Kotlin-Multiplatform applicato allo sviluppo di applicazioni Android e iOS

Ingegneria e scienze informatiche magistrale Cesena

Relatrice: prof. Antonella Carbonaro

Laureando: Andrea Procucci

7 marzo 2021

Sommario

Questa tesi ha il compito di determinare se *Kotlin-Multiplatform*, per la progettazione e lo sviluppo di un'applicazione sia *Android* che *iOS*, allo stato attuale della tecnologia può portare il risultato aspettato. L'obiettivo della tesi è replicare l'applicazione *Android* di gestione del magazzino *Warehouse* preesistente utilizzando *Kotlin-Multiplatform*.
Gli argomenti sono suddivisi nel seguente modo, un primo capitolo introduce il perché continuano ad essere ricercati e sviluppati framework per realizzare applicazioni *Android* e *iOS* che riducano i tempi di sviluppo e di manutenzione. Con il capitolo di "Introduzione" viene presentata la tecnologia utilizzata assieme all'obiettivo della tesi e a una descrizione completa del caso di studio. Nel capitolo di "Progettazione" e in quello di "Implementazione" si argomentano le scelte di progettazione e come sono state realizzate nella pratica.
Le applicazioni sviluppate sono state svolte per l'azienda *Twinlogix* con il fine di realizzare la versione *iOS* dell'applicazione *Warehouse* che non possedevano.

Indice

1. L'ecosistema delle applicazioni

1.1 Un po' di dati

Con l'incremento del numero di persone in possesso di dispositivi mobile, dovuto anche ad un incremento della loro potenza computazionale che ha portato una riduzione del costo di accesso, il mondo delle applicazioni continua a prosperare. Come mostrato in figura 1.1, sono 3,5 miliardi i possessori di smartphone nel 2020 e si prospetta un incremento di 0,3 miliardi nel prossimo anno.

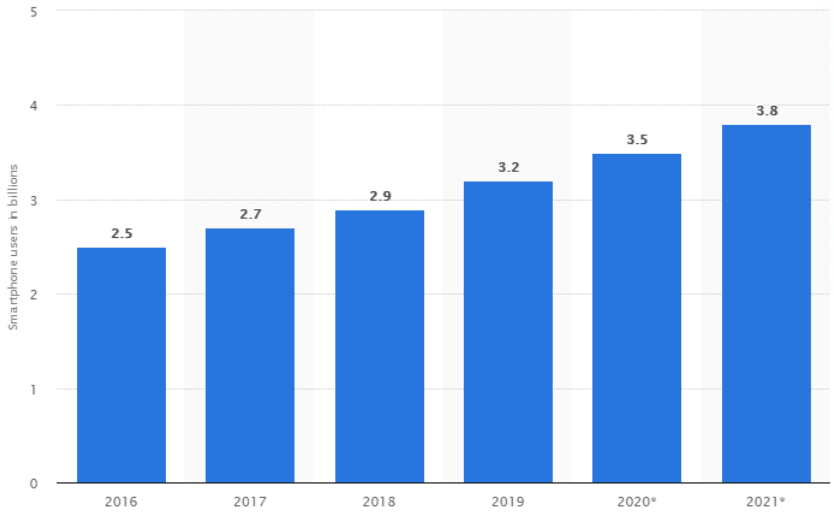


Figura 1.1: Numero di possessori di smartphones e crescita prevista per il prossimo anno. Fonte *statista.com*.

Tutto ciò non fa altro che aumentare l'importanza delle applicazioni e come mostrato nella figura 1.2, i ricavi dei due maggiori store di applicazioni continuano ad aumentare. Rimane anche pressoché costante il rapporto dei guadagni dello store *Apple* e dello store *Google*.

Tra i guadagni sono presenti anche quelli del mondo del gaming che, anche grazie al miglioramento delle prestazioni degli smartphones, ha permesso lo sviluppo di videogiochi. Questi tramite pubblicità, acquisti in app e l'acquisto del videogioco stesso hanno fatto

incrementare il ritorno economico. L'esempio più eclatante del periodo è *Genshin Impact* che solo per il settore mobile in circa 2 mesi ha prodotto 245 milioni di dollari.

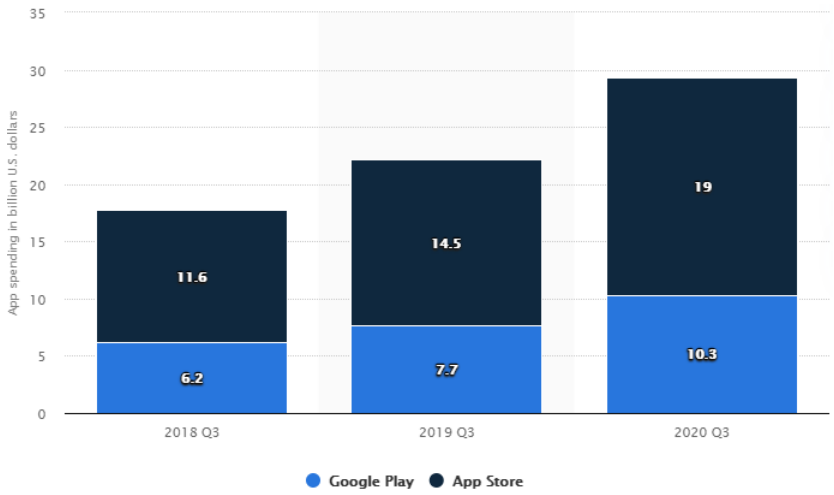


Figura 1.2: Guadagni dei due principali store di applicazioni. Fonte *statista.com*.

Oggigiorno esistono applicazioni per ogni cosa, da applicazioni per la consegna del cibo a domicilio ad altre che ci permettono di controllare la casa domotica. Resta il fatto che tutto ciò contribuisce all'aumentare l'utilizzo degli smartphone che va a sua volta a incrementare l'interesse delle aziende di far parte di questo mercato.
In linea con questo trend anche *Twinlogix* già dagli albori non ha potuto non sfruttare questa opportunità andando a sviluppare applicazioni, principalmente per il mercato *Apple*, senza però tralasciare il mondo Web anch'esso in crescita.

1.2 Piattaforme mobile

Come anticipato nella sezione precedente oggi esistono solo due piattaforme, i dispositivi *Apple* con sistema operativo *iOS* e i dispositivi con sistema operativo *Android*. Nel momento in cui si decide di voler realizzare un'applicazione è necessario come prima cosa scegliere per quale piattaforma svilupparla. Se si scegliesse *iOS* ci si troverebbe con un minor quantitativo di dispositivi ma con un insieme di utilizzatori disposto a pagare di più, al contrario scegliendo *Android* si avrebbe accesso a un maggior numero di persone rinunciando però ai guadagni maggiori di *iOS*.
Ovviamente nulla vieta lo sviluppo dell'applicazione per entrambe le piattaforme.

1.3 Approcci per lo sviluppo

Scegliendo di realizzare l'applicazione per entrambe le piattaforme è necessario stabilire che approccio utilizzare per lo sviluppo. Le scelte sono tra:

- Approccio nativo, lo sviluppo dell'applicazione *iOS* può essere svolto in parallelo con lo sviluppo dell'applicazione *Android*. É la modalità di sviluppo base nella quale vengono definiti due progetti differenti, ciò però comporta la duplicazione di codice tra le due applicazioni. Si hanno benefici in termini prestazionali in quanto le due applicazioni vengono realizzate utilizzando linguaggio e strumenti di sviluppo propri della piattaforma.

- Approccio multi-piattaforma, l'idea alla base di questo approccio è tentare di rimuovere la duplicazione di codice tra le due applicazioni. Ciò rende più semplice la manutenzione delle applicazioni e potenzialmente riduce anche i tempi di sviluppo. Lo svantaggio principale è che ancora non esistono veri approcci multi-piattaforma che lascino allo sviluppatore la libertà e il livello di prestazioni che un approccio nativo fornisce.

- Approccio Web, rappresenta un'alternativa ai due casi precedenti, vengono utilizzati linguaggi e strumenti di sviluppo del mondo Web per produrre una soluzione che ricordi l'utilizzo di un'applicazione. Ciò è possibile perché negli ultimi anni è aumentata la facilità di utilizzo delle funzionalità degli smartphone anche da applicazione Web. Anche il livello di prestazioni è ottimo in quanto la parte computazionale viene svolta lato server. Uno dei fattori più importanti che le separa dall'essere applicazioni native è la necessità di utilizzare un *browser* per potervi accedere.

- Approccio *Progressive Web Application*, si intende la volontà di produrre applicazioni web per il mondo mobile che rispecchino le applicazioni native in tutti gli aspetti come: prestazioni, *User eXperience*, integrazione con il dispositivo e affidabilità d'uso anche in assenza di connessione o con connessioni instabili. Vanno a rimuovere la necessità di utilizzare un *browser* per accedere alle loro funzionalità.

- Approccio tramite *WebView*, con questo approccio si va a unire il mondo native con quello delle web app. Il funzionamento alla base è il seguente, lo sviluppatore realizza l'applicazione utilizzando linguaggi e strumenti del mondo Web. Dopodiché tramite

le *WebView* (componenti nativi alle piattaforme) vengono visualizzate all'interno dell'applicazione le pagine Web precedentemente create.

2. Introduzione

2.1 Contesto aziendale

TwinLogix è un'azienda che dal 2009 si è concentrata sullo sviluppo di nuove tecnologie per il mercato Mobile, questo le ha permesso oggi di vantare la realizzazione di soluzioni leader in ambito sanitario e lo sviluppo di una delle App di Mobile Banking più apprezzate nei mercati Apple e Android.

2.2 Kotlin multiplatform

Attualmente quando è necessario effettuare lo sviluppo di un'applicazione che sia utilizzabile su dispositivi *Android* e *iOS* è necessario realizzare due diverse soluzioni per le due piattaforme. Ciò si traduce nella necessità di avere due risorse, ognuna specializzata in una piattaforma, che sviluppano in parallelo le applicazioni.
Il problema maggiore di questo approccio è che la maggior parte del codice che viene prodotto dalle due risorse risulta essere duplicato, con tutti gli effetti negativi che ne conseguono.
Per risolvere questo problema sono nate diverse soluzioni ma *Kotlin Multiplatform*, che da adesso per comodità verrà chiamato *KMP*, permette di scrivere una volta sola la business logic lasciando alle singole piattaforme lo sviluppo della *User Interface*. *KMP* raggiunge rispetto ai competitor delle buone performance rimuovendo la necessità di avere livelli intermedi tra la business logic e i progetti delle singole applicazioni.
Di fatto la parte comune di business logic viene, lato *Android* compilata come .jar e importata nel progetto, mentre lato *iOS* viene compilato un *Native framework* con header *Objective-C*.
Riassumendo, utilizzare *KMP* permette di risolvere il problema della duplicazione del co-

dice mantenendo lo sviluppo delle *UI* separato, ciò permette agli sviluppatori di realizzarle utilizzando le specifiche funzionalità grafiche delle diverse piattaforme.

2.3 Altri framework

Le principali alternative a KMP, restando nella stessa ottica di sviluppo, sono *Xamarin* e *Flutter*. Come mostrato in figura 2.1, nonostante la sua attuale instabilità, KMP ha già iniziato ad essere utilizzato per lo sviluppo di applicazioni *cross-platform*.

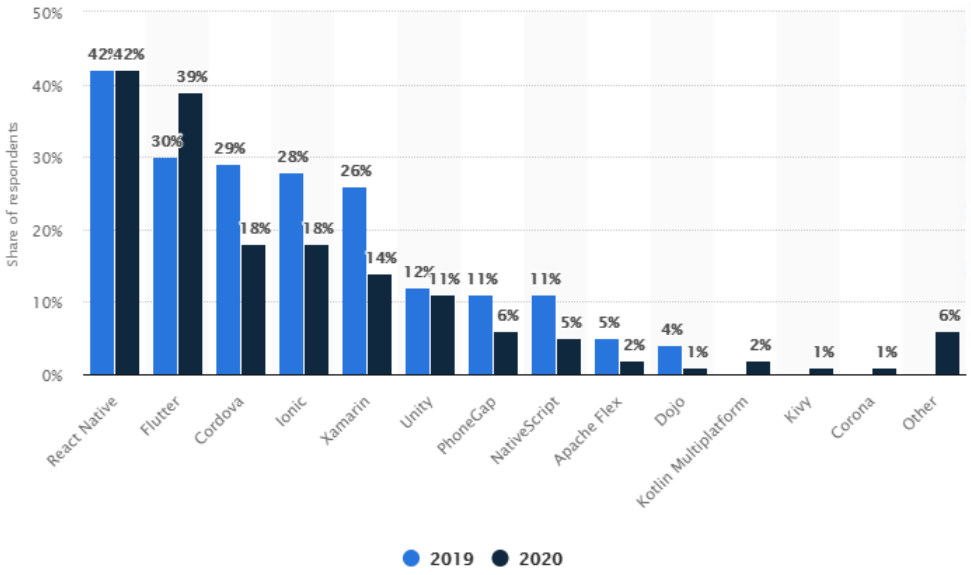


Figura 2.1: Grafico che illustra la percentuale di utilizzo dei framework *cross-platform* da parte degli sviluppatori. Fonte *statista.com*.

2.3.1 Flutter

Flutter è un SDK mobile opensource utilizzabile per la creazione di applicazioni *cross-platform Android* e *iOS*. Queste mantengano un'interfaccia simile a una soluzione native ma possiedono una stessa base di codice condiviso. Il linguaggio utilizzato è *Dart* che viene poi compilato in codice per le diverse piattaforme native.

L'architettura di *Flutter* come mostrato in figura 2.2, si compone di tre livelli ognuno dei quali ha delle librerie interne indipendenti e rimpiazzabili.

Il livello *embedded* è il livello che deve interagire con il sistema operativo della piattaforma e viene scritto in un linguaggio appropriato ad essa.

Il livello intermedio è l'*engine* di *Flutter*, scritto in C++ fornisce le primitive necessarie

per supportare le applicazioni *Flutter*.

Il livello con il quale in genere gli sviluppatori interagiscono con *Flutter* è il *framework* scritto in *Dart*. Fornisce un insieme di piattaforme, layout e librerie di base tutto strutturato a livelli che aiutano gli sviluppatori nella realizzazione delle applicazioni.

Tra i vantaggi di *Flutter* c'è sicuramente il fatto che è una tecnologia sempre più utilizzata come mostrato nel grafico 2.1. Costruire una buona community dietro ad una tecnologia permette ai nuovi sviluppatori di accedere a materiale e tutorial già pronti. Di contro abbiamo che le applicazioni hanno una dimensione maggiore rispetto a se sviluppate con approccio native (un'app *Flutter* da 4MB peserebbe 400Kb se realizzata con approccio native) e, dato che *Flutter* è un prodotto *Google*, si ha un supporto alle applicazioni *iOS* inferiore. Resta comunque il fatto che *Flutter* si presta molto bene come tecnologia per realizzare *Minimum Viable Product*.

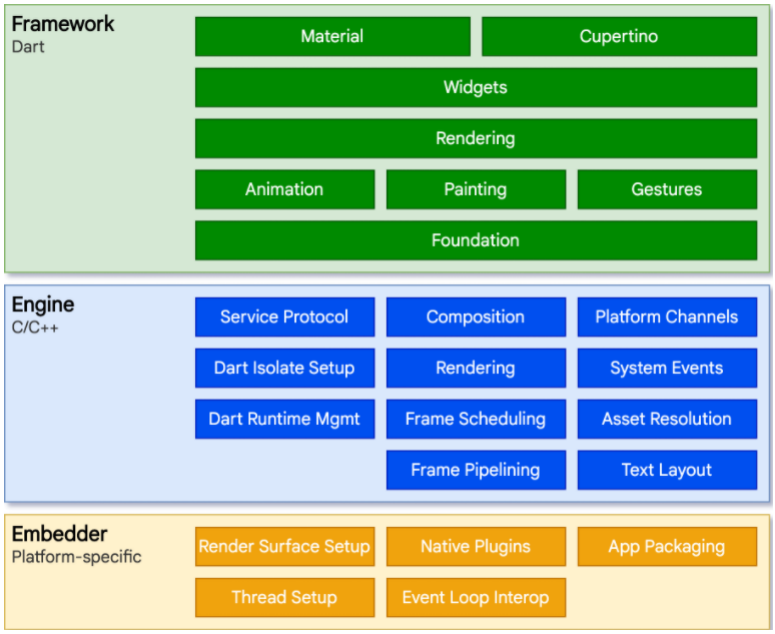


Figura 2.2: Livelli architetturali di *Flutter*.

2.3.2 Xamarin

Come riportato dal sito, Xamarin è una piattaforma open source per la compilazione di applicazioni per iOS, Android e Windows con .NET. Xamarin è un livello di astrazione che gestisce la comunicazione del codice condiviso con il codice della piattaforma sottostante. Consente agli sviluppatori di condividere una media del 90% delle applicazioni su più piattaforme. Questo modello consente agli sviluppatori di scrivere utilizzando C#

come linguaggio tutta la logica di business (o di riutilizzare il codice dell'applicazione esistente), replicando in ogni piattaforma le prestazioni e l'aspetto originari.
Le applicazioni Xamarin possono essere scritte su PC o Mac e vengono compilate in pacchetti dell'applicazione nativa, ad esempio un file APK in Android o un file IPA in iOS, figura 2.3.
Ovviamente ci sono anche dei contro come ad esempio la dimensione dell'applicazione finale che tendenzialmente risulta essere maggiore rispetto alla stessa sviluppata con approccio native. Oppure problemi di compatibilità con con librerie e strumenti di terze parti che non sono presenti nello store *Xamarin*.

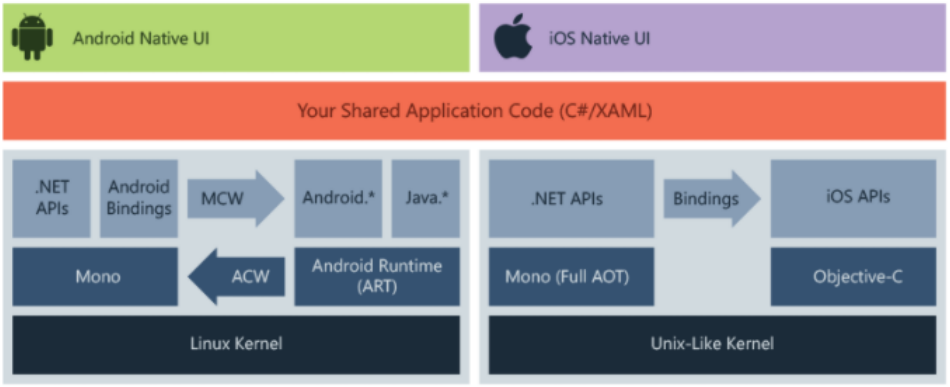


Figura 2.3: Funzionamento *Xamarin*.

2.4 Obiettivo della tesi

L'obiettivo della tesi è quello di valutare l'utilizzo di *KMP* per lo sviluppo di un'applicazione che deve essere utilizzabile sia su dispositivi *Android* che su dispositivi *iOS*.

2.5 Caso di studio e specifiche

Il caso di studio è rappresentato dall'applicazione di gestione del magazzino che attualmente hanno solo in versione *Android*. Nella figura 2.4 sono riportate le diverse viste dell'applicazione (le immagini fanno riferimento all'applicazione *Android* prodotta durante il tirocinio che però eguagliano per funzionalità quelle dell'applicazione preesistente in azienda).
Nel dettaglio, l'applicazione ha una parte iniziale di *Login* e scelta del *Punto vendita* del

quale si vuole gestire il magazzino.

Effettuata questa scelta tramite la vista del *Menu* si hanno i seguenti quattro casi d'uso:

- *Purchase* (Acquisto) : Effettuare un ordine di acquisto.
- *Unlod* (Scarico) : Gestire gli scarichi.
- *Load* (Carico) : Gestire i carichi.
- *Stock* (Inventario) : Effettuare l'inventario del magazzino.

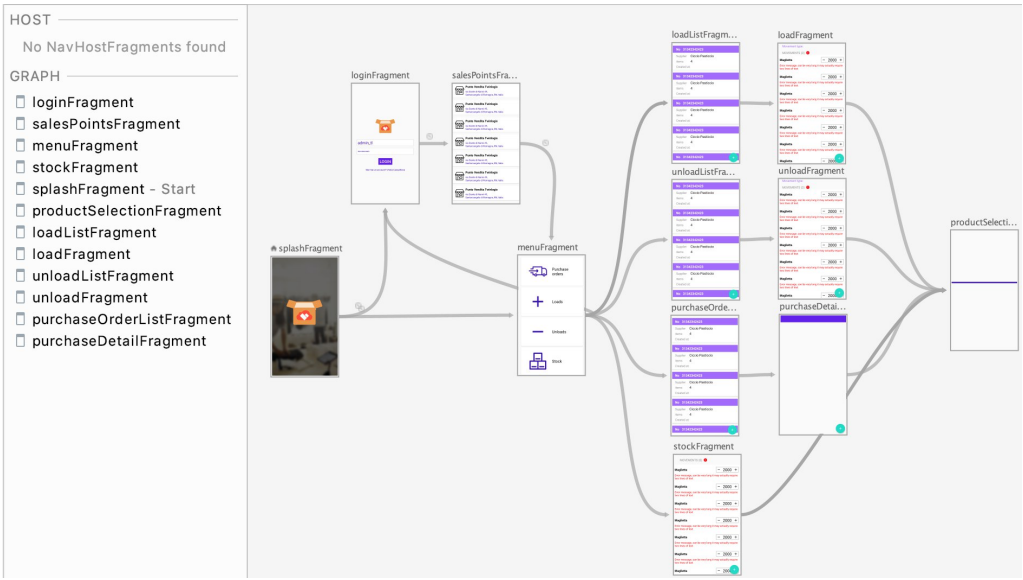


Figura 2.4: Viste dell'applicazione e navigazione (immagine presa dal Navigation component di Android).

2.5.1 Login e scelta del punto vendita

Come mostrato in figura 2.5, inizialmente l'utente deve effettuare il login e scegliere il punto vendita del quale vuole gestire il magazzino. A tal punto l'utente si trova nella schermata di menu nella quale ha accesso ai quattro casi d'uso dell'applicazione.

L'utente dopo il login acquisisce l'accesso anche al menu drawer, figura 2.6. Nel drawer vengono mostrate informazioni sull'utente registrato e sul punto di vendita selezionato.

Per tornare alla schermata di login l'utente deve effettuare il logout, funzione presente nel drawer dell'applicazione figura 2.6, oppure attendere lo scadere del token di autenticazione. Difatti fintanto che il token è valido non deve essere richiesto all'utente di ripetere il login.

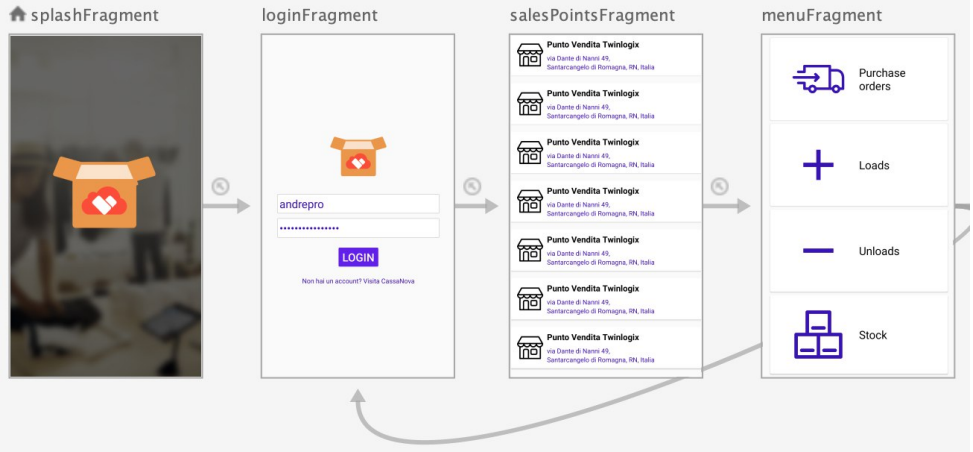


Figura 2.5: Viste di Login e scelta del punto vendita.

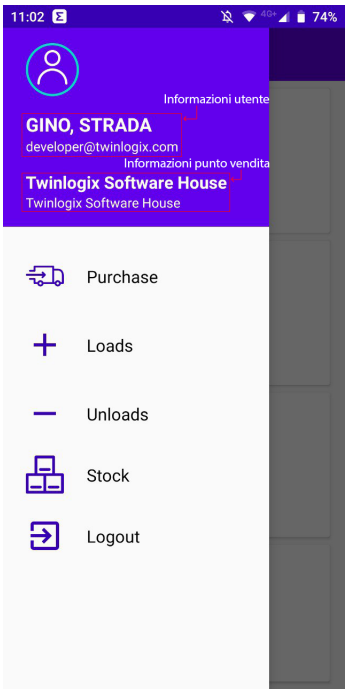


Figura 2.6: Drawer applicazione Android.

2.5.2 Acquisto

Per aggiungere un nuovo acquisto l'utente deve navigare dal menu alla schermata di *Purchase details*, come mostrato in figura 2.7.
La schermata tra il menu e la *Purchase details* mostra, se presenti, la lista di ordini di acquisto che l'utente ha salvato come bozze. Questa lista è mantenuta solo localmente all'interno del database dell'applicazione.
Quando l'utente si trova nella schermata che mostra la lista delle bozze di ordini, può decidere di creare un nuovo ordine d'acquisto o può scegliere di modificare, se presente, una bozza che aveva precedentemente salvato. In ogni caso viene rimandato alla *Purchase details* dove può creare/modificare l'ordine.
La *Purchase details* si compone di tre diverse tab (come mostrato in figura 2.8):

- Nella prima tab si effettua la scelta della numerazione e del fornitore.
- Nella seconda si ha la visualizzazione dei prodotti dell'ordine e la possibilità di modificare la quantità dei singoli.
- Nella terza si possono aggiungere note sull'acquisto.

Sempre in questa schermata si possono eseguire, premendo gli appositi bottoni, le seguenti azioni:

- Controllo dei prodotti selezionati: viene rieseguito il controllo sui prodotti selezionati alla ricerca di possibili errori.
- Cancellazione di tutti i prodotti selezionati.
- Salvataggio dell'ordine nel database.
- Invio dell'ordine al server.

Premendo invece il bottone presente alla fine della vista (figura 2.9) si può:

- Aggiungere prodotti sotto scorta.
- Cercare e aggiungere nuovi prodotti, questo porta l'apertura della vista di selezione dei prodotti (figura 2.10).

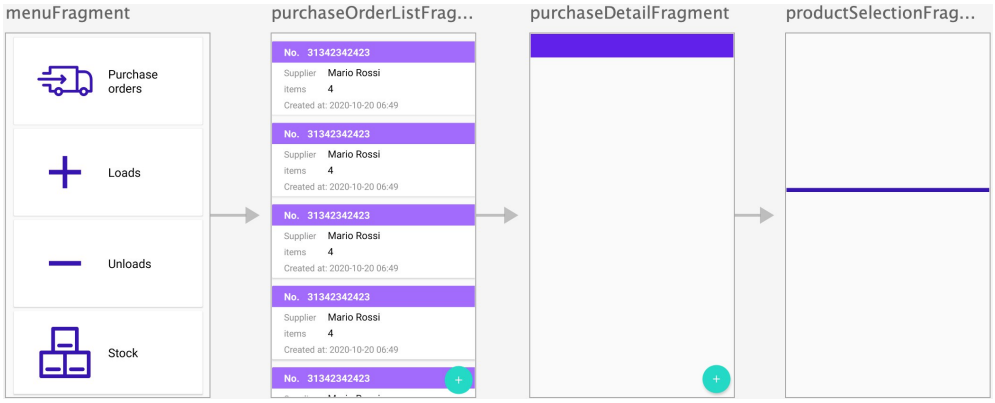


Figura 2.7: Viste relative al caso d'uso Acquisto.

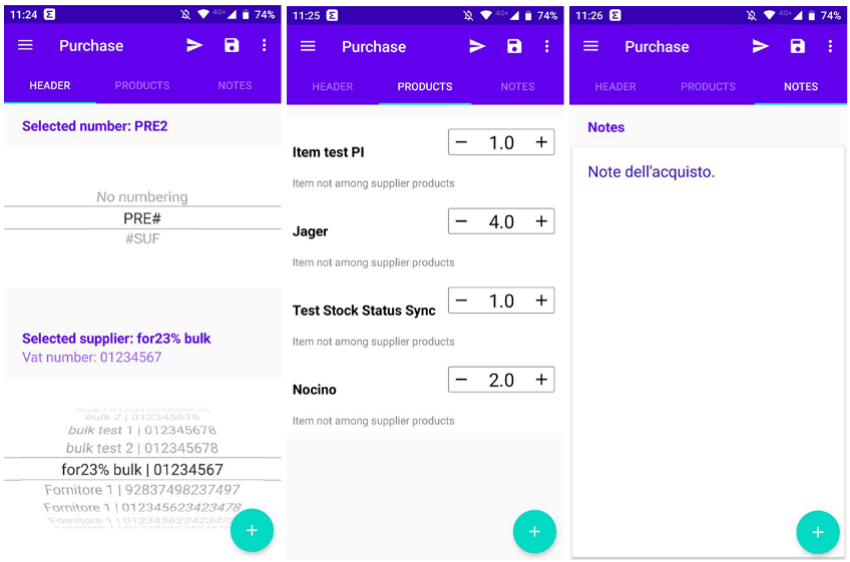


Figura 2.8: Tab della vista *Purchase details*.

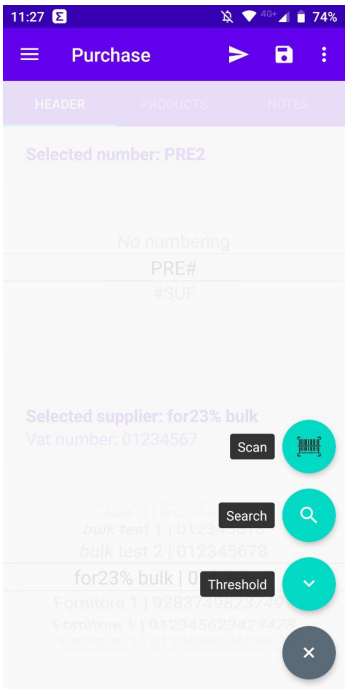


Figura 2.9: Bottoni per la navigazione da *Purchase details* alla vista di selezione dei prodotti e per l'aggiunta di prodotti sotto scorta.

2.5.3 Carico e Scarico

Questi due casi d'uso sono molto simili, registrare un carico va ad aggiungere prodotti nel magazzino mentre uno scarico va a rimuoverli.

Le viste per questi due casi d'uso sono riportate nella figura 2.11.

Come per gli ordini di acquisto anche per i carichi e scarichi l'utente può creare delle bozze, queste sono salvate nel database dell'applicazione e vengono mostrate inizialmente sotto forma di lista. L'utente può scegliere di modificare una bozza o può creare un nuovo carico/scarico.

Prendendo ad esempio che l'utente voglia creare un nuovo carico, nella vista di *Load* può, tramite il bottone in fondo, accedere alla selezione dei prodotti come per gli ordini d'acquisto, figura 2.10.

Le altre azioni che l'utente può svolgere sono analoghe a quelle della vista di creazione/modifica di ordini di acquisto 2.5.2, l'unica differenza è che non c'è il bottone per l'aggiunta di prodotti sotto scorta come mostrato in figura 2.12.

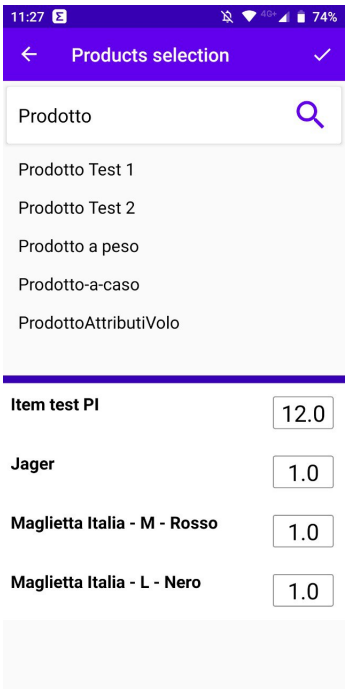


Figura 2.10: Vista di selezione dei prodotti, viene usata da tutti i casi d'uso.

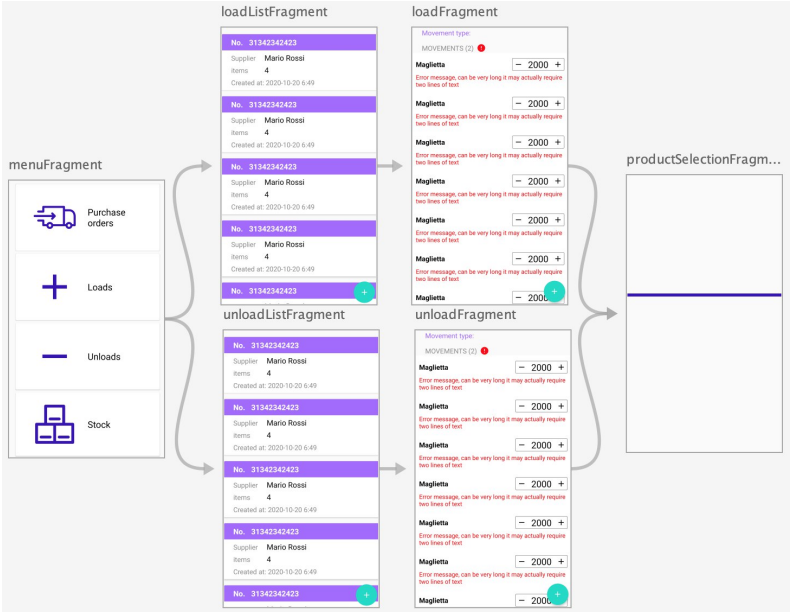


Figura 2.11: Navigazione nell'applicazione per i casi d'uso di carico e scarico.

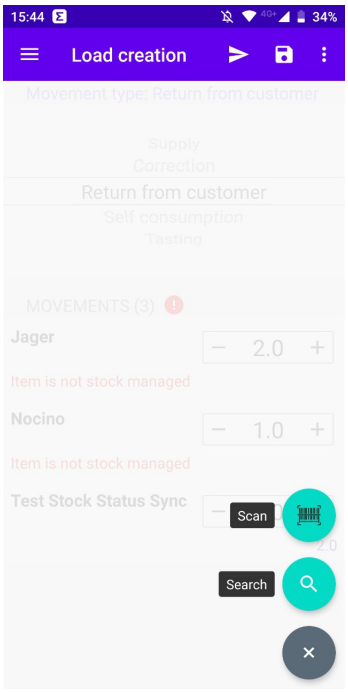


Figura 2.12: Bottoni presenti nella vista di creazione/modifica di un carico (Per gli scarichi esiste una schermata analoga).

2.5.4 Inventario

Il caso d'uso comprende unicamente due viste, figura 2.13.

L'utente deve poter eseguire l'inventario del magazzino andando ad impostare gli attuali valori di quantità dei prodotti presenti, figura 2.14.

Anche in questo caso può mantenere una bozza in memoria locale prima di decidere di inviare l'inventario al server.

Per la scelta dei prodotti viene utilizzata nuovamente la vista di selezione dei prodotti, figura 2.10. Le azioni che si possono compiere sono le stesse dei casi d'uso di carico e scarico 2.5.3.

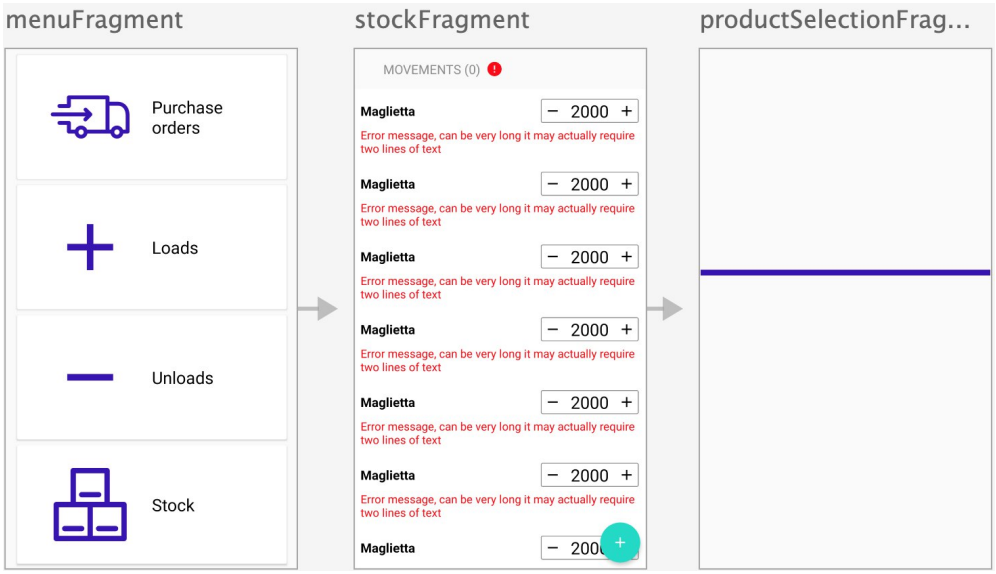


Figura 2.13: Viste relative al caso d'uso dell'inventario.

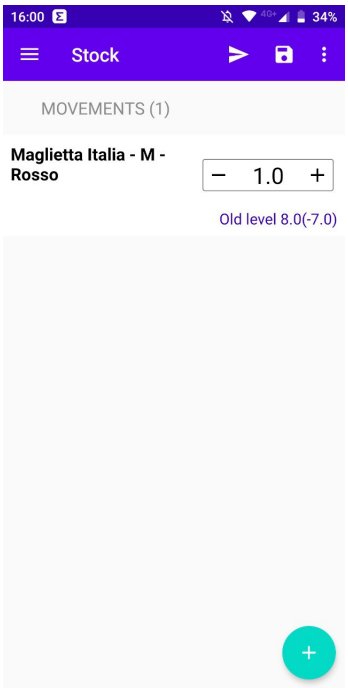


Figura 2.14: Vista dell'inventario.

3. Progettazione

3.1 Architettura

La scelta dell'architettura è guidata dalle necessità della tesi in se, ovvero ridurre quanto più possibile la duplicazione di codice tra le due applicazioni, lasciando però alle singole piattaforme il compito di definire l'interfaccia grafica. Per questo motivo si è adottata la *Clean Architecture* e la *Model-View-ViewModel*.

3.1.1 Clean Architecture

Adottare la *Clean Architecture* permette di mantenere disaccoppiata la parte di business logic dalle singole tecnologie utilizzate, ad esempio dal framework utilizzato per effettuare le chiamate remote.
Dato che *KMP* è ancora in alpha e nuove librerie verranno create, è possibile che nel breve futuro si vogliano adottare diverse librerie da quelle attualmente utilizzate. Strutturando il progetto secondo la *Clean Architecture* effettuare questo cambiamento non comporta costi eccessivi.
Come mostrato in figura 3.1, la *Clean Architecture* si presenta come una sequenza di cerchi concentrici la cui idea alla base è che i cerchi all'interno non devono avere conoscenza a riguardo dei cerhi all'esterno.
Ognuno dei quattro cerchi ha uno specifico significato:

- Entity: sono i business object dell'applicazione, incapsulano al loro interno le regole più generali e ad alto livello. É la porzione di software meno soggetta a cambiamenti.
- Use case: rappresenta tutti i casi d'uso dell'applicazione, gestiscono il flusso di dati in ingresso e in uscita dalle *Entity*.

- Interface Adapter: contiene gli adapter che convertono, ad esempio, i dati dal formato più conveniente per i casi d'uso e le entità, al formato più conveniente per il Database.
- Framework e Driver: è il livello in cui si inseriscono i dettagli come ad esempio il database o il framework web.

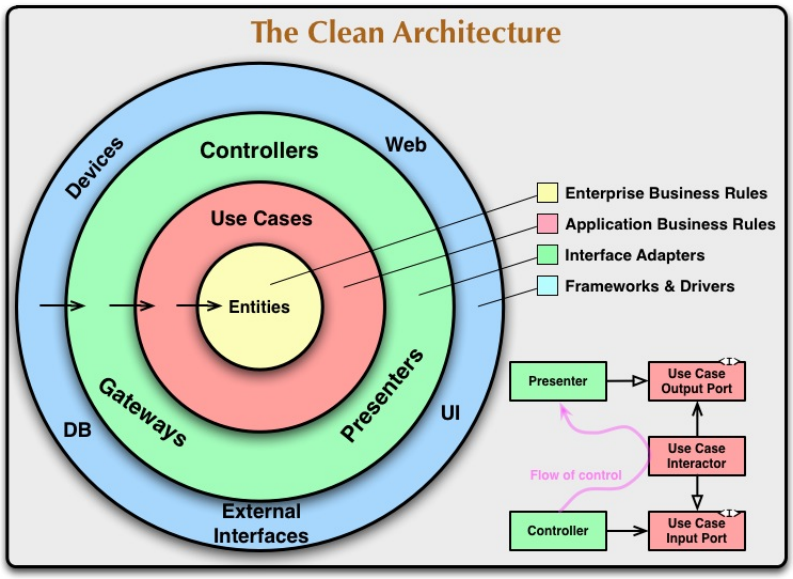


Figura 3.1: Clean Architecture.

3.1.2 Model-View-ViewModel

La scelta di utilizzare l'architettura *Model-View-ViewModel* (figura 3.2) invece del *Model-View-Presenter* è stata dettata dal fatto che si vuole disaccoppiare il più possibile la *View* dalla business logic che dovrà risiedere nella parte condivisa. Di fatto nella *MVP* il *Presenter* deve avere connoscenza della *View* cosa che riduce il disaccoppiamento tra le due parti.
La *View* viene notificata dei cambiamenti di stato tramite i *Kotlin Flow* come richiesto dall'azienda.

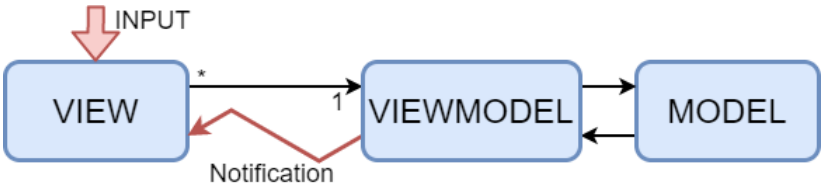


Figura 3.2: Model-View-ViewModel.

3.1.3 Architettura nel progetto

Come mostrato in figura 3.3 per la quale ho usato gli stessi colori dell'immagine 3.1, il progetto si compone dei seguenti livelli:

- Source: rappresenta il livello più esterno dedito all'interazione con il server (*RemoteSource*) e con il database (*LocalSource*).
- Repository: ha l'istanza della *Source* e si occupa di convertire i *Data Transfer Object*/Oggetti ottenuti dal database in *Entity* e viceversa.
- ServiceLocator: ha le uniche istanze dei *Repository* e degli *Use Case* dell'applicazione. Svolge una funzione di *Dependency Injection* andando ad iniettare nel *ViewModel* i casi d'uso dei quali necessita.

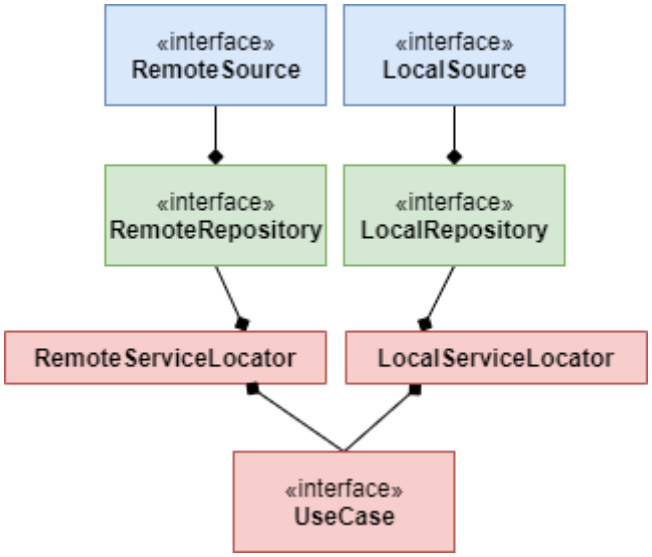


Figura 3.3: Architettura del progetto: *Source*, *Repository*, *UseCase* e *ServiceLocator*.

Prendendo come esempio il Login in figura 3.4, per completare l'obiettivo di mantenere il più semplice possibile la realizzazione delle viste delle diverse piattaforme, si ha la seguente logica:

- ViewModel: si occupa di gestire lo stato del *Model* e dell'aggiornamento della *View*.
- BasicView: per ogni vista dell'applicazione esiste una BasicView che incapsula al suo interno tutte le reazioni in risposta a tutte le azioni che l'utente può compiere nella vista.
- Viste: le viste (*LoginFragment* e *LoginViewController*) estendono una *BasicView*, hanno come unico compito quello di rilevare le azioni eseguite dall'utente nell'interfaccia grafica e, per ognuna di esse, attivare la specifica reazione.

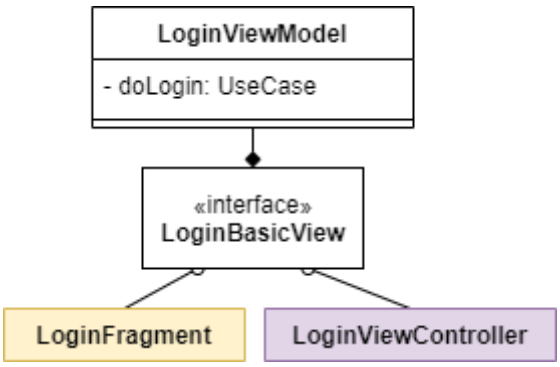


Figura 3.4: Architettura del progetto, esempio di Login: ViewModel, BasicView con LoginFragment (Android) e LoginViewController (iOS).

Come anticipato in precedenza, i *ViewModel* si avvalgono dei *Kotlin Flow* per notificare cambiamenti alle *View*.
Riprendendo l'esempio in figura 3.4, il *LoginViewModel* mantiene al suo interno un flow di Token (figura 3.5) e in seguito alla richiesta di login, quando ottiene la risposta dal server (caso d'uso *doLogin*) aggiorna il flow. L'aggiornamento del valore del flow comporta che una notifica alla vista di login che si è registrata ad esso in precedenza tramite l'*observeTokenFlow*.

```
private val tokenFlow: StateFlowWrapper<Token> = StateFlowWrapper(Token.emptyToken())

fun doLoginAndGetAuthToken(credentials: Credentials) {
    clientScope.launch { this: CoroutineScope
        val result: Token? = doLogin(credentials)
        tokenFlow.updateValue(result!!)
    }
}
@InternalCoroutinesApi
fun observeTokenFlow(flowCollector: FlowCollector<Token>) {
    clientScope.launch { this: CoroutineScope
        tokenFlow.collect(flowCollector)
    }
}
```

Figura 3.5: *Kotlin flow* nel *ViewModel*.

3.2 Configurazione del progetto

Un progetto *KMP* si presenta come mostrato in figura 3.6.
Nella radice dell'albero è necessario di configurare *Gradle*, in particolare andando ad inserire nel *build.gradle* i *repository* e i *classpath* che vengono utilizzati per risolvere le dipendenze nei sotto progetti.
Nel secondo livello dell'albero, colore verde, sono definiti i tre sotto progetti:

- androidApp: è il progetto *Android* dell'applicazione.
- iosApp: è il progetto *iOS* dell'applicazione.
- shared: è la porzione di codice condivisa dai precedenti due progetti.

A sua volta anche *shared* contiene tre progetti:

- commonMain: definisce il codice condiviso. Per dettagli che richiedono logica dipendente dalla specifica piattaforma, viene utilizzato il meccanismo degli *expect/actual*. In questo sotto progetto vengono definiti gli *expect*, esempio in figura 3.7.
- androidMain: per ogni *expect* presente nel *commonMain* qui viene riportato l'*actual* per la piattaforma *Android*, esempio in figura 3.8.
- iosMain: per ogni *expect* presente nel *commonMain* qui viene riportato l'*actual* per la piattaforma *iOS*, esempio in figura 3.9.

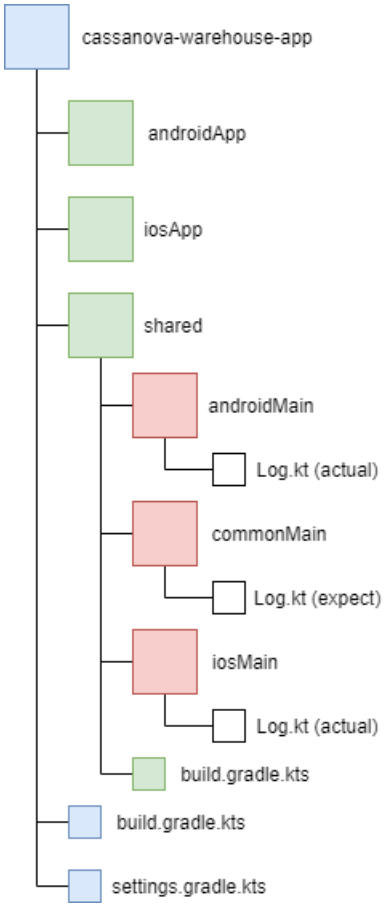


Figura 3.6: Struttura base del progetto.

Log.kt

```
1  package com.twinlogix.warehouse.shared.configuration.logger
2
3  expect object Log {
4      fun e(tag: Tag, message: String, cause: Throwable)
5      fun d(tag: Tag, message: String)
6  }
```

Figura 3.7: Esempio di *expect* definito nella parte *commonMain*.

Log.kt

```
1  package com.twinlogix.warehouse.shared.configuration.logger
2
3  import android.util.Log
4
5  actual object Log {
6      actual fun e(tag: Tag, message: String, cause: Throwable) {
7          Log.e(tag.toString(), message, cause)
8      }
9
10     actual fun d(tag: Tag, message: String) {
11         Log.d(tag.toString(), message)
12     }
13 }
```

Figura 3.8: Implementazione *Android* dell'*expect* in figura 3.7.

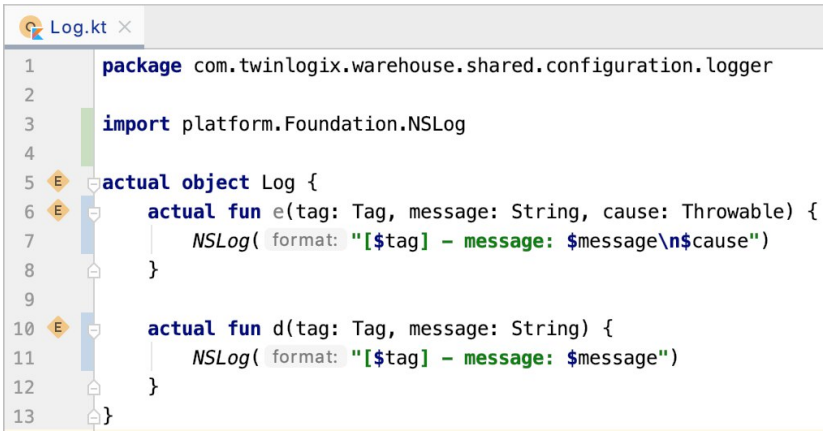


Figura 3.9: Implementazione *iOS* dell'*expect* in figura 3.7.

Il file *build.gradle* presente nel sotto progetto shared deve definire al suo interno:

- Plugin *KMP*: permette la creazione e la gestione di un progetto in *KMP*.
- Target: il target è la parte della *build* responsabile di: compilazione, test e creazione del package di codice condiviso che verrà utilizzato per lo specifico target. Nel caso del progetto in questione sono da configurare due target, *Android* e *iOS*. La configurazione del target *iOS* richiede di definire un target diverso a seconda di se si vuole compilare l'applicazione per un dispositivo fisico o per un simulatore.
- Source set: questo blocco descrive i *Source set* del progetto. Alcune dipendenze richiedono della logica specifica per le singole piattaforme quindi si definiscono tre *Source set*:
  - *Common source set*: contiene le dipendenze per la parte *commonMain* del progetto *shared*.
  - *Android source set*: contiene dipendenze specifiche per la parte *androidMain*.
  - *iOS source set*: contiene dipendenze specifiche per la parte *iosMain*.
- packForXcode: task *Gradle* con il compito di generare un framework contenente la porzione di codice condivisa che possa essere importato in *Xcode*.

3.3 Viste dell'applicazione e schema del database

La progettazione delle viste dell'applicazione (in termine di numero e funzionalità di ciascuna vista) è già stata svolta dall'azienda ed è spiegata nel dettaglio nella sezione 2.5.
Lo schema del database in progettazione ha subito una minima modifica rispetto all'originale, le tabelle che mantengono le bozze di carichi e scarichi dell'utente sono state fuse. Per identificare il tipo di movimento sono stati aggiunti gli attributi *loadMovementType* e *unloadMovementType* al *movement_header*, figura 3.10.

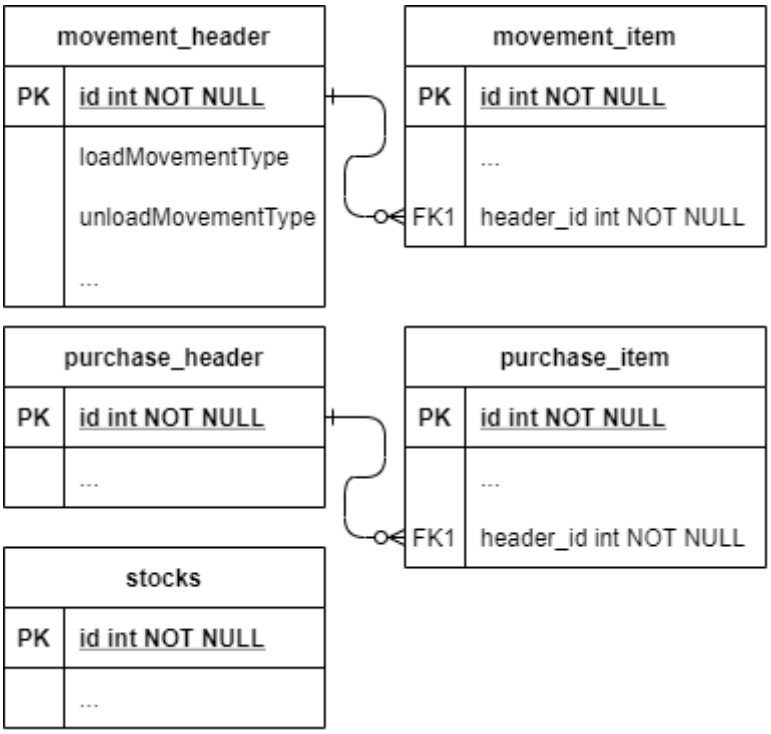


Figura 3.10: Schema ER del database dell'applicazione.

4. Implementazione

4.1 Tecnologie utilizzate

Per lo sviluppo del progetto sono state scelte/sono richieste le seguenti tecnologie:

- *Kotlin Multiplatform Mobile* un SDK che consente di estrarre la business logic dall'applicazione che si vuole creare permettendo poi di riusarla nelle diverse piattaforme.
- *Android Studio* e *Kotlin* rispettivamente come IDE e come linguaggio di sviluppo per la parte *shared* e per l'applicazione *Android*.
- *Xcode* e *Swift* rispettivamente come IDE e come linguaggio di sviluppo per la parte *iOS*.
- *Ktor Client* come client http per poter effettuare chiamate remote.
- *SQLDelight* come libreria per l'utilizzo del database delle due piattaforme e *SQL* come linguaggio di database.
- *Kotlin Flow* come libreria per gestire i flussi dati. É una nuova libreria di stream asincroni creata sulle *Coroutines* da JetBrains. É simile agli stream *Rx*. I *Flow* sono cold e per collezionarne i valori si utilizza la funzione *collect* come mostrato nella figura 3.5.

Il progetto è stato svolto utilizzando *Git* come sistema di *version control* e *Gradle* per la gestione delle dipendenze. Il progetto risiede in un repository *GitHub*.
Lo sviluppo, data la necessità di produrre un'applicazione *iOS*, richiede l'utilizzo del sistema operativo *MacOS*.

4.1.1 Kotlin Multiplatform

Kotlin Multiplatform nasce con l'idea di permettere di estrapolare la parte di business logic comune alle diverse applicazioni inserendola in una parte *shared*, questa viene poi utilizzata dai diversi *target* (i quali si dovranno preoccupare unicamente di realizzare i dettagli specifici della singola piattaforma).
Il vantaggio risulta da subito essere la minor quantità di codice duplicata che porta a facilitarne la manutenzione e i test.
Se è necessario accedere, dal codice condiviso, ad API specifiche delle singole piattaforme si utilizza il meccanismo KMP delle dichiarazioni *expect*/*actual*.
Con questo meccanismo, nella parte comune si dichiara ciò che ci si aspetta (*expect*) e le piattaforme devono fornire la dichiarazione effettiva (*actual*), come mostrato nell'esempio delle figure 3.7, 3.8 e 3.9.
KMP è ancora una tecnologia nuova entrata in fase alpha a fine agosto e questo è sicuramente il problema principale perché la rende soggetta a frequenti cambiamenti.
Nel dettaglio del progetto viene utilizzato meramente *Kotlin Multiplatform Mobile* in quanto l'interesse è nello sviluppo di un'applicazione *Android* e *iOS*.

4.1.2 Ktor e SQLDelight

Sono le due soluzioni compatibili con *Kotlin Multiplatform* che permettono di definire nella parte *shared* le chiamate remote (con *Ktor*) e l'utilizzo del database (con *SQLDelight*).
Più nel dettaglio:

- *Ktor* è un framework di rete basato sulle *coroutine* di Jetbrains scritto in *Kotlin*.
- *SQLDelight* è una libreria che genera, a partire da del codice *SQL*, delle API *type-safe* in Kotlin che permettono di gestire il database.

Attualmente sono le soluzioni migliori, se non le uniche, per effettuare queste due operazioni.

4.2 Implementazione del progetto

Per semplificare l'implementazione, in particolare il debug della parte *shared*, è consigliato realizzare la parte condivisa in parallelo con l'applicazione *Android* lasciando per ultimo

lo sviluppo *iOS*.

4.2.1 Creazione del progetto e configurazione iniziale

La prima operazione da svolgere, come riportato nella fase di progettazione 3.2, è quella di creare un progetto che abbia una struttura identica a quella riportata nella figura 3.6. Dopodiché è necessario configurare i file *Gradle* per renderlo un progetto *KMP*.
Al termine di questa operazione la parte condivisa può essere compilata per le diverse piattaforme.
Per importare *shared* nel sotto progetto *Android*, è sufficiente aggiungere come dipendenza *Gradle* il .jar ottenuto dalla compilazione della parte condivisa.
Per *iOS* è necessario, dopo aver creato il framework native con il task packForXcode, indicare, agendo dall'IDE *Xcode*, la sua posizione in modo tale che possa essere importato. L'intera procedura è descritta dettagliatamente in diversi tutorial online ragion per cui qui non saranno riportati i dettagli.
In seguito creare anche la struttura dei *package* dell'architettura scelta nella fase di progettazione, sezione 3.1.3.

4.3 Livelli esterni della Clean Architecture

Realizzazione dei due livelli più esterni, blu e verde, della *Clean Architecture* in figura 3.3.

4.3.1 Configurazione ed utilizzo di Ktor

In quanto tecnologia nuova *Ktor* non possiede un'adeguata documentazione, inoltre il fatto che possa essere utilizzata sia lato *Client* che lato *Server* rende ancora più complessa la ricerca delle informazioni.

Client HTTP

Per poter svolgere le chiamate remote è necessario configurare il *Client HTTP Ktor*.
Come mostrato in figura 4.1 è necessario aggiungere:

- Logging: per questioni di *debug* è utile avere un sistema di *logging* che stampi le chiamate effettuate e le risposte ricevute. Il sistema di *logging* è realizzato utilizzan-

do il meccanismo degli *expect/actual*, come mostrato nelle figure *expect*-3.7, *actual Android*-3.8 e *actual iOS*-3.9.

- JsonFeature: necessario per configurare il serializzatore di *JSON*. Viene utilizzata la libreria *kotlinx.serialization*.

- HttpResponseValidator: opzionale, è possibile configurare il *client HTTP* in modo tale che possa gestire al suo interno la validazione delle risposte ottenute dal *server*.

```
class KtorClientHttp(clientEngine: HttpClientEngine) {
    @UnstableDefault
    @ExperimentalStdlibApi
    val client: HttpClient by lazy {
        HttpClient(clientEngine) { this: HttpClientConfig<*>

            install(Logging) {...}

            install(JsonFeature) {...}

            HttpResponseValidator {...}
        }
    }

    init {
        ensureNeverFrozen()
    }
}
```

Figura 4.1: Client HTTP Ktor.

Realizzazione delle API

La realizzazione di ogni singola *API* segue il seguente percorso:

1. Analisi dell'*API* in termini di impostazione della *richiesta* e comprensione della *risposta*. Il risultato di questa fase è la realizzazione dei *Data Transfer Object* (DTO) e delle *Entity* compresa l'aggiunta della richiesta nel *RemoteSource*, figura 4.2.
   Nella figura 4.4 è mostrato l'esempio di DTO relativo al caso d'uso in analisi.
   Ogni DTO deve avere una controparte *Entity*. I DTO devono estendere l'interfaccia *RemoteDataAdapter* e quindi implementare il metodo *toEntity()* che li converte nella *Entity* corrispondente.
   La classe *SyncronizedEntity* invece è necessaria unicamente per come sono gestiti questi DTO lato server, va a definire i campi utilizzati internamente per la sincronizzazione dei dati.

2. Implementazione della richiesta all'interno del *RemoteSourceImpl*, figure 4.5.

3. Aggiunta della richiesta nel *RemoteRepository*, figura 4.6 e 4.7. Il *Repository* si occupa di gestire la conversione da *Entity* a DTO e viceversa.

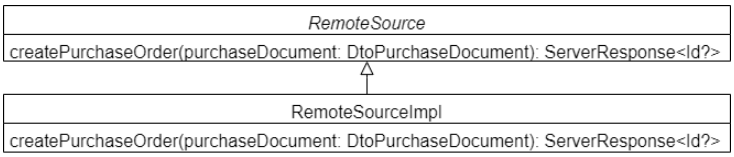


Figura 4.2: Aggiunta dell'*API* di creazione di un ordine di acquisto

Per semplificare l'esecuzione delle richieste sono da aggiungere due metodi di utilità che permettono di effettuare chiamate remote con o senza autenticazione, figura 4.3.
Come si evince dalle immagini sono necessarie le *Kotlin coroutine*, e quindi, andando a sfruttare un'ulteriore volta il meccanismo degli *expect/actual*, si vanno a definire i *Dispatcher* utilizzati nell'esecuzione delle *coroutine*, figure *expect*-4.8, *actual Android*-4.9 e *actual iOS*-4.10.

```kotlin
class RequestHandler<D> {
    // Exceptions are handled into the hppt client, see the configuration in KtorClientHttp.kt
    suspend fun doRequest(request: suspend () -> D?): D? =
        try { request() } catch (exception: Exception) { null }

    suspend fun doAuthRequest(request: suspend (cookieHeader: CookieHeader) -> D?): D? =
        try { RemoteUtils.getCookieHeaderFromPreferences()?.let { request(it) } }
        catch (exception: Exception) { null }
}
```

Figura 4.3: Funzioni di utilità per l'esecuzione delle richieste a remoto.

```kotlin
@Serializable
class DtoPurchaseDocument(...) : SynchronizedEntity(), Dto, RemoteDataAdapter<PurchaseDocument> {

    constructor(purchaseDocument: PurchaseDocument): this (
        live = purchaseDocument.live,
        idSalesPoint = purchaseDocument.idSalesPoint,
        userType = purchaseDocument.userType,
        idUserFO = purchaseDocument.idUserFO,
        idSupplier = purchaseDocument.idSupplier,
        taxFree = purchaseDocument.taxFree,
        amount = purchaseDocument.amount,
        date = purchaseDocument.date,
        datetime = purchaseDocument.datetime,
        confirmed = purchaseDocument.confirmed,
        documentType = purchaseDocument.documentType,
        supplier = purchaseDocument.supplier,
        purchaseDocumentItems = purchaseDocument.purchaseDocumentItems?.map { DtoPurchaseDocumentItem(it) },
        order = purchaseDocument.order
    )

    override fun toEntity(): PurchaseDocument = {...}
}
```

Figura 4.4: DTO del documento di acquisto.

```kotlin
@ImplicitReflectionSerializer
override suspend fun createPurchaseOrder(purchaseDocument: DtoPurchaseDocument): ServerResponse<Id?>? {
    return RequestHandler<ServerResponse<Id?>>().doAuthRequest { it: CookieHeader
        var id: Id? = null
        var error: RemoteError? = null
        try {
            id = client.post<Id>(Api.Post.PURCHASE_ORDER) { this: HttpRequestBuilder
                header(it.key(), it.value)
                contentType(ContentType.Application.Json)
                body = purchaseDocument
            }
        } catch (e: ServerResponseException) {
            error = RemoteError.SERVER_ERROR
        } catch (e: ClientRequestException) {
            error = RemoteError.CONNECTION_ERROR
        }
        ServerResponse(error = error, response = id) ^doAuthRequest
    }
}
```

Figura 4.5: Implementazione dell'*API* di creazione di un ordine di acquisto.

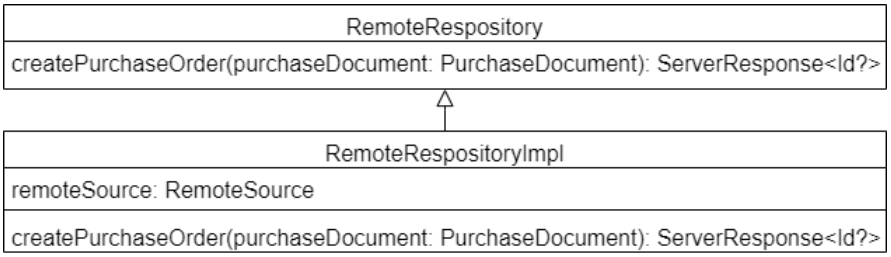


Figura 4.6: Aggiunta al *RemoteRespository* della richiesta di creazione di un ordine di acquisto.

```kotlin
override suspend fun createPurchaseOrder(purchaseDocument: PurchaseDocument): ServerResponse<Id?>? {
    return withContext(Dispatcher.io) { this: CoroutineScope
        remoteSource.createPurchaseOrder(DtoPurchaseDocument(purchaseDocument))
    }
}
```

Figura 4.7: Implementazione del metodo di richiesta di creazione di un ordine di acquisto nel *RemoteRepositoryImpl*.

```kotlin
5 A expect object Dispatcher {
6 A     val main: CoroutineDispatcher
7 A     val io: CoroutineDispatcher
```

Figura 4.8: *Expect* dei *dispatcher* utilizzati per le *coroutine*.

```kotlin
actual object Dispatcher {
    actual val main: CoroutineDispatcher by lazy { Dispatchers.Main }
    actual val io: CoroutineDispatcher by lazy { Dispatchers.IO }
}
```

Figura 4.9: *Actual Android* relativo all'immagine 4.8.

```kotlin
import kotlinx.coroutines.CoroutineDispatcher
import kotlinx.coroutines.Runnable
import platform.darwin.dispatch_async
import platform.darwin.dispatch_get_main_queue
import kotlin.coroutines.CoroutineContext

actual object Dispatcher {
    actual val main: CoroutineDispatcher by lazy { IosMainDispatcher }
    actual val io: CoroutineDispatcher by lazy { IosMainDispatcher }
}

private object IosMainDispatcher : CoroutineDispatcher() {
    override fun dispatch(context: CoroutineContext, block: Runnable) {
        dispatch_async(dispatch_get_main_queue()) { block.run() }
    }
}
```

Figura 4.10: *Actual iOS* relativo all'immagine 4.8.

Configurazione delle shared preferences

La gestione dei dati che non devono essere persi alla chiusura dell'applicazione, come il *token* di autenticazione o alcuni dati dell'utente attualmente registrato, richiede l'utilizzo di un meccanismo che in *Android* è chiamato *Shared Preferences*.
Attualmente per KMP esistono diverse alternative per realizzare questa funzione, da requisiti si vuole che in *Android* le *Shared Prefecences* siano criptate mentre su *iOS* non si vuole usare il *Key Chain*, in quanto i dati salvati in esso restano anche dopo la disinstallazione dell'applicazione.
L'utilizzo della libreria *Multiplatform Settings* fornisce, tramite l'uso del meccanismo *expect/actual*, la possibilità di soddisfare i requisiti, figure *expect*-4.11, *actual Android*-4.12 e *actual-iOS*-4.13.
Come mostrato in figura 4.12 la crittografia delle *Shared Preferences* in *Android* è stata svolta utilizzando la libreria *androidx.security:security-crypto*.

```kotlin
import com.russhwolf.settings.Settings

expect object PreferencesSettings {
    fun settings(): Settings
}
```

Figura 4.11: *Shared Preferences expect*.

```kotlin
actual object PreferencesSettings {
    private const val preferencesFileName = "secret_shared_prefs"

    var context: Context? = null
    private var masterKeyAlias: String? = null
    private var delegate: SharedPreferences? = null
    private var settings: Settings? = null

    actual fun settings(): Settings {
        if (settings != null) return settings!! // If settings exists return
        createMasterKey() // else create the settings
        createDelegate(context)
        createSettings()
        settings?.let { return it }
        throw NullPointerException(ERROR_SHARED_PREFERENCES_CREATION)
    }

    fun setupContext(context: Context) { PreferencesSettings.context = context }

    private fun createSettings() : Unit?  = delegate?.run {...}

    private fun createDelegate(context: Context?) {
        if (masterKeyAlias != null || context != null) {
            delegate = EncryptedSharedPreferences.create(
                preferencesFileName, // fileName
                masterKeyAlias!!, // masterKeyAlias used for the encryption
                context!!, // context in order to access the stored preferences
                EncryptedSharedPreferences.PrefKeyEncryptionScheme.AES256_SIV,
                EncryptedSharedPreferences.PrefValueEncryptionScheme.AES256_GCM
            )
        }
    }

    private fun createMasterKey() {...}
}
```

Figura 4.12: *Shared Preferences actual Android* relativo alla figura 4.11.

```kotlin
import com.russhwolf.settings.AppleSettings
import com.russhwolf.settings.Settings
import platform.Foundation.NSUserDefaults

actual object PreferencesSettings {
    private val delegate: NSUserDefaults = NSUserDefaults()
    private var settings: Settings = AppleSettings(delegate)

    actual fun settings(): Settings =
        settings
}
```

Figura 4.13: *Shared Preferences actual iOS* relativo alla figura 4.11.

4.3.2 Configurazione di SQLDelight

Per il salvataggio in memoria locale di ordini di acquisto, carichi/scarichi e operazioni di inventario che l'utente in un determinato momento non può/non vuole inviare al server, è necessario utilizzare il database dell'applicazione.
Configurare *SQLDelight* richiede di:

- Definire i driver che dipendono dai diversi target, figure *expect*-4.14, *actual Android*-4.15 e *actual iOS*-4.16. Come per i precedenti casi, quando si usa *expect/actual*, si dovranno aggiungere nel *build.gradle.kts* le specifiche dipendenze delle diverse piattaforme, in questo caso per *Android*:

  com.squareup.sqldelight:android-driver

  mentre per *iOS*:

  com.squareup.sqldelight:native-driver.

- Definire gli *Adapter* per le colonne che hanno campi con tipi non gestiti nativamente da *SQLDelight*. Un esempio è l'attributo *barcodes* in figura 4.17 che richiede di definire un *Adapter* personalizzato per il tipo di dato *List⟨String⟩*. In figura 4.18 è presente l'implementazione dell'*Adapter*.

- Creare un file con estensione .sq nel quale inserire il codice *SQL*.
  É necessario poi nel file *build.gradle.kts* andare a cofigurare *SQLDelight*, in particolare inserendo la posizione del file .sq all'interno dei *package*.

- Creazione delle classi utilizzate per i dati che devono essere salvati su database o che devono essere ritornati agli anelli centrali dell'architettura. Ognuna di queste classi ha una rispettiva *Entity* in cui può essere convertita.

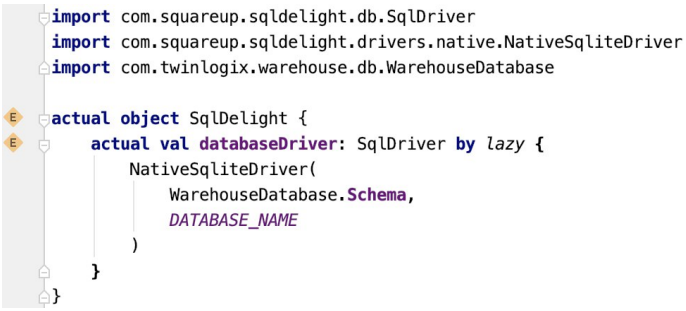
Terminate queste operazioni, durante la *build* del progetto automaticamente *SQLDelight* genera le classi *Kotlin* che, tramite l'oggetto *dao* (figura 4.18), permettono l'utilizzo del database direttamente nel progetto *shared*.

```
expect object SqlDelight {
    val databaseDriver: SqlDriver
}
```

Figura 4.14: *Expect* dei driver del database nella parte condivisa.

```
import com.squareup.sqldelight.android.AndroidSqliteDriver
import com.squareup.sqldelight.db.SqlDriver
import com.twinlogix.warehouse.db.WarehouseDatabase

actual object SqlDelight {
    actual val databaseDriver: SqlDriver by lazy {
        AndroidSqliteDriver(
            WarehouseDatabase.Schema,
            context!!,
            DATABASE_NAME
        )
    }
}
```

Figura 4.15: *Actual Android* dei driver del database, figura 4.14.



```
import com.squareup.sqldelight.db.SqlDriver
import com.squareup.sqldelight.drivers.native.NativeSqliteDriver
import com.twinlogix.warehouse.db.WarehouseDatabase

actual object SqlDelight {
    actual val databaseDriver: SqlDriver by lazy {
        NativeSqliteDriver(
            WarehouseDatabase.Schema,
            DATABASE_NAME
        )
    }
}
```

Figura 4.16: *Actual iOS* dei driver del database, figura 4.14.

```
CREATE TABLE stocks (
    id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
    user_id INTEGER NOT NULL,
    sales_point_id INTEGER NOT NULL,
    barcode TEXT,
    id_sku TEXT,
    stock_status_id TEXT,
    warehouse_enabled INTEGER AS Boolean,
    scan_note TEXT,
    description TEXT,
    barcodes TEXT AS List<String>,
    quantity REAL
);

insertStock:
INSERT OR REPLACE INTO stocks (user_id, sales_point_id, barcode, id_sku, stock_status_id,
warehouse_enabled, scan_note, description, barcodes, quantity)
VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?);

selectAllStocks:
SELECT *
FROM stocks
WHERE user_id = ? AND sales_point_id = ?
ORDER BY id ;

deleteAllStocks:
DELETE FROM stocks
WHERE user_id = ? AND sales_point_id = ?;

deleteSingleStock:
DELETE FROM stocks
WHERE id = ?;
```

Figura 4.17: Codice *SQL* per la creazione della tabella di inventario e per la creazione delle query sulla stessa.

```
object Database {
    private val listOfStringsAdapter: ColumnAdapter<List<String>, String> =
        object : ColumnAdapter<List<String>, String> {
            override fun decode(databaseValue: String): List<String> =
                databaseValue.split( ...delimiters: ",").toCollection(arrayListOf())
            override fun encode(value: List<String>) : String  = value.joinToString(separator = ",")
        }

    val dao: WarehouseDatabase = WarehouseDatabase(
        driver = SqlDelight.databaseDriver,
        stocksAdapter = Stocks.Adapter(
            barcodesAdapter = listOfStringsAdapter
        )
    )
}
```

Figura 4.18: Creazione dell'oggetto *dao* che permette di utilizzare il database. Richiede i driver dipendenti dalla piattaforma e, se necessari, gli *Adapter* che si occupano delle colonne che hanno tipi di dato non gestiti nativamente da *SQLDelight*.

4.3.3 Utilizzo del database nella parte comune

L'utilizzo del database è simile alla gestione delle chiamate remote. Tutte le operazioni che possono essere svolte sul database sono inserite nel *LocalSource*, figura 4.19. Per separare le *Entity* dai dati utilizzati dal database viene adottato lo stesso approccio dei DTO, sezione 4.3.1.

Il *LocalRepository* si occupa di gestire la conversione da *Entity* alle classi utilizzate per i dati del database e viceversa, come mostrato nell'esempio in figura 4.20.

```
class LocalSourceImpl(private val db: WarehouseDatabase) : LocalSource {

    override fun retrieveStocks(salesPointId: Long, userId: Long): List<DbStock> =
        db.warehouseDatabaseQueries.selectAllStocks(
            user_id = userId,
            sales_point_id = salesPointId
        ).executeAsList().map { DbStock(it) }
```

Figura 4.19: *LocalSource* esempio di richiesta dell'inventario salvato nel database.

```
class LocalRepositoryImpl(val localSource: LocalSource) : LocalRepository {

    override fun retrieveStocks(salesPointId: Long, userId: Long): List<Stock> =
        localSource.retrieveStocks(
            salesPointId = salesPointId,
            userId = userId
        ).map { it.toEntity() }
```

Figura 4.20: *LocalRepository* esempio di richiesta dell'inventario salvato nel database.

4.4 Realizzazione dei livelli interni della Clean Architecture

Realizzazione dei due livelli più interni, rosso e giallo, della *Clean Architecture* in figura 3.3. La creazione delle *Entity* che hanno una relazione biunivoca con i DTO è stata trattata nella sezione 4.3.1.

4.4.1 Casi d'uso e ServiceLocator

Casi d'uso

Per ogni operazione necessaria al funzionamento dall'applicazione esiste un caso d'uso, come mostrato nelle figure 4.21 e 4.22. I casi d'uso per lo svolgimento delle funzionalità utilizzano i *Repository* definiti in precedenza.
La scelta di utilizzare l'operatore *invoke* è dettata da una mera scelta di pulizia del codice.

```
class CreatePurchaseOrder(private val repository: RemoteRepository) : UseCase {
    suspend operator fun invoke(purchaseDocument: PurchaseDocument): ServerResponse<Id?>? =
        repository.createPurchaseOrder(purchaseDocument)
}
```

Figura 4.21: Caso d'uso che utilizza il *RemoteRespository* per richiedere la creazione di un ordine di acquisto.

```
class RetrieveStocks (val repository: LocalRepository) : UseCase {
    operator fun invoke(salesPointId: Long, userId: Long): List<Stock> =
        repository.retrieveStocks(
            salesPointId = salesPointId,
            userId = userId
        )
}
```

Figura 4.22: Caso d'uso che utilizza il *LocalRespository* per richiedere l'inventario salvato su database.

4.4.2 ServiceLocator

I *ServiceLocator*, come anticipato nel capitolo di Architettura del progetto, svolgono una funzione simil *Dependency Injection*.
Al loro interno il *RemoteServiceLocator* (figura 4.23) e il *DbServiceLocator* (figura 4.24) creano e mantengono le istanze dei seguenti elementi:

- Casi d'uso.
- Remote e Local Source.
- Remote e Local Repository.

```
object RemoteServiceLocator {
    private val remoteSource = RemoteSourceImpl(httpClientEngine)
    private val remoteRepository: RemoteRepository = RemoteRepositoryImpl(remoteSource)

    // Use case
    val createPurchaseDocument = CreatePurchaseOrder(remoteRepository)
}
```

Figura 4.23: *RemoteServiceLocator* contiene al suo interno casi d'uso, *Source* e *Repository* inerenti con le chiamate remote.

```
object DbServiceLocator {
    private val localSource = LocalSourceImpl(Database.dao)
    private val localRepository = LocalRepositoryImpl(localSource)

    // Use case
    val retrieveStocks = RetrieveStocks(localRepository)
}
```

Figura 4.24: *DbServiceLocator* contiene al suo interno casi d'uso, *Source* e *Repository* inerenti con l'utilizzo del database.

4.4.3 ViewModel

La business logic è gestita dai diversi *ViewModel* che utilizzando le *Entity* e i casi d'uso determinano le funzioni dell'applicazione.
Come mostrato in figura 4.25, ogni *ViewModel* può estendere o il *BaseViewModelImpl* o il *WithErrorViewModelImpl*. La differenza tra i due è data dal fatto che il *WithError-ViewModelImpl* gestisce un flusso di errori che non sempre è necessario.
Il *BaseViewModelImpl*, in figura 4.25, è marcato di arancione in quanto viene utilizzato il meccanismo degli *expect/actual* per definire lo *Scope* di esecuzione delle *Coroutine* nello

specifico *ViewModel*. Ciò ci permette quando il *ViewModel* non è più necessario di, chiamando la *onDestroyViewModel()*, terminare tutti i *Job* attualmente in esecuzione.
Utilizzando la figura 4.26 segue la spiegazione della composizione dei *ViewModel*.

Inizializzazione del ViewModel

I *ViewModel* prendono in ingresso unicamente i casi d'uso dei quali necessitano per svolgere le loro funzioni, ed estendono una classe tra la *BaseViewModelImpl* o la *WithErrorViewModelImpl*. La creazione di un *ViewModel* viene semplificata aggiungendo la funzione statica *create()* che ritorna un'istanza del *ViewModel*.

MutableStateFlow

Ogni *ViewModel* definisce uno o più *Kotlin MutableStateFlow* (esempio: *tokenFlow*) ed espongono un metodo che permetta di effettuare la *collect* del *flow*, sempre nell'esempio il metodo è l'*observeTokenFlow*.
Per questioni di pulizia del codice del *ViewModel* i *Kotlin MutableStateFlow* possono essere decorati come mostrato nell'esempio in figura 4.26. Lo *StateFlowWrapper* aggiunge al *Kotlin MutableStateFlow* uno stato che indica se il valore è cambiato. Ciò è utile in fase di debug o nei *ViewModel* dove è necessario sapere se alla chiusura della vista si devono aggiornare i dati del database.

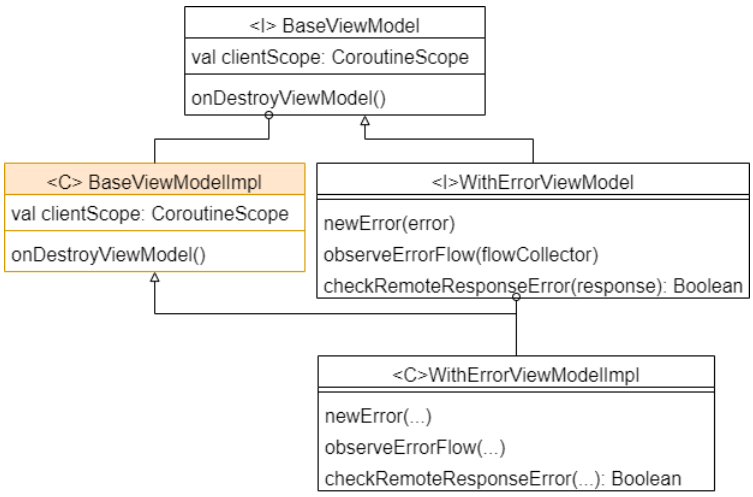


Figura 4.25: Struttura dei *ViewModel*.

Figura 4.26: Esempio di *ViewModel*.

4.4.4 BasicView

Uno degli obiettivi principali del progetto è riuscire a limitare alla creazione delle viste il codice da scrivere nelle singole piattaforme. Per questo motivo sono presenti le *BasicView*, il loro scopo è quello di definire le funzionalità dell'applicazione in modo tale che, nelle singole piattaforme, ci si debba limitare a collegare queste funzionalità con gli eventi generati dagli utenti.

Come mostrato nella figura 4.27, le *BasicView* possono estendere o l'interfaccia *BasicView* o la classe astratta *BasicViewWithError*. La differenza tra le due è che la seconda gestisce al suo interno un *WithErrorViewModel* e quindi aggiunge metodi per registarsi al flusso di errori.

Nelle figure 4.28 e 4.29 è riportato un esempio di *BasicView*. Le *BasicView* si compongono di un'interfaccia e di un singleton (*Kotlin object*) che la implementa. É necessario utilizzare questo approccio in quanto, attualmente, KMP non riesce a convertire un'interfaccia

con metodi implementati in codice *native* per *iOS*. Allo stesso tempo è necessario che la *BasicView* sia un'interfaccia perché le viste devono già, di base, estendere una classe (*Fragment* in *Android* e *ViewController* in *iOS*) e non è supportata ereditarietà multipla.

Interfaccia

Nell'interfaccia delle *BasicView* figura 4.28, vengono definite tutte le funzioni richieste dalla specifica vista che la estenderà. Con *Android*, estesa l'interfaccia, si possono già utilizzare i metodi in quanto già implementati con le definizioni del singleton associato all'interfaccia. Per *iOS* la procedura è trattata in seguito nella sezione 4.6.1.

Singleton

Il singleton in figura 4.29, possiede i *ViewModel* dei quali ha necessità per definire i metodi dell'interfaccia che implementa.

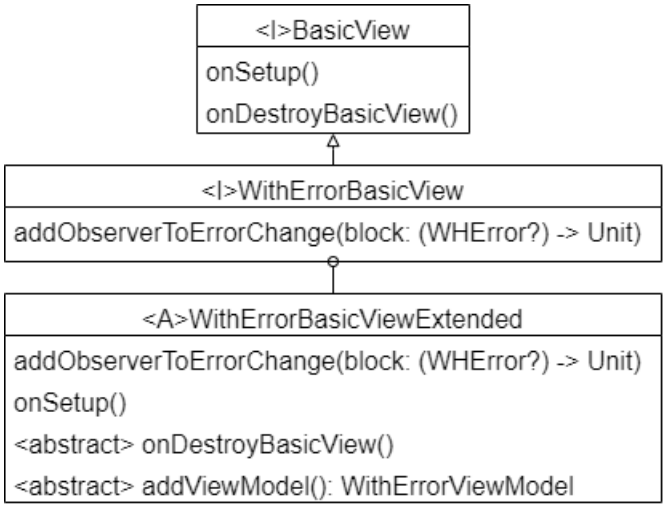
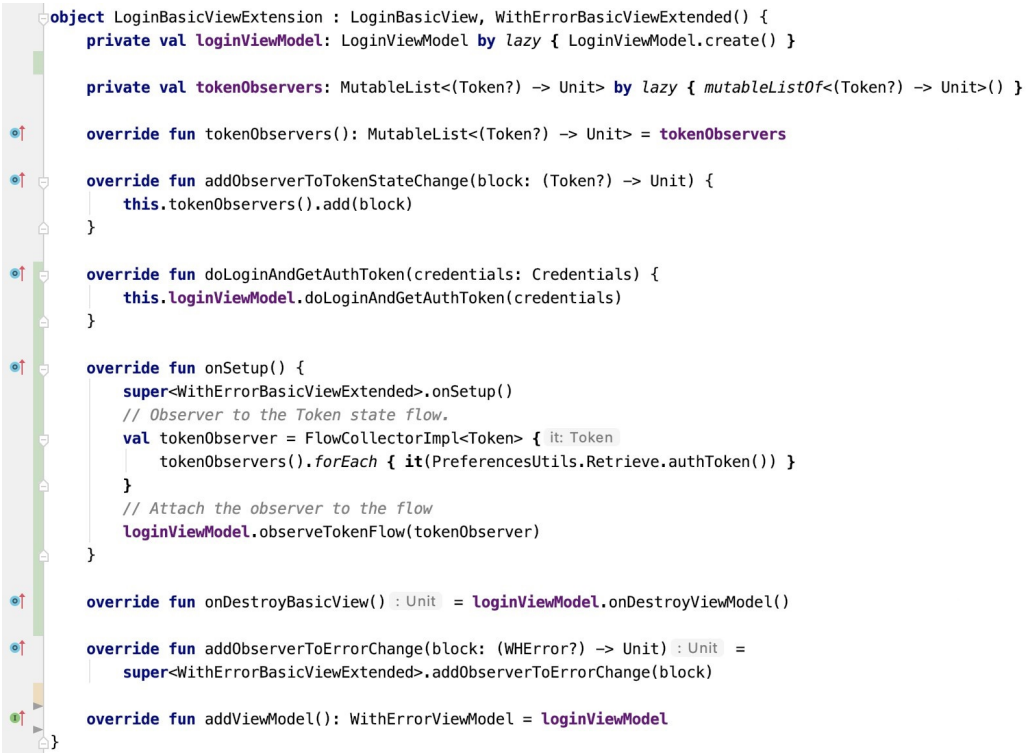


Figura 4.27: Struttura delle *BasicView*.

```
interface LoginBasicView : WithErrorBasicView {
    fun tokenObservers(): MutableList<(Token?) -> Unit> = LoginBasicViewExtension.tokenObservers()

    fun addObserverToTokenStateChange(block: (Token?) -> Unit) : Unit =
        LoginBasicViewExtension.addObserverToTokenStateChange(block)

    fun doLoginAndGetAuthToken(credentials: Credentials) : Unit =
        LoginBasicViewExtension.doLoginAndGetAuthToken(credentials)

    override fun onSetup() : Unit = LoginBasicViewExtension.onSetup()

    override fun onDestroyBasicView() : Unit = LoginBasicViewExtension.onDestroyBasicView()

    override fun addObserverToErrorChange(block: (WHError?) -> Unit) : Unit =
        LoginBasicViewExtension.addObserverToErrorChange(block)
}
```

Figura 4.28: Interfaccia *LoginBasicView*. L'implementazione dei metodi è presa dal *singleton* ad essa associato, figura 4.29.



```
object LoginBasicViewExtension : LoginBasicView, WithErrorBasicViewExtended() {
    private val loginViewModel: LoginViewModel by lazy { LoginViewModel.create() }

    private val tokenObservers: MutableList<(Token?) -> Unit> by lazy { mutableListOf<(Token?) -> Unit>() }

    override fun tokenObservers(): MutableList<(Token?) -> Unit> = tokenObservers

    override fun addObserverToTokenStateChange(block: (Token?) -> Unit) {
        this.tokenObservers().add(block)
    }

    override fun doLoginAndGetAuthToken(credentials: Credentials) {
        this.loginViewModel.doLoginAndGetAuthToken(credentials)
    }

    override fun onSetup() {
        super<WithErrorBasicViewExtended>.onSetup()
        // Observer to the Token state flow.
        val tokenObserver = FlowCollectorImpl<Token> { it: Token
            tokenObservers().forEach { it(PreferencesUtils.Retrieve.authToken()) }
        }
        // Attach the observer to the flow
        loginViewModel.observeTokenFlow(tokenObserver)
    }

    override fun onDestroyBasicView() : Unit = loginViewModel.onDestroyViewModel()

    override fun addObserverToErrorChange(block: (WHError?) -> Unit) : Unit =
        super<WithErrorBasicViewExtended>.addObserverToErrorChange(block)

    override fun addViewModel(): WithErrorViewModel = loginViewModel
}
```

Figura 4.29: *Singleton* che implementa l'interfaccia *LoginBasicView*, figura 4.28.

4.4.5 Navigazione

Anche la navigazione tra le viste dell'applicazione deve risiedere nella parte condivisa. Per ottenere questo obiettivo non viene utilizzato il meccanismo *expect/actual* ma viene definita l'interfaccia *Navigator* che definisce tutte le possibili navigazioni, annessa a questa interfaccia viene definita una variabile globale *navigator* di tipo *Navigator* come *lateinit*. É compito delle diverse piattaforme implementare l'interfaccia e inizializzare la variabile all'avvio dell'applicazione.

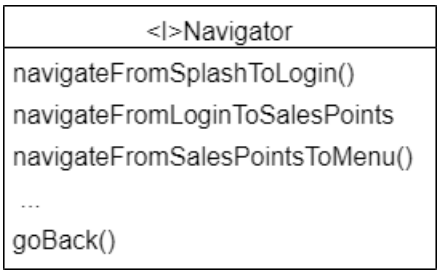


Figura 4.30: Interfaccia *Navigator* definita nella parte condivisa. Deve contenere tutte le navigazioni definite nella figura 2.4.

4.4.6 Testing

Per testare il codice prodotto tramite test *JUnit*, nel progetto condiviso è necessario aggiungere alla struttura del progetto 3.6, dentro *shared*, altri 3 package: *androidTest*, *commonTest* e *iosTest*. Inoltre è necessario aggiungere anche nel *build.gradle.kts* di *shared* questi 3 sottoprogetti in modo da poterne definire le dipendenze.
Per far si che i test siano eseguiti per tutti i target, questi devono essere definiti nel *commonTest* altrimenti verranno lanciati solo per il target nel quale sono stati posizionati.
Per lanciare i test si può utilizzare il *GradleWrapper* con il comando *./gradlew :shared:test*, in tal modo però non verranno eseguiti i test per il target *iOS*. Difatti anche per l'esecuzione dei test lato *iOS* è aggiungere definire un task nel *build.gradle.kts* di *shared* che svolga questo compito.
Terminata questa configurazione iniziale è possibile iniziare a scrivere i test in *commonTest*.
Alcune librerie come *Multiplatform settings* o *Ktor* forniscono dei *mock* direttamente utilizzabili importandoli nel *build.gradle* di shared nel sourceSet dei test. Ma comunque esistono librerie come *Mockk* che permettono di simulare il comportamento di qualsiasi

elemento creato. L'unico limite attualmente è che funzionando solo per *Android* e costringono a scrivere i test nel sotto progetto *androidTest* in modo che non vengano eseguiti lato *iOS*.

4.5 Android

4.5.1 Utilizzo della parte comune

Lo sviluppo dell'applicazione *Android* non risente molto dell'utilizzo di KMP. Difatti essendo il linguaggio di programmazione lo stesso, collegare le viste alle *BasicView* non presenta problemi. Anche importare la parte condivisa è semplice, richiede unicamente di aggiungere *shared* come dipendenza al progetto.
Resta comunque da realizzare l'intera parte grafica per la quale però non si hanno limiti dettati da KMP.
Il debug non è particolarmente complesso, utilizzando *Android Studio* come IDE è possibile sapere dove avvengono i problemi, anche se situati nella parte condivisa, e quindi risolverli. Questo è sicuramente il vantaggio principale per il quale andare a sviluppare l'applicazione *Android* in parallelo con *shared*.
La figura 4.32 è un esempio di vista dell'applicazione *Android*, estendendo la *BasicView* è possibile utilizzare direttamente i metodi definiti in essa (nell'immagine in arancione).

```
class MenuFragment : WithToolbarFragment(), MenuBasicView {
    private lateinit var menuFragmentComponents: MenuFragmentComponents

    override fun onDrawerItemClick(navigation: () -> Unit) : Unit  = navigation()

    @InternalCoroutinesApi
    @ExperimentalStdlibApi
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
        val view = super.onCreateView(inflater, container, savedInstanceState)

        inflateLayoutInMainContainer(view!!, R.layout.fragment_menu)

        menuFragmentComponents = MenuFragmentComponents(view)
        menuFragmentComponents.stockOnClick { navigateToStock() }
        menuFragmentComponents.loadOnClick { navigateToLoadList() }
        menuFragmentComponents.unloadOnClick { navigateToUnloadList() }
        menuFragmentComponents.purchaseOnClick { navigateToPurchaseOrderList() }

        hideAndLockDrawer(hideAndLock = false)
        setupCurrentPositionInTheApp(ViewsWithDrawer.MENU)

        return view
    }

    override fun onBackPressed() {
        CloseAppDialog.create(
            context = context!!, positiveButtonListener = { activity?.finish() }
        )
    }

    override fun onDestroyBasicView() : Unit  = super<WithToolbarFragment>.onDestroyBasicView()

    override fun onSetup() : Unit  = super<WithToolbarFragment>.onSetup()

    companion object {
        @JvmStatic
        fun newInstance() : MenuFragment  = MenuFragment()
    }
}
```

Figura 4.31: Vista esempio dell'applicazione *Android*, fa riferimento all'ultima vista della figura 2.5.

4.5.2 Navigazione

Come definito nella sezione precedente 4.5.2, è necessario implementare l'interfaccia *Navigator*. In *Android* utilizzando il *NavigationComponent* per definire la navigazione è sufficiente, all'avvio dell'applicazione, accedere al *NavController* del *Fragment* nel quale verranno inserite le diverse viste. Ciò ci permette di accedere a tutte le navigazioni che sono state definite nel tool grafico del *NavigationComponent*, figura 2.4.

4.5.3 Svilluppo UI

Il layout della *User Interface* è possibile realizzarlo liberamente inserendo nel *build.gradle.kts* del progetto *Android* le dipendenze necessarie. Ciò è possibile unicamente perchè KMP non imposta limiti sulla parte grafica dell'applicazione.
Come anticipato nella sezione di navigazione 4.5.2, per gestire la navigazione tra le viste è stato utilizzato il *NavigationComponent*. Questo gestisce la navigazione inserendo le viste all'interno di un *Fragment* principale, di conseguenza ogni vista dell'applicazione deve essere un *Fragment*.
In figura 4.32, è mostrata la struttura delle viste. Ciascuna estende *WithToolbarFragment* che fornisce l'accesso al menu *Drawer* e al sistema di *blur* della vista. Tutti i riferimenti ai componenti grafici e i metodi di utilità per le specifiche viste, vengono mantenuti nei *FragmentComponents* di cui ogni vista ha il proprio.
Le viste realizzate in *Android* sono quelle mostrate nel capitolo 2.5.

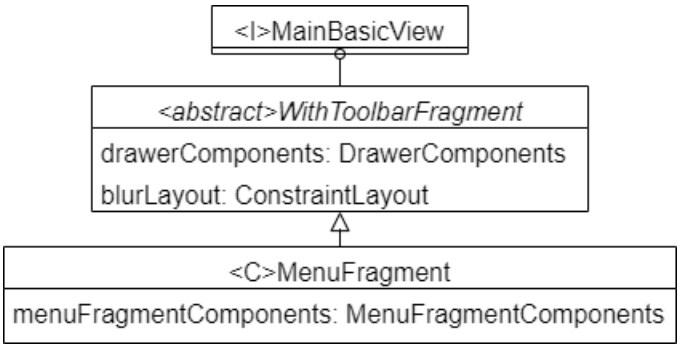


Figura 4.32: Esempio di struttura delle viste *Android*.

4.6 iOS

4.6.1 Utilizzo della parte comune

Lo sviluppo dell'applicazione *iOS* risulta essere più complesso rispetto a quella *Android*. In particolare ciò è dovuto al fatto che non è possibile effettuare il debug anche sul *framework shared* che deve essere importato. Presenti anche le difficoltà, ancora non tutte risolte, dovute alla differenza di linguaggio che richiede una traduzione da *Kotlin* a *Objective-C*. Importare il *framework* nel progetto *Xcode* non è immediato come per *Android*, per poter effettuare la creazione del *framework* è necessario creare un apposito task *Gradle* nel *build.gradle.kts* del progetto *shared* oppure configurare l'utilizzo di *CocoaPods* definendo come ottenere il *framework*. *CocoaPods* è un sistema di gestione delle dipendenze per progetti *Xcode*.
Comunque, ottenuto il *framework* e importatolo correttamente è possibile iniziare a sviluppare l'applicazione *iOS* utilizzando la parte condivisa, per farlo è sufficiente importare il *framework* dove è necessario.
In figura 4.33 è presente la stessa vista portata come esempio per *Android* (immagine 4.32).
In arancione come per l'immagine *Android* sono segnati gli utilizzi delle funzioni della *BasicView* che, anche in questo caso, sono attaccate agli eventi di pressione dei pulsanti della vista.
Come anticipato nella sezione 4.4.4, KMP non è in grado di convertire interfacce con metodi implementati *Kotlin* in qualcosa di equivalente *Objective-C*, quindi i *ViewController* che estendono le *BasicView* non ne vedono l'implementazione e devono provvedere loro stesso a fornirne una. Definendo per ogni *BasicView* un singleton *BasicViewExtension* che ne fornisce l'implementazione è sufficiente utilizzarlo, come mostrato nella figura 4.33 in azzurro, per dare un'implementazione ai metodi della *BasicView* che si sta realizzando.

```
class MenuViewController: UIViewController,
                          MenuBasicView,
                          ViewControllerInNavigation {

    override func viewDidLoad() {
        super.viewDidLoad()
        setCurrentViewController(viewController: self)
        onSetup()
    }

    override func viewWillAppear(_ animated: Bool) {
        super.viewWillAppear(true)
        // Sets the ViewController currently shown to the user.
        setCurrentViewController(viewController: self)
    }

    @IBAction func purchaseButtonOnClick(_ sender: Any) {
        navigateToPurchaseOrderList()
    }
    @IBAction func loadButtonOnClick(_ sender: Any) {
        navigateToLoadList()
    }
    @IBAction func unloadButtonOnClick(_ sender: Any) {
        navigateToUnloadList()
    }
    @IBAction func stockButtonOnClick(_ sender: Any) {
        navigateToStock()
    }

    func navigateToLoadList() {
        MenuBasicViewExtension().navigateToLoadList()
    }

    func navigateToPurchaseOrderList() {
        MenuBasicViewExtension().navigateToPurchaseOrderList()
    }

    func navigateToStock() {
        MenuBasicViewExtension().navigateToStock()
    }

    func navigateToUnloadList() {
        MenuBasicViewExtension().navigateToUnloadList()
    }

    func onDestroyBasicView() {
        MenuBasicViewExtension().onDestroyBasicView()
    }

    func onSetup() {
        MenuBasicViewExtension().onSetup()
    }
}
```

Figura 4.33: Vista esempio dell'applicazione *iOS*, fa riferimento all'ultima vista della figura 2.5.

4.6.2 Navigazione

Lato *iOS* la procedura di navigazione resta simile a quella utilizzata per *Android*. La differenza è che risulta essere necessario impostare manualmente il *ViewController* che attualmente viene mostrato all'utente. Nell'immagine 4.33, estendendo il protocollo *ViewControllerInNavigation* è possibile impostare l'attuale *ViewController* (nell'immagine in rosso).
Avendo un riferimento all'istanza del *ViewController* è possibile accedere alle navigazioni che partono da esso oppure, eseguire l'operazione di *pop* dallo *stack* dei *ViewController*.

4.6.3 Sviluppo UI

L'applicazione *iOS* ha le stesse viste di quella *Android* come mostrato in figura 4.35. Oltre alle minori differenze estetiche o di posizionamento dei bottoni, l'unico cambiamento è la mancanza del menu *Drawer* e l'aggiunta di una vista, accessibile dal menu, che mostra le informazioni dell'utente registrato e del *SalesPoint* selezionato, figura 4.38.
Data la mancanza del *Drawer* la struttura delle viste è più semplice, come mostrato in figura 4.34, ogni *ViewController* si limita ad estendere la *BasicView* e la *ViewControllerInNavigation*. Quest'ultima fornisce delle funzionalità di utilità e tramite il meccanismo degli *extension* di *Swift* ha già i metodi implementati pronti per essere utilizzati.

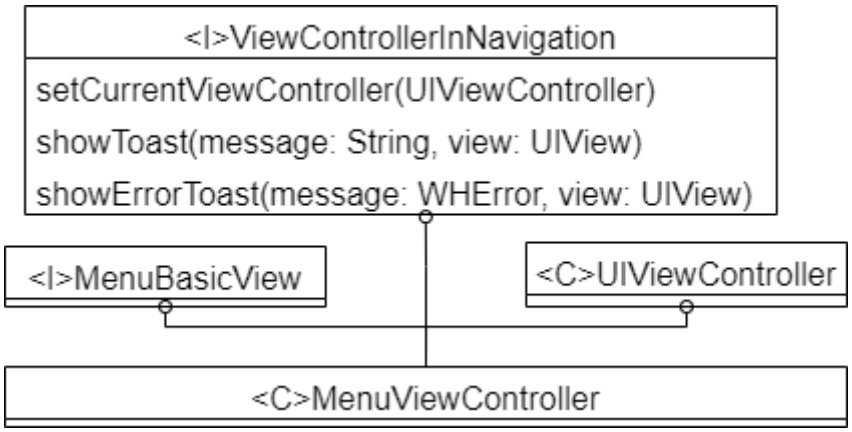


Figura 4.34: Esempio di struttura delle viste *iOS*.

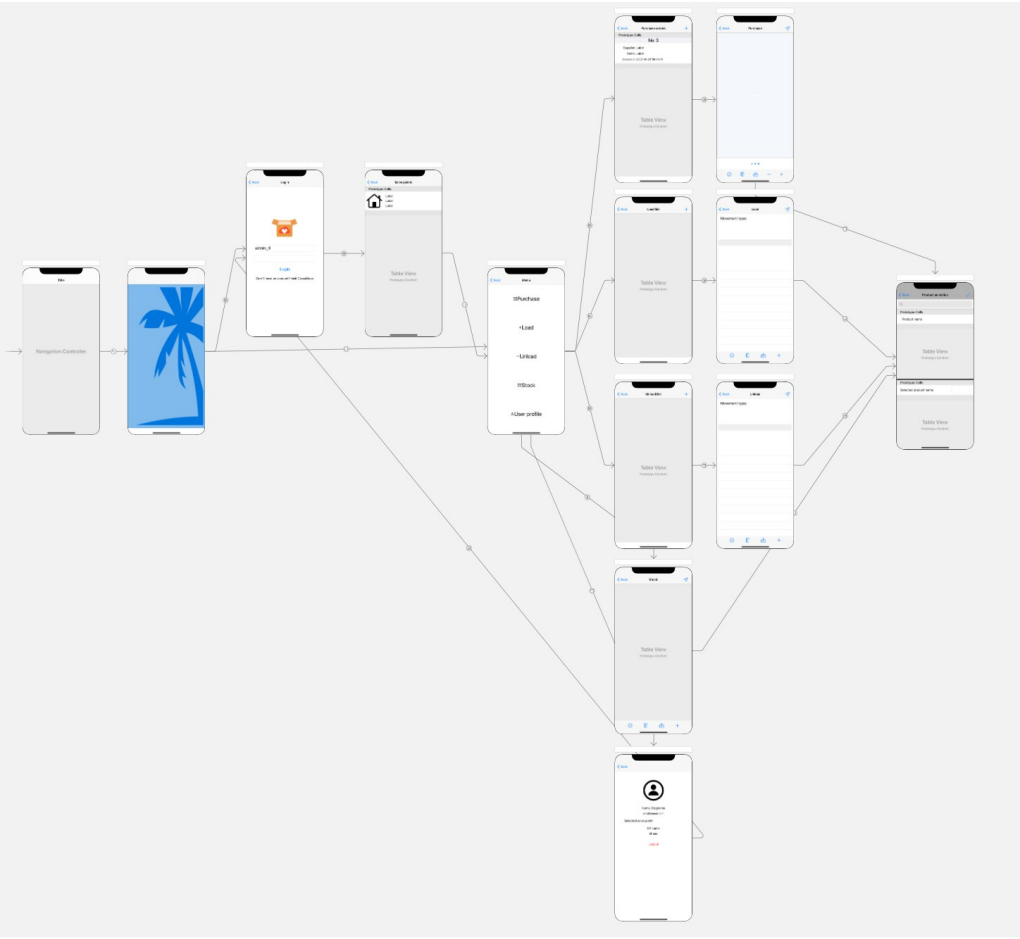


Figura 4.35: Viste dell'applicazione *iOS*.

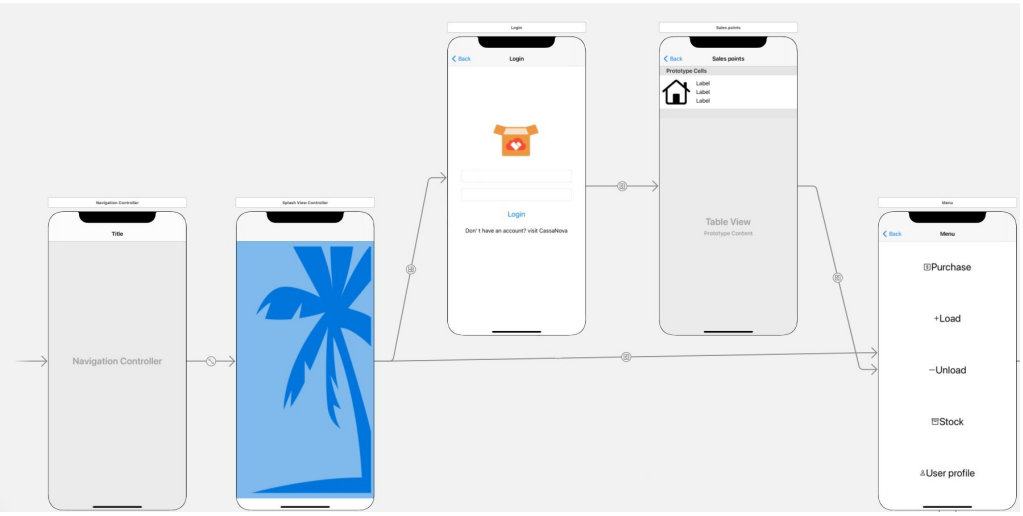


Figura 4.36: Prime viste dell'applicazione *iOS*.

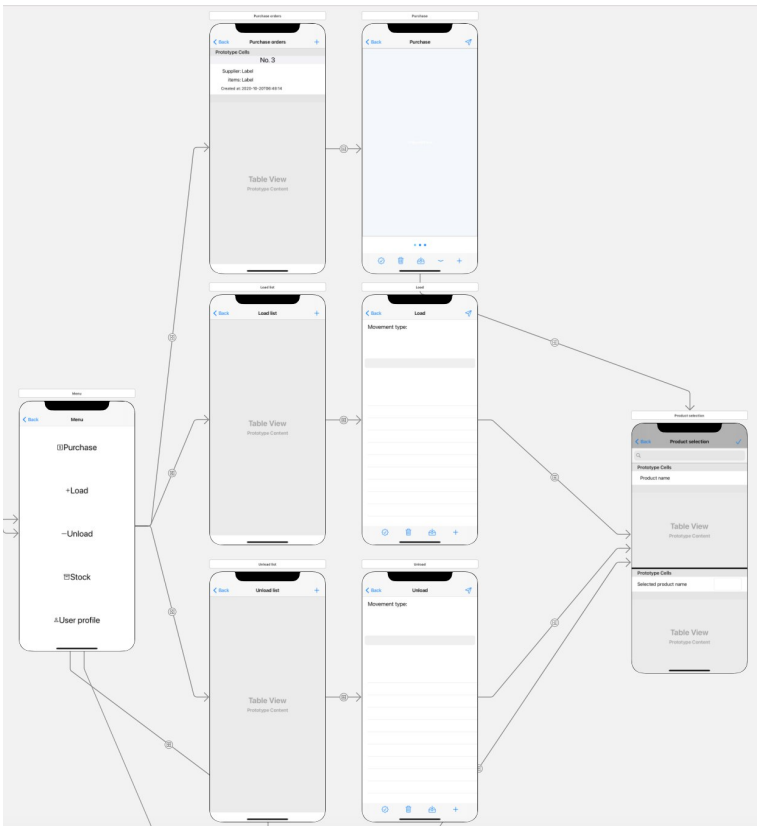


Figura 4.37: Viste di *Purchase*, *Load* e *Unload* dell'applicazione *iOS*.

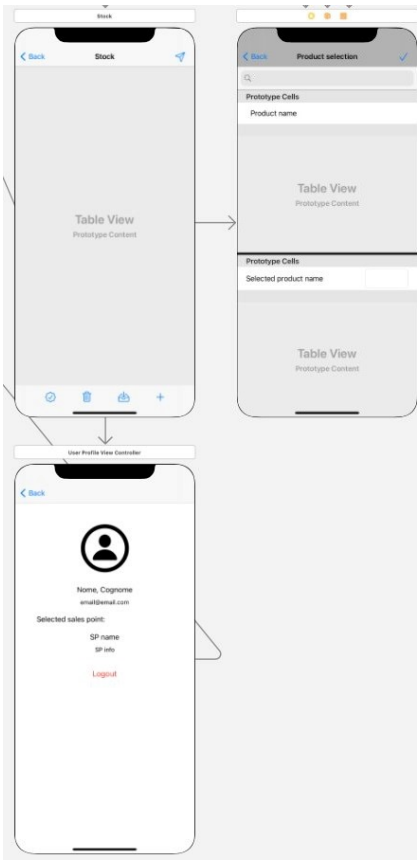


Figura 4.38: Viste di *Stock* e *UserProfile* dell'applicazione *iOS*.

5. Conclusioni

5.1 Vantaggi e svantaggi percepiti

Come riportato nell'*abstract*, qui vado a elencare i vantaggi e gli svantaggi di KMP che ho percepito confrontandoli con uno svolgimento classico di progetto dove l'applicazione *Android* e *iOS* vengono realizzate in modo indipendente.

5.1.1 Vantaggi

Viene notevolmente ridotto il quantitativo di codice duplicato tra le due applicazioni. Ciò va a ridurre i tempi di sviluppo ma rimuove anche la possibilità di avere codice inconsistente tra le due applicazioni. Inoltre il codice prodotto è quasi tutto nativo e rende più semplice la comprensione del progetto.
La rimozione di codice duplicato comporta anche il beneficio di dover scrivere i test una volta sola, lasciando a KMP l'onere di eseguirli per i diversi *target*.
Anche aggiungere nuove funzionalità riguardanti la busines logic trae beneficio dalla base di codice condiviso, di fatti nelle singole piattaforme sarà unicamente necessario ricompilare la parte comune e agganciare le nuove funzioni alla UI.
Tutte queste caratteristiche impattano sulla manutenibilità del codice che risulta essere più semplice ed efficace rispetto ad un approccio classico.
Se il codice comune viene ben modularizzato, come nel caso spiegato in questa tesi, lo sviluppatore delle singole piattaforme si dovrà occupare unicamente della parte grafica, quindi i progetti *Android* e *iOS* conterranno unicamente file relativi alla UI. Ciò riduce la complessità iniziale che incontra una nuova risorsa spostata nel progetto.
A riguardo della UI, KMP rispetto a soluzioni simili lascia totale libertà. Non agisce in ottica di ridurre anche il codice duplicato per la creazione della parte grafica, ma preferisce permettere agli sviluppatori di utilizzare tutte le funzionalità fornite nativamente dalle diverse piattaforme.

5.1.2 Svantaggi

Il primo svantaggio è dovuto al fatto che attualmente KMP è una tecnologia nuova per la quale non esistono ancora librerie che permettano di svolgere o di accedere, in modo semplice, ad alcune funzionalità dei dispositivi *Android* e *iOS*. Ciò, secondo me, limita il range di applicazione per le quali KMP può essere considerata una tecnologia conveniente in termini di tempi di sviluppo. Inoltre, essendo soggetta a cambiamenti frequenti, potrebbe essere richiesto mantenere il progetto aggiornato con le ultime versioni per poter sfruttare nuove librerie che in futuro verranno realizzate. Aggiornare il progetto a una nuova versione di KMP però può richiedere modifiche non banali.
La configurazione iniziale del progetto risulta essere particolarmente lunga e complessa ma è necessario definirla sin da subito assieme all'architettura che si intende utilizzare. Attualmente esistono tutorial che mostrano esempi di configurazione e definizione dell'architettura di progetti KMP, ma sono tutti esempi molto semplici che necessitano di modifiche per poter essere adattati a progetti di dimensioni e complessità maggiore.
Un altro svantaggio è il debug in *iOS* che al contrario di quello in *Android* non permette di debuggare anche la porzione di codice condivisa. Questo rende la scoperta della causa di un errore particolarmente complessa in quanto l'eccezione fornita non è sempre di particolare aiuto e attualmente online non sono presenti molte informazioni su KMP e sui vari problemi esistenti. É da considerare che il tempo necessario per la risoluzione degli errori durante lo sviluppo *iOS* aumenta. Per riportare un semplice caso d'esempio, con *SQLDelight* si può ottenere l'id dell'ultimo elemento inserito nel database. Per farlo è necessario aggiungere l'apposita query nel file *.sq* dopodichè viene generata la funzione *Kotlin* ad essa associata. Questa funzione viene utilizzata nel *LocalSource* e quindi nella parte comune. Con *Android* è possibile richiamarla nel seguente modo:

Algorithm 1: Esempio di problema nel codice comune.

```
1 fun getLastInsertedRowId();
  Output: id : Long
2 [...]
3 return db.getLastInsertedRowId()
```

Ma durante lo sviluppo *iOS* che riguardava l'utilizzo del database ho ricevuto un'eccezione che, dopo varie ricerche, si è rivelata essere dovuta al fatto che è necessario per *iOS* richiamare la funzione *db.getLastInsertedRowId()* dentro una transazione.

Il *testing* deve essere migliorato, come riportato nella sezione 4.4.6 ancora non esistono librerie che permettano di creare dei *mock* direttamente nel progetto *shared*. Al momento questo limita l'esecuzione su piattaforma *iOS* di test della parte comune.

5.2 Considerazioni finali

Anche considerando la mia limitata esperienza nel campo ho percepito che KMP è una tecnologia che in alcuni progetti può portare benefici, anche in termini di riduzione dei tempi di sviluppo, ma ancora non è il *silver bullet* dei progetti multi-piattaforma in quanto ancora molto recente e soggetta a cambiamenti.
Lasciare libertà nello svolgimento della parte grafica permette di realizzare applicazioni più distintive ma allo stesso tempo KMP rimuovere la duplicazione della business logic permettendo di avere un codice più facilmente testabile e manutenibile.

5.3 Sviluppi futuri

Il progetto è quasi totalmente ultimato. Resta da definire una gestione più specifica degli errori mostrati all'utente e aggiungere all'applicazione *iOS* alcuni dettagli grafici che migliorino la *User eXperience*.
Inoltre, prima di mettere le due applicazioni in produzione, è necessario svolgere un adeguato periodo di test in quanto, a causa del tempo limitato, non ho potuto dedicare molto tempo alla realizzazione di test automatizzati.