UNIVERSITÀ DI PISA

DIPARTIMENTO DI INFORMATICA

Corso di Laurea Triennale in Informatica

Refactoring e aggiornamento di un'applicazione mobile per l'agricoltura digitale basati sugli studi di usabilità.

Relatori:

Prof. Stefano Chessa

Prof. Paolo Milazzo

Controrelatore:

Prof. Tommaso Turchi

Candidata:

Sara Grecu

ANNO ACCADEMICO 2022/2023

Capitolo 1

Ringraziamenti

In questo momento speciale, desidero esprimere la mia sincera gratitudine a Stefano Chessa e Paolo Milazzo, i miei relatori di tesi, per la loro inestimabile disponibilità e generosità nel fornirmi consigli preziosi. Il loro supporto ha giocato un ruolo fondamentale nel raggiungimento di questo traguardo così importante. Ringrazio, inoltre, la dott.ssa Marina Buzzi dell' Unità Tecnologica Human-Centered Technologies, per i suoi preziosi consigli.

Vorrei manifestare la mia profonda riconoscenza alle mie care amiche Martina, Aurora, Mary e Giulia. Il loro sostegno ha avuto un impatto significativo nella mia vita, contribuendo in modo tangibile al mio percorso. Un caloroso ringraziamento va anche alla mia famiglia, sempre presente e solida nel sostenermi in ogni circostanza.

Inoltre, desidero ringraziare Giacomo: la sua costante presenza durante questi cambiamenti ha rappresentato un punto di riferimento cruciale, offrendomi il suo sostegno e aprendo la mia visione verso un mondo di infinite possibilità, dimostrandomi quanto la vita potesse offrire se solo avessi avuto il coraggio di abbracciare nuove opportunità.

Financial support has been provided by PRIMA, a programme supported by European Union Member States, Horizon 2020 Associated Countries and Mediterranean Partner Countries, in the context of the project "Optimization of Halophyte-based Farming systems in salt-affected Mediterranean Soils (HaloFarMs)".

Capitolo 2

Introduzione

L'agricoltura digitale, comunemente conosciuta come agricoltura 4.0, rappresenta l'adozione di tecnologie digitali nell'ambito agricolo. Queste tecnologie consentono agli agricoltori di raccogliere e analizzare dati in maniera più efficiente, prendere decisioni più informate e, di conseguenza, migliorare la produttività e la sostenibilità delle loro attività agricole.

Le tecnologie impiegate nell'ambito dell'agricoltura digitale comprendono l'Internet delle Cose (IoT), l'intelligenza artificiale, la robotica, l'analisi di grandi quantità di dati (Big Data) e l'utilizzo di droni.

Tra i vantaggi dell'agricoltura digitale rientrano l'aumento dell'efficienza, grazie alla raccolta e all'analisi più efficiente dei dati, che consente agli agricoltori di prendere decisioni più informate e migliorare la produttività. Queste tecnologie promuovono anche la sostenibilità, in quanto possono contribuire a ridurre l'uso di risorse, come acqua, fertilizzanti e pesticidi. Inoltre, favoriscono la produzione di prodotti di migliore qualità e più nutrienti.

L'agricoltura digitale è una tendenza in crescita rapida che sta rivoluzionando il settore agricolo. In Italia, sebbene sia ancora in fase di sviluppo, sta crescendo rapidamente.

Un rilevante progetto nell'ambito dell'agricoltura digitale è il progetto HaloFarMs [7], condotto dall'Università di Pisa in collaborazione con vari paesi, tra cui Tunisia, Italia, Egitto, Spagna, Francia e Portogallo. L'obiettivo di HaloFarMs è ottimizzare i sistemi agricoli e produttivi nella regione del Mediterraneo, sfruttando le piante alofite, che resistono al sale. Gli obiettivi principali includono l'aumento della produttività e della qualità delle colture commerciali su terreni salini, la riduzione della salinità del suolo, il ripristino della biodiversità attraverso la coltivazione di piante diverse e la diversificazione della produzione agricola [12], [14], [16], [15]. Questo mira a incrementare il reddito degli agricoltori, preservando l'ambiente e migliorando la resilienza del sistema alimentare globale.

Il progetto prevede l'implementazione di diverse innovazioni, tra cui la coltivazione congiunta o in rotazione di piante alofite e colture commerciali su terreni salini. Le piante alofite, in grado di assorbire e accumulare il sale, possono migliorare la

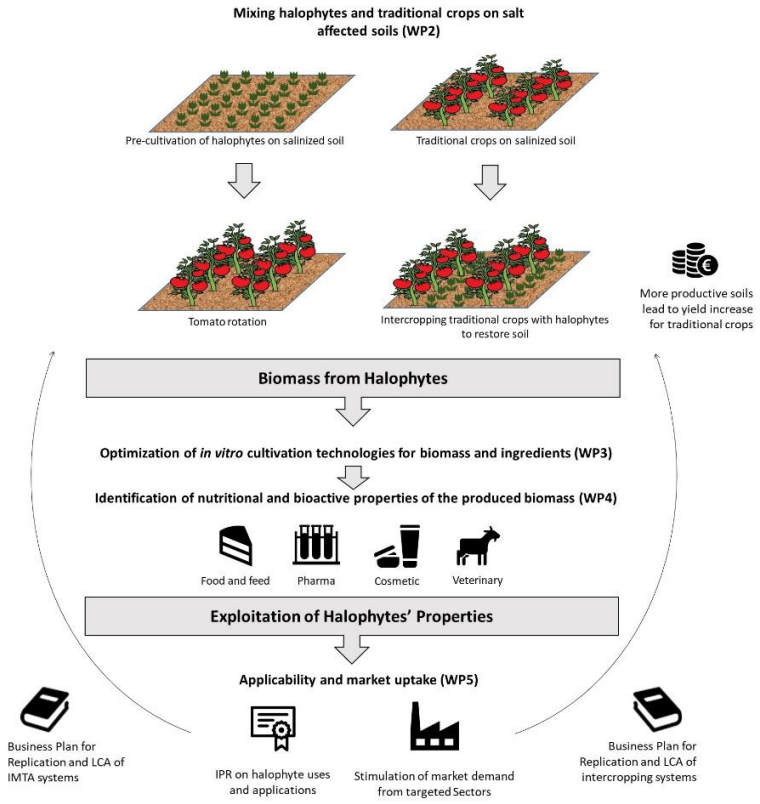

Figura 2.1: Struttura del progetto

crescita delle colture sensibili al sale. Una rappresentazione grafica del progetto è riportata nella figura 2.1.

Un elemento fondamentale del progetto è lo sviluppo di un'applicazione per smartphone che assiste gli agricoltori nella selezione del tipo di coltivazione più adatto alle condizioni del terreno. Questa app offre funzionalità come l'identificazione del perimetro dell'area da coltivare tramite il GPS, la determinazione dei punti di campionamento del terreno per le analisi, la creazione di una heatmap del terreno basata sui risultati delle analisi e il mantenimento di un database locale con backup in cloud per registrare lo storico delle analisi.

Questi compiti specifici non sono stati implementati in modo completo in alcuna delle app attualmente disponibili sul mercato. Il tirocinio si è concentrato sull'aggiornamento di un'applicazione preesistente e sui relativi test, nell'ambito del progetto *"PRIMA Section 2 call multi-topics 2019 Halofarms (Sviluppo e Ottimizzazione dei Sistemi Agricoli Basati su Alofite in Terreni Mediterranei Afflitti da Salinità)"*.

Capitolo 3

Stato dell'Arte

3.1 Architetture software

In questa Sezione verranno illustrate due architetture note nello sviluppo Android.

3.1.1 Clean architecture

La Clean Architecture (Architettura Pulita) è un approccio architetturale per la progettazione e lo sviluppo del software, ideato da Robert C. Martin intorno al 2012 [9]. L'obiettivo principale della Clean Architecture è di creare sistemi software che siano altamente modulari, indipendenti dalle tecnologie e facili da testare, manutenere e scalare nel tempo.

La Clean Architecture si basa su alcuni principi chiave:

- **Separazione delle responsabilità**: La struttura dell'applicazione è suddivisa in diverse "cerchie" concentriche o livelli, ognuna con un diverso grado di astrazione e responsabilità. Queste cerchie vengono organizzate in modo che le dipendenze vadano dall'esterno verso l'interno, permettendo a ciascun livello di conoscere solo ciò che è strettamente necessario.

- **Indipendenza dalle tecnologie**: La struttura dell'applicazione dovrebbe essere indipendente dalle tecnologie specifiche utilizzate. Questo significa che le decisioni tecniche, come la scelta del database o del framework di interfaccia utente, non dovrebbero influenzare la struttura dell'architettura.

- **Testabilità**: La Clean Architecture promuove l'alta testabilità del software. Poiché i livelli sono fortemente separati, è possibile testare ogni livello separatamente, sostituendo le dipendenze esterne con simulazioni o mock durante i test.

- **Scalabilità**: L'architettura è progettata in modo da poter affrontare facilmente i cambiamenti e crescere nel tempo. L'isolamento dei componenti e la chiara separazione delle responsabilità consentono di apportare modifiche senza influenzare l'intero sistema.

- **Chiarezza delle responsabilità**: Ogni livello dell'architettura ha un ruolo ben definito e specifiche responsabilità. Ad esempio, il nucleo dell'applicazione contiene le regole di business, mentre i livelli esterni si occupano delle interfacce utente, dei dettagli di infrastruttura e delle connessioni con il mondo esterno.

Gli elementi chiave della Clean Architecture includono:

- **Entità**: Rappresentano le regole di business e contengono la logica fondamentale dell'applicazione.
- **Casi d'uso**: Contengono le interazioni tra le entità e definiscono gli scenari di utilizzo dell'applicazione.
- **Interfacce utente**: Gestiscono l'interazione con l'utente e presentano i dati provenienti dai casi d'uso.
- **Framework e driver**: Contengono i dettagli tecnologici, come i framework web o i driver di database, e si trovano ai margini dell'architettura.

La Clean Architecture non è legata a un linguaggio di programmazione specifico e può essere implementata in vari ambienti, come applicazioni mobili, web o desktop. Tuttavia, richiede un impegno iniziale nella progettazione dell'architettura e nel mantenimento dei confini tra i vari livelli.

3.1.2 Architettura Model–View–Viewmodel

L' MVVM (Model–View–Viewmodel) è un pattern software architetturale o schema di progettazione software. È una variante del pattern "Presentation Model design" di Martin Fowler [4]. L' MVVM astrae lo stato della View e il comportamento. Mentre il modello di "presentazione" astrae una vista (crea un view model) di modo tale che non dipende da una specifica interfaccia utente.

Componenti del pattern:

- **Model**: Il Model è una implementazione del modello di dominio (domain model) della applicazione che include un modello di dati insieme con la logica di business e di validazione. Esempi di oggetti del modello comprendono repository, oggetti di business, oggetti di trasferimento dei dati (DTOs), Plain Old CLR Objects (POCOS), e oggetti generati di entità e proxy.
- **View**: La View è responsabile della definizione della struttura, il layout e l'aspetto di ciò che l'utente vede sullo schermo.
- **View Model**: Il View Model fa da intermediario tra la vista e il modello, ed è responsabile per la gestione della logica della vista. In genere, il View Model interagisce con il modello invocando metodi nelle classi del modello. Il View Model fornisce quindi i dati dal modello in una forma che la vista può usare facilmente.

- **Binder**: Il principale meccanismo di questo pattern assicura una costante sincronizzazione tra il View Model e la View, di solito attraverso una sintassi dichiarativa all'interno della View stessa. Questo significa che le modifiche ai dati fatte dall'utente tramite la View saranno automaticamente riflesse nel View Model, alleviando lo sviluppatore da questo compito. Allo stesso modo, qualsiasi modifica effettuata ai dati contenuti nel View Model verrà automaticamente visualizzata nella View.

In questo progetto è stata utilizzata il suddetto tipo di architettura.

3.2 Paradigma offline-first

Un'applicazione offline-first è un'applicazione in grado di svolgere tutta o una parte critica delle sue funzionalità principali senza accesso a Internet. In altre parole, può eseguire parte o tutto il suo codice di business offline.

Le considerazioni per la creazione di un'applicazione offline-first iniziano dal livello dei dati, che offre l'accesso ai dati dell'applicazione e alla business logic. L'applicazione potrebbe aver bisogno di aggiornare periodicamente questi dati da fonti esterne al dispositivo. In tal caso, potrebbe dover utilizzare risorse di rete per rimanere aggiornata.

La disponibilità di una connessione di rete non è sempre garantita. I dispositivi spesso hanno periodi di connessione di rete intermittente o lenta. Infatti, gli utenti potrebbero riscontrare i seguenti problemi:

- Banda internet limitata
- Interruzioni transitorie della connessione
- Accesso intermittente ai dati

Indipendentemente dalla ragione, spesso è possibile per un'applicazione funzionare adeguatamente in queste circostanze. Per garantire che l'applicazione funzioni correttamente offline, dovrebbe essere in grado di:

- Rimanere utilizzabile anche senza una connessione di rete affidabile.
- Presentare immediatamente agli utenti i dati locali anziché attendere il completamento o il fallimento della prima chiamata di rete.
- Recuperare i dati in modo consapevole dello stato della batteria e dei dati. Ad esempio, richiedendo il recupero dei dati solo in condizioni ottimali.

Un'applicazione che può soddisfare i criteri sopra descritti viene spesso definita un'applicazione offline-first.

3.2.1 Model data in una applicazione offline-first

Un'applicazione offline-first ha almeno due fonti di dati per ogni repository che utilizza risorse di rete:

- Fonte locale dei dati: è la fonte primaria di informazioni per un'applicazione e dovrebbe essere l'unica fonte da cui vengono letti i dati dai livelli superiori dell'applicazione. Questo assicura la coerenza dei dati tra i diversi stati di connessione. L'origine dati locale viene spesso supportata da uno storage persistente su disco e può includere sorgenti di dati strutturati come database relazionali e dati non strutturati come buffer di protocollo o file semplici.
- Fonte di rete: rappresenta lo stato effettivo dell'applicazione. Idealmente, l'origine dati locale e quella di rete dovrebbero essere sincronizzate, ma possono anche essere leggermente asincrone. In questo caso, l'applicazione deve essere aggiornata una volta che torna online. Al contrario, l'origine dati di rete può essere meno aggiornata rispetto a quella locale e l'applicazione deve essere in grado di aggiornarla quando viene ripristinata la connettività. Tuttavia, il dominio e i livelli dell'interfaccia utente dell'applicazione non devono interagire direttamente con il livello di rete. Questa è responsabilità del repository di hosting, che comunica con il livello di rete e lo utilizza per aggiornare l'origine dati locale.

3.2.2 Letture

La lettura dei dati è essenziale in un'applicazione offline-first e deve essere garantito che l'applicazione sia in grado di leggere e visualizzare i nuovi dati non appena sono disponibili. Un'applicazione reattiva rende questo possibile tramite l'utilizzo di API di lettura con tipi osservabili. Nel caso specifico riportato nel Listing 3.2.2, viene restituito un flusso per tutte le API di lettura, consentendo di aggiornare i lettori quando arrivano nuovi dati dalla fonte di rete. Questo significa che `OfflineFirstTopicRepository` può apportare modifiche quando la fonte di dati locale viene invalidata. I reader di `OfflineFirstTopicRepository` devono essere pronti ad affrontare queste modifiche quando viene ripristinata la connessione di rete. Inoltre, `OfflineFirstTopicRepository` legge direttamente i dati dalla fonte dati locale e può notificare i lettori solo dopo aver aggiornato i dati nella fonte locale.

3.2.3 Scritture

Mentre il modo consigliato per leggere i dati in un'applicazione offline-first è l'uso di tipi osservabili, l'equivalente per le API di scrittura sono API asincrone come le funzioni di sospensione. Questo evita di bloccare il thread dell'interfaccia utente e aiuta a gestire gli errori, perché le scritture nelle applicazioni offline-first possono fallire quando eccedono i limiti della rete.

```
class OfflineFirstTopicsRepository(
    private val topicDao: TopicDao,
    private val network: NiaNetworkDataSource,
) : TopicsRepository {

    override fun getTopicsStream(): Flow<List<Topic>> =
        topicDao.getTopicEntitiesStream()
            .map { it.map(TopicEntity::asExternalModel) }
}
```

Quando si scrivono dati in applicazioni offline-first, si possono prendere in considerazione tre strategie. La scelta dipende dal tipo di dati da scrivere e dai requisiti dell'applicazione:

Scritture online-only

Tentano di scrivereIn caso di successo, aggiorna l'origine dati locale, altrimenti lancia un'eccezione e lascia al chiamante il compito di rispondere in modo appropriato.

Questa strategia viene spesso utilizzata per le transazioni scritte che devono avvenire online quasi in tempo reale, ad esempio, un trasferimento bancario. Poiché le scritture possono fallire, è spesso necessario comunicare all'utente che la scrittura non è andata a buon fine, oppure evitare che l'utente tenti di scrivere i dati. Alcune strategie da adottare in questi scenari possono essere le seguenti:

- Se un'applicazione richiede l'accesso a Internet per scrivere i dati, può scegliere di non presentare all'utente un'interfaccia utente che gli consenta di scrivere i dati, o almeno di disabilitarla.
- È possibile utilizzare un messaggio pop-up che l'utente non può eliminare, o un prompt transitorio, per notificare all'utente che è offline.

Queued writes

Quando si dispone di un oggetto che si desidera scrivere, inserirlo in una coda. Procedere a svuotare la coda con un back-off esponenziale quando l'applicazione torna online. Su Android, svuotare una coda offline è un lavoro persistente che spesso viene delegato a WorkManager. Questo approccio è una buona scelta se:

- Non è essenziale che i dati vengano scritti sulla rete.
- La transazione non è time sensitive.
- Non è essenziale che l'utente sia informato se l'operazione fallisce.

I casi d'uso di questo approccio includono gli eventi di analisi e il log.

Lazy writes

Questo tipo di operazioni scrivono prima in locale, e poi mettono in coda le scritture da notificare alla rete il prima possibile. Questa operazione non è banale, poiché potrebbero esserci conflitti tra la rete e le fonti di dati locali quando l'applicazione torna online. Per approfondire la risoluzione dei conflitti si rimanda alla Sezione 3.2.4. Questo approccio è la scelta giusta se i dati sono critici per l'applicazione; ad esempio, in un'applicazione di elenchi di cose da fare offline, è essenziale che tutte le attività aggiunte dall'utente offline siano memorizzate localmente per evitare il rischio di perdita di dati.

3.2.4 Sincronizzazione e risoluzione dei conflitti

Quando un'applicazione offline-first ripristina la propria connettività, deve riconciliare i dati della propria sorgente dati locale con quelli della sorgente di rete. Questo processo è detto sincronizzazione. Esistono due modi principali in cui un'applicazione può sincronizzarsi con la propria origine dati di rete: sincronizzazione pull-based, sincronizzazione push-based e sincronizzazione ibrida.

Sincronizzazione pull-based

La sincronizzazione pull-based consiste nel recuperare i dati più recenti dalla rete solo quando necessario, come ad esempio immediatamente prima di presentarli all'utente. Questo approccio è particolarmente adatto per le applicazioni che devono gestire brevi o medi periodi di mancanza di connettività, in quanto gli aggiornamenti dei dati avvengono solo quando la connessione è disponibile. Tuttavia, se l'utente tenta di accedere alle risorse dell'applicazione durante un periodo prolungato di mancanza di connessione, potrebbe essere visualizzata una cache obsoleta o vuota.

Sincronizzazione push-based

Nella sincronizzazione push-based, l'applicazione riceve attivamente nuovi dati dalla rete tramite notifiche push quando la connessione è nuovamente disponibile. Questo approccio è efficace per le applicazioni che richiedono una sincronizzazione in tempo reale e che devono reagire immediatamente ai cambiamenti dei dati.

Sincronizzazione ibrida

Alcune applicazioni utilizzano un approccio ibrido basato su pull o push, a seconda dei dati. Ad esempio, un'applicazione di social media può utilizzare la sincronizzazione pull-based per recuperare il feed dei follower dell'utente su richiesta, a causa dell'elevata frequenza degli aggiornamenti del feed. La stessa applicazione può scegliere di utilizzare la sincronizzazione push per i dati relativi all'utente connesso, tra cui il nome utente, l'immagine del profilo e così via.

In definitiva, la scelta di quale tipologia di sincronizzazione utilizzare dipende dai requisiti del prodotto e dall'infrastruttura tecnica disponibile.

Conflict resolution

La risoluzione dei conflitti nelle applicazioni che scrivono dati offline spesso richiede il versionamento e l'uso di metadati per determinare quale dato è più recente. Un'approccio comune per la risoluzione dei conflitti è che "l'ultima scrittura vince", consentendo ai dati più recenti di essere accettati.

3.3 JetPack Compose

JetPack Compose è un moderno toolkit per lo sviluppo di interfacce utente Android native che semplifica ed accelera lo sviluppo grazie all'utilizzo del *declarative programming* per la creazione di UI [5]. Il declarative programming prevede che il programmatore debba semplicemente descrivere l'interfaccia utente e Compose si prenda carico del resto. Con Compose è possibile costruire componenti piccole e stateless che non sono vincolate a nessuna activity o fragment. Questo le rende piu' facili da riusare e testare.

Jetpack Compose si basa su funzioni componibili. Queste funzioni consentono di definire programmaticamente l'interfaccia utente dell'applicazione, descrivendone l'aspetto e fornendo le dipendenze dai dati, anziché concentrarsi sul processo di costruzione dell'interfaccia (inizializzazione di un elemento, collegamento a un genitore, ecc.) Per creare una funzione `composable`, basta aggiungere l'annotazione `@Composable` al nome della funzione.

In Compose, lo stato è esplicito e passato al *composable*, così che ci sia una *single source of truth*.

In questo progetto non si è scelto di utilizzare i tradizionali file XML, ma JetPack Compose, così da poter garantire maggior flessibilità ed espressività nell'interfaccia.

3.3.1 Accelerate development

Compose è integrabile anche con codice che non ne prevedeva il suo utilizzo inizialmente: infatti, si può chiamare del codice con Compose da Views e da Views chiamare del codice con Compose. La gran parte delle librerie, come quella di Navigazione, ViewModel e le coroutine sono compatibili con Compose, così che possa essere adottato nel momento in cui è piu' utile allo sviluppatore. Inoltre, Compose permette di creare applicazione con il built-in support per Material Design, Dark Theme, animazioni ed altre feature.

```
1     suspend fun fetchDocs() {
2         val result = get("https://unipi.it")
3         show(result)
4     }
5     suspend fun get(url: String) = withContext(Dispatchers.IO)
      { /* ... */ }
```

Listing 3.1: Esempio coroutine

3.4 Coroutine

Una coroutine è un design pattern per la gestione della concorrenza usato su Android per semplificare l'implementazione di codice eseguito asincronamente. Su Android, le coroutine aiutano a gestire task di lunga durata che potrebbero bloccare il main thread.

3.4.1 Feature

- **Lightweight**: è possibile eseguire piu' coroutine su un singolo thread grazie al suspension support, che consente di risparmiare memoria rispetto al blocco di un thread e supporta molte operazioni in parallelo;
- **Minori memory leak**: una coroutine usa la structured concurrency per eseguire operazioni in uno scope;
- **Built-in cancellation support**: la cancellation è propagata automaticamente all'interno della gerarchia della coroutine in esecuzione;
- **Integrazione a JetPack Compose**: all'interno di Compose vengono largamente usate le coroutine; alcune librerie forniscono addirittura il loro coroutine scope, così che possano essere usate per una structured concurrency.

Le coroutine si basano su due operazioni: suspend e resume. Eccone le caratteristiche:

- **suspend**, mette in pausa l'esecuzione della coroutine corrente, salvando tutte le variabili locali;
- **resume**, prosegue l'esecuzione di una coroutine sospesa dal luogo in cui è stata sospesa.

È possibile chiamare funzioni suspend solo da altre funzioni suspend o utilizzando `launch` per avviare una nuova coroutine. Il Listing 3.1 mostra una semplice implementazione di coroutine per un'ipotetica attività di lunga durata:

Nel Listing 3.1, la `get()` viene ancora eseguito sul **thread principale**, ma avvia la coroutine prima di avviare la richiesta di rete. Quando la richiesta viene completata, la `get()` ripristina la coroutine sospesa invece di utilizzare una callback per inviare una notifica al thread principale.

Kotlin utilizza uno stack frame per gestire quale funzione è in esecuzione con eventuali variabili locali. Quando una **coroutine** si sospende, lo stack attuale viene **copiato** e **salvato** per un secondo momento. Quando viene ripristinato, lo stack frame viene copiato dal punto in cui è stato salvato e la funzione viene avviata di nuovo. Anche se può sembrare che nel codice vi sia una richiesta bloccante, la coroutine assicura che tale richiesta di rete eviti di bloccare il thread principale.

3.4.2 Main-safety

Le coroutine Kotlin utilizzano i **Dispatchers** per determinare quali thread vengono utilizzati per l'esecuzione della coroutine. Per eseguire codice al di fuori del thread principale, lo sviluppatore può **specificare** su quale Dispatcher eseguirlo.

In Kotlin, tutte le **coroutine** devono essere eseguite in un **Dispatcher**, anche quando sono in esecuzione sul thread principale. Le Coroutine possono sospendersi e il Dispatcher è responsabile del loro ripristino.

Kotlin fornisce tre dispatchers:

- **Dispatchers.Main**: questo Dispatcher esegue una coroutine sul thread principale di Android. Deve essere utilizzato solo per interagire con l'interfaccia utente ed eseguire operazioni rapide. Alcuni esempi sono: chiamata di funzioni suspend, esecuzione di operazioni del framework dell'interfaccia utente Android e aggiornamento di oggetti LiveData.

- **Dispatchers.IO**: questo supervisore è ottimizzato per eseguire l'I/O del disco o della rete al di fuori del thread principale. Alcuni esempi sono l'utilizzo del componente Room, la lettura o la scrittura di file e l'esecuzione di qualsiasi operazione di rete.

- **Dispatchers.Default**: questo dispatcher è ottimizzato per eseguire operazioni che richiedono un uso intensivo delle CPU al di fuori del thread principale. Tra i casi d'uso di esempio ci sono l'ordinamento di un elenco e l'analisi di JSON.

Riprendendo il Listing 3.1, vengono utilizzati i dispatchers per ridefinire la funzione `get()`. Nel corpo di `get()`, viene chiamato `withContext(Dispatchers.IO)` per creare un blocco che verrà eseguito sul pool di thread dell'IO. Il codice inserito all'interno del blocco verrà eseguito sempre tramite il Dispatchers.IO. Poiché `withContext()` è una funzione suspend, anche la funzione `get()` lo è.

Con le coroutine è possibile avere un controllo dei thread **granulare**: poiché `withContext()` consente di controllare il pool di thread di qualsiasi riga di codice senza introdurre callback, si può applicare a funzioni molto brevi come la lettura da un database o l'esecuzione di una richiesta di rete. È buona norma utilizzare

```
suspend fun fetchDocs() {
    val result = get("unipi.it")
    show(result)
}

suspend fun get(url: String) =
    withContext(Dispatchers.IO) {
        /* perform network IO here */
    }
}
```

Listing 3.2: Esempio coroutine

`withContext()` per assicurarsi che ogni funzione abbia main-safety, il che significa che può essere chiamata dal thread principale. In questo modo, il chiamante non deve mai pensare esplicitamente a quale thread utilizzare per eseguire la funzione, ma questa operazione viene garantita dal dispatcher.

Nel Listing 3.2, `fetchDocs()` viene eseguito sul thread principale; tuttavia, può chiamare in modo sicuro `get`, che esegue una richiesta di rete in background. Poiché le coroutine supportano `suspend` e `resume`, la coroutine nel thread principale viene ripristinata con il risultato della `get()` non appena viene completato il blocco `withContext()`. Si noti che l'uso di `suspend` non indica a Kotlin di eseguire una funzione su un thread in background; anzi, è normale che le funzioni `suspend` vengano eseguite sul thread principale.

3.4.3 Avvio delle coroutine

Le coroutine possono essere avviate in due modi:

- **launch** avvia una nuova coroutine e non restituisce il risultato al chiamante. Qualsiasi attività "fire and forget" può essere avviata utilizzando launch.
- **async** avvia una nuova coroutine e consente di restituire un risultato con la funzione `await`. In genere, bisogna avviare una nuova coroutine da una funzione regolare, poiché questa non può chiamare la `await`. `async` si deve utilizzare solo all'interno di un'altra coroutine o all'interno di una funzione suspend ed esegui la scomposizione parallela.

3.4.4 Parallel decomposition

Tutte le coroutine che vengono avviate all'interno di una funzione suspend devono essere interrotte quando la funzione termina, quindi è necessario garantire il completamento di tali coroutine prima della fine. Con la **structured concurrency** in Kotlin, si può definire un `CoroutineScope` che avvia una o più coroutine. Quindi,

```
suspend fun fetchTwoDocs() =
    coroutineScope {
        val deferredOne = async { fetchDoc(1) }
        val deferredTwo = async { fetchDoc(2) }
        deferredOne.await()
        deferredTwo.await()
    }
```

Listing 3.3: Esempio coroutine

```
suspend fun fetchTwoDocs() =
    coroutineScope {
        val deferreds = listOf(
            async { fetchDoc(1) },
            async { fetchDoc(2) }
        )
        deferreds.awaitAll()
    }
```

Listing 3.4: Esempio coroutine

utilizzando `await()` (per una singola coroutine) o `awaitAll()` (per più coroutine), ci si può assicurare che queste coroutine terminino prima di ritornare al chiamante.

Ad esempio, nel Listing 3.3 un CoroutineScope che recupera due documenti in modo asincrono. Richiamando `await()` a ogni riferimento differito, garantiamo che entrambe le operazioni async vengano completate prima di restituire un valore. Oppure, può anche essere utilizzato `awaitAll()` per più coroutine, come illustrato nel Listing 3.4.

Anche se `fetchTwoDocs()` lancia nuove coroutine con async, la funzione usa `awaitAll()` per attendere che le coroutine lanciate vengano completate prima di tornare. Tuttavia, anche se non fosse stata chiamata la `awaitAll()`, il builder CoroutineScope non riprende la coroutine che ha chiamato `fetchTwoDocs` fino al completamento di tutte le nuove coroutine.

Inoltre, `CoroutineScope` intercetta le eventuali eccezioni lanciate dalle coroutine e le reindirizza al chiamante.

3.4.5 CoroutineScope

Un CoroutineScope monitora qualsiasi coroutine creata utilizzando `launch` o `async`. Le coroutine in esecuzione possono essere annullate chiamando `scope.cancel()` in qualsiasi momento. In Android, alcune librerie KTX forniscono i propri `CoroutineScope`

per determinate classi del ciclo di vita. Ad esempio, ViewModel ha un ViewModelScope e Lifecycle ha lifecycleScope.

Uno scope cancellato non può creare altre coroutine. Pertanto, `scope.cancel()` Si può chiamare solo quando la classe che ne controlla il ciclo di vita viene eliminata. Quando si utilizza ViewModelScope, la classe ViewModel annulla automaticamente lo scope nel metodo `onCleared()` di ViewModel.

3.4.6 Job

Ogni coroutine creata con launch o async restituisce un'istanza Job che identifica in modo univoco la coroutine e ne gestisce il ciclo di vita.

3.4.7 Coroutine context

Il CoroutineContext definisce il comportamento di una coroutine utilizzando il seguente insieme di elementi:

- **Job**: controlla il ciclo di vita della coroutine.
- **CoroutineDispatcher**: invia il lavoro al thread specificato.
- **CoroutineName**: il nome della coroutine, utile per il debug.
- **CoroutineExceptionHandler**: gestisce le eccezioni non rilevate.

Alle nuove coroutine create in uno scope, viene assegnata una nuova istanza Job e gli altri elementi CoroutineContext vengono ereditati dallo scope che li contiene. Si possono sostituire gli elementi ereditati passando un nuovo CoroutineContext alla funzione launch o async. Si noti che il passaggio di Job a launch o async non ha alcun effetto, poiché una nuova istanza di Job viene sempre assegnata a una nuova coroutine.

3.5 Room

La libreria Room, introdotta con Android Architecture Components, è una libreria dedicata alla persistenza su Sqlite, caratterizzata da un approccio O/RM [1]. Tecnicamente, si tratta di un abstraction layer, uno strato software che permette di sfruttare tutte le potenzialità del database senza la preoccupazione di affrontare ogni aspetto nei dettagli.

3.5.1 Gerarchia delle componenti

In un'applicazione Room-based, le componenti da implementare possono essere classificate in tre tipologie, come illustrato dalla figura 3.1: In particolare:

[1] https://it.wikipedia.org/wiki/Object-relational_mapping

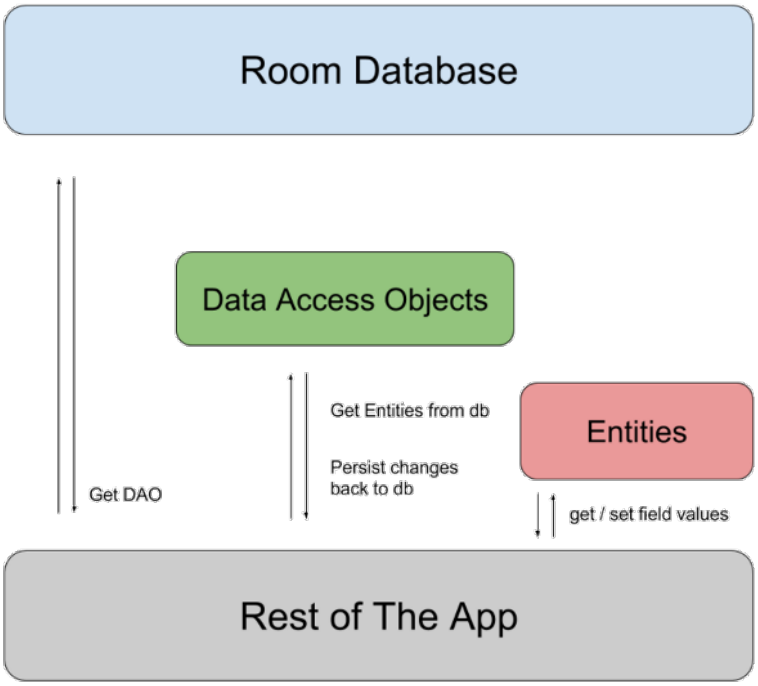

Figura 3.1: Room figure

- Entity: ogni classe di questo tipo rappresenta una tabella del database. All'interno delle classi Entity devono essere predisposte tante variabili d'istanza quanti sono i campi previsti dallo schema della tabella, più altri eventuali membri che si renderanno necessari;
- DAO (Data Access Object): un DAO viene utilizzato per incapsulare il codice che agirà sui dati, e conterrà i veri e propri comandi per le operazioni CRUD. Tipicamente esistono più DAO in un'applicazione, ma non è necessario ve ne siano tanti quante le Entity. In genere, ogni DAO raccoglie tutte le interazioni che riguardano un sottosistema del software;
- Room Database: rappresenta l'astrazione del database su cui si vuole lavorare. Verrà creata una classe di questo tipo che costituirà il centro di accesso ai dati: qualsiasi operazione da svolgere passerà da qui.

Vediamo alcune annotazioni interessanti e i relativi attributi:

- @Entity;
- @Dao;
- @Database;
- @PrimaryKey(autoGenerate = true), segnala che la variabile d'istanza è la chiave primaria, ed il flag `autogenerate` indica che è autogenerata;

- @Ignore, segnala che quel campo verrà ignorato da Room;
- @Query(...) indica una query in linguaggio sequel;
- @Delete, indica la query Delete;
- @Insert, indica la query Insert.

3.6 Soluzione realizzata in precedenza

Nel 2021 è stata sviluppata una precedente versione dell'app HaloFarms, che aveva obiettivi simili e molte delle funzionalità attualmente presenti. In questa sezione verranno dettagliate le tecnologie utilizzate in questa prima versione: Java, Firebase, file XML.

3.6.1 Strumenti Utilizzati

Questa versione è stata sviluppata usando il linguaggio di programmazione Java, insieme a file XML per la creazione dell'interfaccia. Inoltre, Firebase è stato utilizzato per gestire sia l'autenticazione degli utenti che la conservazione dei dati nel cloud. Si evidenzia che in questa versione non è stata implementata alcuna architettura specifica di sviluppo. Va sottolineato anche l'importante enfasi posta sull'esigenza di valutare approfonditamente l'usabilità.

Java Java è uno dei linguaggi principali utilizzati nello sviluppo di applicazioni per Android [3], insieme a Kotlin, che è stato proposto successivamente. L'IDE ufficiale per lo sviluppo Android, Android Studio, supporta sia Java che Kotlin.

Nel processo di sviluppo di un'app Android con Java, si fa uso del Software Development Kit (SDK) Android, che mette a disposizione una vasta gamma di strumenti e librerie per la costruzione di applicazioni Android [2]. L'SDK include una serie di classi e metodi Java che consentono di creare componenti dell'interfaccia utente, gestire l'input degli utenti, accedere all'hardware del dispositivo e molto altro.

Firebase Firebase appartiene alla categoria dei servizi online noti come "backend as-a-service," che mettono a disposizione, tramite API, servizi come autenticazione, archiviazione dati, notifiche push, comunicazione tra utenti e molto altro [10].

Uno dei principali benefici di Firebase è la sua capacità di fornire una sincronizzazione in tempo reale dei dati tra dispositivi e piattaforme diverse. Ciò implica che qualsiasi modifica apportata ai dati memorizzati in Firebase viene immediatamente riflesso su tutti i dispositivi connessi, agevolando la collaborazione senza interruzioni e l'accesso a informazioni sempre aggiornate [10].

Firebase offre anche una serie di funzionalità di sicurezza, tra cui autenticazione e autorizzazione, per garantire la protezione dei dati degli utenti. In aggiunta, Firebase

fornisce strumenti di analisi che assistono i developer nell'analisi del comportamento degli utenti e nell'ottimizzazione delle prestazioni dell'app.

In sintesi, Firebase costituisce una potente piattaforma che semplifica lo sviluppo delle app e fornisce un'infrastruttura scalabile per app in crescita. Le sue capacità di sincronizzazione dati in tempo reale risultano particolarmente utili per le app collaborative, mentre le funzionalità di sicurezza contribuiscono a proteggere i dati sensibili degli utenti.

File XML Nel processo di sviluppo per Android, l'utilizzo di file XML è estensivo e serve a configurare numerosi aspetti dell'applicazione, tra cui il manifesto, i layout, gli stili e le risorse grafiche. XML è un linguaggio di markup altamente versatile che consente ai developer di creare tag personalizzati e definizioni, rendendolo una scelta ideale per la definizione di layout e stili dell'interfaccia utente.

All'interno dell'ambiente Android, i file XML sono impiegati per definire la struttura e l'aspetto delle interfacce utente. I developer possono creare e modificare tali file XML per specificare il layout delle attività, dei widget e di altri elementi dell'interfaccia utente. Tra i tipi di file XML più comunemente utilizzati nello sviluppo Android troviamo:

- Il file del Manifest: Questo documento contiene informazioni essenziali sull'applicazione, come il nome, la versione, le autorizzazioni e l'icona.
- I file XML di Layout: Questi file definiscono il posizionamento degli elementi dell'interfaccia utente, come pulsanti, campi di testo, immagini e altri. I developer hanno la possibilità di creare e personalizzare i file XML di layout per organizzare gli elementi dell'interfaccia utente sulla schermata.
- I file XML di Stringhe: Questi documenti memorizzano i valori delle stringhe utilizzate nell'applicazione, come etichette dei pulsanti e voci di menu.
- I file XML di Stili: I file XML di stili consentono di definire temi personalizzati e stili per l'interfaccia utente, permettendo ai developer di modificare l'aspetto dell'applicazione in modo coerente.
- I file XML di Risorse Grafiche: Questi documenti contengono grafiche e animazioni utilizzate nell'applicazione, tra cui icone, sfondi e barre di avanzamento.

Capitolo 4

Progettazione

4.1 Requisiti iniziali dell'applicazione

L'obiettivo principale dell'app è agevolare gli agricoltori nella gestione e nella memorizzazione dei dati relativi ai campionamenti dei campi agricoli, permettendo di accedere alla cronologia di tali dati. Gli utenti possono aggiungere nuovi campi all'elenco, visualizzare un elenco di quelli già esistenti e selezionarne uno per visualizzarne i dettagli sulla mappa. Tramite l'applicazione è possibile disegnare il perimetro del campo, inserire manualmente i punti di campionamento oppure utilizzare una griglia automatica. È inoltre possibile assegnare uno stato di analisi (da analizzare/non da analizzare/analizzati) ai punti e, nel caso siano stati analizzati, inserire i relativi valori associati.

4.2 Studi sull'usabilità

Nella fase iniziale di questo studio di tesi, è stata condotta un'approfondita analisi sull'usabilità, con l'obiettivo di migliorare l'esperienza dell'utente. Durante questo processo, sono state consultate diverse fonti di informazione, che includevano i testi presenti in letteratura e riguardanti l'esperienza utente e il materiale fornito da Google.

Utente

L'utente di questa applicazione è un *agronomo*, quindi si assume che abbia una conoscenza base dei dispositivi Android e una conoscenza avanzata del dominio di applicazione. Data questa considerazione, è stata aumentata la chiarezza dei componenti di tutta l'applicazione [13]. Tale chiarezza va dal colore scelto dei pulsanti alla velocità di caricamento della lista dei campi, compresi i relativi punti di campionamento. Inoltre, considerando la possibilità che l'utente possa parlare francese tunisino, in base alla lingua impostata sul dispositivo, si procederà automaticamente al cambio di lingua da inglese a francese tunisino.

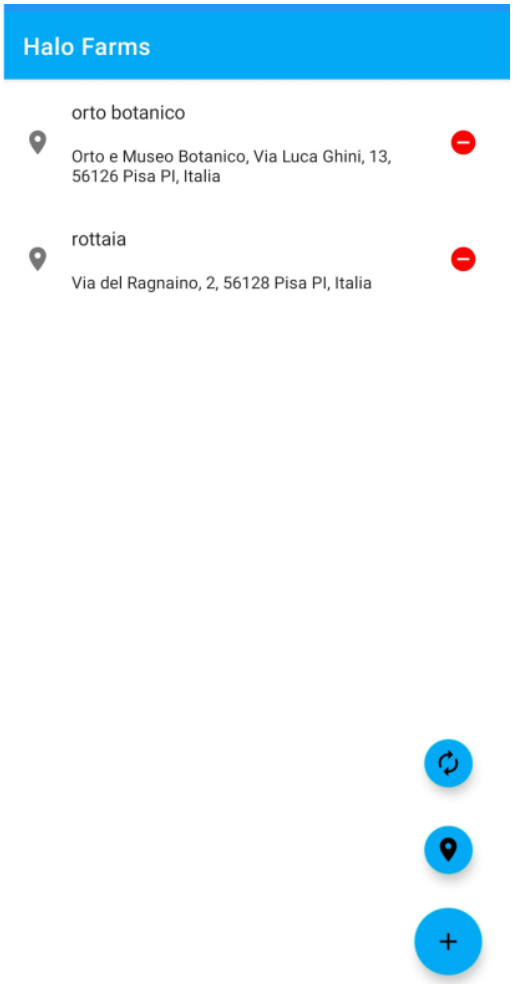

Figura 4.2: Home della versione precedente

Home

La schermata iniziale implementata nella versione precedente (figura 4.2) dell'applicazione è stata oggetto di modifiche come segue :

- ogni elemento presente nell'elenco dei campi è stato rappresentato in modo tale da garantire all'utente la chiara comprensione della sua interattività, evidenziando la possibilità di fare click su di esso;
- la sincronizzazione tra il dispositivo mobile e il servizio cloud si attua in maniera automatizzata, eliminando la precedente necessità di utilizzare il pulsante di `refresh`. Tale miglioramento è stato apportato al fine di agevolare e semplificare l'esperienza dell'utente;

halofarms

Figura 4.1: Logo

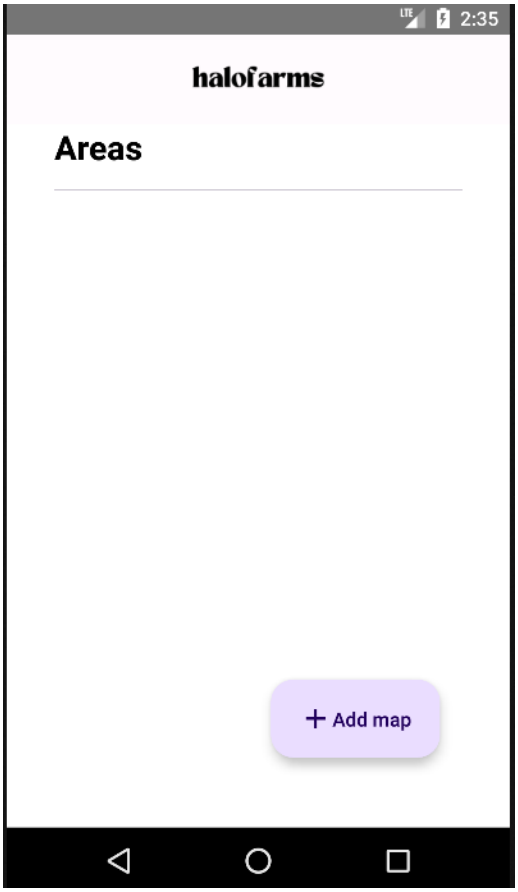

Figura 4.3: Home attuale (senza campi)

Figura 4.4: Home attuale

- per procedere all'eliminazione di un campo ora è richiesto un click prolungato su di esso, seguito dall'apertura di una finestra di dialogo che sollecita una conferma da parte dell'utente prima di completare effettivamente l'azione. È stato inoltre deciso di abbandonare l'opzione precedentemente disponibile, che consentiva di eliminare il campo attraverso un pulsante visibile, poiché tale pulsante è stato giudicato fuorviante nell'interfaccia e rendeva troppo facile l'eliminazione di un elemento;

- vi erano due pulsanti che prevedevano di aggiungere un campo: uno tramite GPS e l'altro tramite inserimento manuale dell'indirizzo, ma dato che entrambi eseguono azioni sostanzialmente simili, la presenza di due pulsanti con icone differenti e dimensioni variabili potrebbe risultare fuorviante, dando l'impressione che indichino azioni del tutto distinte. Pertanto, si è provveduto a unificarli in un unico pulsante, consentendo all'utente di effettuare la scelta desiderata in modo più chiaro.

Il risultato finale (figure 4.3, 4.4) presenta diversi elementi chiave: un logo iniziale, un chiaro testo esplicativo per guidare l'utente nella sua esperienza, e due colonne di pulsanti colorati in grigio per indicare la loro interattività, contenente il nome del campo associato.

Successivamente, troviamo il pulsante di aggiunta di un nuovo campo. La scelta di un colore differente per questo pulsante rispetto a quelli associati ai campi sottolinea chiaramente la distinzione nell'azione da compiere. L'uso del simbolo `+` e della scritta `Add` su questo pulsante offrono un'indicazione intuitiva della sua funzione di aggiunta.

La finestra di dialogo per l'aggiunta di un campo (figura 4.5) consente all'utente di specificare il nome desiderato per il campo e di decidere se utilizzare l'indirizzo GPS o inserirne uno manualmente.

Queste modifiche mirano a fornire un'esperienza utente intuitiva e personalizzata.

Map screen

Nella versione precedente dell'applicazione, la schermata della mappa presentava la mappa centrata sull'indirizzo specificato, inserito manualmente o tramite il GPS. In aggiunta, erano presenti vari pulsanti interattivi il cui significato poteva variare a seconda dell'azione dell'utente. Tuttavia, sono state apportate modifiche significative in questo contesto. La principale ragione di tali modifiche risiede nella complessità derivante dal fatto che il significato o il comportamento di un pulsante variava a seconda se l'utente ci passasse sopra col cursore, ci facesse click o lo mettesse a fuoco. Questa variazione poteva creare difficoltà agli utenti nel comprendere la funzione

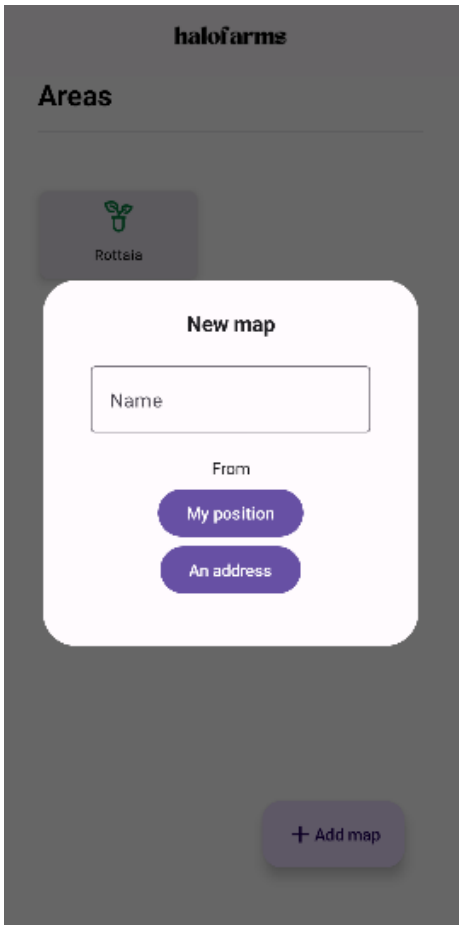

Figura 4.5: Finestra di dialogo di aggiunta di un campo

del pulsante o nel capire come interagire correttamente con esso. Come sottolineato da Nielsen [11], la coerenza è un principio fondamentale. Mantenere la coerenza nel comportamento dei pulsanti aiuta gli utenti a sviluppare la memoria muscolare e a interagire in modo intuitivo con le interfacce.

Inoltre, nella versione precedente, nell'angolo in alto a sinistra era presente un'icona ambigua che non risultava chiaramente interattiva e non rendeva evidente la possibilità di selezionare la visualizzazione tra i punti e la heatmap in base ai diversi valori (ad esempio, heatmap per l'EC, heatmap per il SAR, heatmap per il CEC, heatmap per il pH). Poiché la sua funzione non era immediatamente comprensibile, è stata apportata una modifica significativa per rendere più chiara e accessibile questa opzione, dato che risultava particolarmente difficile da interpretare.

Per navigare tra i vari timestamps, era necessario effettuare un clic prolungato su un campo (non risultava immediatamente evidente che fosse interattivo). Questa azione generava una schermata di dialogo contenente una data e un pulsante di rimozione. Poiché è stata implementata la funzione di eliminazione, si è deciso di spostare questa opzione direttamente nella schermata della mappa.

Pulsanti di interazione I pulsanti di interazione comprendevano opzioni di disegno (*Draw* o *Handfree*), la funzione di eliminazione della mappa, il pulsante QR code e il marker, che permette di inserire la posizione dell'utente tra i punti di campionamento. Come già accennato, l'inconveniente principale riguardava la dinamica di tali pulsanti, i quali mutavano il proprio significato in base alle azioni dell'utente [8]. Per esempio, il pulsante per la modalità di disegno *Handfree* veniva sostituito dal pulsante QR code al momento della selezione di un punto sulla mappa.

Di conseguenza, è stata implementata una soluzione (figura 4.6) che prevede:

- l'utilizzo di tre pulsanti di navigazione, finalizzati a semplificare la transizione tra diverse schermate, comprese quelle dedicate alla visualizzazione dei punti sulla mappa, alla heatmap e alla navigazione tra i timestamp;
- uno switch per cambiare modalità di disegno del perimetro;

La selezione della modalità di disegno, precedentemente gestita tramite i menzionati pulsanti di interazione, è stata ristrutturata per evitare l'ambiguità derivante da cambiamenti nel significato dei pulsanti. Ora tale selezione è stata spostata in un interruttore (`Switch`) posizionato nella barra superiore della schermata. Per una delle modalità, l'interruttore si colora di rosso, mentre per l'altra modalità assume una colorazione nera, al fine di garantire una chiara comprensione da parte dell'utente riguardo al cambiamento tra le due opzioni.

Navigazione tra timestamp La navigazione tra i diversi timestamp di una mappa è stata progettata per garantire un flusso di utilizzo lineare e coerente. Inizialmente, l'utente parte dalla schermata principale (*Home*) e poi può spostarsi nella schermata della mappa. Questa progettazione è stata adottata per assicurare che ciascuna schermata offra azioni pertinenti e specifiche solo a quella schermata.

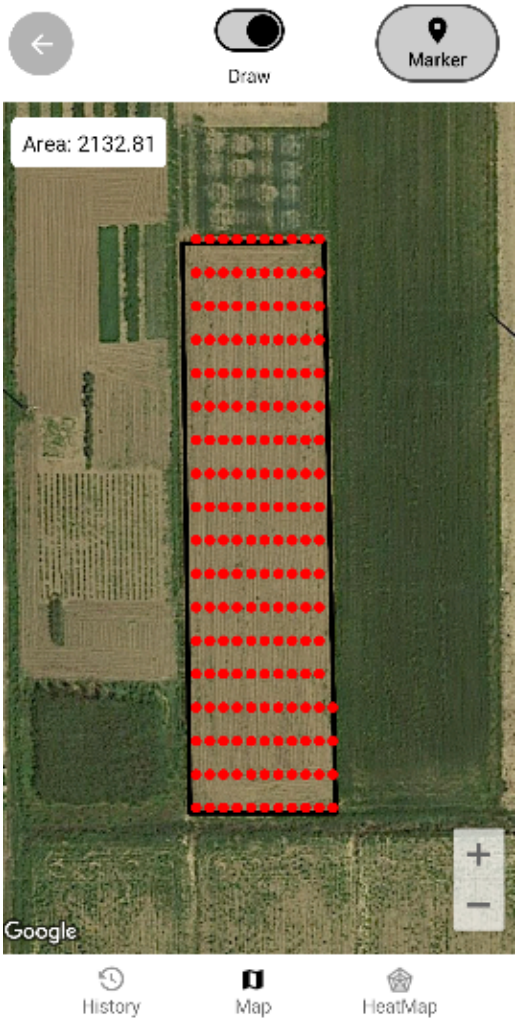

Figura 4.6: Esempio di una schermata di una mappa

Figura 4.7: Bottom bar menu per la navigazione all'interno della schermata della mappa

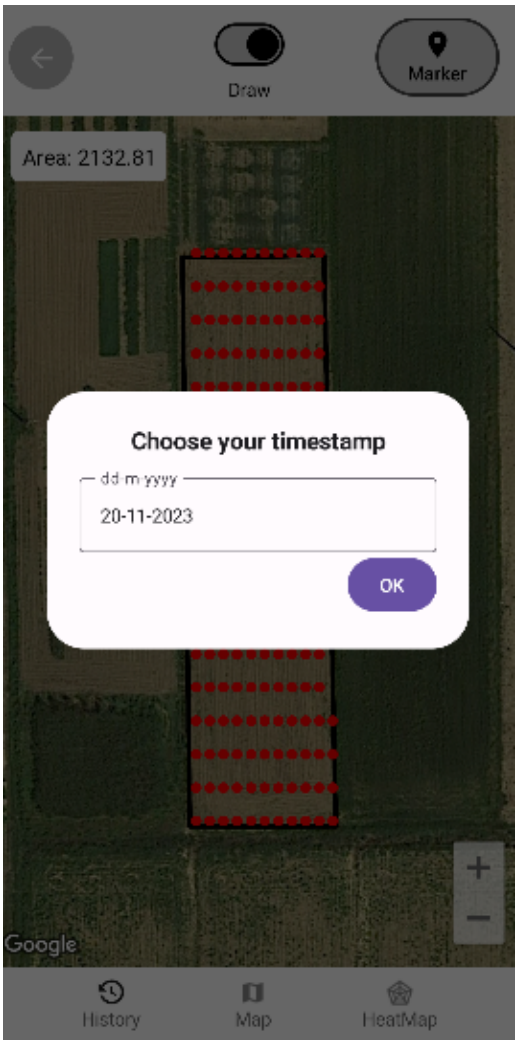

Figura 4.8: Timestamp menu

Figura 4.9: Esempio bottom sheet di Google Maps

Per semplificare la navigazione tra i diversi timestamp sulla mappa, è stato incluso un menu in basso (`bottom bar menu`, figura 4.7) che consente all'utente di selezionare un timestamp specifico. Questa selezione attiva automaticamente una finestra di dialogo che permette di decidere il timestamp desiderato (vedi Figura 4.8).

Finestra di dialogo informativa sui valori del punto selezionato La finestra di dialogo relativa alle informazioni associate a ciascun punto è stata aggiornata (figure 4.11, 4.10, 4.12) per assomigliare il più possibile al componente `sheet` presente in Google Maps (figura 4.9), al fine di presentare all'utente un elemento con cui è già familiare. A tale scopo, è stata progettata in modo tale che un pannello contenente dettagliate informazioni sul punto, tra cui il nome, lo stato di analisi, la necessità di analisi, i valori associati (nel caso di analisi completate) e il QR code corrispondente, emerga dalla parte inferiore della finestra di dialogo.

4.3 Flusso di utilizzo

4.3.1 Home

La Home è stata progettata in modo tale da mostrare all'utente tutte le mappe che aveva inserito in precedenza, e la possibilità di poterne aggiungere una nuova. In questo ultimo caso, l'applicazione offre all'utente la possibilità di scegliere come creare la mappa, se tramite GPS o inserendo un indirizzo. Se l'utente clicca su uno degli elementi della lista, verrà portato alla mappa corrispondente. Il flusso di utilizzo all'interno della Home è rappresentato nella figura 4.13.

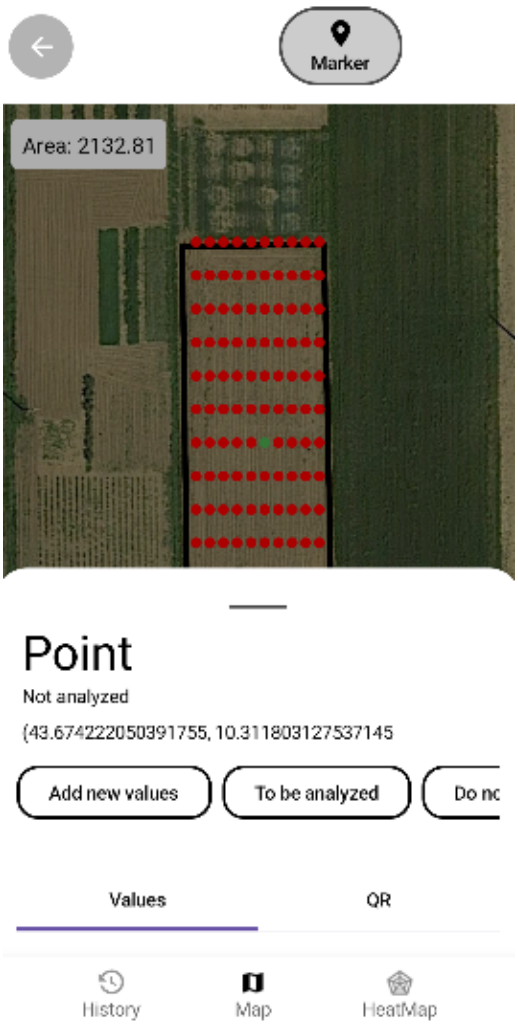

Figura 4.10: Attuale bottom sheet non totalmente espanso (punto non analizzato)

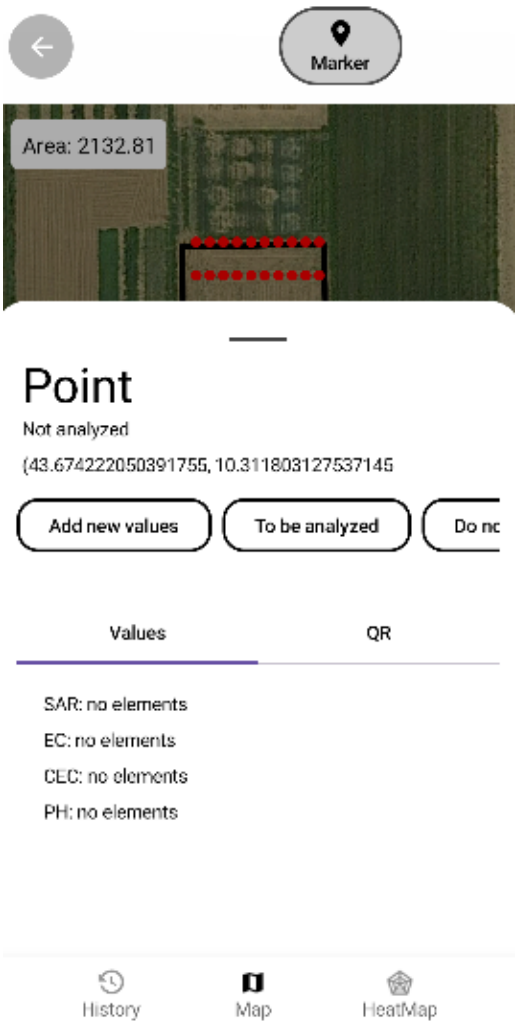

Figura 4.11: Attuale bottom sheet espanso (punto non analizzato)

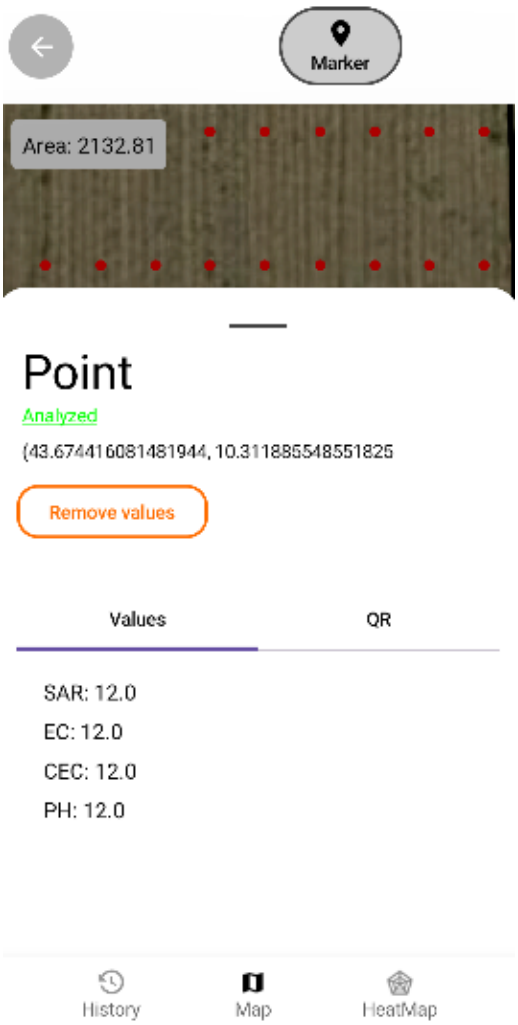

Figura 4.12: Attuale bottom sheet espanso (punto analizzato)

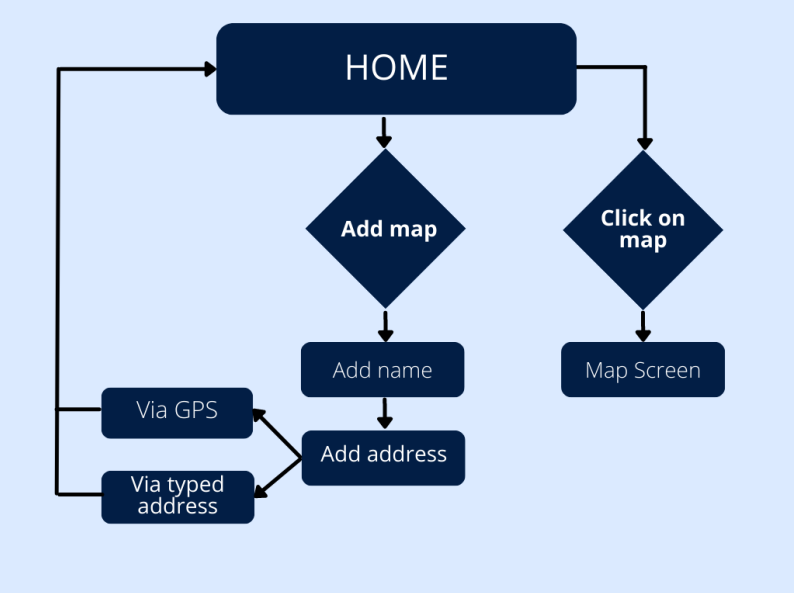

Figura 4.13: Home

4.3.2 Map Screen

La schermata della mappa (come mostrato nella figura 4.14) è stata progettata per consentire all'utente di tracciare il perimetro del campo desiderato dall'agronomo. Una volta completato, l'utente deve premere il pulsante `Done`. Successivamente, può aggiungere i punti di campionamento utilizzando due modalità: *Handfree* o *Draw*. La prima modalità consente all'utente di aggiungere un punto cliccando sulla mappa, mentre la seconda crea automaticamente una griglia di punti.

Inoltre, è possibile per l'utente aggiungere la propria posizione come punto sulla mappa tramite un pulsante aggiuntivo.

Ogni click su un punto mostra un pannello con tutti gli identificativi del punto selezionato: coordinate, valori attuali e QR code. In questo pannello, l'utente potrà decidere se il punto deve essere analizzato, se già lo è, aggiungere nuovi valori o rimuoverli se già presenti.

In aggiunta (come mostrato nella figura 4.15), è presente una barra inferiore che consente la navigazione tra la mappa e le relative heatmap, accompagnata da una finestra di dialogo che permette all'utente di selezionare il timestamp desiderato.

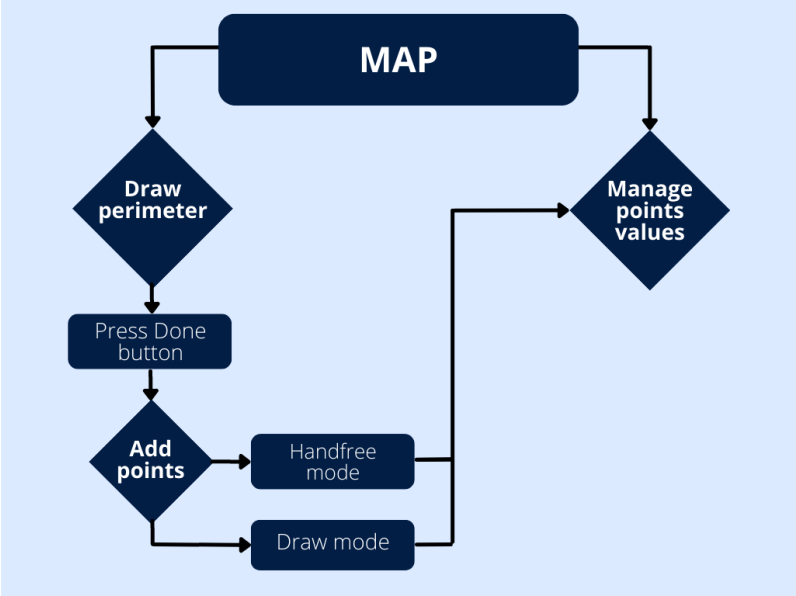

Figura 4.14: Map screen

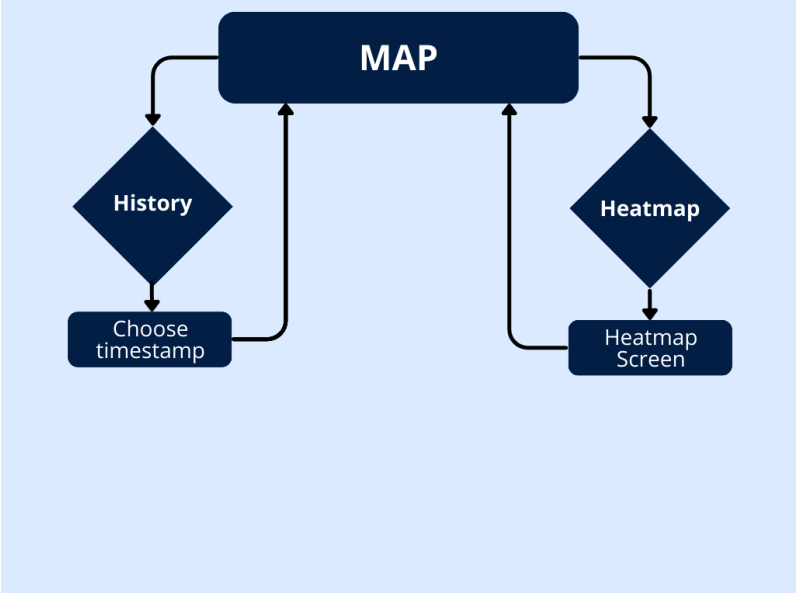

Figura 4.15: Bottom Bar Navigation

Capitolo 5

Implementazione

In questa Sezione vengono presentati i dettagli dell'implementazione di questa versione di HaloFarms. Come già scritto nel Capitolo 3 (Stato dell'arte), utilizza un'architettura MVVM (Model-View-ViewModel), dove sono presenti una Sezione riguardante la memorizzazione dei dati (Sezione database), una per la gestione della comunicazione con il cloud (paragrafo cloud) e una relativa agli screen presenti nell'applicazione. Successivamente, nelle sezioni seguenti, sono forniti dettagli su una serie di file, tra cui il file relativo alla registrazione e altri contenenti funzioni utili sia per l'interfaccia grafica che per l'implementazione dell'applicazione.

5.1 Database

Room [1] è stato impiegato come intermediario tra l'applicazione e il cloud, garantendo così un approccio *offline-first*. Questa scelta si traduce in una sincronizzazione più rapida dei dati e migliora la compatibilità dell'applicazione con potenziali modifiche ai server remoti in futuro.

La struttura `Map` rappresenta la mappa memorizzata e include diversi attributi di rilievo, quali il nome, le coordinate di latitudine e longitudine, la modalità di disegno dei punti di campionamento, l'area del campo, lo stato di completamento del disegno e la data associata.

- Il campo `Name` indica il nome assegnato dall'utente durante l'aggiunta ed è rappresentato come una stringa.
- Le coordinate GPS del campo sono contenute nei campi `latitude` e `longitude` e utilizzano il tipo *double*.
- Il campo `Mode` specifica la modalità di disegno dei punti di campionamento, che può essere *Draw* (disegno a griglia) o *Handfree* (disegno a mano), ed è rappresentato con uno *string*.
- `Area` rappresenta l'area del campo disegnato ed è un valore di tipo *double*.

- Il campo `Done` indica se il disegno del campo è stato completato ed è rappresentato da un valore booleano.

- `Date` rappresenta la data in cui il campo è stato disegnato, ma questa informazione verrà sovrascritta quando almeno un punto di campionamento verrà analizzato, riportando la data di tale analisi.

Per quanto riguarda l'oggetto `PerimeterPoint`, questo è caratterizzato da tre attributi principali. La latitudine è rappresentata con un valore di tipo *double*, così come la longitudine, mentre il nome del campo di riferimento è una stringa. La precisione delle coordinate GPS è fondamentale, in quanto consente di regolare la posizione del campo nel caso in cui l'utente utilizzi la propria posizione per il disegno del campo. Questo assicura che il campo venga disegnato in una posizione corretta, evitando errori di posizionamento rispetto alla propria ubicazione.

Nel caso dell'oggetto `Point`, esso è composto da quattro attributi principali: latitudine (*double*), longitudine (*double*), nome del campo di appartenenza (*string*) e un codice QR identificativo. Questi attributi forniscono informazioni essenziali sulla posizione del punto di campionamento e la sua relazione con un campo specifico.

Infine, l'oggetto `Sample` include diverse informazioni chiave: latitudine (*double*), longitudine (*double*), valori da analizzare o già analizzati (SAR, pH, EC, CEC - tutti rappresentati con *double*), data di analisi (*string*), uno stato booleano che indica se il punto richiede analisi e il nome del campo di appartenenza (*string*). Questi dettagli permettono di gestire in modo completo e accurato le informazioni relative ai punti di campionamento, compresa la possibilità di analizzare i dati raccolti.

A titolo di esempio, è presentata l'implementazione dell'oggetto `Sample` (listing 5.1) con le relative annotazioni necessarie per il suo utilizzo all'interno del database (listing 5.2).

La gestione della comunicazione con il database richiede la creazione di una classe oggetto (ad esempio, `Sample`) in cui il database stesso viene inizializzato. Inoltre, due ulteriori file, il `DAO` (Data Access Object) e il `Repository`, sono responsabili dell'implementazione delle funzioni di comunicazione con il database.

Una figura rappresentativa della struttura del database è presente nella figura 5.1.

5.2 Cloud data-store

Il data-store cloud utilizzato è Firestore [6], una componente di Firebase, e la comunicazione con esso avviene attraverso questa piattaforma. Si utilizzano diverse liste, ognuna dedicata a una specifica struttura dati: una lista per i campioni (Sample), una per i punti di perimetro (Perimeter Point), una per i punti valore (Point Value) e una per le mappe.

Di seguito, è presentata un esempio di funzione per l'aggiunta di una mappa (listing 5.3).

```
@Parcelize
@Entity(
    tableName = "samples",
    primaryKeys = ["latitude", "longitude", "date"]
)
data class Sample(
    @ColumnInfo(name = "latitude")
    var latitude: Double,
    @ColumnInfo(name = "longitude")
    var longitude: Double,
    @ColumnInfo(name = "sar")
    var sar: Double,
    @ColumnInfo(name = "ph")
    var ph: Double,
    @ColumnInfo(name = "ec")
    var ec: Double,
    @ColumnInfo(name = "cec")
    var cec: Double,
    @ColumnInfo(name = "date")
    var date: String,
    @ColumnInfo(name = "toBeAnalyzed")
    var toBeAnalyzed: Boolean,
    @ColumnInfo(name = "zoneName")
    var zoneName: String,
) : Parcelable {
    constructor() : this(0.0, 0.0, 0.0, 0.0, 0.0, 0.0, "null",
    false, "null")
}
```

Listing 5.1: Implementazione dell'oggetto Sample

```
@Volatile
private var INSTANCE: HaloFarmsDatabase? = null

fun getInstance(context: Context): HaloFarmsDatabase {
    synchronized(this) {
        var instance = INSTANCE

        if (instance == null) {
            instance = Room.databaseBuilder(
                context.applicationContext,
                HaloFarmsDatabase::class.java,
                "halofarms_database")
                .fallbackToDestructiveMigration()
                .build()

            INSTANCE = instance
        }
        return instance
    }
}
```

Listing 5.2: Funzione che crea il database

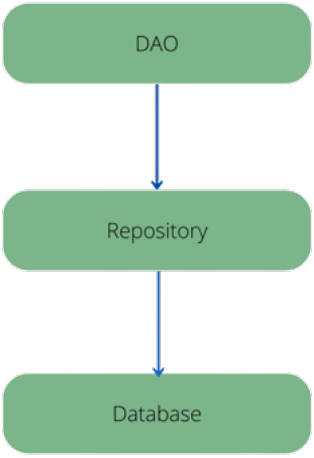

Figura 5.1: Schema database

```
    fun addMap(map: Map) {
        val z = hashMapOf(
            "name" to map.name,
            "latitude" to map.latitude,
            "longitude" to map.longitude,
            "mode" to map.mode,
            "area" to map.area,
        )

        // Sends the map to Firestore
        db().collection("$username-maps").document(map.name).
    set(z)
            .addOnSuccessListener {
                Log.d(TAG, "Map ${map.name} written")
            }
            .addOnFailureListener { e ->
                Log.w(TAG, "Error adding map ${map.name}", e)
            }
    }
```

Listing 5.3: Funzione che aggiunge una mappa su Firestore

5.3 Geolocalizzazione

Quando un utente aggiunge un campo, come accennato precedentemente, ha la possibilità di utilizzare sia il GPS sia di inserire manualmente un indirizzo. L'implementazione di entrambe le opzioni avviene in questo modo: per l'inserimento manuale dell'indirizzo, si è fatto uso dell'autocompletamento dei luoghi, un servizio fornito da Google che consente l'estrazione delle coordinate corrispondenti al luogo inserito [1]. Queste coordinate vengono quindi salvate all'interno di un oggetto Map. Nel caso del GPS, le coordinate vengono ottenute attraverso l'utilizzo della funzione `LocationServices.getFusedLocationProviderClient(context)`.

5.4 Permessi

I permessi richiesti all'utente per l'implementazione della geolocalizzazione sono stati gestiti con l'ausilio della libreria *Accompanist* [2]. Quest'ultima è stata sviluppata per arricchire Jetpack Compose con funzionalità non ancora disponibili ma comunemente richieste dagli sviluppatori.

[1] https://developers.google.com/maps/documentation/places/web-service/autocomplete?hl=it

[2] https://google.github.io/accompanist/

L'obiettivo di Accompanist è colmare le lacune presenti nel toolkit di Compose, sperimentare nuove API e acquisire esperienza nello sviluppo di librerie Compose. L'intento finale di queste librerie è l'integrazione nel toolkit ufficiale, momento in cui verranno deprecate e rimosse da Accompanist.

5.5 Registrazione

La registrazione dell'utente segue un'interfaccia predefinita fornita da Google (vedi codice nel listing 5.4). L'utente ha la possibilità di registrarsi utilizzando il proprio account Google a sua discrezione e otterrà un account su HaloFarms.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".register.RegisterActivity">

</androidx.constraintlayout.widget.ConstraintLayout>
```

Listing 5.4: Finestra di registrazione

5.6 Screen

La struttura della cartella dedicata alle schermate è divisa in due sezioni principali: *Home*, che comprende la schermata principale, e *Map*, che racchiude le schermate per la Heatmap, la Mappa, oltre al relativo View Model associato sia ad ogni mappa che a ciascun punto.

5.6.1 HomeScreen

Si tratta di una funzione `@Composable` che riceve lo stato dei permessi richiesti dall'app all'utente, l'elenco delle mappe presenti e una funzione che consente la navigazione verso una mappa al clic dell'utente. È costruita su uno `Scaffold`, utilizzando le componenti `topBar`, `floatingActionButton`, `floatingActionButtonPosition` e `containerColor`.

Per visualizzare l'elenco dei campi, viene utilizzata una `LazyVerticalGrid` che supporta due azioni per ogni elemento: `onLongPress` e `onTap`. L'`onLongPress` abilita la richiesta di rimozione di un campo, mentre l'`onTap` indirizza l'utente alla mappa selezionata. Ogni singolo campo è presentato come mostrato nel listing 5.5.

```
@Composable
fun MapView(
    map: Map,
    modifier: Modifier = Modifier
) {
    Column(
        modifier = modifier
            .clip(MaterialTheme.shapes.medium)
    ) {
        // Zone's icon
        Icon(
            painter = painterResource(id = R.drawable.provetta64),
            contentDescription = stringResource(R.string.zone_icon),
            modifier = modifier
                .align(Alignment.CenterHorizontally)
                .size(32.dp),
            tint = MediumGreen
        )
        // Zone's name
        androidx.compose.material.Text(
            map.name,
            style = MaterialTheme.typography.bodySmall,
            modifier = modifier.padding(8.dp)
        )
    }
}
```

Listing 5.5: Map View

5.6.2 MapScreen

La schermata della mappa ha il compito di rappresentare ogni elemento di una mappa, inclusi perimetro e punti con i loro dati associati. Si avvale di uno `Scaffold` che comprende la `bottom bar` per la navigazione tra le diverse visualizzazioni della mappa, un `floatingActionButton` e una `topBar`. I dati sono mostrati attraverso un `ModalBottomSheetLayout` che visualizza la mappa stessa e un foglio informativo per ogni punto cliccato.

La rappresentazione della mappa avviene mediante l'utilizzo della funzione composable [3]`GoogleMap()`, introdotta all'inizio del 2023 . Tra i suoi parametri rilevanti, troviamo `properties`, `cameraPositionState`, `uiSettings` e `onMapClick`. Le `properties` consentono di determinare la modalità di visualizzazione della mappa, ad esempio, è possibile scegliere la modalità satellitare. Il `cameraPositionState` definisce la posizione specifica in cui la mappa è visualizzata, mentre le `uiSettings` consentono all'utente di eseguire azioni come zoom e scorrimento della mappa. L'`onMapClick` rappresenta l'azione che avviene quando l'utente clicca sulla mappa.

Nel contesto di un clic sulla mappa, se il perimetro è già stato disegnato, il click può essere interpretato come un clic su un punto specifico oppure come un click neutro che non attiva nessuna azione aggiuntiva (nessun bottom sheet viene visualizzato). Al contrario, se il perimetro non è ancora stato definito, il clic viene considerato come la definizione di un punto che farà parte del perimetro stesso. Una parte dell'implementazione di questa funzionalità, escludendone i contenuti associati, è descritta nel listing 5.6.

All'interno di MapScreen, viene implementata una serie di funzioni, tra cui quella dedicata al disegno del perimetro (5.7), che include anche la logica per ordinare i punti cliccati dall'utente in modo che seguano sempre un ordine orario.

Modalità Draw

La modalità *Draw* consiste nel tracciare automaticamente una griglia di punti. Questi punti sono posizionati ad intervalli specifici, calcolati sulla base della distanza sferica tra il nord-ovest e il nord-est, divisa per un certo valore intero.

Il procedimento di creazione della griglia, riportata sotto forma di pseudocodice nell'Algoritmo 1, è il seguente:

1. Si calcola la distanza tra il nord-ovest e il nord-est (shortDistance);
2. Si calcola la distanza tra il nord-ovest e il sud-ovest (longDistance);
3. Si inizializzano due variabili: **left**, con il sud-ovest, e **right**, con il sud-est;
4. Se la latitudine di **left** è minore di quella del nord-ovest, si disegna una fila di punti separati da una certa distanza;

[3]https://googlemaps.github.io/android-maps-compose/maps-compose/com.google.maps.android.compose/-google-map.html

```
GoogleMap(
            modifier = Modifier.fillMaxSize(),
            properties = MapProperties(isMyLocationEnabled =
    true, mapType = MapType.HYBRID),
            cameraPositionState = cameraPositionState,
            uiSettings = MapUiSettings(compassEnabled = true,
    indoorLevelPickerEnabled = true, scrollGesturesEnabled =
    true, zoomGesturesEnabled = true),
            onMapClick = { latlng ->
                if (!map.done) {
                    mapsViewModel.addPerimeterPoint(latlng.
    latitude, latlng.longitude, map.name)
                } else if (PolyUtil.containsLocation(latlng,
    fromDbToLatLng(pPoints), false)) {
                    pointViewModel.addPoint(latlng.latitude,
    latlng.longitude, map.name, date)
                }
            }
        )
```

Listing 5.6: Google Map

```
for (p in perimeterPoints.toList()) {
        Marker(
            state = rememberMarkerState(position = p),
            visible = !done,
            draggable = true
        )

        if (perimeterPoints.size > 0) {
            area.doubleValue = String.format("%.2f",
    SphericalDrawUtil.computeArea(perimeterPoints)).toDouble()

            sortVertices(perimeterPoints)
            Polygon(
                points = perimeterPoints.toList(),
                fillColor = Color.Transparent,
                visible = true
            )
        }
    }
```

Listing 5.7: Parte della funzione che disegna il perimetro

5. Successivamente, si aumentano i valori di **left** e **right** utilizzando shortDistance e longDistance e si procede alla riga successiva, continuando fino a uscire dal poligono disegnato dall'utente.

Algorithm 1 Disegno di punti valore in modalità Draw

$shortDistance \leftarrow \text{DISTANCE}(nordOvest, nordEst)$;
$longDistance \leftarrow \text{DISTANCE}(nordOvest, sudOvest)$;
$left \leftarrow sudOvest$
$right \leftarrow SudEst$
while $left.latitude < nordOvest.latitude$ **do**
    Disegna una fila di punti separati da distanza pari a $shortDistance$;
    $left \leftarrow (left.latitude + longDistance, left.longitude)$;
    $right \leftarrow (right.latitude + longDistance, right.longitude)$;
end while

È importante notare che i punti vengono disegnati solamente se fanno parte del perimetro delineato dall'utente.

5.6.3 HeatMapScreen

La HeatmapScreen visualizza l'heatmap associata a un valore specifico (EC, SAR, PH, CEC) in un timestamp specifico. Per esempio, nel listato 5.8 è presente l'implementazione della HeatMap associata al valore EC.

5.6.4 MapViewModel e PointViewModel

All'interno dei View Model avviene l'interazione tra gli screen e il database. Qui vengono effettuate le richieste e mantenuto lo stato corrente della mappa (MapViewModel) e quello dei punti ad essa associati (PointViewModel).

5.7 Grafica

Per implementare l'interfaccia grafica dell'applicazione è stato utilizzato Jetpack Compose, il quale ha permesso di sviluppare in modo più efficiente numerose funzionalità.

Ad esempio, nel listato 5.9 è disponibile la visualizzazione dell'implementazione della funzione correlata allo stato di un punto, che indica se il punto è stato analizzato, è da analizzare o non è previsto per l'analisi.

```
@Composable
fun EcHeatMap(sample: Sample) {
    val data = LatLng(sample.latitude, sample.longitude)
    val startPoints = floatArrayOf(
        0.1f, 0.3f, 0.6f, 1f
    )

    val colors = setPointColor(sample = sample, mode = "ec")

    if(colors.size == 4) {
        val gradient = Gradient(colors, startPoints)
        val heatMapProvider = HeatmapTileProvider.Builder()
            .data(listOf(data))
            .gradient(gradient)
            .radius(50)
            .build()

        TileOverlay(tileProvider = heatMapProvider, visible =
    true, fadeIn = true)
    }
}
```

Listing 5.8: EC HeatMap

5.8 Testing

Inizialmente, sono stati condotti test sul campo utilizzando la precedente versione dell'app al fine di analizzare le esigenze degli utenti presso il Podere Rottaia (Via Ragnaino, 2 – San Piero a Grado, PI).

Successivamente, durante il processo di refactoring, sono stati effettuati test su diversi emulatori e dispositivi fisici, che spaziavano da Android 24 fino ad Android 33. Tra i dispositivi utilizzati vi erano Motorola G22, Pixel 4, Nexus 5, Nexus 5X, Pixel e OnePlus Nord.

5.8.1 Test grafici

Sono stati condotti test sui singoli componenti grafici utilizzando la funzionalità `@Preview` fornita da JetPack Compose, la quale consente la visualizzazione di un singolo componente dello schermo anche senza la presenza dell'intero layout.

```
1
2 @Composable
3 fun PointState(currentPoint: PointValue, samples: List<Sample
    >) {
4     val sample = samples.find { sample -> sample.latitude ==
    currentPoint.latitude && sample.longitude == currentPoint.
    longitude }
5     when {
6         sample == null -> {
7             Text(text = stringResource(R.string.not_analyzed))
8         }
9         sample.ph > 0.00 -> {
10            Text(
11                text = stringResource(R.string.analyzed_),
12                color = Color.Green,
13                textDecoration = TextDecoration.Underline
14            )
15        }
16        sample.toBeAnalyzed -> {
17            Text(text = stringResource(R.string.
    to_be_analyzed_))
18        }
19
20    }
21 }
```

Listing 5.9: Stato di un punto

Capitolo 6

Conclusioni

La realizzazione di questo progetto è iniziata con una revisione approfondita dell'applicazione precedentemente presentata, per identificare eventuali criticità e opportunità di miglioramento. Successivamente, è stato condotto un esame dettagliato dell'usabilità e dell'esperienza degli utenti, per comprenderne i requisiti. In parallelo, sono state esplorate le più recenti tecnologie emergenti nell'ambito dello sviluppo di applicazioni Android, per individuare soluzioni innovative che potessero migliorare la qualità del prodotto. In seguito, si è lavorato per integrare le tecnologie menzionate e gli studi di usabilità precedentemente svolti all'interno della soluzione software.

La stesura di questa tesi è stata un'esperienza incredibilmente formativa, poiché mi ha permesso di acquisire competenze sostanziali nell'implementazione di un'applicazione Android. Ho approfondito sia gli aspetti legati all'ingegneria del software e alla progettazione, sia quelli implementativi, aprendomi a numerose tecnologie nuove, alcune delle quali del tutto innovative e di frontiera.

Ho mirato a rendere il codice il più flessibile possibile, separandolo il più che potevo dalle specifiche tecnologie di Firebase. Questo approccio consentirà un agevole adattamento a eventuali cambiamenti nell'utilizzo di server differenti in futuro, riducendo al minimo le complicazioni e il vendor lock-in.

Ho altresì dedicato cura nel rendere il codice il più chiaro possibile, fornendo commenti e documentazione esaustiva. Il mio obiettivo è stato agevolare qualsiasi futuro programmatore nella comprensione, lettura e modificazione del codice senza intoppi.

Bibliografia

[1] Android Developers. Room Persistence Library, 2023. Accessed: November 2023.

[2] Android Developers. What is android. *Dosegljivo: http://www. academia. edu/download/30551848/andoid–tech. pdf*, 2011.

[3] Joyce Farrell. *Java programming*. Course Technology Press, 2015.

[4] Martin Fowler. The presentation model design pattern. *Martin Fowler.com*, July 19 2004.

[5] Google. Jetpack compose: A modern toolkit for building native android ui, 2021. Pagina Web.

[6] Ram Kesavan, David Gay, Daniel Thevessen, Jimit Shah, and C Mohan. Firestore: The nosql serverless database for the application developer. 2023.

[7] Alexander Kocian, Paolo Milazzo, Antonella Castagna, Annamaria Ranieri, José Antonio Hernandez, Pedro Diaz Vivancos, Gregorio Barba Espin, Karim Ben Hamed, Aida Selmi, Nesrine Kalboussi, and Stefano Chessa. Predictive model for the growth rate of tomatoes in saline substrate cultivation. In *IEEE International Workshop on Metrology for Agriculture and Forestry (MetroAgriFor)*, Pisa, Italy, November 6-8 2023.

[8] Steve Krug. *Don't Make Me Think! A Common Sense Approach to Web Usability*. New Riders Publishing, 2006.

[9] Robert C Martin. Clean architecture, 2017.

[10] Laurence Moroney and Laurence Moroney. The firebase realtime database. *The Definitive Guide to Firebase: Build Android Apps on Google's Mobile Platform*, pages 51–71, 2017.

[11] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann Publishers, 2010.

[12] C. Oliveira and et al. Can saline irrigation improve the quality of tomato fruits? *Agronomy Journal*, 114(2):900–914, 2022.

[13] John Smith and Mary Johnson. Usability and interface design: A tutorial. *Interactions*, 14(4):20–23, 2007.

[14] C. Sonneveld and G. Welles. Yield and quality of rockwool-grown tomatoes as affected by variations in ec-value and climatic conditions. *Plant and Soil*, 111(1):37–42, 1988.

[15] I. Tüzel, Y. Tüzel, A. Gül, and R. Eltez. Effects of ec level of the nutrient solution on yield and fruit quality of tomatoes. *Acta Horticulturae*, 559:587–592, 2001.

[16] H. Yang, T. Du, X. Mao, and M. K. Shukla. Modeling tomato evapotranspiration and yield responses to salinity using different macroscopic reduction functions. *Vadose Zone Journal*, 19(1), 2020.