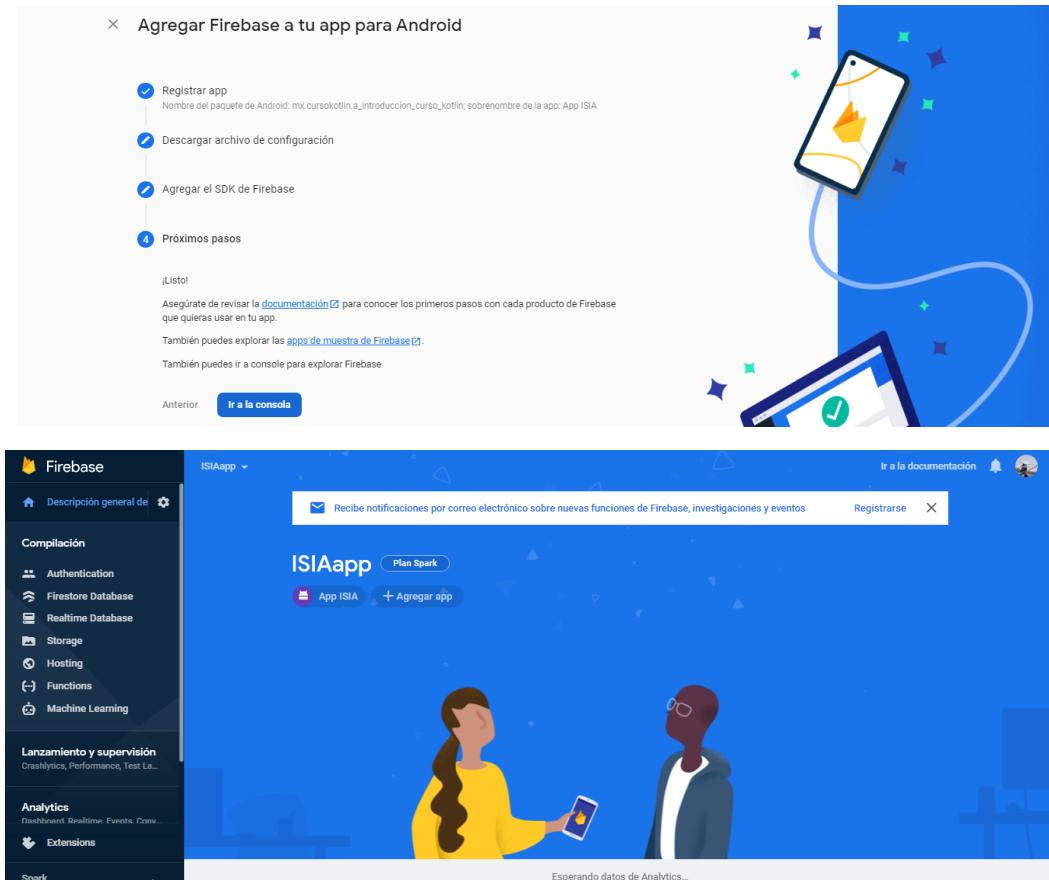
```
implementation 'com.google.firebaseio:firebase-analytics:17.2.2'
```



Ya añadidas todas las dependencias al proyecto Android, debemos dar clic en el botón de Siguiente en la página de Firebase, esto se debe hacer antes de dar clic en Sync Now en los archivos Gradle.



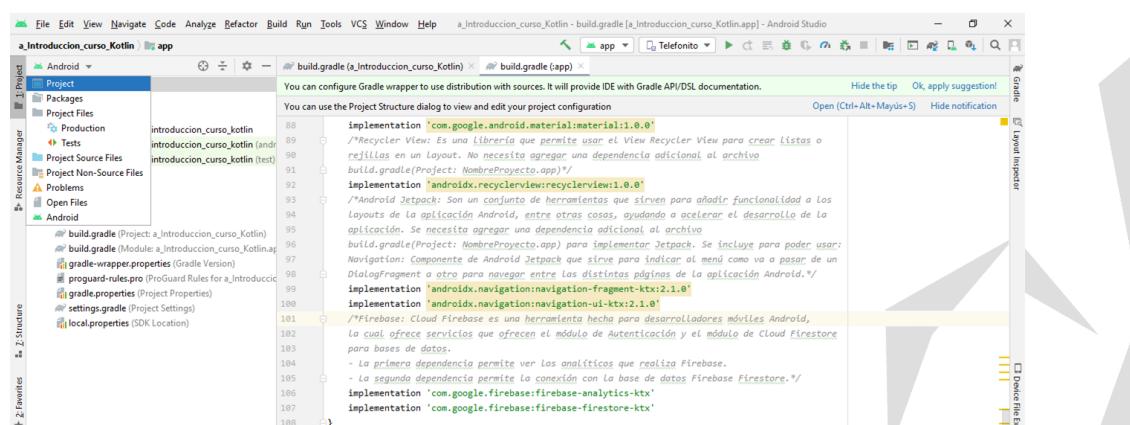
Y finalmente podremos regresar a la consola del proyecto ya enlazado al dar clic en el botón de Ir a la consola.



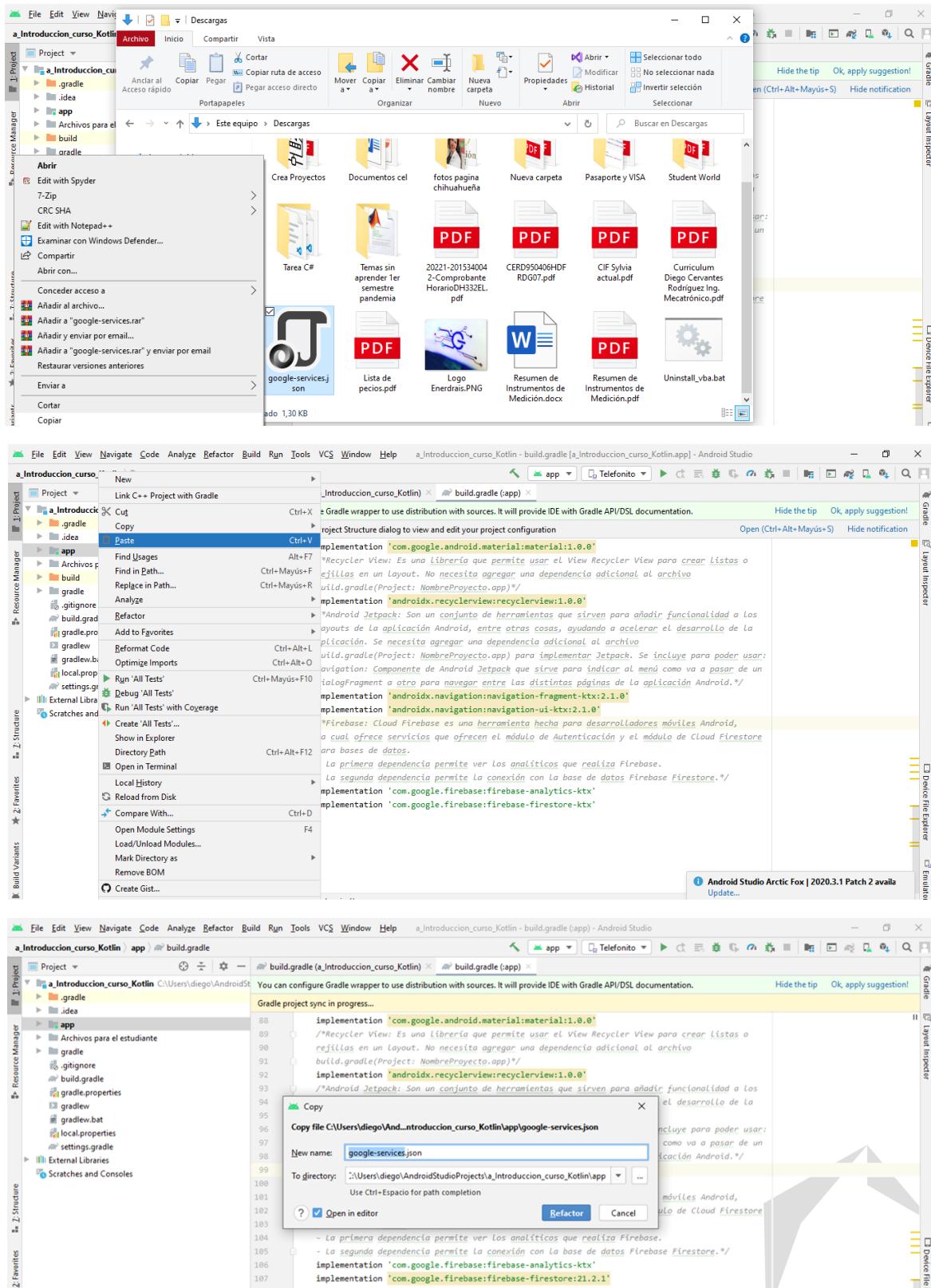
Posteriormente, para poder utilizar la base de datos de Firebase debemos agregar una dependencia más en el archivo de **app/Gradle Scripts/ build.gradle(Module: NombreProyecto.app)** llamada:

```
implementation 'com.google.firebaseio:firebase-firebase:21.2.1'
```

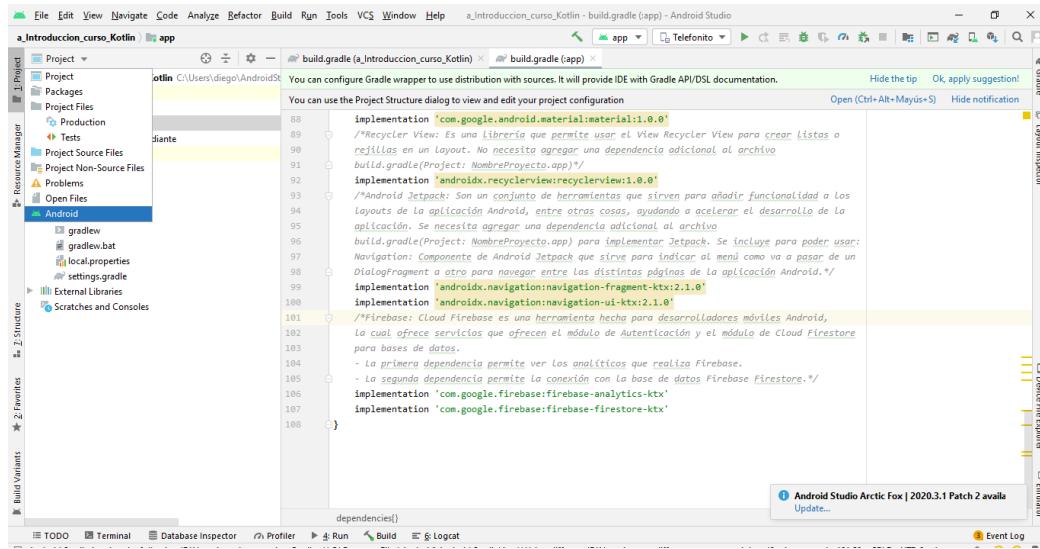
Antes de poder sincronizar el proyecto para activar las nuevas dependencias se debe agregar el archivo descargado de la página de Firebase llamado google-services.json, este se debe añadir a la carpeta **C:\Users\nombre\_usuario\AndroidStudioProjects\ nombreProyecto\app** del ordenador, para ello es útil usar la vista de Project en vez de la de Android un segundo:



Para poder pegar el archivo json primero debo dar clic en copiar sobre el archivo donde sea que se encuentre y después pegarlo en la carpeta de app:



Y ahora ya podremos regresar a la vista de Android como estaba originalmente.

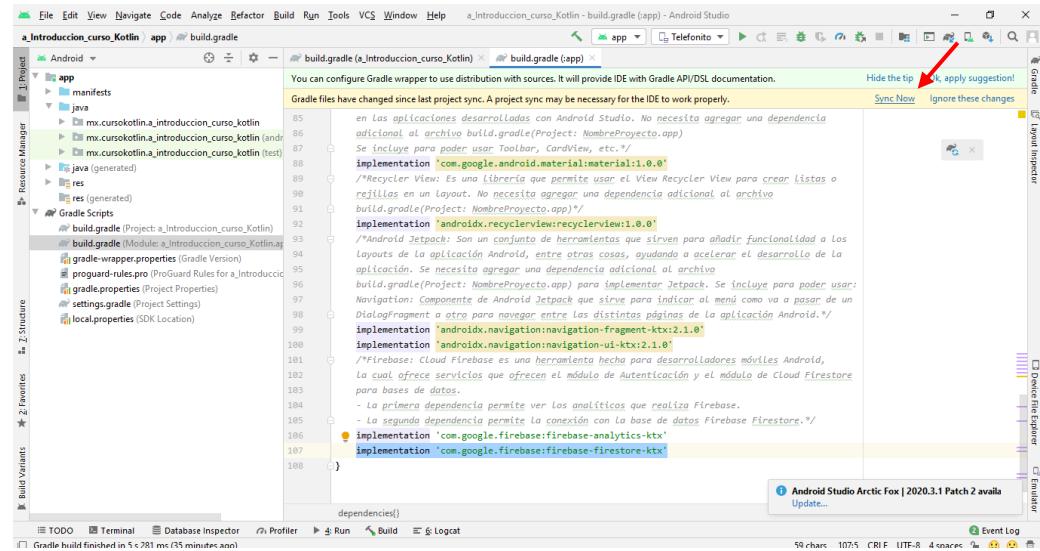


```

dependencies {
    implementation 'com.google.android.material:material:1.0.0'
    //Recycler View: Es una librería que permite usar el View Recycler View para crear listas o rejillas en un layout. No necesita agregar una dependencia adicional al archivo build.gradle(Project: NombreProyecto.app)
    implementation 'androidx.recyclerview:recyclerview:1.0.0'
    //Android Jetpack: Son un conjunto de herramientas que sirven para añadir funcionalidad a los layouts de la aplicación Android, entre otras cosas, ayudando a acelerar el desarrollo de la aplicación. Se necesita agregar una dependencia adicional al archivo build.gradle(Project: NombreProyecto.app) para implementar Jetpack. Se incluye para poder usar: Navigation: Componente de Android Jetpack que sirve para indicar al menú como va a pasar de un DialogFragment a otro para navegar entre las distintas páginas de la aplicación Android./*
    implementation 'androidx.navigation:navigation-fragment-ktx:2.1.0'
    implementation 'androidx.navigation:navigation-ui-ktx:2.1.0'
    //Firebase: Cloud Firebase es una herramienta hecha para desarrolladores móviles Android, la cual ofrece servicios que ofrecen el módulo de Autenticación y el módulo de Cloud Firestore para bases de datos.
    - La primera dependencia permite ver los analíticos que realiza Firebase.
    - La segunda dependencia permite la conexión con la base de datos Firebase Firestore./*
    implementation 'com.google.firebase.firebaseio-analytics-ktx'
    implementation 'com.google.firebase.firebaseio-firestore-ktx'
}

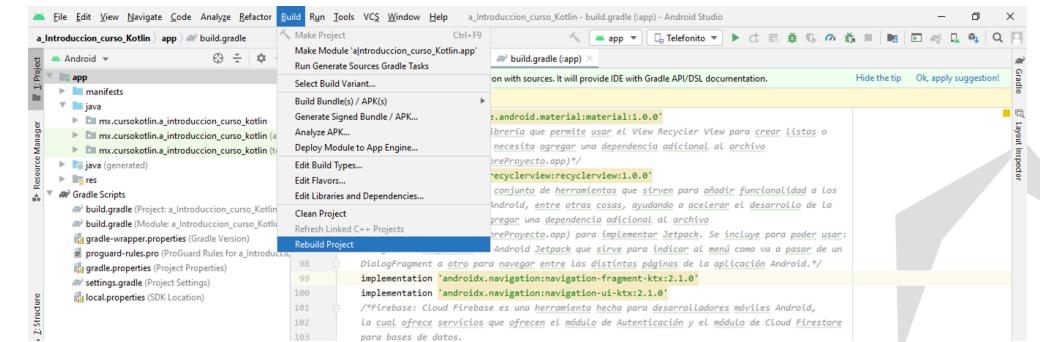
```

Ahora ya deberemos dar clic en el botón de Sync Now para sincronizar el proyecto con las nuevas dependencias agregadas.



The screenshot shows the same build.gradle file as above, but the "Sync Now" button in the toolbar is highlighted with a red arrow. The status bar at the bottom indicates "Gradle build finished in 5 s 281 ms (35 minutes ago)".

Si por alguna razón ya había dado clic en Sync Now antes de agregar el archivo, puedo dar clic en la opción del menú Build → Rebuild Project.



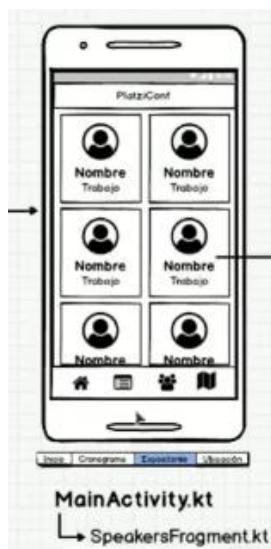
The screenshot shows the "Build" menu open, with "Rebuild Project" highlighted. The status bar at the bottom indicates "59 chars 1075 CRLF 1UTF-8 4 spaces".

Con eso ya se habrá conectado también la base de datos de Firestore, después lo que se debe hacer es ingresar los datos de la aplicación, los cuales se podrán mostrar en forma de lista o rejilla por medio de la etiqueta Recycler View que se crea en la carpeta de interfaces de usuario **en la carpeta app/res/layout** por medio del lenguaje XML:

- **Listas:** Es un View donde su contenido y el formato en el que se muestra se va a repetir, esto sirve para crear un enlistado de elementos como horarios y descripción de conferencias, precio y descripción de productos, etc.
  - **ListView:** Es un tipo de View muy antiguo que se utilizaba para crear listas en Android, pero en la actualidad se opta por mejor utilizar RecyclerView que también sirve para crear listas.
  - **RecyclerView:** Es un View que permite colocar elementos en forma de lista o rejilla:
    - **Lista (List):** View que coloca un elemento tras otro en forma de párrafo.



- **Rejilla o grilla (Grid):** View que ordena varios elementos en contenedores cuadrados de dos en dos, tres en tres, etc.



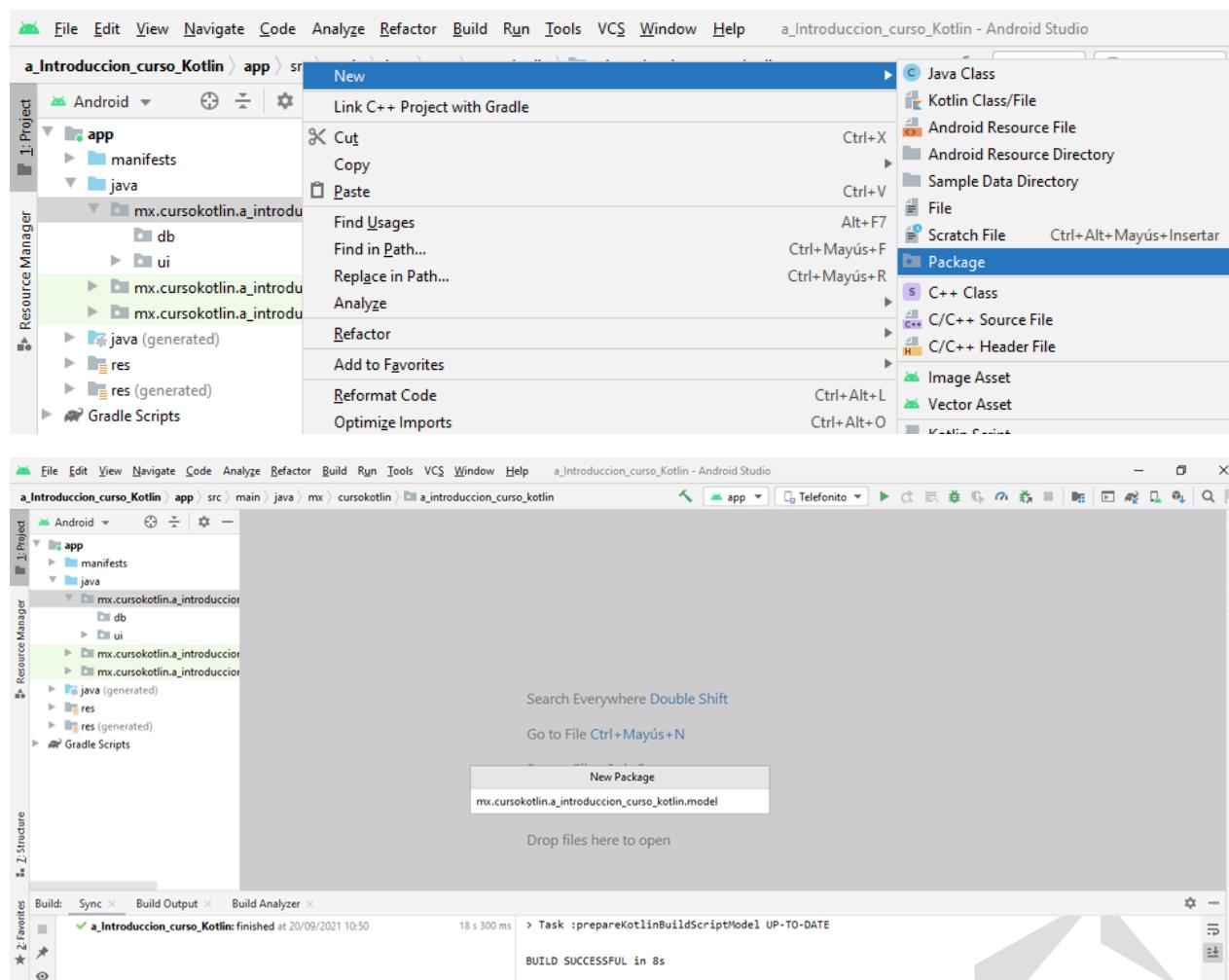
- Para poder utilizar el RecyclerView, debo descargar una dependencia en el archivo de build.gradle(Module: NombreProyecto.app), esto para poder utilizar la librería de **Recycler View** por medio del siguiente código:

```
o implementation 'androidx.recyclerview:recyclerview:1.0.0'
```

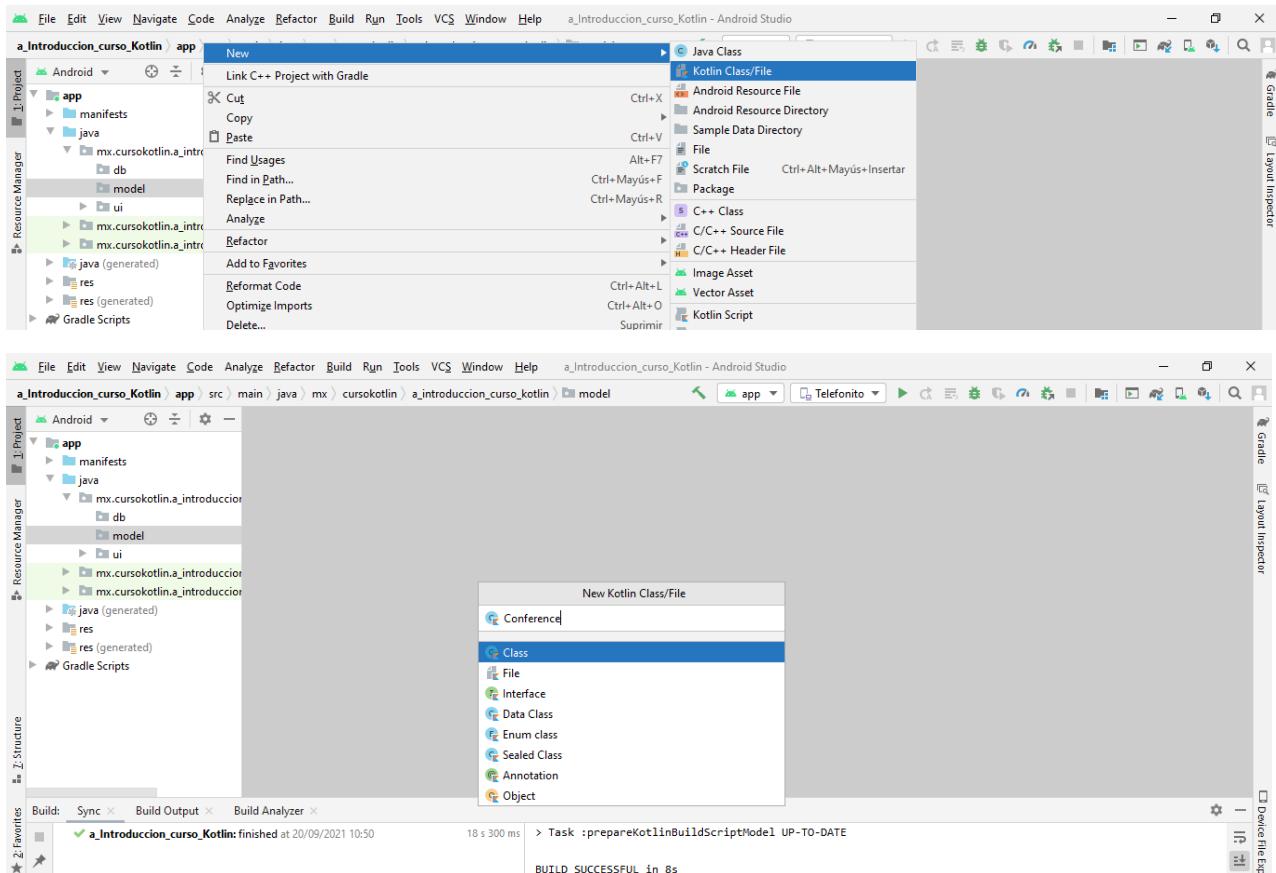
Otra función que tienen las bases de datos de Firestore es la de mostrar ubicaciones por medio de los mapas de Google Maps.

## Creación de recursos: app/java/res

Todos los datos se van a mapear en clases, para ello se crea un nuevo paquete en la carpeta app/java/com.nombreDominio.nombreProyecto, esto se hace dando clic derecho en el paquete com.nombreDominio.nombreProyecto y seleccionando la opción de New → Package → Nombre del nuevo paquete: Model, de esta misma manera se crean todos los demás paquetes del proyecto:



Dentro del nuevo paquete se da clic derecho y se selecciona la opción de New → Kotlin File/Class → Class → Nombre del nuevo archivo: Conference.

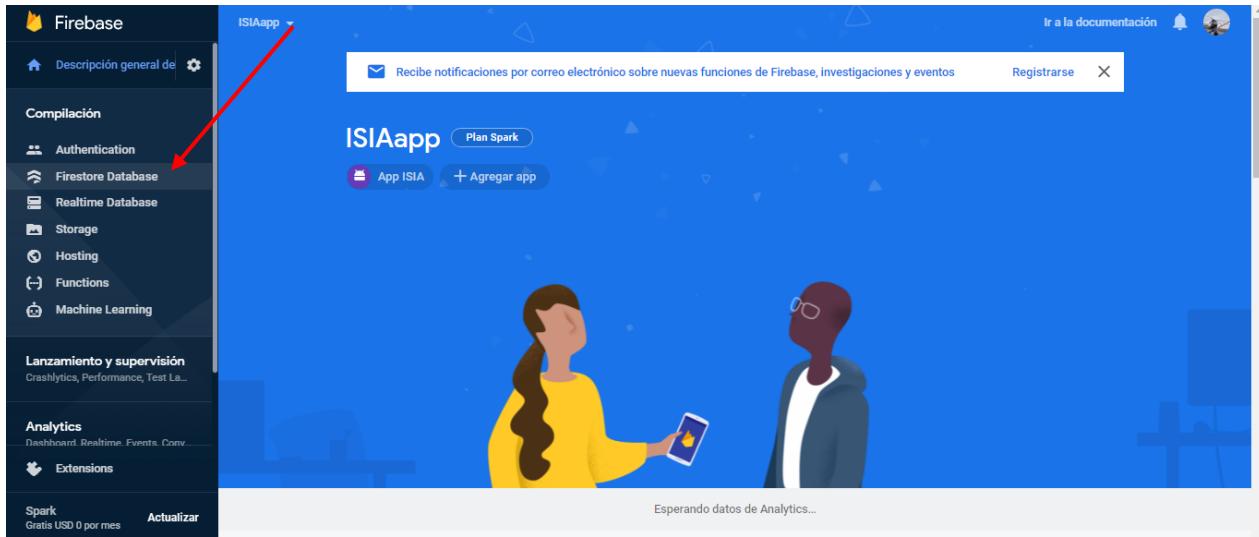


Se creará una clase por cada serie de datos (llamadas colecciones de datos) que se quiera subir a la base de datos. Dentro de las clases creadas para recibir datos se crearán variables que almacenen cada dato requerido de la serie de datos.

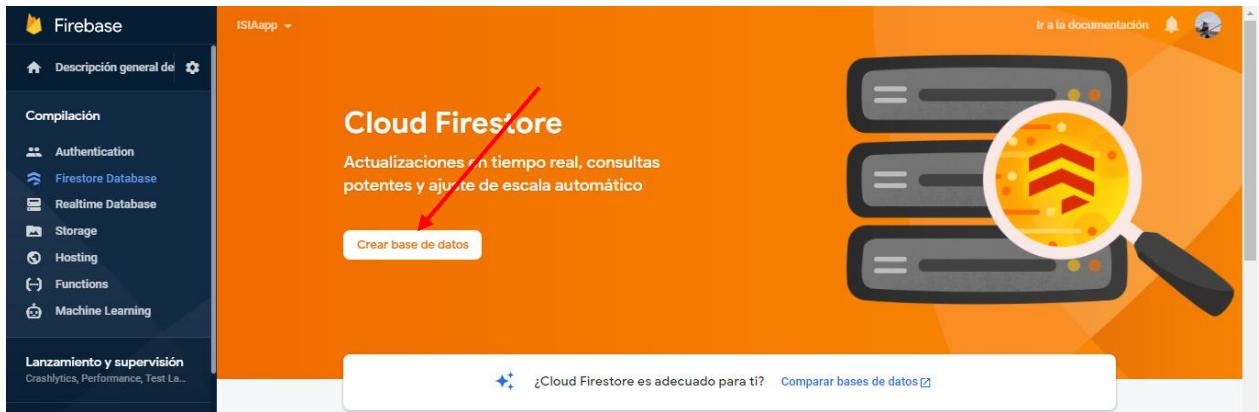
- **model:** Paquete que contiene todas las colecciones de datos que se pretenda subir a la base de datos en forma de Clases Kotlin, las colecciones se refieren a los grupos de datos que se quiere subir a la base de datos FireCloud.
- **network:** Paquete que contiene las clases necesarias para extraer y mostrar las colecciones de datos requeridos en la interfaz de la aplicación, las colecciones se refieren a los grupos de datos que se quiere mostrar en las listas o rejillas de cada pantalla.
- **viewmodel:** Paquete que permite comunicar las clases del paquete network y model con las Activities y dialogFragments de la interfaz de la aplicación Android.

## Creación de la base de datos en Firebase

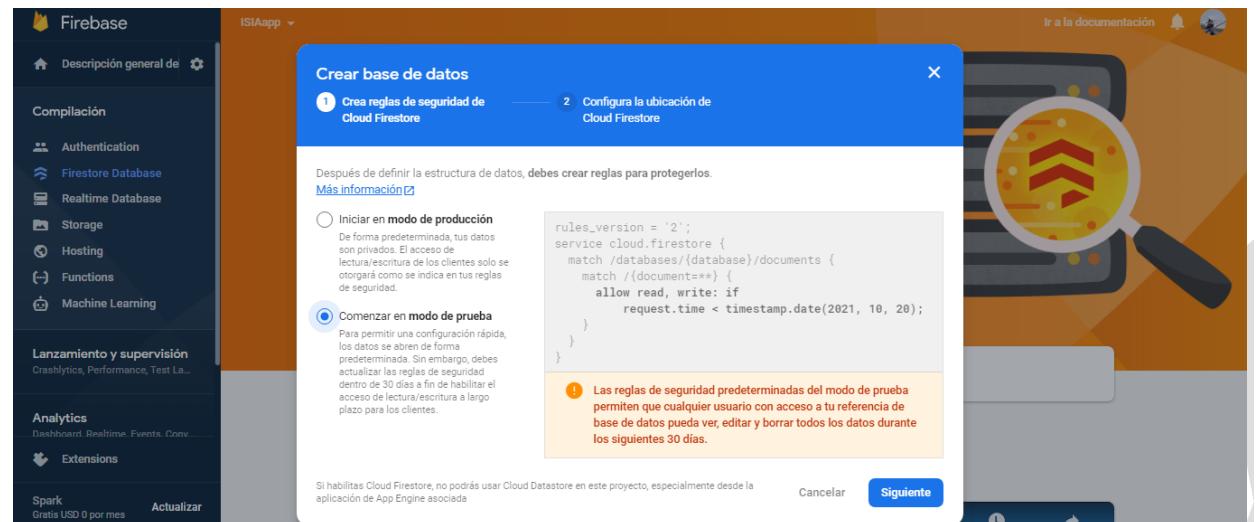
Ya que se haya enlazado el proyecto con la base de datos Firebase y creado las clases en el proyecto que reciban los datos de la base de datos se usará la consola en línea del proyecto, introduciéndonos en la opción de Firestore Database que es uno de los servicios de Firebase.



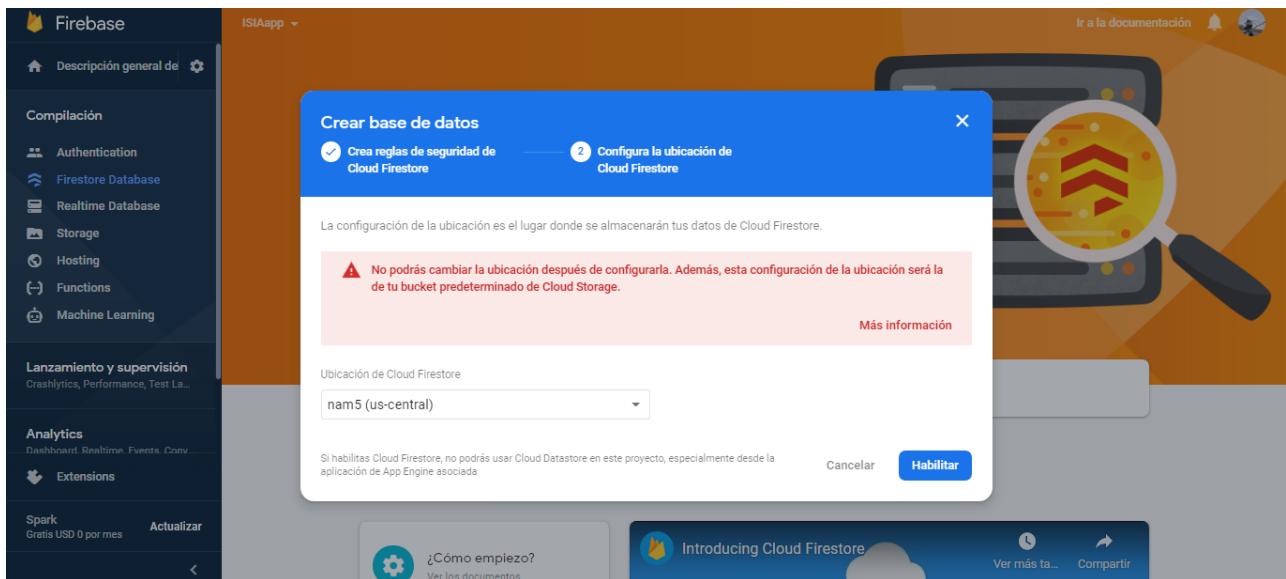
En esta ventana se da clic en el botón de Crear base de datos.



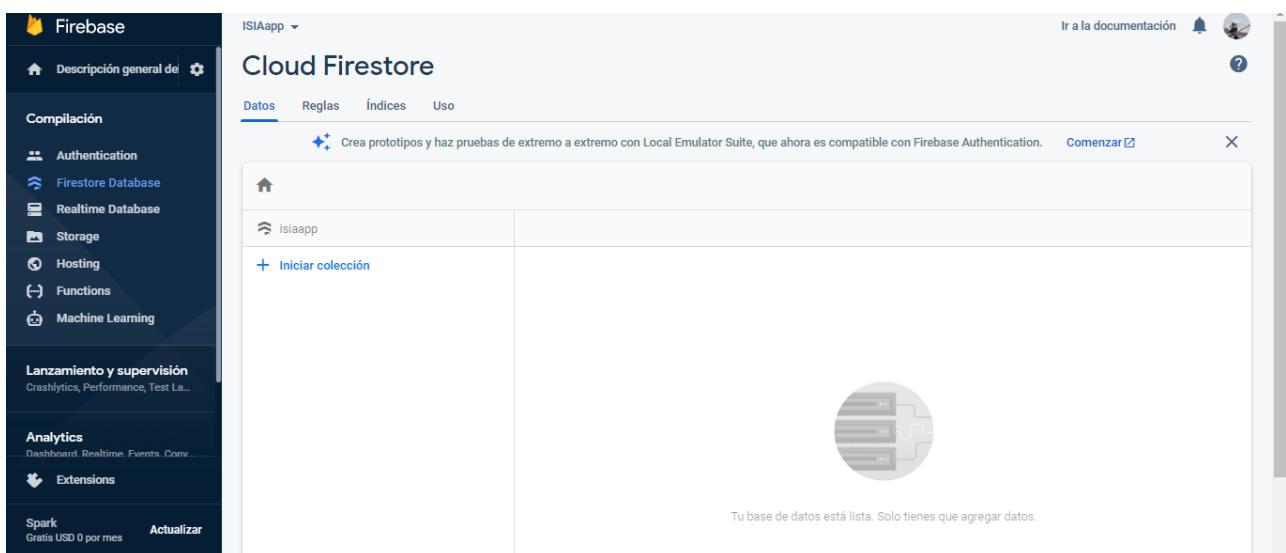
Y en este caso vamos a seleccionar el modo de prueba y dar clic en el botón de Siguiente:



Y posteriormente se selecciona la ubicación del servidor de la base de datos, usualmente se deja la que ya viene de forma predeterminada y se da clic en el botón de Habilitar:



En esta parte es donde ya se suben los datos de la aplicación:



## Subir datos a la base de datos FireCloud:

Para poder subir datos a la base de datos Firecloud de Firebase ya se debe contar con el paquete **model**, además los datos deben estar ingresados en forma de JSON y se deben subir desde el código Kotlin de Android Studio, para ello se necesita lo siguiente:

- **JSON de los datos que se quieren introducir a la base de datos:** Todos los datos que se quieran introducir a la base de datos deben estar en formato JSON, sus siglas significan JavaScript Object Notation y simplemente es un pedazo de código que sirve para la transmisión de datos, este llegó a reemplazar a XML que es un lenguaje de marcado para la transmisión de datos más antiguo y pesado.

- Los JSON se componen principalmente de dos partes, **key (clave)** y **value (valor)**, ambos rodeadas por llaves, cada **key** es seguida de dos puntos para distinguirlas de los **values** y los datos del JSON son separados entre sí por llaves de apertura y cierre:
  - `{"ciudad":"Nueva York", "país":"Estados Unidos"}`
- Pueden existir JSON compuestos por arrays, donde cada elemento del array estará separado por comas excepto el último, en el siguiente ejemplo se puede observar como un array se usó para describir varios **values** para un mismo **key**, que a su vez tienen sus propios **keys** y **values**, aunque esto se puede usar de distintas maneras:
  - `"ciudad": [  
 {"primerNombre":"Tom", "Apellido":"Gutiérrez"},  
 {"primerNombre":"Tom", "Apellido":"Gutiérrez"},  
 {"primerNombre":"Tom", "Apellido":"Gutiérrez"}  
]`
- En Kotlin los JSON se almacenan en una **variable inmutable** (declarada con la palabra reservada **val**, donde no se les puede modificar ni borrar su valor) y se declaran por medio de una clase en específico, dependiendo del tipo de JSON que sea, que puede ser de los siguientes tres tipos, en donde se usan distintas clases para convertirlos a objetos Kotlin:
  - **JSON simple:** Es un JSON normal, rodeado de llaves de inicio y cierre, donde sus elementos solo tienen **key** y **value**.
    - Este tipo de JSON se transforma a un objeto **JSONObject("")**.
    - Para obtener algún **value** del JSON y guardarlo en una variable se pueden usar distintos métodos dependiendo del tipo primitivo del dato que se quiere obtener que está asignado a un **key** en específico, como lo son:
      - `JSONObject.getString("nombre_key")`
      - `JSONObject.getInt("nombre_key")`
      - `JSONObject.getLong("nombre_key"), etc.`
  - **JSON tipo Array:** La estructura de este tipo de JSON comienza con corchetes de array en vez de con una llave simple, esto porque dentro contiene varios JSON simples, cada uno con una serie de keys y values que usualmente son los mismos entre sí.
    - Este tipo de JSON se transforma a un objeto **JSONArray("")**.
    - Para obtener los **values** del **JSONArray** que usen un **key** en específico se debe utilizar un **bucle for** que recorra el JSON completo usando el método **JSONArray.length()** para limitar la ejecución del bucle y la variable **"i"** del bucle **for** en conjunto con alguno de los siguientes métodos dependiendo del tipo primitivo del **value** asignado a ese **key**:
      - `val variableAuxiliar = JSONArray.get(i) as JSONObject`
      - `variableAuxiliar.getString("nombre_key")`
      - `variableAuxiliar.getInt("nombre_key")`
      - `variableAuxiliar.getLong("nombre_key"), etc.`
    - Si se quiere obtener el **value** de algún **key** que no aparezca en todos los JSONs internos del **JSONArray** se debe usar el método **opt** en vez de **get**:
      - `val variableAuxiliar = JSONArray.get(i) as JSONObject`
      - `variableAuxiliar.optString("nombre_key")`
      - `variableAuxiliar.optInt("nombre_key")`

- `variableAuxiliar.optLong("nombre_key")`, etc.
- **JSON Anidado:** Es una combinación de los dos anteriores y se deben mezclar los métodos anteriores para recorrer y acceder a sus values.
- Ya que todos los tipos de datos en Kotlin son considerados como objetos, hasta los tipos primitivos son descritos como clases, si se quiere solamente declarar un JSON, se debe usar la clase correspondiente dependiendo del tipo de JSON que sea y su valor se pone entre las comillas de la clase.
  - **JSON simple:**
  - `val jsonArr = JSONObject("{"ciudad": "Nueva York", "país": "Estados Unidos"}")`
  - **JSON tipo Array:**
  - `val jsonArr = JSONArray("[ {"ciudad": "Nueva York", "país": "Estados Unidos"}, {"ciudad": "Los Ángeles", "país": "Estados Unidos"}, {"ciudad": "CDMX", "país": "México"} ]")`
- **Clases pertenecientes al paquete model de la carpeta app/java/com.nombreDominio.nombreProyecto:** Estas clases lo único que van a hacer es declarar **variables mutables** que tengan **el mismo nombre** y **tipo primitivo** que **el key y value** de algún elemento específico del JSON, esto para que sus datos se almacenen en esa variable llamada **atributo de la clase** (las variables mutables son declaradas con la palabra reservada **var** y a estas si se les puede modificar o borrar su valor), con tipo primitivo nos referimos a si el dato es un string, int, double, datetime (osea una fecha, objeto Date), etc.

*Se debe crear una **variable de atributo de clase** para cada **key** y **value** perteneciente al JSON que se quiere subir, para que así todos los datos sean almacenados en la base de datos.*

- `val jsonArr = JSONArray("[{"key" : "value"}]")`
  - `val nombre_variable: tipo_primitivo`
- **Instancia de la clase Firebase Firestore:** Para instanciar la clase `FirebaseFirestore` se utiliza el método `FirebaseFirestore.getInstance()` y se almacena en una **variable inmutable** declarada con la palabra reservada **val**, para que no pueda cambiar su valor, ya que después de haber recorrido con un bucle `for` **las variables de atributo de clase** de alguna clase del paquete `model`, se usará el siguiente método dentro del mismo bucle `for` para subir la colección de datos a FireCloud:
 

```
.collection("nombre_coleccion").document().set(objeto_clase_paquete_model)
```
- **Bucles for que recorran las variables de atributo de clase y por medio del objeto FirebaseFirestore suban colecciones de datos a Firebase:** Los **bucles for** lo que harán es ejecutarse el mismo número de veces que el tamaño del JSON que dictó los datos a subir a Firebase, para ello se debe utilizar:
  - Un **objeto JSON** para aplicar el método `.length()` de la clase `JSONArray`, con el fin de recorrer **las variables de atributo de clase** de alguna clase creada en el paquete `model` que recopiló datos de ese mismo JSON que se utilizó para dictar las veces que se ejecuta el bucle.

```
for (i in 0 until objJSONArray.length()){
    //contenido for
}
```

- Una variable auxiliar que use el método `.get()` de la clase `JSONArray` y que reciba como parámetro la variable `i` del bucle `for`, por lo que debe utilizar un objeto `JSON`, se debe usar el mismo objeto que se usó para ejecutar el método que limitó la ejecución del `for`.

```
for (i in 0 until objJSONArray.length()){
    val variable_auxiliar = objJSONArray.get(i) as JSONObject
}
```

- Una instancia de la clase que se quiere recorrer perteneciente al paquete `model`:

```
for (i in 0 until objJSONArray.length()){
    val variable_auxiliar = objJSONArray.get(i) as JSONObject
    val instancia_clase_paquete_model = ClasePaqueteModel()
}
```

*Esta instancia lo que hará es extraer individualmente las variables de atributo de clase utilizando la variable auxiliar y los métodos `get` que permitan extraer cada dato primitivo y metiendo dentro de su paréntesis el nombre que se le quiera dar a la variable de atributo de clase dentro de FireCloud y debe ser igual al nombre de atributo de clase original sino el programa dará error, como lo son: `.getString()`, `.getInt()`, `.getLong()`, etc.*

```
for (i in 0 until objJSONArray.length()){
    val variable_auxiliar = objJSONArray.get(i) as JSONObject
    val instancia_clase_paquete_model = ClasePaqueteModel()
    instancia_clase_paquete_model.variable_de_atributo.getTipoPrimitivo("nombre")
}
```

- La instancia de la clase `FirebaseFirestore` creada previamente se utiliza para que después de haber recorrido con un bucle `for` las variables de atributo de clase de alguna clase del paquete `model` que obtuvieron todos los datos de cada key y value del `JSON`, se utilice el método `.collection()` dentro del mismo bucle `for` para subir los datos recopilados a la base de datos:

```
for (i in 0 until objJSONArray.length()){
    val variable_auxiliar = objJSONArray.get(i) as JSONObject
    val instancia_clase_paquete_model = ClasePaqueteModel()
    instancia_clase_paquete_model.variable_de_atributo.getTipoPrimitivo("nombre")
    instancia_Firebase.collection("nombre_colección").document().set(instancia_clase_paquete_model)
}
```

*Básicamente estas instancias lo que harán es jalar los datos del JSON de las clases creadas en el paquete `model` desde una clase distinta para así subir los datos a FireCloud y crear colecciones de datos y que se puedan ver en la consola. Se debe crear un bucle `for` por cada clase que pertenezca al paquete `model`.*



```
[{"Fam_id":1,"Fam_tipo":"Batientes","Fam_descripcion":"La gama de motores para puertas batientes industriales y comunitarias es adecuada para puertas pesadas, de grandes dimensiones y de mucho tráfico, tanto en entornos industriales como en comunidades de vecinos.", "Fam_imagenurl":"https://www.erreka.com/wp-content/uploads/2020/11/producto-motores-puertas-batientes-comunitarias.jpg"}, {"Fam_id":2,"Fam_tipo":"Corredizas","Fam_descripcion":"Motores adaptados para su instalación en puertas de hasta 4.000 kg. Disponemos de la solución más adecuada para cualquier puerta corredera por grande y pesada que este sea.", "Fam_imagenurl":"https://www.erreka.com/wp-content/uploads/2020/11/producto-motores-puertas-corredoras-industriales-comunitarias.jpg"}, {"Fam_id":3,"Fam_tipo":"Secccionales","Fam_descripcion":"Disponemos de una amplia gama de motores que nos permite automatizar puertas seccionales de cualquier fabricante por muy grande que estas sean.", "Fam_imagenurl":"https://www.erreka.com/wp-content/uploads/2020/11/motores-puertas-secccionales-industriales-comunitarias.jpg"}, {"Fam_id":4,"Fam_tipo":"Enrollable","Fam_descripcion":"Nuestra gama de automáticos es válida para portones de garaje, toldos, claustrales, cubiertas y garajes para particulares y profesionales.", "Fam_imagenurl":"https://www.erreka.com/wp-content/uploads/2020/11/automatismos-toldos-persianas-claustrales-enrollable.jpg"}, {"Fam_id":5,"Fam_tipo":"Barieras","Fam_descripcion":"Barieras automáticas ideales para controlar el acceso y control de los vehículos en zonas comunitarias, aparcamientos, industrias, etc.", "Fam_imagenurl":"https://www.erreka.com/wp-content/uploads/2020/11/barieras-automaticas-img-2.jpg"}, {"Fam_id":6,"Fam_tipo":"Radios","Fam_descripcion":"Disponemos de emisores y receptores para poder abrir fácil y cómodamente a distancia la puerta de tu garaje.", "Fam_imagenurl":"https://www.erreka.com/wp-content/uploads/2020/11/radio-emisores-receptores-mandos-garaje.jpg"}, {"Fam_id":7,"Fam_tipo":"Electrónica","Fam_descripcion":"Nuestra amplia gama de cuadros de maniobra permite automatizar tanto nuestros motores como los de otros fabricantes.", "Fam_imagenurl":"https://www.erreka.com/wp-content/uploads/2020/11/electronica-cuadros-maniobra.jpg"}]
```

The screenshot shows the Android Studio interface. On the left is the Project Navigational Drawer, which includes sections for app, manifests, java, db, model, network, ui, and resources. The 'model' section is currently selected, showing a file named 'familiasSIA.kt'. The code editor on the right contains the following Kotlin code:

```

71 //Para la clase familiasSIA.kt del paquete model
72 val jsonArr = JSONArray(jsonObject.toString())
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100

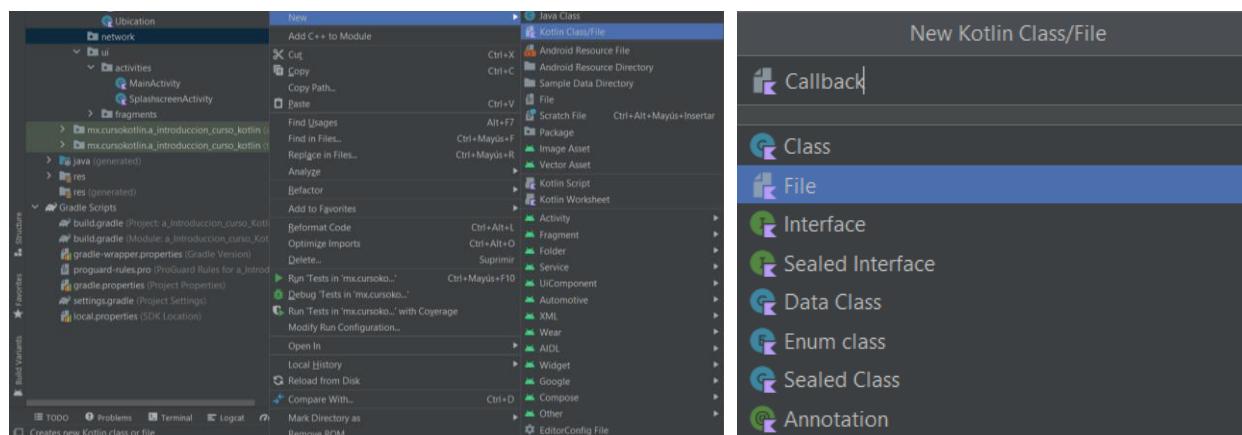
```

The code is a JSON array definition for a family model, with fields like Fam\_id, Fam\_tipo, Fam\_descripcion, Fam\_imagenurl, and Fam\_id. It includes several nested JSON objects for different family types (Batientes, Corredizas, Motorizadas, etc.) with their respective descriptions and URLs.

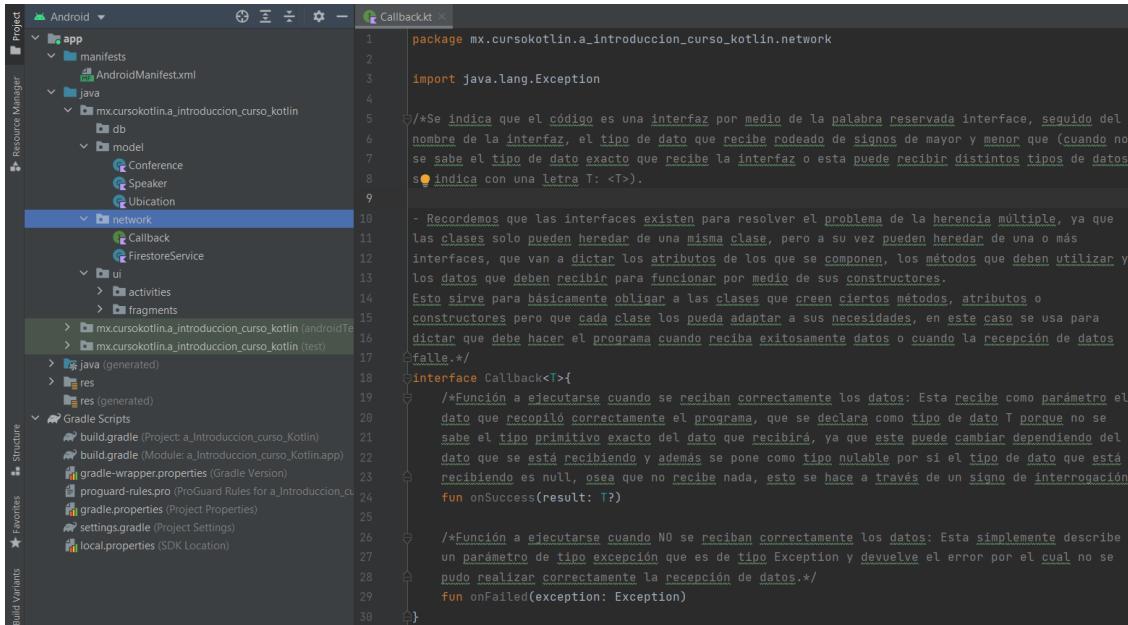
## Extracción de datos de la base de datos FireCloud:

Ya habiendo subido los datos a la base de datos Firecloud de Firebase y contando con el paquete **network**, se debe crear una clase y una interfaz dentro del paquete para que se puedan extraer las colecciones de datos subidos a Firecloud, ambas serán creadas como archivos tipo File de Kotlin y después se indicará que una es una interfaz y la otra es una clase:

- **Interfaz Callback:** Su función es indicar las funciones a ejecutarse cuando se han recibido correcta o incorrectamente los datos de un servicio externo al proyecto de Android Studio, como en este caso lo es la base de datos de Firebase.
  - Para ello se debe crear un archivo llamado **Callback** dentro del paquete **network** que sea de tipo Kotlin File, para luego indicar que es una interfaz en el código por medio de la palabra reservada **interface**, seguido del nombre de la interfaz, el tipo de dato que recibe rodeado de signos de mayor y menor que (cuando no se sabe el tipo de dato exacto que recibe la interfaz o esta puede recibir distintos tipos de datos, se indica con una letra T: <T>) y llaves de apertura y cierre.



- Recordemos que las interfaces existen para resolver el problema de la **herencia múltiple**, ya que las clases solo pueden heredar de una misma clase, pero a su vez pueden heredar de una o más interfaces, que van a dictar **los atributos de los que se componen, los métodos que deben utilizar y los datos que deben recibir para funcionar por medio de sus constructores**. Esto sirve para básicamente obligar a las clases que crean ciertos métodos, atributos o constructores pero que cada clase los pueda adaptar a sus necesidades, en este caso se usa para dictar que debe hacer el programa cuando reciba exitosamente datos o cuando la recepción de datos falle.
  - Las funciones descritas en la interfaz no deben definir su comportamiento de forma explícita, solo deben indicar su nombre y el nombre del parámetro o parámetros que reciben con todo y su tipo de dato: **fun nombreFunción(nombre\_parámetro: tipo\_de\_dato)**
  - **Función a ejecutarse cuando se reciban correctamente los datos:** Esta recibe como parámetro el dato que recopiló correctamente el programa, que se declara como tipo de dato **T** porque no se sabe el tipo primitivo exacto del dato que recibirá, ya que este puede cambiar dependiendo del dato que se está recibiendo y además se pone como tipo **nutable** por si el tipo de dato que está recibiendo es **null**, osea que no recibe nada, esto se hace a través de un signo de interrogación (**?**).
    - **fun onSuccess(nombre\_parámetro: T?)**
  - **Función a ejecutarse cuando NO se reciban correctamente los datos:** Esta simplemente describe un parámetro de tipo excepción que es de tipo **Exception** y devuelve el error por el cual no se pudo realizar correctamente la recepción de datos.
    - **fun onFailed(nombre\_parámetro: Exception)**



The screenshot shows the Android Studio interface with the Project and Structure tabs selected. The project structure is visible on the left, showing modules like app, manifests, java, and network. The code editor on the right contains the following Java code for an interface named Callback:

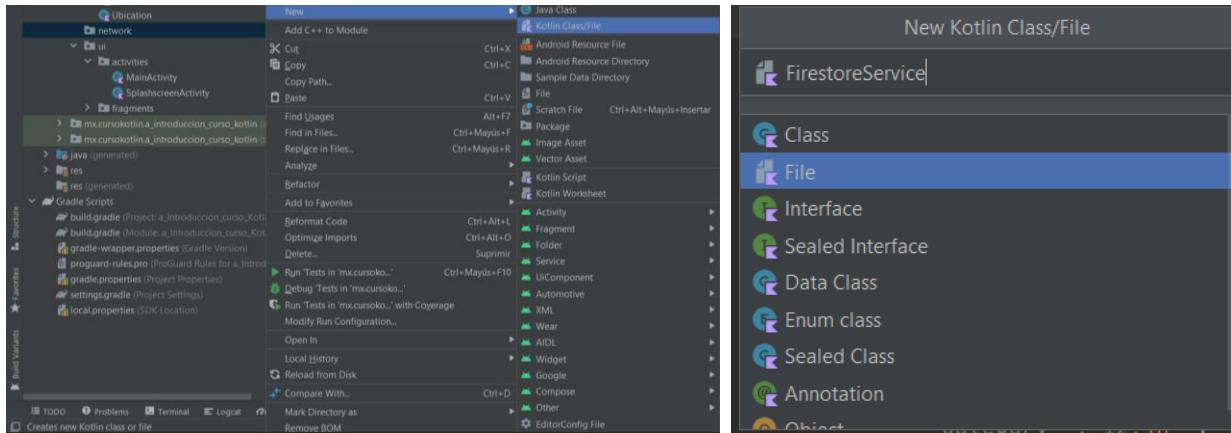
```

1 package mx.curso kotlin.a_introduccion_curso_kotlin.network
2
3 import java.lang.Exception
4
5 /*Se indica que el código es una interfaz por medio de la palabra reservada interface, seguido del nombre de la interfaz, el tipo de dato que recibe rodeado de signos de mayor y menor que (cuando no se sabe el tipo de dato exacto que recibe la interfaz o esta puede recibir distintos tipos de datos, se indica con una letra T: <T>).
6
7 - Recordemos que las interfaces existen para resolver el problema de la herencia múltiple, ya que las clases solo pueden heredar de una misma clase, pero a su vez pueden heredar de una o más interfaces, que van a dictar los atributos de los que se componen, los métodos que deben utilizar y los datos que deben recibir para funcionar por medio de sus constructores.
8 Esto sirve para básicamente obligar a las clases que crean ciertos métodos, atributos o constructores pero que cada clase los pueda adaptar a sus necesidades, en este caso se usa para dictar que debe hacer el programa cuando reciba exitosamente datos o cuando la recepción de datos falle.*/
9
10 interface Callback<T>{
11     /*Función a ejecutarse cuando se reciban correctamente los datos: Esta recibe como parámetro el dato que recopiló correctamente el programa, que se declara como tipo de dato T porque no se sabe el tipo primitivo exacto del dato que recibirá, ya que este puede cambiar dependiendo del dato que se está recibiendo y además se pone como tipo nutable por si el tipo de dato que está recibiendo es null, osea que no recibe nada, esto se hace a través de un signo de interrogación.*/
12     void onSuccess(result: T?)
13
14     /*Función a ejecutarse cuando NO se reciban correctamente los datos: Esta simplemente describe un parámetro de tipo excepción que es de tipo Exception y devuelve el error por el cual no se pudo realizar correctamente la recepción de datos.*/
15     void onFailed(exception: Exception)
16 }
17
18
19
20
21
22
23
24
25
26
27
28
29
30

```

- **Clase FirestoreService:** Su función es realizar la conexión directa con la base de datos y configurar la conexión para que se puedan obtener los datos de la base de datos Firecloud aun estando fuera de línea.

- Para ello se debe crear un archivo llamado FirestoreService dentro del paquete **network** que sea de tipo Kotlin File, para luego indicar que es una clase por medio de la palabra reservada **class**, seguido del nombre de la clase y llaves de apertura y cierre.



- La clase **FirestoreService** debe:
  - **Establecer la configuración de Firestore para que se puedan obtener los datos de la base de datos aun estando fuera de línea:** Esto se hace por medio de dos variables inmutables declaradas con la palabra reservada **val**, una que instancia la clase **FirebaseFirestore** y la otra que instancia la clase **FirebaseFirestoreSettings** para que al juntarse y colocarse dentro de un inicializador **init{ }** (que es básicamente el constructor de la clase que se ejecuta al inicio de esta), configuren la conexión como se desea.

```

package mx.curso kotlin.a_introduccion_curso_kotlin.network

import com.google.firebase.firestore.FirebaseFirestore
import com.google.firebase.firestore.FirebaseFirestoreSettings

class FirestoreService {
    /*Conexión con la base de datos Firecloud: Se hace a través de una instancia de la clase
    FirebaseFirestore creada con el método .getInstance() para establecer una conexión directa
    hacia la base de datos de Firebase.*/
    val firebaseFirestore = FirebaseFirestore.getInstance()

    /*Configuración de la base de datos Firecloud: Se ejecuta esta configuración de Firebase para
    obtener los datos de la base de datos aun estando fuera de linea o descargarlos, esto se hace a
    través de una instancia de la clase FirebaseFirestoreSettings en conjunto con el método
    .Builder().setPersistenceEnabled().build(), pasándole como parámetro el valor de true.*/
    val settings = FirebaseFirestoreSettings.Builder().setPersistenceEnabled(true).build()

    /*Incializador init: Este es exclusivo de Kotlin y es más o menos como un constructor que
    funciona como un método main, ejecutándose justo cuando inicia esta clase.*/
    init {
        /*Dentro del inicializador se accederá al atributo firestoreSettings del objeto
        firebaseFirestore y se le asignará el objeto settings que instanció la clase
        FirebaseFirestoreSettings y ejecutó el método setPersistenceEnabled() con un valor true para
        poder obtener los datos de la base de datos aun estando fuera de linea.*/
        firebaseFirestore.firestoreSettings = settings
    }
}

```

- **Conectarse con las clases del paquete model para recibir los datos:** Se declara una función dentro de la clase **FirestoreService** por cada clase declarada en el paquete **model**, estas funciones reciben como parámetro un callback, que pertenece a la interfaz **Callback** declarada dentro del mismo paquete **network**, ya que esta dicta que es lo que tiene que hacer el programa cuando reciba

correcta o incorrectamente los datos y recibe a su vez una lista de objetos que pertenezcan a la clase que describa el conjunto de datos, osea cada **clase del paquete model** que describa **las variables de atributo de clase**.

```
fun nombreFunción(nombre_callback: Callback<List<nombre_clase_paquete_model>>){ }
```

- Dentro de las funciones usadas para recibir los datos se utiliza el método `.collection("nombre_colección")` aplicado a la instancia de la clase **FirebaseFirestore**, donde se indica dentro de su paréntesis el nombre de la colección de datos que se quiere obtener.
  - Para evitar errores, se debe crear constantes fuera de la clase que indiquen el nombre de las colecciones que se quieren extraer de la base de datos:
    - **CONSTANTES:** Estas deben ir definidas fuera de la función o clase y antes de ellas, se declara con la palabra reservada `const val` y su valor no debe cambiar nunca, su sintaxis es la siguiente:
      - `const val nombre_constante: tipo_de_dato = valor`
- Después de haber obtenido la colección de la base de datos se puede utilizar el método `.orderBy("nombre_variable_de_atributo_de_clase")` en una línea de código debajo de donde se obtuvo la colección para ordenar los datos recopilados en función de .
- Posteriormente se aplica el método `.get()` para introducir los datos obtenidos de la colección al programa.
- Finalmente se utilizan los siguientes elementos para crear todos los objetos que contengan los datos recopilados para que se puedan mostrar dentro del programa:
  - **Método `.addOnSuccessListener{variable_de_ingreso -> }`:** Se aplica una línea debajo del código anterior y recibe una variable de ingreso con el resultado de los datos recopilados de la colección indicada en el método `FirebaseFirestore.collection("nombre_colección")`.
  - **Bucle for:** Se declara dentro del método `.addOnSuccessListener{}` para que recorra los datos recopilados en la `variable_de_ingreso` y los mande hacia la interfaz **Callback** con el fin de poderlos ingresar a los layouts de la aplicación
    - Se crea una **variable\_auxiliar** dentro del **bucle for** que aplique el método `.toObjects()` a la `variable_de_ingreso` para que cree las instancias necesarias de la **clase del paquete model** de donde se extrajeron los datos:

```
val variable_auxiliar = variable_de_ingreso.toObjects(clase_paquete_model::class.java)
```

- Despues dentro del mismo **bucle for** se aplica la función `onSuccess()` perteneciente a la interfaz `callback` y se le pasa como parámetro la

**variable\_auxiliar**, para ello se debe usar el parámetro de tipo callback **nombre\_callback** declarado al iniciar la función:

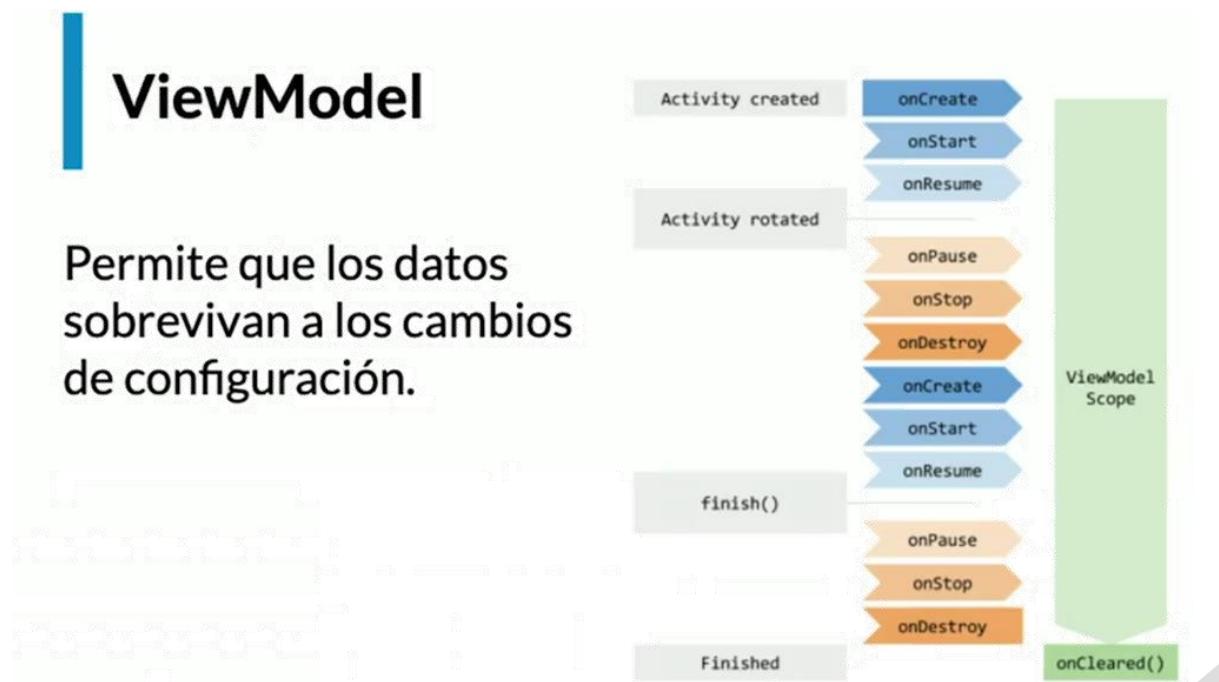
```
nombre_callback.onSuccess(variable_auxiliar)
```

- Finalmente se pone la instrucción **break** para terminar la ejecución del bucle for cuando se termine de crear las instancias de la **clase del paquete model** que muestren los distintos datos recopilados de la interfaz y termine la ejecución de la función.

## Librería Android Jetpack:

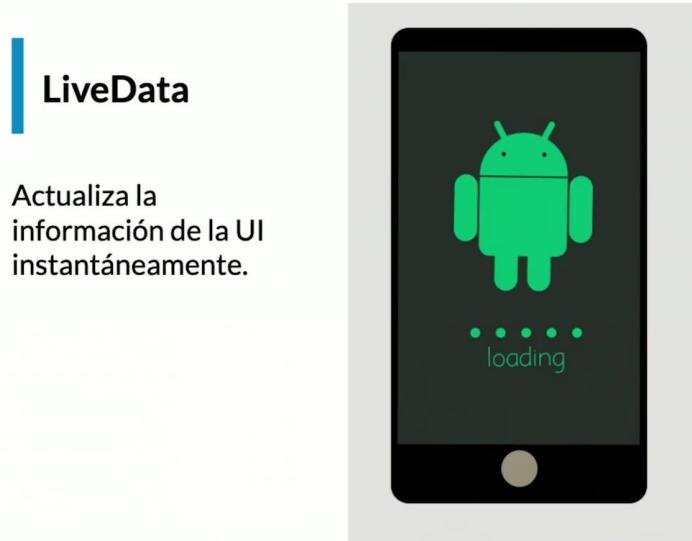
Es una librería que permite extraer datos de la base de datos y mostrarlos en la interfaz, los elementos necesarios para poder realizar esto son los siguientes:

- **ViewModel:** Permite que los datos sobrevivan a cambios de configuración y ciclo de vida.

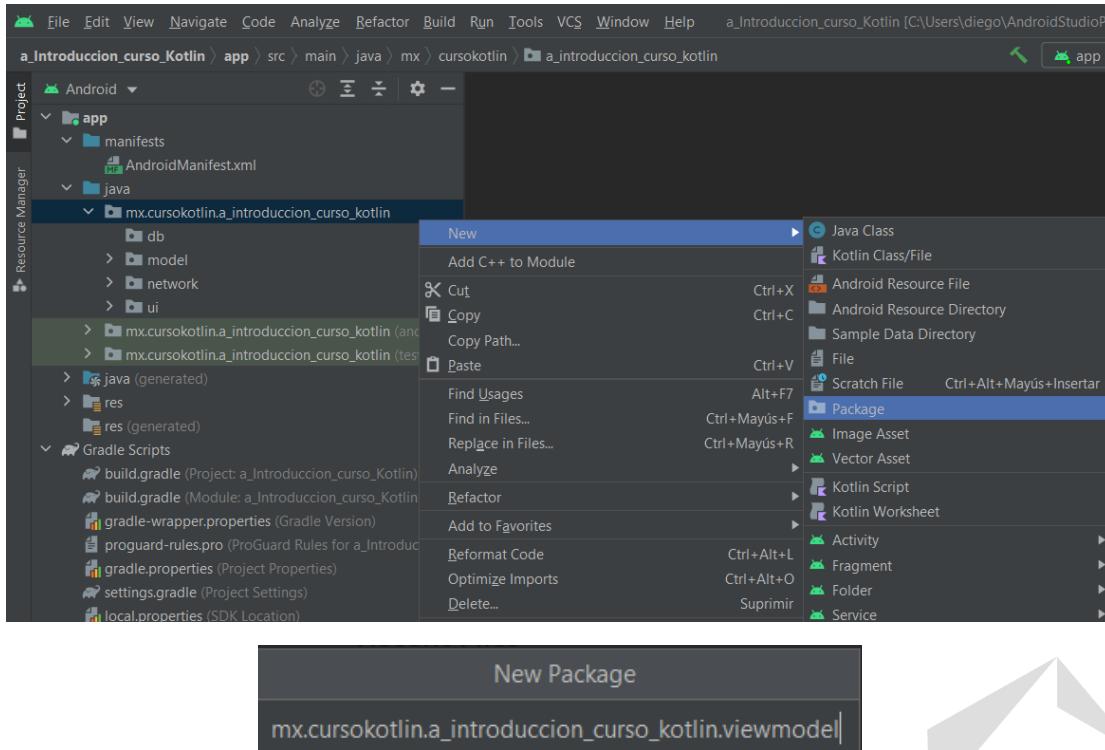


Lo que puede pasar en la aplicación si no se usa el elemento ViewModel de la librería Jetpack es que si ocurre un cambio de pantalla, ya sea porque se dio clic para pasar de una pantalla a otra o si se rotó la pantalla del celular, los datos se pueden borrar y perder ya que se ejecutan los métodos **onCreate()**, **onStart()** y **onResume()** del ciclo de vida de la actividad o fragmento por ocurrir el evento de rotar la pantalla o dar clic para pasar de una a otra, mientras que con el elemento ViewModel se borran los datos nadamás cuando se ejecute el método **onCleared()** que está completamente aparte del ciclo de vida de la interfaz.

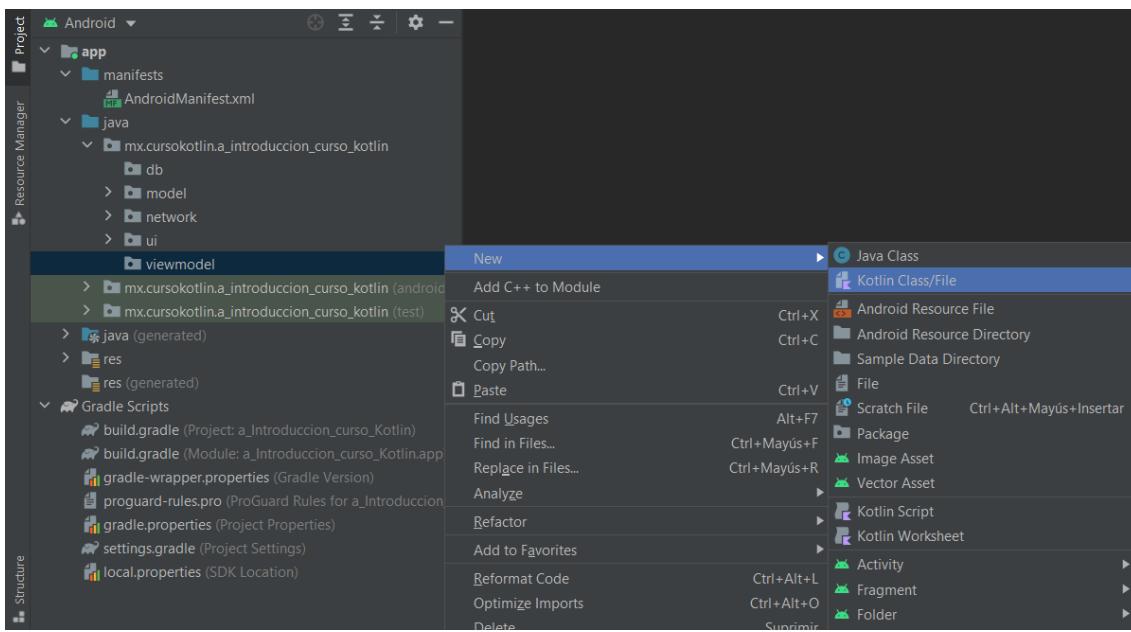
- **LiveData**: Realiza la actualización de los datos en la interfaz de forma automática, sin tener la necesidad de tener que llamar manualmente una función que actualice los datos cuando se adicione o elimine alguno.



Ya habiendo creado las clases del paquete **model** y **network** para que se puedan subir y luego extraer las colecciones de datos subidos a Firecloud, se debe crear el paquete **viewmodel** para crear el modelo de comunicación entre los datos recopilados de la base de datos y las actividades y fragmentos de los layouts de la aplicación.



La acción de establecer la comunicación entre los datos recopilados de Firebase y las pantallas de la aplicación se realiza a través de dos clases Kotlin que se encuentran dentro del paquete **viewmodel**:



- **Clase ViewModel:** Para poder comunicar las colecciones de datos recopilados con las pantallas de la aplicación se utiliza lo siguiente:

- **Instancia de la clase FirestoreService:** Sirve para extraer la colección de datos recopilados con todo y la configuración conexión para que se puedan obtener los datos de la base de datos Firecloud aun estando fuera de línea, se debe declarar como variable inmutable, donde no pueden ser modificada ni borrado su valor y debe ser declarada con la palabra reservada **val**.

```
val instancia_firestoreService = FirestoreService()
```

- **Lista de tipo primitivo MutableLiveData que instancia la misma clase MutableLiveData:** El tipo primitivo MutableLiveData lo que hace es fusionar la funcionalidad de los elementos ViewModel con LiveData, esto lo que hará es dejar que los datos sean actualizados de forma automática, permitiendo que los datos recopilados sobrevivan a cambios de configuración y cambios de ciclo de vida en las actividades y fragmentos de la aplicación, el tipo es además de tipo mutable por lo que se puede cambiar y borrar el valor de la variable y debe ser declarado con la palabra reservada **var**.

```
var lista_datos_recopilados : MutableLiveData<List<>>
```

- El tipo de dato que recibirá la variable de tipo **MutableLiveData** es una lista, descrita con el tipo primitivo **List<>**, que es parecido a un array porque son una serie de valores almacenados en una misma variable, pero cuenta con otros métodos exclusivos que no se pueden aplicar a arrays.

```
var lista_datos_recopilados : MutableLiveData<List<>>
```

- El tipo de dato que recibirá la lista es de la **clase perteneciente al paquete model** a la cual pertenece la colección de datos que se quiere conectar con los **layouts** de la aplicación y al final igual se indica que es instancia de la clase **MutableLiveData**.

```
var lista_datos_recopilados : MutableLiveData<List<clase_paquete_model>> = MutableLiveData()
```

- **Variable booleana para actualizar los datos en la interfaz:** Utiliza el tipo primitivo MutableLiveData pero recibe como tipo primitivo un valor booleano y sirve para poder cambiar su valor dentro de las funciones de esta misma clase **ViewModel** para indicar si un proceso de recopilación de datos ha terminado, para así dejar de mostrar la pantalla de carga que se colocó sobre el view que muestra el listado de elementos obtenidos de la base de datos Firebase.

```
var actualizacionDatos : MutableLiveData<Boolean>
```

- **Función refresh:** Esta función puede tener este nombre o cualquier otro, pero lo importante es que se encarga de correr otra función que se encuentra dentro de la misma clase para que ejecute una función de la clase **FirebaseService** que recopiló los datos de alguna clase perteneciente al paquete model.
- **Función getFromFirebase:** Esta función es llamada desde la función refresh y lo que hace es utilizar la **Instancia de la clase FirebaseService** para ejecutar alguna de sus funciones que se conectan con las clases del paquete model para recibir sus datos, como estas funciones reciben como parámetro un **Callback**, se le debe mandar como parámetro un objeto que tenga de tipo primitivo una lista, descrita con el tipo primitivo **List<>**, que a su vez recibirá como tipo de dato primitivo la **clase perteneciente al paquete model** de donde quiere extraer la colección de datos.

```
fun getFromFirebase(){  
    instancia_firestoreService.funcionFirestoreService(object: Callback<List<clase_paquete_model>> {  
        //Implementación de los métodos del callback  
    })  
}
```

- **Implementación de los métodos de la interfaz callback:** Como estos métodos de la clase **FirebaseService** reciben como parámetro un callback, se deben ejecutar los métodos del callback, esto se hace a través de la instrucción **override fun** seguido del nombre de las funciones declaradas en la interfaz, en este punto es donde **la interfaz obliga a la clase a crear ciertos métodos, atributos o constructores, pero adaptándolos a sus necesidades**, por lo que se deben crear los métodos:

- **onSuccess(nombre\_parámetro: T?):**

- Dentro de la función se utilizará el método **.postValue()** aplicado a la **Lista de tipo primitivo MutableLiveData**, esta recibirá como parámetro lo que viene en el parámetro de la función **onSuccess()**. El método sirve para actualizar los datos de un objeto **MutableLiveData**, a este se le pasa como parámetro la lista de valores que se quiere actualizar.
- Además, se ejecuta una función que cambie el valor de la **variable booleana para actualizar los datos en la interfaz** a true y que el programa sepa que ya se dejó de correr un proceso de

recopilación de datos, por lo que puede remover la pantalla de carga de los layouts donde se muestren estos datos.

- **onFailed(nombre\_parámetro: Exception):**
  - Solamente se ejecuta una función que cambie el valor de la **variable booleana para actualizar los datos en la interfaz** a true.

## Referencias:

Platzi, Gustavo Lizárraga, “Curso de Kotlin para Android”, [Online], Available: <https://platzi.com/clases/1836-kotlin-android/26986-por-que-desarrollar-para-android-usando-kotlin/>

Platzi, Juan Guillermo Gómez Torres, “Curso de Firebase 5: Cloud Functions”, [Online], Available: <https://platzi.com/clases/1472-firebase-cloud/16629-introduccion5134/>

