PROGRAMMAZIONE PER DISPOSITIVI MOBILI
Anno 2024/2025

SmartShop
Documentazione

Università di Torino
Michele Cascione **978699**, Pierluigi Boscaglia **983553**
michele.cascione@edu.unito.it pierluigi.boscaglia@edu.unito.it

Indice

1 Introduzione

L'applicazione **SmartShop** è stata pensata per essere utilizzata in un **supermercato** che necessita di **sincronizzazione istantanea** per fornire i seguenti servizi.
Per il cliente:

- **Spesa online** con richiesta di ritiro nel **supermercato** o consegna a casa.

Per i dipendenti e i responsabili:

- **Gestione degli ordini** in arrivo e conclusi
- **Sincronizzazione istantanea** dei prodotti presenti tra gli scaffali del **supermercato** e nel **magazzino**;
- Possibilità di gestire il **magazzino** ed effettuare dei **riordini** dai diversi fornitori.

Tutti gli utenti disporranno della possibilità di **registrarsi** e modificare il proprio **account** e visualizzare lo **storico** degli ordini effettuati. La **sincronizzazione** tra scaffali del supermercato, magazzino, ordini e riordini della merce è gestita accuratamente grazie ad un **server** ed un **database**. Quest'ultimo è stato costruito con l'intento di poterlo utilizzare su app di larga scala.

2 Requisiti

L'applicazione si basa sull'utilizzo di **3 UI** diverse in base al tipo di utente che ha eseguito l'accesso. Per ogni utente saranno disponibili diverse sezioni e azioni da poter eseguire. Gli utenti che potranno utilizzare l'applicazione saranno: **Cliente**, **Dipendente** e **Responsabile**.

2.1 Registrazione & Login

Prima di accedere alle varie sezioni dell'app è possibile effettuare il **Login** o la **Registrazione**. La registrazione è pensata per essere usata principalmente dai **Clienti**: ogni utente registrato sarà salvato nel database come cliente. Il gestore del DB può accedervi e permettere agli utenti destinati ad essere **"Admin"** (**Responsabili** e **Dipendenti**) di passare al ruolo più alto. Dopo aver effettuato correttamente il **login**, l'applicazione ci reindirizzerà alla schermata associata al nostro account.

2.2 Cliente

Il **cliente** grazie alla **barra di navigazione** potrà viaggiare nelle seguenti schermate: **Catalogo** (voce "Ordina"), **Storico** e **Account**

2.2.1 Catalogo

Il **catalogo** è una delle colonne portanti dell'applicazione, esso infatti rappresenta tutti i **prodotti** che il cliente può ordinare. La quantità dei prodotti è sincronizzata grazie al **database** e all'intervento dei dipendenti nell'aggiornare la quantità dei prodotti presenti negli scaffali del negozio.

Schermata Principale, Filtri, Menù Categorie & Ricerca
Il catalogo si presenta come un elenco di prodotti mostrati con i dati utili (nome, marca, prezzo, **tag** e foto). Ogni prodotto potrà essere aggiunto al **carrello**, oppure sarà possibile cliccarci per visualizzare tutti i suoi dati nel dettaglio. I prodotti non saranno aggiunti al carrello se questi fossero non disponibili (e quindi non presenti tra gli scaffali del negozio).
I tag associati ad ogni prodotto permetteranno all'utente di filtrare per determinati prodotti (vegani, eco friendly, bio, proteico...) e di mostrare solo i prodotti in **offerta**.
Il **menù Categorie** permetterà all'utente di visualizzare i prodotti associati ad una singola categoria (esempio: Casa → Detersivi | Pesce → Crudo |
Frutta → Esotica | Carne → Pollo). Sarà possibile tornare alla situazione originale cliccando **"Tutte le categorie"**.
Una **box di testo** può essere usata dal cliente per digitare il prodotto che sta cercando, così da visualizzarlo nella categoria in cui si trova.

Preferiti e Carrello
Posti di fianco al menù categorie il cliente troverà due icone: un **cuore** e un **carrello della spesa**.

- **Preferiti** ❤: grazie all'uso della **memoria locale**, il cliente potrà tenere in una **sezione dedicata** i prodotti che compra spesso, semplicemente cliccando il cuore associato ad ogni prodotto all'interno del catalogo. Entrando nella sezione preferiti sarà possibile inoltre aggiungere tutti i prodotti salvati all'interno del **carrello**.

- **Carrello** 🛒: il carrello mostrerà al cliente un riepilogo di tutti i prodotti aggiunti, permettendogli di aggiungere o rimuovere ogni prodotto. Il cliente potrà scegliere se ritirare il suo ordine direttamente nel supermercato all'interno di un **locker** (così da saltare la fila) oppure optare per la **consegna a domicilio**, che utilizzerà l'indirizzo associato al suo account.

2.2.2 Storico

Lo **storico** conterrà tutti gli **ordini** effettuati dal cliente, suddividendoli tra quelli ancora in elaborazione e quelli conclusi. Per gli ordini in **elaborazione** il dipendente potrà aggiornare lo **stato dell'ordine** (in preparazione, spedito, annullato, concluso): il cliente potrà visualizzare tali aggiornamenti in tempo reale. Per gli ordini da assegnare al **locker**, il dipendente porrà la spesa nel primo locker disponibile associatogli dall'applicazione. Dopo che l'ordine sarà stato preparato, il cliente potrà cliccare "**Mostra QR**" così da effettuare in sicurezza il ritiro del suo ordine. L'ordine sarà segnato come **concluso** dopo il ritiro.
Per quanto rigaurda gli ordini a domicilio, il dipendente potrà segnare l'ordine come "**Consegnato**" (e quindi sarà concluso) quando riceverà dal rider che la consegna è stata effettuata con successo.
Nella sezione "**Conclusi**", il cliente sarà in grado di visualizzare ordinati per data tutti gli ordini effettuati, con un riepilogo delle informazioni e dei prodotti comprati.

2.3 Dipendente

Il **dipendente** è colui che si occupa della **gestione degli ordini** mandati dai clienti: dopo aver preso in incarico un ordine, si occupa di prepararlo usufruendo di **SmartShop**.

Infatti, dopo l'**assegnazione** dell'ordine, il responsabile potrà utilizzare la **mappa interattiva** del negozio che lo guiderà negli scaffali dove sono presenti i prodotti dell'ordine. Grazie ad una **CheckList** associata alla mappa, potrà segnare di volta in volta i prodotti di cui ha concluso la preparazione, per poi aggiornare lo stato dell'ordine. Grazie alla **barra di navigazione** potrà accedere alle diverse schermate.

2.3.1 Mappa & Assegnazione Ordine

La **mappa** permette al dipendente di orientarsi meglio all'interno del negozio, infatti ad ogni **scaffale** del negozio presente sulla mappa sarà associata un'**area cliccabile**. Al click di un'area, il dipendente potrà visualizzare tutti i prodotti disponibili in quello scaffale e visualizzarne i dettagli. Grazie al bottone "**Scegli un ordine**" sarà reindirizzato alla voce **Assegna** della barra di navigazione che gli permetterà, tra un elenco di ordini da gestire, di prenderne uno in carico. A seguire la presa in carico di un ordine è presente il reindirizzamento alla mappa, dove la sezione dell'ordine sarà riempità. Più in dettaglio, verranno visualizzati tutti i prodotti associati all'ordine, la **quantità** per ogni prodotto, e la **corsia/scaffale** associata a quell'ordine. Il pulsante "**Vedi**" associato ad ogni prodotto attiverà lo scroll automatico verso la mappa, nella quale sarà evidenziato lo scaffale/corsia dove è riposto il prodotto interessato. La **CheckList** permetterà di segnare tutti i prodotti preparati. Quando tutti i prodotti saranno preparati, il completamento della CheckList permetterà al dipendente di segnare il prossimo stato dell'ordine (**Pronto per il ritiro**, **Spedito**, **Consegnato**).

2.3.2 Storico

Lo **storico** del dipendente, diviso in ordini "**Da Gestire**" e "**Conclusi**", permette di visualizzare tutti gli ordini e le relative informazioni. Per quelli da gestire è visualizzato se questi sono appena stati creati o sono in attesa di essere ritirati/conclusi. Gli ordini conclusi mostrano lo storico degli ordini che sono stati "chiusi" con il relativo stato: "**Concluso**" per gli ordini ritirati al **locker**, "**Consegnato**" per gli ordini con consegna.

2.4 Responsabile

Il **responsabile** è colui che si occupa della **gestione del magazzino**. E' affidato a lui il compito di **riordinare** i prodotti che stanno per terminare, selezionare i **fornitori** per ogni prodotto e segnare nel sistema lo **spostamento** di prodotti dal **magazzino** agli **scaffali**, così da mantenere costante la sincronizzazione. Il responsabile potrà eseguire queste operazioni utilizzando le schermate raggiungibili dalla **barra di navigazione**.

Riordine

La sezione **Riordine** è utilizzata per ordinare una **quantità precisa** di un prodotto da un **fornitore** selezionato, che una volta arrivata sarà aggiunta all'interno del **magazzino**. Il **menù "Categorie"** e il **tasto "Cerca"** funzionano allo stesso modo del catalogo; la differenza è che qui vengono mostrati al responsabile non solo le info di ogni prodotto ma anche le **quantità** di questo sia nel magazzino che tra gli scaffali del supermercato.
Una volta selezionato il prodotto desiderato e consultata la quantità all'interno del magazzino, il responsabile può riordinarlo selezionando uno dei **fornitori disponibili** e inserendo la quantità di prodotto desiderata.

Storico

Lo **storico** permette al responsabile di visualizzare i **riordini** che sono stati creati ma non sono ancora stati consegnati al supermercato, e i riordini che sono arrivati e si trovato nelle **giacenze** del magazzino. Ogni ordine mostrato contiene un riepilogo di tutte le informazioni.

Trasferisci

Tutto ciò è finalizzato dalla schermata "**Trasferisci**". La prima sezione è la stessa che è stata introdotta nello storico: un **menù**, una **barra di ricerca** e dei prodotti con le relative **quantità** in **magazzino** e nel **catalogo** (tra gli scaffali).
La sezione centrale della pagina cambia: viene mostrata la **mappa del negozio** usata dal dipendente. Rispetto a quest'ultimo che la usava per preparare con facilità gli ordini, il responsabile ha il compito di popolare il catalogo, e quindi segnare nel sistema uno **spostamento** di una quantità di prodotti dal magazzino ad uno **scaffale/corsia** specifico/a del supermercato, selezionabile grazie alla mappa.

2.5 Gestione Profilo e LogOut

Tutti gli utenti avranno accesso ai dati del proprio **account** cliccando sulla propria foto profilo o mail della barra superiore dell'app, oppure cliccando sulla voce **Profilo** ⚙ presente nella **navbar inferiore** (entrambi i metodi portano alla stessa schermata). I dati come nome, indirizzo, foto profilo ecc. potranno essere cambiati e salvati nel **server**. Nella parte destra della navbar superiore il tasto escì permetterà agli utenti di uscire dal proprio account e tornare alla pagina di **Login** e **Registrazione**.

2.6 Aggiorna

Ultimo requisito importante per il funzionamento dell'app è il **pulsante aggiorna** che rieffettuerà le **query** al **database** e l'**aggiornamento istantaneo** via UI per mostrare se alcuni dati sono cambiati (es. ordine di prodotto disponibile fallito = prodotto non disponibile ma non aggiornato in UI).

Catalogo
Aggiornamento delle quantità dei prodotti presenti tra gli scaffali del negozio;

Storico & Assegnazione
Aggiornamento di ordini/riordini per controllare se il loro stato è stato cambiato (In Preparazione, In Consegna, Creato, Consegnato...) o se ve ne sono di nuovi;

Riordina & Trasferisci
Aggiorna la quantità di tutti i prodotti presenti sia nel magazzino che tra gli scaffali del supermercato. Utile per verificare che lo spostamento dei prodotti magazzino→catalogo siano andati a buon fine nel sistema, oppure per verificare se il sistema ha registrato l'arrivo della merce arrivata tramite il riordino.

2.7 Requisiti Non Funzionali

Il database e il server che ne permette l'interfacciamento permettono all'applicazione una **sincronizzazione in tempo reale**. Tutti i dati vengono salvati istantaneamente sul database e indicizzati grazie all'uso di **tabelle** e **chiavi esterne**, permettendo di fare anche query complesse. Con 3 istanze dell'applicazione su 3 dispositivi diversi, basterà cliccare il tasto aggiorna dopo l'azione di un utente per verificare che i dati sono stati istantaneamente salvati nel **DB**. Questa soluzione evita che si verifichino errori gravi, come l'ordine di un prodotto che è stato esaurito, oppure lo spostamento tra magazzino e catalogo di prodotti che nel magazzino non ci sono.
Il progetto è stato sviluppato per essere compatibile a partire **Android 8 (SDK 26)** fino all'ultima versione, **Android 16 (SDK 36)**. L'interfaccia è stata costruita per essere utilizzata in modo facile ed intuitivo grazie all'interfaccia **user-friendly**: le barre di navigazione nelle parti inferiore e superiore hanno aiutato in questo intento.
Le mappe sono state una soluzione che ci ha permesso di sviluppare un'interazione facile con gli utenti facenti parte del personale del supermercato.

3 Design

ENTRY POINT

MainActivity.kt

Inizializzazione dell'app: Login & Registrazione con successivo instradamento verso lo Screen (ViewLayer) corretto.

VIEW LAYER

```
ui/screens/
  ├ LoginScreen.kt
  ├ CatalogScreen.kt
  ├ EmployeeScreen.kt
  ├ ManagerScreen.kt
  └ AccountSettingsScreen.kt

ui/theme/
  ├ Color.kt
  ├ Theme.kt
  └ Type.kt

ui/components/
  ├ ProductImage.kt
  ├ StoreMapComposable.kt
  └ NavBarDecor.kt

ui/map/
  ├ SupermarketPolygons.kt
  ├ MapJsonLoader.kt
  └ MapImageLoader.kt
```

In questo Layer risiedono: Screen (Schermate) differenti dell'applicazione; Componenti utilizzati per mostrare immagini e mappa interattiva; Palette colori e Tipografia.

VIEW MODEL LAYER

```
viewModel/
  ├ LoginViewModel.kt     →  LoginUiState
  ├ CatalogViewModel.kt   →  CatalogUiState
  ├ EmployeeViewModel.kt  →  EmployeeUiState
  ├ ManagerViewModel.kt   →  ManagerUiState
  └ AccountPreferencesViewModel.kt  →  AccountState
```

ViewModelLayer. Si occupa di: Logica di presentazione; Ricezione dei dati dal Model Layer; Notifica il View Layer di aggiornare i dati (StateFlow Usage).

ModelLayer. Layer robusto che si interfaccia con: Database Locale App (Room); Server Express collegato a DB SQL; Datastore (Preferenze Utente e sessione Login). L'interfacciamento è permesso dai Repository e le Entities.

MODEL LAYER

REPOSITORY

```
data/repository/
  ├ AuthRepository.kt       →  Login, Registrazione
  ├ ProductRepository.kt    →  Gestione prodotti
  ├ CategoryRepository.kt   →  Gestione categorie
  ├ OrderRepository.kt      →  Gestione ordini
  ├ SupplierRepository.kt   →  Gestione fornitori
  ├ RestockRepository.kt    →  Richieste rifornimento
  ├ InventoryRepository.kt  →  Inventario magazzino
  ├ ShelfRepository.kt      →  Gestione scaffali
  └ UserRepository.kt       →  Profilo utente
```

DATASTORE

```
data/datastore/
  ├ SessionDataStore.kt       (Sessione utente corrente)
  ├ FavoritesDataStore.kt     (Prodotti preferiti)
  └ AccountPreferencesDataStore.kt (Settings profilo)
```

DOMAIN

```
domain/
  └ UserRole.kt   (enum: CUSTOMER, EMPLOYEE, MANAGER)
```

REMOTE DATA SOURCE

```
data/remote/
  ├ RetrofitInstance.kt
  └ SmartShopApiService.kt
```

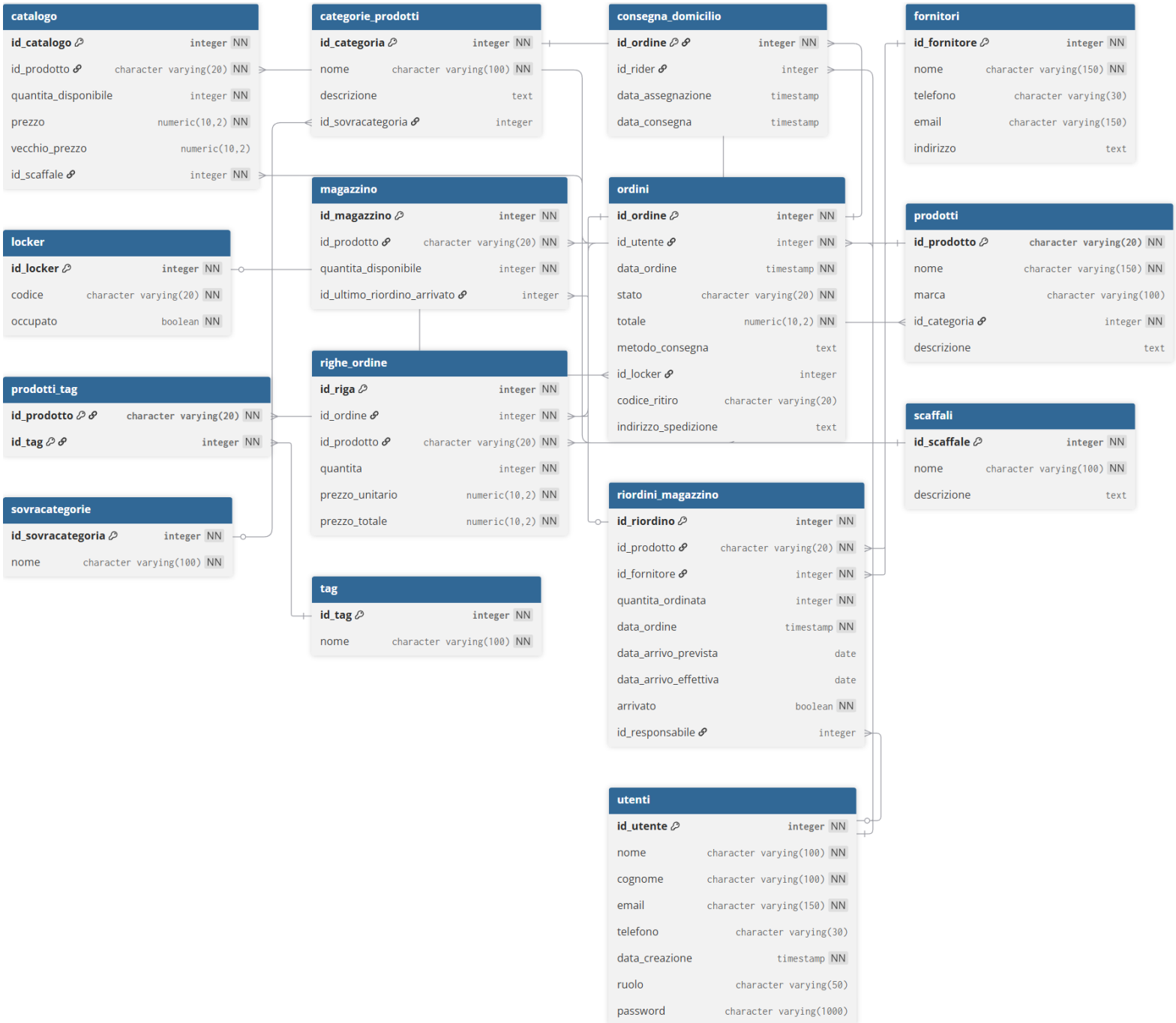
LOCAL DATA SOURCE

```
data/database/
├ SmartShopDatabase.kt
└ Converters.kt

data/dao/
├ ProductDao.kt
├ CategoryDao.kt
├ OrderDao.kt
├ SupplierDao.kt
├ RestockDao.kt
└ ShelfDao.kt
```

ENTITY

```
data/entity/
  ├ User.kt          (@Entity)
  ├ Product.kt       (@Entity)
  ├ Category.kt      (@Entity)
  ├ Order.kt         (@Entity)
  ├ Supplier.kt      (@Entity)
  ├ Restock.kt       (@Entity)
  ├ Shelf.kt         (@Entity)
  ├ StockTransfer.kt (Data class - DTO)
  ├ MapPolygon.kt    (Data class - DTO)
  └ UpdateUserRequest.kt (Data class - DTO per API)
```


catalogo
id_catalogo integer NN
id_prodotto character varying(20) NN
quantita_disponibile integer NN
prezzo numeric(10,2) NN
vecchio_prezzo numeric(10,2)
id_scaffale integer NN
categorie_prodotti
id_categoria integer NN
nome character varying(100) NN
descrizione text
id_sovracategoria integer
consegna_domicilio
id_ordine integer NN
id_rider integer
data_assegnazione timestamp
data_consegna timestamp
fornitori
id_fornitore integer NN
nome character varying(150) NN
telefono character varying(30)
email character varying(150)
indirizzo text
magazzino
id_magazzino integer NN
id_prodotto character varying(20) NN
quantita_disponibile integer NN
id_ultimo_riordino_arrivato integer
ordini
id_ordine integer NN
id_utente integer NN
data_ordine timestamp NN
stato character varying(20) NN
totale numeric(10,2) NN
metodo_consegna text
id_locker integer
codice_ritiro character varying(20)
indirizzo_spedizione text
prodotti
id_prodotto character varying(20) NN
nome character varying(150) NN
marca character varying(100)
id_categoria integer NN
descrizione text
locker
id_locker integer NN
codice character varying(20) NN
occupato boolean NN
righe_ordine
id_riga integer NN
id_ordine integer NN
id_prodotto character varying(20) NN
quantita integer NN
prezzo_unitario numeric(10,2) NN
prezzo_totale numeric(10,2) NN
prodotti_tag
id_prodotto character varying(20) NN
id_tag integer NN
scaffali
id_scaffale integer NN
nome character varying(100) NN
descrizione text
riordini_magazzino
id_riordino integer NN
id_prodotto character varying(20) NN
id_fornitore integer NN
quantita_ordinata integer NN
data_ordine timestamp NN
data_arrivo_prevista date
data_arrivo_effettiva date
arrivato boolean NN
id_responsabile integer
sovracategorie
id_sovracategoria integer NN
nome character varying(100) NN
tag
id_tag integer NN
nome character varying(100) NN
utenti
id_utente integer NN
nome character varying(100) NN
cognome character varying(100) NN
email character varying(150) NN
telefono character varying(30)
data_creazione timestamp NN
ruolo character varying(50)
password character varying(1000)

4 Implementazione

Di seguito vengono riportate le **tecnologie utilizzate** per la realizzazione dell'app, seguendo la suddivisione a partire da: **Model**, **ViewModel** & **View**.

4.1 Model Layer: Interfacciamento con Dati Locali ed Esterni

Progettazione del DB
L'obiettivo da raggiungere durante la realizzazione di questo progetto è stato quello di realizzare un **sistema perfettamente sincronizzato** e user-friendly, pensato per un utilizzo reale all'interno di un supermercato.
La prima parte, che ha ricoperto gran parte dell'operato, è stata quella di progettare un **Database SQL** (usando **PgAdmin4**) che sfruttando vincoli di **chiavi primarie**, **esterne**, e tutti i tipi di **relazioni n x n**, rappresentasse una base solida per l'applicazione.

Server Express
Seguendo le linee guida di IUMTWEB, abbiamo implementato un **server Node.js** con framework **Express**. Questo ci ha permesso di:

- **Mantenere** le foto dei prodotti e dei profili registrati in un server sempre raggiungibile;
- Configurare l'accesso al **DB PostGres** locale;
- Definire le **Route**, utili per il recupero di dati dal database grazie alle **Query** e la relativa gestione degli errori;
- Definire una documentazione **Swagger** accessibile da browser per testare e rendere trasparenti le Routes disponibili.

Retrofit
Le librerie **Retrofit** e **okHttp** sono state usate per costruire il **client HTTP**, appoggiandosi su **SmartShopApiService**, file contenente tutti gli endpoint REST. Quest'ultimo contiene **@Annotation** usate per tradurre le chiamate HTTP verso il server.

Room Database & Entities: il Database Locale
L'applicazione utilizza **Room Database** per mantenere i dati anche dopo il riavvio, così da mantenerli in **cache** in caso di problemi col Server.
Room, il database locale, viene istanziato proprio come un vero e proprio DB (SQL per esempio). La sua creazione è eseguita sfruttando le **Entities** che grazie alle **@Annotation** (@PrimaryKey, @ColumnInfo) permettono di rappresentare lo stesso schema relazionale che risiede su PgAdmin4.
Oltre alle Entities, Room utilizza un **Converter** per rendere possibile il salvataggio di dati complessi come liste di stringhe (converte liste di stringhe in JSON e viceversa).

Repository e Dao
I file contrassegnati come **Repository** utilizzano sia i **DAO** che l'**API** associata a Retrofit per le richieste al Server. Questo permette di seguire la politica **Offline-First**: i dati che inizialmente vengono mostrati sono quelli presenti nel **DB Room**. Successivamente viene usata l'API per comunicare col server e prendere i dati. Non è detto che

quest'ultima operazione vada a buon fine (Server Express non raggiungibile): Room però permetterà all'utente di visualizzare tutti i dati salvati in Room nell'ultimo avvio. Il ruolo dei DAO in questo contesto è quello di definire le **Query** che possono essere eseguite sul DB Room facendo uso delle @Annotation (@Query, @Insert).

Coroutine, State & Flow
Tutte le operazioni asincrone all'interno dell'app sono eseguite grazie ad un'istruzione non bloccante: ***suspend fun***, chiamata **Coroutine**. Le istruzioni asincrone maggiormente utilizzato riguardando la comunicazione con il **DB Room** e il **Server Express** (ricordiamo, collegato a Postgres). Le Coroutine svolgono un ruolo molto importante perchè oltre a permetterci di eseguire operazioni non bloccanti, aggiornano in tempo reale la UI grazie a **State & Flow**, il tipo di dato ritornato da Room e da Retrofit. Quando Room si aggiorna emette l'update tramite **Flow**; il ViewModel riceve il Flow e aggiorna il proprio **StateFlow** usando *DAO.function.collect*. Grazie all'istruzione ***collectAsState*** la UI, che stava osservando lo StateFlow, si ricomporrà mostrando i dati aggiornati .

DataStore
Datastore è un'architettura di **archiviazione chiave-valore** leggera, usata per memorizzare le **preferenze** dell'utente e utilizza.
Nel nostro progetto, è stata utilizzata in 3 sezioni:

- **SessionDataStore**: Utilizzato per salvare l'accesso dell'utente. Questo gli permetterà di chiudere l'app e, all'avvio, trovare l'account già loggato;
- **AccountPreferencesDataStore**: sviluppato per la sezione Profilo degli utenti. Memorizza i dati dell'account e, in caso di cambiamento, viene aggiornato (nome, cognome, indirizzo, foto profilo);
- **FavoritesDataStore**: riservato ai clienti. Permette di archiviare i prodotti del catalogo come preferiti, così da permetterne l'acquisto veloce.

4.2 ViewModel Layer: Logica di Presentazione

Il **ViewModel Layer** comprende tutta la **logica di presentazione**: mantiene lo **stato della UI**, lancia le **Coroutines** per chiamare le **Repository** e acquisire i **Flow**, e gestire gli input dell'utente per inviare i dati al Server e a Room.
Tutte le schermate - o tutti gli **Screen** - sono regolate dal ViewModel a loro associato, così da seguire questa "filosofia" di separazione UI - Logica correttamente.

Main Activity & MainViewModel
Main Activity istanzia il **Root @Composable**. Crea i **viewModel**, applica i temi e instrada l'utente verso la schermata giusta (Cliente, Respoonsabile o Dipendente) grazie a ***MainViewModel.uiState.selectedRole***. Quando nel Datastore non è presente alcuna sessione, viene mostrato lo Screen di **Registrazione/Login**, altrimenti sarà visualizzato lo Screen associato all'**Account**. Una volta eseguito il login, il Main Activity si occupa di creare la **Bottom Navigation Bar** e la **Top App Navigation**: quella inferiore aiuterà l'utente a navigare tra le schermate ad egli accessibili, mentre la superiore gli permetterà di eseguire il **LogOut** oppure accedere alla sezione **Profilo**.

4.3 View Layer

Le differenti schermate

LoginScreen: come accennato prima, presenta il form di **Login/Registrazione**;

CatalogScreen: Home del Cliente. Presenta **catalogo** (lista di prodotti), **filtri** applicabili, **preferiti**, **carrello**, **storico ordini**.

EmployeeScreen: Home del Dipendente. Suddivisa in **Storico** e **Preparazione Ordini**.

ManagerScreen: Home del Responsabile. Presenta **Trasferimento Scorte** e **Riordino Prodotti**.
Gli screen vengono arricchiti grazie a strutture presenti all'interno di **Components** e **Map**.

Map

SupermarketPolygons: Definisce la forma geometrica degli scaffali del supermercato come poligoni (punti X,Y che formano figure chiuse). Contiene un **algoritmo**" che determina se un punto (il tap dell'utente) è dentro o fuori un poligono, usato per capire quale scaffale l'utente ha toccato sulla mappa.

MapJsonLoader: Carica il file **JSON** dalla cartella assets che contiene le coordinate di tutti i poligoni scaffali (es: scaffale A1 ha questi punti X,Y). Gestisce anche il **ridimensionamento automatico** delle immagini troppo grandi per evitare crash.

MapImageLoader: Carica l'immagine di sfondo della planimetria del supermercato dalla cartella assets (assets/map/supermarket.jpg). Usa remember() di Compose per caricare l'immagine una sola volta e tenerla in cache, così non viene ricaricata ogni volta che l'utente torna sulla schermata mappa.

Components

ProductImage: Un componente che mostra l'immagine di un prodotto scaricandola dal **backend** (http://localhost:3000/images/products/prd-xx.png). Usa la libreria **Coil** per caricare le immagini in modo asincrono. Conserva le immagini in **cache**.

NavBarDecor: Un semplice divisore decorativo con **gradiente** orizzontale che va da trasparente ai bordi a visibile al centro. Viene usato sopra la **NavigationBar** in basso per separare visivamente il contenuto della schermata dalla barra di navigazione. Usa i colori del tema **Material 3** per adattarsi automaticamente al tema chiaro/scuro.

StoreMapComposable: Il componente principale che disegna la **mappa interattiva** usando **Compose Canvas**. Mostra l'immagine di sfondo, ci disegna i **poligoni** degli scaffali sopra, rileva i **tap** dell'utente per capire quale scaffale è stato toccato, e cerchio quando l'utente clicca.

Package Theme (Color, Type, Theme)
Contiene la definizione di **colori**, **tipografie** e **temi** per la modalità sia **chiara** che **scura**.

Domain Package
Definizione di una **classe enumerata**, utilizzata per assegnare un tipo preciso per ciascuno dei 3 ruoli dell'utente: ***CUSTOMER, EMPLOYEE, MANAGER***.

5 Test

I **Test** sono stati svilupatti all'interno del progetto *SmartShop* seguendo le linee guida del **Mobile Testing**.

5.1 ViewModel Unit

UnitTest eseguiti sui **ViewModel** singoli. I Test sono:

- AccountPreferencesViewModelTest
- CatalogViewModelTest
- EmployeeViewModelTest
- LoginViewModelTest
- MainViewModelTest
- ManagerViewModelTest

Tecnologie & librerie utilizzate

- **MockK** (mock (simulazione) di dipendenze, comportamenti e flussi di dati)
- **Robolectric** (esecuzione di componenti Android su JVM per test realistici senza device)
- **DataStore** + **Flow** (simulazione persistenza e reattività dati)
- **Repository/DAO mockati** (simulano accessi a rete e database)

L'Obiettivo
Testare la **logica** di business dei **ViewModel** in **isolamento**: gestione stato, validazione input, reazioni agli update dei flussi, gestione errori, aggiornamento interfaccia tramite eventi, interazioni con repository e controlli sulle operazioni dell'utente (login, catalogo, ordini, riordini, ecc.). Mirano a **verificare** che ogni ViewModel produca **output** coerenti e prevedibili rispetto alle sue dipendenze simulate.

5.2 Integration Test

UnitTest che verificano la **comunicazione** tra i **ViewModel**. I Test sono:

- RepositoryIntegrationTest
- OrderRepositoryIntegrationTest
- RestockInventoryIntegrationTest

- UserRepositoryIntegrationTest
- AuthSessionIntegrationTest
- FavoritesDataStoreIntegrationTest

Tecnologie & librerie utilizzate

- **MockWebServer** (server HTTP finto per simulare API reali)
- **Room** (test del database locale con dati persistenti veri)
- **DataStore in contesto isolato** (persistenza reale, ma non condivisa con app)

L'Obiettivo

Verificare l'**integrazione** reale tra **repository**, **rete** e **database**: download e popolamento cache, salvataggio nuovi dati, persistenza, comportamento in caso di errori server, coerenza dei Flow emessi, gestione sessione, isolamento dati utente. Misurano se la catena "API → Repository → Cache/Flow" funziona correttamente in ogni contesto.

5.3 UI

I test sono:

- LoginFlowUiTest
- CustomerCatalogFlowUiTest
- CustomerHomeDeliveryUiTest
- CustomerProfileUpdateUiTest
- FavoritesPersistenceUiTest
- EmployeeFlowUiTest
- EmployeePickupFlowUiTest
- ManagerFlowUiTest

Tecnologie & librerie utilizzate

- **Jetpack Compose Testing** (per interagire con la UI: tap, inserimento testo, ricerca)
- **Instrumentation / AndroidJUnitRunner** (esecuzione su device o emulatori reali)
- **waitUntil / query node** (sincronizzazione con stato UI)

L'Obiettivo

Simulare i flussi utente reali nell'app (**azioni** da svolgere usando l'**interfaccia** grafica): login, navigazione tra schermate, ricerca prodotti, aggiunta al carrello, invio ordini, interazioni col profilo, flussi operativi dei dipendenti e del manager. Controllano il corretto comportamento delle entità grafiche - come l'apertura di una nuova schermata o l'aggiornamento di un dato nell'interfaccia; stati, conferme di operazioni eseguite con successo, e l'effettiva **risposta** dell'interfaccia agli **input**.

5.4 Test con Demo Applicazione

Come citato precedentemente, l'applicazione è stata creata con lo scopo di **sincronizzarsi** in tempo reale. Un **test "finale"** può essere eseguito, infatti, creando **3** istanze dell'**app** collegate allo stesso server: ciascuno dei 3 utenti sarà collegato con un **ruolo differente** dagli altri. Su ciascuna delle 3 istanze, eseguire una o più operazioni che si **riflettono** su due o **più** utenti e verificare l'**aggiornamento** durante l'utilizzo dell'app (magari **scorrendo** tra le varie pagine della Navigation Bar oppure forzando l'update grazie al pulsante ***Aggiorna***).

5.5 Jacoco Code Coverage

Per fornire una visualizzazione precisa della **percentuale** di codice **testato** rispetto a quello totale dell'applicazione, è stato utilizzato il tool ***Jacoco***. Questo ci ha permesso di visualizzare, grazie ad un task personalizzato, le percentuali precise di codice testato dai test all'interno di un file **HTML** creato dal tool. I risultati sono poi stati manipolati per essere rappresentati su un grafico costruito con l'ausilio di ***Jupyter Notebook***. Osservando il grafico riportato si può notare come i Test - rispettivamente Unit e UI - per ciascuno dei Package dell'App, ricoprano una determinata percentuale di codice.