Firebase (Anexo)

Descargar estos apuntes pdf o html

Índice

Introducción

- **Persistencia en la nube con Firebase**
  - Documentación oficial: **Firebase documentation**
  - Video Tutorial (Inglés): **Firebase**
  - Video Tutorial (Castellano): **DevExpert**
  - Video Tutorial (Castellano): **AristiDevs**
  - Video Tutorial (Castellano): **AristiDevs**

La plataforma Firebase proporciona múltiples herramientas para el almacenamiento, consulta y gestión de datos, así como otros servicios en la nube. Algunas de las más destacadas incluyen:

- **Cloud Firestore**: Base de datos NoSQL en la nube con sincronización en tiempo real.
- **Realtime Database**: Base de datos en tiempo real que permite actualizaciones instantáneas entre clientes.
- **Firebase Authentication**: Servicio para gestionar autenticación de usuarios con correo, Google, Facebook, etc.
- **Firebase Cloud Storage**: Almacenamiento de archivos en la nube, ideal para imágenes, videos y otros documentos.
- **Firebase Cloud Messaging (FCM)**: Servicio para enviar notificaciones push y mensajes a dispositivos.
- **Firebase Remote Config**: Permite cambiar la configuración de la app de forma remota sin necesidad de actualizarla.
- **Firebase Crashlytics**: Herramienta para la detección y análisis de errores en tiempo real.
- **Firebase Analytics**: Plataforma para el análisis del comportamiento de los usuarios dentro de la aplicación.

Firebase proporciona un ecosistema completo para desarrollar aplicaciones escalables sin necesidad de administrar servidores. 🚀

> **Nota**
>
> Existen otras opciones de terceros como **Supabase** (alternativa basada en SQL con PostgreSQL) o **Appwrite** (un backend ligero y multiplataforma con soporte para Flutter, Kotlin Multiplatform y más), pero no están tan integradas con el ecosistema de Google como Firebase. Además, Firebase ofrece un plan gratuito con ciertas limitaciones, lo que lo convierte en una opción atractiva para proyectos pequeños o en fase de prueba.

Nosotros nos vamos a centrar en Firebase, aunque no entraremos en el diseño de bases de datos NoSQL ni en el uso avanzado de esta tecnología. Para conocer toda la funcionalidad de Firebase, se recomienda consultar la documentación oficial. El problema de trabajar directamente con esta API es que puede requerir un manejo detallado de la sincronización de datos y manejo de la red, lo que puede llevar a errores si no se gestiona correctamente.

Por este motivo, Firebase ofrece diversas herramientas y servicios como Firebase Authentication, Firebase Storage y Firebase Cloud Messaging, que funcionan como una capa de abstracción y simplifican el proceso de implementación, optimizando la gestión de datos, la autenticación de usuarios, el almacenamiento de archivos y las notificaciones push.

Modificando la Agenda para usar Firestore

> **Proyecto**
>
> Puedes descargar el proyecto de la agenda con todo el código visto en el tema desde el siguiente enlace **AgendaFirebase**

Creando un proyecto de Firebase

Lo primero que debemos hacer, es crear una cuenta de Google e iniciar sesión en la **consola** de Firebase. Antes de comenzar a trabajar en el proyecto de Android, es necesario configurar un proyecto de Firebase en la consola. Una vez que hayas creado y configurado el proyecto en Firebase, podrás ver los proyectos que tienes y empezar a integrarlos con tu aplicación Android.

> **Aviso**
>
> Firebase impone un límite en la cantidad de proyectos que puedes crear. Si alcanzas este límite, tendrás que enviar una solicitud para pedir una cantidad específica de proyectos, indicando el motivo. En caso de que necesites proyectos ilimitados, puedes solicitar la versión Spark, explicando las razones por las que debería ser concedida.

Los pasos para **crear un proyecto en Firebase** son los siguientes:

1. Indicamos el **nombre del proyecto** por ejemplo ***agenda***.
2. Habilitamos Google Analytics en nuestro proyecto.
3. Elegimos la cuenta por defecto de Google Analytics.
4. Hacemos clic en "***Crear proyecto***".

Ahora, los pasos para configurar un proyecto de Android en Firebase son los siguientes:

1. Registrar la aplicación Android pulsando sobre el **botón con el icono de Android**:
   - Introduce el **nombre del paquete** de tu aplicación Android.
     Este se encuentra en El archivo `build.gradle.kts` del módulo `app`.

```
android {
    namespace = "com.pmdm.agenda"
    compileSdk = 34
    ...
}
```

- Si es necesario, puedes agregar el SHA-1 de la firma de tu app (esto es importante si vas a usar servicios como Firebase Authentication o Firebase Dynamic Links).
- Haz **clic en Registrar** la aplicación.

2. Descargar el archivo `google-services.json`:
   - Una vez registrada la aplicación, se te pedirá descargar el archivo `google-services.json`.
   - Descarga este archivo y sigue las instrucciones proporcionadas en el sitio web.

> **Aviso**
>
> Es importante no cambiar el nombre del archivo `google-services.json` ya que, si lo haces, podrías tener problemas al conectarte con Firebase. Asegúrate de colocarlo en la carpeta `app` dentro de tu proyecto. Además, cada vez que añadas un nuevo servicio en tu proyecto de Firebase, deberás revisar este archivo y descargarlo por si ha cambiado algo en la configuración.

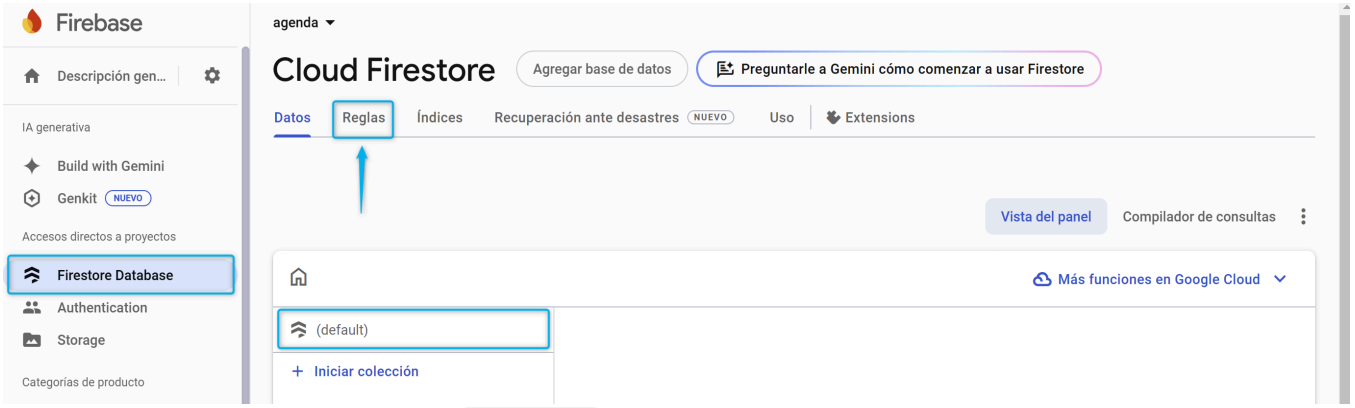
Activación del servicio de Firestore

- **Documentación Oficial**
- **API de Firestore en Kotlin**

Es una BD NoSQL flexible y escalable para aplicaciones móviles, web y servidores, que organiza los datos en **documentos** y **colecciones**. Ofrece sincronización en tiempo real y soporte sin conexión.

Una vez hemos accedido a nuestro proyecto:

- Marcamos la opción **Cloud Firestore** 1 .
- Nos aparecerá la opción 2 de firestore, y al pulsar seleccionamos **Agregar base de datos** dejamos la BD por defecto `(default)` que es la gratuita seleccionando como ubicación por ejemplo `europe3`. Firebase proveerá en ese momento los recursos necesarios para que la BD funcione correctamente.
- En 3 y 4 podremos acceder a la configuración de nuestro proyecto y desde la misma podremos volver el archivo de configuración `google-services.json`.
- Una vez creada nos aparecerá la BD vacía y deberemos configurar las reglas de acceso en las pestaña reglas. I



- Deberemos pondremos la regla `if true;` para no tener ningún tipo de restricción en la BD ya que no estamos utilizando ningún tipo de autenticación.

```
rules_version = '2';

service cloud.firestore {
    match /databases/{database}/documents {
        match /{document=**} {
6           allow read, write: if true;
        }
    }
}
```

Añadiendo permisos en el `manifest.xml`

Puesto que Firestore es una BD en la nube, vamos a necesitar permisos de internet. Pare ello, en el archivo `AndroidManifest.xml` deberemos añadir el permiso de acceso a internet para que el servicio pueda acceder al API.

```
<manifest ...>
    ...
3   <uses-permission android:name="android.permission.INTERNET"/>
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
5   <uses-permission android:name="android.permission.ACCESS_WIFI_STATE"/>
    ...
    <application ...>
        <!-- Permitir tráfico http en lugar de https -->
9       android:usesCleartextTraffic="true"
        ...
    </application>
</manifest>
```

Dependencias y Plugins de en Gradle

> **Nota**
>
> Puede que todas las dependencias relacionadas con **kotlinx-serialization-json** las tengas ya añadidas en tu proyecto porque se usan en la nueva navegación segura de Jetpack Compose. Si es así, fíjate bien pues no es necesario añadirlas de nuevo.

En el catálogo de versiones `lib.versions.toml` deberemos comprobar que hemos definido tener:

```
# Recuerda además que las entradas van en una sola línea.
# Algunas se han indentado para que se vean mejor.

[versions]
googleServices = "4.4.2"
firebaseFirestoreKtx = "25.1.1"
# Solo si no está ya incluido
kotlinxSerializationJson = "1.7.3"

[libraries]
google-firebase-bom = {
    group = "com.google.firebase", name = "firebase-bom", version.ref = "firebase"
}
firebase-firestore-ktx = {
    group = "com.google.firebase", name = "firebase-firestore-ktx",
    version.ref = "firebaseFirestoreKtx"
}
# Solo si no está ya incluido
kotlinx-serializarion-json = {
    group = "org.jetbrains.kotlinx", name = "kotlinx-serialization-json",
    version.ref = "kotlinxSerializationJson"
}

[plugins]
google-services = { id = "com.google.gms.google-services", version.ref = "googleServices" }
# Solo si no está ya incluido
kotlinx-serialization = {
    id = "org.jetbrains.kotlin.plugin.serialization", version.ref = "kotlin"
}
```

En el `build.gradle.kts` raíz del proyecto añadiremos el siguiente plugin:

```
plugins {
    alias(libs.plugins.google.services) apply false
    // Solo si no está ya incluido
    alias(libs.plugins.kotlinx.serialization) apply false
}
```

En el `build.gradle.kts` del **módulo de la aplicación** (app) añadiremos:

```
plugins {
    alias(libs.plugins.google.services)
    // Solo si no está ya incluido
    alias(libs.plugins.kotlinx.serialization)
}
...
dependencies {
    implementation(platform(libs.google.firebase.bom))
}
```

Modificando el proyecto

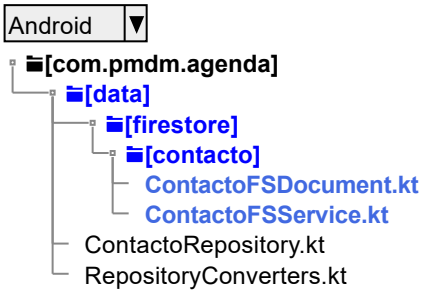
Definiendo el proveedor de Firestore con Dagger-Hilt

Definiremos el proveedor de Firestore en el archivo `AppModule.kt` que me creará una instancia de Firestore de acuerdo a la configuración de mi proyecto definida en el archivo `google-services.json` y me la inyectará con **Dagger-Hilt**.

```
@Module
@InstallIn(SingletonComponent::class)
class AppModule {
    ...
    @Singleton
    @Provides
    fun provideFirestore(): FirebaseFirestore = Firebase.firestore
}
```

Definiendo las clases para operar con Firestore

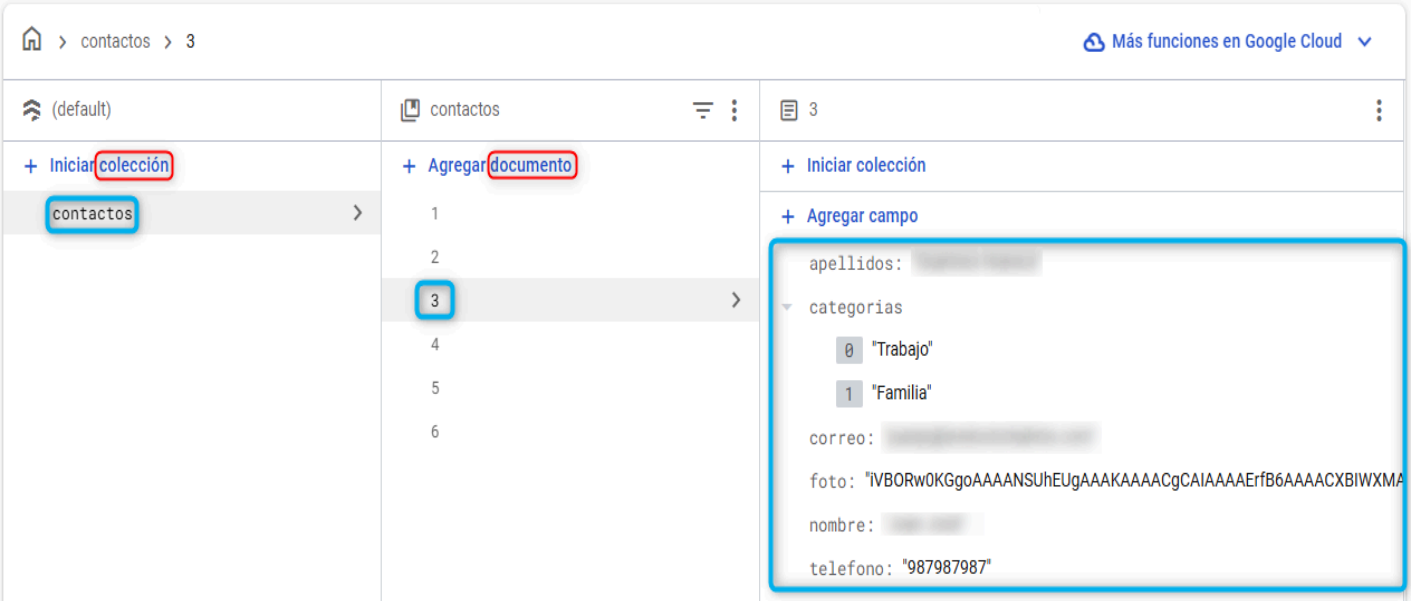
Definiremos el **paquete** `data.firestore.contacto` en nuestro proyecto. El documento a almacenar en Firestore y el servicio que hara las veces de DAO y me proporcionará las operaciones CRUD que usamos en nuestro repositorio.



Primero definiremos la clase que modelizará el documento a almacenar en Firestore. En este caso, será un **Contacto** y que definiremos en el fichero `ContactoFSDocument.kt`.

Para ello, tenemos que entender primero **cómo se almacenan los datos en Firestore**.

Nuestro modelo será la **BD** `(default)` con la **colección** `contactos` y cada **documento** dentro de la misma tendrá un **id** que se tratará como cadena de texto y se recomienda que sea un UUID. En nuestro caso será el id que definimos en el '*mock*' por lo que nos debería quedar algo similar a esto:



A la hora de **agregar o crear un documento** ejecutaremos algo similar a:

```
Firebase.firestore.collection("contactos").document("3").set(contacto)
```

Donde el método document nos devolverá una **referencia al documento** (**DocumentReference**) con el id `"3"` dentro de la colección `contactos` o **lo creará si no existe** devolviendo la referencia al nuevo documento. Si no pasamos el id, Firestore lo generará automáticamente y lo podremos obtener a través del objeto `DocumentReference` que devuelve document.

Definiendo el documento en Kotlin

Ahora ya podemos definir el **objeto personalizado** que modelizará el documento a almacenar en Firestore dentro de `ContactoFSDocument.kt` siguiendo las siguientes recomendaciones:

1. Al crear objetos para Firestore, es necesario inicializarlos con valores por defecto; de lo contrario, se generará un error.
2. Si usamos '*Hash Maps*', solo podrán usar claves de tipo string.
3. Si usas un campo en inglés, como por ejemplo `isBlocked`, es recomendable usar `blocked` en su lugar, ya que obtendrmos errores si no usamos la anotación `@field:JvmField` sobre el campo.

```kotlin
data class ContactoFSDocument(
    // Excluimos el id pues se almacenará a parte de los datos del documento.
    @get:Exclude
    val id: Int = 0,
    val nombre: String = "",
    val apellidos: String = "",
    val foto: String? = null,
    val correo: String = "",
    val telefono: String = "",
    val categorias: List<String> = emptyList()
)
```

Fíjate que podemos guardar las categorías como una lista de cadenas de texto. Esto nos permitirá realizar consultas más avanzadas en Firestore, como buscar todos los contactos que pertenecen a una categoría específica.

Definiendo los '*mapeos*' al modelo

Una vez tenemos modelado el documento en `RepositoryConverters.kt` definiremos el mapeo del documento a nuestro modelo `Contacto` y viceversa.

```
private fun List<String>.toEnumSetCategorias(): EnumSet<Contacto.Categorias> =
    this.mapNotNull { categoria ->
        runCatching { Contacto.Categorias.valueOf(categoria) }.getOrNull()
    }
    .takeIf { it.isNotEmpty() }
    ?.let { EnumSet.copyOf(it) }
    ?: EnumSet.noneOf(Contacto.Categorias::class.java)


fun ContactoFSDocument.toContacto() = Contacto(
    id = id,
    nombre = nombre,
    apellidos = apellidos,
    foto = foto,
    correo = correo,
    telefono = telefono,
    categorias = categorias.toEnumSetCategorias()
)

fun Contacto.toContactoFSDocument() = ContactoFSDocument(
    id = id,
    nombre = nombre,
    apellidos = apellidos,
    foto = foto,
    correo = correo,
    telefono = telefono,
    categorias = categorias.map { c -> c.name }
)
```

Definiendo el servicio de Firestore con el CRUD

Una vez tenemos los '*mapeos*' entre `ContactoFSDocumento` y `Contacto` definiremos `ContactoFSService.kt` donde definiremos las operaciones CRUD que necesitamos para interactuar con Firestore.

Definiremos una excepción personalizada `FirestoreException` que nos permitirá lanzar excepciones hacia el ViewModel.

```
class FirestoreException(message: String) : Exception(message)
```

Posteriormente inyectaremos la la instancia de Firestore creada a través de Dagger-Hilt en el servicio `ContactoFSService` y lo definiremos como un **singleton** para no tener que definir un

provider para `ContactoFSService` en el `AppModule.kt`.

```
@Singleton
class ContactoFSService @Inject constructor(
    private val firestore: FirebaseFirestore
) {
    // Definición de constantes a nivel de clase
    private companion object {
        const val COLLECTION = "contactos"
        const val TAG = "Firestore"
    }

    // Método de extensión privado para centralizar la gestión de errores.
    private fun Exception.gestionaError(mensaje: String) {
        Log.e(TAG, "$mensaje: ${this.localizedMessage}", this)
        throw FirestoreException(mensaje)
    }

    ...
}
```

Definimos las diferentes operaciones como **métodos de suspensión** siguiendo un esquema similar. Por ejemplo, para recuperar **todos los contactos** de la colección.

```
suspend fun get(): List<ContactoFSDocument> {
    return firestore
        // Recuperamos la colección de contactos CollectionReference
        .collection(COLLECTION)
        // Lanzamos la query que hayamos definido a firestore
        // en nuestro caso no hay ningún filtro obteniendo una
        // tarea de tipo Task<QuerySnapshot>
        .get()
        // Ante cualquier error lanzamos nuestra excepción personalizada
        .addOnFailureListener { e ->
            e.gestionaError("Error al obtener los contactos")
        }
        // Función de suspensión que bloquea la corrutina hasta que
        // se resuelva la tarea con la consulta devolviendo un QuerySnapshot
        .await()
        // Una fez obtenido los datos mapeamos cada DocumentSnapshot
        // a un objeto ContactoFSDocument con toObject
        .documents.map { document ->
            document.toObject(ContactoFSDocument::class.java)!!.copy(
                // Como al recuperar el documento no obtenemos un id por el Exclude
                // lo añadimos tras el mapeo obteniéndolo del documento
                id = document.id.toInt()
            )
        }
}
```

El resto de consultas para completar el API del repositorio serán similares a la anterior.

```
    suspend fun get(id: Int): ContactoFSDocument? {
        return firestore
            .collection(COLLECTION)
            .document(id.toString())
            .get()
            .addOnFailureListener { e ->
                e.gestionaError("Error al obtener el contacto $id")
            }
            .await()
            ?.let { document ->
                document.toObject(ContactoFSDocument::class.java)?.copy(id = document.id.toIr
            }
    }

    suspend fun insert(contacto: ContactoFSDocument) {
        firestore
            .collection(COLLECTION)
            .document(contacto.id.toString())
            .set(contacto)
            .addOnFailureListener { e ->
                e.gestionaError("Error al insertar el contacto ${contacto.id}")
            }
            .await()
    }

    suspend fun update(contacto: ContactoFSDocument) {
        firestore
            .collection(COLLECTION)
            .document(contacto.id.toString())
            .set(contacto, SetOptions.merge())
            .addOnFailureListener { e ->
                e.gestionaError("Error al actualizar el contacto ${contacto.id}")
            }
            .await()
    }

    suspend fun delete(id: Int) {
        firestore
            .collection(COLLECTION)
            .document(id.toString())
            .delete()
            .addOnFailureListener { e ->
                e.gestionaError("Error al borrar el contacto $id")
```

```
            }
            .await()
    }

    suspend fun count(): Int {
        val querySnapshot = firestore
            .collection(COLLECTION)
            .get()
            .addOnFailureListener { e ->
                e.gestionaError("Error al obtener el número de contactos")
            }
            .await()
        return querySnapshot.size()
    }
```

Puedes ver más información sobre **cómo realizar consultas en Firestore** en la **documentación oficial**.

Por ejemplo, para obtener los contactos que pertenezcan a un determinado número de categorías para cumplir el filtro podríamos hacer una consulta similar a la siguiente...

```
suspend fun get(
    ascendente: Boolean,
    categorias: List<String>,
): List<ContactoFSDocument> = firestore
    .collection(COLLECTION)
    .orderBy(
        Fields.NOMBRE,
        if (ascendente) Query.Direction.ASCENDING
        else Query.Direction.DESCENDING
    )
    .whereArrayContainsAny(Fields.CATEGORIAS, categorias)
    .get()
    .addOnFailureListener { e ->
        e.gestionaError("Error al obtener contactos por categorías.")
    }
    .await()
    .documents.map { document ->
        document.toObject(ContactoFSDocument::class.java)!!.copy(id = document.id.toInt())
    }
```

> ⚠ **Aviso**
>
> Si estableces muchas condiciones en una consulta, al ejecutar la aplicación podrías encontrarte con un error. En el 😾 Logcat, revisa el mensaje de error, ya que te indicará que necesitas generar índices para optimizar las consultas. Haz clic en el enlace proporcionado en el Logcat y serás dirigido a la consola, donde se generará automáticamente el índice necesario.

Usando el servicio de Firestore en el repositorio

Dentro del paquete `data`, crearemos el archivo `ContactoRepository.kt` como en casos anteriores. Esta clase implementará las operaciones CRUD (Create, Read, Update, Delete) sobre los **objetos del modelo** `Contacto` y se encargará de gestionarlos en la fuente de datos, en este caso, Firestore, de forma transparente.

```
@Singleton
class ContactoRepository @Inject constructor(
    private val dao: ContactoFSService
) {
    suspend fun get(): List<Contacto> = withContext(Dispatchers.IO) {
        dao.get().map { it.toContacto() }.toList()
    }
    ...
}
```