Pràctica introductòria #5 - Persistència remota amb *Firestore* i patró repositori

Projecte Integrat de Software (2024/25)
Facultat de Matemàtiques i Informàtica
Universitat de Barcelona

1 Introducció

En la majoria d'aplicacions, no és necessari ni convenient tenir tots els objectes del domini en memòria alhora, sinó només aquells que han de ser manipulats amb immediatesa. Quan necessitem els objectes, els obtenim de la persistència, els manipulem i els tornem a fer persistir. Això ens permet estalviar memòria i evitar la pèrdua d'informació o inconsistències en cas d'errors que acabin l'execució de l'aplicació. Si la persistència és remota, obtenim – addicionalment – la capacitat de compartir dades entre dispositius i, per tant, entre usuaris de l'aplicació en temps pràcticament real.

Entre diverses possibilitats de bases de dades remotes, optarem per *Firestore* de la plataforma *Firebase* de Google – per la seva facilitat d'ús i gratuïtat (dins d'uns paràmetres d'ús generosos). Podrem consultar/guardar informació de/a la base de dades mitjançant la API de *Firestore* que permet crear tasques i escoltadors per gestionar l'asincronia de les operacions amb la base de dades de manera senzilla. Aquesta implementació concreta quedarà "amagada" darrere d'un *patró repositori*, que el codi client de la capa de domini pugui accedir a la persistència sense haver-ne de conèixer els detalls.

Requeriments

- PI1 - Primers passos amb Android Studio
- PI2 - Activitats i d'altres components fonamentals
- PI3 - Model–Vista–Model de Vista
- PI4 - Vistes reciclables

Objectius

1. Conèixer els conceptes i el funcionament bàsic de la API de *Firestore* (Sec. 2).
2. Veure la implementació concreta d'un *patró repositori* que faci anar la API *Firestore* (Secs. 3-4).
3. Crear la vostra pròpia base de dades *Firestore* i enllaçar-la amb una aplicació d'Android (Ap. A).
4. Pujar dades a la *Firestore* des de fora de l'aplicació (Ap. B) per evitar un *cold start*.

Recursos

Al Campus Virtual, juntament amb aquest guió, us proporcionem:

- `FirestoreRepositoryExample.zip`: projecte d'Android Studio descarregable del campus virtual, per fer el seguiment d'aquesta pràctica introductòra i/o que serveix de codi base resoldre els exercicis proposats.
- `FirestoreUploaderExample.zip`: projecte d'IntelliJ descarregable del campus virtual, per fer el seguiment de l'Ap. B, el qual explica com pujar a la *Firestore* des de fora de l'aplicació a partir d'un fitxer JSON.

2 Conceptes i el funcionament bàsic de la API de *Firestore*

Les *Cloud Firestore* (o, directament, *Firestore*) són bases de dades **remotes** (emmagatzemades al núvol) de la plataforma *Firebase*. Són també bases de dades **no relacionals**, és a dir, no estan estructurades mitjançant taules d'entitat i de relació. En lloc d'això, utilitzen una estructura basada en *col·leccions*, *documents* i *camps*. Si féssim el paral·lelisme amb les bases de dades relacionals, podríem dir que les col·leccions són anàlogues a les taules, els documents – cadascun amb un identificador únic – a les files d'una taula i els camps les columnes de les taules. Ara bé, a diferència d'una base de dades relacional, no és necessari que tots els documents dins d'una col·lecció tinguin el mateix nombre i/o tipus de camps.

Els camps dels documents poden ser de tipus simple (cadenes de text, números, etc) o estructures més complexes (llistes, mapes o fins i tot sub-col·leccions). Els camps d'un document es poden pensar com atributs d'una classe d'un llenguatge orientat a objectes. De fet, és fàcil convertir documents a objectes i viceversa mitjançant mètodes proporcionats per l'API de *Firestore*; cosa que facilita la seva integració en aplicacions desenvolupades en Java (o

Kotlin).

L'API de *Cloud Firestore* es basa en la definició de **tasques asíncrones** que permeten realitzar les típiques operacions **CRUD** (crear, llegir, actualitzar[1] i esborrar dades) i avisar-nos un cop acabada la tasca mitjançant escoltadors. En aquesta pràctica, veurem com definir tasques de creació d'un document, lectura d'un document i lectura de tots els documents d'una col·lecció. Per a la resta d'operacions, podeu consultar la documentació a https://firebase.google.com/docs/firestore/manage-data/add-data.

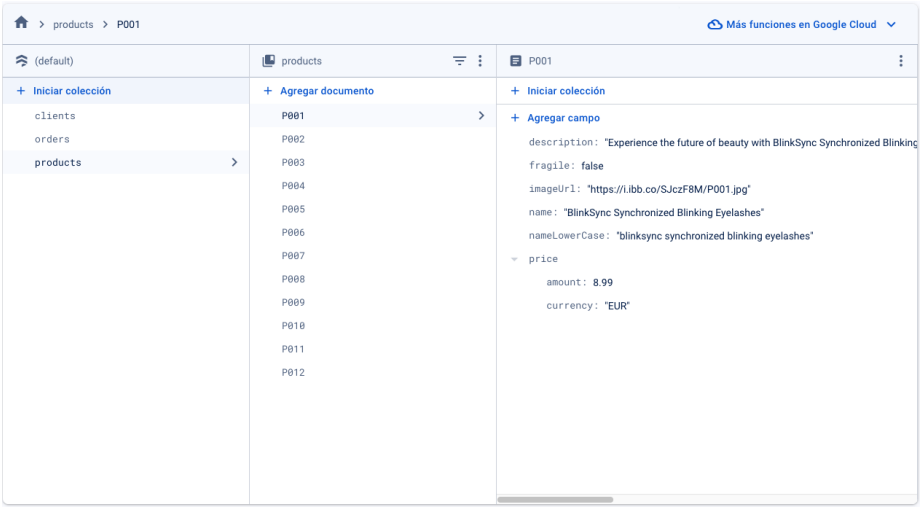

Figure 1: Exemple de base de dades *Firestore* amb tres col·leccions (`clients`, `orders` i `products`). Per la col·lecció de `products`, es mostren dotze documents (`P001`, ..., `P012`). I, pel document, `P1` els seus atributs (`description`, `fragile`, `imageUrl`, etc). Noteu que els camps – en aquest cas `price`, poden contenir altres camps (`amount` i `currency`), de manera jeràrquica.

Limitacions de les bases de dades *Firestore*. En primer lloc, les *Firestore* tenen quotes d'utilització i limitacions per al seu ús gratuït[2]. També són suficients pel projecte que us demanem i molt difícilment les assolireu (a no ser que en feu un mal ús). En segon lloc, les *Firestore* no estan pensades per guardar-hi imatges. Les imatges s'han de guardar externament i a la base de dades guardar-hi una URL. Això és perquè *Firestore* imposa un límit de mida dels documents. que existeixen certes limitacions. En primer lloc, la mida dels documents dins de les col·leccions és de màxim 1 MB. És més que suficient,

[1] Us pot ser útil si planejeu escriptura simultània de documents. Consulteu l'Ap. **??**
[2] https://firebase.google.com/docs/firestore/quotas

tenint en compte que no s'hi ha de guardar contingut multimèdia. Si voleu guardar imatges en algun lloc, haureu d'utilitzar una altra eina. Per exemple, podeu fer servir l'eina `Storage`[3] també de Firebase o qualsevol altre lloc de pujada d'imatges per obtenir una URL. I seria la URL el que guardaríeu a la base de dades *Firestore*.

A més de les *Firestore* i de l'*Storage*, *Firebase* ofereix altres eines per donar suport al desenvolupament d'aplicacions, incloent-hi diversos serveis (autenticació, analítiques, etc.). Podeu trobar més informació a https://firebase.google.com/.

3 Persistència amb el patró repositori

Un **repositori** actua com a mediador entre els objectes del domini i les capa de persistència, funcionant – des del punt de vista del domini – com una col·lecció en memòria d'objectes del domini "persistibles". Els objectes clients interactuen amb la **interfície del repositori** de manera declarativa per afegir, obtenir, actualitzar o eliminar objectes, sense preocupar-se dels mecanismes subjacents de persistència – és a dir, de la seva implementació.

La **implementació del repositori** haurà de coordinar l'accés a les dades per un determinat tipus d'objecte del domini. Pot encarregar-se'n directament o pot, alternativament, delegar en DAOs. En el nostre cas, tenint en compte el tipus de base de dades i la dimensió de les aplicacions que desenvolupem en el context de l'assignatura, no voldrà la pena definir DAOs.

D'altra banda, els repositoris **operen a nivell d'agregat** – si seguim el *patró agregat* del *Domain-Driven Design* (DDD). Això facilita mantenir les fronteres de consistència dels agregats i evita, a més, la duplicitat de dades.

4 Implementació d'un repositori de productes amb la API de *Firestore*

Veurem les operacions d'afegir un producte (document), d'obtenció d'un producte amb un identificador determinat i la d'obtenir tots els productes (documents) de la col·lecció.

Aquesta serà la interfície del repositori:

```
1 public interface ProductRepository {
2    interface Callback<T> {
3     void onSuccess(T result);
4     void onError(Throwable error);
```

[3] https://firebase.google.com/docs/storage/android/start

```
    }

    void add(Product product, Callback<Boolean> callback);
    void getById(String id, Callback<Product> callback);
    void getAll(Callback<List<Product>> callback);
    ...
}
```

La interfície serà part de la capa del domini (paquet `domini` en el codi base) i el codi client de la interfície serà el servei `ProductStoreService`. La implementació del repositori `ProductFirestoreRepository` serà part de la capa de persistència (paquet `data` al codi base). Veure la Fig. 2 on s'il·lustra l'estructura dels paquests `data` i `domain`.

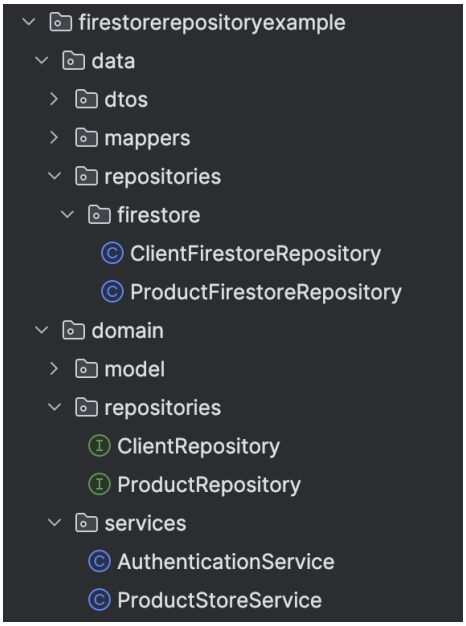

Figure 2: Estructura de directoris dels paquets `data` i `domain`. Les interfícies dels repositoris són part del domini (`domain`), igual que el seu codi client i les implementacions de persistència (`data`).

4.1 Afegir un producte al repositori

A continuació, es mostra la implementació del mètode `add(...)` per afegir un `Product` (document) a la col·lecció "productes" d'una *Firestore*:

```
public class ProductFirestoreRepository implements
    ProductRepository {
    /* Atributs */
```

```
3       private final String PRODUCTS_COLLECTION_NAME = "products";
4       private final FirebaseFirestore db; // Singleton
5
6       public ProductRepository() {
7           db = FirebaseFirestore.getInstance();
8       }
9
10      ...
11
12      /**
13       * Add a product to Firebase Cloud Firestore.
14       *
15       * @param product The product to add.
16       * @param callback The callback to be called when the
             operation is done.
17       */
18      public void add(Product product, Callback<Void> callback) {
19        ProductFirestoreDto productDto = DTOToDomainMapper.map(
              product, ProductFirestoreDto.class);
20
21        // Creació de la tasca d'escriptura a Firestore
22        Task<Void> task = db
23            .collection(PRODUCTS_COLLECTION_NAME)
24            .document(product.getId().toString())
25            .set(productDto);
26
27        // Gestió del resultat de la tasca
28        task.addOnFailureListener(new OnFailureListener() {
29          @Override
30          public void onFailure(@NonNull Exception exception) {
31            // Error, per alguna raó
32            callback.onError(new Throwable("Error adding product"
                  , exception));
33          }
34        })
35        .addOnSuccessListener(new OnSuccessListener<Void>() {
36          @Override
37          public void onSuccess(Void ignored) {
38            callback.onSuccess(null);
39          }
40        });
41      }
42
43      ...
44  }
```

Hi ha diverses observacions a fer sobre el codi anterior:

Objectes de transferència de dades i mapejador genèric. Observeu com creem un `Data Transfer Object` (DTO), amb la classe `ProductFirestoreDto` (línia 28). Els DTO són objectes contenidors que més tard es converteixen en objectes del domini. Els motius d'utilitzar DTO són, bàsicament, dos:

- A diferència de l'objecte del domini, són lleugers i sense comportament (només *getters*), de manera que és fàcil transferir-los entre components, capes o sistemes, i assegurar que les dades contingudes seran transferides sense mutacions degudes a comportaments invocats per cap codi client; cosa que no podríem assegurar si estesàssim parlant d'objectes del domini. *(No és un punt especialment rellevant en aquest context, perquè al final el mateix repositori és qui crea els objectes del domini i, per tant, té la capacitat d'invocar-ne comportament. Ara bé, si tinguéssim DAO per sota del repositori, sí. Justament, els DTO serien l'únic tipus de dades que els DAO farien anar.)*
- L'aplicació del criteri GRASP de *variació protegida* als objectes del domini, ja que, canvis en l'estructura de la base de dades no causaran (directament) variacions en els objectes del domini, sinó en els DTO.

Les conversions entre entre objectes del domini i DTO, les fem amb un mapejador genèric `DTOToDomainMapper`, del qual trobareu més detalls a la Sec. 4.5.

Definició de la `Task<T>`. Hem de definir el tipus de dada que esperem de la tasca. `Firestore` ens pot retornar diversos tipus d'objecte que no són objectes del domini ni DTOs, sinó els seus propis objectes. En el cas de l'`set` (línia 34), no ens retornaria cap dada (només ens assegura la crida d'un *callback*). Fixeu-vos que la `Task<T>` s'obté d'invocar un seguit de mètodes (línies 31-34) a partir de la instància de la classe `FirebaseFirestore` (singleton) – en aquest cas, `db`. Indiquem la col·lecció a la que volem accedir (mitjançant el nom), fem referència al document on voldrem escriure (que en cas de no indicar-lo, serà un hash numèric auto-generat) i, finalment, la comanda `set` (amb la instància del DTO).

Asincronia i *callbacks*. Fixeu-vos que definim escoltadors per la tasca, `OnSuccessListener` i `OnFailureListener`, revelant-nos que és asíncrona. I com que la `Task` és asíncrona, la resta del nostre codi també ho serà. Per això, definim una interfície pròpia, `Callback<T>` (línies 14-17), que el codi client del repositori haurà d'implementar per decidir com processa el resultat `T` obtingut. En aquest cas, el mètode `add` del repositori, igual que la tasca, no retorna res (`Void`) i, per tant serà un `Callback<Void>`. Volem saber només si s'ha fet l'addició del producte o si ha tingut lloc un error.

Execució de la tasca. És important tenir en compte que la tasca s'inicia en el moment d'afegir-li l'`OnSuccessListener`. Per aquest motiu, és recomanable

assignar primer l'`OnFailureListener` (línia 37) abans de l'`OnSuccessListener` (línia 44). Això garanteix que, en el cas (poc probable) que la tasca s'executi molt ràpidament, la gestió de fallada ja estigui preparada, evitant així possibles situacions en què els errors no siguin capturats correctament.

4.2 Obtenir un producte del repositori mitjançant l'identificador

Veiem a continuació la implementació d'un mètode `getById(...)` per obtenir un `Product` (document) concret d'una col·lecció de productes mitjançant l'identificador del producte:

```
1  public class ProductFirestoreRepository implements
       ProductRepository {
2    ...
3
4    /**
5     * Get product from a Firestore querying by identifier.
6     * @return The product.
7     */
8    public void getById(ProductId productId, Callback<Product>
         callback) {
9      /* Creació de la tasca d'obtenció */
10     Task<DocumentSnapshot> task = db
11       .collection(PRODUCTS_COLLECTION_NAME)
12       .document(productId.getId())
13       .get();
14
15     task.addOnFailureListener(exception -> {
16       callback.onError(exception);
17     })
18     .addOnSuccessListener(new OnSuccessListener<
           DocumentSnapshot>() {
19       @Override
20       public void onSuccess(DocumentSnapshot ds) {
21         // DocumentSnapshot -> ProductFirestoreDto
22         ProductFirestoreDto productDto = ds.toObject(
               ProductFirestoreDto.class);
23         // ProductFirestoreDto -> Product
24         Product product = DTOToDomainMapper.map(productDto,
               Product.class);
25         // Product pel callback
26         callback.onSuccess(product);
27       }
28     });
29   }
30
31   ...
32 }
```

En aquest cas, si tot va bé, la tasca ens farà arribar un objecte`DocumentSnapshot`, que representa la "instantània" del document en el moment de la lectura i que és, directament, convertible al tipus d'objecte Java que era originalment quan vàrem fer l'afegit a la base de dades – en aquest cas, un `ProductFirestoreDto`. Trobareu més detall sobre aquesta conversió a la Sec. 4.5.

4.3 Obtenir tots els productes del repositori

Si enlloc d'un `Product` (document) concret, volguéssim recuperar tots els productes, ometríem la part de la instrucció `.document(...)` en el moment de creació de la tasca. De la definició de la tasca n'obtindríem un `QuerySnapshot`, que pot ser iterat per obtenir diveross `DocumentSnapshot`.

Veiem la implementació d'un mètode `getAll(...)` del repositori:

```
1  public class ProductFirestoreRepository implements
       ProductRepository {
2    ...
3
4    /**
5     * Get all products from a Firestore.
6     * @return The product.
7     */
8    public void getAll(Callback<List<Product>> callback) {
9      // Creació de la tasca d'obtenció
10     Task<QuerySnapshot> task = db
11         .collection(PRODUCTS_COLLECTION_NAME)
12         .get();
13
14     task.addOnFailureListener(exception -> {
15       callback.onError(exception);
16     })
17     .addOnSuccessListener(new OnSuccessListener<QuerySnapshot
           >() {
18       @Override
19       public void onSuccess(QuerySnapshot querySnapshot) {
20         // Construïm una llista amb els Product
21         List<Product> products = new ArrayList<>();
22         // Iterem sobre els documents resultants de la query
23         for (DocumentSnapshot ds : querySnapshot) {
24           ProductFirestoreDto productDto = ds.toObject(
               ProductFirestoreDto.class);
25           Product product = DTOToDomainMapper.map(productDto,
               Product.class);
26           products.add(product);
27         }
28         // Llista de productes pel callback
29         callback.onSuccess(products);
```

```
30          }
31        });
32    }
33
34    ...
35  }
```

Per veure com es construeixen les tasques d'altres tipus (actualització o esborrat), podeu consultar vosaltres mateixos el següent recurs: https://firebase.google.com/docs/firestore/quickstart.

4.4 Codi client del repositori (`ProductStoreService`)

Aquest objecte del domini (un servei) serà el codi client del repositori de productes. Des del seu punt de vista, el repositori és un `ProductRepository` (i no un `ProductFirestoreRepository`):

```
1  public class ProductStoreService {
2     private final ProductRepository productRepository;
3
4    ...
5
6    @SuppressWarnings("unused")
7    public ProductStoreService() {
8      productRepository = new ProductFirestoreRepository();
9    }
10
11   public void getAllProducts(OnGetProductsListener listener) {
12       productRepository.getAll(
13           new ProductRepository.Callback<List<Product>>() {
14             @Override
15             public void onSuccess(List<Product> products) {
16               listener.onSuccess(products);
17             }
18             @Override
19             public void onError(Throwable error) {
20               listener.onError(error);
21             }
22           }
23     );
24   }
25
26   ...
27 }
```

Idealment, el codi client no hauria de dependre de la implementació concreta del repositori tal com ho fa aquí (línia 8). I això ho solucionarem amb injecció de dependències a la pròxima PI.

D'altra banda, som conscients que aquest codi client resulta ben poc interessant per la manca de lògica dels mètodes de `ProductStoreService`. Si examineu el `AuthenticarService` del codi base, veureu que els seus mètodes fan coses més interessants a partir de les respostes dels *callbacks* dels repositoris. Ens hem basat en el`ProductFirestoreRepository` i, per tant, en aquest codi client perquè està lligat a l'`Exercici 2`.

4.5 Conversions de tipus

4.5.1 DTO ↔ objecte del domini

Fem anar la classe `DTOToDomainMapper` per convertir de DTO a objecte del domini o viceversa. Aquesta classe estén de la classe `ModelMapper` d'una llibreria externa `org.modelmapper:modelmapper`[4] i permet convertir un objecte a un altre de qualsevol tipus, fent còpia dels valors dels atributs de l'objecte d'origen a l'objecte de destí. Per a fer el mapeig, cal que els noms dels atributs coincideixin. No és un requisit que el tipus d'atribut sigui el mateix, sempre que definim un mapeig entre tipus – tal com veurem a continuació.

Aquesta és la classe `DTOToDomainMapper`:

```
1  public class DTOToDomainMapper extends ModelMapper {
2
3    public DTOToDomainMapper() {
4      super();
5
6      super.getConfiguration()
7        .setFieldMatchingEnabled(true)
8        .setFieldAccessLevel(Configuration.AccessLevel.PRIVATE)
9        .setMatchingStrategy(MatchingStrategies.LOOSE);
10
11     /* Conversió entre tipus no coincidents */
12     super.addConverter(new AbstractConverter<String, ClientId
           >() {
13       @Override
14       protected ClientId convert(String source) {
15         return new ClientId(source);
16       }
17     });
18     super.addConverter(new AbstractConverter<String, ProductId
           >() {
19       @Override
20       protected ProductId convert(String source) {
21         return new ProductId(source);
22       }
23     });
```

[4] En el codi base ja teniu afegida la llibreria. Sinó, caldria afegir la línia `implementation 'org.modelmapper:modelmapper:2.4.4'` al bloc `dependencies {...}` del gradle a nivell de mòdul.

```
24      super.addConverter(new AbstractConverter<ClientId, String
            >() {
25        @Override
26        protected String convert(ClientId clientId) {
27          return clientId.toString();
28        }
29      });
30      super.addConverter(new AbstractConverter<ProductId, String
            >() {
31        @Override
32        protected String convert(ProductId productId) {
33          return productId.getId();
34        }
35      });
36    }
37
38    ...
39  }
```

Hi **establim la configuració** (línies 6-9) per dir-li que accedeixi als atributs sense necessitat d'un mètode *setter* (línia 7), perquè – recordeu – els nostres DTO no tenen *setters*. Després li donem permís per accedir als atributs encara que siguin privats (línia 8) i, finalment, li diem que el mapeig d'atributs no ha de ser 1:1 (línia 9). És a dir, que la classe destí i la classe d'origen poden tenir un número diferent o tenir el mateix número d'atributs però no haver-hi una correspondència de noms perfecta. Precisament, aquesta és l'estratègia adequada per la conversió entre DTO i objectes del domini, ja que, els DTO podrien contenir més dades de les que necessitem en el domini. Poseu per cas una base de dades compartida entre diverses aplicacions.

I per definir el **mapeig entre atributs de tipus no coincident**, afegim al `super` tots els conversors necessaris (línies 12-35). En aquest cas, convertim entre objectes identificador de producte (`ProductId`) o de client (`ClientId`) a `String`. Això ho fem, perquè els nostres DTO han de tenir un identificador de tipus `String` si volem fer-lo servir com a clau única dels documents de *Firestore*.

4.5.2 `DocumentSnapshot` → DTO

La conversió de `DTO` → `DocumentSnapshot` és implícita quan "guardem" el DTO a la *Firestore* amb una tasca creada a partir de la instrucció `set(...)`. No ho és en l'altre sentit.

Hem vist que amb les tasques d'obtenció (`get(...)`), la API de *Firestore* no retorna directament un DTO, sinó un `DocumentSnapshot` (o `QuerySnapshot`). I també que el mètode de conversió `toObject` dels `DocumentSnap-`

`shot` accepta un paràmetre que indica la classe a la que volem convertir l'objecte (p. ex. `ProductFirestoreDto.class`).

El mètode `toObject` té un funcionament semblant al `map` del `DTOToDomainObject`, però té alguna particularitat:

- Hem d'especificar-li al DTO quin és l'atribut que representarà l'identificador amb un decorador `"@DocumentId"`.
- Cal que els DTO implementin un constructor buit[5].

Podeu veure-ho a la implementació del `ProductFirestoreDto`:

```
public class ProductFirestoreDto {
  /* Attributes */
  @DocumentId
  private String id;
  private String name;
  private String nameLowerCase;
  private String description;
  private PriceFirestoreDto price;
  private String imageUrl;
  private Boolean fragile;

  /**
   * Empty constructor required for Firestore.
   */
  @SuppressWarnings("unused")
  public ProductFirestoreDto() { }

  /* Getters */
  public String getId() { return id; }
  public String getName() { return name; }
  public String getNameLowerCase() { return nameLowerCase; }
  public String getDescription() { return description; }
  public PriceFirestoreDto getPrice() { return price; }
  public String getImageUrl() { return imageUrl; }
  public Boolean getFragile() { return fragile; }
}
}
```

Finalment, podeu tornar a fer un cop d'ull a la Fig. 1 (part de la dreta) per veure com els camps del document (i, per tant, els atributs del `DocumentSnapshot` que obtinguem de la API) i del `ProductFirestoreDto` coincideixen. L'únic camp que no trobareu juntament amb la resta de camps és l'identificador, que és – justament – la clau del document i per això s'ha

[5]Android Studio us avisarà amb un *warning* de que el constructor no està sent utilitzat en el vostre codi, però sí que està sent utilitzat pel mètode `toObject` del `DocumentSnapshot`. Per tant, si us molesta el *warning*, podeu afegir un decorador `"@@SuppressWarnings("unused")"` al constructor buit.

d'especificar amb `@DocumentId` (de manera que `toObject` faci el mapeig d'aquest atribut correctament).

Exercicis

Exercici 1. Creeu una base de dades pròpia i enllaceu-la amb el projecte base, ja que la que hi ha enllaçada actualment no permet l'escriptura. Els passos són:

1. Crear un projecte de Firebase amb una base de dades *Firestore* i reemplaçar-la amb la nostra en el projecte del codi base. Per fer-ho, seguiu les instruccions de l'Ap. A.

2. Aneu a la **consola de *Firebase*** per veure el contingut de la vostra nova *Firestore*. Hauríeu de trobar-vos-la és completament buida (sense col·leccions o documents). Per pujar-hi dades, seguiu l'Ap. B. Un cop pujades, reviseu mitjançant la consola de Firebase que la pujada s'ha fet bé. Pot ser que us calgui refrescar la pàgina del vostre navegador web.

3. Arranqueu l'aplicació i loguejeu-vos amb admin/admin, per a comprovar que l'aplicació té accés a la base de dades, concretament a la col·lecció de clients.

Exercici 2. Implementeu l'esborrat d'un producte de la base de dades quan feu clic al botó d'esborrar un element de la vista reciclable de productes. A la PI anterior teníeu implementada la funcionalitat, però no era un canvi permanent. Ara ho hauria de ser. No us donem passos, però seguireu trobant pistes de què i on heu d'implementar, buscant "EXERCICI 2" en el codi base (Ctrl + Shift + f).

Si alguna cosa va malament i esborreu tots els elements, recordeu que sempre podeu tornar a pujar les dades a *Firestore* tal com us hem ensenyat a l'Ap. B.

Appendix A Crear i enllaçar base de dades *Firebase Firestore* a la vostra aplicació Android

Per a seguir aquestes instruccions, necessitareu un usuari de Google. Creu-vos-en un si no en teniu cap.

A.1 Crear projecte de Firebase i base de dades

Assumint que ja el teniu, per a **crear un projecte de Firebase**:

1. Accedir a https://firebase.google.com/ i clicar al botó "Get started" (o "Comenzar" en funció de l'idioma), que us demanarà loguejar-vos amb el vostre usuari de Google i, seguidament, us enviarà a una pantalla on hi veureu els vostres projectes de Firebase.

2. En la nova pantalla, tindreu la possibilitat d'afegir un nou projecte mitjançant un botó "+ Agregar proyecto" (veure la Fig. 3). Fent-hi clic, inicieu un procés de 3 passos senzills:

Figure 3: Afegir nou projecte.

   (a) Especificar un nom de projecte (p. ex. ElMeuProjecte), que no cal que coincideixi amb el nom del projecte d'Android Studio de la vostra aplicació, i feu clic a "Continuar".
   (b) Habilitar o no les *Google analytics* per aquest projecte. És opcional. Decidiu-ho i feu clic a "Continuar".
   (c) Creació (automàtica) del nou projecte Firebase. Un cop estigui llest, us apareixerà el que us mostrem a la Fig. 4.

Ja teniu el projecte de Firebase creat (sense base de dades). Per a **crear la base de dades Cloud Firestore**:

Figure 4: Un cop creat un projecte de Firebase us apareixerà la consola de Firebase. Al menú lateral hi teniu totes les opcions.

1. Cliqueu a la part inferior dreta, on veureu que hi diu "Cloud Firestore" (a la mateixa Fig. 4), entre d'altres eines.

2. En la nova pantalla, premeu el botó "Crear base de datos" i en la finestra emergent que se us obri, heu d'especificar la localització de la base de dades, "eur3 (Europa)", i feu clic a "Continuar".

3. Trieu ara el mode de funcionament "modo de prueba" i feu clic a "Continuar". Aquest mode no requereix d'autenticació per utilitzar la base de dades. Us facilitarà la vida, a canvi d'una mica menys de seguretat.

Arribats a aquest punt, heu creat un nou projecte de Firebase amb una *Firestore* buida. Falta enllaçar el projecte i, per tant, la base de dades.

A.2 Enllaçar projecte de Firebase amb aplicació d'Android

Per enllaçar el projecte de Firebase (i base de dades) cal seguir aquests passos:

1. Al menú lateral de l'esquerra de la consola de Firebase, mireu que al costat d'on diu "Descripción general" hi apareix un engranatge (veure la Fig. 5). Cliqueu-lo per veure un desplegable amb 3 opcions i cliqueu la que diu "Configuración del proyecto". Això us obrirà una nova pantalla (Fig. 6).

2. Dins de la pestanya "General", baixeu fins la secció "Tus apps" i veureu que no hi teniu cap app associada. Llavors, cliqueu a la icona d'Android (tornar a veure la Fig. 6) per obrir una nova pantalla "Agrega Firebase a tu app para Android" i seguir un assistent de 4 passos:

Figure 5: Accedir a la configuració del projecte des de la consola de Firebase.

(a) Especificar el paquet arrel de la vostra app d'Android Studio. **IMPORTANT**: si no coincideix, no funcionarà l'enllaç. En el cas del projecte que us hem compartit per aquest tutorial, seria `edu.ub.2425.firestorerepositoryexample`. La resta de la informació del primer pas podeu deixar-la en blanc. Feu clic a "Continuar".

(b) Generació del fitxer `google-services.json`, que haureu de baixar i posar dins del vostre projecte. Concretament, en la localització que us especifica el mateix l'assistent. Es pot fer ara o més tard, però no us n'oblideu de fer-ho. Torneu a fer clic a "Continuar".

(c) Modificar els fitxers `gradle` del projecte (a nivell de projecte i a nivell de mòdul) tal com us indiquen les instruccions per afegir-hi plugins i dependències. **IMPORTANT**: copieu i enganxeu les línies a la secció que correspongui (`plugins {...}` o `dependencies {...}`), perquè aquestes seccions ja existeixen (no dupliqueu les seccions). Feu clic a "Continuar" fins a sortir de l'assistent.

Un cop modificats els gradle, no us oblideu de fer `Sync now...` (a la barra superior groga que us apareix sempre que editeu el fitxer).

Enhorabona! **Ja teniu el vostre projecte de Firebase enllaçat**. Si voleu consultar el contingut de la base de dades (buida quan l'acabeu de crear), amb l'explorador d'Internet, podeu anar a la barra lateral de l'esquerra de la consola de Firebase i clicar "Firestore Database". Veure Fig. 7.

Finalment, tingueu en compte que heu configurat la base de dades en mode de test (no de producció), el qual té un temps d'ús limitat a 30 dies. Podeu estendre aquest temps des de la consola de Firebase, editant-ne les regles tal com es mostra a la Fig. 8. Si volguéssiu passar-la a mode de producció per restringir l'accés a través d'autenticació, podríeu fer-ho també editant les regles sense crear una nova base de dades. Més detalls sobre això últim: `https:`

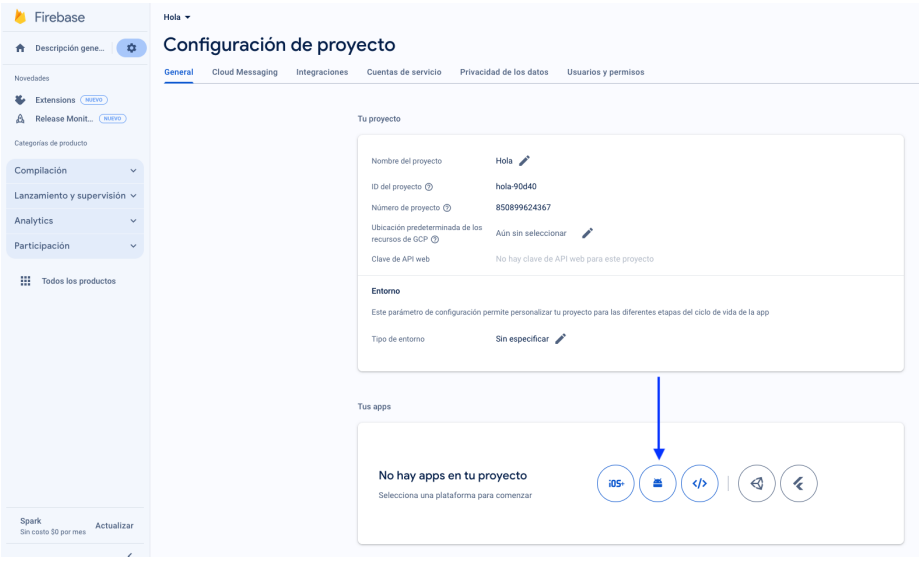

Figure 6: Enllaçar amb apliació Android.

//firebase.google.com/docs/rules/basics#cloud-firestore.

Appendix B Pujar dades a *Firestore* des de fora d'una aplicació Android

Hi haurà dades de la base de dades de la vostra aplicació que es poblaran a mesura que els usuaris hi interactuïn, però – molt possiblement – voldreu que l'aplicació tingui unes dades inicials abans que cap usuari hi faci res.

Fins ara, només sabeu com interactuar amb la base de dades des de la vostra aplicació. Per pujar-hi dades, ho hauríeu de fer des d'aquesta o des de la consola de Firebase. Això últim podeu investigar-ho vosaltres mateixos. No us ho mostrem perquè no resulta pràctic. El que sí trobem convenient és tenir un programa, i no necessàriament una aplicació Android, sinó de Java (projecte d'IntelliJ), que pugui llegir fitxers de dades (en format JSON) i pujar-les directament a la base de dades. Això us evitarà el que es coneix com a *cold start* (tenir una aplicació sense dades inicials).

Per a pujar dades des de fora d'una aplicació, necessitareu dues coses: (1) una clau privada per accedir al compte de servei del vostre projecte de Firebase (Sec. B.1) i (2) el projecte d'IntelliJ contingut en el fitxer `FirestoreUploaderExample.zip` que us proporcionem i que fa ús de la llibreria `firebase-`

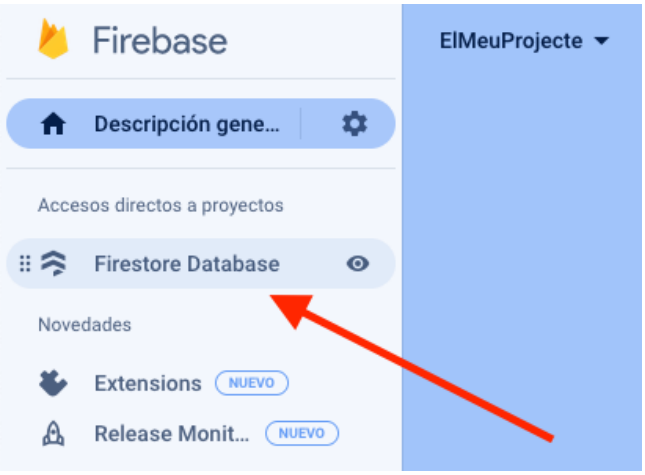

Figure 7: Accedir a la Firestore creada des de la consola de Firebase (barra lateral esquerra).

`admin` per operar amb la *Firestore* i pujar-hi dades sense passar per cap aplicació Android (Sec. B.2).

B.1 Generar clau privada de compte de servei del projecte Firebase

Per generar una clau privada i utilitzar el compte de servei d'un projecte de Firebase que permet accedir-hi com a administrador, heu d'anar a la consola de Firebase del vostre projecte. A la barra lateral esquerra, fer clic a l'engranatge que hi ha a la dreta de "Descripción general" > "Configuración del proyecto", i en el desplegable, seleccionar l'opció "Cuentas de servicio". A continuació trieu "Java" i feu clic al botó blau "Generar nova clau privada" (veure Fig. 9). En fer això, podreu descarregar un fitxer `json`. Aquest fitxer, a diferència del fitxer que utilitzeu en la vostra aplicació per accedir a la base de dades en mode no-administrador, s'utilitzarà en el projecte d'IntelliJ de la secció següent (Sec. B.2).

B.2 Pujar dades amb aplicació externa

La pujada de dades us la donem implementada en el codi Java (projecte IntelliJ, no d'Android Studio) empaquetat en el fitxer `FirestoreUploaderExample.zip`. Haureu d'haver seguit, com a mínim, els passos de la Sec. A.1 i Sec. B.1. Un cop fet:

1. Descarregueu-vos el projecte `FirestoreUploaderExample.zip`, descomprimiu-lo i obriu-lo amb IntelliJ.

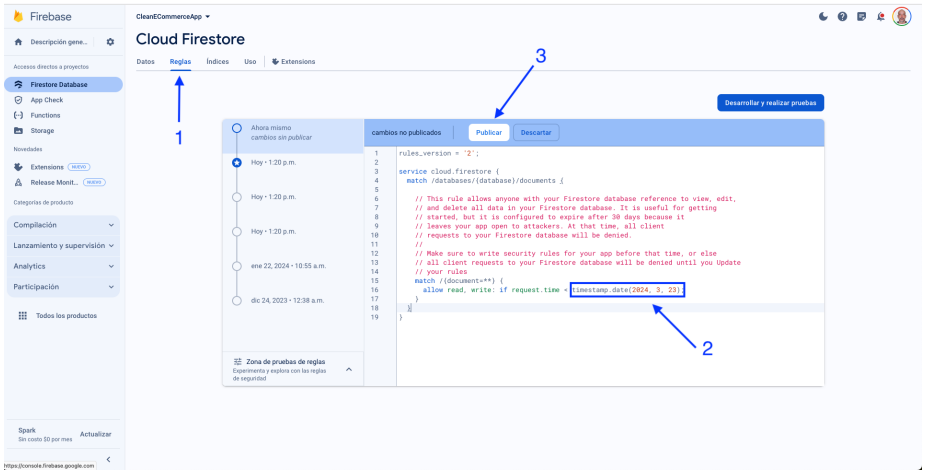

Figure 8: Editar restriccions d'accés per estendre l'ús de la base de dades Firestore configurada en mode de prova durant més de 30 dies (establert per defecte).

2. Comproveu que té les següents llibreries afegides:

   - `com.google.firebase:firebase-admin:9.4.3`
   - `org.slf4j:slf4j-nop:2.0.11` (dependència de `firebase-admin`)
   - `com.google.code.gson:gson:2.10`

   Recordeu que per consultar/instal·lar llibreries a IntellJ, heu d'anar a "File" > "Project Structure..." > "Project Settings" (a la barra lateral esquerra) > "Libraries". En cas d'haver-les d'Instal·lar, fer clic al botó "+" i triar "Maven". A la finestra emergent, escriure-hi el nom i versió de la llibreria (p. ex. `com.google.firebase:firebase-admin:9.2.0`).

3. Copieu el json del compte de servei (descarregat seguint les instruccions de la Sec. B) al directori `resources`. Veure Fig. 10.

4. Navegueu al `Main.java` del projecte i, llavors, modifiqueu el valor de la constant `SERVICE_ACCOUNT_KEY_FILEPATH` amb la ruta del vostre `json` del compte de servei. (En principi, només us caldria canviar el nom del fitxer, la part de la ruta que fa referència al directori no us caldria tocar-la si l'heu posat a la carpeta `resources`).

Abans d'executar el codi, acabeu d'inspeccionar el mètode `main` de la classe `Main`. Fixeu-vos en les crides `uploader.uploadJsonToCollection(...)` i els seus paràmetres. El que fan és crear cadascuna una col·lecció a la *Firestore* i hi pujar-hi dades llegint els dos fitxers – també – en format `json` que hi ha a la carpeta de `resources` del projecte: `resources/clients.json` i `resources/products.json`. Examineu-los per veure com hi hem formatat les

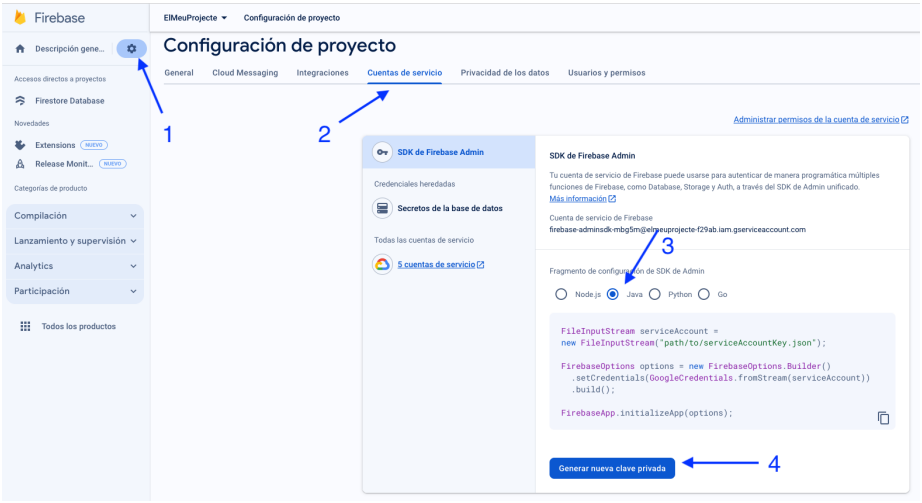

Figure 9: Lloc de la consola on s'obté la clau privada per l'accés al projecte de Firebase en mode administrador mitjançant compte de servei.

Figure 10: Posar el json del compte de servei al directori `resources`.

dades.

Finalment, executeu el codi i aneu a veure si se us ha omplert la vostra base de dades.

Appendix C Actualització dels documents i transaccions de *Firestore*

(Es necessari haver llegit el guió per entendre aquest apèndix.)

La operació d'actualització seria la que utilitzaríem en una aplicació on diversos usuaris causessin la modificació del mateix document simultàniament. Per exemple, per una aplicació de xarxa social on els documents representessin publicacions i les publicacions tinguessin una llista (o subcol·lecció) de comentaris. No seria estrany que diversos usuaris escrivissin comentaris alhora i la operació d'actualització causés inconsistències en el document, és dir, la pèrdua de comentaris.

Per a l'actualització de dades és important, doncs, conèixer el concepte de *transacció*. Les **transaccions** són un mecanisme de les bases de dades que assegura l'execució íntegra d'una seqüència d'operacions sobre un o diversos registres – en aquest cas, els camps dels documents. Quan es defineix una transacció, es crea un bloc de codi que llegeix, modifica i actualitza un o més camps d'un document. Si durant l'execució del bloc es detecta que el document ha canviat, la transacció es *rollbackeja* automàticament i es torna a intentar indefinidament o número finit de vegades – depenent de com estigui implementat.

No us ho explicarem aquí en detall com fer-ho amb *Firestore*, però us direm que és possible i que està explicat a https://firebase.google.com/docs/firestore/manage-data/transactions.