Relazione progetto
di
Laboratorio di applicazioni mobili
A.A. 2023/2024
Università di Bologna

Daniele D'Ugo - 0001027741
(daniele.dugo@studio.unibo.it)

"HomeCook's companion"

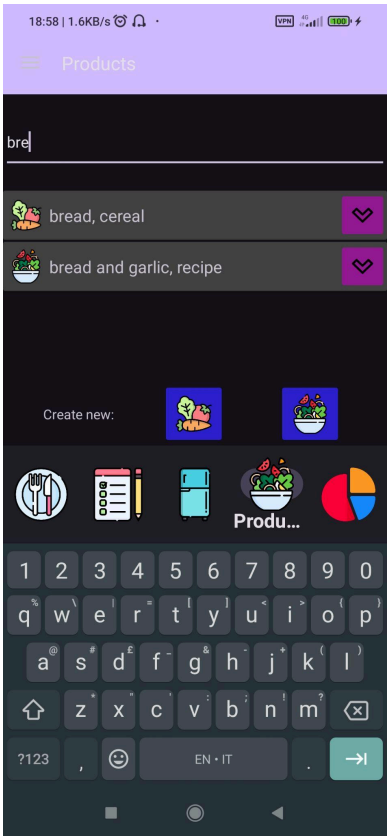
Introduzione

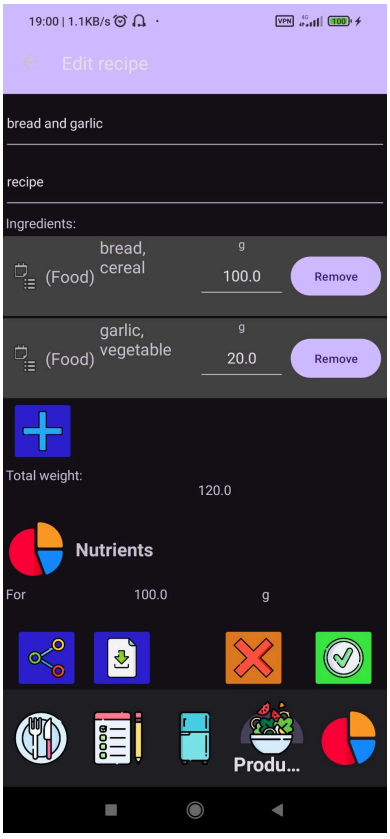
L’applicazione presentata come progetto per il corso di Laboratorio di applicazioni mobili, “HomeCook’s companion”, vuole essere un aiuto per gli acquisti nella vita quotidiana, e in particolare per l’alimentazione, in tutte le sue parti. Nel suo uso principale, l’utente può salvare una spesa effettuata, pianificare i suoi pasti - con calcolo di calorie e macronutrienti - in base alle sue scorte e tenere traccia dei prodotti quasi esauriti, salvandoli in una lista della spesa e ricominciando così il ciclo (compra-pianifica-consuma).
L’applicazione è stata sviluppata con un requisito minimo di versione di *SDK* 30 (*Android 11*).

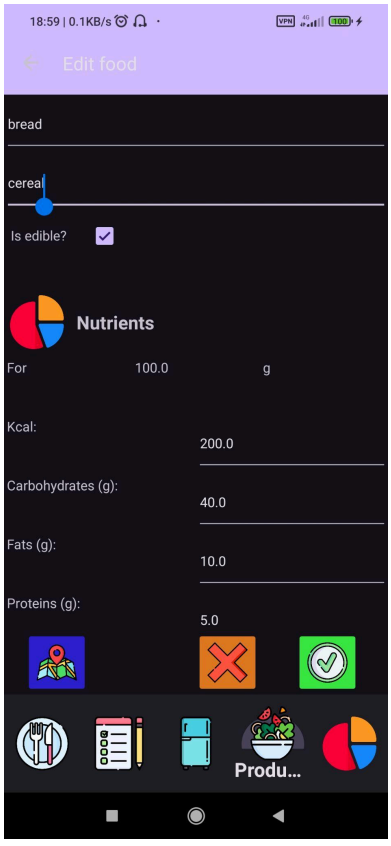
Funzionalità

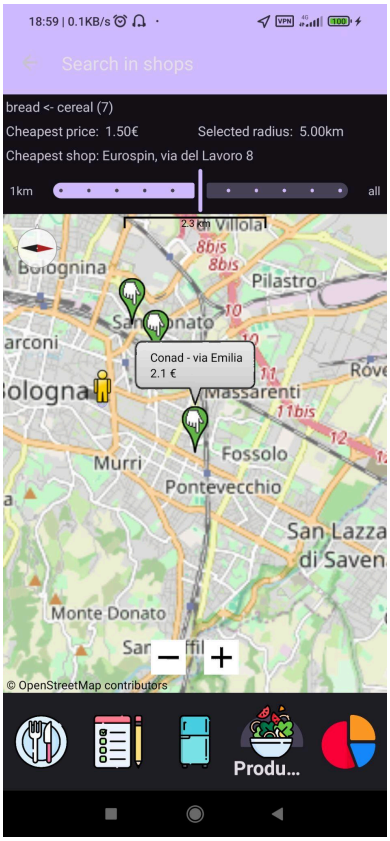
Prodotti

Principalmente, l'applicazione permette di gestire prodotti quotidiani, con particolare attenzione per gli alimenti, di cui si hanno le kilocalorie e i (grammi di) macronutrienti per *100g* di prodotto. Un alimento, poi, può anche essere una ricetta: in tal caso, gli si associano degli ingredienti, in varie quantità (in grammi), e i suoi valori nutrizionali equivalgono alla loro media su *100g*. Un prodotto è identificato da due campi - un nome e un'eventuale categoria (*parent*) -, che vengono semplicemente concatenati per mostrarlo (per esempio, "pomodoro, verdure"). Un prodotto (o una ricetta) può essere creato o modificato in ogni momento dalla schermata dei prodotti, o solo visualizzato e modificato cliccandoci sopra da un'altra schermata. Accedendo a un prodotto, è possibile anche aprire una mappa che mostri i negozi salvati, ognuno con il miglior prezzo che si è trovato per esso (prezzo assoluto), e il negozio con il prezzo migliore in assoluto, tutto in un raggio (regolabile) dalla propria posizione.







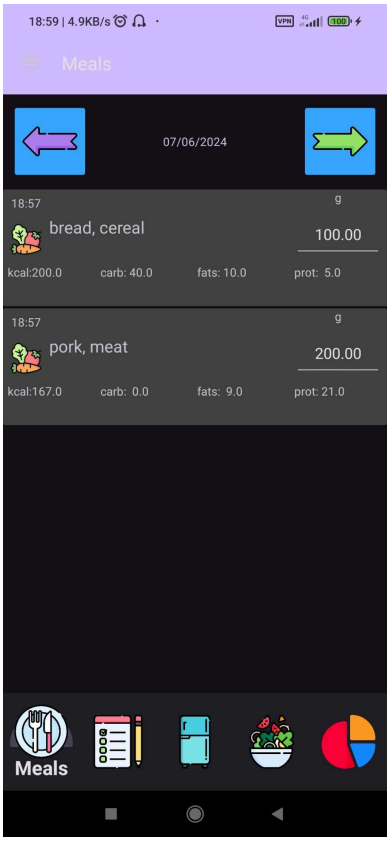


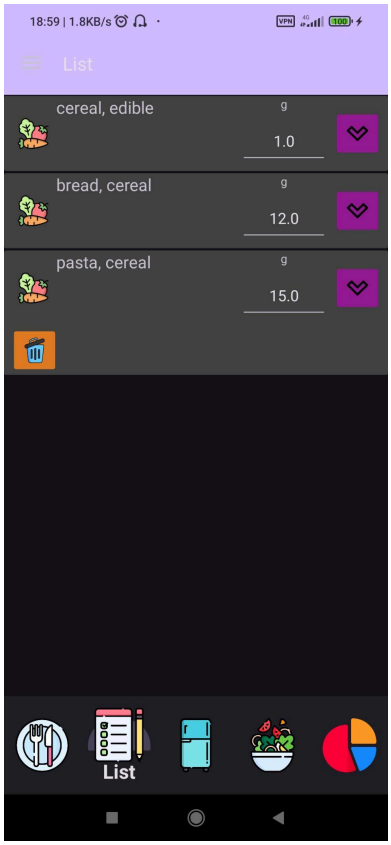
La schermata dell'elenco di tutti i prodotti permette anche di cercarli per sottostringa sul nome.

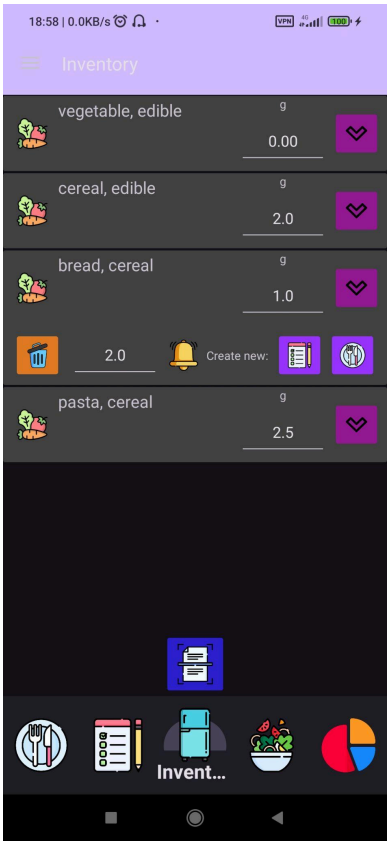
Inventario, lista, pasti

Una certa quantità di un prodotto può essere aggiunta, modificata o rimossa da tre elenchi, con degli appositi *Dialog*:

- la lista della spesa (che non ha particolari funzionalità aggiuntive);
- l'inventario, che permette anche di impostare una soglia (*alert*) per un prodotto in modo che, se la sua quantità posseduta scende al di sotto, si verrà notificati ogni giorno intorno alle 08:00.
- i pasti pianificati, suddivisi per giorno, che mostrano l'orario e i nutrienti di ogni prodotto aggiunto; all'aggiunta di un pasto, infatti, si possono indicare data e ora con appositi *Dialog*.

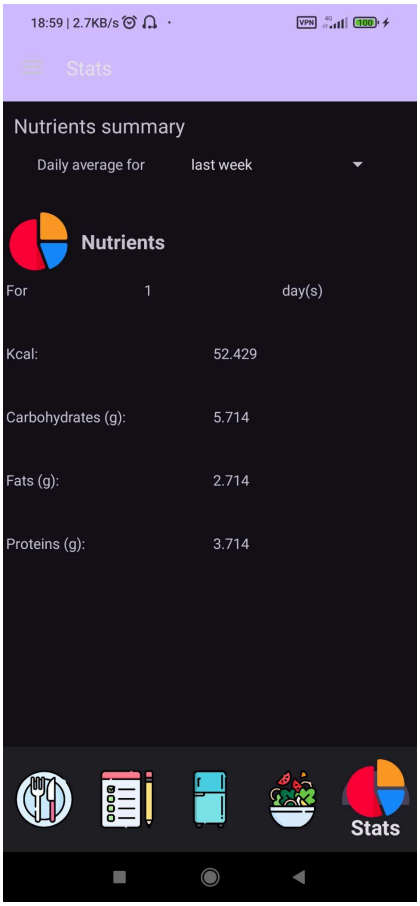






Statistiche

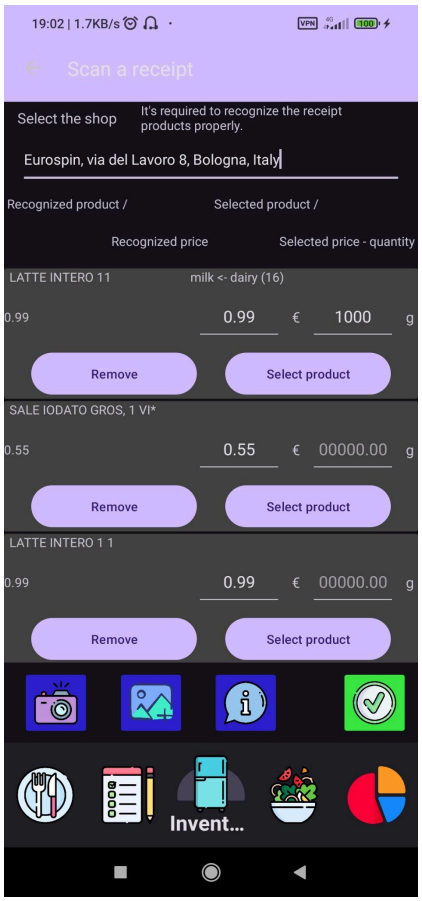
Esiste anche una schermata di statistiche (*stats*), da cui è possibile visualizzare la media dei nutrienti consumati nell'ultima giornata, settimana, mese o anno.



Scansione

Oltre che manualmente, si possono aggiungere prodotti all'inventario registrando una spesa tramite l'immagine di uno scontrino. Dalla schermata apposita (accessibile dall'inventario), bisogna seguire i seguenti passaggi:

1. selezionare un'immagine o scattare una foto
2. verificare che i prodotti e prezzi riconosciuti automaticamente siano corretti: si può rimuovere un elemento, cambiarne la quantità o il prezzo oppure cambiare il prodotto riconosciuto con un altro già salvato
3. solo alla prima volta che viene riconosciuto un nuovo testo sullo scontrino, bisogna associarlo manualmente a un prodotto, in modo che l'applicazione possa ricordarlo per la prossima volta
4. selezionare un negozio, tra quelli salvati (con l'aiuto dell'autocompletamento)
5. confermare

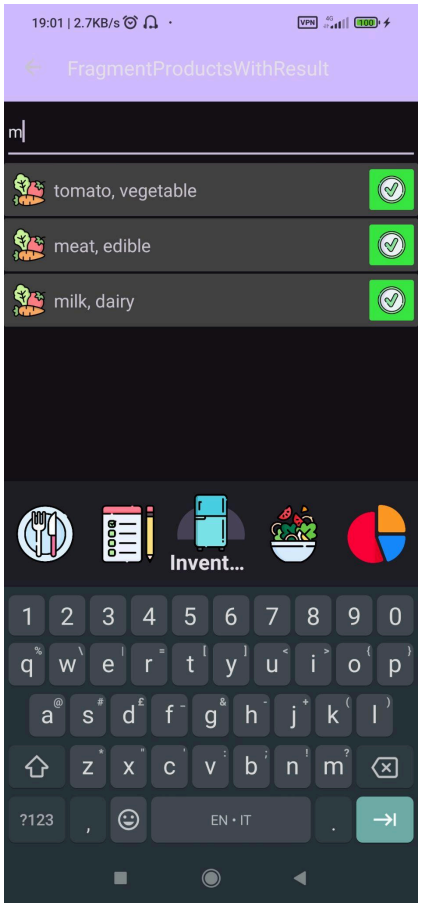


Perché lo scontrino venga riconosciuto il più fedelmente possibile, bisogna prendere alcuni accorgimenti nello scattare la foto:

- l'immagine deve contenere solo l'elenco dei prodotti, altrimenti altre righe compariranno nel risultato e si potranno comunque scartare manualmente; si può dunque piegare prima lo scontrino per tagliare le parti indesiderate, cercare di scattare una foto precisa oppure ritagliare l'immagine successivamente con i propri mezzi;
- poiché le righe di testo devono essere allineate orizzontalmente quanto più possibile, nel fare la foto, bisogna fare attenzione a tenere il telefono parallelo allo scontrino e ad allineare i bordi superiore e inferiore della foto alla prima e ultima riga dello scontrino;
- la foto non dovrebbe essere sfocata, e funziona bene una qualsiasi risoluzione a partire da circa *Full HD* (*1080x1920*, o *2 MegaPixel*);
- con uno scontrino molto lungo, di più di 30-40 righe, potrebbe risultare più difficile scattare la foto e, soprattutto, farla bene, e potrebbe servire anche una risoluzione maggiore (che comunque, tipicamente, non è un problema per i telefoni da diversi anni), per cui potrebbe essere consigliabile dividere lo scontrino su più immagini;
- uno scontrino in buone condizioni è preferibile.

In alternativa, si può ovviamente usare lo screenshot di uno scontrino elettronico preso direttamente dall'applicazione di un negozio.

Si noti infine che il testo riconosciuto dall'applicazione potrebbe apparire strano (per esempio, il numero "1" e la lettera minuscola "l" o "i" potrebbero essere scambiati, oppure potrebbero apparire degli spazi in mezzo a delle parole): in tal caso si può proseguire normalmente a selezionare a mano il prodotto giusto, e anche associare più testi diversi (in scontrini diversi) a uno stesso prodotto non è un problema.



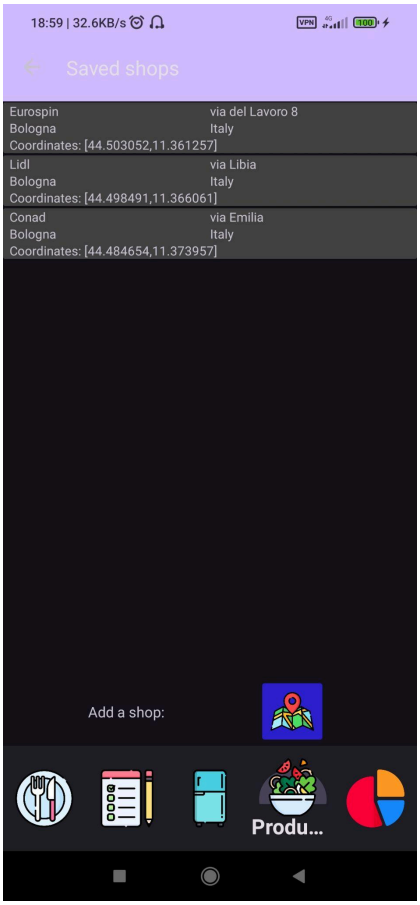
Quando si cerca un cibo da associare a una riga dello scontrino, si apre una schermata di selezione, da cui è possibile visualizzare i nutrienti (solo per un cibo) e selezionarlo con l'apposito tasto.

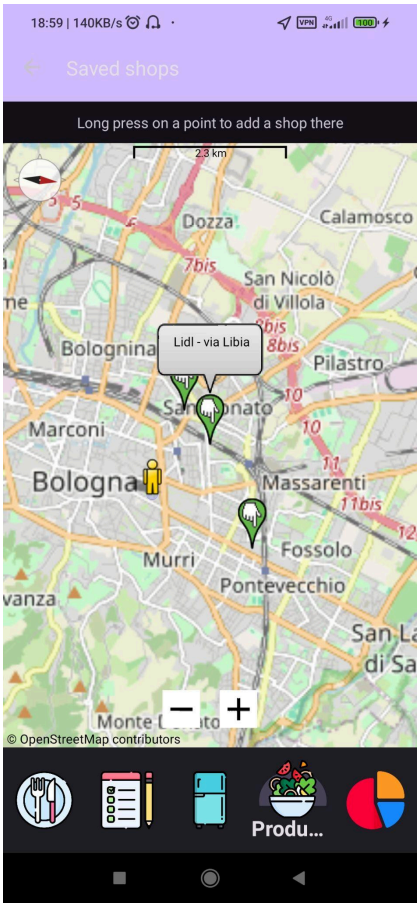
Import/Export

Una ricetta, invece, andando sulla sua visualizzazione, può essere condivisa (come file *json*) con le modalità possibili sul proprio dispositivo (per esempio, anche salvare il file localmente), e importata dalla pagina dei prodotti, selezionando il suo file dalla schermata che appare.

Negozi

Si possono visualizzare i negozi salvati, o aggiungerne altri, attraverso l'apposita pagina, accessibile dal menù laterale.
Un negozio è composto da un *brand* (per esempio *Lidl*), una coppia di coordinate (latitudine, longitudine) e indirizzo, città e Stato. Dalla stessa schermata si può accedere a una mappa, dove si possono vedere i negozi salvati e cliccare su di essi per avere informazioni, oppure se ne può aggiungere un altro cliccando su un altro punto.





Notifiche, impostazioni

Sono previsti due tipi di notifiche:

- gli *alert*, già spiegati, che si attivano automaticamente all'apertura dell'app;
- le notifiche per quando si entra nel raggio di *150m* di distanza da un negozio salvato, che devono essere attivate dalle impostazioni (nel menù laterale).

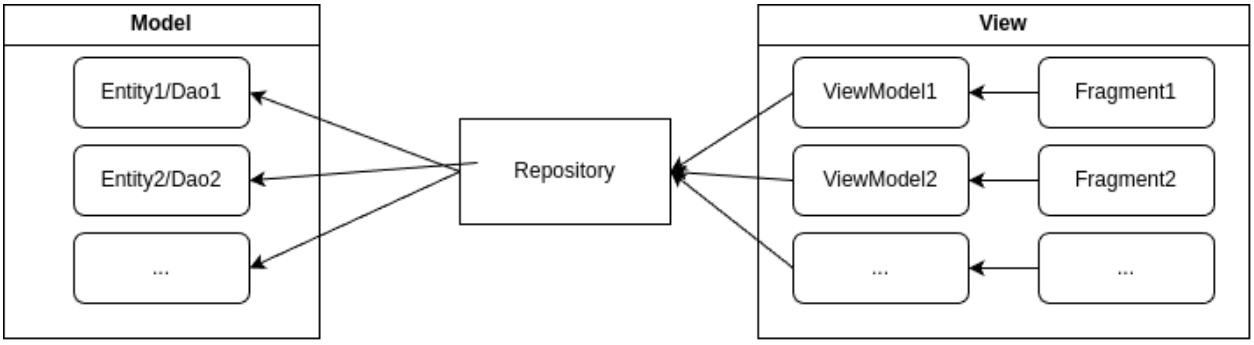
Entrambe le notifiche permettono di aprire l'applicazione cliccandoci sopra, ed entrambe devono essere riattivate se si è riavviato il dispositivo.

Sviluppo

Non avendo avuto altre esperienze di programmazione per *Android*, diverse cose sono cambiate in corso di sviluppo, per esigenze implementative o per allinearsi alle pratiche più diffuse. La base di partenza sono state le demo viste durante il corso - per struttura e organizzazione del progetto - e le lezioni - per una rassegna di oggetti, funzioni, metodologie e convenzioni più comuni -, arricchite via via con le nozioni apprese sul momento per ogni esigenza specifica, cercando di seguire gli approcci più moderni ed efficienti suggeriti dalla documentazione ufficiale.
La scelta di *API 30* come minima versione di *SDK* è dovuta a vari fattori: da un lato, per avere a disposizione le ultime funzionalità di *Android* e un migliore supporto, sembrava una buona idea usare una versione più aggiornata possibile; dall'altro, sarebbe stato comodo avere un dispositivo reale supportato dall'applicazione, sia per testare meglio, sia per poterla utilizzare, eventualmente, in futuro. Inoltre, rendere l'applicazione utilizzabile anche su un sistema operativo più vecchio può costituire un lavoro aggiuntivo, dovendo sempre permettere di utilizzare sia metodi vecchi che moderni. Di qui la scelta di *Android 11* come versione minima, essendo il mio telefono *Android 11*.

Struttura

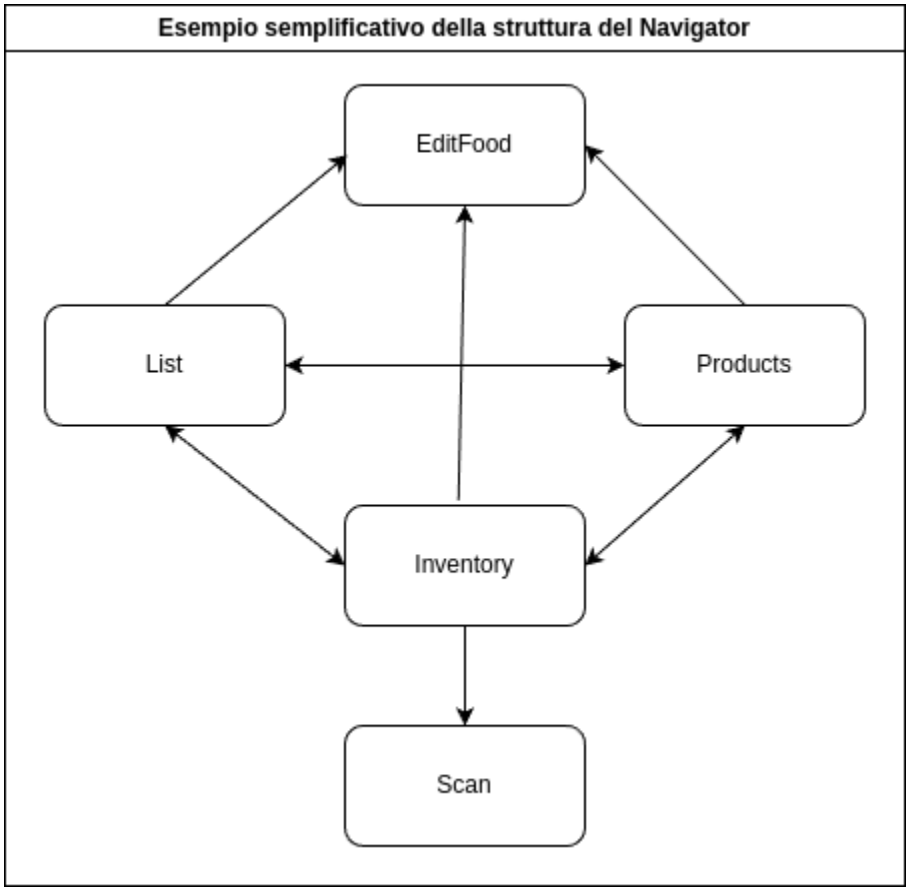
La struttura del progetto segue fondamentalmente il modello Model-View-ViewModel.
Il Model è composto da un Database basato su Room, contenente svariati Entity e Dao, e da una Repository per gestirne l'uso dall'esterno (view).
La View è composta dalle schermate visibili all'utente, e ogni pagina usa il suo ViewModel per ricevere i dati locali del Model filtrati appropriatamente.



View

Poiché l'applicazione ha un gran numero di schermate visibili, si è scelto di usare una singola (Main) Activity e tanti Fragment, coordinati attraverso un Navigator. La *MainActivity*, dunque, si occupa solo di istanziare un *NavController* (che permette ai Fragment di avvicendarsi sulla base di eventi dell'interfaccia) e i due menù che l'utente ha a disposizione per cambiare pagina: una *BottomNavBar* per quelle principali e un menù laterale a scorrimento per le altre, come le impostazioni.
Nello specifico, i frammenti sono:

- quelli per le pagine principali (*meals*, *list*, *inventory*, *products*, *stats*);
- quelli per le pagine secondarie (*settings*, *shops*);
- due per visualizzare o modificare, rispettivamente, un cibo o una ricetta;
- due mappe, di cui una per la ricerca di un prodotto e una per visualizzare i negozi o aggiungerne uno;
- uno per l'aggiunta di uno scontrino;
- uno che può essere invocato per selezionare un prodotto;
- i *Dialog* per aggiungere un prodotto a pasti, lista o inventario, o un negozio.



Le pagine principali hanno tutte un design e delle funzionalità molto simili: ognuno ha una *RecyclerView* che mostra un elenco di prodotti di un determinato tipo (inventario, lista, pasti o tutti), ognuno con varie informazioni aggiuntive e possibili azioni di modifica. Tuttavia, anche se sarebbe stato bello semplificare il codice creando un'unica *View* generalizzata (a cui, eventualmente, effettuare piccole modifiche con l'eredità tra classi), ogni Fragment è implementato indipendentemente, sia perché, con i continui cambiamenti soprattutto nelle prime fasi di sviluppo, era difficile astrarne un'interfaccia che potesse andare bene per tutte, sia perché, in ogni caso, avrebbe richiesto tempo e particolare attenzione per non ridurre la flessibilità del design di ogni elemento, per cui ci si è concentrati sull'implementazione di altre funzionalità e si lasciano le generalizzazioni al futuro.

Tutte le icone utilizzate, utili per rendere più intuibile, compatta, amichevole e semplice (si spera) l'interfaccia, provengono da https://www.flaticon.com/, che le mette a disposizione gratuitamente in cambio di una citazione (presente nel menù, in *About*).

Comunicazione tra Fragment

Alcuni dei frammenti secondari richiedono dei parametri: è il caso dei frammenti che mostrano informazioni specifiche su un solo prodotto, come quelli di editing e la mappa di ricerca. A tale scopo si è sfruttata la funzionalità apposita del *Navigator*.

Il frammento di selezione di un prodotto, invece, ne "ritorna" uno attraverso il suo *ViewModel*, associato alla *MainActivity*, il quale permette ad esso di impostare un valore da restituire, e al fragment chiamante di leggerlo (al suo ritorno).

Mappe

Per le mappe, si è preferito provare a usare *osmdroid* (la libreria di *OpenStreetMap* per android) rispetto a *Maps SDK* di *Google*, essendo open source, gratuito senza limiti d'uso e non richiedendo l'associazione di una carta di credito.
A meno di problemi nell'integrazione iniziale, la libreria è stata facile da usare - per le funzionalità richieste da questa applicazione - seguendo e sperimentando gli esempi da essa forniti.

OCR

Per la *Optical Character Recognition* (OCR), invece, si è tornati dalla parte di Google con la sua libreria *ML Kit*, scelta dopo varie ricerche e tentativi.
Nel suo unico uso nell'applicazione, si è usata la sua funzionalità di *text recognition* mista a una "tecnica algoritmica" per estrarre il testo dalla foto di uno scontrino in maniera corretta: il *recognizer* di *ML Kit*, infatti, suddivide i caratteri presenti nell'immagine in *blocchi* (insiemi di testo più o meno adiacenti), a loro volta suddivisi in *linee* (blocchi con caratteri allineati solo orizzontalmente). Tuttavia il testo estratto, senza ulteriore lavorazione, non appare adatto al nostro scopo: vengono inseriti diversi spazi inesistenti nel mezzo di parole e, soprattutto, i *blocchi* non sono formati nel modo migliore possibile per uno scontrino, poiché, ad esempio, i nomi dei prodotti appaiono completamente scollegati dai relativi prezzi. Di conseguenza, l'algoritmo (con il testo estratto preso in input):

1. identifica una *linea* come nome di prodotto, iva, prezzo, prezzo al peso o al pezzo in base a espressioni regolari;
2. sempre con espressioni regolari, prova a rimuovere spazi inutili;
3. associa le informazioni relative allo stesso prodotto cercando gli insiemi di *linee* che abbiano ognuna il centro che, proiettato orizzontalmente, cade all'interno del *rettangolo* dell'altra (infatti, ogni *linea* è circondata da un *rettangolo*, di cui si hanno le coordinate sull'immagine);
4. nel caso delle righe contenenti prezzo al peso o pezzo, le si associa alla riga appena inferiore se questa è una riga di un prodotto, altrimenti le si scarta.

Si può notare come questo approccio richieda foto scattate con una certa precisione, per fare in modo che tutte le righe siano effettivamente orizzontali, ma negli esperimenti personali non ci sono state molte difficoltà a far riconoscere uno scontrino (ovviamente non si può sperare che venga riconosciuta una foto fatta male o di fretta). Sono stati provati anche altri approcci per riconoscere una riga, ma questo è risultato il migliore:

- proiettare il centro di ogni *rettangolo*, invece che orizzontalmente, seguendo l'inclinazione media del *rettangolo* stesso, produce risultati peggiori;
- creare righe come catene di *linee* adiacenti associate a due a due (con uno dei due metodi di proiezione di un centro dentro l'altra), invece che come insieme di *linee* che siano tutte associate le une con le altre, tende a unire più righe consecutive in una sola.

Un'altra limitazione è che bisogna inquadrare solo i prodotti (o ritagliare l'immagine prima), poiché l'algoritmo non prevede ulteriore processamento, anche se sarebbe stato comodo riconoscere automaticamente altre informazioni nell'intestazione dello scontrino. In ogni caso, l'utente potrebbe sempre scartare righe inutili dall'output visualizzato a schermo.

Ovviamente, le intelligenze artificiali sono uno strumento molto potente, e una formulata appositamente potrebbe svolgere il compito molto meglio, ma ci si è accontentati di un approccio "sufficiente" per poter lavorare anche su tutto il resto del progetto. Potrebbe anche essere che *ML Kit* fornisca già dei parametri che, modificati a dovere, permettano di ottenere risultati migliori, ma non c'è stato tempo di approfondire.

Si precisa poi che, seppur riconosciuti, i prezzi per peso o pezzi non sono al momento utilizzati nell'implementazione, ma si spera di includerli in futuro per migliorare l'esperienza (e, per esempio, anche per cercare il miglior prezzo per quantità invece del solo prezzo assoluto).
Un'altra aggiunta possibile per l'applicazione potrebbe prevedere la scansione anche delle etichette dei valori nutrizionali dei prodotti.

Import/Export

Le funzioni di importazione ed esportazione di una ricetta in *json* sono gestite da una classe apposita. L'unico punto critico qui sta nella sincronizzazione di ricette su dispositivi diversi, con database locali diversi, che avviene nel *ViewModel* delle ricette dopo aver importato una ricetta. Si potrebbero avere conflitti se, per esempio, entrambi hanno una ricetta con quel nome, che potrebbe essere la stessa, con gli stessi ingredienti (nel cui caso non si importa nuovamente la ricetta e se ne notifica l'utente), oppure una diversa (quindi si importa la ricetta cambiandone il nome, aggiungendo un numero alla fine e notificando l'utente); inoltre, bisogna eseguire gli stessi procedimenti per ogni ingrediente, e aggiungerlo nel caso in cui non esista o abbia diversi nutrienti. Per semplicità, con l'implementazione attuale, se una ricetta ha un'altra ricetta tra i suoi ingredienti, questa viene "convertita" in un cibo - altrimenti, in futuro, si potrebbe espandere l'importazione/esportazione per supportare questa ricorsione.
Per l'accesso ai file, essendo file generici, si usa un *FileProvider*, definito con una propria sottoclasse essendo una soluzione più stabile, secondo la documentazione.

Servizi

Come già visto, *HomeCook's companion* interagisce con l'utente anche da chiusa, in due modi: quando questo si avvicina a un negozio salvato e quando un prodotto nell'inventario è sotto la soglia dell'*alert* impostato.
Per la prima esigenza, dall'interno dell'applicazione (aperta) si registrano dei *geofence*, uno per ogni negozio salvato. Le geofence sono gestite direttamente da android, che, alla loro attivazione, invoca un BroadcastReceiver registrato, il quale invia una notifica al dispositivo. Per l'attivazione di geofence per nuovi negozi appena aggiunti è necessario premere nuovamente il bottone di attivazione, che registra nuovamente tutti i negozi salvati: usando sempre lo stesso id per uno stesso shop, i vecchi geofence verranno semplicemente sovrascritti. Inoltre, essendoci un limite massimo di geofence registrabili per applicazione (100), la lista di negozi salvati viene, al momento, tagliata: comunque, probabilmente, un utente difficilmente aggiungerà tanti *negozi*.
Per gli *alert*, invece, si lancia un *job* periodico per il *WorkManager*, che, ogni giorno intorno alle 08:00, controlla se ci siano prodotti nell'inventario sotto la soglia e, nel caso, invia una notifica. Per il momento, il *job* ha bisogno di un primo avvio dell'applicazione dopo ogni volta che il dispositivo è stato acceso: in futuro, sarà sicuramente utile registrare un *BroadcastReceiver* che sia chiamato all'accensione (con permesso *RECEIVE_BOOT_COMPLETED*).

Si osservi, infine, che android, dalle ultime versioni dell'*API*, richiede espressamente che la notifica per le *geofence* sia attivata dall'interno dell'applicazione, in modo che l'utente sia consapevole di star dando il permesso di eseguire un'attività di geolocalizzazione in background, e necessariamente dopo che siano già stati dati i permessi meno restrittivi sulla *location* (*fine/coarse location*). Con il *WorkManager*, invece, non ci sono tali restrizioni, essendo gestito direttamente dal sistema operativo e garantendo meno poteri al servizio da eseguire.

Permessi, intent

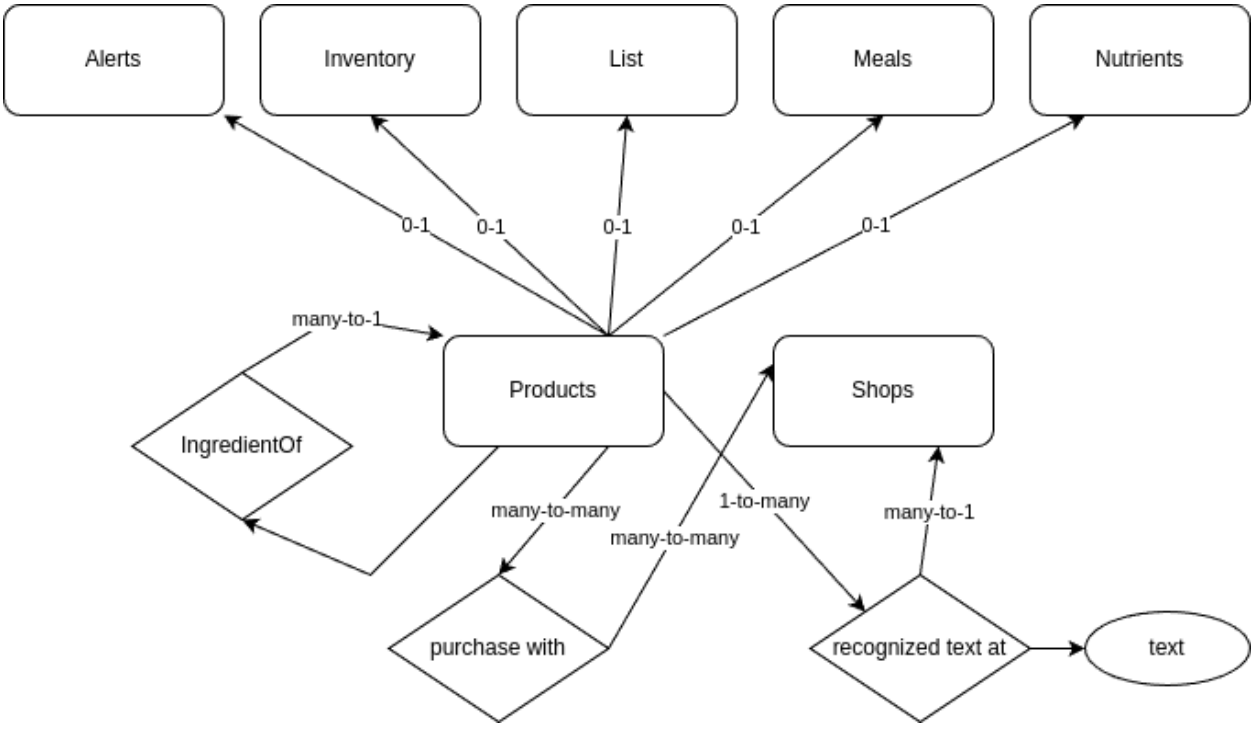
Infine, gli svariati permessi richiesti dall'applicazione, quando non possono essere semplicemente inclusi nel *Manifest* per politiche di *Android* (come la geolocalizzazione che abbiamo appena visto), sono gestiti con i moderni *ActivityResultContract*, che controllano che un permesso sia stato garantito e, in caso positivo, lanciano la funzione specificata; i permessi quindi, nell'implementazione, vengono richiesti appena prima di lanciare un contratto.
Anche gli *Intent*, come la telecamera per la scansione e la selezione di un file da importare come ricetta, usano i *Contract*.

Model

Entity

Ogni prodotto (edibile, ricetta o nessuno dei due) è rappresentato da un'*entry* nella tabella *Products*, e identificato da un *id* univoco; allo stesso modo, *Shops* e *Purchases* contengono, insieme a un *id*, rispettivamente, ogni negozio e ogni acquisto salvato.
Gli *id* sono così usati come *foreign key* per memorizzare gli elementi nelle tabelle per lista, inventario, pasti, nutrienti (associati a un cibo), *IngredientOf* (che collega una ricetta ai suoi

ingredienti), *Alerts* (con l'alert impostato per ogni prodotto), *Purchases* (che può associare uno o più *shop* a uno o più prodotti) e *ProductRecognized* (per associare un prodotto a uno o più testi riconosciuti con *OCR*, e un testo riconosciuto, per un dato negozio, a non più di un prodotto). I nutrienti vengono memorizzati anche per le ricette, per evitare di doverli ricalcolare in base agli ingredienti ogni qual volta si vogliano conoscere; ciò ovviamente implica che, a ogni aggiornamento di ricetta o ingrediente, si debba aggiornare anche la tabella *Nutrients*: se ne occupa la *Repository* con un metodo univoco per inserire o aggiornare una ricetta con tutti i suoi ingredienti, mediando l'accesso dei *ViewModel* al *Database*.
Oltre a queste entità principali, sono presenti diverse relazioni (di *Room*) per semplificare accessi, inserimenti, modifiche a più tabelle contemporaneamente: per esempio, i prodotti in *List*, *Inventory*, *Meals* vengono sempre mostrati con il nome, che viene salvato solo in *Products*, e l'inventario mostra sempre gli elementi di *Inventory* insieme a quelli di *Alerts*, e *Meals* insieme a *Nutrients*.



Funzioni di accesso

Ogni *Entity* può essere acceduta tramite il suo *Dao*, e la *Repository* cerca di mediare il loro uso da parte della *View*, permettendo anche di definire alcune query un po' più complesse dei *Dao* che coinvolgano più tabelle.
I valori di ritorno sono tipicamente *LiveData*, convenienti per l'interfaccia così che si possa tenere aggiornata in maniera asincrona; tuttavia esistono anche metodi non-*LiveData*, per esigenze specifiche o di sincronizzazione,come per le parti di codice che vengono già eseguite in thread di background e non hanno bisogno di *observer*: è il caso del *Worker* per le notifiche per gli *alert*, che non deve fare altro che aspettare i dati dal database prima di decidere se e

come lanciare la notifica, e dell'importazione di una ricetta, che viene lanciata - dalla *View* - come funzione in background - nel *ViewModel* -, e il cui risultato si può osservare con il *MutableLiveData importResult*.
Con la discreta complessità del database e tanti *LiveData* da dover gestire, è tornata molto utile la loro funzione *switchMap*, che permette di annidare chiamate che ritornino *LiveData* in modo che il cambiamento di un valore di ritorno (in un livello di annidamento) più esterno provochi, a cascata, l'aggiornamento di tutte le chiamate innestate. Oltre che in poche funzioni della *Repository*, gli *switchMap* sono stati ampiamente utilizzati in diversi *ViewModel*. Per esempio, quello per la ricerca dei prezzi migliori per un prodotto, per effettuare la chiamata alla *Repository* che ritorni i negozi giusti, deve tenere in conto del raggio selezionato, prodotto e posizione attuale: dunque, ognuno di questi parametri rappresenta un *MutableLiveData*, che può essere aggiornato dalla *View* in base alle interazioni dell'utente, in modo da poter notificare il cambiamento anche alla query; poiché questo annidamento di *switchMap* è una funzione del *ViewModel*, alla *View* basta solo continuare ad osservare il suo valore di ritorno. Inoltre, per evitare aggiornamenti inutili, si è cercato di mantenere i parametri aggiornati meno frequentemente in un livello di annidamento più esterno possibile (per esempio, i negozi salvati non cambieranno mentre l'utente è sulla mappa, ma la sua posizione o il raggio di visualizzazione selezionato potrebbero farlo con molta probabilità).

Testing

Il testing, dopo l'accurata scelta della minima versione dell'*API Android*, è avvenuto principalmente sul dispositivo personale, e poco, soprattutto all'inizio, con *AVD*. Oltre che per esigenze hardware, visto che la quantità di *RAM* richiesta dall'emulatore portava spesso il computer a crashare (a volte per il fisso da *16GB*, sempre per il portatile da *10GB*), usare un telefono reale ha permesso un testing migliore, in tutte le funzionalità dell'applicazione, più vicino all'uso che un utente potrebbe farne.
Dunque, si è cercato di provare ogni interazione possibile prevista dall'app per verificare che fosse eseguita correttamente e non desse errori.
Per la scansione di scontrini, sono state fatte diverse prove per vedere come cambiasse il risultato con scontrini diversi, più o meno deteriorati, in formati diversi, con foto con caratteristiche diverse; ciò è servito a delineare l'algoritmo e le istruzioni per il suo corretto utilizzo, già visti. Inoltre, degli esperimenti sono serviti anche a cercare di migliorare i risultati, anche perché la documentazione di *ML Kit* consiglia di avere un numero minimo di pixel per carattere e suggerisce che, a volte, una risoluzione troppo alta potrebbe comportarsi peggio: personalmente, sembra che tutte le foto di scontrini con una risoluzione a partire dal *Full HD* (*1080x1920*, o *2 MegaPixel*) vadano ugualmente bene (ma risoluzioni inferiori sono peggiori), considerando che la foto di uno scontrino avrà più o meno sempre le stesse dimensioni (con uno scontrino non esagerato, fino a massimo 30-40 righe).
Le parti più difficili da testare sono state i due tipi di notifiche, poiché non erano immediati, richiedevano vari passaggi, attese e avevano un debugging più difficile, dovendoli testare anche con applicazione chiusa e senza logging. In particolare, le notifiche periodiche con il *WorkManager* avevano il problema che non possono essere schedulate per meno di 15 minuti, per cui richiedevano di attendere un po' tra un test e l'altro; per le *geofence*, invece, è stato

molto utile usare un'applicazione di *mock location* per cambiare la posizione artificialmente - quella usata è *Lockito*, disponibile su *Play Store*.

Per finire il database, al momento, viene popolato automaticamente con pochi valori per ogni tabella, per testare.