Report Finale: Track My Pantry

Marco Benito Tomasone 873909

04/09/2021

1 Introduzione

Questo documento è parte integrante del progetto finale del corso di Laboratorio di Applicazioni Mobili, dell'Alma Mater Studiorum - Università di Bologna, dell'A.S. 2020/2021.
Il progetto prevedeva la produzione di un'applicazione che permetteva ai suoi utenti di tenere traccia dei prodotti nella propria dispensa e di creare un database collaborativo di barcode di prodotti e delle loro caratteristiche.
Questo report finale è una rendicontazione del lavoro svolto, una spiegazione delle scelte implementative ed una presentazione delle feature prodotte.

2 Feature del Progetto

2.1 Requisiti Minimi

Le feature ufficialmente richieste, e minime, per questo progetto sono principalmente due:

- **Migliorare il database collaborativo:** tutti gli utenti dell'applicazione devono essere in grado di partecipare alla costruzione di un database collaborativo e collettivo, in cui gli utenti possono cercare i loro prodotti inserendo il barcode , inserirne di nuovi nel caso non fossero presenti ed esprimere una loro valutazione positiva relativamente ad un prodotto in particolare. Per accedere a questo database gli utenti devono essere in grado di registrarsi e di autenticarsi correttamente al database.
- **Tenere traccia dei prodotti nella dispensa:** tutti i prodotti ottenuti/inseriti dal database collaborativo devono essere correttamente inseriti in un database locale che deve permettere agli utenti di tenere traccia dei prodotti in loro possesso ed eliminarli a fine utilizzo.

2.2 Feature Aggiuntive

Particolarmente indicata era la possibilità di inserire delle feature aggiuntive al progetto. Qui di seguito la lista delle feature da me aggiunte:

2.2.1 Categorie

La prima feature aggiuntiva che ho inserito nel progetto è una divisione dei prodotti per categorie. L'utente può inserire e gestire i prodotti in una categoria particolare, es. Carne, Pesce, Lievitati, Dolci, Bevande. Si potrebbe anche pensare ad un utilizzo delle categorie per macronutrienti, quindi carboidrati, grassi e proteine. Le operazioni possibili sulle categorie sono: inserimento, modifica ed eliminazione.

2.2.2 Quantità dei prodotti

La seconda feature aggiuntiva è stata l'aggiunta della quantità dei prodotti. Al momento dell'inserimento di un nuovo prodotto la quantità base è pari ad 1. Successivamente è possibile aumentarne e diminuirne la quantità durante la visualizzazione della propria dispensa. Dal momento in cui la quantità di un prodotto scende a 0, il prodotto non viene eliminato dal database ma viene inserito nella Lista Della Spesa.

2.2.3 Scansionamento del Barcode

Oltre che inserire il barcode manualmente è possibile scansionare il barcode tramite la fotocamera ed attivare il flash durante la scansione premendo il tasto Volume Su.

2.2.4 Filtrare i prodotti per nome

Durante la visualizzazione della lista dei prodotti per categoria è possibile tramite un'apposita barra di ricerca filtrare i prodotti per nome, carattere per carattere durante l'inserimento.

2.2.5 Immagini

Per i prodotti nel database collaborativo per i quali è stata inserita un'immagine è possibile visualizzarla in tutte le parti dell'applicazione in cui è prevista la visualizzazione del prodotto (Lista dei prodotti, Lista della spesa, Sezione di ricerca del prodotto). Per i prodotti per i quali non è stata opportunamente caricata una foto, verrà visualizzata un'immagine fittizia.

2.2.6 Inserimento di un nuovo prodotto scattando una foto

Al momento dell'inserimento di un nuovo prodotto nel database collaborativo è possibile scattare una foto del prodotto ed allegarla ad esso durante l'inserimento, in modo tale da visualizzarla per tutti i successivi usi del prodotto stesso. L'app è pensata per essere utilizzata in semplicità e principalmente scansionando i codici a barre dei prodotti, quindi avendoli fisicamente durante l'utilizzo. E' difficile pensare ad un utilizzo dell'applicazione in cui l'utente inserisce il codice a barre manualmente, questo è il motivo per cui è possibile fare una foto del prodotto per allegarla ma non è possibile caricarne una dalla galleria.

2.2.7 Lista della Spesa

Dal momento in cui un prodotto raggiunge una quantità pari a 0, viene eliminato dalla lista dei prodotti ed inserito nella lista della spesa. Questa serve come reminder dei prodotti da acquistare in negozio. Dal momento in cui un prodotto viene nuovamente scannerizzato ed inserito nella lista dei prodotti viene automaticamente eliminato dalla lista della spesa.

2.2.8 Gestione degli utenti

Per accedere all'app ed utilizzare il database collettivo viene richiesto un login. La procedura di login viene sfruttata per separare i prodotti nel database locale di utenti diversi. E' possibile quindi utilizzare l'applicazione con due utenti diversi ed i prodotti verranno correttamente separati fra i due.

3 Aspetti Implementativi

In questa sezione spiegherò quelle che sono state le scelte implementative del progetto, allegando degli snippet di codice. Durante la stesura di questo progetto è stato utilizzato il pattern Model View View Model (MVVM), il database locale è stato gestito tramite Room, mentre le chiamate alle API del database collaborativo sono state gestite mendiante Retrofit.

3.1 SplashActivity

Questa è l'activity di caricamento dell'applicazione. E' un'activity prettamente grafica, durante il suo caricamento sceglie quale è la prossima activity da invocare. Se il token di login è ancora valido carica direttamente l'activity con la lista delle categorie, altrimenti viene caricata l'activity di login.

```
1
2         int oneWeekAsSec = 604800;
3         SharedPreferences pref =
              getApplication().getApplicationContext().getSharedPreferences("MY_PREFERENCES",
              Context.MODE_PRIVATE);
4         long currentDate =
              Calendar.getInstance(Locale.getDefault()).getTimeInMillis()
              / 1000;
5         Long savedAccessTokenData =
              Long.valueOf(pref.getString("DATA", String.valueOf(-1)));
6         Intent intent = null;
7         if(savedAccessTokenData != -1) {
8             if (currentDate < savedAccessTokenData + oneWeekAsSec)
9                 intent = new Intent(SplashActivity.this,
                      CategoryListActivity.class);
10            else{
11                Toast.makeText(SplashActivity.this,"Token Expired!
                      Please Log In", Toast.LENGTH_SHORT).show();
12                intent = new Intent(SplashActivity.this,
                      LoginActivity.class);
13            }
14        }
15        else
16                intent = new Intent(SplashActivity.this,
                      LoginActivity.class);
17    }
```

La scelta di usare per i dati della login le SharedPreferences e non magari il database è data dalla volontà di sperimentare più soluzioni possibili per apprendere il più possibile. Può succedere infatti che delle scelte implementative per questo progetto non siano le migliori e che avrei potuto fare scelte diverse, la ragione è proprio quella di sperimentare il più possibile con il codice.

3.2 Login & registrazione

L'activity di Login e Registrazione contiene un campo nascosto il campo name, che viene visualizzato o meno nel momento in cui si preme il testo per passare tra una vista e l'altra. Questa activity ha un view Model che si occupa di gestire le richieste. Il view model è composto da solo due chiamate la login e la register ed esso sono associati delle classi opportunamente create per la gestione dei dati richiesti, come la LoginData e la RegisterData.

3.3 Database & chiamate http

In entrambe le classi per la gestione del database con Room e per la gestione delle chiamate Http con Retrofit è stato utilizzato il design pattern creazionale Singleton.

```
        @Database(entities = {Category.class, Product.class},
            version = 10)
public abstract class AppDataBase extends RoomDatabase {
    public abstract PantryDao pantryDao();

    public static AppDataBase INSTANCE;

    public static AppDataBase getDataBaseInstance(Context context){
        if(INSTANCE == null){
            //create the database
            INSTANCE =
                Room.databaseBuilder(context.getApplicationContext(),
                AppDataBase.class, "ProductDatabase")
                    .fallbackToDestructiveMigration()
                    .build();
        }
        return INSTANCE;

    }
}
```

Allego solo il codice della classe AppDataBase dal momento in cui la classe RetroIstance è simile. Per il DB è stata creata poi una interfaccia DAO, mentre per Retrofit è stata creata un'interfaccia per i Service HTTP. Allego in questo caso la seconda.

```
        public interface APIService {
    @Headers("Content-Type: application/json")
    @POST("users")
```

```
    Call<Authentication> registrationMethod(@Body RegisterData
        registerData);

    @Headers("Content-Type: application/json")
    @GET("users/me")
    Call<Authentication> getUserData(@Header("Authorization")
        String accessToken);

    @Headers("Content-Type: application/json")
    @POST("auth/login")
    Call<AccessToken> loginMethod(@Body LoginData loginData);

    @Headers("Content-Type: application/json")
    @POST("products")
    Call<Product> insertNewProduct(@Body CreateProductSchema
        product, @Header("Authorization") String accessToken);

    @Headers("Content-Type: application/json")
    @GET("products")
    Call<GetProductSchema> getProductByBarcode(@Query("barcode")
        String Barcode, @Header("Authorization") String accessToken);

    @Headers("Content-Type: application/json")
    @POST("votes")
    Call<Void> insertVote(@Body CreateVoteSchema voteSchema,
        @Header("Authorization") String accessToken);
}
```

3.4 Activity Categorie

Per la gestione delle categorie è stata creata una tabella apposita all'interno del database. La gestione grafica delle categorie è affidata ad una Recycler View. Utilizzando il pattern MVVM, a questa classe sono collegati un View-Model ed un Adapter che servono per la gestione delle categorie. Il view model è così fatto:

```
        public class CategoryListViewModel extends AndroidViewModel {

    private MutableLiveData<List<Category>> listOfCategories;
    private AppDataBase appDataBase;
    private SharedPreferences pref;

    public CategoryListViewModel(Application application){
```

```
8           super(application);
9           listOfCategories = new MutableLiveData<>();
10          appDataBase =
                AppDataBase.getDataBaseInstance(getApplication().getApplicationContext());
11          pref =
                getApplication().getApplicationContext().getSharedPreferences("MY_PREFERENCES",
                Context.MODE_PRIVATE);
12          getAllCategories();
13      }
14      //Observer for the liveData, so for the categories you have
15      public MutableLiveData<List<Category>>
            getListOfCategoriesObserver(){
16          return listOfCategories;
17      }
18
19      //Function to get all the categories in the DataBase and post
            them into the liveData
20      public void getAllCategories(){
21          List<Category> categoriesList =
                appDataBase.pantryDao().getAllCategories(pref.getString("EMAIL",
                null));
22          if(categoriesList.size() > 0)
23              listOfCategories.postValue(categoriesList);
24          else
25          listOfCategories.postValue(null);
26          }
27          public void insertCategory(String categoryName){
28              }
29      public void updateCategory(Category category){
30          }
31          public void deleteCategory(Category category){
32              }
33  }
```

Il codice delle funzioni non è stato inserito per semplicità dal momento in cui verrà allegato a questo report. La particolarità principale è il MutableLiveData che permette di essere aggiornato tratime quell'Observer ed essere osservato dall'Adapter che permette un aggiornamento automatico all'applicazione delle operazioni come inserimento, modifica ed eliminazione di una categoria. Dal momento in cui si decide di eliminare una categoria dal proprio database verranno anche contestualmente eliminati tutti i prodotti in essa contenuti. La modifica permette, invece, di cambiare il nome della categoria. Il campo email che viene passato alla funzione getAllCategories()

serve per ottenere solo i prodotti di un determinato utente, stessa cosa verrà fatta per i prodotti. Per semplicità dell'Adapter verranno riportate solo tre parti di codice:

```
1        public void setCategoryList(List<Category> categoryList){
2            this.categoryList = categoryList;
3            notifyDataSetChanged();
4            }
```

La prima è questa funzione che serve per notificare alla View che i dati sono cambiati e di conseguenza riaggiornare la view.

```
1
2        @Override //Set data to our TextView
3        public void onBindViewHolder(@NonNull
             CategoryListAdapter.MyViewHolder holder, int position) {
4            holder.tvCategoryName.setText(this.categoryList.get(position).categoryName);
5
6            holder.itemView.setOnClickListener(new
                 View.OnClickListener() {
7                @Override
8                public void onClick(View v) {
9                    clickListener.itemClick(categoryList.get(position));
10               }
11               });
12
13               holder.editCategory.setOnClickListener(new
                     View.OnClickListener() {
14               @Override
15               public void onClick(View v) {
16                   clickListener.editClick(categoryList.get(position));
17               }
18               });
19
20               holder.deleteCategory.setOnClickListener(new
                     View.OnClickListener() {
21                   @Override
22                   public void onClick(View v) {
23                       clickListener.deleteClick(categoryList.get(position));
24                       }
25                       });
26                       }
```

Questa funzone serve invece per fare binding tra i dati e le parti della View.

Come è possibile vedere implementano delle funzioni di un clickListener, questo è proprio il dataType sottostante, il ViewHolder. Ce ne saranno di simili per la lista della spesa e per la lista dei prodotti.

```
1
2       public class MyViewHolder extends RecyclerView.ViewHolder{
3           TextView tvCategoryName;
4           ImageView deleteCategory;
5           ImageView editCategory;
6
7           public MyViewHolder(View view){
8               super(view);
9               tvCategoryName =
                    view.findViewById(R.id.textViewCategoryName);
10              deleteCategory =
                    view.findViewById(R.id.image_delete_category);
11              editCategory =
                    view.findViewById(R.id.image_edit_category);
12              }
13              }
14
15              public interface HandleCategoryClick {
16                  void itemClick(Category category);
17          void deleteClick(Category category);
18          void editClick(Category category);
19          }
20          }
```

3.5 Activity Lista di Prodotti

Per la lista dei prodotti l'architettura è simile che per la gestione delle categorie, c'è quindi il ViewModel, l'Adapter, ma, in aggiunta è stato utilizzato il pattern Repository per implementare la SingleSource of Truth. In questo caso quindi il View Model si occupa di collegare le chiamate della view al Repository che al contrario si occupa delle chiamate e della consistenza dei dati nel db interno. Il Repository è unico per tutti i prodotti, sia quelli nella lista della spesa che per quelli nella lista dei prodotti. Si allega ad esempio la funzione per la creazione di un nuovo prodotto. A seguito di una chiamata http con successo il prodotto oltre ad essere inserito nel db collaborativo viene anche automaticamente inserito nel database locale dell'utente.

```
1       public void createNewProduct(CreateProductSchema
            productSchema, int categoryId, SearchProductsActivity
```

```
            searchProductsActivity) {
        APIService apiService =
            RetroInstance.getRetroClient().create(APIService.class);
        Call<Product> call =
            apiService.insertNewProduct(productSchema, "Bearer " +
            pref.getString("ACCESS_TOKEN",""));
        call.enqueue(new Callback<Product>() {
            @Override
            public void onResponse(Call<Product> call,
                Response<Product> response) {
                if (response.isSuccessful()) {
                    Product product = (Product) response.body();
                    product.setCategoryId(categoryId);
                    product.setUserEmail(pref.getString("EMAIL",
                        null));
                    insertProduct(product);
                } else{
                    Toast.makeText(searchProductsActivity.getApplicationContext(),"Error
                        creating Product!",
                        Toast.LENGTH_SHORT).show();
                }
            }
            @Override
            public void onFailure(Call<Product> call, Throwable t) {
                Log.e("ERROR", t.getMessage());
            }
        });
    }
```

3.6 Filtrare i prodotti

Il filtro dei prodotti per nome è stato semplicemente implementato mediante una TextView e ad ogni carattere inserito tramite un textwatcher viene chiamata una funzione dell'adapter per cercare e mostrare i prodotti contenenti quella sottostringa tra tutti i prodotti posseduti in quella categoria.

```
        @Override
        public void afterTextChanged(Editable s) {
            filterList(s.toString());
        }
```

3.7 Ricerca di un prodotto

La ricerca di un prodotto viene fatta tramite le chiamate, al ritorno da una ricerca si ottiene la lista di tutti i prodotti con quel barcode. Se si clicca su un prodotto questo viene aggiunto alla lista dei prodotti nel database locale, ma prima viene chiesto di darne una valutazione da 1 a 5 stelle, tramite una rating bar.

3.8 Inserimento di un nuovo prodotto

Se non si è trovato prodotti che rispecchiano i parametri di ricerca è possibile inserirne di nuovi, l'inserimento di un nuovo prodotto tramite l'apposito button richiederà il nome, la descrizione di un prodotto ed eventualmente è possibile scattare una foto al prodotto tramite la fotocamera.

```
1        camera.setOnClickListener(new View.OnClickListener() {
2            @Override
3            public void onClick(View v) {
4                if
                     (ContextCompat.checkSelfPermission(SearchProductsActivity.this,
                     Manifest.permission.CAMERA) !=
                     PackageManager.PERMISSION_GRANTED) {
5                    ActivityCompat.requestPermissions(SearchProductsActivity.this,
                         new String[]{
6                            Manifest.permission.CAMERA}, 100);
7                    };
8                Intent intent = new
                     Intent(MediaStore.ACTION_IMAGE_CAPTURE);
9                startActivityForResult(intent, 100);
10
11           }
12       });
13
14       protected void onActivityResult(int requestCode, int
             resultCode, Intent data) {
15           super.onActivityResult(requestCode, resultCode, data);
16           if(requestCode == 100){
17               Bitmap bitmap = (Bitmap) data.getExtras().get("data");
18               imageView.setImageBitmap(bitmap);
19               String base64conversion = encodeImage(bitmap);
20               base64 = "" +
                     base64conversion; }
```

3.9 Scanning Barcode Fotocamera

E' possibile scannerizzare il barcode dei prodotti con la fotocamera cliccando su un'icona nel dialog di ricerca. Il codice per renderlo possibile:

```
1        camera.setOnClickListener(new View.OnClickListener() {
2            @Override
3            public void onClick(View v) {
4                dialogBuilder.dismiss();
5                IntentIntegrator intentIntegrator = new
                     IntentIntegrator(SearchProductsActivity.this);
6                intentIntegrator.setCaptureActivity(ScanActivity.class);
7                intentIntegrator.setDesiredBarcodeFormats(intentIntegrator.ALL_CODE_TYPES);
8                intentIntegrator.setBeepEnabled(true);
9                intentIntegrator.setOrientationLocked(true);
10               intentIntegrator.setPrompt("For Flash use volume up
                     Key");
11               intentIntegrator.initiateScan();
12           }
13       });
```

3.10 Immagini dei prodotti

Le immagini dei prodotti sono state caricate mediante la libreria Glide. Se l'immagine c'è viene correttamente visualizzata altrimenti viene visualizzata un'immagine con il testo "No image avaible". Questa scelta si pone in essere dal momento in cui in un utilizzo meno accademico e più reale dell'applicazione, pochi saranno i prodotti non accompagnati da un'immagine.

```
1    if (this.productList.get(position).getImg() != null) {
2        if
             (!this.productList.get(position).getImg().startsWith(""))
3            this.productList.get(position).setImg(""
                 + this.productList.get(position).getImg());
4        Glide.with(context).load(this.productList.get(position).getImg())
5                .apply(RequestOptions.centerCropTransform()).into(holder.productImage);
6        } else
7           Glide.with(context).load(R.drawable.no_image_avaible).into(holder.productImage);
```

3.11 Lista della spesa

La lista della spesa è implementata sempre tramite una RecyclerView che visualizzerà tutti i prodotti con quantità pari a 0. Anch'essa un Adapter e si appoggia al ViewModel dei prodotti che ritorna un observer per i soli prodotti che hanno un quantità pari a 0. Il tasto delete nella lista della spesa elimina un prodotto nel database locale, mentre se si riacquista un prodotto che è nella lista della spesa viene tolto da quest'ultima e inserito nella categoria di appartenenza.

4 Conclusioni

Il progetto è stato molto utile per affacciarmi e imparare delle nuove tecnologie che non avevo mai sperimentato prima. Probabilemte non tutte le scelte sono state corrette a livello tecnico ma molte sono state prese per diversificare e sperimentare con una nuova tecnologia e cercare di apprendere il più possibile.