UNIVERSITÀ
POLITECNICA
DELLE MARCHE

FACOLTÀ DI INGEGNERIA

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea in Ingegneria Informatica e dell'Automazione

Progettazione e sviluppo di un'applicazione mobile Android per il supporto alle attività di un ristorante.

Design and development of an Android mobile application to support restaurant activities.

Relatore:
Prof. Emanuele STORTI

Tesi di Laurea di:
Stefano MARINUCCI

Anno Accademico 2023/2024

Sommario

Introduzione **4**

1 Analisi dei requisiti **5**
1.1 Requisiti funzionali 5
1.1.1 Descrizione dei requisiti funzionali 6
1.2 Requisiti non funzionali 8
1.2.1 Descrizione dei requisiti non funzionali 8

2 Casi d'uso **9**
2.1 Diagramma dei casi d'uso 9
2.1.1 Descrizione dei casi d'uso 10

3 Strumenti utilizzati **16**
3.1 Ambiente di sviluppo 16
3.2 Firebase 18
3.3 Moduli e dipendenze 19
3.3.1 Build.gradle.kts 19
3.3.1.1 Build.gradle relativo al progetto 19
3.3.1.2 Build.gradle relativo al modulo 20
3.3.2 Catalogo delle versioni 22
3.3.3 Dipendenze dichiarate 23
3.4 Componenti software 25
3.4.1 Activity e Fragment 25
3.4.2 Navigazione all'interno dell'applicazione 29
3.4.3 Linguaggio XML 30
3.4.4 View Binding e Data Binding 31
3.4.5 Recycler View e Adapter 33
3.4.6 Coroutine di Kotlin 34
3.4.7 Database relazionale con Room 35

4 Sviluppo Software **38**
4.1 Architettura 38
4.1.1 Architettura Android 38
4.1.2 Architettura MVVM 40
4.1.3 Architettura del database centrale 41
4.1.4 Architettura del database locale 45
4.2 Il file Android Manifest 46
4.3 Struttura in Android Studio 47
4.4 Logica dell'applicazione 50
4.4.1 Package generale 50
4.4.2 Package database 51
4.4.3 Package adapter 57
4.4.4 Package ui/order 64

4.4.5 Package ui/loginregister 71
4.4.6 Package ui/account 74
4.4.7 Package ui/userorders 77
4.5 Schermate dell'applicazione 80
4.5.1 Schermata iniziale 80
4.5.2 Schermata dei prodotti 82
4.5.3 Schermata per personalizzare il prodotto 83
4.5.4 Schermata del carrello 84
4.5.5 Schermata per modificare un prodotto nel carrello 85
4.5.6 Schermata per aggiungere una nota all'ordine 86
4.5.7 Schermata di autenticazione 87
4.5.8 Schermata di recupero della password 88
4.5.9 Schermata per la creazione di un nuovo profilo 89
4.5.10 Schermata del profilo utente 90
4.5.11 Schermata per modificare i dati dell'utente 91
4.5.12 Schermata per selezionare un indirizzo di spedizione 92
4.5.13 Schermata per aggiungere un nuovo indirizzo 93
4.5.14 Schermata degli ordini effettuati di recente 94
4.5.15 Schermata per visualizzare i prodotti di un ordine 95

Conclusione **96**

Bibliografia **97**

Introduzione

Nello svolgere un'attività di ristorazione è fondamentale saper fornire un servizio di qualità in relazione alla fascia di prezzo in cui si opera. Per ottenere questo risultato è necessario offrire cibi e bevande di qualità, oltre che garantire un'ambiente capace di mettere a proprio agio le persone, indipendentemente dal fatto di essere membri dello staff o semplici clienti.
In un paese con standard culinari molto elevati come l'Italia, in particolare, queste due condizioni assumono un'importanza ancora maggiore.
Allo stesso tempo, se si vuole rimanere rilevanti sul mercato, è necessario sapersi innovare, sia nella selezione di prodotti disponibili ai clienti sia nel trovare nuovi modi per poter garantire il servizio.
Un buon modo per soddisfare l'ultima condizione è dare la possibilità alle persone di poter ordinare i prodotti desiderati in totale autonomia dove vogliono, quando vogliono ed avendo la possibilità di personalizzare gli ordini a loro discrezione.
Ed è proprio da quest'idea che nasce questo progetto. In un mondo sempre più connesso e digitalizzato, è infatti impensabile per un'attività commerciale continuare ad operare senza una propria rappresentanza virtuale. Il progetto ha come obbiettivi la progettazione e lo sviluppo di un prototipo di un'applicazione mobile a supporto delle attività di ristorazione, ed è stato commissionato da un ristorante situato a Roma. L'app offre all'utente diverse funzionalità, tra cui:

- Interfaccia user-friendly: l'interfaccia grafica è progettata per essere user-friendly ed intuitiva, in modo da essere accessibile ad un pubblico più vasto possibile.
- Ordinazione di una vasta gamma di cibi e bevande, raggruppati fra loro in base al loro tipo.
- Possibilità di personalizzare l'ordine del singolo elemento, attraverso la scelta delle quantità da ordinare e, dove possibile, della quantità dei singoli ingredienti presenti.
- Creazione di una nota personalizzata da parte dell'utente che sarà aggiunta all'ordine finale al momento della creazione.
- Possibilità di modificare un prodotto nel carrello senza doverlo necessariamente eliminare e reinserirlo da capo.
- Visualizzazione di uno storico degli ordini più recenti effettuati dall'utente, e dei singoli prodotti presenti in ciascun ordine

Di seguito è riportato un riassunto dei vari capitoli:

- Nel capitolo 1 è presente l'analisi dei requisiti, con relativa classificazione tra quelli funzionali e non funzionali. I requisiti funzionali sono tutti i servizi che l'app deve fornire all'utente, mentre quelli non funzionali rappresentano i vincoli collegati a questi ultimi che vanno necessariamente presi in considerazione.
- Nel Capitolo 2 viene mostrato il diagramma dei casi d'uso, in cui sono indicate tutte le possibili azioni che possono essere svolte interagendo con l'applicazione. In seguito, si procede ad una breve spiegazione per ciascuna di esse.
- Nel Capitolo 3 sono presentati i vari strumenti (editor grafici, IDE, servizi esterni, librerie…) utilizzati durante lo sviluppo del progetto.
- Nel Capitolo 4 sono riportate le varie fasi di progettazione. Si parte dalla descrizione dell'architettura utilizzata, passando poi per la descrizione di come sono organizzate le informazioni e concludendo con lo sviluppo dell'applicazione vera e propria.

1. Analisi dei requisiti

L'analisi dei requisiti è una fase fondamentale nello sviluppo di un qualsiasi progetto. I prodotti finiti per essere considerati validi, infatti, devono soddisfare una serie di vincoli concordati con il committente, oltre che ad una serie di standard di qualità che ne garantiscono l'utilizzo a lungo termine. Nel corso dell'analisi i requisiti individuati possono essere divisi in due sottocategorie:

- Requisiti funzionali: Sono tutte le funzionalità che si vogliono fornire all'utente finale.
- Requisiti non funzionali: Corrispondono a tutti quei vincoli da soddisfare che sono inevitabilmente connessi ai requisiti funzionali.

1.1 Requisiti funzionali

Requisiti funzionali
o RF1 - Registrazione utente o RF2 - Autenticazione o RF3 - Recupero della password o RF4 - Visualizzazione dati utente o RF5 - Modifica dati utente o RF6 - Visualizzazione dei prodotti disponibili o RF7 - Ricerca del prodotto o RF8 - Personalizzazione prodotto o RF9 - Eliminazione di un prodotto nel carrello o RF10 - Modifica di un prodotto nel carrello o RF11 - Ordinazione prodotti o RF12 - Aggiunta di una nota all'ordine o RF13 - Visualizzazione storico degli ordini più recenti o RF14 - Logout

1.1.1 Descrizione dei requisiti funzionali

Requisito	Descrizione
RF1 - Registrazione utente	Gestione della registrazione di un nuovo utente al sistema.
RF2 - Autenticazione	Gestione dell'autenticazione degli utenti registrati.
RF3 - Recupero della password	Procedura per generare una nuova password in caso di smarrimento.
RF4 - Visualizzazione dati utente	Possibilità da parte dell'utente di visualizzare i propri dati.
RF5 - Modifica dati utente	Possibilità da parte dell'utente di modificare i propri dati.
RF6 - Visualizzazione dei prodotti disponibili	Gestione del catalogo dei prodotti disponibili alla vendita.
RF7 - Ricerca del prodotto	Possibilità di cercare un prodotto specifico in base a parole chiave.
RF8 - Personalizzazione prodotto	Possibilità di personalizzare un determinato prodotto prima di inserirlo nel carrello.
RF9 - Eliminazione di un prodotto nel carrello	Possibilità di eliminare un prodotto già presente nel carrello.
RF10 - Modifica di un prodotto nel carrello	Possibilità di modificare un prodotto già presente nel carrello.
RF11 - Ordinazione prodotti	Il sistema deve permettere all'utente di ordinare i prodotti desiderati.
RF12 - Aggiunta di una nota all'ordine	Possibilità di aggiungere un messaggio personalizzato all'ordine.
RF13 - Visualizzazione storico degli ordini più recenti	Gestione dell'elenco degli ordini più recenti effettuati dall'utente.
RF14 - Logout	Il sistema deve permettere all'utente di disconnettersi quando richiesto.

L'utente deve avere la possibilità di visualizzare tutti i possibili prodotti presenti nel database, potendone visualizzare una foto, una breve descrizione ed il suo prezzo unitario. L'app deve permettere all'utente di poter cercare determinati prodotti in base al loro nome o alla loro descrizione. Una volta scelto un determinato prodotto, l'utente ha la possibilità di sceglierne la quantità desiderata e, dove possibile, personalizzare gli ingredienti che lo compongono. Nel personalizzare gli ingredienti l'utente potrà scegliere per uno di essi se rimuoverlo del tutto o aggiungerne in quantità maggiore rispetto alla norma. L'app permette all'utente di modificare un prodotto già presente nel carrello (quantità desiderata e scelta ingredienti) oppure di eliminarlo del tutto. Il sistema deve consentire all'utente di prenotare un ordine, permettendogli di decidere luogo e data di consegna. L'app consente di inserire una nota personalizzata al momento dell'ordinazione. Il sistema deve consentire di visualizzare gli ordini più recenti dell'utente, mostrando per ciascuno data dell'ordine, data di consegna, numero di prodotti ordinati, prezzo totale e lista dei prodotti.
Per ogni prodotto nella lista viene mostrato il nome, una foto, una lista degli ingredienti in quantità aggiuntive (se presenti), una lista degli ingredienti rimossi (se presenti), la quantità ordinata ed il prezzo. L'utente deve avere la possibilità di modificare o semplicemente visualizzare le proprie informazioni (nome, cognome, e-mail, indirizzo di consegna), di cambiare la propria password anche senza averla dimenticata e di aggiungere nuovi indirizzi di spedizione in caso di necessità.

1.2 Requisiti non funzionali

Requisiti non funzionali
o RNF1 - Implementazione o RNF2 - Interfaccia grafica o RNF3 - Facilità d'uso o RNF4 - Database centrale o RNF5 - Localizzazione o RNF6 - Permessi di sistema

1.2.1 Descrizione dei requisiti non funzionali

Requisito	Spiegazione
RNF1 - Implementazione	L'app deve essere realizzata in modo che possa essere eseguita nativamente in Android.
RNF2 - Interfaccia grafica	L'app deve essere dotata di un'interfaccia grafica.
RNF3 - Facilità d'uso	L'app deve essere facile da comprendere e da utilizzare da parte dell'utente.
RNF4 - Database centrale	Il sistema deve dotarsi di un database centrale per memorizzare le informazioni fondamentali per il suo funzionamento.
RNF5 - Localizzazione	L'app deve essere progettata in modo da implementare e supportare più lingue, facilitando così l'utilizzo da parte degli utenti che non conoscono l'italiano.
RNF6 - Permessi di sistema	L'app deve richiedere al sistema operativo Android i permessi necessari per il suo corretto funzionamento.

2. Casi d'uso

Il diagramma dei casi d'uso[1] è uno strumento che rappresenta tutte le possibili azioni che un determinato attore può svolgere all'interno di un sistema. In generale, le azioni indicate per ciascun attore corrispondono ad un sottoinsieme dei requisiti funzionali previsti per il prodotto finito.
Il diagramma consente anche di visualizzare tali requisiti in maniera più comprensibile, permettendo di indicare delle relazioni tra di loro. Un esempio di relazione è l'estensione, che si verifica quando una certa azione per poter essere svolta richiede necessariamente il completamento di un'altra.

2.1 Diagramma dei casi d'uso

Le convenzioni dell'ingegneria del software prevedono di rappresentare l'attore come un omino stilizzato con una breve descrizione che ne indica il ruolo, mentre le attività come ellissi contenenti una loro breve descrizione. Le relazioni tra attività sono indicate tramite delle frecce tratteggiate accompagnate da un'etichetta che ne indica il tipo.
La Figura 2.1 mostra un esempio di diagramma dei casi d'uso, in particolare quello relativo all'applicazione sviluppata.

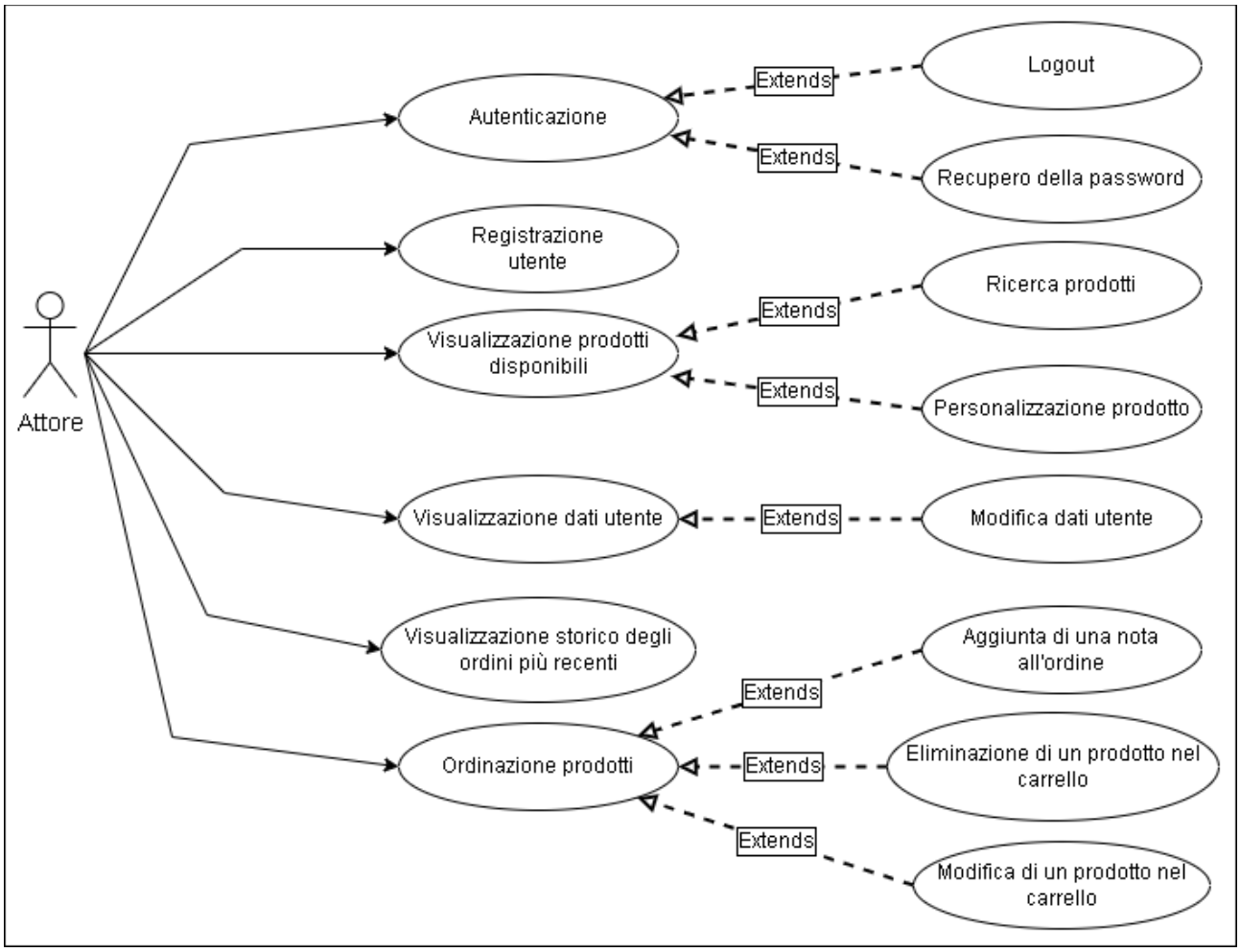

Figura 2.1 - Diagramma dei casi d'uso dell'applicazione.

2.1.1 Descrizione dei casi d'uso

Caso d'uso: Registrazione Utente
Attori: Utente.
Questo caso d'uso si verifica quando si vuole registrare un nuovo account utente nel sistema.
Pre-condizioni: l'account scelto non esiste nel sistema.
Post-condizioni: il nuovo account viene registrato nel sistema (o non è stato possibile registrarlo).
Sequenza degli eventi principale:

1. Il caso d'uso inizia quando l'utente decide di creare un nuovo account.
2. L'utente avvia l'applicazione e avvia la procedura di accesso.
3. L'applicazione visualizza a schermo la schermata di login.
4. L'utente richiede di accedere alla schermata di registrazione.
5. L'applicazione visualizza la schermata di registrazione.
6. L'utente inserisce le informazioni necessarie.
7. L'utente avvia la procedura di registrazione.
8. Il sistema registra il nuovo account appena creato nel database centrale.

Sequenza degli eventi alternativa:
La sequenza alternativa inizia dal punto 7.

1. Le informazioni inserite dall'utente sono incomplete o non valide.
2. La procedura di registrazione non va a buon fine.

Caso d'uso: Autenticazione
Attori: Utente.
Questo caso d'uso si verifica quando l'utente effettua il login per la prima volta o ha effettuato un logout in precedenza.
Pre-condizioni: l'utente ha le credenziali di accesso.
Post-condizioni: l'utente accede all'applicazione (o non è stato possibile permettere l'accesso).
Sequenza degli eventi principale:

1. Il caso d'uso inizia quando l'utente vuole autenticarsi per utilizzare l'applicazione.
2. L'utente accede alla schermata di login.
3. L'utente inserisce le proprie credenziali di accesso (e-mail e password).
4. L'utente avvia la procedura di autenticazione.
5. L'applicazione fa tornare l'utente alla schermata iniziale.

Sequenza degli eventi alternativa:
La sequenza alternativa inizia dal punto 4.

1. Le credenziali inserite sono errate o non valide.
2. La procedura di autenticazione non va a buon fine.

Caso d'uso: Recupero della password
Attori: Utente.
Questo caso d'uso si verifica quando l'utente non riesce ad effettuare il login perché ha dimenticato la password.
Pre-condizioni: l'utente è registrato nel sistema.
Post-condizioni: l'utente reimposta la password, permettendogli così di poter accedere al sistema.
Sequenza degli eventi principale:
1. Il caso d'uso inizia quando l'utente non riesce ad accedere poiché ha dimenticato la password.
2. L'utente accede alla schermata di login ed avvia la procedura di recupero.
3. L'utente inserisce l'indirizzo e-mail associato al suo account.
4. Il sistema provvede ad inviare una e-mail contenente un link per impostare la nuova password.
5. L'utente inserisce la nuova password desiderata.
6. L'utente può ora autenticarsi sfruttando le credenziali aggiornate.

Sequenza degli eventi alternativa:
La sequenza alternativa inizia dal punto 3.
1. L'indirizzo e-mail inserito non è valido oppure non è associato al proprio account.
2. La procedura di recupero non va a buon fine.

Caso d'uso: Logout
Attori: Utente.
Questo caso d'uso si verifica quando l'utente vuole disconnettersi dal sistema.
Pre-condizioni: l'utente ha effettuato il login.
Post-condizioni: l'utente si disconnette dal sistema.
Sequenza degli eventi principale:
1. Il caso d'uso inizia quando l'utente vuole effettuare il logout.
2. L'utente accede alla schermata del suo profilo.
3. L'utente seleziona l'opzione per disconnettersi.
4. Il sistema provvede a disconnettere l'utente da esso.
5. L'applicazione fa tornare l'utente alla schermata iniziale.

Sequenza degli eventi alternativa:
La sequenza alternativa inizia dal punto 4.
1. Si verifica un errore nel sistema.
2. Il sistema non consente all'utente di effettuare il logout.

Caso d'uso: Visualizzazione dati utente
Attori: Utente.
Questo caso d'uso si verifica quando l'utente vuole visualizzare i suoi dati memorizzati nel sistema.
Pre-condizioni: l'utente ha effettuato il login.
Post-condizioni: l'utente visualizza i suoi dati.
Sequenza degli eventi principale:
1. Il caso d'uso inizia quando l'utente vuole accedere ai propri dati nel sistema.
2. L'utente accede alla schermata del suo profilo.
3. L'applicazione mostra all'utente le sue informazioni memorizzate nel sistema.

Sequenza degli eventi alternativa:
La sequenza alternativa inizia dal punto 2.
1. Si verifica un errore nel sistema.
2. Le informazioni non vengono mostrate all'utente.

Caso d'uso: Modifica dati utente

Attori: Utente.
Questo caso d'uso si verifica quando l'utente vuole modificare i suoi dati memorizzati nel sistema.
Pre-condizioni: l'utente ha visualizzato a schermo i propri dati.
Post-condizioni: l'utente modifica i suoi dati.
Sequenza degli eventi principale:

1. Il caso d'uso inizia quando l'utente vuole modificare i propri dati nel sistema.
2. L'utente accede alla schermata del suo profilo.
3. L'applicazione mostra all'utente le sue informazioni memorizzate nel sistema.
4. L'utente accede alla schermata per modificare i propri dati.
5. L'utente modifica le proprie informazioni.

Sequenza degli eventi alternativa:
La sequenza alternativa inizia dal punto 4.

1. Si verifica un errore nel sistema.
2. L'utente torna alla schermata iniziale.

Caso d'uso: Visualizzazione dei prodotti disponibili

Attori: Utente.
Questo caso d'uso si verifica quando l'utente vuole visualizzare i prodotti disponibili alla vendita.
Pre-condizioni: nessuna.
Post-condizioni: l'utente visualizza i prodotti interessati.
Sequenza degli eventi principale:

1. Il caso d'uso inizia quando l'utente vuole visualizzare i prodotti in vendita.
2. L'utente accede alla schermata iniziale.
3. L'utente sceglie il tipo di prodotti da visualizzare.
4. L'applicazione mostra all'utente tutti i prodotti del tipo scelto.

Sequenza degli eventi alternativa:
La sequenza alternativa inizia dal punto 3.

1. Si verifica un errore nel sistema o non ci sono prodotti del tipo scelto.
2. L'utente deve scegliere un altro tipo di prodotto.

Caso d'uso: Ricerca del prodotto

Attori: Utente.
Questo caso d'uso si verifica quando l'utente vuole cercare determinati prodotti in base al loro nome oppure alla loro breve descrizione.
Pre-condizioni: l'utente ha visualizzato i prodotti di un determinato tipo.
Post-condizioni: l'utente visualizza i prodotti che rispettano i parametri di ricerca.
Sequenza degli eventi principale:

1. Il caso d'uso inizia quando l'utente vuole visualizzare cercare determinati prodotti.
2. L'utente accede alla schermata della lista dei prodotti.
3. L'utente scrive nella casella di ricerca una parola chiave oppure una frase.
4. L'applicazione mostra all'utente tutti i prodotti che contengono tale parola o frase nel nome oppure nella descrizione.

Sequenza degli eventi alternativa:
La sequenza alternativa inizia dal punto 3.

1. L'utente inserisce una parola una frase che non è presente nei prodotti presenti nella lista.
2. L'applicazione mostra all'utente una schermata vuota.

Caso d'uso: Personalizzazione prodotto

Attori: Utente.
Questo caso d'uso si verifica quando l'utente vuole personalizzare il prodotto scelto prima di inserirlo nel carrello.
Pre-condizioni: l'utente ha effettuato il login.
Post-condizioni: l'utente personalizza il prodotto desiderato e lo inserisce nel carrello.
Sequenza degli eventi principale:

1. Il caso d'uso inizia quando l'utente vuole personalizzare un prodotto da ordinare.
2. L'utente accede alla schermata per visualizzare le informazioni sul prodotto.
3. L'utente personalizza il prodotto scegliendone la quantità desiderata e, dove possibile, scegliendo per ogni ingrediente se aggiungerne in quantità maggiori o rimuoverlo del tutto.
4. L'utente inserisce il prodotto personalizzato nel carrello.
5. L'applicazione fa tornare l'utente alla lista dei prodotti.

Sequenza degli eventi alternativa:
La sequenza alternativa inizia dal punto 3.

1. L'utente non è autenticato oppure si verifica un errore nel sistema.
2. L'applicazione mostra all'utente un messaggio per informarlo del mancato inserimento.

Caso d'uso: Eliminazione di un prodotto nel carrello

Attori: Utente.
Questo caso d'uso si verifica quando l'utente vuole eliminare un prodotto nel carrello.
Pre-condizioni: il prodotto da eliminare si trova nel carrello.
Post-condizioni: il prodotto viene rimosso dal carrello.
Sequenza degli eventi principale:

1. Il caso d'uso inizia quando l'utente vuole eliminare un prodotto inserito in precedenza.
2. L'utente accede alla schermata del carrello.
3. L'utente individua il prodotto nella lista e clicca il pulsante per rimuoverlo.
4. Il sistema provvede ad eliminare il prodotto dal carrello.
5. Una volta eliminato l'applicazione rimuove il prodotto dalla lista.

Sequenza degli eventi alternativa:
La sequenza alternativa inizia dal punto 3.

1. L'utente non è online oppure si verifica un errore nel sistema.
2. L'applicazione mostra all'utente un messaggio per avvisarlo della mancata eliminazione.

Caso d'uso: Modifica di un prodotto nel carrello

Attori: Utente.
Questo caso d'uso si verifica quando l'utente vuole modificare un prodotto nel carrello.
Pre-condizioni: il prodotto da modificare si trova nel carrello.
Post-condizioni: la modifica viene aggiornata nel carrello.
Sequenza degli eventi principale:

1. Il caso d'uso inizia quando l'utente vuole modificare un prodotto inserito in precedenza.
2. L'utente accede alla schermata del carrello.
3. L'utente individua il prodotto nella lista e clicca il pulsante per modificarlo.
4. L'applicazione porta l'utente nella schermata per modificare il prodotto
5. L'utente apporta le modifiche desiderate (quantità totale e/o scelta degli ingredienti).
6. Il sistema provvede ad aggiornare il prodotto nel carrello.
7. Una volta finito l'applicazione mostra il prodotto aggiornato nella lista.

Sequenza degli eventi alternativa:
La sequenza alternativa inizia dal punto 4.

1. L'utente non è online oppure si verifica un errore nel sistema.

L'applicazione mostra all'utente un messaggio per avvisarlo della mancata modifica.

Caso d'uso: Ordinazione prodotti

Attori: Utente.
Questo caso d'uso si verifica quando l'utente vuole ordinare i prodotti presenti nel carrello.
Pre-condizioni:

1. Il carrello deve contenere almeno un elemento.
2. L'utente deve essere online ed autenticato.

Post-condizioni: l'ordine viene registrato nel sistema.
Sequenza degli eventi principale:

1. Il caso d'uso inizia quando l'utente vuole ordinare tutti i prodotti scelti in precedenza.
2. L'utente accede alla schermata del carrello.
3. L'utente sceglie il luogo in cui deve avvenire la consegna.
4. L'utente sceglie la data e l'ora in cui deve avvenire la consegna.
5. L'utente procede all'ordine.
6. Il sistema provvede a registrare l'ordine nel database.
7. Il sistema provvede a svuotare il carrello.
8. L'applicazione riporta l'utente nella schermata iniziale.

Sequenza degli eventi alternativa:
La sequenza alternativa inizia dal punto 5.

1. L'utente non è online, non è autenticato oppure si verifica un errore nel sistema.
2. L'applicazione mostra all'utente un messaggio per avvisarlo della mancata ordinazione.

Caso d'uso: Aggiunta di una nota all'ordine

Attori: Utente.
Questo caso d'uso si verifica quando l'utente vuole aggiungere una nota personalizzata all'ordine.
Pre-condizioni: l'utente deve avere almeno un prodotto nel carrello.
Post-condizioni: il messaggio personalizzato viene associato all'ordine.
Sequenza degli eventi principale:

1. Il caso d'uso inizia quando l'utente vuole aggiungere un messaggio personalizzato all'ordine.
2. L'utente accede alla schermata del carrello.
3. L'utente preme il tasto per aggiungere una nota.
4. L'applicazione porta l'utente nella schermata per scrivere/modificare la nota da allegare.
5. L'utente scrive il messaggio che desidera e preme il tasto per aggiungerlo.
6. Il sistema provvede a salvare la nota nel profilo dell'utente.
7. Al momento dell'ordine la nota viene associata ad esso ed eliminata dal profilo dell'utente.

Sequenza degli eventi alternativa:
La sequenza alternativa inizia dal punto 5.

1. L'utente non è online oppure si verifica un errore nel sistema.
2. L'applicazione mostra all'utente un messaggio per avvisarlo della mancata aggiunta.

Caso d'uso: Visualizzazione storico degli ordini più recenti

Attori: Utente.
Questo caso d'uso si verifica quando l'utente vuole visualizzare un elenco degli ordini più recenti che ha effettuato.
Pre-condizioni: l'utente deve essere autenticato.
Post-condizioni: l'utente può visualizzare uno storico degli ordini più recenti effettuati.
Sequenza degli eventi principale:

1. Il caso d'uso inizia quando l'utente vuole vedere un elenco degli ordini più recenti effettuati.
2. Dalla schermata iniziale l'utente accede alla schermata dei suoi ordini recenti.
3. Il sistema recupera dal database i dati sugli ultimi ordini effettuati.
4. L'applicazione mostra all'utente una lista degli ordini recenti.
5. L'utente cliccando su un elemento può vedere un elenco dei prodotti presenti in quell'ordine.
6. L'applicazione provvede a mostrare all'utente una lista dei prodotti.

Sequenza degli eventi alternativa:
La sequenza alternativa inizia dal punto 2.

1. Si verifica un errore nel sistema.
2. L'applicazione mostra all'utente una lista vuota.

3. Strumenti utilizzati

Un requisito fondamentale per lo sviluppo di un qualsiasi progetto è conoscere e saper utilizzare gli strumenti di sviluppo adeguati. Questo capitolo si occupa di mostrare e descrivere brevemente i programmi, le librerie, i servizi esterni ed altri tipi di risorse utilizzate durante lo sviluppo dell'applicazione.

3.1 Ambiente di sviluppo

Android Studio[2] è l'ambiente di sviluppo integrato (IDE, Integrated Development Enviroment) ufficiale per lo sviluppo di app Android. È sviluppato da Google ed è ampiamente utilizzato dagli sviluppatori per creare ed eseguire il debug di applicazioni. L'IDE offre una serie di strumenti e funzionalità che semplificano il processo di sviluppo delle app, e permette di aggiungerne molti altri grazie ai numerosi plugin sviluppati dalla community. La sua interfaccia grafica (Figura 3.1) è molto semplice e facilmente comprensibile, questo consente ai nuovi utenti di ambientarsi facilmente una volta iniziato ad usare l'IDE.

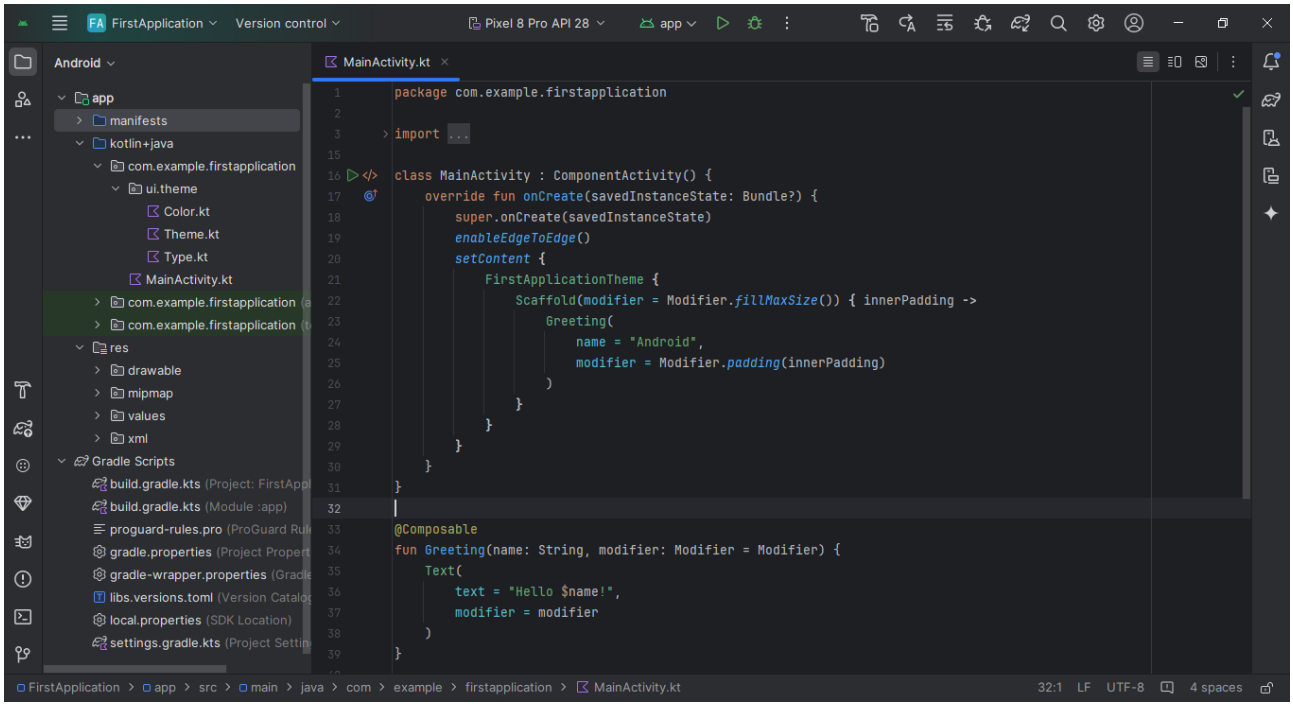

Figura 3.1 - Interfaccia utente di Android Studio.

Alcuni aspetti e caratteristiche chiave di Android Studio:

- **Interfaccia utente:** Android Studio ha un'interfaccia intuitiva con vari pannelli e finestre per la codifica, la progettazione di layout e l'anteprima delle schermate delle app.

- **Editor di codice:** offre un potente editor di codice con funzionalità come il completamento del codice, la navigazione al suo interno e la messa in evidenza della sintassi. Le app possono essere scritte in Kotlin[3] (come Google consiglia da maggio 2019), in Java oppure negli altri linguaggi supportati dall'IDE.

- **Editor di layout:** Android Studio include un editor di layout visivo che consente agli sviluppatori di progettare interfacce utente utilizzando un generatore di interfacce drag-and-drop. È possibile visualizzare un'anteprima dell'aspetto dell'app su dispositivi e dimensioni dello schermo diversi.

- **Emulatore:** Android Studio fornisce un emulatore integrato che consente di testare l'app su diverse configurazioni di dispositivi Android. Tramite questa funzionalità è possibile testare l'app su dispositivi con versioni di Android, specifiche tecniche e dimensioni dello schermo configurabili dall'utente. Risulta molto utile perché evita agli sviluppatori di dover reperire un dispositivo per ogni configurazione su cui si vuole effettuare dei test.

- **Strumenti di debug:** offre strumenti di debug per identificare e risolvere gli errori nel codice. Per individuare i bug è possibile impostare punti di interruzione, esaminare l'evolvere delle variabili durante l'esecuzione e scorrere il codice istruzione per istruzione.

- **Profilazione delle prestazioni:** Android Studio include strumenti per la profilazione delle prestazioni dell'app, aiutando l'utente nell' identificare i colli di bottiglia ed ottimizzare il codice per migliorarne le prestazioni.

- **Integrazione con sistemi per controllo delle versioni:** l'IDE è integrato con sistemi come Git, semplificando la gestione e l'archiviazione del codice sorgente del progetto.

- **Gradle Build System:** Android Studio utilizza il sistema di compilazione Gradle[4], che consente di gestire le dipendenze dell'app e di definire e personalizzare il processo di compilazione.

- **Firma delle app:** Android Studio semplifica il processo di firma dell'app, questo passaggio è necessario per poterla distribuire sul Google Play Store e/o su altri app store.

- **Modelli e procedure guidate:** fornisce modelli e procedure guidate per iniziare le attività comuni di sviluppo, come la creazione di una nuova attività, la configurazione della navigazione o l'aggiunta di un database.

- **Android Jetpack:** l'IDE si integra con Android Jetpack, un insieme di librerie, strumenti e best practice per aiutare gli sviluppatori a creare più facilmente app Android di alta qualità.

- **Integrazione con Google Play Console:** permette la pubblicazione delle app sul Google Play Store da Android Studio e di monitorarne le prestazioni utilizzando Google Play Console.

3.2 Firebase

Firebase[5] è una piattaforma per lo sviluppo di applicazioni mobili e web sviluppata da Google. Fornisce agli sviluppatori una vasta gamma di strumenti e servizi per aiutarli a creare applicazioni sicure e ricche di funzionalità. I servizi offerti da Firebase sono progettati per gestire vari aspetti dello sviluppo di un'applicazione, come ad esempio la registrazione e l'autenticazione degli utenti, l'uso di un database centralizzato oppure l'archiviazione via cloud. La Figura 3.2 mostra la schermata iniziale della console dell'app, da cui è possibile configurare tutte le funzionalità citate e molte altre.

Figura 3.2 - La console dell'applicazione in Firebase.

I componenti che sono stati utilizzati nell'app *Ristorante* sono i seguenti:

- **Firebase Authentication**[6]**:** Firebase offre diversi servizi di autenticazione, tra cui autenticazione tramite e-mail/password, autenticazione tramite social (come ad esempio Google, Facebook e X) e autenticazione del numero di telefono. È stato integrato nell'applicazione con la funzione di autentificare e registrare i vari profili associati agli utenti.
- **Firestore**[7]**:** È il database NoSQL più avanzato di Firebase. Offre funzionalità aggiuntive come scalabilità, potenti funzionalità di query e sincronizzazione dei dati offline. È adatto per una vasta gamma di applicazioni, dalle app mobili alle app Web. Tale servizio è stato utilizzato per memorizzare i dati relativi a:
  - Tipi di prodotto.
  - Prodotti (cibi e bevande).
  - Ingredienti.
  - Account utenti, per ogni account vengono salvate una lista dei prodotti nel suo carrello ed una lista degli indirizzi di spedizione aggiunti.
  - Ordini.

- **Cloud Storage**[8]: Firebase Cloud Storage fornisce un cloud storage sicuro e scalabile per i contenuti che si vogliono rendere recuperabili tramite un repository centrale. È possibile integrarlo con Firebase Authentication in modo da renderlo utilizzabile solo dagli account registrati ed autenticati. È stato utilizzato per memorizzare le immagini dei prodotti e degli ingredienti (in formato .jpeg).

3.3 Moduli e Dipendenze

Nel contesto dello sviluppo in Android Studio, non si può fare a meno dei concetti di modulo, dipendenza e plugin Gradle. Ad un primo sguardo alcuni termini possono sembrare sinonimi ma in realtà ci sono alcune differenze nel come vengono usati.

Modulo: è un componente autonomo di un progetto Android contenente codice sorgente, file di risorse, file di build, tra cui il file Android Manifest. Ogni modulo viene progettato e testato in modo indipendente dagli altri, facilitando la manutenzione del progetto.
Si può pensare ad un modulo come un mini-progetto all'interno di un progetto più grande.

Dipendenza: è un modulo o una libreria esterna di cui un progetto Android ha bisogno per funzionare correttamente. Possono essere sia locali, ossia memorizzate nella cartella del progetto, oppure remote, ovvero da recuperare da un repository. Le dipendenze sono fondamentali nello sviluppo Android poiché forniscono funzionalità e/o servizi allo sviluppatore senza la necessità di doverli progettare, implementare e testare. Gli sviluppatori delle dipendenze si faranno carico di questi impegni, oltre ad effettuare la loro manutenzione nel corso del tempo.

Plugin Gradle: è un componente che estende il sistema di build di Gradle. Questi componenti permettono di integrare funzionalità specifiche per la creazione di applicazioni Android.

Per utilizzare un plugin Gradle in un progetto, è necessario aggiungerlo nel file *build.gradle.kts* relativo all'intero progetto. Per usare una dipendenza invece è necessario aggiungerla nel file *build.gradle.kts* relativo al modulo specifico in cui la si vuole utilizzare.

3.3.1 Build.gradle.kts

I file *build.gradle.kts* sono i file di configurazione usati nei progetti Android. Oltre a specificare le dipendenze, contiene anche metadati ed altre impostazioni centrali nello sviluppo.

3.3.1.1 Build.gradle relativo al progetto[9]

Contiene configurazioni di build comuni per ogni modulo del progetto, e lo trova nella sua cartella principale. Spesso definisce le versioni dei plugin usati dai moduli che compongono il progetto (Figura 3.3). L'opzione *apply false* indica di aggiungere il plugin Gradle come dipendenza di build ma di non applicarlo al progetto corrente.

```
// Top-level build file where you can add configuration options common to all sub-projects/modules.
plugins {
    alias(libs.plugins.android.application) apply false
    alias(libs.plugins.jetbrains.kotlin.android) apply false
    alias(libs.plugins.google.gms.google.services) apply false
    alias(libs.plugins.com.google.devtools.ksp) apply false
    alias(libs.plugins.androidx.navigation.safeargs.kotlin) apply false
    alias(libs.plugins.androidx.room) apply false
}
```

Figura 3.3 - La sezione dei plugin presente nel file del progetto.

3.3.1.2 Build.gradle relativo al modulo[10]

Contiene configurazioni di build specifiche per un modulo, e lo si può trovare nella sua cartella principale. Ecco una panoramica del contenuto del file e del suo significato:

Sezione dei plugin: Spesso è la prima sezione nel documento e serve ad applicare i plugin di Gradle in questa build e di conseguenza includere le dipendenze globali del progetto (Figura 3.4).

```
plugins {
    alias(libs.plugins.android.application)
    alias(libs.plugins.jetbrains.kotlin.android)
    alias(libs.plugins.google.gms.google.services)
    alias(libs.plugins.com.google.devtools.ksp)
    alias(libs.plugins.androidx.navigation.safeargs.kotlin)
    alias(libs.plugins.androidx.room)
}
```

Figura 3.4 - La sezione dei plugin usata nel file del modulo.

Impostazioni SDK Android: include le versioni dell'Android SDK usate nel modulo (Figura 3.5). Ci sono tre campi da tenere in considerazione:

- ***compileSdk***: specifica quale API Android viene utilizzata nella compilazione del codice.
- ***minSdk***: indica la versione minima di Android che si vuole supportare, agendo su questo valore si modifica il gruppo di dispositivi che possono eseguire il codice.
- ***targetSdk***: specifica qual è la versione più recente della libreria Android che è stata propriamente testata. Il suo valore deve essere compreso tra *minSdk* e *compileSdk*.

namespace: il namespace (spazio dei nomi) serve a identificare il codice e le risorse associate al modulo. Ciò consente di usare gli stessi nomi per le risorse, le classi e le interfacce, semplificando la comprensione e lo sviluppo (Figura 3.5).

versionName: definisce un nome della versione corrente comprensibile all'utente (Figura 3.5).

```
android {
    namespace = "com.projectrestaurant"
    compileSdk = 35

    defaultConfig {
        applicationId = "com.projectrestaurant"
        minSdk = 28
        targetSdk = 35
        versionCode = 1
        versionName = "1.0"
        testInstrumentationRunner = "androidx.test.runner.AndroidJUnitRunner"
    }
```

Figura 3.5 - Valori delle impostazioni SDK, del parametro "namespace" e di "versionName".

Dipendenze: Specifica le dipendenze necessarie per la build del modulo (Figura 3.6).

```
dependencies {
    implementation(libs.androidx.core.ktx)
    implementation(libs.androidx.appcompat)
    implementation(libs.material)
    implementation(libs.androidx.constraintlayout)
    implementation(libs.androidx.lifecycle.livedata.ktx)
    implementation(libs.androidx.lifecycle.viewmodel.ktx)
    implementation(libs.androidx.navigation.fragment.ktx)
    implementation(libs.androidx.navigation.ui.ktx)
    implementation(libs.androidx.coordinatorlayout)
    implementation(libs.firebase.auth)
    implementation(libs.firebase.firestore)
    implementation(libs.androidx.room.runtime)
    implementation(libs.androidx.room.ktx) // optional - Kotlin Extensions and Coroutines support for Room
    implementation(libs.bumptech.glide)
    annotationProcessor(libs.androidx.room.compiler)
    ksp(libs.androidx.room.compiler) // To use Kotlin Symbol Processing (KSP)
    testImplementation(libs.junit)
    testImplementation(libs.androidx.room.testing) // optional - Test helpers for Room
    androidTestImplementation(libs.androidx.junit)
    androidTestImplementation(libs.androidx.espresso.core)
}
```

Figura 3.6 - Elenco delle dipendenze usate nel modulo.

3.3.2 Catalogo delle versioni

Il catalogo delle versioni di Gradle[11] (Gradle version catalog) permette di aggiungere e mantenere dipendenze e plugin in modo semplice e scalabile. Risulta particolarmente utile quando si hanno progetti composti da molti moduli. Il catalogo delle versioni è un repository centralizzato di plugin e dipendenze che possono essere referenziate dai diversi moduli del progetto, evitando così di modificare i singoli file di build ogni volta che si deve aggiornare una dipendenza.

```
[versions]
agp = "8.7.0"
kotlin = "2.0.20"
coreKtx = "1.13.1"
junit = "4.13.2"
junitVersion = "1.2.1"
espressoCore = "3.6.1"
appcompat = "1.7.0"
material = "1.12.0"
constraintlayout = "2.1.4"
lifecycleLivedataKtx = "2.8.6"
lifecycleViewmodelKtx = "2.8.6"
navigationFragmentKtx = "2.8.2"
navigationUiKtx = "2.8.2"
coordinatorlayout = "1.2.0"
glide = "4.16.0"
googleGmsGoogleServices = "4.4.2"
firebaseAuth = "23.0.0"
firebaseFirestore = "25.1.0"
roomKtx = "2.6.1"
ksp = "2.0.20-1.0.24"
```

Figura 3.7 - Struttura della sezione "versions" nel file del progetto.

Il file del catalogo si trova nella sottocartella *gradle* della cartella del progetto, ed è chiamato quasi sempre *libs.version.toml* (dato che Gradle per impostazione predefinita cerca il catalogo proprio in un file con quel nome). È composto da tre sezioni:

- ***versions***: questa sezione contiene una lista di variabili per memorizzare le versioni delle dipendenze e dei plugin. Queste variabili vengono poi utilizzate nelle sezioni successive. Un esempio di tale sezione lo si può vedere nella Figura 3.7.

- ***libraries***: questa sezione contiene una lista delle dipendenze (Figura 3.8). Per ciascuna di esse vanno indicati tre parametri:
  - ***group*:** indica il namespace oppure il package che identifica la dipendenza.
  - ***name*:** indica il nome della dipendenza all'interno del namespace o del package di riferimento.
  - ***version.ref*:** contiene il nome della variabile nella sezione *versions* che contiene la versione della dipendenza.

```
[libraries]
androidx-core-ktx = { group = "androidx.core", name = "core-ktx", version.ref = "coreKtx" }
junit = { group = "junit", name = "junit", version.ref = "junit" }
androidx-junit = { group = "androidx.test.ext", name = "junit", version.ref = "junitVersion" }
androidx-espresso-core = { group = "androidx.test.espresso", name = "espresso-core", version.ref = "espressoCore" }
androidx-appcompat = { group = "androidx.appcompat", name = "appcompat", version.ref = "appcompat" }
material = { group = "com.google.android.material", name = "material", version.ref = "material" }
androidx-constraintlayout = { group = "androidx.constraintlayout", name = "constraintlayout", version.ref = "constraintlayout" }
androidx-lifecycle-livedata-ktx = { group = "androidx.lifecycle", name = "lifecycle-livedata-ktx", version.ref = "lifecycleLivedataKtx" }
androidx-lifecycle-viewmodel-ktx = { group = "androidx.lifecycle", name = "lifecycle-viewmodel-ktx", version.ref = "lifecycleViewmodelKtx" }
androidx-navigation-fragment-ktx = { group = "androidx.navigation", name = "navigation-fragment-ktx", version.ref = "navigationFragmentKtx" }
androidx-navigation-ui-ktx = { group = "androidx.navigation", name = "navigation-ui-ktx", version.ref = "navigationUiKtx" }
androidx-coordinatorlayout = { group = "androidx.coordinatorlayout", name = "coordinatorlayout", version.ref = "coordinatorlayout" }
firebase-auth = { group = "com.google.firebase", name = "firebase-auth", version.ref = "firebaseAuth" }
firebase-firestore = { group = "com.google.firebase", name = "firebase-firestore", version.ref = "firebaseFirestore" }
androidx-room-runtime = { group = "androidx.room", name = "room-runtime", version.ref = "roomKtx" }
androidx-room-compiler = { group = "androidx.room", name = "room-compiler", version.ref = "roomKtx" }
androidx-room-ktx = { group = "androidx.room", name = "room-ktx", version.ref = "roomKtx" }
androidx-room-testing = { group = "androidx.room", name = "room-testing", version.ref = "roomKtx" }
bumptech-glide = { group = "com.github.bumptech.glide", name = "glide", version.ref = "glide" }
```

Figura 3.8 - Come è strutturata la sezione "libraries" nel file del progetto.

- ***plugins***: questa sezione contiene una lista dei plugin (Figura 3.9). Per ciascuno di loro vanno indicati due parametri:
  - ***id*:** è un identificatore univoco per il plugin.
  - ***version.ref*:** contiene il nome della variabile nella sezione *versions* che contiene la versione del plugin.

```
[plugins]
android-application = { id = "com.android.application", version.ref = "agp" }
jetbrains-kotlin-android = { id = "org.jetbrains.kotlin.android", version.ref = "kotlin" }
google-gms-google-services = { id = "com.google.gms.google-services", version.ref = "googleGmsGoogleServices" }
com-google-devtools-ksp = { id = "com.google.devtools.ksp", version.ref = "ksp" }
androidx-navigation-safeargs-kotlin = { id = "androidx.navigation.safeargs.kotlin", version.ref = "navigationFragmentKtx" }
androidx-room = { id = "androidx.room", version.ref = "roomKtx"}
```

Figura 3.9 - Come è strutturata la sezione "plugins" nel file del progetto.

3.3.3 Dipendenze dichiarate

Visto che sono state citate le dipendenze di sviluppo, nell'applicazione *Ristorante* sono state utilizzate le seguenti dipendenze e plugin:

Dipendenze e plugin Firebase (Figura 3.10)

- **Plugin Google Services:** plugin per usare le API di Firebase nel progetto.
- **Firebase Authentication:** permette di implementare il sistema di autenticazione di Firebase.
- **Firebase Firestore:** permette l'accesso al database NoSQL di Firebase.

```
alias(libs.plugins.google.gms.google.services)

implementation(libs.firebase.auth)
implementation(libs.firebase.firestore)
```

Figura 3.10 - Dichiarazione delle dipendenze Firebase all'interno del progetto.

Glide[12] (Figura 3.11)
Glide è un gestore di immagini, video e gif veloce ed open source. Permette di gestire in modo semplice il recupero, la ridimensione e la visualizzazione di questi media disponibili in remoto. Un classico esempio è il recupero e la visualizzazione di un'immagine raggiungibile tramite URL.

```
implementation(libs.bumptech.glide)
```

Figura 13.11 - Dichiarazione di Glide.

Dipendenze e plugin Room e KSP (Figura 3.12)
Room è una libreria che fornisce un livello di astrazione per il linguaggio SQLite, permettendo un accesso facilitato al database locale dell'applicazione. Nella pratica fornisce i seguenti vantaggi:

- Verifica in fase di compilazione delle query SQLite, riducendo la possibilità di bug.
- Annotazioni pratiche per ridurre codice ripetitivo e più soggetto ad errori.
- Semplifica il processo di migrazione del database.

KSP (Kotlin Symbol Processing) è un plugin per il compilatore Kotlin sviluppato da Google che consente di creare annotazioni personalizzate. Permette di generare codice, analizzare la struttura del codice e modificarla al momento della compilazione.
Attraverso l'analisi e la modifica a compile-time KSP può effettuare controlli ed ottimizzazioni del codice prima della sua compilazione in bytecode.

```
alias(libs.plugins.com.google.devtools.ksp)
alias(libs.plugins.androidx.room)
implementation(libs.androidx.room.runtime)
implementation(libs.androidx.room.ktx) // optional - Kotlin Extensions and Coroutines support for Room
annotationProcessor(libs.androidx.room.compiler)
ksp(libs.androidx.room.compiler) // To use Kotlin Symbol Processing (KSP)
testImplementation(libs.androidx.room.testing) // optional - Test helpers for Room
```

Figura 3.12 - Dichiarazione delle dipendenze Room e KSP.

Dipendenze e plugin per la navigazione in-app[13] (Figura 3.13)
Sono componenti che consentono all'utente di navigare attraverso i vari contenuti all'interno dell'app. Attraverso loro è possibile gestire svariati casi d'uso per la navigazione, dal semplice click su un pulsante a cose più complesse come le barre di navigazione.
Il plugin *SafeArgs* permette il passaggio di dati durante la navigazione in modo semplice e sicuro, dato che garantisce la sicurezza dei tipi per i singoli dati.

```
alias(libs.plugins.androidx.navigation.safeargs.kotlin)
implementation(libs.androidx.navigation.fragment.ktx)
implementation(libs.androidx.navigation.ui.ktx)
```

Figura 3.13 - Dichiarazione delle dipendenze all'interno del progetto.

Dipendenze per l'interfaccia grafica (Figura 3.14)

- **Material Design:** fornisce strumenti per creare interfacce grafiche consistenti tra le varie applicazioni Android. In particolare:
  - Creazione di un tema generale da applicare a tutti gli elementi dell'interfaccia, permettendo così un aspetto consistente in tutta l'app.
  - Elementi grafici più sofisticati rispetto a quelli base forniti da Android. Questi elementi permettono agli sviluppatori una maggiore personalizzazione.
  - Creazioni di animazioni personalizzate per diversi eventi che si verificano nell'uso dell'app (come, ad esempio, il tocco su un elemento grafico o il passaggio da una schermata ad un'altra).
- **Constraint Layout:** permette di posizionare gli elementi grafici sullo schermo in modo più semplice ed efficiente, contribuendo alla creazione di interfacce grafiche più complesse senza penalizzare le prestazioni dell'app.

```
implementation(libs.androidx.constraintlayout)
implementation(libs.androidx.coordinatorlayout)
implementation(libs.androidx.lifecycle.livedata.ktx)
```

Figura 3.14 - Dichiarazione dipendenze di layout.

Dipendenze per l'architettura MVVM (Figura 3.15)
Permettono di implementare il pattern MVVM (Model-View-ViewModel). Questo pattern di progettazione consente di separare le varie componenti logiche dell'app, facilitando la loro implementazione e manutenzione nel tempo.

```
implementation(libs.androidx.lifecycle.livedata.ktx)
implementation(libs.androidx.lifecycle.viewmodel.ktx)
```

Figura 3.15 - Dichiarazione delle dipendenze MVVM.

3.4 Componenti software

Una volta specificate le dipendenze è necessario descrivere con dettaglio alcune componenti software tipicamente utilizzate nelle applicazioni Android. Per ciascuna di queste è necessario spiegare la sua utilità, la sua logica di funzionamento e l'effettiva procedura per implementarla e renderla operativa in una qualsiasi applicazione.

3.4.1 Activity e Fragment

Un componente fondamentale di cui non si può fare a meno nella realizzazione di una qualsiasi app Android sono le Activity[14] (o Attività in italiano). Sono responsabili per la gestione dell'interfaccia utente e dell'interazione con l'applicazione. Nella pratica, un'activity corrisponde ad una singola schermata oppure ad una singola operazione che l'utente può svolgere.
Una caratteristica importante delle activity risiede nel loro ciclo di vita o lifecycle. Ogni activity passa attraverso una serie di stati durante il suo ciclo di vita, per ciascuno dei quali è possibile decidere il loro comportamento. Ad ogni cambio di stato, infatti, le API Android mettono a disposizione dei metodi che, una volta sovrascritti, sono eseguiti automaticamente secondo un ordine ben preciso (Figura 3.16). Esistono sei possibili stati:

1. ***created***: L'activity viene creata ed inizializzata. Per farlo si esegue il metodo *onCreate()*, in cui vengono eseguite delle operazioni necessarie per consentire il funzionamento nel passaggio agli stati successivi, come ad esempio l'inflating dell'interfaccia utente. Questo metodo deve essere necessariamente implementato per ogni activity su cui si lavora durante lo sviluppo. Nel metodo è possibile ripristinare l'eventuale stato delle variabili salvato in precedenza.
2. ***started***: È visibile all'utente, ma quest'ultimo non può ancora interagirci. Nel passaggio a questo stato va in esecuzione *onStart()*, se l'activity è stata appena creata, oppure *onRestart()* se era stata fermata (stopped) ma non distrutta. All'interno di questi metodi si inizializzano gli elementi di layout con le informazioni necessarie. Dopo questo metodo va in esecuzione *onRestoreInstanceState()*, in cui è possibile ripristinare lo stato delle variabili salvato in precedenza se non è stato già fatto in *onCreate()*.
3. ***resumed***: Ha acquisito il focus, ossia è in primo piano e l'utente può ora interagirci. L'activity rimane in questo stato finché non viene messa in pausa. Per l'occasione, viene eseguito il metodo *onResume()*.
4. ***paused***: L'activity perde il focus, ma è ancora visibile dall'utente. Ciò può accadere, ad esempio, se appare un popup che informa della poca carica residua nella batteria. Quando entra in questo stato viene eseguito il metodo *onPause()*.

5. ***stopped***: L'activity esiste ancora ma non è più visibile. Gran parte delle risorse che usa vengono rilasciate per renderle utilizzabili da altre activity o da altre applicazioni. Lo stato delle variabili definite all'interno dell'activity non fa parte delle risorse rilasciate. Nel passaggio in questo stato viene eseguito *onStop()*. Se l'activity viene messa in background viene eseguito anche *onSaveInstanceState()*, in cui è possibile salvare lo stato delle variabili in modo da poterlo ripristinare in futuro.
6. ***destroyed***: L'activity sta per essere distrutta. Si può arrivare in questo stato per richiesta esplicita dell'utente oppure a causa di cambiamenti nella configurazione del dispositivo. In entrambi i casi viene eseguito *onDestroy()*. Nel primo caso le risorse residue vengono definitivamente liberate. Il secondo caso si verifica quando l'utente ruota il dispositivo oppure quando il sistema chiude l'activity in background per mancanza di memoria a disposizione. Nel secondo caso quindi, quando l'utente riapre l'activity, essa viene ricreata da capo.

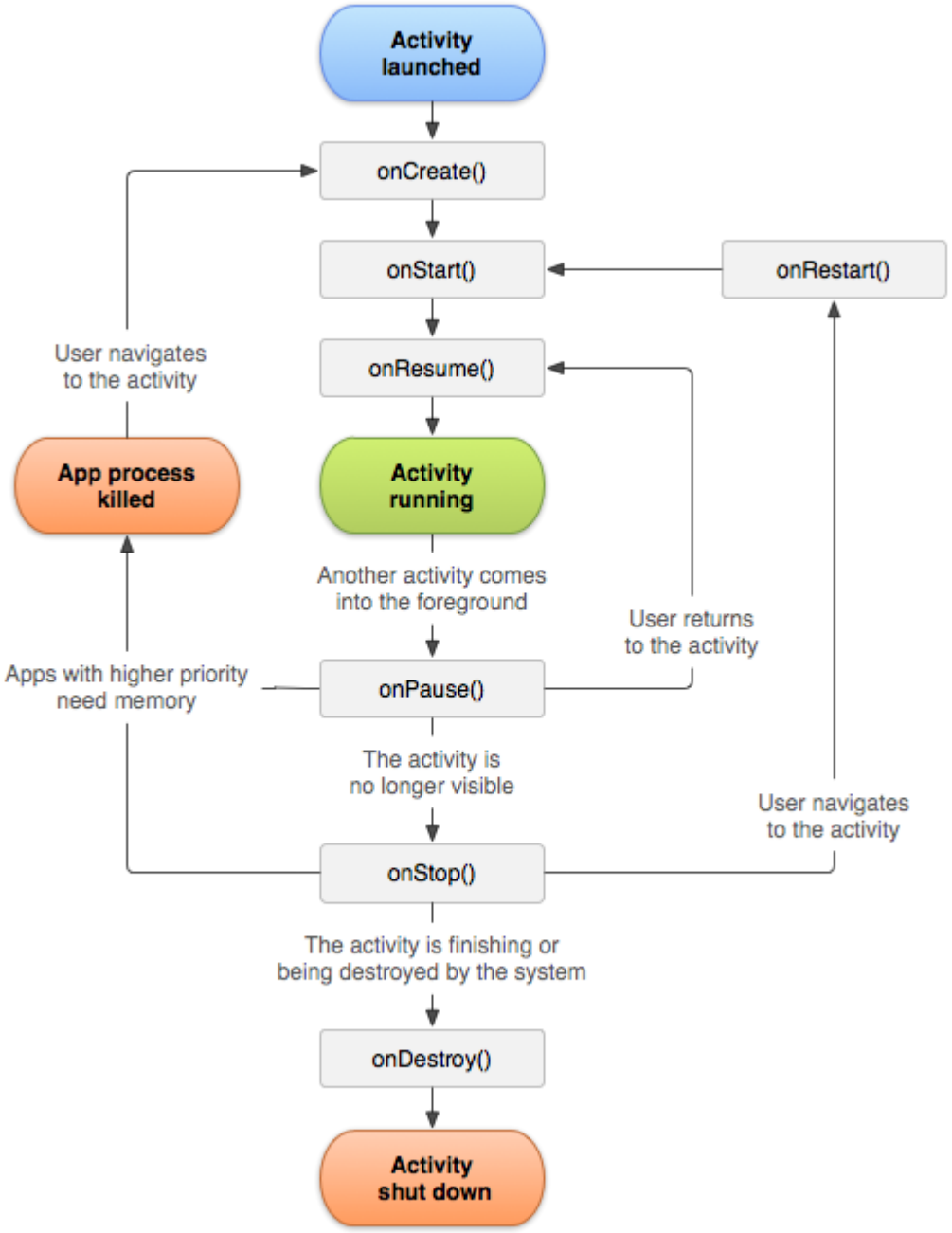

Figura 3.16 - Schema semplificato del lifecycle di un'activity.

Per il salvataggio ed il recupero dello stato delle variabili si sfruttano i Bundle, strutture dati capaci di memorizzare coppie chiavi-valore. La chiave è di tipo stringa e serve a identificare univocamente il dato ad essa associato, il dato può essere di qualsiasi tipo. Una volta creatone uno, è possibile aggiungere un nuovo elemento con il metodo *putX()* (dove X è il tipo di elemento da inserire), specificando come parametri la coppia chiave-valore. Per estrarlo, si deve invece usare *getX()*, inserendo come parametro la chiave.

Un altro elemento molto importante nella gestione delle interfacce sono i Fragment[15] (frammenti). Un fragment rappresenta una porzione dell'interfaccia utente dell'applicazione e, di conseguenza, definisce e gestisce la sua interfaccia grafica, gestisce gli eventi che intercetta ed ha un proprio ciclo di vita, separato da quello delle activity.
Nonostante la loro natura diversa dalle activity, i fragment non possono esistere in autonomia. Per poter esistere, infatti, un fragment deve essere ospitato da un'activity oppure da un altro fragment, e la sua distruzione avviene in automatico durante la distruzione dell'ospitante.
I benefici più importanti nell'uso dei fragment consistono nella modularità e nel riutilizzo di componenti dell'interfaccia, e sono applicabili in diversi modi:

- Definita un'activity e l'operazione ad essa associata, è possibile implementare al suo interno tutti gli elementi grafici in comune nelle varie sotto-operazioni in cui è suddivisa l'operazione originaria. Per ciascuna di queste fasi, invece, si implementa un fragment contenente tutti gli elementi grafici specifici per poter essere completata.
- È possibile usare lo stesso fragment, anche se riempito con dati diversi, più volte nella stessa activity oppure in activity diverse.
- È possibile visualizzare un numero variabile di fragment nello stesso momento in base alle dimensioni dello schermo (ad esempio, se si visualizza un'activity su uno smartphone si ha un solo fragment, se lo si fa su un tablet ne compaiono due contemporaneamente).

Il ciclo di vita dei fragment è simile a quello delle activity. La differenza sostanziale sta nella mancanza degli stati *paused* e *stopped* (Figura 3.17). I possibili stati, e relativi metodi, diventano:

- ***created***: Il fragment viene creato ed inizializzato, per l'occasione viene eseguito *onCreate()*. Si torna in questo stato anche quando scompare dalla vista dell'utente e parte delle risorse che occupa vengono rilasciate. In questo caso viene eseguito *onStop()*. A seguito di quest'ultimo si esegue *onSaveInstanceState()* per salvare lo stato delle variabili in modo analogo a quanto visto per le activity.
- ***started***: Il fragment diventa visibile ma non è ancora possibile interagirci, per l'occasione viene eseguito *onStart()*. Si torna in questo stato anche quando il fragment perde il focus ma rimanendo visibile all'utente. Per l'occasione viene eseguito il metodo *onPause()*.
- ***resumed***: Il fragment è pronto ad interagire con l'utente, per l'occasione viene eseguito il metodo *onResume()*.
- ***destroyed***: Il fragment sta per essere distrutto. Le ultime risorse ancora occupate vengono rilasciate e viene eseguito *onDestroy()*.

Un'altra caratteristica che separa i fragment dalle activity sta nel fatto che le interfacce (o view) in essi contenuti hanno un loro ciclo di vita, separato da quello dei loro ospitanti (Figura 3.17). Nonostante questo, ciascun fragment assume la proprietà del ciclo di vita dell'interfaccia in esso contenuto. Di conseguenza, la sua distruzione comporta il termine di entrambi i lifecycle. A causa di questa separazione, il loro ciclo di vita è caratterizzato da metodi diversi:

- ***onCreateView()***: Viene eseguito dopo *onCreate()*, in questo metodo viene creata l'interfaccia ed avviene l'inflating del layout grafico.
- ***onViewCreated()***: Viene eseguito subito dopo *onCreateView()*. È il luogo ideale per inizializzare gli elementi del layout con i dati necessari, iniziare ad osservare eventuali live data ed inizializzare gli adapter per le liste.
- ***onViewStateRestored()***: È il luogo per aggiornare i dati a schermo con i valori salvati in precedenza in *onSaveInstanceState()*. Viene eseguito subito dopo *onViewCreated()*.
- ***onDestroyView()***: Luogo per svolgere le operazioni a seguito della distruzione dell'interfaccia. Viene eseguito subito dopo *onSaveInstanceState()* e prima di *onDestroy()*.

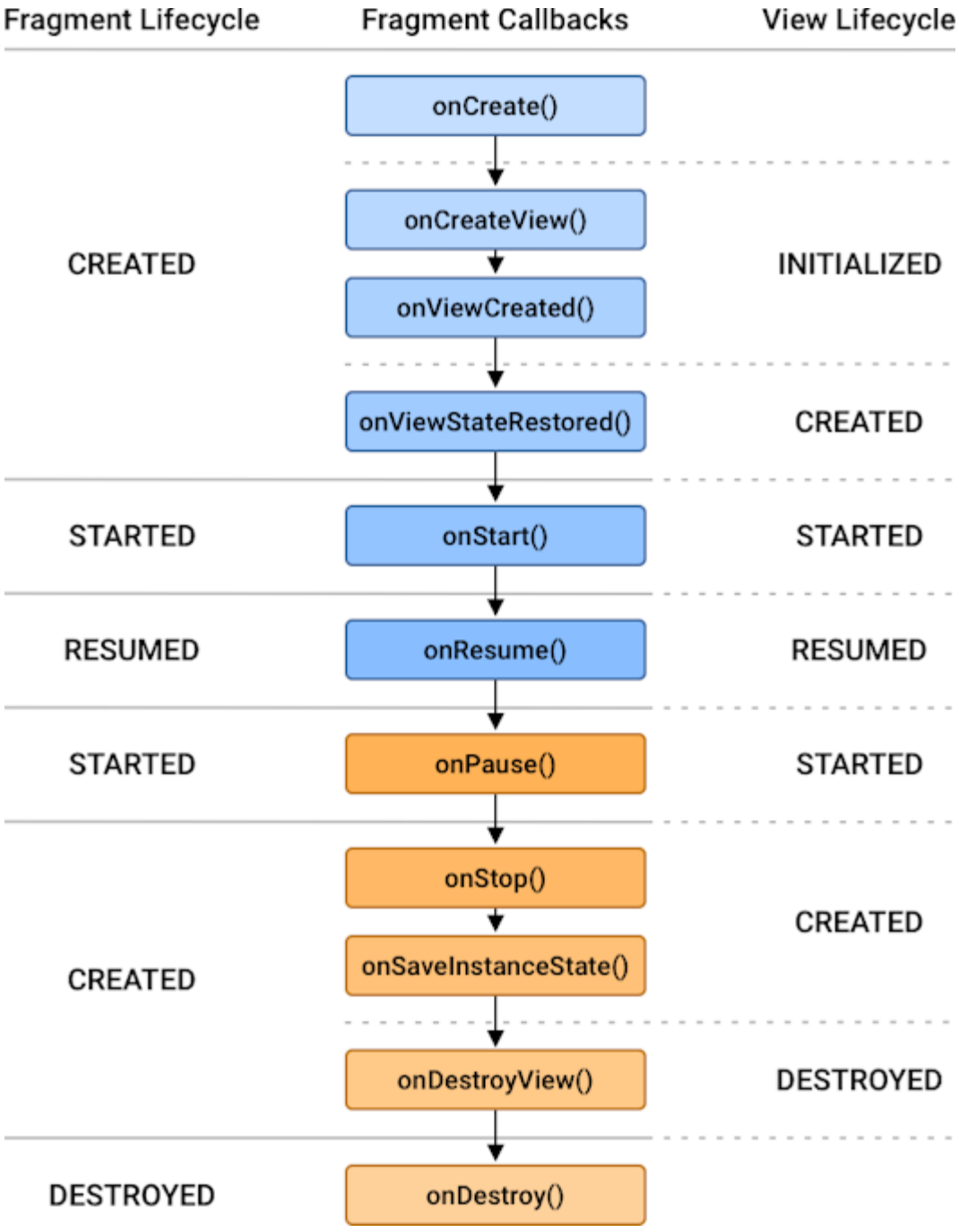

Figura 3.17 - Schema semplificato del lifecycle dei fragment e delle view.

3.4.2 Navigazione all'interno dell'applicazione

In una qualsiasi applicazione Android, la navigazione si riferisce ad una serie di interazioni che consentono all'utente di spostarsi tra i vari contenuti dell'app.
Lo spostamento tra due activity è implementabile attraverso gli Intent[16], particolari oggetti progettati per effettuare una richiesta al sistema operativo Android. In questo caso, la richiesta consiste nell'avviare un'activity, e si possono distinguere due tipi di intent:

- **Espliciti**: Viene richiesta la creazione di una specifica activity, che può appartenere alla stessa applicazione di partenza oppure ad un'app totalmente diversa. Per farlo è necessario creare un oggetto Intent, usando il costruttore dell'omonima classe, e poi usarlo per lanciare l'activity desiderata. Nella creazione dell'oggetto è necessario indicare il nome della classe che implementa la destinazione, mentre per l'avvio è necessario usare il metodo *startActivity()*.
- **Impliciti**: Viene richiesto ad Android di trovare un'activity in grado di eseguire una determinata azione scelta dall'utente. Per farlo la procedura è analoga al caso esplicito, ma bisogna specificare l'azione da svolgere al posto della classe. Prima di lanciare l'intent, inoltre, conviene eseguire *resolveActivity()* per verificare se esiste un'activity capace di svolgere l'azione.

In entrambi i casi, è possibile aggiungere dei dati da inviare al ricevente attraverso i bundle. Per lo spostamento tra due fragment è necessario usare le dipendenze già elencate nella sezione 3.3 e, di conseguenza, è necessario implementare alcuni oggetti particolari:

- **Host**: È un elemento dell'interfaccia che ha il compito di visualizzare la destinazione corrente. Per funzionare, deve essere necessariamente ospitato all'interno di un'activity.
- **Destinazione**: Corrisponde ad un fragment raggiungibile dall'utente, visualizzarla corrisponde a mostrare gli elementi definiti nel fragment. Durante la navigazione, l'app di fatto cambia la destinazione all'interno dell'host. È spesso indicata come *NavDestination*.
- **Grafo**: Una struttura dati che definisce tutte le destinazioni raggiungibili durante la navigazione, rappresentate dai nodi, e come sono collegate fra loro, rappresentate invece dagli archi. Viene indicato con il termine *NavGraph*.
- **Azione**: Identifica una transizione tra due destinazioni e gli eventuali dati richiesti da quella di arrivo, ciascuna azione è identificata univocamente attraverso un id.
- **Controller**: Gestisce la navigazione tra destinazioni. Quest'oggetto offre metodi per effettuare la navigazione, gestire il *back stack* e molto altro. È indicato con *NavController*.

Il *back stack* è una struttura dati contenente tutte le destinazioni visitate, durante la navigazione il *NavController* aggiunge o rimuove destinazione da esso in base alle azioni dell'utente. Viene gestita con approccio LIFO (Last In First Out), gli elementi contenuti vengono sempre aggiunti e rimossi partendo dagli ultimi presenti. Un possibile elemento utilizzabile come host è il *NavHostFragment*, un particolare fragment progettato appositamente per questo scopo. A ciascun *NavHostFragment* è associato un *NavController*, al quale si può accedere usando il metodo *findNavController()*. Questo metodo può essere eseguito all'interno dell'activity contenente l'host oppure dentro un qualsiasi fragment visualizzabile al suo interno.
Per effettuare lo spostamento in sé si utilizza il metodo *navigate()*, definito nel *NavController*, passando come parametro l'identificatore associato all'azione desiderata.
Nonostante questa funzionalità sia stata pensata per la navigazione tra fragment senza dover cambiare activity, è possibile usarla per navigare anche tra queste ultime, anche se in questi casi è più conveniente usare gli intent.

3.4.3 Linguaggio XML

Oltre a Kotlin/Java, un progetto Android richiede spesso l'utilizzo di un altro linguaggio, ossia il linguaggio di markup XML[17] (EXtensible Markup Language).
Gli elementi di base di ogni linguaggio di markup sono i tag, ossia stringhe racchiuse all'interno di due parentesi angolari. Un elemento è definito da una coppia di tag, uno di apertura ed uno di chiusura, quello di chiusura ha la stessa sintassi di quello di apertura salvo per l'aggiunta del carattere / dopo la parentesi iniziale. La stringa contenuta all'interno delle parentesi definisce il nome dell'elemento. Al posto di utilizzare il tag di chiusura è possibile semplificare la struttura, dove possibile, aggiungendo il carattere nel tag iniziale prima della parentesi di chiusura. Ogni documento creato in XML ha una struttura gerarchica: ciascun elemento può contenere al suo interno altri elementi oppure del testo normale. La caratteristica più importante di XML è il fatto di essere un metalinguaggio per creare altri linguaggi di markup. Oltre agli elementi predefiniti, infatti, è possibile aggiungerne altri personalizzati, definendo di fatto nuovi linguaggi in base alle necessità. Gli elementi possono contenere anche degli attributi, ossia coppie chiave-valore contenente informazioni aggiuntive. Questi vanno dichiarati all'interno del tag di apertura, subito dopo il nome.
Data l'estrema modularità, per costruire un linguaggio partendo da XML è necessario costruire uno schema che ne definisce la struttura, ossia:

- Quali tag sono permessi e quai relazioni gerarchiche esistono fra di loro.
- Quali tipi di dato si possono associare a ciascun tag.
- Quali attributi può contenere ciascun tag. Per ciascun attributo deve essere indicato che tipo di dati sono consentiti, se sono opzionali oppure obbligatori e, nel caso fossero opzionali, il loro valore di default.

Lo standard attuale per la creazione di uno schema è l'XSD (XML Schema Definition). Una volta implementato, lo schema funge anche da spazio dei nomi. Per fare in modo che un tag rispetti i vincoli di un determinato schema, infatti, è necessario anteporre il suo nome al nome del tag.
L'importanza di schemi e spazio dei nomi aumenta notevolmente se si considera che non esiste un limite al loro numero in un documento XML. È molto probabile, quindi, che due schemi diversi usino gli stessi nomi per tag e/o attributi. Per dichiarare l'utilizzo di uno schema all'interno di un documento è necessario usare l'attributo *xmlns* all'interno del tag radice (Figura 3.18). Questo attributo richiede una coppia di valori: il primo è il nome dato allo schema ed il secondo è un URI al file che lo implementa (spesso l'URI corrisponde ad un URL da cui è possibile recuperare il file).
Nel contesto Android sono espressi tramite XML varie componenti:

- I file Android Manifest, documenti che contengono e descrivono alcune caratteristiche di base di un'applicazione (come, ad esempio, le activity che la compongono).
- Le risorse non sotto forma di codice eseguibile: layout grafici, elementi di un menu, grafi per la navigazione, definizione di stringhe, array di valori, colori, icone, dimensioni, temi, stili e tanto altro.

Tutti gli elementi definiti per lo sviluppo Android hanno a disposizione l'attributo *id*. Quest'ultimo è particolarmente importante poiché consente di definire un identificatore univoco agli elementi all'interno del file XML. L'identificatore prende il nome di Resource ID, e viene usato per accedere ad un particolare elemento all'interno del codice dell'app.

Un tag, invece, particolarmente importante per il contesto Android è *data*, il quale definisce una sezione in cui è possibile dichiarare delle variabili (Figura 3.18). Una variabile è definita usando l'elemento *variable*, ed ha due attributi a disposizione:

- ***name***: Il nome della variabile.
- ***type***: Il tipo di variabile, spesso contiene la locazione della classe che implementa l'oggetto.

L'utilizzo di questa sezione è legato ad una funzionalità di Android Studio chiamata Data Binding.

```
<layout
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    tools:context=".ui.order.FragmentFoodIngredients">
    <data>
        <variable
            name="viewModel"
            type="com.projectrestaurant.viewmodel.FoodOrderViewModel"/>
    </data>
```

Figura 3.18 - Esempio di sezione contenente variabili. Da notare anche la definizione di più schemi nell'elemento posto come radice.

3.4.4 View Binding e Data Binding

Per rendere un'activity oppure un fragment in grado di interagire con l'utente è necessario ottenere all'interno del loro file sorgente dei riferimenti agli oggetti grafici (o View) in cui si vuole implementare l'interazione. Il modo più semplice per farlo è usare il metodo findViewById() passandogli come parametro il resource id della risorsa cercata. Questo metodo restituisce il riferimento alla view desiderata. Assegnando il riferimento ad una variabile si ottiene di fatto un oggetto, gli attributi definiti nel file XML sono ora accessibili come normali attributi di un oggetto. Questo sistema ha però alcuni svantaggi:

- Il metodo può restituire un riferimento nullo, causando un crash dell'applicazione a runtime. Questo accade se la risorsa cercata non esiste nel layout corrente.
- Si rischia di assegnare il riferimento ad una variabile di tipo diverso da quello effettivo, causando così un errore di conversione a runtime.
- È necessario evocare il metodo per ogni view a cui si vuole associare un oggetto.

Per facilitare il lavoro, Android Studio mette a disposizione una funzionalità chiamata View Binding[18]. Quest'ultima genera automaticamente, per ogni file di layout, una classe contenente gli oggetti associati agli elementi grafici presenti nel layout.
I nomi delle classi generate seguono una specifica convenzione: si aggiunge il suffisso *Binding* al nome del file XML opportunamente modificato per renderlo conforme alla convenzione usata per le classi (ad esempio, per il file *activity_main.xml* si ottiene la classe *ActivityMainBinding*).

La generazione automatica di queste classi porta a diversi benefici:

- Codice semplificato: Non è più necessario scrivere diverse righe di codice per recuperare gli oggetti desiderati, ma si può fare il tutto in poche righe.
- Evita errori di conversioni: Le view sono già accessibili con il tipo corretto (Type Safety).
- Evita la possibilità di riferimenti nulli: I riferimenti riguardano solo le view attualmente presenti nel layout, evitando così elementi inesistenti (Null Safety). In caso di modifiche al layout, inoltre, i riferimenti si aggiornano automaticamente in modo opportuno.

Per permettere al view binding di generare un oggetto a partire da un elemento nel layout, quest'ultimo deve avere un resource id. Questo perché i nomi degli oggetti corrispondono alle stringhe nel resource id, opportunamente modificate secondo la convenzione usata per i nomi di variabile (ad esempio, la stringa *main_textview* viene convertita in *mainTextview*).
Una volta generata la classe, è necessario ottenere una sua istanza per avere i riferimenti alle view ed effettuare l'inflating del layout, entrambe queste operazioni vengono svolte nel metodo *onCreate()* (per le activity) oppure *onCreateView()* (per i fragment). Per la prima parte è necessario eseguire il metodo statico *inflate()*, incluso di default nella classe generata. Per la seconda fase, invece, bisogna eseguire *setContentView()* passandogli il riferimento della view radice (la view che contiene tutte le altre) incluso nell'oggetto di binding.

Un'altra funzionalità di Android Studio simile al view binding è il Data Binding[19]. Quest'ultima consente di legare fra loro elementi di layout e dati definiti nell'app usando un approccio dichiarativo. L'utilizzo di questa funzionalità comporta due importanti vantaggi:

- Facilità di implementazione e di manutenzione nel lungo periodo.
- Semplificazione del codice, rendendolo così più comprensibile ad altri e riducendo contemporaneamente la probabilità di errori.

Per consentire al layout grafico di operare con in data binding, è necessario racchiuderlo all'interno del tag *layout*. Fatto questo, si può procedere a definire i dati usando i tag *data* e *variable* e ad effettuare il collegamento vero e proprio dove necessario. L'accesso ad una variabile all'interno del file di layout avviene racchiudendola all'interno di una coppia di parentesi graffe precedute da un asterisco. Tramite questo sistema è possibile anche utilizzare espressioni all'interno dei file di layout, ad esempio per effettuare semplici operazioni di confronto.

Sia il data binding che il view binding sono stati pensati per lavorare con i moduli. È necessario, quindi, attivarli per ogni modulo che ci interessa. L'attivazione avviene agendo sul file *build.gradle.kts*, in particolare sulla sezione *buildFeatures*. Una volta raggiunta questa sezione, bisogna impostare al valore *true* rispettivamente le opzioni *dataBinding* e *viewBinding* (Figura 3.19). Il loro utilizzo non è mutualmente esclusivo, di conseguenza conviene usarli entrambi, dove possibile. Nella Figura 3.20 mostra invece un esempio di utilizzo del data binding all'interno di file XML.

```
buildFeatures {
    viewBinding = true
    //noinspection DataBindingWithoutKapt
    dataBinding = true
}
```

Figura 3.19 - Attivazione delle due funzionalità.

```
<data>
    <variable
        name="viewModel"
        type="com.projectrestaurant.viewmodel.FoodOrderViewModel"/>
</data>
<TextView
    android:id="@+id/text_view_quantity"
    style="@style/TextViewTitle"
    android:layout_height="50dp"
    android:layout_marginTop="16dp"
    android:text="@{viewModel.foodQuantity.toString()}"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/materialDivider"/>
```

Figura 3.20 - Esempio di utilizzo del data binding.

3.4.5 Recycler View e Adapter

Se si vuole mostrare a schermo delle liste bisogna ricorrere ad un particolare elemento di layout chiamato Recycler View[20]. Di fatto, un recycler view funge da contenitore in cui è possibile visualizzare gli elementi della lista. Prima di implementare un recycler view, è necessario creare un file di layout per i singoli elementi della lista. Analogamente ai file per activity e fragment, è possibile sfruttare view binding e data binding per semplificare l'integrazione con il codice.
Per poter funzionare, oltre alla sorgente di dati in sé, richiede la presenza di altri oggetti:

- Un Layout Manager: si occupa di definire l'orientamento degli elementi nella lista. Si può, ad esempio, scegliere di mostrarli in sequenza orizzontale, verticale oppure a griglia.
- Un Adapter: gestisce la visualizzazione a schermo dei singoli elementi, oltre che gestire la loro interazione con l'utente.

Oltre a questi due, ne esiste un altro opzionale chiamato Item Decoration che si occupa di disegnare decorazioni aggiuntive, come ad esempio dei separatori tra i singoli elementi.
Per ogni lista l'adapter, a differenza degli altri componenti, deve essere implementato dagli sviluppatori estendendo le classi astratte *Adapter* oppure *ListAdapter*.
Per farlo, è necessario in entrambi i casi implementare una classe interna e sovrascrivere due metodi:

- La classe interna estende a sua volta *ViewHolder*, contiene i riferimenti alle view presenti nel layout, oltre ad eventuali dati aggiuntivi. Per farlo, bisogna passare al costruttore l'elemento radice del layout grafico.
- Il metodo *onCreateViewHolder()* crea una nuova istanza di *ViewHolder* per ogni elemento della lista. Per farlo è necessario prima effettuare l'inflating del layout grafico.
- Il metodo *onBindViewHolder()* inizializza gli elementi contenuti nel nuovo oggetto *ViewHolder* appena creato con i dati della sorgente.

Nel primo caso è necessario sovrascrivere anche il metodo *getItemCount()*, che restituisce il numero di elementi presenti nella sorgente dati. La classe *Adapter*, inoltre fornisce il metodo *notifyDataSetChanged()* per aggiornare tutti gli elementi a schermo in caso di modifiche alla sorgente dati, a prescindere se il singolo elemento sia stato modificato oppure no. In caso di aggiunte, rimozioni o modifiche di specifici elementi, si possono usare rispettivamente *notifyItemInserted()*, *notifyItemRemoved()* e *notifyItemChanged()*. Il problema principale riguardo questi quattro eventi sta nel fatto che devono essere manualmente gestiti dagli sviluppatori con del codice su misura, rendendo di fatto la classe *Adapter* poco pratica per sorgenti di dati dinamiche durante l'esecuzione.
La classe *ListAdapter*, invece, si distingue per la sua forte integrazione con la classe DiffUtil. Quest'ultima fornisce un modo per calcolare la differenza tra due sorgenti di dati ed aggiornare automaticamente solo gli elementi della lista che hanno subito una modifica. Il calcolo avviene in un thread secondario, senza così compromettere le prestazioni dell'applicazione. Per rendere operativa la funzionalità, è necessario implementare una classe che estende DiffUtil.Callback, in questa nuova classe andranno sovrascritti i metodi *areItemsTheSame()* e *areContentsTheSame()*. Questi due implementano rispettivamente i criteri, definiti dagli sviluppatori dell'app, per stabilire se due elementi sono in realtà lo stesso oggetto e se due elementi diversi hanno lo stesso contenuto. La classe astratta mette a disposizione anche il metodo *submitList()*, che prende come parametro la nuova sorgente dati da mostrare a schermo. Una volta eseguito, provvede automaticamente a calcolare la differenza con la sorgente già esistente e di conseguenza aggiornare gli elementi a schermo secondo le modalità descritte. Gli sviluppatori, quindi, devono solo preoccuparsi delle modifiche alla sorgente dati e alla definizione dei criteri per calcolare le differenze tra due elementi.

3.4.6 Coroutine di Kotlin

Tra le funzionalità di Kotlin vi sono le Coroutine[21], ossia un design pattern che consente l'esecuzione di codice in modo asincrono. Il loro caso d'uso sta nella gestione di processi con tempi di esecuzione maggiori rispetto al resto del codice, come ad esempio il recupero di dati da un sito web oppure il loro salvataggio in una memoria di archiviazione. L'esecuzione asincrona evita che questi processi occupino il thread principale dell'app, rendendola di conseguenza incapace di gestire le interazioni dell'utente. Le coroutine hanno diverse caratteristiche:

- Sono progettate in modo da non essere strettamente legate ad un thread. Più coroutine possono essere eseguite sullo stesso thread grazie al meccanismo della sospensione. Una coroutine, infatti, può essere sospesa senza bloccare il thread in cui essa opera, permettendo così ad altre coroutine di prendere il suo posto ed andare in esecuzione. Kotlin usa uno stack per gestire la coroutine in esecuzione e le sue variabili locali. Nel sospenderla, il suo stack viene copiato e salvato per poter essere riutilizzato in seguito. Nella riattivazione lo stack viene recuperato dal punto in cui era stato salvato e ciò consente alla funzione al suo interno di tornare in esecuzione.
- Consentono una riduzione dei memory leak grazie alla concorrenza strutturata. Questo meccanismo è ottenibile grazie al CoroutineScope (ambito della coroutine), un ambiente in cui più coroutine possono operare e che delimita il loro ciclo di vita.

È possibile implementare delle funzioni in grado di sospendersi tramite il modificatore *suspend*. Questi tipi di funzioni possono essere lanciate solo all'interno di una coroutine oppure di un'altra funzione suspend. Allo stesso tempo, una coroutine può essere creata solo all'interno di una di questo tipo di funzione.

La creazione di una coroutine inizia specificando il CoroutineScope di riferimento ed usando il suo metodo dedicato. Esistono due possibili metodi:

- ***launch()***: Viene usato per le coroutine che non deve restituire alcun risultato al chiamante.
- ***async()***: Si usa per lanciare coroutine che devono restituire un risultato tramite la funzione suspend ***await()***.

In Kotlin, le coroutine devono disporre anche dei Dispatchers per poter funzionare. Un dispatcher è un oggetto che specifica in quale thread essa deve essere eseguita, e si occupa della loro riattivazione quando necessario. Ne esistono tre tipi:

- ***Dispatchers.Main***: per lanciare la coroutine nel thread principale di Android. Per evitare rallentamenti, andrebbe usato solo per lavori brevi o aggiornamenti dell'interfaccia.
- ***Dispatchers.IO***: è ottimizzato per lavori di input/output su memorie di archiviazione o tramite reti. Alcuni esempi sono il recupero di dati tramite un sito web o l'interfacciarsi con un database relazionale.
- ***Dispatchers.Default***: pensato per lavori intensivi da svolgere per la CPU come, ad esempio, l'analisi di un file JSON.

Il dispatcher usato va indicato come parametro nei metodi per la creazione delle coroutine. È possibile usare *withContext()* per ridefinire un dispatcher all'interno di un corpo di una funzione suspend.

3.4.7 Database relazionale con Room

L'utilizzo di Room[22] per interfacciarsi con un database relazionale comporta l'implementazione di alcune componenti necessarie per il corretto funzionamento della libreria:

- La classe che implementa il database. Serve come punto di accesso principale per lo strato dell'applicazione contenente i dati permanenti.
- Le Entità, ossia classi rappresentanti le tabelle del database. Le singole istanze di queste classi vengono convertite in righe delle tabelle e viceversa.
- Un Data Access Object (DAO) per ogni entità implementata. Fornisce i metodi con cui è possibile interrogare il database, inserire, modificare e cancellare dati.

La classe del database deve rispettare a sua volta dei requisiti ben precisi:

- Deve essere dichiarata come astratta e deve estendere la classe *RoomDatabase*, che viene fornita dalla libreria.
- La dichiarazione deve essere preceduta dall'annotazione *@Database*, in modo da essere riconosciuta da Room. Come parametri dell'annotazione vi sono i metadati del database, come ad esempio il numero di versione e la lista delle classi corrispondenti alle tabelle contenute.
- Al suo interno deve essere dichiarato per ogni entità un metodo astratto che ha come tipo di ritorno il DAO associato.

Se si utilizza Kotlin come linguaggio di programmazione, le entità sono generalmente implementate tramite delle data class, ovvero classi pensate per contenere esclusivamente attributi. Per ogni classe rappresentante un'entità, la dichiarazione va preceduta con l'annotazione *@Entity*, in modo da essere riconosciuta e processata dalla libreria.

Tra i vari parametri che si possono includere nell'annotazione, i più importanti sono:

- ***tableName***: Contiene il nome che avrà la tabella nel database. Se non viene specificato, Room userà lo stesso nome della classe.
- ***primaryKeys***: Un vettore contenete i nomi degli attributi che avranno la funzione di chiave primaria. Va usato solo se si prevede di usare come chiave primaria la combinazione dei valori di più colonne.
- ***ignoredColumns***: Vettore contenente i nomi degli attributi che non devono essere presenti come colonne nella tabella finale. Questo consente di riutilizzare la classe in altri contesti dell'applicazione dove è però necessario l'uso di più attributi.
- ***foreignKeys***: Vettore contenente le chiavi esterne presenti nella tabella. Per ciascun elemento del vettore va indicato: la classe contenente la chiave primaria, il nome della tabella corrente, il nome dell'attributo nella tabella in cui funge da chiave primaria e cosa fare in caso di modifica e cancellazione della chiave primaria.
- ***indices***: Vettore contente gli indici definiti per la tabella. Per ogni indice è importante specificare la lista delle colonne della tabella da indicizzare.

Allo stesso tempo, esistono annotazioni anche per i singoli attributi:

- ***@Ignore***: Ha una funzionalità analoga al parametro *ignoredColumns*.
- ***@PrimaryKey***: Indica l'attributo come chiave primaria. È possibile usarla una sola volta per classe, e ne è obbligatoria la presenza in caso di assenza di *primaryKeys* nell'annotazione dell'entità. Ha anche il parametro *autoGenerate*, che di default è impostato a *false*, per specificare a Room di assegnare automaticamente un valore all'attributo nel momento della creazione di una nuova istanza.
- ***@ColumnInfo***: Permette una personalizzazione aggiuntiva dell'attributo. Ad esempio, è possibile usare il parametro *name* per specificare un nome diverso alla colonna della tabella oppure *defaultValue* per specificare il valore di default.

Room si occupa automaticamente di convertire i tipi di dati supportati nei tipi implementati dal linguaggio SQLite, in modo da permettere la memorizzazione dei loro valori. I tipi supportati corrispondono a: numeri (sia interi che decimali), booleani, singoli caratteri e stringhe. Per memorizzare un tipo di dato complesso in una singola colonna è necessario ricorrere a dei convertitori di tipi, ossia dei metodi che indicano a Room come effettuare la conversione da un tipo di dato personalizzato ad uno noto e viceversa. Il modo migliore per implementarli è tramite una classe definita nello stesso file che implementa l'entità. Per permettere a Room di effettuare le conversioni in maniera corretta i singoli metodi devono essere preceduti da *@TypeConverter*, mentre bisogna indicare la classe che li implementa come parametro di *@TypeConverters*. Quest'ultimo va aggiunto alla classe che implementa il database.

Per implementare il DAO associato ad un'entità è necessario creare un'interfaccia o una classe astratta ed annotarla con *@Dao*. Al suo interno andranno inseriti i metodi per svolgere le operazioni desiderate ed annotarle in modo opportuno. Esistono quattro annotazioni possibili:

- ***@Insert***: Definisce un metodo che inserisce i suoi parametri all'interno della tabella appropriata. Ha a disposizione due proprietà, ovvero *entity* ed *onConflict*. La prima specifica l'entità a cui si riferisce il metodo, mentre la seconda cosa fare in caso di conflitti nell'inserimento. Per impostazione predefinita la procedura viene annullata. Un conflitto si verifica nell'inserimento di un'entità con chiavi primarie già esistenti nella tabella.

- ***@Update***: Specifica un metodo in grado di aggiornare specifiche righe della tabella. Le righe in questione vengono automaticamente individuate in base ai valori delle chiavi primarie. In caso di assenza di una riga con la stessa chiave primaria, la relativa modifica non viene apportata. Opzionalmente, è possibile indicare al metodo di restituire un numero intero che rappresenta il numero di righe aggiornate con successo.
- ***@Delete***: Specifica un metodo in grado di eliminare specifiche righe della tabella. La procedura di individuazione delle righe e la gestione di righe mancanti è analoga al caso dell'aggiornamento. Anche qui, è possibile fare restituire al metodo un intero, anche se in questo caso indica il numero di righe eliminate con successo.
- ***@Query***: Permette di implementare query più complesse rispetto a quelle precedenti. La query va passata all'annotazione come parametro di tipo stringa. Se la query personalizzata prevede la restituzione di una o più righe, è necessario indicare un tipo di ritorno opportuno al metodo. È possibile usare i parametri dichiarati nel metodo all'interno della query. Per consentire a Room di effettuare il collegamento, metodo e query devono usare gli stessi nomi per i parametri, e tali nomi devono essere preceduti dai due punti *:* all'interno della query.
- ***@Transaction***: Permette di implementare una transazione, ossia una serie di operazioni che devono essere necessariamente svolte in sequenza e senza interruzioni esterne. All'interno del metodo associato devono essere indicati i metodi per svolgere le singole operazioni.

4. Sviluppo Software

In questo capitolo si entra finalmente nel dettaglio dello sviluppo dell'applicazione. Si partirà da una descrizione dell'architettura utilizzata passando poi alla descrizione del database centrale con cui l'applicazione dovrà interfacciarsi. Infine, si descriverà con dettaglio l'applicazione in sé, sia dal punto di vista delle schermate che della sua implementazione.

4.1 Architettura

Nella definizione dell'architettura si identificano le principali componenti del sistema e come queste interagiscono fra loro. Una volta completata questa fase, si definisce la struttura generale dell'applicazione.

4.1.1 Architettura Android

È molto importante considerare l'architettura nativa di Android[23]. In modo simile a tutti gli altri sistemi operativi esso è diviso in una serie di strati, ognuno dei quali sfrutta lo strato sottostante per fornire servizi a quello sovrastante (Figura 4.1). Partendo da quello più in alto, tali strati sono:

- **Applicazioni:** Include sia le applicazioni di sistema, cioè quelle preinstallate nel dispositivo sia quelle utente, cioè installate a discrezione di quest'ultimo. Le applicazioni di sistema non hanno uno status speciale rispetto a quelle utente, ma forniscono funzionalità utili agli sviluppatori.
- **Java API Framework:** Sono API scritte in Java e Kotlin, e forniscono tutte le funzionalità del sistema operativo. Tra le varie funzionalità ci sono, ad esempio, classi per costruire l'interfaccia grafica di un'applicazione, gestione delle notifiche e gestione del ciclo di vita delle schermate.
- **Android Runtime (ART):** È il luogo dove le applicazioni vengono effettivamente eseguite e ne gestisce tutte le fasi di esecuzione (ad esempio, quando un'applicazione è ancora aperta ma non appare più a schermo). Ogni app è eseguita come processo separato all'interno di un'istanza dell'Android Runtime.
- **Librerie C/C++ native:** Permettono l'accesso a componenti e servizi nativi di Android. Ad esempio, vi sono SQLite, usato per interrogare i database relazionali, ed OpenGL ES, impiegato per il rendering di grafica tridimensionale. In aggiunta a tutto questo vi è la libreria standard di C (Libc), fondamentale per il loro funzionamento.
- **Hardware Abstraction Layer:** Sono interfacce che espongono funzionalità dei dispositivi hardware come librerie, e vengono utilizzate dalle librerie C/C++ native. Alcuni esempi di queste funzionalità sono: le fotocamere, il modulo bluetooth, i sensori e tanti altri.
- **Kernel Linux:** È lo strato più basso, al di sotto di esso vi è l'hardware vero e proprio. Si occupa della gestione di basso livello dei processi, dei thread e della memoria, fornisce funzionalità di sicurezza ed include i driver per le varie componenti hardware.

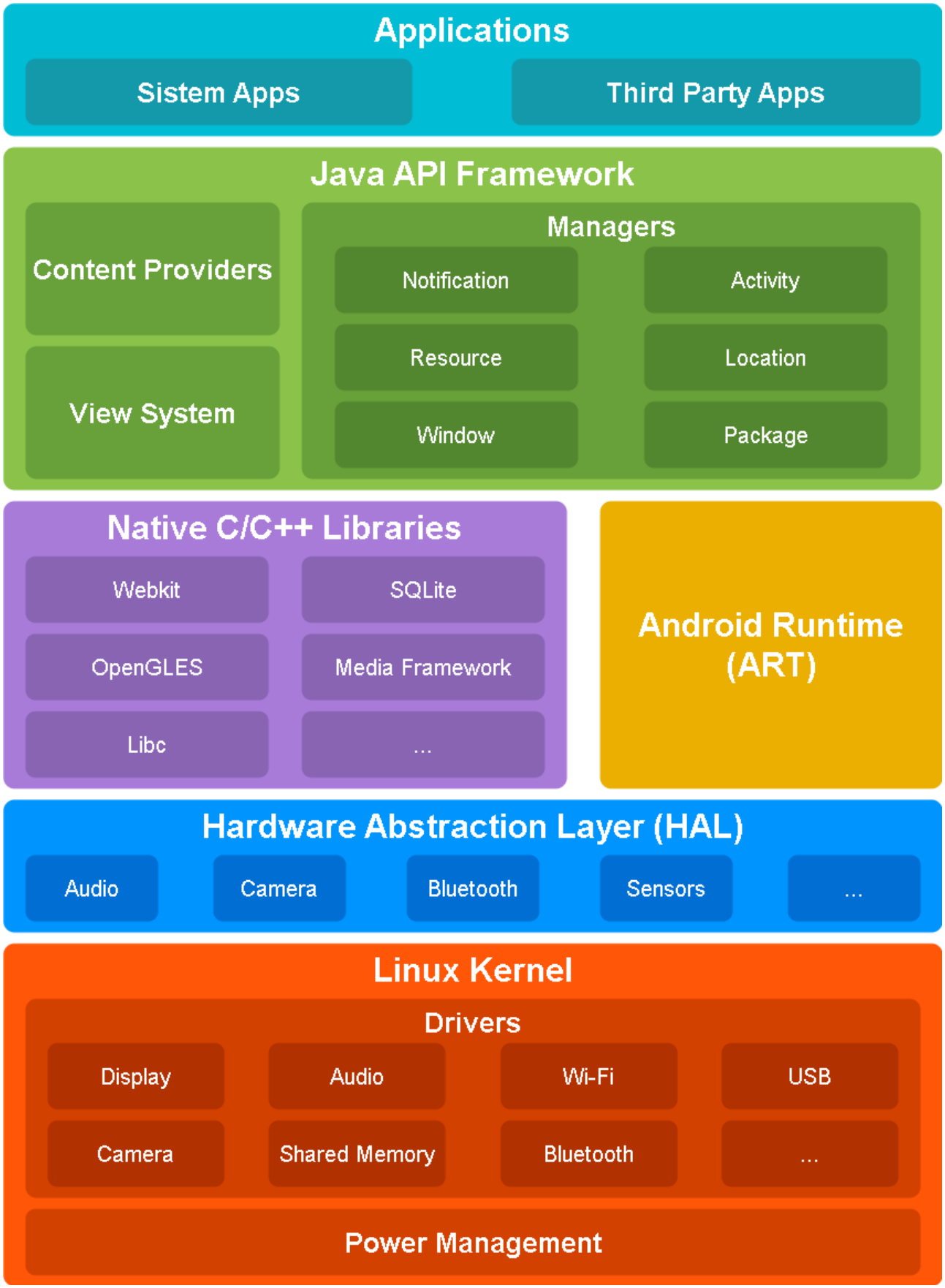

Figura 4.1 - Schema dell'architettura Android

4.1.2 Architettura MVVM

MVVM (Model-View-ViewModel)[24] è un pattern architetturale che permette maggiore scalabilità e manutenzione del codice, per questo motivo è largamente diffuso tra gli sviluppatori, anche al di fuori del contesto Android.
Sfrutta il principio della *separation of concerns*, ovvero le varie componenti di cui è composto il codice devono essere progettate e realizzate in modo separato e, in aggiunta, devono essere poste in una relazione gerarchica. Ciascuna componente può accedere solo agli elementi presenti in quella di livello inferiore (Figura 4.2). Le tre componenti nello specifico sono:

- **View:** Contiene tutte le classi in cui viene gestita l'interfaccia grafica.
- **Model:** Contiene le classi in cui vengono gestiti i dati usati dall'applicazione.
- **ViewModel:** Contiene le classi per gestire gli eventi intercettati dalle View. Sfrutta i dati contenuti nel Model per aggiornare, dove possibile, le informazioni mostrate nelle interfacce attraverso i LiveData, che sono particolari oggetti osservabili dalle View.

I lifecycle delle componenti non sono strettamente legati fra loro, ciascuna istanza delle tre può essere creata e distrutta in momenti diversi durante l'esecuzione.

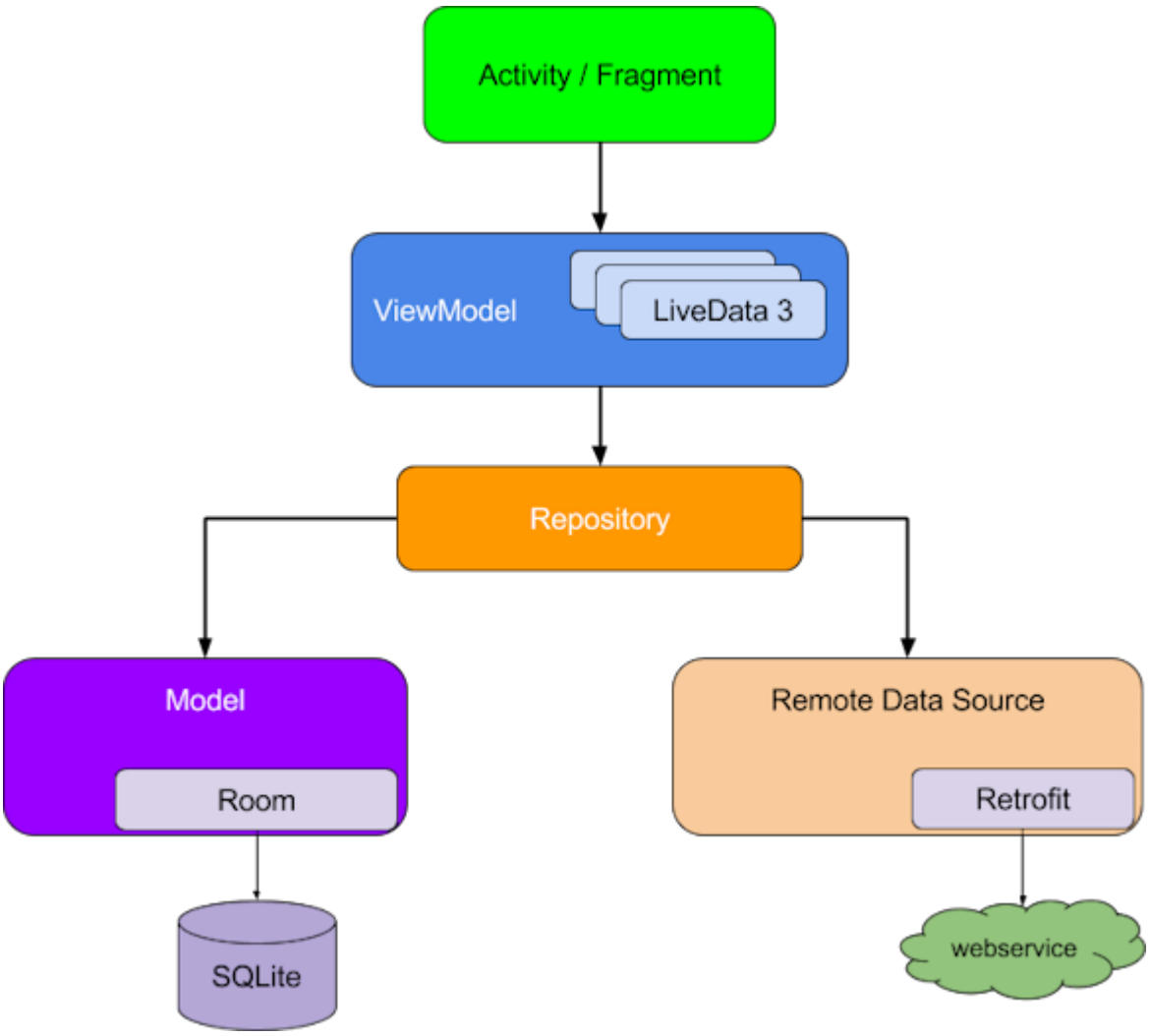

Figura 4.2 - Schema del pattern MVVM.

4.1.3 Architettura del database centrale

L'applicazione usa un Firebase Firestore per memorizzare e gestire i prodotti e gli ordini effettuati dagli utenti in un database centrale accessibile dall'applicazione. In questa sezione viene mostrato e descritta la struttura di questo database, partendo dalle Collection (raccolte) di documenti e finendo ai singoli elementi che compongono i vari documenti.
Collection e documenti possono essere annidati fra loro, un documento può contenere sia campi, ossia coppie chiave-valore a cui viene associato un determinato tipo di dato, sia altre collection.
Nel database è presente la collection *users* che memorizza tutti i documenti relativi agli account utente. Ad ogni account, infatti, è associato un documento che è identificato da un ID alfanumerico univoco. Un discorso analogo riguarda la collection *orders*, che memorizza tutti i documenti relativi agli ordini effettuati dagli utenti.
Le collection *food_types*, *foods* ed *ingredients* memorizzano rispettivamente i documenti riguardanti tipi di prodotti, i prodotti ordinabili e gli ingredienti personalizzabili. A differenza delle collection citate prima, i documenti memorizzati in queste tre usano come identificativi dei numeri interi. Il motivo di questa scelta sarà chiaro più avanti nel descrivere il processo di localizzazione.

Ogni documento della collection *users* contiene, in generale (Figura 4.3), i seguenti elementi:

- ***shopping_cart*** (collection): Collection contenente tutti i prodotti al momento presenti nel carrello della spesa associato. Viene generata nel momento in cui l'utente inserisce qualcosa nel carrello per la prima volta.
- ***delivery_addresses*** (collection): Collection contenente tutti gli indirizzi di consegna memorizzati nel profilo. Viene generata nel momento in cui l'utente registra per la prima volta un indirizzo di consegna.
- ***name*** (stringa): Nome della persona associata al profilo.
- ***surname*** (stringa): Cognome della persona associata al profilo.
- ***email*** (stringa): Indirizzo e-mail associato al profilo.
- ***accepted_privacy_policy*** (booleano): Indica se chi gestisce il profilo ha accettato oppure no l'informativa sulla privacy.
- ***default_delivery_address*** (riferimento): Riferimento al documento contenente l'indirizzo di consegna impostato come quello di default.
- ***order_note*** (stringa): Contiene la nota personalizzata che l'utente vuole aggiungere all'ordine.

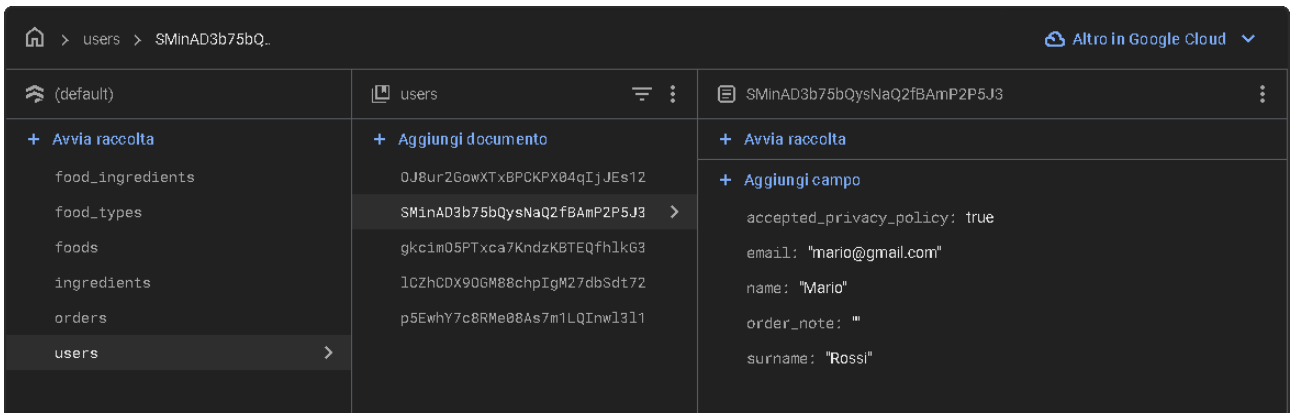

Figura 4.3 - Struttura di un documento in "users" nel database.

Ogni documento nella collection *orders* (Figura 4.4) contiene:

- ***products*** (collection): Contiene tutti i prodotti associati all'ordine.
- ***user*** (riferimento): Riferimento all'account utente che ha effettuato l'ordine.
- ***note*** (stringa): Contiene l'eventuale nota inclusa nell'ordine. Contiene una stringa nulla se non è stata definita nessuna nota.
- ***total_*price** (numero): Il prezzo totale pagato per l'ordine.
- ***order_date_time*** (timestamp): Timestamp (data e ora) in cui è stato effettuato l'ordine.
- ***delivery_date_time*** (timestamp): Timestamp in cui è avvenuta o avverrà la consegna.
- **delivery_*address*** (riferimento): Riferimento all'indirizzo di consegna scelto. È nullo se l'utente sceglie di ritirare i prodotti ordinati direttamente al ristorante.

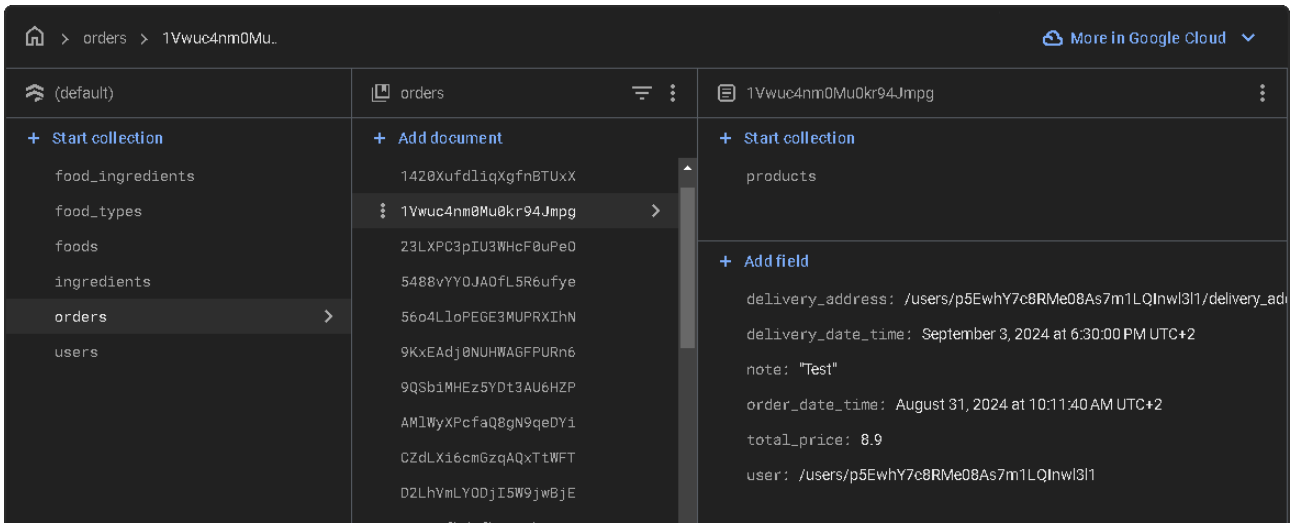

Figura 4.4 - Struttura di un documento in "orders" nel database.

Ogni documento in *food_types* (Figura 4.5) è composto esattamente da tre attributi:

- ***name*** (stringa): Nome del tipo.
- ***image_uri*** (stringa): Indirizzo in cui si trova l'immagine che identifica quel tipo di prodotto.

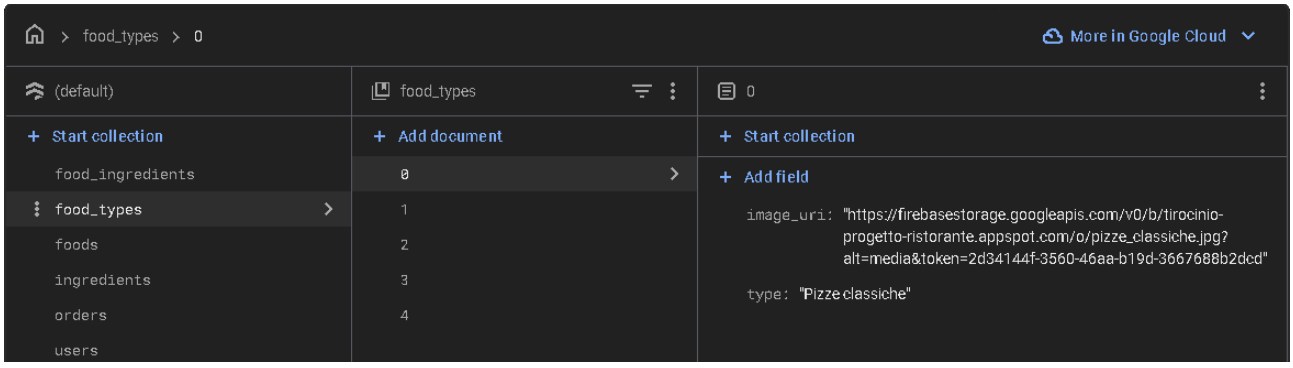

Figura 4.5 - Struttura di un documento in "food_types" nel database.

Ogni documento in *foods* (Figura 4.6) si compone esattamente da quattro campi:

- ***name*** (stringa): Nome del prodotto.
- ***unit_price*** (numero): Prezzo unitario del prodotto.
- ***type*** (riferimento): Riferimento al tipo a cui quel prodotto appartiene.
- ***image_uri*** (stringa): Indirizzo in cui si trova l'immagine che raffigura quel prodotto.

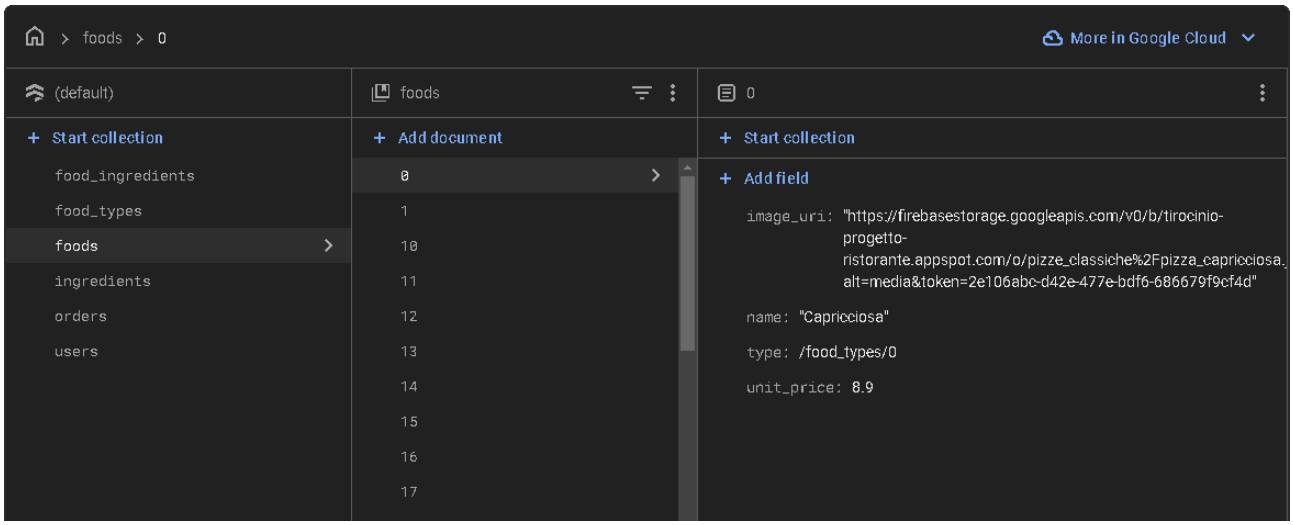

Figura 4.6 - Struttura di un documento in "foods" nel database.

Ogni documento in *ingredients* (Figura 4.7) è costituito esattamente da quattro campi:

- ***name*** (stringa): Nome dell'ingrediente.
- ***unit_price*** (numero): Prezzo unitario dell'ingrediente.
- ***image_uri*** (string): Indirizzo in cui si trova l'immagine che raffigura quell'ingrediente.

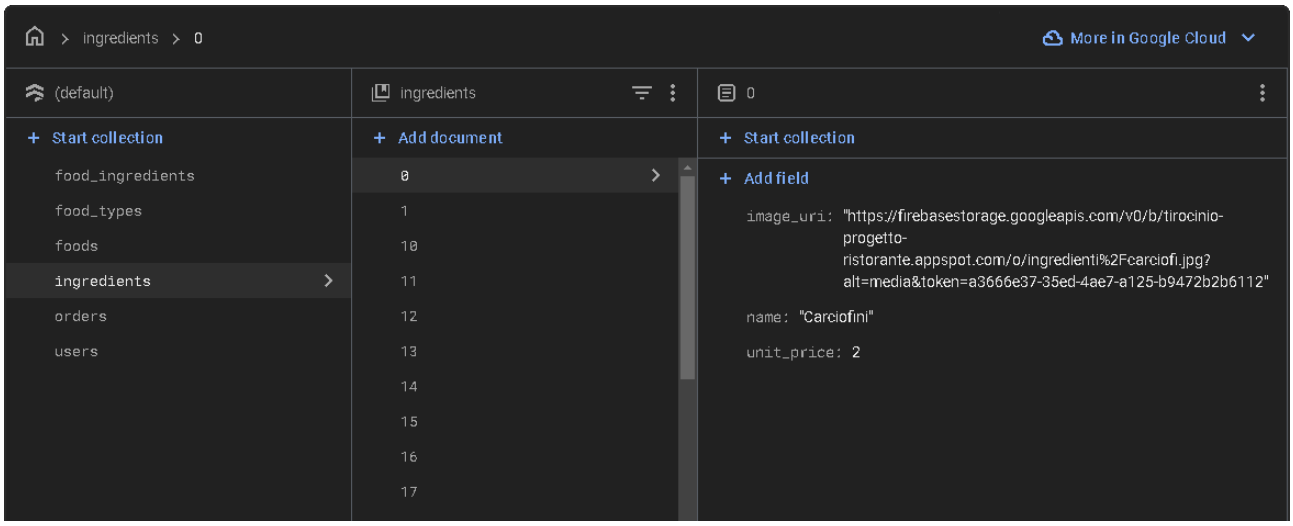

Figura 4.7 - Struttura di un documento in "ingredients" nel database.

Le collection *products*, *shopping_cart* e *delivery_addresses* non appartengono a quelle di primo livello, ma sono importanti tanto quanto queste ultime per il funzionamento dell'app. I documenti presenti in queste collection sono identificati da un ID alfanumerico.

I documenti presenti in *products* e *shopping_cart* (Figura 4.8) condividono la stessa struttura:

- ***extra_ingredients*** (vettore): Contiene i riferimenti agli ingredienti da aggiungere in quantità maggiori rispetto al normale.
- ***removed_ingredients*** (vettore): Contiene i riferimenti agli ingredienti da rimuovere del tutto.
- ***food*** (riferimento): Riferimento al prodotto.
- ***quantity*** (numero): Quantità da ordinare / ordinata.
- ***price*** (numero): Prezzo totale dei prodotti.

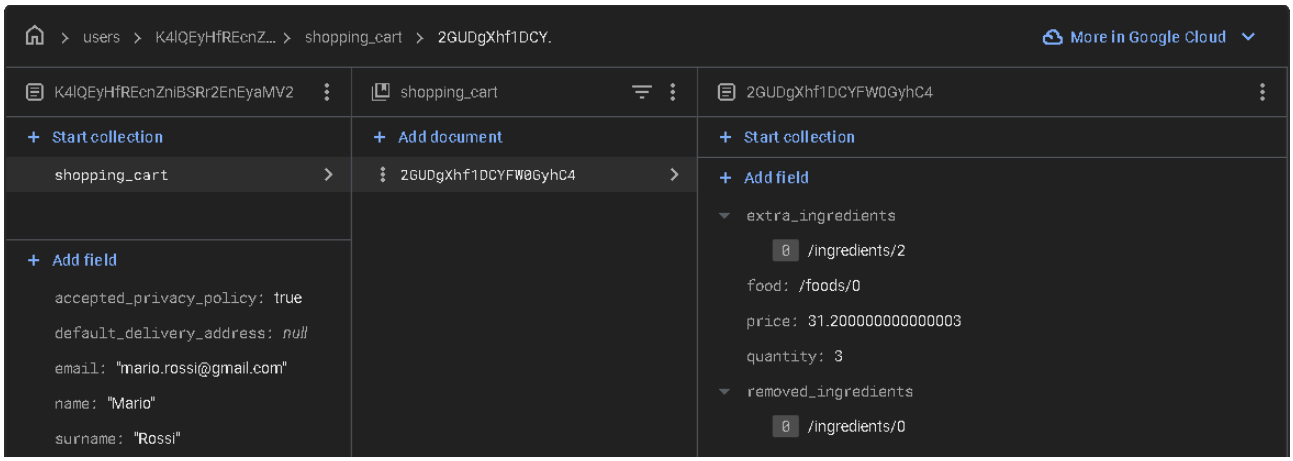

Figura 4.8 - Struttura di un documento in "shopping_cart" (e "products") nel database.

Ogni documento in *delivery_addresses* (Figura 4.9) è composto da:

- ***address*** (stringa): Indirizzo contenente anche il numero civico.
- ***cap*** (stringa): Codice di avviamento postale.
- ***city*** (stringa): Città in cui si trova l'indirizzo.
- ***province*** (stringa): Provincia in cui si trova la città.

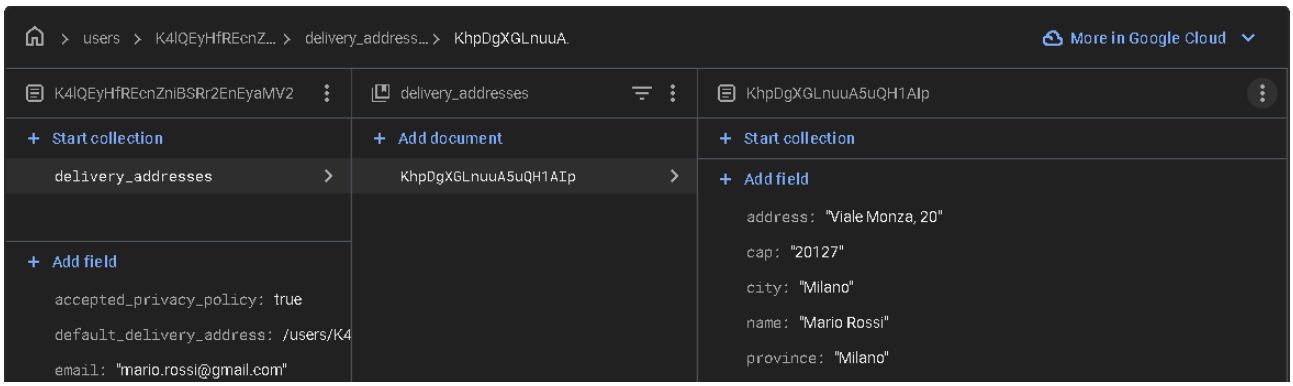

Figura 4.9 - Struttura di un documento in "delivery_addresses" nel database.

4.1.4 Architettura del database locale

L'applicazione sfrutta anche un database relazionale memorizzato sul dispositivo. Questo database viene usato per accedere ai dati quando si è offline è quando non è necessario accedere ai dati sfruttando il database centrale.
Un database relazionale è costituito da tabelle (o entità), in ciascuna di esse vengono memorizzate le singole istanze che fanno parte. In ciascuna tabella, le righe rappresentano le singole istanze memorizzate mentre le colonne gli attributi che le caratterizzano. Ciascuna riga deve essere identificata univocamente nella tabella da almeno un attributo, detto chiave primaria (primary key).
Come attributi di una tabella è possibile definire anche delle chiavi esterne (foreign key), tali chiavi permettono ad ogni istanza di collegarsi con un'altra appartenente ad un'altra tabella. Per implementare il collegamento è necessario creare un attributo dello stesso tipo di quello della chiave primaria a cui ci si vuole collegare e poi inserire per ogni riga il particolare valore della chiave a cui ci si vuole riferire. Per poter interagire con il database è necessario conoscere ed utilizzare il linguaggio SQL (Structured Query Language), attraverso il quale è possibile interrogare il database e ottenere le informazioni richieste.
La Figura 4.10 mostra la struttura del database relazionale implementato nell'applicazione.

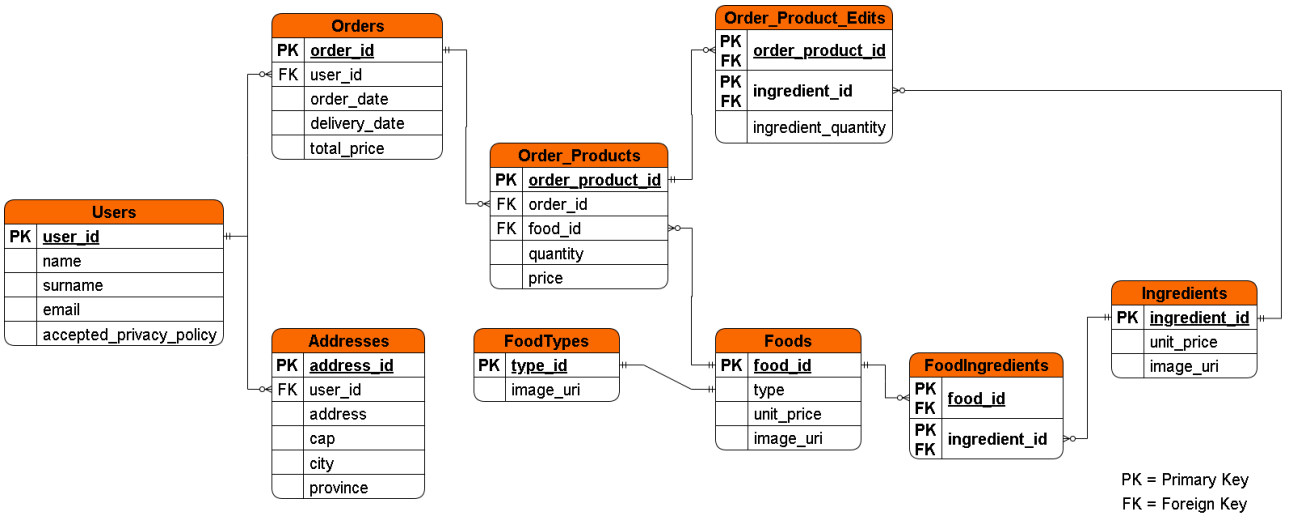

Figura 4.10 - Schema del database relazionale.

La differenza sostanziale rispetto al database centrale sta in come vengono memorizzati gli ordini degli utenti. La lista dei prodotti ordinati è memorizzata nella tabella *Order_Products*, mentre quella delle eventuali modifiche agli ingredienti in *Order_Product_Edits*.
Ciascuna riga in *Order_Products* è identificata dai seguenti attributi:

- ***order_product_id*** (stringa): Chiave primaria.
- ***order_id*** (stringa): Chiave esterna, collega la riga all'ordine specifico a cui appartiene.
- ***food_id*** (numero intero): Chiave esterna, identifica il particolare prodotto ordinato.
- ***quantity*** (numero intero): Quantità ordinata.
- ***price*** (numero decimale): Prezzo totale per quel prodotto.

Ciascuna riga in "Order_Product_Edits" è identificata dai seguenti attributi:

- ***order_product_id*** (stringa): Funge sia da chiave primaria che secondaria.
- ***ingredient_id*** (numero intero): Funge sia da chiave primaria che secondaria.
- ***ingredient_quantity*** (enumerativo): Può assumere solo due valori: *INGREDIENT_REMOVED* e *INGREDIENT_EXTRA*. Indica quale modifica è stata apportata ad un ingrediente in un prodotto di un ordine.

4.2 Il file Android Manifest

Il file Android Manifest[25] è un componente cruciale di un'applicazione Android. È un file XML che fornisce al sistema operativo e al Google Store informazioni sulla struttura dell'app, sui requisiti necessari per funzionare ed i suoi metadati.
Il documento è interamente contenuto all'interno dell'elemento radice *manifest*, e quest'ultimo è diviso al suo interno da diverse sezioni. La più importante di queste è quella contenente i metadati, ed è delimitata dall'elemento *application*. I metadati includono, ad esempio: il titolo, l'icona, il tema utilizzato, la memoria di archiviazione in cui installarla e tanto altro (Figura 4.11).
In questa sezione sono anche elencate le varie activity implementate nell'applicazione, per indicare ciascuna di esse bisogna usare l'elemento *activity* e indicare nell'attributo *name* il percorso alla classe che la implementa. Se una particolare activity è avviabile tramite intent impliciti, bisogna indicare quali azioni può fare e a quali categorie appartiene. Per ciascun intent è necessario creare la sottosezione *intent-filter* ed inserirvi le informazioni opportune su azione e categoria (Figura 4.11).
Nelle altre sezioni in cui è composto *manifest* sono indicate altri tipi di informazioni sull'app:

- Indicazioni su SDK: È delineata dall'elemento uses-sdk, contiene i valori di *minSdkVersion* e *targetSdkVersion* specifici dell'applicazione.
- Permessi: Elenca i permessi di sistema di cui l'app ha bisogno per accedere a componenti protette del sistema oppure svolgere operazioni che coinvolgono dati dell'utente. Ciascun permesso è racchiuso nell'elemento *uses-permission*.
- Librerie condivise: Elenca le librerie condivise richieste dall'applicazione, il sistema operativo provvederà ad aggiungerle al class loader del package. Ciascuna libreria è indicata tramite l'elemento *uses-library*.

```
<uses-permission android:name="android.permission.INTERNET"/>
<application
    android:allowBackup="true"
    android:dataExtractionRules="@xml/data_extraction_rules"
    android:fullBackupContent="@xml/backup_rules"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/Theme.ProjectRestaurant"
    tools:targetApi="31">
    <activity
        android:name=".ui.order.ActivityOrder"
        android:windowSoftInputMode="adjustNothing"
        android:exported="true">
        <intent-filter>
            <action android:name="android.intent.action.MAIN"/>
            <category android:name="android.intent.category.LAUNCHER"/>
        </intent-filter>
    </activity>
```

Figura 4.11 - Parte del file manifest usato dall'applicazione.

4.3 Struttura in Android Studio

Qualsiasi applicazione in Android Studio è strutturata in una maniera ben precisa e facilmente comprensibile anche da chi è alle prime armi. In particolare, la parte dedicata ai file sorgenti è suddivisibile in tre macrogruppi:

- Il file Android Manifest specifico dell'applicazione, sotto il nome di *androidManifest.xml*.
- I file che contengono codice eseguibile, posizionati all'interno del package generale indicato nel file manifest.
- File per tutte le altre risorse necessarie, posizionati all'interno della cartelle *res*[26]. Salvo casi particolari, sono tutti documenti XML.

In questa applicazione, in particolare, il codice è ulteriormente diviso in altri package (Figura 4.12):

- ***ui***: Contiene le classi che si occupano di costruire le activity ed i fragment. È a sua volta diviso in vari sotto-package per le varie sezioni accessibili nell'app:
  - ***aboutus***: Sezione che mostra informazioni aggiuntive sul ristorante.
  - ***account***: Sezione per la gestione del profilo.
  - ***loginregister***: Sezione per creare un nuovo account, fare login o recuperare la password.

    - ***order***: Sezione per l'ordinazione di prodotti.
    - ***privacypolicy***: Sezione che spiega l'informativa sulla privacy del ristorante.
    - ***settings***: Sezione delle impostazioni.
    - ***userorders***: Sezione per visualizzare gli ordini effettuati in passato.
- ***viewmodel***: Contiene tutte le classi che implementano i ViewModel.
- ***database***: Contiene tutte le classi per implementare il database relazionale.
- ***adapter***: Contiene le classi che implementano gli adapter necessari.

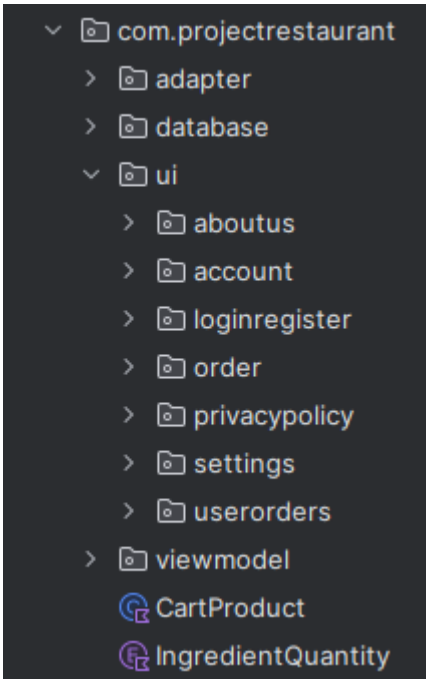

Figura 4.12 - Struttura dei package dell'applicazione in Android Studio.

Sono presenti, inoltre, due file che non appartengono a nessuno dei package appena descritti. Il primo, *IngredientQuantity.kt* implementa un oggetto enumerativo utilizzato durante il processo di modifica degli ingredienti. Il secondo, *CartProduct.kt*, definisce una classe per gestire i prodotti mentre si trovano all'interno del carrello.
La cartella *res* contiene varie sottocartelle in base alla funzione assunta dalle risorse che contengono (Figura 4.13):

- ***color***: Contiene file che definiscono i colori da associare a determinati stati. Si usano, ad esempio, per creare pulsanti personalizzati partendo da altri elementi grafici.
- ***drawable***: Immagini in formato bitmap (PNG, JPEG e tanti altri) oppure XML, quest'ultimo è usato per le immagini in formato vettoriale.
- ***layout***: Documenti XML che definiscono i layout di activity, fragment, ed elementi delle liste. Il view binding genera una classe per ogni elemento presente in questa cartella.
- ***menu***: Contiene file che definiscono le voci presenti nei menu delle app.
- ***mipmap***: Contiene l'icona di avvio applicazione, memorizzata in versioni specifiche per le varie densità in uso dai dispositivi disponibili in commercio.
- ***navigaion***: Contiene i grafi utilizzati per la navigazione all'interno delle app.
- ***values***: File XML contenente valori semplici per rappresentare colori, stringhe, numeri, stili e temi. Dato che può contenere più tipi di risorse, è necessario distinguerle senza creare confusione. Per questo motivo una risorsa deve essere contenuta all'interno dell'elemento *resource* per essere riconosciuta, di conseguenza è possibile scegliere un nome arbitrario per i singoli file. Per convenzione, tuttavia, è conveniente scegliere i seguenti nomi:
    - *arrays.xml* per gli array di valori.
    - *colors.xml* per i colori.
    - *dimens.xml* per i valori rappresentanti dimensioni.
    - *strings.xml* per le stringhe.
    - *styles.xml* per gli stili grafici.
    - *themes.xml* per i temi

- ***xml***: Contiene file xml arbitrari.
- ***fonts***: File per i font con estensioni TTF, OTF o TTC. È possibile usare anche dei file XML contenente l'elemento *font-family*.
- ***raw***: Contiene file non XML arbitrari.

È possibile definire versioni alternative della cartella *values* creando cartelle con lo stesso nome ma aggiungendo suffissi opportuni. Un esempio è applicabile al processo di localizzazione dell'app, è possibile creare una cartella alternativa in cui conservare le stringhe ed i valori specifici per una specifica lingua. Il suffisso da aggiungere è il codice della lingua in cui effettuare la traduzione (ad esempio, se si vuole supportare la lingua francese bisogna creare una cartella *values-fr* ed inserire al suo interno un file contenente le stringhe tradotte). Android studio si occupa di raggruppare e visualizzare file di versioni diverse fra loro all'interno di cartelle immaginarie (come si può vedere nella Figura 4.13). Un ragionamento analogo vale anche nel caso di dimensioni da utilizzare per le diverse risoluzioni degli schermi. All'interno dell'applicazione questa funzionalità è sfruttata anche per la localizzazione dei prodotti, dei loro tipi, delle loro descrizioni e degli ingredienti. Le stringhe sono state memorizzate dentro array separati. Questo è la motivazione dietro all'uso di numeri interi come identificatori per questi elementi all'interno dei database. Il valore dell'ID corrisponde al valore dell'indice, nel corrispondente array, a cui è possibile recuperare la stringa tradotta.

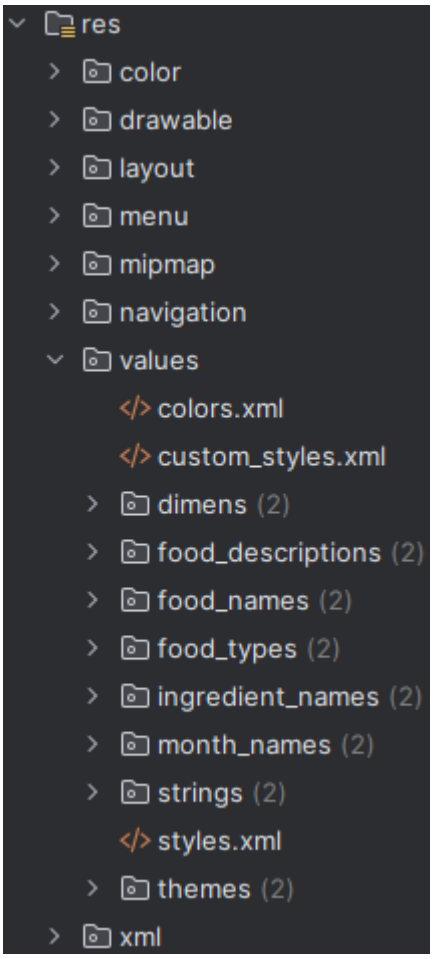

Figura 4.13 - Struttura della cartella "res" dell'applicazione in Android Studio.

4.4 Logica dell'applicazione

In questa sezione viene mostrata e descritta nel suo funzionamento, per i package più importanti e per i file contenuti al loro interno, l'effettiva logica implementata nell'applicazione.

4.4.1 Package generale

Esistono due file che sono presenti nel package generale, il loro posizionamento è dovuto al fatto che non hanno un ambito specifico ma sono utilizzati in più punti del codice.
Il primo è *IngredientQuantity.kt*, al suo interno è definita la classe enumerativa per contenere i tre possibili stati di un ingrediente (Figura 4.14):

- ***INGREDIENT_REMOVED***: L'ingrediente è stato rimosso dal prodotto.
- ***INGREDIENT_NORMAL***: È presente nelle quantità predefinite per quel prodotto specifico.
- ***INGREDIENT_EXTRA***: È presente ma in quantità superiori alla norma.

```
enum class IngredientQuantity {
    INGREDIENT_REMOVED, INGREDIENT_NORMAL, INGREDIENT_EXTRA
}
```

Figura 4.14 - Implementazione della classe enumerativa.

Il secondo file è *CartProduct.kt*, esso implementa la classe per rappresentare i prodotti nel carrello. Nella classe viene sovrascritto il metodo equals() per poter verificare se due istanze sono uguali. Ogni prodotto è costituito da diversi attributi (Figura 4.15):

- ***cartProductId*** (stringa): Contiene l'identificatore del prodotto.
- ***food*** (oggetto): Riferimento all'oggetto rappresentante il prodotto in sé.
- ***extraIngredients*** (vettore): Una lista contenente gli ingredienti in quantità extra.
- ***removedIngredients*** (vettore): Una lista contenente gli ingredienti rimossi.
- ***quantity*** (int): Quantità del prodotto scelta.
- ***price*** (double): Prezzo totale del prodotto.

```
class CartProduct(
    val cartProductId: String,
    val food: Food,
    var extraIngredients: ArrayList<Ingredient>,
    var removedIngredients: ArrayList<Ingredient>,
    var quantity: Int,
    var price: Double
)
```

Figura 4.15 - Attributi della classe.

4.4.2 Package database

In questo package risiedono tutti gli elementi necessari per creare ed interagire con il database relazionale. La classe rappresentante il database, *RestaurantDB*, è contenuta nel file *RestaurantDatabase.kt*, al suo interno sono indicati i metodi per ottenere i DAO (Figura 4.16).

```
@Database(entities = [User::class, Order::class, Ingredient::class, Food::class, FoodIngredient::class,
                      OrderProductEdit::class, FoodType::class, OrderProduct::class, Address::class], version = 1)
abstract class RestaurantDB: RoomDatabase() {
    abstract fun userDao(): UserDao
    abstract fun foodDao(): FoodDao
    abstract fun foodTypeDao(): FoodTypeDao
    abstract fun orderDao(): OrderDao
    abstract fun ingredientDao(): IngredientDao
    abstract fun foodIngredientDao(): FoodIngredientDao
    abstract fun orderProductDao(): OrderProductDao
    abstract fun orderProductEditDao(): OrderProductEditDao
    abstract fun addressDao(): AddressDao
```

Figura 4.16 - Dichiarazione dei DAO necessari per il database.

Ingredients.kt contiene l'entità *Ingredient* ed il corrispondente Dao *IngredientDao*. L'entità, in particolare, presenta l'attributo *name* per contenere il nome dell'ingrediente, che non esiste come colonna nella tabella finale (Figura 4.17). La classe, inoltre, implementa l'interfaccia *Parcelable*, necessaria per poter trasferire le sue istanze come argomenti durante la navigazione.

```
@Entity(tableName = "Ingredients")
data class Ingredient(
    @ColumnInfo(name = "ingredient_id") @PrimaryKey val ingredientId: Int,
    @Ignore var name: String = "",
    @ColumnInfo(name = "unit_price") var unitPrice: Double,
    @ColumnInfo(name = "image_uri") var imageUri: String
): Parcelable
```

Figura 4.17 - Implementazione dell'entità "Ingredient".

Il Dao include sia metodi standard per l'inserimento e la cancellazione, sia i metodi personalizzati *getIngredientById()* ed *exists()*. Questi ultimi permettono, rispettivamente, di ottenere un ingrediente specificando il suo il suo ID e di controllare se la tabella esiste (Figura 4.18).

```
@Dao
interface IngredientDao{
    @Insert(onConflict = OnConflictStrategy.REPLACE) suspend fun insert(vararg ingredients: Ingredient)
    @Delete suspend fun delete(ingredient: Ingredient)
    @Query(value = "Select * from Ingredients where ingredient_id = :id") suspend fun getIngredientById(id: Int): Ingredient?
    @Query(value = "Select case when exists(Select 1 from Ingredients) then 1 else 0 end") suspend fun exists(): Boolean
}
```

Figura 4.18 - Implementazione del DAO con relativi metodi.

Addresses.kt contiene entità e DAO relativi agli indirizzi di spedizione. Nella dichiarazione dell'entità *Address*, dato che la tabella finale contiene una chiave esterna, è necessario implementare la proprietà *foreignKeys*. In questo caso, la chiave esterna *user_id* è legata all'omonima chiave primaria presente nella tabella legata all'entità *User*, ed ogni azione sull'elemento associato a quest'ultima si ripercuote anche nella tabella corrente (Figura 4.19). Sempre nella dichiarazione, viene specificato di indicizzare questo attributo. La classe, inoltre, implementa l'interfaccia *Parcelable*.

```
@Entity(tableName = "Addresses", indices = [Index(value = ["user_id"])], foreignKeys = [
    ForeignKey(User::class, ["user_id"], ["user_id"], ForeignKey.CASCADE, ForeignKey.CASCADE)])
data class Address(
    @PrimaryKey @ColumnInfo(name = "address_id") val addressId: String,
    @ColumnInfo(name = "user_id") val userId: String,
    val address: String,
    val cap: String,
    val city: String,
    val province: String,
    @ColumnInfo(name = "default_address") var defaultAddress: Boolean = false
): Parcelable {
```

Figura 4.19 - Implementazione dell'entità "Address".

All'interno del DAO sono dichiarati, oltre a quelli predefiniti, una serie di metodi personalizzati (Figura 4.20). Uno di questi, in particolare, implementa una transazione:

- ***getAddressesByUserId()***: Dato l'ID di un utente, restituisce una collezione degli indirizzi a lui associati.
- ***exists()***: Prende come parametro l'ID di un utente e verifica la presenza di indirizzi associati.
- ***getDefaultDeliveryAddress()***: Restituisce, se esiste, l'indirizzo di spedizione predefinito dell'utente.
- ***removeDefaultAddress()***: Identifica l'indirizzo predefinito, se esiste, ed imposta a *false* il suo attributo *default_address*.
- ***addDefaultAddress()***: Prende come parametri gli ID di un'utente e di un indirizzo. Una volta individuato l'indirizzo corrispondente, se esiste, imposta a *true* il suo attributo *default_address*.
- ***setAsDefaultDeliveryAddress()***: Transazione che consente il cambio di indirizzo predefinito. Al suo interno vengono eseguiti in sequenza *removeDefaultAddress()* e *addDefaultAddress()*.

```
@Dao
interface AddressDao {
    @Insert(onConflict = OnConflictStrategy.REPLACE) suspend fun insert(vararg addresses: Address)
    @Delete suspend fun delete(address: Address)
    @Query("Select * from Addresses where user_id = :id") suspend fun getAddressesByUserId(id: String): List<Address>
    @Query("Select case when exists(Select address_id from Addresses where user_id = :id) then 1 else 0 end")
    suspend fun exists(id: String): Boolean
    @Query("Select * from Addresses where user_id = :id and default_address = 1") suspend fun getDefaultDeliveryAddress(id: String): Address?
    @Query("Update Addresses set default_address = 0 where user_id = :userId and default_address = 1")
    fun removeDefaultAddress(userId: String)
    @Query("Update Addresses set default_address = 1 where user_id = :userId and address_id = :addressId")
    fun addDefaultAddress(userId: String, addressId: String)
    @Transaction suspend fun setAsDefaultDeliveryAddress(userId: String, addressId: String) {
        removeDefaultAddress(userId)
        addDefaultAddress(userId, addressId)
    }
}
```

Figura 4.20 - Implementazione del DAO di "Address" con relativi metodi.

Il file *FoodTypes.kt* contiene l'entità ed il DAO relativi ai tipi di prodotti. L'entità, in particolare, contiene l'attributo name, il quale non esiste nella tabella finale. Al suo interno viene memorizzato il nome da associare al tipo. Esiste anche un costruttore secondario in cui non è necessario specificare un valore di questo attributo, ma viene inizializzato con una stringa vuota.
Il DAO include due metodi standard per l'inserimento e la cancellazione e due metodi personalizzati: *getAllFoodTypes()* ed *exists()*. Il primo permette di recuperare tutti i tipi memorizzati nella tabella, mentre il secondo serve per verificare l'esistenza della tabella.
Nella figura 4.21 è possibile vedere l'implementazione di entrambe le componenti.

```
@Entity(tableName = "FoodTypes")
data class FoodType(
    @ColumnInfo(name = "type_id") @PrimaryKey val typeId: Int,
    @Ignore var name: String = "",
    @ColumnInfo(name = "image_uri") var imageUri: String = ""
)
{ constructor(typeId: Int, imageUri: String): this(typeId, name: "", imageUri) }

@Dao
interface FoodTypeDao {
    @Insert(onConflict = OnConflictStrategy.REPLACE) suspend fun insert(vararg types: FoodType)
    @Delete suspend fun delete(type: FoodType)
    @Query(value = "Select * from FoodTypes") suspend fun getAllFoodTypes(): List<FoodType>
    @Query(value = "Select case when exists(Select 1 from FoodTypes) then 1 else 0 end") suspend fun exists(): Boolean
}
```

Figura 4.21 - Implementazione di entità e DAO relativi a tipi di prodotti.

Foods.kt implementa le componenti relative ai prodotti ordinabili. Nell'entità *Food* vengono definiti gli attributi specifici *name* e *description*, contenenti rispettivamente nome e breve descrizione del prodotto. Viene definito, in aggiunta, un indice per l'attributo *type* (Figura 4.22). L'entità implementa anche l'interfaccia *Parcelable*.

```
@Entity(tableName = "Foods", indices = [Index(value = ["type"])])
data class Food (
    @ColumnInfo(name = "food_id") @PrimaryKey val foodId: Int,
    @Ignore var name: String = "",
    @Ignore var description: String = "",
    val type: Int,
    @ColumnInfo(name = "unit_price") var unitPrice: Double,
    @ColumnInfo(name = "image_uri") var imageUri: String = ""
) : Parcelable {
```

Figura 4.22 - Implementazione dell'entità "Foods".

Nel DAO sono definiti tre metodi personalizzati (Figura 4.23):

- ***getFoodById()***: Consente di recuperare il prodotto, se esiste, il cui identificatore combacia con quello specificato come parametro.
- ***getFoodsByType()***: Restituisce una lista di prodotti il cui tipo combacia con quello specificato.
- ***exists()***: Verifica se la tabella esiste oppure no.

```kotlin
@Dao
interface FoodDao {
    @Insert(onConflict = OnConflictStrategy.REPLACE) suspend fun insert(vararg foods: Food)
    @Delete suspend fun delete(food: Food)
    @Query(value = "Select * from Foods where food_id = :id") suspend fun getFoodById(id: Int): Food?
    @Query(value = "Select * from Foods where type = :type") suspend fun getFoodsByType(type: Int): List<Food>
    @Query(value = "Select case when exists(Select 1 from Foods) then 1 else 0 end") suspend fun exists(): Boolean
}
```

Figura 4.23 - Implementazione del DAO associato all'entità "Food".

Il file *Users.kt* implementa l'entità ed il DAO associati agli account degli utenti, la Figura 4.24 mostra la loro implementazione. L'entità *User* ha un indice associato all'attributo *email*.
Nel DAO, oltre ai metodi predefiniti per l'aggiunta, la modifica e la cancellazione delle righe, sono presenti sei metodi personalizzati:

- ***getUserById()***: Permette di ottenere l'utente, se esiste, il cui identificatore coincide con quello indicato.
- ***deleteAllUsers()***: Elimina tutti gli utenti presenti nella tabella.
- ***exists()***: Verifica se esiste un utente il cui identificatore coincide con quello passato come parametro.
- ***changeName()***: Cerca l'utente in base all'ID passatogli e modifica il suo nome sostituendolo con quello indicato come parametro.
- ***changeSurname()***: Svolge un lavoro analogo a *changeName()*, ma stavolta agisce sul cognome.
- ***changeEmail()***: Simile a *changeName()*, ma rivolto all'e-mail.

```kotlin
@Entity(tableName = "Users", indices = [Index(value = ["email"])])
data class User(
    @ColumnInfo(name = "user_id") @PrimaryKey val userId: String,
    var name: String,
    var surname: String,
    var email: String,
    @ColumnInfo(name = "accepted_privacy_policy") var acceptedPrivacyPolicy: Boolean
)

@Dao
interface UserDao {
    @Insert(onConflict = OnConflictStrategy.REPLACE) suspend fun insert(vararg users: User)
    @Update suspend fun update(user: User)
    @Delete suspend fun delete(user: User)
    @Query("Select * from Users where user_id = :uid") suspend fun getUserById(uid: String): User?
    @Query("Delete from Users") suspend fun deleteAllUsers()
    @Query("Select case when exists(Select user_id from Users where user_id = :id) then 1 else 0 end")
    suspend fun exists(id: String): Boolean
    @Query("Update Users set name = :newName where user_id = :id") suspend fun changeName(newName: String, id: String)
    @Query("Update Users set name = :newSurname where user_id = :id") suspend fun changeSurname(newSurname: String, id: String)
    @Query("Update Users set name = :newEmail where user_id = :id") suspend fun changeEmail(newEmail: String, id: String)
}
```

Figura 4.24 - Implementazione di entità e DAO.

Orders.kt implementa le componenti dedicate agli ordini degli utenti. Al suo interno è anche presente la costante *MILLISECONDS_PER_MONTH*, che contiene il numero di millisecondi presenti in trenta giorni. L'entità *Order*, in particolare, ha una struttura più complessa rispetto alle altre (Figura 4.25):

- Due attributi sono di tipo *Date*, non supportato da Room. Per questo motivo è necessaria la presenza della classe *DateConverter* contenente i metodi per effettuare le dovute conversioni dal tipo *Date* a *Long* e viceversa (Figura 4.25).
- L'attributo *user_id* della tabella funge da chiave esterna, ed è collegata all'omonima chiave primaria della tabella associata all'entità *User*.
- Vi è un indice definito per la colonna *user_id* della tabella.

```
private val MILLISECONDS_PER_MONTH = 2592000000  //Milliseconds in 30 days
private class DateConverter { // Timestamp = number of milliseconds passed since January 1, 1970, 00:00:00 UTC
    @TypeConverter fun dateFromTimestamp(timestamp: Long): Date = Date(timestamp)
    @TypeConverter fun timestampFromDate(date: Date): Long = date.time
}

@Entity(tableName = "Orders", indices = [Index(value = ["user_id"])],
    foreignKeys = [ForeignKey(User::class, ["user_id"], ["user_id"], ForeignKey.CASCADE, ForeignKey.CASCADE)])
@TypeConverters(DateConverter::class)
data class Order(
    @ColumnInfo(name = "order_id") @PrimaryKey val orderId: String,
    @ColumnInfo(name = "user_id") var userId: String?,
    @ColumnInfo(name = "order_date") val orderDate: Date,
    @ColumnInfo(name = "delivery_date") val deliveryDate: Date,
    @ColumnInfo(name = "total_price") val totalPrice: Double
): Parcelable {
```

Figura 4.25 - Implementazione dell'entità "Order" e di "DateConverter".

Nel DAO sono presenti sia metodi standard sia metodi che non lo sono (Figura 4.26):

- ***getOrderById()***: Restituisce l'ordine, se esiste, il cui ID combacia con quello specificato.
- ***deleteAllOrders()***: Elimina tutti gli ordini memorizzati nella tabella.
- ***deleteLeastRecentOrder()***: Elimina l'ordine più datato memorizzato nella tabella.
- ***getAllOrders()***: Restituisce una collezione contenente tutti gli ordini nella tabella.
- ***exists()***: Verifica se esiste un ordine il cui identificatore combacia con quello indicato.
- ***deleteOlderOrders()***: Elimina gli ordini più datati, ossia quelli effettuati più di trenta giorni prima della data specificata come parametro. Per svolgere l'operazione ricorre all'uso di *MILLISECONDS_PER_MONTH*.

```
@Dao
interface OrderDao {
    @Insert(onConflict = OnConflictStrategy.REPLACE) suspend fun insert(vararg orders: Order)
    @Delete suspend fun delete(order: Order)
    @Query("Select * from Orders where order_id = :id") suspend fun getOrderById(id: String): Order?
    @Query("Delete from Orders") suspend fun deleteAllOrders()
    @Query("Delete from Orders where order_id in (Select order_id from Orders where user_id = :id order by order_date asc limit 1)")
    suspend fun deleteLeastRecentOrder(id: String)
    @Query("Select * from Orders where user_id = :id order by order_date desc") suspend fun getAllOrders(id: String): List<Order>
    @Query("Select case when exists(Select order_id from Orders where user_id = :id) then 1 else 0 end")
    suspend fun exists(id: String): Boolean
    @Query("Delete from Orders where user_id = :userId and order_date < (:currentDate - :millisecondsReference)")
    suspend fun deleteOlderOrders(userId: String ,currentDate: Long = Date().time, millisecondsReference: Long = MILLISECONDS_PER_MONTH)
}
```

Figura 4.26 - Implementazione del DAO associato all'entità "Order".

Il file *OrderProducts.kt* contiene entità e DAO per i prodotti memorizzati negli ordini. Nell'implementazione dell'entità *OrderProduct* si definiscono due indici per gli identificatori dell'ordine e del prodotto. I due attributi, allo stesso tempo, sono indicati come chiavi esterne.
Nel DAO, invece, è presente come metodo non standard solamente *getOrderProductsByOrderId()*. Quest'ultimo consente di recuperare tutti i prodotti il cui ID dell'ordine di appartenenza coincide con quello passato come parametro. La Figura 4.27 mostra l'implementazione di entrambe le componenti.

```
@Entity(tableName = "Order_Products", indices = [Index(value = ["order_id"]), Index(value = ["food_id"])],
    foreignKeys = [ForeignKey(Order::class, ["order_id"], ["order_id"], ForeignKey.CASCADE, ForeignKey.CASCADE),
    ForeignKey(Food::class, ["food_id"], ["food_id"], ForeignKey.SET_NULL, ForeignKey.CASCADE)])
data class OrderProduct(
    @ColumnInfo(name = "order_product_id") @PrimaryKey val orderProductId: String,
    @ColumnInfo(name = "order_id") val orderId: String?,
    @ColumnInfo(name = "food_id") val foodId: Int,
    val quantity: Int,
    val price: Double,
)

@Dao
interface OrderProductDao {
    @Insert(onConflict = OnConflictStrategy.REPLACE) suspend fun insert(vararg cartProduct: OrderProduct)
    @Delete suspend fun delete(cartProduct: OrderProduct)
    @Query(value = "Select * from Order_Products where order_id = :id") suspend fun getOrderProductsByOrderId(id: String): List<OrderProduct>
}
```

Figura 4.27 - Implementazione di entrambe le componenti.

OrderProductEdits.kt contiene l'entità ed il DAO per le modifiche apportate ai prodotti di un ordine. L'entità *OrderProductEdit* contiene due attributi, *orderProductId* ed *ingredientId*, che godono di diverse proprietà:

- Le loro controparti nella tabella sono definite come chiavi primarie, così come indicato dalla proprietà *primaryKeys*.
- Sono allo stesso tempo definite come chiavi esterne, così come indicato da *foreignKeys*. In entrambi i casi una qualsiasi azione svolta sulle chiavi primarie a cui sono legati si ripercuote anche su di loro.
- Vi sono due indici definiti per le corrispondenti colonne nella tabella.

Nel DAO è presente solamente *getOrderProductEditsByProductId()* come metodo personalizzato. Quest'ultimo restituisce le modifiche, se esistono, apportate al prodotto identificato dall'ID passatogli come parametro. La Figura 4.28 mostra l'implementazione di entrambe le componenti.

```
@Entity(tableName = "Order_Product_Edits", indices = [Index(value = ["ingredient_id"]), Index(value = ["order_product_id"])],
    foreignKeys = [ForeignKey(Ingredient::class, ["ingredient_id"], ["ingredient_id"], ForeignKey.CASCADE, ForeignKey.CASCADE),
    ForeignKey(OrderProduct::class, ["order_product_id"], ["order_product_id"], ForeignKey.CASCADE, ForeignKey.CASCADE)],
    primaryKeys = ["order_product_id", "ingredient_id"])
data class OrderProductEdit(
    @ColumnInfo(name = "order_product_id") val orderProductId: String,
    @ColumnInfo(name = "ingredient_id") val ingredientId: Int,
    @ColumnInfo(name = "ingredient_quantity") val ingredientQuantity: IngredientQuantity
)

@Dao
interface OrderProductEditDao {
    @Insert(onConflict = OnConflictStrategy.REPLACE) suspend fun insert(vararg cartProductEdit: OrderProductEdit)
    @Delete suspend fun delete(cartProductEdit: OrderProductEdit)
    @Query(value = "Select * from Order_Product_Edits where order_product_id = :id")
    suspend fun getOrderProductEditsByProductId(id: String): List<OrderProductEdit>
}
```

Figura 4.28 - Implementazione dell'entità "OrderProductEdit" e del relativo DAO.

4.4.3 Package adapter

All'interno di questo package sono presenti tutti gli adapter utilizzati nelle varie liste di elementi che vengono mostrate a schermo durante la navigazione. Il primo di questi, *FoodTypeAdapter*, è presente all'interno dell'omonimo file ed implementa l'adapter utilizzato per la lista dei tipi di prodotto.
Nel costruttore viene passato il riferimento al NavController necessario per la navigazione in app.
I dati da mostrare sono contenuti all'interno dell'attributo *data*, mentre il metodo *setFoodTypeData()* si occupa di inizializzarlo e di mostrare a schermo le informazioni con *submitList()*.
Il metodo *onCreateViewHolder()* è implementato in solo due righe. Si sfrutta il view binding per ottenere l'oggetto contenente tutti i riferimenti agli elementi grafici necessari e lo si passa come parametro al costruttore di *FoodTypeViewHolder*. La sua classe, inoltre, non ha un corpo ed il costruttore chiama il corrispondente del super tipo passandogli l'elemento radice del layout. Questo accade perché, nella maggior parte dei casi, tutti i riferimenti agli elementi grafici sono già presenti nell'oggetto di binding e non è necessario dichiarare dati aggiuntivi.
Nella Figura 4.29 viene mostrata l'implementazione delle componenti appena descritte.

```
class FoodTypeAdapter(private val navController: NavController, private val application: Application):
    ListAdapter<FoodType , FoodTypeAdapter.FoodTypeViewHolder>(ItemDiffCallback()) {

    private var data: List<FoodType> = listOf()

    //Single element of the list
    inner class FoodTypeViewHolder(val binding: ItemRecyclerViewFoodTypeBinding): RecyclerView.ViewHolder(binding.root)

    fun setFoodTypeData(data: List<FoodType>) {
        this.data = data
        submitList(this.data)
    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): FoodTypeViewHolder {
        val binding = ItemRecyclerViewFoodTypeBinding.inflate(LayoutInflater.from(parent.context), parent, attachToParent: false)
        return FoodTypeViewHolder(binding)
    }
```

Figura 4.29 - Implementazione della sorgente dati, del ViewHolder e di "onCreateViewHolder()".

In *onBindViewHolder()*, oltre ad inizializzare con i dati gli elementi grafici, per ciascun elemento viene implementato un Listener (ascoltatore) per intercettare il click dell'utente (Figura 4.30). Il listener, intercettato l'evento, esegue il metodo (detto anche Callback) a lui passatogli. In questo caso il callback fa navigare l'utente al fragment contenente la lista dei prodotti appartenenti al tipo selezionato.

```
override fun onBindViewHolder(holder: FoodTypeViewHolder, position: Int) {
    with(holder) {
        with(data[position]) {
            Glide.with(application).load(imageUri).placeholder(com.projectrestaurant.R.drawable.placeholder)
                .error(com.projectrestaurant.R.drawable.placeholder).into(binding.imageViewFoodType)
            binding.imageViewFoodType.contentDescription = name
            binding.textViewFoodType.text = name
            holder.itemView.setOnClickListener{
                val action = FragmentFoodTypeDirections.actionFragmentFoodTypeToFragmentFoodList(position)
                navController.navigate(action)
            }
        }
    }
```

Figura 4.30 - Implementazione di "onBindViewHolder()".

FoodListAdapter.kt implementa l'adapter usato nella lista dei prodotti dello stesso tipo. Contiene al suo interno due sorgenti per i dati: *fullData* e *currentData*. La prima contiene tutti gli elementi non filtrati mentre la seconda quelli filtrati secondo i termini indicati dall'utente. Tramite *setFoodData()* vengono inizializzati entrambi e la lista viene visualizzata sullo schermo (Figura 4.31).

```
private lateinit var fullData: List<Food>
private lateinit var currentData: List<Food>

fun setFoodData(data: List<Food>) {
    this.fullData = data
    this.currentData = data
    submitList(currentData)
}
```

Figura 4.31 - Inizializzazione della lista.

onBindViewHolder(), oltre ad inizializzare gli elementi di layout, implementa anche un listener per l'elemento in se. Il callback associato naviga l'utente al fragment in cui è possibile personalizzare il prodotto scelto ed inserirlo nel carrello.
Il filtro è implementato in modo asincrono attraverso la classe Filter, in questo modo si evita di rallentare l'applicazione durante il processamento dei dati (Figura 4.32). Per implementare una sua istanza è necessario sovrascrivere due metodi:

- ***performFiltering()***: Effettua il filtraggio vero e proprio. È stato implementato in modo da restituire l'intera sorgente di dati se l'utente non ha inserito alcun termine. Nel caso contrario, invece, è stato implementato in modo da ignorare lo stampatello dei caratteri nel confrontare le stringhe. In entrambi i casi, il risultato è inserito in *currentData* che a sua volta viene inserito in un oggetto di tipo *FilterResults*, il tipo di ritorno del metodo.
- ***publishResults()***: Prende in ingresso l'oggetto restituito dal metodo precedente e lo usa per mostrare a schermo la lista aggiornata.

```
override fun getFilter(): Filter { return searchFilter }

@Suppress( ...names: "UNCHECKED_CAST")
private val searchFilter: Filter = object: Filter() {
    override fun performFiltering(constraint: CharSequence?): FilterResults {
        currentData = if(constraint.isNullOrEmpty()) fullData
        else {
            val tmp = mutableSetOf<Food>()
            tmp.addAll(fullData.filter { it.name.startsWith(constraint.toString(), ignoreCase = true) ||
                    it.description.contains(constraint.toString(), ignoreCase = true) })
            tmp.toList()
        }
        return FilterResults().apply {
            count = currentData.size
            values = currentData
        }
    }
    override fun publishResults(constraint: CharSequence?, results: FilterResults?) { submitList(results!!.values as List<Food>) }
}
```

Figura 4.32 - Implementazione del filtro di ricerca.

Il file *IngredientAdapter.kt* contiene l'omologo adapter utilizzato per la lista degli ingredienti nei fragment dedicati alla personalizzazione dei prodotti. Al suo interno, in particolare, vengono definiti due attributi: *extraIngredients* e *removedIngredients*, che contengono rispettivamente le collezioni degli ingredienti in quantità maggiorate e gli ingredienti rimossi.
L'implementazione di *onBindViewHolder()*, in particolare, è più complessa rispetto agli altri adapter:

1. Inizializzazione delle componenti di layout.

2. Si verifica se l'elemento ingrediente è presente in una delle due collezioni presenti nell'adapter. Nel caso uno dei controlli ha esito positivo, si modifica l'elemento nella lista in modo da mostrare gli elementi grafici opportuni.
3. Si implementa un callback per il pulsante dedicato a ridurne la quantità (Figura 4.33). Nel caso il valore corrente è *INGREDIENT_NORMAL* lo si modifica in *INGREDIENT_REMOVED*, si modifica l'elemento nella lista e lo si inserisce in *removedIngredients* tramite il metodo *addToRemovedIngredients()*. Nel caso di *INGREDIENT_EXTRA* lo si modifica in *INGREDIENT_NORMAL*, si modifica il prezzo totale in modo opportuno attraverso *addToPrice()*, si modifica l'elemento nella lista e lo si rimuove da *extraIngredients*.
4. Si implementa un callback al il pulsante per aumentarne la quantità. L'implementazione è duale rispetto a quella fatta per l'altro pulsante (Figura 4.33). Se il valore corrente è *INGREDIENT_NORMAL* lo si modifica in *INGREDIENT_EXTRA*, si modifica in modo opportuno il prezzo con *addToPrice()* e l'elemento nella lista e si aggiunge l'ingrediente in *extraIngredients* tramite *addToExtraIngredients()*. Se invece il valore è *INGREDIENT_REMOVED* esso viene cambiato in *INGREDIENT_NORMAL*, si modifica l'elemento e si rimuove l'ingrediente da *removedIngredients*.

```
binding.buttonDecrement.setOnClickListener {
    when(viewModel.getIngredientQuantity(data[position].ingredientId)) {
        IngredientQuantity.INGREDIENT_REMOVED -> return@setOnClickListener
        IngredientQuantity.INGREDIENT_NORMAL -> {
            binding.textViewIngredientName.paintFlags = Paint.STRIKE_THRU_TEXT_FLAG
            viewModel.setIngredientQuantity(data[position].ingredientId, IngredientQuantity.INGREDIENT_REMOVED)
            addToRemovedIngredients(data[position])
        }
        IngredientQuantity.INGREDIENT_EXTRA -> {
            binding.textViewIngredientPrice.isVisible = false
            viewModel.setIngredientQuantity(data[position].ingredientId, IngredientQuantity.INGREDIENT_NORMAL)
            viewModel.addToPrice( value: -data[position].unitPrice * viewModel.foodQuantity.value!!)
            extraIngredients.remove(data[position])
        }
    }
}
binding.buttonIncrement.setOnClickListener {
    when(viewModel.getIngredientQuantity(data[position].ingredientId)) {
        IngredientQuantity.INGREDIENT_EXTRA -> return@setOnClickListener
        IngredientQuantity.INGREDIENT_NORMAL -> {
            binding.textViewIngredientPrice.isVisible = true
            viewModel.setIngredientQuantity(data[position].ingredientId, IngredientQuantity.INGREDIENT_EXTRA)
            viewModel.addToPrice( value: data[position].unitPrice * viewModel.foodQuantity.value!!)
            addToExtraIngredients(data[position])
        }
        IngredientQuantity.INGREDIENT_REMOVED -> {
            binding.textViewIngredientName.paintFlags = Paint.ANTI_ALIAS_FLAG
            viewModel.setIngredientQuantity(data[position].ingredientId, IngredientQuantity.INGREDIENT_NORMAL)
            removedIngredients.remove(data[position])
        }
    }
}
```

Figura 4.33 - Implementazione di entrambi i callback.

CartProductAdapter.kt implementa l'adapter utilizzato nella lista dei prodotti nel carrello. Al suo interno, sono presenti *productList* e *foodImages*, che contengono rispettivamente i dati dei prodotti e le loro immagini. È presente anche *stringBuilder*, usato per costruire una stringa in modo graduale. Il metodo *setData()* inizializza le collezioni e mostra a schermo la lista dei prodotti. Il ViewHolder, inoltre, contiene *cartProductId* come attributo aggiuntivo, esso serve per contenere l'identificatore del prodotto associato. La Figura 4.34 mostra l'implementazione delle componenti appena citate.

```
private var stringBuilder: StringBuilder = StringBuilder()
lateinit var productsList: MutableSet<CartProduct>; private set
private lateinit var foodImages: HashMap<Int,String?>

fun setData(data: MutableSet<CartProduct>, images: HashMap<Int,String?>) {
    productsList = data
    foodImages = images
    submitList(productsList.toList())
}

//Single element of the list
class CartProductViewHolder(val binding: ItemRecyclerViewCartProductBinding) : RecyclerView.ViewHolder(binding.root) {
    lateinit var cartProductId: String
}
```

Figura 4.34 - Implementazione di "setData()" e del ViewHolder.

All'interno di *onBindViewHolder()*:

1. Si verifica se il prodotto ha degli ingredienti in quantità maggiorata e, nel caso, si aggiungono i loro nomi alla coda di *stringBuilder*. In entrambi i casi si modifica il layout grafico in modo opportuno, mostrando o nascondendo la casella di testo contenente i nomi degli ingredienti.
2. Si svuota la coda di *stringBuilder* e si ripete un procedimento analogo per gli ingredienti rimossi. Alla fine del lavoro si svuota nuovamente la coda per consentire agli altri elementi nella lista di usare a loro volta lo string builder.
3. Si implementa un listener al pulsante per l'eliminazione del prodotto. Il callback mostra a schermo un messaggio in cui chiede all'utente se vuole confermare la scelta. In caso affermativo provvede ad eliminarlo nel carrello memorizzato in Firebase tramite *deleteProductFromCart()*, lo elimina dalla sorgente dati locale e dalla lista a schermo ed infine controlla se quest'ultima è vuota oppure no. Se il controllo da esito positivo chiude automaticamente il fragment corrente.
4. Si implementa un callback anche al pulsante per la modifica. Il callback inserisce tutti i dati del prodotto in un bundle e lo invia al fragment per la modifica del prodotto, facendo allo stesso tempo navigare l'utente verso quest'ultimo.

È presente anche il metodo *updateCartProduct()*. Quest'ultimo cerca la posizione del prodotto nella sorgente e, se lo trova, provvede aggiornare i dati ed anche l'elemento nella lista in modo da visualizzare i cambiamenti (Figura 4.35).

```
fun updateCartProduct(newVersion: CartProduct) {
    val position: Int = productsList.indexOf(productsList.find { it.cartProductId == newVersion.cartProductId })
    if(position != -1) {
        with(productsList.elementAt(position)) {
            extraIngredients = newVersion.extraIngredients; removedIngredients = newVersion.removedIngredients
            price = newVersion.price; quantity = newVersion.quantity
        }
        notifyItemChanged(position)
    }
```

Figura 4.35 - Implementazione di "updateCartProduct()".

OrderAdapter.kt contiene l'omonimo adapter utilizzato nella lista che mostra gli ordini effettuati in passato. Al suo interno sono definiti diversi attributi (Figura 4.36):

- ***fullOrders***: Contiene tutti gli ordini presenti nel database.
- ***currentOrders***: Contiene i dati degli ordini che sono presenti nella lista a schermo.
- ***fullOrderProducts***: Contiene i dati dei prodotti associati agli ordini.
- ***stringBuilder***: Necessario per costruire stringhe gradualmente.
- ***PRODUCTS_PER_PAGE***: Costante che memorizza il numero massimo di elementi nella lista presenti in una pagina. È inizializzato al valore quindici.
- ***numberOfPages***: Numero di pagine presenti.
- ***currentPage*** e ***_currentPage***: Sono LiveData che memorizzano il numero di pagina corrente.

Il metodo *setData()* inizializza *fullOrders* e *fullOrderProducts* con i dati necessari, calcola il numero di pagine in totale e lo memorizza in *numberOfPages*, inizializza *currentOrders* in base al valore appena calcolato ed infine visualizza a schermo gli ordini contenuti al suo interno (Figura 4.36).
Nel ViewHolder è presente un attributo aggiuntivo che memorizza l'identificatore dell'ordine.

```
private var fullOrders = listOf<Order>()
private var fullOrderProducts = listOf<OrderProduct>()
private var currentOrders = listOf<Order>()
private var stringBuilder = StringBuilder()
private val PRODUCTS_PER_PAGE = 15
var numberOfPages: Int = 1; private set
private val _currentPage = MutableLiveData( value: 1)
val currentPage: LiveData<Int> get() = _currentPage

fun setData(orders: List<Order>, orderProducts: List<OrderProduct>) {
    fullOrders = orders
    fullOrderProducts = orderProducts
    numberOfPages = if(fullOrders.size % PRODUCTS_PER_PAGE != 0) (fullOrders.size / PRODUCTS_PER_PAGE) + 1
                    else fullOrders.size / PRODUCTS_PER_PAGE
    currentOrders = if(numberOfPages > 1) fullOrders.subList(0, PRODUCTS_PER_PAGE) else fullOrders
    submitList(currentOrders)
}

//Single element of the list
inner class OrderViewHolder(val binding: ItemRecyclerViewOrderBinding): RecyclerView.ViewHolder(binding.root) {
    lateinit var orderId: String
}
```

Figura 4.36 - Implementazione di attributi e "setData()".

In *onBindViewHolder()*, in particolare, si raggruppano i prodotti associati ad un ordine e li si conta in modo da mostrare queste informazioni a schermo (Figura 4.37).

1. Si inizia filtrando l'insieme dei prodotti recuperando solo quelli appartenenti all'ordine corrente ed ordinandoli in base al loro identificatore.
2. Si scorre la collezione filtrata raggruppando e contando tutti quelli con lo stesso identificatore, tenendo conto delle quantità indicate in ogni prodotto ed ignorando eventuali modifiche apportate agli ingredienti. Per ogni gruppo si aggiunge una porzione di stringa che contiene nome e quantità a *stringBuilder*. Le singole quantità vengono sommate in modo da ottenere la quantità totale di prodotti.

3. Si costruisce la stringa finale con *stringBuilder* e la si visualizza a schermo inizializzando l'elemento di layout opportuno. Si procede in modo analogo in modo da mostrare a schermo la quantità totale.

Sempre all'interno del metodo si implementa un callback per l'elemento stesso in modo che, se premuto, naviga l'utente al fragment in cui può vedere informazioni aggiuntive sui prodotti dell'ordine (Figura 4.37).

```
override fun onBindViewHolder(holder: OrderViewHolder, position: Int) {
    val calendar: Calendar = Calendar.getInstance()
    holder.orderId = currentOrders[position].orderId
    with(holder.binding) {
        val tmp = fullOrderProducts.filter { it.orderId == currentOrders[position].orderId }.sortedBy { it.foodId }
        if(tmp.isNotEmpty()) {
            var total = 0 ; var i = 0 ; var j = 1; var count = tmp[i].quantity
            while(i < tmp.size) {
                while(j < tmp.size && tmp[j].foodId == tmp[i].foodId) { count += tmp[j].quantity; j++ } // tmp[i]'s foodId is the cu
                stringBuilder.append("$count - ${viewModel.foodNames[tmp[i].foodId]}\n")
                total += count; i = j; j++; count = if(i < tmp.size) tmp[i].quantity else 0
            }
            textViewTotalQuantity.text = "Total quantity: {total.toString()}"
            textViewProductsList.text = stringBuilder.deleteCharAt(stringBuilder.lastIndex)
            stringBuilder = stringBuilder.clear()
        }
        textViewTotalPrice.text = "€ {String.format("%.2f", currentOrders[position].totalP..."
        calendar.timeInMillis = currentOrders[position].orderDate.time
        textViewOrderDate.text = "Order date: {calendar.get(Calendar.DAY_OF_MONTH).toStri..."
        calendar.timeInMillis = currentOrders[position].deliveryDate.time
        textViewOrderDeliveryDate.text = "Delivery date: {calendar.get(Calendar.DAY_OF_MONTH).toS..."
        cardViewOrder.setOnClickListener {
            val action = FragmentUserOrdersDirections.actionFragmentUserOrdersToFragmentUserOrderedProducts(currentOrders[position])
            navController.navigate(action)
        }
    }
```

Figura 4.37 - Implementazione di "onBindViewHolder()".

OrderProductAdapter.kt è l'ultimo file presente in questo package. Esso implementa l'adapter *OrderProductAdapter*, che viene utilizzato nella lista dei prodotti presenti in un determinato ordine effettuato in passato. Gli attributi implementati sono molto simili al caso di *OrderAdapter*, ma ci sono delle eccezioni:

- ***fullOrderProducts***: Contiene tutti i prodotti presenti nell'ordine.
- ***currentOrderProducts***: Contiene solamente gli ordini mostrati a schermo.
- ***fullOrderProductEdits***: Contiene tutte le modifiche (aggiunte e rimozioni di ingredienti) effettuate sui prodotti.

Gli attributi *stringBuilder*, *PRODUCTS_PER_PAGE*, *numberOfPages*, *_currentPage* e *currentPage* hanno gli stessi scopi di quelli visti in *OrderAdapter*. Il metodo *setData()* è di conseguenza implementato in modo analogo alla controparte nell'altro adapter. Anche qui, nel ViewHolder è presente un attributo aggiuntivo, in questo caso memorizza l'identificatore del prodotto.
La Figura 4.38 mostra l'implementazione di tutte le componenti appena descritte.

```
private lateinit var fullOrderProducts: List<OrderProduct>
private lateinit var currentOrderProducts: List<OrderProduct>
private lateinit var fullOrderProductEdits: List<OrderProductEdit>
private lateinit var productImages: HashMap<Int,String?>
private var stringBuilder = StringBuilder()
private val PRODUCTS_PER_PAGE = 15
var numberOfPages: Int = 1; private set
private val _currentPage = MutableLiveData( value: 1)
val currentPage: LiveData<Int> get() = _currentPage

fun setData(orderProducts: List<OrderProduct>, orderProductEdits: List<OrderProductEdit>, images: HashMap<Int,String?>) {
    fullOrderProducts = orderProducts
    fullOrderProductEdits = orderProductEdits
    productImages = images
    numberOfPages = if(fullOrderProducts.size % PRODUCTS_PER_PAGE != 0) (fullOrderProducts.size / PRODUCTS_PER_PAGE) + 1
                    else fullOrderProducts.size / PRODUCTS_PER_PAGE
    currentOrderProducts = if(numberOfPages > 1) fullOrderProducts.subList(0, PRODUCTS_PER_PAGE) else fullOrderProducts
    submitList(currentOrderProducts)
}

//Single element of the list
class OrderProductViewHolder(val binding: ItemRecyclerViewOrderProductBinding): RecyclerView.ViewHolder(binding.root) {
    lateinit var productId: String
}
```

Figura 4.38 - Implementazione di attributi, ViewHolder e del metodo "setData()".

In *onBindViewHolder()* si procede in modo concettualmente simile alla controparte in *OrderAdapter*, ma stavolta incentrato sulle modifiche dei prodotti (Figura 4.39):

1. Si filtrano tutte le modifiche associate al prodotto, se esistono.
2. Se esistono, si dividono in due gruppi: quelle che aggiungono ingredienti e quelle che li rimuovono. Se invece non esistono si modifica l'elemento a schermo per rimuovere le caselle di testo contenente la lista delle modifiche.
3. Per il primo gruppo, se esiste, si costruisce una stringa usando i loro nomi. La stringa finale viene poi mostrata a schermo. Se invece non esiste si modifica l'elemento a schermo per nascondere la casella di testo contenente il testo. Si ripetono i passaggi in modo analogo anche per il secondo gruppo.

```
val editList = fullOrderProductEdits.filter{ it.orderProductId == currentOrderProducts[position].orderProductId }
if(editList.isNotEmpty()) {
    val tmpList = mutableListOf<OrderProductEdit>()
    tmpList.addAll(editList.filter { it.ingredientQuantity == IngredientQuantity.INGREDIENT_EXTRA })
    if(tmpList.isEmpty()) textViewExtraIngredients.visibility = View.GONE
    else {
        for(edit in tmpList) stringBuilder.append("${"Additional"} ${viewModel.ingredientNames[edit.ingredientId]}\n")
        textViewExtraIngredients.text = stringBuilder.deleteCharAt(stringBuilder.lastIndex)
    }
    tmpList.clear(); stringBuilder = stringBuilder.clear()
    tmpList.addAll(editList.filter { it.ingredientQuantity == IngredientQuantity.INGREDIENT_REMOVED })
    if(tmpList.isEmpty()) textViewRemovedIngredients.visibility = View.GONE
    else {
        for(edit in tmpList) stringBuilder.append("${viewModel.ingredientNames[edit.ingredientId]}\n")
        textViewRemovedIngredients.text = stringBuilder.delete(stringBuilder.lastIndex, stringBuilder.lastIndex)
        textViewRemovedIngredients.paintFlags = Paint.STRIKE_THRU_TEXT_FLAG
    }
    stringBuilder = stringBuilder.clear()
} else { textViewExtraIngredients.visibility = View.GONE; textViewRemovedIngredients.visibility = View.GONE }
```

Figura 4.39 - Implementazione della procedura.

4.4.4 Package ui/order

In questo package risiede il file *ActivityOrder.kt*, che implementa l'activity iniziale, ossia quella dedicata all'ordinazione dei prodotti.
In *onCreate()* (Figura 4.40) si fa uso delle shared preferences per recuperare un numero intero necessario per impostare correttamente il tema da usare nell'applicazione. Una volta caricato in memoria, infatti, se ne controlla il valore e si imposta il tema in modo opportuno. Di default, il tema si adatta a quello usato dal dispositivo su cui l'app è installata, ma è possibile comunque forzarne l'uso di uno chiaro o di uno scuro.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    foodOrderViewModel = FoodOrderViewModel(application)
    accountViewModel = AccountViewModel(application)
    FirebaseApp.initializeApp(application)
    sharedPrefs = getSharedPreferences("preferences", Context.MODE_PRIVATE)
    val themeMode = sharedPrefs.getInt("theme", AppCompatDelegate.MODE_NIGHT_FOLLOW_SYSTEM)
    when (themeMode) {
        AppCompatDelegate.MODE_NIGHT_FOLLOW_SYSTEM -> AppCompatDelegate.setDefaultNightMode(AppCompatDelegate.MODE_NIGHT_FOLLOW_SYSTEM)
        AppCompatDelegate.MODE_NIGHT_YES -> AppCompatDelegate.setDefaultNightMode(AppCompatDelegate.MODE_NIGHT_YES)
        AppCompatDelegate.MODE_NIGHT_NO -> AppCompatDelegate.setDefaultNightMode(AppCompatDelegate.MODE_NIGHT_NO)
    }
    binding = ActivityOrderBinding.inflate(layoutInflater)
    setContentView(binding.root)
    setSupportActionBar(binding.appBarMain.toolbarOrder)
    navController = findNavController(com.projectrestaurant.R.id.nav_host_fragment_order)
    appBarConfiguration = AppBarConfiguration(navController.graph, binding.drawerLayout)
    setupActionBarWithNavController(navController, appBarConfiguration)
    binding.navView.setupWithNavController(navController)
    binding.lifecycleOwner = this
}
```

Figura 4.40 - Metodo eseguito alla creazione dell'activity iniziale.

Successivamente alla creazione, si controlla se l'utente è al momento autenticato nel sistema controllando se la variabile *currentUser*, già fornita dai servizi Firebase, contiene un valore nullo oppure no. In base all'esito si modifica il drawer menu in maniera opportuna. (Figura 4.41)

```
override fun onStart() {
    super.onStart()
    if(auth.currentUser != null) {
        Log.i( tag: "FirebaseAuth",  msg: "${ActivityOrder::class.java.name} - Authentication state changed to ${auth.currentUser?.uid}")
        binding.navView.menu.findItem(com.projectrestaurant.R.id.nav_login_register).isVisible = false
        binding.navView.menu.findItem(com.projectrestaurant.R.id.nav_account).isVisible = true
    } else {
        Log.i( tag: "FirebaseAuth",  msg: "${ActivityOrder::class.java.name} - Authentication state changed to null")
        binding.navView.menu.findItem(com.projectrestaurant.R.id.nav_login_register).isVisible = true
        binding.navView.menu.findItem(com.projectrestaurant.R.id.nav_account).isVisible = false
    }
}
```

Figura 4.41 - Controllo per verificare se l'utente è autenticato.

Il file *FragmentFoodType.kt* implementa il fragment dedicato a mostrare la schermata iniziale dell'applicazione. Nel metodo eseguito alla sua creazione, viene inizializzato un oggetto di tipo *FoodTypeAdapter*, necessario per mostrare a schermo la lista dei tipi di prodotto (Figura 4.42).

```
override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?): View {
    super.onCreateView(inflater, container, savedInstanceState)
    binding = FragmentFoodTypeBinding.inflate(inflater, container, attachToRoot: false)
    binding.recyclerViewFoodType.layoutManager = LinearLayoutManager(requireActivity(), LinearLayoutManager.VERTICAL, reverseLayout: false)
    navController = findNavController()
    adapter = FoodTypeAdapter(navController, requireActivity().application)
    return binding.root
}
```

Figura 4.42 - Metodo eseguito all'avvio del fragment.

Successivamente alla creazione (Figura 4.43), si controlla se il dispositivo è connesso ad internet con il metodo *isOnline()*. Se il controllo da esito positivo:

1. Si recuperano i dati relativi ai tipi di prodotti grazie a *getFoodTypes()* e li si memorizza nella lista *foodTypeList*.
2. Si esegue *isLoggedIn()* per verificare se l'utente è autenticato. Se la risposta è sì, viene mostrata a schermo anche la barra di navigazione inferiore, e si imposta a ciascun elemento che ne fa parte quale activity o fragment aprire in caso di selezione. Nel mentre si controlla se il carrello è vuoto con *isShoppingCartEmpty()* e nel caso si rimuove la relativa opzione nella barra.
3. Si passa *foodTypeList* all'adapter con il metodo *setFoodTypeData()*.

Come ultima cosa si passa l'adapter, ormai fornito di tutti gli attributi necessari, al recycler view.

```
if(viewModel.isOnline(requireActivity().application)) {
    viewLifecycleOwner.lifecycleScope.launch(Dispatchers.IO) {
        val foodTypeList = viewModel.getFoodTypes()
        withContext(Dispatchers.Main) {
            if(viewModel.isLoggedIn) {
                binding.bottomNavbar.visibility = View.VISIBLE
                binding.bottomNavbar.setOnItemSelectedListener { menuItem ->
                    when(menuItem.itemId) {
                        com.projectrestaurant.R.id.nav_orders -> {
                            startActivity(Intent(requireActivity(), ActivityUserOrders::class.java))
                            return@setOnItemSelectedListener true
                        }
                        com.projectrestaurant.R.id.nav_shopping_cart -> {
                            navController.navigate(com.projectrestaurant.R.id.action_fragment_food_type_to_fragment_shopping_cart)
                            return@setOnItemSelectedListener true
                        }
                        else -> return@setOnItemSelectedListener false
                    }
                }
                if(!(viewModel.isShoppingCartEmpty()))
                    binding.bottomNavbar.menu.findItem(com.projectrestaurant.R.id.nav_shopping_cart).isVisible = true
            } else binding.bottomNavbar.visibility = View.GONE
            adapter.setFoodTypeData(foodTypeList)
```

Figura 4.43 - Codice per mostrare elementi a schermo se il dispositivo è connesso ad internet.

FragmentFoodList.kt implementa il fragment in cui viene mostrata la lista dei prodotti appartenenti a un determinato tipo. Nel metodo *onCreateView()* si eseguono tutte le operazioni classiche per inizializzare un fragment, ovvero inflating del layout grafico, aggiunta di un layout manager al recycler view e recupero del riferimento per il *navController* associato.
In *onViewCreated()* si provvede a mostrare la lista a schermo e aggiornare opportunamente altri elementi dell'interfaccia (Figura 4.44), la procedura è divisa in diverse fasi:

1. Creazione di un ListAdapter.
2. Aggiunta nella barra superiore dell'opzione per cercare i prodotti in base a determinate parole. Per farlo, dato che la barra appartiene all'activity e non al fragment, è necessario aggiungere un particolare oggetto chiamato MenuProvider all'activity tramite il metodo *addMenuProvider()*. Nel metodo va specificato il provider, ovvero il componente che vuole aggiungere qualcosa nella barra, e lo stato in cui si deve trovare quest'ultimo per effettuare l'operazione in sé.
3. Si recuperano le informazioni sui prodotti da mostrare, per farlo si invoca *getFoodList()* indicando come parametro il tipo a cui appartengono.
4. Si controlla se l'utente è già autenticato e se ci sono già elementi nel carrello rispettivamente con *isLoggedIn()* e la negazione del risultato di *isShoppingCartEmpty()*. Se il controllo da esito positivo viene mostrato a schermo anche un pulsante per accedere direttamente al carrello.
5. Si passa la sorgente dati all'adapter con *setFoodData()* e si passa quest'ultimo al recycler view.
6. Si aggiunge un listener per la casella di testo in cui effettuare la ricerca per intercettare cambiamenti del testo inserito. Il listener, intercettato l'evento, esegue il callback per filtrare la lista in base al contenuto.
7. Si aggiunge un listener anche al pulsante del carrello per gestire l'interazione con l'utente.

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)
    adapter = FoodListAdapter(navController, requireActivity().application)
    (requireActivity() as MenuHost).addMenuProvider( provider: this, viewLifecycleOwner, Lifecycle.State.STARTED)
    with(binding) {
        progressBar.isIndeterminate = true
        constraintLayout.overlay.add(progressBar)
        progressBar.visibility = View.VISIBLE
        viewLifecycleOwner.lifecycleScope.launch(Dispatchers.IO) {
            val foodList = viewModel.getFoodList(args.foodType)
            withContext(Dispatchers.Main) {
                progressBar.visibility = View.GONE
                progressBar.isIndeterminate = false
                constraintLayout.overlay.remove(progressBar)
                if(viewModel.isLoggedIn && !(viewModel.isShoppingCartEmpty())) buttonShoppingCart.visibility = View.VISIBL
                adapter.setFoodData(foodList)
                recyclerViewFoodList.adapter = adapter
                editTextSearch.addTextChangedListener { adapter.filter.filter(it) }
                buttonShoppingCart.setOnClickListener {
                    viewModel.resetPrice()
                    navController.navigate(com.projectrestaurant.R.id.action_fragment_food_list_to_fragment_shopping_cart)
                }
            }
        } }
}
```

Figura 4.44 - Implementazione del metodo "onViewCreated()".

FragmentFoodList.kt implementa il fragment in cui è possibile scegliere la quantità e personalizzare il prodotto desiderato attraverso una lista di ingredienti. Il prodotto selezionato viene ricevuto tramite i SafeArgs. In *onCreateView()*, oltre alle operazioni classiche, si provvede ad aggiungere delle decorazioni agli elementi della lista e a collegare l'oggeto *viewModel* a quella definita nel layout grafico tramite il data binding (Figura 4.45).

```
override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?): View {
    super.onCreateView(inflater, container, savedInstanceState)
    binding = FragmentFoodIngredientsBinding.inflate(inflater, container, attachToRoot: false)
    binding.recyclerViewFoodIngredients.layoutManager = LinearLayoutManager(requireActivity(), LinearLayoutManager.VERTICAL, reverseLayout: false)
    binding.recyclerViewFoodIngredients.addItemDecoration(DividerItemDecoration(requireContext(), LinearLayoutManager.VERTICAL))
    binding.viewModel = viewModel
    binding.lifecycleOwner = this
    navController = findNavController()
    return binding.root
}
```

Figura 4.45 – Creazione dell'interfaccia del fragment e preparazione del NavController.

In onViewCreated(), invece, si procede a mostrare a schermo la lista degli ingredienti e rendere interattivi elementi dell'interfaccia:

1. Si aggiungono dei callback per i pulsanti per modificare la quantità da ordinare (Figura 4.46). Le due procedure provvedono a modificare quantità, per poi aggiornare il prezzo totale tenendo conto dei prezzi per gli ingredienti da aggiungere in quantità superiori alla norma. Si crea una variabile di appoggio per memorizzare il prezzo da aggiungere o sottrarre, e la si inizializza con il prezzo unitario del prodotto. Si procede, con *getQuantities()*, a recuperare una struttura dati che mostra per ogni ingrediente la quantità al momento selezionata, selezionando poi solo quelli con valore pari a *INGREDIENT_EXTRA*. Per ciascuno di essi si aggiungere il loro prezzo al valore della variabile. Come ultimo si aggiorna il prezzo totale con *addToPrice()* usando il valore finale memorizzato nella variabile.

```
binding.buttonIncrement.setOnClickListener {
    viewModel.incrementFoodQuantity()
    var toAdd: Double = args.food.unitPrice
    for(element in viewModel.getQuantities())
        if(element.value == IngredientQuantity.INGREDIENT_EXTRA)
            toAdd += adapter.data.find { it.ingredientId == element.key }!!.unitPrice
    viewModel.addToPrice(toAdd)
}
binding.buttonDecrement.setOnClickListener {
    if(viewModel.foodQuantity.value!! > 1) {
        viewModel.decrementFoodQuantity()
        var toRemove: Double = args.food.unitPrice
        for(element in viewModel.getQuantities())
            if(element.value == IngredientQuantity.INGREDIENT_EXTRA)
                toRemove += adapter.data.find { it.ingredientId == element.key }!!.unitPrice
        viewModel.addToPrice(-toRemove)
    }
}
```

Figura 4.46 - Implementazione dei callback per i due pulsanti.

2. Si inizializza il prezzo totale con il prezzo unitario del prodotto.
3. Si inizializzano gli elementi che mostrano nome, descrizione ed immagine del prodotto.
4. Si aggiunge un listener al pulsante per aggiungere il prodotto al carrello (Figura 4.47). Il callback controlla se l'utente è autenticato e provvede ad aggiungere il prodotto nel carrello se la risposta è positiva. L'aggiunta avviene con *addProductToShoppingCart()* ed include sia la quantità che le modifiche apportate. Se l'aggiunta va a buon fine l'applicazione fa tornare l'utente alla schermata precedente tramite *navigateUp()*, fornito dal NavController. Se uno dei due controlli da esito negativo viene mostrato a schermo un dialog contenente un opportuno messaggio di errore per informare l'utente dell'impossibilità di procedere. Il dialog viene creato sfruttando la classe predefinita *AlertDialog*.

```
binding.cardViewShoppingCart.setOnClickListener {
    it.isClickable = false
    binding.progressBar.isIndeterminate = true
    binding.constraintLayout.overlay.add(binding.progressBar)
    binding.progressBar.visibility = View.VISIBLE
    if(viewModel.isLoggedIn) {
        viewLifecycleOwner.lifecycleScope.launch(Dispatchers.Main) {
            val result = withContext(Dispatchers.IO) {
                viewModel.addProductToShoppingCart(args.food, adapter.getExtraIngredients(), adapter.getRemovedIngredients())
            }
            if(result) navController.navigateUp()
            else {
                AlertDialog.Builder(requireContext()).setTitle(com.projectrestaurant.R.string.shopping_cart_error_title)
                    .setMessage(com.projectrestaurant.R.string.shopping_cart_add_product_error_message)
                    .setNeutralButton(com.projectrestaurant.R.string.ok) {
                            _, _ -> navController.navigateUp() }.show()
            }
        }
    } else {
        AlertDialog.Builder(requireContext()).setTitle(com.projectrestaurant.R.string.shopping_cart_error_title)
            .setMessage(com.projectrestaurant.R.string.shopping_cart_add_product_error_message_no_auth)
            .setNeutralButton(com.projectrestaurant.R.string.ok) { _, _ -> navController.navigateUp() }.show()
    }

}
```

Figura 4.47 - Implementazione del callback per il pulsante del carrello.

Nel fragment è implementato anche onDestroy() per gestire il suo processo di distruzione (Figura 4.48). In questo caso si procede a reimpostare la quantità dei prodotti ad uno e ad azzerare il prezzo totale, rispettivamente con i metodi *setFoodQuantity()* e *resetPrice()*.

.

```
override fun onDestroy() {
    super.onDestroy()
    viewModel.setFoodQuantity(1)
    viewModel.resetPrice()
}
```

Figura 4.48 - Implementazione di "onDestroy()".

FragmentShoppingCart.kt implementa il fragment in cui si può visualizzare il carrello con la lista dei prodotti al suo interno e procedere al completamento dell'ordine. In *onCreateView()*, in particolare, viene impostato un listener in grado di intercettare determinati dati inviati da altri fragment (Figura 4.49). L'implementazione avviene tramite *setFragmentResultListener()* e specificando una requestKey, ossia un'etichetta con cui identificare il tipo di risultato da ricevere. I dati sono ricevuti sottoforma di un bundle. In questo caso il callback gestisce l'aggiornamento di un prodotto nella lista a causa di una modifica apportata dell'utente. L'aggiornamento avviene con il metodo *updateCartProduct()*, passandogli come parametri le informazioni aggiornate.

```
setFragmentResultListener( requestKey: "modifiedCartProduct") { _, bundle ->
    adapter.updateCartProduct(CartProduct(bundle.getString( key: "cartProductId")!!,
        bundle.getParcelable( key: "food")!!, bundle.getParcelableArrayList( key: "extraIngredients")!!,
        bundle.getParcelableArrayList( key: "removedIngredients")!!,
        bundle.getInt( key: "quantity"), bundle.getDouble( key: "price")))
}
```

Figura 4.49 - Implementazione del listener per intercettare il prodotto modificato.

In *onViewCreated()* si provvede a recuperare e mostrare all'utente le informazioni sui prodotti nel carrello e a rendere interattiva l'interfaccia:

1. Si controlla se l'utente è autenticato e, se il controllo fallisce, si provvede a chiudere il fragment e tornare a quello precedente. Se invece il controllo fallisce si recuperano le informazioni sull'indirizzo di spedizione predefinito e sui prodotti nel carrello, rispettivamente con *getDefaultDeliveryAddress()* e *getCartProducts()*.
2. Si effettuano controlli sui dati appena recuperati: se non ci sono prodotti nel carrello si chiude il fragment, mentre se non esiste un indirizzo di spedizione predefinito si rimuove la relativa opzione di consegna.
3. Si crea l'adapter, gli si assegnano i prodotti da visualizzare con *setData()* e lo si assegna al recycler view del fragment. Si provvede successivamente a recuperare una lista di possibili giorni di consegna con *getDeliveryDays()* per poi mostrarli a schermo con *addDays()*. Nella visualizzazione a schermo i giorni vengono aggiunti ad una sezione implementata in modo tale è possibile selezionare solo un elemento alla volta.
4. Si implementa un listener alla sezione contenente la lista dei giorni per intercettare cambiamenti nella selezione tramite il metodo *setOnCheckedStateChangeListener()* (Figura 4.50). Il callback prende come parametro d'ingresso una lista degli elementi selezionati. Si controlla se la sezione non contiene elementi, se il controllo da esito positivo si rimuovono tutti gli elementi nella sezione delle ore. Se la lista degli elementi attivi non è vuota si provvede ad aggiornare la data di consegna finale con il giorno selezionato e a recuperare una lista delle ore di consegna ammissibili per quel giorno con il metodo *getHours()*, e con *addHours()* le si visualizza a schermo nelle modalità del tutto analoghe a quelle dei giorni.
5. Si implementa un listener alla sezione delle ore. Il callback controlla la presenza di un elemento selezionato e nel caso provvede ad aggiornare la data di consegna finale (Figura 4.50).
6. L'ultimo listener implementato è quello per il pulsante per effettuare l'ordine. Il callback controlla se la data è stata impostata correttamente e, se il controllo ha esito positivo, anche se il dispositivo è connesso ad internet. Se anche quest'ultimo termina con successo procede con la creazione effettiva dell'ordine nel database centrale dell'app con *createOrder()*. In caso uno dei controlli elencati fallisce si provvede a mostrare a schermo un dialog con l'opportuno messaggio di errore.

```
chipGroupDays.setOnCheckedStateChangeListener { chipGroup, checkedIds ->
    chipGroup.isEnabled = false
    if(chipGroupHours.childCount != 0) { chipGroupHours.removeAllViews() }
    if(checkedIds.isEmpty()) { chipGroup.isEnabled = true }
    else {
        val calendar = Calendar.getInstance()
        calendar.timeInMillis = dayList[checkedIds[0]]
        hourList = foodOrderViewModel.getHours(calendar)
        addHours(hourList, chipGroupHours)
        chipGroup.isEnabled = true
    }
}
chipGroupHours.setOnCheckedStateChangeListener { chipGroup, checkedIds ->
    chipGroup.isEnabled = false
    if(checkedIds.isNotEmpty()) { finalCalendar.timeInMillis = hourList[checkedIds[0]] }
    chipGroup.isEnabled = true
}
```

Figura 4.50 - Implementazione dei callback per le due sezioni della data di consegna.

FragmentEditCartProduct.kt implementa il fragment in cui è possibile modificare un prodotto già presente nel carrello. L'implementazione è molto simile a quella vista per il fragment della personalizzazione del prodotto, dato che anche le due interfacce sono quasi identiche. La differenza sostanziale sta nel callback del pulsante per tornare al carrello (Figura 4.51):

1. Si provvede ad aggiungere nel bundle i dati aggiornati del prodotto.
2. Si aggiorna il prodotto nel carrello memorizzato nel database centrale con updateCartProduct(), passandogli come parametri i nuovi dati.
3. Se l'aggiornamento è andato a buon fine, si passa il bundle con i dati aggiornati al fragment richiedente con il metodo *setFragmentResult()*, passando come parametro anche la *requestKey* per identificare il risultato da intercettare. Fatto questo, il fragment si chiude e si torna a quello precedente. Se invece l'aggiornamento non è stato completato con successo si mostra a schermo un dialog con il messaggio di errore opportuno.

```
args.bundle.putParcelableArrayList("extraIngredients", adapter.getExtraIngredients())
args.bundle.putParcelableArrayList("removedIngredients", adapter.getRemovedIngredients())
args.bundle.putInt("quantity", viewModel.foodQuantity.value!!)
args.bundle.putDouble("price", viewModel.totalPrice.value!!)
viewLifecycleOwner.lifecycleScope.launch(Dispatchers.Main) {
    val result = withContext(Dispatchers.IO){
        viewModel.updateCartProduct(CartProduct(
            args.bundle.getString( key: "cartProductId")!!, args.bundle.getParcelable( key: "food")!!,
            args.bundle.getParcelableArrayList( key: "extraIngredients")!!,
            args.bundle.getParcelableArrayList( key: "removedIngredients")!!,
            args.bundle.getInt( key: "quantity"), args.bundle.getDouble( key: "price")))
    }
    if(result) { setFragmentResult( requestKey: "modifiedCartProduct", args.bundle); navController.navigateUp() }
```

Figura 4.51 - Sezione più importante del listener del pulsante per salvare le modifiche.

FragmentFoodNote.kt contiene l'ultima classe presente nel package, ed implementa il fragment per impostare una nota personalizzata da aggiungere all'ordine. In *onViewCreated()* si procede a visualizzare a schermo l'eventuale nota già salvata e a gestire il salvataggio di una sua nuova versione nel database centrale (Figura 4.52).

- Si recupera l'eventuale nota con *getOrderNote()* e la si inserisce nella casella di testo modificabile con *SpannableStringBuilder()*.
- Si implementa un callback per il pulsante di salvataggio. Si salva la nota aggiornata con *setOrderNote()*, se il salvataggio è andato a buon fine si ritorna al fragment precedente. In caso contrario si mostra a schermo un dialog con un messaggio di errore.

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)
    viewLifecycleOwner.lifecycleScope.launch(Dispatchers.Main) {
        val note = withContext(Dispatchers.IO) { viewModel.getOrderNote() }
        binding.editText1.text = SpannableStringBuilder(note)
    }
    binding.buttonAddNote.setOnClickListener {
        it.isClickable = false
        viewLifecycleOwner.lifecycleScope.launch(Dispatchers.Main) {
            it.isClickable = false
            binding.progressBar.isIndeterminate = true
            binding.constraintLayout.overlay.add(binding.progressBar)
            binding.progressBar.visibility = View.VISIBLE
            val result = withContext(Dispatchers.IO) { viewModel.setOrderNote(binding.editText1.text.toString()) }
            if(result) navController.navigateUp()
            else {
                AlertDialog.Builder(requireContext()).setTitle("Something went wrong")
                    .setMessage("Error while adding a note in your order, please try again")
                    .setNeutralButton("OK") { _, _ -> navController.navigateUp() }.show()
            }
        }
    }
}
```

Figura 4.52 - Implementazione di "onViewCreated()" del fragment.

4.4.5 Package ui/loginregister

In questo package è presente il file *ActivityLoginRegister.kt*, che implementa l'activity che consente all'utente di registrarsi, autenticarsi e recuperare la password in caso di smarrimento. Nell'*onCreate()* avvengono le classiche operazioni descritte in precedenza, ossia inflating del layout grafico ed inizializzazione del NavController per effettuare la navigazione (Figura 4.53).

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    binding = ActivityLoginRegisterBinding.inflate(layoutInflater)
    setContentView(binding.root); setSupportActionBar(binding.toolbarLoginRegister)
    binding.lifecycleOwner = this
    navController = findNavController(com.projectrestaurant.R.id.nav_host_fragment_login_register)
    setupActionBarWithNavController(navController, AppBarConfiguration(navController.graph))
    supportActionBar?.setDisplayHomeAsUpEnabled(true)
}
```

Figura 4.53 - Implementazione del metodo per la creazione dell'activity.

FragmentLogin.kt implementa la classe associata al fragment responsabile dell'autenticazione dell'utente. Nell'implementazione di *onViewCreated()* è particolarmente importante il callback per il pulsante di autenticazione:

1. Si valida l'e-mail inserita dall'utente con il metodo *validateEmail()*, la validazione consiste nel controllare se la stringa inserita non è vuota e se è conforme con la sintattica delle e-mail (Figura 4.54). Il metodo restituisce un map contenente il risultato per ciascuno dei test. Se almeno uno di loro ha esito negativo si mostra un messaggio di errore e si annulla l'intera operazione.

```
var checks = viewModel.validateEmail(binding.editTextEmail.text.toString())
if (checks.containsValue(false)) {
    binding.emailContainer.isErrorEnabled = true
    if (checks["isBlank"]!!) {
        binding.emailContainer.error = "The e-mail field is blank"
        binding.buttonLogin.isClickable = true
        return@setOnClickListener }
    if (!checks["isValid"]!!) {
        binding.emailContainer.error = "Invalid e-mail"
        binding.buttonLogin.isClickable = true
        return@setOnClickListener }
}
```

Figura 4.54 - Controlli effettuati sull'e-mail.

2. Si procede in modo analogo con la verifica della password inserita con *validatePassword()*. I test presenti nella validazione servono a verificare se la password inserita: non è vuota, è composta da almeno otto caratteri, ha almeno due minuscole, due maiuscole, due cifre e un carattere speciale.
3. Se entrambe le valutazioni danno esito positivo si effettua l'autenticazione con *login()*, passandogli come parametri i dati inseriti dall'utente. Se la procedura ha esito positivo si distrugge l'activity corrente e si torna a quella iniziale con *finish()*. In entrambi i casi viene mostrato a schermo un messaggio per informare l'utente (Figura 4.55).

```
val result = withContext(Dispatchers.IO) {
    viewModel.logIn(binding.editTextEmail.text.toString(), binding.editTextPassword.text.toString()) }
if (result) {
    Toast.makeText(requireActivity(), "Successfully logged in", Toast.LENGTH_LONG).show()
    startActivity(Intent(requireActivity(), ActivityOrder::class.java))
    activity?.finish()
}
else {
    binding.progressBarLogin.clearAnimation()
    binding.progressBarLogin.visibility = View.GONE
    AlertDialog.Builder(requireContext()).setTitle("Unable to log in")
        .setMessage("Check if your e-mail and password are correct and try a...")
        .setPositiveButton("OK") { _, _ -> }.show()
    binding.buttonLogin.isClickable = true }
```

Figura 4.55 - Implementazione del login e relativi controlli sul risultato.

FragmentRegister.kt implementa il fragment in cui l'utente può registrare un nuovo account. Per registrarsi è necessario inserire nome, cognome, e-mail e due volte la password e accettare la polizza della privacy del ristorante. Nell'implementazione di *onViewCreated()* è importante il callback per il pulsante di registrazione:

1. Si validano le informazioni inserite. Per nome e cognome si controlla che non siano vuoti, per l'e-mail la procedura è invariata rispetto al caso del login, mentre per la password si ha un controllo aggiuntivo per verificare se i due valori inseriti combaciano. Superati tutti questi si verifica se l'utente ha accettato la polizza della privacy agendo sul valore della spunta.
2. Se tutti i controlli hanno avuto esito positivo, si esegue *register()* per registrare l'account all'interno di Firebase ed aggiungere i suoi dati al database centrale (Figura 4.56).
3. Se la registrazione va a buon fine si distrugge l'activity corrente in modo analogo al caso del login, altrimenti si mostra a schermo un messaggio di errore.

```
viewLifecycleOwner.lifecycleScope.launch(Dispatchers.Main) {
    binding.progressBarRegister.visibility = View.VISIBLE
    binding.progressBarRegister.animate()
    val result = withContext(Dispatchers.IO) { viewModel.register(binding.editTextName.text.toString(),
        binding.editTextSurname.text.toString(), binding.editTextEmail.text.toString(),
        binding.editTextPassword.text.toString(), binding.checkBoxPrivacyPolicy.isChecked) }
    if(result) {
        Toast.makeText(requireContext(), "Successfully registered", Toast.LENGTH_LONG).show()
        startActivity(Intent(requireActivity(), ActivityOrder::class.java))
        activity?.finish()
    }
    else {
        binding.progressBarRegister.clearAnimation()
        binding.progressBarRegister.visibility = View.GONE
        AlertDialog.Builder(requireContext()).setTitle("Unable to register")
            .setMessage("Please try again later").setPositiveButton("OK") { _, _ -> }.show()
        it.isClickable = true }
}
```

Figura 4.56 - Implementazione della registrazione e relativi controlli sul risultato.

FragmentNewPassword.kt è l'ultimo file presente in questo package. Implementa al suo interno il fragment che consente all'utente di recuperare la password in caso di smarrimento. Per farlo, l'utente deve inserire il proprio indirizzo e-mail e seguire le istruzioni contenute in un'e-mail inviata all'indirizzo specificato. L'e-mail inserita dall'utente è sottoposta alle stesse validazioni già viste negli altri due fragment. In caso di test falliti, viene mostrato a schermo un messaggio di errore. Una volta superati tutti i test, si esegue *resetPassword()* per consentire a Firebase di avviare la procedura di recupero. In base al risultato viene mostrato a schermo un messaggio che conferma l'invio dell'e-mail oppure un messaggio di errore.

4.4.6 Package ui/account

Nel package, il file ActivityAccount.kt implementa l'activity per consentire all'utente di gestire il suo profilo ed effettuare il logout. Nell'*onCreate()* avvengono tutte le operazioni di routine già illustrate in precedenza (Figura 4.57).

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    binding = ActivityAccountBinding.inflate(layoutInflater)
    setContentView(binding.root)
    binding.lifecycleOwner = this
    setSupportActionBar(binding.toolbarAccount)
    navController = findNavController(com.projectrestaurant.R.id.nav_host_fragment_account)
    setupActionBarWithNavController(navController, AppBarConfiguration(navController.graph))
    supportActionBar?.setDisplayHomeAsUpEnabled(true)
}
```

Figura 4.57 - Implementazione del metodo iniziale dell'activity.

Il file *FragmentAccount.kt* implementa il fragment per poter visualizzare le informazioni principali associate all'account e per poter effettuare il logout. In *onViewCreated()* è possibile distinguere tre fasi principali (Figura 4.58):

1. Recupero dei dati dell'utente attraverso *getUserData()*, e memorizzazione in dei LiveData.
2. Uso di un MenuProvider per aggiungere nella barra di navigazione dell'activity la voce per modificare le informazioni che, una volta selezionata, naviga l'utente al fragment opportuno.
3. Si impostano dei callback per il pulsante di logout e per il cambio dell'indirizzo di spedizione. Per effettuare il logout si esegue il metodo *logout()*.

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)
    viewLifecycleOwner.lifecycleScope.launch(Dispatchers.IO) { viewModel.getUserData() }
    val menuHost = requireActivity() as MenuHost
    menuHost.addMenuProvider( provider: this, viewLifecycleOwner, Lifecycle.State.STARTED)
    if (activity is ActivityAccount) (activity as ActivityAccount).supportActionBar?.setDisplayHomeAsUpEnabled(true)
    binding.buttonLogout.setOnClickListener { viewModel.logout() }
    binding.buttonChangeAddress.setOnClickListener {
        navController.navigate(com.projectrestaurant.R.id.action_fragment_account_to_fragment_change_delivery_address)
    }
}
```

Figura 4.58 - Implementazione delle tre fasi principali.

Nel fragment è presente anche il metodo *onStart()*, al suo interno viene implementato un callback per chiudere l'activity in caso di logout dell'utente. Il callback è implementato tramite il metodo di Firebase *addAuthStateListener()* ed agendo sulla variabile *currentUser* (Figura 4.59).

```
override fun onStart() {
    super.onStart()
    auth.addAuthStateListener { if(it.currentUser == null) activity?.finish() }
}
```

Figura 4.59 - Implementazione del listener per intercettare la disconnessione.

FragmentEditUserData.kt implementa il fragment in cui l'utente può modificare nome, cognome ed e-mail. Per salvare le modifiche nel database, tuttavia, è necessario l'inserimento della password. La procedura di modifica è divisa nelle seguenti fasi:

1. Validazione dei dati inseriti. Il processo di validazione è lo stesso di quello implementato nei fragment per la registrazione ed il login. Per la password si controlla solamente che il campo per l'inserimento non è vuoto.
2. Se tutti i controlli danno esito positivo, si procede all'aggiornamento vero e proprio di nome, cognome ed e-mail tramite i metodi *changeUserNameAndSurname()* e *changeUserEmail()*, passando loro come parametri i dati da aggiornare (Figura 4.60).
3. Se entrambi gli aggiornamenti sono stati completati con successo viene mostrato un messaggio a schermo e l'applicazione fa tornare l'utente al fragment visitato in precedenza.
4. Se ci sono stati errori durante l'aggiornamento viene mostrato un dialog con un messaggio di errore opportuno.

```
viewLifecycleOwner.lifecycleScope.launch(Dispatchers.Main) {
    val result1 = withContext(Dispatchers.IO) {
        viewModel.changeUserNameAndSurname(binding.editTextName.text.toString(), binding.editTextSurname.text.toString())
    }
    val result2 = withContext(Dispatchers.IO) {
        viewModel.changeUserEmail(binding.editTextEmail.text.toString(), binding.editTextPassword.text.toString())
    }
    if(result1 && result2) {
        AlertDialog.Builder(requireContext()).setMessage("Update completed!")
            .setNeutralButton("OK") { _, _ -> navController.navigateUp() }.show()
    }
    if(!result1) {
        AlertDialog.Builder(requireContext()).setMessage("Unable to update your name and surname!")
            .setNeutralButton("OK") { _, _ -> navController.navigateUp() }.show()
    }
    if(!result2) {
        AlertDialog.Builder(requireContext()).setTitle("Unable to update email")
            .setMessage("Please check if the password is correct or try again later")
            .setNeutralButton("OK") { _, _ -> navController.navigateUp() }.show()
    }
}
```

Figura 4.60 - Aggiornamento dei dati.

FragmentChangeDeliveryAddress.kt contiene l'omonima classe del fragment che consente all'utente di cambiare l'indirizzo di spedizione predefinito, scegliendone uno da una lista contenente tutti quelli memorizzati nell'account. La lista è implementata in modo tale da rendere la scelta mutualmente esclusiva. In *onCreateView()*, in particolare, viene inizializzato un altro listener per intercettare dati inviati da un altro fragment come risultato (Figura 4.61). In questo caso, i dati ricevuti corrispondono ad un nuovo indirizzo e sono identificati dalla request key *addedNewAddress*. Il callback si occupa di controllare se l'oggetto indirizzo esiste e nel caso lo inserisce nella lista mostrata a schermo tramite il metodo *addToRadioGroup()*.

```
setFragmentResultListener( requestKey: "addedNewAddress") { _, bundle ->
    val new = bundle.getParcelable<Address>( key: "newAddress")
    if(new != null) addToRadioGroup(new, binding.radioGroupAddresses, requireContext(), addToList: true)
}
```

Figura 4.61 - Implementazione del listener per intercettare il risultato di un altro fragment.

In *onViewCreated()* si procede a mostrare a schermo la lista degli indirizzi, selezionando quello al momento impostato come predefinito, e a gestire gli eventi causati dall'utente (Figura 4.62):

1. Si recuperano gli indirizzi tramite getDeliveryAddresses() e li si inserisce nella lista con *addToRadioGroup()*.
2. Si aggiunge un listener per il pulsante per salvare la nuova scelta. Il callback controlla se è presente un elemento selezionato e, nel caso di risposta positiva, imposta il nuovo indirizzo predefinito con *setDefaultDeliveryAddress()* e successivamente fa tornare l'utente alla schermata precedente.
3. Si aggiunge un listener anche per il pulsante per aggiungere un nuovo indirizzo. Il pulsante, se premuto, naviga l'utente al fragment dedicato.

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)
    viewLifecycleOwner.lifecycleScope.launch(Dispatchers.IO) {
        list = viewModel.getDeliveryAddresses().toMutableList()
        withContext(Dispatchers.Main) { for(element in list) addToRadioGroup(element, binding.radioGroupAddresses, requireContext()) }
    }
    binding.buttonChangeAddress.setOnClickListener {
        viewLifecycleOwner.lifecycleScope.launch(Dispatchers.IO) {
            if(binding.radioGroupAddresses.checkedRadioButtonId != View.NO_ID) {
                viewModel.setDefaultDeliveryAddress(list[binding.radioGroupAddresses.checkedRadioButtonId].addressId)
                withContext(Dispatchers.Main) { navController.navigateUp() }
            }
        }
    }
    binding.buttonAddAddress.setOnClickListener {
        navController.navigate(com.projectrestaurant.R.id.action_fragment_change_delivery_address_to_fragment_new_address)
    }
}
```

Figura 4.62 - Recupero degli indirizzi e visualizzazione nello schermo.

FragmentNewAddress.kt implementa il fragment dedicato all'aggiunta di un nuovo indirizzo di spedizione. La parte più importante si trova in *onViewCreated()*, ed è l'implementazione del callback del pulsante per la sua registrazione nell'account (Figura 4.63). La procedura implementata è divisa in tre fasi principali:

1. Si verifica ogni stringa inserita per verificare la presenza di stringhe vuote e, nel caso, si provvede a mostrare un messaggio di errore e si interrompe l'intera procedura.
2. Superati tutti i test, il nuovo indirizzo viene memorizzato nell'account tramite *addDeliveryAddress()*.
3. Si controlla l'esito dell'aggiunta. Se è andata a buon fine si inserisce l'oggetto rappresentante il nuovo indirizzo in un bundle. Il bundle viene spedito come risultato al fragment che lo ha richiesto tramite *setFragmentResult()*. In questo caso il fragment richiedente corrisponde a *FragmentChangeDeliveryAddress*, dato che viene indicata la stessa request key. Se invece la procedura non è terminata con successo viene mostrato a schermo un dialog contenente un messaggio di errore.

```
viewLifecycleOwner.lifecycleScope.launch(Dispatchers.Main) {
    val address = withContext(Dispatchers.IO) {
        viewModel.addDeliveryAddress(editTextAddress.text.toString(),
            editTextPostcode.text.toString(), editTextCity.text.toString(),
            editTextProvince.text.toString(), checkBoxDefaultAddress.isChecked)
    }
    if(address != null) {
        val bundle = Bundle()
        Toast.makeText(requireContext(), "Added new address", Toast.LENGTH_LONG).show()
        bundle.putParcelable("newAddress", address)
        setFragmentResult( requestKey: "addedNewAddress", bundle)
        navController.navigateUp()
    } else {
        AlertDialog.Builder(requireContext()).setTitle("Something went wrong")
            .setMessage("Error while creating this address, check if it already ...")
            .setPositiveButton("OK") { _, _ -> }.show()
    }
}
```

Figura 4.63 - Sezione principale del callback.

4.4.7 Package ui/userorders

Questo package contiene *ActivityUserOrders.kt*, che implementa l'activity dedicata alla visualizzazione degli ordini effettuati in passato. L'implementazione della classe è composta solo dal metodo *onCreate()*. Quest'ultimo svolge le sole operazioni di routine per la creazione di un'activity, ossia inflating del layout grafico e preparazione del NavController per la navigazione (Figura 4.64).

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    viewModel = OrdersViewModel(application)
    binding = ActivityUserOrdersBinding.inflate(layoutInflater)
    setContentView(binding.root)
    setSupportActionBar(binding.toolbarUserOrders)
    binding.lifecycleOwner = this
    navController = findNavController(com.projectrestaurant.R.id.nav_host_fragment_user_orders)
    setupActionBarWithNavController(navController, AppBarConfiguration(navController.graph))
    supportActionBar?.setDisplayHomeAsUpEnabled(true)
}
```

Figura 4.64 - Implementazione del metodo iniziale.

FragmentUserOrders.kt implementa il fragment per la visualizzazione di una lista contenente gli ordini effettuati di recente. Nel metodo *onCreateView()* viene inizializzato l'adapter necessario per la lista e viene definito il layout manager per il recycler view (Figura 4.65).

```
override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?): View {
    super.onCreateView(inflater, container, savedInstanceState)
    binding = FragmentUserOrdersBinding.inflate(inflater, container, attachToRoot: false)
    adapter = OrderAdapter(requireActivity().application, viewModel, findNavController())
    binding.recyclerViewOrders.layoutManager = LinearLayoutManager(requireContext(), LinearLayoutManager.VERTICAL, reverseLayout: false)
    return binding.root
}
```

Figura 4.65 - Implementazione di "onCreateView".

La procedura implementata in *onViewCreated()* è composta da diverse fasi principali (Figura 4.66):

1. Recupero degli ordini tramite *getOrders()*, per ogni ordine si recuperano anche i relativi prodotti ordinati con *getOrderProducts()* e li si memorizza in un'unica struttura dati.
2. Si passano i dati recuperati all'adapter con *setData()*.
3. Esecuzione di *hasMultiplePages()* per controllare se il numero di elementi da visualizzare è abbastanza elevato da dover suddividere la lista in sezioni disposte in più pagine. Se il controllo da esito positivo, si imposta un LiveData per visualizzare dinamicamente il numero di pagina corrente e si implementano i listener per i due pulsanti per navigare tra di esse. I callback eseguono *goToNextPage()* e *goToPreviousPage()*, rispettivamente per scorrere in avanti e all'indietro.

```
viewLifecycleOwner.lifecycleScope.launch(Dispatchers.IO) {
    val orderList = viewModel.getOrders()
    val productList = mutableListOf<OrderProduct>()
    for(order in orderList) productList.addAll(viewModel.getOrderProducts(order))
    withContext(Dispatchers.Main) {
        progressBar.isIndeterminate = false
        constraintLayout.overlay.remove(progressBar)
        progressBar.visibility = View.GONE
        recyclerViewOrders.adapter = adapter
        adapter.setData(orderList, productList)
        if(adapter.hasMultiplePages()) {
            currentPageObserver = Observer { newValue ->
                textViewPagesList.text = "{newValue.toString()} / {adapter.numberOfPages.toString()}"
            }
            adapter.currentPage.observe(viewLifecycleOwner, currentPageObserver)
            buttonNextPage.setOnClickListener { adapter.goToNextPage() }
            buttonPreviousPage.setOnClickListener { adapter.goToPreviousPage() }
```

Figura 4.66 - Implementazione della porzione principale.

FragmentUserOrderedProducts.kt implementa il fragment in cui l'utente può visualizzare tutti i prodotti contenuti in un determinato ordine. Il metodo *onCreateView()* è totalmente analogo a quello del fragment precedente. In *onViewCreated()*, invece, la procedura eseguita è differente (Figura 4.67):

1. Recupero dei prodotti dell'ordine con *getOrderProducts()* e delle modifiche apportate a tali prodotti con *getOrderProductEdits()*. Vengono recuperate anche le immagini dei prodotti tramite *getFoodImage()*, in modo da visualizzarle nella lista.
2. Si passano i dati recuperati all'adapter con *setData()*.
3. Esecuzione di *hasMultiplePages()* per controllare se il numero di elementi da visualizzare è abbastanza elevato da dover suddividere la lista in sezioni disposte in più pagine. Se il controllo da esito positivo, si imposta un LiveData per visualizzare dinamicamente il numero di pagina corrente e si implementano i listener per i due pulsanti per navigare tra di esse. I callback eseguono *goToNextPage()* e *goToPreviousPage()*, rispettivamente per scorrere in avanti e all'indietro.

```
viewLifecycleOwner.lifecycleScope.launch(Dispatchers.IO) {
    val productList = viewModel.getOrderProducts(args.order)
    val productEditsList = mutableListOf<OrderProductEdit>()
    val productImages = hashMapOf<Int, String?>()
    for(product in productList) {
        productEditsList.addAll(viewModel.getOrderProductEdits(product))
        productImages[product.foodId] = viewModel.getFoodImage(product.foodId)
    }
    withContext(Dispatchers.Main) {
        progressBar.isIndeterminate = false
        constraintLayout.overlay.remove(progressBar)
        progressBar.visibility = View.GONE
        recyclerViewOrderedProducts.adapter = adapter
        adapter.setData(productList, productEditsList, productImages)
        if(adapter.hasMultiplePages()) {
            currentPageObserver = Observer { newValue ->
                textViewPagesList.text = "{newValue.toString()} / {adapter.numberOfPages.toString()}"
            }
            adapter.currentPage.observe(viewLifecycleOwner, currentPageObserver)
            buttonNextPage.setOnClickListener { adapter.goToNextPage() }
            buttonPreviousPage.setOnClickListener { adapter.goToPreviousPage() }
```

Figura 4.67 - Implementazione della sezione principale.

4.5 Schermate dell'applicazione

In questa sezione verranno mostrate le varie schermate che compongono l'applicazione, ne verrà descritto il loro contenuto e quali azioni l'utente può fare. Tali schermate sono state ricavate da uno schermo di risoluzione 720 x 1480 pixel.

4.5.1 Schermata iniziale

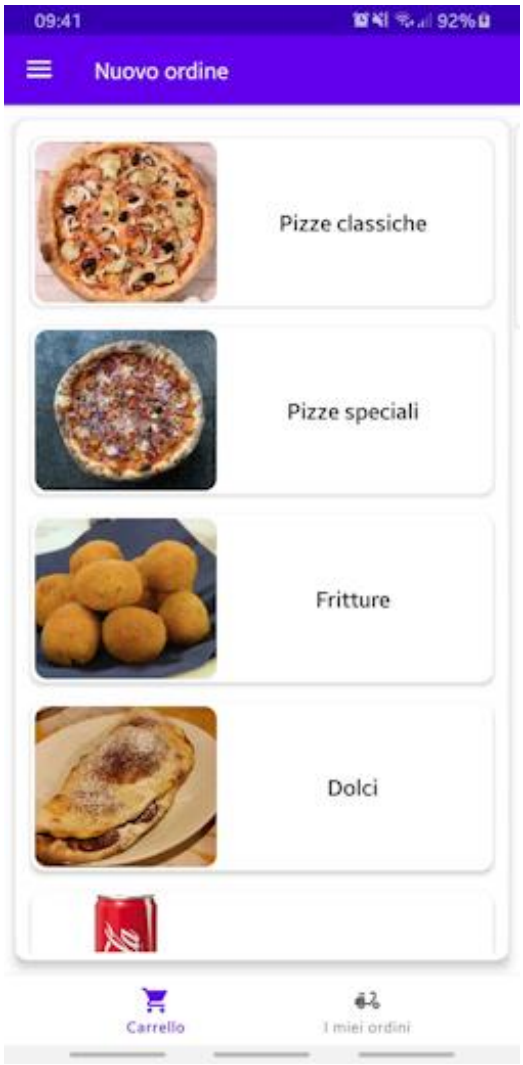

Figura 4.68 - Schermata iniziale dell'applicazione con barra di navigazione visibile.

La prima cosa che viene mostrata all'utente è una lista contenente i tipi di prodotto ordinabili, per ogni elemento ne viene mostrato il nome ed un'immagine descrittiva per facilitare l'interazione.
Se l'utente è autenticato, sotto la lista comparirà una barra di navigazione con due possibili opzioni:

- I miei ordini: È accompagnata dall'icona di uno scooter da consegne. Permette di accedere alla schermata in cui è possibile visualizzare gli ordini fatti di recente.
- Carrello: È accompagnata dall'icona di un carrello della spesa, e compare solo se ci sono già dei prodotti al suo interno. Permette di accedere alla schermata in cui è possibile visualizzare i prodotti inseriti, modificarli oppure completare l'ordine.

Nella Figura 4.68 viene mostrata la schermata iniziale contenente anche la barra di navigazione appena descritta.

Cliccando sull'icona in alto a sinistra (l'icona delle tre linee orizzontali) o scorrendo con il dito da sinistra verso destra è possibile aprire un menu a comparsa (Figura 4.69). Nel menù sono presenti diverse voci:

- Impostazioni
- Chi siamo: Contiene informazioni sul ristorante non strettamente legate all'ordinazione.
- Polizza sulla privacy: Specifica le modalità con cui verranno trattati i dati sensibili degli utenti e le leggi a cui tali modalità fanno riferimento.

Queste tre voci sono sempre presenti indipendentemente se l'utente si è già autenticato oppure no. Nel menu è presente una quarta voce che, a differenza delle altre, varia in base ai due casi citati:

- Accedi / Registrati: Appare se non si ha effettuato il login. Cliccando su di essa l'applicazione mostra all'utente una schermata in cui autenticarsi oppure, se non dispone ancora di un account, di crearne uno nuovo.
- Profilo: Appare se si ha effettuato il login in precedenza e non si ha ancora effettuato il logout. Attraverso questa voce l'utente può visualizzare una schermata in cui visualizzare e modificare le proprie informazioni oppure effettuare il logout.

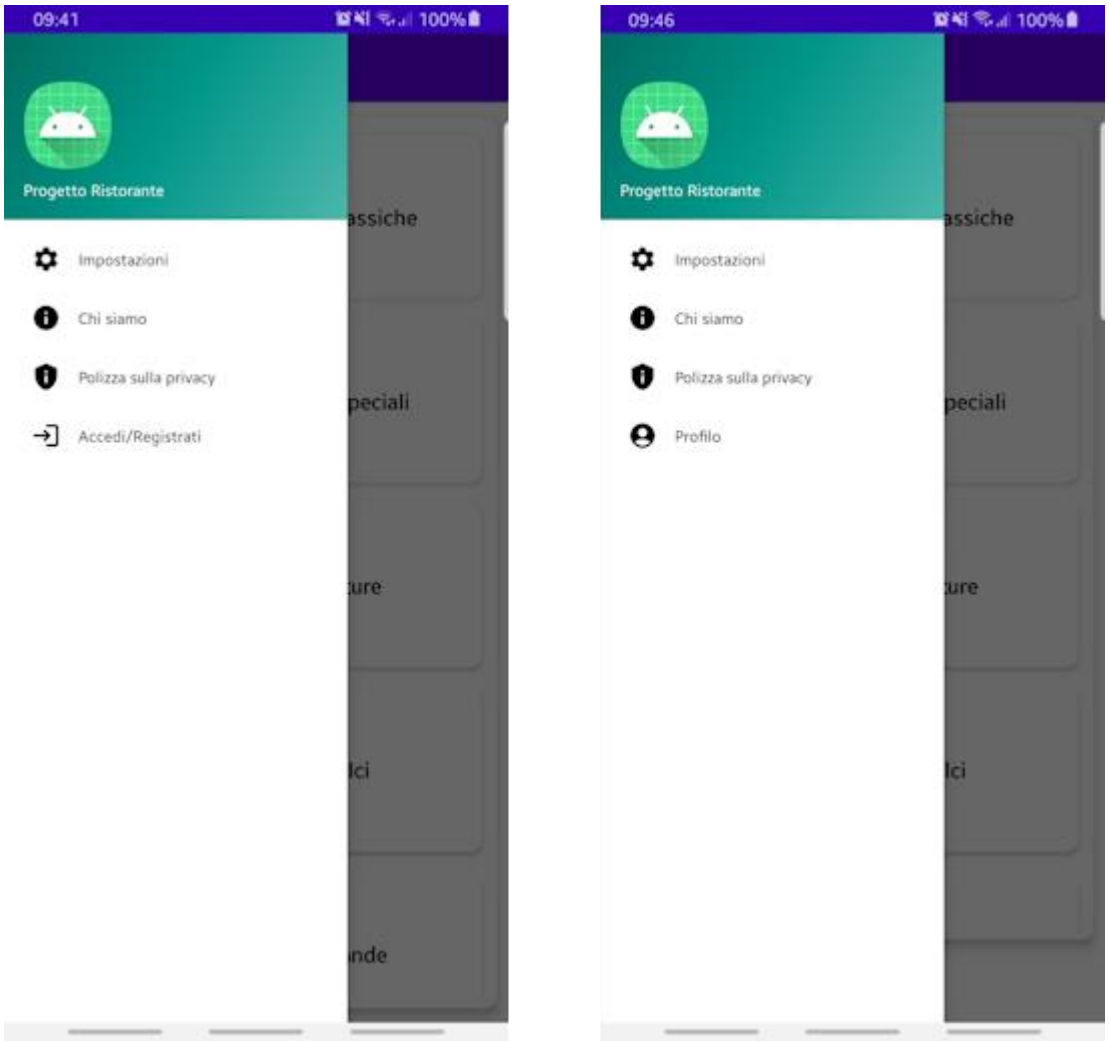

Figura 4.69 - Come appare il menù quando non si è autenticati (a sinistra) rispetto a quando invece sì (a destra).

4.5.2 Schermata dei prodotti

È accessibile selezionando uno degli elementi presenti della lista iniziale. Il corpo principale è costituito da una lista dei prodotti appartenenti al tipo scelto, per ciascun prodotto vengono mostrati:

- Il nome, indicato in grassetto.
- Una breve descrizione.
- Il prezzo unitario, indicato in grassetto.
- Un'immagine descrittiva.

In alto a destra, nella barra di navigazione, è presente un'icona di una lente d'ingrandimento. Una volta premuta si apre una finestra con cui l'utente può inserire del testo per cercare dei prodotti in base ad uno o più termini presente nel nome e/o nella descrizione. In alto a sinistra invece è presente un'icona di una freccia che permette di tornare alla schermata precedente. Questo elemento è presente in tutte le schermate eccetto quella iniziale.
Se l'utente è già autenticato e ha già dei prodotti nel carrello, in basso compare il pulsante "VAI AL CARRELLO" che consente di accedere direttamente alla schermata dedicata per modificarne il contenuto oppure completare l'ordine definitivo (Figura 4.70).

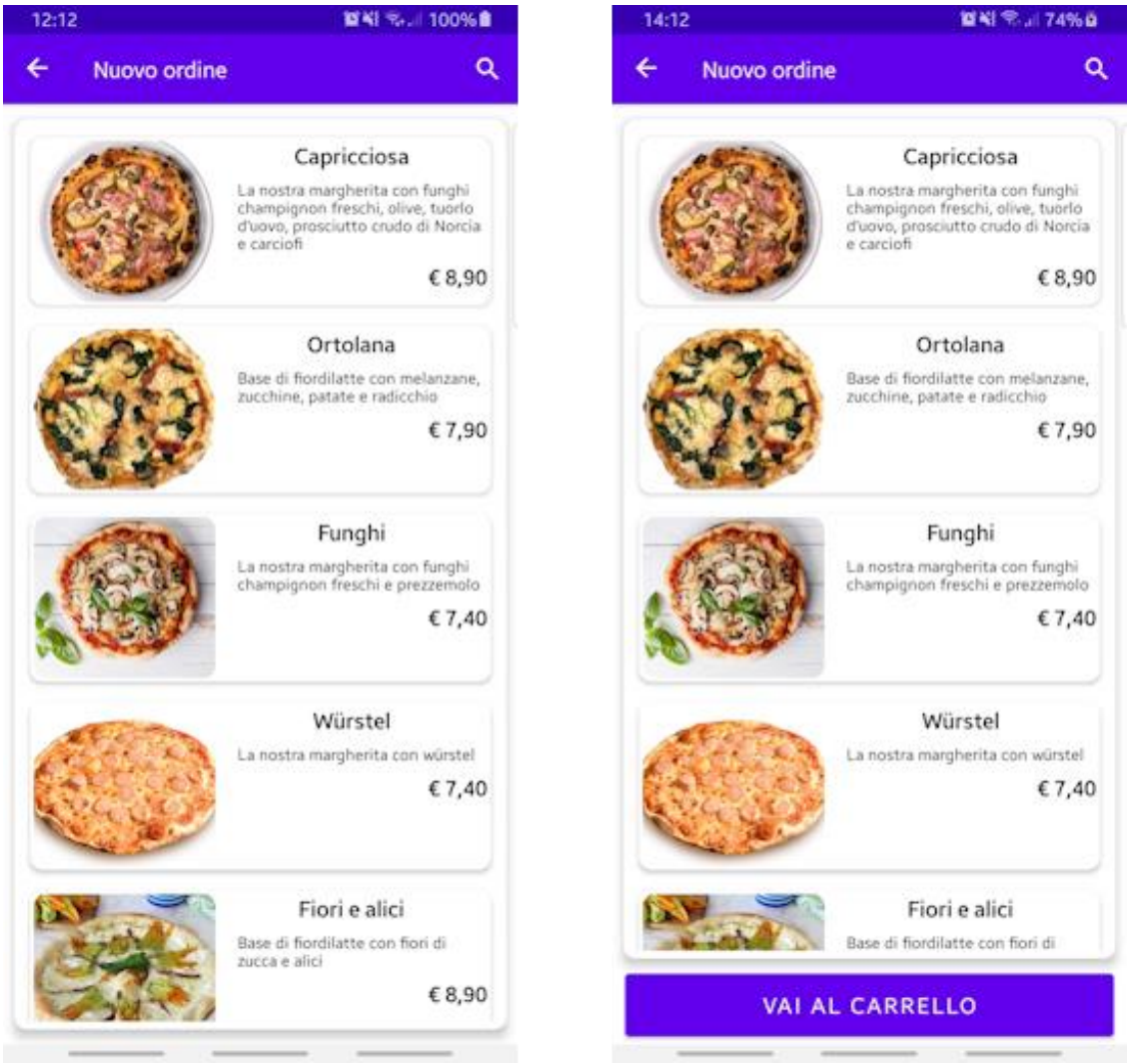

Figura 4.70 - Schermata di selezione prodotto con e senza il tasto per accedere al carrello.

4.5.3 Schermata per personalizzare il prodotto

Cliccando sul prodotto scelto si accede a questa schermata (Figura 4.71). Nella parte superiore vengono rimostrati il nome, la descrizione e l'immagine, mentre nella parte centrale è possibile modificarne la quantità da ordinare e, dove possibile, gli ingredienti presenti.
La sezione per modificare la quantità è composta da tre elementi:

- Pulsante per aumentarla: Contiene un'icona del segno "+".
- Pulsante per diminuirla: Contiene un'icona del segno "-".
- Contatore del valore corrente: È posizionato a metà tra i due pulsanti.

Di default, la quantità di prodotto è impostata ad uno. Per ciascun ingrediente, invece, viene mostrato:

- Nome
- Immagine descrittiva
- Prezzo unitario
- Pulsanti per modificarne la quantità, hanno lo stesso design descritto prima.

Di default, ciascun ingrediente è impostato sulla quantità necessaria per quel prodotto. In questo caso premendo il pulsante "-" l'ingrediente associato verrà rimosso del tutto dal prodotto ed il suo nome comparirà sbarrato. Premendo invece il pulsane "+", l'ingrediente viene aggiunto in quantità maggiore alla norma, in questo caso viene mostrato il relativo prezzo aggiuntivo per singolo prodotto ed il prezzo totale si modifica in modo opportuno. Se l'ingrediente è rimosso del tutto, premendo "+" esso è riportato alla quantità predefinita e viene rimossa la sbarratura nel nome. Se invece è presente in quantità maggiore, premendo "-" viene riportato alla quantità predefinita, nascondendo il prezzo unitario e modificando opportunamente il prezzo totale.

Una volta apportate tutte le modifiche desiderate, premendo il tasto "AGGIUNGI AL CARRELLO", presente nella parte inferiore è possibile aggiungere il tutto nel carrello. Il testo nel pulsante mostra la quantità scelta ed il prezzo totale del prodotto, e si aggiorna dinamicamente in base alle azioni compiute dall'utente.
Dopo averlo aggiunto nel carrello l'applicazione riporta l'utente alla schermata della lista prodotti.

Figura 4.71 - Schermata per personalizzare il prodotto.

4.5.4 Schermata del carrello

Qui è possibile visualizzare tutti i prodotti contenuti, decidere se modificarne o eliminarne alcuni e svolgere gli ultimi passaggi per effettuare l'ordine vero e proprio (Figura 4.72).
La lista dei prodotti si trova nella parte superiore, per ciascun prodotto viene mostrato:

- Il nome, scritto in grassetto.
- L'immagine descrittiva
- Una lista degli eventuali ingredienti rimossi, vengono indicati con il nome sbarrato.
- Una lista degli eventuali ingredienti maggiorati, viene anteposto il prefisso "Extra" davanti al loro nome.
- La quantità scelta
- Il prezzo totale, scritto in grassetto.
- Un tasto per rimuoverlo dal carrello: Ha al suo interno l'icona di un cestino.
- Un tasto per modificarlo: Ha al suo interno l'icona di una matita.

Una volta premuto il pulsante per modificare un determinato prodotto, l'applicazione navigherà l'utente alla schermata dedicata. La cancellazione di un prodotto non è reversibile, l'utente dovrà aggiungerlo di nuovo nel carrello se lo desidera.

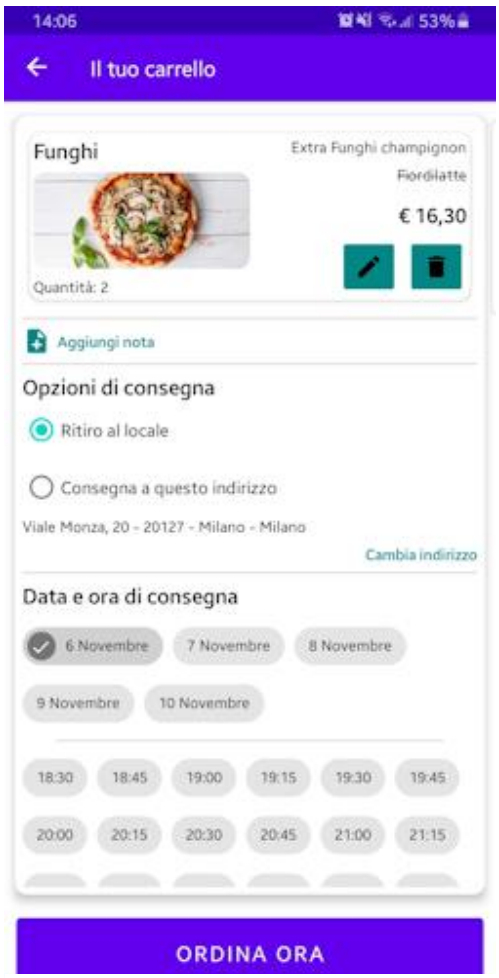

Figura 4.72 - Schermata del carrello.

Al di sotto della lista vi è la scritta "Aggiungi nota" per aggiungere una nota personalizzata all'ordine. Una volta premuto, l'applicazione porta l'utente alla schermata per configurarla.
Procedendo ancora verso il basso si trovano le impostazioni per il luogo, la data e l'ora di consegna. Nella scelta del luogo l'utente può decidere di ritirare ciò che ha ordinato direttamene al ristorante oppure decidere che vengano consegnati ad un indirizzo specificato. Come indirizzo l'applicazione mostrerà quello scelto come predefinito dall'utente, se esiste. Premendo sulla scritta "Cambia indirizzo" l'utente può aggiungerne di nuovi oppure sceglierne uno diverso da quello di default.
Nell'impostare la data, l'utente può scegliere il giorno corrente, se il ristorante risulta ancora aperto, oppure dei giorni successivi. Una volta scelta la data sarà possibile scegliere l'ora selezionandola da una lista di quelle disponibili che comparirà per l'occasione.

Come ultimo elemento, in basso si trova il pulsante "ORDINA ORA" per effettuare l'ordine. Una volta premuto e completato l'ordine, l'utente è riportato alla schermata della lista prodotti. Per poter completare un ordine, tuttavia, è obbligatorio aver selezionato almeno luogo, data ed ora di consegna, altrimenti l'operazione non andrà a buon fine.

4.5.5 Schermata per modificare un prodotto nel carrello

Questa schermata è praticamente identica a quella per personalizzare il prodotto. Le poche differenze tra le due sono le seguenti:

- Il pulsante in fondo alla schermata ora contiene la scritta "AGGIORNA CARRELLO" ed il prezzo totale del prodotto (Figura 4.73), ed anche qui il tutto si aggiorna dinamicamente a seconda delle azioni svolte dall'utente.
- Premendo tale pulsante l'applicazione provvederà ad apportare le modifiche desiderate al prodotto, per poi far tornare l'utente alla schermata del carrello.

Figura 4.73 - Schermata per modificare un prodotto.

4.5.6 Schermata per aggiungere una nota all'ordine

È costituita principalmente da una casella di testo ed un pulsante (Figura 4.74).
All'apertura, la casella contiene il testo della nota, se è già stata impostata in precedenza. Una volta scritto il testo o apportate le modifiche volute, premendo il pulsante "AGGIUNGI NOTA" è possibile salvare nel profilo il lavoro svolto, rendendolo così accessibile anche da altri dispositivi. Per assicurarsi che le note inserite siano brevi, la casella è stata impostata con un limite massimo di caratteri per il testo, oltre il quale non è più possibile scrivere.
Una volta effettuato l'ordine, una copia della nota verrà associata a quest'ultimo, mentre quella principale verrà eliminata dal profilo utente.

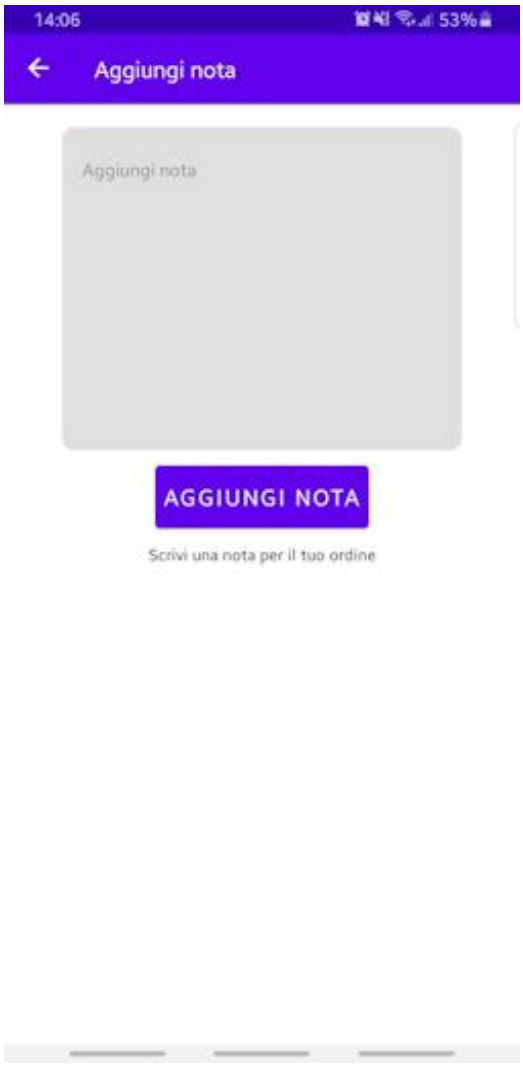

Figura 4.74 - Schermata per impostare una nota personalizzata.

4.5.7 Schermata di autenticazione

Scegliendo l'opzione "Accedi / Registrati" nel menù della homepage l'applicazione naviga l'utente verso questa schermata (Figura 4.75).
Nella parte superiore vi è il pulsante "Registrati" che, una volta premuto, permette di accedere alla schermata dedicata alla creazione di un nuovo profilo.
Nel corpo centrale vi sono due caselle di testo in cui bisogna inserire le credenziali necessarie per effettuare l'accesso, ossia e-mail e password. Di default, i caratteri della password sono oscurati per questioni di sicurezza, ma è possibile visualizzarli in chiaro premendo l'icona dell'occhio posta dentro la casella. È possibile ritornare alla visualizzazione oscurata premendo di nuovo l'icona.
Una volta compilati tutti i campi, premendo il pulsante "ACCEDI" l'applicazione autenticherà l'utente per poi riportarlo alla schermata iniziale. Se, invece, le credenziali sono errate mostrerà un messaggio di errore chiedendo di modificarle e riprovare.
Se l'utente ha dimenticato la password di accesso, cliccando sulla scritta "Password dimenticata?", posta sotto la casella della password, l'applicazione navigherà l'utente verso la schermata dedicata.

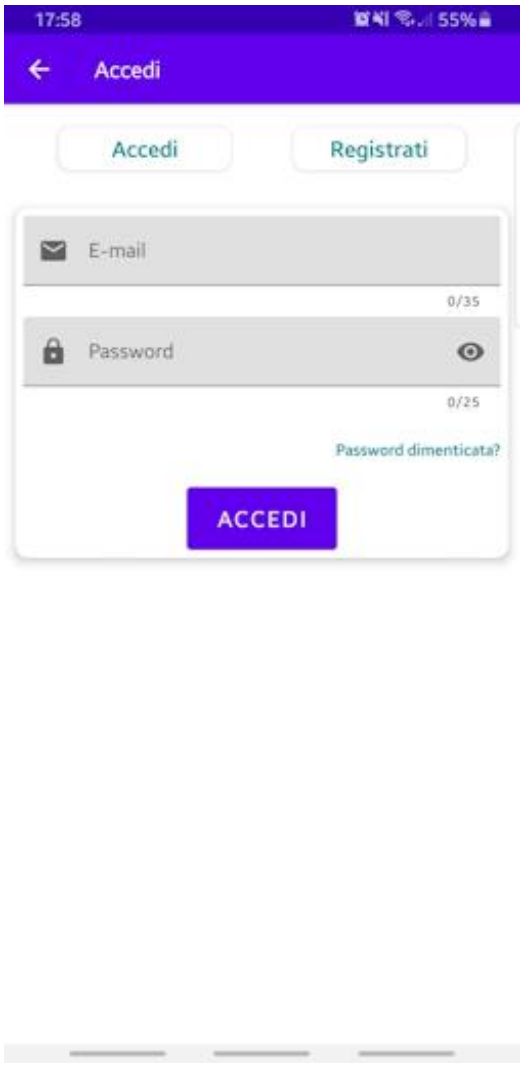

Figura 4.75 - Schermata per effettuare il login.

4.5.8 Schermata di recupero della password

Questa schermata è molto semplice, è composta infatti solamente da una casella di testo e da un pulsante (Figura 4.76). Per recuperare la password l'utente deve inserire la propria e-mail e successivamente premere il pulsante "REIMPOSTA PASSWORD".
Fatto ciò, l'applicazione verifica prima la correttezza dell'informazione inserita e, se i controlli danno esito positivo, invia un'e-mail automatica a quell'indirizzo. In questa e-mail sono contenute le istruzioni per consentire al sistema di generare una nuova password.

Figura 4.76 - Schermata per reimpostare la password.

4.5.9 Schermata per la creazione di un nuovo profilo

In questa schermata l'utente deve inserire i dati necessari per la creazione del profilo ed accettare la polizza sulla privacy del ristorante. Nella parte superiore vi è il pulsante "Accedi" per tornare alla schermata di acceso. Nel corpo centrale vi sono una serie di campi da compilare con i seguenti dati:

- Nome
- Cognome
- Indirizzo e-mail
- Password

Tutte le caselle di testo hanno un limite di caratteri preimpostato, oltre il quale non si può più scrivere. Per ragioni di sicurezza la password è soggetta a dei vincoli aggiuntivi:

- Deve contenere almeno due caratteri minuscoli, due maiuscoli, due cifre ed almeno un carattere speciale. In aggiunta a tutto questo, deve essere inserita due volte.
- Così come accade nella schermata di login, i suoi caratteri sono di default oscurati.

Una volta compilati tutti i campi e messo la spunta per accettare la polizza sulla privacy, premendo il pulsante "REGISTRATI" poso in basso, l'applicazione controllerà la correttezza delle informazioni e provvederà a registrare il profilo utente nel sistema per poi autenticarlo ed infine riportarlo alla schermata iniziale. Nel caso di informazioni errate e/o mancanti mostrerà un messaggio di errore chiedendo all'utente di modificare i dati e riprovare.
Nella Figura 4.77 è possibile vedere l'aspetto della schermata.

Figura 4.77 - Schermata per registrare un nuovo profilo utente.

4.5.10 Schermata del profilo utente

Scegliendo la voce "Profilo" nel menù della homepage si accede alla schermata in cui l'utente può visualizzare i suoi dati memorizzati nel profilo oppure effettuare il logout (Figura 4.78).
Nella parte centrale sono presenti delle caselle con cui non è possibile interagire che contengono:

- Il nome
- Il cognome
- L'indirizzo e-mail
- L'indirizzo di spedizione predefinito, se impostato

Per modificare l'indirizzo di spedizione predefinito, a prescindere ci esista oppure no, l'utente deve cliccare sul pulsante "Cambia indirizzo" posto sotto la sua cesella. Tale pulsante porta l'utente alla schermata per selezionare un indirizzo.
Per effettuare il logout l'utente deve premere il pulsante "ESCI": una volta completata la procedura l'applicazione riporta l'utente alla schermata iniziale.
A destra nella barra superiore è presente un pulsante con l'icona di una matita: una volta premuto l'utente viene portato nella schermata per modificare le proprie informazioni salvate nel profilo.

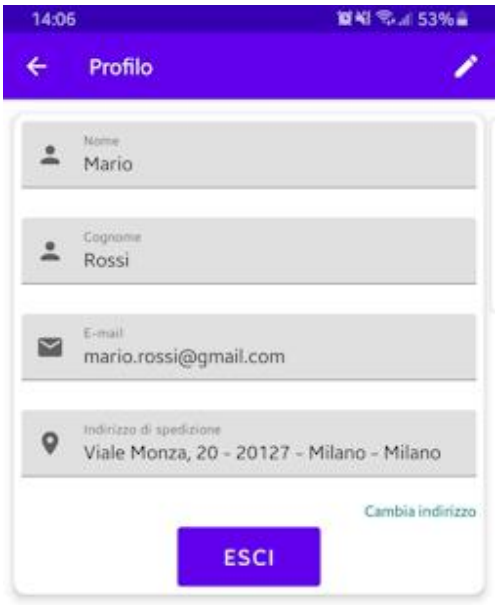

Figura 4.78 - Schermata del profilo utente.

4.5.11 Schermata per modificare i dati dell'utente

Questa schermata gode di una struttura molto simile a quella precedente. All'avvio, le prime tre caselle di testo contengono le stesse informazioni viste nella schermata del profilo, mentre quella contenente l'indirizzo di spedizione è stata sostituita da una casella vuota in cui è richiesto l'inserimento della password associata all'account (Figura 4.79). Rispetto alla schermata precedente l'utente può ora interagire con queste caselle e modificarne il contenuto.

Una volta apportate le modifiche desiderate ai propri dati, l'utente deve inserire la password dell'account per renderle effettive. Nel premere il pulsante "SALVA MODIFICHE" l'applicazione controlla se la password inserita coincide con quella memorizzata nel sistema e, se la risposta è positiva, procede ad aggiornare i campi desiderati. Se, invece, la password è assente oppure è errata, l'app mostra un messaggio di errore chiedendo all'utente di riprovare inserendo la password corretta.

Per rendere i dati consistenti con quelli immessi in fase di registrazione, le caselle sono state dotate da un limite massimo di caratteri oltre il quale non si può più scrivere. Per una maggiore sicurezza, anche qui i caratteri della password sono oscurati di default.

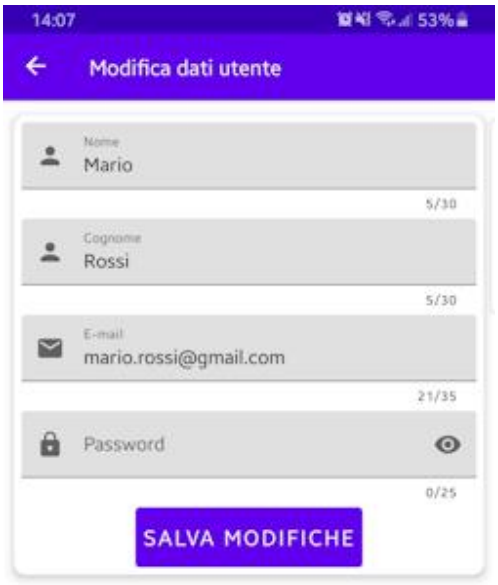

Figura 4.79 - Schermata per modificare i propri dati.

4.5.12 Schermata per selezionare un indirizzo di spedizione

Questa schermata è accessibile sia da quella del profilo sia da quella del carrello. Qui l'utente può cambiare l'indirizzo di spedizione predefinito da utilizzare negli ordini. Nella parte centrale vi è una lista contenente tutti gli indirizzi memorizzati nel profilo. Dato che un solo indirizzo per volta può essere impostato come predefinito, la selezione di un elemento nella lista avviene in modo mutualmente esclusivo.
Una vola selezionato l'indirizzo scelto, premendo il pulsante "SELEZIONA INDIRIZZO" l'applicazione provvederà ad aggiornare i dati nel sistema per impostarlo come quello di default, per poi far tornare l'utente alla schermata del profilo.
Se invece l'indirizzo desiderato non è presente nella lista, cliccando sul pulsate con il simbolo "+" posto in basso a destra è possibile accedere alla schermata in cui aggiungerne uno nuovo.
La Figura 4.80 mostra l'aspetto della schermata appena descritta.

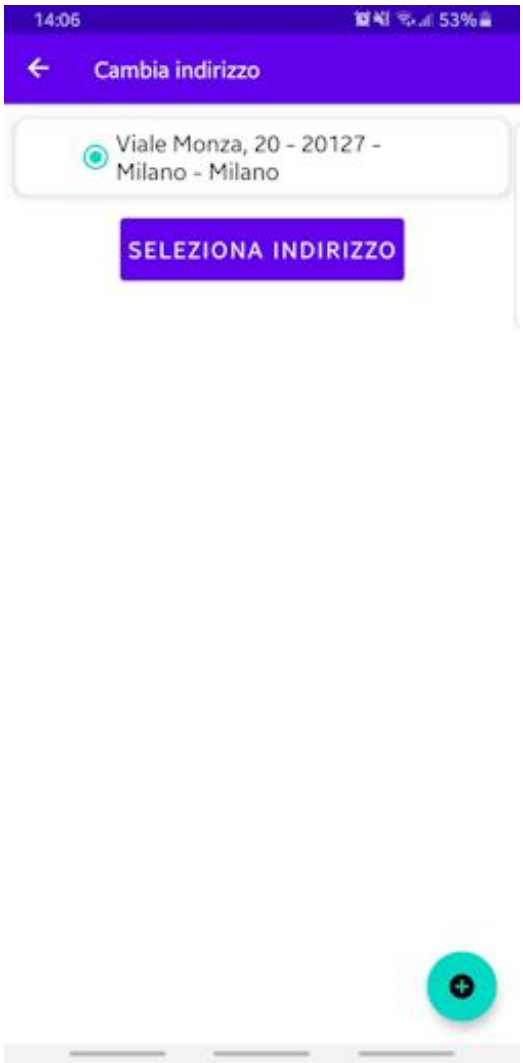

Figura 4.80 - Schermata di selezione dell'indirizzo.

4.5.13 Schermata per aggiungere un nuovo indirizzo

In questa schermata l'utente, oltre ad aggiungere un nuovo indirizzo di spedizione, può anche specificare se impostarlo fin da subito come indirizzo predefinito oppure no.
La schermata è composta principalmente da dei campi da compilare e da un pulsante (Figura 4.81), in questi campi l'utente deve inserire:

- L'indirizzo in sé, composto da via e numero civico.
- Il codice di avviamento postale (CAP)
- La città
- La provincia

In modo analogo al caso della registrazione, anche qui le caselle di testo hanno un numero limitato di caratteri utilizzabili. Per impostare il nuovo indirizzo come quello di default è necessario mettere una spunta nell'opzione dedicata che si trova sotto l'ultima casella di testo.
Una volta compilato tutti i campi, premendo il pulsante "AGGIUNGI INDIRIZZO" l'applicazione provvederà ad aggiungerlo nei dati associati al profilo e ad aggiornare quello legato all'indirizzo predefinito, se necessario. Una volta fatto tutto questo, l'applicazione provvede a far tornare l'utente alla schermata della lista degli indirizzi.

Figura 4.81 - Schermata per aggiungere un nuovo indirizzo.

4.5.14 Schermata degli ordini effettuati di recente

È accessibile scegliendo l'opzione "I miei ordini" nella barra di navigazione presente nella schermata iniziale. Qui è possibile visualizzare una lista contenente tutti quelli effettuati di recente (Figura 4.82). Per ciascun elemento della lista vengono mostrati:

- I prodotti ordinati, indicando per ciascuno le rispettive quantità.
- La quantità totale di prodotti ordinati.
- Il prezzo totale dell'ordine, indicato in grassetto.
- Data in cui è stato effettuato l'ordine.
- Data in cui è avvenuta, oppure avverrà, la consegna.

Premendo su un qualsiasi elemento della lista, l'applicazione mostra all'utente una schermata in cui può visualizzare con maggior dettaglio i singoli prodotti presenti in quell'ordine.
Se la lista contiene molti elementi, l'applicazione la suddivide in più parti per non comprimere il tutto in un'unica schermata. In questo caso nella parte inferiore della schermata, al di sotto di ciascuna sezione della lista, vengono aggiunti due pulsanti ed una scritta. I due pulsanti "PRECEDENTE" e "SUCCESSIVO" permettono rispettivamente di passare alla parte precedente e successiva, se possibile. La scritta, posta a metà strada tra i due pulsanti, da informazioni sulla parte corrente e sul loro numero complessivo, e si aggiorna dinamicamente a seconda delle azioni dell'utente.

Figura 4.82 - Schermata per visualizzare gli ordini più recenti.

4.5.15 Schermata per visualizzare i prodotti di un ordine

In questa schermata viene mostrata una lista di tutti i prodotti contenuti in un determinato ordine fatto di recente dall'utente (Figura 4.83).
Per ciascun prodotto nella lista vengono mostrate le stesse informazioni viste nella schermata del carrello, ossia:

- Nome, scritto in grassetto.
- Immagine descrittiva.
- Lista degli ingredienti maggiorati, se presenti. Il loro nome è preceduto dal prefisso "Extra".
- Lista degli ingredienti rimossi, se presenti. Sono mostrati con il nome sbarrato.
- Quantità ordinata.
- Prezzo totale, scritto in grassetto.

I prodotti sono separati in base a come sono stati personalizzati gli ingredienti. Questo significa che se lo stesso prodotto è stato personalizzato in modi diversi, allora comparirà più volte nella lista, una per ogni variante ordinata. Anche qui, come nella schermata precedente, se vi sono troppi elementi nella lista, l'applicazione la divide in sezione. Vengono aggiunti in fondo alla schermata, per l'occasione, gli stessi elementi descritti in precedenza con le stesse funzioni.

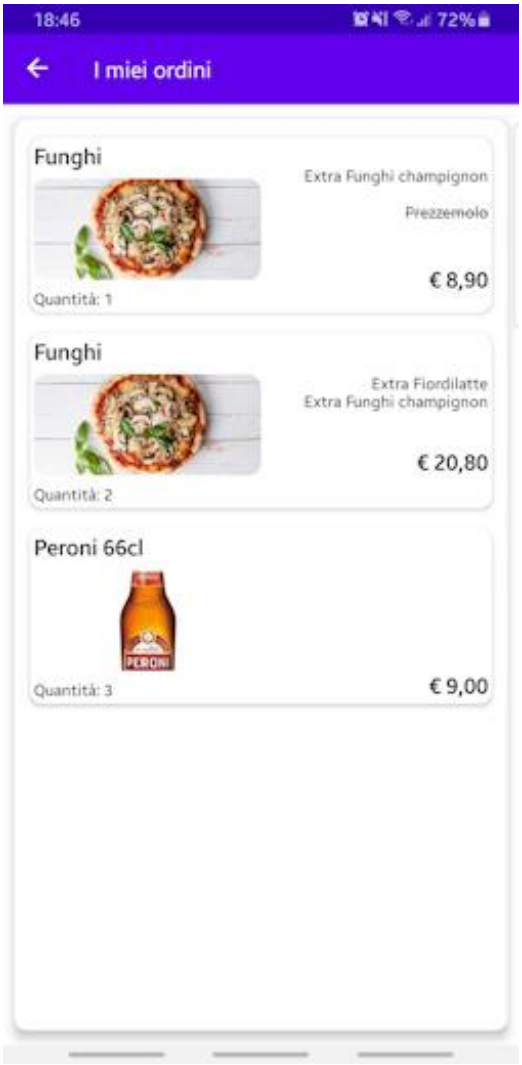

Figura 4.83 - Schermata della lista dei prodotti di un ordine.

Conclusione

In questa tesi, è stato ampiamente descritto un tipo caso di sviluppo di un'applicazione per dispositivi mobili. Si è partiti dal processo di analisi dei requisiti, passando successivamente all'analisi dei casi d'uso, alla progettazione delle singole funzionalità, per terminare infine con l'implementazione vera e propria. Nel documentare le varie fasi è stato fondamentale riservare un trattamento simile anche agli strumenti di sviluppo utilizzati, partendo dal linguaggio di programmazione fino ad interi servizi web che consentono di implementare facilmente operazioni altrimenti complesse.
Come è facile immaginare, tuttavia, quelli descritti in questo documento sono solo una piccola parte di quelli al momento disponibili all'uso. Grazie al rapido avanzamento tecnologico, ed in particolare alla sempre più ampia diffusione dell'intelligenza artificiale, la selezione di strumenti utilizzabili è in continua evoluzione.
Altro aspetto in continua evoluzione sono le abitudini e le esigenze dei consumatori, e ciò comporta per gli sviluppatori applicare nuove filosofie di sviluppo per avere la possibilità di rendere rilevanti ed appetibili i propri prodotti e servizi digitali.
Proprio per questi due fattori, questa tesi non rappresenta la fine dello sviluppo dell'applicazione Ristorante, nuove funzionalità ed aggiornamenti a quelle già esistenti saranno fondamentali per la sua sopravvivenza sul mercato. Alcuni esempi di queste nuove funzionalità potrebbero essere un sistema di offerte a tempo limitato per determinati prodotti, oppure la possibilità da parte degli utenti di accumulare coupon spendibili in app. In entrambi i casi si avrebbe un miglioramento dell'interazione da parte degli utenti, i quali avrebbero motivazioni aggiuntive per continuare ad usare l'app.
Per concludere, l'esperienza accumulata nello sviluppo di questo progetto non è strettamente legata a quest'ultimo, ma può fornire utili insegnamenti ed ottimizzazioni applicabili anche ad altri progetti in ambito mobile.

Bibliografia

[1] Use Case Diagram - https://www.geeksforgeeks.org/use-case-diagram

[2] Android Studio, introduction - https://developer.android.com/studio/intro

[3] Get started with Kotlin - https://kotlinlang.org/docs/getting-started.html

[4] Gradle for Kotlin - https://kotlinlang.org/docs/gradle.html

[5] Firebase - https://firebase.google.com

[6] Introduction to Firebase Authentication - https://firebase.google.com/docs/auth

[7] Introduction to Firebase Firestore - https://firebase.google.com/docs/firestore

[8] Introduction to Firebase Cloud Storage - https://firebase.google.com/docs/storage

[9] The top-level build file - https://developer.android.com/build#top-level

[10] The module-level build file - https://developer.android.com/build#module-level

[11] Version Catalogs in Android - https://developer.android.com/build/dependencies#add-dependency

[12] Glide Github page - https://github.com/bumptech/glide

[13] Navigation in Android - https://developer.android.com/guide/navigation

[14] Introduction to Activities - https://developer.android.com/reference/android/app/Activity

[15] Introduction to Fragments - https://developer.android.com/guide/fragments

[16] What is an Intent - https://developer.android.com/reference/android/content/Intent

[17] EXtensible Markup Language, introduction - https://www.w3.org/XML/

[18] View Binding in Android - https://medium.com/@sandeepkella23/what-is-view-binding-in-android-and-how-it-works-internally-74c9ce0e5422

[19] Fast lane to Data Binding in Android - https://medium.com/mindful-engineering/data-binding-in-android-9ed92a68f9f6

[20] Recycler View, overview - https://developer.android.com/develop/ui/views/layout/recyclerview

[21] Coroutines in Kotlin - https://kotlinlang.org/docs/coroutines-basics.html

[22] Save data in a local database using Room - https://developer.android.com/training/data-storage/room

[23] Android platform architecture - https://developer.android.com/guide/platform

[24] MVVM Architecture in Android - https://www.geeksforgeeks.org/mvvm-model-view-viewmodel-architecture-pattern-in-android/

[25] Android Manifest overview - https://developer.android.com/guide/topics/manifest/manifest-intro

[26] App resources overview - https://developer.android.com/guide/topics/resources/providing-resources