Università degli studi di Padova

Dipartimento di Matematica “Tullio Levi-Civita"

Corso di laurea in Informatica

Progettazione e sviluppo di un'applicazione mobile Android con backend RESTful AWS

Laureando
Ncola Ravagnan
Matricola
2008088

Relatore
Prof. Franceso Ranzato

Anno accademico 2022/2023

Abstract

Questo documento descrive il lavoro svolto durante il periodo di stage della durata di 320 ore presso l'azienda *Zero12 s.r.l.* Lo scopo dello stage è stata la partecipazione al progetto *Smart Offices* che consisteva nello sviluppo di una piattaforma per la consultazione, ricerca, caricamento e salvataggio di luoghi dove fare *smart working*; utenti registrati inoltre possono caricare recensioni dei luoghi con valutazioni. Io mi sono occupato di sviluppare l'applicazione mobile *Android*, che comunicava con il back-end *RestFul JSON* sviluppato da altri colleghi. L'applicazione è stata sviluppata usando tecnologie native *Android*, in particolare il linguaggio *Kotlin* e il moderno toolkit di UI *Jetpack Compose*.

Inizialmente il progetto è stato preceduto da una parte di training dove si apprendevano i vari concetti fondamentali di *Kotlin*, di *Jetpack Compose* e dell'architettura di un'applicazione *Android* finendo con una breve demo per mettere in pratica ciò che è stato appreso. In seguito, si è proceduto con lo sviluppo effettivo dell'applicazione, con design dell'esperienza utente e codifica delle varie funzionalità dell'applicazione. Si è usata una metodologia di lavoro agile, con stand-up giornalieri di 15 minuti per allinearsi con il lavoro fatto.

Ringraziamenti

In primo luogo, ci tengo a ringraziare il relatore Prof. Francesco Ranzato, per avermi aiutato e sostenuto durante il mio lavoro.

Ringrazio con affetto tutta la mia famiglia, per la vicinanza che mi hanno dato in qualsiasi momento del mio periodo di studio.

Ringrazio inoltre tutti i collaboratori e i tirocinanti dell'azienda per la grande ospitalità e la disponibilità durante il periodo di tirocinio.

Infine, ringrazio tutti gli amici e le persone che ho incontrato all'università che mi hanno accompagnato in questa avventura.

Tabella dei contenuti

Lista delle tabelle

Tabella delle figure

1 Introduzione

1.1 Scopo del progetto

Il progetto *Smart Officies* è nato con l'idea di creare una piattaforma di ricerca di luoghi dove poter lavorare in remoto o in *smart working* dovuta ad una grande diffusione di questa modalità di lavoro in conseguenza al periodo pandemico del COVID-19, rimasta rilevante anche dopo la pandemia dato che molte persone hanno visto un migliore bilancio vita-lavoro e le aziende hanno visto un risparmio nei costi e un aumento nella produttività.

Il progetto quindi si propone di creare una piattaforma che permette di trovare luoghi dove poter lavorare in remoto: per esempio bar, ristoranti, biblioteche, ecc. che offrono la possibilità di lavorare in remoto; La piattaforma offre funzionalità di ricerca e filtri per trovare il luogo più adatto alle proprie esigenze. Un utente può inoltre registrarsi e gli utenti registrati, oltre a consultare i luoghi, possono aggiungere nuovi luoghi, lasciare recensioni sui luoghi e salvare i luoghi nei preferiti. Il progetto nell'insieme è suddiviso in tre parti:

- Parte **back-end** che si occupa di esporre i servizi per la ricerca luoghi, il caricamento dei luoghi, dei utenti e delle recensioni; fornisce i dati all'applicazione tramite API REST, è stato sviluppato usando servizi e tecnologie *AWS* ed è scritto in *TypeScript* [2];

Figura 2: Logo di *Amazon Web Services.*

- Parte **mobile** che si occupa di mostrare i dati forniti dal back-end all'utente su dispositivi mobili: questa si divide in due applicazioni, una scritta in *Kotlin* [3] per *Android* e l'altra in *Swift* [4] per *iOS*; la parte *iOS*, sviluppata da un altro collega, utilizza il framework di UI *SwiftUI* mentre la parte Android utilizza il framework di UI *Jetpack Compose* [5].

Figura 3: Logo di *Jetpack Compose.*

Il mio progetto di stage si è concentrato sull'applicazione scritta in **Kotlin** per **Android**. Il mio scopo era quello di effettuare le chiamate dal back-end remoto ed implementare le funzionalità dell'applicazione, cioè la ricerca dei luoghi, la visualizzazione dei dettagli di un luogo, il caricamento di un nuovo luogo, l'autenticazione utente, la registrazione di un nuovo account, la pagina del profilo utente, la visualizzazione delle recensioni di un luogo e la creazione di nuove recensioni, il tutto utilizzando le best practices di un'architettura che separa la logica di business dalla logica di presentazione.

Il progetto è stato preceduto da un breve periodo di formazione tecnica seguito da una breve demo.

Il progetto è stato accompagnato da una metodologia di lavoro agile [6], con stand-up giornalieri di 15 minuti dove con il tutor aziendale si discuteva del lavoro fatto e del lavoro da fare.

1.2 L'azienda

ZERO12
a vargroup company

Figura 4: Logo di *Zero12 s.r.l.*

Zero12 s.r.l è una software house che propone prodotti innovativi e servizi di consulenza per la trasformazione digitale, è parte del gruppo *Vargroup*. L'azienda offre principalmente prodotti basati su *Amazon Web Services*, come infrastrutture cloud, software web, mobile e intelligenza artificiale. Altri campi di interesse sono la *Augmented Reality* e l'*Internet of Things*.

L'azienda è in continua crescità e si divide in due sedi, una a Padova e una a Empoli.

1.3 Motivazione della scelta

La prima volta che ho conosciuto l'azienda è stata durante il primo periodo del corso di ingegneria del software quando il mio gruppo doveva scegliere il capitolato per il progetto, più tardi mi sono presentato all'evento **Stage IT 2023** dove sono andato a conoscenza dei loro progetti.

Ho scelto questo progetto perché volevo cimentarmi in qualcosa di nuovo ma allo stesso tempo cercavo uno stage legato allo sviluppo mobile o web; quindi, ciò che mi ha spinto a scegliere questo progetto è stata la possibilità di lavorare con tecnologie mobile *Android* e la possibilità di lavorare con un servizio remoto.

2 Il progetto

In questo capitolo viene descritto il progetto di stage in largo, in particolare come è stato pianificato, le tecnologie usate per realizzare il prodotto, gli strumenti utilizzati per lo sviluppo, gli strumenti utilizzati per la gestione del lavoro e in breve cosa si è fatto nel periodo di formazione.

2.1 Pianificazione

Prima dell'inizio del tirocinio è stato pianificato un piano di lavoro organizzando il programma di lavoro in 5 periodi, ognuno di due settimane a eccezione del primo e del ultimo periodo che durano una sola settimana, con un obiettivo da raggiungere per ogni periodo.

2.1.1 *Vincoli temporali*

Il tirocinio aveva una durata prevista di 8 settimane con 40 ore per settimana, con orario di lavoro dalle 8:30 alle 17:30 con un'ora di pausa pranzo dalle ore 13:00 alle 14:00 e altre due pause intermedie, una alla mattina dalle 10:30 alle 11:00 e l'altra dalle 15:30 alle 16:00, per un totale di 8 ore giornaliere.

In seguito, il programma di lavoro inizialmente pianificato per il tirocinio:

Periodo	**Descrizione**
Periodo 1 (1 settimana)	Introduzione al linguaggio di programmazione *Kotlin* e modalità di deploy automatico in ambito mobile
Periodo 2 (2 settimane)	Progettazione e sviluppo sezione dell'applicazione relativa a schermate di registrazione, login e recupero password
Periodo 3 (2 settimane)	Progettazione e sviluppo sezione dell'applicazione relativa alla gestione profilo e dati personali e per la ricerca di location per fare smart working
Periodo 4 (2 settimane)	Progettazione e sviluppo sezione dell'applicazione relativa alla segnalazione location di smart working e le funzioni per la recensire location e/o segnalatore
Periodo 5 (1 settimana)	Testing finale e scrittura documentazione di quanto sviluppato

Tabella 1: Programma di lavoro pianificato.

2.1.2 *Obiettivi formativi*

Nel piano di lavoro sono stati identificati i seguenti obiettivi formativi:

- **O-1**: Approfondire le tematiche di sviluppo mobile;
- **O-2**: Approfondire le tematiche di interazione con servizi esterni;
- **O-3**: Apprendere metodologie di lavoro agile.

2.1.3 *Analisi dei rischi*

I possibili rischi individuati nel piano di lavoro sono stati i seguenti:

- **R-1**: Difficoltà apprendimento del linguaggio di programmazione e sviluppo mobile;
- **R-2**: Difficoltà apprendimento API back-end;
- **R-3**: Difficoltà nello sviluppo applicazione mobile;
- **R-4**: Difficoltà nello sviluppo e gestione del deploy attraverso sistemi di CI/CD in ambito mobile.

2.1.4 *Prodotti aspettati*

In seguito, vengono elencati i prodotti inizialmente pianificati nel piano di lavoro:

- **P-1**: Sviluppare applicazione mobile funzionante;
- **P-2**: Sviluppare test automatici;
- **P-3**: Documentazione dell'intero progetto di stage;
- **P-4**: Articolo per il blog di *Zero12* dove raccontare l'esperienza vissuta in azienda.

2.2 Tecnologie utilizzate

Le seguenti tecnologie sono state usate per lo sviluppo del prodotto:

- **Kotlin** [3]: linguaggio di programmazione *general purpose*, multi-paradigma, svillupato da *JetBrains* e utilizzato per lo sviluppo dell'applicazione *Android* e per la scrittura dei test automatici;
- **Gradle** [7]: sistema di automazione dello sviluppo basato su *Apache Ant* e *Maven*, utilizzato per la compilazione del codice sorgente e la gestione delle dipendenze; rispetto a *Maven*, *Gradle* usa un DSL basato su *Groovy* invece di XML;

- **Jetpack Compose** [5]: moderno toolkit di UI per *Android* per lo sviluppo di UI native in *Kotlin*, utilizzato per lo sviluppo dell'interfaccia grafica dell'applicazione *Android*. Permette di implementare interfacce grafiche usando un linguaggio conciso, scrivendo meno righe di codice ed è compatibile con codice *Android* già esistente;
- **Material Design 3** [8]: sistema di design open-source sviluppato da *Google*, definisce le componenti e le linee guida dell'interfaccia grafica supportando le migliori best practices; per il progetto è stato usato con *Jetpack Compose*;
- **Hilt** [9]: libreria di *Dependency injection* per *Android*, sviluppata al di sopra della libreria *Dagger* e utilizzata per implementare il pattern di *Dependency injection* nell'applicazione *Android*;
- **Ktor** [10]: framework per la creazione di client e server web asincroni; per il progetto ho utilizzato il client di *Ktor* per fare chiamate al back-end remoto;
- **Room** [11]: una libreria di persistenza locale che fornisce un layer d'astrazione per SQLite, un database relazionale; utilizzato per la creazione e la gestione del database locale;
- **AWS Amplify** [12]: framework di sviluppo per applicazioni web e mobili di *Amazon Web Services*, utilizzato per configurare *AWS Cognito*;
- **AWS Cognito** [13]: servizio fornito da *Amazon Web Services* utilizzato per la registrazione e l'autenticazione degli utenti all'interno dell'applicazione; permette di gestire gli utenti e le loro autorizzazioni.

2.3 Strumenti di sviluppo

2.3.1 *Sistema operativo*

Per l'intera durata dello stage ho utilizzato il notebook fornito dall'azienda, cioè un *MacBook Pro 2017* con processore *Intel* con installato *macOS Ventura 13.4*

2.3.2 *Android Studio*

Per la codifica del prodotto mi sono servito di *Android Studio* [14], l'ambiente di sviluppo integrato gratuito fornito da *JetBrains*, essendo ufficiale per lo sviluppo di applicazioni *Android* semplifica il processo di build dell'applicazione con *Gradle* e nell'installazione è incluso un emulatore *Android*. Per il debugging e l'installazione dell'applicazione su dispositivi *Android* mi sono servito del tool *Android Debug Bridge* (o *ADB*).

2.3.3 *Altri strumenti*

Per il versionamento del codice sorgente ho utilizzato *Git*, un sistema di controllo versione distribuito *open source*, e *Bitbucket* come repository del codice remota. Durante lo stage ho utilizzato anche *Fork*, un'interfaccia grafica per *Git*, e *Postman*, un client API che ho usato per testare le API del back-end.

2.4 Strumenti organizzativi

Per la gestione del lavoro è stata utilizzata la suite di *Atlassian*:

- **Jira**: software di *Issue tracking system* che consente il bug tracking e la gestione dei progetti sviluppati con metodologie agili. Durante lo stage è stata utilizzata la *Scrum* board per il tracciamento del lavoro contrassegnando il lavoro fatto e visualizzando i prossimi lavori da fare;

- **Bitbucket**: servizio di hosting remoto del repository del codice sorgente basato su *Git*. Per lo stage è stato utilizzato come repository remoto del codice sorgente ed è stato utilizzato per le revisioni del codice scritto tramite *Pull Request*.

Per la comunicazione e la collaborazione sono stati utilizzati i seguenti strumenti:

- **Slack**: software di collaborazione aziendale sviluppato da *Slack technologies*, utilizzato dall'azienda per comunicazioni rapide, con possibilità di creare canali all'interno dello stesso server ed effettuare incontri. Per lo stage è stato creato un canale condiviso con altri collaboratori e tirocinanti per comunicare e condividere informazioni;
- **Google Meet**: applicazione per effettuare teleconferenze sviluppata da *Google*, usata in azienda anche nel corso del tirocinio per effettuare gli stand-up giornalieri.

2.5 Formazione

Nella prima fase del tirocinio si è concentrato sulla formazione tecnica necessaria, utilizzando risorse online come il corso ufficiale **Android Basics with Compose** e la documentazione di *Google*.

Ho cominciato lo studio delle tecnologie con il linguaggio di programmazione **Kotlin**, studiandone la sintassi e i costrutti, in seguito sono passato a studiare il framework di UI *Jetpack Compose* e i fondamenti dell'architettura di un'applicazione *Android*, quindi l'utilizzo del pattern Model-view-ViewModel in un'applicazione *Android*, l'architettura a tre strati (UI, dominio, data), il *Dependency injection* tramite la libreria *Hilt* e l'utilizzo della libreria *Room*.

Il periodo di formazione poi finì con una breve demo tramite lo sviluppo controllato di una app, con poche funzionalità utili ma strutturata. Durante la demo, ogni funzionalità creata passava sotto revisione tramite **Pull Request** su *Bitbucket*.

3 Analisi dei Requisiti

I requisiti dell'applicazione sono stati individuati attraverso un'analisi dei casi d'uso e dei requisiti funzionali insieme al tutor aziendale e ad altri collaboratori durante il corso del progetto.

All'inizio è stato pianificato che l'applicazione doveva avere le seguenti funzionalità:

- **F1**: Lista dei luoghi (con funzione di ordinamento e filtraggio);
- **F2**: Mappa dei luoghi (con funzione di filtraggio);
- **F3**: Visualizzazione in dettaglio di un luogo;
- **F4**: Caricamento di un luogo;
- **F5**: Pagina di login;
- **F6**: Pagina del profilo utente, con lista dei luoghi caricati e dei luoghi salvati;
- **F7**: Registrazione di un nuovo account;
- **F8**: Lista delle recensioni di un luogo;
- **F9**: Caricamento di una nuova recensione di un luogo.

3.1 Tipi di utenti

I tipi di utenti che utilizzano l'applicazione sono:

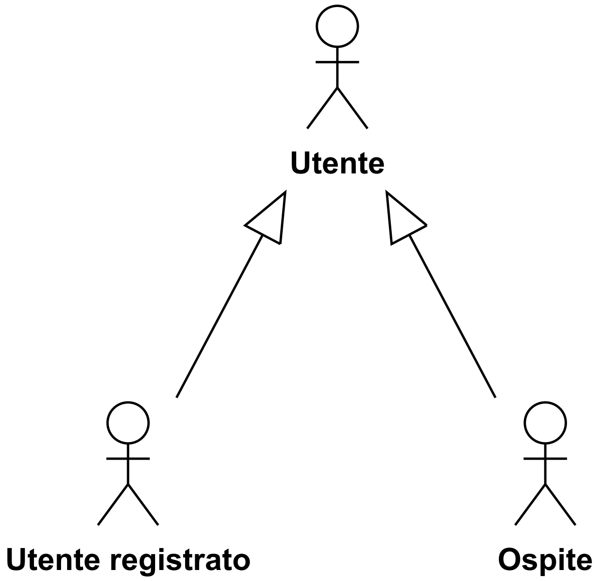


Figura 5: Gerarchia dei tipi di utente.

- **Utente**: un utente generico, registrato o non che utilizza l'applicazione per visualizzare la lista dei luoghi, le informazioni in dettaglio dei luoghi, la mappa dei luoghi e la lista delle recensioni di un luogo;
- **Ospite**: un utente che non ha effettuato l'accesso al suo account e che oltre ad aver accesso alle funzionalità dell'utente generico, può accedere al proprio account o registrarsi;
- **Utente registrato**: un utente che ha effettuato l'accesso al suo account e che può quindi accedere alle funzionalità riservate agli utenti registrati, come caricare un nuovo luogo, salvare un luogo nei preferiti o caricare una nuova recensione.

3.2 Requisiti

Dalle funzionalità riportate sopra sono stati ricavati dei requisiti associati (per la lista completa dei requisiti, vedere *lista requisiti* nell'appendice).

Alcuni requisiti sono:

- Per la **lista dei luoghi** l'utente selezionando un luogo viene portato alla pagina di dettaglio del luogo e un utente vuole ordinare la lista dei luoghi per distanza, valutazione o data d'inserimento;
- Per la **mappa dei luoghi**, l'utente selezionando un luogo viene portato alla pagina di dettaglio del luogo similmente alla lista dei luoghi;
- Per la **lista dei luoghi** e la **mappa dei luoghi** un utente può filtrare i luoghi per nome, prezzo, caratteristiche o orario di apertura;
- Per la **visualizzazione in dettaglio di un luogo** dovrebbero essere visualizzati i dettagli del luogo, tra cui nome, indirizzo, descrizione, orari di apertura e chiusura, contatti, prezzo e una o più immagini;
- Per la **lista dei luoghi**, la **mappa dei luoghi** e la **visualizzazione in dettaglio di un luogo** un utente registrato può salvare un luogo nei preferiti o rimuovere un luogo dai preferiti;

- Per il **caricamento di un luogo** l'utente registrato deve inserire i dati richiesti per inserire il luogo, tra i quali: nome, indirizzo, descrizione, orari di apertura e chiusura, contatti, prezzo e una o più immagini;
- Per la **pagina di login** un ospite può effettuare l'accesso con un account interno o con un account *Google*;
- Per la **pagina del profilo utente** l'utente registrato può visualizzare la lista dei luoghi caricati, la lista dei luoghi salvati o effettuare il logout;
- Per la **registrazione di un nuovo account** l'ospite deve inserire e-mail e password;
- Per il **caricamento di una recensione** l'utente registrato deve inserire il testo della recensione e una valutazione.

Nell'analisi dei requisiti sono stati individuati requisiti che sono condivisi tra più funzionalità, come per esempio il filtraggio dei luoghi che è presente sia nella lista che nella mappa e il salvataggio di un luogo nei preferiti che è presente nella lista, nella mappa e nel dettaglio del luogo.

Oltre ai requisiti funzionali, ci sono dei **requisiti di vincolo** che sono stati individuati durante l'analisi dei requisiti, come l'utilizzo del linguaggio di programmazione *Kotlin* e del toolkit UI *Jetpack Compose* per lo sviluppo dell'applicazione.

Per garantire la qualità del codice e dell'applicazione, sono stati individuati dei **requisiti di qualità**: il codice del progetto deve trovarsi sul repository aziendale, deve passare tutte le *Pull requests* e l'applicazione deve essere funzionante anche in assenza di connessione ad Internet.

Tipo di requisito	**Numero**
Funzionali	49
Di vincolo	4
Di qualità	3
Totali	56

Tabella 2: Conteggio requisiti dell'applicazione *Android*.

4 Progettazione

Qui verrà spiegata la progettazione dell'applicazione, in particolare verrà spiegata l'architettura dell'applicazione e le scelte progettuali fatte, in sintesi ho deciso di utilizzare un'architettura a tre livelli e il pattern *MVVM* (*Model-View-ViewModel*).

Per quanto riguarda il design dell'interfaccia grafica, sono partito da un mock-up già fatto del risultato finale come punto di partenza, modificando alcuni componenti per adattarli meglio al design di *Android* e per renderli più semplici da interagire.

In generale per la struttura architetturale ho cercato di seguire le best practices dell'architettura moderna di un'applicazione *Android* quindi ho deciso di utilizzare un'architettura a tre livelli (o *layers*), composta dai seguenti livelli:

- Il **livello UI** (*UI Layer*): contiene l'interfaccia grafica che mostra i dati all'utente e permette di interagire con l'applicazione.
- Il **livello dati** (*Data Layer*): contiene i dati dell'applicazione, la logica di business su di essi e le sorgenti dati dell'applicazione.

- Il **livello dominio** (*Domain Layer*): è un livello facoltativo situato tra il livello UI e il livello dati ed è utilizzato per semplificare o riutilizzare codice.

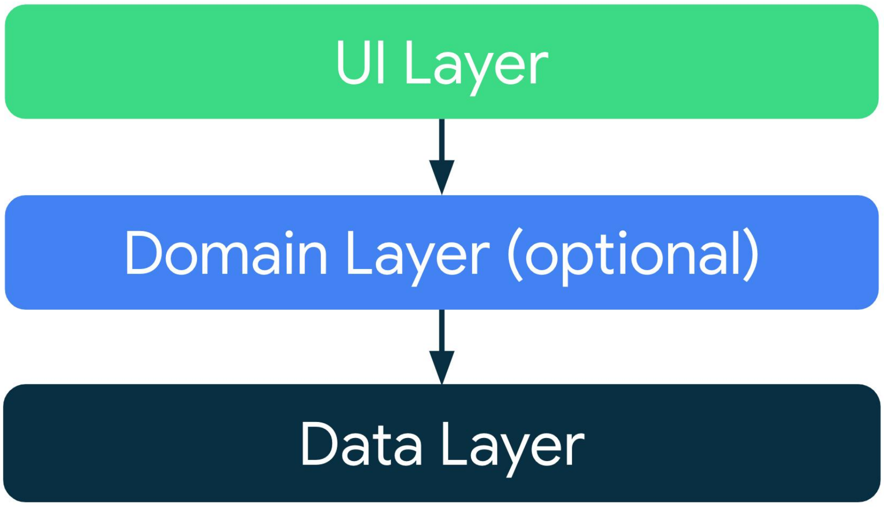


Figura 6: Schema architettura a livelli, da *Guide to app architecture* [1].

L'utilizzo di questa architettura porta un codice mantenibile, testabile, scalabile e con una buona separazione delle responsabilità tra i vari livelli.

4.1 Livello UI

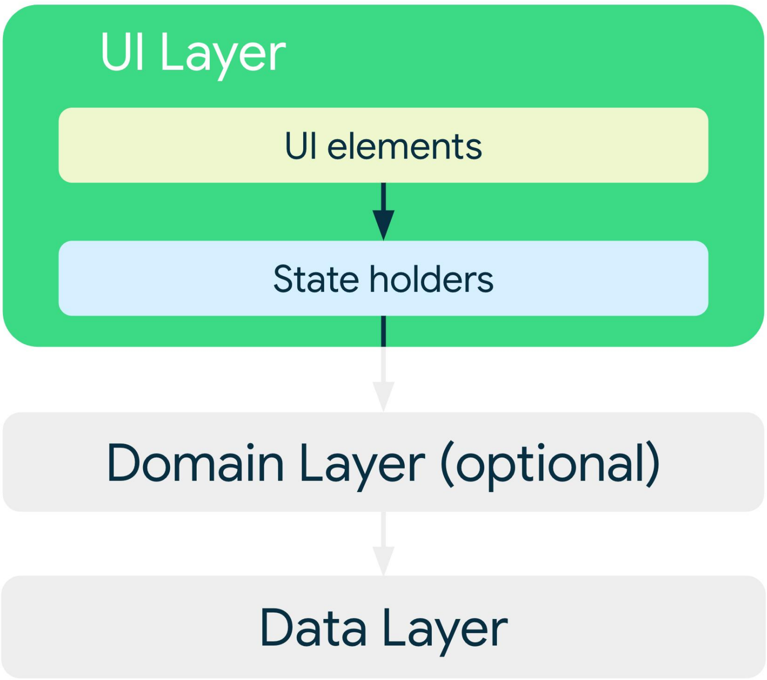


Figura 7: Schema livello UI, da *Guide to app architecture* [1].

Il **livello UI** (o **livello di presentazione**) si occupa di presentare i dati all'utente e di reagire all'input dell'utente. Ad ogni interazione utente, l'interfaccia utente si aggiorna per riflettere le modifiche.

L'interfaccia grafica è rappresentata dai *Composable* di *Jetpack Compose*, cioè delle funzioni *Kotlin* che rappresentano parti dell'interfaccia grafica come superfici, pulsanti, immagini, o elementi di layout come colonne, righe e griglie: ogni schermata dell'applicazione è un insieme di *Composable* innestati.

Ogni schermata ha un *ViewModel* associato che si occupa di gestire la logica di presentazione della schermata e rappresenta il titolare dello stato dell'interfaccia grafica: contiene i dati da mostrare all'interfaccia grafica e scambia i dati con gli altri livelli. Per esempio, per la lista dei luoghi ci sarà un *ViewModel* associato che carica la lista dei luoghi dalla sorgente dati e li presenta a schermo, inoltre viene gestito l'input utente come la selezione di un filtro per la lista.

4.2 Livello dati

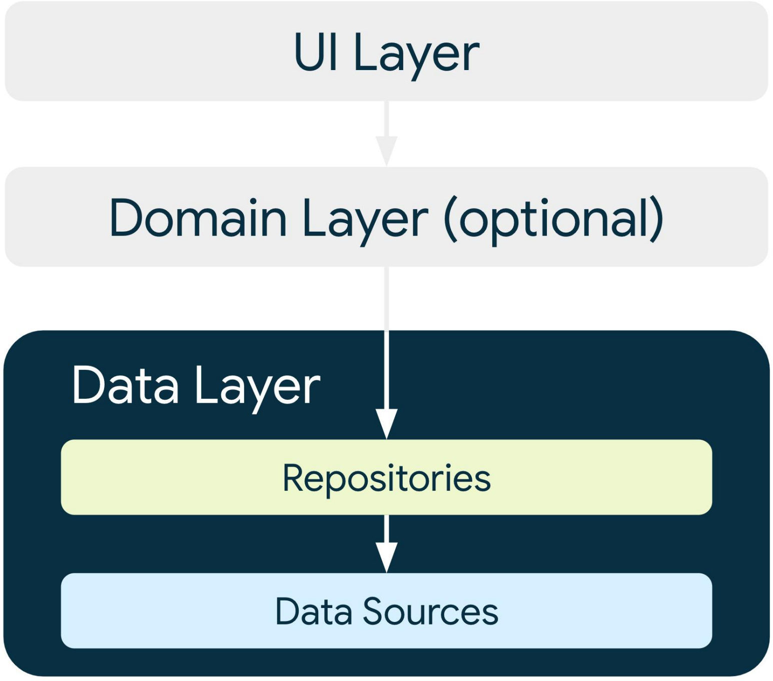


Figura 8: Schema livello dati, da *Guide to app architecture* [1].

Il **livello dati** di un'applicazione *Android* contiene la logica di business e si occupa ci come l'applicazione crea, archivia e modifica i dati.

Per il livello dati ho utilizzato il pattern repository, i **repository** si occupano di esporre i dati agli altri livelli, senza che essi sappiano da dove vengono questi dati, e si occupano della logica di business sui dati. Nel progetto ho creato un repository per ogni tipo di risorsa (luogo, utente …) e questi repository scambiano dati con due sorgenti dati:

- Il **database locale**, cioè il database *Room*, è la fonte di riferimento dei dati in quanto è la più veloce da accedere e permette di avere i dati disponibili anche offline. Il database locale è composto da una classe di database che crea il database locale se non esiste già e, attraverso una classe DAO (*Data Access Object*), permette di prelevare o modificare i dati da esso; la classe DAO consente l'accesso alle tabelle del database locale. Il database locale è relazionale, per vederne la struttura consultare il diagramma ER nell'*appendice*, in sintesi ho cercato di avere una struttura simile alle risposte JSON del back-end remoto;
- Il **back-end remoto**, cioè il servizio API REST che fornisce i dati. Il back-end remoto è la sorgente secondaria dei dati, in quanto richiede una connessione a Internet, ma permette di avere i dati sempre aggiornati. Una classe di servizio API viene creata per ogni risorsa.

Nel progetto quando l'applicazione carica i dati, prima carica i dati dalla fonte di riferimento e poi sincronizza i dati del database locale con il back-end remoto, salvando i dati nuovi nel database e aggiornando quelli già presenti. In questo modo, l'applicazione può mostrare i dati anche offline e può mostrare i dati più aggiornati quando è connessa ad Internet: questo approccio è detto sincronizzazione basata sul pull ed è quello che ho implementato nel progetto.

4.3 Livello di dominio

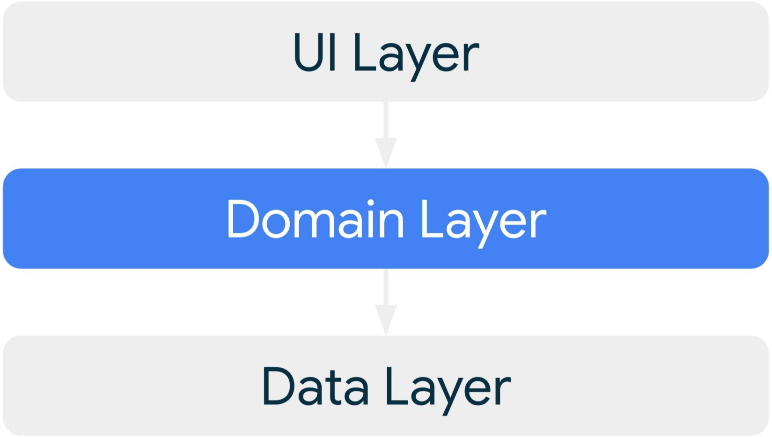


Figura 9: Schema livello di dominio, da *Guide to app architecture* [1].

Il **livello di dominio** è un livello posto in mezzo tra il livello dati e il livello di presentazione, è responsabile dell'incapsulamento di una logica di business più complessa o di logica che viene riutilizzata su più *ViewModel*.

In questo livello si trovano le classi **UseCase** che sono responsabili ciascuna di una singola funzionalità (si occupano di un solo *caso d'uso*).

I *ViewModel* possono utilizzare le classi **UseCase** per ottenere dati complessi che necessitano di combinare dati da più repository o per ottenere dati che necessitano di una elaborazione più complessa.

Un esempio può essere l'ottenimento dei dettagli di un luogo con l'autore associato, in questo caso viene creata una classe *UseCase* che prende i dati del luogo dal repository dei luoghi e i dati dell'autore dal repository degli utenti e li combina in un unico oggetto che viene mandato al *ViewModel*.

Questo livello non è necessario se il progetto è abbastanza piccolo, a volte è possibile utilizzare direttamente i repository nei *ViewModel*, nel progetto è stato usato più a scopo educativo.

4.4 Dependency injection

Il pattern **Dependency Injection** (In breve **DI**) è un pattern che permette di semplificare la gestione delle dipendenze tra le classi. In questo modo, invece di creare le dipendenze all'interno delle classi, le dipendenze vengono create all'esterno e poi passate alle classi che ne necessitano. Questo permette di semplificare la gestione delle dipendenze e di rendere il codice più testabile.

Nel progetto è stato utilizzato il framework *Hilt* per implementare il pattern *DI*. Per esempio, è stato utilizzato per iniettare le classi *UseCase* e i repository nei *ViewModel* e le sorgenti dati nei repository.

5 Codifica

In questo capitolo vengono descritte come le varie parti dell'architettura sono state codificate nel progetto e il funzionamento delle singole schermate dell'applicazione.

5.1 Interfaccia grafica

Durante la codifica sono stato in grado di utilizzare gli strumenti offerti dal toolkit UI **Jetpack Compose** per creare l'interfaccia grafica dell'applicazione, in particolare in *Compose* ogni elemento dell'interfaccia grafica è rappresentato da un **Composable**, cioè una funzione *Kotlin* annotata con l'annotazione `@Composable` che consente anche di codificare nuovi *Composable*; questi componenti rappresentano elementi dell'interfaccia grafica, come per esempio pulsanti, testi o immagini, oppure elementi di layout come colonne o righe.

Per lo stile dei *Composable* ho usato *Material Design 3* dato che *Jetpack Compose* fornisce un'implementazione immediata di *Material Design 3* e dei suoi elementi UI; lo stile dei *Composable* può essere cambiato passando dei parametri alle funzioni *Composable* (per esempio, font del testo, forma o colore).

Un modo per modificare l'aspetto e il comportamento di un *Composable* è passargli un `Modifier`, utile per modificare le dimensioni, il layout, il comportamento, l'aspetto e per elaborare l'input utente.

Le destinazioni dei schermi sono state implementate usando una libreria esterna che permette di definire le destinazioni tramite funzioni annotate con l'annotazione `@Destination` (vedi *Destinazione di navigazione* nell'appendice) e sono usate per navigare tra le schermate dell'applicazione: per esempio quando si vuole passare dalla schermata della lista dei luoghi alla schermata del dettaglio di un luogo, si usa la funzione `navigate` che, quando un luogo viene selezionato dalla lista, prende come parametro l'identificatore del luogo selezionato. In queste destinazioni viene estratto lo stato dell'interfaccia dal *ViewModel* e lo viene fatto passare alla schermata di destinazione, che sarà un *Composable*, inoltre vengono impostate le funzioni del *ViewModel* da invocare in risposta del input utente.

Il punto d'inizio dell'applicazione è la classe `MainActivity` (vedi *MainActivity* nell'appendice), una classe che estende `ComponentActivity` e possiede un ciclo di vita associato al ciclo di vita dell'applicazione (per esempio quando si fa partire l'applicazione o viene messa in background). Nel progetto ho dovuto semplicemente implementare la funzione `onCreate` che viene chiamata quando viene creato il processo dell'applicazione e redirige l'utente verso la destinazione iniziale, in questo caso la lista dei luoghi.

5.2 Classi ViewModel

Per le classi **ViewModel** ho creato delle classi che ereditano dalla classe *ViewModel* del framework di *Android* (vedi *Classi ViewModel* nell'appendice), una classe *ViewModel* per schermata. Il ciclo di vita di queste classi è gestito dal framework stesso e una caratteristica delle classi *ViewModel* è che rappresentano dei titolari di stato della schermata a cui sono associati: lo stato della schermata, infatti, viene codificato in un oggetto che ogni volta che subisce una modifica, aggiorna la schermata riflettendo la modifica. Usando *Hilt*, sono stato in grado di iniettare le dipendenze all'interno dei *ViewModel*, che possono essere repository o classi *UseCase* se richiesto: per esempio il *ViewModel* della schermata di visualizzazione del dettaglio di un luogo ha bisogno del repository dei luoghi e del repository degli utenti, allora viene creata una classe *UseCase* (vedi *Classi UseCase* nell'appendice) che contiene i due repository iniettati (tramite *Hilt*): la caratteristica principale delle classi *UseCase* è che contengono una sola funzione che viene poi invocata nel *ViewModel*.

5.3 Sorgenti dati

Per la gestione e la persistenza dei dati come già scritto l'applicazione usufruisce di due sorgenti dati, quella locale e quella remota:

- Il **database locale**, la fonte di riferimento principale implementata utilizzando *Room*: innanzitutto ho creato una classe di database (*Classe Database* nell'appendice) che crea il database, se non esiste già, ed offre dei metodi per ottenere i DAO (Vedi *Interfacce DAO* nell'appendice) per ogni tipo di risorsa dati (luoghi, utenti …), ogni DAO offre dei metodi per ottenere, inserire, aggiornare e cancellare elementi del database; ogni metodo dei DAO è annotato con l'annotazione `@Insert` o `@Update` o `@Delete` o `@Query`: queste annotazioni sono usate da *Room* per generare il codice necessario per eseguire le query al database. I dati vengono salvati e ritornati come oggetti immutabili.
- La comunicazione alla sorgente dati **remota** viene effettuata tramite *Ktor*: per ogni tipo di risorsa dati (luoghi, utenti …) ho creato una classe che si occupa di caricare e di ottenere dati per quella risorsa (Vedi *Classi API* nell'appendice), le risposte sono in JSON e vengono deserializzate in oggetti *Kotlin* usando la libreria **Kotlinx.serialization**, libreria che viene usata anche per serializzare oggetti *Kotlin* in JSON per l'invio di dati. Le risposte possono essere:
  - `200`: la risposta contiene i dati richiesti;
  - `201`: L'utente ha creato nuovi dati con successo;
  - `422`: la risposta contiene un messaggio di errore a causa di un errore di validazione dei dati, l'utente deve correggere i dati e riprovare;
  - `404`: la risposta contiene un messaggio di errore da parte del client, può essere causato da un errore di connessione.

Entrambe le sorgenti dati vengono iniettate nei repository tramite *Hilt*: per esempio nel repository dei luoghi ho iniettato il DAO dei luoghi, che consente di accedere ai luoghi nel database locale, e la classe API, che si occupa di caricare i luoghi dalla sorgente dati remota.

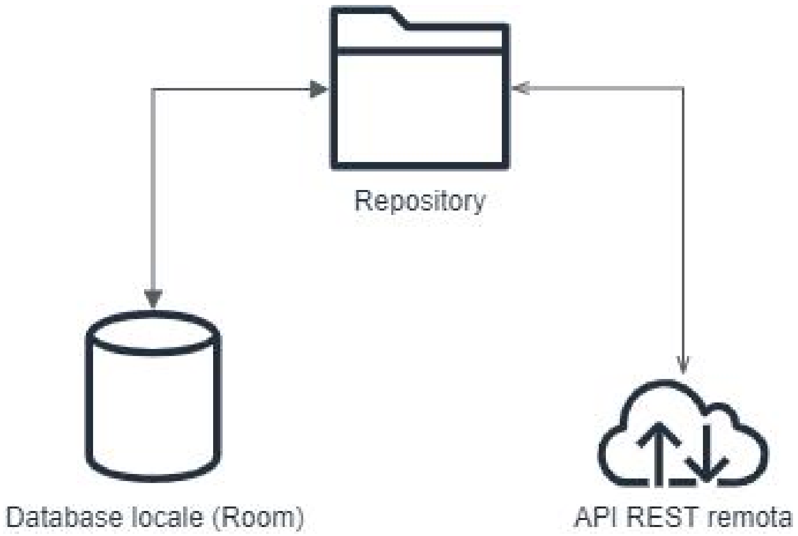


Figura 10: Relazione tra repository e sorgenti dati.

5.4 Autenticazione utenti

Per quanto riguarda l'autenticazione utenti mi sono servito di **AWS Cognito**: quando un utente accede con il suo account, gli viene conferito un token segreto con il quale è in grado di accedere al suo profilo, di caricare nuovi luoghi o recensioni e di salvare nuovi luoghi nel proprio account; questo token viene allegato nel header della richiesta HTTP, se non è un token valido allora la richiesta ritorna `401 unauthorized`.

5.5 Schermate applicazione

In seguito, descriverò più in dettaglio il funzionamento delle singole schermate dell'applicazione.

5.5.1 *Schermata iniziale: lista dei luoghi*

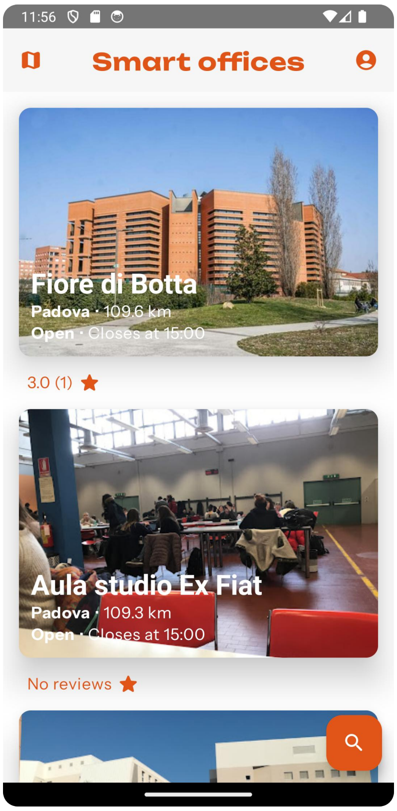


Figura 11: Schermata della lista dei luoghi.

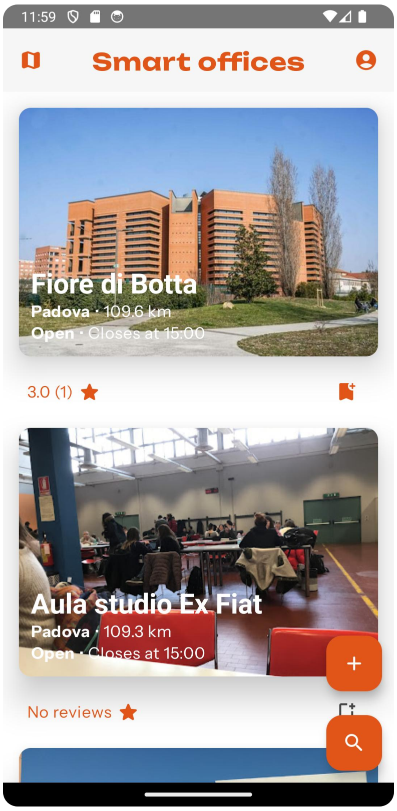


Figura 12: Schermata della lista dei luoghi con utente registrato.

La schermata principale che viene presentata non appena l'utente apre l'applicazione, viene mostrata la lista dei luoghi più vicini all'utente. Ogni elemento della lista mostra:

- Il nome del luogo;
- La città nel quale si trova;
- La distanza del luogo dall'utente;
- Se il luogo è aperto o no;
- L'orario di chiusura, se è aperto, o di apertura, se chiuso;
- La valutazione media con numero di recensioni, se ci sono;
- Per gli utenti registrati, un segnalibro per salvare i luoghi nei preferiti toccandolo.

Toccando un luogo nella lista si accede alla schermata dei dettagli del luogo selezionato. In alto a sinistra della schermata si può accedere alla mappa dei luoghi, mentre in alto a destra si può accedere al proprio profilo, se si è già registrato, o un ospite può effettuare l'accesso. In basso a destra si trova il pulsante di ricerca e filtraggio luoghi e, per gli utenti registrati, è possibile aggiungere un nuovo luogo, tramite il pulsante in basso a destra con l'icona *+*, sopra al pulsante di ricerca e filtraggio luoghi.

Questa schermata ha associato un *ViewModel* che tramite un *UseCase* si occupa di prelevare la lista dei luoghi dal database locale e se è disponibile la connessione ad Internet i dati saranno aggiornati con quelli remoti: questo *UseCase* chiama un metodo del repository dei luoghi per ottenere la lista dei luoghi. Per salvare un luogo viene invocato un *UseCase* che si occupa di salvare un luogo sull'account personale o rimuovere un luogo se già salvato nei preferiti, come se fosse un *toggle*. Altri metodi del *ViewModel* si occupano di applicare i filtri e l'ordinamento alla lista dei luoghi, richiamando i dati dal repository dei luoghi con i parametri di filtraggio e ordinamento.

5.5.2 *Mappa dei luoghi*

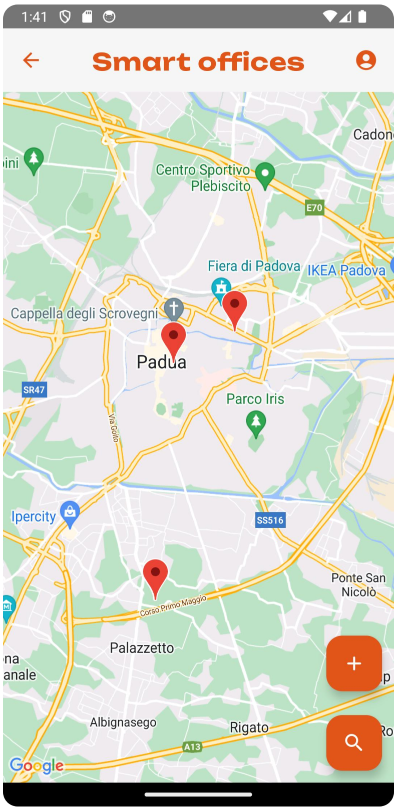
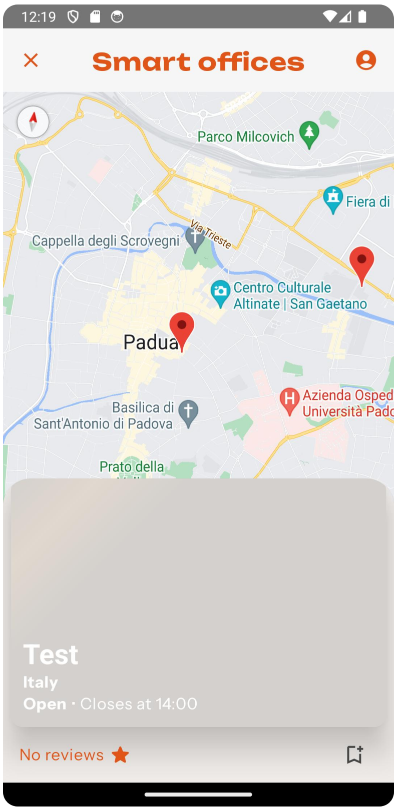


Figura 13: Schermata della mappa dei luoghi.


12:19
Smart offices
Parco Milcovich
Fiera di
Cappella degli Scrovegni
Via Trieste
Centro Culturale Altinate | San Gaetano
Padua
Basilica di Sant'Antonio di Padova
Azienda Osped Università Pado
Prato della

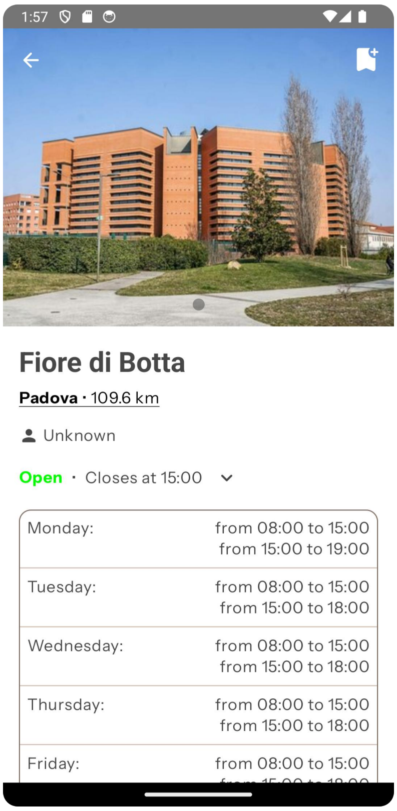
Test
Italy
Open · Closes at 14:00
No reviews

Figura 14: Schermata della mappa dei luoghi selezionato un segnaposto.

Dalla lista dei luoghi si può accedere alla mappa dei luoghi, dove viene mostrata una mappa globale con i luoghi rappresentati come segnaposto: questa mappa è una mappa *Google Maps* e selezionando un segnaposto viene mostrata la scheda del luogo, che mostra le stesse informazioni della lista. Se si seleziona la scheda del luogo si accede alla schermata dei dettagli del luogo selezionato mentre selezionando il segnalibro un utente registrato può salvare il luogo (o rimuovere un luogo salvato se già salvato). In alto a sinistra della schermata si può tornare alla lista dei luoghi, mentre in alto a destra si può accedere al proprio profilo, se si è già registrato, o un ospite può effettuare l'accesso. In basso a destra si trovano i medesimi pulsanti che si trovano nella lista dei luoghi: le funzioni di ricerca e filtraggio sono disponibili pure nella mappa dei luoghi (eccetto l'ordinamento) e un utente registrato può caricare un nuovo luogo.

Simile alla lista dei luoghi, questa schermata ha un *ViewModel* che chiama lo stesso metodo del repository dei luoghi per ottenere la lista dei luoghi, poi di ogni luogo della lista vengono presi gli attributi di posizione geografica e vengono creati i segnaposto nella mappa. Altri metodi del *ViewModel* si occupano di mostrare un luogo selezionato un segnaposto nella mappa e applicare i filtri alla mappa dei luoghi, richiamando i dati dal repository dei luoghi con i parametri di filtraggio. Per salvare un luogo nei preferiti funziona in modo analogo alla lista dei luoghi.

5.5.3 *Dettaglio di un luogo*



Fiore di Botta

Padova · 109.6 km

Unknown

Open · Closes at 15:00

Monday:	from 08:00 to 15:00
	from 15:00 to 19:00

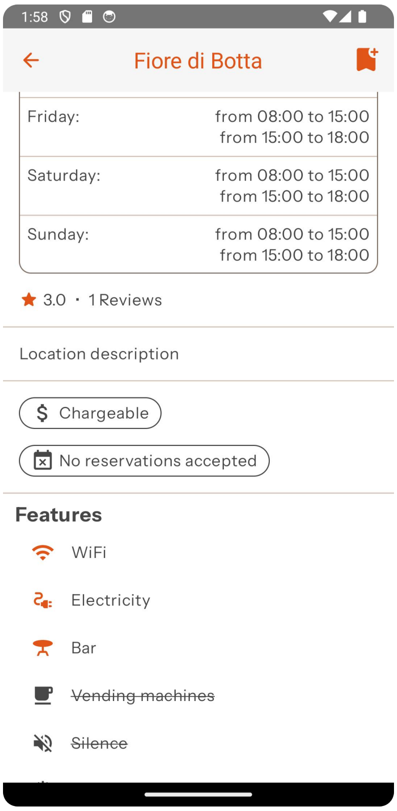
Tuesday:	from 08:00 to 15:00
	from 15:00 to 18:00

Wednesday:	from 08:00 to 15:00
	from 15:00 to 18:00

Thursday:	from 08:00 to 15:00
	from 15:00 to 18:00

Friday:	from 08:00 to 15:00

Figura 15: Schermata del dettaglio di un luogo: nome, dove si trova, orari di apertura.



Friday:	from 08:00 to 15:00
	from 15:00 to 18:00

Saturday:	from 08:00 to 15:00
	from 15:00 to 18:00

Sunday:	from 08:00 to 15:00
	from 15:00 to 18:00

★ 3.0 · 1 Reviews

Location description

$ Chargeable

No reservations accepted

Features

WiFi

Electricity

Bar

~~Vending machines~~

~~Silence~~

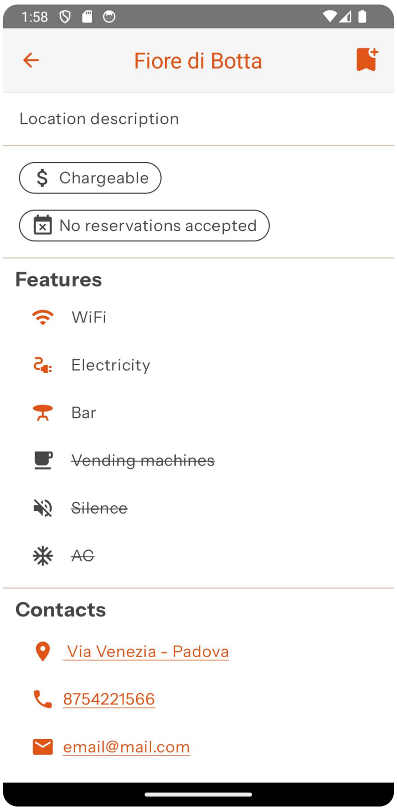
Figura 16: Schermata del dettaglio di un luogo: descrizione, caratteristiche e numero recensioni.

Selezionando un luogo, o dalla lista o dalla mappa, si può vederne in più dettaglio le sue caratteristiche:

- Un'anteprima del luogo, con una o più immagini: se ci si tocca sopra vengono mostrate le immagini a schermo intero;
- Il nome del luogo;
- Dove si trova, toccandoci sopra verrà visualizzata una mappa che mostra il posto;
- L'utente che ha postato il luogo;
- Se il luogo è aperto o chiuso, con pulsante che mostra gli orari di apertura;
- Valutazione media e numero recensioni;
- Descrizione luogo;
- Se il luogo è a pagamento e se il luogo accetta prenotazioni;
- Le caratteristiche del luogo, se in scuro barrato vuol dire che quella caratteristica non è offerta nel luogo; alcune caratteristiche possono essere: presenza di corrente elettrica, Wi-Fi, aria condizionata, etc.;
- I contatti del luogo: posizione, numero di telefono, mail ed eventuali link a siti web, tutti quanti toccabili (per esempio, toccando la mail possiamo mandare una mail).

Toccando in alto a sinistra si torna alla schermata precedente e toccando in alto a destra si può salvare il luogo nei preferiti se si ha effettuato l'accesso.

Il *ViewModel* associato a questa schermata chiama un metodo del repository dei luoghi per ottenere il luogo selezionato passandogli come parametro l'identificatore del luogo, mentre per salvare il luogo funziona in modo simile alla lista e alla mappa, cioè quando si tocca il pulsante per salvare nei preferiti, una classe *UseCase* si occupa di salvare il luogo nei preferiti nell'account utente, toccando ancora il pulsante si rimuove il luogo dai preferiti. Altri metodi del *ViewModel* consentono di visualizzare le immagini a schermo intero, di visualizzare o meno la scheda con gli orari di apertura e di visualizzare o meno la posizione del luogo nella mappa.



Location description

Chargeable

No reservations accepted

Features

- WiFi
- Electricity
- Bar
- ~~Vending machines~~
- ~~Silence~~
- ~~AC~~

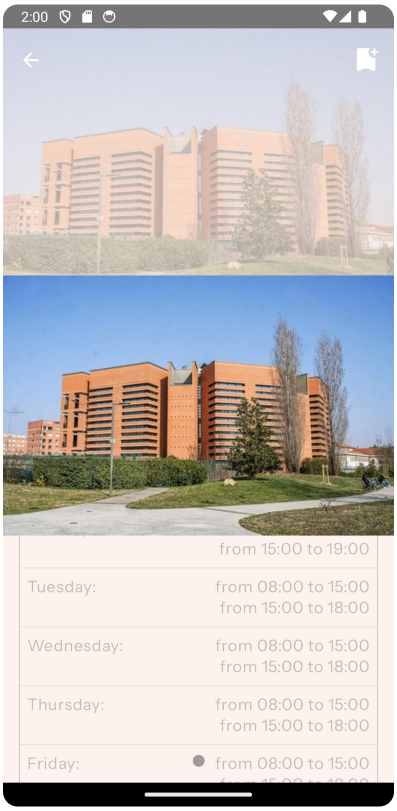
Contacts

Via Venezia - Padova

8754221566

email@mail.com

Figura 17: Schermata del dettaglio di un luogo: contatti.



	from 15:00 to 19:00

Tuesday:	from 08:00 to 15:00
	from 15:00 to 18:00

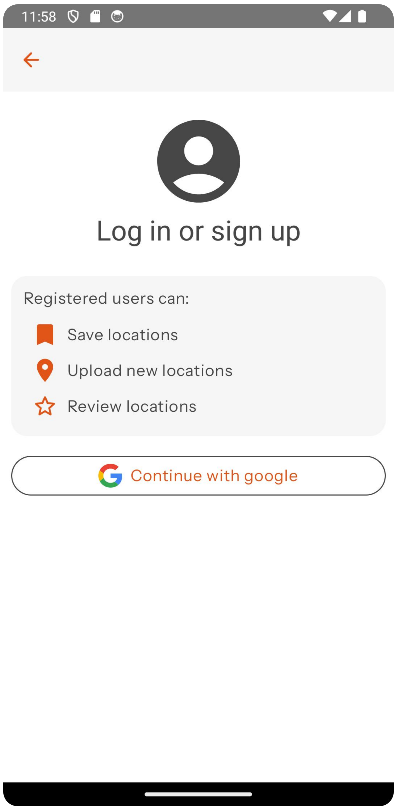
Wednesday:	from 08:00 to 15:00
	from 15:00 to 18:00

Thursday:	from 08:00 to 15:00
	from 15:00 to 18:00

Friday:	from 08:00 to 15:00

Figura 18: Schermata del dettaglio di un luogo: anteprima del luogo a schermo intero.

5.5.4 *Schermata login e profilo utente*

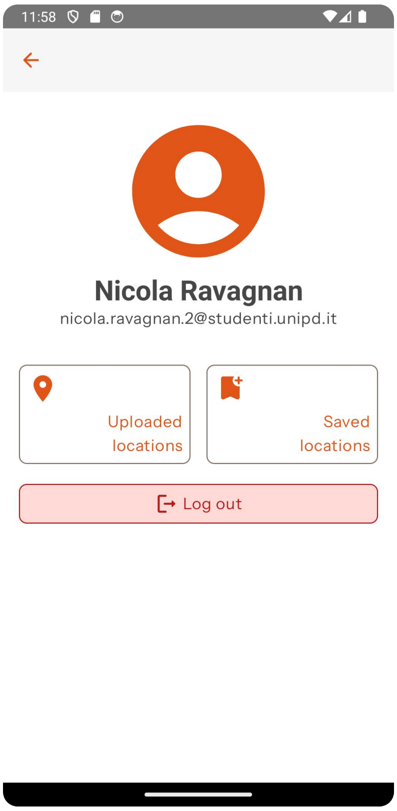


Log in or sign up

Registered users can:

- Save locations
- Upload new locations
- Review locations

Figura 19: Schermata di login.



Nicola Ravagnan

nicola.ravagnan.2@studenti.unipd.it

Figura 20: Schermata del profilo utente.

Dalla lista dei luoghi (o dalla mappa), toccando l'icona del profilo in alto a destra si può accedere alla schermata di login, dove si può effettuare il login al proprio account utilizzando un account *Google*. Se si è già registrati invece verrà visualizzata la schermata del profilo utente, dove si può vedere:

- La foto profilo;
- Il nome utente;
- La propria mail.

Da questa schermata l'utente registrato può;

- Selezionare di vedere i luoghi che ha caricato;
- Selezionare di vedere i luoghi preferiti che ha salvato;
- Effettuare il logout.

In alto a sinistra si può tornare alla schermata precedente.

La schermata di login ha un suo *ViewModel* associato, nel quale viene iniettata una classe di *Amplify* utilizzata per configurare *Cognito* e, tramite un metodo del *ViewModel* che viene chiamato quando un utente vuole accedere con un account *Google*, redige alla pagina di accesso con *Google*.

La schermata del profilo utente ha un suo *ViewModel* associato, con un metodo che, chiamando un metodo del repository degli utenti, si occupa di prelevare i dati dell'utente che ha effettuato l'accesso e un metodo per effettuare il logout.

Dalla schermata del profilo utente si può accedere alla lista dei luoghi salvati e alla lista dei luoghi caricati:

- La lista dei luoghi salvati mostra i luoghi che l'utente ha salvato nei preferiti, toccando un luogo si accede alla schermata dei dettagli del luogo selezionato; questa lista ha un *ViewModel* associato che tramite una classe *UseCase* chiama un metodo del repository dei luoghi che restituisce la lista dei luoghi salvati dall'utente;
- La lista dei luoghi caricati mostra i luoghi che l'utente ha caricato, toccando un luogo si accede alla schermata dei dettagli del luogo selezionato; questa lista ha un *ViewModel* associato che tramite una classe *UseCase* chiama un metodo del repository dei luoghi che restituisce la lista dei luoghi caricati dall'utente.

5.5.5 *Caricamento di un luogo*

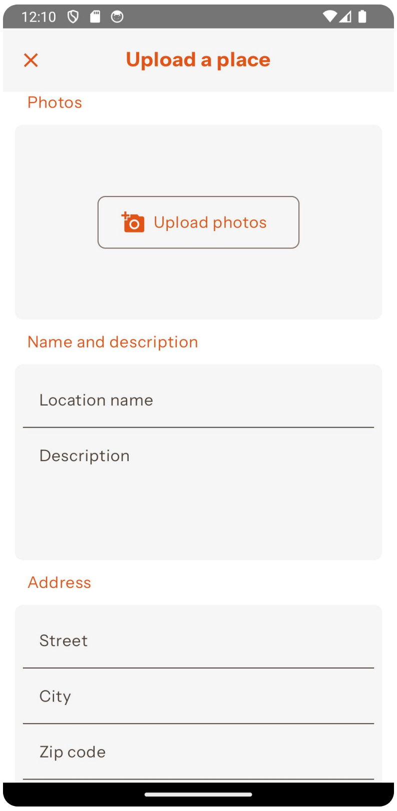


Figura 21: Schermata di caricamento luogo: foto, nome e descrizione.

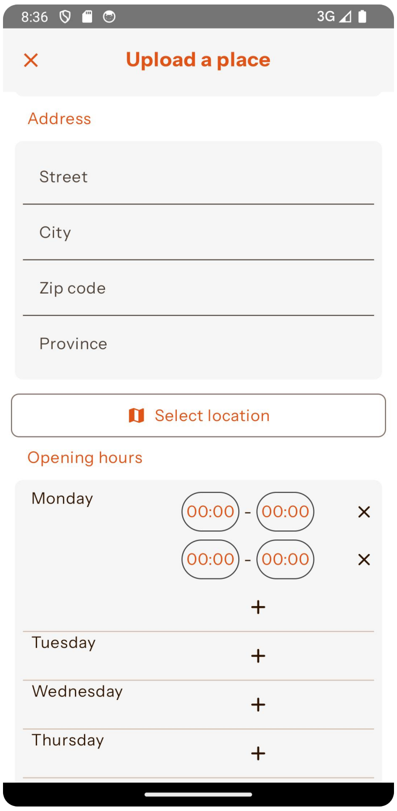


Figura 22: Schermata di caricamento luogo: indirizzo e orari di apertura.

Selezionando il pulsante di aggiunta (come utente registrato) si può accedere alla schermata di caricamento di un luogo, dove si può inserire:

- Una o più foto del luogo;
- Il nome del luogo;
- La descrizione del luogo;
- La posizione del luogo, con strada, città, codice postale e provincia: toccando il pulsante sotto *Seleziona luogo* si può selezionare la posizione da una mappa;
- Gli orari di apertura del luogo: per ogni giorno si può aggiungere un intervallo orario con ora di apertura e ora di chiusura;
- Il prezzo;
- La possibilità di prenotare un posto;

- Le caratteristiche del luogo, selezionando quelle che offre;
- I contatti del luogo, con mail, numero di telefono e la possibilità di aggiungere uno o più sito web.

Questa schermata ha un *ViewModel* associato, che contiene metodi per impostare la posizione del luogo, gli attributi del luogo (nome, descrizione, etc.), le caratteristiche del luogo, per mostrare o meno la mappa dove selezionare la posizione, un metodo per aggiungere o rimuovere un orario di apertura/chiusura, un metodo per aggiungere un link, un metodo per rimuovere un link e un metodo per caricare il luogo, tramite una classe *UseCase* che chiama un metodo del repository dei luoghi.

Con il pulsante in fondo si può confermare il caricamento del luogo, l'esito può essere:

- Caricamento avvenuto con successo, viene mostrato un messaggio in basso a schermo;
- Caricameto fallito a causa di dati immessi in modo non corretto, i campi interessati saranno segnati in rosso e un messaggio di errore verrà mostrato in basso a schermo;
- Se l'utente non ha effettuato l'accesso al suo account, il caricamento fallisce, ma questa circostanza non dovrebbe succedere dato che la schermata è accessibile solo se si ha effettuato l'accesso;
- Caricamento fallito a causa di un errore di connessione, verrà mostrato un messaggio di errore in basso a schermo.

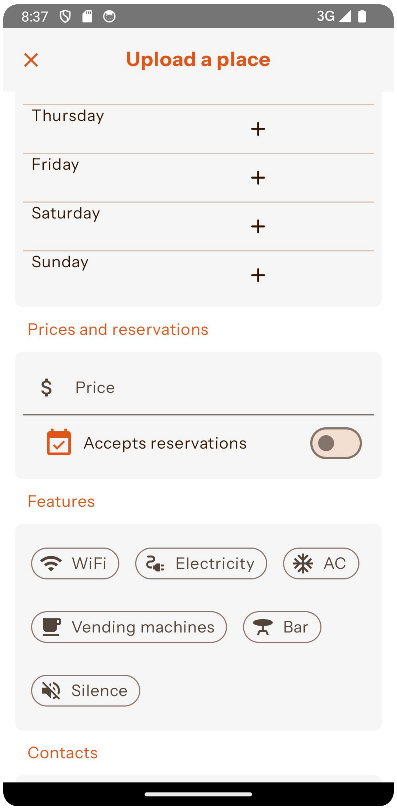


Figura 23: Schermata di caricamento luogo: prezzo e caratteristiche.

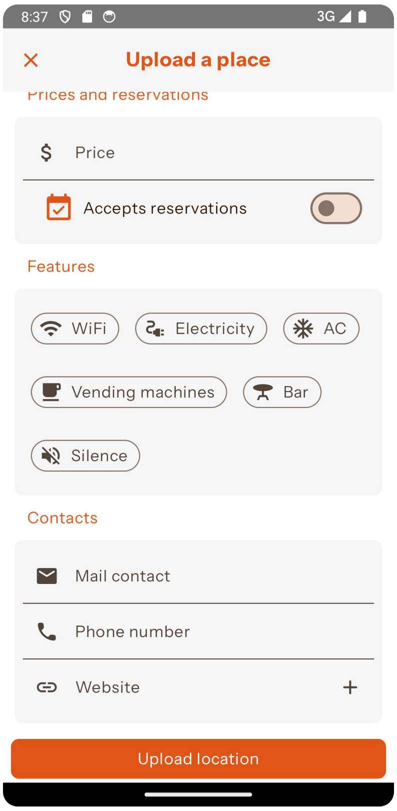


Figura 24: Schermata di caricamento luogo: contatti.

5.5.6 *Filtri e ricerca*

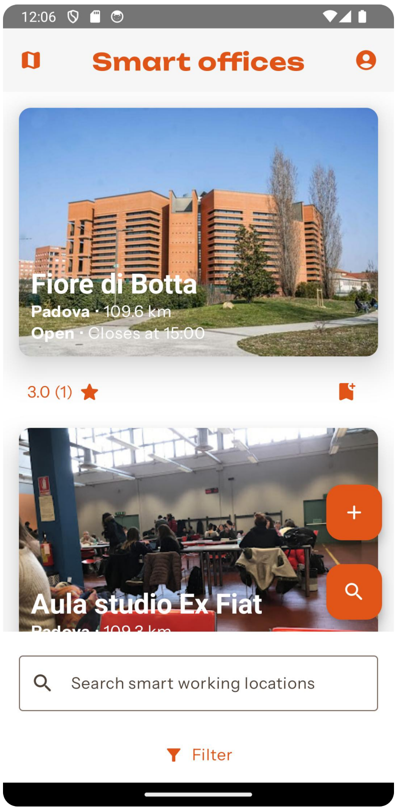


Figura 25: Casella di ricerca nella lista luoghi.

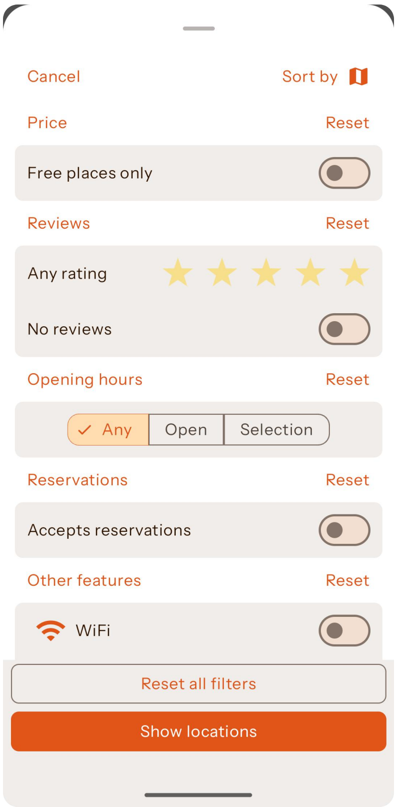


Figura 26: Schermata filtri di ricerca.

Selezionando il tasto di ricerca in basso a destra nella schermata della lista dei luoghi o della mappa dei luoghi appare una casella di ricerca nella quale si può cercare un luogo per nome. Se si seleziona il pulsante di filtraggio in basso alla casella di ricerca si può filtrare la lista dei luoghi (o la mappa) a seconda delle proprie preferenze, si può filtrare per:

- Solo posti gratuiti;
- Per valutazione media;
- Per ora di apertura oppure se sono aperti nel momento in cui si effettua la ricerca;
- Solo posti che accettano prenotazioni;
- Per caratteristiche, selezionando quelle che si vogliono cercare.

Per la lista luoghi, in alto a destra della finestra dei filtri si può ordinare la lista per:

- Distanza;
- Valutazione media;
- Data di aggiornamento.

Dopo aver selezionato i filtri si può applicare la ricerca toccando il pulsante *Mostra luoghi* in fondo, per ripristinare i filtri si tocca invece il pulsante *Ripristina tutti i filtri*.

La finestra dei filtri non ha un suo *ViewModel* associato ma è integrata nella lista dei luoghi e nella mappa dei luoghi: applicando i filtri, quindi, verranno chiamati i rispettivi metodi che modificheranno lo stato della schermata e aggiorneranno la lista/mappa.

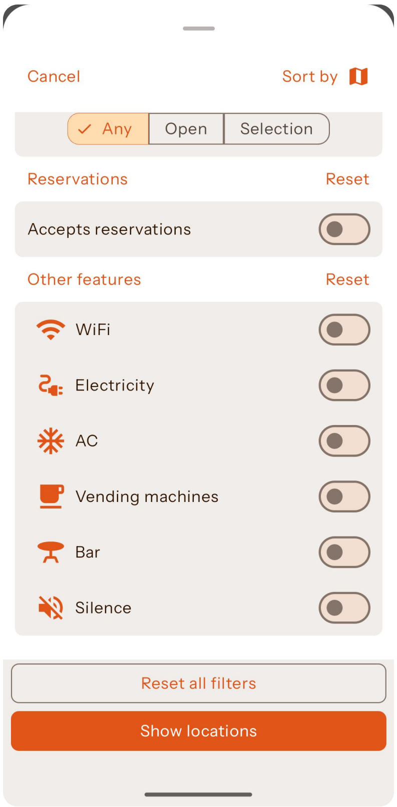


Figura 27: Schermata filtri di ricerca: filtraggio per caratteristica.

,

5.6 Repository codice

Il codice dell'applicazione è stato caricato su un repository **Bitbucket** aziendale. Per la gestione dei *branch* si è utilizzato un *gitflow* standard: ogni funzionalità è stata sviluppata su un ramo di sviluppo separato. Quando una funzionalità veniva considerata finita, veniva aperta una *Pull request*, nella quale il codice veniva controllato dai colleghi in azienda, e dopo che la *Pull request* veniva approvata, si passava al merge con il ramo di sviluppo principale.

6 Validazione

Alla fine del tirocinio le funzionalità che effettivamente sono state codificate sono:

- **F1**: Lista dei luoghi (con funzione di ordinamento e filtraggio);
- **F2**: Mappa dei luoghi (con funzione di filtraggio);
- **F3**: Visualizzazione in dettaglio di un luogo;
- **F4**: Caricamento di un luogo;
- **F5**: Pagina di login;
- **F6**: Pagina del profilo utente, con lista dei luoghi caricati e dei luoghi salvati.

Per mancanza di tempo è stato deciso di tralasciare le seguenti funzionalità che erano state pianificate:

- **F7**: Registrazione di un nuovo account;
- **F8**: Lista delle recensioni di un luogo;
- **F9**: Caricamento di una nuova recensione di un luogo.

6.1 Requisiti soddisfati

In conclusione, alla codifica sono stati soddisfatti i seguenti requisiti (per vedere la tabella dei requisiti soddisfatti, vedi *Tabella requisiti soddisfatti* nell'appendice):

- Per la **lista dei luoghi** tutti i requisiti sono stati soddisfatti: un utente può visualizzare la lista dei luoghi, selezionare un luogo dalla lista per vederne i dettagli, ordinare la lista dei luoghi per distanza, valutazione o data di caricamento e filtrare la lista dei luoghi per caratteristiche, per prezzo, per nome o per orario di apertura;
- Per la **mappa dei luoghi** tutti i requisiti sono stati soddisfatti: un utente può visualizzare la mappa dei luoghi, selezionare un luogo dalla mappa per vederne i dettagli e filtrare la mappa dei luoghi come per la lista dei luoghi (ma non di ordinarla);
- Per la **visualizzazione in dettaglio di un luogo** tutti i requisiti sono stati soddisfatti: un utente può visualizzare le informazioni in dettaglio di un luogo, cioè la posizione geografica, le caratteristiche di un luogo, gli orari di apertura, i contatti, le immagini di un luogo e il numero di recensioni;
- Nella **lista dei luoghi**, nella **mappa dei luoghi** e nella **visualizzazione in dettaglio di un luogo**, se l'utente è registrato, è possibile salvare un luogo nei preferiti o rimuovere un luogo già salvato dai preferiti;
- Per il **caricamento di un luogo** tutti i requisiti sono stati soddisfatti: un utente registrato può caricare un nuovo luogo, inserendo il nome del luogo, la posizione geografica del luogo, i contatti del luogo, una descrizione del luogo, gli orari di apertura del luogo e una o più immagini del luogo;
- Per la **pagina di login** un ospite può accedere solo con un account *Google*, dato che non è stata implementata la funzionalità di registrazione di un nuovo account;
- Per la **pagina del profilo utente** un utente registrato può visualizzare il suo profilo utente, visualizzare i suoi dati personali, visualizzare la lista dei luoghi preferiti salvati, visualizzare la lista dei luoghi caricati da lui e di effettuare il logout;
- I **requisiti di vincolo** sono stati soddisfatti tutti eccetto che le componenti sviluppate non sono state documentate per mancanza di tempo;
- I **requisiti di qualità** sono stati soddisfatti tutti.

6.2 Requisiti non soddisfatti

I seguenti requisiti non sono stati soddisfatti:

- La funzionalità di **registrazione di un nuovo account** non è stata implementata: un ospite non può creare un nuovo account, ma può accedere solo usando un account *Google*;
- La funzionalità di **visualizzazione della lista delle recensioni di un luogo** non è stata implementata: un utente non può visualizzare la lista delle recensioni di un luogo;
- La funzionalità di **caricamento di una nuova recensione** di un luogo non è stata implementata: un utente registrato non può caricare una nuova recensione di un luogo;

- L'unico **requisito di vincolo** non soddisfatto è che le componenti sviluppate non sono state documentate per mancanza di tempo.

6.3 Riepilogo requisiti

Tipo di requisito	Sodisfatti	Totali	Percentuale
Funzionali	41	49	83.67%
Vincolo	3	4	75%
Qualità	3	3	100%
Totali	47	56	83.92%

Tabella 3: Riepilogo requisiti soddisfatti dopo la codifica.

Tutti requisiti di qualità sono stati soddisfatti, mentre alcuni requisiti funzionali e di vincolo non sono stati soddisfatti; In totale sono stati soddisfatti più dell'80% dei requisiti totali.

7 Conclusione

Alla fine il tirocinio è durato in totale 320 ore di lavoro (o 8 settimane lavorative di 40 ore di lavoro), con la fine effettiva del tirocinio il 25/08/2023. In seguito, verranno segnati il consuntivo periodico rispetto al piano di lavoro iniziale, il resoconto degli obiettivi e le eventuali variazioni rispetto a quanto pianificato.

7.1 Consuntivo periodico

7.1.1 *Periodo 1 (28/06/2023 - 05/07/2023)*

In questo periodo ho iniziato il periodo di formazione partendo con un rapido studio del linguaggio di programmazione *Kotlin*, studiandone la sintassi e i costrutti, e subito dopo ho iniziato con il corso online **Android Basics with Compose** partendo con l'apprendimento di *Jetpack Compose*.

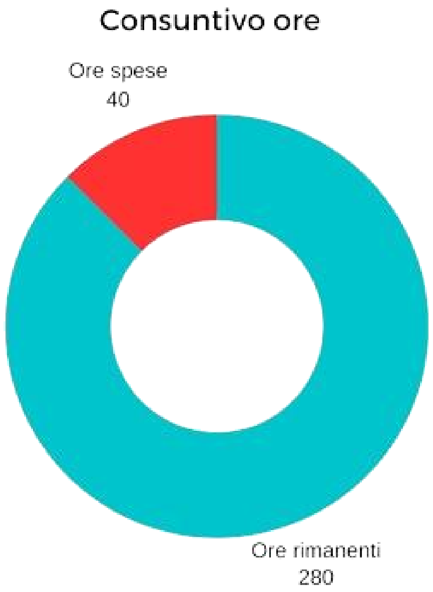


Figura 28: Grafico consuntivo ore periodo 1.

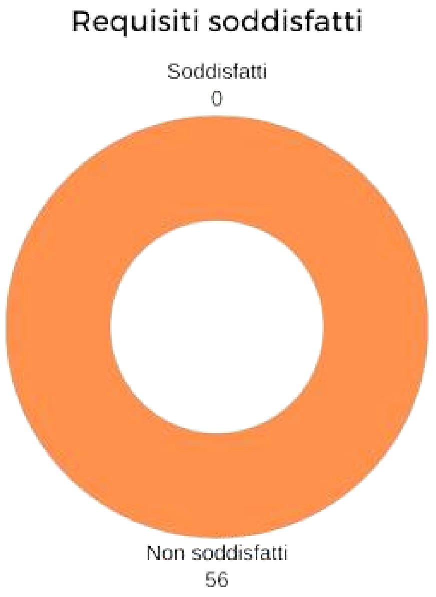


Figura 29: Grafico requisiti soddisfatti periodo 1.

Vedendo dai grafici, in questo periodo ho speso 40 ore di formazione ma non ho soddisfatto ancora nessun requisito, dato che ero ancora in periodo di formazione.

7.1.2 *Periodo 2 (06/07/2023 - 19/07/2023)*

In questo periodo:

- Ho continuato il periodo di formazione, imparando la struttura architetturale, ad usare client come *Ktor* per scambiare dati remoti e ad usare *Room*;
- Ho finito il periodo di formazione con una breve demo finale;
- Ho iniziato a lavorare sul progetto effettivo, partendo con la creazione dell'applicazione e l'inizio dell'implementazione della lista dei luoghi, partendo dall'interfaccia grafica.

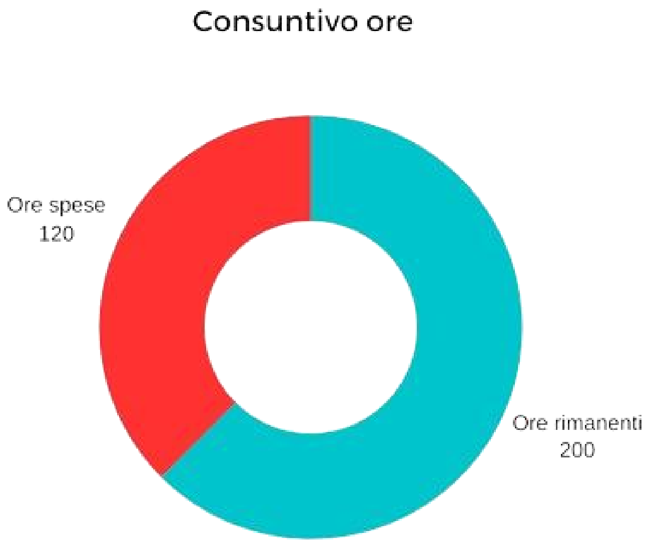


Figura 30: Grafico consuntivo ore periodo 2.

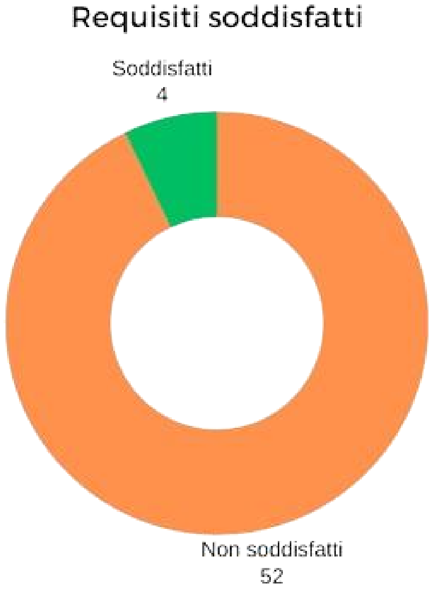


Figura 31: Grafico requisiti soddisfatti periodo 2.

Vedendo dai grafici, in questo periodo ho speso 80 ore lavorative e ho soddisfatto in totale 4 requisiti su 56.

7.1.3 *Periodo 3 (20/07/2023 - 02/08/2023)*

In questo periodo:

- Ho continuato e finito la schermata della lista dei luoghi, implementando i filtri e l'ordinamento della lista;
- Ho iniziato e finito l'implementazione della schermata di dettaglio del luogo;
- Ho sviluppato anche la classe di repository dei luoghi e la classe per le chiamate alle API dei luoghi;

- Alla fine del periodo ho iniziato anche a lavorare sull'interfaccia grafica della schermata del caricamento di un nuovo luogo.

Le funzionalità completate in questo periodo sono state:

- **F1**: Lista dei luoghi (con funzione di ordinamento e filtraggio);
- **F3**: Visualizzazione in dettaglio di un luogo.

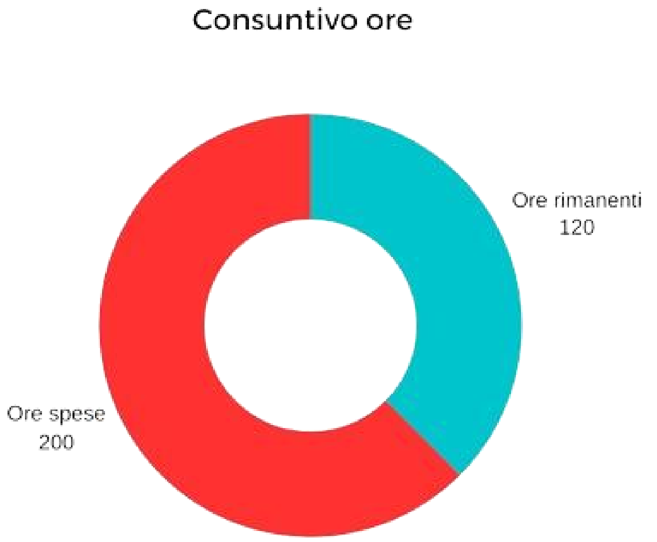


Figura 32: Grafico consuntivo ore periodo 3.

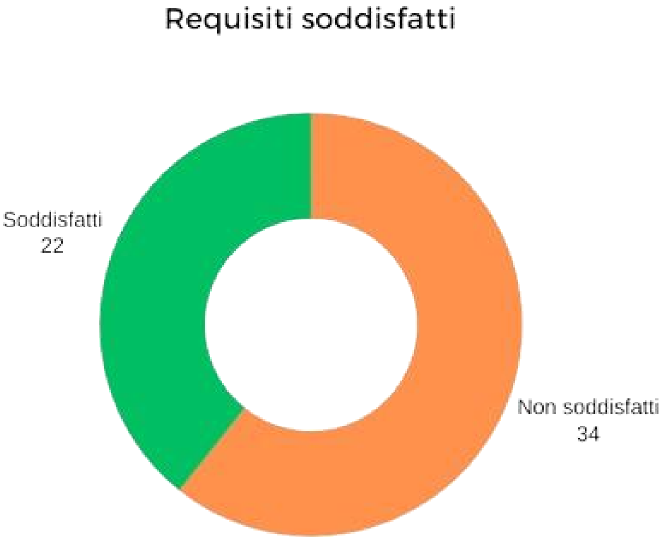


Figura 33: Grafico requisiti soddisfatti periodo 3.

Vedendo dai grafici, in questo periodo ho speso 80 ore lavorative e ho soddisfatto in totale 22 requisiti su 56.

7.1.4 *Periodo 4 (03/08/2023 - 16/08/2023)*

In questo periodo:

- Ho implementato la chiamata al back-end per il caricamento di un nuovo luogo, concludendo la schermata;
- Ho impostato la classe per configurare *AWS Cognito* e ho implementato l'interfaccia grafica e le funzionalità della schermata di login;
- Ho sviluppato la pagina del profilo utente, con la creazione della classe di repository degli utenti e la classe API per gli utenti;
- Ho sviluppato la mappa dei luoghi con funzione di filtraggio, ci ho messo meno tempo dato che possiede una logica simile alla lista dei luoghi.

Le funzionalità completate in questo periodo sono state:

- **F2**: Mappa dei luoghi (con funzione di filtraggio);
- **F4**: Caricamento di un luogo;
- **F5**: Pagina di login;
- **F6**: Pagina del profilo utente, con lista dei luoghi caricati e dei luoghi salvati.

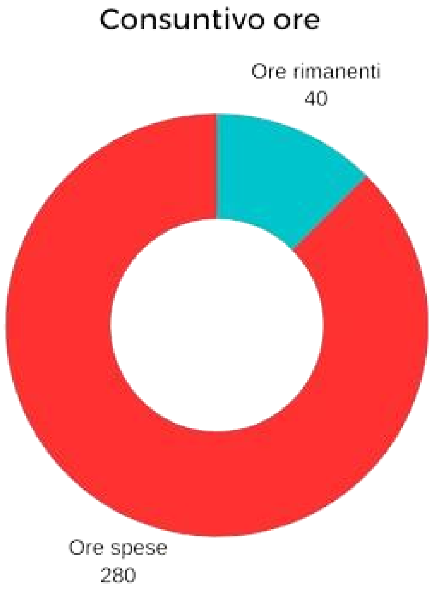


Figura 34: Grafico consuntivo ore periodo 4.

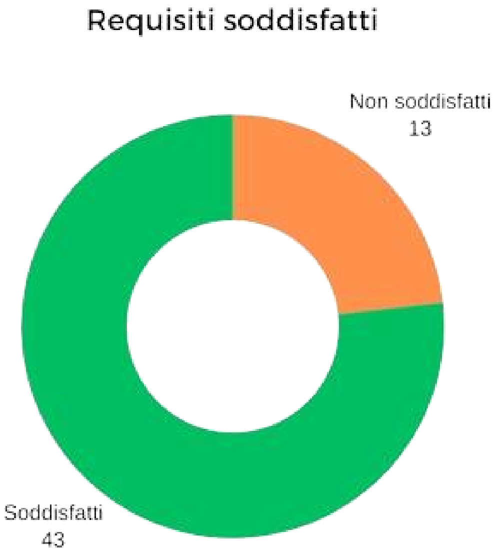


Figura 35: Grafico requisiti soddisfatti periodo 4.

Vedendo dai grafici, in questo periodo ho speso 80 ore lavorative e ho soddisfatto in totale 43 requisiti su 56.

7.1.5 *Periodo 5 (17/08/2023 - 25/08/2023)*

Nel periodo finale:

- Mi sono impegnato a impostare il database locale *Room*;
- Ho implementato la funzionalità di salvataggio di un luogo;
- Ho dato gli ultimi ritocchi all'applicazione sistemando alcuni problemi noti.

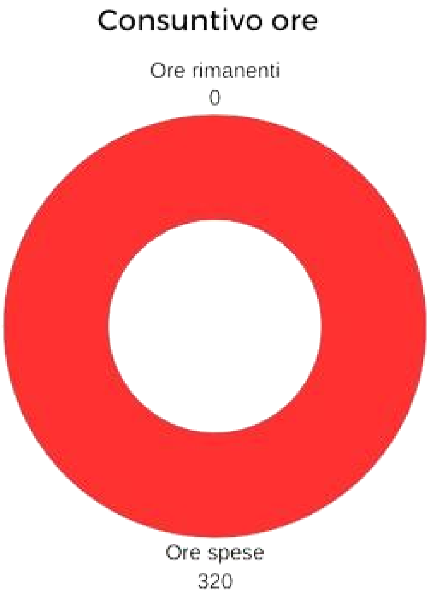


Figura 36: Grafico consuntivo ore periodo 5.

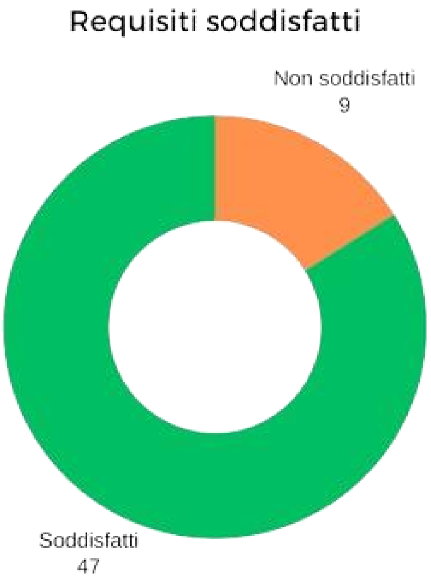


Figura 37: Grafico requisiti soddisfatti periodo 5.

Vedendo dai grafici, in questo periodo ho speso 40 ore lavorative, con due giorni di pausa in mezzo, e ho soddisfatto in totale 47 requisiti su 56.

7.1.6 *Variazioni rispetto alla pianificazione*

Rispetto il piano di lavoro iniziale, il periodo di formazione è durato una settimana in più del previsto, in quanto ho avuto bisogno di più tempo per finire il corso online. Inoltre, le schermate di login e profilo utente sono state sviluppate in un periodo diverso da quello previsto, in quanto è stato trovato più comodo ad iniziare con la schermata della lista dei luoghi e la visualizzazione dei dettagli dei luoghi. Le attività pianificate per il quinto periodo non sono state realizzate per mancanza di tempo.

7.2 Resoconto obiettivi e prodotti attesi

Alla fine del tirocinio ho raggiunto i seguenti obiettivi:

Obiettivo	**Resoconto**	**Grado di completamento**
O-1	Sono riuscito a comprendere lo sviluppo di un'applicazione mobile e di sviluppare a pieno un'applicazione funzionante, sperimentando lo sviluppo sia della logica di business che dell'interfaccia grafica	Completato
O-2	Sono riuscito a sperimentarmi con l'interazione con un servizio remoto RestFul JSON, sia per ricevere dati che per inviare dati, e sono riuscito a presentare con successo i dati ricevuti nell'applicazione	Completato
O-3	Grazie agli stand-up giornalieri ho avuto esperienza con una tipologia di lavoro agile che si concentra sul lavoro di squadra con il constante aggiornamento del progresso di lavoro	Completato

Tabella 4: Riepilogo obiettivi formativi tirocinio.

Per gli obiettivi formativi, in sintesi, sono riuscito a raggiungere tutti gli obiettivi con successo.

Per quanto riguardano i prodotti finali attesi:

Prodotto/i	**Resoconto**	**Grado di completamento**
P-1	L'applicazione è stata sviluppata ed è funzionante, con la maggior parte dei requisiti soddisfatti	Completato
P-2	I test automatici sono stati sviluppati nella parte di formazione ma non sono stati sviluppati per l'applicazione in sé	Parzialmente completato
P-3 e **P-4**	Non sono stati fatti per mancanza di tempo	Non completati

Tabella 5: Riepilogo prodotti attesi tirocinio.

Questa mancanza di tempo è dovuta ad un ritardo dell'inizio dello sviluppo del progetto effettivo a causa di una durata maggiore del previsto per la formazione, che è durata 1 settimana in più del previsto.

Per quanto riguardano i rischi:

- Il rischio che si è presentato in maniera più evidente è stato **R-1** che come già detto ha portato ad un ritardo nello sviluppo del progetto effettivo, ma che non ha portato a nessun problema di sorta, in quanto l'applicazione è stata sviluppata e funzionante, anche se non con tutti i requisiti soddisfatti;
- Il rischio **R-2** non si è presentato in quanto già avevo una conoscenza di base di API REST dagli anni passati in università, l'unica cosa che ho dovuto apprendere è stata l'autenticazione utente con il token generato da *Cognito*;
- Il rischio **R-3** non si è presentato in quanto durante il progetto sono stato in grado di lavorare con abbastanza autonomia, alcune volte chiedendo pareri o riportando il progresso del lavoro e raramente mi sono trovato in situazioni di stallo;

- Il rischio **R-4** non si è presentato in quanto tutte le funzionalità sono state integrate senza problemi e tutte le *Pull requests* venivano approvate in giornata.

7.3 Considerazioni finali

Tutto sommato mi ritengo soddisfatto di questa esperienza di tirocinio, in quanto mi ha permesso di imparare a sviluppare un'applicazione mobile completa utilizzando le tecnologie più recenti e mi ha permesso di lavorare con un team di sviluppo in un ambiente professionale come l'azienda. Mi sarebbe piaciuto poter completare il progetto con tutti i prodotti attesi completati: infatti, personalmente una cosa che avrei migliorato sarebbe stata la mia gestione del tempo in generale, in quanto ciò ha portato via tempo allo sviluppo del progetto effettivo. Concludendo penso che tutto ciò che ho imparato durante questa esperienza certamente la userò in futuro, sia per quanto riguarda lo sviluppo di applicazioni mobile, ma soprattutto per quanto riguarda la gestione del tempo e del lavoro in un team di sviluppo.

Appendice

A.1 Lista requisiti

I requisiti sono classificati con la seguente codifica:

R[Tipo]-numero

- **R**: Acronimo di *Requisito*
- **Tipo**: Tipo di requisito, può essere:
  - **F**: *Funzionale*
  - **V**: *Vincolo*
  - **Q**: *Qualità*

Requisito	Descrizione	Funzionalità
RF-1	L'utente vuole visualizzare la lista dei luoghi	F1
RF-2	L'utente vuole selezionare un luogo dalla lista per vederne i dettagli	F1
RF-3	L'utente vuole visualizzare i dettagli di un luogo	F3
RF-4	L'utente vuole visualizzare la posizione geografica un luogo	F3
RF-5	L'utente vuole visualizzare i contatti di un luogo	F3
RF-6	L'utente vuole visualizzare le caratteristiche di un luogo	F3
RF-7	L'utente vuole visualizzare gli orari di apertura di un luogo	F3
RF-8	L'utente vuole visualizzare le immagini di un luogo	F3
RF-9	L'utente vuole visualizzare il numero di recensioni di un luogo	F3
RF-10	L'utente vuole visualizzare la mappa dei luoghi	F2
RF-11	L'utente vuole selezionare un luogo dalla mappa per vederne i dettagli	F2
RF-12	L'utente visualizza un messaggio a causa di un errore nel caricamento dei dati	F1, F2, F3
RF-13	L'utente visualizza un messaggio che indica che la lista dei luoghi è vuota	F1
RF-14	L'ospite vuole effettuare il login all'suo profilo utente	F5
RF-15	L'ospite vuole effettuare il login all'suo profilo utente utilizzando un Account *Google*	F5
RF-16	L'ospite vuole creare un nuovo account per registrarsi sull'applicazione	F7
RF-17	L'ospite inserisce un nome utente per registrarsi	F7
RF-18	L'ospite inserisce una e-mail per registrarsi	F7
RF-19	L'utente vuole effettuare il logout dal suo profilo utente	F6
RF-20	L'utente registrato vuole visualizzare il suo profilo utente	F6
RF-21	L'utente registrato visualizza i suoi dati personali	F6
RF-22	L'utente registrato vuole caricare un nuovo luogo	F4
RF-23	L'utente registrato inserisce il nome del luogo da caricare	F4
RF-24	L'utente registrato inserisce una descrizione del luogo da caricare	F4

RF-25	L'utente registrato inserisce la posizione geografica del luogo da caricare	F4
RF-26	L'utente registrato inserisce le caratteristiche del luogo da caricare	F4
RF-27	L'utente registrato inserisce gli orari di apertura del luogo da caricare	F4
RF-28	L'utente registrato inserisce i contatti del luogo da caricare	F4
RF-29	L'utente registrato inserisce una o più immagini del luogo da caricare	F4
RF-30	L'utente visualizza un messaggio che indica che non ha inserito tutte le informazioni richieste del luogo da caricare	F4
RF-31	L'utente visualizza un messaggio a causa di un errore nel caricamento del nuovo luogo	F4
RF-32	L'utente registrato vuole salvare un luogo nei preferiti	F1, F2, F3
RF-33	L'utente registrato vuole rimuovere un luogo dai preferiti	F1, F2, F3
RF-34	L'utente registrato vuole visualizzare la lista dei luoghi preferiti salvati	F6
RF-35	L'utente registrato vuole selezionare un luogo dalla lista dei luoghi preferiti salvati	F6
RF-36	L'utente vuole visualizzare la lista dei luoghi caricati da lui	F6
RF-37	L'utente vuole selezionare un luogo dalla lista dei luoghi caricati da lui	F6
RF-38	L'utente vuole visualizzare la lista delle recensioni di un luogo	F8
RF-39	L'utente registrato vuole caricare una nuova recensione di un luogo	F9
RF-40	L'utente registrato vuole inserire il testo di una nuova recensione	F9
RF-41	L'utente registrato vuole inserire una valutazione insieme alla recensione	F9
RF-42	L'utente registrato visualizza un messaggio a causa di un errore nel caricamento della nuova recensione	F9
RF-43	L'utente vuole ordinare la lista dei luoghi per distanza	F1
RF-44	L'utente vuole ordinare la lista dei luoghi per valutazione	F1
RF-45	L'utente vuole ordinare la lista dei luoghi per data di caricamento	F1
RF-46	L'utente vuole filtrare la lista o la mappa dei luoghi per nome	F1, F2
RF-47	L'utente vuole filtrare la lista o la mappa dei luoghi per prezzo	F1, F2
RF-48	L'utente vuole filtrare la lista o la mappa dei luoghi per le caratteristiche scelte	F1, F2
RF-49	L'utente vuole filtrare la lista o la mappa dei luoghi per orario di apertura	F1, F2

Tabella 6: Requisiti funzionali dell'applicazione *Android*.

Requisito	**Descrizione**
RV-1	L'applicazione deve essere sviluppata utilizzando il toolkit UI *Jetpack Compose*
RV-2	L'applicazione deve essere sviluppata utilizzando il linguaggio *Kotlin*
RV-3	L'applicazione finale deve essere utilizzabile da dispositivi *Android* dalla versione 13.0
RV-4	Le componenti sviluppate devono essere documentate

Tabella 7: Requisiti di vincolo dell'applicazione *Android*.

Requisito	**Descrizione**
RQ-1	L'applicazione deve essere fruibile anche in assenza di connessione ad Internet
RQ-2	Il codice dell'applicazione deve essere presente nel repository *Bitbucket* aziendale
RQ-3	Il codice del progetto deve passare tutte le *Pull requests*

Tabella 8: Requisiti di qualità dell'applicazione *Android*.

A.2 Tabella requisiti soddisfatti

In seguito, viene riportata la tabella dei requisiti con il seguente grado di soddisfazione:

- **S**: Soddisfatto
- **NS**: Non soddisfatto

Requisito	**S/NS**
RF-1	S
RF-2	S
RF-3	S
RF-4	S
RF-5	S
RF-6	S
RF-7	S
RF-8	S
RF-9	S
RF-10	S
RF-11	S
RF-12	S
RF-13	S
RF-14	S
RF-15	S
RF-16	NS
RF-17	NS
RF-18	NS
RF-19	S
RF-20	S
RF-21	S
RF-22	S
RF-23	S
RF-24	S
RF-25	S

RF-26	S
RF-27	S
RF-28	S
RF-29	S
RF-30	S
RF-31	S
RF-32	S
RF-33	S
RF-34	S
RF-35	S
RF-36	S
RF-37	S
RF-38	NS
RF-39	NS
RF-40	NS
RF-41	NS
RF-42	NS
RF-44	S
RF-45	S
RF-46	S
RF-47	S
RF-48	S
RF-49	S
RV-1	S
RV-2	S
RV-3	S
RV-4	NS
RQ-1	S
RQ-2	S
RV-3	S

Tabella 9: Tabella requisiti soddisfatti e non soddisfatti.

A.3 MainActivity

```
@AndroidEntryPoint
class MainActivity : ComponentActivity() {

    @Inject
    lateinit var amplifyManager: AmplifyManager
    @OptIn(ExperimentalAnimationApi::class, ExperimentalMaterialNavigationApi::class)
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        setContent {
            SmartOfficesTheme() {
                // A surface container using the 'background' color from the theme
                Surface(modifier = Modifier.fillMaxSize(), color =
MaterialTheme.colorScheme.background) {
                    val navController = rememberAnimatedNavController()

                    val navHostEngine = rememberAnimatedNavHostEngine(
                        /**
                        ...
                        */
                    )
                    DestinationsNavHost(
                       navGraph = NavGraphs.root,
                       navController = navController,
                       engine = navHostEngine
                   ){
                        composable(AccountRedirectRouteDestination){
                            AccountRedirectRoute(
                                navigator = destinationsNavigator,
                                amplify = amplifyManager
                            )
                        }
                   }
                }
            }
        }
    }
}
```

Classe che estende `ComponentActivity` ed è la prima classe che genera contenuti a schermo: inizializza il tema dell'applicazione ed inizializza il `NavHost` e il `NavController` che gestiscono la navigazione tra i vari schermi. Ho implementato solamente la funzione `onCreate` che viene chiamata quando il processo viene creato, tenendo il comportamento di default per le altre funzioni. L'annotazione `@AndroidEntryPoint` permette di iniettare le dipendenze con *Hilt*, in questo caso ho iniettato l'oggetto `AmplifyManager` che gestisce le operazioni di autenticazione utenti per poi passarlo alla schermata di login/profilo utente.

A.4 Destinazione di navigazione

```
@RootNavGraph(
    start = true
)
@Destination
@Composable
fun PlacesListRoute(
    viewModel: PlacesListViewModel = hiltViewModel(),
    navigator: DestinationsNavigator
) {

    val uiState by viewModel.uiState.collectAsState()

    /**
    ...
    */

    when(uiState.response){
        is ListState.Success -> PlacesListScreen(
            filters = uiState.filters,
            showFilters = uiState.showFilters,
            placesList = uiState.placesList,
            onUploadScreenClick = {navigator.navigate(UploadPlaceRouteDestination)},
            onFiltersClick = { viewModel.toggleFilters(true) },
            onDismissFilters = { viewModel.toggleFilters(false) },
            onSearchFilter = { viewModel.updateSearchFilter(it) },
            onFilter = { viewModel.updateFilters(it) },
            onSelectAccountBox = { navigator.navigate(AccountRedirectRouteDestination) },
            onResetAllFilters = viewModel::resetAllFilters,
            onApplyFilters = viewModel::applyFilters,
            onClickCard = {navigator.navigate(PlacesDetailsRouteDestination(it))},
            onAddFavoritesCard = {viewModel.onToggleLocation(it)},
            onMapClick = { navigator.navigate(PlacesMapRouteDestination) },
            onToggleBottomBar = viewModel::toggleBottomBar,
            showBottomBar = uiState.showBottomBar,
            onRetry = {
                viewModel.resetAllFilters()
                viewModel.getLocationList()
            },
            isLoggedIn = uiState.isLoggedIn
        )
        is ListState.Loading -> LoadingScreen(
            modifier = Modifier
                .fillMaxSize()
                .size(100.dp)
        )
        else -> PlacesListErrorScreen(
            message = R.string.location_list_error_msg,
            onRetry = {
                viewModel.resetAllFilters()
                viewModel.getLocationList()
            },
            onAccountBoxClick = {
                navigator.navigate(AccountRedirectRouteDestination)
            },
            onNavigationScreen = {
                navigator.navigate(UploadPlaceRouteDestination)
            }
        )
    }
}
```

Una destinazione è una funzione annotata con `@Destination` e prende come parametri il `navigator` usato per navigare da una schermata all'altra e il *ViewModel* della schermata associata; dal *ViewModel* poi viene estratto lo stato dell'interfaccia che viene fatto passare al *Composable* della schermata. Oltre allo stato dell'interfaccia vengono passate anche le funzioni invocate all'input dell'utente. La notazione `@RootNavGraph` indica che si tratta del nodo iniziale, cioè la prima schermata visualizzata avviata l'applicazione.

A.5 Classi ViewModel

```
@HiltViewModel
class PlaceDetailsViewModel @Inject constructor(
    private val getLocalDetailsStreamUseCase: GetLocalDetailsStreamUseCase,
    private val toggleSavedLocationUseCase: ToggleSavedLocationUseCase,
    private val fetchCurrentUserDataUseCase: FetchCurrentUserDataUseCase
) : ViewModel() {

    private val _uiState: MutableStateFlow<PlaceDetailsUiState> =
        MutableStateFlow(PlaceDetailsUiState())
    val uiState: StateFlow<PlaceDetailsUiState> = _uiState.asStateFlow()

    private val mutex = Mutex()

    fun toggleGallery(enabled : Boolean){
        _uiState.update {
            it.copy(showGallery = enabled)
        }
    }

    init {
        viewModelScope.launch {
            fetchCurrentUserDataUseCase().collect {
                _uiState.update { state ->
                    state.copy(isLoggedIn = it)
                }
            }
        }
    }

    fun toggleMap(enabled: Boolean){
        _uiState.update {
            it.copy(showMap = enabled)
        }
    }

    fun toggleTimetable(enabled: Boolean){
        _uiState.update {
            it.copy(showHours = enabled)
        }
    }

    fun toggleSavedLocation(){
        viewModelScope.launch {
            if(!mutex.isLocked) {
                mutex.lock()
                val details = uiState.value.details
                details?.let {
                    val placeId = it.placeDetails.id
                    toggleSavedLocationUseCase(placeId)
                }
                mutex.unlock()
            }
        }
    }
    fun getLocationDetails(id : String) {
        viewModelScope.launch {
            getLocalDetailsStreamUseCase(id).collect { loc ->
                _uiState.update { state ->
                    state.copy(response = ListState.Success, details = loc)
                }
            }
        }
    }
}
```

Le classi *ViewModel* sono i titolari di stato di una schermata, estendono la classe `ViewModel` del framework *Android* e il loro ciclo di vita è regolato dal framework stesso. Con l'annotazione `@HiltViewModel` è possibile iniettare la classe nell'interfaccia grafica e con l'annotazione `@Inject` nel costruttore è possono iniettare le dipendenze con *Hilt*. In questo esempio la classe gestisce lo stato della schermata di dettaglio di un luogo. All'interno della classe è contenuto lo stato dell'interfaccia grafica, che è uno `StateFlow`: la sua caratteristica è che ad ogni sua modifica l'interfaccia grafica viene aggiornata. La classe contiene anche metodi per modificare lo stato dell'interfaccia, dato che lo stato è mutabile solo all'interno della classe e all'esterno viene esposto solo lo stato in sola lettura.

A.6 Classi UseCase

```
@ViewModelScoped
class GetLocalListUseCase @Inject constructor(
    private val locationRepository: LocationRepository,
    private val savedLocationRepository: SavedLocationRepository,
    private val userRepository: UserRepository
) {

    suspend operator fun invoke(filters : PlacesListFilters) : Flow<List<PlaceListItemWithSavedModel>>
{

        return
combine(locationRepository.getLocationListStream(filters),userRepository.getLoggedUserFlow()){
            locStream, user ->
            locStream.map {
                PlaceListItemWithSavedModel(
                    place = it,
                    isSaved = if(user != null) savedLocationRepository.isLocationSaved(it.id) else
false
                )
            }
        }
    }
}
```

Le classi `UseCase` sono classi particolari che contengono una sola funzione: vengono usate quando un *ViewModel* richiede un'elaborazione dei dati più complessa o di dati che provengono da più repository diversi. La loro caratteristica è appunto di avere una sola funzione `invoke`, un operatore di *Kotlin*. In questo esempio il `UseCase` è usato per ottenere la lista dei luoghi, che è composta da dati provenienti da più repository, in questo caso il repository dei luoghi e quello degli utenti: viene usato il metodo `combine` di *Kotlin* per combinare i dati provenienti da più flussi in un unico flusso. L'annotazione `@ViewModelScoped` permette di iniettare la classe con

Hilt e di mantenere lo stesso oggetto per tutta la durata del ciclo di vita del *ViewModel*, sempre con *Hilt* vengono iniettati i repository con l'annotazione `@inject` nel costruttore.

A.7 Classe Database

```
@Database(
    entities = [
        LocationEntity::class,
        PhotosEntity::class,
        LinksEntity::class,
        HourRangeEntity::class,
        LoggedUserEntity::class,
        UserFavoriteLocations::class,
        UserFullNameEntity::class], version = 5)
abstract class SmartOfficesDatabase : RoomDatabase(){
    abstract fun locationDao() : LocationDao

    abstract fun userDao() : UserDao
}
```

La classe `SmartOfficesDatabase` estende `RoomDatabase` e rappresenta il database locale. Questa classe contiene i metodi per ottenere i DAO, in questo caso dalla classe si può richiedere due DAO, uno per accedere alle tabelle dei luoghi e uno per accedere alle tabelle degli utenti. L'annotazione `@Database` permette di definire le entità del database. La classe in sé è astratta, la classe concreta viene implementata tramite una build di *Gradle*.

A.8 Schema ER database locale

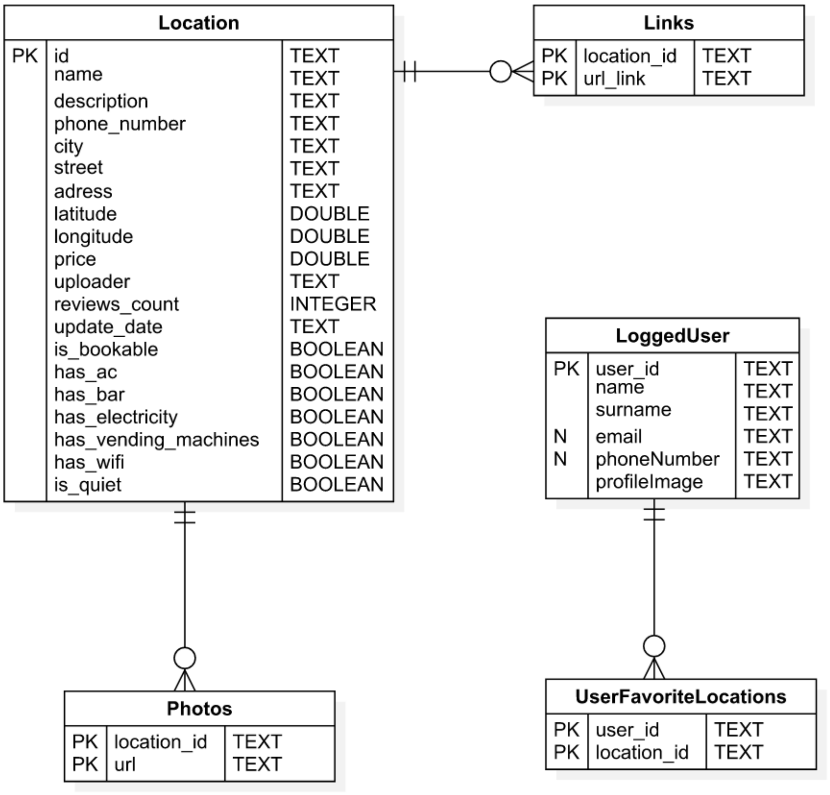


Figura 38: Schema ER del database locale.

Essendo un database relazionale, il database locale è composto da tabelle che sono collegate tra loro tramite chiavi esterne, queste tabelle sono state progettate per avere una struttura simile alle risposte JSON del API remota, dato che quando vengono prelevati i dati dal back-end remoto, questi vengono salvati nel database locale. in questo caso il database è composto da cinque tabelle, che sono:

- `location`: tabella che contiene i dati dei luoghi, con nome, descrizione, indirizzo, posizione geografica, contatti, caratteristiche, orari di apertura e numero di recensioni;
- `photos`: tabella che contiene i link delle immagini dei luoghi salvate su *Amazon S3*, una `location` ha una relazione uno a molti con `photos`;
- `links`: tabella che contiene i link a siti esterni, una `location` ha una relazione uno a molti con `links`;
- `LoggedUser`: tabella che contiene i dati dell'utente che ha effettuato l'accesso, cioè il nome, cognome, possibile e-mail, possibile numero di telefono e l'immagine del profilo;
- `UserFavoriteLocations`: tabella che contiene i luoghi preferiti salvati dall'utente, `LoggedUser` ha una relazione uno a molti con `UserFavoriteLocations`.

A.9 Interfacce DAO

```
@Dao
interface LocationDao {

    /*...

    */

    @Transaction
    @Query("""
        SELECT * FROM location
        ...query lunga...
         """)
    fun getLocationsStream(
        /**
        ...
        */
        ) : Flow<List<CompleteLocation>>

    @Transaction
    @Query("SELECT * FROM location WHERE uploader = :userId")
    fun getUserUploadedLocations(userId : String) : Flow<List<CompleteLocation>>

    @Transaction
    @Query("""SELECT * FROM location INNER JOIN user_favorites ON location_id = id WHERE user_id
= :userId """)
```

```
    fun getUserFavoriteLocations(userId: String) : Flow<List<CompleteLocation>>

    @Transaction
    @Query("SELECT * FROM location WHERE id IN (:ids)")
    fun getLocationsFromIds(ids : Set<String>) : Flow<List<CompleteLocation>>

    @Transaction
    @Query("SELECT * FROM location WHERE id = :id")
    fun getLocationById(id : String) : Flow<CompleteLocation>

    @Transaction
    @Upsert
    fun insertOrUpdateLocations(
        locations : List<LocationEntity>,
        hours : List<HourRangeEntity> = emptyList(),
        photos : List<PhotosEntity> = emptyList(),
        links : List<LinksEntity> = emptyList()
    )
}
```

Le interfacce DAO di *Room* vengono utilizzate per implementare il codice effettivo per fare le query dal database locale, il codice delle classi concrete viene generato con una build di *Gradle*. In questo caso la classe `LocationDao` contiene tutte le query necessarie per ottenere i dati dei luoghi. Le query sono scritte in SQL, ma *Room* permette di scrivere query più complesse, come query che ritornano più tabelle, usando l'annotazione `@Transaction` che permette di eseguire più query in una transazione. Le query ritornano un `Flow`, un flusso di dati che ha la caratteristica che, quando viene aggiornato, chi lo osserva viene notificato. Il tipo di query viene indicato con l'annotazione `@Query`, `@Insert`, `@Delete`, `@Update` e `@Upsert`, quest'ultimo effettua l'*Insert* se la chiave principale non è presente, altrimenti l'*Update*.

A.10 Classi API

```
class LocationApiService(private val ktor: HttpClient) {

    suspend fun getLocation(id: String): Result<PlaceDetailsResponse> {
        return try {
            val response = ktor.get("location/$id")

            when (response.status.value) {
                200 -> Result.success(response.body())
                422 ->
Result.failure(ValidationException(response.body<ValidationErrorResponse>().description[0]))
                else -> Result.failure(Exception(response.body<ErrorResponse>().message))
            }
        }catch (ex: Exception){
            Result.failure(ex)
        }
    }

    suspend fun getLocationList(
        /**
        ...
        */
    ): Result<PlacesListResponse> {
        try {
            val response = ktor.get("location/") {
                /**
                ...
                */
            }

            return when (response.status.value) {
                200 -> {
                    val responseBody = response.body<PlacesListResponse>()
                    Result.success(responseBody)
                }

                422 -> {
                    val responseBody = response.body<ValidationErrorResponse>()
                    Result.failure(ValidationException(responseBody.description[0]))
                }

                else -> {
                    val responseBody = response.body<ErrorResponse>()
                    Result.failure(Exception(responseBody.message))
                }
            }
        }catch (ex :Exception) {
            return Result.failure(ex)
        }
    }

    suspend fun uploadNewLocation(module: ResponseModule, token: String?): Result<SuccessResponse> {
        try {

            val response = ktor.post("location/") {
                contentType(ContentType.Application.Json)
                header(HttpHeaders.Authorization, "Bearer $token")
                setBody(module)
            }

            return when (response.status.value) {
                201 -> Result.success(response.body())
                422 ->
Result.failure(ValidationException(response.body<ValidationErrorResponse>().description[0]))
                else -> Result.failure(Exception(response.body<ErrorResponse>().message))
            }
        }catch (ex: Exception){
            return Result.failure(ex)
        }
    }
}
```

Le classi API sono classi che contengono metodi per effettuare chiamate al API remota. Il client *Ktor* fa una chiamata all'API remota con l'indirizzo del back-end più l'endpoint interessato. I metodi sono scritti in modo tale da ritornare un oggetto `Result` che è un oggetto che può contenere un valore o un'eccezione, sarà il chiamante poi a gestire i vari casi. Le chiamate sono scritte in modo da essere effetuate in modo asincrono. In questo esempio vengono gestite le chiamate sul endpoint `/location` che permette di ottenere i dati dei luoghi e di caricare nuovi luoghi: il primo metodo ottiene i dati di un luogo, il secondo ottiene la lista dei luoghi, il terzo carica un nuovo luogo, notare come per caricare un luogo bisogna accedere al proprio account così da poter usare un token valido. Le risposte vengono convertite in oggetti serializzabili e per inviare dati al back-end viene convertito un oggetto serializzabile in JSON. Come per i luoghi, anche per gli utenti esiste una classe API che contiene i metodi per effettuare chiamate al API remota sul endpoint `/user`.

Bibliografia

[1] Google, «Guide to app architecture». [Online]. Disponibile su: *https://developer.android.com/topic/architecture*

[2] Microsoft, «TypeScript». [Online]. Disponibile su: *https://www.typescriptlang.org/*

[3] JetBrains, «Kotlin». [Online]. Disponibile su: *https://kotlinlang.org/*

[4] Apple, «Swift». [Online]. Disponibile su: *https://www.apple.com/it/swift/*

[5] Google, «Jetpack Compose». [Online]. Disponibile su: *https://developer.android.com/jetpack/compose*

[6] Kent Beck, Robert C. Martin, Martin Fowler, e altri, «Manifesto for Agile Software Development». [Online]. Disponibile su: *https://agilemanifesto.org/*

[7] Gradle.inc, «Gradle». [Online]. Disponibile su: *https://docs.gradle.org/current/userguide/userguide.html*

[8] Google, «Material Design». [Online]. Disponibile su: *https://m3.material.io/*

[9] «Hilt». [Online]. Disponibile su: *https://dagger.dev/hilt/*

[10] JetBrains, «Ktor». [Online]. Disponibile su: *https://ktor.io/docs/welcome.html*

[11] Google, «Room». [Online]. Disponibile su: *https://developer.android.com/jetpack/androidx/releases/room*

[12] Amazon Web Services, «AWS Amplify». [Online]. Disponibile su: *https://docs.aws.amazon.com/amplify/*

[13] Amazon Web Services, «AWS Cognito». [Online]. Disponibile su: *https://aws.amazon.com/it/cognito/*

[14] «Android Studio». [Online]. Disponibile su: *https://developer.android.com/studio*