Politecnico di Torino

Corso di Laurea Magistrale in INGEGNERIA DEL CINEMA E DEI MEZZI DI COMUNICAZIONE

Tesi di Laurea Magistrale

Studio di Kotlin Multiplatform Mobile come strumento per lo sviluppo di applicazioni mobile cross-platform

Relatore

Prof. Giovanni MALNATI

Candidato

Fabio PELLONE

Luglio 2022

Abstract

Negli ultimi anni l'utilizzo di smartphone è cresciuto notevolmente, portando in molti a decidere di sviluppare applicazioni per dispositivi mobile. Una delle scelte più diffuse è quella di sviluppare nativamente per una delle due piattaforme più popolari, Android ed iOS, ma sono nati, nel tempo, anche degli strumenti per lo sviluppo di applicazioni che possano essere eseguite su entrambe. In questa tesi è stato studiato Kotlin Multiplatform Mobile, un framework per lo sviluppo mobile cross-platform di recente creazione sviluppato da JetBrains, azienda che ha creato anche il linguaggio attualmente più utilizzato per sviluppare applicazioni per Android, Kotlin. Per analizzare il framework e verificare se questo possa essere utilizzato in produzione, è stata creata un'applicazione cross-platform prendendo spunto da un'app Android già esistente, Kilood, sviluppata per il corso di Digital Interaction Design. In fase di progettazione è stata posta particolare attenzione sulla scelta dell'architettura più efficace e delle librerie più utili tra quelle compatibili. Nello specifico, è stato scelto di seguire il pattern architetturale Model-View-ViewModel (MVVM) seguendo i principi della Clean Architecture. Dal momento che la tesi è stata scritta in collaborazione con Synesthesia Srl, è stato utilizzato il pattern architetturale attualmente in uso nell'azienda, che presenta un'ulteriore suddivisione dei livelli logici definiti da MVVM e che consente una migliore organizzazione del codice in un progetto di questo tipo. Per il codice comune sono state scelte delle librerie già molto utilizzate per lo sviluppo multiplatform per via della loro ottima integrazione con il framework. Il codice che gestisce la composizione della UI deve essere specifico per le piattaforme, perciò è stato necessario scegliere due librerie differenti per implementarlo: Jetpack Compose per Android e SwiftUI per iOS. Le due librerie, in ogni caso, sono molto simili in quanto entrambe sfruttano il paradigma della programmazione dichiarativa per rendere la creazione della UI molto più intuitiva e semplice, rendendo il codice più efficiente, oltre che più facile da leggere. Durante lo sviluppo è stato necessario sopperire ad alcune limitazioni del framework creando delle classi e funzioni specifiche per permettere, in particolare, l'utilizzo dei Flow anche sulla piattaforma iOS ed una gestione delle chiamate asincrone efficiente su entrambe le piattaforme.

Indice

Elenco delle figure

Elenco dei listati

Capitolo 1

Storia del mercato mobile

Negli ultimi anni l'utilizzo e la diffusione degli smartphone è cresciuto vertiginosamente. Nel 2016, ad esempio, il numero di utenti connessi ad internet mediante dispositivi mobile superava per la prima volta quello di utenti connessi con un computer desktop[1][2]. La loro popolarità si deve, in parte, anche al grande numero di applicazioni disponibili per tali dispositivi che spaziano dalla produttività all'intrattenimento, dalle piattaforme social ai giochi. Ad oggi sviluppare e distribuire un'app mobile è molto semplice, i linguaggi di sviluppo si sono evoluti per essere di facile utilizzo e gli store delle piattaforme riducono, se non addirittura eliminano, l'onere della distribuzione agli sviluppatori. Inizialmente, però, il mercato di software mobile era molto diverso, frammentato e complesso, sia a causa dell'assenza di piattaforme di distribuzione e kit di sviluppo, sia nativi che multipiattaforma, intuitivi e di facile utilizzo, sia per via dell'immaturità del mercato.

1.1 La nascita del mercato di software mobile

Nel 1984 fu commercializzato lo Psion Organiser[3], un dispositivo dall'aspetto di una calcolatrice, ma che univa le funzionalità delle calcolatrici programmabili a quelle delle agende digitali, ovvero dei database portatili nei quali immagazzinare informazioni come indirizzi e numeri di telefono, considerato il primo vero precursore degli smartphone. Una delle caratteristiche più rilevanti dell'Organiser era la compatibilità con dei supporti di archiviazione molto compatti, grandi quanto una gomma da cancellare, detti Datapak, che potevano contenere programmi di vario tipo, in alcuni casi anche scritti dagli utenti[4]. Questi erano inizialmente prodotti esclusivamente dalla Psion stessa, ma, grazie alla popolarità che il secondo dispositivo della serie, l'Organiser II[5], riuscì a raggiungere, anche altre aziende come la Maritek e la Harvester Information Ltd. cominciarono a produrre dei Datapak con software di diverso tipo. In breve tempo vennero distribuiti applicativi scientifici,

per la produttività, per la gestione delle finanze e finanche delle collezioni di giochi[6] e nacque, così, il primo vero mercato di applicativi per dispositivi tascabili o, per usare un termine più moderno, mobile.

Il successo dei dispositivi della serie Organiser, ai quali si aggiunse nel 1991 lo Psion Series 3[7] che, differentemente dai predecessori, aveva l'aspetto di un vero e proprio computer in miniatura, con uno schermo a cristalli liquidi più grande ed una tastiera QWERTY racchiusi in un design a conchiglia, portò anche altre aziende a sviluppare i propri computer tascabili. Nei primi anni '90, infatti, Apple lanciò il Newton MessagePad[8] e Tandy lo Zoomer[9], entrambi profondamente diversi dai dispositivi prodotti fino a quel momento dalla Psion in quanto presentavano un numero estremamente ridotto di tasti ed un touch screen resistivo. Da quel momento a questi dispositivi venne dato il nome Personal Digital Assistant, o PDA, coniato da Apple al CES del 1992[10]. Nonostante il newton MessagePad e lo Zoomer non riuscirono ad eguagliare il successo dello Psion Series 3, la presenza di concorrenza e di alternative fece crescere l'interesse nel pubblico. La maturità del linguaggio OPL[11], ovvero Organiser Programming Language, utilizzato sul PDA della Psion e la semplicità di NewtonScript[12], linguaggio creato da Apple per facilitare lo sviluppo su Newton OS, invogliarono, inoltre un gran numero di sviluppatori a cominciare a creare applicativi per questi dispositivi e condividerli con altri utenti. Quelli successivi al lancio di Psion Series 3 e MessagePad, infatti, furono i primi anni del World Wide Web e furono diversi i siti web[13][14] che nacquero per raccogliere applicativi distribuiti sia gratuitamente che con licenza shareware, molto popolare in quel periodo, che permetteva di scaricare e provare il software per un periodo limitato per poi pagare lo sviluppatore per continuare ad utilizzarlo. Nacque, così, grazie al Web un mercato di software per dispositivi mobile anche per i piccoli sviluppatori che non potevano permettersi di distribuire copie fisiche dei propri programmi o che semplicemente volevano condividerli con altri utenti.

Nella seconda metà degli anni '90 il mercato dei PDA si espanse ulteriormente e questi divennero oggetti di massa. In particolare riscossero un grande successo i dispositivi della serie Pilot della Palm, azienda fondata dall'ideatore del sopracitato Tandy Zoomer Jeff Hawikins[15], ed al sistema operativo Windows CE, sviluppato da Microsoft per portare l'interfaccia e le funzionalità di Windows 95 su una grande varietà di prodotti compatti e con diverse architetture di processori[16]. I dispositivi prodotti da Palm riscossero un grande successo grazie alla loro compattezza, comodità d'uso, convenienza e ad un sistema operativo, denominato Palm OS, intelligentemente progettato con un'interfaccia grafica fortemente ottimizzata per l'utilizzo del touchscreen che rendeva l'esperienza degli utenti semplice ed intuitiva. Windows CE, diversamente, era stato progettato per fornire un'esperienza omogenea su dispositivi prodotti da altre aziende, come LG, Casio, Compaq e

HP, con un fattore di forma più classico, quindi con schermi più piccoli e tastiera QWERTY. Benché l'interfaccia di Windows 95 lo rendesse familiare per molti utenti, i limiti dettati dalle caratteristiche dei PDA dell'epoca rendevano il sistema operativo meno usabile di Palm OS impedendo così, anche a causa dei prezzi più elevati, ai dispositivi con Windows CE, successivamente denominati Pocket PC, di raggiungere la popolarità dei Palm Pilot[17]. In entrambi i casi, però, le community di utenti furono molto attive nel creare e distribuire su siti web appositi software creato dagli utenti stessi[18], come già successo per i dispositivi precedenti. Entrambi i sistemi operativi permettevano, infatti, di eseguire programmi sviluppati in C e C++, due linguaggi molto popolari in quegli anni, rendendo così molto semplice lo sviluppo di applicativi più o meno complessi per PDA. Nonostante ciò, a causa delle differenze tra le loro architetture, software dedicati ad una piattaforma non erano compatibili con l'altra e viceversa. Per la prima volta conoscendo un solo linguaggio era possibile sviluppare applicativi per più piattaforme mobile, ma anche se così alcune parti potevano essere riutilizzate, era ancora necessario riadattare buona parte del codice per ogni specifica piattaforma. Si era ancora, quindi, ben lungi dallo sviluppo multipiattaforma mobile vero e proprio.

1.2 I telefoni diventano smart

Alla fine degli anni '90, come visto, il mercato di applicativi per dispositivi compatti era interamente incentrato a software nativi dedicato ai Personal Digital Assistant, ma nello stesso periodo anche i telefoni cellulari si stavano evolvendo, implementando sempre più funzionalità che fino a poco tempo prima erano esclusivamente presenti in dispositivi più performanti come, appunto, i PDA. Era sempre più facile trovare funzioni di calcolatrice, calendario, agenda e, negli ultimi anni del decennio, anche i primi giochi, come il popolarissimo Snake che ha fatto il suo debutto sul Nokia 6110 nel 1998[19]. Per diversi anni, però, gli unici applicativi disponibili per cellulari furano solo quelli già presenti nativamente, ma non fu necessario aspettare a lungo perché le cose cambiassero. Nei primissimi anni 2000 nacquero molti strumenti e tecnologie che cambiarono il modo di concepire i dispositivi mobile, a partire dal 2001 con la commercializzazione dei primi telefoni con Symbian 6.1[20] e, soprattutto, di telefoni compatibili con Java Micro Edition[21]. Symbian 6.1 era il nuovo nome dato dalla sua sesta versione ad EPOC32, il sistema operativo sviluppato da Psion per i suoi dispositivi palmari a partire dall'Organiser Series 3, dopo che la stessa Psion aveva unito le forze con Ericsson, Nokia e Motorola per sviluppare software per telefoni cellulari[22]. Tale sistema operativo era stato intelligentemente creato con più di un'interfaccia utente per poter essere adattato a dispositivi diversi, le varie interfacce che sono state nel tempo create erano, infatti, ottimizzate per sfruttare al meglio schermi più grandi, touchscreen,

tastiere o una combinazione di queste. Questa caratteristica, però, per quanto lo rendesse versatile, limitava molto gli sviluppatori in quanto software creato per un'interfaccia non poteva essere utilizzato sulle altre, costringendoli a dover scrivere più volte la stessa applicazione per renderla disponibile su tutti i dispositivi Symbian. Ciò si andava, poi, ad aggiungere alla difficoltà che molti avevano nel cominciare a sviluppare per la piattaforma a causa della scelta di supportare solo i linguaggi OPL e Symbian C++, una versione del C++ leggermente modificata. I cellulari Symbian, però, furono anche tra i primi a supportare Java Micro Edition[23], o più semplicemente Java ME o J2ME, una piattaforma nata poco tempo prima con lo scopo di semplificare lo sviluppo e la distribuzione di software per dispositivi embedded e mobile basato, come suggerisce il nome, sul linguaggio di programmazione ad oggetti Java, un linguaggio estremamente popolare e molto utilizzato ancora adesso. Lo scopo di J2ME era di espandere il motto di Java, WORA, ovvero "Write Once, Run Anywhere", anche a dispositivi con capacità di calcolo e specifiche più basse. Alla base della piattaforma vi erano due concetti utilizzati per distinguere gruppi di dispositivi, le configurazioni, definite come la combinazione di una macchina virtuale ed un set di API rappresentative di una relativamente ampia gamma di dispositivi, ed i profili, ovvero un set di API specifiche per una tipologia ristretta di dispositivi. Nel 1999 Motorola, col supporto di aziende tra cui Psion, Siemens ed Ericsson, propose di creare, sulla base della "Connected, Limited Device Configuration", o CLDC, una configurazione dedicata a dispositivi con scarse capacità di calcolo, un profilo specifico per dispositivi piccoli, con risorse limitate, alimentati a batteria, con supporto a connessioni wireless e con interfacce più o meno sofisticate[24]. La creazione di tale profilo, denominato Mobile Information Device Profile, o MIDP, facilitò notevolmente lo sviluppo di software supportati da un vasto numero di dispositivi mobile in quanto, come già accennato, in breve tempo anche le altre aziende produttrici di telefoni cellulari si accorsero del potenziale di J2ME ed aggiunsero il supporto al MIDP ai loro dispositivi. Nonostante la coppia CLDC e MIDP fosse pensata anche per dispositivi con caratteristiche simili a quelle dei PDA, Sun Microsystem, azienda produttrice di tutte le edizioni di Java, rilasciò un'implementazione del sopracitato profilo solamente per Palm OS[25], ignorando altri sistemi operativi altrettanto diffusi come Windows CE. Per quest'ultimo, infatti, le uniche implementazioni disponibili furono proprietarie di aziende terze, come CrEme di NSIcom[26], o dedicate a Java Standard Edition, come Mysaifu JVM [27]. Ciò, chiaramente, ridusse il numero di Pocket PC sui quali fosse possibile eseguire applicativi sviluppati per Java Micro Edition. Con la rapida diffusione di Java ME gli sviluppatori ebbero per la prima volta la possibilità di scegliere se creare applicazioni native più performanti, anche per il promettente Symbian OS, o applicazioni eseguibili su una gamma di dispositivi più vasta e varia, anche se al prezzo di prestazioni leggermente inferiori. Il mercato dei software mobile crebbe, così, ulteriormente, con il numero di

applicativi per telefoni cellulari che aumentava a discapito di quelli per PDA[28]. L'interesse per questi ultimi, infatti, andò scemando perché, nonostante l'aggiunta delle funzionalità di telefonia nei nuovi modelli, cominciarono ad essere considerati più come dispositivi specifici per la produttività rispetto ai cellulari e, di conseguenza, cedettero il passo. Anche il mercato dei software mobile si spostò verso gli applicativi per telefoni cellulari, in particolare i giochi, relativamente semplici da sviluppare e con un pubblico più vasto rispetto a software per la produttività. Piccoli sviluppatori e utenti alle prime armi iniziarono a distribuire i propri applicativi su forum e siti aggregatori[29][30], mentre case di sviluppo e sviluppatori più grandi potevano usare mezzi più efficaci e semplici per gli utenti come i portali WAP, che consentivano agli utenti di scaricare i software collegandosi a dei link specifici[31]. Nonostante tali metodi di distribuzione non fossero particolarmente efficienti, richiedendo agli utenti un lavoro di ricerca notevole, e non sempre particolarmente semplice, il mercato continuò a crescere in questa direzione, almeno fino 2008, quando Apple lo stravolse completamente.

1.3 Il primo iPhone e l'App Store

La rivoluzione di Apple cominciò nel 2007, quando commercializzarono il primo iPhone, presentato come un dispositivo rivoluzionario, in grado di unire le funzionalità di un telefono, quelle di un iPod e quelle di un palmare, il tutto in un fattore di forma inusuale, i tasti presenti erano solo quattro, due per la regolazione del volume, uno per blocco e spegnimento ed uno per tornare alla schermata principale, cosa resa possibile dal grande schermo capacitivo[32]. Questo fu una delle sue grandi innovazioni, il primo smartphone con schermo capacitivo era uscito appena qualche settimana prima ed il pubblico era ancora abituato a schermi resistivi, poco precisi se non utilizzando un pennino, perciò lo schermo dell'iPhone risultò quasi rivoluzionario ai più. La già enorme popolarità dell'azienda e il fascino del dispositivo fece sì che, con un solo prodotto in vendita, Apple riuscisse ad ottenere quasi il 30% del mercato degli smartphone negli Stati Uniti in meno di sei mesi[33]. Inizialmente, però, le uniche applicazioni utilizzabili sul dispositivo erano quelle già installate all'acquisto e, nonostante Apple invitasse gli sviluppatori di terze parti a sviluppare web app, supportate dall'iPhone, questa scelta non fu molto apprezzata e, dopo mesi di richieste, nel marzo del 2008 venne annunciato un aggiornamento software denominato iPhone 2.0 che includeva il primo SDK per sviluppare applicazioni nativamente per la piattaforma[34]. Nella stessa conferenza, però, Apple annunciò, tra le altre cose, anche una delle novità più rivoluzionarie per il mercato del software mobile: l'App Store, un'applicazione dove trovare tutte le applicazioni disponibili per iPhone. Grazie all'App Store Apple semplificò notevolmente la distribuzione sia per gli utenti, semplificando la ricerca

di applicazioni compatibili per il proprio dispositivo, che per gli sviluppatori, riducendo l'onere della distribuzione e rendendo molto più immediato ed efficiente la vendita. Un tentativo simile era stato fatto da Nokia tra il 2006 ed il 2007, con due diversi servizi, Nokia Donwload![35], inizialmente denominato Catalogs e contenente principalmente prodotti a pagamento, e MOSH[36], contrazione di "Mobilize and Share", nato come social network in cui gli utenti potessero caricare ogni tipo di contenuto, applicazioni comprese, permettendo, così, agli sviluppatori di caricare i propri applicativi rendendoli disponibili gratuitamente. L'esperienza era, però, molto più complessa e frustrante per gli utenti in quanto frammentata su due diverse piattaforme, una delle quali, MOSH, priva di controllo e, di conseguenza, ricca di materiale di pessima qualità se non anche illegale. L'esperienza molto più semplice ed efficiente dell'App Store, invece, garantì a questo un successo immediato, ufficialmente aperto a luglio con 500 applicazioni, a settembre 2008 ne contava già 3000, con ben 100 milioni di download[37] ed il suo successo portò molti concorrenti a creare qualcosa di simile. Nokia unificò le sue piattaforme Download! e MOSH per creare Ovi Store nel 2009[38], nello stesso anno anche Microsoft aprì il Windows Marketplace for mobile per tutti i dispositivi con Windows Mobile 6.5[39], sistema operativo per smartphone basato su kernel Windows CE, così come Palm con il Palm Pre App Catalog[40] e BlackBerry con il BlackBerry App World[41]. Questa scelta risultò troppo tardiva in quanto Apple aveva ormai consolidato la propria presenza sul mercato anche con l'uscita del suo secondo smartphone, iPhone 3G, a discapito di tutti gli altri produttori. A farne le spese furono soprattutto i dispositivi Symbian, sistema operativo più diffuso prima della commercializzazione di iPhone, ma che dinanzi al software di quest'ultimo mostrava tutti i suoi più grandi difetti, in primis l'inefficienza e la già discussa frammentazione delle interfacce grafiche, oltre ad un'esperienza d'uso più macchinosa e meno piacevole[42]. Non servì a molto neanche il tentativo di risollevare le sorti del sistema operativo da parte di Nokia che, nel 2008, ne divenne unica proprietaria acquistandone tutte le quote azionarie e annunciò di volerlo rendere open source con lo scopo di permettere a tutti, aziende e sviluppatori, di avere accesso alla sua tecnologia così da favorire la possibilità di scelta degli utenti[43]. Purtroppo, però, nello stesso anno vide la luce anche un altro sistema operativo open source destinato ad avere una sorte decisamente migliore. Nell'ottobre del 2008, venne commercializzato il primo smartphone con Android[44], sistema operativo open source di Google che forniva un'esperienza più similare a quella offerta dagli iPhone, più responsivo e ottimizzato e, soprattutto, sin da subito provvisto del proprio negozio, con anche un buon numero di applicazioni. La diffusione dei dispositivi Android non fu rapida come quella degli smartphone Apple, fu più lenta e lineare, ma nel giro di soli quattro anni il sistema operativo riuscì a raggiungere e superare le quote di mercato globali sia di iOS che di Symbian, stabilendosi, poi, saldamente come il più diffuso negli anni a seguire[45].

1.4 Web app ed i primi framework multipiattaforma

Negli anni immediatamente successivi alla commercializzazione del primo iPhone il mercato dei sistemi operativi per smartphone era, quindi, notevolmente frammentato. Dopo l'esplosione iniziale, iOS si assestò intorno al 20% abbastanza stabilmente, Symbian subì un leggero calo, ma riuscì a resistere fino al 2012, dopo che Nokia annunciò di voler adottare come sistema principale per i propri smartphone Windows Phone, successore di Windows Mobile e sistema piuttosto di nicchia, BlackBerry crebbe lentamente fino al 2010, quando cominciò a perdere terreno in favore di Android, che continuava costante la sua crescita. Se da un lato la presenza di tanti sistemi operativi dava agli utenti la possibilità di scegliere quello che più si adattasse ai loro gusti e preferenze, dall'altro rendeva più complesso sviluppare applicazioni che potessero raggiungere un grande numero di persone. Ogni piattaforma, infatti, utilizzava architetture, linguaggi ed SDK differenti e sviluppare un'applicazione nativa per ognuna sarebbe stato davvero oneroso. La piattaforma Java Micro Edition, che per qualche anno grazie a MIDP e CLDC era riuscita ad assicurare la compatibilità delle applicativi tra più piattaforme, era, poi, ormai supportata solo da Symbian ed in parte da BlackBerry OS che, essendo basato su un superset di Java ME[46] consentiva di eseguire alcune applicazioni. Android, iOS e Windows Phone, al contrario, supportavano soltanto software nativi per le rispettive piattaforme. In un primo periodo l'unico modo per raggiungere dispositivi mobile con sistemi operativi diversi senza sviluppare applicazioni native furono le web app che, tuttavia, come accennato nel paragrafo precedente, non erano molto apprezzate per via della loro eccessiva semplicità e della scomodità di utilizzo. Erano, infatti, sostanzialmente dei siti web ottimizzati per utilizzo da dispositivi mobile, quindi potevano essere raggiunte solo tramite un browser e potevano accedere a pochissime delle funzionalità native dei dispositivi.

Nel 2009, però, Nitobi creò PhoneGap[47], un framework open source che permetteva di creare, utilizzando solo strumenti per lo sviluppo web, applicazioni che potevano essere eseguite su piattaforme mobile differenti, inizialmente su Android ed iOS e successivamente anche su Symbian, Palm OS, BlackBerry OS e Windows Phone. Le applicazioni create non erano vere e proprie applicazioni compilate in codice nativo, ma applicazioni native in cui un contenitore eseguiva una web app fornendo, in tal modo, anche l'accesso ad alcune funzionalità native del dispositivo. PhoneGap fu, quindi, il primo framework per la creazione di applicazioni ibride, trattate nel dettaglio nel capitolo 3.1, e che confluì successivamente nel progetto Apache Cordova, uno degli strumenti più utilizzati anche attualmente nello sviluppo di applicazioni ibride. Nello stesso periodo fecero la loro comparsa anche framework multipiattaforma come Titanium Appcelerator e MoSync che,

diversamente da PhoneGap, compilavano codice scritto in HTML e JavaScript in codice nativo delle piattaforme, creando delle applicazioni native a tutti gli effetti. Framework di questo tipo permettevano di sviluppare applicazioni con prestazioni migliori delle applicazioni ibride e fornivano l'accesso ad un numero tendenzialmente maggiore di funzionalità native del dispositivo. Molte aziende e fondazioni decisero, quindi, di creare i propri framework multipiattaforma o per applicazioni ibride, fornendo, così, un gran numero di alternative da poter scegliere per sviluppare applicazioni cross-platform. Supportare le sempre nuove funzionalità native di diverse piattaforme, però, ne rendeva il mantenimento spesso oneroso e poteva richiedere anche diverso tempo. Per questo motivo e per il declino che negli anni successivi subirono molti dei sistemi operativi mobile, col tempo i framework iniziarono a supportare sempre meno piattaforme, fino a quando non ne rimasero solo due: Android ed iOS.

1.5 L'affermazione del duopolio

Con la progressiva scomparsa del già poco diffuso BlackBerry OS ed il rapido declino subito da Symbian, infatti, l'unico sistema operativo rimasto a tentare di competere con il duopolio di Google ed Apple dopo il 2012 è stato quello di Microsoft denominato Windows Phone 8 prima e Windows 10 Mobile poi. Sfortunatamente, nonostante fosse apprezzato da buona parte della sua utenza, rimase sempre una piattaforma di nicchia, senza mai riuscire a raggiungere una grande diffusione. Anche per questo motivo risultava molto poco appetibile agli sviluppatori, che spesso distribuivano le loro applicazioni soltanto su Android ed iOS, e molti dei framework multipiatatforma rilasciati nella seconda metà degli anni '10 del 2000 scelsero di non fornire il loro supporto al sistema operativo. Si generò, così, una grande mancanza di applicazioni rispetto alle altre due piattaforme mobile e ciò, insieme alla sua già scarsa popolarità e ad alcune scelte delle aziende concorrenti che lo isolarono ulteriormente[48], condannò definitivamente il sistema operativo di Microsoft. Dopo numerosi tentativi di risollevare le sue sorti, infatti, nel 2018 Microsoft annunciò la fine del supporto a Windows 10 Mobile[49], legittimando definitivamente la sola presenza di Google ed Apple nel mercato dei sistemi operativi mobile.

Capitolo 2

Android ed iOS

Come discusso nel capitolo precedente, ormai il mercato dei sistemi operativi mobile è dominato da due soli attori: Android ed iOS. I due sistemi, nati con obiettivi differenti, seguono tutt'ora dei paradigmi distinti. Il primo resta una piattaforma aperta, con lo scopo di dare la possibilità agli utenti di personalizzarlo, di installare app da fonti diverse, finanche di modificarlo e riadattarlo per crearne la propria versione. Il sistema operativo di Apple, diversamente, permette meno libertà ai propri utenti in cambio di una migliore ottimizzazione, un maggiore controllo del software utilizzato ed una migliore esperienza d'uso, specialmente all'interno dell'ecosistema di dispositivi Apple.

2.1 Android

Android è il sistema operativo mobile open source di Google basato su kernel Linux e scritto prevalentemente in C, C++ e Java. Tramite l'Android Open Source Project (AOSP) Google rende disponibile il codice sorgente del sistema operativo così che chiunque possa riadattarlo e compilarlo per conto proprio. Potenzialmente, infatti, fornendo tutti i driver necessari è possibile utilizzare Android su qualsiasi dispositivo, smartphone, smartwatch, console di gioco, televisori e perfino frigoriferi. Principalmente, però, il sistema operativo è utilizzato per smartphone e la sua natura open source permette sia ai produttori di inserire facilmente i propri driver per rendere compatibili le componenti hardware dei loro dispositivi, che agli utenti di creare distribuzioni di Android personalizzate come LineageOS[50].

2.1.1 Architettura di Android

L'architettura del sistema operativo è suddivisibile in sei componenti principali, mostrati nella seguente immagine.

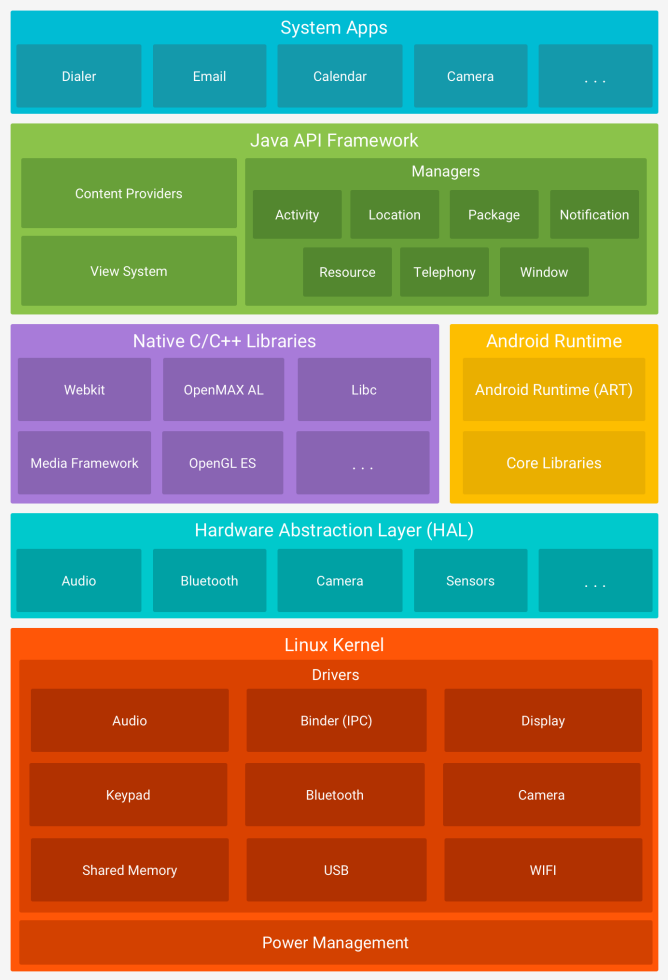

Figura 2.1: Architettura di Android

Kernel

L'architettura di Android si basa su kernel Linux che fornisce i driver alle varie componenti dello smartphone, quali lo schermo, la camera, le antenne Bluetooth e così via, ed un livello di astrazione tra l'hardware del dispositivo ed il resto dell'architettura.

Hardware Abstraction Layer

Il livello di astrazione dell'hardware (HAL) si occupa di fornire delle interfacce standard che espongono al framework delle API Java, che si trova ad un livello superiore, le funzionalità hardware del dispositivo. Contiene diversi moduli, ognuno dei quali implementa un'interfaccia per uno specifico componente hardware. Tali moduli consentono di implementare funzionalità ad un livello più basso senza che ciò influenzi o modifichi i livelli superiori.

Android Runtime

L'ambiente Android Runtime (ART) è una parte fondamentale del sistema operativo in quanto costituisce la base per il livello di Application framework fornendo sistemi di compilazione Ahead Of Time (AOT) e Just In Time (JIT), discussi più nel dettaglio al paragrafo 2.1.2, il garbage collector, sistemi di debugging e delle librerie di runtime Java utilizzate, per l'appunto, dal framework di API Java.

Librerie native

Le librerie native, scritte in C e C++, sono necessarie per il funzionamento di numerose parti e servizi del sistema, come i già citati HAL e ART. Le funzionalità di alcune di queste librerie, come OpenGL per disegnare e manipolare grafica 2D o 3D, SQLite per la gestione di database o librerie per la registrazione e riproduzione dei media, sono esposte alle applicazioni mediante delle API del framework Java discusso di seguito. Se nello sviluppo di un'applicazione si rendesse necessario l'accesso diretto a tali librerie, Android mette a disposizione il Native Development Kit (NDK) che permette di richiamarle all'interno del codice in Java o Kotlin utilizzando la Java Native Interface (JNI).

Java API Framework

Il framework delle API Java contiene le API scritte, per l'appunto, in Java che rendono disponibili tutte le funzionalità e servizi del sistema operativo che le applicazioni possono utilizzare. Queste comprendono:

- Il sistema di viste usate per comporre l'interfaccia utente, come bottoni, liste, caselle di testo, checkbox ed altre

- Il gestore delle risorse, che fornisce l'accesso ad una serie di utili risorse quali stringhe di traduzione, componenti grafiche e file di layout

- Il gestore delle notifiche, che consente all'applicazione di mostrare, appunto, delle notifiche nella barra di stato del sistema

- Il gestore delle attività, che gestisce il ciclo di vita delle applicazioni fornendo uno stack di navigazione comune

- I Content Provider, che permettono all'applicazione di accedere a dati di altre app e di condividere con esse i propri

App di sistema

In ultimo vi sono le applicazioni di sistema, ovvero quelle applicazioni presenti di default sul sistema operativo che permettono di utilizzare le funzionalità base del dispositivo come le email, gli SMS, il calendario, il browser di internet o i contatti. Molte di queste applicazioni possono essere sostituite dall'utente con applicazioni di terze parti, con qualche eccezione, come l'app delle impostazioni di sistema. Il sistema operativo permette, inoltre, agli sviluppatori di invocare le applicazioni di sistema fornendo per ogni funzionalità direttamente l'applicazione scelta dall'utente.

Nello sviluppo di un'applicazione per Android si fa uso delle varie API e librerie messe a disposizione dal sistema per utilizzare i componenti hardware del dispositivo e comporre un'interfaccia con la quale mostrare dati all'utente. Quando, poi, l'applicazione viene compilata viene generato un APK, ovvero Android Package, al cui interno sono poste le risorse, gli asset, i certificati, i file manifest ed il *bytecode* dell'app, ovvero il codice sorgente contenente le operazioni del programma. Perché l'applicazione venga eseguita, quest'ultimo deve essere convertito in linguaggio macchina, operazione di cui si occupa, nello specifico, l'Android Runtime.

2.1.2 Android Runtime

In origine, poiché molti dei dispositivi mobile avevano molta poca memoria, per compilare il codice dell'app veniva utilizzata la *Dalvik Virtual Machine* (DVM), basata sulla strategia *Just In Time* (JIT), che compilava il codice durante l'esecuzione, riducendo, appunto, l'utilizzo della RAM. Sfortunatamente questo metodo impattava molto sulle prestazioni, rallentando, in alcuni casi anche notevolmente, l'esecuzione.

A partire dalla versione 4.4 del sistema operativo, Android KitKat, però, è stato introdotto un nuovo sistema: *Android Runtime* (ART). Differentemente da DVM, ART non utilizzava la strategia di compilazione JIT, ma la strategia *Ahead Of Time* (AOT), compilando il codice prima dell'esecuzione dell'applicazione, migliorando notevolmente le prestazioni e velocizzando l'esecuzione.

Lo svantaggio principale di questo sistema era il maggiore consumo di RAM, bilanciato da una tendenza ad inserire memorie sempre più capienti negli smartphone, e un maggiore tempo necessario all'installazione, dato che la compilazione dell'APK, infatti, avveniva durante l'installazione dell'applicazione.

Per questo motivo dalla versione 7.0 del sistema, Android Nougat, nell'Android Runtime è stato incluso un compilatore Just In Time per adottare una soluzione ibrida definita *Profile Guided Optimization* (PGO). La compilazione, infatti, sarebbe nuovamente avvenuta all'avvio dell'applicazione, ma l'ART avrebbe rilevato quali parti del codice venissero eseguite più frequentemente, le avrebbe precompilate e salvate, generando un *profilo*, così che all'avvio successivo non sarebbe stato necessario compilarle nuovamente. La precompilazione non avveniva durante l'esecuzione, ma quando il dispositivo era in idle ed in carica, così da non impattare sulle prestazioni della batteria. Tale strategia ridusse nuovamente i tempi di installazione, ma prevedeva, ovviamente, che l'utente utilizzasse l'app perché l'ART rilevasse le porzioni di codice usate più spesso, perciò era possibile che i primi utilizzi dell'applicazione risultassero più lenti in quanto in quel momento il codice veniva compilato interamente durante l'esecuzione.

Per ovviare a questo problema nella versione 9.0 di Android, Pie, Google aggiunse la possibilità di scaricare un profilo insieme all'APK dell'applicazione così che sin dal primo avvio questa potesse essere eseguita con delle ottime prestazioni. Per creare questo profilo venivano usati i dati degli utenti basandosi sull'idea che la maggior parte delle persone utilizza un'applicazione allo stesso modo. Il processo, perciò, prevede che dopo la creazione da parte dell'ART di un profilo sul dispositivo di un utente, questo venga caricato sul Play Cloud, sul quale verrà, successivamente, creato un profilo aggregato a partire da quelli caricati. In questo modo già dalla prima installazione sul dispositivo l'applicazione possiede un profilo che verrà aggiornato in base all'uso dell'utente dopo qualche utilizzo e caricato a sua volta sul Play Cloud.

Anche questo sistema, però, ha i suoi difetti. Collezionare ed aggregare i profili sul Play Cloud può richiedere anche qualche giorno e ciò potrebbe rappresentare un problema per applicazioni che ricevono aggiornamenti ad intervalli molto brevi in quanto i profili aggregati potrebbero non aggiornarsi in tempo. C'è da considerare, inoltre, che alcune applicazioni potrebbero non avere un numero di utenti sufficiente a creare un profilo efficace. Recentemente, quindi, sono stati introdotti i *Baseline Profiles*, ovvero dei profili creati direttamente in fase di sviluppo dell'applicazione selezionando classi e metodi da includere nel profilo. In tal modo gli sviluppatori possono rendere disponibile fin da subito sul Play Store con l'applicazione un profilo che potrà, in seguito, essere integrato con i profili aggregati degli utenti che la utilizzano.

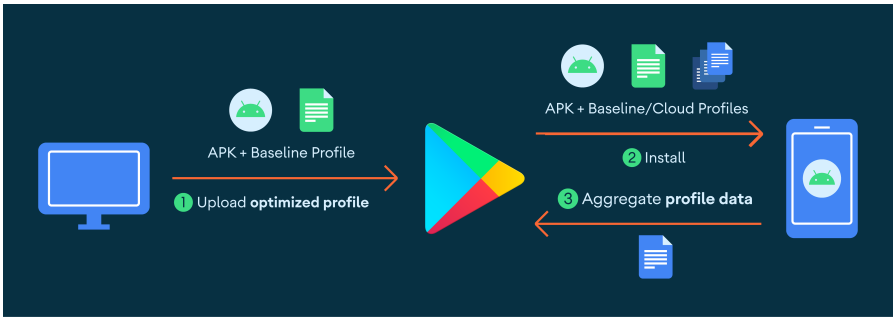

Figura 2.2: Workflow del Baseline Profile

2.1.3 Componenti di un'applicazione

Come già specificato al paragrafo 2.1.1, Android fornisce numerose librerie, alcune delle quali sono utilizzate per fornire agli sviluppatori gli elementi necessari alla costruzione di un'applicazione: i componenti. I componenti sono i blocchi fondamentali di un'applicazione ed ognuno di essi costituisce un punto d'ingresso attraverso il quale il sistema può accedere all'app. I componenti possono essere di quattro tipi, *Activity*, *Service*, *Broadcast receiver* e *Content provider*, ed ognuno di questi è destinato ad uno specifico tipo di interazione all'interno del sistema operativo.

Activity

Una activity è il fulcro dell'interazione con l'utente, rappresenta una singola schermata dell'applicazione ed è preposta a consentire lo svolgimento di uno specifico compito al suo interno. Un'applicazione è tendenzialmente costituita da una o più attività, ognuna indipendente dalle altre. Ciò consente al sistema o ad altre app di aprire direttamente una qualunque activity dell'applicazione se questa lo permette, così da fornire l'accesso diretto ad una sua specifica funzionalità.

Service

Un servizio è un componente che consente di mantenere l'applicazione in esecuzione in background. Non è dotato di interfaccia grafica e viene solitamente eseguito per svolgere operazioni di lunga durata come riprodurre della musica mentre è in utilizzo un'altra applicazione o ricevere dati dalla rete mentre l'utente sta visualizzando ed interagendo con un'activity. I servizi sono componenti molto utilizzati, per via della loro duttilità per la creazione di funzionalità come screen saver, live wallpapaer e servizi di accessibilità. I servizi possono essere di due tipi:

- *Started service*, comunica al sistema di tenerlo in esecuzione finché il lavoro del servizio non è terminato

- *Bound service*, viene eseguito perché il sistema, o un'altra app, ha bisogno di utilizzarlo, potrebbe essere considerato il servizio che fornisce una API ad un altro processo

Broadcast receiver

Un broadcast receiver è un componente che attende messaggi provenienti dal sistema, come un avviso di spegnimento dello schermo o di batteria in esaurimento, o da altre app, come la comunicazione della fine della trasmissione di un file. Tali messaggi possono essere consegnati all'applicazione anche quando questa non è in esecuzione. Nonostante i broadcast receiver non abbiano un'interfaccia grafica, è possibile creare una notifica nella barra di stato per informare l'utente della loro esecuzione.

Content provider

Un content provider gestisce un insieme di dati applicativi che possono essere memorizzati nel file system, in un database, sul web o su qualunque altro sistema di archiviazione persistente accessibile all'applicazione. Per consentire ad altre app di ricevere e salvare dati gestiti dall'applicazione, il content provider implementa un insieme di metodi che non vengono richiamati direttamente, ma tramite un content resolver. Per permettere di specificare il tipo di dati da ricevere od inviare, l'applicazione fornisce, poi, degli schemi URI alle altre entità così che queste possano utilizzarli per accedere ai dati.

Tutte le funzionalità offerte da un'applicazione sono contenute nel suo *manifest file*, un documento in cui sono definiti i singoli componenti che la costituiscono, i permessi necessari ad accedere a parti protette del sistema o di altre app ed eventuali informazioni di configurazione. Le informazioni contenute nel manifest consentono al sistema operativo di sapere quali componenti è necessario istanziare quando l'applicazione deve essere eseguita perché avviata dal sistema o da un evento esterno.

2.1.4 Kotlin

La maggior parte delle API di Android sono scritte in Java[51] e proprio questo è stato per lungo tempo il linguaggio di programmazione ufficiale del sistema operativo, ma dal 2019 è stato sostituito da *Kotlin*[52], un sistema operativo open source

più giovane e moderno sviluppato da JetBrains che ha in breve tempo acquisito una notevole popolarità.

Kotlin è un linguaggio di programmazione a oggetti staticamente tipato ispirato principalmente da linguaggi come Scala e lo stesso Java. È nato con l'obiettivo di essere compatibile con l'ambiente Java ed il suo già grande ecosistema, ma con un linguaggio più semplice e rapido da utilizzare. Kotlin si basa, infatti, sulla *Java Virtual Machine* (JVM), ma può essere compilato anche in JavaScript e linguaggio macchina nativo utilizzando il compilatore Kotlin/Native. Sono diverse le funzionalità per cui è, ormai, preferito a Java, tra le quali figurano la pulizia della sua sintassi, che permette di avere un codice molto meno verboso, la sicurezza delle chiamate di variabili potenzialmente null, la totale interoperabilità con Java e le funzioni di gestione della concorrenza.

Sintassi pulita

Sono diverse le caratteristiche che rendono la sintassi di Kotlin tanto pulita permettendo, così, di ottenere un codice chiaro e compatto. Qui di seguito sono riportate alcune tra le caratteristiche principali.

- È possibile omettere diversi elementi come i punti e virgola, il return type quando questo è Unit, la keyword *new* quando viene creato un nuovo oggetto o il tipo di una variabile quando questa viene inizializzata.
- Sono, poi, disponibili le parole chiave *var* e *val* per inizializzare facilmente, rispettivamente, variabili riscrivibili e variabili final.
- Le stringhe possono contenere delle *template expression*, ovvero delle espressioni precedute da un simbolo del dollaro il cui risultato è inserito nella stringa nella posizione alla quale si trova l'espressione.
- È possibile utilizzare, in sostituzione dei poco chiari operatori ternari, una espressione if che, pur essendo di poco più lunga, rende notevolmente più leggibile il codice.

```
fun printName(outerString: String) {
    val name = "Mario"
    val surname = (if outerString.isNotEmpty()) outerString else "Unknown"
    println("Name: $name, Surname: $surname")
}
```

Listato 2.1: Esempio di funzione per mostrare la sintassi di Kotlin

Null safety

In alcuni linguaggi quando una variabile viene dichiarata senza fornire un valore iniziale esplicito a questa viene assegnato il valore null che, se non gestito bene, può portare facilmente a delle eccezioni. Uno dei principali obiettivi di progettazione di Kotlin era quello di ridurre drasticamente il rischio di incappare in problemi simili. Per questo motivo in Kotlin una variabile può contenere il valore null soltanto se questa viene dichiarata di un tipo *nullable*, ovvero aggiungendo come suffisso al tipo un punto interrogativo.

Ciò consente di poter accedere a proprietà e funzioni degli oggetti di tipo non-nullable senza dover verificare che questi non siano null. È possibile, poi, verificare se una variabile nullable sia null o meno esplicitamente con una condizione all'interno di un if o con l'operatore di chiamata sicura fornito da Kotlin. Per utilizzare tale operatore è sufficiente inserire un punto interrogativo prima del punto con il quale viene richiamata la proprietà o la funzione dell'oggetto.

È possibile, infine, aggirare questo sistema utilizzando l'operatore di asserzione di non nullità, utilizzabile inserendo due punti esclamativi come suffisso della variabile, che la converte in un oggetto di tipo non-nullable e ritorna un errore se questo è null.

```
var string: String = "hello"
string = null           // Genera un errore di compilazione

var nullableString: String? = "hello"
nullableString = null   // Viene eseguito senza generare errori

var length: Int? = (if nullableString != null) nullableString.length else -1
↪  // length vale -1

length = nullableString?.length     // length è null
length = nullableString!!.length    // Genera un'eccezione di puntatore nullo
```

Listato 2.2: Esempio di variabile nullable

Interoperabilità con Java

Essendo costruito proprio con lo scopo di essere compatibile con Java, Kotlin consente di utilizzare facilmente codice scritto in Java in maniera naturale e viceversa. Gli elementi che differiscono nei due linguaggi sono efficacemente convertiti, come i getter e setter degli oggetti presenti in Java che vengono convertiti in proprietà

in Kotlin o i metodi che in Java ritornano *void* che in Kotlin ritornano automaticamente Unit. Una delle differenze principali è la appena descritta gestione delle variabili null. Poiché ogni oggetto in Java può essere null, i tipi dichiarati in Java vengono definiti *tipi di piattaforma* e su di essi non vengono effettuati i controlli sulla loro possibile nullability, ottenendo così, di fatto, le stesse garanzie che si avrebbero in Java. Quando viene richiamato un metodo di una variabile di un tipo di piattaforma non viene generato alcun errore durante la compilazione, ma potrebbe comunque presentarsi una eccezione di puntatore nullo durante l'esecuzione dell'app.

Concorrenza strutturata

Il principio della concorrenza strutturata prevede che nell'esecuzione simultanea di più compiti un'operazione sia completa solo quando anche le operazioni figlie siano state completate e che, si conseguenza, dopo il termine dell'operazione nessuna operazione figlia sia ancora in esecuzione. Seguendo questo principio, Kotlin fornisce le *coroutine*, delle istanze di esecuzioni sospendibili, concettualmente simili ai thread, ma slegate da questi. Le coroutine possono essere eseguite solo in uno specifico *CoroutineScope* che ne delimita il ciclo di vita, garantendo il rispetto del principio della concorrenza strutturata in quanto ogni scope non può essere completato finché tutte le coroutine al suo interno non siano completate. È possibile contrassegnare, poi, delle funzioni come *suspend* per indicare che queste hanno un comportamento asincrono e possono essere richiamate esclusivamente all'interno di coroutine o di altre funzioni suspend. Le coroutine offrono, inoltre, una serie di funzionalità molto utili in fase di sviluppo come il contesto, ovvero un insieme di elementi che la definisce, in particolare il Job ed il Dispatcher. Il Job è un oggetto che gestisce l'esecuzione della coroutine e può essere utilizzato per attendere esplicitamente il suo completamento o cancellarla. Il Dispatcher determina quali thread utilizzare per l'esecuzione della coroutine, può confinarla su un solo thread o lasciarla eseguire su qualunque thread. È possibile creare dei thread personalizzati o utilizzare quelli forniti da Kotlin:

- *Default*, utilizzato se non viene specificato nessun Dispatcher nel contesto della coroutine, usa un gruppo comune di thread di background condivisi ed è la scelta migliore per esecuzioni che consumano molte risorse della CPU
- *IO*, usa un gruppo condiviso di thread creati su richiesta ed è progettato per gestire al meglio operazioni bloccanti di Input/Output come il recupero di dati da un database locale o da un server remoto
- *Unconfined*, esegue la coroutine nell'attuale stack di chiamate fino alla prima sospensione, viene poi ripresa successivamente nel thread utilizzato dalla funzione che l'ha richiamata

2.2 iOS

iOS è il sistema operativo sviluppato da Apple unicamente per i suoi dispositivi mobile. Attualmente è utilizzato solo sugli iPhone, ma inizialmente era dedicato anche agli iPod Touch, ormai non più prodotti, ed agli iPad, per i quali dal 2019 è utilizzata una sua variante, iPadOS. iOS è anche la base di altri sistemi operativi dedicati a specifiche famiglie di dispositivi Apple come il già citato iPadOS per gli iPad, tvOS per le Apple TV e watchOS per gli Apple Watch. Il sistema operativo è proprietario, ma alcune sue parti sono open source pubblicate con Apple Public Source License, come il kernel XNU, o con altre licenze, come WebKit o il linguaggio di programmazione Swift.

2.2.1 Architettura di iOS

L'architettura del sistema operativo mobile di Apple è composta di quattro elementi principali, mostrati nella seguente immagine.

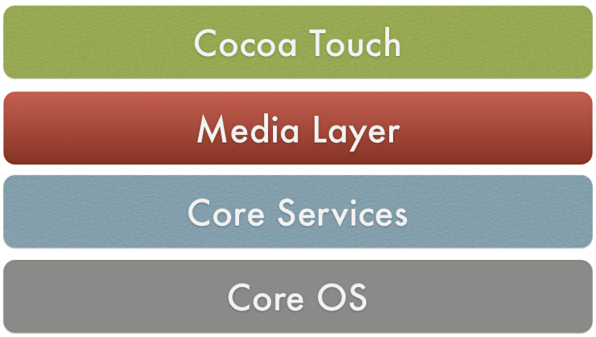

Figura 2.3: Architettura di iOS

Core OS

L'architettura di iOS si basa su Darwin OS, sul quale si basano anche tutti gli altri sistemi operativi sviluppati da Apple. Darwin è a sua volta costruito su XNU, un kernel ibrido scritto in C, C++ ed Assembly contenente caratteristiche del kernel monolitico BSD e del microkernel Mach. Il livello di Core OS si occupa di gestire le attività del file system e del sistema operativo, l'utilizzo della memoria e la comunicazione, oltre a fornire i driver per il funzionamento delle componenti hardware del dispositivo. Contiene i framework che forniscono servizi di basso livello relativi a componenti hardware di basso livello, astraendo ai livelli superiori i servizi del livello sottostante, tra cui:

- *Accelerate*, consente di effettuare calcoli matematici complessi ed elaborazione digitale delle immagini

- *Core Bluetooth*, gestisce, come suggerisce il nome, la comunicazione con tecnologia Bluetooth

- *Local Authentication*, fornisce funzioni di autenticazione, in particolare gestendo in maniera sicura i dati biometrici legati all'autenticazione mediante TouchID e FaceID

- *External Accessory*, consente di comunicare con accessori che vengono collegati al dispositivo tramite il connettore Lightning o tecnologia Bluetooth

Core Services

Il livello Core Services si occupa di implementare i servizi di base del dispositivo utilizzati dal sistema e dalle applicazioni come networking, preferenze di sistema e gestione dei dati. Nello specifico, tra le librerie più utilizzate vi sono:

- *Address Book*, fornisce l'accesso alle informazioni dei contatti ed al database degli utenti

- *Core Foundation*, fornisce le funzioni di recupero e gestione dei dati

- *Core Motion*, dà accesso alle informazioni relative a sensori del dispositivo come accelerometro, giroscopio e pedometro

- *Social*, fornisce una semplice interfaccia per accedere agli account social dell'utente

Media

Il livello Media contiene le librerie che consentono la riproduzione di contenuti multimediali e la gestione della grafica sul dispositivo. Le librerie di questo livello, infatti, sono utilizzate per il rendering di immagini 2D, 3D e vettoriali, la visualizzazione di animazioni ed il controllo di riproduzione e registrazione di video e audio.

Cocoa Touch

Cocoa Touch l'ambiente di sviluppo delle applicazioni per iOS, watchOS e tvOS che contiene i framework utilizzati per gestire la logica di più alto livello e l'interfaccia utente. Contiene, tra le altre, le librerie che permettono di incorporare le mappe nella propria applicazione, implementare un'interfaccia per interagire con rubrica,

calendario e messaggi, gestire le notifiche ed aggiungere funzionalità di social-gaming. Nello specifico, i due framework più utilizzati sono:

- *Foundation*, fornisce le funzionalità di base per applicazioni e framework come memorizzazione e persistenza dei dati, elaborazione del testo, ordinamento e filtraggio dei dati e funzioni di networking contenendo le definizioni di classi, protocolli e tipi utilizzati nell'intero SDK.

- *UIKit*, fornisce l'architettura di finestre e view che permette di implementare l'interfaccia utente, l'infrastruttura che gestisce gli eventi di input ed i cicli di eventi che si occupano dell'interazione tra l'utente, il sistema e l'applicazione

2.2.2 Swift

Buona parte del sistema operativo è scritto in Objective-C, un linguaggio di programmazione ad oggetti creato negli anni '80 come estensione del C, con il quale mantiene la completa compatibilità, influenzato, in particolare, da Smalltalk. Molti dei sistemi operativi sviluppati da Apple a partire da Mac OS X sono stati scritti principalmente in Objective-C ed iOS non fa eccezione. Per questo motivo inizialmente Objective-C è stato anche il linguaggio con cui sviluppare le applicazioni per iOS, ma nel 2014 Apple ha rilasciato un nuovo linguaggio con l'obiettivo di sostituirlo, Swift. Swift è un linguaggio di programmazione ad oggetti open source progettato per essere veloce, moderno, sicuro ed interattivo. Sin dagli inizi è stato supportato dalla comunità di sviluppatori e molto apprezzato per le funzionalità che offriva, evolutesi nel tempo e con gli aggiornamenti del linguaggio di versione in versione. Essendo stato progettato con obiettivi simili, Swift condivide alcune caratteristiche con la sua controparte per piattaforma Android, Kotlin, come l'utilizzo di tipi opzionali per semplificare la gestione di variabili null, la possibilità di omettere punti e virgola o di usare template expression all'interno delle stringhe che garantiscono la pulizia della sintassi. Swift, offre, però anche delle funzionalità specifiche, alcune delle quali molto apprezzate dagli sviluppatori, e presenta qualche differenza sostanziale rispetto a Kotlin.

Strutture

In Swift, differentemente da molti altri linguaggi di programmazione ad oggetti, è possibile creare sia oggetti che sfruttino il passaggio per riferimento che oggetti che utilizzino il passaggio per valore. Nel primo caso il tipo dell'oggetto è definito come *reference type* ed il suo passaggio verso una variabile implica semplicemente il passaggio del puntatore al suo valore in memoria. Ciò comporta che la seconda variabile non contenga una copia dell'oggetto, ma soltanto il riferimento al suo valore. Nel secondo caso l'oggetto è definito di tipo *value type* ed il suo passaggio

ad una seconda variabile comporta la creazione di un oggetto identico, ma che non sarà influenzato da future modifiche del primo oggetto. Per la creazione di oggetti value type, Swift fornisce il tipo nominale *struct*. Le strutture sono assimilabili alle classi, con le quali condividono numerose funzionalità come il meccanismo delle estensioni o la possibilità di contenere metodi e implementare protocolli, che in altri linguaggi sono tendenzialmente definiti interfacce, ma mentre le classi generano oggetti reference type, le strutture generano, appunto, oggetti value type. Per rendere più efficiente la gestione delle strutture, specialmente di quelle molto grandi, Swift utilizza la pratica copy-on-write che prevede che un oggetto non venga copiato finché non si tenta di cambiare il valore contenuto al suo interno. In tal modo se un oggetto viene passato ad una seconda variabile, continuerà ad esistere una sola istanza in memoria alla quale potranno accedere entrambe finché non verrà effettuata un'operazione di modifica.

Concorrenza Strutturata

Dalla sua versione 5.5, rilasciata nel settembre 2019, anche Swift ha introdotto la possibilità di scrivere codice asincrono in maniera strutturata grazie alla sintassi *async/await*. Tale sintassi prevede che l'identificazione di funzioni asincrone avvenga usando la parola chiave *async* e che per il loro utilizzo sia necessaria la parola chiave *await*, similmente a come avviene in altri lunguaggi come C# o JavaScript. Contrassegnare una funzione come async esplicita la possibilità che questa possa essere sospesa prima di concludere il suo compito e richiamandola con la parola await si riconosce che questa sia asincrona e possa essere sospesa. Una funzione async, però, non può essere richiamata da qualunque punto del codice, ma soltanto da un contesto asincrono, ovvero all'interno del metodo statico *main()* di una struttura o classe con l'annotazione *@main*, di un'altra funzione, metodo, proprietà asincrona o di un Task. I *Task* sono oggetti che permettono di eseguire delle operazioni concorrenti individualmente fornendo, poi, la possibilità di sospenderle, cancellarle e verificare il loro stato. Nel momento della sua creazione è, inoltre, possibile assegnare al task una priorità rispetto ad altre operazioni asincrone che potrebbero essere in esecuzione nello stesso momento. Per operazioni più complesse Swift fornisce anche la struttura *TaskGroup*, che consente, come suggerisce il nome, di raggruppare più task simili, ovvero che ritornino tutti lo stesso tipo di oggetti, per gestirli insieme. Per utilizzarla, però, non è sufficiente inizializzare un oggetto di tipo TaskGroup, ma è necessario richiamare la funzione asincrona *withTaskGroup* alla quale fornire il tipo degli oggetti che i task interni restituiranno.

Automatic Reference Counting

In Swift per la gestione della memoria è utilizzato il sistema *Automatic Reference Counting* (ARC), un sistema che si occupa di allocare e deallocare porzioni di memoria automaticamente senza che se ne debba occupare lo sviluppatore in fase di scrittura del codice. Ogni volta che viene creata una nuova istanza di una classe, ARC alloca una porzione della memoria dedicata a questa in cui verrà conservata l'informazione relativa al suo tipo e il valore di ogni proprietà ad essa associata. Durante la compilazione, ARC verifica quante proprietà, costanti e variabili fanno riferimento ad ogni istanza, così da poter verificare sempre quanti riferimenti attivi ad un'istanza esistono. Se non esiste alcun riferimento ad un'istanza, questa è considerata irraggiungibile e ARC dealloca la porzione di memoria utilizzata. È possibile, tuttavia, che un'istanza abbia sempre almeno un riferimento, come nel caso dello *Strong Reference Cycle* che indica una situazione in cui due istanze facciano riferimento l'un l'altra. Si verifica, ad esempio, una situazione di questo tipo quando si hanno due classi che contengono ognuna un parametro del tipo dell'altra. In questo caso, anche se si prova ad eliminare o sostituire un oggetto, non sarà mai possibile deallocarlo e questo causerà memory leak. Per risolvere questo problema Swift fornisce il property wrapper *weak* che permette al sistema ARC di deallocare la memoria anche se è ancora presente un riferimento perché questo è considerato debole.

Listato 2.3: Strong Reference Cycle e weak reference

```
struct Person {
    let name: String
    var apartment: Apartment?
}

struct Apartment {
    let unit: String
    weak var tenant: Person?
}

var mario: Person?
var unit3B: Apartment?

mario = Person(name: "Mario Rossi")
unit3B = Apartment(unit: "3B")
```

```
mario!.apartment = unit3B
unit3B!.tenant = mario
// Viene generato un riferimento tra le due istanze

unit3B = nil
// L'istanza dell'oggetto Apartment non è deallocata perché la proprietà in
↪  Person non è weak, perciò genera un riferimento forte

mario = nil
// Poiché la proprietà tenant della classe Apartment è weak, il riferimento
↪  che generava era debole, perciò ora l'oggetto Apartment può essere
↪  deallocato, eliminando così anche il riferimento forte generato dalla
↪  proprietà apartment della classe Person e permettendo di deallocare anche
↪  l'istanza dell'oggetto Person
```

Il vantaggio di tale sistema rispetto al Garbage Collector utilizzato da molti altri linguaggi, tra cui Kotlin, è che ARC non ha bisogno di essere eseguito in background, cosa che potrebbe rallentare l'esecuzione dell'applicazione e richiedere più memoria.

2.2.3 Compilatore di Swift

Per la compilazione del codice in linguaggio macchina Swift sfrutta l'infrastruttura di compilazione LLVM, un progetto nato nell'Università dell'Illinois nel 2000 per studiare le tecniche di compilazione dinamica per linguaggi dinamici e statici evolutosi, poi, in una raccolta di compilatori modulari e riutilizzabili e tecnologie di toolchain. Attualmente LLVM consente di compilare numerosi linguaggi, tra cui Swift, Objective-C, Rust ed altri, per diverse piattaforme come x86-64, ARM o MIPS.

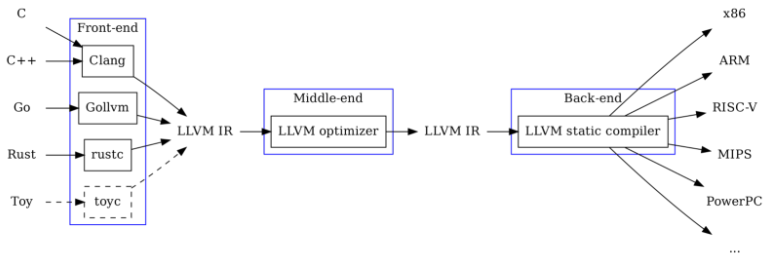

Figura 2.4: Schema di funzionamento dell'infrastruttura LLVM

La struttura su cui si basa fornisce un front-end per ogni linguaggio supportato che converte il codice in un linguaggio di basso livello simile ad Assembly, definito *Intermediate Representation* (IR), indipendente dalla macchina. Il codice intermedio viene poi a sua volta compilato in linguaggio macchina mediante un back-end che fornisce i set di istruzioni della specifica architettura sulla quale l'applicazione è in esecuzione. Il compilatore di Swift si occupa di importare l'infrastruttura di front-end e tooling del progetto LLVM dedicata al linguaggio ed è costituito di sette componenti:

- *Parsing*, un parser di discesa ricorsivo, ovvero un parser top-down costituito da procedure mutualmente ricorsive, si occupa di generare un *Abstract Syntax Tree* (AST) senza alcuna semantica e verificare la presenza di problemi nella grammatica del codice sorgente

- *Semantic Analysis*, si occupa di verificare che nell'AST generato dal parser non siano presenti problemi semantici come l'inferenza di tipo e, in caso di successo, passa l'AST, ora type-checked, al componente successivo

- *Clang Importer*, importa i moduli di Clang, l'infrastruttura di front-end e tooling del progetto LLVM, e mappa le API in C o Objective-C che tali moduli esportano nelle corrispondenti API in Swift

- *SIL Generator*, converte l'AST type-checked in codice in *Swift Intermediate Language* (SIL) "grezzo", un linguaggio intermedio di alto livello specifico di Swift utilizzato per analizzare ed ottimizzare ulteriormente il codice Swift

- *SIL Guaranteed Transformations*, esegue un ulteriore controllo che verifica la correttezza di un programma generando, al termine del processo, del codice in SIL "canonico"

- *SIL Optimizations*, esegue delle ottimizzazioni aggiuntive di alto livello al programma come, ad esempio, devirtualizzazione o Automatic Reference Counting, l'ottimizzazione dell'allocazione di memoria in relazione al reference count di un oggetto discussa in precedenza

- *LLVM IR Generation*, converte il SIL in LLVM Intermediate Representation affinché il middle-end di LLVM possa effettuare ulteriori ottimizzazioni e generare il codice in linguaggio macchina

2.2.4 Stabilità della ABI di Swift

I vantaggi di Swift rispetto al linguaggio su cui si basa e che dovrebbe sostituire, Objective-C, sono molti, principalmente legati alla facilità di utilizzo, semplicità di lettura del codice e sicurezza. Nei primi anni di vita, però, molte critiche

furono mosse al linguaggio, specialmente in relazione all'intenzione di renderlo il sostituto di Objective-C, per via dell'instabilità della sua ABI. La ABI, ovvero *Application Binary Interface*, è l'interfaccia alla quale il compiler deve aderire e definisce strutture e metodi che l'applicazione compilata utilizza per accedere a librerie di più basso livello. È fondamentalmente il livello di astrazione intermedio tra il codice macchina ed il codice compilato. In alcuni casi, nuove versioni di un linguaggio portano anche modifiche alla ABI, rendendole incompatibili con le versioni precedenti. La stabilità dell'ABI, quindi, garantisce un supporto nel tempo di versioni successive di un linguaggio. Objective-C, costruito sul linguaggio C, si basa quasi interamente sulla ABI di questo, ad esempio per le chiamate di funzione o per il memory layout delle strutture. Alcune implementazioni, però, sono specifiche del linguaggio e, per questo, inserite in framework forniti da Apple come il framework *Objective-C Runtime*. Tutti i framework implementano l'ABI del linguaggio Objective-C che è fornita dal sistema operativo ed è, ormai, stabile. Swift ha accesso all'ambiente di runtime del linguaggio su cui si basa, Objective-C, ciò garantisce un'alta interoperabilità tra i due consentendo di utilizzare codice in entrambi i linguaggi nello stesso progetto. Ciononostante, i due linguaggi non condividono l'ABI e, per via della sua continua evoluzione, l'ABI di Swift è cambiata spesso nei suoi primi anni di vita. Il problema principale era quindi che, non essendo stabile, l'ABI di Swift non poteva essere implementata all'interno del sistema operativo in quanto, per poter compilare applicazioni scritte in versioni di Swift differenti, sarebbe stato necessario implementare librerie di supporto per ogni versione. Per questo motivo le librerie di supporto contenenti l'implementazione dell'ABI dovevano essere distribuite insieme all'applicazione, aumentandone in questo modo le dimensioni. Nel 2019, però, con il rilascio di Swift 5, l'ABI venne dichiarata stabile e le librerie dinamiche che la implementavano vennero finalmente integrate in iOS a partire dalla versione 12.2, rendendo non più necessario includerle nelle applicazioni[53]. I vantaggi di tale scelta furono diversi, il più evidente fu la riduzione della dimensione delle app quando installate su una versione del sistema operativo uguale o superiore alla 12.2. L'inclusione delle librerie nel sistema operativo, poi, permetteva di condividerle tra tutte le applicazioni memorizzandole nella cache del dispositivo, permettendo un avvio delle stesse più rapido. Inoltre, veniva così assicurata la compatibilità tra applicazioni e librerie compilate con versioni del compilatore differenti, risparmiando continui controlli ad ogni aggiornamento del linguaggio.

Capitolo 3

Sviluppo Cross-Platform

Sviluppare in nativo permette certamente di creare applicazioni potenti, complete e ben ottimizzate, ma se si prevede di distribuirle su più piattaforme ciò può diventare piuttosto oneroso. Sarebbe, infatti, necessario scrivere più volte la stessa applicazione in quanto, come anche descritto nel capitolo 2, spesso per ogni sistema operativo vengono usati linguaggi e strumenti differenti. Se si dispone delle risorse necessarie per poter portare avanti più progetti contemporaneamente è senz'altro una strada percorribile, ma in caso contrario è probabilmente preferibile sviluppare un'applicazione cross-platform. Lo sviluppo cross-platform prevede che il codice venga scritto una volta sola per poi essere compilato nativamente per più piattaforme o eseguito all'interno di un contenitore. In entrambi i casi, tuttavia, la comodità di un codice condiviso può andare a discapito delle prestazioni, delle funzionalità o dell'aspetto grafico, dovendo effettuare scelte che siano compatibili con tutte le piattaforme. In ogni caso, riuscendo a trovare i giusti compromessi, lo sviluppo cross-platform può essere un ottimo strumento per raggiungere un'utenza più ampia riducendo notevolmente il carico di lavoro.

3.1 Applicazioni ibride e PWA

Un primo approccio per lo sviluppo cross-platform si basa sull'utilizzo di *applicazioni ibride*, ovvero applicazioni web eseguite all'interno di un wrapper. Un'applicazione web, o web app, è un'applicazione scritta in HTML, CSS e JavaScript distribuita tramite web e tendenzialmente fruita mediante un browser. Si tratta fondamentalmente di siti web ottimizzati per l'utilizzo su dispositivi mobile e, proprio per questo, sono utilizzabili soltanto quando il dispositivo è online e non hanno accesso alle funzionalità specifiche di quest'ultimo. Oltre a ciò, le web app hanno un'interfaccia unica, non specifica per le piattaforme e che quindi potrebbe poco integrarsi con queste. Per superare alcune di queste limitazioni è possibile

utilizzare strumenti come Apache Cordova che generano un'applicazione nativa per ogni piattaforma che incorpora l'applicazione web e fa da ponte tra questa ed il dispositivo per consentire l'utilizzo delle funzionalità del dispositivo tramite API apposite. Vi sono, poi, framework che non si limitano a creare un'applicazione che incorpori una web app già esistente, ma che permettono di creare interamente un'applicazione ibrida ottimizzata per mobile. Framework come Ionic, infatti, consentono di scrivere l'intera web app utilizzando anche varie librerie frontend JavaScript come Angular, React o Vue fornendo strumenti per creare interfacce che si avvicinino il più possibile alle interfacce native delle piattaforme mobile. In tal modo lo sviluppatore è supportato nel seguire le best practice relative all'aspetto grafico di ogni piattaforma ed è facilitato nella creazione di applicazioni mobile ibride che abbiano un aspetto molto vicino ai componenti della piattaforma stessa. Dopo aver creato l'interfaccia della web app, poi, tali framework forniscono gli strumenti di cui sopra per distribuirla come applicazione nativa per le varie piattaforme. Le applicazioni ibride sono senz'altro un'ottima soluzione per sviluppare per mobile facilmente se si deve creare un'applicazione semplice o se si ha già un sito web ben ottimizzato per mobile e con funzionalità sufficienti. I loro vincoli sono comunque molti e possono essere notevolmente limitanti se si ha l'obiettivo di creare un'applicazione complessa o si ha necessità di fare uso di funzionalità specifiche della piattaforma non supportate dal framework.

Recentemente, oltre alle applicazioni ibride, è stata introdota una nuova modalità di distribuzione le web app su dispositivi mobile che si affida alle sempre maggiori capacità dei moderni browser. Il nuovo approccio è quello delle *Progressive Web App* (PWA), ovvero applicazioni web che sfruttano funzionalità del dispositivo accessibili dal browser, funzionalità che vengono, come suggerisce il nome, progressivamente supportate nel corso del tempo. I browser che le supportano, inoltre, consentono di installare le PWA sul proprio dispositivo proprio come se fossero delle applicazioni native ed eseguite in una finestra indipendente invece che in una scheda del browser. Il vantaggio principale rispetto alle applicazioni ibride risiederebbe, quindi, nell'utilizzo del browser del dispositivo come ponte per utilizzare le sue funzionalità che renderebbe necessario il solo sviluppo della web app. D'altro canto, la dipendenza dal browser del dispositivo è anche una grande limitazione in quanto non tutti i browser forniscono le stesse funzionalità e si rischia, quindi, che su dispositivi diversi della stessa piattaforma la PWA non possa funzionare allo stesso modo o possa essere limitata in alcune sue parti. Rimangono, poi, anche le limitazioni delle applicazioni ibride, in particolare relativamente all'interfaccia dato che, per quanto possa essere ottimizzata per il mobile, dovrà essere unica per tutte le piattaforme. C'è anche da considerare che, nonostante si possa pensare che essendo distribuite sul web possano raggiungere più utenti, è probabile che in molti considerino le PWA soltanto come siti web e non considerino, quindi, l'idea di installarle sul dispositivo. Anche per questo motivo, Google

ha integrato nel Play Store il supporto alle PWA così che gli utenti possano trovarle come una qualunque altra applicazione, mentre Apple non ha ancora reso disponibile la loro distribuzione sul suo App Store. Le PWA rappresentano, perciò, una valida alternativa alle applicazioni ibride, condividendone alcuni difetti come l'interfaccia unica e funzionalità ridotte rispetto ad applicazioni native, ma avendo il vantaggio di poter essere distribuite semplicemente sul web senza dover essere compilate.

3.2 Framework multipiattaforma

Un altro approccio, molto popolare per lo sviluppo di applicazioni cross-platform, prevede l'utilizzo di framework multipiattaforma. I framework multipiattaforma consentono di scrivere del codice che viene, poi, compilato per generare un'applicazione che può essere eseguita su una piattaforma come un'applicazione nativa. Similmente ad applicazioni ibride e PWA, il vantaggio principale di questo metodo risiede nella possibilità di utilizzare un solo linguaggio per sviluppare l'applicazione, ma anche in questo caso potrebbero esserci delle limitazioni nell'utilizzo delle funzionalità specifiche dei dispositivi. Tali funzionalità, infatti, necessitano di essere implementate dal framework e potrebbero essere disponibili anche diverso tempo dopo essere state rilasciate nativamente o non essere affatto disponibili. I vari framework esistenti si distinguono per i differenti paradigmi in relazione alla quantità di codice condiviso.

Di seguito verranno analizzati tre di questi paradigmi prendendo d'esempio i framework multipiattaforma attualmente più utilizzati: Flutter, React Native e Xamarin.

Flutter

Flutter[54] è un framework open source sviluppato da Google che permette di compilare in nativo, nello specifico in librerie ARM scritte in C e C++, applicazioni scritte in Dart. Il paradigma che seguono framework come Flutter è quello di una condivisione del codice totale, sia per quanto riguarda le logiche di business che per l'interfaccia. Flutter fornisce, infatti, dei componenti denominati *widget* che possono essere utilizzati per creare interfacce che seguano i principi del Material Design[55], spesso utilizzati nello sviluppo di applicazioni Android, ma che poco si sposano con lo stile grafico tipico di iOS. I widget, però, sono molti e permettono di costruire delle interfacce sufficientemente articolate ed utilizzabili che restituiscano una buona esperienza d'uso. Oltre a dover condividere l'interfaccia sulle varie piattaforme, anche in questo caso potrebbero esserci problemi con l'utilizzo di alcune funzionalità native che potrebbero non essere supportate dal framework o supportate solo in parte. D'altro canto, l'utilizzo di un solo linguaggio consente

di creare e manutenere con maggiore facilità applicazioni compilate e distribuibili per le piattaforme Android ed iOS. Framework di questo tipo, perciò, sono ottimi per la creazione di applicazioni semplici che non necessitino di utilizzare componenti native specifiche, non richiedano funzionalità complesse e non debbano avere interfacce specifiche per le piattaforme.

React Native

Un approccio leggermente diverso è utilizzato da framework come React Native[56], che prevedono la compilazione in codice nativo di applicazioni scritte interamente in un altro linguaggio, ma consentono comunque di differenziare i componenti dell'interfaccia in base alla piattaforma. Mentre le logiche di business sono interamente condivise, nel codice relativo alla composizione dell'interfaccia è possibile indicare, dipendentemente dalle esigenze, quali elementi utilizzare per ogni piattaforma. Ciò aumenta la quantità di codice, ma permette di costruire delle interfacce che si integrino meglio nella piattaforma sulla quale l'applicazione viene eseguita. Tali framework lasciano, inoltre, la possibilità di inserire nel progetto nuovi componenti e funzionalità utilizzando il codice nativo della piattaforma, così da poter accedere a funzionalità non supportate dal framework ed avere un maggiore controllo. Scegliere di incorporare porzioni di codice nativo potrebbe, tuttavia, aumentare la complessità dell'applicazione e rendere più difficile il suo mantenimento, oltre a richiedere la conoscenza dei linguaggi utilizzati sulle piattaforme, perdendo così il principale vantaggio dello sviluppo con framework multipiattaforma. Un ulteriore problema potrebbe essere rappresentato dall'utilizzo di un linguaggio con notevoli differenze strutturali rispetto a quelli nativi delle piattaforme. React Native, ad esempio, utilizza JavaScript, in particolare la libreria React, la cui differenza principale rispetto sia al linguaggio usato per Android che a quello usato per iOS risiede nella sua natura single-thread. Per questo motivo React Native, così come altri framework che utilizzano JavaScript, potrebbe essere meno adatto per applicazioni che necessitino di avere alte prestazioni in tempo reale. Nonostante questa sua caratteristica, JavaScript è molto apprezzato per soluzioni di questo tipo per via della sua semplicità di utilizzo, della sua maturità e della sua enorme popolarità, che ha portato alla creazione di numerose librerie di componenti di interfaccia grafica compatibili con framework come React Native.

Xamarin

Il terzo paradigma è quello adottato, tra gli altri, da Xamarin[57], un framework di proprietà di Microsoft che prevede la condivisione del solo codice relativo alle logiche di business. In Xamarin, infatti, le logiche di business sono comuni alle due piattaforme e sviluppate in stile .NET, mentre le interfacce sono sviluppate specificamente per le piattaforme utilizzando degli SDK specifici. Tale approccio

consente di creare delle interfacce coerenti con quelle della piattaforma sulla quale l'applicazione è eseguita utilizzando gli strumenti messi a disposizione dagli SDK di Android ed iOS fornendo, così, una maggiore libertà nella loro creazione, ma aumenta notevolmente la complessità del codice. Nonostante le logiche di business compongano tendenzialmente buona parte di un'applicazione, scrivere anche delle parti di codice specifiche per le piattaforme rende necessario utilizzare tre SDK diversi, andando a perdere uno dei vantaggi di un framework multipiattaforma. Per questo motivo, da qualche anno è stato reso disponibile Xamarin.Forms, una libreria per la creazione di interfacce multipiattaforma che consente quindi, più similmente a quanto possibile con i framework precedentemente discussi, di sviluppare anche l'interfaccia utilizzando un solo SDK. L'approccio iniziale adottato da Xamarin resta uno dei più validi per semplificare lo sviluppo di applicazioni mobile per più piattaforme condividendo buona parte del codice fornendo al contempo un'esperienza utente più vicina a quella nativa.

3.3 Kotlin Multiplatform Mobile

Un approccio simile a quello appena descritto è stato adottato anche da JetBrains per la creazione di *Kotlin Multiplatform Mobile* (KMM) [58], un nuovo framework multipiattaforma basato, come suggerisce il suo nome, sul linguaggio Kotlin ed oggetto di studio di questa tesi. Allo stesso modo di Xamarin, infatti, KMM nasce con lo scopo di rendere le logiche di business condivise e la composizione dell'interfaccia specifica per ogni piattaforma. Il grande vantaggio rispetto al sopracitato framework consiste nella scelta di utilizzare per lo sviluppo del codice comune lo stesso linguaggio usato per lo sviluppo nativo per Android, Kotlin. In questo modo non sarebbero solo le logiche di business ad essere scritte in questo linguaggio, ma anche il codice specifico per una delle due piattaforme, così da ridurre la complessità. L'utilizzo di Kotlin, poi, assicura una già ottima integrazione sulla piattaforma Android, mentre per la compilazione per iOS è stato sviluppato un compilatore specifico integrato nell'infrastruttura Kotlin/Native che fornisce interoperabilità bidirezionale con i framework e le librerie native scritte in C e C++.

3.3.1 Kotlin Multiplatform

Kotlin Multiplatform Mobile è una specializzazione di *Kotlin Multiplatform* (KMP) un framework più ampio che permette di eseguire codice scritto in Kotlin su diverse piattaforme. Oltre ad applicazioni mobile, infatti, KMP è in grado di compilare anche applicazioni web full-stack e applicazioni per JVM grazie ai compilatori presenti in Kotlin/JVM, che compilano per Java Virtual Machine, e Kotlin/JS, che compilano per JavaScript.

KMP garantisce, così, una grande versatilità consentendo di combinare l'utilizzo dei diversi compilatori per creare, ad esempio, anche librerie multipiattaforma o web app che condividano la logica di applicazioni mobile native. Il funzionamento del framework può essere riassunto nel seguente schema.

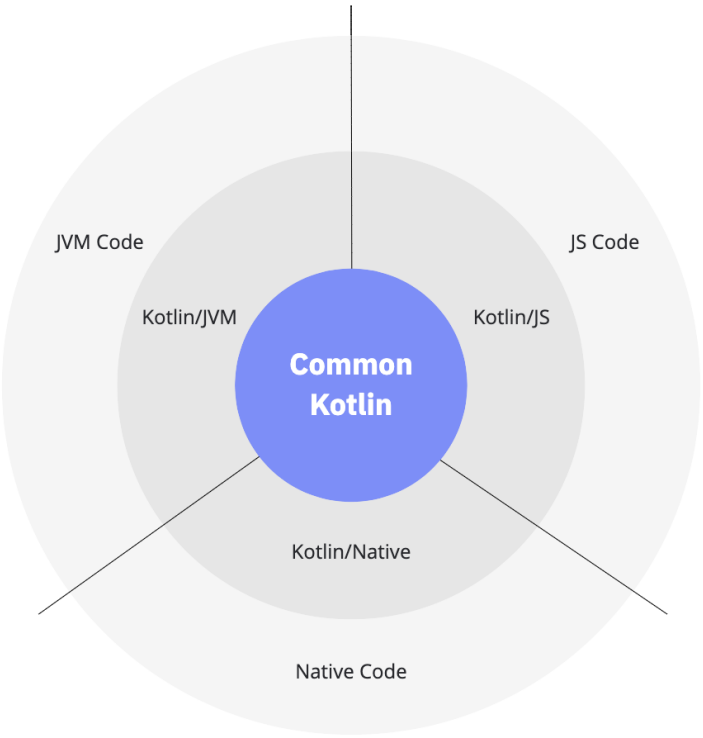

Figura 3.1: Schema di Kotlin Multiplatform

Nello schema, il centro, definito *Common Kotlin*, rappresenta la porzione di codice in Kotlin comune a tutte le piattaforme, che include librerie e strumenti di base e il linguaggio.

Nel primo anello sono presenti le infrastrutture che contengono i compilatori, ovvero delle versioni specifiche di Kotlin che forniscono l'interoperabilità con le piattaforme ed includono estensioni al linguaggio di base, oltre che strumenti e librerie specifiche per la piattaforma.

L'anello più esterno è composto dal codice nativo delle piattaforme al quale è possibile accedere per sfruttare funzionalità e librerie native grazie all'utilizzo dei compilatori.

Condivisione del codice

Grazie alla sua struttura, Kotlin Multiplatform consente non solo di condividere il codice con più piattaforme, ma anche di scegliere di condividere porzioni di codice solo con alcune piattaforme e non con tutte. KMP, infatti, permette di organizzare i progetti in strutture gerarchiche per creare dei set di istruzioni specifici condivisi tra più piattaforme.

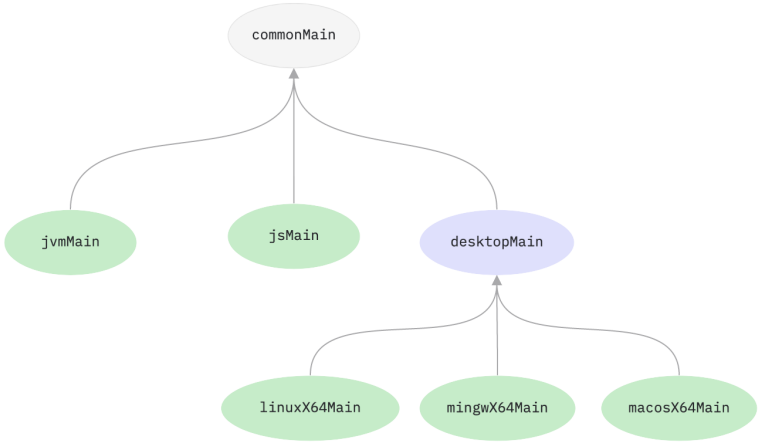

Figura 3.2: Esempio di una struttura gerarchica

In figura 3.2 è presente una struttura gerarchica d'esempio per un progetto che preveda la compilazione per JVM, JavaScript, Linux, Windows e macOS. Il progetto prevede che queste ultime tre piattaforme possano condividere una porzione maggiore di codice che non può essere, però, condiviso con le altre due. Per questo motivo, oltre al set commonMain che contiene il codice condiviso tra tutte le piattaforme, è stato creato un set intermedio definito desktopMain che contenga la logica la logica comune alle tre piattaforme. In Kotlin Multiplatfom generare dipendenze tra set è molto semplice in quanto il framework fornisce la relazione *dependOn*, da utilizzare in fase di impostazione del progetto.

Listato 3.4: Creazione del set desktopMain

```
kotlin{
    sourceSets {
        val desktopMain by creating {
            dependsOn(commonMain)
        }
```

```
        val linuxX64Main by getting {
            dependsOn(desktopMain)
        }

        val mingwX64Main by getting {
            dependsOn(desktopMain)
        }

        val macosX64Main by getting {
            dependsOn(desktopMain)
        }
    }
}
```

Come visibile in figura 3.2 e nel codice in 3.4 le piattaforme vengono definite da dei *target* legati all'architettura di destinazione. Per ogni piattaforma, infatti, possono essere presenti più target come avviene per Windows con i target *mingwX64* e *mingwX32*, destinati rispettivamente alle versioni a 64 e 32 bit del sistema operativo, o per macOS con i target *macosX64* e *macosArm64*, destinati alle piattaforme x86_64 e Apple Silicon. Per ridurre la complessità del codice in fase di impostazione Kotlin fornisce, inoltre, delle scorciatoie per creare semplicemente delle strutture che contengano combinazioni di target solitamente molto utilizzate, come la scorciatoia *ios* che crea una struttura gerarchica con all'interno i target *iosArm64* e *iosX64*.

Meccanismo *expect/actual*

Oltre alla possibilità di specificare tra quali piattaforme condividere una o più porzioni di codice, KMP fornisce un'utile funzionalità per implementare delle API specifiche per le piattaforme, le dichiarazioni *expect* ed *actual*. Tale meccanismo permette di definire nel codice comune una dichiarazione *expect* che dovrà essere implementata come dichiarazione *actual* nel codice specifico della piattaforma. Le parole chiave *expect* ed *actual* sono utilizzabili per la maggior parte delle dichiarazioni, come classi, interfacce, funzioni, annotazioni e proprietà. In fase di compilazione viene verificato che ogni dichiarazione *expect* abbia un suo corrispettivo *actual* nel modulo di ogni piattaforma.

Le dichiarazioni *expect* ed i loro corrispettivi *actual* devono avere lo stesso nome e, soprattutto, trovarsi nello stesso pacchetto, ovvero nello stesso livello della struttura del progetto sia nel modulo comune che in quelli specifici per le piattaforme. Le dichiarazioni *expect*, inoltre, sono sempre astratte per definizione e non possono contenere alcuna implementazione e se la dichiarazione è di un'interfaccia

anche le sue funzioni non possono avere un corpo. D'altro canto, le funzioni del corrispettivo *actual* possono anche non essere astratte ed avere un corpo, così che le classi eredi dell'interfaccia situate nel codice comune non necessitino di implementare le funzioni in quanto queste hanno delle implementazioni specifiche per le piattaforme.

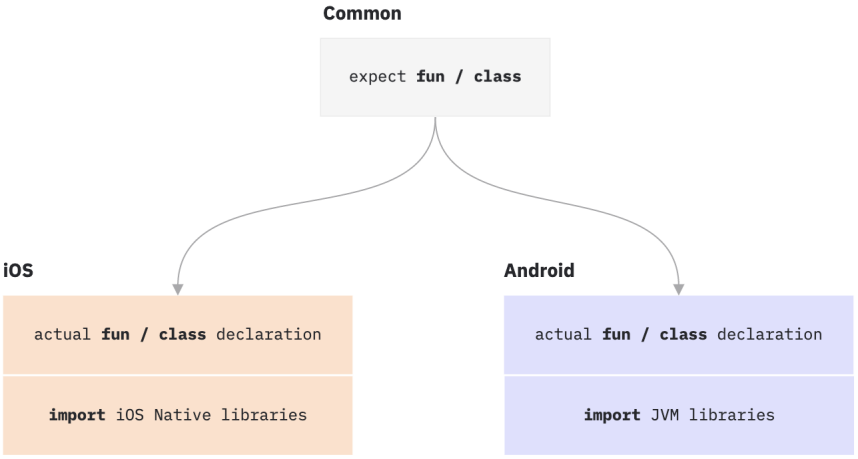

Figura 3.3: Dichiarazioni *expect/actual*

È possibile trovare implementazioni specifiche del meccanismo *expect/actual* usate nel progetto alla base di questa tesi nei capitoli 4.2 e 4.3.

Test

Kotlin Multiplatform consente anche di eseguire dei test per molte delle piattaforme supporatte, nello specifico per JVM, JavaScript, Android , Linux, Windows, macOS ed i simulatori di iOS, watchOS e tvOS. È possibile eseguire dei test anche su altri target, ma è necessario configurarli manualmente nell'apposito framework, ambiente od emulatore. Per effettuare i test Kotlin mette a disposizione la libreria *kotlin.test* che fornisce le annotazioni per contrassegnare le funzioni di test ed un set di funzioni che permettono di eseguire asserzioni nei test indipendentemente dal framework di test utilizzato. La libreria è, a sua volta, composta di diversi moduli, ognuno dei quali fornisce asserzioni, annotazioni e loro implementazioni compatibili con vari framework di test come JUnit o TestNG.

È possibile eseguire dei test sia sul codice comune sia su piattaforme specifiche, aggiungendo ai set specifici e comuni le dipendenze apposite in fase di impostazione. Ciò permette di analizzare l'integrazione del codice specifico per le piattaforme con il codice comune e, in particolar modo, di verificare la corretta implementazione di dichiarazioni *expect* ed *actual*.

Listato 3.5: Test per una dichiarazione *expect/actual*

```
// commonMain
expect object Platform {
    val name: String
}

fun hello(): String = "Hello, ${Platform.name}"

class Proxy {
    fun proxyHello() = hello()
}

// iosMain
actual class Platform actual constructor() {
    actual val platform: String = UIDevice.currentDevice.systemName() + " " +
    ↪  UIDevice.currentDevice.systemVersion
}

// androidMain
actual class Platform actual constructor() {
    actual val platform: String = "Android
    ↪  ${android.os.Build.VERSION.SDK_INT}"
}

// commonTest
class SampleTests {
    @Test
    fun testProxy() {
        assertTrue(Proxy().proxyHello().isNotEmpty())
    }
}

// iosTest
class SampleTestsIOS {
    @Test
    fun testHello() {
        assertTrue("iOS" in hello())
    }
}
```

```
// androidTest
class SampleTestsAndroid {
    @Test
    fun testHello() {
        assertTrue("Android" in hello())
    }
}
```

3.3.2 Kotlin/Native

Come mostrato in figura 3.1, le infrastrutture messe a disposizione da Kotlin Multiplatform forniscono un compilatore per ogni piattaforma per accedere alle librerie e funzionalità specifiche. Mentre Kotlin/JVM e Kotlin/JS sono utilizzati per generare codice destinato a macchine virtuali, Kotlin/Native è progettato principalmente per permettere la compilazione per piattaforme sulle quali non è preferibile o possibile eseguire una macchina virtuale. L'utilizzo ideale dell'infrastruttura è, infatti, per la produzione di programmi che debbano essere eseguiti nativamente sulla piattaforma senza necessità di ulteriori ambienti di runtime o, come detto, macchine virtuali. Kotlin/Native, poi, permette di generare da codice in Kotlin dei binari nativi per numerose piattaforme, fornendo diversi target per ognuna di esse[59]:

- *Android NDK*, destinato a produrre nativi per piattaforme Android con architettura x86, x86_64, ARM32 o ARM64
- *iOS*, destinato al sistema operativo mobile di Apple, nello specifico con target ARM32 per iPhone 5 e precedenti, ARM64 per iPhone 5s e successivi, X64 per la simulazione su sistemi a x86_64 e SimulatorArm64 per la simulazione su sistemi Apple Silicon
- *watchOS*, con target Arm32 per Apple Watch Series 3 e precedenti, Arm64 per Apple Watch Series 4 e successivi, X86 per la simulazione su sistemi x86_64 di watchOS a 32 bit, X64 per la simulazione su sistemi x86_64 di watchOS a 64 bit e SimulatorArm64 per la simulazione su sistemi Apple Silicon
- *tvOS*, con target Arm64 per Apple TV di quarta generazione o successive, X64 per la simulazione su sistemi x86_64 e SimulatorArm64 per la simulazione su sistemi Apple Silicon
- *macOS*, destinato al sistema operativo desktop di Apple, con target X64 per sistemi x86_64 e Arm64 per sistemi Apple Silicon

- *Linux*, con target X64 per piattaforme x86_64, Arm64 per piattaforme ARM64 come Raspberry Pi, Arm32Hfp per piattaforme ARM32 con calcolo a virgola mobile hardware, Mips32 per piattaforme MIPS e Mipsel32 per piattaforme MIPS little-endian

- *Windows*, destinato al sistema operativo desktop di Microsoft, nello specifico con target X64 e X86 per macchine rispettivamente a 64 e 32 bit

- *WebAssembly*, con target wasm32, che genera un binario conforme allo standard WebAssembly eseguibile nelle pagine web

Tutti i target relativi a piattaforme Apple richiedono che la macchina utilizzata per compilare abbia il sistema operativo macOS con installati Xcode ed i suoi strumenti da riga di comando.

Il supporto alle piattaforme Apple, inoltre, prevede anche la totale interoperabilità bidirezionale di Kotlin con Objective-C, le cui librerie e framework possono essere importati nel porgetto ed utilizzati nel codice in Kotlin. Attualmente le librerie scritte in Swift possono essere utilizzate nel codice in Kotlin solo se le loro API sono esportate in Objective-C in quanto i moduli interamente in Swift non sono ancora supportati. In ogni caso, Kotlin/Native fornisce un mapping, spesso bidirezionale, dei concetti di Kotlin a concetti similari in Swift ed Objective-C per poter passare dal codice comune a quello specifico senza soluzione di continuità. La parola chiave *null*, ad esempio, è convertita automaticamente in *nil*, le liste sono convertite in array, le mappe in dizionari e così via. Non tutti i mapping sono, però, così immediati dato che non tutte le funzionalità di Kotlin hanno delle controparti in Swift ed Objective-C. In particolare, una delle differenze principali dei linguaggi, che si riflette anche in una maggiore complessità nell'interoperabilità, è la gestione della concorrenza. Una delle funzionalità specifiche di Kotlin più utilizzate è la parola chiave *suspend*, trattata al capitolo 2.1.4, che permette di indicare facilmente una funzione asincrona e si integra con il meccanismo delle coroutine. In Swift ed Objective-C questa funzionalità è assente, per questo motivo Kotlin/Native converte le funzioni suspend in completion handler. L'utilizzo dei completion handler è leggermente più verboso dell'utilizzo delle funzioni suspend in Kotlin, ma la differenza maggiore risiede nella gestione delle eccezioni. Kotlin, infatti, a differenza di Swift ed Objective-C, non utilizza eccezioni checked, ma solo unchecked, perciò se il codice scritto in Swift od Objective-C richiama una funzione Kotlin che potrebbe generare un'eccezione è necessario esplicitarlo. Per farlo occorre contrassegnare la funzione con l'annotazione *@Throws* specificando una lista di eccezioni che potrebbero essere lanciate durante l'esecuzione. I completion handler generati da funzioni suspend contengono sempre un parametro di tipo NSError* in Objective-C ed Error in Swift con il quale è possibile specificare come gestire l'eventuale eccezione generata. Quando una funzione Kotlin richiamata

da codice in Swift od Objective-C genera un'eccezione di un tipo dichiarato nell'annotazione *@Throws* questa viene propagata come NSError, mentre se la classe dell'eccezione non è tra quelle dichiarate questa causa la cessazione dell'esecuzione dell'applicazione. Le funzioni suspend che non presentano un'annotazione *@Throws* propagano soltanto eccezioni di tipo CancellationException come NSError in Swift ed Objective-C.

```kotlin
class MyClass {

    /*...*/

    private suspend fun mySuspendFunction(): String {
        var string: String

        // codice asincrono per recuperare dati da inserire in string

        return string
    }
}
```

Listato 3.6: Definizione della funzione suspend in Kotlin

```swift
myObj = MyClass()

myObj.mySuspendFunction { result, error in
    if let result = result {
        print(result)
    } else if let error = error {
        print("Error: \(error)")
    }
}
```

Listato 3.7: Utilizzo della funzione suspend da Swift

Dalla versione 5.5 di Swift sono disponibili le funzioni async, come trattato al capitolo 2.2.2, ma attualmente l'utilizzo di queste in luogo dei completion handler come controparte delle funzioni suspend è ancora in fase sperimentale.

3.3.3 Gestione della concorrenza

Kotlin Multiplatform Mobile fornisce un modello per la concorrenza e gli stati per iOS differente da quello utilizzato quando si sviluppa esclusivamente per piattaforma Android. Swift ed Objective-C, come anche altri linguaggi, permettono a più thread di accedere allo stesso stato senza alcuna restrizione, pratica piuttosto rischiosa e propensa ad errori. I problemi generati dalla gestione della concorrenza sono difficili da identificare e riprodurre, potrebbero non presentarsi mai in fase di sviluppo, ma solo sporadicamente in produzione, magari con un utilizzo intenso dell'applicazione, perciò rappresentano un grande rischio. Per evitare di incappare in problemi relativi alla concorrenza, Kotlin/Native regola la condivisione di stati tra thread basandosi su un semplice principio: se uno stato è mutabile può essere visto da un solo thread, se è immutabile da più thread. Perché uno stato sia immutabile è necessario che anche tutti i suoi componenti siano immutabili. Una classe, ad esempio, è considerata immutabile solo se tutte le sue proprietà sono dichiarate come val e sono di un tipo finale o immutabile.

```
// classe immutabile perché ha parametri val e di tipi finali
data class MyClass(val s: String, val i: Int)

// classe mutabile perché uno dei parametri è var
data class MyClass(var s: String, val i: Int)

// classe mutabile perché il tipo MyInterface potrebbe essere mutabile
data class MyClass(val s: String, val myObject: MyInterface)
```

Listato 3.8: Definizione di una classe mutabile e di una immutabile

Nel codice in 3.8 l'ultimo esempio contiene un parametro di tipo MyInterface, ma non è possibile sapere durante la compilazione cosa questo farà e se, quindi, è mutabile o immutabile. Kotlin/Native può verificare la mutabilità di parte di uno stato solo in fase di esecuzione, ma farlo per ogni parte di ogni oggetto ad ogni esecuzione influirebbe senz'altro sulle prestazioni. Per questo motivo è stato definito un nuovo stato chiamato *frozen* che prevede che nessuna parte di esso possa essere modificata, generando un'eccezione del tipo *InvalidMutabilityException* se si provasse a farlo. Se l'istanza di un oggetto è frozen, inoltre, ogni altro oggetto referenziato da questo e tutti gli oggetti a loro volta referenziati da questi ricorsivamente divengono frozen e non possono, quindi, essere modificati. Ciò permette di determinare facilmente in fase di esecuzione se un oggetto può essere condiviso o meno tra più thread.

Per sfruttare questo meccanismo, Kotlin Multiplatform Mobile aggiunge a tutte le classi l'extension function *freeze* che rende l'oggetto sulla quale è richiamata e tutto ciò che è referenziato da questo frozen, ovvero completamente immutabile.

```
data class MoreData(val stringData: String, var width: Float)
data class SomeData(val moreData: MoreData, var count: Int)

/*...*/

var moreData = MoreData("abc", 10.0)
val someData = SomeData(moreData, 0)

someData.freeze()
// ora sia l'oggetto someData, che moreData e tutti i loro parametri sono
↪  frozen e non possono essere più modificati
```

Listato 3.9: Esempio di utilizzo della funzione *freeze*

Una volta utilizzata la funzione freeze su un oggetto questo non potrà più essere modificato in quanto non è possibile uscire dallo stato frozen. La funzione freeze è, inoltre, utilizzabile soltanto nel codice specifico delle piattaforme, non nel codice comune, dove la gestione della condivisione di uno stato tra più thread può essere gestito utilizzando librerie specifiche per la concorrenza come kotlinx.coroutines.

Anche in Kotlin Multiplatform Mobile è possibile definire uno stato disponibile globalmente, ma, per rispettare il principio dell'immutabilità, questo sarà automaticamente frozen e non potrà, quindi, essere modificato. Per permettere ad uno stato globale di essere mutabile, Kotlin/Native fornisce l'annotazione *@ThreadLocal* che genera una una copia dello stato diversa per ogni thread in cui è utilizzato. Ciò, tuttavia, implica che i dati dello stato potrebbero risultare differenti in thread differenti. *@ThreadLocal* può essere utilizzata per contrassegnare sia oggetti globali che companion object, ma anche proprietà globali che, però, hanno delle caratteristiche diverse dagli altri due. Le proprietà globali sono, infatti, disponibili soltanto nel main thread come stati mutabili. Con l'annotazione *@ThreadLocal* è possibile renderli disponibili anche agli altri thread e, in più, per le proprietà globali è disponibile l'annotazione *@SharedImmutable* che li rende disponibili globalmente, ma come stati frozen.

```
object MyGlobalObject{
    var count = 0
    fun add(){
        count++     // l'oggetto è frozen di default, perciò genera
        ↪  un'eccezione
    }
}

@ThreadLocal
object MyLocalObject{
    var count = 0   // leggendo count da differenti thread si potrebbero
    ↪  ottenere valori differenti
    fun add(){
        count++     // non viene generata alcuna eccezione
    }
}
```

Listato 3.10: Esempio di utilizzo dell'annotazione @ThreadLocal

Capitolo 4

Progettazione dell'applicazione

Per valutare l'utilizzo di Kotlin Multiplatform Mobile come strumento per sviluppare applicazioni cross-platform per Android e iOS è stato scelto di partire dal progetto di un'applicazione già esistente creata per il corso di Digital Interaction Design di Ingegneria del Cinema e dei Mezzi di Comunicazione. Kilood, questo è il nome dell'applicazione, è stata sviluppata per Android e consiste in una companion app per una bilancia smart, con la quale comunica per ottenere dati sulle pesate dell'utente e che integra al suo interno funzioni di storico delle pesate, dei pasti consumati e di obiettivi prefissati dall'utente. L'accesso all'applicazione è gestito mediante una logica di registrazione o login che consente di salvare i dati dell'utente su un server così da potervi accedere anche da dispositivi diversi.

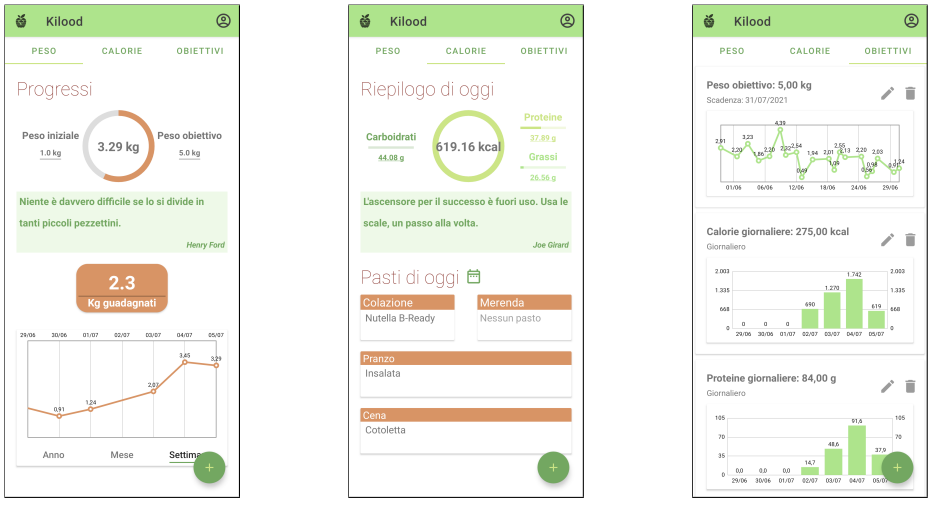

Figura 4.1: Schermate principali di Kilood

Le tre schermate principali, mostrate dopo l'accesso e tra le quali è possibile spostarsi con uno swipe laterale, mostrano le informazioni rispettivamente sul peso dell'utente, sui pasti consumati nel giorno corrente e sugli obiettivi prefissati. La prima schermata mostra, oltre allo storico delle pesate dell'utente in un intervallo temporale a scelta, anche il peso attuale ed il progresso rispetto all'eventuale obiettivo attuale. Nella seconda schermata è possibile consultare la lista dei pasti consumati nella giornata corrente e l'andamento degli eventuali obiettivi relativi al numero di calorie, carboidrati, proteine o grassi da assumere giornalmente. Nella terza schermata è presente la lista dagli obiettivi attualmente inseriti nel profilo dell'utente corredati da informazioni ad essi relative e grafici che ne mostrano l'andamento. Sovrapposto a tutte le schermate è presente un *Floating Action Button* che, se cliccato, mostra con un'animazione quattro bottoni che rimandano rispettivamente alle schermate di aggiunta di un nuovo obiettivo, di aggiunta di un nuovo pasto consumato, di consultazione della lista dei pasti preferiti salvati e di collegamento alla bilancia per l'aggiunta di una nuova pesata. Tutte le informazioni relative a nuovi obiettivi o nuovi pasti preferiti possono essere aggiunte manualmente, ma nel secondo caso è possibile anche scansionare un codice a barre con la fotocamera per ottenere i valori nutrizionali di un prodotto ricercandoli tramite il codice EAN sul database Open Food Facts[60]. Per la gestione dei pasti preferiti e dei pasti consumati sono presenti più schermate in quanto la logica scelta consiste nel permettere all'utente di salvare in una lista dei pasti preferiti contenenti le informazioni su ingredienti e valori nutrizionali in modo tale da non doverli inserire nuovamente ogni volta e poi, da una pagina differente, scegliere uno dei pasti salvati dalla suddetta lista per per indicare di averlo consumato nella giornata attuale indicando se è stato consumato per colazione, pranzo, cena o merenda. Da tutte le tre schermate principali è possibile, inoltre, cliccando sull'icona posta sulla destra nella toolbar superiore, accedere ad una schermata con le informazioni dell'utente in cui è possibile modificarle in qualunque momento.

Ai fini del progetto di tesi non è stata convertita l'intera l'applicazione, ma soltanto le sue parti fondamentali, utili a valutare l'efficacia di Kotlin Multiplatform Mobile nello sviluppo ed utilizzo delle principali funzioni di un'applicazione. Sono state tralasciate le funzionalità di registrazione e login, di scansione di codici a barre e di collegamento alla bilancia, è stata semplificata l'interfaccia eliminando grafici, animazioni e navigazione tramite gestures per renderla più funzionale ed è stata notevolmente semplificata la gestione dei pasti, eliminando la logica dei pasti preferiti. In questo progetto ci si è concentrati maggiormente sull'utilizzo di API RESTful per l'invio e la ricezione di dati, sul loro salvataggio in un database locale e sulla loro gestione ed elaborazione al fine di fornire su entrambe le piattaforme un'esperienza d'uso fluida e piacevole. Per questo motivo in fase di progettazione è stata prestata molta attenzione alla scelta del tipo di architettura dell'applicazione e a librerie, toolkit e framework da utilizzare.

4.1 Architettura: MVVM e Clean Architecture

Per questo progetto è stato scelto di utilizzare il pattern architetturale *Model-View-ViewModel* (MVVM) seguendo i principi della *Clean Architecture*[61]. La Clean Architecture è un insieme di principi teorizzati da Robert C. Martin, in arte Uncle Bob, che hanno l'obiettivo di creare del software semplice da scalare, da manutenere e da comprendere.

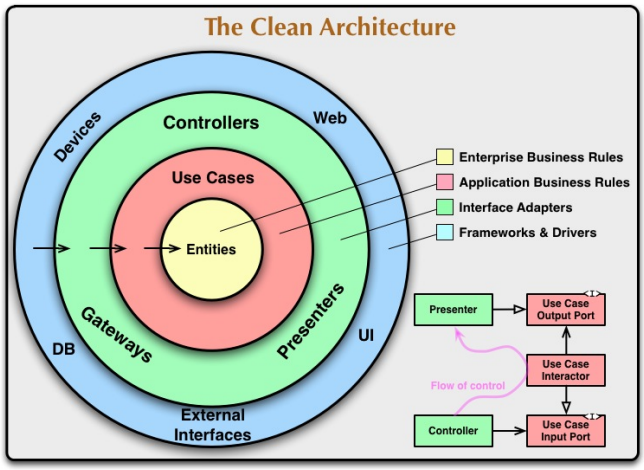

Figura 4.2: Schema della Clean Architecture

Tali principi si basano sulla separazione degli ambiti, ovvero sulla divisione degli elementi di un progetto in livelli, rappresentabili visivamente da un cerchio concentrico in cui agli anelli più esterni vi sono politiche di basso livello, mentre in quelli più interni vi sono politiche di alto livello. Il principio fondamentale di questa architettura prevede che le dipendenze nel codice sorgente puntino solo verso l'interno, ovvero dal livello basso a quello alto, e che ciò che si trova in un cerchio interno non conosca nulla di ciò che si trova in qualunque cerchio più esterno.

Al livello più alto, ovvero nel centro del grafico della Clean Architecture, si trovano gli elementi definiti entità che, nel caso specifico della creazione di un'applicazione, rappresentano gli oggetti operativi che contengono le regole generali e di alto livello. Sono, in generale, gli elementi che meno vengono affetti da cambiamenti che non li riguardano direttamente, come modifiche nel sistema di navigazione o nell'interazione con un server.

Nel secondo cerchio più interno vi sono le regole operative dell'applicazione, ovvero l'implementazione di tutti i suoi casi d'uso. Il codice contenuto in questo livello gestisce il flusso di dati inviati alle entità dai livelli più bassi e viceversa. Per

il principio di separazione degli ambiti, così come modifiche dei casi d'uso non influenzano le entità, anche questi non sono influenzati da modifiche a codice presente in livelli più bassi, ma solo da cambiamenti del funzionamento dell'applicazione.

Il livello subito inferiore contiene degli adattatori che convertono i dati ottenuti da attori esterni, come un database o il Web, nel formato più conveniente per entità e casi d'uso e viceversa. Allo stesso modo gli adattatori presenti in questo livello si occupano anche di convertire dati manipolati o inseriti dall'utente nella forma più conveniente per un eventuale framework di persistenza utilizzato, ad esempio un database locale. Ancora per il principio di separazione degli ambiti, il codice di tali adattatori non deve conoscere nulla del database che viene utilizzato.

Il cerchio più esterno del grafico della Clean Architecture contiene tutti gli strumenti e framework atti alla comunicazione con gli attori esterni all'applicazione. Eseguendo principalmente la funzione di passaggio di dati dall'interno dell'applicazione all'esterno e viceversa, il codice contenuto in questo livello è tendenzialmente poco e relativamente semplice.

Come già accennato, per questo progetto sono stati applicati i principi della Clean Architecture al pattern architetturale Model-View-ViewModel. Tale pattern divide l'applicazione in tre layer principali denominati, appunto, *Model*, *View* e *ViewModel*.

Il livello Model è il più basso e contiene tutte le classi, gli algoritmi e le strutture dati che recuperano dati dall'esterno e li trasformano in oggetti nativi Kotlin e viceversa.

Nel livello ViewModel vi sono le classi, gli algoritmi e le strutture dati che interagiscono con quelle nel livello Model e trasformano i dati nativi da questo forniti in dati pronti per essere presentati all'utente. È qui che viene applicata l'eventuale business logic del progetto.

Il livello più alto, il livello View, infine, contiene classi, algoritmi e strutture dati che ricevono dal livello ViewModel i dati da visualizzare a schermo e, nel caso in cui l'utente compia azioni che necessitino di elaborazione locale o remota, le passa al ViewModel stesso.

Applicando il principio di separazione degli ambiti della Clean Architecture, il livello Model, essendo il più basso, non dipende direttamente da nessun altro livello, si limita ad esporre valori e funzioni al livello ViewModel senza conoscere nulla del ViewModel stesso o dell'uso che ne fa. Allo stesso modo, il ViewModel espone dei valori che vengono osservati dal livello View, ma non espone nulla al livello Model, che si limita ad osservare, e il livello View non espone nulla ad alcun livello sottostante, ma può contenere oggetti che, dipendendo da essi, hanno accesso diretto a oggetti appartenenti al livello ViewModel. Tutto ciò è efficacemente riassunto nella seguente immagine, in cui le linee continue indicano una dipendenza diretta, mentre quelle tratteggiate indicano osservazione nel tempo.

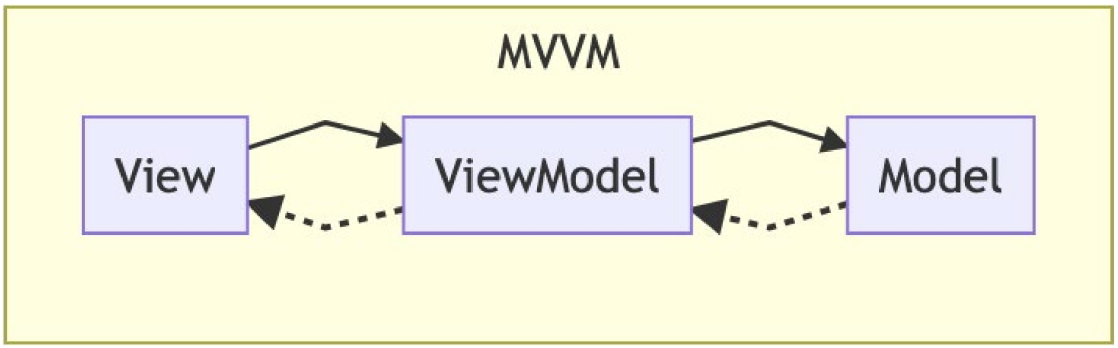

Figura 4.3: Pattern Model-View-ViewModel

A livello di codice, però, è stata applicata un'ulteriore divisione, basata sull'architettura utilizzata per lo sviluppo di applicazioni Android da Synesthesia. Nello specifico la suddivisione prevede la presenza di tre moduli, App, Data e Domain secondo lo schema di seguito riportato.

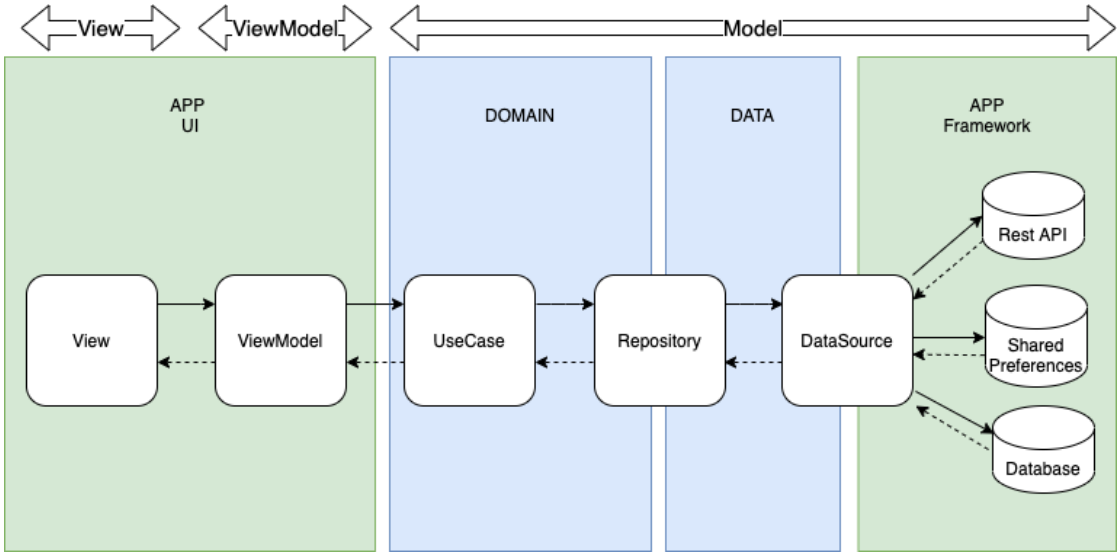

Figura 4.4: Schema dell'architettura usata da Synesthesia

Il modulo Domain contiene gli UseCase, che, come già accennato, rappresentano le logiche di business, le Entity, gli oggetti atomici che riflettono le suddette logiche, e le interfacce dei Repository, contenenti le dichiarazioni dei metodi che consentono agli UseCase di recuperare dati dal livello inferiore. Ogni UseCase può svolgere esclusivamente un singolo compito, ma può interfacciarsi con più di un Repository.

Il modulo Data può vedere le interfacce dei Repository dichiarate in Domain e può implementarle, per questo motivo in Figura 4.4 il Repository si trova a cavallo

tra i due livelli. Oltre alle implementazioni dei Repository, in questo livello sono presenti anche le interfacce dei DataSource, ovvero le fonti di dati locali, come un database, o remoti, come delle API. Così come uno UseCase può sfruttare più Repository, allo stesso modo un Repository può interfacciarsi con più DataSource.

Il modulo App è a sua volta suddiviso in due sezioni che rappresentano il livello più basso dello schema, Framework, ed il livello più alto, UI. Il livello Framework contiene l'implementazione dei DataSource, dei quali può vedere le interfacce presenti nel livello superiore, motivo per cui in Figura 4.4, così come i Repository, anche i DataSource si trovano a cavallo tra due livelli, e nei quali i dati ottenuti dalle strutture dati locali o remote vengono elaborati per essere, poi, esposti. Per questo motivo in questo livello sono presenti sia le response, oggetti ricevuti direttamente dalle API, che i mapper necessari a convertirle in Entity da poter utilizzare in livelli superiori. Tale scelta è motivata dalla volontà di rendere l'applicazione indipendente dalle strutture dati in arrivo dai backend. Il livello UI, infine, si occupa di manipolare i dati provenienti dai livelli inferiori per poi mostrarli tramite l'interfaccia all'utente. Più nello specifico tale manipolazione avviene all'interno dei ViewModel, nei quali i dati ottenuti dagli UseCase vengono trattati e semplificati per essere, successivamente, esposti ad Activity e Fragment. Questi si occupano semplicemente di osservarli e, senza modificarli in alcun modo, li utilizzano per aggiornare le View, ovvero i componenti che gestiscono la loro visualizzazione. Per rendere più semplice l'osservazione delle variabili, Kotlin fornisce delle classi Observable specifiche, come *LiveData*[62] e *Flow*[63], che permettono di definire molto semplicemente le azioni da compiere quando il valore della variabile cambia, come ad esempio aggiornare la vista che lo mostra o una progress bar.

Grazie ad una tale suddivisione, oltre al principio di separazione degli ambiti, questa architettura permette di applicare anche il principio di inversione delle dipendenze, secondo cui è necessario che le dipendenze vadano dal basso verso l'alto, ovvero che moduli di livello più alto non dovrebbero dipendere da moduli di livello più basso, e che le dipendenze dovrebbero fare riferimento solo ad astrazioni, che in questo caso sono le interfacce[64][65]. È per rispettare tale principio, ad esempio, che il modulo Domain, che si trova più in alto, contiene le interfacce dei Repository le cui implementazioni, che dipendono da esse, si trovano nel modulo Data.

Mentre per lo sviluppo di applicazioni per iOS spesso sono usati pattern architetturali differenti, come il Model-View-Controller, il pattern Model-View-ViewModel è quasi uno standard nello sviluppo di applicazioni Android, anche in virtù del fatto che Kotlin consente di rendere il proprio progetto conforme con estrema facilità grazie a funzioni e classi apposite, come la classe sopracitata Flow per l'utilizzo di variabili osservabili o la classe ViewModel che, come suggerisce il nome, semplifica la creazione e gestione di un ViewModel. Nello specifico, però, per questo progetto

è stato scelto il pattern MVVM specialmente per via dell'autonomia dei livelli inferiori, contenenti le logiche di business, rispetto al livello più alto, quello della UI. Ciò torna molto utile nella creazione di un'applicazione con Kotlin Multiplatform Mobile in quanto questo prevede che proprio il codice con le logiche di business venga condiviso tra le due piattaforme e che sia necessario scriverne di specifico per ognuna soltanto per la UI.

L'applicazione da convertire per questo progetto, Kilood, è stata già costruita seguendo il pattern Model-View-ViewModel, ma con una struttura meno precisa e pulita di quella descritta in precedenza.

Al livello più basso dell'applicazione si trovano i package *api*, in cui si trovano le API REST per comunicare con fonti di dati remote, e *data*, il quale, differentemente da ciò che avviene nell'architettura di Synesthesia, non rappresenta un modulo contenente le implementazioni dei Repository e le interfacce dei DataSource, ma contiene le dichiarazioni dei metodi usati per interagire con il database locale.

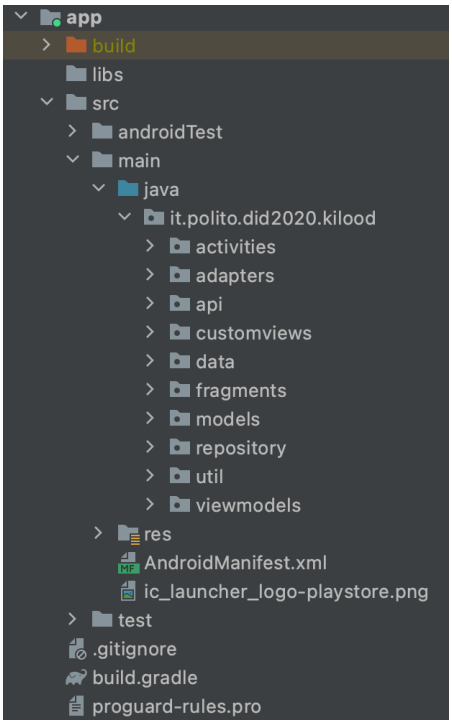

Figura 4.5: Struttura del progetto di Kilood

Ad un livello superiore vi sono i package *models*, in cui sono presenti gli equivalenti delle Entity precedentemente discusse, e *repository*, che contiene, appunto, le definizioni delle classi dei Repository che, in questo caso, svolgono anche il ruolo di DataSource, recuperando i dati direttamente dalle fonti locali o remote e rendendoli fruibili ai livelli superiori. Ad osservare valori e funzioni esposti dai Repository non sono gli UseCase, assenti in questa struttura, ma direttamente i ViewModel che, quindi, manipolano i dati ricevuti per renderli pronti per essere visualizzati.

Al livello più alto appartengono i package *activities*, *fragments*, *adapters* e *customviews*, in cui si trovano le classi che gestiscono l'interfaccia utente delle schermate, nello specifico Activity e Fragment si occupano di recuperare i dati dai ViewModel e di mostrarli usando elementi grafici personalizzati, situati nel package *customviews*, o nativi. Gli Adapter contenuti nel package *adapters* sono strumenti per gestire facilmente la visualizzazione di dati complessi come le liste.

Il package *util*, infine, contiene semplicemente delle classi e funzioni custom create per rendere più efficiente lo sviluppo.

Una delle differenze più evidenti tra la struttura di Kilood e quella dell'architettura usata da Synesthesia è senz'altro l'assenza di DataSource e UseCase. Oltre a ciò mancano anche le interfacce dei Repository, che sono, invece direttamente implementati, riducendo la modularità in quanto il modulo Domain non ha, in questa struttura, motivo di esistere. Tutto ciò incide notevolmente sull'applicazione del principio di separazione degli ambiti, aumentando i compiti da svolgere da Repository e ViewModel e rendendo meno modificabile, riparabile e testabile il codice. In fase di progettazione dell'applicazione cross-platform è stato, quindi, necessario aggiornare la struttura per renderla più fedele a quella adottata per le applicazioni di Synesthesia e conforme ai principi della Clean Architecture.

4.2 Database: SQLDelight

Per la gestione del database locale è stata scelta *SQLDelight*[66], una libreria che, a partire da delle query in SQL, genera delle API typesafe per Kotlin validandole e convertendole in codice che può essere usato da Kotlin Multiplatform Mobile prevenendo errori di tipo. Per via della sua facilità d'uso ed efficienza, la libreria è divenuta una delle più popolari per progetti multipiattaforma. L'intera gestione del database avviene mediante un file .sq, automaticamente identificato dalla libreria, al cui interno andranno inseriti gli schemi delle tabelle e tutte le query di inserimento, recupero ed eliminazione delle entry.

```
CREATE TABLE MyTable (
    id INTEGER NOT NULL PRIMARY KEY,
    value TEXT NOT NULL
);

insertEntry:
INSERT INTO MyTable(id, value)
VALUES(?, ?);

selectEntry:
SELECT * FROM MyTable
WHERE id = ?;

removeEntry:
DELETE FROM MyTable
WHERE id = ?;
```

Listato 4.11: File .sq di esempio

Una volta compilato il progetto, SQLDelight creerà una prima interfaccia relativa al database ed una seconda relativa specificamente alle query definite nel file .sq che espone una funzione per ogni query creata. I tipi degli oggetti che queste funzioni restituiscono o ricevono in input sono definiti automaticamente dalla libreria in base ai tipi delle relative colonne nello schema della tabella. Per inizializzare il database è necessario passare ad esso un'istanza di un *SQLite* driver, tipo di cui SQLDelight fornisce un'implementazione specifica per ogni piattaforma. Risulta, perciò, necessario in questo caso creare due istanze differenti per la piattaforma Android e per la piattaforma iOS. Per creare una classe che generi per ognuna delle due piattaforme il giusto driver SQLite è, dunque, possibile sfruttare il meccanismo *expect/actual* fornito da Kotlin Multiplatform Mobile come mostrato di seguito.

```
// codice comune
expect class DatabaseDriverFactory {
    fun createDriver(): SqlDriver
}

// codice specifico Android
actual class DatabaseDriverFactory(private val context: Context) {
    actual fun createDriver(): SqlDriver {
        return AndroidSqliteDriver(AppDatabase.Schema, context, "database.db")
    }
}

// codice specifico iOS
actual class DatabaseDriverFactory {
    actual fun createDriver(): SqlDriver {
        return NativeSqliteDriver(AppDatabase.Schema, "database.db")
    }
}
```

Listato 4.12: Creazione dell'SqlDriver con meccanismo *expect/actual*

Una volta implementato il driver SQLite questo può essere utilizzato per creare un oggetto con il quale poter gestire il database e le sue query. Se, ad esempio, al database fosse stato assegnato il nome *AppDatabase* nelle impostazioni del progetto, sarebbe possibile scrivere delle funzioni per interagirvi mediante le query create in maniera simile a come mostrato nel seguente codice.

```
private val database = AppDatabase(databaseDriverFactory.createDriver())
private val dbQueries = database.appDatabaseQueries

fun addEntry(
    id: Int,
    value: String
) {
    dbQueries.insertEntry(
        id,
        value
    )
}

fun getEntry(id: Int): Entry {
    return dbQueries.selectEntry(id, ::mapEntry).executeAsOne()
}

private fun mapEntry(
    id: Int,
    value: String
): Entity {
    return Entity (
        id = id,
        value = value
    )
}

fun deleteEntry(id: Int) {
    dbQueries.removeEntry(id)
}
```

Listato 4.13: Esempio di funzioni per effettuare interrogazioni al database

L'unica accortezza necessaria, come mostrato nel codice in 4.13, è la necessità di creare un mapper da fornire alle funzioni create da SQLDelight a partire dalle query per convertire gli eventuali risultati della query in oggetti utilizzabili in livelli più alti del codice. Queste stesse funzioni, inoltre, restituiscono degli oggetti di tipo Query, una classe della libreria che consente di elaborarli in vari modi o, usando la funzione *executeAsOne*, di convertirli semplicemente in oggetti del tipo specificato nel mapper.

4.3 Networking: KTOR

Per il networking è stato utilizzato Ktor[67], un framework asincrono open source per Kotlin sviluppato da JetBrains che sfrutta funzionalità specifiche del linguaggio come le coroutines per gestire efficientemente più richieste contemporaneamente. Grazie all'eccellente integrazione in Kotlin e per l'ottima implementazione dello sviluppo multipiattaforma è ormai diventato lo standard per progetti in KMM.

Per poter effettuare una richiesta con Ktor è necessario creare un *HttpClient* usando la funzione omonima fornita dal framework, con la quale è possibile specificare l'engine da utilizzare. Una specifica piattaforma, infatti, potrebbe richiedere uno specifico engine per processare le richieste. Anche per le piattaforme Android ed iOS esistono degli engine appositi, perciò è utile gestire anche la creazione dell'HttpClient inserendo una funzione expect nel codice comune ed implementando le rispettive actual per le specifiche piattaforme.

```
// codice comune
expect fun httpClient(config: HttpClientConfig<*>.()-> Unit = {}): HttpClient

// codice specifico Android
actual fun httpClient(config: HttpClientConfig<*>.()-> Unit = {}): HttpClient{
    return HttpClient(OkHttp) {
        config(this)
        engine {
            config {
                retryOnConnectionFailure(true)
                connectTimeout(5, TimeUnit.SECONDS)
            }
        }
    }
}

// codice specifico iOS
actual fun httpClient(config: HttpClientConfig<*>.()-> Unit = {}): HttpClient{
    return HttpClient(Darwin) {
        config(this)
        engine {
            configureRequest(requestConfig)
        }
    }
}
```

Listato 4.14: Creazione dell'HttpClient con meccanismo *expect/actual*

Come mostrato, ogni engine utilizza parametri di configurazione differenti, specifici per la piattaforma. L'engine utilizzato per iOS, inoltre, è denominato Darwin in quanto compatibile con tutti i sistemi operativi basati su Darwin, ovvero tutti i sistemi operativi progettati da Apple.

Una volta configurati gli HttpClient, effettuare delle richieste è estremamente semplice, è sufficiente creare dove necessario un client ed utilizzare le intuitive funzioni messe a disposizione da Ktor.

```
val client = httpClient()

val getResponse: HttpResponse = client.get("https://www.polito.it")

val postResponse: HttpResponse = client.post("http://localhost:8080/post") {
    setBody("Body content")
}
```

Listato 4.15: Esempi di richieste con Ktor

4.4 Dependency Injection: Koin

Uno dei *design pattern* più utilizzati per semplificare lo sviluppo con linguaggi di programmazione a oggetti è la *Dependency Injection*[68]. Tale pattern prevede che oggetti complessi non istanzino altri oggetti all'interno del proprio costruttore o dei propri metodi, ma che tali oggetti e le loro relative dipendenze siano costruiti da un componente esterno e siano fornite mediante *injection*. Anche per questo progetto è stato utilizzato il pattern della Dependency Injection e per la sua implementazione è stato scelto *Koin*[69], un framework per Kotlin sviluppato da Kotzilla molto semplice da configurare e facilmente implementabile anche in progetti multipiattaforma.

Per configurare Koin è necessario fornire le definizioni dei componenti da iniettare nell'applicazione, tali definizioni vanno inserite nei *moduli*. Il framework consente di dichiarare i componenti come singleton o factory, rispettivamente con le funzioni *single* e *factory*. Nel primo caso verrà creata un'istanza unica del componente che verrà fornita ogni volta che sarà richiesto di iniettarla. Nel secondo caso, invece, ogni volta che il componente verrà richiesto verrà creata una nuova istanza, distinta dalle altre. È possibile utilizzare la Dependency Injection anche quando si definiscono i componenti all'interno del modulo, ad esempio nel caso in cui all'interno del costruttore di un componente da iniettare si voglia, a sua volta, iniettare un oggetto. Per fare ciò, Koin mette a disposizione la comoda funzione *get*.

```
class MySingleton()
class MyFactory(singleton: MySingleton)

val myModule = module {
    single {
        MySingleton()
    }
    factory {
        MyFactory(get())
    }
}
```

Listato 4.16: Configurazione d'esempio di un modulo di Koin

Una volta configurati i moduli, bisogna configurare l'inizializzazione di Koin all'avvio dell'applicazione con la funzione apposita *startKoin*.

```
class MyApplication : Application() {
    override fun onCreate() {
        super.onCreate()
        startKoin {
            modules(myModule)
        }
    }
}
```

Listato 4.17: Inizializzazione di Koin

È, infine, possibile fornire ad una classe la capacità di utilizzare le funzionalità di Koin facendole estendere l'interfaccia *KoinComponent*. Al suo interno sarà, quindi, possibile iniettare istanze delle classi presenti nei moduli in due diversi modi, con la funzione *by inject* per ottenere un istanza con valutazione lazy, ovvero senza inizializzare l'oggetto fin quando questo non viene richiesto, o con la funzione *get* che, invece, fornisce l'istanza inizializzando subito l'oggetto.

```
class MyClass : KoinComponent {
    val lazySingleton: MySingleton by inject()
    val eagerSingleton: MySingleton = get()
}
```

Listato 4.18: Esempi di injection

4.5 UI: Jetpack Compose e SwiftUI

Come già accennato, in un progetto in Kotlin Multiplatform Mobile non tutto il codice può essere condiviso, quello che si occupa della composizione della UI deve, infatti, essere specifico della piattaforma, per cui i framework utilizzati sono due, uno per Android ed uno per iOS.

Negli ultimi anni si è assistito ad uno spostamento dagli strumenti utilizzati solitamente per la composizione della UI sulle due piattaforme, XML su Android e UIKit su iOS, verso dei nuovi framework basati sul paradigma della programmazione dichiarativa. A differenza di quella imperativa, questo tipo di programmazione prevede, infatti, che lo sviluppatore descriva l'obiettivo da raggiungere piuttosto che i passaggi da compiere per raggiungerlo. Ciò si presta perfettamente alla composizione della UI, in quanto utilizzando la programmazione dichiarativa è sufficiente che il codice descriva quale dovrebbe essere l'aspetto dell'interfaccia, semplificando l'intero processo. Usando questo paradigma, ad esempio, è possibile specificare facilmente quali componenti dell'interfaccia aggiornare quando lo stato dell'applicazione cambia e il codice scritto in questo modo è anche molto più semplice da leggere e modificare. Anche per questo progetto sono stati scelti i due framework UI dichiarativi recentemente creati per Android ed iOS denominati, rispettivamente, *Jetpack Compose*[70] e *SwiftUI*[71].

4.5.1 Jetpack Compose

La UI dell'applicazione alla base di questo progetto, Kilood, è stata costruita utilizzando file XML, contenenti le informazioni sul layout dell'interfaccia. Tale metodo, però, rende la modifica della UI da codice molto complessa e, di conseguenza, più complicata da implementare. Per l'applicazione cross-platform del progetto è stato, quindi, scelto di ricreare l'intera interfaccia utente utilizzando il toolkit Jetpack Compose, in quanto la programmazione dichiarativa risulta essere la soluzione ideale specialmente per applicazioni conformi ai principi della Clean Architecture.

Jetpack Compose è un toolkit sviluppato da Google che fa parte della suite Android Jetpack, contenente delle librerie che aiutano a seguire le best practice di sviluppo e semplificare la scrittura di codice che possa lavorare in maniera consistente su differenti dispositivi e versioni di Android. Come già accennato, Compose consente di costruire la UI di un'applicazione seguendo un approccio dichiarativo e permettendo, così, di scrivere meno codice. Alla base dell'utilizzo del toolkit vi sono le funzioni *composable*, con le quali è possibile definire l'interfaccia utente programmaticamente descrivendo l'aspetto che dovrebbe avere e fornendo le dipendenze che comporteranno eventuali aggiornamenti dell'interfaccia. È possibile

creare una funzione composable semplicemente aggiungendo l'annotazione *@Composable* al suo nome. Il toolkit fornisce numerose funzioni composable per creare elementi dell'interfaccia di base, come testi, immagini o bottoni. Differentemente da altri toolkit di UI, in Compose gli elementi dell'interfaccia non espongono metodi setter e getter, i valori da mostrare vengono, invece, passati nel momento in cui viene richiamata la funzione composable. È possibile, inoltre, modificare alcune proprietà del composable, come il padding assegnatogli o le sue dimensioni, semplicemente applicando dei modificatori come parametri.

```
@Composable
fun Greeting(name: List<String>) {
    Text(
        text = "Hello \$name",
        modifier = Modifier
            .padding(24.dp)
            .size(width = 200.dp, height = 100.dp)
    )
}
```

Listato 4.19: Esempio di funzione composable

Ogni funzione di questo tipo ha una *Composition* iniziale che viene eseguita solo nel momento in cui il contenuto della funzione deve essere visualizzato. Quando l'utente interagisce con l'interfaccia vengono generati degli eventi come l'*onClick*, che vengono propagati alla logica dell'app che si occupa, a sua volta, di gestire eventuali conseguenze. Se tali eventi dovessero causare delle modifiche dei dati da cui dipendono le informazioni visualizzate a schermo, le funzioni composable verrebbero chiamate nuovamente passando i nuovi dati, tale processo è definito *Recomposition*.

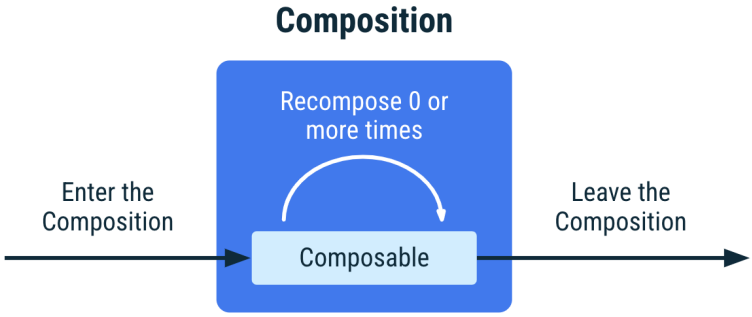

Figura 4.6: Ciclo di vita di una Composition

Se, ad esempio, si volesse disegnare un bottone che mostri il numero di volte che è stato premuto sarebbe sufficiente scrivere il codice sottostante.

```
@Composable
fun ClickCounter(clicks: Int, onClick: () -> Unit) {
    Button(onClick = onClick) {
        Text("Sono stato premuto \$clicks volte")
    }
}
```

Listato 4.20: Esempio di aggiornamento di una funzione composable

Il processo di recomposition, però, risulterebbe piuttosto inefficiente se ad ogni piccolo cambiamento venisse ricaricata l'intera interfaccia dell'applicazione, per questo motivo Jetpack Compose utilizza una *intelligent recomposition*. Tale processo prevede che soltanto le funzioni composable il cui input è cambiato vengano chiamate, in tal modo solo gli elementi che utilizzano i dati che sono cambiati vengono effettivamente aggiornati. Per gestire più facilmente l'aggiornamento delle variabili Compose fornisce gli *State*, degli oggetti che possono incapsulare un valore e che possono notificare un suo aggiornamento. In Kotlin sono già disponibili delle classi con uno scopo simile, i LiveData e i Flow, che è possibile convertire in State grazie a delle funzioni apposite fornite dal toolkit, così da permettere a Compose di aggiornare gli elementi della UI necessari quando uno stato cambia.

Con Jetpack Compose, infine, è anche possibile gestire semplicemente la navigazione tra composable. Per farlo è necessario creare un *NavController* in un punto del codice che permetta a tutti i composable di accedervi. È possibile, quindi, associare al NavController un grafo di navigazione che specifichi quali sono i composable tra cui navigare creando un *NavHost* al quale fornire il NavController stesso ed il composable di partenza. All'interno del NavHost è possibile definire i composable navigabili con il metodo *composable* al quale è necessario fornire un nome da associare alla destinazione e la funzione composable che la rappresenta. È, inoltre, possibile navigare passando degli argomenti tra le destinazioni aggiungendoli al nome come il percorso di un deep link. Di default gli argomenti vengono trattati come stringhe, ma è possibile specificare un tipo differente usando il parametro *arguments*.

Listato 4.21: Creazione del NavHost

```
@Composable
fun FirstComposable() {
    // Contenuto del composable
}
```

```
@Composable
fun SecondComposable(intInput: Int) {
    // Contenuto del composable
}

/*...*/

NavHost(navController = navController, startDestination = "first") {
    composable("first") { FirstComposable() }
    composable(
        "second/argument",
        arguments = listOf(navArgument("argument") { type = NavType.IntType })
    ) { backStackEntry ->
        SecondComposable(backStackEntry.arguments?.getInt("argument"))
    }
}
```

È possibile passare il navController a funzioni composable interne per gestire la navigazione al loro interno, ma è preferibile alzare la gestione della navigazione in un solo punto superiore a tutti i composable navigabili e passare ai singoli composable solo delle callback da invocare quando è necessario spostarsi.

```
// Composable che riceve in input il navController
@Composable
fun ControllerComposable(navController: NavController) {
    Button(onClick = { navController.navigate("otherComposable") }) {
        Text(text = "Navigate to other composable")
    }
    /*...*/
}

// Composable che riceve in input una callback
@Composable
fun CallbackComposable(navigateToOtherComposable: () -> Unit) {
    Button(onClick = { navigateToOtherComposable.invoke() }) {
        Text(text = "Navigate to other composable")
    }
    /*...*/
}
```

Listato 4.22: Due implementazioni della navigazione

Per le sue caratteristiche, Jetpack Compose risulta un'ottima scelta per un'applicazione basata su pattern architetturale MVVM. Esponendo al livello di ViewModel degli Observable che contengano i dati da mostrare all'utente, questi possono essere convertiti in State e, di conseguenza, passati direttamente alle funzioni composable.

4.5.2 SwiftUI

Lo strumento per comporre la UI su piattaforma iOS scelto per questo progetto è SwiftUI, un framework dichiarativo per la composizione di interfacce nato per sostituire il framework precedentemente utilizzato, UIKit. Anche SwiftUI, come Jetpack Compose, permette di rendere una view, questo è il nome degli elementi dell'interfaccia in Swift, funzione dello stato dell'applicazione, facendo sì che l'interfaccia cambi automaticamente al suo cambiamento. Per fare ciò è sufficiente che l'elemento creato sia conforme al protocollo View, il cui unico requisito è la proprietà *body*, a sua volta di tipo *some View*, che ritorna un'ulteriore view. Similmente a quanto avviene in Jetpack Compose, anche in SwiftUI è possibile modificare delle proprietà delle view, come tipo di font o colore di sfondo, utilizzando dei semplici modificatori.

```
struct Greeting: View {
    let name: String

    var body: some View {
        Text("Hello \(name)")
            .font(.largeTitle)
            .background(.blue)
    }
}
```

Listato 4.23: Esempio di oggetto conforme al protocollo View

Anche in SwiftUI, seguendo il paradigma della programmazione dichiarativa, le view sono dipendenti dai dati che vengono loro forniti, perciò nel momento in cui lo stato dell'app dovesse cambiare anche queste sarebbero aggiornate. Chiaramente, ancora per rendere il processo efficiente, non vengono aggiornate tutte le view ad ogni cambiamento dello stato, ma soltanto quelle dipendenti da dati che sono stati modificati. È buona pratica anche in questo caso, per tenere il codice pulito, non specificare nella dichiarazione della view eventuali conseguenze di un'interazione dell'utente, ma utilizzare delle callback, definite *closure* in Swift, per indicare le azioni da compiere in un punto più alto del codice.

Riprendendo l'esempio utilizzato precedentemente, per ottenere un bottone che mostri il numero di volte che è stato premuto in SwiftUI basterebbe scrivere il seguente codice.

```
struct ClickCounter: View {
    let clicks: Int
    let onClick: () -> Void

    var body: some View {
        Button("Sono stato premuto \(clicks) volte") {
            onClick()
        }
    }
}
```

Listato 4.24: Esempio di aggiornamento di una view

Differentemente da quanto avviene con Jetpack Compose, per facilitare l'osservazione di variabili, SwiftUI non fornisce una classe, ma il property wrapper *@State*, applicabile a proprietà di qualunque tipo e che consente a SwiftUI di gestire la sua conservazione indipendentemente dallo stato della struttura che, quindi, può essere distrutta e ricreata senza che lo stato venga perso. È preferibile che il wrapper *@State* venga usato solo su tipi semplici come String o Int e la proprietà alla quale è assegnato non dovrebbe essere condivisa tra più view. Per fare ciò, SwiftUI mette a disposizione altri property wrapper, *@ObservedObject* e *@EnvironmentObject*, che fanno sì che tutte le views vengano aggiornate quando il dato cambia. È possibile, però, applicare il wrapper *@ObservedObject* solo a classi che estendano il protocollo *ObservableObject*. Per far sì che le view che osservano un oggetto di questo tipo si aggiornino quando una o più delle sue proprietà vengono modificate è necessario che queste siano segnate come *@Published*.

Anche con SwiftUI è possibile gestire in maniera semplice la navigazione tra views e per farlo non viene utilizzato un grafo di navigazione che definisca da subito i possibili percorsi, bensì delle specifiche view, *NavigationView* e *NavigationLink*. La prima va posta nella view più esterna, così da farle contenere tutte le view sottostanti tra le quali sarà, quindi, possibile navigare. A tutte le view all'interno della NavigationView è possibile assegnare un titolo con il modificatore *navigationTitle*. La navigazione da una view all'altra è resa possibile inserendo la view di partenza in un NavigationLink che fa sì che al tap dell'utente su questa venga mostrata quella di destinazione. È possibile, infine, passare dei parametri alla view di destinazione semplicemente inserendoli nel suo costruttore.

```
struct SecondView: View {
    var parameter: String

    var body: some View {
        // Contenuto della view
    }
}

struct FirstView: View {
    let myString: String

    var body: some View {
        NavigationView {
            NavigationLink(destination: ResultView(parameter: myString)) {
                // Contenuto della view
            }
            .navigationTitle("Navigation")
        }
    }
}
```

Listato 4.25: Esempio di implementazione della navigazione

Capitolo 5

Scrittura dell'applicazione

In questo capitolo verrà descritto come è stata costruita l'applicazione, con particolare attenzione verso la corretta implementazione della parte di codice comune alle due piattaforme.

5.1 Importazione delle librerie

Dopo aver creato il progetto, per prima cosa è necessario aggiungere tutte le dipendenze per il corretto funzionamento di framework e librerie utilizzati. Kotlin Multiplatform Mobile consente di inserire le dipendenze per le librerie da utilizzare nel codice comune alle due piattaforme fornendo un file *build.gradle.kts* nel modulo condiviso. All'interno di questo file è possibile inserire diversi set specifici per ogni piattaforma, in cui inserire eventuali dipendenze di una libreria specifica.

Listato 5.26: Dipendenze del modulo condiviso

```
val coroutinesVersion = "1.6.3-native-mt"
val serialVersion = "1.3.3"
val ktorVersion = "2.0.2"
val koinVersion = "3.2"
val sqlDelightVersion = 1.5.3

// Set per dipendenze comuni
val commonMain by getting {
    dependencies {
        implementation(
            "org.jetbrains.kotlinx:kotlinx-coroutines-core:$coroutinesVersion"
        )
```

```
        implementation(
            "org.jetbrains.kotlinx:kotlinx-serialization-core:$serialVersion"
        )
        // Ktor
        implementation("io.ktor:ktor-client-core:$ktorVersion")
        implementation("io.ktor:ktor-client-serialization:$ktorVersion")
        // Koin
        implementation("io.insert-koin:koin-core:$koinVersion")
        // SQLDelight
        implementation("com.squareup.sqldelight:runtime:$sqlDelightVersion")
        implementation(
            "com.squareup.sqldelight:coroutines-extensions:$sqlDelightVersion"
        )
    }
}

// Set per dipendenze specifiche per piattaforma Android
val androidMain by getting {
    dependencies {
        // Ktor
        implementation("io.ktor:ktor-client-okhttp:$ktorVersion")
        // Koin
        implementation("io.insert-koin:koin-android:$koinVersion")
        // SQLDelight
        implementation(
            "com.squareup.sqldelight:android-driver:$sqlDelightVersion"
        )
    }
}

// Set per dipendenze specifiche per piattaforma iOS
val iosMain by getting {
    dependencies {
        implementation(
            "org.jetbrains.kotlinx:kotlinx-coroutines-core:$coroutinesVersion"
        ) {
            version {
                strictly(coroutinesVersion)
            }
        }
```

```
        // Ktor
        implementation("io.ktor:ktor-client-ios:$ktorVersion")
        // SQLDelight
        implementation(
            "com.squareup.sqldelight:native-driver:$sqlDelightVersion"
        )
    }
}
```

Nel codice qui mostrato le versioni delle librerie sono gestite tramite variabili per rendere più semplice il loro aggiornamento. È stato, inoltre, necessario specificare forzatamente la versione delle dipendenze per la libreria delle coroutine nel set della piattaforma iOS per evitare conflitti con altre librerie, in particolare Ktor che richiede espressamente l'utilizzo di una versione *native-mt*.

5.2 Funzioni e classi custom

Durante lo sviluppo dell'applicazione una delle sfide principali è stata rappresentata dalla differente gestione della concorrenza delle due piattaforme. Kotlin/Native compila il codice traducendo gli elementi scritti in Kotlin in elementi compatibili con Swift, ma non tutti gli elementi hanno un corrispettivo in Swift, ad esempio Observable come i LiveData e i Flow, e in alcuni casi questi sono più complessi da utilizzare delle loro controparti in Kotlin. Ne è un esempio la conversione che il framework fa delle coroutines che vengono trattate in Swift come completion handler, trattati nel capitolo 3.3.2, che, però, sono più complessi da utilizzare, oltre che più verbosi, e non forniscono alcun metodo per gestire la cancellazione della chiamata in quanto il Job della coroutine non è esposto.

5.2.1 runInsideOfCancelableCoroutine

La prima possibile soluzione che è stata presa in considerazione prevedeva due approcci differenti per le due piattaforme. Poiché il codice specifico per Android è scritto in Kotlin, l'idea prevedeva di propagare le coroutine fino al livello ViewModel in questa parte di codice per poter sfruttare tutte le funzionalità messe a disposizione dalle stesse e garantire una buona pulizia del codice. Per la piattaforma iOS, invece, sarebbe stato necessario creare una funzione specifica che convertisse efficacemente le coroutine fornendone tutte le funzionalità, compresa la loro cancellazione. Per rendere più semplice e, soprattutto, uniformare lo sviluppo su entrambe le piattaforme è stato successivamente scelto di evitare di propagare le

coroutine ed adottare anche nel codice specifico per piattaforma Android lo stesso approccio, incapsulando le coroutine in una funzione.

La funzione creata per tale scopo, *runInsideOfCancelableCoroutine*, riceve in input la coroutine da eseguire ed una callback nella quale viene fornito il risultato della coroutine stessa.

Listato 5.27: Definizione della funzione *runInsideOfCancelableCoroutine*

```
fun <T> runInsideOfCancelableCoroutine(
    callback: (Result<T>) -> Unit = { },
    task: suspend () -> T
): Cancelable {
    val job = Job()

    CoroutineScope(Dispatchers.Default + job).launch {
        ensureActive()
        val result = executeAndHandleExceptions(task)
        withContext(Dispatchers.Main) {
            callback.invoke(result)
        }
    }

    return object : Cancelable {
        override fun cancel() {
            job.cancel()
        }
    }
}
```

Listato 5.28: Definizione della funzione *executeAndHandleExceptions*

```
private suspend fun <T> executeAndHandleExceptions(
    action: suspend () -> T?
): Result<T> {
    return try {
        val data = action.invoke()
        Result(
            status = Status.SUCCESS,
            value = data,
            error = null
        )
```

```
    } catch (t: Throwable) {
        Result(
            status = Status.FAIL,
            value = null,
            error = t.stackTraceToString()
        )
    }
}
```

Il compito della funzione è quello di eseguire la coroutine in uno scope creato usando il dispatcher Default ed un job creato in precedenza. L'utilizzo del dispatcher Default, reso necessario dall'universalità di questa funzione, comporta una piccola perdita in termini di efficienza nella gestione di processi bloccanti rispetto al dispatcher IO, ma la potenza dei dispositivi mobile attuali rende tale perdita quasi trascurabile e, di conseguenza, questo compromesso più che accettabile.

All'interno dello scope viene prima verificato che la chiamata non sia stata cancellata, poi la coroutine viene eseguita mediante l'utilizzo della funzione *executeAndHandleExceptions* ed il suo risultato viene salvato in una variabile. La funzione *executeAndHandleExceptions* ha il compito di provare ad eseguire la coroutine e gestire eventuali errori che questa possa generare. Sia in caso di successo che in caso di fallimento ritornerà un oggetto di tipo Result.

```
class Result<T>(
    val status: Status,
    val value: T? = null,
    val error: String? = null
)

enum class Status {
    SUCCESS, FAIL, LOADING
}
```

Listato 5.29: Definizione della classe Result

Un oggetto Result contiene informazioni sullo stato dell'esecuzione della coroutine, se questa ha successo viene ritornato il suo risultato come parametro *value* ed il parametro *status* viene settato a SUCCESS. Al contrario, in caso dovesse essere sollevato un errore il parametro *status* è settato a FAIL e viene ritornata la descrizione del messaggio di errore come parametro *error*.

```
interface Cancelable {
    fun cancel()
}
```

Listato 5.30: Definizione dell'interfaccia Cancelable

La funzione *runInsideOfCancelableCoroutine*, infine, restituisce un oggetto di tipo Cancelable, un'interfaccia creata ad hoc che richiede soltanto l'implementazione del metodo *cancel* e la cui implementazione, in questo caso, consiste semplicemente nella cancellazione del job legato alla coroutine.

Per minimizzare l'impatto sulla scrittura di codice di livello più alto, queste funzioni vengono utilizzate all'interno dei Repository, che saranno discussi al paragrafo 5.5.1 e che si trovano al livello più basso in cui gestire effettivamente l'esecuzione di coroutines. Ciò consente di ridurre la complessità nel modulo Domain, così come anche in ViewModel e View, propagando soltanto callback ed oggetti di tipo Cancelable e Return.

Per questo progetto, infine, è stato scelto di mantenere la funzione molto semplice, ad esempio non implementando la logica del caricamento del processo, ma la sua struttura permette di implementare numerose altre funzionalità delle coroutine e di gestire l'esecuzione della coroutine in modi differenti.

5.2.2 CustomFlow

Come già accennato in precedenza, Kotlin/Native non fornisce un corrispettivo per i Flow in Swift e sarebbe, quindi, impossibile utilizzarli al livello ViewModel. Per questo motivo è stata realizzata una classe wrapper denominata *CustomFlow* che espone la funzione *watch* alla quale è possibile passare una callback in cui viene restituito il valore del Flow. Tale funzione è costruita similmente a *runInsideOfCancelableCoroutine* per poter sfruttare la funzione suspend *collect* fornita dalla classe Flow che consente di eseguire del codice ogni volta che il valore del Flow è aggiornato. In tal modo è possibile eseguire la callback passata in input a *watch* ad ogni aggiornamento del valore. CustomFlow estende, inoltre, la classe Flow, perciò eredita anche tutte le sue funzioni permettendo, così, di utilizzare dove possibile, ovvero al livello ViewModel su piattaforma Android, oggetti CustomFlow come normali Flow e sfruttarne tutte le funzionalità.

Listato 5.31: Definizione della classe CustomFlow

```
class CustomFlow<T>(flow: Flow<T>) : Flow<T> by flow {
    fun watch(onEach: (T) -> Unit): Cancelable {
        val job = Job()
```

```
        CoroutineScope(Dispatchers.Main + job).launch {
            ensureActive()
            try {
                collect {
                    onEach(it)
                }
            } catch (t: Throwable) {
                print(t.message)
            }
        }

        return object : Cancelable {
            override fun cancel() {
                job.cancel()
            }
        }
    }
}
```

Anche la funzione *watch*, come *runInsideOfCancelableCoroutine*, restituisce un oggetto di tipo Cancelable consentendo, in tal modo, di cancellare il processo dedicato all'osservazione del Flow quando non è più necessario ricevere il suo valore aggiornato.

5.3 Struttura dell'applicazione

Come è stato discusso nel paragrafo 4.1, la struttura dell'applicazione è stata costruita seguendo il pattern architetturale Model-View-ViewModel e i principi della Clean Architecture. Come già accennato, inoltre, con Kotlin Multiplatform Mobile è possibile condividere il codice che contiene delle logiche di business, mentre quello per la composizione della UI è specifico per ogni piattaforma. Per questo motivo in questo progetto il codice condiviso comprende l'intero livello Model descritto dal pattern MVVM, ovvero quelli che nell'architettura usata da Synesthesia sarebbero i moduli Domain e Data e la porzione Framework del modulo App, mentre per i livelli View e ViewModel è stato scritto del codice specifico per le due piattaforme. Per organizzare classi e metodi sono state, quindi, analizzate le funzionalità dell'applicazione per decidere come suddividere i compiti.

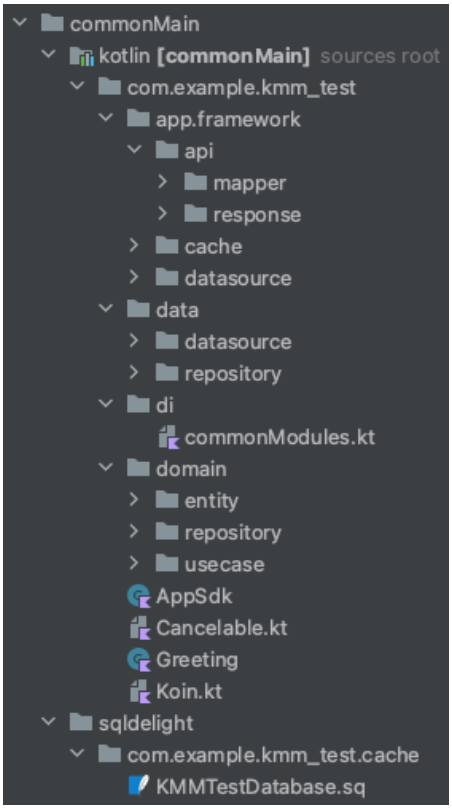

Figura 5.1: Struttura del progetto

L'idea di base è che all'avvio l'applicazione contatti un server remoto per recuperare dei dati aggiornati e li salvi nel database locale aggiornando quelli già presenti. Nel mentre verrebbe mostrata la prima schermata, quella relativa ai pesi, recuperando i dati da visualizzare dal database locale. Per semplicità non è stata implementata una logica di attesa del completamento della richiesta al server remoto, in caso la richiesta non venga completata prima che i dati vengano mostrati questi si aggiorneranno non appena verranno aggiornati quelli del database. I dati grezzi recuperati dal database dovrebbero, inoltre, essere manipolati nei ViewModel per renderli pronti per essere mostrati all'utente. L'interfaccia dovrebbe mettere a disposizione dell'utente un bottone che permetta di navigare verso una schermata nella quale, dipendentemente da quale fosse la schermata mostrata in precedenza, l'utente possa aggiungere dei nuovi dati. Perciò un tap sul bottone quando l'utente si trova nella schermata dei pesi doverebbe portare ad una schermata di aggiunta di un nuovo peso e, allo stesso modo se il tap avviene quando ci si trova nella schermata dei pasti o degli obiettivi, la schermata di destinazione sarà, rispettivamente, quella di aggiunta di un nuovo pasto o di aggiunta di un nuovo obiettivo. Se da una qualunque di queste schermate l'utente aggiungesse

un nuovo peso, pasto od obiettivo, questo andrebbe sia memorizzato nel database locale che inviato al server remoto. Nella schermata dei pasti dovrebbe esserci, poi, un bottone che consenta, una volta cliccato, di selezionare da un calendario il giorno di cui si vogliono visualizzare i pasti selezionati. Sia nella schermata dei pesi che in quella dei pasti è, inoltre, presente un box contenente una frase motivazionale. Nella schermata degli obiettivi dovrebbero, infine, essere presenti anche dei bottoni che forniscano all'utente la possibilità di modificare od eliminare un obiettivo. Il bottone di modifica dovrebbe aprire, similmente a quello di aggiunta, una schermata contenente i dati attuali dell'obiettivo recuperati nuovamente dal database locale e che fornisca all'utente la possibilità di modificarli. In caso di modifica di tali dati, l'obiettivo aggiornato verrebbe inviato al server remoto e memorizzato nel database locale sostituendo quello già presente al suo interno.

5.4 Modulo Domain

Poiché tutto il codice relativo alla UI è specifico delle due piattaforme, il modulo più alto presente nel codice comune è il modulo Domain. Al suo interno sono presenti le definizioni delle Entity e degli UseCase e le interfacce dei Repository.

5.4.1 Entity

Come già descritto nel paragrafo 4.1 le Entity sono gli oggetti atomici che riflettono le logiche di business, sono costruite come semplici *data class* contenenti i parametri che identificano l'oggetto.

Poiché in questo progetto gli oggetti da gestire sono di tre tipi, ovvero pesi, pasti ed obiettivi, sono state create tre Entity:

WeightEntity Contiene le informazioni di una pesata, i suoi unici parametri sono *date*, che rappresenta la data e l'ora della pesata, e *weight*, ovvero il valore della pesata espresso in chili

MealEntity Contiene le informazioni relative ad un pasto, possiede i parametri:

- *id*, l'id univoco assegnato al pasto
- *date*, il giorno in cui il pasto è stato aggiunto
- *type*, quando il pasto è stato consumato durante la giornata, se a colazione, pranzo, cena o merenda
- *name*, il nome assegnato al pasto
- *calories*, la quantità di calorie contenute in 100 g
- *proteins*, la quantità di proteine contenute in 100 g

- *fats*, la quantità di grassi contenuti in 100 g
- *carbs*, la quantità di carboidrati contenuti in 100 g
- *grams*, la quantità in grammi di pasto consumato

GoalEntity Contiene le informazioni di un obiettivo, i suoi parametri sono:

- *category*, il tipo di obiettivo, se relativo ad un peso da raggiungere entro una certa data o la quantità giornaliera da consumare di un determinato valore nutrizionale
- *deadline*, usata solo se l'obiettivo riguarda un peso da raggiungere, rappresenta la data entro la quale lo si vuole raggiungere
- *startingValue*, anche questa è usata solo per un obiettivo relativo al peso ed identifica il peso che si aveva quando l'obiettivo è stato aggiunto
- *targetValue*, il valore da raggiungere, può essere il peso ideale o la quantità di calorie, grassi, proteine o carboidrati da consumare giornalmente

```
data class GoalEntity(
    val category: Int,
    val deadline: String = "",
    val startingValue: Float = -1f,
    val targetValue: Float
)
```

Listato 5.32: Definizione della classe GoalEntity

5.4.2 UseCase

Gli UseCase rappresentano, come suggerisce il nome, tutti i casi d'uso, ovvero le logiche di business vere e proprie dell'applicazione. Ogni UseCase è specifico per una singola logica ed è definito come una classe che espone una singola funzione che svolga il compito dello UseCase. Nel costruttore della classe vengono iniettati i Repository necessari a svolgere tale compito. Per questo progetto sono stati creati i seguenti UseCase:

AddWeightUseCase Si occupa dell'aggiunta di una nuova pesata, espone la funzione *addWeight* che riceve in input la data ed il valore della pesata e li passa al Repository che gestisce i pesi

AddMealUseCase Si occupa dell'aggiunta di un nuovo pasto, espone la funzione *addMeal* che riceve in input i parametri del pasto e li passa al Repository che gestisce i pasti

AddGoalUseCase Si occupa dell'aggiunta di un nuovo obiettivo, espone la funzione *addGoal* che riceve in input i parametri dell'obiettivo inseriti dall'utente e li passa al Repository che gestisce gli obiettivi

DeleteGoalUseCase Si occupa dell'eliminazione di un obiettivo creato in precedenza, espone la funzione *deleteGoal* che riceve in input la categoria dell'obiettivo e lo passa al Repository che gestisce gli obiettivi

EditGoalUseCase Si occupa della modifica di un obiettivo già esistente, espone la funzione *editGoal* che riceve in input i nuovi parametri dell'obiettivo e li passa al Repository che gestisce gli obiettivi

GetWeightsUseCase Si occupa di recuperare tutte le pesate presenti nel database locale, espone la funzione *getWeights* alla quale è possibile passare in input una callback in cui viene fornito un oggetto di tipo Result che incapsula la lista di pesate

GetMealsUseCase Si occupa di recuperare tutti i pasti presenti nel database locale, espone la funzione *getMeals* alla quale è possibile passare in input una callback in cui viene fornito un oggetto di tipo Result che incapsula la lista di pasti

GetMealsFromDayUseCase Si occupa di recuperare tutti i pasti di un determinato giorno presenti nel database locale, espone la funzione *getMeals* che riceve in input il giorno scelto ed una callback in cui viene fornito un oggetto di tipo Result che incapsula la lista di pasti richiesti

GetGoalUseCase Si occupa di recuperare uno specifico obiettivo dal database locale, espone la funzione *getMeals* che riceve in input la categoria dell'obiettivo ed una callback in cui viene fornito un oggetto di tipo Result che incapsula l'obiettivo richiesto

GetGoalsUseCase Si occupa di recuperare tutti gli obiettivi presenti nel database locale, espone la funzione *getGoals* alla quale è possibile passare in input una callback in cui viene fornito un oggetto di tipo Result che incapsula la lista di obiettivi

LoadWeightsUseCase Utilizzato per aggiornare i dati dei pesi nel database locale recuperando le nuove informazioni dal server remoto, espone la funzione *loadWeights* che semplicemente richiama l'omonima funzione del Repository che gestisce i pesi

LoadMealsUseCase Utilizzato per aggiornare i dati dei pasti nel database locale recuperando le nuove informazioni dal server remoto, espone la funzione *loadMeals* che semplicemente richiama l'omonima funzione del Repository che gestisce i pasti

LoadGoalUseCase Utilizzato per aggiornare i dati degli obiettivi nel database locale recuperando le nuove informazioni dal server remoto, espone la funzione *loadGoals* che semplicemente richiama l'omonima funzione del Repository che gestisce gli obiettivi

```
class AddWeightUseCase(private val weightRepository: WeightRepository) {
    fun addWeight(date: Long, weight: Float) =
        weightRepository.addWeight(date, weight)
}
```

Listato 5.33: Definizione della classe AddWeightUseCase

```
class GetMealsFromDayUseCase(private val mealRepository: MealRepository) {
    fun getMeals(
        date: String,
        callback: (Result<CustomFlow<List<MealEntity>>>) -> Unit
    ) = mealRepository.getMealsFromDay(date, callback)
}
```

Listato 5.34: Definizione della classe GetMealsFromDayUseCase

```
class EditGoalUseCase(private val mealRepository: GoalRepository) {
    fun editGoal(
        category: Int,
        deadline: String,
        startingValue: Float,
        targetValue: Float
    ) = mealRepository.editGoal(category, deadline, startingValue,
    ↪  targetValue)
}
```

Listato 5.35: Definizione della classe EditGoalUseCase

5.4.3 Interfacce dei Repository

Per rispettare il principio della separazione degli ambiti, in questo livello sono presenti solo le interfacce dei Repository, così da rendere il codice presente nel modulo indipendente dalla loro implementazione e migliorare scalabilità e modificabilità. È stato creato un Repository per ogni Entity, esistono quindi tre Repository: *WeightRepository*, *MealRepository* e *GoalRepository*.

Nell'esempio sotto riportato si può notare che tutte le funzioni non restituiscono Entity o liste di Entity, ma oggetti di tipo Cancelable, classe descritta nel paragrafo 5.2.1. I risultati delle chiamate al database sono forniti all'interno della callback passata come input alle funzioni incapsulati in oggetti di tipo Result e CustomFlow, classi descritte anch'esse rispettivamente nei paragrafi 5.2.1 e 5.2.2.

```
interface GoalRepository {
    fun loadGoals(): Cancelable

    fun getGoals(
        callback: (Result<CustomFlow<List<GoalEntity>>>) -> Unit
    ): Cancelable

    fun getGoal(
        category: Int, callback: (Result<CustomFlow<GoalEntity?>>) -> Unit
    ): Cancelable

    fun addGoal(
        category: Int,
        deadline: String,
        startingValue: Float,
        targetValue: Float
    ): Cancelable

    fun editGoal(
        category: Int,
        deadline: String,
        startingValue: Float,
        targetValue: Float
    ): Cancelable

    fun deleteGoal(category: Int): Cancelable
}
```

Listato 5.36: Definizione dell'interfaccia del GoalRepository

5.5 Modulo Data

In questo modulo i dati recuperati dai DataSource vengono convertiti per essere gestiti più facilmente dai livelli più alti dell'applicazione. In particolare, all'interno dei Repository vengono utilizzate le funzioni e le classi presentate al paragrafo 5.2 per processi asincroni ed oggetti osservabili utilizzabili anche nelle porzioni di codice scritte in Swift.

5.5.1 Implementazioni dei Repository

All'interno del modulo Data sono presenti le implementazioni delle interfacce dei Repository presenti nel modulo Domain. Alla classe di ogni implementazione viene assegnato come nome quello dell'interfaccia seguito dal suffisso *Impl*, ovvero *WeightRepositoryImpl*, *MealRepositoryImpl* e *GoalRepositoryImpl*. Data la semplicità dell'applicazione le azioni da svolgere nei Repository sono molto simili, per cui anche le loro funzioni sono simili. Per questo motivo in questo paragrafo verranno descritte nello specifico solo le funzioni del GoalRepository:

- *loadGoals*, ottiene dal DataSource remoto i dati aggiornati degli obiettivi e li passa al DataSource locale; i suoi corrispettivi nei Repository dei pesi e dei pasti sono chiamati, rispettivamente, *loadWeights* e *loadMeals*

- *getGoals*, recupera dal DataSource locale la lista di tutti gli obiettivi presenti; i suoi corrispettivi nei Repository dei pesi e dei pasti sono chiamati, rispettivamente, *getWeights* e *getMeals*

- *getGoal*, recupera dal DataSource locale l'obiettivo desiderato in base alla sua categoria; non ha corrispettivi negli altri Repository

- *addGoal*, riceve tutti i parametri dell'obiettivo da aggiungere, li invia al DataSource remoto e li passa al DataSource locale; i suoi corrispettivi nei Repository dei pesi e dei pasti sono chiamati, rispettivamente, *addWeight* e *addMeal*

- *editGoal*, riceve tutti i parametri dell'obiettivo da modificare, li invia al DataSource remoto e li passa al DataSource locale; non ha corrispettivi negli altri Repository

- *deleteGoal*, riceve la categoria dell'obiettivo da eliminare, la invia al DataSource remoto e la passa alla funzione del DataSource locale per eliminare l'obiettivo; non ha corrispettivi negli altri Repository

L'unica funzione presente in un altro Repository, ma assente in quello degli obiettivi si trova nel MealRepository ed è *getMealsFromDay*, che recupera dal DataSource locale la lista di pasti Consumati nel giorno specificato.

```
class WeightRepositoryImpl(
    private val remoteDataSource: WeightRemoteDataSource,
    private val localDataSource: WeightLocalDataSource
) : WeightRepository {

    override fun loadWeights() = runInsideOfCancelableCoroutine {
        with(remoteDataSource.getWeights()) {
            localDataSource.addWeights(this)
        }
    }

    override fun getWeights(
        callback: (Result<CustomFlow<List<WeightEntity>>>) -> Unit
    ): Cancelable =
        runInsideOfCancelableCoroutine(callback) {
            CustomFlow(localDataSource.getWeights())
        }

    override fun addWeight(
        date: Long,
        weight: Float
    ) = runInsideOfCancelableCoroutine {
        remoteDataSource.addWeight(date, weight)
        localDataSource.addWeight(date, weight)
    }

}
```

Listato 5.37: Definizione della classe WeightRepositoryImpl

Poiché tutte le funzioni presenti nei Repository dovrebbero eseguire delle coroutine per svolgere il loro compito, per ognuna di queste è stata utilizzata la funzione *runInsideOfCancelableCoroutine*, discussa nel paragrafo 5.2.1, per evitare di propagare la coroutine stessa, dato che non sarebbe gestibile in Swift, e fornire, invece, il suo risultato in una callback. Tale risultato è incapsulato in un oggetto di tipo Result che consente di gestire anche eventuali errori sollevati dall'esecuzione della coroutine. Le funzioni del Repository restituiscono un oggetto di tipo Cancelable,

classe anch'essa discussa al paragrafo 5.2.1, che espone soltanto un metodo che consente di cancellare l'esecuzione della coroutine. Le funzioni che si occupano di recuperare dati dal database e ricevono, quindi, dai DataSource locali degli oggetti di tipo Flow li convertono in CustomFlow per permettere il loro utilizzo anche nel codice specifico per la piattaforma iOS scritto in Swift, come discusso al paragrafo 5.2.2.

5.5.2 Interfacce dei DataSource

Come per i Repository, anche per i DataSource le interfacce si trovano in questo modulo e le loro implementazioni nel modulo sottostante. Per ogni Repository sono stati creati un DataSource locale, che ha il compito di interfacciarsi con il database, ed un DataSource remoto, che deve, invece, interfacciarsi con il server remoto. I DataSource presenti nell'applicazione sono, quindi: *WeightLocalDataSource*, *WeightRemoteDataSource*, *MealLocalDataSource*, *MealRemoteDataSource*, *GoalLocalDataSource* e *GoalRemoteDataSource*.

```
interface GoalRemoteDataSource {
    suspend fun getGoals(): List<GoalEntity>

    suspend fun addGoal(
        category: Int,
        deadline: String,
        startingValue: Float,
        targetValue: Float
    )

    suspend fun editGoal(
        category: Int,
        deadline: String,
        startingValue: Float,
        targetValue: Float
    )

    suspend fun deleteGoal(category: Int)
}
```

Listato 5.38: Definizione dell'interfaccia del GoalRemoteDataSource

Listato 5.39: Definizione dell'interfaccia del GoalLocalDataSource

```
class GoalLocalDataSourceImpl(
    private val database: KMMTestDatabase
) : GoalLocalDataSource {

    override fun getGoals() =
        database.kMMTestDatabaseQueries.selectAllGoals(::mapGoal)
            .asFlow().mapToList()

    override fun getGoal(category: Int) =
        database.kMMTestDatabaseQueries.selectGoal(category, ::mapGoal)
            .asFlow().mapToOneOrNull()

    override fun addGoal(
        category: Int,
        deadline: String,
        startingValue: Float,
        targetValue: Float
    ) =
        database.kMMTestDatabaseQueries.insertGoal(
            category,
            deadline,
            startingValue,
            targetValue
        )

    override fun addGoals(newGoals: List<GoalEntity>) =
        database.kMMTestDatabaseQueries.transaction {
            database.kMMTestDatabaseQueries.removeAllGoals()
            newGoals.forEach {
                database.kMMTestDatabaseQueries.insertGoal(
                    it.category,
                    it.deadline,
                    it.startingValue,
                    it.targetValue
                )
            }
        }
```

```
    override fun editGoal(
        category: Int,
        deadline: String,
        startingValue: Float,
        targetValue: Float
    ) =
        database.kMMTestDatabaseQueries.insertGoal(
            category,
            deadline,
            startingValue,
            targetValue
        )

    override fun deleteGoal(category: Int) =
        database.kMMTestDatabaseQueries.removeGoal(category)

    private fun mapGoal(
        category: Int,
        deadline: String,
        startingValue: Float,
        targetValue: Float
    ): GoalEntity {
        return GoalEntity(
            category = category,
            deadline = deadline,
            startingValue = startingValue,
            targetValue = targetValue
        )
    }
}
```

5.6 Modulo App - Framework

Seguendo il pattern architetturale utilizzato da Synesthesia descritto al paragrafo 4.1, il modulo App è stato separato in due sezioni, Framework e UI. Il supporto allo sviluppo multipiattaforma delle librerie Ktor e SQLDelight consente di condividere il codice di Framework tra le due piattaforme, ma quello di UI deve necessariamente essere specifico per ognuna. In questo paragrafo verrà descritto esclusivamente il codice di Framework, mentre quello di UI verrà descritto in seguito.

5.6.1 Implementazioni dei DataSource

In questo livello sono presenti le implementazioni dei DataSource, che si occupano di interagire con le fonti di dati locale e remota per recuperare, memorizzare, modificare o eliminare dati. Anche in questo caso l'implementazione di ogni interfaccia di un DataSource mantiene il nome dell'interfaccia al quale viene aggiunto il suffisso *Impl*, perciò in questo progetto sono presenti *WeightLocalDataSourceImpl*, *WeightRemoteDataSourceImpl*, *MealLocalDataSourceImpl*, *MealRemoteDataSourceImpl*, *GoalLocalDataSourceImpl* e *GoalRemoteDataSourceImpl*. Poiché le funzioni che ogni DataSource implementa sono le stesse del Repository relativo alla stessa Entity, anche i DataSource tendono a somigliarsi tra loro e verranno, quindi, descritte in questo paragrafo soltanto le funzioni di GoalLocalDataSourceImpl e GoalRemoteDataSourceImpl, con riferimento ad eventuali corrispettivi negli altri DataSource.

GoalLocalDataSourceImpl

- *getGoals*, esegue una query sul database locale per ottenere la lista di tutti gli obiettivi; i suoi corrispettivi nei LocalDataSource dei pesi e dei pasti sono, rispettivamente, getWeights e getMeals

- *getGoal*, esegue una query sul database locale per ottenere l'obiettivo relativo alla categoria ricevuta in input; non ha corrispettivi negli altri LocalDataSource

- *addGoal*, esegue una query sul database locale per inserire un nuovo obiettivo del quale riceve tutti i parametri in input; i suoi corrispettivi nei LocalDataSource dei pesi e dei pasti sono, rispettivamente, addWeight e addMeal

- *addGoals*, esegue una query sul database locale per inserire una lista di obiettivi ricevuta in input; i suoi corrispettivi nei LocalDataSource dei pesi e dei pasti sono, rispettivamente, addWeights e addMeals

- *editGoal*, esegue una query sul database locale per modificare un obiettivo ricevendo in input i suoi parametri modificati; non ha corrispettivi negli altri LocalDataSource

- *deleteGoal*, esegue una query sul database locale per eliminare l'obiettivo relativo alla categoria ricevuta in input; non ha corrispettivi negli altri LocalDataSource

- *mapGoal*, funzione utilizzata per convertire la entry presente nel database locale in un oggetto di tipo GoalEntity; i suoi corrispettivi nei LocalDataSource dei pesi e dei pasti sono, rispettivamente, mapWeight e mapMeal

Come avviene per i Repository, l'unica funzione presente in un altro LocalDataSource, ma assente in quello degli obiettivi si trova nel MealLocalDataSource ed è *getMealsFromDay* che esegue una query sul database locale per ottenere la lista di pasti consumati nel giorno specificato.

Tutte le funzioni dei LocalDataSource che recuperano oggetti dal database ritornano tali oggetti incapsulati in dei Flow che, grazie all'ottima implementazione della classe da parte della libreria SQLDelight, permettono di utilizzare facilmente ed efficientemente dati sempre aggiornati.

Listato 5.40: Definizione della classe GoalLocalDataSourceImpl

```
class GoalLocalDataSourceImpl(
    private val database: KMMTestDatabase
) : GoalLocalDataSource {
    override fun getGoals() =
        database
            .kMMTestDatabaseQueries
            .selectAllGoals(::mapGoal)
            .asFlow()
            .mapToList()

    override fun getGoal(category: Int) =
        database
            .kMMTestDatabaseQueries
            .selectGoal(category, ::mapGoal)
            .asFlow()
            .mapToOneOrNull()

    override fun addGoal(
        category: Int,
        deadline: String,
        startingValue: Float,
        targetValue: Float
    ) =
        database.kMMTestDatabaseQueries.insertGoal(
            category,
            deadline,
            startingValue,
            targetValue
        )
```

```
    override fun addGoals(newGoals: List<GoalEntity>) =
        database.kMMTestDatabaseQueries.transaction {
            database.kMMTestDatabaseQueries.removeAllGoals()
            newGoals.forEach {
                database.kMMTestDatabaseQueries.insertGoal(
                    it.category,
                    it.deadline,
                    it.startingValue,
                    it.targetValue
                )
            }
        }
    override fun editGoal(
        category: Int,
        deadline: String,
        startingValue: Float,
        targetValue: Float
    ) =
        database.kMMTestDatabaseQueries.insertGoal(
            category,
            deadline,
            startingValue,
            targetValue
        )
    override fun deleteGoal(category: Int) =
        database.kMMTestDatabaseQueries.removeGoal(category)
    private fun mapGoal(
        category: Int,
        deadline: String,
        startingValue: Float,
        targetValue: Float
    ): GoalEntity {
        return GoalEntity(
            category = category,
            deadline = deadline,
            startingValue = startingValue,
            targetValue = targetValue
        )
    }
}
```

GoalRemoteDataSourceImpl

- *getGoals*, effettua una richiesta HTTP GET per ottenere dal server remoto la lista di tutti gli obiettivi; i suoi corrispettivi nei RemoteDataSource dei pesi e dei pasti sono, rispettivamente, getWeights e getMeals

- *addGoal*, effettua una richiesta HTTP POST inviando al server remoto i parametri di un nuovo obiettivo ricevuti in input; i suoi corrispettivi nei RemoteDataSource dei pesi e dei pasti sono, rispettivamente, addWeight e addMeal

- *editGoal*, effettua una richiesta HTTP POST inviando al server remoto i nuovi parametri di un obiettivo già esistente ricevuti in input; non ha corrispettivi negli altri RemoteDataSource

- *deleteGoal*, effettua una richiesta HTTP POST inviando la categoria di un obiettivo da eliminare ricevuta in input; non ha corrispettivi negli altri RemoteDataSource

Ad ogni RemoteDataSource è iniettato nel costruttore un HttpClient con il quale è possibile deserializzare gli oggetti JSON ricevuti dal server in oggetti Remote, che verranno discussi nel paragrafo 5.6.3 e, viceversa, di serializzare oggetti di questo tipo per inviarli come JSON al server. All'interno delle funzioni atte al recupero di dati da remoto, gli oggetti Remote ricevuti vengono convertiti nelle relative Entity grazie a dei mapper appositi, discussi anch'essi nel paragrafo 5.6.3.

```
class WeightRemoteDataSourceImpl(
    private val httpClient: HttpClient
): WeightRemoteDataSource {
    override suspend fun getWeights(): List<WeightEntity> =
        httpClient.get<List<WeightRemote>>(path = "weights").map {
            WeightEntityMapper.fromRemote(it)
        }

    override suspend fun addWeight(date: Long, weight: Float) {
        httpClient.post<HttpResponse>(path = "newWeight") {
            contentType(ContentType.Application.Json)
            body = WeightRemote(date = date, weight = weight)
        }
    }
}
```

Listato 5.41: Definizione della classe WeightRemoteDataSourceImpl

5.6.2 Cache

In questa sezione di Framework sono soltanto presenti le definizioni delle DatabaseDriverFactory necessarie per il corretto funzionamento della libreria SQLDelight. Come già discusso al paragrafo 4.2, poiché il Driver è creato differentemente sulle due piattaforme, è necessario utilizzare il meccanismo *expect/actual* fornito da Kotlin Multiplatform Mobile.

```
// codice comune
expect class DatabaseDriverFactory {
    fun createDriver(): SqlDriver
}

// codice specifico Android
actual class DatabaseDriverFactory(private val context: Context) {
    actual fun createDriver(): SqlDriver {
        return AndroidSqliteDriver(
            KMMTestDatabase.Schema,
            context,
            "kmmtest.db"
        )
    }
}

// codice specifico iOS
actual class DatabaseDriverFactory {
    actual fun createDriver(): SqlDriver {
        return NativeSqliteDriver(KMMTestDatabase.Schema, "kmmtest.db")
    }
}
```

Listato 5.42: Definizione della classe DatabaseDriverFactory

5.6.3 API

In quest'ultima sezione di Framework sono presenti le definizioni di classi e oggetti che consentono di deserializzare gli oggetti ricevuti in formato JSON dal server remoto.

Response insieme delle classi Remote, *WeightRemote*, *MealRemote* e *GoalRemote*, che consentono di deserializzare gli oggetti JSON ricevuti dal server remoto

Mapper insieme dei mapper, *WeightEntityMapper*, *MealEntityMapper* e *GoalEntityMapper*, che convertono gli oggetti Remote nelle corrispettive Entity

```
@Serializable
 data class GoalRemote(
     @SerialName("category")
     val category: Int,
     @SerialName("deadline")
     val deadline: String,
     @SerialName("startingValue")
     val startingValue: Float,
     @SerialName("targetValue")
     val targetValue: Float
 )
```

Listato 5.43: Definizione della classe GoalRemote

```
object GoalEntityMapper {
     fun fromRemote(remote: GoalRemote) =
         GoalEntity(
             category = remote.category,
             deadline = remote.deadline,
             startingValue = remote.startingValue,
             targetValue = remote.targetValue
         )

 }
```

Listato 5.44: Definizione del GoalEntityMapper

5.7 Modulo App - UI

Idealmente, scrivendo applicazioni multipiattaforma con Kotlin Multiplatform Mobile l'unico codice specifico per ogni piattaforma che sarebbe necessario scrivere è quello relativo alla composizione della UI, ma in questo caso, per essere maggiormente fedeli all'architettura scelta e rendere il codice più conforme al principio di separazione degli ambiti, è stato scelto di rendere specifica per ogni piattaforma l'intera porzione UI del modulo App. Per questo motivo anche l'implementazione dei ViewModel è differente sulle due piattaforme. Relativamente all'interfaccia, inoltre, è necessario specificare che è stata costruita principalmente per essere funzionale e consentire di testare in modo efficiente il funzionamento dell'applicazione, senza necessariamente seguire le best practice di composizione della UI e della User Experience.

5.7.1 Android

Per la piattaforma Android sono stati creati cinque ViewModel, tre per le schermate principali, *WeightsViewModel*, *MealsViewModel* e *GoalsViewModel*, uno per la schermata di aggiunta di un nuovo pasto, *AddMealViewModel*, ed uno utilizzato soltanto per aggiornare i dati all'avvio dell'applicazione, *MainViewModel*. Poiché il codice specifico per la piattaforma Android è scritto in Kotlin, è possibile, estendendo KoinComponent, iniettare direttamente all'interno dei ViewModel delle istanze degli UseCase.

Listato 5.45: Definizione del MainViewModel

```
class MainViewModel : ViewModel(), KoinComponent {
    private val loadWeightsUseCase: LoadWeightsUseCase by inject()
    private val loadMealsUseCase: LoadMealsUseCase by inject()
    private val loadGoalsUseCase: LoadGoalsUseCase by inject()
    private val cancelables = mutableListOf<Cancelable>()

    fun loadData() {
        cancelables.addAll(
            listOf(
                loadWeightsUseCase.loadWeights(),
                loadMealsUseCase.loadMeals(),
                loadGoalsUseCase.loadGoals()
            )
        )
    }
```

```
    override fun onCleared() {
        super.onCleared()
        cancelables.forEach {
            it.cancel()
        }
    }
}
```

Siccome, per i motivi descritti al paragrafo 5.2.1, è stato scelto di non propagare le coroutine, non è possibile legare i processi che vengono eseguiti mediante gli UseCase al ciclo di vita del ViewModel, rischiando di generare memory leak nel caso in cui questo venga distrutto prima che il processo termini. Per risolvere questo problema è stata implementata una soluzione non pulitissima, ma funzionale. Quando viene richiamata una funzione esposta da uno UseCase, il Cancelable restituito viene salvato in una lista creata all'inizializzazione del ViewModel con lo scopo di contenere i Cancelable di tutti i processi fatti partire. In tal modo è possibile, nel momento in cui il ViewModel venga distrutto, cancellare tutti i processi semplicemente richiamando la funzione *cancel* su ogni elemento della lista.

Sfruttando gli UseCase iniettati con Koin, ogni ViewModel recupera i dati necessari, li manipola per renderli pronti per essere mostrati all'utente e li espone alle classi del livello superiore come oggetti osservabili di tipo LiveData. I dati arrivano al ViewModel come oggetti di tipo CustomFlow che, ereditando tutte le funzioni dei Flow, potrebbero essere convertiti direttamente in LiveData utilizzando *asLiveData*, ma poiché questi vengono ricevuti all'interno delle callback passate in input alle funzioni degli UseCase, non è possibile salvare i LiveData così ottenuti in nuove variabili visibili nel resto del ViewModel. Per questo motivo è stata creata una extension function per i CustomFlow che si occupa di inserire, ad ogni suo aggiornamento, il valore del Flow in un LiveData fornitole in input.

```
fun <T> CustomFlow<T>.toLiveData(
    scope: CoroutineScope,
    liveData: MutableLiveData<T>
) = scope.launch { collect { liveData.postValue(it) } }
```

Listato 5.46: Definizione dell'extension function *toLiveData*

In molti casi non è sufficiente esporre i dati ottenuti, ma è necessario elaborarli e combinarli tra loro per ricavare esattamente i dati da mostrare. Utilizzando i LiveData è possibile farlo molto semplicemente. Nell'esempio sotto riportato, estratto dal ViewModel della schermata dei pesi, vengono esposti, tra gli altri, il valore del peso attuale come stringa e il progresso dell'obiettivo creato dall'utente, funzione del peso attuale, del peso iniziale e del peso che si vuole raggiungere.

```
private var _weights = MutableLiveData<List<WeightEntity>>()
private val _weightGoal = MutableLiveData<GoalEntity?>()
private val cancelables = mutableListOf(
    getWeightsUseCase.getWeights {
        if (it.status == Status.SUCCESS)
            it.value?.toLiveData(viewModelScope, _weights)
    },
    getGoalUseCase.getGoal(0) {
        if (it.status == Status.SUCCESS)
            it.value?.toLiveData(viewModelScope, _weightGoal)
    }
)
private val _actualWeight = _weights.map { it.maxByOrNull {it.date}?.weight }
private val _startingWeight = _weightGoal.map { it?.startingValue }
private val _targetWeight = _weightGoal.map { it?.targetValue }

val actualWeight: LiveData<String?>
    get() = _actualWeight.map { it?.toRoundedString(2) }
val startingWeight: LiveData<String>
    get() = _startingWeight.map { it?.toRoundedString(1) ?: "" }
val targetWeight: LiveData<String>
    get() = _targetWeight.map { it?.toRoundedString(1) ?: "" }
val difference =
    _startingWeight.combineWith(_actualWeight) { startingValue, actualValue ->
        if (actualValue != null && startingValue != null)
            actualValue - startingValue
        else
            0f
    }
val progress = _actualWeight.combineWith(_weightGoal) { actualValue, g ->
    if (actualValue != null && g != null)
        (actualValue - g.startingValue) / (g.targetValue - g.startingValue)
    else
        0f
}

fun Float.toRoundedString(decimalDigits: Int) =
((this*10f.pow(decimalDigits)).roundToInt()/10f.pow(decimalDigits)).toString()
```

Listato 5.47: Estratto del WeightsViewModel

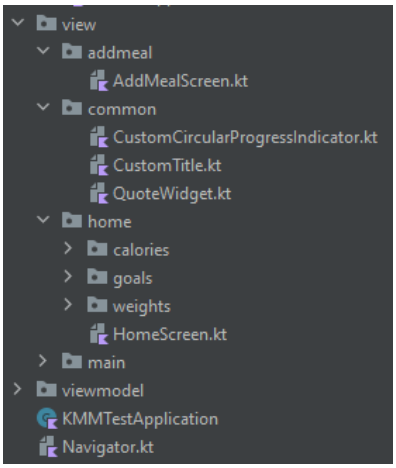

Figura 5.2: Struttura del package view

L'interfaccia vera e propria dell'applicazione è gestita da poche classi inserite nel package denominato *view*. La MainActivity contiene le schermate create con Jetpack Compose e si occupa solo di far partire l'aggiornamento dei dati usando l'apposita funzione esposta da MainViewModel e di caricare il *Navigator*, una funzione composable che crea il NavController ed imposta il NavHost. La navigazione è piuttosto semplice, dato che le uniche due schermate presenti nell'applicazione sono la schermata iniziale e quella di aggiunta di un nuovo pasto. Non sono presenti schermate di aggiunta di un nuovo peso e di un nuovo obiettivo in quanto queste funzionalità sono implementate tramite la comparsa di una dialog.

```
@Composable
fun Navigator() {
    val navController = rememberNavController()
    NavHost(
        navController = navController,
        startDestination = "homeScreen",
        builder = {
            composable("homeScreen") {
                HomeScreen(navController)
            }
            composable("addMealScreen") {
                AddMealScreen(navController)
            }
        }
    )
}
```

Listato 5.48: Definizione del Navigator

Dalla schermata principale è possibile spostarsi lateralmente per visualizzare le schermate dei pesi, dei pasti e degli obiettivi. Ogni schermata è definita come una funzione composable al cui interno vengono prima osservati come stati tutti gli oggetti esposti dal proprio ViewModel e poi descritti gli elementi dell'interfaccia, che sono a loro volta funzioni composable. In questa tesi non si andrà molto nel

dettaglio sulla creazione di funzioni composable complesse in quanto poco inerente allo studio di Kotlin Multiplatform Mobile, ma di seguito è riportato un esempio di schermata costruita in Compose, *GoalsPage*. Al suo interno è possibile notare la semplicità e la chiarezza del linguaggio dichiarativo e sono presenti anche delle funzioni composable create ad hoc come *GoalCard* e *NewGoalDialog*. Quest'ultima, in particolare rappresenta la dialog di aggiunta di un nuovo obiettivo, e mostra una delle grandi potenzialità del linguaggio dichiarativo. La sua comparsa è infatti regolata semplicemente da una condizione *if* legata ad un booleano fornito in input a GoalsPage quando questa viene richiamata dalla schermata principale. Il booleano può, così, essere modificato da elementi presenti in quest'ultima e può essere passato ad una qualunque funzione composable, rendendo possibile gestire la comparsa di una dialog differente dipendentemente dalla schermata mostrata.

Listato 5.49: Definizione di GoalsPage

```
@Composable
fun GoalsPage(
    showNewGoalDialog: Boolean = false,
    closeDialog: () -> Unit = {}
) {

    val goalsViewModel = getViewModel<GoalsViewModel>()
    val goals = goalsViewModel.goals.observeAsState()

    if (goals.value.orEmpty().isEmpty())
        Box(
            modifier = Modifier
                .fillMaxSize()
                .padding(horizontal = 20.dp),
            contentAlignment = Alignment.Center
        ) {

            Text(
                modifier = Modifier.padding(),
                text = "Add a goal to see it in this page.",
                fontSize = 30.sp,
                textAlign = TextAlign.Center
            )

        }
```

```
    LazyColumn(modifier = Modifier.padding(horizontal = 20.dp)) {
        items(
            count = goals.value.orEmpty().size,
            itemContent = { index ->
                GoalCard(
                    goal = goals.value.orEmpty()[index],
                    editGoal = { category, deadline, startingValue,
                    ↪  targetValue ->
                        goalsViewModel.editGoal(
                            category,
                            deadline,
                            startingValue,
                            targetValue
                        )
                        closeDialog.invoke()
                    },
                    deleteGoal = {
                        goalsViewModel.deleteGoal(it)
                        closeDialog.invoke()
                    }
                )
            }
        )
    }

    if (showNewGoalDialog)
        NewGoalDialog(
            addGoal = {
                    category, deadline, startingValue, targetValue ->
                goalsViewModel.addGoal(
                    category,
                    deadline,
                    startingValue,
                    targetValue
                )
                closeDialog.invoke()
            },
            closeDialog = closeDialog
        )
}
```

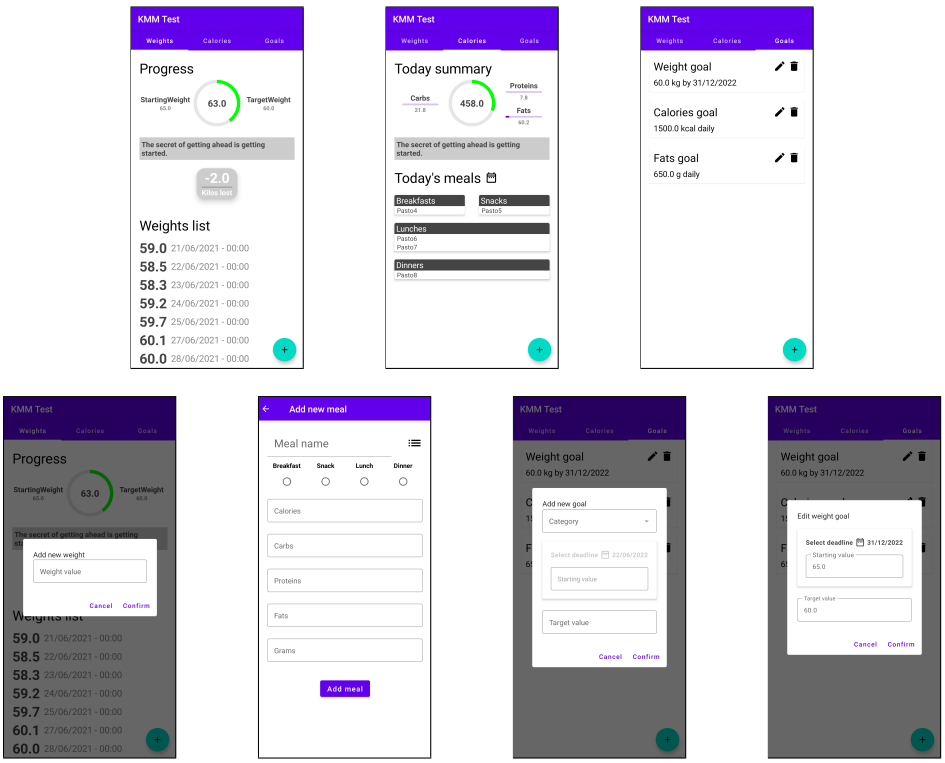

Figura 5.3: Schermate dell'applicazione su Android

5.7.2 iOS

Anche per la piattaforma iOS sono stati creati tre ViewModel per le schermate principali, *ObservableWeightsModel*, *ObservableMealsModel* e *ObservableGoalsModel* ed uno utilizzato per aggiornare i dati all'avvio dell'applicazione, *ObservableMainModel*, ma non è stato necessario crearne uno anche per l'aggiunta di un nuovo pasto in quanto è stato possibile implementare la funzione in quello dei pasti. Non potendo utilizzare direttamente la dependency injection di Koin come nel codice per la pittaforma Android, è stato necessario creare una classe che facesse da ponte tra il codice condiviso e quello specifico della piattaforma, che è stata denominata *AppSDK*. AppSDK, che estende KoinComponent, espone delle semplici funzioni che restituiscono delle istanze degli UseCase ottenute iniettandole con Koin.

```
fun getGoalsIos(): GetGoalsUseCase {
    val getGoalsUseCase: GetGoalsUseCase by inject()
    return getGoalsUseCase
}
```

Listato 5.50: Definizione di una delle funzioni esposte da AppSDK

Per ottenere uno UseCase all'interno di un ViewModel sarà, quindi, sufficiente creare un oggetto di tipo AppSDK e richiamare la funzione relativa allo UseCase. La gestione degli oggetti Cancelable legati ai processi fatti partire è similare a quella effettuata su piattaforma Android, ogni volta che la funzione di uno UseCase viene richiamata, il Cancelable restituito viene memorizzato in un vettore e quando il ViewModel viene distrutto viene richiamata la funzione *cancel* su ognuno degli oggetti presenti al suo interno. Similare è anche il modo in cui i dati vengono esposti alle classi del livello superiore. I dati ricevuti nelle callback passate agli UseCase, infatti, vengono manipolati e salvati in delle variabili pubbliche. Tali variabili devono essere contrassegnate dal property wrapper *@Published*, così che qualunque loro modifica generi un aggiornamento delle view che osservano il ViewModel.

Listato 5.51: Struttura dell'ObservableWeightsModel

```
class ObservableWeightsModel: ObservableObject {
    private let appSdk = AppSdk()
    private var cancelables: [Cancelable] = []

    @Published var product = "Started"
    @Published var startingWeight = 0.0
    @Published var targetWeight = 0.0
    @Published var actualWeight = 0.0
    @Published var progress = 0.0
    @Published var weights: [WeightEntity] = []

    init() {
        cancelables.append(contentsOf: [
            appSdk.getWeightsIos().getWeights() { result in
                result.value?.watch { weights in
                    if let weightsArray = weights as? [WeightEntity] {
                        self.weights = weightsArray
                        if let value = weightsArray.last?.weight {
                            self.actualWeight = Double(value)
                            self.updateProgress()
                        }
                    }
                }
            },
```

```
            appSdk.getGoalIos().getGoal(category: 0) { result in
                result.value?.watch { goal in
                    if let goal = goal {
                        self.startingWeight = Double(goal.startingValue)
                        self.targetWeight = Double(goal.targetValue)
                        if (self.actualWeight != 0.0) {
                            self.updateProgress()
                        }
                    }
                }
            }
        ])
    }

    private func updateProgress() {
        progress = (actualWeight - startingWeight)/(targetWeight -
        ↪  startingWeight)
    }

    func cancel() {
        for cancelable in cancelables {
            cancelable.cancel()
        }
    }
}
```

Similmente alla piattaforma Android, anche in iOS l'interfaccia è composta usando un linguaggio dichiarativo, fornito dal framework SwiftUI, semplificando di molto la scrittura e consentendo di fare tutto usando solo poche strutture. La schermata principale è gestita dalla ContentView che, analogamente a quanto descritto nel paragrafo 4.5.2, incapsula in una NavigationView una TabView che consente di spostarsi tra le schermate di pesi, pasti ed obiettivi. Differentemente da quanto fatto per piattaforma Android, le view di aggiunta di nuovi pesi, pasti od obiettivi non sono mostrate in schermate o dialog, ma utilizzando delle *sheet*. Anche l'interfaccia costruita per iOS non verrà discussa nel dettaglio, ma nell'esempio sottostante è possibile notare quanto la struttura di una view composta in SwiftUI sia molto simile a quella di una funzione composable.

```
struct WeightsView: View {
    @ObservedObject var vm = ObservableWeightsModel()

    var body: some View {
        ScrollView{
            LazyVStack(
                alignment: .leading,
                spacing: 10
            ) {
                HStack(spacing: 20) {
                    WeightProgress(
                        progressText: "Starting weight",
                        progressValue: vm.startingWeight
                    )
                        .frame(maxWidth: .infinity)
                    CustomCircularProgressBar(
                        progress: vm.progress,
                        color: Color.green,
                        value: vm.actualWeight
                    )
                        .frame(maxWidth: .infinity)
                    WeightProgress(
                        progressText: "Target weight",
                        progressValue: vm.targetWeight
                    )
                        .frame(maxWidth: .infinity)
                }
                WeightsList(weights: vm.weights)
            }
            .navigationBarTitle("Weights")
            .padding(20)
        }
    }
}
```

Listato 5.52: Definizione di WeightsView

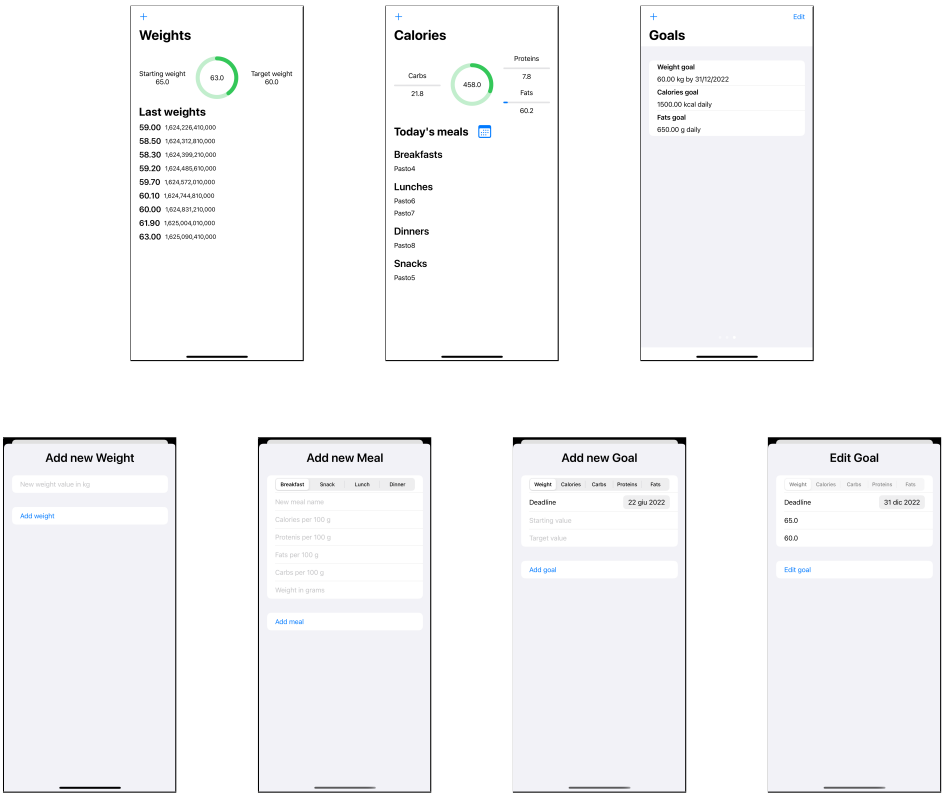

Figura 5.4: Schermate dell'applicazione su iOS

Capitolo 6

Conclusioni

Con questo progetto è stato possibile constatare come Kotlin Multiplatform Mobile sia sicuramente uno strumento molto potente, ma con ancora delle imperfezioni che rendono necessari alcuni piccoli sacrifici.

Sviluppare l'applicazione in Kotlin Multiplatform Mobile è stato, per la maggior parte, simile a sviluppare nativamente per Android in Kotlin, sia per il linguaggio utilizzato che per l'architettura. Il pattern Model-View-ViewModel è già uno dei più diffusi nello sviluppo per il sistema operativo di Google e, unito ai principi della Clean Architecture, risulta essere ideale per progetti cross-platform con questo framework. Il principio di separazione degli ambiti ed il principio di inversione garantiscono, infatti, che il codice specifico per le piattaforme sia indipendente da quello contenente le logiche di business, che si trova ad un livello più basso. L'utilizzo di un'architettura tale rende anche i moduli del codice condiviso completamente testabili con JUnit, permettendo, così, di verificare la stabilità dell'applicazione e che ogni unità di sviluppo rispetti i requisiti ed assolva le sue funzioni.

Sia funzioni semplici che più complesse possono, poi, essere implementate facilmente anche grazie alle numerose librerie compatibili con il framework. Tutte le librerie utilizzate per il progetto di questa tesi sono già mature ed offrono tutte le funzionalità necessarie per lo sviluppo di un'applicazione mobile. Oltre a quelle qui menzionate ne esistono anche molte altre che rispondono ad ogni tipo di esigenza e che sono già molto utilizzate.

L'interoperabilità fornita da Kotlin/Native è buona, converte efficacemente molti degli elementi di Kotlin in Swift e viceversa permettendo di passare da un linguaggio all'altro senza soluzione di continuità. È vero che per alcuni elementi di Kotlin molto utili ed utilizzati non viene fornito un corrispettivo in Swift, in particolare Flow e LiveData, ma è possibile creare abbastanza facilmente classi wrapper più o meno complesse a seconda delle esigenze per sopperire a tale mancanza.

L'unico aspetto un po' più complesso del framework è l'implementazione di processi multithread. Come già discusso, a causa della differente gestione della

concorrenza delle due piattaforme, non è possibile utilizzare le coroutine come si farebbe sviluppando nativamente in Kotlin, ma si è limitati dall'impossibilità di usare dispatcher specifici, ad esempio il dispatcher dedicato all'IO, e condividere oggetti tra thread in background a meno che non siano stati resi immutabili con la funzione *freeze*. Per questo motivo in alcuni casi è necessario ingegnarsi per trovare il modo migliore per implementare processi più complessi senza rischiare memory leak od incappare in errori.

Ad ogni modo, al netto di inevitabili difetti, Kotlin Multiplatform Mobile è già un framework sufficientemente maturo da permettere la creazione di applicazioni cross-platform ricche, funzionali e stabili. Utilizzarlo è decisamente immediato per chi conosce già Kotlin, essendo la maggior parte dell'applicazione scritta in questo linguaggio, e ha già dimestichezza con lo sviluppo mobile, anche se, specialmente per progetti personali, risulta necessario conoscere anche Swift, linguaggio comunque non molto dissimile da Kotlin e, perciò, abbastanza semplice da imparare per chi conosce già quest'ultimo. Viceversa, chi conosce già Swift, ma non Kotlin, dovrà abituarsi ad utilizzare per lo più il linguaggio sviluppato da JetBrains, ma non dovrebbe comunque avere difficoltà ad imparare ad utilizzarlo.

6.1 Utilizzo in produzione

Per valutare se possa essere utilizzato in produzione da un'azienda, però, occorre valutare anche altri aspetti oltre a quello tecnico, principalmente l'aspetto economico e di risorse umane.

I principali problemi di Kotlin Multiplatform Mobile, riscontrati anche in questo progetto di tesi, sono facilmente risolvibili con l'utilizzo di un'architettura dedicata e di classi e funzioni create appositamente per rendere lo sviluppo più semplice ed uniformare la struttura delle applicazioni, processo tendenzialmente già messo in atto dalle aziende che sviluppano per mobile. La natura del framework, poi, rende semplice sia la creazione di nuove applicazioni che la conversione di progetti già esistenti di applicazioni native in Kotlin, a patto che presentino un'architettura che rispetti i principi di separazione degli ambiti e inversione delle dipendenze. In casi del genere, infatti, basterebbe soltanto apportare qualche piccola modifica al codice delle logiche di business per poter aggiungere anche il modulo di UI per la piattaforma iOS.

L'ottima implementazione delle funzionalità native permette di condividere buona parte del codice, migliorando la testabilità e riducendo il lavoro necessario sia per lo sviluppo che per il mantenimento. Ciò rende lo sviluppo di applicazioni cross-platform con Kotlin Multiplatform Mobile notevolmente più efficiente dello sviluppo di due distinte applicazioni native. Anche rispetto ad altri framework

per lo sviluppo multipiattaforma risultano esserci diversi vantaggi. In primis, poiché tutte le funzionalità native sono supportate ed utilizzabili in Kotlin, non vi è necessità di integrare parti del codice comune con codice nativo. Il linguaggio utilizzato ed il compiler Kotlin/Native permettono, anche se, come detto, con qualche accorgimento, di sfruttare la natura multithread di entrambe le piattaforme. In ultimo, ma non per importanza, la possibilità di comporre la UI nativamente sulle due piattaforme elimina le limitazioni che sarebbero inevitabilmente presenti nel creare un'interfaccia comune tra le due applicazioni.

Paradossalmente, il problema principale dell'utilizzo di Kotlin Multiplatform Mobile in produzione potrebbe non essere di natura tecnica. In un progetto con Kotlin Multiplatform Mobile sono, infatti, necessarie sia conoscenze di Kotlin che di Swift, perciò il vero problema potrebbe sorgere nella composizione del team di sviluppo. Se si volesse comporre il team in base alla quantità di codice di un linguaggio o dell'altro ci sarebbero molti più sviluppatori Kotlin che sviluppatori Swift ed ogni sviluppatore potrebbe lavorare solo sulla parte di sua competenza. La logica soluzione è che tutti i componenti del team conoscano almeno un po' entrambi i linguaggi, con qualche sviluppatore esperto di Kotlin e qualche altro esperto di Swift. Il problema di tale soluzione è che sarebbe, quindi, necessario che tutti gli sviluppatori venissero istruiti sul linguaggio non di loro competenza, e questo per un'azienda ha un costo, oltre al fatto che, nonostante imparare un linguaggio conoscendo già l'altro non sia molto difficile, gli sviluppatori dovrebbero comunque imparare un nuovo linguaggio. Per un'azienda che deve decidere se usare Kotlin Multiplatform Mobile in produzione c'è, perciò, da valutare principalmente se ci sia la possibilità di formare un team adeguato e se il costo della formazione dei propri dipendenti possa valere il risparmio ottenuto utilizzando il framework.

6.2 Il futuro di Kotlin Multiplatform Mobile

Essendo ancora in Alpha, nel prossimo futuro Kotlin Multiplatform Mobile riceverà sicuramente degli aggiornamenti con correzioni e nuove funzionalità, alcune delle quali sono anche già state annunciate. Entro la fine del 2022 dovrebbe essere rilasciata la Beta che, tra le altre migliorie tecniche, dovrebbe contenere anche un nuovo gestore della memoria per Kotlin/Native. Il nuovo gestore eliminerà le restrizioni sulla condivisione di oggetti tra thread e fornirà delle nuove primitive di programmazione concorrente sicure che impediranno memory leak. Ciò renderà sicuramente più semplice la gestione di chiamate asincrone ed il loro passaggio dal codice comune al codice specifico della piattaforma, in particolare quello per iOS, scritto in Swift. Le nuove versioni delle librerie kotlinxinx.coroutines e Ktor, inoltre, permetteranno di lanciare le coroutine usando un qualunque dispatcher

multithreaded anche senza applicare la funzione *freeze* sugli oggetti che si trovano in thread in background.

Oltre a questi aggiornamenti certi, una aggiunta di cui sicuramente il framework beneficerebbe potrebbe essere un metodo per spostare il livello ViewModel nel codice comune dell'applicazione in modo da lasciare che soltanto il codice della UI sia specifico per la piattaforma. Per farlo sarebbe necessario trovare il modo di far esporre ai ViewModel degli oggetti osservabili sia in Kotlin che in Swift in modo da aggiornare le interfacce quando questi subiscono modifiche. Tali oggetti si dovrebbero quindi comportare come Observable in Kotlin e come proprietà *@Published* in Swift. Probabilmente sarebbe anche necessario che il ViewModel venga identificato come un oggetto che estende il protocollo *@ObservableObject* dal codice specifico per la piattaforma iOS. Il progetto è supportato da una grande community per cui sono già presenti diverse proposte di architetture alternative per rendere condiviso il ViewModel[72], ma una soluzione come quella sopra descritta sarebbe probabilmente più efficiente e di semplice utilizzo.

Allo stato attuale, Kotlin Multiplatform Mobile, anche se con ancora qualche difetto, è già un ottimo framework, utilizzato e supportato e con grandi potenzialità. Con gli aggiornamenti già annunciati e gli ulteriori possibili miglioramenti proposti dalla community si stanno gettando le basi affinché possa diventare uno dei framework di riferimento per lo sviluppo di applicazioni mobile cross-platform. Anche se lo si ritenesse ancora poco maturo per essere utilizzato per costruire un progetto in questo momento, cominciare a conoscerlo già da ora potrebbe rivelarsi molto utile per il futuro per chiunque voglia sviluppare per dispositivi mobile.

Bibliografia

[1] StatCounter. *Mobile and tablet internet usage exceeds desktop for first time worldwide*. Nov. 2016. URL: https://gs.statcounter.com/press/mobile-and-tablet-internet-usage-exceeds-desktop-for-first-time-worldwide (visitato il 12/07/2022) (cit. a p. 1).

[2] StatCounter. *Desktop vs Mobile Market Share Worldwide (Jan 2009 - May 2022)*. URL: https://gs.statcounter.com/platform-market-share/desktop-mobile/worldwide/#monthly-200901-202205 (visitato il 12/07/2022) (cit. a p. 1).

[3] Psion, cur. *Psion Organiser Brochure*. Brochure. 1984. URL: https://www.jaapsch.net/psion/pdffiles/p1brochure.pdf (visitato il 12/07/2022) (cit. a p. 1).

[4] Dick Pountain. «A plethora of portables». In: *Byte Magazine* (nov. 1984). URL: https://archive.org/details/byte-magazine-1984-11/page/n413 (visitato il 12/07/2022) (cit. a p. 1).

[5] Steve Litchfield. *The History of Psion*. 2005. URL: https://stevelitchfield.com/historyofpsion.htm (visitato il 12/07/2022) (cit. a p. 1).

[6] Jaap Scherphuis. *Third party Datapaks for Psion Organiser II*. URL: https://www.jaapsch.net/psion/galpacks.htm#thirdparty (visitato il 12/07/2022) (cit. a p. 2).

[7] Raphael Needleman. «Psion: It doesn't get any more portable». In: *InfoWorld* (ott. 1991). URL: https://archive.org/details/bub_gb_dT0EAAAAMBAJ/page/n81 (visitato il 12/07/2022) (cit. a p. 2).

[8] Russell Ito. «Newton's World». In: *MacUser* (ago. 1992). URL: https://archive.org/details/MacUser9208August1992/page/n45 (visitato il 12/07/2022) (cit. a p. 2).

[9] Daniel Holden. «Personal Devices Pick Up Steam Across Industry». In: *Electronic News* (lug. 1993). URL: https://archive.org/details/sim_electronic-news_1993-06-07_39_1966 (visitato il 12/07/2022) (cit. a p. 2).

[10] Gerardo Greco. «Personal Digital Assistant: la tecnologia Newton di Apple al CES di Chicago». In: *MC Microcomputer* (lug. 1992). URL: https://archive.org/details/MC_microcomputer-120/page/n79 (visitato il 12/07/2022) (cit. a p. 2).

[11] Psion. *Psion Series 3a Programming Reference*. English. Ver. 1.0. Psion. Ago. 1993. 216 pp. URL: http://basic.hopto.org/basic/manual/PSION%203a%20programming%20Reference.pdf (visitato il 12/07/2022) (cit. a p. 2).

[12] Adrian Yacub, Cheryl Chambers e Christopher Bey. *The NewtonScript Programming Language*. English. Apple Comupter, Inc. 1996. 109 pp. URL: https://web.archive.org/web/20150124193723/http://manuals.info.apple.com/MANUALS/1000/MA1508/en_US/NewtonScriptProgramLanguage.PDF (visitato il 12/07/2022) (cit. a p. 2).

[13] Prosoft. *Psion Series 3 games*. URL: https://www.fogma.co.uk/prosoft/series3/classic/s3_classic_games.html (visitato il 12/07/2022) (cit. a p. 2).

[14] Sprightly. *Psion Series 3 miscellaneous software*. URL: http://www.sprightly.co.uk/psion3/bitty.html (visitato il 12/07/2022) (cit. a p. 2).

[15] Shawn Barnett. «Jeff Hawkins, The man who almost single-handedly revived the handheld computer industry». In: *Pen Computing* (apr. 2000). URL: https://www.pencomputing.com/palm/Pen33/hawkins1.html (visitato il 12/07/2022) (cit. a p. 2).

[16] Esther V. V. Reed. *Software Development for an Embedded System*. Rapp. tecn. Master's Project Report. Dept. of Computer Science e Engineering, Michigan State University, 1999. URL: https://www.cse.msu.edu/~reedevv/msproj/paper/index.html (visitato il 12/07/2022) (cit. a p. 2).

[17] Walter S. Mossberg. «The Palm Pilot Has Rivals But No True Competition». In: *Wall Street Journal* (lug. 1998). URL: https://www.wsj.com/articles/SB899329954715175000 (visitato il 12/07/2022) (cit. a p. 3).

[18] TascalSoft. *Windows CE softwares written by TascalSoft*. URL: https://web.archive.org/web/20000818031204/http://www2r.biglobe.ne.jp/~tascal/download/wce/index_e.htm (visitato il 12/07/2022) (cit. a p. 3).

[19] *Nokia 6110*. URL: https://en.wikipedia.org/wiki/Nokia_6110 (visitato il 12/07/2022) (cit. a p. 3).

[20] Bohdan Dovhaliuk. *History of Symbian OS, Symbian 6.0 - 6.1*. Ott. 2021. URL: https://smart2000s.com/2021/10/10/history-of-symbian-os/#2-1-1 (visitato il 12/07/2022) (cit. a p. 3).

[21] Edward J. Correia. «Embedded Market Comes to Fore at JavaOne». In: *SD Times* (lug. 2001). URL: https://archive.org/details/sdtimes033/page/n31 (visitato il 12/07/2022) (cit. a p. 3).

[22] Norris Parker Smith. «Psion? Who's that? Ericsson, Nokia and Motorola like it». In: *HPCwire* (lug. 1998). URL: https://www.hpcwire.com/1998/07/03/psion-whos-ericsson-nokia-motorola-like (visitato il 12/07/2022) (cit. a p. 3).

[23] *Java Platform, Micro Edition (Java ME)*. URL: https://www.oracle.com/java/technologies/javameoverview.html (visitato il 12/07/2022) (cit. a p. 4).

[24] Mark VandenBrink. *JSR 37: Mobile Information Device Profile for the J2METM Platform*. Set. 1999. URL: https://www.jcp.org/en/jsr/detail?id=37 (visitato il 12/07/2022) (cit. a p. 4).

[25] *Cattura del 2001 della pagina di introduzione a MIDP for Palm OS*. URL: https://web.archive.org/web/20011110111418/http://java.sun.com/products/midp4palm (visitato il 12/07/2022) (cit. a p. 4).

[26] *Cattura del 2001 della pagina di introduzione a CrEme*. URL: https://web.archive.org/web/20010206121506/http://www.nsicom.com/products/creme.asp (visitato il 12/07/2022) (cit. a p. 4).

[27] *Cattura del 2005 della pagina di introduzione a MySaifu*. URL: https://web.archive.org/web/20051001074848/http://www2s.biglobe.ne.jp/~dat/java/project/jvm/index_en.html (visitato il 12/07/2022) (cit. a p. 4).

[28] Tom Krazit. «PDA market continues to decline». In: *ComputerWeekly* (ott. 2004). URL: https://www.computerweekly.com/news/2240058478/PDA-market-continues-to-decline (visitato il 12/07/2022) (cit. a p. 5).

[29] *Cattura del 2005 dell'aggregatore di applicativi J2ME GetJar*. URL: https://web.archive.org/web/20050129040151/http://www.getjar.com/software (visitato il 12/07/2022) (cit. a p. 5).

[30] *Cattura del 2003 dell'aggregatore di applicativi per Symbian OS SearchAmateur*. URL: https://web.archive.org/web/20031212053329/http://www.searchamateur.com:80/Symbian-OS-Software/Symbian-Games.htm (visitato il 12/07/2022) (cit. a p. 5).

[31] Chris Wright. *A Brief History of Mobile Games: 2002 - Wake up and smell the coffee*. Mar. 2016. URL: https://www.pocketgamer.biz/feature/10705/a-brief-history-of-mobile-games-2002-wake-up-and-smell-the-coffee (visitato il 12/07/2022) (cit. a p. 5).

[32] *Presentazione del primo iPhone*. URL: https://youtu.be/MnrJzXM7a6o (visitato il 12/07/2022) (cit. a p. 5).

[33] Jade Charles. «iPhone owns 28 percent of US smartphone market». In: *Ars Technica* (mag. 2008). URL: https://arstechnica.com/gadgets/2008/02/iphone-owns-28-percent-of-us-smartphone-market (visitato il 12/07/2022) (cit. a p. 5).

[34] *Apple annuncia la beta software iPhone 2.0*. Mar. 2008. URL: https://www.apple.com/it/newsroom/2008/03/06Apple-Announces-iPhone-2-0-Software-Beta (visitato il 12/07/2022) (cit. a p. 5).

[35] *Cattura dell'annuncio della creazione di Nokia Catalogs, poi rinominato Nokia Download!* URL: https://web.archive.org/web/20061209073944/http://www.nokia.com/A4228134 (visitato il 12/07/2022) (cit. a p. 6).

[36] Will Park. «Nokia launches mobile social network – MOSH by Nokia!» In: *IntoMobile* (ago. 2007). URL: https://www.intomobile.com/2007/08/09/nokia-launches-mobile-social-network-mosh-by-nokia (visitato il 12/07/2022) (cit. a p. 6).

[37] Katerina Zolotareva. «Infographic: The Evolution (History) of The App Stores». In: *TheTool* (ago. 2017). URL: https://thetool.io/2017/evolution-app-stores-infographic (visitato il 12/07/2022) (cit. a p. 6).

[38] Roberto Colombo. «Nokia attiva Ovi Store a livello mondiale e in Italia». In: *Hardware Upgrade* (mag. 2009). URL: https://www.hwupgrade.it/news/telefonia/nokia-attiva-ovi-store-a-livello-mondiale-e-in-italia_29102.html (visitato il 12/07/2022) (cit. a p. 6).

[39] Nilay Patel. «Microsoft announces Windows Marketplace and My Phone for Windows Mobile». In: *Engadget* (feb. 2009). URL: https://www.engadget.com/2009-02-16-microsoft-announces-windows-marketplace-for-windows-mobile.html (visitato il 12/07/2022) (cit. a p. 6).

[40] Priya Ganapati. «Palm Pre App Catalog Makes a Slow Start». In: *Wired* (mag. 2009). URL: https://www.wired.com/2009/06/palm-pre-apps (visitato il 12/07/2022) (cit. a p. 6).

[41] Jessica Dolcourt. «BlackBerry App World has landed». In: *CNET* (apr. 2009). URL: https://www.cnet.com/culture/blackberry-app-world-has-landed (visitato il 12/07/2022) (cit. a p. 6).

[42] Joel West e David Wood. «Evolving an Open Ecosystem: The Rise and Fall of the Symbian Platform». In: vol. 30. Lug. 2013, pp. 27–67. ISBN: 978-1-78190-826-6. DOI: 10.1108/S0742-3322(2013)0000030005 (cit. a p. 6).

[43] Jemima Kiss. «Nokia buys Symbian in web push». In: *The Guardian* (giu. 2008). URL: https://www.theguardian.com/media/2008/jun/24/digitalmedia.mediabusiness (visitato il 12/07/2022) (cit. a p. 6).

[44] Priya Ganapati. «HTC Dream Coming October 20». In: *Wired* (set. 2008). URL: https://www.wired.com/2008/09/htc-dream-comin (visitato il 12/07/2022) (cit. a p. 6).

[45] StatCounter. *Mobile Operating System Market Share Worldwide (Jan 2009 - Dec 2014)*. URL: https://gs.statcounter.com/os-market-share/mobile/worldwide/#monthly-200901-201412 (visitato il 12/07/2022) (cit. a p. 6).

[46] Chris King. «Getting Started with the BlackBerry Mobile Development Platform». In: *Developer.com* (lug. 2010). URL: https://www.developer.com/mobile/java-me/getting-started-with-the-blackberry-mobile-development-platform (visitato il 12/07/2022) (cit. a p. 7).

[47] *Cattura del 2009 della home page del sito di PhoneGap*. URL: https://web.archive.org/web/20090416065604/http://phonegap.com (visitato il 12/07/2022) (cit. a p. 7).

[48] Stuart Dredge. «Google disables Microsoft's Windows Phone YouTube app (again)». In: *The Guardian* (ago. 2013). URL: https://www.theguardian.com/technology/appsblog/2013/aug/15/google-disables-windows-phone-youtube-app (visitato il 12/07/2022) (cit. a p. 8).

[49] Sean Endicott. «Windows Phone Store shuts down today, Windows 10 Mobile support extended to January». In: *Windows Central* (dic. 2019). URL: https://www.windowscentral.com/window-phone-store-shuts-down-today-windows-10-mobile-supported-extended-january (visitato il 12/07/2022) (cit. a p. 8).

[50] *LineageOS*. URL: https://lineageos.org (visitato il 12/07/2022) (cit. a p. 9).

[51] *Java*. URL: https://www.java.com/it (visitato il 12/07/2022) (cit. a p. 15).

[52] *Kotlin*. URL: https://kotlinlang.org (visitato il 12/07/2022) (cit. a p. 15).

[53] Jordan Rose. «ABI Stability and More». In: (feb. 2019). URL: https://www.swift.org/blog/abi-stability-and-more (visitato il 12/07/2022) (cit. a p. 26).

[54] *Flutter*. URL: https://flutter.dev (visitato il 12/07/2022) (cit. a p. 29).

[55] *Material Design*. URL: https://material.io/design (visitato il 12/07/2022) (cit. a p. 29).

[56] *React Native*. URL: https://reactnative.dev (visitato il 12/07/2022) (cit. a p. 30).

[57] *Xamarin*. URL: https://dotnet.microsoft.com/en-us/apps/xamarin (visitato il 12/07/2022) (cit. a p. 30).

[58] *Kotlin Multiplatform Mobile*. URL: https://kotlinlang.org/lp/mobile/ (visitato il 12/07/2022) (cit. a p. 31).

[59] *Target di Kotlin Multiplatform*. URL: https://kotlinlang.org/docs/multiplatform-dsl-reference.html#targets (visitato il 12/07/2022) (cit. a p. 37).

[60] *Open Food Facts Open Database*. URL: https://world.openfoodfacts.org (visitato il 12/07/2022) (cit. a p. 44).

[61] Robert C. Martin. *The Clean Architecture*. Ago. 2012. URL: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html (visitato il 12/07/2022) (cit. a p. 45).

[62] *LiveData Overview*. URL: https://developer.android.com/topic/libraries/architecture/livedata (visitato il 12/07/2022) (cit. a p. 48).

[63] *Flow*. URL: https://developer.android.com/kotlin/flow (visitato il 12/07/2022) (cit. a p. 48).

[64] R.C. Martin, J.M. Rabaey, A.P. Chandrakasan, J.W. Newkirk, B. Nikolić e R.S. Koss. *Agile Software Development: Principles, Patterns, and Practices*. Alan Apt series. Pearson Education, 2003. ISBN: 9780135974445. URL: https://books.google.it/books?id=0HYhAQAAIAAJ (cit. a p. 48).

[65] R.C. Martin. *Clean Architecture: Guida per diventare abili progettisti di architetture software*. Maestri di programmazione. Feltrinelli Editore, 2018. ISBN: 9788850318261. URL: https://books.google.it/books?id=pBhWDwAAQBAJ (cit. a p. 48).

[66] *SQLDelight*. URL: https://cashapp.github.io/sqldelight (visitato il 12/07/2022) (cit. a p. 50).

[67] *Ktor*. URL: https://ktor.io (visitato il 12/07/2022) (cit. a p. 53).

[68] Stefano Leli. *La Dependency Injection*. Gen. 2010. URL: https://www.html.it/pag/18718/la-dependency-injection (visitato il 12/07/2022) (cit. a p. 54).

[69] *Koin*. URL: https://insert-koin.io (visitato il 12/07/2022) (cit. a p. 54).

[70] *Jetpack Compose*. URL: https://developer.android.com/jetpack/compose (visitato il 12/07/2022) (cit. a p. 56).

[71] *SwiftUI*. URL: https://developer.apple.com/xcode/swiftui (visitato il 12/07/2022) (cit. a p. 56).

[72] Federico Torres. *Kotlin Multiplatform Mobile and how to share ViewModel: An architecture proposal*. Feb. 2022. URL: https://www.droidcon.com/2022/02/02/kotlin-multiplatform-mobile-and-how-to-share-viewmodel-an-architecture-proposal/ (visitato il 12/07/2022) (cit. a p. 102).

[73] Jaap Scherphuis. *Gallery of Psion Organiser II devices and accessories made by Psion*. URL: https://www.jaapsch.net/psion/galdev1.htm (visitato il 12/07/2022).

[74] Psion. *Psion Series 3a User Guide*. English. Ver. 1.0. Psion. Lug. 1993. 260 pp. URL: https://arvutimuuseum.ee/pdas/download/006/Psion_Series_3a_User_Guide.pdf (visitato il 12/07/2022).

[75] Marty Jerome. «PDAs Are Here: Zoomer vs. Newton». In: *PC Computing* (ott. 1993). URL: https://archive.org/details/pc-computing-magazine-v6i10/page/n197 (visitato il 12/07/2022).

[76] Steve Mann. «An Introduction to the GEOS Operating System». In: *Pen Computing* (feb. 1996). URL: https://www.pencomputing.com/developer/geos_introduction.html (visitato il 12/07/2022).

[77] *Certificazione Android Play Protect*. URL: https://www.android.com/certified (visitato il 12/07/2022).

[78] Aurora Vassallo. «La Clean Architecture applicata allo sviluppo mobile in ambiente Android = Clean Architecture applied to mobile development in the Android environment». Dic. 2020. URL: https://webthesis.biblio.polito.it/16662 (visitato il 12/07/2022).

Ringraziamenti

In conclusione di questa tesi vorrei ringraziare le persone che hanno reso possibile la sua stesura e quelle che mi sono state vicine in questi lunghi anni universitari.

In primis ringrazio il professor Giovanni Malnati per avermi fatto da relatore e per essere stato, prima ancora, un professore eccezionale.

Vorrei ringraziare anche Synesthesia, realtà accogliente che mi ha dato la splendida opportunità di lavorare a questa tesi, in particolare Marco, che ha proposto l'argomento e mi ha seguito e supportato nello sviluppo del progetto.

Ringrazio la mia famiglia, specialmente i miei genitori, che mi hanno sostenuto ogni giorno e mi hanno sempre fatto sentire la loro vicinanza, e mio fratello, senza il quale sono certo che questi ultimi anni sarebbero stati molto più difficili.

Desidero ringraziare anche i professori che mi hanno formato in questi anni universitari, in particolare il professor Gliozzi, che mi ha insegnato che determinazione e passione portano sempre risultati, e il professor De Rita, che tanto mi ha dato anche, e soprattutto, a livello umano.

Vorrei, poi, ringraziare gli amici vicini e lontani, vecchi e nuovi, che mi hanno accompagnato durante questi anni.

Anzitutto Giacinto, che è sempre stato presente, anche se eravamo a centinaia di chilometri di distanza.

Poi Alessandro, Ivan, Leonardo, Gabriele, Stefano e Roberta, che per primi mi hanno dimostrato come anche in un'aula universitaria possano nascere delle splendide amicizie. E Alex, Gabriella, Kamilia, Sara, Andrea, Luca e Bonifacio per avermi mostrato come queste possano diventare legami così forti che neanche il tempo e la lontananza possono rompere.

Andrea, Cristian, Sara, Federico e, in particolare, Alessandra, con i quali ho condiviso splendide giornate di studio all'inseguimento di un esame che sembrava irraggiungibile.

Giulia, Beatrice, Gaia, Samuele, Alessandro, Cesare, Valeria, Jacopo, Edoardo, Rossella, Claudia, Salvo, Sara, Vincenzo, Daniele, e Roberta, con i quali ho vissuto momenti fantastici, in aula e fuori, tra lezioni, gruppi di studio, caffè, pranzi e foto di dolci, momenti che porterò con me per sempre.

Mara, per essermi stata sempre vicina ed essere stata una guida in davvero tante occasioni.

Carlo, con cui è sempre un piacere scambiare messaggi vocali infiniti e che nonostante la distanza, sia geografica che ideologica, mi ha sempre capito molto bene e dato consigli utilissimi.

Leonardo, una delle poche persone capaci di tenere testa alla mia stupidità, che è stato mio complice per buona parte di questi anni e su cui so di poter sempre contare.

Irene, che con la sua energia riesce a tirarmi su in qualunque momento, che supporta anche le mie idee più insensate e che quando ne ho bisogno mi fa tornare a credere in me stesso.

Federica, che ho sempre ammirato per la sua determinazione e la sua forza e che tanto mi ha fatto crescere come persona, probabilmente più di quanto lei immagini.

Elisa, che mi è sempre stata accanto quando ne avevo bisogno, che spronandomi ed incoraggiandomi in ogni momento ha sempre creduto in me, anche quando non ci credevo neppure io.

Infine Aurora, che è stata un'amica eccezionale, ha salvato la mia sanità mentale più volte, è stata la voce della mia coscienza quando la mia coscienza non sapeva che dire e mi spinge costantemente ad essere una persona migliore.