UNIVERSITÀ POLITECNICA DELLE MARCHE
FACOLTÀ DI INGEGNERIA
Dipartimento di Ingegneria dell'Informazione
Corso di Laurea in Ingegneria Informatica e dell'Automazione

TESI DI LAUREA

Progettazione e implementazione di un social network in tecnologia Swift per gli amanti dei film

Design and implementation of a social network in Swift technology for movie fans

Relatore

Prof. Domenico Ursino

Correlatore

Dott. Enrico Corradini

Candidato

Nicolò Bartolini

ANNO ACCADEMICO 2022-2023

Sommario

In un'epoca contraddistinta dall'intersezione tra tecnologia e intrattenimento, lo sviluppo di applicazioni mobili nel settore cinematografico e televisivo emerge come una sfida affascinante e potenzialmente proficua. La presente tesi mira a esplorare lo sviluppo di un'applicazione iOS in Swift, funzionante come social network per appassionati di film e serie TV. L'indagine iniziale si concentra sull'ecosistema Apple e sul linguaggio Swift, stabilendo una solida base teorica. Segue un'analisi dettagliata dei requisiti funzionali e non funzionali, supportata dalla realizzazione dei diagrammi dei casi d'uso. La progettazione sfrutta altri diagrammi UML per offrire una visione architettonica coerente, facilitando la successiva fase di implementazione. Qui, viene realizzato un prototipo funzionante dell'applicazione. Una guida utente e un'analisi SWOT completano la ricerca, affiancate da un confronto con applicazioni simili, come TVTime e Letterboxd. Tali ultime discussioni permettono l'individuazione dei punti di debolezza dell'applicativo realizzato e dei suoi possibili sviluppi futuri.

Keyword: Programmazione Mobile, iOS, Swift, Social Network, Film, Serie TV, Ingegneria del Software, SWOT Analysis

Indice

Introduzione **1**

1 Introduzione a Swift **4**
1.1 La programmazione mobile . . . 4
1.1.1 Storia e sviluppo della programmazione mobile . . . 4
1.1.2 Principali piattaforme mobile: iOS e Android . . . 5
1.1.3 Principali linguaggi di programmazione per lo sviluppo mobile . . . 6
1.2 Storia di Swift . . . 6
1.2.1 Genesi e sviluppo di Swift . . . 6
1.2.2 Cronologia delle versioni e cambiamenti principali . . . 7
1.3 Caratteristiche principali di Swift . . . 7
1.3.1 Sicurezza e robustezza . . . 8
1.3.2 Facilità d'uso e leggibilità . . . 8
1.3.3 Efficienza e performance . . . 8
1.4 Sintassi di base di Swift . . . 8
1.4.1 Variabili e costanti . . . 8
1.4.2 Tipi di dati . . . 9
1.4.3 Operatori . . . 12
1.4.4 Strutture di controllo . . . 14
1.4.5 Funzioni . . . 15
1.4.6 Programmazione ad oggetti in Swift . . . 16
1.4.7 Programmazione asincrona in Swift . . . 19
1.5 Confronto tra Swift e altri linguaggi . . . 20
1.5.1 Confronto con Objective-C . . . 21
1.5.2 Confronto con altri linguaggi di programmazione mobile . . . 22
1.6 Ambienti di sviluppo . . . 22
1.6.1 AppCode . . . 22
1.6.2 Swift Playgrounds . . . 23
1.6.3 Xcode . . . 24
1.7 SwiftUI . . . 24
1.7.1 Caratteristiche principali di SwiftUI . . . 25
1.7.2 Confronto con Storyboard . . . 26
1.8 Swift e lo sviluppo iOS . . . 26
1.8.1 SwiftData vs Core Data . . . 26
1.8.2 CloudKit . . . 27

1.8.3 Integrazione di Firebase in un progetto Swift 27
1.8.4 Richieste HTTP: la libreria Alamofire 28
1.8.5 Gestione delle immagini: la libreria Kingfisher 29

2 Specifica e analisi dei requisiti 30
2.1 Definizione del problema 30
2.1.1 Contesto 30
2.1.2 Obiettivi del progetto 30
2.1.3 Nome dell'applicazione 31
2.1.4 Glossario di progetto 31
2.2 Analisi dei requisiti 33
2.2.1 Requisiti funzionali 33
2.2.2 Requisiti non funzionali 35
2.3 Analisi della concorrenza 35
2.3.1 Analisi di TVTime 36
2.3.2 Analisi di TMDB 37
2.3.3 Analisi di Letterboxd 37
2.4 Casi d'uso 39
2.4.1 Definizione degli attori 39
2.4.2 Definizione dei casi d'uso 39
2.4.3 Diagrammi dei casi d'uso 43
2.5 Matrice di mapping dei requisiti 46
2.6 Conclusioni 47

3 Progettazione 49
3.1 Architettura dell'applicazione 49
3.1.1 Pattern Model-View-ViewModel (MVVM) 49
3.2 Progettazione del database 50
3.2.1 Caratteristiche di Cloud Firestore 50
3.2.2 Struttura del database 51
3.3 Progettazione dell'interfaccia utente 51
3.3.1 Mappa dell'applicazione 51
3.3.2 Logo dell'applicazione 52
3.3.3 Mockup dell'applicazione 53
3.4 Diagrammi 54
3.4.1 Diagrammi di sequenza 54
3.4.2 Diagrammi di attività 56
3.4.3 Diagramma delle classi 57
3.5 Ottimizzazione 59
3.5.1 Ottimizzazione delle prestazioni 60
3.5.2 Gestione delle notifiche push 60
3.6 Conclusioni 61

4 Implementazione 62
4.1 Configurazione iniziale del progetto 62
4.1.1 Creazione del progetto Xcode e del repository Git 62
4.1.2 Installazione di CocoaPods e confronto con Swift Package Manager . . 63
4.1.3 Creazione e impostazione del progetto Firebase 64
4.2 Librerie esterne utilizzate 64
4.2.1 Librerie funzionali 65
4.2.2 Librerie di SwiftUI 65

4.3 Implementazione dell'architettura di WatchWise 66
4.3.1 Classi Model 66
4.3.2 Classi ViewModel 67
4.3.3 Classi View 67
4.3.4 Classi Repository 69
4.3.5 Classi ausiliarie 69
4.4 Analisi del codice implementativo 70
4.4.1 Avvio dell'applicazione 70
4.4.2 Flusso di autenticazione 71
4.4.3 `HomeNavigationView` 78
4.4.4 Schermate di dettaglio 80
4.4.5 Schermate di personalizzazione 83
4.5 Conclusioni 84

5 Manuale utente **87**
5.1 Presentazione di WatchWise e dei suoi obiettivi 87
5.2 Autenticazione 88
5.2.1 Login 88
5.2.2 Registrazione 88
5.2.3 Reset della password 89
5.3 Schermata principale 89
5.3.1 Schermata Home 90
5.3.2 Schermata Feed 90
5.3.3 Schermata Esplora 91
5.3.4 Schermata Episodi 92
5.3.5 Schermata Profilo 93
5.4 Dettagli 95
5.4.1 Dettaglio dei film 96
5.4.2 Dettaglio delle serie TV 96
5.4.3 Dettaglio delle persone 97
5.5 Funzionalità social 98
5.5.1 Dettaglio degli utenti 98
5.5.2 Valutazioni e recensioni 99
5.6 Conclusioni 100

6 Discussione **101**
6.1 SWOT Analysis 101
6.1.1 Punti di Forza 101
6.1.2 Punti di Debolezza 102
6.1.3 Opportunità 103
6.1.4 Minacce 104
6.2 Confronto con sistemi correlati 104
6.2.1 Confronto con IMDb 105
6.2.2 Confronto con JustWatch 105
6.2.3 Confronto con Hobi 107

Conclusioni **109**

Bibliografia **111**

Ringraziamenti **113**

Elenco delle figure

1.1 Martin Cooper con un prototipo del DynaTAC 4
1.2 Steve Jobs con il primo iPhone 5
1.3 Esempio di una schermata di AppCode 23
1.4 Esempio di una schermata di Swift Playgrounds 23
1.5 Esempio di una schermata di Xcode 24

2.1 Pagina di tracciamento degli episodi dell'app TVTime 36
2.2 Pagina principale di TMDB (`https://www.themoviedb.org/`) 37
2.3 Pagina di condivisione delle liste dell'app Letterboxd 38
2.4 Diagramma dei casi d'uso per la sezione "Autenticazione e Registrazione" . . 43
2.5 Diagramma dei casi d'uso per la sezione "Esplorazione dei contenuti" 44
2.6 Diagramma dei casi d'uso per la sezione "Interazioni sociali" 44
2.7 Diagramma dei casi d'uso per la sezione "Gestione delle liste" 45
2.8 Diagramma dei casi d'uso per la sezione "Valutazioni e recensioni" 45
2.9 Diagramma dei casi d'uso per la sezione "Gestione del profilo utente" 46
2.10 Diagramma dei casi d'uso per la sezione "Tracciamento degli episodi" 46

3.1 Mappa dell'applicazione WatchWise 52
3.2 Logo di WatchWise 52
3.3 Mockup di WatchWise 53
3.4 Diagramma di sequenza relativo alla visualizzazione del profilo utente 55
3.5 Diagramma di sequenza relativo all'inserimento (o alla rimozione) di un film nella watchlist 55
3.6 Diagramma di sequenza relativo all'invio di un consiglio 56
3.7 Diagramma di attività relativo alla registrazione di un nuovo utente 57
3.8 Diagramma di attività relativo all'inserimento di un episodio nella lista degli episodi visti 58
3.9 Diagramma di attività relativo alla visualizzazione dei prossimi episodi da vedere 58
3.10 Diagramma delle classi della sezione *Model* 59
3.11 Diagramma delle classi della sezione *Repository* 59

4.1 Schermata di creazione di un progetto Xcode 63

5.1 Schermata di login 88
5.2 Schermata di login con errore 88

5.3 Schermata di registrazione (fase I) . . . 89
5.4 Schermata di registrazione (fase II) . . . 89
5.5 Barra di navigazione inferiore . . . 90
5.6 Schermata Home . . . 90
5.7 Schermata delle chat . . . 91
5.8 Esempio di chat . . . 91
5.9 Schermata Esplora . . . 92
5.10 Esempi di ricerche . . . 92
5.11 Schermata Episodi . . . 93
5.12 Schermata Profilo . . . 94
5.13 Schermata di modifica del profilo . . . 94
5.14 Schermata di modifica dell' immagine di sfondo . . . 94
5.15 Esempio di schermata di dettaglio di una lista . . . 95
5.16 Schermata di creazione lista personalizzata . . . 95
5.17 Esempio di schermata di dettaglio di un film (I) . . . 96
5.18 Esempio di schermata di dettaglio di un film (II) . . . 96
5.19 Esempio di schermata di dettaglio di una serie TV . . . 97
5.20 Esempio di schermata di dettaglio di una stagione di una serie TV . . . 97
5.21 Esempio di schermata di dettaglio di una persona (I) . . . 98
5.22 Esempio di schermata di dettaglio di una persona (II) . . . 98
5.23 Istogramma delle valutazioni . . . 99
5.24 Schermata di elenco delle recensioni . . . 99
5.25 Menu "Liste e condivisione" . . . 100
5.26 Menu di condivisione . . . 100

6.1 Informazioni particolari reperibili nella schermata Home di IMDb . . . 106
6.2 Informazioni sui servizi di visione offerte da JustWatch . . . 107
6.3 Alcune delle statistiche offerte da Hobi . . . 108

Elenco delle tabelle

2.1 Glossario di progetto 33
2.2 Matrice di mapping dei requisiti (Parte 1) 47
2.3 Matrice di mapping dei requisiti (Parte 2) 47

Elenco dei listati

1.1 Sintassi per la dichiarazione e l'inizializzazione di variabili o costanti 8
1.2 Esempio di dichiarazione e inizializzazione di variabili e costanti 9
1.3 Esempio di separazione di dichiarazione e inizializzazione di variabili 9
1.4 Esempio di interpolazione nelle stringhe 10
1.5 Esempio di inizializzazione di un array ed accesso ad un suo elemento 10
1.6 Esempi di utilizzo di tuple 11
1.7 Esempio di utilizzo di un dizionario 11
1.8 Esempio dimostrativo del fatto che i set non contengono duplicati 11
1.9 Utilizzo degli optional in Swift 12
1.10 Esempio di utilizzo dell'operatore di coalescenza nulla 13
1.11 Esempio di utilizzo dell'istruzione `switch` in Swift 15
1.12 Esempio di utilizzo del ciclo `for-in` in Swift 15
1.13 Esempio di definizione e chiamata di una funzione in Swift 15
1.14 Esempio di utilizzo di una closure in Swift 16
1.15 Esempio di definizione e utilizzo di un `enum` in Swift 17
1.16 Esempio di definizione e utilizzo di una `struct` in Swift 17
1.17 Esempio di definizione di una classe in Swift 18
1.18 Esempio di creazione di un'istanza di una classe in Swift 18
1.19 Esempio di ereditarietà in Swift 18
1.20 Esempio di definizione e utilizzo di un metodo in Swift 19
1.21 Esempio di utilizzo della tecnologia Grand Central Dispatch in Swift 20
1.22 Esempio di utilizzo delle Operation Queue in Swift 20
1.23 Esempio di utilizzo delle API asincrone in Swift 5.5 21
1.24 Esempio di codice SwiftUI 25
1.25 Esempio di codice CloudKit 28
1.26 Esempio di codice Firebase 28
1.27 Esempio di richiesta GET con Alamofire 28
1.28 Esempio di richiesta POST con Alamofire 29
1.29 Esempio di utilizzo di Kingfisher 29
4.1 Inizializzazione del progetto Firebase nel codice sorgente 65
4.2 Entry point dell'applicazione 71
4.3 Inizializzazione della classe `AuthManager` 72
4.4 Login con indirizzo email e password 73
4.5 Logica della registrazione tramite email e password 74
4.6 Estensioni per la validazione di email e password 74

4.7 Codice eseguito dal pulsante "Entra con Google" 75
4.8 Metodo per l'autenticazione tramite Google 76
4.9 Logica di gestione dell'immagine di profilo 77
4.10 Struttura della vista `HomeNavigationView` 78
4.11 Funzionalità di ricerca dei film 79
4.12 Implementazione dell'aggiunta e della rimozione di un film in una lista . . . 81
4.13 Logica di ottenimento dei dettagli delle persone 82
4.14 Metodo per la conversione di minuti in mesi, giorni e ore 83
4.15 Logica di ottenimento dei dettagli delle liste 84
4.16 Logica di modifica dell'immagine di profilo 85

Introduzione

Nell'era digitale contemporanea, la fusione tra tecnologia e intrattenimento non è solo inevitabile, ma anche imperativa, emergendo come un pilastro fondamentale nella struttura del nostro stile di vita quotidiano. In un mondo sempre più interconnesso, la passione per la programmazione informatica, e in particolare per lo sviluppo di applicazioni mobili, trova un terreno fertile e incredibilmente promettente nel settore dei film e delle serie TV. Questo comparto, una volta confinato ai grandi schermi e alle sale di proiezione, ha subito una metamorfosi radicale e ha conosciuto una crescita esponenziale negli anni recenti. Questa ascesa vertiginosa è alimentata non soltanto dalla straordinaria qualità delle produzioni cinematografiche e dalla profondità della narrazione, ma anche e soprattutto dal desiderio intrinseco dell'essere umano di connettersi, comunicare e condividere esperienze. Inoltre, l'avvento delle piattaforme di streaming e l'accessibilità a un'ampia gamma di contenuti hanno catalizzato un interesse ancora maggiore in questo settore, rendendo imperativo per gli sviluppatori riconoscere e capitalizzare su questa tendenza. Pertanto, la decisione di dedicare una tesi allo sviluppo di un'applicazione iOS in Swift, pensata e progettata come un social network per gli entusiasti di film e serie TV, appare non solo assolutamente calzante, ma anche straordinariamente opportuna. Questa applicazione ambisce a fornire una piattaforma intuitiva e coinvolgente dove gli utenti possono non solo scoprire nuovi contenuti, ma anche interagire in modo significativo con altri appassionati, contribuendo a costruire una comunità unita e interconnessa intorno a un interesse comune.

La motivazione che si cela dietro alla stesura di questa tesi, inoltre, è radicata in un'analisi ponderata e meticolosa delle tendenze di mercato in atto e delle proiezioni per il futuro. L'ascesa esponenziale degli smartphone, abbinata all'ubiquità delle applicazioni mobili, ha irrevocabilmente rivoluzionato il modo in cui le persone interagiscono con il mondo circostante, estendendo il loro raggio d'azione ben oltre i confini fisici e temporalmente definiti. In tale scenario, le reti sociali specializzate, che una volta erano considerate un sottogenere marginale nel vasto ecosistema dei social media, stanno guadagnando una crescente rilevanza. Queste piattaforme tematiche sono diventate spesso il principale punto di riferimento per utenti alla ricerca di contenuti di nicchia, offrendo un livello di interazione e personalizzazione difficilmente raggiungibile attraverso canali più generalisti.

Ma il discorso non si limita semplicemente all'analisi delle tendenze di mercato o alla scelta opportunistica di un settore in crescita. Il progetto in questione ha il potenziale di inserirsi in una tela più vasta, ovvero quella dell'evoluzione continua delle piattaforme di intrattenimento e della loro progressiva e inevitabile integrazione tecnologica. In un panorama già saturo e dominato da giganti dello streaming, come Netflix, Amazon Prime Video e Disney+, un'applicazione che funge da punto d'incontro specializzato per gli appassionati di

film e serie TV non solo offrirebbe un servizio altamente focalizzato e curato, ma potrebbe anche aprire la strada a nuove opportunità commerciali, sinergie strategiche e partenariati con i colossi esistenti dell'industria dell'intrattenimento.

La sfida qui non è solo quella di creare un prodotto attraente, ma di posizionarlo in un ecosistema in continua evoluzione, una sfida che richiede una visione a 360 gradi che tenga conto di fattori come l'engagement dell'utente, la monetizzazione e la sostenibilità a lungo termine. Questo si collega intimamente con la scelta degli strumenti e delle tecnologie impiegate per sviluppare l'applicazione. Dal punto di vista tecnico, l'utilizzo di Swift come linguaggio di programmazione e della piattaforma iOS come ambiente di distribuzione non è stato casuale, ma è stato, piuttosto, il risultato di considerazioni sia tecniche che strategiche. Swift è ampiamente apprezzato per la sua efficienza di esecuzione e per una sintassi chiara e concisa che facilita lo sviluppo rapido, mentre iOS gode non solo di un ecosistema robusto ma anche di una base utente ampia, variegata e generalmente disposta a spendere per contenuti di qualità.

È, quindi, nel contesto di queste dinamiche complesse e interconnesse che la presente tesi si inserisce, con l'obiettivo di esaminare in dettaglio il potenziale di un'applicazione iOS in Swift dedicata a una comunità di appassionati di film e serie TV.

Nel contesto di questa ricerca, la fase iniziale è stata dedicata a un'esplorazione scrupolosa dell'ambiente della programmazione mobile, con un'enfasi particolare sull'ecosistema Apple e sul linguaggio di programmazione Swift. Questo non solo ha fornito un fondamento teorico solido per il lavoro a seguire, ma ha anche permesso di delineare le peculiarità e i vantaggi offerti da queste tecnologie rispetto ad altre soluzioni disponibili nel mercato.

Successivamente, il focus si è spostato verso l'ingegnerizzazione vera e propria dell'applicazione iOS. In questa fase, è stata condotta un'analisi meticolosa dei requisiti funzionali e non funzionali, complementata dalla delineazione dei casi d'uso specifici per l'applicazione in questione. L'integrazione e l'allineamento tra questi elementi sono stati rigorosamente verificati attraverso una matrice di mapping dei requisiti, assicurando una congruenza ottimale tra le specifiche del progetto e le funzionalità desiderate.

A seguire, è stata avviata la fase di progettazione del software. In questo segmento, è stato elaborato un insieme di diagrammi UML per fornire una visione architettonica coerente e dettagliata dell'applicazione. Questi diagrammi hanno avuto un ruolo cruciale nell'anticipare e delineare la struttura complessiva del software, agevolando considerevolmente le fasi di implementazione che sarebbero seguite.

Nella fase di implementazione, infine, si sono materializzate la scrittura del codice Swift e la realizzazione pratica dell'applicazione. Questa fase culminante ha visto la comparsa di un prototipo funzionante del social network tematico, il quale rappresenta non solo la sintesi di tutto il processo di sviluppo, ma anche una significativa contribuzione all'innovazione nel campo dell'ingegneria informatica e dell'automazione. Attraverso questo prototipo, non solo si è riusciti a concretizzare un concetto che era rimasto teorico, ma si è anche aperta la strada per ulteriori sviluppi e raffinamenti, contribuendo in maniera sostanziale al corpo di conoscenze accademiche e industriali nel settore.

Per fornire una visione comprensiva del progetto, la tesi è strutturata nei seguenti capitoli:

- Nel Capitolo 1, si offre un'introduzione alla programmazione mobile, focalizzandosi su Swift. Nello specifico, sono state descritte le principali funzionalità del linguaggio di programmazione, le modalità di sviluppo attraverso esso ed è stato posto un accento sul framework SwiftUI.

- Nel Capitolo 2, si conduce un'analisi dei requisiti e una specifica dei casi d'uso. Vengono definiti dettagliatamente gli obiettivi del progetto, vengono delineati i requisiti funzionali e non funzionali e vengono realizzati i diagrammi dei casi d'uso.

- Nel Capitolo 3 viene illustrata la fase di progettazione del progetto software. Viene, innanzitutto, definita l'architettura dell'applicazione e, successivamente, vengono illustrati i diagrammi di progettazione (ovvero, diagrammi di sequenza, diagrammi di attività e diagrammi delle classi). Il capitolo si conclude con una discussione su questioni di ottimizzazione.
- Nel Capitolo 4, si esamina l'implementazione del codice, descrivendo, innanzitutto, i processi di configurazione del progetto, passando, poi, alla presentazione delle classi con le quali è stata implementata l'architettura precedentemente progettata, e spiegando, infine, segmenti di codice particolarmente rilevanti.
- Nel Capitolo 5, si redige una guida utente completa, delineando tutte le funzionalità dell'applicazione.
- Nel Capitolo 6, si svolge un'analisi SWOT dell'applicazione e una comparazione con applicazioni simili, quali TVTime, Letterboxd, IMDb, JustWatch e Hobi. Tali discussioni consentono di identificare dettagliatamente eventuali problematiche dell'applicazione e possibili spunti per sviluppi e miglioramenti futuri.
- Nel Capitolo 7, si delineano le conclusioni, effettuando una discussione sull'intero progetto svolto.

CAPITOLO 1

Introduzione a Swift

Questo capitolo fornisce una panoramica della programmazione mobile, concentrandosi sul linguaggio Swift. Tratterà la storia degli sviluppi in programmazione mobile e le piattaforme che la utilizzano. Poi, esaminerà la genesi del linguaggio Swift, analizzandone in dettaglio le caratteristiche e la sintassi fondamentale. Verrà, quindi, effettuato un breve confronto tra Swift e alcuni linguaggi alternativi, seguito da un'analisi degli ambienti di sviluppo per le applicazioni Swift. Infine, si fornirà una visione d'insieme del framework SwiftUI e di aspetti chiave dello sviluppo iOS con Swift. L'obiettivo del capitolo, dunque, è un'introduzione al contesto della programmazione mobile, con un focus particolare sullo sviluppo di app iOS in Swift.

1.1 La programmazione mobile

Il termine *mobile* è un aggettivo che, in relazione a un dispositivo, può assumere diversi significati. Può riferirsi alla mobilità intrinseca del dispositivo stesso, come nel caso di droni o robot, può indicare la mobilità del collegamento tra il dispositivo e gli altri dispositivi a cui è connesso, oppure può riferirsi alla mobilità dell'utente del dispositivo. Quando si parla di programmazione mobile, ci si riferisce a quest'ultima interpretazione del termine. Pertanto, la programmazione mobile riguarda lo sviluppo di applicazioni per dispositivi mobili e, più specificamente, si occupa di *mobile computing*, ovvero quella tipologia di calcolo computazionale che è accessibile mentre si è in movimento.

1.1.1 Storia e sviluppo della programmazione mobile

Figura 1.1: Martin Cooper con un prototipo del DynaTAC

La storia dei dispositivi mobile, in particolare dei telefoni cellulari, inizia nel 1983 con il rilascio del *DynaTAC 8000X* da parte di Motorola (Figura 1.1), il primo telefono cellulare. Questo dispositivo, seppur rivoluzionario, non abbracciava ancora il concetto di programmazione mobile, limitandosi a permettere di effettuare telefonate in movimento.

La programmazione mobile inizia a prendere forma nel 1993, 10 anni dopo, quando IBM rilascia *Simon*, il primo "smartphone". Simon includeva le prime applicazioni mobile, come calendario, rubrica e blocchi di appunti, sebbene fossero tutte sviluppate internamente da IBM. Nel 1994, il rilascio di *Palm OS* amplia le possibilità della programmazione mobile, permettendo una maggiore personalizzazione delle applicazioni.

Nel 1996, Nokia rilascia il *9000 Communicator*, il primo telefono cellulare con piena connettività Internet. Nel 1998, *Symbian OS* introduce la

possibilità di sviluppare applicazioni di terze parti, un'innovazione che avrebbe avuto un impatto significativo sulla programmazione mobile.

L'era moderna dei dispositivi mobile inizia nel 2003 con il rilascio della prima versione di *Android* da parte di Android Inc., prima dell'acquisizione da parte di Google nel 2005. Nel 2007, Apple rilascia il primo *iPhone* (Figura 1.2) e la prima versione di *iOS*, offrendo una nuova piattaforma per lo sviluppo di applicazioni mobile e un hardware rivoluzionario, in particolare grazie alla tecnologia multi-touch.

Nel 2008, Apple e Google rilasciano rispettivamente l'*App-Store* (con l'*iOS SDK*) e l'*Android Market* (ora *Google Play Store*, con l'*Android SDK*), aprendo la strada a molti sviluppatori per entrare nel mondo della programmazione mobile, che fino a quel momento era un percorso pieno di ostacoli.

Figura 1.2: Steve Jobs con il primo iPhone

Nel 2010, viene rilasciato *PhoneGap* (ora *Apache Cordova*), il primo framework multipiattaforma che permette di sviluppare applicazioni utilizzando HTML, CSS e JavaScript. Questo segna l'inizio della diffusione del paradigma multipiattaforma nella programmazione mobile. Nel 2017, Google rilascia Flutter, uno dei framework multipiattaforma più utilizzati nel mondo della programmazione mobile.

In conclusione, la programmazione mobile ha percorso un lungo cammino dalla sua nascita. Da semplici dispositivi per effettuare chiamate in movimento, si è passati a sofisticati smartphone che possono eseguire una vasta gamma di applicazioni. Questo progresso è stato possibile grazie a un processo di continua evoluzione nel campo del software e dell'hardware che sta continuando anche oggi.

1.1.2 Principali piattaforme mobile: iOS e Android

Nel panorama attuale dei dispositivi mobile, due giganti tecnologici dominano il mercato: Apple e Google. Queste due aziende hanno sviluppato, e continuano a supportare, le due principali piattaforme mobile in uso oggi: *iOS* e *Android*. Entrambi sono sistemi operativi incentrati sulle applicazioni (proprietarie o di terze parti), ma presentano differenze sostanziali che rendono interessante un'analisi di ciascuno di essi.

iOS

iOS è il sistema operativo mobile sviluppato e mantenuto da Apple. È un sistema operativo esclusivo per i dispositivi Apple, tra cui iPhone e iPad. iOS è noto per il suo design elegante, l'interfaccia utente intuitiva e la sua fluidità. La caratteristica che maggiormente lo distingue da Android è che Apple controlla strettamente l'ecosistema di iOS, compreso l'App Store, che è l'unico modo "legittimo" per scaricare e installare applicazioni su un dispositivo Apple. Questo controllo stretto permette ad Apple di mantenere elevati standard di qualità per le applicazioni e di garantire la sicurezza dei suoi utenti. Tuttavia, questo approccio può limitare la personalizzazione e la flessibilità per gli sviluppatori e gli utenti.

Android

Android, al contrario, è un sistema operativo *open source* sviluppato da Google e utilizzato da una vasta gamma di produttori di dispositivi, tra cui Samsung, LG, Sony e molti altri. Questa diversità di produttori porta a una grande varietà di dispositivi Android disponibili, con una vasta gamma di specifiche hardware, dimensioni dello schermo e caratteristiche. Android è noto per la sua flessibilità e personalizzazione, permettendo agli sviluppatori e agli

utenti di modificare molti aspetti del sistema operativo. Tuttavia, questa apertura può portare a problemi di frammentazione, con dispositivi che eseguono versioni diverse di Android e con livelli di supporto variabili. Inoltre, mentre Google Play Store è il principale marketplace per le applicazioni Android, esistono altri store di terze parti, il che può portare a problemi di sicurezza.

1.1.3 Principali linguaggi di programmazione per lo sviluppo mobile

Nel campo dello sviluppo di applicazioni mobile, esistono vari linguaggi di programmazione che gli sviluppatori possono utilizzare. La scelta del linguaggio di programmazione dipende da vari fattori, tra cui la piattaforma target e le specifiche dell'applicazione. Tuttavia, spesso, la piattaforma target è il fattore che "obbliga" una scelta specifica per il linguaggio di programmazione da utilizzare.

Per lo sviluppo di applicazioni native[1] per iOS, i linguaggi di programmazione principali sono *Swift* (oggetto di questo capitolo) e *Objective-C*. Swift è un linguaggio di programmazione potente e intuitivo utilizzabile per lo sviluppo di applicazioni native per macOS, iOS, iPadOS, watchOS e tvOS. Sviluppato da Apple stessa, Swift è progettato per essere facile da usare e per offrire prestazioni elevate. Objective-C, invece, è un linguaggio di programmazione più vecchio, ma ancora utilizzato, che ha servito come base per lo sviluppo di applicazioni iOS per molti anni prima dell'introduzione di Swift.

Per lo sviluppo di applicazioni native per Android, i linguaggi di programmazione principali sono *Kotlin* e *Java*. Kotlin è un linguaggio di programmazione moderno che combina concetti di programmazione orientata agli oggetti e programmazione funzionale. È stato sviluppato da JetBrains e successivamente adottato da Google come linguaggio di programmazione standard per lo sviluppo di applicazioni Android. Java, d'altra parte, è stato il linguaggio di programmazione standard per lo sviluppo di applicazioni Android per molti anni prima dell'introduzione di Kotlin.

Esistono anche vari linguaggi di programmazione e framework che permettono lo sviluppo di applicazioni mobile cross-platform[2]. Tra questi, *React Native*, *Flutter* e *Xamarin* sono tra i più popolari. React Native, sviluppato da Meta, permette di scrivere applicazioni in JavaScript che vengono poi eseguite in modo nativo sul dispositivo. Flutter, sviluppato da Google, utilizza il linguaggio di programmazione Dart e offre un'esperienza di sviluppo rapida e fluida. Xamarin, infine, permette di scrivere applicazioni in C# che possono essere eseguite su molteplici piattaforme. Questi linguaggi e framework permettono agli sviluppatori di scrivere un unico codice sorgente che può essere eseguito su più sistemi operativi, riducendo, così, il tempo e lo sforzo necessari per lo sviluppo di applicazioni per diverse piattaforme.

1.2 Storia di Swift

Come accennato nella sezione precedente, Swift è il linguaggio di programmazione ufficiale per lo sviluppo di applicazioni iOS. È stato presentato per la prima volta durante la Worldwide Developers Conference (WWDC) di Apple nel 2014.

1.2.1 Genesi e sviluppo di Swift

Il progetto Swift è stato avviato da Chris Lattner, con il contributo di molti altri ingegneri di Apple, nel 2010. L'obiettivo era quello di creare un linguaggio di programmazione che

[1] Applicazioni che vengono sviluppate e poi compilate per uno specifico sistema operativo e possono essere eseguite solo su di esso.

[2] Applicazioni multipiattaforma che vengono sviluppate per più sistemi operativi e vengono eseguite nativamente in base al sistema operativo sul quale vengono avviate.

combinasse la sicurezza e le prestazioni di linguaggi compilati staticamente come C++ con l'agilità, la leggibilità e l'interattività di linguaggi di scripting come Python. Swift è stato progettato per essere potente e intuitivo, offrendo un'esperienza di programmazione piacevole e interattiva.

Il linguaggio è stato sviluppato in segreto per diversi anni prima di essere rivelato al pubblico nel 2014. Da allora, Swift ha visto un rapido sviluppo e un alto tasso di adozione, diventando uno dei linguaggi di programmazione più popolari per lo sviluppo di applicazioni mobile.

1.2.2 Cronologia delle versioni e cambiamenti principali

Dal suo rilascio, Swift ha ricevuto numerosi aggiornamenti che hanno fornito ad esso, volta per volta, tutte le funzionalità che caratterizzano attualmente il linguaggio. Nel seguito verranno illustrate le principali caratteristiche delle varie versioni di Swift:

- *Swift 1.0*: rilasciato nel 2014 dopo le versioni preliminari a cui Apple ha lavorato segretamente negli anni precedenti, ha introdotto una serie di caratteristiche innovative, tra cui la gestione automatica della memoria, la sicurezza dei tipi e un sistema di gestione degli errori robusto.
- *Swift 2.0*: rilasciato nel 2015, ha introdotto miglioramenti significativi nelle prestazioni, una nuova sintassi per la gestione degli errori e il supporto per la programmazione funzionale.
- *Swift 3.0*: nel 2015, Apple ha annunciato che Swift sarebbe diventato un progetto open source[3], permettendo alla comunità di sviluppatori di contribuire al suo sviluppo. Ciò ha portato nel 2016 al rilascio di Swift 3.0, che ha introdotto una serie di cambiamenti significativi nella sintassi del linguaggio per rendere Swift più coerente e facile da usare.
- *Swift 4.0*: rilasciato nel 2017, ha introdotto una serie di miglioramenti nella stabilità del linguaggio e nella compatibilità delle API, rendendo più facile per gli sviluppatori la migrazione del loro codice esistente a Swift.
- *Swift 5.0*: rilasciato nel 2019, ha introdotto il supporto per l'ABI[4] stabile, rendendo Swift un linguaggio più maturo e affidabile per lo sviluppo di applicazioni a lungo termine. Questa versione ha anche introdotto il supporto per le stringhe raw.

Attualmente, l'ultima versione rilasciata e supportata ufficialmente è Swift 5.8, ma la versione Swift 5.9 è già stata rilasciata in versione beta. Questo a dimostrazione del fatto che Swift è un linguaggio di programmazione in continua evoluzione.

1.3 Caratteristiche principali di Swift

Swift è un linguaggio di programmazione potente e intuitivo, progettato per essere sicuro, veloce e interattivo. In questa sezione, verranno analizzati i suoi principali punti di forza che rappresentano le caratteristiche che hanno guidato il suo sviluppo.

[3] `https://github.com/apple/swift`
[4] Application Binary Interface

1.3.1 Sicurezza e robustezza

Swift è stato progettato con un forte focus sulla sicurezza. Il linguaggio include una serie di caratteristiche che aiutano a prevenire errori comuni di programmazione, come l'accesso a puntatori nulli (*null-safety*) o l'overflow degli interi. Swift utilizza un sistema di tipi statico che aiuta a prevenire errori di tipo (*type-safety*) e fornisce un sistema di gestione degli errori robusto che permette agli sviluppatori di gestire le eccezioni in modo sicuro e prevedibile. Inoltre, Swift include il supporto per la gestione automatica della memoria, che aiuta a prevenire errori comuni, come memory leak o riferimenti a oggetti già deallocati. Questo focus sulla sicurezza rende Swift un linguaggio di programmazione particolarmente adatto per lo sviluppo di applicazioni che richiedono un alto livello di affidabilità e stabilità.

1.3.2 Facilità d'uso e leggibilità

Swift è stato progettato per essere facile da usare e per avere una buona leggibilità. La sintassi del linguaggio è chiara e concisa, rendendo il codice Swift facile da leggere e da scrivere. Swift include anche una serie di caratteristiche che rendono il linguaggio più espressivo e flessibile, come le pattern matching function, le closure e le higher-order function. Queste caratteristiche, insieme alla sua sintassi pulita e alla sua enfasi sulla leggibilità, rendono Swift un linguaggio di programmazione allo stesso tempo facile e potente.

1.3.3 Efficienza e performance

Swift è stato progettato per essere veloce. Il linguaggio utilizza un compilatore ottimizzato che produce codice binario efficiente e veloce. Swift include anche una serie di caratteristiche che aiutano a migliorare le performance, come il supporto per la programmazione parallela e concorrente. Questo significa che Swift è in grado di sfruttare al meglio le moderne architetture di CPU multicore, che lo rendono un linguaggio di programmazione ideale per lo sviluppo di applicazioni che richiedono un alto livello di performance.

1.4 Sintassi di base di Swift

Dopo aver analizzato i punti di forza di Swift, questa sezione si addentrerà nel dettaglio della sua sintassi e, dunque, di come viene utilizzato.

1.4.1 Variabili e costanti

In Swift, le variabili e le costanti vengono gestite come nella maggioranza dei linguaggi di programmazione più utilizzati. Nello specifico, una variabile (o una costante) in Swift è un nome identificativo associato ad un valore di un tipo specifico. La differenza tra variabili e costanti risiede nel fatto che il valore associato alle prime può essere modificato in seguito alla loro inizializzazione, mentre il valore associato alle seconde non può essere modificato in seguito alla loro inizializzazione.

Swift offre due parole chiave per la dichiarazione e l'inizializzazione di variabili e costanti. La sintassi generica è mostrata nel Listato 1.1.

```
1  <var/let> <nomeVariabile>[: <tipo>] [= <valore>]
```

Listato 1.1: Sintassi per la dichiarazione e l'inizializzazione di variabili o costanti

Il Listato 1.2 contiene, invece, un esempio specifico di dichiarazione e inizializzazione.

```
1 var valoreVariabile = 10
2 let valoreCostante = "Stringa";
```

Listato 1.2: Esempio di dichiarazione e inizializzazione di variabili e costanti

Riguardo tale esempio è possibile notare che nella riga 1 non è stato utilizzato il *punto e virgola* (`;`). Questo per indicare il fatto che in Swift il punto e virgola può essere aggiunto se si vuole, ma il compilatore lo ignorerà. L'unico caso in cui il punto e virgola è obbligatorio è quando si vogliono inserire più istruzioni nella stessa riga.

Dichiarazione e inizializzazione

Swift consente di separare le operazioni di dichiarazione e inizializzazione di variabili. Ovviamente queste operazioni possono essere eseguite contemporaneamente, come è stato fatto nell'esempio precedente e come viene fatto nella grande maggioranza dei casi pratici. Tuttavia, qualora fosse necessario separarle, questo può essere effettuato solo con le variabili (e non con le costanti) utilizzando la cosiddetta *type annotation*, cioè l'operazione con cui viene dichiarata una variabile e il suo tipo, senza però assegnarvi un valore. È importante, però, ricordare che finché la variabile dichiarata non verrà inizializzata (cioè finché non verrà assegnato ad essa un valore), questa non potrà essere utilizzata senza causare errori. Un esempio della separazione di dichiarazione e inizializzazione è mostrato nel Listato 1.3.

```
1 var variabileDichiarata: String
2 print(variabileDichiarata) // ERRORE: variabile non inizializzata
3 variabileDichiarata = 12 // ERRORE: il tipo non corrisponde
4 variabileDichiarata = "Stringa di prova"
5 print(variabileDichiarata) // -> Stringa di prova
```

Listato 1.3: Esempio di separazione di dichiarazione e inizializzazione di variabili

Nomi di variabili e costanti

I nomi di variabili e costanti in Swift, contrariamente a quanto succede in molti linguaggi di programmazione, possono contenere quasi ogni carattere, inclusi caratteri Unicode multi-byte (i.e.: π, α, 🗿, ecc. . .)

I nomi di costanti e variabili non possono contenere spazi, operatori, frecce e caratteri Unicode riservati. Inoltre, non possono iniziare con dei numeri, ma questi possono essere presenti dopo il primo carattere. Infine, per convenzione, i nomi di costanti e variabili dovrebbero essere scritti in "lowerCamelCase".

1.4.2 Tipi di dati

Swift offre una serie di tipi di dati fondamentali. In questa sezione verranno analizzati tali tipi di dati.

Numeri

Swift offre diversi tipi di dati numerici, tra cui interi (`Int`) e numeri in virgola mobile (`Float` e `Double`). Gli interi possono essere sia positivi che negativi. Swift supporta sia interi a 32 bit (`Int32`) che a 64 bit (`Int64`). Questi possono anche essere senza segno (`UInt32` e `UInt64`). I numeri in virgola mobile possono essere utilizzati per rappresentare numeri reali con una precisione di 32 bit (`Float`) o 64 bit (`Double`).

Booleani

Il tipo di dato booleano in Swift è chiamato `Bool` e può assumere i soliti due valori: `true` o `false`. Questo tipo di dato, come in praticamente tutti i linguaggi di programmazione, è utilizzato per rappresentare condizioni di verità.

Stringhe

Le stringhe in Swift sono rappresentate dal tipo di dato `String`. Le stringhe possono contenere qualsiasi carattere Unicode e supportano una serie di operazioni molto utili per lavorare con i testi, tra cui la concatenazione o l'interpolazione. Quando vengono definite in modo letterale, le stringhe devono essere incluse tra doppi apici (`"`) o tra apici singoli (`'`).

Per quanto riguarda l'interpolazione, questa può essere effettuata inserendo la variabile o l'espressione da interpolare all'interno di `\()`, come nell'esempio mostrato nel Listato 1.4.

```
1 var interpolazione = "Il doppio di 5 è \(5 * 2)"
```

Listato 1.4: Esempio di interpolazione nelle stringhe

Infine, Swift offre il tipo `Character` che rappresenta un carattere Unicode da 16 bit.

Collezioni

Swift offre una serie di collezioni per memorizzare un insieme di dati. Le collezioni offerte da Swift sono le seguenti:

- *Array*: gli array in Swift sono collezioni ordinate di elementi dello stesso tipo. Essi sono tipizzati, il che significa che è necessario specificare il tipo di elementi che possono contenere. Per accedere a un elemento di un array si utilizza il nome dell'array seguito dall'indice dell'elemento desiderato racchiuso tra parentesi quadre (Listato 1.5).

```
1 let array: [Int] = [1, 2, 3, 4, 5]
2 print(array[3]) // -> 4
```

Listato 1.5: Esempio di inizializzazione di un array ed accesso ad un suo elemento

- *Tuple*: le tuple in Swift sono gruppi di valori che possono essere di tipi diversi. Esse sono particolarmente utili quando si desidera raggruppare insieme valori correlati. Gli elementi di una tupla possono essere una lista di valori, ma anche una lista di valori con un nome. Per accedere agli elementi della tupla è possibile, quindi, utilizzare l'indice dell'elemento oppure, se presente, il suo nome. Alcuni esempi di utilizzo di tuple sono mostrati nel Listato 1.6.

```swift
let tupla: (Int, String) = (1, "serpente")
let tuplaConNome = (giorno: 30, mese: "luglio", anno: 2023)
let (numero, animale) = tupla
print(animale) // -> serpente
print("num=\(tupla.0), anim=\(tupla.1)") // -> num=1, anim=serpente
print("siamo nel \(tuplaConNome.anno)") // -> siamo nel 2023
```

Listato 1.6: Esempi di utilizzo di tuple

- *Dizionari*: i dizionari in Swift sono collezioni non ordinate di coppie chiave-valore dello stesso tipo. Come gli array, anche i dizionari in Swift sono tipizzati. Per accedere agli elementi di un dizionario si utilizzano le parentesi quadre come negli array, ma inserendovi la chiave dell'elemento desiderato (Listato 1.7).

```swift
var dizionario: [String: Int] = ["aa": 1, "ab": 2, "ac": 3]
print(dizionario["ab"]) // -> 2
dizionario["ab"] = 7
print(dizionario["ab"]) // -> 7
```

Listato 1.7: Esempio di utilizzo di un dizionario

- *Set*: I set in Swift sono collezioni non ordinate di elementi unici dello stesso tipo. A differenza degli array e dei dizionari, i set non mantengono un ordine specifico dei loro elementi e non permettono duplicati (Listato 1.8).

```swift
let set: Set<Int> = [1, 2, 3, 4, 5, 5]
// Anche inserendo il numero 5 due volte,
// il set conterrà solo un'occorrenza di 5.
print(set) // -> [1, 2, 3, 4, 5]
```

Listato 1.8: Esempio dimostrativo del fatto che i set non contengono duplicati

Oltre a fornire metodi per l'accesso e la manipolazione degli elementi, Swift fornisce anche una serie di higher-order function che permettono di lavorare con le collezioni in modo molto espressivo e conciso. Ad esempio, è possibile utilizzare funzioni come `map`, `filter` e `reduce` per trasformare, filtrare o combinare gli elementi di una collezione in un unico valore. Inoltre, Swift supporta la creazione di collezioni che possono essere di qualsiasi tipo, inclusi i tipi definiti dall'utente.

Optional

Gli *optional* in Swift rappresentano un concetto fondamentale, introdotto dal linguaggio, che garantisce la null-safety citata nella Sezione 1.3.1. Un optional in Swift è un tipo di dato che può contenere un valore o nessun valore (`nil`). Questo tipo è utilizzato per gestire situazioni in cui un valore potrebbe non essere disponibile. In molti linguaggi di programmazione, l'assenza di un valore è spesso rappresentata da un valore speciale come `null` o `nil`, che, essendo associabile anche ai tipi primitivi, può portare a errori se non gestito correttamente. Swift affronta questo problema introducendo il concetto di optional.

Un optional in Swift è essenzialmente un contenitore per un valore. Esso può contenere un valore (in tal caso si dice che l'optional è "non vuoto") o può non contenere alcun valore (in tal caso si dice che l'optional è "vuoto", o `nil`). Ciò permette di distinguere in modo esplicito tra un valore reale e l'assenza di un valore, riducendo la possibilità di errori.

Per dichiarare una variabile come optional in Swift si utilizza il punto interrogativo (`?`) inserito dopo la specifica del tipo di dato. Per accedere al valore contenuto in un optional, se si è sicuri che quest'ultimo contenga un valore, è necessario "scollegarlo" utilizzando il punto esclamativo (`!`). Tuttavia, è importante notare che scollegare un optional che, in realtà, non contiene un valore, e quindi contiene `nil`, causerà un errore. Per evitare questo, Swift fornisce il costrutto `if let`, che permette di scollegare un optional in modo più sicuro rispetto all'utilizzo del punto esclamativo. Il Listato 1.9 contiene alcuni esempi dell'utilizzo degli optional in Swift.

```
1  var stringa: String? // Dichiarazione di un optional
2  stringa = "Ciao, Swift!" // Assegnazione di un valore all'optional
3
4  if let stringaScollegata = stringa { // Scollegamento sicuro
5  print(stringaScollegata) // -> Ciao, Swift!
6  } else {
7  print("stringa è un optional vuoto")
8  }
9
10 // Se si ha la certezza che l'optional sia non vuoto, si può usare !
11 print(stringa!) // -> Ciao, Swift!
```

Listato 1.9: Utilizzo degli optional in Swift

1.4.3 Operatori

Un operatore è un simbolo o una frase speciale utilizzato per controllare, modificare o combinare valori. Swift supporta una serie di operatori, molti dei quali potrebbero essere già noti a chi ha esperienza con linguaggi come C. Tuttavia, Swift introduce anche nuovi operatori e modifica il comportamento classico di alcuni operatori già noti.

Operatori di assegnazione

L'operatore di assegnazione (=) è l'operatore che permette di assegnare un valore ad una variabile o ad una costante. In Swift, al contrario di come succede in molti linguaggi di programmazione, l'operatore di assegnazione non restituisce alcun valore, al fine di prevenire il suo utilizzo errato quando si intende utilizzare l'operatore di uguaglianza (==).

Operatori aritmetici

Gli operatori aritmetici di Swift sono i seguenti:

- *Operatore addizione* (`a + b`): permette di effettuare addizioni tra numeri.
- *Operatore sottrazione* (`a - b`): permette di effettuare sottrazioni tra numeri.
- *Operatore moltiplicazione* (`a * b`): permette di effettuare moltiplicazioni tra numeri.
- *Operatore divisione* (`a / b`): permette di effettuare divisioni tra numeri.

- *Operatore modulo* (`a % b`): permette di calcolare il resto della divisione intera tra numeri.
- *Operatori unari di segno* (`-a` oppure `+b`): permette di cambiare il segno di una variabile numerica.
- *Operatori di incremento/decremento e assegnazione* (`a += 3` oppure `b -= 7`): permette di addizionare o sottrarre al valore del primo operando il valore del secondo operando.

Come è evidente, questi operatori hanno lo stesso funzionamento dei classici operatori aritmetici che si possono trovare nella maggioranza dei linguaggi di programmazione. L'unica piccola differenza consiste nel fatto che gli operatori aritmetici in Swift rilevano e non consentono l'overflow dei valori, per evitare risultati inaspettati quando si lavora con numeri che diventano più grandi o più piccoli del range di valori consentito dal tipo che li memorizza.

Operatori logici

Swift supporta gli operatori logici standard come AND (`&&`), OR (`||`) e NOT (`!`). Questi operatori sono utilizzati per creare espressioni logiche complesse.

Operatori di confronto

Swift include una serie di operatori di confronto, tra cui uguaglianza (==), non uguaglianza (`!=`), maggiore (>), minore (<), maggiore o uguale (>=) e minore o uguale (<=). Gli operatori di confronto ritornano sempre un `Bool` che indica se il confronto è stato rispettato o meno.

Operatori di intervallo

Swift fornisce operatori di intervallo che non si trovano in C, come `a..<b` e `a...b`. Questi operatori vengono utilizzati come scorciatoie per la creazione di intervalli numerici. Ad esempio, l'operatore `2..<6` crea un intervallo numerico che va da `2` fino a `5`. L'omissione, in questo operatore, del simbolo < crea un intervallo in cui l'estremo finale è incluso.

Operatori ternari

Swift supporta l'operatore ternario condizionale (`a ? b : c`), che ha tre parti: una condizione booleana, un risultato per quando la condizione è vera e un risultato per quando la condizione è falsa. In altre parole, questo operatore ternario è una sorta di costrutto `if` (Sezione 1.4.4) compatto.

Operatori di coalescenza nulla

Swift introduce un operatore di coalescenza nulla (`a ?? b`) che è un modo rapido di fornire un valore predefinito se un optional è nullo. In termini tecnici, quando si deve accedere al valore di un optional, piuttosto che usare lo scollegamento sicuro (Listato 1.9), si può usare l'operatore di coalescenza nulla per restituire un valore di default nel caso in cui l'optional sia vuoto. Un esempio di utilizzo di tale operatore è mostrato nel Listato 1.10.

```
1 let stringaScollegata = stringa ?? "stringa è un optional vuoto"
```

Listato 1.10: Esempio di utilizzo dell'operatore di coalescenza nulla

Questi sono solo i principali e più importanti operatori disponibili in Swift. Il linguaggio offre, anche, la possibilità di definire operatori personalizzati e di implementare gli operatori standard per i propri tipi personalizzati.

1.4.4 Strutture di controllo

Swift fornisce una varietà di strutture di controllo del flusso. Queste includono cicli `while` per eseguire un'attività più volte, costrutti `if`, `guard`, e `switch` per eseguire diversi rami di codice in base a determinate condizioni, e istruzioni come `break` e `continue` per trasferire il flusso di esecuzione ad un altro punto nel codice.

Cicli `while` e `repeat-while`

I cicli `while` eseguono un blocco di istruzioni finché una condizione specificata è vera. Il ciclo `repeat-while`, simile al ciclo `do-while` in altri linguaggi, esegue un blocco di istruzioni almeno una volta e poi continua a ripeterlo finché la condizione specificata è vera.

Costrutti `if` e `guard`

I costrutti `if` in Swift vengono utilizzati per eseguire diversi rami di codice in base a determinate condizioni. Swift supporta anche le estensioni `else if` e `else` per gestire più di una condizione. Il costrutto `guard` è un costrutto che viene utilizzato per gestire le condizioni di uscita anticipata, migliorando la leggibilità del codice in modo da evitare l'annidamento eccessivo.

Costrutto `switch`

Il costrutto `switch` in Swift è notevolmente più potente rispetto al suo corrispondente in molti altri linguaggi di programmazione. I `case` possono corrispondere a molti modelli diversi, tra cui corrispondenze di intervalli, tuple e cast a un tipo specifico. I valori corrispondenti in un `case` di `switch` possono essere legati a costanti temporanee o variabili per l'uso all'interno del corpo del `case`; le condizioni di corrispondenza complesse possono essere espresse con una clausola `where` per ogni `case`. Il codice nel Listato 1.11 mostra un esempio di utilizzo dell'istruzione `switch` in Swift. In questo esempio, l'istruzione `switch` verifica il tipo e il valore della variabile `someValue`. Ogni `case` verifica una condizione diversa: se il valore è un intero o un double, se è una stringa che inizia con "Swift", se è una tupla di interi, e così via. Se nessuno dei casi corrisponde, viene eseguito il codice nel blocco `default`.

Ciclo `for-in`

Swift fornisce un ciclo `for-in` che rende semplice l'iterazione su array, dizionari, intervalli, stringhe e altre sequenze. Il codice nel Listato 1.12 mostra un esempio di utilizzo del ciclo `for-in` per iterare su un array.

Istruzioni `break` e `continue`

Le istruzioni `break` e `continue` in Swift vengono utilizzate per alterare il flusso di controllo all'interno dei cicli e delle istruzioni `switch`. L'istruzione `break` termina l'esecuzione del ciclo o dell'istruzione `switch` più interna, mentre l'istruzione `continue` termina l'iterazione corrente del ciclo più interno e inizia immediatamente la successiva senza eseguire eventuali istruzioni rimanenti.

```
let someValue: Any = 5
switch someValue {
    case 0 as Int:
        print("0 Int")
    case 0 as Double:
        print("0 Double")
    case let x where x is Int:
        print("Un valore intero")
    case let (x, y) as (Int, Int):
        print("Una tupla di due elementi Int: (\(x), \(y))")
    case let string as String where string.hasPrefix("Swift"):
        print("\(string) comincia con 'Swift'")
    default:
        print("Altro")
}
```

Listato 1.11: Esempio di utilizzo dell'istruzione `switch` in Swift

```
let numbers = [1, 2, 3, 4, 5]
for number in numbers {
    print(number)
}
```

Listato 1.12: Esempio di utilizzo del ciclo `for-in` in Swift

1.4.5 Funzioni

Le funzioni sono blocchi di codice che eseguono un compito specifico. Ogni funzione ha un nome che identifica il compito che svolge; questo nome viene utilizzato per chiamare la funzione in modo da eseguire il suo compito quando necessario.

Sintassi delle funzioni

La sintassi unificata delle funzioni di Swift è abbastanza flessibile da esprimere qualsiasi cosa, da una semplice funzione in stile C senza nomi di parametri a un complesso metodo in stile Objective-C con nomi e etichette di argomenti per ciascun parametro. I parametri possono possedere dei valori di default per semplificare le chiamate alle funzioni e possono essere passati come parametri in-out, che modificano la variabile passata una volta che la funzione ha completato la sua esecuzione. Il codice nel Listato 1.13 mostra un esempio di definizione e chiamata di una funzione in Swift.

```
func saluta(persona: String, giorno: String) -> String {
    return "Ciao (persona), oggi è (giorno)."
}
let saluto = saluta(person: "Mario", day: "domenica")
print(saluto) // -> Ciao Mario, oggi è domenica.
```

Listato 1.13: Esempio di definizione e chiamata di una funzione in Swift

Tipi di funzione

Ogni funzione in Swift ha un tipo, composto dai tipi dei parametri della funzione e dal tipo di ritorno. Questo tipo può essere utilizzato come se fosse un qualsiasi altro tipo di Swift, il che rende facile passare le funzioni come parametri ad altre funzioni e restituire funzioni dalle funzioni.

Funzioni annidate

Le funzioni possono anche essere scritte all'interno di altre funzioni con il fine di incapsulare delle funzionalità utili solo alla funzione "genitore"; tuttavia, ciò può avvenire solo in determinate situazioni. Queste funzioni annidate hanno accesso alle variabili del loro ambito esterno e possono essere restituite come valori da altre funzioni.

Closure

Le closure in Swift sono blocchi di codice che possono essere passati e restituiti in una funzione. Sono simili alle funzioni, ma possono essere assegnate a variabili e possono catturare e memorizzare riferimenti a variabili e costanti dal contesto circostante. Le closure sono particolarmente utili quando si lavora con funzioni che accettano altre funzioni come parametri o che restituiscono funzioni come risultato. Il codice nel Listato 1.14 mostra un esempio di utilizzo di una closure in Swift.

```
1 let numeri = [1, 2, 3, 4, 5]
2 let numeriAlQuadrato = numeri.map { $0 * $0 } // Closure nelle graffe
3 print(numeriAlQuadrato) // -> [1, 4, 9, 16, 25]
```

Listato 1.14: Esempio di utilizzo di una closure in Swift

1.4.6 Programmazione ad oggetti in Swift

La programmazione ad oggetti è un paradigma di programmazione che fornisce un modo di strutturare i programmi in modo che le proprietà e i comportamenti siano racchiusi in singoli oggetti (un concetto noto come incapsulamento). In Swift, come in molti altri linguaggi di programmazione moderni, la programmazione ad oggetti è un elemento chiave.

Enumerazioni

Le enumerazioni, o `enum`, rappresentano un tipo speciale che consente di definire un gruppo comune di valori correlati. Gli `enum` in Swift sono molto più flessibili rispetto ad altri linguaggi. Oltre a valori semplici, gli `enum` possono contenere valori associati di qualsiasi tipo e possono avere metodi, similmente alle classi. Il codice nel Listato 1.15 mostra un esempio di definizione e utilizzo di un `enum`.

Strutture

Le strutture, o `struct`, sono uno dei tipi di dati fondamentali in Swift. Una `struct` è un tipo composto che può contenere più proprietà e metodi. Le `struct` in Swift sono molto simili alle classi, ma hanno alcune differenze importanti. Ad esempio, le `struct` sono tipi di valore, il che significa che vengono copiate quando vengono passate come parametri nelle

```
enum Bussola {
    case nord
    case sud
    case est
    case ovest
}

var direzione = Bussola.ovest
direzione = .nord
```

Listato 1.15: Esempio di definizione e utilizzo di un `enum` in Swift

funzioni, mentre le classi sono tipi di riferimento. Il codice nel Listato 1.16 mostra un esempio di definizione e utilizzo di una `struct`.

```
struct Punto {
    var x: Int
    var y: Int
}

var punto = Punto(x: 1, y: 2)
punto.x = 3
```

Listato 1.16: Esempio di definizione e utilizzo di una `struct` in Swift

Classi

Le classi sono il cuore della programmazione ad oggetti in Swift. Una classe è una struttura di codice che combina metodi (funzioni) e proprietà (variabili) per creare un oggetto. Le classi possono ereditare metodi, proprietà e altre caratteristiche da altre classi, un concetto noto come ereditarietà. Le classi supportano anche le funzionalità di incapsulamento e polimorfismo, che sono concetti fondamentali della programmazione ad oggetti.

Definizione di una classe

Una classe in Swift viene definita utilizzando la parola chiave `class`, seguita dal nome della classe. La convenzione dice che il nome di una classe deve essere un sostantivo il più breve possibile e che spieghi al meglio il fine della classe; inoltre deve essere scritto in "UpperCamelCase". Il corpo della classe è racchiuso tra parentesi graffe. Il codice nel Listato 1.17 mostra un esempio di definizione di una classe. In questo esempio, la classe `Cane` ha due proprietà, `nome` e `razza`, e un metodo costruttore, `init`, che viene chiamato quando si crea un'istanza della classe.

Creazione di un'istanza di una classe

Per creare un'istanza di una classe, si utilizza la sintassi di inizializzazione; questa prevede l'utilizzo del nome della classe seguito da parentesi tonde contenenti i valori per gli eventuali argomenti della classe. Il codice nel Listato 1.18 mostra un esempio di creazione di un'istanza

```
class Cane {
    var nome: String
    var razza: String

    init(nome: String, razza: String) {
        self.nome = nome
        self.razza = razza
    }
}
```

Listato 1.17: Esempio di definizione di una classe in Swift

della classe `Cane`. In questo esempio, `max` è un'istanza della classe `Cane`. Le proprietà `nome` e `razza` dell'istanza sono inizializzate con i valori "Max" e "Labrador", rispettivamente.

```
let max = Cane(nome: "Max", razza: "Labrador")
```

Listato 1.18: Esempio di creazione di un'istanza di una classe in Swift

Ereditarietà

L'ereditarietà è un principio fondamentale della programmazione ad oggetti che permette di creare una nuova classe basata su una classe esistente. La nuova classe eredita tutte le proprietà e i metodi della classe esistente e può aggiungere nuove proprietà e metodi o sovrascrivere (override) quelli esistenti. Il codice nel Listato 1.19 mostra un esempio di ereditarietà. In questo esempio, la classe `Barboncino` è una sottoclasse della classe `Cane`. Essa eredita le proprietà `nome` e `razza` e il metodo `init` dalla classe `Cane`, e aggiunge una nuova proprietà, `colore`, e un nuovo metodo `init`. Il metodo `init` della classe `Barboncino` chiama il metodo `init` della classe `Cane` utilizzando la parola chiave `super`.

```
class Barboncino: Cane {
    var colore: String

    init(nome: String, colore: String) {
        self.colore = colore
        super.init(nome: nome, razza: "Barboncino")
    }
}

let nuvola = Barboncino(nome: "Nuvola", colore: "Bianco")
```

Listato 1.19: Esempio di ereditarietà in Swift

Metodi

I metodi sono funzioni associate a una particolare classe. I metodi in Swift possono accedere e modificare le proprietà della classe e possono essere utilizzati per implementare il

suo comportamento. Il codice nel Listato 1.20 mostra un esempio di definizione e utilizzo di un metodo. In questo esempio, la classe `Cane` ha un metodo `abbaia` che stampa un messaggio quando viene chiamato. Il metodo `abbaia` può accedere alla proprietà `nome` della classe utilizzando la parola chiave `self`, che rappresenta l'istanza della classe su cui viene chiamato il metodo.

```
class Cane {
    var nome: String
    var razza: String

    init(nome: String, razza: String) {
        self.nome = nome
        self.razza = razza
    }

    func abbaia() {
        print("\(self.nome) ha abbaiato!")
    }
}

let max = Cane(nome: "Max", razza: "Labrador")
max.abbaia() // -> Max ha abbaiato!
```

Listato 1.20: Esempio di definizione e utilizzo di un metodo in Swift

1.4.7 Programmazione asincrona in Swift

La programmazione asincrona è un paradigma di programmazione che consente di eseguire operazioni in background e di gestire più attività contemporaneamente senza bloccare l'esecuzione del programma nel thread principale. Ciò è particolarmente utile in applicazioni che richiedono operazioni di lunga durata, come il download di file da Internet, operazioni di I/O su disco o richieste di rete. In Swift, la programmazione asincrona può essere gestita in vari modi, tra cui l'utilizzo della Grand Central Dispatch (GCD), Operation Queue e, a partire da Swift 5.5, le nuove API asincrone.

Grand Central Dispatch

Grand Central Dispatch (GCD) è una tecnologia di basso livello che gestisce l'esecuzione di compiti asincroni. GCD fornisce e gestisce le cosiddette *code di dispatch*, che sono strutture dati utilizzate per gestire più compiti (o task) che devono essere eseguiti in parallelo. I task possono essere sottomessi a queste code per l'esecuzione asincrona su un insieme di thread del sistema.

GCD fornisce due tipi di code: seriali e concorrenti. Le code seriali eseguono un compito alla volta in ordine di arrivo, mentre le code concorrenti possono eseguire più compiti contemporaneamente. Il codice nel Listato 1.21 mostra un esempio di utilizzo della GCD. In questo esempio, viene creata una coda di dispatch seriale e vengono sottomessi due task per l'esecuzione asincrona. I task vengono eseguiti in ordine di arrivo, quindi "Task 1" viene stampato prima di "Task 2".

```
let coda = DispatchQueue(label: "com.example.miacoda")
queue.async { print("Task 1") }
queue.async { print("Task 2") }
```

Listato 1.21: Esempio di utilizzo della tecnologia Grand Central Dispatch in Swift

Operation Queue

Le Operation Queue sono un'alternativa di più alto livello alla GCD fornita dal framework *Foundation* di Swift. A differenza di GCD, le Operation Queue supportano le operazioni (o operation), che sono unità di lavoro incapsulate in un oggetto `Operation`. Le operazioni possono avere dipendenze, che consentono di specificare un ordine di esecuzione tra diverse operazioni. Inoltre, esse possono essere annullate, sospese e riprese, offrendo un controllo più granulare sull'esecuzione delle stesse. Il codice nel Listato 1.22 mostra un esempio di utilizzo delle Operation Queue. In esso, viene creata una Operation Queue e vengono create due operazioni. L'operazione 2 ha una dipendenza dall'operazione 1, il che significa che l'operazione 2 non verrà eseguita fino a quando l'operazione 1 non sarà stata completata. Le operazioni vengono, quindi, aggiunte alla coda per l'esecuzione asincrona.

```
let operationQueue = OperationQueue()
let operation1 = BlockOperation {
    print("Operation 1")
}
let operation2 = BlockOperation {
    print("Operation 2")
}
operation2.addDependency(operation1)
operationQueue.addOperation(operation1)
operationQueue.addOperation(operation2)
```

Listato 1.22: Esempio di utilizzo delle Operation Queue in Swift

API asincrone in Swift 5.5

A partire da Swift 5.5, è stato introdotto un nuovo modello di programmazione asincrona con l'aggiunta delle parole chiave `async` e `await`. Queste nuove funzionalità semplificano notevolmente la scrittura di codice asincrono in Swift, rendendo il codice più leggibile e più facile da scrivere. Il codice nel Listato 1.23 mostra un esempio di utilizzo delle API asincrone di Swift 5.5. In questo esempio, la funzione `ottieniDati` è marcata con la parola chiave `async`, indicando che esegue un'operazione asincrona. La chiamata a `ottieniDati` è preceduta dalla parola chiave `await`, che indica che l'esecuzione del codice dovrebbe essere sospesa fino a quando l'operazione asincrona non è completata.

1.5 Confronto tra Swift e altri linguaggi

Dopo aver analizzato nel dettaglio il funzionamento e la sintassi di Swift, risulta interessante confrontarlo con altri linguaggi di programmazione per comprendere meglio i suoi vantaggi e svantaggi. In questa sezione, dunque, Swift sarà confrontato con Objective-C,

```
func ottieniDati() async -> Data { // Simulazione richiesta di rete
    await Task.sleep(2 * 1_000_000_000) // Si ferma per 2 secondi
    return Data()
}
async {
    let dati = await ottieniDati()
    print("Dati ottenuti: (dati)")
}
```

Listato 1.23: Esempio di utilizzo delle API asincrone in Swift 5.5

che è stato a lungo il linguaggio principale per lo sviluppo di applicazioni iOS, e con altri linguaggi di programmazione mobile comuni.

1.5.1 Confronto con Objective-C

Objective-C è stato il linguaggio principale per lo sviluppo di applicazioni iOS fino all'introduzione di Swift nel 2014. Sebbene Swift sia stato progettato per essere compatibile con Objective-C, ci sono molte differenze tra i due linguaggi.

Sintassi

La sintassi di Swift è più concisa e meno verbosa rispetto a quella di Objective-C. Ad esempio, in Swift, non è necessario terminare ogni istruzione con un punto e virgola, a meno che non ci siano più istruzioni sulla stessa riga. Inoltre, Swift utilizza una sintassi più leggibile per le chiamate di funzioni e metodi, con parametri nominati che rendono il codice più descrittivo. Al contrario, la sintassi di Objective-C può essere più difficile da leggere e da scrivere, soprattutto per i programmatori che non hanno familiarità con la sintassi di tipo Smalltalk[5] utilizzata da Objective-C.

Sicurezza

Swift è stato progettato con un forte focus sulla type-safety e sulla gestione degli errori. Il linguaggio include una serie di caratteristiche che aiutano a prevenire errori comuni di programmazione, come la null-safety e l'overflow degli interi. Inoltre, Swift fornisce un sistema di gestione degli errori robusto che permette agli sviluppatori di gestire le eccezioni in modo sicuro e prevedibile. Al contrario, Objective-C è meno sicuro in termini di tipi e gestione degli errori. Ad esempio, in Objective-C, l'accesso a un puntatore nullo non causa un errore di runtime, ma restituisce semplicemente un valore nullo.

Performance

In termini di performance, Swift tende ad avere un vantaggio rispetto a Objective-C. Swift è stato progettato per essere veloce e utilizza un compilatore ottimizzato che produce codice binario efficiente. Inoltre, Swift supporta la programmazione parallela e concorrente, che può migliorare le performance per le operazioni di lunga durata o con un utilizzo intensivo di CPU. Al contrario, sebbene Objective-C sia generalmente veloce per la maggior parte delle operazioni, può essere più lento rispetto a Swift per alcune operazioni, in particolare quelle che richiedono calcoli intensivi.

[5]Smalltalk è un linguaggio di programmazione con un paradigma di programmazione riflessivo.

1.5.2 Confronto con altri linguaggi di programmazione mobile

Oltre a Objective-C, ci sono molti altri linguaggi di programmazione utilizzati per lo sviluppo di applicazioni mobile. Questi includono Java e Kotlin per Android, e linguaggi multipiattaforma come JavaScript (con framework come React Native e Cordova), Dart (con Flutter), e C# (con Xamarin). Ognuno di questi linguaggi ha i suoi vantaggi e svantaggi, e la scelta del linguaggio dipende dalle esigenze specifiche del progetto.

Java e Kotlin

Java è stato il linguaggio principale per lo sviluppo di applicazioni Android fino all'introduzione di Kotlin nel 2017. Kotlin, come Swift, offre una sintassi più moderna e concisa rispetto a Java, e include molte caratteristiche avanzate come il supporto per la programmazione funzionale e la null-safety. Tuttavia, a differenza di Swift con Objective-C, Kotlin è completamente interoperabile con Java, il che significa che i programmatori possono utilizzare liberamente le librerie Java esistenti nel loro codice Kotlin. Per altre caratteristiche sintattiche, invece, Swift e Kotlin risultano molto simili.

Linguaggi multipiattaforma

I linguaggi multipiattaforma come JavaScript, Dart e C# permettono di scrivere codice che può essere eseguito su più piattaforme, come iOS e Android. Ciò può ridurre il tempo e lo sforzo necessari per sviluppare e mantenere un'applicazione su più piattaforme. Tuttavia, queste soluzioni multipiattaforma possono avere svantaggi in termini di performance e accesso alle API native. Ad esempio, le applicazioni React Native o Flutter potrebbero non essere in grado di utilizzare le ultime API iOS o Android fino a quando non vengono aggiornate per supportarle. Inoltre, le applicazioni multipiattaforma possono avere performance inferiori rispetto alle applicazioni native, soprattutto per le operazioni di grafica intensiva o per le operazioni che richiedono un accesso diretto alle API del sistema operativo.

1.6 Ambienti di sviluppo

Ora che sono state analizzate le caratteristiche fondamentali di Swift e della sua sintassi, oltre ai suoi vantaggi e svantaggi, ci si può affacciare sulle possibilità che si hanno per lo sviluppo di applicazioni tramite Swift. Queste possibilità si concretizzano nei tre IDE[6] che si possono adottare: AppCode, Swift Playgrounds e Xcode.

1.6.1 AppCode

AppCode (Figura 1.3) è un IDE realizzato da JetBrains, l'azienda che ha creato popolari IDE come IntelliJ IDEA e PyCharm e che ha sviluppato il linguaggio di programmazione Kotlin. AppCode supporta diversi linguaggi di programmazione, tra cui Swift, Objective-C, C e C++.

Esso offre una serie di funzionalità che possono aiutare a migliorare la produttività dello sviluppatore, tra cui il rilevamento intelligente degli errori, il refactoring del codice, l'integrazione con Git e GitHub, e il supporto per il testing unitario. Inoltre, esso include un potente debugger e un profiler di performance.

Tuttavia, AppCode non è gratuito. JetBrains offre una licenza commerciale per AppCode, ma offre anche sconti per gli studenti e per le organizzazioni no-profit. Inoltre, JetBrains

[6] Integrated Development Environment

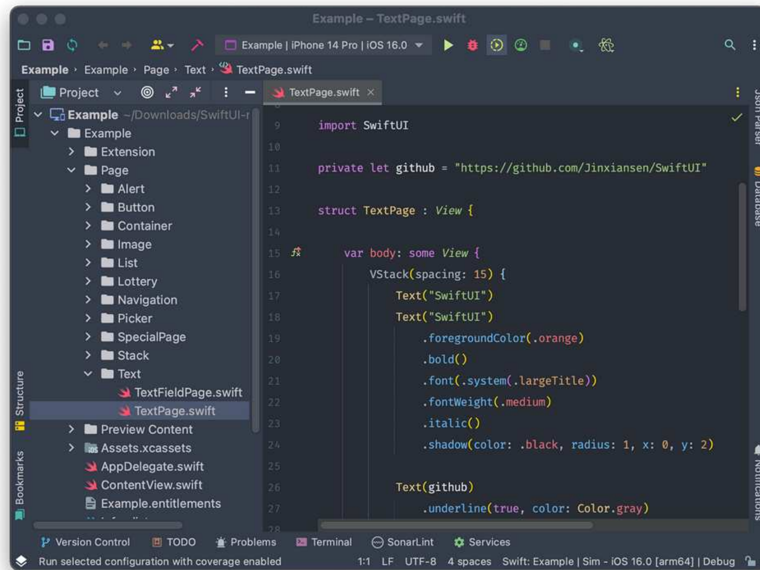

Figura 1.3: Esempio di una schermata di AppCode

ha ufficializzato la fine di AppCode il 14 dicembre 2022 annunciando che non verrà più aggiornato a partire dal 31 dicembre 2023.

1.6.2 Swift Playgrounds

Swift Playgrounds (Figura 1.4) è un'applicazione per iPad e Mac sviluppata da Apple che permette di scrivere codice Swift in un ambiente interattivo. Swift Playgrounds non è un vero e proprio IDE, infatti è stato progettato per essere un modo divertente e facile per imparare a programmare in Swift, con una serie di "playgrounds" interattivi che guidano l'utente attraverso vari concetti di programmazione e di sviluppo app.

Ogni playground presenta una serie di sfide di programmazione che l'utente deve risolvere scrivendo codice Swift. Mentre l'utente scrive il codice, può vedere immediatamente i risultati in un'area di visualizzazione interattiva.

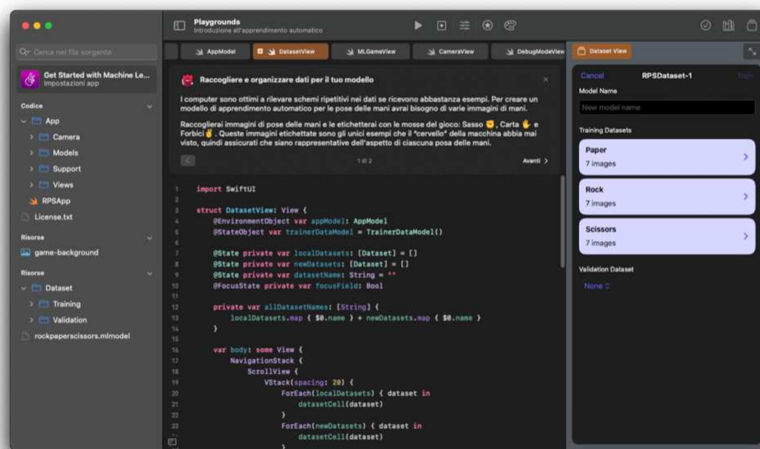

Figura 1.4: Esempio di una schermata di Swift Playgrounds

Inoltre, ogni utente ha la possibilità di creare i propri playground personalizzati in cui può mettere in atto le conoscenze acquisite creando delle app su Swift Playgrounds.

1.6.3 Xcode

Xcode (Figura 1.5) è l'IDE ufficiale di Apple per lo sviluppo di applicazioni per iOS, iPadOS, macOS, watchOS e tvOS. Xcode supporta sia Swift che Objective-C, e include una serie di strumenti per lo sviluppo di applicazioni, tra cui un editor di codice, un debugger, un profiler di performance, un designer di interfaccia utente e un gestore di pacchetti.

Xcode offre anche un'ampia integrazione con altri servizi di Apple, tra cui il Mac App Store, iCloud, Game Center, e TestFlight, per il testing beta delle applicazioni.

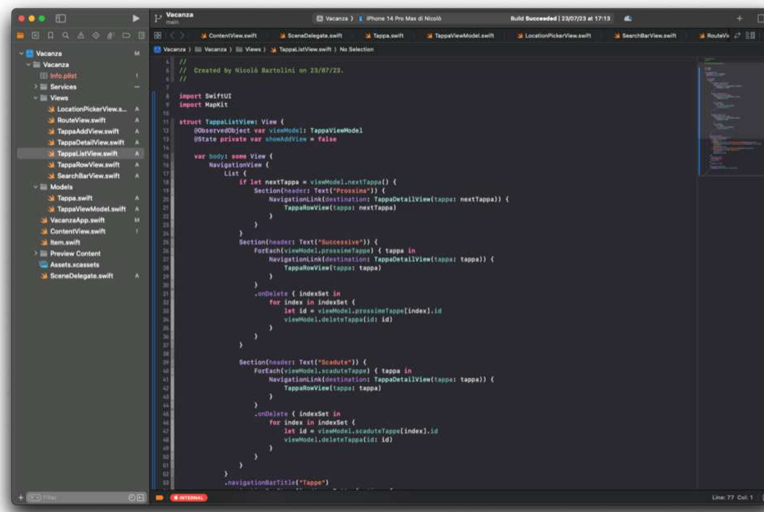

Figura 1.5: Esempio di una schermata di Xcode

Uno dei principali vantaggi di Xcode è il suo designer di interfaccia utente, chiamato Interface Builder, che permette di creare interfacce utente per le applicazioni tramite il trascinamento degli elementi UI su un canvas. Interface Builder supporta la creazione di interfacce utente sia con Auto Layout, che permette di creare interfacce utente che si adattano alle diverse dimensioni dello schermo, sia con SwiftUI (Sezione 1.7), il framework di Apple per la creazione di interfacce utente dichiarative.

Inoltre, Xcode include un emulatore di iOS che permette di testare le applicazioni su una varietà di dispositivi e versioni di iOS senza avere fisicamente quei dispositivi.

Xcode è gratuito e può essere scaricato dal Mac App Store e installato solo su macOS.

1.7 SwiftUI

SwiftUI è un framework di interfaccia utente innovativo introdotto da Apple nel 2019 che consente di progettare app in modo dichiarativo. Con SwiftUI si descrive (o dichiara) l'interfaccia utente che si desidera ottenere e SwiftUI si assicura che l'interfaccia utente corrisponda a quella descrizione. È un modo completamente rivoluzionario rispetto al modo precedente di costruire le interfacce utente e ha l'obiettivo di essere sia più semplice che più potente, con meno codice e una maggiore interattività.

1.7.1 Caratteristiche principali di SwiftUI

SwiftUI offre una serie di caratteristiche che lo rendono un'opzione attraente per lo sviluppo di interfacce utente:

- *Dichiaratività:* con SwiftUI, si descrive *cosa* si vuole, non *come* ottenerlo. Si definiscono l'interfaccia utente e il comportamento dell'app in termini di stato e di mutazioni di stato. SwiftUI si occupa di aggiornare l'interfaccia utente per riflettere tali modifiche.
- *Componibilità:* le interfacce utente in SwiftUI sono costruite componendo piccoli pezzi di codice riutilizzabile. Ciò rende più facile costruire interfacce utente complesse e personalizzate.
- *Consistenza:* SwiftUI offre un'interfaccia utente coerente su tutte le piattaforme Apple. Il codice si scrive una volta e si può eseguire su iOS, iPadOS, macOS, watchOS e tvOS.
- *Interattività:* SwiftUI offre potenti strumenti per l'interattività, tra cui animazioni e gesture.
- *Integrazione:* SwiftUI è profondamente integrato con Xcode, rendendo facile la progettazione, il debug e il testing delle interfacce utente.

Il codice nel Listato 1.24 mostra una semplice interfaccia utente SwiftUI che visualizza un testo con il messaggio "Esempio di SwiftUI" in caratteri grandi e di colore blu.

```
1  import SwiftUI
2
3  struct ContentView: View {
4      var body: some View {
5          Text("Esempio di SwiftUI")
6              .font(.largeTitle)
7              .foregroundColor(.blue)
8      }
9  }
10
11 struct ContentView_Previews: PreviewProvider {
12     static var previews: some View {
13         ContentView()
14     }
15 }
```

Listato 1.24: Esempio di codice SwiftUI

Nel Listato 1.24 è possibile notare che sono presenti due `struct`. La prima rappresenta la vera e propria interfaccia utente; essa contiene una descrizione di ciò che si vuole vedere, cioè un testo con scritto "Esempio di SwiftUI" con un font grande e un colore blu. La seconda `struct`, invece, è un'aggiunta opzionale che consente ad Xcode di visualizzare le modifiche che si effettuano alla prima in un'anteprima (chiamata canvas) che si aggiorna quasi in tempo reale, senza la necessità di effettuare il build dell'applicazione ogni volta che si effettuano delle modifiche.

1.7.2 Confronto con Storyboard

Storyboard è un altro strumento di Apple per la progettazione di interfacce utente ed è stato lo standard fino al rilascio di SwiftUI. È un editor visivo che consente di creare e modificare interfacce utente utilizzando un'interfaccia drag-and-drop. Mentre Storyboard è potente e flessibile, ha anche alcuni svantaggi rispetto a SwiftUI, che lo hanno reso una scelta secondaria rispetto a SwiftUI. Questi sono:

- *Imperatività:* con Storyboard si specifica come l'interfaccia utente dovrebbe cambiare in risposta a vari eventi. Con SwiftUI, invece, si descrive lo stato dell'interfaccia utente e come dovrebbe apparire in base a quello stato. Ciò rende SwiftUI più semplice e più intuitivo da usare.
- *Riutilizzo del codice:* con Storyboard, è difficile riutilizzare le parti dell'interfaccia utente tra diverse schermate o nell'ambito della stessa app. Con SwiftUI, invece, si possono creare componenti dell'interfaccia utente riutilizzabili che possono essere messi insieme per creare interfacce utente complesse.
- *Gestione dello stato:* la gestione dello stato può essere complicata con Storyboard. Con SwiftUI, invece, lo stato dell'interfaccia utente è una parte fondamentale del design dell'interfaccia stessa, rendendo più facile capire come lo stato dell'app influisce sull'interfaccia utente.
- *Compatibilità:* Storyboard è compatibile solo con iOS e tvOS, mentre SwiftUI è compatibile con tutte le piattaforme Apple.
- *Preview in tempo reale:* con SwiftUI, è possibile vedere una preview in tempo reale del layout mentre si scrive il codice; questo non è possibile con Storyboard.

Per riassumere questo breve confronto, mentre Storyboard offre un potente editor visivo per la creazione di interfacce utente, SwiftUI offre un approccio più moderno e dichiarativo alla progettazione dell'interfaccia utente che può risultare più semplice e intuitiva per molti sviluppatori.

1.8 Swift e lo sviluppo iOS

Swift è il linguaggio di programmazione ufficiale per lo sviluppo di applicazioni iOS. Apple ha progettato Swift per essere potente e facile da usare, con una serie di funzionalità che lo rendono ideale per lo sviluppo di applicazioni iOS. Queste funzionalità sono state analizzate nel dettaglio nel corso di questo capitolo. In quest'ultima sezione verranno analizzati alcuni concetti e librerie utili proprio per lo sviluppo di applicazioni iOS e che torneranno utili anche nel progetto relativo alla presente tesi.

1.8.1 SwiftData vs Core Data

SwiftData e Core Data sono due framework per la gestione dei dati nell'ambito della programmazione in linguaggio Swift. Questi due framework adottano entrambi l'Object-Relational Mapping (ORM[7]), fornendo strumenti efficaci per memorizzare, recuperare e modificare i dati all'interno di un'applicazione.

[7] L'Object-Relational Mapping è un paradigma di programmazione che consente una mappatura tra i dati mantenuti in un database e gli oggetti di un linguaggio di programmazione orientato agli oggetti.

SwiftData

SwiftData è un framework di più recente concezione rispetto a Core Data. È stato progettato con l'intento di semplificare l'esperienza dell'utente e aumentare l'efficienza della gestione dei dati. Inoltre, SwiftData è stato ottimizzato e integrato in maniera impeccabile con SwiftUI, offrendo un ambiente di lavoro altamente sinergico.

Nonostante la sua recente nascita, SwiftData ha già un solido insieme di vantaggi rispetto a Core Data, tra cui una facilità d'uso superiore, una maggiore efficienza e una migliore integrazione con altre tecnologie. Tuttavia, a fronte di questi vantaggi, SwiftData presenta anche alcuni limiti, tra i quali una minore gamma di funzionalità complesse rispetto a quelle offerte da Core Data e una minore maturità dovuta alla giovane età del framework.

Core Data

Core Data, al contrario, è un framework con una storia più lunga e consolidata rispetto a SwiftData, ed offre una serie di funzionalità più articolate e complete. La sua complessità, però, lo rende più difficile da utilizzare rispetto a SwiftData, e la sua integrazione con SwiftUI non è altrettanto ottimizzata.

Inoltre, Core Data è il framework standard de facto per le applicazioni che necessitano di una gestione sofisticata e avanzata dei dati, grazie alla sua vasta gamma di strumenti e funzionalità. SwiftData, invece, risulta più adatto per quelle applicazioni che richiedono una gestione dei dati più basilare, ma che puntano alla semplicità di utilizzo e all'efficienza del framework.

1.8.2 CloudKit

CloudKit è un framework di servizi cloud fornito da Apple che consente agli sviluppatori di integrare le funzionalità del cloud nelle loro applicazioni iOS. CloudKit fornisce API per l'autenticazione degli utenti, il salvataggio e la sincronizzazione dei dati dell'applicazione, l'invio di notifiche push e la condivisione di dati tra gli utenti.

CloudKit è strettamente integrato con l'ecosistema di Apple, il che significa che gli sviluppatori possono sfruttare le funzionalità di iCloud, come l'autenticazione degli utenti e la sincronizzazione dei dati, senza dover implementare tali funzionalità da soli. Inoltre, CloudKit supporta la programmazione asincrona e offre un modello di programmazione basato su *callback*, il che lo rende un buon complemento per Swift.

Un esempio di utilizzo di CloudKit potrebbe essere il salvataggio di un record di dati nel cloud, mostrato nel Listato 1.25.

1.8.3 Integrazione di Firebase in un progetto Swift

Firebase è una piattaforma di sviluppo di applicazioni mobile fornita da Google che offre una serie di servizi cloud, tra cui l'autenticazione degli utenti, il database in tempo reale, l'hosting, lo storage, le notifiche push e l'analisi delle applicazioni.

Firebase offre SDK per vari linguaggi di programmazione, tra cui Swift. Questo significa che gli sviluppatori Swift possono integrare facilmente le funzionalità di Firebase nelle loro applicazioni iOS.

Per integrare Firebase in un progetto Swift è necessario, innanzitutto, installare l'SDK di Firebase utilizzando un gestore di pacchetti come CocoaPods o Swift Package Manager. Dopo aver installato l'SDK, è possibile iniziare a utilizzare le API di Firebase per accedere ai suoi servizi.

Un esempio di utilizzo di Firebase in un progetto Swift potrebbe essere l'autenticazione degli utenti, mostrata nel Listato 1.26.

```swift
import CloudKit
let container = CKContainer.default()
let privateDatabase = container.privateCloudDatabase
let record = CKRecord(recordType: "User")
record["name"] = "Prova"
record["email"] = "prova@prova.it"
privateDatabase.save(record) { savedRecord, error in
    if let error = error {
        print("Errore durante il salvataggio del record: (error)")
    } else {
        print("Record salvato con successo")
    }
}
```

Listato 1.25: Esempio di codice CloudKit

```swift
import FirebaseAuth
FirebaseAuth.auth().createUser(
            withEmail: "prova@prova.it",
            password: "password") { authResult, error in
    if let error = error {
        print("Errore durante la creazione dell'utente: (error)")
    } else {
        print("Utente creato con successo")
    }
}
```

Listato 1.26: Esempio di codice Firebase

1.8.4 Richieste HTTP: la libreria Alamofire

Alamofire è una libreria Swift per la gestione delle richieste HTTP. È basata sul framework Foundation Networking di Apple e offre un'interfaccia di alto livello per la gestione delle richieste di rete, rendendo più semplice l'esecuzione di operazioni comuni come l'invio di richieste GET e POST, il download e l'upload di file e la gestione delle risposte JSON.

Un esempio di come si può utilizzare Alamofire per inviare una richiesta GET a un endpoint API è mostrato nel Listato 1.27.

```swift
import Alamofire
Alamofire.request("https://api.com/users").responseJSON { response in
    switch response.result {
        case .success(let value):
            print("Data: (value)")
        case .failure(let error):
            print("Error: (error)")
    }
}
```

Listato 1.27: Esempio di richiesta GET con Alamofire

Alamofire può anche essere utilizzato per inviare richieste POST, come mostrato nel Listato 1.28.

```swift
import Alamofire

let parameters: [String: Any] = [
    "name": "John Doe",
    "email": "john.doe@example.com"
]

Alamofire.request(
            "https://api.com/users",
            method: .post,
            parameters: parameters).responseJSON { response in
    switch response.result {
        case .success(let value):
            print("Data: (value)")
        case .failure(let error):
            print("Error: (error)")
    }
}
```

Listato 1.28: Esempio di richiesta POST con Alamofire

1.8.5 Gestione delle immagini: la libreria Kingfisher

Kingfisher è una libreria Swift per la gestione delle immagini. Essa fornisce un'interfaccia di alto livello per il download e la cache delle immagini, rendendo più semplice l'esecuzione di operazioni comuni come il download di immagini da un URL e la loro visualizzazione in una view di immagine.

Un esempio di come si può utilizzare Kingfisher per scaricare un'immagine da un URL e visualizzarla in una view è mostrato nel Listato 1.29.

```swift
import Kingfisher

let url = URL(string: "https://example.com/image.jpg")
let imageView = UIImageView()

imageView.kf.setImage(with: url)
```

Listato 1.29: Esempio di utilizzo di Kingfisher

CAPITOLO 2

Specifica e analisi dei requisiti

Nel presente capitolo, viene fornita una dettagliata analisi dei requisiti per la realizzazione di un'applicazione mobile per iOS, avente lo scopo di fungere da social network per gli amanti dei film (e delle serie TV). Questo capitolo costituisce il punto di partenza per lo sviluppo dell'intero progetto e si articola in diverse sezioni che forniscono le fondamenta per la successiva progettazione e implementazione. In particolare, verrà innanzitutto effettuata un'analisi del contesto in cui dovrà operare l'applicazione, passando poi ad un'analisi dei requisiti funzionali e non funzionali. Successivamente, verranno effettuate delle analisi comparative con dei prodotti già esistenti nel campo, al fine di identificare le caratteristiche chiave di tali soluzioni e di individuare le opportunità di miglioramento. Infine, verranno definiti i casi d'uso dell'applicazione e verrà stilata una matrice di mapping dei requisiti per chiarire la correlazione tra i casi d'uso identificati e i requisiti specifici dell'applicazione.

2.1 Definizione del problema

2.1.1 Contesto

Nella società contemporanea, la passione per i film e le serie TV è diventata una componente essenziale della vita quotidiana di molte persone. Le piattaforme di streaming online e le numerose produzioni cinematografiche e televisive hanno reso accessibile un'ampia varietà di contenuti di intrattenimento. Tuttavia, l'abbondanza di opzioni può rendere difficile per gli appassionati gestire e organizzare le proprie preferenze, perdendo di vista quali contenuti siano in programma per il futuro, quali episodi di una serie TV siano già stati visti oppure quali sono i dettagli positivi o negativi trovati durante la visione di un prodotto.

Con tale contesto in mente, il presente progetto si pone l'obiettivo di sviluppare un'applicazione iOS dedicata agli amanti dei film e delle serie TV. L'applicazione consentirà agli utenti di scoprire e condividere contenuti cinematografici e televisivi, fornendo una piattaforma interattiva e sociale per esprimere le proprie opinioni, creare liste personalizzate di contenuti preferiti e seguire le serie TV in modo organizzato, tenendo traccia degli episodi già guardati e dei prossimi da guardare.

2.1.2 Obiettivi del progetto

Come già accennato nella sezione precedente, l'obiettivo principale del progetto è lo sviluppo di un'applicazione mobile iOS in grado di fornire le seguenti funzionalità principali:

- Visualizzazione dei dettagli completi di film e serie TV, inclusi trame, cast, stagioni ed episodi e altri dettagli secondari.

- Visualizzazione dei dettagli completi di persone famose nel mondo del cinema (attori, registi, sceneggiatori, etc.).
- Possibilità di lasciare valutazioni e recensioni ai prodotti e di visualizzare le recensioni e le valutazioni di altri utenti, nonché di condividerle internamente o esternamente all'applicazione.
- Popolamento di liste predefinite (watchlist, preferiti, etc.) da riempire con prodotti specifici e creazione di liste personalizzate.
- Possibilità di seguire ed interagire con altri utenti, specificatamente attraverso l'invio di consigli.
- Possibilità di tracciamento degli episodi delle serie TV in visione.
- Personalizzazione del profilo personale in base ai prodotti preferiti.

2.1.3 Nome dell'applicazione

Il nome di un'applicazione è una componente molto importante che deve rappresentare il contenuto e le funzionalità offerte da essa in modo rapido ed efficace. Per l'applicazione del presente progetto è stato scelto come nome *WatchWise*.

WatchWise combina due elementi chiave dell'applicazione: *Watch* e *Wise*. Il termine Watch si riferisce all'azione di guardare film e serie TV, evidenziando il focus principale dell'applicazione sulla fruizione di contenuti cinematografici e televisivi. Allo stesso tempo, il termine Wise suggerisce l'idea di saggezza, intelligenza e consapevolezza.

L'obiettivo dell'applicazione è fornire una piattaforma che aiuti gli utenti a prendere decisioni informate riguardo ai contenuti da guardare, offrendo dettagli, organizzazione e possibilità di lasciare e leggere recensioni e valutazioni aggiornate. Pertanto, WatchWise invita gli utenti a essere saggi nella loro scelta di cosa guardare, guidandoli verso contenuti che potrebbero essere di loro interesse e soddisfare le loro preferenze.

2.1.4 Glossario di progetto

In questa sezione, vengono presentati i termini chiave e il glossario di progetto per l'applicazione WatchWise. Al fine di garantire una comprensione chiara e univoca dei concetti e delle funzionalità dell'applicazione, vengono fornite le definizioni dei termini più rilevanti utilizzati all'interno del progetto. Tale glossario rappresenta un punto di riferimento essenziale per gli utilizzatori dell'applicazione, promuovendo una comunicazione chiara e coerente durante tutte le fasi del processo di sviluppo. Il glossario di progetto è mostrato nella Tabella 2.1.

Termine	Definizione	Sinonimi
Backdrop	Un'immagine di sfondo utilizzata per fornire contesto visivo a un prodotto.	Immagine di sfondo
Cast	Il gruppo di attori che hanno partecipato alla realizzazione di un prodotto.	Personaggi
Crew	Il gruppo di individui coinvolto nella produzione di un prodotto, come sceneggiatori, registi, e altri membri dello staff tecnico.	Squadra tecnica, Staff

Continua nella pagina successiva

Tabella 2.1 – continua dalla pagina precedente

Termine	Definizione	Sinonimi
Episodio	Contenuto facente parte di una serie TV, costituente un prodotto a se stante all'interno dell'insieme più ampio della serie TV, con una trama unica o collegata agli altri episodi.	Puntata
Feed	La sezione dell'applicazione che mostra i contenuti recenti degli utenti seguiti.	Attività
Film	Un prodotto cinematografico, costituito da una singola opera audiovisiva.	Prodotto, Lungometraggio
Follower	Un utente che segue un altro utente per ricevere gli aggiornamenti dei suoi contenuti.	Seguace
Home	La schermata principale dell'applicazione che presenta i film attualmente in onda, i film e le serie TV con una valutazione più alta fornita dagli utenti.	Pagina iniziale
In onda	Un elenco di film attualmente in programmazione nei cinema.	Nessuno
Lista	Un insieme di prodotti selezionati dall'utente, organizzati secondo una categoria specifica.	Nessuno
Login	Il processo attraverso il quale un utente accede al proprio account nell'applicazione.	Accesso
Persona	Individuo coinvolto nella produzione dei prodotti, ad esempio un attore o un membro della crew.	Attore, Regista, Creatore
Piattaforma	Un servizio o un'azienda che offre contenuti cinematografici o televisivi.	Servizio di streaming
Poster	L'immagine promozionale principale di un prodotto.	Locandina
Prodotto	Un contenuto audiovisivo, che può essere un film o una serie TV.	Contenuto
Profilo	La schermata dell'utente corrente, che include le sue informazioni, le sue liste e altri dettagli personalizzati.	Account
Recensione	Una considerazione testuale scritta dall'utente riguardo a un prodotto.	Commento
Registrazione	Il processo attraverso il quale un nuovo utente crea un account nell'applicazione.	Iscrizione
Ricerca	La schermata di esplorazione/ricerca che contiene una barra di ricerca, film e serie TV popolari e di tendenza.	Esplora
Serie TV	Una serie di contenuti audiovisivi organizzati in episodi collegati tra loro, presentati in sequenza e appartenenti a una singola narrativa o tema.	Prodotto, Serie, Show

Continua nella pagina successiva

Tabella 2.1 – continua dalla pagina precedente

Termine	Definizione	Sinonimi
Stagione	Un gruppo di episodi di una serie TV presentati nella stessa fase di produzione.	Nessuno
Utente	Un individuo registrato e autenticato nell'applicazione, avente un proprio profilo.	Utente registrato, Utilizzatore
Valutazione	Un punteggio numerico assegnato da un utente per esprimere il suo giudizio su un prodotto.	Voto

Tabella 2.1: Glossario di progetto

2.2 Analisi dei requisiti

Lo sviluppo di qualsiasi progetto software richiede un'analisi molto dettagliata dei requisiti necessari all'applicativo. Nella fase di analisi dei requisiti è dunque molto importante delineare in modo chiaro e dettagliato l'elenco delle funzionalità che dovrà soddisfare l'applicazione e dei vincoli che essa dovrà rispettare. I requisiti sono generalmente divisi in due categorie principali: requisiti funzionali e requisiti non funzionali. I requisiti funzionali definiscono le funzionalità concrete che dovranno essere implementate dall'applicazione; i requisiti non funzionali, invece, riguardano i vincoli e le caratteristiche che determineranno la qualità, l'usabilità, le prestazioni e lo sviluppo stesso dell'applicazione.

In questa sezione saranno identificati e analizzati in dettaglio i requisiti funzionali e non funzionali dell'applicazione WatchWise.

2.2.1 Requisiti funzionali

Relativamente ai requisiti funzionali, sono state delineate una serie di funzionalità chiave che l'applicazione WatchWise dovrà fornire. Le seguenti caratteristiche, dunque, costituiscono le funzionalità di base che dovranno essere offerte:

RF1: L'applicazione dovrà permettere di creare un account utente e di effettuarvi il login.

RF2: L'applicazione dovrà fornire degli elenchi di film e serie TV ottenuti dall'API di TMDB.
(L'applicazione dovrà mostrare, nello specifico, una lista di film attualmente in onda al cinema e delle liste di film e serie TV popolari e di tendenza restituiti dall'API di TMDB)

RF3: L'applicazione dovrà permettere di visualizzare una schermata di informazioni per ogni film contenente i suoi dettagli, oltre alla lista del cast.

RF4: L'applicazione dovrà permettere di visualizzare una schermata di informazioni per ogni serie TV contenente i suoi dettagli, oltre alla lista del cast e a quella di stagioni ed episodi.

RF5: L'applicazione dovrà permettere di visualizzare una schermata di informazioni per ogni persona contenente i suoi dettagli, oltre ad una lista dei prodotti più importanti in cui ha partecipato.

RF6: L'applicazione dovrà permettere di ricercare film, serie TV, personaggi e altri utenti.

RF7: L'applicazione dovrà permettere di aggiungere un film o una serie TV alla lista dei film guardati o delle serie TV in visione.

RF8: L'applicazione dovrà permettere di aggiungere un film o una serie TV alla lista dei film preferiti o delle serie TV preferite.

RF9: L'applicazione dovrà permettere di aggiungere un film o una serie TV alla watchlist dei film o quella delle serie TV.

RF10: L'applicazione dovrà permettere di creare delle liste personalizzate e di popolarle con film o serie TV.

RF11: L'applicazione dovrà permettere di valutare un film o una serie TV su una scala che va da 0.5 a 5 stelle.

RF12: L'applicazione dovrà permettere di recensire un film o una serie TV.

RF13: L'applicazione dovrà permettere di visualizzare tutte le recensioni per un dato film o serie TV.

RF14: L'applicazione dovrà permettere di visualizzare una pagina profilo per ogni utente in cui sono mostrate le informazioni principali e le statistiche sull'utente, le sue liste e gli elenchi di follower e utenti seguiti.

RF15: L'applicazione dovrà permettere agli utenti di seguire altri utenti.

RF16: L'applicazione dovrà mostrare una schermata di feed contenente funzionalità sociali, come un centro notifiche contenente i consigli di prodotto inviati da seguaci, recensioni degli utenti seguiti o nuovi prodotti relativi a persone seguite.

RF17: L'applicazione dovrà fornire una lista delle persone seguite.

RF18: L'applicazione dovrà permettere di tracciare gli episodi delle serie TV in visione mostrando il prossimo episodio da vedere in base agli episodi segnalati come visti.

RF19: L'applicazione dovrà permettere agli utenti di modificare il proprio profilo.

RF20: L'applicazione dovrà permettere agli utenti di inviare consigli riguardanti prodotti ad altri utenti.

RF21: L'applicazione dovrà mostrare una classifica di film e serie TV con una media di valutazione più alta fornita dagli utenti dell'app.

RF22: L'applicazione dovrà permettere agli utenti di effettuare il reset della propria password, qualora la dimenticassero.

RF23: L'applicazione dovrà permettere la ricezione di notifiche push per aggiornamenti importanti, come nuove stagioni di serie TV preferite oppure nuovi follower.

RF24: L'applicazione dovrà permettere di condividere recensioni effettuate in WatchWise anche all'esterno dell'app stessa.

Questi requisiti definiscono gli obiettivi funzionali del software, rappresentando anche una sorta di roadmap per lo sviluppo e la successiva fase di progettazione delle funzionalità dell'applicazione.

2.2.2 Requisiti non funzionali

Per quanto concerne i requisiti non funzionali, sono stati identificati una serie di aspetti e vincoli che guideranno l'implementazione delle funzionalità dell'applicazione. Questi includono:

RNF1: L'applicazione dovrà essere sviluppata con l'IDE Xcode.

RNF2: L'applicazione dovrà essere sviluppata utilizzando principalmente il linguaggio di programmazione Swift e il framework SwiftUI.

RNF3: L'interfaccia utente dell'applicazione dovrà seguire le linee guida stilistiche di Apple per iOS[1], al fine di fornire un'esperienza utente coerente con l'ecosistema iOS.

RNF4: L'applicazione dovrà garantire un'esperienza utente ottimale, garantendo tempi di risposta rapidi, animazioni fluide e un'interfaccia utente intuitiva.

RNF5: L'applicazione dovrà usare Firebase Authentication come sistema di autenticazione e Cloud Firestore come database, al fine di garantire sicurezza e protezione per i dati dell'utente.

RNF6: L'applicazione dovrà supportare le versioni di iOS più recenti e, per massimizzare la base utenti potenziale, dovrebbe essere compatibile con versioni precedenti di iOS, almeno a partire da iOS 14.

RNF7: L'applicazione dovrà gestire correttamente le richieste all'API di TMDB, utilizzando la libreria Alamofire (Sezione 1.8.4) per effettuare chiamate RESTful e per gestire la ricezione e l'elaborazione dei dati provenienti dal server di TMDB.

RNF8: L'applicazione dovrà supportare diverse lingue e regioni, fornendo una traduzione adeguata del contenuto dell'applicazione per raggiungere un pubblico globale. (Inizialmente verranno supportate le lingue Italiano e Inglese e le regioni IT e US, ma l'approccio di sviluppo dovrà garantire una facile scalabilità di tale funzione)

RNF10: L'applicazione dovrà fornire tempi di risposta rapidi, riducendo al minimo i tempi di caricamento e fornendo una risposta immediata alle azioni dell'utente.

RNF11: L'applicazione dovrà essere stabile e affidabile, evitando crash e malfunzionamenti e garantendo una corretta gestione degli errori.

I requisiti non funzionali influenzeranno le decisioni progettuali e implementative nel corso dello sviluppo, garantendo che il software soddisfi gli standard di qualità, prestazioni e usabilità. Essi garantiscono che il software sia non solo funzionale, ma anche efficiente, sicuro, facile da usare e affidabile.

2.3 Analisi della concorrenza

In questa sezione, verranno analizzate due app concorrenti e un sito che operano nel campo dei social network per gli amanti di film e serie TV. Tale analisi permetterà di comprendere meglio le caratteristiche distintive di ognuna di esse, i punti di forza e le eventuali lacune, al fine di trarre spunti e ispirazione per il progetto WatchWise.

[1] `https://developer.apple.com/design/human-interface-guidelines`

2.3.1 Analisi di TVTime

TVTime (la cui schermata principale è mostrata in Figura 2.1) è un'applicazione dedicata principalmente agli appassionati di serie TV, che offre una vasta gamma di funzionalità per tenere traccia delle serie TV in visione, condividere recensioni, commenti e scambiare opinioni con altri utenti. Le principali caratteristiche di TVTime includono:

- *Tracciamento degli episodi:* gli utenti possono segnare gli episodi delle serie TV che hanno visto e l'app mostrerà i prossimi episodi da vedere (Figura 2.1). TVTime include anche una funzionalità che rende possibile gestire lo stato di visione delle serie TV ("in pausa", "guarda più tardi", etc.) e di segnare le serie TV e gli episodi come visti più volte.
- *Agenda delle serie:* TVTime fornisce agli utenti un calendario con le date di uscita degli episodi delle serie TV seguite, in modo da poter essere sempre aggiornati.
- *Comunità di utenti:* gli utenti possono interagire con altri appassionati di serie TV, condividere recensioni, commenti e partecipare a discussioni sulle serie TV preferite.
- *Classifiche delle serie:* l'app offre una classifica delle serie TV più popolari e meglio valutate dagli utenti della piattaforma.
- *Valutazioni e statistiche sugli episodi*: TVTime consente di recensire e valutare gli episodi delle serie TV singolarmente, fornendo, inoltre, la possibilità di votare il personaggio migliore dell'episodio, segnalare lo stato d'animo alla fine della visione e anche indicare la piattaforma utilizzata per la visione.
- *Film:* dopo uno degli ultimi aggiornamenti, l'app ha introdotto la possibilità di monitorare anche i film, nonostante questa funzione non sia sviluppata tanto quanto quella relativa alle serie TV.

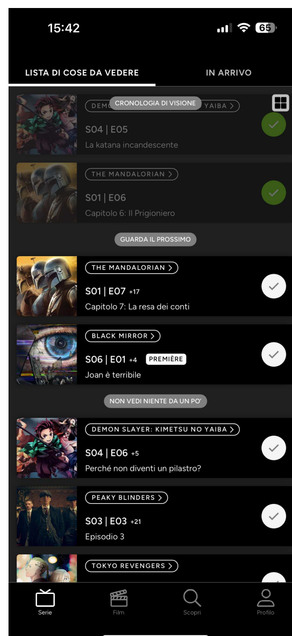

Figura 2.1: Pagina di tracciamento degli episodi dell'app TVTime

2.3.2 Analisi di TMDB

TMDB (The Movie Database) è una piattaforma online (Figura 2.2) che fornisce una vasta raccolta di informazioni su film, serie TV e attori. La piattaforma offre un'API che consente alle applicazioni di accedere ai suoi dati e fornire dettagli aggiornati e pertinenti sul mondo del cinema e delle serie TV. Alcune caratteristiche chiave di TMDB includono:

- *Informazioni dettagliate:* TMDB offre un vasto database contenente dettagli accurati su film, serie TV, attori, registi e altro ancora, in molteplici lingue.
- *Immagini e trailer:* la piattaforma offre una vasta gamma di immagini, poster e trailer relativi ai film e alle serie TV presenti nel database.
- *Raccomandazioni personalizzate:* TMDB offre funzionalità di raccomandazione che offrono una lista di prodotti consigliati basata su film e serie TV specifici.

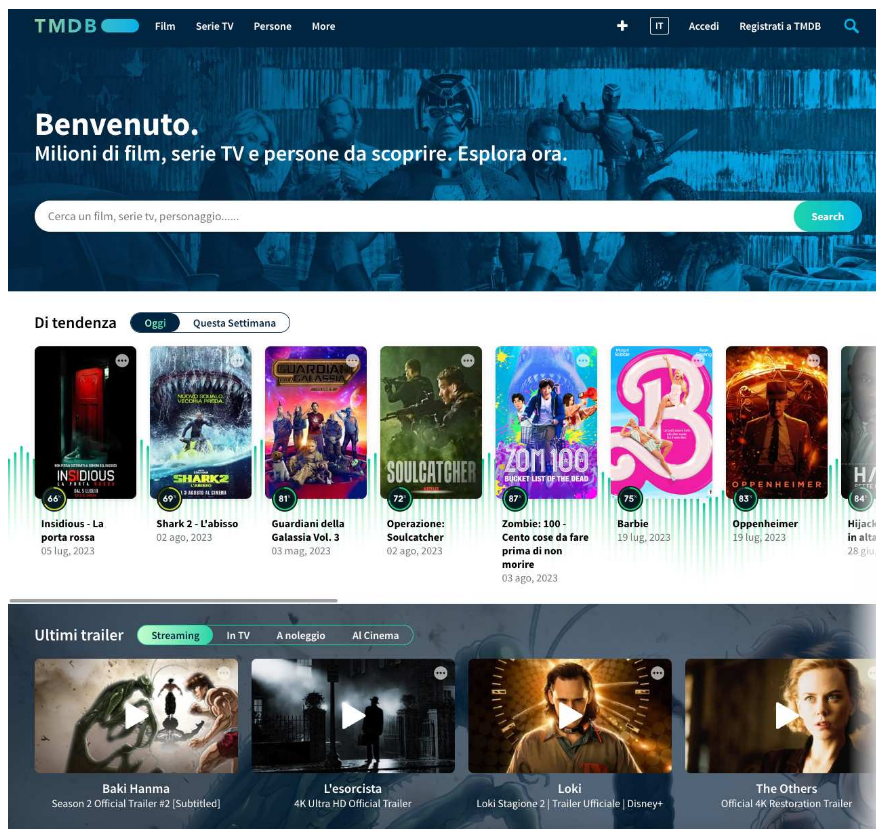

Figura 2.2: Pagina principale di TMDB (`https://www.themoviedb.org/`)

2.3.3 Analisi di Letterboxd

Letterboxd (di cui una schermata di esempio è mostrata in Figura 2.3) è un'applicazione sociale dedicata agli amanti del cinema, con un'attenzione particolare ai film. Tale piattaforma consente agli utenti di scoprire nuovi film, segnare i film visti e da vedere, scrivere recensioni e interagire con altri cinefili. Le principali caratteristiche di Letterboxd includono:

- *Diario dei film:* gli utenti possono segnare i film che hanno visto o che desiderano vedere in un diario personale, tenendo traccia delle proprie esperienze cinematografiche.

- *Recensioni e voti:* gli utenti possono scrivere recensioni sui film visti e assegnare loro un voto, condividendo le proprie opinioni con la comunità.
- *Feed di attività:* gli utenti possono seguire altri utenti e visualizzare il loro feed di attività, cioè l'insieme delle recensioni scritte, i voti assegnati e i film segnati come visti.
- *Liste personalizzate:* Letterboxd offre agli utenti la possibilità di creare liste personalizzate di film, come liste dei preferiti, classifiche tematiche o liste di film da vedere in base a specifici criteri e di condividerle al pubblico (Figura 2.3).
- *Journal:* l'app consente agli utenti di creare una sorta di blog personale in cui discutere pubblicamente dei film che hanno visto.

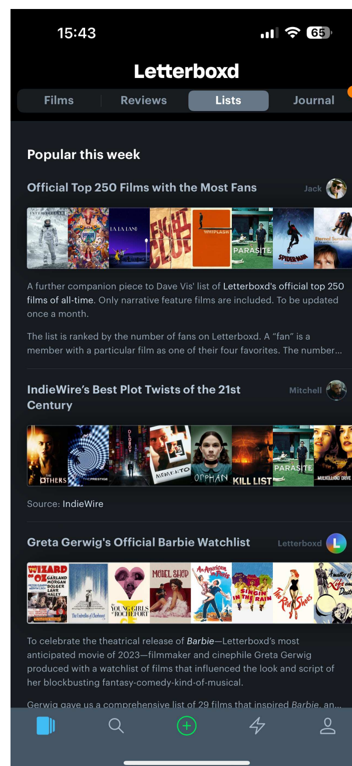

Figura 2.3: Pagina di condivisione delle liste dell'app Letterboxd

Confronto e considerazioni

Analizzando queste soluzioni concorrenti, possiamo osservare diverse funzionalità comuni, come la possibilità di scrivere recensioni e votare film o serie TV, e l'interazione con una comunità di utenti. Tuttavia, alcune differenze significative emergono tra le risorse analizzate.

TVTime si concentra principalmente sulle serie TV, fornendo una funzionalità di agenda delle serie e mettendo in contatto un'ampia community di appassionati che commentano e valutano gli episodi singolarmente. TMDB, d'altra parte, si focalizza sulla fornitura di dati e informazioni complete sul cinema e le serie TV, tramite la sua API (che verrà utilizzata nel progetto della presente tesi). Letterboxd, infine, si distingue per la sua enfasi sui film e per il diario personale che permette agli utenti di creare una sorta di blog incentrato sui lungometraggi.

Il progetto della presente tesi, WatchWise, come si è potuto evincere dai requisiti analizzati nella Sezione 2.2, avrà l'obiettivo di combinare le migliori caratteristiche delle app concorrenti,

offrendo una piattaforma completa per gli amanti di film e serie TV, senza focalizzarsi su uno dei due ambiti. Verranno fornite funzionalità per il tracciamento degli episodi delle serie TV, la scrittura di recensioni per film e serie TV, la creazione di liste personalizzate e l'interazione con altri utenti attraverso un feed di attività e la possibilità di seguire utenti e ricevere notifiche.

Nonostante le funzionalità da implementare fossero già state definite nella Sezione 2.2, tale analisi ha fornito una panoramica utile per definire maggiormente in che modo implementare le funzionalità già esplicate.

2.4 Casi d'uso

Nella fase di analisi dei requisiti, lo step finale consiste nella definizione rigorosa dei diagrammi dei casi d'uso dell'applicazione a partire dai requisiti precedentemente analizzati. I diagrammi dei casi d'uso rappresentano e sono composti da due elementi fondamentali: attori e casi d'uso.

2.4.1 Definizione degli attori

Gli attori rappresentano le diverse entità che svolgono un ruolo significativo nell'utilizzo dell'applicazione. Ogni attore avrà un'interazione specifica con il sistema e svolgerà un ruolo chiave nel raggiungimento degli obiettivi dell'applicazione.

Di seguito vengono presentati gli attori dell'applicazione WatchWise identificati:

- *Utente non autenticato*: è l'utilizzatore dell'applicazione WatchWise che non ha ancora creato un account utente o non ha effettuato l'accesso allo stesso. Tale attore non ha a disposizione le funzionalità base dell'applicazione, ma può soltanto creare un account o effettuare il login ad un account esistente.

- *Utente autenticato*: è il principale fruitore dell'applicazione WatchWise. Rappresenta un utente che ha effettuato l'accesso all'applicazione tramite il proprio account personale. Questo attore, al contrario del precedente, avrà la possibilità di accedere a tutte le funzionalità dell'applicazione.

2.4.2 Definizione dei casi d'uso

I casi d'uso descrivono in modo dettagliato le interazioni tra gli attori e il sistema, fornendo una visione chiara e completa delle funzionalità che l'applicazione dovrà offrire. Ogni caso d'uso rappresenta uno scenario specifico in cui gli attori interagiscono con il sistema per raggiungere un obiettivo particolare.

In questa sezione verrà presentata una lista esaustiva dei casi d'uso identificati. I vari casi d'uso sono stati suddivisi in 7 categorie diverse.

Autenticazione e registrazione

CU1 - Login Questo caso d'uso consente all'utente non autenticato di accedere all'applicazione fornendo le proprie credenziali di accesso (email e password) o accedendo tramite Google. Il sistema verificherà l'identità dell'utente tramite Firebase Authentication e, se le credenziali sono corrette, gli permetterà di accedere all'interfaccia come utente autenticato. In caso di credenziali errate o di utente non registrato, verrà visualizzato un messaggio di errore appropriato.

CU2 - Registrazione Questo caso d'uso consente all'utente non autenticato di registrarsi all'applicazione fornendo le informazioni necessarie. Il sistema verificherà la validità delle informazioni inserite e creerà un nuovo account utilizzando Firebase Authentication. Successivamente, l'utente potrà accedere all'applicazione come utente autenticato attraverso il caso d'uso CU1.

CU3 - Recupero password Questo caso d'uso consente all'utente non autenticato di recuperare la propria password qualora l'avesse dimenticata. L'utente inserirà il proprio indirizzo email e, qualora esso corrisponda ad un account registrato in Firebase Authentication, il sistema provvederà ad inviare una mail all'utente contenente un link per il reset della password.

Esplorazione dei contenuti

CU4 - Visualizzazione elenchi film e serie TV Questo caso d'uso consente all'utente autenticato di visualizzare degli elenchi di film e serie TV o di ricercare prodotti o persone. L'utente potrà raggiungere una tra le apposite schermate dell'applicazione atte all'esplorazione di prodotti dove egli potrà visualizzare i suddetti elenchi o inserire un parametro di ricerca. Il sistema provvederà a popolare gli elenchi attraverso apposite chiamate all'API di TMDB. Le schermate di esplorazione fornite dall'applicazione saranno la schermata di ricerca, la Home Page e la schermata Feed. In quest'ultima, saranno presenti anche funzionalità sociali, tra cui un centro notifiche e una lista di informazioni degli utenti seguiti.

CU5 - Visualizzazione film attualmente al cinema Questo caso d'uso consente all'utente autenticato di visualizzare un elenco di film attualmente in onda nei cinema della regione selezionata. L'utente potrà visualizzare tale elenco nella Home Page e il sistema provvederà a popolarlo con una chiamata all'endpoint `/now_playing` dell'API di TMDB.

CU6 - Visualizzazione dettagli film Questo caso d'uso consente all'utente autenticato di visualizzare i dettagli completi di un film selezionato attraverso una delle modalità fornite dall'applicazione. L'applicazione recupererà i dati relativi al film attraverso un'apposita chiamata all'API di TMDB e mostrerà all'utente una schermata con le informazioni principali del film, tra cui titolo, durata, trama, regista, cast e altro. L'utente avrà, anche, la possibilità di aggiungere il film ad una delle proprie liste, oppure di lasciare un voto o una recensione.

CU7 - Visualizzazione dettagli serie TV Questo caso d'uso consente all'utente autenticato di visualizzare i dettagli completi di una serie TV selezionata attraverso una delle modalità fornite dall'applicazione. L'applicazione recupererà i dati relativi alla serie attraverso un'apposita chiamata all'API di TMDB e mostrerà all'utente una schermata con le informazioni principali della serie, tra cui titolo, durata, trama, regista, cast e altro. Il sistema mostrerà, inoltre, una sezione contenente la lista di stagioni ed episodi della serie TV, in cui l'utente potrà segnare gli episodi come visti. L'utente avrà, anche, la possibilità di aggiungere la serie TV ad una delle proprie liste, oppure di lasciare un voto o una recensione.

CU8 - Visualizzazione dettagli persona Questo caso d'uso consente all'utente autenticato di visualizzare i dettagli completi di una persona selezionata attraverso una delle modalità fornite dall'applicazione. Il sistema recupererà i dati relativi alla persona attraverso un'apposita chiamata all'API di TMDB e mostrerà all'utente una schermata con le informazioni principali della persona, tra cui nome, data e luogo di nascita, biografia e una breve lista dei prodotti più famosi a cui ha partecipato.

Interazioni sociali

CU9 - Follow di un utente Questo caso d'uso consente all'utente autenticato di iniziare a seguire un altro utente. L'utente autenticato raggiungerà la pagina profilo dell'utente che vuole seguire attraverso una delle modalità fornite dall'applicazione e cliccando su un apposito pulsante potrà iniziare a seguire l'utente (o smetterlo di seguirlo).

CU10 - Invio di un consiglio Questo caso d'uso consente all'utente autenticato di inviare un prodotto come consiglio ad uno degli utenti che segue. L'utente raggiungerà la schermata di dettaglio del prodotto da consigliare e, attraverso un apposito pulsante, potrà inviare tale prodotto come consiglio ad uno degli utenti che segue.

CU11 - Ricezione notifiche Questo caso d'uso consente all'utente autenticato di ricevere notifiche push riguardanti nuovi seguaci o nuovi prodotti che potrebbero interessargli. Quando l'utente verrà seguito da un altro utente, un sistema in backend di notifiche push si occuperà di inviare la notifica all'utente autenticato. Un discorso analogo avverrà quando verrà rilasciato un nuovo prodotto che potrebbe interessare all'utente autenticato.

CU12 - Condivisione esterna di una recensione Questo caso d'uso consente all'utente autenticato di condividere esternamente una recensione di un prodotto. L'utente autenticato visualizzerà la lista delle recensioni di un prodotto e potrà cliccare su un apposito pulsante per condividere esternamente il testo e l'autore della recensione.

Gestione delle liste

CU13 - Aggiunta di un prodotto in una delle corrispondenti liste predefinite Questo caso d'uso consente all'utente autenticato di aggiungere un prodotto ad una delle liste predefinite dell'applicazione ("Guardati/In visione", "Preferiti", "Watchlist"). L'utente raggiungerà la pagina di dettaglio del prodotto attraverso una delle modalità fornite dall'applicazione e potrà cliccare sul pulsante di aggiunta alla lista desiderata. Il sistema aggiungerà l'ID del prodotto alla corrispettiva lista e informerà l'utente del successo con un messaggio di conferma.

CU14 - Aggiunta di una serie TV nella lista "In visione" Questo caso d'uso consente all'utente autenticato di aggiungere una serie TV alla lista "In visione". L'utente aggiungerà la serie TV così come nel caso d'uso CU13. In questo caso, però, il sistema provvederà anche ad aggiungere la serie TV alla lista delle serie TV per il monitoraggio degli episodi.

CU15 - Inserimento di una serie TV nella lista "Completate" Questo caso d'uso consente all'utente autenticato di aggiungere indirettamente una serie TV alla lista delle serie TV completate. Quando l'utente segnerà tutti gli episodi (esclusi gli episodi speciali) di una serie TV come visti, il sistema provvederà a spostare la serie TV dalla lista "In visione" alla lista "Completate".

CU16 - CRUD lista personalizzata Questo caso d'uso consente all'utente autenticato di creare, visualizzare, modificare ed eliminare una lista personalizzata. Attraverso opportune sezioni dell'applicazione l'utente sarà in grado di effettuare queste operazioni sulle liste personalizzate e il sistema provvederà ad effettuare tali modifiche nel database.

CU17 - Aggiunta di un prodotto in una lista personalizzata Questo caso d'uso consente all'utente autenticato di aggiungere un prodotto ad una delle sue liste personalizzate. L'utente raggiungerà la pagina di dettaglio del prodotto attraverso una delle modalità fornite dall'applicazione e cliccherà sul pulsante di aggiunta ad una lista personalizzata. Il sistema mostrerà all'utente l'elenco delle sue liste personalizzate e l'utente selezionerà la lista in cui vuole aggiungere il prodotto. Il sistema provvederà ad aggiungere il prodotto alla lista e mostrerà un messaggio di conferma all'utente.

Valutazioni e recensioni

CU18 - Valutazione prodotto Questo caso d'uso consente all'utente autenticato di valutare un prodotto. L'utente raggiungerà la pagina di dettaglio del prodotto desiderato, dove troverà uno slider sotto forma di stelle in cui inserire la propria valutazione (da 0.5 a 5 stelle). L'utente inserirà la sua valutazione e cliccherà su un apposito pulsante. Il sistema provvederà a salvare la valutazione e ad aggiornare la media di voto del prodotto. Infine, il sistema avviserà l'utente del successo con un messaggio di conferma.

CU19 - Recensione prodotto Questo caso d'uso consente all'utente autenticato di recensire un prodotto. L'utente raggiungerà la pagina di dettaglio del prodotto desiderato, dove troverà una casella di testo in cui scrivere la propria recensione. L'utente scriverà la propria recensione e cliccherà su un apposito pulsante. Il sistema provvederà a salvare la recensione. Successivamente, quando l'utente rientrerà nella stessa pagina di dettaglio, troverà già la sua recensione con un pulsante che permetterà la modifica.

CU20 - Visualizzazione recensioni prodotto Questo caso d'uso consente all'utente autenticato di visualizzare tutte le recensioni di un prodotto. L'utente raggiungerà la pagina di dettaglio del prodotto desiderato, dove troverà un pulsante che porterà alla schermata contenente tutte le recensioni del prodotto stesso.

CU21 - Visualizzazione classifica di prodotti con voto medio maggiore Questo caso d'uso consente all'utente autenticato di visualizzare una classifica di film e serie TV basata sul voto medio. L'utente raggiungerà la Home Page dell'applicazione. Il sistema, allora, calcolerà le classifiche attraverso i dati statistici del database e mostrerà i due elenchi.

Gestione del profilo utente

CU22 - Modifica profilo Questo caso d'uso consente all'utente autenticato di modificare il proprio profilo. L'utente raggiungerà la pagina di modifica del profilo attraverso l'apposito pulsante presente nella sua pagina di profilo. Nella schermata, l'utente potrà effettuare le modifiche desiderate, ad esempio cambiare il proprio nome visualizzato, la propria immagine del profilo oppure la propria immagine di sfondo.

CU23 - Visualizzazione profilo dell'utente corrente Questo caso d'uso consente all'utente autenticato di visualizzare la propria pagina di profilo. Attraverso la barra di navigazione dell'applicazione, l'utente cliccherà sull'apposito pulsante per visualizzare il proprio profilo e il sistema provvederà a mostrarlo, popolandolo con i dati presenti nel database.

CU24 - Visualizzazione profilo di un altro utente Questo caso d'uso consente all'utente autenticato di visualizzare la pagina di profilo di un altro utente. L'utente raggiungerà l'apposita schermata attraverso una delle modalità fornite dall'applicazione. Il sistema popolerà la schermata con i dati presenti nel proprio database.

Tracciamento degli episodi

CU25 - Aggiunta di un episodio alla lista degli episodi visti Questo caso d'uso consente all'utente autenticato di segnare un episodio di una serie TV come visto. L'utente raggiungerà la pagina di dettaglio della serie TV e cliccherà sull'apposito pulsante presente accanto all'episodio desiderato. Il sistema provvederà ad aggiungere l'episodio alla lista degli episodi visti e ad effettuare le relative operazioni.

CU26 - Visualizzazione prossimi episodi Questo caso d'uso consente all'utente autenticato di visualizzare quali sono i prossimi episodi da vedere. L'utente raggiungerà la schermata dell'applicazione relativa agli episodi da vedere e il sistema provvederà a calcolare quali sono i prossimi episodi da vedere e a mostrarli nella schermata.

2.4.3 Diagrammi dei casi d'uso

I diagrammi dei casi d'uso sono una potente forma di visualizzazione che consente di comprendere le interazioni tra gli attori e il sistema in modo chiaro e intuitivo.

Ognuno dei casi d'uso definiti nella sezione precedente sarà, in questa sezione, rappresentato graficamente e messo in relazione con gli attori del sistema attraverso lo strumento dei diagrammi dei casi d'uso. I diagrammi dei casi d'uso dell'applicazione WatchWise sono stati suddivisi così come i casi d'uso stessi, in modo da evitare confusione derivata dalla complessità di un diagramma unico.

Il diagramma dei casi d'uso mostrato in Figura 2.4 definisce la relazione tra l'attore "Utente non autenticato" e i casi d'uso relativi alla sezione "Autenticazione e Registrazione".

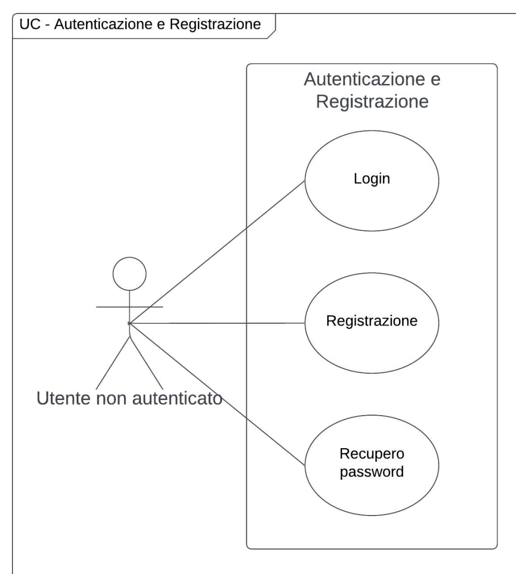

Figura 2.4: Diagramma dei casi d'uso per la sezione "Autenticazione e Registrazione"

Il diagramma dei casi d'uso mostrato in Figura 2.5 definisce la relazione tra l'attore "Utente non autenticato" e i casi d'uso relativi alla sezione "Esplorazione dei contenuti".

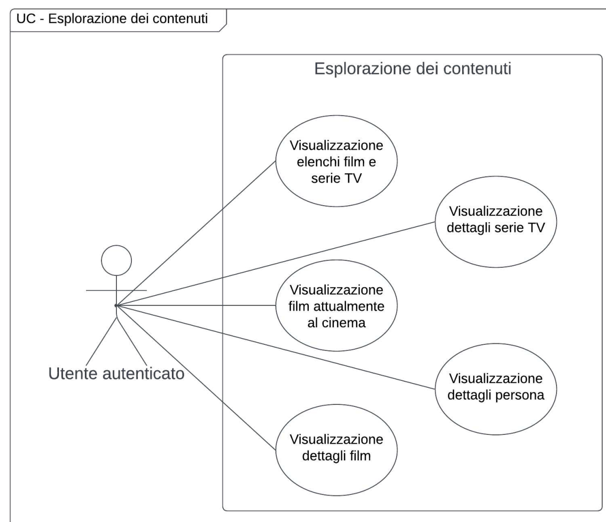

Figura 2.5: Diagramma dei casi d'uso per la sezione "Esplorazione dei contenuti"

Il diagramma dei casi d'uso mostrato in Figura 2.6 definisce la relazione tra l'attore "Utente autenticato" e i casi d'uso relativi alla sezione "Interazioni sociali".

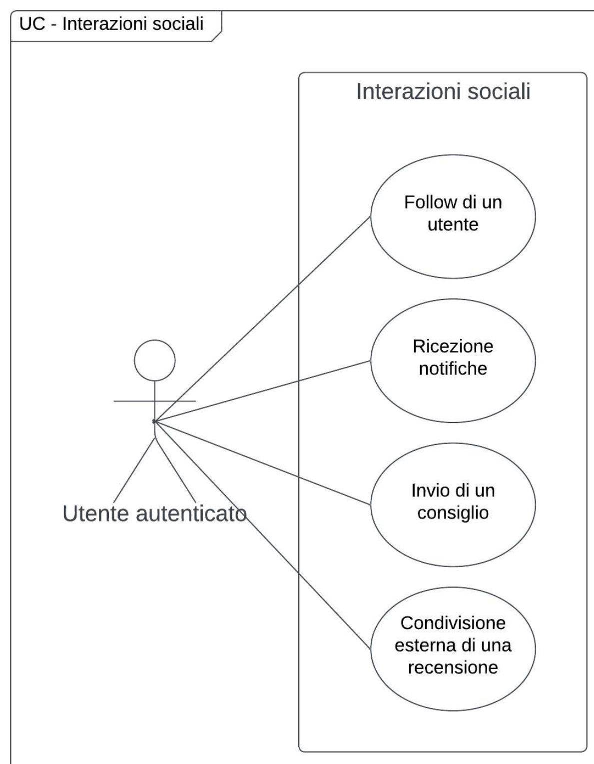

Figura 2.6: Diagramma dei casi d'uso per la sezione "Interazioni sociali"

Il diagramma dei casi d'uso mostrato in Figura 2.7 definisce la relazione tra l'attore "Utente autenticato" e i casi d'uso relativi alla sezione "Gestione delle liste".

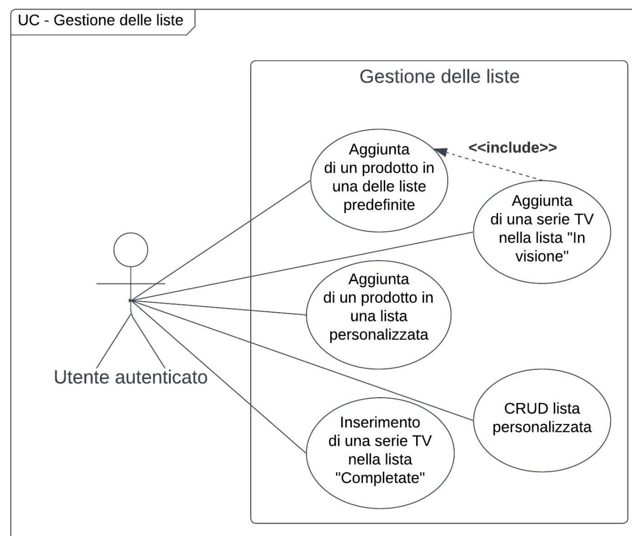

Figura 2.7: Diagramma dei casi d'uso per la sezione "Gestione delle liste"

Il diagramma dei casi d'uso mostrato in Figura 2.8 definisce la relazione tra l'attore "Utente autenticato" e i casi d'uso relativi alla sezione "Valutazioni e recensioni".

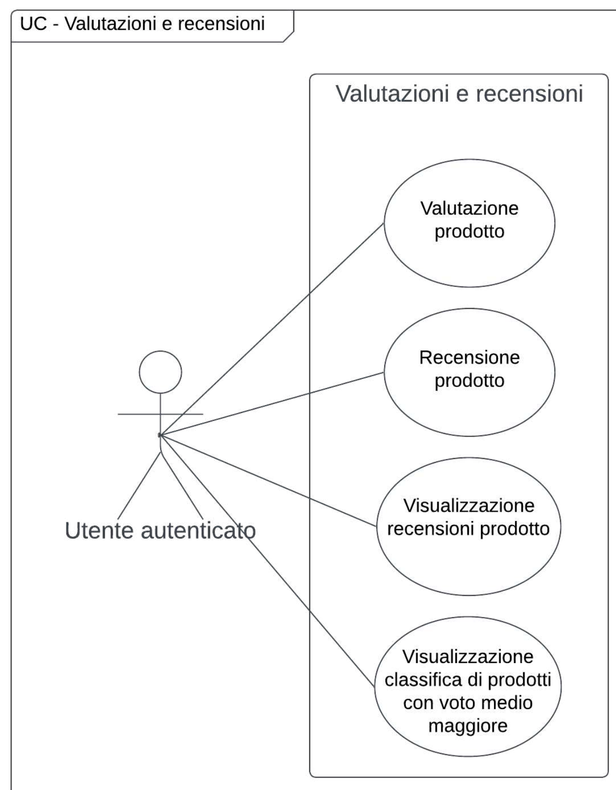

Figura 2.8: Diagramma dei casi d'uso per la sezione "Valutazioni e recensioni"

Il diagramma dei casi d'uso mostrato in Figura 2.9 definisce la relazione tra l'attore "Utente autenticato" e i casi d'uso relativi alla sezione "Gestione del profilo utente".

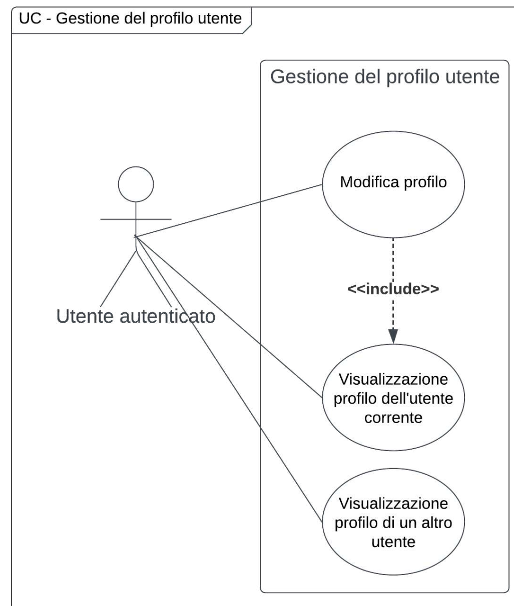

Figura 2.9: Diagramma dei casi d'uso per la sezione "Gestione del profilo utente"

Il diagramma dei casi d'uso mostrato in Figura 2.10 definisce la relazione tra l'attore "Utente autenticato" e i casi d'uso relativi alla sezione "Tracciamento degli episodi".

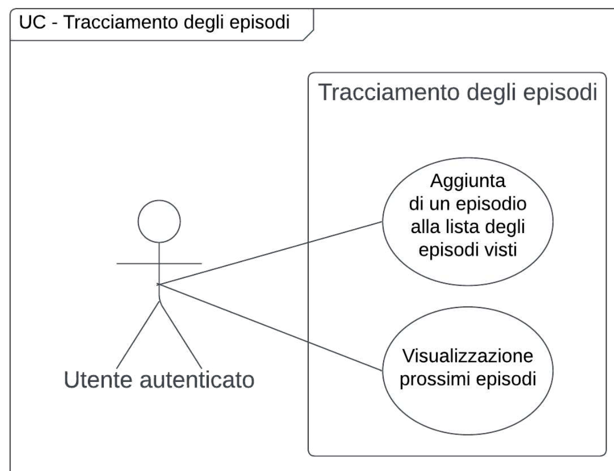

Figura 2.10: Diagramma dei casi d'uso per la sezione "Tracciamento degli episodi"

2.5 Matrice di mapping dei requisiti

La matrice di mapping dei requisiti è una rappresentazione visiva che aiuta ad identificare le relazioni tra requisiti funzionali (descritti nella Sezione 2.2) e i casi d'uso (descritti

nella sezione precedente). Tale rappresentazione grafica è una matrice avente come righe i casi d'uso definiti e come colonne i requisiti funzionali. Ogni cella, quindi, rappresenta una relazione caso d'uso-requisito funzionale; se tale caso d'uso soddisfa il corrispondente requisito, allora la cella contiene una "X". Le Tabelle 2.2 e 2.3 illustrano la matrice di mapping dei requisiti dell'applicazione WatchWise.

	R1	**R2**	**R3**	**R4**	**R5**	**R6**	**R7**	**R8**	**R9**	**R10**	**R11**	**R12**	**R13**
CU1	X												
CU2	X												
CU4		X				X							
CU5		X											
CU6			X										
CU7				X									
CU8					X								
CU13							X	X	X				
CU14							X						
CU15							X						
CU16										X			
CU17										X			
CU18											X		
CU19												X	
CU20													X

Tabella 2.2: Matrice di mapping dei requisiti (Parte 1)

	R14	**R15**	**R16**	**R17**	**R18**	**R19**	**R20**	**R21**	**R22**	**R23**	**R24**
CU3									X		
CU4			X	X							
CU9		X									
CU10							X				
CU11										X	
CU12											X
CU21								X			
CU22						X					
CU23	X			X							
CU24	X			X							
CU25					X						
CU26					X						

Tabella 2.3: Matrice di mapping dei requisiti (Parte 2)

Come si evince dalla matrice di mapping dei requisiti, ogni requisito è soddisfatto da almeno un caso d'uso. Ciò significa che tutti i requisiti potranno essere soddisfatti in fase di progettazione.

2.6 Conclusioni

Questo capitolo ha delineato in modo esauriente la specifica e l'analisi dei requisiti per l'applicazione WatchWise. La chiara definizione delle funzionalità e dei vincoli tecnici permetterà di guidare lo sviluppo e la progettazione dell'applicazione, garantendo un prodotto di alta qualità e soddisfacente per gli utenti.

L'obiettivo più importante raggiunto in questo capitolo è la costruzione di una base su cui cominciare la progettazione per poi proseguire con lo sviluppo. La fase del progetto affrontata in questo capitolo, infatti, rappresenta una fase fondamentale di qualsiasi progetto software, che se non eseguita correttamente potrebbe causare molti problemi nelle fasi successive, quando tornare indietro diventa sempre più ostico.

Si può, dunque, passare alla fase di progettazione dell'applicazione WatchWise.

CAPITOLO 3

Progettazione

Questo capitolo presenta il dettaglio della progettazione dell'applicazione WatchWise. Nello specifico, il capitolo inizia con un'analisi dell'architettura dell'applicazione dettagliata da una panoramica sul pattern Model-View-ViewModel (MVVM). In seguito, il capitolo illustra la progettazione del database dell'applicazione elencando e spiegando le collezioni inserite nel database Firebase Cloud Firestore. Successivamente, viene discussa la progettazione dell'interfaccia utente, ponendo un accento sulla mappa delle schermate dell'applicazione e sui suoi mockup. Dopo di ciò, vengono mostrati i diagrammi di progettazione rappresentanti uno sviluppo dei casi d'uso presentati nel capitolo precedente e una base sulla quale iniziare l'implementazione. Infine, vengono discussi brevemente dei concetti di ottimizzazione specificatamente per le prestazioni dell'applicazione e per la gestione delle notifiche push.

3.1 Architettura dell'applicazione

Nell'era moderna dello sviluppo software, la progettazione di un'applicazione non riguarda solo la creazione di funzionalità efficienti e affidabili, ma anche la costruzione di un'architettura robusta e scalabile. L'architettura di un'applicazione definisce la sua struttura complessiva, determinando come le varie componenti interagiscono tra loro e come l'applicazione si comporta nel suo complesso. In questa sezione, verrà analizzata l'architettura adottata per lo sviluppo dell'applicazione WatchWise, descrivendone le caratteristiche principali e i suoi vantaggi peculiari.

3.1.1 Pattern Model-View-ViewModel (MVVM)

Il *Model-View-ViewModel* (*MVVM*) è un pattern architetturale che separa lo sviluppo dell'interfaccia grafica dell'applicazione dalla logica di business. Questo pattern è diventato sempre più comune nella programmazione mobile negli ultimi anni, poiché offre una chiara separazione delle responsabilità e facilita la manutenzione e l'espansione del codice. I tre componenti fondamentali del pattern MVVM sono descritti di seguito:

- *Model*: rappresenta i dati e la logica di business dell'applicazione. Il Model è responsabile dell'acquisizione, della manipolazione e del salvataggio dei dati. Non ha alcuna conoscenza dell'interfaccia utente e comunica con il ViewModel attraverso degli eventi o dei meccanismi di data binding.
- *View*: rappresenta l'interfaccia utente dell'applicazione. La View è responsabile della visualizzazione dei dati e dell'interazione con l'utente. Non contiene logica di business,

non si occupa di manipolare i dati che ottiene e comunica con il ViewModel attraverso il data binding.

- *ViewModel*: funge da intermediario tra il Model e la View. Il ViewModel acquisisce i dati dal Model, li elabora se necessario, e li fornisce alla View in un formato adatto per la visualizzazione. Inoltre, risponde agli input dell'utente e invoca le azioni appropriate nel Model.

Il grande vantaggio dell'architettura MVVM è la sua modularità. Poiché ogni componente è in possesso di una responsabilità chiaramente definita, è possibile modificare o sostituire uno di essi senza influenzare gli altri. Ad esempio, si potrebbe cambiare l'interfaccia utente dell'applicazione (la View) senza dover modificare la logica di business (il Model) o il codice che gestisce la comunicazione tra i due (il ViewModel). Inoltre, il pattern MVVM facilita la creazione di test automatizzati; poiché la logica di business è separata dall'interfaccia utente, è possibile testare il Model e il ViewModel in un ambiente isolato, senza la necessità di interagire con la View.

Nel contesto dell'applicazione WatchWise, l'adozione del pattern MVVM assicura che l'applicazione sia flessibile, altamente manutenibile e testabile. Mentre il Model gestisce la comunicazione con il database Firebase Cloud Firestore e la logica associata ai dati dell'applicazione, la View si concentra sulla presentazione e l'interazione con l'utente. Il ViewModel, infine, fa da ponte tra questi due componenti, garantendo che i dati siano presentati in modo appropriato e che le azioni dell'utente siano gestite correttamente.

3.2 Progettazione del database

La scelta del database in un progetto software è un passo fondamentale per garantire prestazioni ottimali, scalabilità, flessibilità e persistenza dei dati. Per l'applicazione WatchWise, come è stato specificato nei requisiti non funzionali (Sezione 2.2.2), è stato scelto Firebase Cloud Firestore, un DBMS NoSQL basato su cloud che offre scalabilità automatica, efficienza e supporto offline.

3.2.1 Caratteristiche di Cloud Firestore

Firebase Cloud Firestore è un DBMS documentale. Ciò significa che i dati sono memorizzati come documenti e organizzati in collezioni. Ogni documento è identificato da un ID univoco e può contenere dati complessi sotto forma di campi. Questi ultimi possono includere sottocampi che, a loro volta, possono contenere dati semplici o complessi, consentendo una struttura di dati altamente flessibile e gerarchica.

A differenza di un DBMS relazionale come MySQL, un DBMS NoSQL come Cloud Firestore non memorizza i dati secondo lo schema rigido e ben definito delle tabelle. Infatti, ogni documento non deve rispettare uno schema preciso per i suoi campi in base alla collezione in cui si trova, ma è libero di possedere campi diversi e contenenti anche altri campi annidati. Inoltre, un DBMS non relazionale, ovviamente, non possiede il concetto di relazioni e, quindi, di chiavi esterne o join. Tuttavia, un comportamento simile può essere ottenuto attraverso l'utilizzo dei documenti nidificati o dei riferimenti ad altri documenti nei campi.

Una delle principali caratteristiche di Cloud Firestore è la sua capacità di scalare automaticamente in base alle esigenze dell'applicazione. Ciò significa che, indipendentemente dal numero di utenti o dalla quantità di dati, Cloud Firestore può gestire il carico, garantendo tempi di risposta rapidi e un'esperienza utente fluida e reattiva.

3.2.2 Struttura del database

Tenendo conto delle esigenze dell'applicazione WatchWise, tra cui anche la necessità di ottimizzare le prestazioni, il database è stato progettato seguendo un approccio di *documenti embedded*. Tale approccio sfrutta la natura gerarchica dei database documentali, consentendo di nidificare documenti all'interno di altri documenti. Ciò riduce la necessità di eseguire molteplici query per ottenere dati correlati, migliorando notevolmente le prestazioni e semplificando la struttura del database.

Nel seguente elenco viene presentata una panoramica delle collezioni progettate per il database e delle loro funzioni:

- *Collezione* `users`: questa collezione memorizza le informazioni relative agli utenti dell'applicazione. Ogni suo documento rappresenta un utente e contiene campi come username, indirizzo email, immagine di profilo e altre informazioni personali. All'interno di ogni documento di questa collezione sono presenti, anche, dei documenti nidificati che rappresentano le interazioni dell'utente con i prodotti (ovvero, liste di film e serie TV, recensioni, episodi, etc.).

- *Collezione* `products`: anche se i dettagli specifici dei film, delle serie TV e delle persone non sono memorizzati nel database (e vengono ottenuti tramite chiamate all'API di TMDB), questa collezione è utile a memorizzare informazioni aggregate, come, ad esempio, il numero totale di utenti che hanno recensito un particolare film oppure la sua media di valutazione.

- *Collezione* `notifications`: questa collezione gestisce le notifiche push inviate agli utenti. È utile a memorizzare il tracciamento temporale degli eventi che dovranno, poi, essere inviati tramite notifiche push e mostrati nell'app.

- *Collezione* `recommendations`: ogni documento di questa collezione rappresenta un consiglio inviato da un utente ad un altro. Contiene campi come l'UID del mittente e del destinatario, l'ID del prodotto, il timestamp del momento in cui il consiglio è stato inviato e uno stato utilizzato per tracciare se il consiglio è stato visualizzato o meno.

3.3 Progettazione dell'interfaccia utente

L'interfaccia utente di un'applicazione mobile gioca un ruolo cruciale nell'esperienza complessiva dell'utente. Un'interfaccia progettata per essere intuitiva, reattiva e visivamente piacevole può fare la differenza tra un'applicazione di successo e una che viene rapidamente disinstallata, rendendo l'estetica e l'intuitività dei valori al pari livello dell'efficienza e delle funzionalità offerte. Per l'applicazione WatchWise, l'obiettivo è quello di fornire un'interfaccia che non solo soddisfi le esigenze funzionali, ma che offra anche un'esperienza utente di alto livello.

3.3.1 Mappa dell'applicazione

La mappa dell'applicazione fornisce una visione d'insieme della struttura e del flusso delle schermate dell'applicazione. Essa rappresenta una guida visiva che mostra come le varie schermate siano collegate tra loro e come l'utente può navigare tra di esse. La Figura 3.1 illustra uno schema rappresentativo della mappa dell'applicazione WatchWise. È importante notare che, con il fine di non complicare eccessivamente lo schema, non sono stati inclusi dei percorsi che, però, esisteranno nell'applicazione finale. Ad esempio, il percorso tra la schermata Home e le schermate di dettaglio dei prodotti.

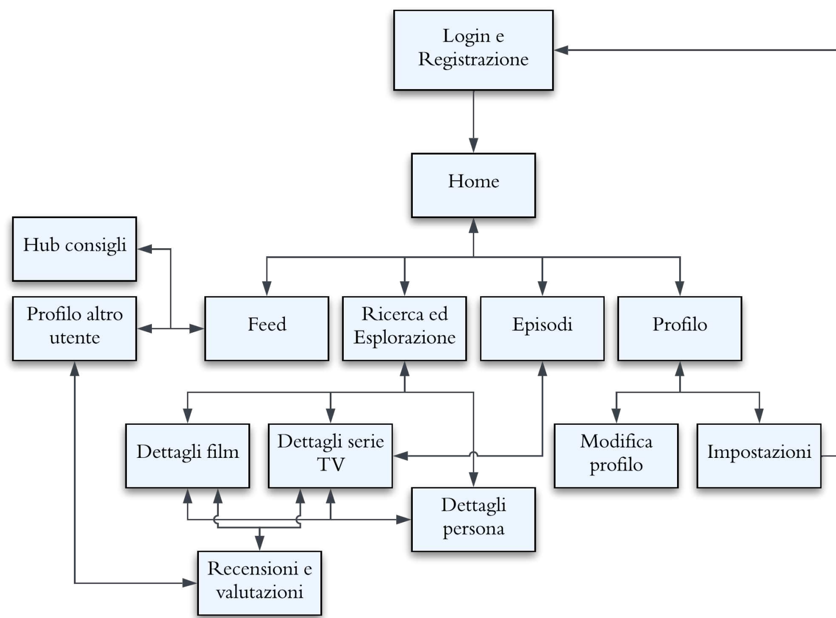

Figura 3.1: Mappa dell'applicazione WatchWise

3.3.2 Logo dell'applicazione

Il logo di WatchWise è stato creato con l'intento di catturare l'essenza dell'applicazione e di fornire un simbolo facilmente riconoscibile e memorabile per gli utenti. Il design del logo è stato influenzato sia dall'estetica moderna e minimalista di iOS che dall'obiettivo specifico dell'applicazione, ovvero fornire una piattaforma sociale per gli amanti di film e serie TV. Per raggiungere questo equilibrio, il logo utilizza una ripetizione del simbolo "play", spesso utilizzato per indicare la riproduzione di un contenuto multimediale, con un design pulito e semplificato, seguendo le linee guida stilistiche di Apple. Il colore e la tipografia del logo sono stati selezionati in modo da allinearsi con la palette di colori e i font utilizzati nell'interfaccia dell'app, garantendo così una coerenza visuale in tutto l'ecosistema dell'applicazione. Il logo è stato realizzato utilizzando il software *Adobe Illustrator* ed è mostrato in Figura 3.2.

Figura 3.2: Logo di WatchWise

3.3.3 Mockup dell'applicazione

I mockup sono delle rappresentazioni visive della progettazione dell'interfaccia utente. Essi forniscono una visione chiara di come apparirà ogni schermata, permettendo di visualizzare la disposizione degli elementi, i colori, le dimensioni e altri dettagli stilistici. Considerando che l'applicazione seguirà le linee guida stilistiche di Apple per iOS, i mockup sono stati progettati per garantire coerenza e fluidità nell'interfaccia.

I mockup sono stati realizzati utilizzando il software *Figma* e la libreria pubblica *Apple Design Resources – iOS 17 and iPadOS 17* fornita da Apple stessa per poter realizzare mockup basati sulle sue linee guida. In Figura 3.3 sono mostrati i mockup realizzati per WatchWise.

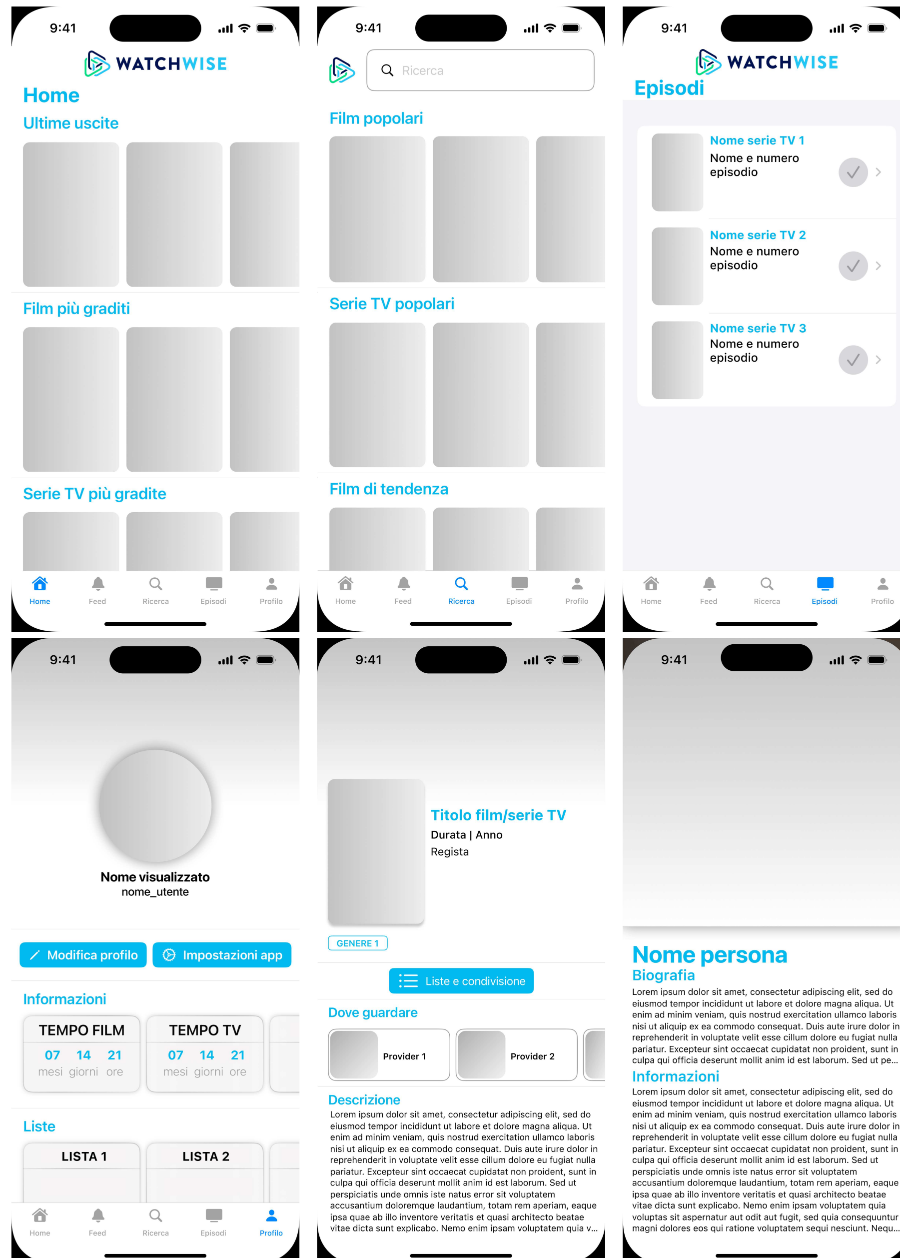

Figura 3.3: Mockup di WatchWise

È importante specificare che i mockup realizzati non rappresentano l'interezza delle schermate che saranno presenti nel prodotto finale, ma solo una frazione di esse. Questa scelta è stata effettuata per velocizzare la progettazione e tenendo in considerazione il fatto

che lo stile e il design delle schermate per cui sono stati realizzati i mockup saranno simili a quelli delle schermate per cui non sono stati realizzati i mockup.

3.4 Diagrammi

Il fulcro della fase di progettazione di un applicativo software è la creazione dei diagrammi di progettazione. Uno degli strumenti principali utilizzati per questa finalità è lo *Unified Modeling Language* (*UML*), che fornisce una rappresentazione visiva e astratta dei vari componenti del software e delle loro interazioni. In questa sezione verranno presentati diversi diagrammi UML che descriveranno visivamente l'architettura e il comportamento dell'applicazione WatchWise.

Prima di procedere con la presentazione dei diagrammi, è essenziale comprendere la natura e lo scopo di ciascun tipo di diagramma UML:

- *Diagrammi di sequenza*: rappresentano la sequenza di messaggi scambiati tra gli oggetti e i componenti del sistema per eseguire uno specifico caso d'uso. Questi diagrammi forniscono una visione chiara delle interazioni e delle responsabilità di ciascun componente.
- *Diagrammi di attività*: mostrano il flusso di attività e le operazioni all'interno del sistema per l'esecuzione di uno specifico caso d'uso.
- *Diagrammi delle classi*: rappresentano le classi, i loro attributi e metodi e le relazioni tra di esse. Questi diagrammi offrono una visione statica dell'architettura del sistema, aiutando a comprendere la struttura del software.

Le best practice di ingegneria del software suggeriscono di realizzare un diagramma di sequenza e uno di attività per ogni caso d'uso. Tuttavia, considerando l'elevato numero di casi d'uso dell'applicazione WatchWise, è stato deciso di presentare un numero ridotto di ciascuno di questi due tipi di diagrammi. Questa scelta aiuterà a ridurre la ridondanza dal momento che molti dei diagrammi sarebbero stati molto simili tra loro. Inoltre, a causa di questa stessa similitudine, l'omissione di molti dei diagrammi non causerà mancanza di informazioni necessarie.

3.4.1 Diagrammi di sequenza

In questa sezione vengono presentati i diagrammi di sequenza che si è scelto di realizzare.

Visualizzazione profilo utente

Il diagramma di sequenza in Figura 3.4 illustra l'interazione tra l'utente autenticato, l'interfaccia dell'applicazione e le varie classi che gestiscono la visualizzazione del profilo dell'utente. L'utente richiede la visualizzazione del profilo cliccando sull'apposito pulsante dell'interfaccia. Il sistema, allora, mediante una serie di messaggi, ottiene le informazioni dell'utente dal database e, tramite una serie di messaggi di risposta, popola l'interfaccia con le informazioni necessarie.

Inserimento/rimozione film in watchlist

Il diagramma di sequenza in Figura 3.5 illustra l'interazione tra l'utente autenticato, l'interfaccia dell'applicazione e le varie classi che gestiscono l'inserimento (o la rimozione) di un film nella watchlist dei film. L'utente richiede l'inserimento (o la rimozione) del film

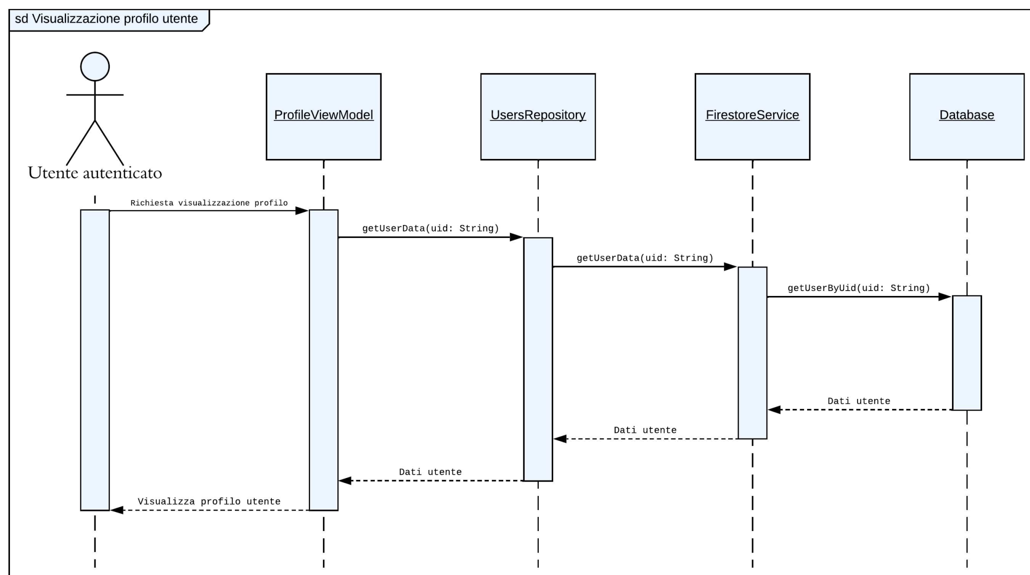

Figura 3.4: Diagramma di sequenza relativo alla visualizzazione del profilo utente

nella watchlist dei film. Il sistema, allora, innanzitutto controlla se il film è già presente nella watchlist; successivamente, con una serie di messaggi, la classe che si occupa dell'interazione con il database inserisce il film nella watchlist o lo rimuove da essa (in base al risultato del controllo effettuato precedentemente). Infine, con una serie di messaggi di risposta, il sistema informa l'utente del completamento dell'operazione.

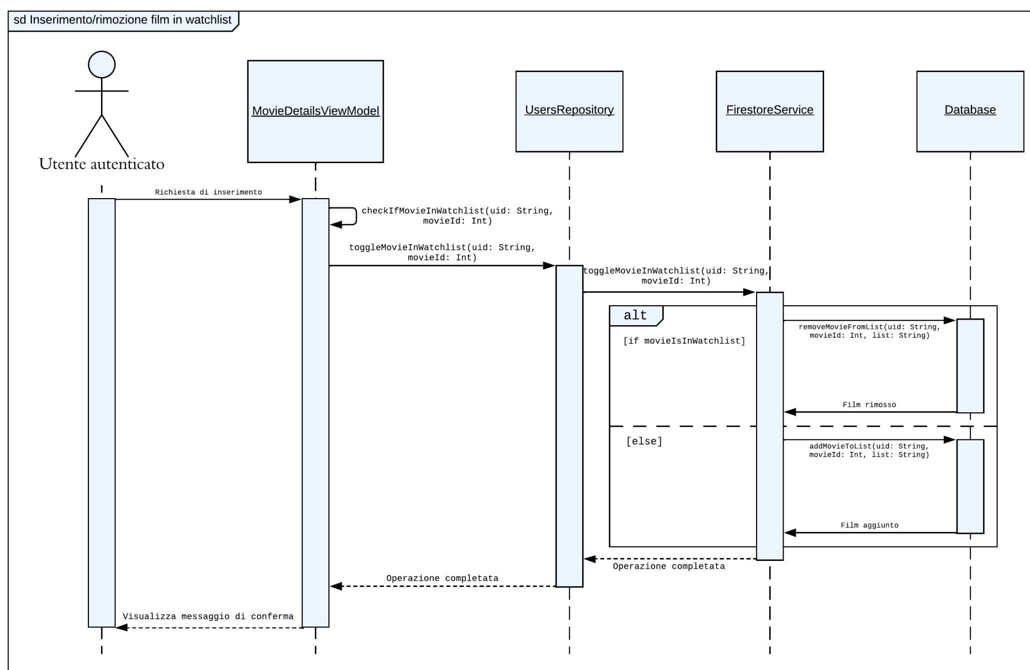

Figura 3.5: Diagramma di sequenza relativo all'inserimento (o alla rimozione) di un film nella watchlist

Invio di un consiglio

Il diagramma di sequenza in Figura 3.6 illustra l'interazione tra l'utente autenticato, l'interfaccia dell'applicazione e le varie classi che gestiscono l'invio di consigli ad altri utenti. L'utente richiede l'invio di un consiglio ad un altro utente. Il sistema, allora, con una serie di messaggi, crea il nuovo consiglio all'interno del database e invia una notifica push all'utente destinatario. Infine, con una serie di messaggi di risposta, l'utente mittente viene informato del successo dell'operazione.

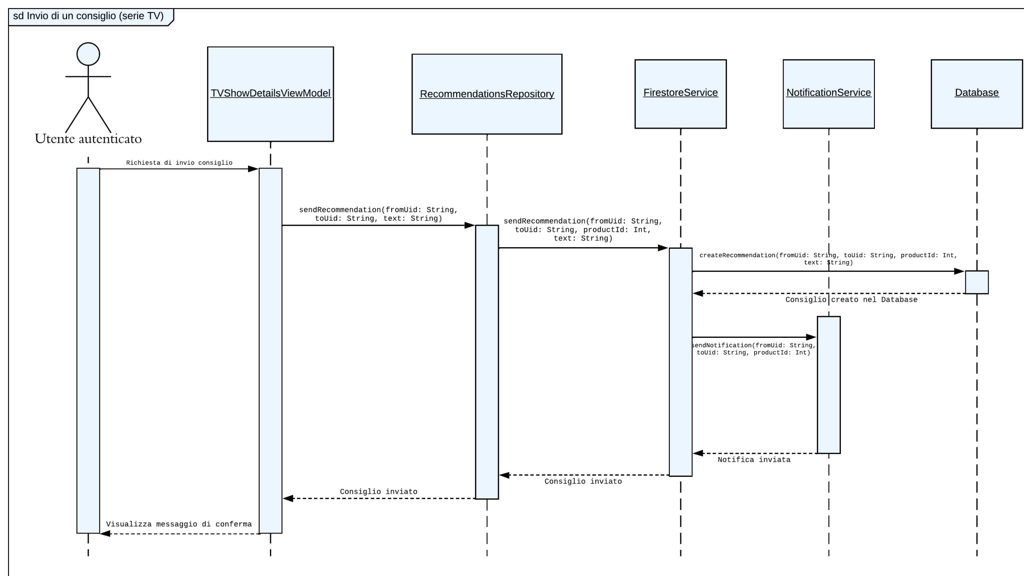

Figura 3.6: Diagramma di sequenza relativo all'invio di un consiglio

3.4.2 Diagrammi di attività

In questa sezione vengono presentati i diagrammi di attività che si è scelto di realizzare.

Registrazione

Il diagramma di attività in Figura 3.7 illustra il flusso delle azioni effettuate dal sistema per la registrazione di un nuovo utente. Alla prima apertura dell'applicazione, l'utente richiede la registrazione. Se l'utente sceglie di registrarsi tramite indirizzo email e la password, egli dovrà inserire tali dati e, se non ci sono errori, passerà all'inserimento dei dettagli del profilo utente. Se l'utente sceglie di registrarsi tramite Google potrà inserire direttamente i dettagli del profilo utente, saltando il primo passo della registrazione tramite indirizzo email. Se non ci sono errori nemmeno nell'inserimento dei dettagli, la registrazione viene completata e il flusso di attività si conclude.

Aggiunta di un episodio alla lista degli episodi visti

Il diagramma di attività in Figura 3.8 illustra il flusso delle azioni effettuate dal sistema per l'inserimento di un episodio nella lista degli episodi visti di una serie TV. L'utente naviga verso la schermata di dettaglio della serie TV di interesse; successivamente apre l'elenco delle stagioni e degli episodi della stessa. Una volta localizzato l'episodio da aggiungere alla lista, l'utente clicca il pulsante per effettuare l'operazione. Il sistema, allora, controlla se l'episodio è già stato aggiunto alla lista degli episodi visti e, in caso affermativo, lo rimuove da tale lista.

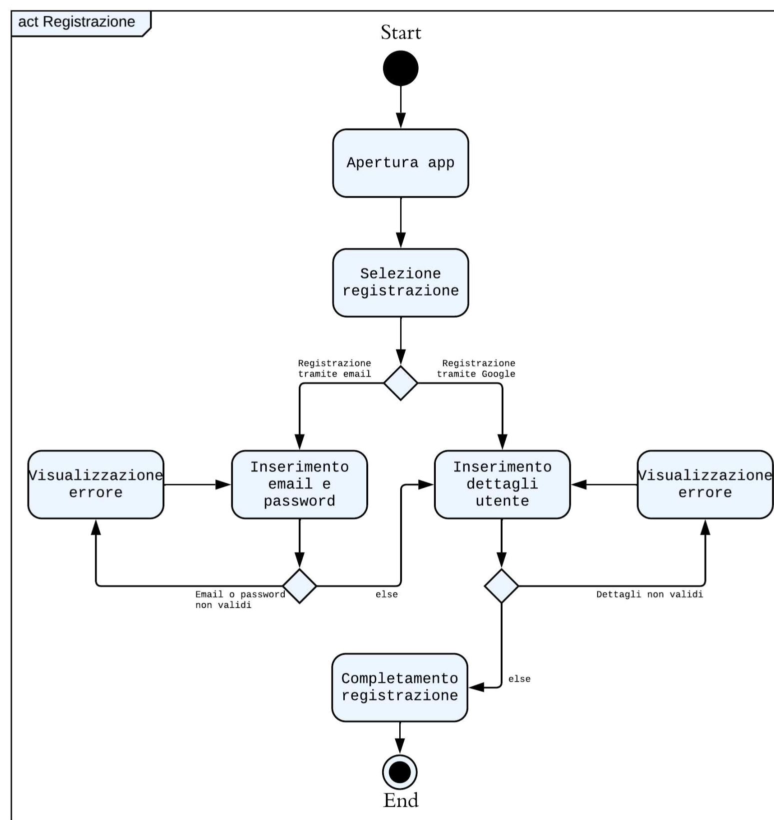

Figura 3.7: Diagramma di attività relativo alla registrazione di un nuovo utente

Nel caso opposto, invece, il sistema aggiunge l'episodio alla lista degli episodi visti. Infine, l'utente viene informato del completamento dell'operazione e il flusso di attività si conclude.

Visualizzazione dei prossimi episodi da vedere

Il diagramma di attività in Figura 3.9 illustra il flusso delle azioni effettuate dal sistema per la visualizzazione dell'elenco degli eventuali prossimi episodi da vedere. L'utente naviga verso la schermata dei prossimi episodi da vedere. Il sistema, allora, effettua delle iterazioni sulle serie TV che l'utente ha inserito nella lista "In visione". Per ognuna delle serie TV in tale lista, il sistema ottiene l'ultimo episodio segnato come visto ed effettua i dovuti calcoli per verificare se esiste un episodio successivo ad esso non segnato come visto. In caso affermativo, il sistema aggiunge tale episodio alla lista dei prossimi episodi da vedere e procede il ciclo con la serie TV successiva. Uscito dal ciclo, il sistema verifica se la lista prodotta è vuota o se contiene almeno un elemento. Nel primo caso, informa l'utente del fatto che non possiede alcun episodio da vedere per le sue serie TV in visione. Nel secondo caso, invece, popola la schermata con gli episodi presenti nella lista. Infine, il flusso di attività si conclude.

3.4.3 Diagramma delle classi

Il diagramma delle classi realizzato per il presente progetto contiene un insieme ridotto delle classi che effettivamente costituiranno il codice completo dell'applicazione. Più della metà delle classi presenti nel codice completo saranno delle classi predefinite del framework SwiftUI, le quali possiedono attributi e metodi la cui signature è predefinita e non può essere personalizzata in base al progetto. Perciò, l'inclusione di tali classi nel diagramma delle classi

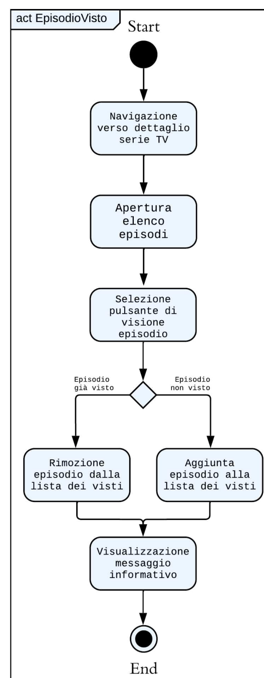

Figura 3.8: Diagramma di attività relativo all'inserimento di un episodio nella lista degli episodi visti

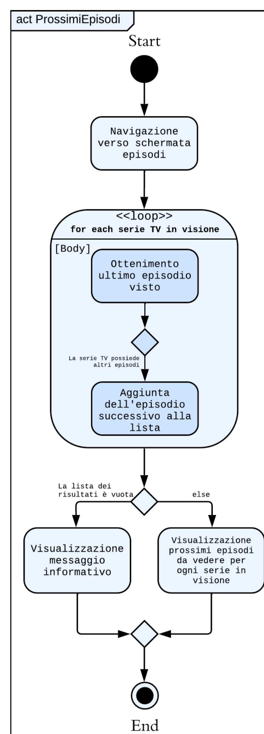

Figura 3.9: Diagramma di attività relativo alla visualizzazione dei prossimi episodi da vedere

avrebbe complicato in modo superfluo il diagramma stesso ed è stato, dunque, scelto di ometterle.

Le Figure 3.10 e 3.11 descrivono il diagramma delle classi (non predefinite) per l'applicazione WatchWise.

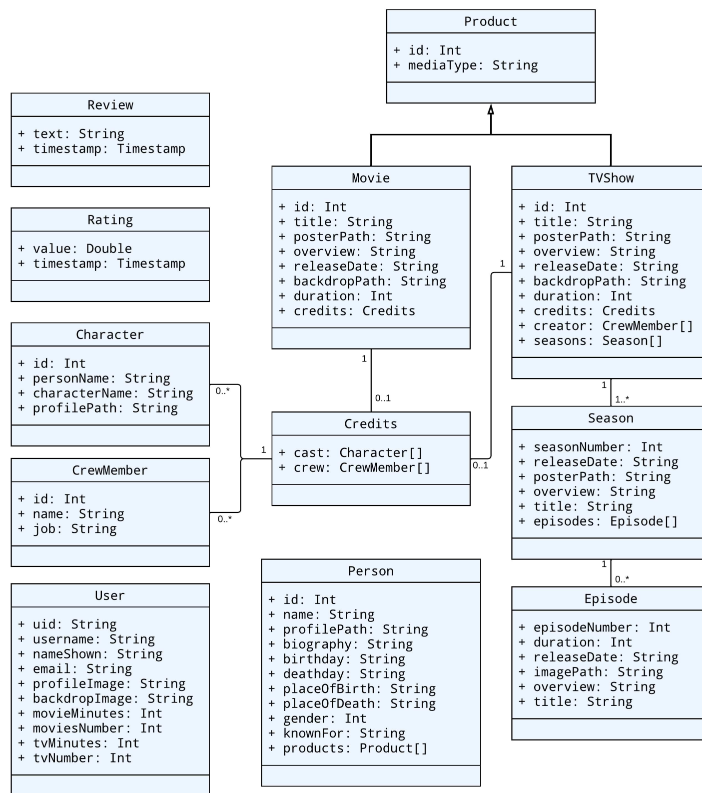

Figura 3.10: Diagramma delle classi della sezione *Model*

Figura 3.11: Diagramma delle classi della sezione *Repository*

3.5 Ottimizzazione

Nello sviluppo di un'applicazione mobile, e specialmente per i social network, l'ottimizzazione del software è una fase fondamentale. Gli utenti mobile, infatti, essendo spesso in movimento, desiderano che le applicazioni che utilizzano quotidianamente abbiano un'alta reattività e dei bassi tempi di attesa. Quindi, nell'ottica di fornire agli utenti di WatchWise una migliore esperienza di utilizzo, sono stati presi in considerazione diversi aspetti riguardanti

l'ottimizzazione, tra i quali è stato posto un focus sull'ottimizzazione delle prestazioni e sulla possibilità di ricevere notifiche push che siano il più possibile in tempo reale.

3.5.1 Ottimizzazione delle prestazioni

Firebase Cloud Firestore, pur essendo un DBMS altamente performante, può trarre beneficio da alcuni metodi di ottimizzazione che mirano a migliorare ulteriormente le prestazioni. Questi due metodi sono l'*indicizzazione* e il *caching locale*.

Indicizzazione

Firestore, di default, crea indici per ogni campo di un documento, semplificando le operazioni di query. Ciò è particolarmente utile quando si effettuano query basate su un solo campo. Tuttavia, in caso di query composte che coinvolgono più campi o query più complesse, è necessario creare degli indici personalizzati. Questi indici possono essere creati e configurati attraverso la console del progetto Firebase.

Ad esempio, uno degli indici creati per l'applicazione WatchWise è quello sullo username degli utenti, che migliora notevolmente la ricerca tramite username di tutti gli utenti registrati all'applicazione.

Caching locale e persistenza dei dati

La reattività dell'applicazione non si basa solo sulla velocità di risposta delle query, ma anche sulla capacità dell'applicazione di funzionare in condizioni di connettività limitata. Firestore offre una potente funzionalità di caching locale proprio per gestire tale problematica. Quando i dati vengono recuperati da Firestore, una copia di essi viene automaticamente salvata in una cache locale. Così, in caso di mancanza di connessione, l'applicazione può riferirsi a questa cache, permettendo una quasi normale operatività. Questa funzione non solo migliora la reattività, ma offre anche la possibilità di operare, momentaneamente, in modalità offline. Poi, una volta che la connettività viene ripristinata, Firestore è in grado di gestire la sincronizzazione dei dati tra la cache locale e il database sul cloud, assicurando che tutte le modifiche vengano propagate e che i dati rimangano consistenti.

3.5.2 Gestione delle notifiche push

Le notifiche push sono uno strumento essenziale per il coinvolgimento degli utenti anche quando non stanno usando direttamente l'applicazione. Per WatchWise, come è stato discusso nel Capitolo 2, sono previste tre tipologie principali di notifiche push, ovvero:

- Notifica inviata all'utente quando un nuovo utente inizia a seguirlo.
- Notifica inviata all'utente quando egli riceve un consiglio su un film o una serie TV da parte di un altro utente.
- Notifica inviata all'utente quando un regista che segue rilascia un nuovo prodotto.

L'implementazione delle notifiche push in tempo reale, tuttavia, richiede una componente server back-end che monitori gli eventi rilevanti e invii le notifiche adeguate anche quando l'applicazione dell'utente destinatario non è in esecuzione. Firebase offre *Firebase Functions*, una soluzione serverless che permette di eseguire del codice JavaScript in risposta a determinati trigger[1]. Questo codice JavaScript è, poi, in grado di notificare i dispositivi degli utenti che sono destinatari della notifica.

[1]Eventi particolari, ad esempio la modifica della lista dei follower di un utente nel database.

In termini di costi, l'attivazione dello strumento Firebase Functions necessita dell'upgrade del progetto Firebase dal piano Spark al piano Blaze, il quale è un piano a consumo. Tuttavia, una revisione della struttura tariffaria di Firebase Functions[2] ha rivelato che le quote gratuite del piano a consumo hanno un margine estremamente alto, renendo questa soluzione ideale.

3.6 Conclusioni

La fase di progettazione dell'applicazione WatchWise è stata essenziale per garantire che lo sviluppo proceda senza intoppi e che il prodotto finale risponda efficacemente alle esigenze degli utenti. Attraverso l'analisi e la progettazione dettagliate, effettuate nei capitoli precedenti, sono state poste le basi per un'applicazione solida, performante e facile da utilizzare.

L'adozione del pattern architetturale MVVM offrirà una struttura chiara e scalabile, promuovendo la separazione delle responsabilità e facilitando eventuali modifiche future. La scelta di Cloud Firestore come database garantirà anch'essa scalabilità, ma anche prestazioni e flessibilità. Il dettagliato design dell'interfaccia utente assicurerà un'esperienza utente fluida ed intuitiva. La stesura dei diagrammi di progettazione, inoltre, faciliterà notevolmente la successiva fase di implementazione, potendo basare la scrittura del codice sui diagrammi già realizzati. Infine, gli sforzi volti all'ottimizzazione, eleveranno ulteriormente il valore dell'applicazione, rendendola reattiva e coinvolgente.

In conclusione, la fase di progettazione è stata fondamentale per il progetto ma, soprattutto, per la successiva fase di implementazione.

[2] https://cloud.google.com/functions/pricing?hl=it

CAPITOLO 4

Implementazione

Nel presente capitolo viene analizzata in maniera dettagliata la fase di implementazione dell'applicazione WatchWise. Questa fase ha rappresentato la trasformazione dei diagrammi elaborati durante la progettazione in un prodotto software concreto e funzionante. L'implementazione ha avuto inizio con la configurazione del progetto, attraverso la definizione delle workspace e l'installazione dei vari componenti aggiuntivi, e successivamente ha proceduto con la stesura del codice. È essenziale sottolineare che l'implementazione non si è limitata alla semplice trasposizione dei risultati della fase di progettazione. Infatti, durante questa fase, sono state intraprese azioni specifiche per ottimizzare l'applicazione, assicurandosi che fosse non solo funzionale, ma anche rapida, esteticamente gradevole e, in particolare, orientata all'utente.

4.1 Configurazione iniziale del progetto

La fase di configurazione iniziale di un progetto software riveste un'importanza cruciale. Una corretta impostazione all'avvio offre, infatti, una solida base su cui edificare l'intera applicazione e garantire, nel contempo, la sua scalabilità in futuro.

4.1.1 Creazione del progetto Xcode e del repository Git

Come indicato nell'analisi dei requisiti non funzionali del Capitolo 2, l'ambiente di sviluppo scelto per il progetto è Xcode, il quale viene descritto nella Sezione 1.6.3. La creazione del progetto Xcode ha rappresentato il primo passo essenziale nella configurazione. Questo processo ha avuto inizio con la selezione della tipologia di app, in questo specifico caso "App iOS", cui è seguita l'immissione delle informazioni fondamentali relative all'applicazione (come mostrato in Figura 4.1). Procedendo con "Next", è stato possibile designare la directory del progetto e, contemporaneamente, decidere di inizializzare un repository Git in quella stessa directory.

Una volta completata la creazione del progetto, sono stati generati vari file e cartelle all'interno della directory scelta. Tra questi, i più rilevanti sono:

- `WatchWiseApp.swift`, responsabile della gestione del ciclo di vita dell'app.
- `ContentView.swift`, che rappresenta la vista principale visualizzata di default all'avvio dell'app nel progetto appena creato.
- `Assets.xcassets`, responsabile della gestione degli asset del progetto (immagini, colori, etc.).

- `Info.plist`, un file di configurazione contenente i metadati dell'applicazione.
- Una directory nascosta `.git`, che rappresenta il repository Git e contiene tutte le informazioni necessarie per il version control.

Concludendo questi passaggi, sono stati effettivamente stabiliti sia il progetto Xcode che il repository Git, offrendo, così, una piattaforma solida per avviare lo sviluppo e la gestione delle versioni del progetto.

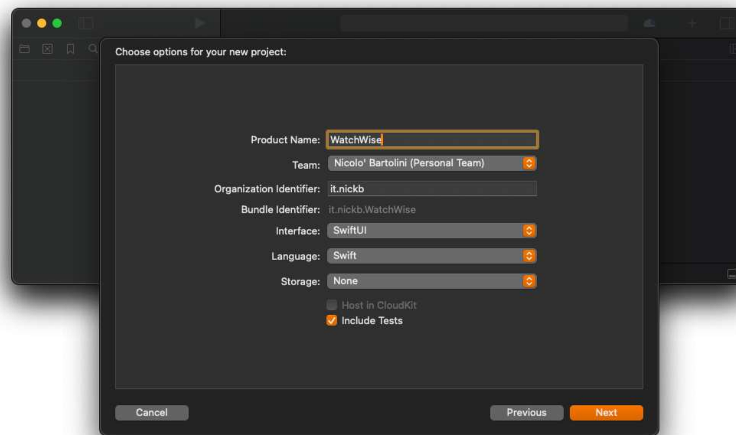

Figura 4.1: Schermata di creazione di un progetto Xcode

4.1.2 Installazione di CocoaPods e confronto con Swift Package Manager

La gestione delle dipendenze riveste anch'essa un ruolo chiave nello sviluppo di applicazioni software contemporanee. Permette agli sviluppatori di integrare in modo semplificato librerie e framework esterni, eliminando la necessità di gestire manualmente il codice di terze parti. Nell'ambito dello sviluppo iOS, *CocoaPods* e *Swift Package Manager* emergono come le principali opzioni per la gestione delle dipendenze. Mentre il primo ha acquisito una solida reputazione come affidabile strumento di gestione delle dipendenze di terze parti, il secondo è un'aggiunta più recente, integrata direttamente in Xcode e proposta da Apple. A causa delle esigenze legate ad alcune librerie specifiche, si è reso necessario ricorrere ad entrambi gli strumenti, ed è per questo che verranno analizzati approfonditamente in questa sezione.

CocoaPods

CocoaPods è uno strumento esterno e necessita di una fase di installazione. Il processo di installazione si avvia con il comando `sudo gem install cocoapods` eseguito nel terminale. Una volta installato CocoaPods sul sistema, è fondamentale creare un file denominato `Podfile` nella directory del progetto tramite il comando `pod init`, in cui vengono specificate le librerie di cui si necessita.

Una caratteristica peculiare di CocoaPods è la sua incidenza sulla struttura del progetto Xcode. Durante l'installazione delle dipendenze, con il comando `pod install` viene generato un file di workspace Xcode che unifica il progetto originale e i Pods, cioè le librerie esterne installate tramite CocoaPods. Questo introduce una differenza nell'architettura iniziale, poiché il file di workspace (con estensione `.xcworkspace`) diviene il fulcro di tutto lo sviluppo, sostituendo, di fatto, il file di progetto originale (estensione `.xcodeproj`).

Swift Package Manager

Swift Package Manager, d'altro canto, presenta una metodologia di gestione delle dipendenze decisamente più integrata. Poiché è incastrato nell'ecosistema Xcode, la sua interazione è intuitiva e diretta, senza necessità di installazioni aggiuntive. La sua implementazione avviene direttamente attraverso l'interfaccia di Xcode, dove è possibile specificare e aggiungere le librerie desiderate, che verranno incorporate direttamente nella struttura di file e cartelle del progetto, mantenendola inalterata.

Sotto l'aspetto pratico, sia CocoaPods che Swift Package Manager esibiscono vantaggi e peculiarità distintive. CocoaPods offre una maggiore versatilità, supportando un ampio ventaglio di librerie, incluse quelle non ancora compatibili con Swift Package Manager. Invece, Swift Package Manager brilla per la sua naturale integrazione con Xcode, semplificando la gestione delle dipendenze.

4.1.3 Creazione e impostazione del progetto Firebase

La scelta di un backend robusto e scalabile garantisce la fluidità operativa di un'applicazione, soprattutto quando si tratta di gestire dati in tempo reale, autenticazioni e altre funzionalità complesse. *Firebase*, offerto da Google, emerge come una delle piattaforme di Backend-as-a-Service (BaaS) più popolari e versatili del mercato attuale, fornendo un insieme di strumenti integrati che agevolano lo sviluppo, la gestione e la scalabilità delle applicazioni.

Creazione del Progetto

La creazione di un nuovo progetto Firebase inizia visitando la Console di Firebase[1] attraverso un browser. Qui, cliccando su "Aggiungi progetto", si avvia una procedura guidata che richiede una serie di informazioni fondamentali relative all'applicazione, come il nome del progetto, la regione di hosting e altri dettagli. Una volta completato questo step, Firebase genera automaticamente un ambiente di lavoro che integra diverse funzionalità, dalle notifiche push all'autenticazione, dallo storage alle analisi in-app.

Integrazione nel Progetto Xcode

Successivamente alla creazione del progetto Firebase, la sua integrazione all'interno dell'ambiente Xcode necessita di alcuni passaggi chiave. Uno dei primi compiti è il download del file `GoogleService-Info.plist` dalla Console di Firebase. Questo file, che contiene configurazioni e credenziali specifiche, deve essere accuratamente inserito nella directory principale del progetto Xcode.

Successivamente, per sfruttare le librerie Firebase all'interno dell'applicazione, è fondamentale installarle, cosa che può essere effettuata sia attraverso CocoaPods che Swift Package Manager, a seconda delle esigenze specifiche. Nel progetto di WatchWise, le librerie di Firebase sono state installate attraverso Swift Package Manager.

L'integrazione di Firebase con l'applicazione non si limita solo all'installazione delle librerie: una volta integrato, è essenziale inizializzare Firebase nel codice sorgente, specificatamente all'interno della classe `AppDelegate`, come mostrato nel Listato 4.1.

4.2 Librerie esterne utilizzate

L'adozione di librerie esterne durante lo sviluppo di un'applicazione può notevolmente semplificare il processo, fornendo soluzioni preconfezionate a problemi comuni e accelerando

[1] `https://console.firebase.google.com/`

```
import Firebase

class AppDelegate: NSObject, UIApplicationDelegate {
    func application(_ application: UIApplication,
                     didFinishLaunchingWithOptions launchOptions:
                     [UIApplication.LaunchOptionsKey : Any]? = nil) -> Bool {
        FirebaseApp.configure()
        return true
    }
}
```

Listato 4.1: Inizializzazione del progetto Firebase nel codice sorgente

la produzione. Durante lo sviluppo di WatchWise, sono state selezionate diverse librerie, suddivise principalmente in due categorie: librerie funzionali e librerie di SwiftUI. Queste librerie hanno agevolato la creazione di caratteristiche avanzate e migliorato l'estetica generale dell'app.

4.2.1 Librerie funzionali

Le seguenti librerie sono state scelte per migliorare e semplificare alcune funzionalità specifiche dell'app:

- *Alamofire* (analizzata nella Sezione 1.8.4): libreria per la gestione delle richieste di rete in Swift. Semplifica la gestione delle chiamate API, garantendo efficienza e sicurezza nelle comunicazioni di rete.
- *Kingfisher* (analizzata nella Sezione 1.8.5): specializzata nella gestione e caching delle immagini. Essa permette di ottimizzare il caricamento e la visualizzazione delle immagini all'interno dell'app.
- *Firebase Libraries*: insieme di librerie utilizzate in vari ambiti del progetto, incluse l'autenticazione, la gestione dei dati con Firestore, l'archiviazione cloud con Firebase Cloud Storage e le funzionalità serverless con Firebase Functions.
- *GoogleSignIn*: garantisce l'integrazione del login attraverso i servizi di Google, offrendo un'esperienza utente semplificata e integrata.

4.2.2 Librerie di SwiftUI

Le seguenti librerie sono state scelte per velocizzare lo sviluppo dell'interfaccia utente e migliorare l'estetica generale dell'applicazione:

- *AlertToast*: arricchendo l'esperienza visiva dell'app, AlertToast fornisce avvisi esteticamente piacevoli che seguono le linee guida di design di Apple. Questi avvisi possono apparire in diverse posizioni e stili, con personalizzazioni che spaziano da testi a icone, fino a animazioni specifiche.
- *AxisRatingBar*: la necessità di raccogliere valutazioni degli utenti per film e serie TV è stata soddisfatta da questa libreria. AxisRatingBar presenta una serie di stelle selezionabili, offrendo un metodo intuitivo e visivamente attraente per raccogliere valutazioni.

- *ExpandableText*: una soluzione elegante ai problemi di visualizzazione del testo di lunghezza variabile. Fornisce una vista che, a seconda della lunghezza del testo inserito, può troncarlo e fornire un meccanismo per espanderlo e visualizzarlo nella sua interezza.

L'inclusione di queste librerie ha notevolmente ampliato le potenzialità dell'applicazione, garantendo al contempo che rimanesse all'avanguardia sia in termini funzionali che estetici.

4.3 Implementazione dell'architettura di WatchWise

Come è stato discusso nella Sezione 3.1.1, il pattern Model-View-ViewModel (MVVM) è uno dei design pattern architetturali più utilizzati nello sviluppo delle applicazioni. Nel corso dello sviluppo di WatchWise, questo pattern è stato implementato con cura, con l'aggiunta di classi Repository e classi ausiliarie per gestire meglio determinate funzionalità e processi.

4.3.1 Classi Model

Le classi Model rappresentano le entità dei dati all'interno dell'applicazione. Queste classi contengono la struttura dati e la logica di business fondamentale, agendo come gli oggetti elementari che compongono le informazioni fornite dall'applicazione.

Nel contesto di WatchWise, le classi Model implementate sono tutte delle `struct`; le più importanti sono racchiuse nel seguente elenco:

- `Movie`: struttura di un film, contenente tutte le sue informazioni nel dettaglio e "mappata" sulla risposta dell'API di TMDB all'endpoint `/movie/movieId`. Il file contenente questa struttura contiene anche altre strutture necessarie per mappare alcune informazioni del film.

- `TVShow`: struttura di una serie TV, contenente tutte le sue informazioni nel dettaglio e "mappata" sulla risposta dell'API di TMDB all'endpoint `/tv/seriesId`. La struttura della serie TV è molto simile a quella del film, ma contiene anche l'elenco di stagioni.

- `Season`: struttura di una stagione di una serie TV, contenente tutte le sue informazioni nel dettaglio e "mappata" sulla risposta dell'API di TMDB all'endpoint di dettaglio delle stagioni (`/tv/seriesId/season/seasonNumber`).

- `Episode`: struttura di un episodio di una serie TV, contenente tutte le sue informazioni nel dettaglio. Le informazioni sull'episodio sono ottenute dalla chiamata all'endpoint di dettaglio della stagione corrispondente.

- `CastMember`: struttura di un membro del cast di un film o di una serie TV, contenente tutte le informazioni sulla persona in questione, oltre al nome del personaggio che interpreta.

- `CrewMember`: struttura di un membro dello staff di un film o di una serie TV, contenente tutte le informazioni sulla persona in questione, oltre al lavoro svolto nello staff.

- `Person`: struttura di una persona, contenente tutte le sue informazioni nel dettaglio e "mappata" sulla risposta dell'API di TMDB all'endpoint `person/personId`.

- `DiscoverMoviesResponse`: struttura ausiliaria "mappata" sulle risposte dell'API di TMDB a chiamate di ricerca o di esplorazione di film. Contiene un array di film trovati e il numero di pagina del risultato.

- `DiscoveredMovie`: struttura contenente le informazioni di un film trovato tramite ricerca o esplorazione. Contiene un insieme ridotto di informazioni sul film in questione e permette di velocizzare e alleggerire le operazioni di ricerca.
- `DiscoverTVShowResponse`: struttura ausiliaria "mappata" sulle risposte dell'API di TMDB a chiamate di ricerca o di esplorazioni di serie TV. Contiene un array di serie TV trovate e il numero di pagina del risultato.
- `DiscoveredTVShow`: struttura contenente le informazioni di una serie TV trovata tramite ricerca o esplorazione. Contiene un insieme ridotto di informazioni sulla serie TV in questione e permette di velocizzare e alleggerire le operazioni di ricerca.
- `DiscoverPeopleResponse`: struttura ausiliaria "mappata" sulle risposte dell'API di TMDB a chiamate di ricerca o di esplorazione di persone. Contiene un array di persone trovate e il numero di pagina del risultato.
- `DiscoveredPerson`: struttura contenente le informazioni di una persona trovata tramite ricerca o esplorazione. Contiene un insieme ridotto di informazioni sulla persona in questione e permette di velocizzare e alleggerire le operazioni di ricerca.
- `Review`: struttura contenente le informazioni su una recensione lasciata da un utente ad un film o ad una serie TV. Contiene le informazioni sull'utente, il testo e la data della recensione.
- `ImagesResponse`: struttura ausiliaria "mappata" sulla risposte dell'API di TMDB agli endpoint `/images`. Contiene solo un array di `Backdrop`.
- `Backdrop`: struttura contenente le informazioni su un backdrop, tra cui percorso, lingua e rapporto d'aspetto dell'immagine.
- `Chat`: struttura contenente le informazioni su una specifica chat. Contiene gli id dei due utenti che interagiscono nella chat e l'ultimo messaggio ricevuto.
- `Message`: struttura contenente le informazioni su uno specifico messaggio in una chat. Contiene l'id del messaggio, l'id del mittente, il contenuto del messaggio e la sua data e ora.

4.3.2 Classi ViewModel

Le classi ViewModel agiscono come un ponte tra il Model e la View. Contengono la logica per presentare i dati alla View. In questo contesto, non sarà necessario elencare tutte le classi ViewModel del progetto dal momento che ne esisterà una per quasi ognuna delle classi View presenti, le quali verranno presentate nella sezione successiva.

4.3.3 Classi View

Le classi View sono responsabili della presentazione dei dati e dell'interazione utente. Secondo il pattern MVVM, la View è generalmente pressoché passiva e si limita a visualizzare ciò che il ViewModel le fornisce.

Nel contesto di WatchWise, le classi View implementate sono tutte delle `struct` (dal momento che è stato utilizzato il framework SwiftUI); le più importanti sono presentate nel seguente elenco:

- `SplashScreen`: vista rappresentante lo *splash screen* dell'applicazione, ovvero quella schermata che viene presentata all'avvio dell'app fintantoché non vengono inizializzati tutti i dati necessari.
- `LoginView`: vista rappresentante la schermata di login, contenente dei campi di testo per effettuare il login e dei pulsanti per confermare il login o per registrarsi.
- `EmailRegistrationView`: vista rappresentante la prima schermata del flusso di registrazione tramite email e password, contenente, appunto, i campi di testo per l'immissione di tali informazioni.
- `CompleteRegistrationView`: vista rappresentante la schermata di completamento del flusso di registrazione (sia tramite email che tramite servizi di terze parti). Contiene campi di testo per l'immissione dei dati necessari al completamento della registrazione e un picker di immagini per scegliere la propria foto profilo.
- `HomeNavigationView`: vista di navigazione principale dell'applicazione, contiene una barra di navigazione inferiore che permette di navigare tra le cinque viste principali dell'applicazione.
- `HomeView`: vista principale dell'applicazione; contiene tre caroselli di poster di film e serie TV con caratteristiche specifiche.
- `FeedView`: vista "social" dell'applicazione; contiene un riassunto delle attività degli utenti seguiti e un pulsante per raggiungere la schermata delle chat.
- `SearchView`: vista di ricerca ed esplorazione dell'applicazione; contiene dei caroselli di poster di film e serie TV oltre ad un campo di testo per inserire una query di ricerca attraverso cui ricercare film, serie TV, persone ed utenti.
- `EpisodesView`: vista di monitoraggio degli episodi; contiene l'elenco del prossimo episodio da vedere per ognuna delle serie TV in visione.
- `MovieDetailsView`: vista di dettaglio di uno specifico film; contiene tutte le informazioni dettagliate sul film, oltre alle varie interfacce per aggiungere il film a delle liste o condividerlo.
- `TVShowDetailsView`: vista di dettaglio di una specifica serie TV; contiene tutte le informazioni dettagliate sulla serie TV, oltre alle varie interfacce per aggiungere la serie TV a delle liste, condividerla oppure monitorare i suoi episodi.
- `SeasonDetailsView`: vista di dettaglio di una stagione di una serie TV; contiene le informazioni principali della stagione, l'elenco dei suoi episodi e le interfacce per il loro monitoraggio.
- `PersonDetailsView`: vista di dettaglio di una persona; contiene le informazioni dettagliate sulla persona e l'elenco dei prodotti relativi ad essa.
- `UserDetailsView`: vista di dettaglio di un utente, anche chiamata profilo utente; contiene le informazioni sull'utente, le sue liste, le sue recensioni e un istogramma con le sue valutazioni.
- `ListDetailsView`: vista di dettaglio di una lista di un utente; contiene una griglia con i poster dei film e delle serie TV aggiunti alla lista.

- `CreateListView`: vista per la creazione di una lista personalizzata; contiene i campi di testo per id e nome della lista e un picker per la tipologia della lista.
- `EditProfileView`: vista per la modifica del profilo utente; contiene le interfacce per modificare nome visualizzato, immagine di profilo e immagine di sfondo.
- `ChangeBackdropView`: vista per la modifica dell'immagine di sfondo; contiene una griglia di immagini di sfondo relative ai prodotti preferiti dall'utente.
- `SettingsView`: vista per le impostazioni dell'app; contiene tutte le personalizzazioni dell'applicazione.
- `ChatsListView`: vista per l'elenco delle chat in cui l'utente è coinvolto. Ogni elemento contiene le informazioni sull'interlocutore e sull'ultimo messaggio ricevuto.
- `ChatView`: vista di dettaglio di una specifica chat; contiene le informazioni sull'interlocutore e l'elenco dei messaggi.
- `HistogramView`: vista personalizzata che, dato un array di valutazioni da 0.5 a 5, crea un istogramma esteticamente appagante basato sui valori dell'array e aggiunge informazioni come il numero totale di valutazioni e la loro media.
- `ClearableTextField`: vista personalizzata che rappresenta un campo di testo il cui contenuto può essere cancellato con un apposito pulsante.
- `ToggleableSecureField`: vista personalizzata che rappresenta un campo di testo sicuro il cui contenuto può essere mostrato con un apposito pulsante.

4.3.4 Classi Repository

Le classi Repository rappresentano un ulteriore livello di astrazione per la gestione dei dati. Nel contesto di WatchWise, esse sono utilizzate per recuperare i dati dal database o dall'API di TMDB. Le classi Repository implementate sono tutte delle `class` e sono presentate nel seguente elenco:

- `MoviesRepository`: questa classe si occupa di recuperare le informazioni relative ai film dall'API di TMDB.
- `TVShowsRepository`: questa classe si occupa di recuperare le informazioni relative alle serie TV e alle stagioni dall'API di TMDB.
- `PeopleRepository`: questa classe si occupa di recuperare le informazioni relative alle persone dall'API di TMDB.
- `UsersRepository`: questa classe si occupa di recuperare le informazioni relative agli utenti di WatchWise dal database Firestore.

4.3.5 Classi ausiliarie

Infine, le classi ausiliarie sono state utilizzate per fornire funzionalità supplementari o per gestire compiti specifici che non rientravano direttamente nelle responsabilità delle altre classi o che dovevano essere svolti da molteplici classi ugualmente. Le classi ausiliarie implementate in WatchWise sono le seguenti:

- `FirestoreService`: questa classe implementa un lungo elenco di metodi che interagiscono con il database Firestore per la lettura, la scrittura, l'aggiornamento e l'eliminazione (CRUD) di dati.
- `AuthManager`: questa classe è una sorta di ViewModel che si interfaccia, però, con tutte le viste dell'intero flusso di splash screen, login e registrazione. Si comporta anche come `@EnvironmentObject` fornendo a molte altre viste dell'applicazione l'id dell'utente corrente.
- `Utils`: questa classe implementa una serie di metodi di utilità, tra cui la conversione di oggetti `Date` in stringhe o la conversione di valori monetari in stringhe.
- *Classi di interfaccia con l'API*: sono quattro classi (`APIConstants`, `APIRouter`, `APIService`, `APIManager`) che si occupano di realizzare la struttura di interfacciamento tra l'app e l'API di TMDB tramite la libreria Alamofire.

4.4 Analisi del codice implementativo

In questa sezione verranno analizzati alcuni punti particolarmente significativi del codice implementativo dell'applicazione, permettendo non solo di comprendere come le funzionalità sono state effettivamente realizzati, ma anche di verificare la qualità del codice, la sua manutenibilità e l'efficienza delle soluzioni adottate.

Prima di cominciare con la spiegazione del codice, è importante porre alcune osservazioni su delle scelte effettuate globalmente per il progetto. Tra queste citiamo le seguenti:

- *Separazione delle directory*: a livello di organizzazione del progetto, sono state create delle directory con un nome chiaro e conciso, ciascuna rappresentante una sezione della logica di business (ovvero, `Repositories`, `ViewModels`, `Views/Details`, etc.).
- *Naming convention*: tutti i file, le classi, le strutture, le funzioni e le variabili sono stati denominati in modo significativo e rigoroso, utilizzando nomi che riflettono chiaramente la loro funzione o il loro scopo ma, soprattutto, mantenendo una coerenza di denominazioni tra elementi della stessa tipologia.
- *Commenti*: nonostante nell'ambito di questa tesi i commenti non verranno mostrati, nel codice sorgente ogni metodo o blocco di codice di particolare importanza è stato adeguatamente commentato.
- *Riutilizzo del codice*: ove possibile, sono state create funzioni o viste riutilizzabili con il fine di evitare la duplicazione del codice, seguendo il principio DRY (Don't Repeat Yourself).

4.4.1 Avvio dell'applicazione

Nel momento in cui l'applicazione WatchWise viene avviata, viene istanziato il componente principale denominato `WatchWiseApp`. Questo componente rappresenta il punto di ingresso dell'intera applicazione e determina quale interfaccia mostrare all'utente in base allo stato di autenticazione attuale, come indicato nel Listato 4.2.

La responsabilità principale di `WatchWiseApp` consiste nel determinare e mostrare una di tre possibili viste basandosi sullo stato corrente dell'autenticazione dell'utente, il quale viene definito in base al risultato dell'inizializzazione della classe ausiliaria `AuthManager` (discussa nella Sezione 4.3.5).

```
@main
struct WatchWiseApp: App {
    @UIApplicationDelegateAdaptor(AppDelegate.self) var delegate

    @StateObject var authManager = AuthManager()
    var body: some Scene {
        WindowGroup {
            switch authManager.authenticationState {
            case .unauthenticated, .authenticating:
                AuthenticationView()
                    .environmentObject(authManager)
            case .authenticated:
                HomeNavigationView()
                    .environmentObject(authManager)
            case .openingApp:
                SplashScreen()
            }
        }
    }
}
```

Listato 4.2: Entry point dell'applicazione

Come mostrato nel Listato 4.3, nel file contenente la classe `AuthManager` è definita anche un'enumerazione, `AuthenticationState`, rappresentante i possibili stati dell'autenticazione dell'utente; questi possono essere non autenticato, autenticazione in corso, autenticato e apertura dell'app in corso. Durante l'inizializzazione della classe `AuthManager`, lo stato iniziale viene impostato come `.openingApp`, facendo sì che l'applicazione mostri la vista `SplashScreen`. Immediatamente dopo, il metodo `registerAuthStateHandler()` viene invocato. Tale metodo controlla lo stato attuale dell'utente, determinando se è autenticato o meno. A seconda dello stato restituito, la `WatchWiseApp` decide di mostrare la vista `AuthenticationView()` (nel caso in cui l'utente non sia autenticato) o la `HomeNavigationView()` (se l'utente è già autenticato).

4.4.2 Flusso di autenticazione

Le diverse viste incaricate della gestione dell'autenticazione dell'utente fanno tutte riferimento ad un unico ViewModel, rappresentato dalla classe `AuthManager`. Questa scelta architetturale è fondamentale per assicurare che le informazioni fornite attraverso le varie viste siano condivise, minimizzando così il rischio di incoerenze. Il processo di autenticazione è suddiviso in tre distinti sotto-flussi: login (tramite credenziali di accesso o servizi di terze parti), registrazione attraverso indirizzo email e password e, infine, registrazione sfruttando servizi esterni.

Flusso di login

L'intero flusso di autenticazione prende avvio dalla vista `LoginView`. Questa interfaccia fornisce all'utente i campi di testo necessari per l'inserimento delle proprie credenziali. Inoltre, sono presenti pulsanti per permettere l'accesso sia attraverso le credenziali standard che mediante servizi terzi. Ulteriori pulsanti guidano l'utente verso le altre opzioni di autenticazione, offrendo un'esperienza d'uso intuitiva.

La pressione del pulsante di login attraverso indirizzo email e password invoca il metodo `signInWithEmailPassword()` appartenente al ViewModel. La specifica implemen-

```swift
enum AuthenticationState {
    case unauthenticated
    case authenticating
    case authenticated
    case openingApp
}

@MainActor
class AuthManager: ObservableObject {
    /* [...] */

    @Published var authenticationState: AuthenticationState = .unauthenticated
    @Published var user: FirebaseAuth.User?
    private var loggingWithGoogle: Bool = false

    /* [...] */

    init() {
        self.authenticationState = .openingApp
        registerAuthStateHandler()

        /* [...] */

    }

    private var authStateHandler: AuthStateDidChangeListenerHandle?

    func registerAuthStateHandler() {
        if authStateHandler == nil {
            authStateHandler = Auth.auth()
                .addStateDidChangeListener { auth, user in
                self.user = user
                self.authenticationState = user == nil ||
                self.loggingWithGoogle ? .unauthenticated : .authenticated
            }
        }
    }

    /* [...] */
}
```

Listato 4.3: Inizializzazione della classe `AuthManager`

tazione di questo metodo è illustrata nel Listato 4.4. Tale codice rappresenta la logica di autenticazione tramite email e password.

In breve, il metodo mostrato nel Listato 4.4 implementa le seguenti attività:

1. *Impostazione dello stato di autenticazione*: all'inizio dell'esecuzione, lo stato di autenticazione è impostato su `.authenticating`, indicando che il processo di verifica delle credenziali è in corso.

2. *Blocco `try-catch`*: qui, l'esecuzione asincrona cerca di effettuare l'accesso utilizzando le credenziali fornite (email e password). Se l'accesso ha successo, il metodo ritorna `true` e, in un'altra sezione del codice, lo stato di autenticazione viene impostato a `.authenticated`.

3. *Gestione degli errori*: nel caso si verifichi un errore durante il tentativo di login (ovvero, credenziali errate), l'errore viene stampato nella console, la variabile `errorLogin`

```
func signInWithEmailPassword() async -> Bool {
        authenticationState = .authenticating
        do {
            try await Auth.auth().signIn(withEmail: self.email,
                password: self.password)
            return true
        }
        catch  {
            print(error)
            errorLogin = true
            authenticationState = .unauthenticated
            return false
        }
    }
```

Listato 4.4: Login con indirizzo email e password

viene impostata su `true` e lo stato di autenticazione ritorna a `.unauthenticated`, segnalando che l'autenticazione non è andata a buon fine. In seguito, il metodo restituisce `false`, fornendo un feedback sul risultato negativo dell'operazione.

Questo approccio, basato sulla programmazione asincrona, garantisce un'esperienza utente fluida, poiché evita blocchi indesiderati nell'interfaccia durante la fase di autenticazione.

Flusso di registrazione tramite email e password

La pressione del pulsante di registrazione tramite email porta all'avvio del corrispondente flusso, il quale inizia con il caricamento della vista `EmailRegistrationView`. Quest'ultima offre diversi campi di testo, ovvero uno per inserire l'indirizzo email, uno per definire una password e un ulteriore campo per confermare la password digitata. Infine, un pulsante di conferma permette di avviare il controllo dei dati inseriti e, se questi sono corretti, di continuare il processo di registrazione. Il codice dietro questa logica è presentato nei Listati 4.5 e 4.6 e svolge le seguenti operazioni:

1. `continueRegistration()`: questa funzione gestisce il flusso di registrazione. Inizializza diversi flag di errore a `false` resettando, così, eventuali errori precedentemente presenti, verifica la validità dell'email e dell'esistenza dell'email nel database e controlla la validità e la corrispondenza della password inserita. Se ci sono errori, viene attivato un feedback aptico, altrimenti viene permesso l'avanzamento nel processo di registrazione. Le operazioni effettuate da questa funzione vengono eseguite attraverso altre funzioni ausiliarie descritte qui di seguito.

2. `checkEmail()`: controlla se l'email inserita è vuota e se corrisponde ad un formato di email valido.

3. `checkPassword()`: verifica se la password inserita è valida e corrisponde alla conferma della password.

4. `triggerHapticFeedback()`: genera un feedback aptico in caso di errori durante la registrazione.

5. `checkEmailExists()`: questa funzione asincrona verifica se l'email fornita esiste già nel database Firestore, utilizzando una query.

```
func continueRegistration() async {
    emailError = false
    emailExistsError = false
    passwordNotValidError = false
    passwordConfirmationError = false
    checkEmail()
    do {
        let emailExists = try await checkEmailExists()
        if emailExists {
            emailExistsError = true
        }
    } catch {
        print(error)
        emailExistsError = true
    }
    checkPassword()
    if emailError || emailExistsError || passwordNotValidError
       || passwordConfirmationError {
        triggerHapticFeedback()
    } else {
        authManager.shouldNavigate = true
    }
}
```

Listato 4.5: Logica della registrazione tramite email e password

6. *Estensioni di `String`*: le estensioni mostrate nel Listato 4.6 aggiungono due metodi per le stringhe: `isValidEmail()` e `isValidPassword()`. Il primo verifica se una stringa è un'email valida attraverso un'espressione regolare, mentre il secondo si assicura che la password contenga almeno 6 caratteri.

```
extension String {
    func isValidEmail() -> Bool {
        let emailRegEx = "[A-Z0-9a-z._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}"
        return NSPredicate(format: "SELF MATCHES %@", emailRegEx)
            .evaluate(with: self)
    }

    func isValidPassword() -> Bool {
        return self.count >= 6
    }
}
```

Listato 4.6: Estensioni per la validazione di email e password

Flusso di registrazione tramite servizi di terze parti

L'applicazione WatchWise consente l'autenticazione tramite due servizi di terze parti, ovvero Google Sign-In e Sign In With Apple. All'interno della vista `LoginView` è possibile effettuare l'autenticazione anche tramite uno di questi due servizi cliccando sull'apposito pulsante. In questa sezione verrà analizzato il caso di Google Sign-In.

Quando l'utente effettua la pressione sul tasto "Entra con Google", viene eseguito il codice mostrato nel Listato 4.7. Tale codice, in breve, si occupa di verificare se l'account Google

utilizzato dall'utente corrisponde ad un utente già registrato a WatchWise o meno. In base al risultato di questo controllo, l'applicazione mostrerà la vista `HomeNavigationView`, se l'utente era già registrato, oppure la vista `CompleteRegistrationView`, se l'account Google non si era già registrato a WatchWise, permettendogli di completare la sua registrazione tramite Google.

```
Task { () -> Void in
    if await authManager.signInWithGoogle() == true {
        let db = Firestore.firestore()
        let usersRef = db.collection("users")
        let querySnapshot = try await usersRef.whereField("email",
            isEqualTo: authManager.googleEmail).getDocuments()
        if querySnapshot.documents.isEmpty {
            authManager.shouldNavigate = true
            authManager.switchFlow()
        } else {
            authManager.authenticationState = .authenticated
        }
    }
}
```

Listato 4.7: Codice eseguito dal pulsante "Entra con Google"

Per effettuare il suddetto controllo, come è evidente dal Listato 4.7, la vista `LoginView` si affida ad una funzione del ViewModel `AuthManager`, ovvero `signInWithGoogle()`. Il codice di tale funzione è mostrato nel Listato 4.8 e le operazioni che esegue sono riassumibili nel seguente modo:

1. Il processo inizia impostando il flag `loggingWithGoogle` a `true`. Ciò permette di mostrare un indicatore di caricamento nella vista aumentando le informazioni restituite dall'interfaccia utente, la quale non lascia l'utente in sospeso.

2. Successivamente, viene estratto il `clientID` dalla configurazione di Firebase dell'applicazione. Tale codice identificativo è essenziale per la configurazione di `GIDSignIn`, ovvero l'oggetto responsabile dell'autenticazione tramite Google.

3. Viene, poi, impostata una configurazione per `GIDSignIn` utilizzando il `clientID` ottenuto precedentemente e si specifica quale controller verrà presentato per il processo di autenticazione. In questo caso, è il `rootViewController` della finestra principale dell'app. Questa operazione permetterà all'applicazione di presentare una scheda che conterrà la pagina web utilizzata per effettuare l'accesso tramite Google.

4. Viene avviato il processo di autenticazione tramite Google. Una volta autenticato, si ottengono i token necessari dall'utente autenticato.

5. I token ottenuti nel passo precedente sono poi utilizzati per creare un oggetto di tipo `credential` di Firebase Authentication, il quale viene utilizzato per autenticare l'utente nel backend di Firebase. Se l'autenticazione ha successo, viene estratta l'email dell'utente e viene salvata nella variabile `googleEmail` del ViewModel (necessaria per altre operazioni).

6. Infine, se si verifica un errore in qualsiasi momento durante il processo, viene stampata una descrizione dell'errore, il flag `loggingWithGoogle` viene reimpostato su `false` e la funzione ritorna `false`. Se, invece, tutto va come previsto, la funzione ritorna `true`.

```
func signInWithGoogle() async -> Bool {
    loggingWithGoogle = true
    guard let clientID = FirebaseApp.app()?.options.clientID else {
        fatalError("No client ID fount in Firebase configuration")
    }
    let config = GIDConfiguration(clientID: clientID)
    GIDSignIn.sharedInstance.configuration = config

    guard let windowScene = UIApplication.shared
            .connectedScenes.first as? UIWindowScene,
          let window = windowScene.windows.first,
          let rootViewController = window.rootViewController else {
        return false
    }

    do {
        let userAuthentication = try await GIDSignIn
            .sharedInstance.signIn(withPresenting: rootViewController)

        let user = userAuthentication.user
        guard let idToken = user.idToken else { throw AuthenticationError
            .tokenError(message: "ID token missing") }
        let accessToken = user.accessToken

        let credential = GoogleAuthProvider
        .credential(withIDToken: idToken.tokenString,
                    accessToken: accessToken.tokenString)

        let result = try await Auth.auth().signIn(with: credential)
        let firebaseUser = result.user
        self.googleEmail = firebaseUser.email ?? "Error"
        loggingWithGoogle = false
        return true
    }
    catch {
        print(error.localizedDescription)
        loggingWithGoogle = false
        return false
    }
}
```

Listato 4.8: Metodo per l’autenticazione tramite Google

Schermata di completamento della registrazione

Quando il metodo `signInWithGoogle()` restituisce `true`, il codice del Listato 4.7 verifica se l’account Google corrisponde ad un utente già registrato a WatchWise. In caso negativo, viene presentata la vista `CompleteRegistrationView`. Tale vista è anche accessibile cliccando sul pulsante "Continua" nella `EmailRegistrationView` e contiene due campi di testo, uno per l’inserimento del nome utente e uno per il nome visualizzato, oltre a un selettore di immagini per l’immagine del profilo.

La peculiarità di questa schermata è proprio il selettore di immagini. Infatti, tale elemento della vista è composto da un’immagine circolare centrale e da un pulsante posto lateralmente. Inizialmente l’immagine circolare centrale, contiene un’icona di sistema rappresentante un utente generico e il pulsante accanto permette di modificare in modo casuale il colore di tale immagine. Tuttavia, toccando l’immagine di sistema verrà aperto il vero e proprio selettore di immagini il quale permetterà all’utente di selezionare un’immagine dalla propria galleria.

```
if let data = data, let uiimage = UIImage(data: data) {
    ZStack {
        PhotosPicker(selection: $profileImage, matching: .images) {
            Image(uiImage: uiimage).resizable().scaledToFill()
                .frame(width: 150, height: 150).clipShape(Circle())
        }.onChange(of: profileImage) { newValue in
            item.loadTransferable(type: Data.self) { result in
                switch result {
                case .success(let data):
                    if let data = data {
                        self.data = data
                        authManager.profileImage = UIImage(data: data)
                    }
                /* [...] */
            }
        }
        Button {
            self.data = nil
            authManager.profileImage = nil
        } label: {
            Image(systemName: "xmark.circle.fill").resizable().tint(.red)
                .frame(width: 36, height: 36)
        }.offset(x: 125, y: 0)
    }
} else {
    ZStack {
        PhotosPicker(selection: $profileImage, matching: .images) {
            Image(systemName: "person.crop.circle.fill").resizable()
                .frame(width: 150, height: 150)
                .foregroundStyle(defaultPropicColor)
        }.onChange(of: profileImage) { newValue in
            item.loadTransferable(type: Data.self) { result in
                switch result {
                case .success(let data):
                    if let data = data {
                        self.data = data
                        authManager.profileImage = UIImage(data: data)
                    }
                /* [...] */
            }
        }
        Button {
            defaultPropicColor = systemColors.randomElement()!
        } label: {
            Image(systemName: "arrow.clockwise.circle.fill").resizable()
                .tint(defaultPropicColor)
                .frame(width: 36, height: 36)
        }.offset(x: 125, y: 0)
    }
}
```

Listato 4.9: Logica di gestione dell'immagine di profilo

Questa nuova immagine sostituirà quella di sistema. Una volta inserita tale immagine, il pulsante per il cambio del colore verrà sostituito con un pulsante che permette di eliminare l'immagine selezionata e di tornare alle immagini di sistema colorate. Il codice che gestisce la logica appena descritta è mostrato nel Listato 4.9.

4.4.3 HomeNavigationView

Nel momento in cui l'utente completa la sua autenticazione, cioè quando lo stato di autenticazione di AuthManager passa a .authenticated, l'entry point WatchWiseApp presenterà la vista HomeNavigationView. Tale vista è il fulcro della navigazione centrale dell'applicazione e consente di raggiungere tutte le schermate principali. Come mostrato nel Listato 4.10, la vista HomeNavigationView presenta una TabView, ovvero la barra di navigazione inferiore di SwiftUI, contenente le cinque viste principali dell'applicazione, ognuna indicata da una Label contenente un testo e un'icona rappresentativa.

```
struct HomeNavigationView: View {
    @EnvironmentObject var authManager: AuthManager
    @State private var tabSelection = 0

    var body: some View {
        TabView(selection: $tabSelection) {
            HomeView()
                .tabItem {
                    Label("Home", systemImage: "house")
                }.tag(0)
            FeedView()
                .tabItem {
                    Label("Feed", systemImage: "bell")
                }.tag(1)
            SearchView()
                .tabItem {
                    Label("Esplora", systemImage: "magnifyingglass")
                }.tag(2)
            EpisodesView(currentUserUid: authManager.currentUserUid)
                .tabItem {
                    Label("Episodi", systemImage: "tv.inset.filled")
                }.tag(3)
            UserDetailsView(uid: authManager.currentUserUid,
                            currentUserUid: authManager.currentUserUid)
                .tabItem {
                    Label("Profilo", systemImage: "person")
                }.tag(4)
                .environmentObject(authManager)
        }
    }
}
```

Listato 4.10: Struttura della vista HomeNavigationView

Schermata di esplorazione

Alcune viste di particolare interesse raggiungibili dalla HomeNavigationView verranno discusse successivamente. Ora, però, verrà analizzata la SearchView. Tale vista fornisce le funzionalità di esplorazione e di ricerca dell'applicazione. Il codice di particolare rilevanza eseguito nel contesto di questa vista è quello che si occupa di effettuare la ricerca e di mostrare i risultati.

Il Listato 4.11 illustra il codice delle funzioni che gestiscono la ricerca dei film all'interno dell'applicazione e come i risultati vengono mostrati all'utente.

La funzione searchMovies ha il compito di avviare il processo di ricerca dei film basato sulla query fornita dall'utente e sulla pagina specificata. Essa effettua le seguenti operazioni:

```
func searchMovies(query: String, page: Int32) {
    moviesTask?.cancel()

    moviesTask = Task {
        await performMoviesSearch(for: query, page: page)
    }
}

func performMoviesSearch(for query: String, page: Int32) async {
    do {
        isLoading = true
        try await Task.sleep(nanoseconds: 1_000_000_000)
        APIManager.searchMovies(query: query,
            page: page) { (result: AFResult<DiscoverMoviesResponse>) in
            isLoading = false
            switch result {
            case .success(let movies):
                searchMoviesResults = movies.results
            case .failure(let error):
                print("Error searching movies: \(error)")
            }
        }
    } catch {
        print(error)
    }
}

func loadMoreMovies() {
    guard !isLoading else {
        return
    }

    isLoading = true
    currentMoviesPage += 1
    APIManager.searchMovies(query: searchText,
        page: currentMoviesPage) { (result: AFResult<DiscoverMoviesResponse>) in
        isLoading = false
        switch result {
        case .success(let movies):
            if currentMoviesPage <= movies.total_pages {
                searchMoviesResults!.append(contentsOf: movies.results)
            }
        case .failure(let error):
            print("Error searching movies: \(error)")
        }
    }
}
```

Listato 4.11: Funzionalità di ricerca dei film

1. *Annullamento delle operazioni in corso*: il primo passo eseguito dalla funzione è l'annullamento di qualsiasi task di ricerca precedentemente avviato. Questo è fondamentale per evitare conflitti o sovrapposizioni tra ricerche multiple, in particolare se l'utente dovesse digitare rapidamente nella barra di ricerca.

2. *Creazione di una nuova task*: la funzione avvia un nuovo task asincrono in cui viene chiamata la funzione `performMoviesSearch`, che si occuperà effettivamente della ricerca.

La funzione `performMoviesSearch` si occupa di effettuare la vera e propria ricerca. Le operazioni che effettua sono le seguenti:

1. *Impostazione dello stato di caricamento*: la variabile `isLoading` viene impostata su `true`, segnalando che una ricerca è in corso.
2. *Ritardo dell'operazione*: viene inserito un ritardo di 1 secondo prima di procedere con la chiamata effettiva all'API. Questo può servire come *debouncing*, garantendo che non vengano effettuate troppe chiamate all'API in breve tempo, soprattutto se l'utente sta digitando rapidamente nella barra di ricerca.
3. *Chiamata API*: viene eseguita una chiamata all'API attraverso, `searchMovies`, una funzione implementata nella classe `APIManager`. Una volta ricevuti i risultati, lo stato di caricamento viene reimpostato su `false` e i risultati vengono assegnati alla variabile appropriata (la quale viene rappresentata graficamente nella vista), oppure viene stampato un errore, se presente.

La funzione `loadMoreMovies` viene chiamata per caricare ulteriori risultati, tipicamente quando l'utente raggiunge la fine della lista dei risultati mostrati. Le operazioni che effettua sono le seguenti:

1. *Verifica dello stato di caricamento*: la funzione si assicura che non ci siano ricerche in corso, se ci sono ferma la sua esecuzione.
2. *Incremento della pagina corrente*: la funzione incrementa la variabile `currentMoviesPage` di uno, indicando che si desidera caricare la prossima pagina di risultati.
3. *Chiamata API*: la funzione effettua una chiamata API similmente a come accade nella funzione `performMoviesSearch`, ma questa volta viene aggiunto un controllo per verificare se la pagina corrente è inferiore o uguale al numero totale di pagine disponibili. Se è così, i nuovi risultati vengono semplicemente aggiunti alla lista esistente.

Infine, è importante notare che il codice sopra presentato è specifico per la ricerca di film. Per altri elementi, come serie TV o persone, il codice sarebbe analogo; quindi, per brevità, non è stato incluso qui.

4.4.4 Schermate di dettaglio

Le schermate di dettaglio di WatchWise sono schermate che mostrano i dettagli di uno specifico elemento. Esistono cinque schermate di dettaglio, ognuna per una specifica tipologia di elemento: film, serie TV, persone, utenti e liste. Tutte queste schermate sono implementate attraverso migliaia di righe di codice contenute in file diversi e presentano moltissime funzionalità. In questa sezione verranno analizzate alcune delle funzionalità più peculiari di queste schermate.

Gestione delle liste

La prima funzionalità di interesse si trova nelle classi `MovieDetailsViewModel` e `TVShowDetailsViewModel` e riguarda l'aggiunta o la rimozione di un prodotto in una lista. Il codice della funzione che implementa questa funzionalità nel caso dei film è mostrato nel Listato 4.12.

```
func toggleMovieToList(listName: String, movieRuntime: Int?) async throws {
    if let isInList = isInList[listName] {
        if isInList {
            firestoreService.removeProductFromList(self.movieId,
                    listName: listName, userId: self.currentUserUid,
                    type: "movies") { error in
                if let error = error {
                    /* [...] */
                    return
                }
                self.isInList[listName] = false
            }
            if listName == "watched_m" {
                try await firestoreService
                          .incrementUserField(userId: currentUserUid,
                          type: "movieMinutes", number: -(movieRuntime ?? 120))
                try await firestoreService
                          .incrementUserField(userId: currentUserUid,
                          type: "movieNumber", number: -1)
            }
        } else {
            firestoreService.addProductToList(self.movieId,
                    listName: listName, userId: self.currentUserUid,
                    type: "movies") { error in
                if let error = error {
                    /* [...] */
                    return
                }
                self.isInList[listName] = true
            }
            if listName == "watched_m" {
                try await firestoreService
                          .incrementUserField(userId: currentUserUid,
                          type: "movieMinutes", number: movieRuntime ?? 120)
                try await firestoreService
                          .incrementUserField(userId: currentUserUid,
                          type: "movieNumber", number: 1)
            }
        }
    }
}
```

Listato 4.12: Implementazione dell'aggiunta e della rimozione di un film in una lista

Tale codice effettua le seguenti operazioni:

1. Verifica se il film è presente nella lista in questione oppure no. Questa verifica viene effettuata attraverso la variabile `isInList` di tipo `[String: Bool]`, che viene inizializzata alla creazione del ViewModel attraverso un'altra funzione.

2. Se il film è già presente nella lista, allora viene chiamata una funzione della classe `FirestoreService` che rimuove il prodotto dalla lista e, successivamente, imposta il valore corrispondente alla lista in questione nella variabile `isInList` a `false`.

3. Se il film non è presente nella lista, allora viene chiamata una funzione della classe `FirestoreService` che si occupa di aggiungere il prodotto alla lista e, poi, imposta il valore corrispondente alla lista in questione nella variabile `isInList` a `true`.

4. Dopo ognuna delle due possibilità, la funzione `toggleMovieToList` effettua un'operazione aggiuntiva nel caso in cui la lista corrente sia la lista dei film guardati; quando la lista viene rimossa, allora viene chiamata una funzione di `FirestoreService` che decrementa i valori dell'utente relativi al numero e alla durata totale dei film guardati; quando la lista viene aggiunta, gli stessi valori vengono incrementati.

Schermata di dettaglio delle persone

Nel ViewModel della schermata di dettaglio delle persone, `PeopleDetailsViewModel`, una funzione di particolare interesse è quella che si occupa di ottenere i dettagli della persona in questione. Il codice di tale funzione è mostrato nel Listato 4.13.

```
1  func getPersonDetails() {
2      repository.getPersonDetails(by: personId) { person in
3          var sortedPerson = person
4
5          if let cast = person?.credits?.cast {
6              sortedPerson?.credits?.cast = cast.sorted(by: {
7              (Double($0.voteCount) * $0.voteAverage) >
8                  (Double($1.voteCount) * $1.voteAverage)
9              })
10         }
11
12         if let crew = person?.credits?.crew {
13             sortedPerson?.credits?.crew = crew.sorted(by: {
14             (Double($0.voteCount) * $0.voteAverage) >
15                 (Double($1.voteCount) * $1.voteAverage)
16             })
17         }
18
19         self.person = sortedPerson
20     }
21 }
```

Listato 4.13: Logica di ottenimento dei dettagli delle persone

La funzione `getPersonDetails`, mostrata nel listato, innanzitutto recupera i dettagli della persona attraverso la chiamata ad un'omonima funzione presente nella classe repository dedicata alle persone. Dopo aver ottenuto i dettagli, effettua un ordinamento dei due array di `cast` e `crew`. L'array di `cast` contiene l'elenco dei prodotti in cui la persona ha partecipato come attore; l'array di `crew` contiene l'elenco dei prodotti in cui la persona ha partipato come staff. Questi due array vengono ordinati in base alle valutazioni dei prodotti, in modo che in alto nella lista vengano mostrati i prodotti con una valutazione maggiore.

Tempi di visione

Nella schermata di dettaglio degli utenti, `UserDetailsView`, tra le informazioni relative all'utente in questione vengono mostrati i tempi di visione di film e di serie TV. Questi tempi vengono memorizzati nel database sotto forma di minuti totali, ma nella schermata vengono mostrati come gruppi di mesi, giorni e ore. È, quindi, stata necessaria una funzione che si occupasse di convertire i minuti in mesi, giorni e ore. Questa funzione si chiama `convertMinutesToTimeComponents` ed è stata implementata nella classe ausiliaria `Utils`. Il codice di questa funzione è mostrato nel Listato 4.14.

```
static func convertMinutesToTimeComponents(minutes: Int) -> (months: Int,
                                                           days: Int, hours: Int) {
    let totalHours = minutes / 60
    let days = totalHours / 24
    let months = days / 30 // Considerando una media di 30 giorni per mese

    let remainingHours = totalHours % 24
    let remainingDays = days % 30

    return (months: months, days: remainingDays, hours: remainingHours)
}
```

Listato 4.14: Metodo per la conversione di minuti in mesi, giorni e ore

Schermata di dettaglio delle liste

Nella classe `ListDetailsViewModel` sono presenti due funzioni che si occupano di recuperare i dettagli delle liste, tra cui il loro nome, la loro tipologie e il loro contenuto. Queste funzioni sono particolarmente interessanti dal momento che le liste di WatchWise sono degli elementi molto flessibili e queste funzioni riescono ad ottimizzare le chiamate all'API e al database in base alle caratteristiche variabili delle liste. Il codice di queste due funzioni è mostrato nel Listato 4.15.

Nello specifico, quando si naviga alla schermata di dettaglio di una lista, viene chiamata la funzione `fetchProducts`, la quale, innanzitutto, chiama la funzione `getListDetails`. Quest'ultima, attraverso la classe ausiliaria `FirestoreService` ottiene i dettagli della lista, ovvero: eventuale elenco di ID di film, eventuale elenco di ID di serie TV, tipologia della lista[2] e nome della lista. Una volta ottenuti i dettagli della lista, la funzione `fetchProducts` si occupa di popolare gli array di prodotti del ViewModel in base alla tipologia della lista. Se la lista è solo di film o di entrambe le tipologie, il ViewModel effettuerà le dovute chiamate al repository dei film per ottenere i dettagli di ogni film da mostrare nella schermata; se, invece, la lista è solo di serie TV o di entrambe le tipologie, il ViewModel effettuerà le dovute chiamate al repository delle serie TV per ottenere i dettagli di ogni serie TV da mostrare nella schermata. In questo modo, è stata implementata una logica che varia in base alla tipologia della lista, in modo da evitare chiamate superflue all'API.

4.4.5 Schermate di personalizzazione

Le schermate di personalizzazione di WatchWise sono la schermata di modifica del profilo, la schermata di modifica dell'immagine di sfondo e la schermata per la creazione di liste personalizzate. La prima schermata, `EditProfileView`, si occupa della personalizzazione della pagina di profilo, permettendo di cambiare alcune informazioni dell'utente, tra cui nome visualizzato, immagine di profilo ed immagine di sfondo. La scelta dell'immagine di sfondo viene, a sua volta, effettuata attraverso la seconda schermata di personalizzazione, `ChangeBackdropView`. La terza schermata di personalizzazione, `CreateListView`, permette, invece, la creazione di liste personalizzate.

Queste schermate hanno ognuna il proprio ViewModel, e tra questi ViewModel è implementata una funzionalità di particolare interesse. La classe `EditProfileViewModel` contiene due funzioni che si occupano di modificare l'immagine del profilo dell'utente. L'immagine di profilo viene scelta, in `EditProfileView`, in modo analogo a quanto avviene

[2] Una lista può essere solo di film, solo di serie TV, oppure di entrambi i tipi di prodotto.

```swift
func fetchProducts() async throws {
    let listDetails = try await getListDetails()

    if let listDetails = listDetails {
        if listDetails.type == "movie" || listDetails.type == "both" {
            for movieId in listDetails.movies {
                moviesRepository.getMovieDetails(by: movieId) { movie in
                    self.moviesList.append(movie!)
                }
            }
        }
        if listDetails.type == "tv" || listDetails.type == "both" {
            for showId in listDetails.shows {
                self.tvShowsList.append(try await tvShowsRepository
                    .getTVShowDetails(showId: showId))
            }
        }
    }
    self.isLoaded = true
}

private func getListDetails() async throws -> (movies: [Int64],
         shows: [Int64], type: String, name: String)? {
    let listDetails = try await firestoreService
        .getUserList(userId: currentUserUid, listId: listId)
    self.listName = listDetails?.name ?? ""
    self.listType = listDetails?.type ?? ""
    return listDetails
}
```

Listato 4.15: Logica di ottenimento dei dettagli delle liste

nella `CompleteRegistrationView` (Listato 4.9). Quando, però, si clicca sul pulsante di conferma del cambiamento, viene eseguito il codice mostrato nel Listato 4.16.

Tale codice illustra le operazioni necessarie per modificare l'immagine di profilo di un utente. La funzione principale, `changeProfileImage`, inizia impostando la variabile booleana `sshowProfileImageLoadingAlert` a `true`, che permette di segnalare, attraverso un alert della libreria `AlertToasts`, che il caricamento dell'immagine è in corso. Si crea, quindi, un riferimento alla posizione in Firebase Cloud Storage in cui verrà salvata l'immagine, utilizzando l'UID dell'utente corrente come nome del file. L'immagine scelta dall'utente viene, quindi, ridimensionata chiamando `resizeImage`, che si occupa di scalare l'immagine ad una larghezza target, mantenendo le proporzioni e comprimendo l'immagine per garantire che la sua dimensione non superi un limite specificato (in questo caso, 50 kB). Dopo aver ridimensionato e compresso l'immagine, la funzione tenta di caricarla nel Cloud Storage e, se il caricamento ha successo, aggiorna il campo `profilePath` nel documento Firestore dell'utente con l'URL dell'immagine caricata. Al termine del caricamento, le variabili di alert vengono aggiornate per segnalare che l'operazione è stata completata con successo.

4.5 Conclusioni

Nell'ambito dell'ingegneria del software, la fase di implementazione riveste una significativa importanza, fungendo da ponte tra la progettazione concettuale e la realizzazione pratica di un'applicazione. Questo capitolo ha offerto uno sguardo dettagliato su alcune parti del

```
func changeProfileImage() async {
    showProfileImageLoadingAlert = true
    let storageRef = Storage.storage().reference()
    let uid = Auth.auth().currentUser!.uid
    let profileImageStorageRef = storageRef.child("users/propics/\(uid).jpg")
    guard let resizedImage = resizeImage(targetSize: 50 * 1024,
                                         targetWidth: 500) else {
        return
    }
    guard let imageData = resizedImage.jpegData(compressionQuality: 0.8) else {
        return
    }
    let metadata = StorageMetadata()
    metadata.contentType = "image/jpeg"
    do {
        _ = try await profileImageStorageRef.putDataAsync(imageData,
                                                          metadata: metadata)
        try await changeUserField(field: "profilePath",
            value: profileImageStorageRef.downloadURL().absoluteString)
    } catch {
        print(error)
        return
    }
    showProfileImageLoadingAlert = false
    showProfileImageCompletedAlert = true
}

func resizeImage(targetSize: Int, targetWidth: CGFloat) -> UIImage? {
    let scale = targetWidth / self.profileImage!.size.width
    let newHeight = self.profileImage!.size.height * scale
    UIGraphicsBeginImageContext(CGSize(width: targetWidth, height: newHeight))
    self.profileImage!.draw(in: CGRect(x: 0, y: 0, width: targetWidth,
                                       height: newHeight))
    let resizedImage = UIGraphicsGetImageFromCurrentImageContext()
    UIGraphicsEndImageContext()
    guard let resized = resizedImage else { return nil }
    var compression: CGFloat = 0.9
    let maxCompression: CGFloat = 0.1
    var imageData = resized.jpegData(compressionQuality: compression)
    while (imageData?.count ?? 0) > targetSize && compression > maxCompression {
        compression -= 0.1
        imageData = resized.jpegData(compressionQuality: compression)
    }
    guard let compressedImageData = imageData else { return nil }
    return UIImage(data: compressedImageData)
}
```

Listato 4.16: Logica di modifica dell'immagine di profilo

processo di implementazione, evidenziando le decisioni tecniche prese e le soluzioni adottate per affrontare specifici problemi.

L'analisi dettagliata del codice ha avuto l'obiettivo di fornire un'analisi concreta delle metodologie e tecniche utilizzate, non solo per illustrare il lavoro svolto, ma anche per offrire un quadro chiaro delle soluzioni adottate in risposta a specifiche sfide tecniche. Questa analisi non solo serve come documentazione tecnica, ma anche come un punto di riflessione su come determinate scelte possono influenzare l'efficienza, la scalabilità e la manutenibilità dell'applicazione.

È opportuno sottolineare che le sezioni di codice analizzate rappresentano solo una frazione del codice totale dell'applicazione. Tuttavia, non è il volume del codice in sé ad essere rilevante, ma piuttosto la qualità delle soluzioni implementative e l'efficacia con cui sono state realizzate. Ci sarebbero molti altri segmenti del codice che meriterebbero un'analisi, ma per questioni di concisione e focalizzazione, si è optato per presentare solo quelle sezioni ritenute più significative dal punto di vista ingegneristico.

In sintesi, questo capitolo ha avuto l'intento di approfondire la fase di implementazione, mettendo in luce le scelte tecniche e le strategie adottate. Ogni decisione presa ha avuto ripercussioni sulla performance, sulla sicurezza e sulla user experience dell'applicazione.

CAPITOLO 5

Manuale utente

Il presente capitolo è dedicato a fornire una descrizione metodica delle interfacce e delle funzionalità di WatchWise, strutturandosi come un manuale utente. Benché l'attenzione sia focalizzata sull'uso dell'applicazione, ogni aspetto è presentato con una rigorosa coerenza e precisione. Le scelte di design, l'interazione tra schermate e l'intuitività delle funzioni vengono analizzate in modo da fornire una panoramica dettagliata e approfondita, essenziale per chi intende comprendere appieno le dinamiche di funzionamento di WatchWise.

5.1 Presentazione di WatchWise e dei suoi obiettivi

WatchWise è un'applicazione innovativa sviluppata per gli appassionati di film e serie TV. Progettata con un'interfaccia intuitiva e accattivante, l'applicazione offre agli utenti un'esperienza simile all'utilizzo di un social network, permettendo loro di esplorare, scoprire e condividere le loro passioni cinematografiche e televisive.

L'obiettivo principale di WatchWise è fornire una piattaforma completa dove gli utenti possono non solo tenere traccia dei film e delle serie TV che hanno visto, ma anche scoprire nuovi contenuti, interagire con altri appassionati, personalizzare la loro esperienza in base ai propri gusti e preferenze. L'applicazione si distingue per la sua capacità di integrare funzionalità social con strumenti di ricerca e personalizzazione.

Un altro obiettivo fondamentale di WatchWise è la creazione di una comunità. Gli utenti possono connettersi con altri appassionati, scambiare consigli e valutare e recensire film e serie TV. Questo aspetto sociale dell'applicazione la rende non solo uno strumento per la scoperta di contenuti, ma anche un luogo dove gli appassionati possono condividere le loro opinioni e creare legami con persone che condividono gli stessi interessi.

In termini di design, WatchWise è stata sviluppata con un'attenzione particolare all'usabilità. L'interfaccia è stata progettata per essere chiara e intuitiva, con icone e menu facilmente riconoscibili. Ogni sezione dell'applicazione è stata pensata per offrire all'utente un'esperienza fluida e piacevole.

In sintesi, WatchWise non è solo un'applicazione per tenere traccia dei film e delle serie TV visti e informarsi su prodotti e persone relativi al mondo del cinema, ma un vero e proprio social network dedicato agli amanti del cinema e della televisione, con l'obiettivo di offrire una piattaforma completa e coinvolgente.

5.2 Autenticazione

L'autenticazione è un aspetto cruciale di WatchWise, garantendo che le informazioni e le preferenze di ogni utente siano protette e personalizzate e permettendo le funzionalità social introdotte nella Sezione 5.1. L'applicazione offre diverse opzioni per i processi relativi all'autenticazione, permettendo agli utenti di accedere, registrarsi o recuperare le proprie credenziali in modo semplice e sicuro.

5.2.1 Login

Una volta avviata l'applicazione, l'utente viene accolto dalla schermata di login (Figura 5.1). Qui è possibile registrarsi, resettare la propria password, oppure inserire le proprie credenziali, ovvero indirizzo email e password, e accedere all'account. Se le credenziali inserite sono corrette, l'utente verrà reindirizzato alla schermata principale dell'applicazione, dove potrà iniziare a navigare e interagire con i contenuti di WatchWise. Se ci sono problemi con le credenziali inserite, un messaggio di errore apparirà, guidando l'utente nella correzione del problema, come illustrato nella Figura 5.2.

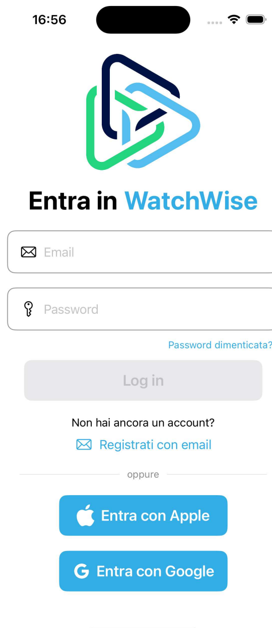

Figura 5.1: Schermata di login

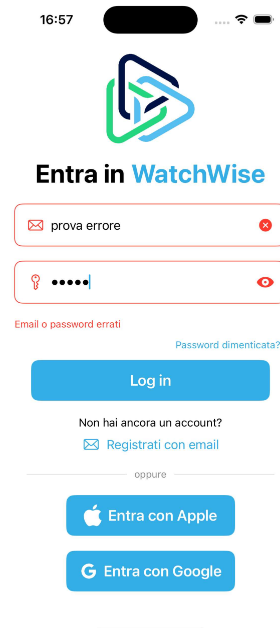

Figura 5.2: Schermata di login con errore

5.2.2 Registrazione

Per gli utenti che non hanno ancora un account su WatchWise, la schermata di login offre anche la possibilità di registrarsi. Toccando l'opzione "Registrati con email", "Entra con Apple" oppure "Entra con Google"[1], l'utente verrà indirizzato alla schermata di registrazione. Qui, sarà necessario fornire alcune informazioni di base, tra cui nome utente, nome visualizzato, indirizzo email e password. La registrazione si compone di due fasi distinte:

- Nella prima fase (Figura 5.3), viene richiesto all'utente l'inserimento di un indirizzo di posta elettronica e di una password, che deve essere confermata.

[1]Negli ultimi due casi, l'utente passa alla fase di registrazione solo se l'account Apple o Google con cui effettua l'accesso non corrisponde ad un account in WatchWise e, in caso accada ciò, l'utente passerà direttamente alla seconda fase di registrazione, mostrata in Figura 5.4.

- Nella seconda fase (Figura 5.4), viene richiesto all'utente l'inserimento di un nome utente - univoco tra tutti gli utenti registrati - seguito da un nome visualizzato. Inoltre, a discrezione dell'utente, viene proposto l'eventuale cambiamento della foto del profilo. Ciò può essere effettuato sia aggiornando il colore della foto del profilo di default, che cliccando sulla foto del profilo di default e scegliendo un'immagine dalla propria galleria.

Una volta completata la registrazione, l'utente potrà accedere a WatchWise e iniziare a personalizzare il proprio profilo e le proprie preferenze oppure ad esplorare i contenuti.

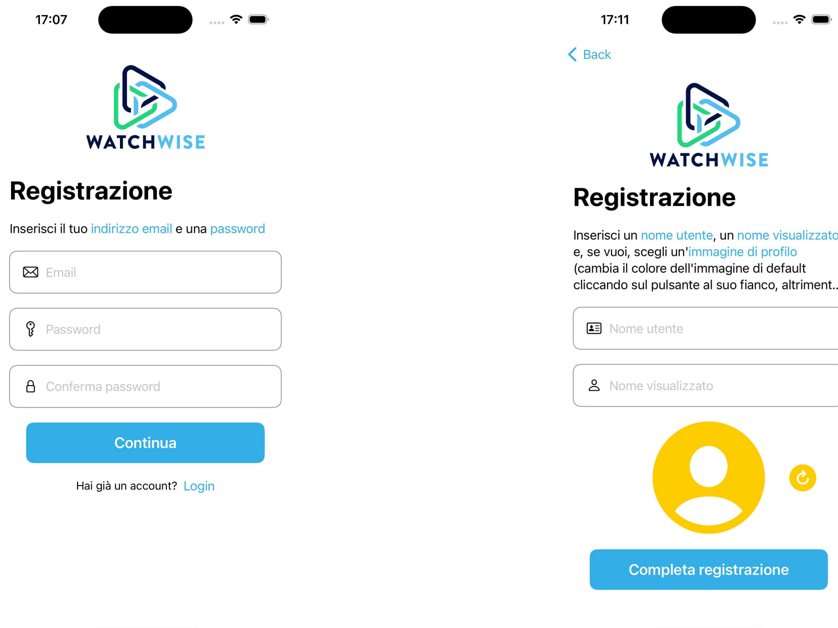

Figura 5.3: Schermata di registrazione (fase I) **Figura 5.4:** Schermata di registrazione (fase II)

5.2.3 Reset della password

Nel caso in cui un utente dimentichi la propria password, WatchWise offre una funzione di reset della stessa. Nella schermata di login, toccando l'opzione "Password dimenticata?", l'utente verrà guidato attraverso un processo che gli permetterà di reimpostare la sua password. Dopo aver inserito l'indirizzo email associato all'account, l'utente riceverà una mail con le istruzioni per creare una nuova password. Seguendo questi passaggi, l'utente potrà facilmente recuperare l'accesso al proprio account, garantendo, al contempo, la sicurezza delle proprie informazioni.

5.3 Schermata principale

La schermata principale di WatchWise è il cuore pulsante dell'applicazione, offrendo agli utenti un accesso rapido e intuitivo alle principali funzionalità e contenuti. Questa schermata è stata progettata per essere il punto di partenza per la navigazione, permettendo agli utenti di esplorare, scoprire e interagire con i contenuti in modo semplice e immediato.

La navigazione può avvenire in modo rapido ed intuitivo grazie alla barra di navigazione inferiore, presente in ognuna delle cinque schermate principali di WatchWise. La barra di navigazione inferiore di WatchWise è mostrata in Figura 5.5.

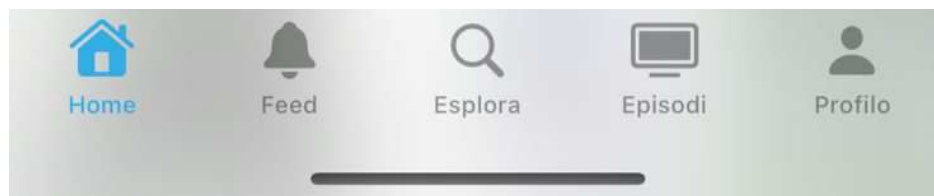

Figura 5.5: Barra di navigazione inferiore

5.3.1 Schermata Home

La schermata *Home* (Figura 5.6) è la prima vista che gli utenti incontrano dopo aver effettuato l'accesso. Essa presenta tre caroselli di poster di film e serie TV, suddivisi in categorie: "Ultime uscite", "Film più graditi" e "Serie TV più gradite". Il primo carosello mostra i film attualmente in onda al cinema e quelli che lo sono stati recentemente; gli altri due caroselli mostrano, rispettivamente, le classifiche di film e serie TV basate sulle valutazioni degli utenti di WatchWise.

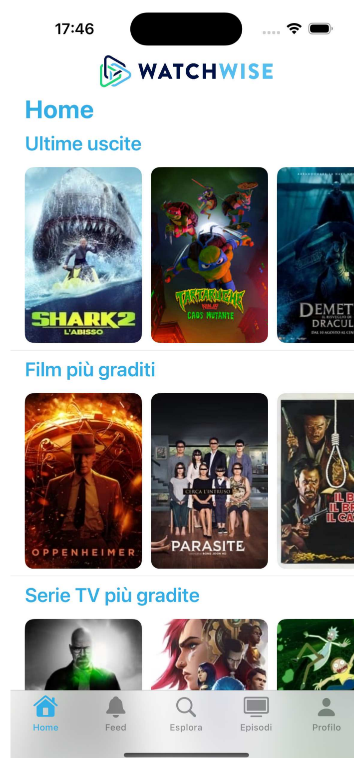

Figura 5.6: Schermata Home

5.3.2 Schermata Feed

La schermata *Feed* offre agli utenti un flusso continuo di aggiornamenti relativi agli utenti seguiti e alle persone per cui si hanno degli interessi, oltre al punto di accesso alla sezione di condivisione. Qui, gli utenti possono vedere le recensioni e le eventuali valutazioni degli utenti che essi seguono, nonché le notizie e gli aggiornamenti sulle ultime uscite. Questa schermata permette, dunque, agli utenti di restare aggiornati sulle attività dei propri amici e sulle novità del mondo cinematografico e televisivo.

Messaggistica

Attraverso la schermata Feed l'utente può anche raggiungere la schermata delle chat, ovvero la schermata in cui vengono elencate tutte le chat a cui egli ha preso parte. La schermata delle chat (Figura 5.7[2]) si presenta come un semplice elenco contenente le varie chat, ognuna rappresentata dall'icona, dal nome utente e dal nome visualizzato dell'interlocutore. Toccando una delle chat, l'utente può navigare alla schermata che contiene la vera e propria chat (Figura 5.8). Tale schermata racchiude lo scambio di consigli tra l'utente corrente e il suo interlocutore.

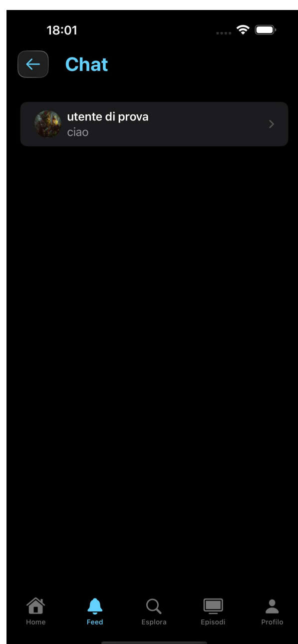

Figura 5.7: Schermata delle chat

Figura 5.8: Esempio di chat

5.3.3 Schermata Esplora

La schermata *Esplora* (Figura 5.9) è dedicata alla scoperta di nuovi contenuti. Appena l'utente accede a tale schermata, saranno presentati quattro caroselli, i quali racchiudono film e serie TV divisi nelle seguenti categorie: "Film popolari", "Serie TV popolari", "Film di tendenza" e "Serie TV di tendenza".

È importante, inoltre, notare che, cliccando su uno qualsiasi dei poster presenti nei caroselli della schermata Esplora (o in altre schermate), si potrà navigare alla schermata di dettaglio del prodotto corrispondente, garantendo, così, un'esperienza utente più fluida e intuitiva.

Ricerca

La schermata Esplora presenta, in alto, una casella di testo. Quando l'utente inizia a digitare del testo in essa, la schermata Esplora si trasformerà in una schermata di ricerca dettagliata. La schermata di ricerca mostrerà all'utente i risultati che corrispondono al testo che ha digitato, fornendogli la possibilità scegliere di visualizzare l'elenco di film, serie TV, persone o utenti di WatchWise corrispondenti al testo digitato. Toccando su uno qualsiasi

[2]Le schermate nelle Figure 5.7 e 5.8 presentano un tema scuro solo per illustrare la disponibilità di un'interfaccia scura in WatchWise.

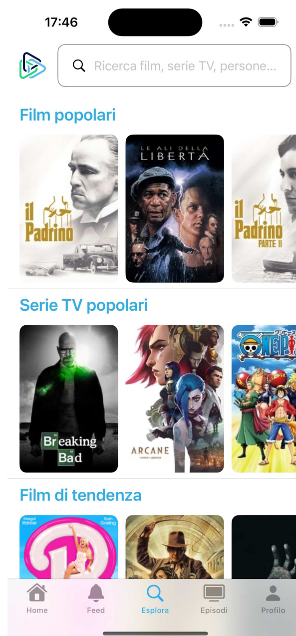

Figura 5.9: Schermata Esplora

degli elementi mostrati, si raggiungerà la corrispondente schermata di dettaglio. Alcuni esempi di ricerca sono mostrati in Figura 5.10.

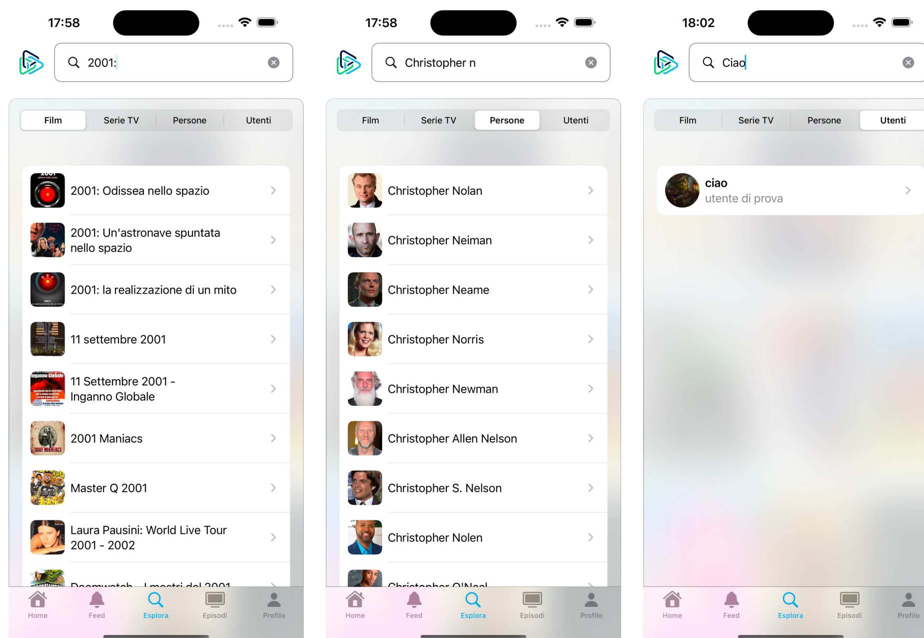

Figura 5.10: Esempi di ricerche

5.3.4 Schermata Episodi

La schermata *Episodi* (Figura 5.11) si concentra esclusivamente sull'offerta di un servizio essenziale agli appassionati di serie TV: mostrare, per ciascuna serie in visione, l'episodio successivo da guardare. Ad esempio, se un utente ha segnato come visti gli episodi 1 e 2 di una serie TV e gli episodi 1, 2 e 3 di un'altra serie TV, la schermata indicherà che l'episodio 3 è

il successivo nella sequenza di visione della prima serie TV, mentre l'episodio 4 è il successivo nella sequenza di visione della seconda serie TV.

All'interno di questa schermata, ogni episodio viene presentato con informazioni chiave: il titolo e il poster della serie TV, il numero della stagione, il numero dell'episodio e il titolo dell'episodio stesso. È anche integrato un pulsante funzionale che consente di contrassegnare l'episodio come "visto" senza dover raggiungere la schermata di dettaglio per farlo, aggiornando contemporaneamente l'elenco mostrato. Se l'utente decide di toccare un episodio al di fuori dell'area del pulsante, verrà reindirizzato alla schermata di dettaglio della serie TV, ottenendo, così, una scorciatoia per l'ottenimento di informazioni aggiuntive sulla serie in questione.

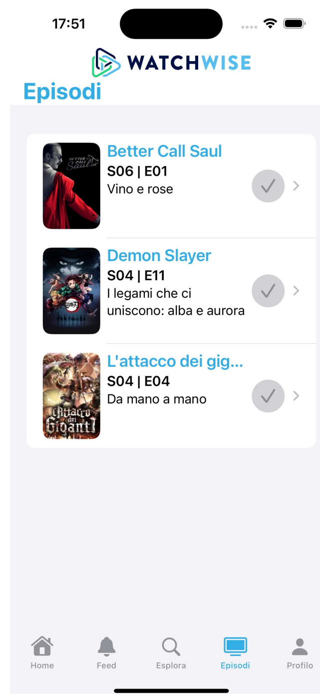

Figura 5.11: Schermata Episodi

5.3.5 Schermata Profilo

La schermata *Profilo* (Figura 5.12) è lo spazio personale di ogni utente all'interno di WatchWise. Qui, gli utenti possono visualizzare e modificare le loro informazioni, vedere le loro liste, controllare le valutazioni e le recensioni lasciate e accedere alle funzionalità di modifica del profilo e alle impostazioni dell'applicazione.

Modifica del profilo

Cliccando il pulsante "Modifica profilo" l'utente può raggiungere la schermata di modifica del profilo (Figura 5.13). Qui, egli ha la possibilità di modificare il proprio nome visualizzato, la propria foto del profilo (così come spiegato nella Sezione 5.2.2) e la propria immagine di sfondo. Per quanto concerne l'immagine di sfondo, questa può essere scelta tra un assortimento di immagini relative ai prodotti inseriti nella propria lista "Preferiti", così come mostrato in Figura 5.14.

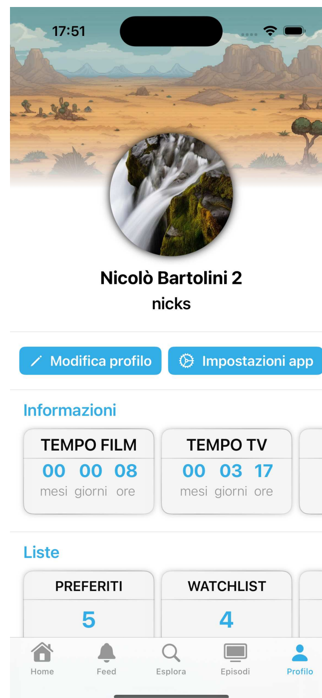

Figura 5.12: Schermata Profilo

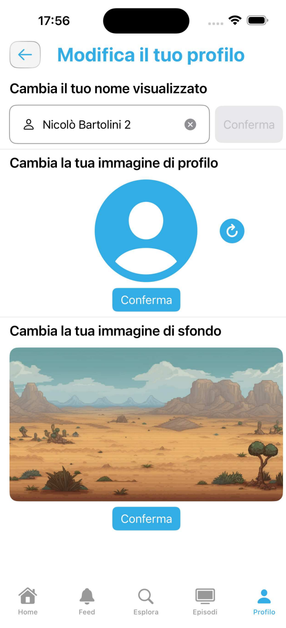

Figura 5.13: Schermata di modifica del profilo

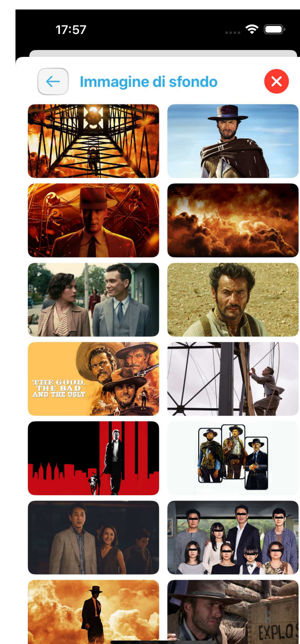

Figura 5.14: Schermata di modifica dell' immagine di sfondo

Liste e liste personalizzate

All'interno della schermata *Profilo*, gli utenti possono accedere in dettaglio alle proprie liste, come illustrato nella Figura 5.15. In questa schermata specifica, il nome della lista è ben visibile e posizionato in modo predominante. Ai fruitori è data la possibilità di selezionare la tipologia di prodotti che desiderano visionare, scelta supportata da un selettore. In posizione centrale, una griglia ben organizzata mostra i poster dei prodotti inclusi nella lista. Consisten-

temente con altre sezioni dell'applicazione, un singolo click su un poster indirizza l'utente alla schermata dettagliata del prodotto selezionato. Un'ulteriore funzionalità disponibile è la rimozione di un prodotto dalla lista: tale azione può essere innescata mediante una pressione prolungata sul poster. Tuttavia, per prevenire cancellazioni accidentali, l'applicazione richiede una conferma esplicita prima di procedere con la rimozione.

Inoltre, la schermata Profilo offre anche la possibilità di ampliare la personalizzazione, consentendo agli utenti di creare nuove liste ad hoc. Scorrendo alla fine dell'elenco orizzontale dedicato alle liste, gli utenti troveranno un pulsante chiaramente etichettato con "Crea una nuova lista". Facendo click su questo pulsante, si apre una nuova schermata, illustrata nella Figura 5.16, pensata per la creazione di liste personalizzate. La schermata presenta tre elementi chiave, ovvero due campi di testo e un selettore. Il primo campo di testo è destinato all'inserimento di un identificatore univoco per la lista, mentre il secondo permette di assegnare un nome visibile e distintivo. Il selettore, poi, serve a definire la tipologia di prodotti da inserire nella lista; questa può essere dedicata esclusivamente ai film, alle serie TV o a una combinazione di entrambi. Concluso il processo di inserimento delle informazioni, un pulsante finalizza la creazione della lista secondo le specifiche fornite dall'utente.

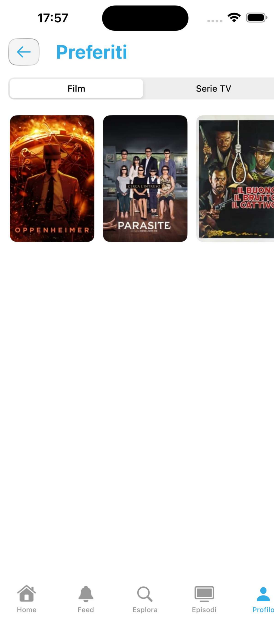

Figura 5.15: Esempio di schermata di dettaglio di una lista

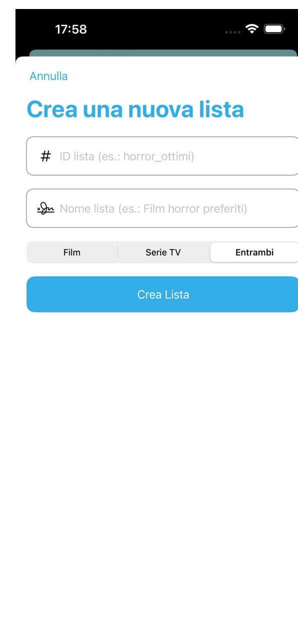

Figura 5.16: Schermata di creazione lista personalizzata

5.4 Dettagli

Le schermate di dettaglio di WatchWise sono state progettate per offrire agli utenti una panoramica approfondita e dettagliata dei contenuti. Ogni elemento, che si tratti di un film, una serie TV o una persona, ha una pagina dedicata che fornisce informazioni dettagliate, valutazioni e altro. Queste pagine sono state progettate per essere visivamente accattivanti e informative, contenendo, anche, alcune informazioni difficilmente reperibili tramite una veloce ricerca su Internet e garantendo agli utenti una comprensione chiara e completa del contenuto.

5.4.1 Dettaglio dei film

La schermata di dettaglio dei film (Figura 5.17) è stata progettata per fornire un'analisi esauriente di un determinato film. Al suo interno, gli utenti hanno la possibilità di accedere a diverse categorie di informazioni. Innanzitutto, è messo in risalto il poster del film, accompagnato dalla sua immagine di sfondo principale, la quale contribuisce a definire l'atmosfera e il tono del film in questione. Seguono, poi, dati fondamentali, quali il titolo, la durata, i generi associati e la trama, che forniscono una comprensione immediata del contenuto e del contesto del film. Ulteriore informazione di rilievo è la presenza di un elenco di eventuali piattaforme di streaming attraverso le quali il film può essere visto, agevolando l'utente nella fruizione del prodotto cinematografico.

La schermata si arricchisce ulteriormente grazie alla sezione dedicata alle valutazioni e recensioni degli utenti. Questa componente è fondamentale in quanto permette di comprendere l'accoglienza e la percezione della comunità riguardo al film, fornendo potenziali spunti critici e interpretativi.

Nella parte terminale della schermata (Figura 5.18), l'attenzione si sposta su dettagli di natura più tecnica e specifica. Qui, l'utente può approfondire le conoscenze sul cast artistico e tecnico del film, avendo, così, una visione chiara dei principali attori e delle figure professionali coinvolte nella realizzazione. In aggiunta, sono presenti dettagli complementari, come le case di produzione responsabili, dati economici come budget e incassi e una serie di contenuti multimediali, quali teaser e trailer, utili per avere un'anteprima dinamica del film. Concludendo, la sezione dedicata ai film correlati consente di suggerire all'utente ulteriori visioni potenzialmente in linea con i suoi interessi.

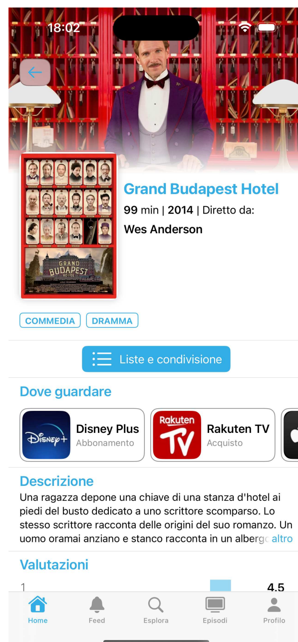

Figura 5.17: Esempio di schermata di dettaglio di un film (I)

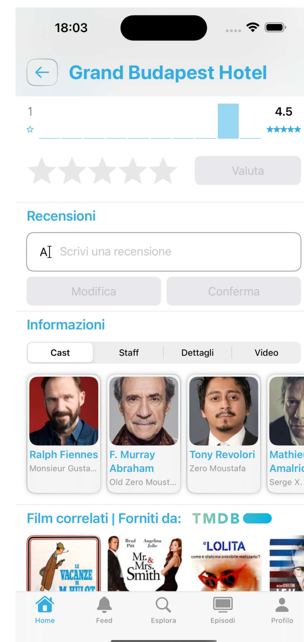

Figura 5.18: Esempio di schermata di dettaglio di un film (II)

5.4.2 Dettaglio delle serie TV

La schermata di dettaglio delle serie TV è progettata per fornire un quadro dettagliato e strutturato di una particolare serie televisiva. Come per la schermata dei film, si inizia presentando al centro dell'attenzione il poster della serie, affiancato dalla sua immagine di sfondo caratteristica, che definisce l'ambiente e la tematica della serie. Seguono informazioni

essenziali, quali il titolo, i generi associati e la trama generale, che delineano il contesto narrativo e tematico della serie.

Oltre a questi elementi basilari e agli altri elementi specifici analizzati nella Sezione 5.4.1, la schermata si distingue per una sezione dedicata specificamente alla navigazione e alla visualizzazione degli episodi che compongono la serie (Figura 5.19). In questa sezione, sono organizzate le diverse stagioni della serie TV. Per ciascuna stagione, sono visibili il poster rappresentativo, il titolo specifico della stagione e un conteggio che indica il numero di episodi già visti dall'utente in relazione al numero totale di episodi della stagione. Accanto a queste informazioni, è presente un pulsante che consente all'utente di segnare l'intera stagione come vista, facilitando il tracciamento del progresso di visione.

Selezionando una particolare stagione, l'utente accede a una schermata più dettagliata dedicata esclusivamente a quella stagione (Figura 5.20). Qui vengono forniti ulteriori dettagli, come la trama specifica della stagione e la data di rilascio. Inoltre, questa schermata presenta un elenco strutturato degli episodi che la compongono. Ciascun episodio, analogamente a quanto visto nella schermata Episodi (Sezione 5.3.4), è accompagnato da un pulsante di tracciamento, consentendo all'utente di tenere sotto controllo il suo avanzamento nella visione della serie.

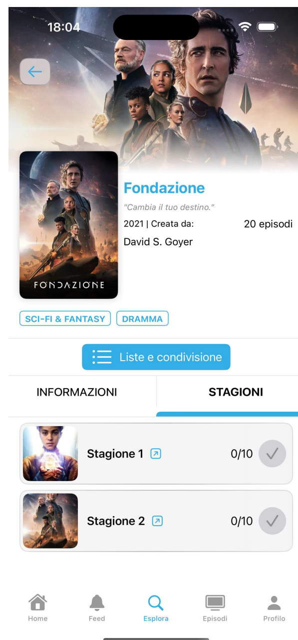

Figura 5.19: Esempio di schermata di dettaglio di una serie TV

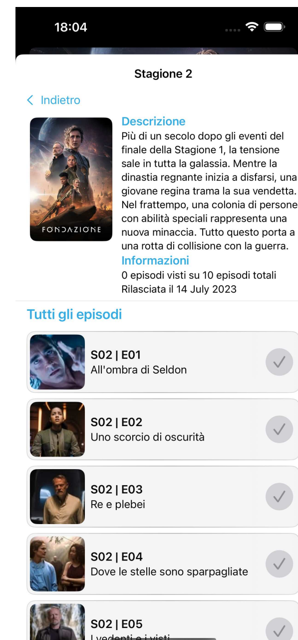

Figura 5.20: Esempio di schermata di dettaglio di una stagione di una serie TV

5.4.3 Dettaglio delle persone

La schermata di dettaglio delle persone (Figure 5.21 e 5.22) si configura come uno strumento essenziale per ottenere informazioni comprensive e articolate su figure chiave nel mondo del cinema e della televisione, quali attori, registi o membri della troupe. Questa componente dell'applicazione è stata progettata per fornire un profilo dettagliato che amalgama aspetti biografici e professionali delle personalità in questione.

All'apertura della schermata, l'utente viene accolto dalla visualizzazione di una fotografia ritraente la persona di interesse, accompagnata da una breve biografia che delinea gli aspetti salienti della sua carriera e vita personale. Questa sezione serve come introduzione essenziale, mettendo a disposizione dell'utente informazioni di base quali la data e il luogo di nascita e altro, qualora queste informazioni siano disponibili e pertinenti.

Un elemento distintivo di questa schermata è la presenza di una lista esaustiva che cataloga i prodotti cinematografici e televisivi a cui la persona ha contribuito nel corso della sua carriera (Figura 5.22). Questa lista è organizzata in maniera tale da facilitare una navigazione intuitiva e una ricerca efficiente. In particolare, è implementata una funzionalità di filtraggio che permette agli utenti di distinguere e selezionare le opere in cui la persona è stata coinvolta in veste di membro del cast o quelle in cui ha fatto parte dello staff, offrendo, così, una panoramica più articolata e strutturata del suo percorso professionale. All'interno di questa lista, ogni voce è strutturata per fornire un'informazione completa e intuitiva; per ogni prodotto sono, infatti, presenti il poster rappresentativo, il titolo dell'opera e, in aggiunta, viene specificato il ruolo interpretato dalla persona, se membro del cast, o la funzione svolta all'interno della produzione, qualora si tratti di un membro dello staff

Inoltre, per garantire un'esperienza utente ancora più immersiva e informativa, ogni elemento dell'elenco è interattivo. Con un semplice click su uno dei prodotti elencati, gli utenti vengono indirizzati alla pagina di dettaglio del prodotto selezionato, dove possono esplorare informazioni ancora più dettagliate relative a quel particolare film o serie TV.

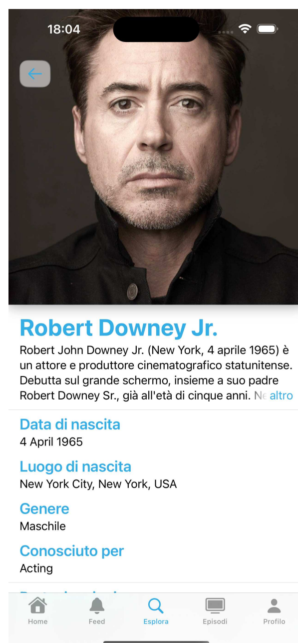

Figura 5.21: Esempio di schermata di dettaglio di una persona (I)

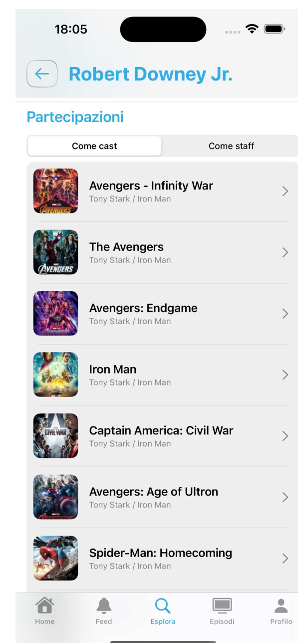

Figura 5.22: Esempio di schermata di dettaglio di una persona (II)

5.5 Funzionalità social

Come è stato esplicato più volte, WatchWise non è solo un'applicazione per la scoperta di contenuti cinematografici e televisivi, ma anche una piattaforma social dove gli utenti possono interagire, condividere e discutere dei loro interessi. Le funzionalità social sono centrali nell'applicazione, permettendo agli utenti di connettersi con altri appassionati e di arricchire la loro esperienza attraverso la condivisione e l'interazione.

5.5.1 Dettaglio degli utenti

Ogni utente di WatchWise ha una pagina di profilo dettagliata, la quale è strutturalmente identica alla schermata Profilo analizzata nella Sezione 5.3.5. Le uniche differenze consistono nel fatto che i pulsanti "Modifica profilo" e "Impostazioni app" sono sostituiti da un pulsante

che permette di seguire o smettere di seguire l'utente in questione e che il pulsante per la creazione di una lista personalizzata non è presente.

5.5.2 Valutazioni e recensioni

WatchWise pone un'enfasi particolare sull'importanza delle opinioni degli utenti, fornendo loro gli strumenti necessari per esprimere e condividere valutazioni e recensioni. Nella specifica schermata di dettaglio relativa a un film o una serie TV, viene data la possibilità all'utente di assegnare un giudizio che può variare da un minimo di 0.5 a un massimo di 5 stelle, e di condividere una recensione articolata. Questi feedback sono accessibili all'intera community, instaurando così un ambiente in cui le opinioni possono essere liberamente manifestate, confrontate e discusse.

All'interno della schermata di dettaglio di ogni prodotto, le valutazioni globali sono elegantemente rappresentate attraverso un istogramma, come illustrato in Figura 5.23. Questo diagramma grafico mette in evidenza la distribuzione delle diverse valutazioni, mostrando, in aggiunta, sia il numero totale di giudizi che la media aritmetica degli stessi. Nota degna di menzione è la presenza di un analogo istogramma all'interno delle schermate di dettaglio degli utenti, il quale sintetizza tutte le valutazioni assegnate da un singolo utente.

Relativamente alle recensioni, queste, se riferite a un determinato prodotto, o se associate a un utente specifico, possono essere consultate in maniera dettagliata mediante la pressione di un pulsante dedicato. La successiva visualizzazione avverrà nella schermata delle recensioni, rappresentata in Figura 5.24.

Figura 5.23: Istogramma delle valutazioni

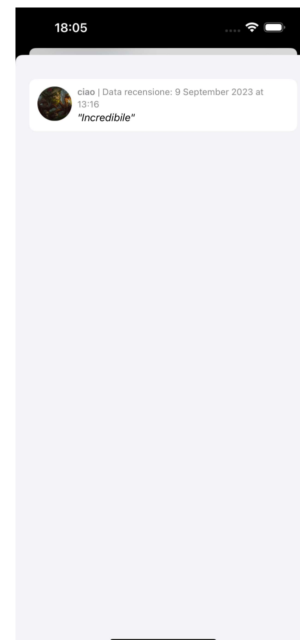

Figura 5.24: Schermata di elenco delle recensioni

Condivisione

Il meccanismo di condivisione offerto da WatchWise rappresenta uno strumento essenziale per potenziare l'interazione tra gli utenti e favorire la divulgazione di prodotti cinematografici o televisivi di particolare interesse. Esso consente ai membri della piattaforma di suggerire un determinato prodotto ad un altro utente che seguono. Quando viene effettuato un tale consiglio, si instaura un dialogo tra i due utenti oppure, se già presente,

il prodotto suggerito viene integrato all'interno della lista dei consigli nella corrispondente chat, come descritto nella Sezione 5.3.2.

Il processo di condivisione inizia con l'accesso al menù denominato "Liste e condivisione", rappresentato nella Figura 5.25. Questo menù viene attivato cliccando sull'apposito pulsante che si trova nelle schermate di dettaglio sia dei film che delle serie TV. Una volta all'interno di "Liste e condivisione", gli utenti possono facilmente accedere al sotto-menù "Condivisione", il cui contenuto è mostrato in Figura 5.26. Questa sezione specifica fornisce una panoramica degli utenti seguiti, permettendo la selezione e l'invio diretto di suggerimenti. Una volta inviato il consiglio a un particolare utente, il destinatario riceverà una notifica all'interno della propria schermata Feed, indicante il nuovo suggerimento. Questa notifica servirà come collegamento diretto, permettendo all'utente stesso di accedere rapidamente alla schermata di dettaglio del prodotto suggerito attraverso la chat di interazione.

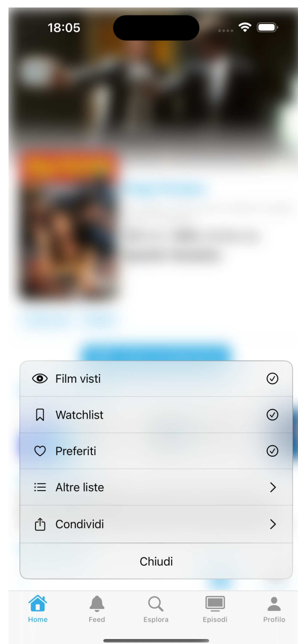

Figura 5.25: Menu "Liste e condivisione"

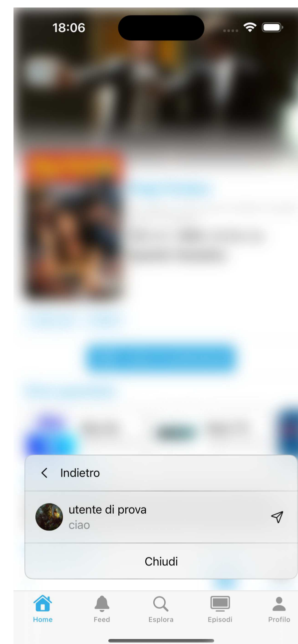

Figura 5.26: Menu di condivisione

5.6 Conclusioni

Il capitolo in esame ha delineato in maniera dettagliata e sistematica le caratteristiche salienti e le funzionalità di WatchWise. Questo capitolo è stato redatto con l'intento di fornire una guida esaustiva che facilitasse la comprensione e l'uso dell'applicazione.

Le diverse sezioni hanno approfondito le componenti chiave dell'esperienza dell'utente, dalla fase di autenticazione, passando per la navigazione delle schermate principali, fino ad arrivare alle funzioni di interazione sociale. Si è posta particolare enfasi sull'architettura intuitiva delle interfacce, progettate per assicurare una navigazione coesa e ottimizzata, oltre che esteticamente appagante.

L'analisi approfondita di funzionalità e interfacce sottolinea l'approccio di WatchWise focalizzato sulla user experience. La piattaforma, oltre a garantire contenuti di elevato calibro, pone l'accento sulla facilità d'accesso e sull'interazione semplificata con l'utente. Con la conclusione di questo capitolo, il progetto di tesi si avvia verso le fasi finali, in cui verranno affrontate le discussioni conclusive e le riflessioni finali sull'intero lavoro.

CAPITOLO 6

Discussione

Questo capitolo è dedicato a una riflessione finale sul progetto dell'applicazione WatchWise. La discussione sarà strutturata in due parti principali: la prima sarà un'analisi SWOT attraverso la quale saranno evidenziati i punti di forza, le debolezze, le opportunità e le minacce legate all'applicazione sviluppata; la seconda parte, invece, proporrà un'analisi comparativa con altre applicazioni dello stesso settore. Quest'ultima analisi sarà fondamentale per identificare potenziali aree di miglioramento e proporre spunti per evoluzioni future di WatchWise.

6.1 SWOT Analysis

L'analisi SWOT (Strengths, Weaknesses, Opportunities, Threats) rappresenta un metodo sistematico e strutturato, impiegato ampiamente nei processi decisionali, in particolare per l'identificazione e la valutazione dei fattori interni ed esterni che possono influenzare la realizzazione e il successo di un progetto. Nel contesto dell'ingegneria del software e, implicitamente, dello sviluppo di applicazioni mobili, questo strumento assume una rilevanza cruciale, permettendo di tracciare una mappa chiara delle risorse e delle sfide presenti, ottimizzando, quindi, la gestione del progetto e anticipando potenziali problematiche.

6.1.1 Punti di Forza

I *punti di forza* si riferiscono a quegli elementi interni che conferiscono un vantaggio competitivo al progetto in esame. Questi possono includere tecnologie avanzate, competenze specifiche del team di sviluppo, metodologie di lavoro efficienti, capacità di integrazione con altri sistemi, o una UX/UI (User Experience/User Interface) particolarmente intuitiva e accattivante.

Nel contesto di WatchWise, i punti di forza identificati sono:

- *Interfaccia utente*: nel processo di sviluppo dell'applicazione è stata attribuita significativa importanza all'ingegnerizzazione dell'interfaccia utente. L'obiettivo principale era realizzare un'interfaccia in linea con le migliori pratiche estetiche e funzionali. Sebbene la percezione estetica possa variare da individuo a individuo, esistono criteri standardizzati che guidano la progettazione di interfacce di qualità. Adottando le linee guida fornite da Apple, si è conseguita una coerenza stilistica e funzionale riconosciuta a livello industriale. Studi hanno dimostrato che gli utenti tendono a favorire applica-

zioni con una buona estetica, anche se ciò potrebbe significare un compromesso sulle funzionalità o sulle prestazioni[1].

- *Esperienza utente*: derivante dalle metodologie adottate nella progettazione dell'interfaccia, l'esperienza utente (UX) ha beneficiato di una qualità elevata. L'adesione alle linee guida proposte da Apple non ha solo implicazioni estetiche ma incide direttamente sulla UX. Pertanto, è stata condotta un'analisi dettagliata su aspetti come il posizionamento, le dimensioni, la palette cromatica e altre specifiche caratteristiche degli elementi dell'interfaccia, al fine di assicurare un'esperienza utente coesa, reattiva e intuitiva.

- *Ottimizzazione della memoria*: grazie all'utilizzo della libreria Kingfisher (Sezione 1.8.5) e all'interfacciamento con le API di TMDB, si è conseguito un significativo risparmio di memoria sul dispositivo. I contenuti grafici associati alle informazioni visualizzate (come poster, sfondi e immagini delle persone) non sono staticamente memorizzati, ma piuttosto richiamati in tempo reale e temporaneamente conservati nella cache del dispositivo, garantendo, così, un efficiente utilizzo delle risorse di memoria.

- *Sicurezza*: la gestione dell'autenticazione utente, sia essa attraverso credenziali email e password, sia mediante l'utilizzo di Sign In With Apple o Google Sign In, è affidata a Firebase Authentication. Allo stesso modo, la persistenza dei dati associati agli utenti e alle loro interazioni all'interno dell'applicazione avviene tramite infrastrutture Firebase. Optare per Firebase assicura un'architettura di sicurezza robusta e affermata, grazie alla sua cifratura end-to-end, alla conformità con gli standard industriali, e alle regolari revisioni di sicurezza svolte da esperti del settore. Questa scelta garantisce la protezione dei dati degli utenti e la resilienza dell'ecosistema dell'applicazione contro potenziali vulnerabilità.

6.1.2 Punti di Debolezza

I *punti di debolezza* delineano le carenze interne che potrebbero rappresentare impedimenti nell'attuazione degli obiettivi prefissati del progetto. Tali carenze potrebbero manifestarsi come insufficienze di competenze in settori critici, vincoli finanziari, una documentazione del software non adeguata o incongruenze nella gestione complessiva del ciclo di vita del prodotto.

Nel contesto di WatchWise, i punti di debolezza identificati sono:

- *Dipendenza dalla connettività internet*: una conseguenza intrinseca dell'efficace strategia di ottimizzazione della memoria è la necessità indispensabile di una connessione internet stabile e affidabile per recuperare immagini e informazioni pertinenti. Sebbene le innovazioni progressive nelle infrastrutture digitali stiano riducendo gradualmente la frequenza di tali inconvenienti, la potenziale insufficienza di connettività rimane una considerazione saliente, qualificandosi così come un possibile punto di debolezza.

- *Limitazioni economiche*: la pubblicazione dell'applicazione nell'App Store e l'introduzione di determinate caratteristiche avanzate, quali il Sign In With Apple o la gestione di notifiche push remote, implica l'obbligatorietà di aderire all'Apple Developer Program, il quale comporta un onere annuale di 99 €. Questo impegno economico, in certe circostanze, può configurarsi come una restrizione di bilancio. Di conseguenza, si potrebbe rendere necessaria l'integrazione di acquisti in-app o di contenuti pubblicitari, soluzioni che potrebbero suscitare perplessità in una frazione degli utenti.

[1] `http://bit.ly/46l3yST`

- *Limitazioni di piattaforma*: la decisione di impiegare Swift ha orientato lo sviluppo verso un'applicazione nativa specificamente per iOS. Benché ciò garantisca vantaggi innegabili in termini di performance e l'adozione di componenti grafici nativi, comporta anche l'intrinseca limitazione di non poter servire l'utenza Android. Questo impone un ulteriore sviluppo ex novo dell'applicazione per tale piattaforma, qualora si intenda espandere la base di utenti.

- *Capacità di filtraggio limitate*: sebbene le funzioni di ricerca dell'applicazione siano solidamente implementate dal punto di vista fondamentale, presentano alcune carenze in termini di funzionalità avanzate. Notabilmente, manca la capacità di raffinare i risultati di ricerca attraverso specifici parametri, quali anno di uscita, genere cinematografico e simili.

6.1.3 Opportunità

Le *opportunità* si distinguono nettamente dai punti di forza e di debolezza, essendo influenze esterne che, se opportunamente capitalizzate, possono propulsare significativamente il progetto verso il successo. Tali opportunità possono manifestarsi in vari modi, come un'evoluzione favorevole delle tendenze di mercato, modifiche legislative che favoriscono lo sviluppo del progetto, o l'adozione di nuove tecnologie compatibili che possono essere integrate per ampliare e migliorare le funzionalità dell'applicazione.

Nel contesto di WatchWise, le opportunità individuate sono:

- *JustWatch*: la piattaforma JustWatch (Sezione 6.2.2) offre la possibilità di instaurare una collaborazione la quale potrebbe rappresentare un'opportunità significativa per WatchWise. JustWatch, previa autorizzazione attraverso una comunicazione formale, mette a disposizione le sue API, permettendo all'applicazione di fornire un servizio di verifica della disponibilità dei contenuti su diverse piattaforme di streaming in modo più accurato e approfondito. Sebbene WatchWise si avvalga già delle API di TMDB per uno scopo simile, l'integrazione con JustWatch potrebbe portare a una maggiore precisione nell'indicazione della disponibilità e offrire una funzionalità di reindirizzamento diretto verso l'applicazione di streaming desiderata.

- *Mercato cinematografico e televisivo*: il panorama cinematografico e, in particolare, quello televisivo ha registrato una crescita significativa negli anni recenti, diventando un punto focale dell'intrattenimento per molti. Con l'emergere di un numero sempre maggiore di appassionati di cinema e serie TV, la necessità di catalogare e organizzare i contenuti visionati diventa sempre più rilevante. Questo trend mette in evidenza un segmento di pubblico crescente che cerca strumenti efficaci per gestire le proprie abitudini di visione, facendo di loro dei candidati ideali per diventare utenti di WatchWise.

- *Mercato social*: il panorama dei social media ha raggiunto una posizione di primaria importanza nel tessuto quotidiano di innumerevoli individui. Le funzionalità social di WatchWise, che fondono la passione per film e serie TV con l'opportunità di condividere e celebrare questi interessi con una community globale, rappresentano indiscutibilmente una notevole occasione di crescita e consolidamento per l'applicazione.

- *Localizzazione*: al momento, WatchWise è disponibile in lingua italiana e in una versione parziale in inglese. Date la strutturata implementazione modulare e l'adozione di stringhe pronte per la localizzazione, c'è un'aperta opportunità per espandere il suo impatto globale rendendo l'app accessibile in diverse lingue, mirando ad una più ampia base di utenti in diversi paesi di tutto il mondo.

6.1.4 Minacce

Le *minacce*, al pari delle opportunità, rappresentano aspetti esterni; tuttavia, in questo scenario, agiscono come potenziali impedimenti al raggiungimento degli obiettivi del progetto. Questi fattori potrebbero manifestarsi come una concorrenza sempre più aggressiva, modifiche legislative restrittive, variazioni nel panorama economico, o il progressivo invecchiamento della tecnologia impiegata.

Nel contesto di WatchWise, le minacce individuate sono le seguenti:

- *Concorrenza*: uno degli ostacoli più significativi proviene dalla concorrenza nel settore. La categoria di applicazioni a cui appartiene WatchWise non è nuova e presenta già una serie di soluzioni consolidate e popolari che potrebbero superare WatchWise in termini di funzionalità ed efficienza. Queste alternative consolidate potrebbero attrarre ad esse utenti che altrimenti potrebbero essere interessati a WatchWise. Per mitigare l'impatto di questa minaccia, sarebbe strategico sviluppare caratteristiche distintive e innovative, come la già implementata funzionalità di consivisione dei consigli, e fornire strumenti per facilitare la migrazione dei dati degli utenti dalle piattaforme concorrenti a WatchWise, una funzionalità attualmente non disponibile.

- *Interesse geografico limitato*: uno degli ostacoli potenziali legati all'espansione geografica dell'applicazione riguarda le specifiche inclinazioni culturali di alcune regioni. Potrebbe accadere che, in certe aree geografiche, l'entusiasmo e l'interesse per i film e le serie TV non siano abbastanza radicati da sostenere un uso costante e prolungato di un'app come WatchWise. Tale scenario implicherebbe che, nonostante gli sforzi e i costi per la localizzazione dell'app, non ci sia un'utenza significativa pronta ad adottarla e utilizzarla regolarmente.

- *Variazioni normative*: l'evoluzione continua delle leggi sulla protezione dei dati personali e sulla privacy, in particolare in regioni come l'Europa con il GDPR, potrebbe imporre nuovi oneri e obblighi a WatchWise. Adattarsi a queste regolamentazioni potrebbe richiedere investimenti significativi, sia in termini di tempo che di risorse, e potrebbe anche influenzare alcune funzionalità chiave dell'app, limitando potenzialmente la sua attrattiva per gli utenti.

- *Dipendenza da servizi esterni*: la forte integrazione con servizi terzi come TMDB rappresenta un rischio in quanto WatchWise è vulnerabile a eventuali cambiamenti nelle politiche, nei prezzi o nelle disponibilità di queste piattaforme. Se uno di questi servizi decidesse di cambiare i propri termini o di interrompere le proprie API, ciò potrebbe avere un impatto diretto e immediato sulle prestazioni e sulla funzionalità di WatchWise, richiedendo soluzioni di contorno o alternative.

6.2 Confronto con sistemi correlati

La minaccia più tangibile ed evidente discussa nella Sezione 6.1.4 è, indubbiamente, quella posta dalla concorrenza. In questa sezione, pertanto, si propone un'analisi dettagliata di cinque piattaforme che operano in un ambito simile a quello di WatchWise: TVTime, Letterboxd, IMDb, JustWatch e Hobi. L'obiettivo è quello di esaminare ogni singolo sistema, mettendo in luce le sue caratteristiche distintive e confrontandole con quelle dell'applicazione in esame. Questo esercizio si rivela fondamentale per comprendere quali potrebbero essere le future integrazioni o le innovazioni da considerare per WatchWise, allo scopo di consolidare la sua posizione nel mercato e di minimizzare l'effetto delle dinamiche competitive.

Le applicazioni TVTime e Letterboxd sono già state analizzate dettagliatamente, rispettivamente, nelle Sezioni 2.3.1 e 2.3.3. Per tale motivo, non verranno trattate in questa sezione.

6.2.1 Confronto con IMDb

IMDb, acronimo di Internet Movie Database, è un sito web che cataloga e archivia film, attori, registi, personale di produzione e programmi televisivi, oltre ai videogiochi. L'app iOS corrispondente offre un'interfaccia per fruire dei servizi offerti dal noto sito. In un'analisi comparativa con WatchWise, alcune delle caratteristiche di IMDb che emergono come particolarmente rilevanti sono le seguenti:

- L'app IMDb presenta un vasto insieme di informazioni sui contenuti. La sua finalità primaria pare essere quella di offrire informazioni estremamente dettagliate su vari argomenti. Le schermate di dettaglio dei prodotti e dei professionisti contengono una mole considerevole di dati, rendendo tali sezioni estensive ma ricche di contenuto. Tale struttura potrebbe essere apprezzata da alcuni utenti, mentre per altri potrebbe apparire eccessivamente complessa o ridondante, specialmente per come è organizzata a livello di esperienza utente.
- La schermata Home di IMDb propone elementi che potrebbero catturare l'interesse di molti utenti, come le classifiche dei film con i maggiori incassi, le celebrità che festeggiano il compleanno in quel giorno o le ultime notizie dal mondo cinematografico (Figura 6.1).
- A differenza di WatchWise, IMDb permette di consultare contenuti e informazioni anche senza la necessità di creare un account, limitando soltanto l'accesso alle funzionalità social.
- IMDb offre uno strumento di ricerca avanzata, arricchito da vari filtri.
- Il profilo utente di IMDb, pur contenendo le liste dell'utente, le sue valutazioni e recensioni, non offre opzioni per personalizzare l'aspetto estetico della pagina.
- L'app di IMDb non integra una funzionalità per il tracciamento degli episodi.
- Un elemento distintivo di IMDb è la possibilità di monitorare e visionare i cinema di diverse località nel mondo, permettendo anche di selezionare cinema preferiti e di consultare la programmazione attuale di un cinema specifico. Questa feature, decisamente originale, potrebbe aumentare significativamente il valore percepito dell'app.

In conclusione, IMDb offre un'ampia gamma di funzionalità che la distinguono da WatchWise. Alcune delle sue funzioni potrebbero fornire spunti utili per future implementazioni o aggiustamenti di WatchWise, mentre altre evidenziano chiaramente i diversi focus dei due strumenti. Mentre IMDb si presenta come un database vasto e dettagliato, WatchWise ha una propensione più marcatamente sociale.

6.2.2 Confronto con JustWatch

JustWatch è un'applicazione focalizzata principalmente sulla fornitura di informazioni relative alle modalità di visione dei prodotti. Analizzando le funzionalità di JustWatch in confronto a WatchWise, emergono le seguenti caratteristiche salienti:

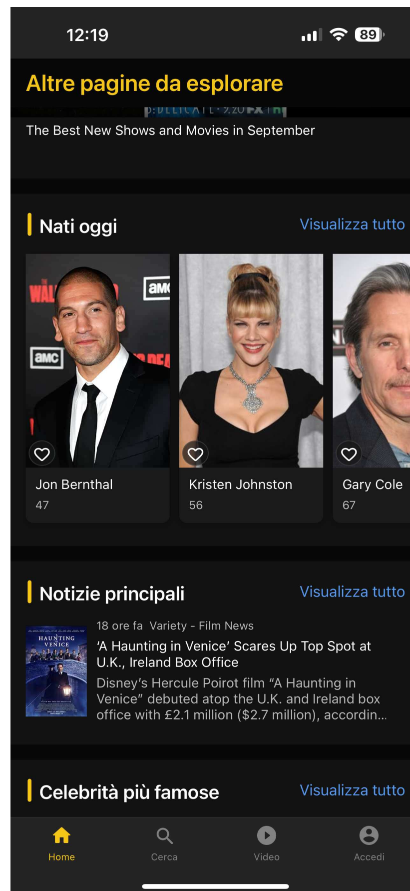

Figura 6.1: Informazioni particolari reperibili nella schermata Home di IMDb

- JustWatch ricava le informazioni mostrate dalle API di IMDb, analogamente a come WatchWise sfrutta le API di TMDB.
- Come già sottolineato, l'essenza di JustWatch risiede nel mostrare le diverse modalità di visione dei contenuti. Per ogni prodotto, presenta un elenco dettagliato delle piattaforme disponibili, specificando la modalità di servizio (abbonamento, noleggio o acquisto). A differenza di WatchWise, fornisce anche dettagli quali il prezzo, la qualità dello streaming e presenta una griglia che ordina i servizi di streaming secondo determinati criteri (Figura 6.2).
- A differenza di WatchWise, JustWatch dedica meno attenzione ai dettagli riguardanti i prodotti, rendendo la sua offerta più orientata a scopi pratici.
- Anche le informazioni relative alle persone (come attori e registi) sono essenziali, concentrandosi principalmente sui prodotti in cui sono stati coinvolti.
- Le funzionalità social offerte da JustWatch sono piuttosto limitate e si basano principalmente sulla valutazione dei prodotti mediante un sistema a stelle e su poche altre funzioni.
- Allo stesso modo di IMDb, JustWatch offre l'accesso a molte delle sue informazioni anche senza la necessità di creare un account, escludendo soltanto le funzionalità social.
- Un aspetto distintivo di JustWatch è la sua capacità di fornire informazioni su eventi sportivi e eSports, sia passati che in diretta.

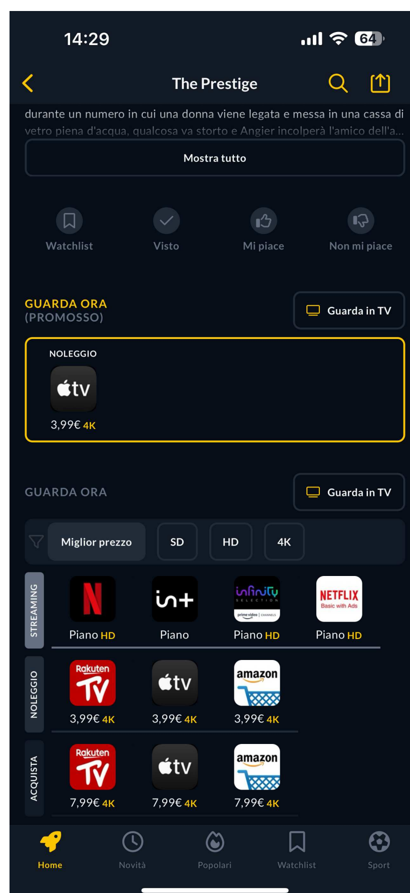

Figura 6.2: Informazioni sui servizi di visione offerte da JustWatch

In conclusione, JustWatch e WatchWise, pur operando nel medesimo ambito, presentano filosofie e funzionalità nettamente diverse. Mentre JustWatch si concentra fortemente sulla fornitura di dettagli sulle modalità di visione e su eventi sportivi, WatchWise offre un'esperienza più ricca dal punto di vista informativo e sociale. Questo confronto sottolinea l'importanza di riconoscere e capitalizzare i propri punti di forza nel panorama delle applicazioni legate al mondo cinematografico e televisivo. Una collaborazione con JustWatch, per usufruire delle sue API, potrebbe, dunque, valorizzare maggiormente le informazioni offerte da WatchWise.

6.2.3 Confronto con Hobi

Hobi è un'applicazione dedicata esclusivamente al tracciamento delle serie televisive. Se confrontata con WatchWise e con le altre applicazioni analizzate in questo capitolo, Hobi presenta un insieme di funzionalità meno ampio. Tuttavia, mettendo a confronto Hobi e WatchWise, si possono sintetizzare le caratteristiche distintive di Hobi nei seguenti punti:

- Hobi si concentra esclusivamente sul tracciamento delle serie TV, lasciando fuori i film dal suo raggio d'azione. Di conseguenza, Hobi non fornisce informazioni sui film, ma si dedica soltanto alle serie TV.
- La sezione di esplorazione di Hobi è strutturata in numerose categorie che facilitano la scoperta di nuovi contenuti, permettendo, così, agli utenti di evitare ricerche e di imbattersi in nuovi spettacoli solo attraverso le correlazioni.

- Tuttavia, le informazioni fornite sulle singole serie TV sono essenziali, con un'attenzione particolare al tracciamento preciso degli episodi visti dall'utente.
- Nonostante la pagina del profilo non sia personalizzabile, Hobi presenta una vasta gamma di statistiche che potrebbero affascinare gli utenti più appassionati.
- Le funzionalità social offerte da Hobi sono limitate, restringendosi principalmente a valutazioni e recensioni.

In conclusione, Hobi e WatchWise, seppur entrambi nell'ambito delle applicazioni di tracciamento, si rivolgono a target di utenti differenti. Mentre WatchWise fornisce un'esperienza più completa e multidimensionale, Hobi si dedica con precisione al mondo delle serie TV, offrendo funzionalità specifiche e soddisfacendo gli utenti desiderosi di un'applicazione focalizzata esclusivamente su questo segmento.

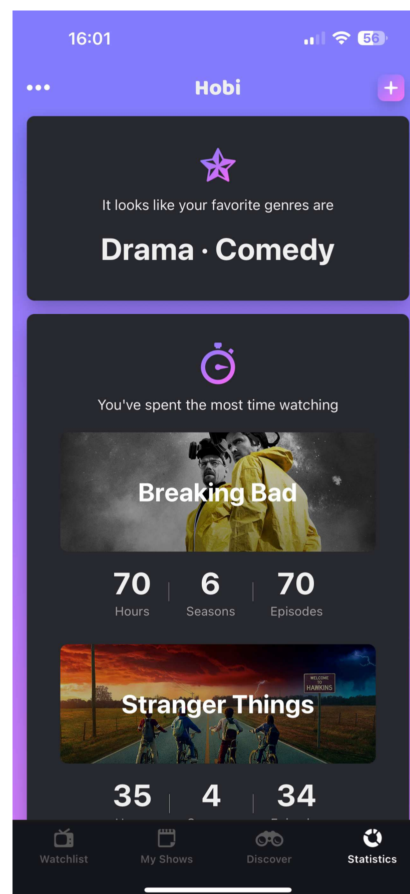

Figura 6.3: Alcune delle statistiche offerte da Hobi

Conclusioni

Nell'odierno e dinamico panorama dell'ingegneria del software, il settore delle applicazioni per dispositivi mobili rappresenta una delle aree più stimolanti e in costante crescita. Particolarmente interessante è lo sviluppo di applicazioni per la piattaforma iOS, che, pur rappresentando una fetta di mercato più ristretta rispetto ad Android, offre enormi opportunità di innovazione e di monetizzazione. In tale contesto, il progetto di ricerca su cui si è basata questa tesi ha avuto come obiettivo principale la creazione di un social network dedicato agli appassionati del mondo cinematografico e delle serie televisive.

La fase preliminare del progetto è stata incentrata su un'analisi approfondita e dettagliata delle esigenze del mercato, delle potenzialità degli utenti e delle scelte tecnologiche più adeguate. Questa analisi ha incluso un'osservazione delle tendenze attuali in materia di applicazioni social e di intrattenimento. Tutto ciò ha permesso di prendere decisioni tecnologiche e stilistiche basate su dati concreti, mirando a realizzare un'app che fosse non solo altamente funzionale ma anche intuitiva, user-friendly e visivamente accattivante.

La scelta del linguaggio di programmazione Swift è stata dettata da una serie di considerazioni, tra cui l'efficienza, la flessibilità e le potenzialità di ottimizzazione del codice, oltre che al fatto che Swift è il linguaggio di programmazione ufficiale per lo sviluppo iOS consigliato da Apple stessa. La sintassi chiara e moderna di Swift, insieme alla sua capacità di interfacciarsi senza problemi con gli strumenti nativi di iOS, ha reso possibile l'ingegnerizzazione di un'app che si amalgama perfettamente con l'ecosistema Apple. Questo, a sua volta, ha garantito un'esperienza utente fluida, reattiva e di alta qualità. Inoltre, l'uso dell'IDE Xcode ha fornito un framework di sviluppo solido e omogeneo che ha assistito lo sviluppo in ogni fase del ciclo di vita del software, dalla concezione iniziale, alla codifica, ai test e, in un ipotetico futuro, al rilascio sul mercato.

L'analisi dettagliata dei requisiti ha giocato un ruolo cruciale nel delineare gli obiettivi e le funzionalità dell'applicazione. Non solo si è cercato di individuare le caratteristiche più desiderate dal pubblico target, ma si è anche dedicata attenzione ai requisiti non funzionali, quali la sicurezza dei dati, la protezione della privacy e l'efficienza nell'esecuzione delle operazioni. Questo ha assicurato che l'app fosse in grado di soddisfare una gamma completa di esigenze, esplicite e latenti, rendendo la gestione dei dati robusta e garantendo un'interazione fluida e piacevole per l'utente.

Per quanto riguarda il design dell'interfaccia utente, si è fatto ricorso a principi e linee guida coerenti con l'estetica e la filosofia di Apple. L'uso di elementi grafici ben progettati e schemi di flusso ha aiutato a creare un'interfaccia che facilita la navigazione e migliora la comprensione delle varie funzionalità offerte dall'app. Il comportamento degli utenti è stato monitorato attraverso l'analisi di feedback di alcuni "tester" per apportare miglioramenti

iterativi alle componenti dell'interfaccia, migliorando così la user experience nel suo insieme.

Nella fase di implementazione, il codice è stato scritto con una metodologia agile, che ha permesso di tradurre in maniera efficace le specifiche di design in un'app funzionante. L'integrazione di API esterne e di servizi forniti da terze parti ha arricchito l'applicazione, ampliando la gamma delle sue funzionalità e rendendo l'esperienza dell'utente ancora più coinvolgente. Un attento processo di gestione degli errori ha garantito l'affidabilità, la stabilità e l'efficienza complessive dell'app, rendendola pronta per la distribuzione, dopo altri accorgimenti funzionali.

L'analisi SWOT condotta in fase finale ha offerto una panoramica completa dell'app, evidenziando i punti di forza e le aree potenziali per ulteriori sviluppi. La comparazione con applicazioni concorrenti e con le tendenze prevalenti nel settore ha offerto ulteriori spunti per lo sviluppo di future funzionalità e per il miglioramento dell'esperienza utente.

Guardando al futuro, le solide basi su cui è stata costruita questa applicazione rendono evidente la sua scalabilità e la sua capacità di adattarsi alle esigenze in rapida evoluzione del mercato. Con la potenzialità di ampliare e perfezionare le funzionalità esistenti, e considerando la vasta gamma di possibili integrazioni e nuove caratteristiche, questa applicazione è ben posizionata per evolversi in risposta alle mutevoli esigenze dei suoi utenti.

In conclusione, questo progetto di ricerca ha portato alla creazione di un'applicazione mobile sofisticata, che coniuga avanzate funzionalità tecniche con un forte focus sull'usabilità e sul design. Le decisioni informate e ben ponderate prese in ogni fase del processo di sviluppo assicurano che questa applicazione abbia solide basi per futuri miglioramenti e adattamenti, con un impegno costante nel fornire un'esperienza utente di alto livello, che comprenda aspetti quali la sicurezza dei dati, la facilità di utilizzo e la capacità di rimanere al passo con un mercato in continua evoluzione.

Bibliografia

APPLE (2014), *The Swift Programming Language (Swift 5.7)*, Apple Inc.

BERLIN, J., RAYWENDERLICH TUTORIAL TEAM e CACHEAUX, R. (2020), *Advanced IOS App Architecture (Third Edition)*, Razeware LLC.

CALABRO, V. G. (2016), *Mobile device and mobile cloud computing forensics.*, Lulu.com.

DEVILLA, J., GANEM, E. e HOLLEMANS, M. (2019), *IOS Apprentice (Eighth Edition): Beginning IOS Development with Swift and UIKit*, Razeware LLC, 8th ed.

FRIESE, P. (2023), *Asynchronous Programming with SwiftUI and Combine*, Apress.

GARCÍA, R. F. (2023), *iOS Architecture Patterns: MVC, MVP, MVVM, VIPER, and VIP in Swift*, Apress.

IN 'T VEEN, T. (2018), *Swift in Depth*, Manning, 1st ed.

KEUR, C. e HILLEGASS, A. (2016), *IOS programming : the Big Nerd Ranch guide.*, Big Nerd Ranch, Atlanta, Ga.

KING, J. (2022), *The WWDC Event*, Independent, 1st ed.

RAYWENDERLICH TUTORIAL TEAM, BELLO, A., MOREFIELD, B., REICHELT, S. e TAM, A. (2021), *SwiftUI by Tutorials (Fourth Edition)*, Razeware LLC.

SADUN, E. e WARDWELL, R. (2014), *The Core iOS Developers Cookbook*, Addison-Wesley Professional.

SMITH, J. (2013), *Advanced Mvvm*, Josh Smith.

TEAM, K. e TODOROV, M. (2023), *Modern Concurrency in Swift (Second Edition)*, Kodeco Incorporated.

TEAM, K., MISHALI, S., PILLET, F. e TODOROV, M. (2023a), *Combine: Asynchronous Programming with Swift*, Kodeco Inc.

TEAM, K., BELLO, A., MOREFIELD, B., REICHELT, S. e TAM, A. (2023b), *SwiftUI by Tutorials (Fifth Edition)*, Razeware LLC, 5th ed.

Siti web consultati

- Apple Developer – `developer.apple.com`
- Firebase – `firebase.google.com`
- Medium – `medium.com`
- Hacking with Swift – `www.hackingwithswift.com`
- Stack Overflow – `stackoverflow.com`
- GitHub – `github.com`
- AppCoda – `www.appcoda.com`
- Sarunw – `sarunw.com`
- Yet Another Swift Blog – `www.vadimbulavin.com`
- FIVE STARS – `www.fivestars.blog`
- Wikipedia, The Free Encyclopedia – `www.wikipedia.org`

Ringraziamenti

Giunto al culmine di questo primo percorso universitario, mi sento in dovere di esprimere la mia più profonda gratitudine a tutte le persone che hanno reso possibile il raggiungimento di questo obiettivo, sostenendomi e accompagnandomi durante i tre anni.

In primo luogo, devo un debito incalcolabile alla mia famiglia, per l'affetto, la motivazione e la presenza costante lungo tutto il viaggio universitario. Grazie per aver sacrificato tanto per assicurare che nulla mi mancasse e per avere costantemente investito nella mia crescita. Questa laurea è tanto mia quanto vostra, e spero che rappresenti un tangibile simbolo del mio apprezzamento per tutto ciò che avete fatto e continuerete a fare per me, fintanto che ne avrete la possibilità.

Desidero dedicare un particolare ringraziamento a mia madre, la cui assenza è un vuoto ancora incolmabile, ma il cui amore e i cui insegnamenti continuano a vivere in me. Grazie per aver gettato le fondamenta della persona che sono oggi, attraverso la tua costante dedizione e il tuo grande amore. La tua influenza è indelebile e mi guida in ogni passo che compio. Nonostante non possiamo condividere questo momento insieme fisicamente, spero di averti resa orgogliosa con i traguardi che sto raggiungendo.

Ringrazio mio padre, la cui passione per la tecnologia e la cultura è stata per me una fonte inesauribile di ispirazione. Grazie per aver instillato in me questi interessi fin dalla mia giovinezza, e per avermi educato a vedere la vita da diverse prospettive. Grazie per aver sempre creduto in me, sostenendomi in tutto e per tutto e grazie, soprattutto, per tutti i grandi sacrifici che fai, quotidianamente, solo per me. Un ringraziamento speciale anche a Dori, che ha contribuito alla tua felicità e, di conseguenza, alla mia.

Ringrazio i miei nonni, che hanno assunto un ruolo paragonabile a quello di secondi genitori nella mia vita. Grazie per aver avuto, ogni giorno, l'obiettivo di farmi stare bene e rendermi felice. Grazie per aver camminato al mio fianco in ogni fase del mio percorso, sostenendomi con una dedizione ineguagliabile. L'ampiezza e la profondità dell'affetto che nutrite per me sono incommensurabili e spero di avervi fatto capire, a modo mio, che il mio affetto nei vostri confronti è altrettanto incalcolabile.

Vorrei, ora, ringraziare con tutto il cuore la mia fidanzata, Francesca, che mi ha permesso di diventare la persona che sono oggi, sicuramente migliore. Da quando ci siamo incontrati per la prima volta, hai dimostrato una dedizione straordinaria e una fiducia incondizionata in me, incoraggiandomi costantemente e credendo nelle mie capacità, anche quando ero io stesso il primo a non farlo. Grazie per avermi costantemente sostenuto negli innumerevoli momenti di difficoltà e per aver sempre contribuito a rimettermi in piedi, anche quando era l'ultima cosa che avrei voluto fare. Grazie per avermi insegnato ad essere più ottimista, cosa che mi ha permesso di non chiudermi totalmente in me stesso, facendomi vivere questo

percorso universitario con un'attitudine più positiva e resiliente. Grazie per avermi trasmesso la tua determinazione, senza la quale non sarei riuscito a raggiungere questo traguardo nel tempo che ci ho impiegato. Grazie davvero di cuore, infine, per tutti i meravigliosi momenti che abbiamo passato insieme, i quali hanno, senza ombra di dubbio, reso questo percorso più facile, e per essere, oltre che la mia fidanzata, la mia migliore amica.

Un grazie va anche ai genitori della mia fidanzata, Daniela e Lino, per avermi accolto e trattato, fin da subito, come un membro della famiglia e per avermi fatto vivere nuove esperienze che, sicuramente, hanno alleggerito alcuni momenti di difficoltà di questo percorso.

Non posso non ringraziare, poi, un grande amico, nonché collega, per cui ricambio l'appellativo di "fratello non di sangue", Nicola. Grazie per avermi insegnato la costanza e il grande impegno che metti in ogni cosa che fai. Grazie per tutti gli esami e i progetti preparati insieme, per tutte le risate e per poter sempre parlare di qualunque cosa. Nonostante ci conosciamo da relativamente poco tempo, hai rivoluzionato il mio modo di vivere l'università, ed è, quindi, sicuramente, anche grazie a te che ora mi trovo qui.

Prima di concludere, vorrei dedicare un ringraziamento anche a tutti i miei amici di Stella, con i quali ho trascorso momenti bellissimi fin dalla scuola materna, Nicolas, Riccardo, Lorenzo, Luca, Stefano e Lorenzo. Grazie anche a mio cugino Marco, che oltre ad essere un membro della famiglia, è anche un grande amico. In particolare, grazie a Francesco per gli innumerevoli pomeriggi trascorsi insieme e per tutti i nostri pranzi e cene dell'ultimo anno, durante i quali mi hai mostrato cosa significa avere una passione smisurata per ciò che si studia.

In conclusione, vorrei esprimere la mia gratitudine al Prof. Domenico Ursino, il mio relatore, per l'inestimabile e costante guida che mi ha offerto durante la redazione di questa tesi. Senza il suo apporto, la qualità del lavoro conseguito non avrebbe raggiunto il livello attuale; devo a lui l'ulteriore accrescimento del mio entusiasmo per questo campo di studio. Un ringraziamento speciale va anche al Dott. Enrico Corradini, il mio correlatore, non solo per gli utili suggerimenti offerti nell'implementazione dell'applicazione, ma anche per il suo spirito incredibilmente accessibile e affabile.