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'app in tecnologia Swift per la gestione di un portafoglio

Design and implementation of an app in Swift technology for the management of a wallet

Relatore

Prof. Domenico Ursino

Candidato

Michelangelo Amoruso Manzari

Correlatore

Dott. Enrico Corradini

ANNO ACCADEMICO 2022-2023

Possiamo ottenere l'approvazione degli altri, se agiamo bene
e ci mettiamo d'impegno nello scopo; ma la nostra stessa approvazione
vale mille volte di più.

Mark Twain

Sommario

Le applicazioni mobili rappresentano un settore in continua espansione. Negli ultimi anni abbiamo visto un aumento del numero di applicazioni, della qualità di sviluppo e delle capacità delle stesse, portandole a sostituire strumenti come siti web o attività commerciali. In questa tesi illustreremo lo sviluppo di un'applicazioni mobile, introducendo il linguaggio di programmazioni utilizzato, per poi passare all'analisi dei requisiti e alla progettazione dell'app. Infine, verranno implementate le scelte effettuate nei capitoli precedenti, esponendo i risultati e, al contempo, spiegando le tecniche utilizzate. Inoltre, l'applicazione verrà descritta all'interno di un manuale utente, compreso di sezione per la risoluzione dei problemi più comuni. Per concludere, verrà applicato il metodo SWOT per analizzare il progetto svolto e confrontarlo ad applicazioni correlate.

Keyword: Applicazioni Mobili, Linguaggio Swift, Sicurezza Informatica, Model-View-Controller, Portafoglio Virtuale, Schema del Database, API, Analisi SWOT

Indice

Indice

Elenco delle figure

Introduzione

Negli ultimi trenta anni, a partire dall'avvento di internet, le tecnologie si sono continuamente e inesorabilmente sostituite l'una all'altra. L'ultimo cambiamento radicale avvenuto può essere datato precisamente al gennaio del 2007, data in cui l'iPhone fu presentato al mondo per la prima volta. Il rilascio di tale tecnologia diede il via allo sviluppo, fino a quel momento ridotto, dei software per dispositivi mobili.

Grazie all'enorme successo degli smartphone, negli anni a venire gli investimenti in tecnologie e piattaforme adibite allo sviluppo di software per dispositivi mobili aumentò esponenzialmente. Già dai primi anni, grazie agli enormi sforzi e investimenti da parte dell'azienda californiana, l'IDE Xcode assunse un ruolo centrale e fondamentale nello sviluppo di questa tipologia di software, nello specifico per i dispositivi con sistema operativo iOS e macOS.

L'implementazione di nuove tecnologie e lo sviluppo del linguaggio Swift da parte di Apple portarono alla sostituzione, da parte degli smartphone, di molte operazioni. Le applicazioni mobili, quindi, diventarono strumenti adibiti a compiti specifici, ad esempio per la compravendita di biglietti elettronici oppure la gestione di un portafoglio virtuale.

L'app sviluppata risulta appartenere alla categoria dei "portafogli virtuali", ovvero applicazioni in grado di gestire informazioni riguardanti le proprie spese e renderle disponibili in ogni momento. L'app fornisce un servizio di inserimento delle proprie transazioni in entrata ed in uscita. Ciò che la contraddistingue da un'applicazione correlata è l'attenzione agli aspetti come design, esperienza utente e, soprattutto, l'attenzione alle necessità dell'utente. Essa, infatti, permette di inserire una transazione e visualizzarla all'interno di una lista. Oltretutto, l'app rende disponibili dei grafici, realizzati automaticamente mediante l'utilizzo dei dati inseriti dall'utente, che permettono di visualizzare le proprie spese e confrontarle tra loro. Essendo l'app incentrata sull'aspetto della consapevolezza delle proprie spese, la visualizzazione delle statistiche viene proposta in molteplici forme.

La presente tesi è composta da sette capitoli strutturati come di seguito specificato:

- Nel Capitolo 1 verrà introdotta la storia di Swift, per poi approfondire il linguaggio, analizzando i punti di forza e le prospettive future.
- Nel Capitolo 2 verrà presentato il progetto, descrivendo i requisiti funzionali e non funzionali, identificando i casi d'uso e gli attori.
- Nel Capitolo 3 saranno introdotti gli elementi grafici dell'applicazione, partendo dai wireframe e la storyboard. Infine, verrà introdotta la progettazione del database.
- Nel Capitolo 4 verranno presentate le tecnologie utilizzate per lo sviluppo e, successivamente, l'implementazione dell'applicazione.

- Nel Capitolo 5 verrà introdotto un manuale utente.
- Nel Capitolo 6 saranno discussi i risultati del processo di pianificazione e sviluppo.
- Nel Capitolo 7 verranno tratte le conclusioni e verranni presentati i possibili aspetti da poter migliorare in futuro.

CAPITOLO 1

Introduzione

In questo capitolo, verrà fornita una panoramica completa del linguaggio di programmazione Swift. Innanzitutto, verranno presentati la storia e lo sviluppo del linguaggio, dalla sua origine fino all'attuale versione. Saranno descritte le ragioni per cui Swift fu sviluppato e le esigenze che doveva soddisfare. Successivamente, verranno presentati i concetti fondamentali di Swift, tra cui la sintassi, i tipi di dati e il controllo del flusso. Verranno, inoltre, descritte le funzionalità avanzate di Swift, tra cui gli optional, i protocol e i generic, evidenziando le loro applicazioni e caratteristiche. Infine, verranno esaminati i principali ambiti di utilizzo di Swift, con particolare riferimento allo sviluppo di app per iOS, macOS e web. Saranno, infine, discusse le prospettive future del linguaggio, con un'analisi degli sviluppi futuri, delle nuove funzionalità e dei progetti in corso. Con questo capitolo si intende dare un'introduzione essenziale per comprendere appieno il linguaggio di programmazione Swift, non solo nella sua attuale forma, ma anche nelle sue possibili evoluzioni future.

1.1 Creazione di Swift: storia e sviluppo del linguaggio

Swift è un potente linguaggio di programmazione ad uso generale, rilasciato nel 2014 e utilizzato per sviluppare applicazioni per vari sistemi operativi, tra cui macOS, tvOS, watchOS, iOS e iPadOS.

Il linguaggio deriva da Objective-C, ma si differenzia per la sua enfasi sulla sicurezza e la chiarezza. Infatti, Swift elimina alcune classi di codice non sicuro presenti in altri linguaggi di programmazione, come, ad esempio, JavaScript, e consente agli sviluppatori di individuare alcuni bug già durante la compilazione del codice.

Inoltre, Swift presenta una sintassi chiara ed espressiva, che consente agli sviluppatori di scrivere programmi più concisi e di facile comprensione, rispetto ad altri linguaggi di programmazione, come Java o Objective-C.

1.1.1 Origine di Swift

L'origine di Swift risale al 2010, quando Chris Lattner, creatore di LLVM e Clang, insieme a Bertrand Serlet, all'epoca capo della Apple Software Team, iniziò a lavorare alla creazione di un linguaggio di programmazione migliore rispetto ad Objective-C. Inizialmente denominato "Shiny", il linguaggio fu presentato ai manager di Apple nel 2011 e divenne un progetto permanente all'interno dell'azienda.

1.1.2 Rilascio e sviluppi successivi

Swift fu presentato ufficialmente durante l'evento annuale WWDC (Worldwide Developers Conference) tenuto da Apple nel giugno 2014, ricevendo subito un'ottima accoglienza da parte degli sviluppatori. Nel dicembre 2015, con il rilascio della Versione 2.2, il codice sorgente del progetto fu reso accessibile e Swift divenne, così, un linguaggio open-source.

Durante la Versione 3.0, Apple dovette affrontare vari problemi di stabilità e compatibilità. Tuttavia, in nove anni di sviluppo, Swift è diventato un linguaggio di programmazione maturo, efficiente e in continua evoluzione, alla base di oltre la metà delle applicazioni disponibili sull'App Store (oltre 3 milioni) e utilizzato anche in altri ambiti, come il Machine Learning e lo sviluppo web.

1.2 Concetti fondamentali di Swift

Questa sezione approfondisce i concetti fondamentali di Swift, come la sintassi, i tipi di dati e il controllo del flusso. Essa fornisce informazioni dettagliate su questi argomenti, essenziali per comprendere la struttura e le funzionalità del linguaggio, che costituiscono la base del percorso di sviluppo di ogni applicazione. Pertanto, queste conoscenze sono fondamentali per ogni sviluppatore.

1.2.1 Sintassi

Il linguaggio Swift, simile al linguaggio C, utilizza variabili per conservare e identificare dati tramite nomi. Inoltre, è possibile utilizzare le costanti, ovvero variabili con valori stabili durante il processo, per rendere il codice più sicuro. La dichiarazione di costanti e variabili avviene tramite sintassi specifica, come mostrato nel codice seguente:

```
let esempioCostante = 100
var esempioVariabile = 50
```

In Swift, è anche possibile dichiarare valori multipli, come mostrato nel seguente codice. Inoltre, Swift non richiede il punto e virgola alla fine delle linee di codice, semplificando, così, la scrittura del codice e migliorando la sua leggibilità.

```
var x = 0.0, y = 2.0, z = 3.0
```

Per funzioni e metodi su più righe, Swift utilizza le parentesi graffe, unite all'indentazione, creando una gerarchia all'interno del codice e mantenendo il testo chiaro e ordinato.

Un'aggiunta interessante al linguaggio sono i Type Alias, utili a prevenire le ripetizioni all'interno del codice. Ad esempio, se si volessero utilizzare più funzioni con la stessa tipologia di dati, è possibile utilizzare i Type Alias fornite da Swift.

```
typealias esempioType = (Int, String, Double) -> (Int, String, Double)
...
func esempio1(esempio1: esempioType) { }
func esempio2(esempio2: esempioType) { }
func esempio3(esempio3: esempioType) { }
```

Swift è stato pensato per garantire la sicurezza del codice, con metodologie come la segnalazione degli errori durante la scrittura del codice. I Swift Object non possono essere di tipologia `nil` e il compilatore blocca direttamente l'esecuzione del codice per ridurre gli errori. Xcode, l'IDE di Swift, suggerisce di aggiungere un "`?`" per gestire eventuali dati `nil`, rendendo il codice più sicuro e ordinato.

Infine, Swift utilizza le Trailing Closure, che semplificano la scrittura del codice e lo rendono più leggibile ed efficiente. Tutte queste funzionalità rendono Swift un linguaggio moderno e innovativo, che si preoccupa della sicurezza del codice e della sua leggibilità.

1.2.2 Tipi di dati

Swift, come molti linguaggi di programmazione, presenta diversi tipi di dati che consentono di conservare e manipolare informazioni all'interno del codice. Alcuni di questi tipi possono essere considerati fondamentali e sono utilizzati in ogni contesto.

Tra i tipi di dati più comuni ci sono `Int`, che viene utilizzato per gestire numeri interi positivi o negativi, e `Double`, che viene utilizzato per gestire numeri con la virgola e offre una maggiore precisione rispetto ad `Int`. Per quanto riguarda i caratteri e le stringhe di testo, Swift utilizza il tipo `String` per conservare e manipolare sequenze di caratteri.

Passando a tipi di dati più complessi, troviamo gli Array e le Tuple. Gli Array sono utilizzati per conservare tipi di dati omogenei tra loro, ciascuno con un metadato che indica la sua posizione all'interno dell'Array. Ad esempio, un Array con 5 elementi avrà 5 posizioni, dalla 0 alla 4.

Le Tuple consentono, invece, di assegnare più dati allo stesso oggetto, in modo da poter passare più dati a una funzione utilizzando un solo oggetto. Le Tuple sono molto utili per mantenere il codice chiaro e conciso.

Approfondendo i dati strutturati, troviamo gli Optional, che possono contenere un valore o essere nulli. Questo evita errori durante l'esecuzione del codice in caso di dato non inizializzato.

Infine, troviamo i dati definiti dall'utente, che includono Struct, Classi ed Enum. Una Struct, va innanzitutto, dichiarata. Al suo interno vanno inserite le variabili appartenenti alla struttura. Nel codice sottostante è osservabile la dichiarazione di una Struct; successivamente viene dichiarata una variabile A `persona1` a cui è assegnata la struct A `Persona`. A `persona1` è ora possibile aggiungere o modificare dati come descritto nel codice. Dopo aver aggiunto nome, età e laurea, possiamo visualizzare il risultato attraverso la funzione A `print`, che stamperà il risultato "35" in console.

```
struct Persona {
    var nome = ""
    var anni = 0
    var laurea = false
}

var persona1 = Persona()

persona1.nome = "Giovanni"
persona1.anni = 35
persona1.laurea = true

print("Anni: \(persona1.anni)")

Output --> Anni: 35
```

1.2.3 Controllo del flusso

Swift fornisce una vasta gamma di flow statement, ovvero istruzioni di controllo del flusso. Tra di esse, troviamo i loop come `for` e `while`, le istruzioni `if`, `guard` e `switch`,

utili per selezionare un ramo di codice da eseguire in base a condizioni. E, infine, le istruzioni `break` e `continue`, `fallthrough` e altre.

I cicli possono essere di tre tipologie: `for-in`, `while`while e `repeat-while`, dove quest'ultimo corrisponde al `do-while` in altri linguaggi.

Il ciclo `for-in` è utilizzato per scorrere valori all'interno di vettori, dizionari o altre tipologie di dati. Spesso viene utilizzato per svolgere operazioni di ricerca di dati. Ad esempio, si potrebbe usare un ciclo `for-in` per scorrere i nomi di una lista e trovare la presenza di uno specifico nome all'interno di essa, come nell'esempio sottostante.

```
let lista = ["Marco", "Giovanni", "Maria"]

for nome in lista {
    if nome == "Giovanni"{
        print("Corrispondenza trovata")
    }
}
```

I cicli `while` e il `repeat-while` differiscono dal ciclo `for` per la condizione di continuazione dell'iterazione. Infatti, i cicli `while` continuano ad iterare il codice finché la condizione del ciclo risulta vera, mentre il ciclo `repeat-while` esegue l'iterazione una volta prima di verificare la condizione.

I cicli rappresentano solo una parte del controllo del flusso offerto da Swift. Infatti, per verificare le condizioni si utilizzano le istruzioni condizionali, come l'`if statement`, che permette di eseguire un blocco di codice solo se la condizione specificata risulta vera. Inoltre, l'istruzione `else` permette di aggiungere un caso alternativo. Ad esempio, se a è maggiore di b, si può scrivere "a è maggiore di b", altrimenti si può scrivere "a è minore di b" utilizzando l'istruzione `else`.

```
if a < b {
        print('a maggiore di b')
} else {
        print('a minore di b')
}
```

Swift offre una vasta gamma di istruzioni di controllo di flusso, tra cui l'istruzione condizionale `switch`. Questa istruzione confronta un valore con una serie di casi e, se trova una corrispondenza, esegue il codice associato a quel caso. È possibile specificare un caso di "default" che verrà eseguito se non viene trovata alcuna corrispondenza.

All'interno dei casi e del caso "default", è possibile utilizzare istruzioni di trasferimento del controllo per eseguire altre parti del codice. Queste istruzioni possono essere di cinque tipi: `continue`, `break`, `fallthrough`, `return` e `throw`. Esse possono essere utilizzate anche all'interno di cicli `while`, `for` e istruzioni `if-else`.

1.3 Funzionalità avanzate di Swift

Swift è un linguaggio di programmazione moderno e potente; questo poiché è composto da funzionalità avanzate di grande utilità per gli sviluppatori. Tra queste troviamo gli optional, che permettono di gestire il valore di una variabile senza sapere se quest'ultima verrà inizializzata come `nil` (variabile con assenza di valore), i protocol, che definiscono un set di metodi, proprietà e ulteriori requisiti che verranno implementati da una classe o struttura, e i generic, utili per scrivere codice riutilizzabile e di facile adattabilità per la gestione di diversi tipi di dati. Grazie a queste funzionalità avanzate, gli sviluppatori hanno la

possibilità di creare applicazioni sofisticate, efficienti e funzionali. In questa sezione verranno esplorate le funzionalità sopraelencate, unendo brevi esempi e casi d'uso per fornire una panoramica completa.

1.3.1 Optional

Gli Optional sono una delle funzionalità che rendono Swift meno prono ad errori. Essi consentono di gestire variabili con valori non inizializzati ed evitare di incorrere in errori quando queste risultano nulle. Infatti, in molti casi, una variabile potrebbe non avere un valore assegnato. Questo normalmente costringerebbe lo sviluppatore a inserire istruzioni `if-else` per gestire gli errori; Apple ha pensato ad una soluzione più elegante. Nell'esempio sottostante sono rappresentate due soluzioni uguali; la seconda impiega l'uso di Optionals, rendendo il codice semplice e ordinato.

```
if (myDelegate != nil) {
    if ([myDelegate class_respondsToSelector:
        @Selector(scrollViewDidScroll:)]) {
        [myDelegate scrollViewDidScroll:myScrollView];
    }
}

myDelegate?.scrollViewDidScroll?(myScrollView)
```

Possiamo, quindi, considerare un Optional come un "contenitore" che può o non può contenere un valore effettivo. Questa funzionalità di Swift permette l'utilizzo di codice robusto e sicuro, poiché evita errori causati da valori nulli.

Tra i vari utilizzi degli operatori Optional troviamo il "Nil-Coalescing Operator", ossia un operatore che offre un'alternativa al valore Optional, in caso esso risulti `nil`. Nel codice sottostante vengono dichiarate due costanti. Per la costante `percorsoProfilo` viene fornito un percorso per l'immagine chiamata `Profilo`; se essa non esiste, grazie agli optional, sarà utilizzato il valore di `percorsoDefault`, così da evitare che `percorsoProfilo` diventi una costante nulla, causando, di conseguenza, un errore.

```
let percorsoDefault = '/immagini/default.png'
let percorsoProfilo = percorsoImmagini['profilo'] ?? percorsoDefault
```

L'ultimo caso d'uso degli Optional è l'Unconditional Wrapping. Questo si adotta per utilizzare un valore Optional quando si è sicuri che esso contenga un valore. La differenza sintattica sta nel segno utilizzato; infatti al posto di un punto interrogativo, si utilizza un punto esclamativo, come da esempio.

```
let valore = Int("13")!
```

1.3.2 Protocol

I protocol di Swift sono una funzionalità potente del linguaggio. Essi consentono di definire insiemi di metodi, proprietà e altro ancora e di utilizzare questi all'interno di classi o struct (strutture). Rappresentano uno dei capisaldi della programmazione orientata agli oggetti e permettono la scrittura di codice modulare e flessibile. Un protocol può essere pensato come una specifica di requisiti che una classe o struttura deve rispettare e fornire. D'ora in poi definiremo classi e strutture come tipi conformi. Questi possono adottare uno o più protocol dichiarandoli durante la loro inizializzazione. Questa funzionalità consente di creare relazioni flessibili, facilmente modificabili e condivisibili tra vari tipi conformi. Inoltre, utilizzando i protocol, il codice risulterà corto e conciso.

Nel codice sottostante viene dichiarato un protocollo, al cui interno possono essere inseriti tutti i metodi e le proprietà del caso. Successivamente viene dichiarata una classe che utilizza `protocollo1` per ereditare le sue proprietà ed evitare di riscriverle all'interno della classe. Così facendo possiamo creare molteplici strutture, come classi e struct, ed evitare di riscrivere il codice comune a tutte più volte.

```
protocol procotollo1 {
        // definizione dei potrocolli
}

class Classe1: protocollo1 {
        // Codice classe
}
```

Tra le varie funzionalità dei protocol troviamo la Delegazione, cioè l'utilizzo di un protocol per delegare le responsabilità di un'azione di un tipo conforme. Prendiamo come esempio una serie di giochi che hanno in comune l'utilizzo di dadi. Si potrebbe utilizzare un protocol per calcolare i tiri dei dadi e, quindi, delegare una funzionalità di tutti i giochi ad un solo protocol.

Un protocol può, inoltre, ereditare uno o più protocol e aggiungere ulteriori requisiti a quelli ereditati. Questo rende i protocol paragonabili a dei blocchi componibili che permettono di adattare facilmente una serie di requisiti per un caso d'uso specifico.

Infine, l'ultima funzione essenziale dei protocol è la possibilità di limitare l'uso degli stessi solo alle classi.

1.3.3 Generic

I generic sono una caratteristica di grande utilità in Swift, in quanto consentono di creare funzioni e tipi che possono essere utilizzati per diversi soggetti, senza dover ripetere il codice. Essi permettono di esprimere in modo astratto e chiaro i requisiti necessari.

Secondo la definizione di Apple, i generic sono una delle funzionalità più potenti di Swift e sono alla base della Swift Standard Library, che include elementi fondamentali, come Array e Dizionari. Sono così ampliamente utilizzati nel linguaggio che si trovano praticamente in ogni codice scritto in Swift.

La loro utilità risiede nell'ampiezza di applicazione. Ad esempio, consideriamo il codice seguente che utilizza una funzione chiamata `ScambiaValori` per scambiare due valori interi. Senza l'uso dei generic, sarebbe necessario creare due funzioni aggiuntive per scambiare valori di tipo `Double` e `String`.

```
func ScambiaValori(_ a: inout Int, _ b: inout Int) {
        let temporaneaA = a
        a = b
        b = temporaneaA
}
```

Grazie ai generic, è possibile riscrivere questa funzione in modo universale per qualsiasi tipo di dato, riducendo, così, le numerose funzioni a una sola. Nel codice seguente è possibile osservare la sintassi di una funzione generica, che presenta somiglianze con la funzione mostrata precedentemente.

```
func ScambiaValori<T>(_ a: inout T, _ b: inout T) {
        let temporaneaA = a
        a = b
        b = temporaneaA
```

}

La differenza nella sintassi è minima; infatti è evidente come l'unica modifica risieda nell'uso di un segnaposto '<T>' a cui non è stato assegnato un tipo di valore specifico.

L'utilità dei generic non si limita alle sole funzioni predefinite; è possibile, infatti, definire i propri tipi generici. Questi tipi possono essere classi, strutture o enumerazioni personalizzate che possono lavorare con qualsiasi tipo di dato in modo simile ad Array e Dizionari.

La vera differenza rispetto alle classi e alle altre strutture personalizzate risiede nella capacità di utilizzare un segnaposto, e quindi rimuovere i vincoli sulla tipologia dei dati all'interno della struttura. Questo rende effettivamente il codice scritto con i generic universale e altamente flessibile.

1.4 Utilizzo di Swift in diverse applicazioni: sviluppo di app iOS, macOS e web

Swift, come detto in precedenza, è un linguaggio versatile, efficiente e, al contempo, potente, in grado di offrire la possibilità di sviluppo per molteplici piattaforme, senza perdere funzionalità.

Grazie agli sforzi di Apple, i punti di forza del linguaggio sono la semplicità e la sicurezza, motivo per cui Swift è diventato la base per la creazione di app per dispositivi iOS, macOS ed anche per il web.

Per sviluppare queste app, Apple offre gratuitamente l'IDE (Integrated Development Environment) Xcode. Questa piattaforma di sviluppo è tra le più avanzate nell'ambito della programmazione di applicazioni grazie alle funzionalità che offre. Questo ambiente di sviluppo è stato creato per Swift, con la sicurezza e l'efficienza come priorità.

Proprio grazie a queste caratteristiche è estremamente versatile e permette di sviluppare ed adattare applicazioni per diverse piattaforme in tempi molto brevi. Inoltre, Swift supporta caratteristiche esclusive delle piattaforme Apple, come l'utilizzo di API e framework nativi, che permettono di sfruttare al massimo l'hardware fornito da Apple.

Per dimostrare la versatilità di Swift basta osservare gli ultimi sviluppi delle piattaforme Apple. Dal 2020 è, infatti, possibile utilizzare applicazioni prodotte per iOS. Su dispositivi che utilizzano macOS. Questa versatilità è oramai una colonna portante del linguaggio.

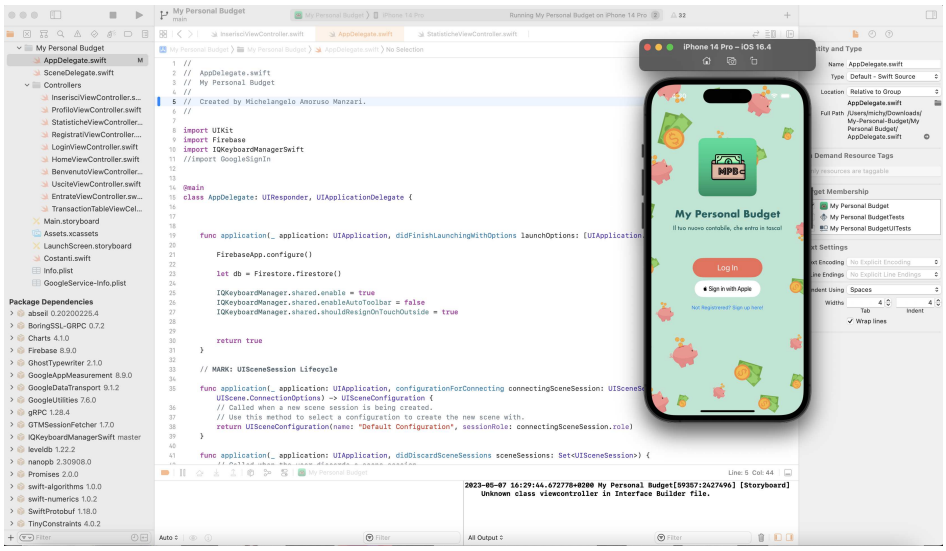

Figura 1.1: L'IDE Xcode con affiancato un emulatore iPhone

Pur essendo progettato per lo sviluppo di applicazioni iOS e macOS, Swift si contraddistingue anche nell'ambiente dello sviluppo web grazie a framework come Vapor e Kitura, forniti da Apple stessa.

L'uso di Swift per il web offre numerosi vantaggi. La sintassi ordinata e concisa rende veloce lo sviluppo; la forte tipizzazione del linguaggio fornisce maggiore sicurezza e rimuove totalmente la possibilità di incappare in errori nella fase di sviluppo. Inoltre, essendo Swift altamente efficiente, esso risulta ottimo per lo sviluppo di codice lato server.

1.5 Prospettive future di Swift: sviluppi futuri, nuove funzionalità e progetti

Il futuro di Swift potrebbe essere visto come codipendente dall'ecosistema Apple e la direzione intrapresa dalla stessa, ma, come detto in precedenza, Swift è open-source. Questa metodologia di sviluppo di un linguaggio, unita alla popolarità dei sistemi che lo utilizzano, hanno allontanato Swift da Apple, rendendolo, quindi, indipendente dalle scelte dell'azienda per quanto possibile.

Una delle direzioni chiave per il futuro di Swift è l'espansione dell'ecosistema al di là delle piattaforme iOS, macOS e web. Swift ha già fatto il suo ingresso nel campo del server-side development, con framework come Vapor e Kitura, consentendo agli sviluppatori di creare applicazioni web scalabili e ad alte prestazioni. Ciò apre la strada a un uso più ampio di Swift nel contesto del cloud computing e dell'IoT (Internet of Things).

Inoltre, Swift continua a evolversi come linguaggio, con l'introduzione di nuove funzionalità e miglioramenti. Apple e la comunità degli sviluppatori stanno lavorando costantemente per rendere Swift più potente, più intuitivo e più facile da imparare. Ciò include il supporto per la programmazione asincrona, che ha avvicinato molti sviluppatori al mondo dello sviluppo server-side, il miglioramento delle performance e l'introduzione di nuovi strumenti di sviluppo.

Il futuro di Swift include anche una maggiore collaborazione e condivisione del codice tra le piattaforme. Si pensi, ad esempio, all'introduzione di SwiftUI, un framework di interfaccia utente dichiarativa, dove gli sviluppatori possono scrivere interfacce utente reattive per iOS, macOS e altre piattaforme utilizzando un'unica base di codice. Questo framework ha portato ad un notevole miglioramento nell'efficienza degli applicativi Swift, nonché a uno standard uguale per tutte le piattaforme.

In conclusione, il futuro di Swift si prospetta entusiasmante, con continui miglioramenti, una comunità estremamente attiva e l'aumento di piattaforme che ne fanno utilizzo. Con la sua combinazione di potenza, sicurezza, efficienza e facilità d'apprensione, Swift continuerà a rivoluzionare il panorama della programmazione, portando ad un pubblico sempre più vasto prodotti innovativi nell'ambito dello sviluppo software.

CAPITOLO 2

Specifica e Analisi dei Requisiti

L'analisi dei requisiti rappresenta la fase iniziale di ogni progetto. In questo capitolo verranno compresi, descritti ed analizzati appieno le esigenze e i requisiti del progetto. Verrà proposta una panoramica dell'approccio utilizzato per definire le funzionalità richieste, gli attori coinvolti, i requisiti funzionali e non funzionali e, infine, i casi d'uso. Saranno prodotti diagrammi con l'intento di visualizzare i diversi scenari di interazione tra l'applicazione e l'utente e saranno definiti i flussi di lavoro dell'applicazione.

2.1 Descrizione del Progetto

L'applicazione sviluppata è un portafoglio virtuale, nata con l'idea di permettere all'utente l'inserimento di transazioni in entrata e in uscita, specificando la categoria d'appartenenza di ogni spesa. La categorizzazione delle spese avviene per poter implementare un riassunto delle spese, presentato mediante l'uso di diagrammi e grafici. La suddivisione delle transazioni in 2 macro-categorie risulta essere il metodo più chiaro per esprimere dati così diversi tra loro, dando, così, all'utente una consapevolezza maggiore in merito alle sue spese. L'applicazione, oltre a fornire i dati sotto forma di grafici, permette di visualizzare le singole transazioni, identificandole attraverso l'uso di colori. L'implementazione dell'applicazione ha come obiettivo un'interfaccia grafica ordinata, sintetica ed intuitiva. Dovrà essere funzionale, ma, al contempo, efficace nel rendere semplice la lettura dei dati dell'utente. L'applicazione verrà divisa in 4 sezioni usando un menù detto Navigation Bar; tale sezioni sono:

- *Transazioni*: all'interno di questa sezione sarà possibile visualizzare le transazioni dell'utente mediante una tabella composta da celle, ciascuna contenente un dato della transazione. Sarà, inoltre, possibile inserire le nuove transazioni.
- *Statistiche*: essa sarà una sezione dedicata alla visualizzazione, mediante appositi grafici, dei dati inseriti dall'utente in un formato diretto e adatto alla tipologia di dati.
- *Aiuto*: in questa sezione verrà mostrato un breve riassunto delle capacità e modalità d'uso dell'applicazione. Questo sarà possibile attraverso l'uso di una webview collegata ad un video tutorial.
- *Profilo*: quest'ultima sezione sarà dedicata alla visualizzazione e all'aggiornamento dei dati relativi al profilo dell'utente. L'utente potrà visualizzare e modificare i propri dati, rispettando i requisiti imposti dall'applicazione.

2.2 Requisiti Funzionali e non Funzionali

Lo sviluppo di un'applicazione mobile richiede un'analisi dei requisiti molto dettagliata. In questa fase è fondamentale definire in maniera chiara e minuziosa tutto ciò che l'applicazione deve concretizzare e quali aspettative soddisfare. I requisiti possono essere suddivisi in due categorie principali: requisiti funzionali e requisiti non funzionali. I requisiti funzionali riguardano le specifiche che definiscono le funzionalità e il comportamento dell'applicazione stessa. Essi stabiliscono ciò che essa deve essere in grado di fare. I requisiti non funzionali, invece, riguardano le caratteristiche non legate alle funzionalità dell'applicazione, ma che risultano essere cruciali per garantire le prestazioni, la sicurezza e una buona qualità generale dell'applicazione.

2.2.1 Requisiti Funzionali

I requisiti funzionali di quest'applicazione dovranno rispettare il paradigma CRUD, acronimo utilizzato per indicare le quattro operazioni di base per la gestione di dati in un database: Create (creazione), Read (Lettura), Update (Aggiornamento) e, infine, Delete (Eliminazione).

Registrazione di un utente e login

Per usufruire dei servizi dell'applicazione, il primo passo è quello di creare un account. La creazione dello stesso avviene necessariamente attraverso l'inserimento di dati, tra cui il nome, il cognome, l'email ed una password. Oltre alla registrazione manuale, viene fornita la possibilità di utilizzare la funzione "Sign in with Apple", un servizio offerto da Apple che permette di automatizzare la registrazione utilizzando l'email del proprio account Apple unita alla generazione automatica di una password. Qualora un utente sia già registrato, è possibile eseguire il login mediante l'inserimento dell'email con cui è avvenuta la registrazione e la password associata all'account. Entrambe le operazioni descritte appartengono alle attività di tipo CRUD.

Visualizzazione ed inserimento delle transazioni

La visualizzazione e la creazione delle transazioni rappresentano le principali funzionalità dell'applicazione. Non appena si effettua il login, l'utente verrà portato alla schermata di visualizzazione delle transazioni. In questa schermata vengono fornite tutte le transazioni registrate dall'utente e viene evidenziata la macro-categoria di appartenenza attraverso l'uso di colori. Per quanto riguarda l'inserimento delle transazioni, questo avviene attraverso l'uso di due schermate, entrambe formate da una serie di immagini stilizzate che ne rappresentano la categoria di appartenenza. L'operazione di aggiunta dei dati di una transazione comprende soltanto l'inserimento della cifra. La data e la categoria di appartenenza vengono associate quando viene premuto il pulsante collegato all'immagine.

Visualizzazione delle statistiche

La gestione della visualizzazione dei dati inseriti avviene mediante l'uso di grafici a torta. Questi grafici rappresentano le entrate, le uscite e il bilancio totale, fornendo così a una panoramica completa delle spese. I grafici vengono creati leggendo i dati permanenti dell'utente. Questa funzionalità, oltretutto, è in continuo aggiornamento, rendendo i grafici perennemente aggiornati all'ultima transazione.

Visualizzazione e modifica dei dati del profilo utente

Questa funzionalità permette di visualizzare i dati inseriti dall'utente durante la fase di registrazione, leggendo i dati utente contenuti all'interno del database. Contemporaneamente alla visualizzazione, è possibile modificare i dati rispettando i requisiti imposti sugli stessi, per poi inviare i dati modificati al database.

2.2.2 Requisiti Non Funzionali

A differenza dei requisiti funzionali, quelli non funzionali trattano funzionalità di design e prestazioni. La descrizione di queste funzionalità si concentrerà sulle scelte intraprese per rendere l'applicazione semplice, intuitiva e con un design dettato dai requisiti forniti da Apple all'interno del documento, "Human Interface Guidelines".

Verranno, inoltre, descritti aspetti dell'applicazione, come la sicurezza dell'account e l'ottimizzazione dell'inserimento delle transazioni.

Sicurezza dei dati utente

La sicurezza dei dati utente è un requisito fondamentale per qualsiasi applicazione, nella fattispecie iOS e l'ecosistema Apple. Per l'appunto, Apple è da anni impegnata nel settore della Cyber ecurity, rendendo la sicurezza dei dati dei propri utenti un requisito necessario per la pubblicazione di applicazioni. Il sistema di autenticazione e gestione dei dati dell'applicazione avviene mediante l'implementazione del servizio Firebase, fornito gratuitamente da Google. Questo servizio permette di implementare un database di tipo NoSQL con funzioni semplificate e pronte all'uso. Tra i vari vantaggi di Firebase troviamo la facilità d'uso, la compatibilità multi-piattaforma e, infine, l'utilizzo di chiavi utente univoche auto-generate nella fase di registrazione. Durante la fase di autenticazione, per ridurre i rischi derivanti dalla registrazione e l'accesso, l'applicazione deve imporre dei requisiti. Tra questi troviamo la necessità di inserire un nome e un cognome, l'impossibilità di inserire un indirizzo email già utilizzato e la necessità di inserire una password che contenga almeno 6 caratteri. Tutte queste limitazioni sono volte ad obbligare l'utente a fornire i dati necessari per un utilizzo corretto dell'applicazione e a rispettare i requisiti del database, come la "@" all'interno dell'email. Mediante l'uso di chiavi univoche e crittografate, l'applicazione risulta più efficiente, rendendo l'accesso ai dati utente semplice, diretto e inequivocabile. L'assegnazione di tali chiavi è gestito direttamente dal servizio Firebase in modo autonomo.

Usabilità dell'applicazione

Uno degli aspetti chiave per una buona esperienza d'uso per l'utente è l'usabilità dell'applicazione. Questo termine, a prima vista generico, può essere espresso usando termini più specifici, come l'interfaccia utente e l'esperienza utente, tradotti dall'acronimo inglese UI/UX. Per garantire una buona esperienza vi sono vari passi da intraprendere. Nel caso dell'applicazione in discussione è stato utilizzato un approccio lineare, dove ogni funzione è sempre disponibile nella sezione inferiore dello schermo, mediante l'utilizzo di una navigation bar. Oltre all'approccio lineare dell'applicazione, è stato implementato un design semplice, intuitivo e che fa uso di icone stilizzate per rimuovere il testo e rendere ogni schermata visivamente semplice. Ad esempio, all'interno dell'inserimento delle transazioni, sono presenti solo icone che rappresentano le singole categorie, esponendo, così, tutte le tipologie di transazioni in una sola pagina. La filosofia di design utilizzata è quella fornita da Apple nella documentazione denominata "Human Interface Guidelines", cioè una serie di scelte standardizzate con l'obiettivo di rendere chiara ed intuitiva l'interfaccia utente. Difatti, Apple

persegue questo design minimalista in ogni sua applicazione. Estendendo queste linee guida alle terze parti, Apple è riuscita a rendere il suo ecosistema estremamente intuitivo anche ai non nativi digitali.

Compatibilità dell'applicazione

Quando si struttura un'applicazione, vi è la necessità di prestare attenzione a tutti i dispositivi che potranno farne uso. Ad esempio, un'applicazione iOS deve necessariamente essere in grado di disporre correttamente la grafica su diverse tipologie di schermi, con misure differenti e colori talvolta non simili. All'interno di Xcode, l'IDE fornito da Apple, sono presenti vari strumenti pensati appositamente per questo scopo. Tra questi troviamo le constraint, ovvero dei vincoli imposti dal programmatore con l'intento di imporre limiti agli oggetti grafici presenti nelle varie schermate. Grazie a queste limitazioni è stato possibile rendere la grafica adatta ad ogni schermo per dispositivi che utilizzano iOS 9.0 e oltre.

2.3 Identificazione degli attori e dei casi d'uso

L'utilizzo dell'applicazione verrà permesso esclusivamente al singolo utente. Egli non solo rappresenterà l'unico attore coinvolto, ma sarà anche il fulcro delle interazioni con le varie funzioni e caratteristiche dell'applicazione. L'applicazione verrà, quindi, progettata tenendo conto delle esigenze e delle aspettative di un utente medio, al fine di raggiungere una qualità d'esperienza soddisfacente. Per ottenere questo obiettivo, saranno implementati un'interfaccia intuitiva e un flusso di utilizzo lineare.

2.4 Creazione dei diagrammi dei casi d'uso

In Figura 2.1 sono state indicate le varie modalità con cui l'Utente potrà interagire con l'applicazione. Egli avrà la possibilità di interagire con le funzioni previste mediante la navigation bar inserita nell'interfaccia dell'applicazione. Ognuna delle funzioni avrà una sezione dedicata, identificabile attraverso icone stilizzate. L'obiettivo è stato quello di immaginare un'interfaccia che renda l'applicazione lineare e, di conseguenza, semplice, tenendo conto delle esigenze dell'utente. In Figura 2.1 sono evidenziati i casi d'uso dell'applicazione, a partire dalla visualizzazione delle transazioni, presentata immediatamente dopo l'autenticazione dell'utente.

Visualizzazione delle transazioni

Questa schermata è la prima ad essere presentata all'utente. Questo poiché essa rappresenta il fulcro dell'applicazione; difatti la sezione "Statistiche" dipende dall'inserimento dei dati in questa sezione. L'organizzazione della grafica è incentrata sulle transazioni; esse vengono, infatti, esposte al centro della pagina. Sono, inoltre, presenti due pulsanti per l'inserimento delle transazioni, il primo per le transazioni relative alle entrate e il secondo per le uscite. L'utente può, quindi, accedere a due schermate con grafica intuitiva, volte a rendere l'inserimento di valori chiaro ed efficace.

Visualizzazione delle statistiche

La rappresentazione dei dati è stata effettuata mediante l'uso di tre grafici a torta. Questi risultano essere la soluzione più adatta a schermi di piccola dimensione quando si tratta di convogliare molti dati in una soluzione semplice ed elegante. L'utente può accedere alla

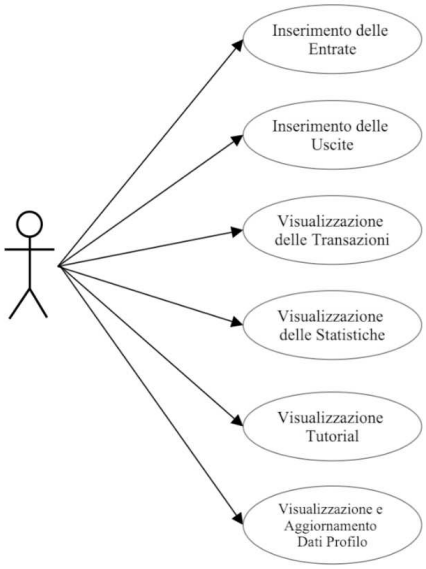

Figura 2.1: Diagramma dei Casi d'Uso

schermata utilizzando la navigation bar e, successivamente, visualizzare i grafici toccando i singoli pulsanti associati ad essi. L'utente, in questo caso, richiede di leggere i dati delle transazioni dal database Firebase.

Visualizzazione e aggiornamento dei dati del profilo

L'utente può svolgere l'azione di richiesta dati e, successivamente, rimodularli all'interno della schermata. L'azione di aggiornamento avviene mediante l'utilizzo di un pulsante che, quando viene premuto, invia un comando di aggiornamento dei dati del profilo dell'utente. In questo caso vengono utilizzate solo due delle quattro azioni CRUD, cioè lettura e aggiornamento.

CAPITOLO 3

Progettazione

La fase di progettazione è un momento critico nello sviluppo di un'applicazione mobile. Durante questa fase, si delineano gli obiettivi, si identificano le esigenze degli utenti e si pianificano l'architettura e l'interfaccia dell'applicazione. In questo capitolo verranno esplicati i processi di sviluppo dell'applicazione, partendo dalla mappa della stessa, per poi passare all'interfaccia grafica. Successivamente verranno analizzate la tecnologia utilizzata per l'organizzazione e la gestione dei dati e, soprattutto, le metodologie impiegate affinché l'applicazione risulti leggera, efficiente e ben organizzata.

3.1 Mappa dell'applicazione

La mappa dell'applicazione è uno strumento utile a definire il percorso per visualizzare le pagine dell'applicazione. Viene descritta partendo dall'alto, come mostrato nella Figura 3.1, con la prima pagina visualizzata, cioè la Splash Page, che si occupa della presentazione dell'applicazione. Successivamente, vengono fornite tre opzioni all'utente: effettuare il login utilizzando le proprie credenziali, accedere o registrarsi mediante la funzione "Sign In with Apple" oppure registrarsi usando l'indirizzo email, il nome ed il cognome. Effettuato l'accesso, l'utente viene direttamente portato alla pagina delle transazioni, dalla quale si può decidere se visualizzare le proprie transazioni o, in alternativa, inserirne delle nuove. Mediante la navigation bar è, inoltre, possibile accedere a tutte le pagine principali dell'applicazione, come la sezione delle Statistiche, la sezione Aiuto e, infine, la sezione Profilo. In tutte e quattro le pagine principali l'utente può decidere, come mostrato nella figura, di effettuare il logout, riportandolo così alla Splash Page. Tornando ora alla sezione delle transazioni, essa permette di accedere a due sezioni denominate "Entrate" e "Uscite", all'interno delle quali è, rispettivamente, possibile inserire nuove transazioni sia in entrata, che in uscita, inserendo la somma della transazione. A seguito troviamo la sezione Statistiche, che permette di accedere, mediante l'uso di pulsanti, a tre grafici, ciascuno contenente dati utili a creare una panoramica delle transazioni inserite nella sezione precedente. Troviamo poi le sezioni Aiuto e Profilo. Grazie alla prima, è possibile visualizzare un breve video introduttivo all'applicazione mediante l'utilizzo di una web-view, ovvero una funzione che permette di visualizzare una pagina web, inclusi video o animazioni. La seconda rappresenta l'ultima delle quattro sezioni principali e permette di visualizzare o modificare i propri dati utente inserendo i dati da aggiornare nelle apposite caselle. Se i dati inseriti rispettano i canoni richiesti dal database, verrà visualizzato un messaggio di inserimento avvenuto correttamente, altrimenti verrà segnalato l'errore e, al contempo, come correggerlo.

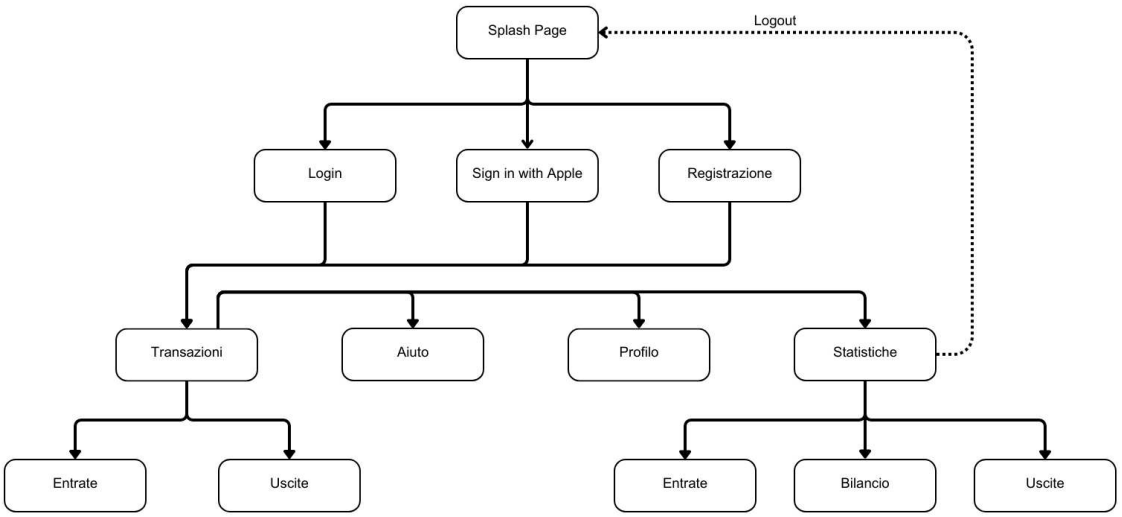

Figura 3.1: Diagramma dei Casi d'Uso

3.2 Wireframe e Storyboard

Per la creazione della grafica e delle funzionalità associate ad ogni pagina dell'applicazione, sono stati realizzati dei wireframe. Essi rappresentano, in maniera sintetica, sia la disposizione delle funzioni principali dell'applicazione, che i fondamenti della grafica. Nella figura sottostante (Figura 3.2) è rappresentata la splash page, ovvero la prima pagina che viene mostrata all'utente all'avvio dell'applicazione. Il design scelto è semplice ed ordinato, creato seguendo le linee guida imposte da Apple e riducendo al minimo l'utilizzo di grafica complessa. Nella figura sono presenti tre sezioni principali: il login mediante email e password, il link per potersi registrare e, infine, il pulsante "Sign in with Apple". Quest'ultimo, fornito dall'API di Apple stessa, permette all'utente di registrarsi o, nel caso avesse precedentemente effettuato la registrazione con questo metodo, di effettuare il login. Nella fase di registrazione automatica, Apple fornisce, inoltre, un servizio aggiuntivo, ovvero la funzione "Hide my email". Questa funzionalità genera un'email gestita da Apple, con l'intento di non fornire i dati dell'utente al database.

Successivamente, qualora l'utente non sia registrato, si procede con la fase di registrazione, che, come detto prima, può avvenire in due modalità: mediante l'utilizzo dell'API Apple oppure attraverso la modalità classica, che richiede l'inserimento manuale dei propri dati da parte dell'utente stesso. Nella Figura 3.3 sono mostrate le due modalità di registrazione disponibili. Nel primo caso, grazie all'API "Sign in with Apple", vengono automaticamente compilati tutti i campi richiesti dall'applicazione. I dati utilizzati vengono presi direttamente dai dati dell'account Apple dell'utente. Inoltre, vi è anche la possibilità di registrarsi utilizzando un'email auto-generata, con inoltro automatico verso la vera email associata all'account. L'utilizzo di questa funzionalità è possibile proprio grazie all'utilizzo del database Firebase. Infatti, in fase di configurazione del Database, è possibile inserire le varie modalità di registrazione che si vogliono rendere disponibili all'utente. Passiamo ora al secondo wireframe presente nella figura, esso rappresenta la modalità di registrazione manuale. In questo caso l'utente dovrà inserire i dati richiesti nelle apposite caselle di testo. All'invio dei dati, questi vengono verificati e, se rispettano i requisiti, vengono salvati, altrimenti viene presentato un messaggio di errore relativo al dato da correggere.

Una volta effettuato l'accesso mediante una delle modalità fornite, l'utente viene portato direttamente alla pagina iniziale, ovvero quella delle transazioni. In questa sezione,

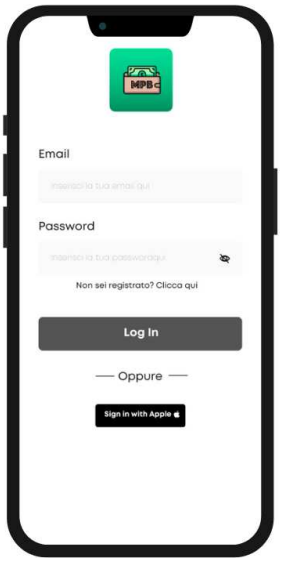

Figura 3.2: Splash Page - Pagina iniziale

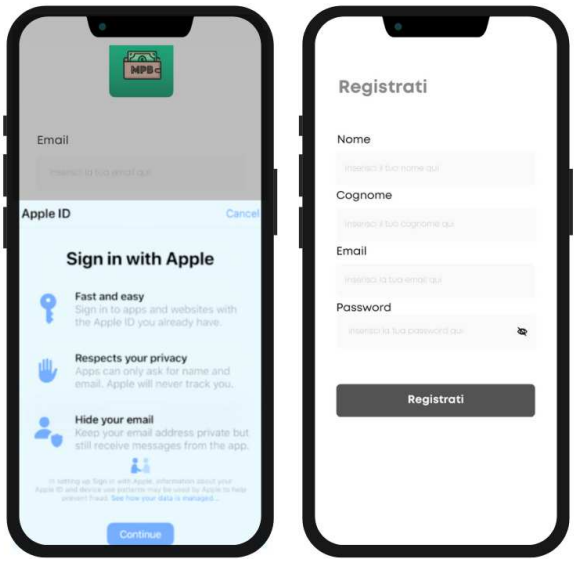

Figura 3.3: Wireframe delle opzioni di registrazione

rappresentata nella Figura 3.4, viene mostrato un breve testo che spiega all'utente come inserire le transazioni. Al di sotto del testo sono presenti le transazioni, inserite in una tabella scorrevole, aggiornate all'ultima inserita. La lista delle transazioni viene reperita dal database ed ordinata per data, dalla più recente alla meno recente. Nella parte inferiore della pagina sono disposti i pulsanti "Entrate" e "Uscite". Quando vengono premuti, l'utente accede alle due pagine d'inserimento delle transazioni, una per quelle appartenenti alla categoria delle entrate e una per le transazioni in uscita. L'ultimo elemento della pagina delle transazioni è il pulsante "Logout". Esso è posizionato nell'angolo superiore dello schermo, così da evitare l'attivazione accidentale della funzione di logout.

Approfondendo le pagine a cui si accede mediante la sezione delle transazioni, possiamo osservare come esse siano strutturate in maniera molto simile, con la distinzione presente solamente nel titolo. Una grossa porzione della schermata è dedicata alle icone delle varie categorie di transazioni. Ogni icona sarà creata in modo da rappresentare in modo intuitivo la tipologia di transazione. Al tocco delle icone, verrà presentato all'utente un pop-up

Figura 3.4: Wireframe della sezione Transazioni

banner, ovvero una finestra temporanea, all'interno della quale egli inserirà la somma della transazione.

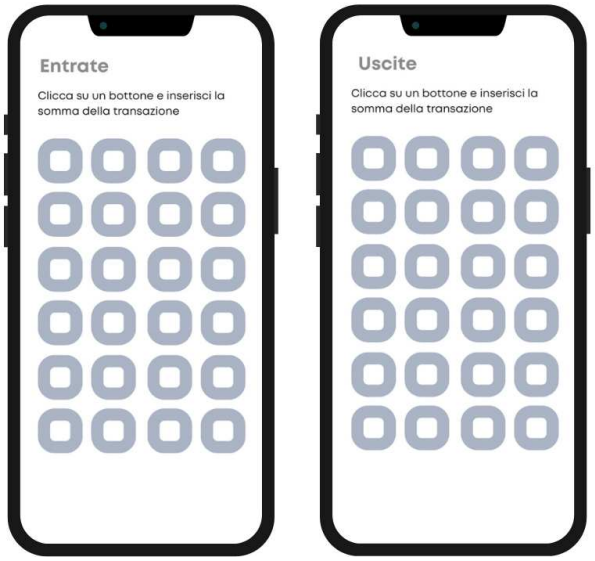

Figura 3.5: Wireframe delle sezioni Entrate e Uscite

All'interno della seconda pagina, corrispondente al secondo elemento da sinistra della navigation bar, la struttura è stata divisa allo stesso modo. In questa sezione, denominata "Statistiche", sono presenti il titolo, disposto con il medesimo stile della pagina delle transazioni, ed un breve riassunto delle funzionalità utilizzabili. Al centro dello schermo è posizionato un grafico a torta, composto dalle varie categorie di transazioni inserite dall'utente. Le categorie vengono mostrate soltanto in caso di presenza di una transazione corrispondente alla stessa categoria nel database. Oltre a questa funzionalità, viene permesso all'utente di scorrere tra vari grafici, con l'intento di visualizzare le singole macro-categorie e, persino, confrontarle. L'utilizzo di colori ad alto contrasto è stato impiegato con l'intento di evitare confusione tra categorie adiacenti; oltretutto, i colori utilizzati vengono scelti in maniera casuale. Sono, inoltre, presenti i nomi di ogni categoria uniti alla percentuale di grafico occupata dalla stessa,

così da fornire una maggiore precisione all'utente. Oltre a queste funzionalità, come nella sezione "Transazioni", è presente il tasto "logout" nell'angolo superiore dello schermo.

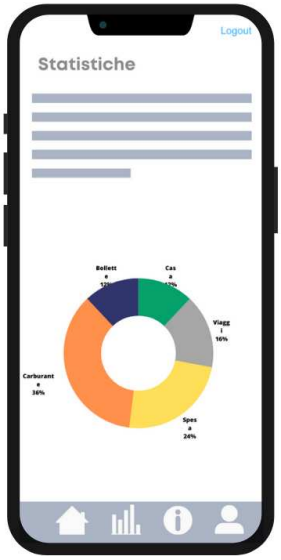

Figura 3.6: Wireframe della sezione Statistiche

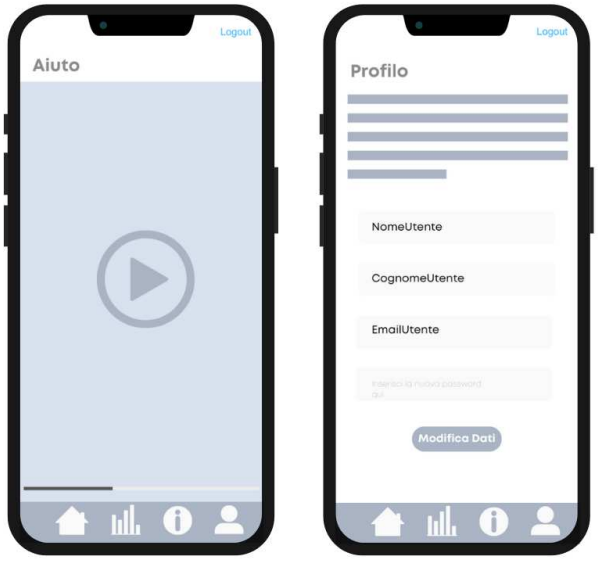

Figura 3.7: Wireframe della sezione Aiuto

Passando ora alle ultime due pagine della navigation bar (Figura 3.7), ovvero le sezioni "Aiuto" e "Profilo". Nella prima viene presentata una sezione contenente un video tutorial. L'utente potrà visualizzare un breve video, utile a mostrare ogni funzionalità dell'applicazione e come utilizzarla. Come in ogni pagina appartenente alla navigation bar, è presente il pulsante "logout". All'interno della sezione "Profilo", vengono disposti i dati dell'utente

inseriti in fase di registrazione, eccetto la password, in modo che possano essere modificati. L'utente può, quindi, toccare la casella che desidera modificare, inserire i dati aggiornati ed inviarli al database mediante la pressione del pulsante "Modifica Dati". Al di sotto del tasto è presente un'etichetta con l'obiettivo di evidenziare eventuali errori derivanti dall'inserimento del dato, come, ad esempio, la mancanza della "@" all'interno del campo email.

3.3 Progettazione del Database

L'organizzazione grafica rappresenta una parte sostanziale dello sviluppo di una buona applicazione, ma lo è al contempo la progettazione dell'organizzazione dei dati. Per garantire l'efficienza della fase di sviluppo, bisogna effettuare una serie di decisioni basate sull'utilizzo previsto, come la scelta del servizio di un sistema di raccolta dati e la struttura dei dati stessi. L'applicazione utilizza un database totalmente gestito online denominato Firebase. Esso è un servizio completo fornito dall'azienda Google in grado di gestire, elaborare ed ordinare dati di ogni genere in maniera semplice e veloce. La scelta di utilizzare il servizio di Google deriva da vari fattori, primo tra tutti la semplicità d'uso. Infatti, Firebase appartiene alla categoria di database NoSQL, ovvero, database che non utilizzano tabelle contenenti righe e colonne, ma modelli chiave-valore strutturati a nodi. Tra i vari vantaggi di Firebase, quelli di interesse maggiore risultano essere la scalabilità automatica, la facilità di organizzazione dei dati all'interno di strutture e la scrittura semplificata di query. I servizi utilizzati dall'applicazione sono i seguenti:

- *Firebase - Firestore*: un servizio NoSQL in grado di memorizzare e sincronizzare i dati in tempo reale. Questo permette di richiedere i dati utente e utilizzarli per produrre grafici in tempo reale.
- *Autenticazione*: permette di autenticare gli utenti mediante la creazione e l'utilizzo di uno Unique Identifier, ovvero un codice univoco auto-generato.
- *Salvataggio*: è una funzionalità essenziale per poter memorizzare i dati di ogni utente, come transazioni e categorie delle spese effettuate.

Servizio Cloud Firestore

Firestore risulta essere il servizio che permette di creare e gestire i dati in modo veloce e semplice. Infatti, esso permette di elaborare ed immagazzinare i dati mediante l'utilizzo di strutture come cartelle e raccolte. Ogni dato memorizzato all'interno del database è composto da una coppia chiave-valore, mentre le cartelle e le raccolte sono denominate secondo nomi e codici identificativi auto-generati. Questa metodologia di ordinamento dei dati permette al programmatore di recupare con facilità i documenti richiesti, riducendo gli errori di omonimia al minimo. Oltretutto, Firestore non distingue i dati a seconda del tipo, dando la possibilità di immagazzinare qualsiasi tipologia di dato senza incorrere in errori.

Autenticazione

L'organizzazione degli utenti risulta spesso ostica all'interno dei database classici, con problemi come l'omonimia e, di conseguenza, la necessità di distinguere ogni utente in base ad ID. Firebase, grazie al servizio "Authentication", permette di evitare errori di questa tipologia già in fase di registrazione. Difatti, tra le varie funzionalità, troviamo la possibilità di associare un codice alfanumerico univoco ad ogni utente, direttamente in fase di registrazione. Al momento della registrazione, è possibile fornire diverse possibilità di accesso all'utente, come

email e password, account Twitter, Facebook, Google e Apple. Questo offre una maggiore flessibilità all'utente, senza appesantire lo sviluppo lato database.

3.3.1 Schema del database

Lo schema del database rappresenta una descrizione logica della struttura del database utilizzato. Nella Figura 3.8 è possibile visualizzare le relazioni tra l'utente, i suoi dati e le transazioni inserite da esso. Lo schema in figura rappresenta la raccolta degli utenti, ciascuno di essi ha al suo interno i dati dell'utente stesso e una raccolta di transazioni. Questa struttura garantisce la sicurezza dei dati, oltre ad evitare di utilizzare dati appartenenti ad altri utenti. Nella parte sinistra della figura sono rappresentati tutti i dati utente, insieme alla tipologia del dato. La freccia indica la relazione tra il dato della singola transazione e l'utente. In ambito di raccolta dati, la transazione viene definita come "child", ovvero "figlio", dell'utente.

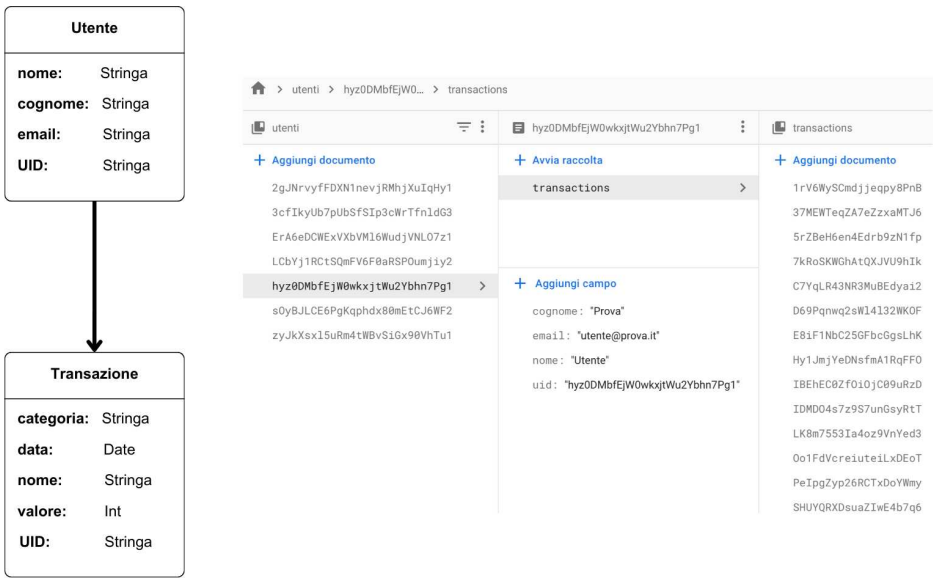

Figura 3.8: Schema del Database

3.3.2 Gestione dell'autenticazione dell'utente

La registrazione e l'autenticazione degli utenti verranno gestite mediante il servizio Firebase denominato "Authentication". In fase di registrazione, il servizio crea un documento all'interno del database, in questo caso all'interno di una raccolta di utenti. Al momento della registrazione, grazie al servizio automatizzato, viene generato un ID utente univoco. Il documento dell'utente viene salvato utilizzando l'ID sopracitato come identificativo. All'interno del documento di ogni singolo utente sono presenti quattro campi: nome, cognome, email e password. La password è inaccessibile all'amministratore del database, infatti, non viene neppure fornita la password criptata. I dati utente vengono compilati indipendentemente dal servizio utilizzato dall'utente in fase di registrazione. Per quanto riguarda la gestione dell'accesso, una volta che l'utente ha fornito le proprie credenziali di accesso, sia in forma di email e password, che mediante un servizio di terze parti come Apple, l'API di Firebase verificherà i dati mediante il metodo "signIn". In caso di credenziali verificate, Firebase fornirà un token di accesso e inizierà a gestire la sessione di autenticazione. Oltre

al metodo "signIn", è reso disponibile anche l'ogetto "Auth", esso consente di monitorare e gestire lo stato di autenticazione corrente dell'utente, rendendo così possibile la richiesta dei dati utente, l'aggiornamento e l'eliminazione. Il servizio fornisce, inoltre, la funzionalità di logout, mediante il quale l'utente può disconnettersi e ritornare alla pagina di login.

3.3.3 Gestione dei dati utente e dei dati permanenti

La gestione dei dati utente e dei dati permanenti viene effettuata mediante il servizio Cloud Firestore. I dati dei singoli utenti sono conservati all'interno di una raccolta denominata "utenti" e sono identificati attraverso l'impiego dell'ID univoco sopracitato. Per ciascun utente sono presenti i dati del profilo utente, come il nome e l'email, e una raccolta delle transazioni. La cartella "transactions" è strutturata in modo tale da ridurre la complessità del database e, soprattutto, per non incorrere in problemi di nesting, ovvero, un fenomeno dove il database in uso risulta essere inutilmente complesso, con dati contenuti in raccolte separate ed innestate una all'interno dell'altra. Per l'appunto, all'interno dei dati permanenti dell'utente, erano inizialmente presenti due raccolte, una per le transazioni positive ed una per quelle negative. Spostando la categoria della transazione all'interno dei dati delle singole transazioni, l'utilizzo dei dati risulta essere più semplice ed efficiente. La struttura di ogni singola transazione risulta essere il più semplice possibile, così da velocizzare i tempi di caricamento di tutte le transazioni all'interno dell'applicazione. Infatti, ogni transazione è composta dalla tipologia, positiva o negativa, dalla categoria di appartenenza, dalla cifra inserita ed, infine, dalla data in cui è stata effettuata. L'insieme di queste informazioni contenute in ogni singola transazione, unite ad ulteriori ID univoci, semplificano la creazione dei grafici all'interno dell'applicazione e rendono inequivocabile la richiesta di una specifica transazione.

CAPITOLO 4

Implementazione

La fase d'implementazione è un passo fondamentale nel processo di sviluppo di un'applicazione mobile. Dopo aver completato la fase di progettazione, si passa alla concretizzazione dei concetti e delle idee pianificate. In questa fase si costruisce effettivamente il prodotto, iniziando dai mockup, costruendo le funzioni che renderanno l'app funzionante, per poi giungere a un prodotto finale. Successivamente avviene la preparazione al rilascio, creando grafiche e i testi per la pubblicazione e analizzando l'utilizzo dei dati degli utenti.

4.1 Scelta delle tecnologie e degli strumenti di sviluppo

La fase di implementazione di un'applicazione mobile richiede l'utilizzo di strumenti adatti allo scopo. La scelta delle tecnologie necessita di una valutazione approfondita delle varie opzioni disponibili, basandosi sulle esigenze del progetto. L'ambiente di sviluppo integrato (IDE) deve risultare veloce e di facile utilizzo, poiché influisce enormemente sulla produttività, sulla fase di debug e sulla gestione di risorse. Le opzioni per poter sviluppare un'applicazione in Swift sono varie; tra le principali troviamo gli IDE Visual Studio Code, CodeRunner e, infine, Xcode. Ognuno di essi offre vantaggi, come la velocità di esecuzione del codice e un processo di debug semplice; la nostra scelta è ricaduta su Xcode in quanto pensato da Apple appositamente per lo sviluppo di app native iOS e, di conseguenza, altamente ottimizzato.

4.1.1 Approfondimento dell'IDE Xcode

Xcode è un ambiente di sviluppo integrato (IDE) creato da Apple per lo sviluppo di applicazioni iOS, macOS, watchOS e tvOS. L'IDE, al tempo denominato "Project Builder", fu inizialmente acquisito dall'azienda californiana nel 2003, con l'intento d'integrare nuove funzionalità e di creare un ambiente di sviluppo proprietario per applicazioni macOS. Nel 2007, con il rilascio del primo iPhone, le capacità di Xcode furono ampliate, permettendo lo sviluppo di applicazioni mobili per il sistema operativo iOS. Con il passare degli anni, Xcode è stato continuamente aggiornato con l'intento di migliorare l'esperienza di sviluppo. Tra le aggiunte fondamentali nel percorso di sviluppo di Xcode troviamo l'integrazione completa con il linguaggio di programmazione Swift, l'aggiunta di librerie di terze parti, utili a semplificare lo sviluppo di applicazioni, e l'interfaccia grafica di progettazione (Interface Builder), uno strumento essenziale per la gestione della grafica delle applicazioni. Oltre a queste funzionalità, oramai parte del DNA di Xcode, nei vari aggiornamenti a cadenza annuale, Apple ha aggiunto componenti e funzionalità che distinguono l'IDE in questione

da altri prodotti simili. Tra questi troviamo il supporto per lo sviluppo di applicazioni per nuovi prodotti come l'Apple Watch, strumenti di testing automatizzati volti a verificare la qualità delle app, per poi giungere a funzionalità come "Swift Playgrounds", ovvero un ambiente interattivo, volto a incentivare l'apprendimento della programmazione in Swift. Tra le varie funzionalità presenti nell'IDE, alcune risultano essere uniche nel mondo dello sviluppo per l'ecosistema Apple. La forte integrazione con il framework di sviluppo iOS offre, infatti, strumenti molto potenti, come il debugger, oppure l'emulatore di dispositivi iOS e iPadOS, quest'ultimo utile a testare la propria applicazione su tutta la linea di dispositivi Apple (pur non possedendone alcuno). Apple ha, quindi, lavorato e migliorato il proprio IDE, prestando attenzione alle richieste dei propri sviluppatori, ma, al contempo, è riuscita a creare un terreno fertile, volto ad introdurre le nuove generazioni alla programmazione.

4.2 Progettazione dell'interfaccia utente

La progettazione dell'interfaccia utente (UI) riveste un ruolo fondamentale nello sviluppo di un'applicazione mobili, nella fattispecie, per lo sviluppo per dispositivi iOS risulta essere di enorme importanza. Un'interfaccia utente semplice, ordinata e intuitiva, è ciò che rende l'esperienza utente (UX) ottimale, determinando molto spesso il successo di un'applicazione. Sin dal primo iPhone, Apple si è contraddistinta per l'attenzione ai dettagli, inserendo delle linee guida per il design e l'utilizzo di componenti già nelle prime fasi di vita di Xcode. Difatti, tra i vari requisiti richiesti per la pubblicazione di un'app sull'App Store, il design intuitivo risulta essere uno dei principali. La scelta della versione del sistema operativo determina la tipologia di design che verrà implementato. Apple richiede ai propri sviluppatori di pubblicare e mantenere aggiornate le proprie applicazioni all'ultima versione di iOS; Di conseguenza, lo sviluppo della UI sarà svoltO seguendo le linee guida per lo sviluppo per iOS 16.

Fase di progettazione

Lo sviluppo dell'interfaccia utente è stato gestito in maniera lineare, partendo dall'analisi delle esigenze principali degli utenti, dei comportamenti di un utente medio e delle esigenze di quest'ultimo. Ad esempio, nei wireframe mostrati nel Capitolo 3, le abilità e le capacità dell'utente medio sono state prese in considerazione in ogni particolare, con l'intento di rendere l'interfaccia semplice, intuitiva ed evitare un primo impatto negativo con l'interfaccia. Partendo da queste considerazioni, il design è, quindi, stato realizzato con l'uso di elementi semplici, contenenti pochi colori ma, soprattutto, riducendo il numero di elementi per pagina allo stretto necessario. Inoltre, l'utilizzo di una palette di colori uguale attraverso l'intera applicazione rende l'esperienza d'uso migliore, dando all'utente informazioni immediate attraverso il metodo del "color coding".

La creazione delle grafiche è stata implementata mediante l'utilizzo di "Sketch", un'applicazione di design in grado di creare file di tipologia "SVG" (Scalable Vector Graphics). I file creati mediante Sketch possiedono qualità ottime per l'utilizzo all'interno di un'app mobile; infatti, essi risultano essere di piccole dimensioni, ma, al contempo, di qualità estremamente alta, grazie alla tipologia di file utilizzata. L'utilizzo di Sketch ha permesso la creazione di componenti come i pulsanti e le caselle di testo utilizzate in fase di login e registrazione. Nella Figura 4.1 vengono mostrati i colori utilizzati attraverso tutta l'applicazione, i vari asset, come pulsanti e caselle di testo, e, infine, la grafica minimalista. Inoltre, l'interfaccia è stata sviluppata con l'intento di porre l'attenzione dell'utente dove necessario. Ad esempio, la registrazione dell'utente risulta essere un'attività che viene svolta singolarmente per ogni utente; una volta effettuata, essa diventerà una funzione non necessaria per un normale uti-

lizzo dell'applicazione; di conseguenza il pulsante per accedere alla pagina di registrazione è stato ridotto ad una linea di testo evidenziata soltanto mediante il colore blu. Questo metodo è oramai lo standard nello sviluppo web e mobile.

Figura 4.1: Splash Page - Pagina d'avvio dell'applicazione

4.3 Implementazione delle funzionalità principali

Lo sviluppo dell'applicazione inizia dalla creazione delle componenti principali. Inizialmente viene creato il repository, ovvero una raccolta con versioning dei file del progetto sulla piattaforma GitHub. Successivamente viene creato il database sulla piattaforma Firebase, per poi giungere alla vera implementazione del progetto, ovvero lo sviluppo del codice sorgente. Il progetto viene clonato dalla piattaforma GitHub all'interno dell'IDE Xcode; si procede, poi, con la creazione della struttura dei file del progetto, implementata seguendo il paradigma Model-View-Controller (MVC). L'MVC è una struttura composta da 3 sezioni, una cartella contenente i controllori, ovvero i file di codice Swift che controllano il comportamento delle pagine dell'applicazione, e altre due sezioni dedicate alla gestione dei dati e alla gestione delle animazioni, della grafica e altre componenti del progetto.

Successivamente, per la gestione dei testi utilizzati all'interno dell'app, è stato creato un file denominato `Costanti.swift`. Esso rende possibile la traduzione delle componenti testuali di tutta l'app in modo semplice e automatico, cambiando la lingua dell'applicazione in base alle impostazioni del dispositivo su cui viene avviata l'applicazione.

Swift Package Manager

Apple, come ogni azienda, fornisce una vasta quantità di librerie native all'interno della piattaforma di sviluppo. Nei primi anni di sviluppo di iOS, l'installazione di queste librerie era gestita da parte di un servizio di terzi denominato "Cocoapods". L'utilizzo di questa piattaforma ha, però, da sempre generato problemi di compatibilità ed efficienza dello

sviluppo. Proprio per questo motivo Apple ha rilasciato SPM con l'aggiormanto a Swift 5 ed Xcode 11. SPM risulta essere una soluzione ottima grazie all'aggiornamento automatico delle librerie, alla compatibilità nativa con Xcode e alla sua natura open-source, che ne garantisce un continuo e veloce sviluppo.

Creazione delle pagine dell'applicazione

La gestione della componente grafica di un'applicazione in Xcode può essere effettuata in due modi. Il primo è l'utilizzo dell'Interface Builder, ovvero, uno strumento che permette allo sviluppatore di organizzare le componenti della propria applicazione, gestendo la dimensione, la posizione, i font, le caratteristiche, e tanto altro, utilizzando un'interfaccia con menù contestuale. Il secondo metodo, oramai sempre più adottato, è SwiftUI, una libreria creata da Apple adibita a scrivere codice con l'intento di modificare il comportamento e l'aspetto delle componenti. L'utilizzo di uno dei metodi non esclude l'altro. Spesso è consigliato da Apple stessa l'utilizzo parallelo delle due tecniche per ottenere i vantaggi di entrambi i metodi.

Una componente fondamentale dello sviluppo di ogni applicazione è data dai constraint. Essi permettono all'applicazione di adattarsi a ogni dispositivo, regolando le componenti grafiche secondo dei limiti impostati dallo sviluppatore. Ad esempio, all'interno della Figura 4.1, tutti i pulsanti presenti al centro dello schermo sono automaticamente allineati al centro verticale del dispositivo, mentre la distanza tra un elemento e il successivo è dinamicamente modificata in base all'altezza dello schermo. Un buon uso dei constraint permette di evitare bug grafici su tutti i dispositivi e mantiene l'esperienza simile su tutte le tipologie di schermo, senza gestirle tutte singolarmente.

Nella Figura 4.2 è possibile osservare l'utilizzo dello strumento Interface Builder per oscurare i caratteri inseriti all'interno della casella di testo per la password. Questa opzione è selezionabile all'interno delle opzioni della casella di testo; nel caso di SwiftUI equivarrebbe a una riga di codice e, di conseguenza, a un file sorgente più complesso. Risulta, quindi, evidente come SwiftUI e Interface Builder vadano usati al contempo, a seconda della funzionalità da implementare.

Successivamente, una volta effettuato l'accesso, l'utente viene indirizzato alla pagina delle transazioni (Figura 4.3). All'interno di questa pagina vengono disposte le transazioni, ordinate per data e colorate in base alla tipologia: verde per le entrate, rosso per le uscite. Per la creazione della tabella è stata impiegata una "Table View", ovvero un elemento grafico contenente un numero di righe e colonne, volte a formare una tabella contenente informazioni. In questo caso, la generazione delle righe è dettata dal numero di elementi presenti nella raccolta `transactions` all'interno del database Firebase. Per gestire la creazione automatica delle righe e delle colonne della tabella è stato creato un file Swift denominato `TransactionTableViewCell`, con l'intento di creare una singola riga della tabella contenente due campi. L'utilizzo di una singola cella è possibile grazie all'utilizzo degli stessi campi all'interno di ogni transazione; tali esempi sono `category`, `name`, `number` e `date`. Così facendo, ogni transazione verrà inserita nella stessa tipologia di casella, generata mediante il file sopracitato.

Una volta che il numero di transazioni è stato determinato, l'applicazione legge la categoria della transazione e assegna ad essa un colore. Successivamente la transazione viene inserita nella riga ed il processo ricomincia da capo fino all'ultimo elemento della raccolta `transactions`. Nella sezione inferiore dello schermo è presente una navigation bar, ovvero una barra di navigazione gestita da un navigation controller. Il controller per gestire la navigazione (Figura 4.4) è stato creato mediante l'utilizzo di Interface Builder, aggiungendo in automatico le animazioni di transizione tra una pagina e l'altra. Per la creazione delle connessioni è, infatti, necessario soltanto collegare il pulsante o l'elemento che darà l'input per cambiare schermata alla e presentare la pagina successiva.

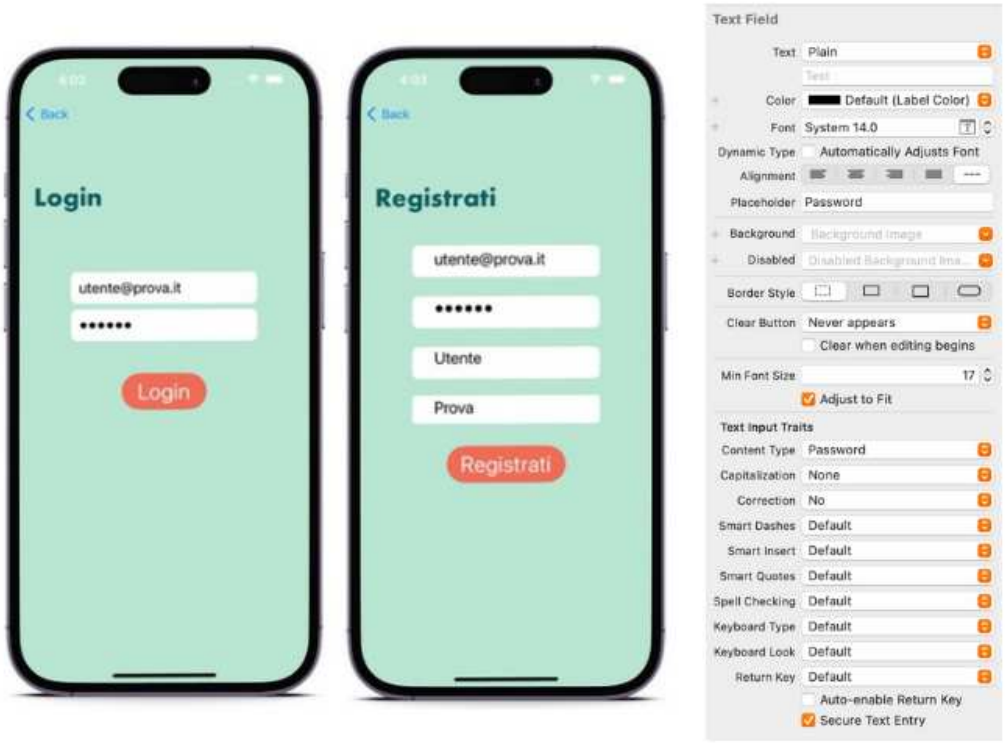

Figura 4.2: Interfaccia della Registrazione e del Login

Nella Figura 4.4 è presente il controller della navigation bar nella parte superiore dell'immagine. Esso è collegato a ogni controller delle pagine sottostanti e ne gestisce il collegamento alle icone della barra di navigazione. Il controller in questione impiega una gestione gerarchica, partendo dal controller principale in alto, per poi arrivare ai controller sottostanti, che, a loro volta, controllano le interfacce grafiche presenti nella parte inferiore dell'immagine.

Inserimento delle transazioni

L'accesso alla sezione dedicata all'inserimento delle transazioni avviene mediante uno dei due pulsanti sottostanti la tabella delle transazioni. Nella Figura 4.6 sono raffigurate le due pagine speculari adibite all'inserimento delle transazioni. Alla pressione di una delle immagini stilizzate, vengono automaticamente selezionate la categoria e la tipologia della transazione; successivamente vengono aperte un tastierino numerico e una "dialog page", ovvero un'interfaccia con testo, una casella per l'inserimento di dati e un pulsante per confermare l'inserimento. La finestra di dialogo, mostrata in Figura 4.5, viene generata mediante l'uso della componente *UIAlertController*; in questo caso d'uso, essa contiene tre valori: il titolo, ovvero "Inserisci Stipendio", il messaggio e, infine, lo stile della finestra, in questo caso sotto forma di un alert, così da obbligare l'utente a inserire i dati necessari per la creazione di una transazione.

La componente sopracitata è parte della libreria Apple UIKit, inclusa nativamente nell'IDE Xcode. Il tastierino numerico con cui l'utente interagisce durante l'inserimento della transazione viene generato includendo il punto decimale. Per far si che il punto decimale sia incluso, è necessario forzare l'inclusione dello stesso, modificando l'aspetto della tastiera

Figura 4.3: Pagine delle Transazioni

attraverso l'uso di SwiftUI. Infine, il numero decimale viene formattato in formato americano, affinché Firebase sia in grado di leggere il dato correttamente.

Visualizzazione delle statistiche

Successivamente, dopo aver inserito delle transazioni, l'utente può passare alla pagina delle statistiche. Questa sezione mantiene una grafica semplice e simile a quella precedente, con l'inserimento di tre pulsanti nella parte inferiore dello schermo e un grafico collegato a ciascuno di essi. L'header risulta speculare, come in ogni pagina dell'applicazione, così da garantire continuità nel percorso. Oltretutto, la palette di colori continua a rimanere uguale, così da rendere facilmente intuibile ogni opzione che può essere effettuata dall'utente. I grafici sono stati generati mediante l'uso di una libreria di terze parti chiamata Charts; essa permette di creare grafici di varie tipologie.

Nel caso in Figura 4.8 è stato usato un grafico a torta, in quanto risulta semplice e immediato, oltretutto, facilmente adattabile a qualsiasi dimensione di schermo in quanto non interagisce con i bordi del dispositivo.

In Figura 4.7 viene esposto il codice impiegato nel calcolo delle proporzioni di ogni categoria all'interno del grafico a torta. I dati vengono ricavati dal database, salvati in variabili temporanee, per poi essere utilizzati nel calcolo delle percentuali. Ogni categoria viene analizzata utilizzando un ciclo `for` che scorre due valori, nome e quantità, per ogni transazione. Successivamente i valori ottenuti vengono confrontati con il totale della categoria in questione e vengono moltiplicati per cento, così da ottenere la percentuale della tipologia della transazione. Le percentuali ottenute vengono poi inserite all'interno del dataset di ogni grafico a torta e i colori di ogni categoria di transazioni vengono scelti casualmente da una lista.

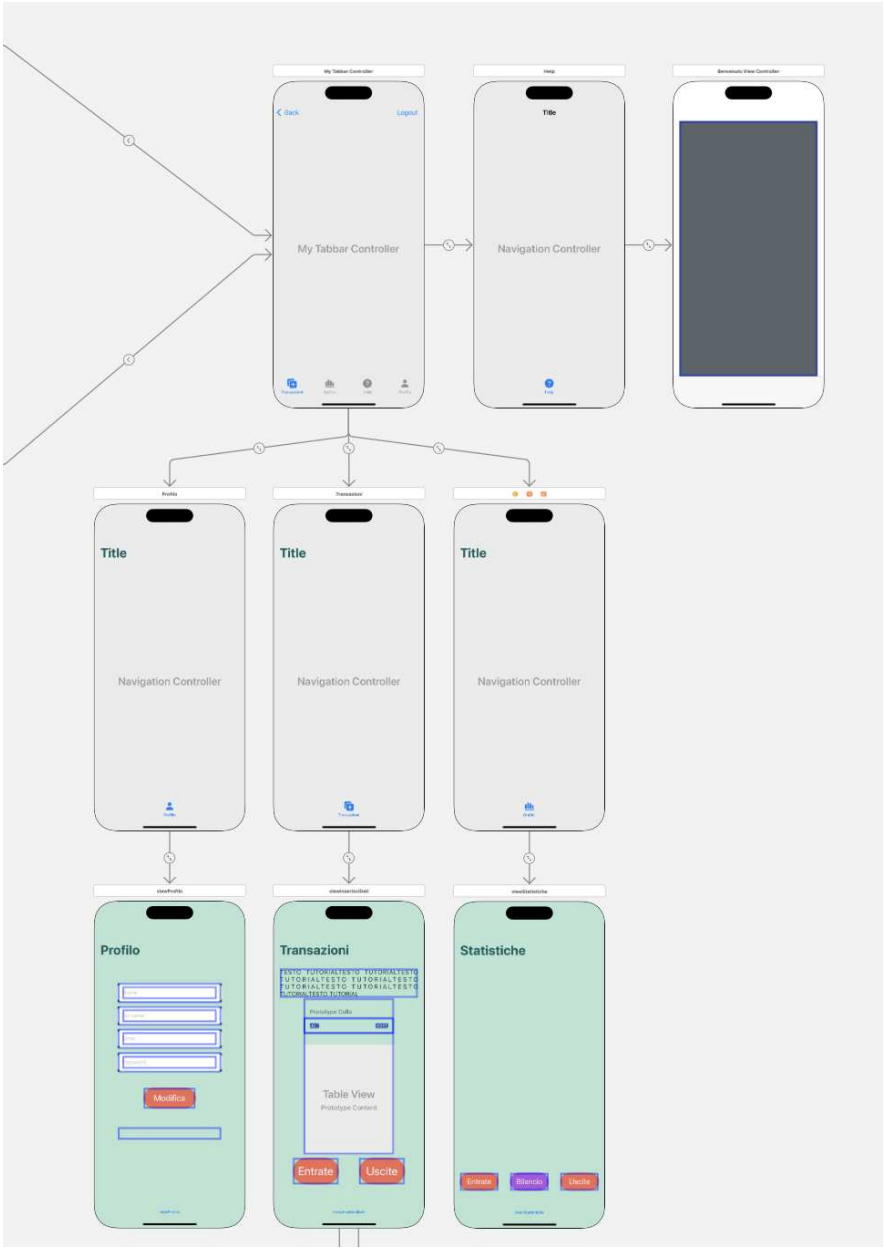

Figura 4.4: Schema del Navigation Controller - Interface Builder

Sezione aiuto

La sezione successiva, raffigurata sotto forma di codice Swift nella Figura 4.9, è la sezione "Aiuto". L'intento di questa pagina è quello di mostrare all'utente un breve video esplicativo, composto da brevi spiegazioni riguardanti ogni funzione dell'applicazione. Per inserire un video all'interno di una schermata dell'applicazione, vi sono varie soluzioni. La prima è possibile grazie all'utilizzo della libreria `AVFoundation`, che permette di inserire componenti media nelle pagine, regolando la grandezza, e altri valori riguardanti la grafica. La seconda, in utilizzo in questo caso, è l'impiego della libreria `WebKit`. La libreria sopracitata, permette di inserire un link che verrà caricato automaticamente dal dispositivo e mostrato utilizzando il framework di Safari, ovvero, il browser proprietario Apple. Nel caso in questione è stato inserito un link a un video, caricato sulla piattaforma YouTube, in formato verticale, in modo che si adatti perfettamente allo schermo del dispositivo. Per far si che il video sia di dimensio-

```
    @IBAction func buttonEntrate(_ sender: UIButton) {
        let alertController = UIAlertController(title: "Inserisci Stipendio", message: "Per favore inserisci un valore", preferredStyle:
.alert)

        alertController.addTextField { (textField : UITextField!) -> Void in
            let numberFormatter = NumberFormatter()
            numberFormatter.locale = Locale(identifier: "en_US")
            textField.keyboardType = .decimalPad
        }

        let insertAction = UIAlertAction(title: "Inserisci Stipendio", style: .default, handler: { alert -> Void in
            if let textField = alertController.textFields?.first, let text = textField.text {
                let enteredNumber = Double(text) ?? 0
                print("Numero immesso: \(enteredNumber)")

                let transactionData = [
                    "name": "wage",
                    "number": enteredNumber,
                    "data": Date(),
                    "category": "positive"
                ] as [String : Any]

                self.db.collection("utenti").document(self.user!.uid).collection("transactions").addDocument(data: transactionData) {
error in
                    if let error = error {
                        print("Errore nell'aggiunta della transazione: \(error.localizedDescription)")
                        self.labelErrore.text = error.localizedDescription
                    } else {
                        print("Transazione aggiunta con successo!")
                        self.labelErrore.text = "Transazione registrata"
                    }
                }
            }
        })
        alertController.addAction(insertAction)

        self.present(alertController, animated: true, completion: nil)
    }
```

Figura 4.5: Codice Swift per l'inserimento di nuove transazioni

Figura 4.6: Pagina per l'inserimento delle transazioni

ni uguali in ogni schermo, è stata usata un'ulteriore libreria denominata `TinyConstraints`, che permette di gestire i constraint dei singoli elementi, mediante codice Swift. Nonostante la necessità di avere una connessione attiva per visualizzare il video, al giorno d'oggi risulta particolarmente difficile trovare applicazioni che non ne fanno uso. Oltretutto, la sola azione di aver effettuato il login ci assicura che il dispositivo sia connesso a internet.

```swift
transactionsRef.getDocuments() { [weak self] querySnapshot, error in
    if let error = error {
        print("Error getting documents: \(error)")
    } else {
        guard let documents = querySnapshot?.documents else {
            print("No documents found")
            return
        }

        var positiveByNames: [String: Double] = [:]
        var negativeByNames: [String: Double] = [:]

        for document in documents {
            let data = document.data()
            let name = data["name"] as? String ?? ""
            let category = data["category"] as? String ?? ""
            let number = data["number"] as? Double ?? 0

            if category == "positive" {
                self?.positiveTotal += number
                if let currentNumber = positiveByNames[name] {
                    positiveByNames[name] = currentNumber + number
                } else {
                    positiveByNames[name] = number
                }
            } else {
                self?.negativeTotal += number
                if let currentNumber = negativeByNames[name] {
                    negativeByNames[name] = currentNumber + number
                } else {
                    negativeByNames[name] = number
                }
            }
        }

        for (name, number) in positiveByNames {
            let percentage = (number / self!.positiveTotal) * 100
            self?.positivePercentagesByNames[name] = Int(percentage)
        }

        for (name, number) in negativeByNames {
            let percentage = (number / self!.negativeTotal) * 100
            self?.negativePercentagesByNames[name] = Int(percentage)
        }

    }
}
```

Figura 4.7: Codice Swift per il calcolo delle percentuali delle categorie

Figura 4.8: Pagine per la visualizzazione delle statistiche

Sezione profilo

L'ultima sezione, accessibile mediante la navigation bar, è la pagina di modifica del profilo. La struttura grafica risulta molto simile alla pagina di registrazione; questo poiché

```swift
let webView = WKWebView()

    override func viewDidLoad() {
        super.viewDidLoad()

        view.addSubview(webView)

        guard let url = URL(string: "https://youtube.com/shorts/clnFmbUaP1Y?feature=share") else {
            return
        }

        webView.translatesAutoresizingMaskIntoConstraints = false

        let g = view.safeAreaLayoutGuide
        NSLayoutConstraint.activate([

            webView.topAnchor.constraint(equalTo: g.topAnchor, constant: -50.0),
            webView.leadingAnchor.constraint(equalTo: g.leadingAnchor, constant: 0.0),
            webView.trailingAnchor.constraint(equalTo: g.trailingAnchor, constant: 0.0),
            webView.bottomAnchor.constraint(equalTo: g.bottomAnchor, constant: 0.0),

        ])
        webView.load(URLRequest(url: url))

    }
```

Figura 4.9: Codice Swift della sezione "Aiuto"

vengono mostrati i dati utente, tranne la password, cosicché l'utente possa modificarli. Difatti, la differenza tra le due sezioni sopra citate, risiede nelle richieste CRUD. La pagina di registrazione utilizza soltanto una tipologia di operazione, ovvero, quella di creazione di dati, mentre la sezione del profilo, dapprima legge in dati e li dispone nelle caselle di testo e, successivamente a una eventuale modifica, invia i nuovi dati con una richiesta di "update", ovvero un aggiornamento dei dati utente. I dati aggiornabili sono: nome, cognome, email e password.

4.3.1 Implementazione delle API esterne e dei servizi di terze parti

L'utilizzo di API esterne è spesso l'unico metodo per ottenere soluzioni a problemi comuni o per implementare funzionalità non disponibili nelle librerie fornite con il linguaggio in uso. L'API che risulta essere fondamentale per il funzionamento corretto dell'applicazione è Firebase. Essa è composta da vari servizi come "Google Utilities", "Google Data Transport" e "Google App Measurments". L'installazione dell'API in questione avviene allo stesso modo di quella relativa alle librerie semplici; infatti, mediante l'utilizzo di SPM (Swift Package Manager) si individua il servizio "Firebase" e lo si installa all'interno del progetto. Successivamente vanno inserite le credenziali del proprio account Google con cui è stato registrato il database e, infine, la chiave per l'utilizzo dell'API. Questa chiave è un codice alfanumerico generato da Google che permette al database di riconoscere e gestire le richieste da parte dell'applicazione. L'interazione tra l'app e il servizio Firebase è quindi possibile grazie a questo codice che, una volta riconosciuto, autorizza l'utente a interagire con i dati presenti sul database. Le componenti aggiuntive dell'API elencate poco fa gestiscono il protocollo utilizzato per il trasferimento dei dati e raccolgono le statistiche sull'utilizzo dell'applicazione. Nel caso di questa applicazione, il secondo servizio risulta superfluo e, di conseguenza, non è stato utilizzato, ciò ci ha consentito di alleggerire le risorse utilizzate dall'applicazione.

Librerie di terze parti

Le librerie di terze parti utilizzate nel progetto hanno principalmente funzioni di carattere grafico. La prima utilizzata, chiamata `GhostTypewriter` risulta essere la più semplice. Essa permette di animare qualsiasi testo all'interno dell'applicazione, così da imitare l'effetto di una macchina da scrivere che, in maniera costante, aggiunge caratteri per formare una frase. Nel caso dell'applicazione, è stata utilizzata per la presentazione del motto dell'applicazione all'interno della splash page, raffigurata nella Figura 4.1. L'aggiunta di funzioni come questa forniscono all'utente una sensazione di maggiore cura dei dettagli da parte dello sviluppatore.

Successivamente, per la gestione della tastiera, è stata implementata una libreria denominata `IQKeyBoardManagerSwift`. Essa permette di gestire in maniera autonoma la grafica dell'applicazione quando la tastiera è in uso. Ad esempio, nel momento in cui l'utente decide di immettere del testo all'interno di una casella posizionata particolarmente in basso nello scheermo, la tastiera viene presentata senza spostare la grafica in alto, andando così a coprire la porzione di schermo dove apparirà il testo inserito. La libreria in questione, in automatico, sposta la grafica fino alla base dell'elemento interessato, andando così a risolvere un problema d'uso molto comune. In inglese, servizi come questo, vengono chiamati "quality of life updates", ovvero, aggiornamenti per migliorare la qualità di vita (d'uso) dell'utente.

Le ultime due librerie, `TinyConstraints` e `Charts`, sono state utilizzate insieme per la gestione dei grafici nella sezione delle statistiche. La libreria `TinyConstraints`, permette allo sviluppatore di gestire elementi grafici in totale autonomia, fornendo dei limiti da rispettare mediante codice Swift. Nel codice sottostante, utilizzato per la gestione dei grafici, è possibile osservare l'uso di metodi forniti dalla libreria.

```
pieChartView.centerInSuperview()
pieChartView.width(to: view)
pieChartView.heightToWidth(of: view)
```

La prima riga di codice permette di ancorare l'elemento in questione al centro dello schermo, qualsiasi esso sia, la seconda riga associa la larghezza del grafico a quella della pagina corrente e, di conseguenza, allo schermo del dispositivo; infine, l'ultima riga indica che l'altezza dell'elemento deve essere proporzionale alla sua larghezza. Per finire, la libreria `Charts` è stata aggiunta per creare, gestire e inserire dati all'interno di grafici. Essa permette di utilizzare grafici di varie tipologie, come grafici a torta, a barre e altri.

La creazione di un grafico (Figura 4.10) avviene in due momenti; inizialmente è necessario creare un dataset per fornire i dati nel formato richiesto dal grafico; successivamente, avviene la modifica dell'aspetto del grafico. Approfondendo la creazione del dataset, esso varia a seconda del grafico in uso; ad esempio, nel caso del grafico a torta, bisogna fornire i dati e dividerli precedentemente in percentuali. In altri casi, come per i grafici a barre, è necessario fornire solo il valore dell'elemento, da affiancare ad altri valori. Per quanto riguarda l'aspetto del grafico, nel caso del grafico delle statistiche, sono stati aggiunti elementi di contorno, come il bordo del grafico, la legenda, i valori di ogni porzione di grafico e l'animazione d'ingrandimento della porzione selezionata.

4.3.2 Implementazione delle funzionalità di autenticazione

Le funzionalità di autenticazione vengono gestite mediante l'API di Firebase descritta in precedenza. L'implementazione dell'API all'interno del progetto avviene inizialmente connettendo l'applicazione al database. Per far ciò, è necessario inizializzare il database all'interno del file `AppDelegate.swift`, spesso riferito come il cuore di ogni applicazione, dove vengono gestiti protocolli e funzioni principali. Nel codice sottostante è evidente

```
var entries : [PieChartDataEntry] = Array()

for (name, number) in positivePercentagesByNames {
    entries.append(PieChartDataEntry(value: Double(number), label: name))
}

let dataSet = PieChartDataSet(entries: entries, label : "")

let set1 = PieChartDataSet(entries: entries, label: "")

pieChartView.holeRadiusPercent = 0.40
set1.sliceSpace = 5

let a_pie_color = UIColor.blue
let b_pie_color = UIColor.red
let c_pie_color = UIColor.green
let d_pie_color = UIColor.gray
let e_pie_color = UIColor.magenta
let f_pie_color = UIColor.orange

let a_text_color = UIColor.green
let b_text_color = UIColor.white

set1.colors = [a_pie_color, b_pie_color, c_pie_color, d_pie_color, e_pie_color,f_pie_color]
set1.valueColors =  [b_text_color, b_text_color, b_text_color, b_text_color]
set1.entryLabelColor = b_text_color
set1.drawValuesEnabled = true
pieChartView.data = PieChartData(dataSet: set1)

pieChartView.rotationAngle = 0
pieChartView.rotationEnabled = false
pieChartView.drawEntryLabelsEnabled = false
pieChartView.drawSlicesUnderHoleEnabled = true
pieChartView.drawEntryLabelsEnabled = false
pieChartView.usePercentValuesEnabled = true
pieChartView.legend.enabled = true
pieChartView.holeColor = self.UIColorFromRGB(0xC2E4D4)

pieChartView.backgroundColor = self.UIColorFromRGB(0xC2E4D4)

let titleChart = NSAttributedString(string: "Entrate", attributes: nil)
pieChartView.centerAttributedText = titleChart


let data = PieChartData(dataSet: set1)
pieChartView.data = data
```

Figura 4.10: Codice Swift per la generazione del grafico a torta delle entrate

la semplicità d'uso di Firebase, dove la configurazione e l'inizializzazione del database all'interno del progetto vengono effettuate soltanto in due righe di codice.

```
FirebaseApp.configure()
let db = Firestore.firestore()
```

Login e Registrazione

All'avvio dell'applicazione, l'utente viene portato alla splash page, dove può decidere se effettuare il login mediante il metodo classico, usare il servizio `Sign in with Apple` oppure registrarsi fornendo i dati manualmente. Firebase Authentication è stata utilizzata

per implementare la funzionalità di autenticazione nell'applicazione. Questa componente è stata fondamentale per garantire una corretta gestione dell'accesso degli utenti e la sicurezza dei dati degli stessi. L'aggiunta di `Sign in with Apple` fornisce agli utenti una sensazione di maggiore cura dei dettagli da parte dello sviluppatore, in quanto è possibile garantire un'esperienza utente fluida e sicura durante il processo di accesso all'applicazione. Approfondendo la componente del codice, nella Figura 4.11 è possibile osservare il codice per la gestione della fase di login. L'email e la password inserite dall'utente vengono passate al metodo `Auth.auth()signIn()` e confrontate con i dati utente presenti nel database. Una volta confrontati, se corretti, viene effettuato l'accesso, altrimenti viene descritto l'errore all'utente mediante del testo che apparirà sotto al pulsante "login".

Nella Figura 4.12 troviamo, invece, il codice per la gestione della registrazione. Confrontando questo codice con quello per la fase di login, possiamo notare come la differenza maggiore risieda nel metodo utilizzato, ovvero `Auth.auth().createUser()`, e nel salvataggio dei dati dell'utente all'interno di una raccolta denominata "utenti". Inoltre, gli utenti vengono indicizzati mediante il loro UID (user identification), come emerge dalla riga di codice:

```
db.collection("utenti").document(user!.uid).setData(userData)
```

```
if let email = emailTextField.text, let password = passwordTextField.text {
            Auth.auth().signIn(withEmail: email, password: password) { authResult, error in
                if let e = error {
                    self.erroreLogin.text = e.localizedDescription
                } else {
                    self.performSegue(withIdentifier: "loginAWelcome", sender:self)
                }
            }
        }
```

Figura 4.11: Codice Swift per la gestione del login

4.3.3 Gestione dei dati e delle operazioni CRUD

La gestione dei dati è interamente implementata mediante Firebase, effettuando le richieste CRUD in Swift. La creazione di un utente inizia durante la fase di registrazione, quando l'utente effettua una richiesta di `create` usando il pulsante "registrati". Come descritto nel paragrafo precedente, al tocco del pulsante, viene creato un documento contenuto all'interno della raccolta `utenti`, indicizzato mediante il codice UID. All'interno del documento sono presenti quattro coppie chiave-valore, ovvero: nome, cognome, email e password. Oltre ai dati, vi è anche una raccolta denominata `transactions` ma, a differenza della raccolta utenti, essa viene creata nel momento in cui l'utente registra la prima transazione. Nella Figura 4.12 si possono osservare le righe di codice necessarie per effettuare una richiesta di tipologia `create`. Analizzando più nello specifico il codice citato, si nota come inizialmente viene creata una struttura dei dati utente associando questi ultimi ad una variabile denominata `userData`; necessariamente viene referenziato il database e, infine, si procede con la richiesta di creare una raccolta `utenti` con i dati contenuti all'interno della variabile `userData`.

```swift
let erroreNome_Cognome = validazioneCampi()

if erroreNome_Cognome != nil {
    mostraErrore(erroreNome_Cognome!)
} else if let email = emailTextField.text, let password = passwordTextField.text {
        Auth.auth().createUser(withEmail: email, password: password) { authResult, error in
            if let e = error { // Uso di nuovo optional binding per errore
                self.labelErrore.text = e.localizedDescription
            } else {
                let nome = self.nomeTextField.text!.trimmingCharacters(in: .whitespacesAndNewlines)
                let cognome = self.cognomeTextField.text!.trimmingCharacters(in: .whitespacesAndNewlines)


                let db = Firestore.firestore()

                let user = Auth.auth().currentUser
                let userData = [
                    "nome": nome,
                    "cognome": cognome,
                    "email": user?.email ?? "",
                    "uid": user?.uid ?? ""
                ]
                db.collection("utenti").document(user!.uid).setData(userData) { err in
                    if let err = err {
                        print("Errore durante l'aggiunta del documento: \(err)")
                    } else {
                        print("Documento aggiunto con successo!")
                    }
                }
                self.performSegue(withIdentifier: "registratiAWelcome", sender:self)
            }
        }
    }
```

Figura 4.12: Codice Swift per la gestione della registrazione

Inserimento di una transazione

L'operazione d'inserimento di una transazione avviene, come detto in precedenza, nella pagina presentata all'utente immediatamente dopo aver effettuato l'accesso. Nella Figura 4.13, sono raffigurate le righe di codice per la creazione della transazione. Inizialmente, viene verificato il numero inserito utilizzando l'optional binding. Successivamente, si procede con la creazione di una struttura dati per la transazione, denominata `transactionData`. In questo caso, essendo il codice riferito a una transazione positiva, nella coppia chiave-valore `category` troviamo `positive`. Infine, viene referenziato il database `db`, indicando a quest'ultimo di aggiungere il documento `transactionData` alla raccolta `transactions`, così da registrare una transazione composta da quattro coppie di valori: nome, cifra, data e categoria.

Visualizzazione delle statistiche e dei dati utente

La visualizzazione delle transazioni inserite avviene mediante dei grafici, raffigurati precedentemente nella Figura 4.8. L'operazione di richiesta dei dati è denominata `read` e si svolge come raffigurato nella Figura 4.14. Inizialmente, dopo aver identificato il percorso da effettuare per ottenere il dato all'interno del database, si procede al salvataggio dello stesso nella variabile `transactionsRef`, per poi essere utilizzato nella richiesta dei dati raffigurata nella Figura 4.15. La richiesta dei dati viene effettuata utilizzando la richiesta `querySnapshot`, ma senza applicazione di filtri, così da selezionare tutte le transazioni presenti all'interno della raccolta. I dati vengono poi divisi in due liste, la prima contenente le transazioni positive, e la seconda quelle negative. Infine, vengono utilizzati una serie di

```swift
if let textField = alertController.textFields?.first, let text = textField.text {
    // Salva il numero immesso nella variabile
    let enteredNumber = Double(text) ?? 0 // se non è un numero valido lo salva a 0
    print("Numero immesso: \(enteredNumber)")

    // Metto il numero nelle transazioni nel DB

    let transactionData = [
        "name": "wage",
        "number": enteredNumber,
        "data": Date(),
        "category": "positive"
    ] as [String : Any]

    self.db.collection("utenti").document(self.user!.uid).collection("transactions").addDocument(data: transactionData) { error in
        if let error = error {
            print("Errore nell'aggiunta della transazione: \(error.localizedDescription)")
            self.labelErrore.text = error.localizedDescription
        } else {
            print("Transazione aggiunta con successo!")
            self.labelErrore.text = "Transazione registrata"
        }
    }

}
```

Figura 4.13: Codice Swift relativo alla creazione di una transazione

`if-else`, usando la pratica del "nesting", in modo da contare il valore totale dei positivi e dei negativi, che verrà, poi, utilizzato nel grafico del bilancio (Figura 4.8).

```swift
let uid = Auth.auth().currentUser?.uid ?? ""
let transactionsRef = db.collection("utenti").document(uid).collection("transactions")
```

Figura 4.14: Codice Swift per memorizzare il percorso delle transazioni all'interno del database

Passando ora ai dati utente, su di essi vengono svolte due operazioni CRUD, inizialmente una di "read" e, successivamente, in caso modifica dei dati utente, l'operazione di "update". Nella Figura 4.16 è mostrato il codice per la richiesta dei dati dell'utente. Una volta letti, i dati vengono inseriti nella pagina di visualizzazione del profilo, dove l'utente può decidere se modificarli. In caso di modifica, ad esempio della sola email, l'applicazione invierà una richiesta di "update" al database mediante il metodo `updateEmail`. Esso sovrascrive il dato precedente, sostituendolo con il nuovo dato, andando, così, ad aggiornare l'informazione.

4.4 Gestione degli errori e delle eccezioni

La gestione degli errori è un aspetto fondamentale dello sviluppo di un'applicazione mobile. Swift, mediante l'utilizzo degli optional, permette di gestire gli errori in maniera semplice. Come spiegato nei capitoli precedenti, grazie agli optional si può evitare un errore in caso di dato mancante o errato; è, inoltre, possibile sostituire il valore ottenuto con una valore di default. In aggiunta all'uso dell'optional binding, Swift permette l'utilizzo del costrutto `try-catch`, mediante il quale è possibile gestire molteplici tipologie di errore allo stesso istante. Nella Figura 4.16, attraverso l'impiego degli optional, è stato possibile utilizzare un'istruzione condizionale `if` per la gestione degli errori. Il codice in figura dapprima richiede l'aggiornamento dei dati, successivamente impiega il metodo `localizedDescription` per ottenere la descrizione dell'eventuale errore. In caso non vi sia un errore, verrà presentato un testo, mediante una `label`, per indicare che l'aggiornamento ha avuto successo. L'applicazione sviluppata non necessita dell'utilizzo delle istruzioni `try-catch` in quanto i possibili

```
transactionsRef.getDocuments() { [weak self] querySnapshot, error in
            if let error = error {
                print("Error getting documents: \(error)")
            } else {
                guard let documents = querySnapshot?.documents else {
                    print("No documents found")
                    return
                }

                var positiveByNames: [String: Double] = [:]
                var negativeByNames: [String: Double] = [:]

                for document in documents {
                    let data = document.data()
                    let name = data["name"] as? String ?? ""
                    let category = data["category"] as? String ?? ""
                    let number = data["number"] as? Double ?? 0

                    if category == "positive" {
                        self?.positiveTotal += number
                        if let currentNumber = positiveByNames[name] {
                            positiveByNames[name] = currentNumber + number
                        } else {
                            positiveByNames[name] = number
                        }
                    } else {
                        self?.negativeTotal += number
                        if let currentNumber = negativeByNames[name] {
                            negativeByNames[name] = currentNumber + number
                        } else {
                            negativeByNames[name] = number
                        }
                    }
                }
```

Figura 4.15: Codice Swift per leggere tutte le transazioni appartenenti all'utente

errori vengono generati dal database Firebase e, di conseguenza, vengono identificati come testo inviato dal database.

4.4.1 Test e debug

Il processo di testing e debug di un'applicazione mobile risulta essere uno dei punti chiave per un buon funzionamento dell'applicazione e dell'esperienza utente. Il processo di test si svolge seguendo una lista di verifiche da effettuare, come la verifica della logica dell'app oppure l'esecuzione dell'app su molteplici dispositivi, così da assicurare il corretto funzionamento su schermi di dimensioni diverse.

```
        db.collection("utenti").document(userID).updateData([
            "nome": textFieldNome.text as Any,
            "cognome": textFieldCognome.text as Any,
            "email": textFieldEmail.text as Any
        ])

        let utenteCorrente = Auth.auth().currentUser

        utenteCorrente?.updateEmail(to: textFieldEmail.text!) { erroreEmail
in          if let erroreEmail = erroreEmail?.localizedDescription {
                self.labelErrore.text = erroreEmail
            } else {
                UIView.animate(withDuration: 3) {
                    self.labelErrore.text = K.Fbase.datiAggiornati
                    self.labelErrore.alpha = 0
                }
                print("Email account aggiornata")
            }
        }
```

Figura 4.16: Codice Swift per leggere tutte le transazioni appartenenti all'utente

Test

Il processo di testing inizia dalla verifica della logica di un'app. In questo caso il processo si è svolto controllando gli algoritmi e verificando il corretto funzionamento della gestione degli errori, inserendo valori errati e assicurandosi che gli errori descritti dall'applicazione risultino corretti. Inoltre, la gestione degli errori implica anche il blocco del codice successivo; ad esempio, nella fase di registrazione, in caso di una email senza "@", oppure di una password che non rispetta i requisiti, l'utente deve visionare l'errore corretto e non deve essere reindirizzato alla pagina successiva. Successivamente è stata svolta una verifica dei constraint, così da assicurare un corretto funzionamento su tutti gli schermi dei dispositivi compatibili con l'applicazione. Per far ciò, Apple fornisce un emulatore per ogni dispositivo attualmente aggiornato all'ultima versione di iOS, con cui è possibile verificare il funzionamento corretto dell'applicazione. Nel caso di questa applicazione, l'orientamento è stato impostato prettamente a quello verticale. Questa fase di testing è stata applicata anche a dispositivi reali, così da determinare il corretto funzionamento delle componenti in risposta al tocco impreciso di un polpastrello.

Debug

Xcode fornisce vari strumenti utili per velocizzare la fase di debug. Uno tra questi è l'utilizzo di "breakpoint" (punti d'interruzione), che permettono di bloccare l'esecuzione del codice da una determinata riga in poi, così da poter identificare il blocco di codice non funzionante. Ad esempio, durante lo sviluppo della sezione delle statistiche, è stato riscontrato un problema nella generazione dei grafici. Utilizzando i breakpoint è stato possibile analizzare le singole componenti adibite al funzionamento dei grafici e identificare l'errore; in questo caso si trattava di un errore nell'inserimento dei dati generati dal database. Oltre ai breakpoint, l'IDE Xcode offre ulteriori strumenti per il debug durante l'esecuzione, come la gestione della memoria e la gestione dell'esecuzione dell'applicazione utilizzando i comandi "Next" e

"Step into". Essi aiutano a individuare gli errori eseguendo le componenti delle applicazioni passo dopo passo, analizzando, così, in modo approfondito ogni variabile o componente utilizzata. A completare questa serie di strumenti è, infine, il pannello per la gestione delle risorse. Questo strumento, raffigurato nella Figura 4.17, permette di visualizzare l'utilizzo della CPU (Central Processing Unit - Processore), della RAM (Random Access Memory - Memoria Temporanea), del disco e, infine, l'utilizzo di internet. Nella figura si può notare come l'utilizzo della CPU sia inizialmente intenso nella fase di avvio dell'applicazione, al contempo vengono caricate in memoria tutte le componenti grafiche dell'applicazione, così da essere utilizzabili istantaneamente. Infine, nel grafico denominato "Network" si può notare un picco derivante dalla richiesta di login; ciò è dovuto al fatto che l'applicazione ha richiesto e ottenuto una verifica dei dati dell'utente.

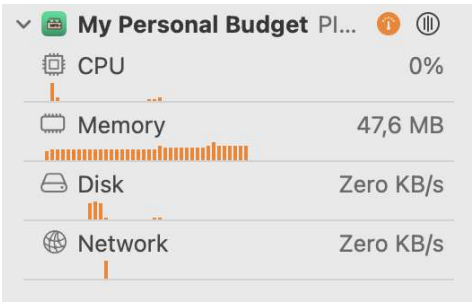

Figura 4.17: Utilizzo delle risorse hardware durante l'esecuzione in Xcode

4.5 Ottimizzazione delle prestazioni e delle risorse

Il processo di ottimizzazione di un'app in Swift è fondamentale per aumentarne le prestazioni e rendere l'esperienza utente fluida e ottimale. Uno dei primi passi da intraprendere per il miglioramento dell'efficienza è quello di ridurre le operazioni di calcolo ripetitive. Risulta, quindi, necessario individuare codice ridondante e ricorsivo all'interno del progetto. La complessità computazionale di un algoritmo è ciò che va analizzato con lo scopo di ridurla il più possibile. Come esempio, l'utilizzo di comandi `if` interni ad altri comandi `if` aumenta enormemente la complessità; a essi, quindi, è da preferirsi un comando di tipo `switch`. In questo modo, la complessità dei calcoli svolti dal dispositivo si riduce.

Nel caso di dispositivi mobili come l'iPhone, nonostante l'aumento della capacità di calcolo negli ultimi anni, è sempre consigliato ridurre al minimo il codice superfluo, così da risparmiare anche l'uso della batteria. Tra le varie ottimizzazioni effettuate, la più comune risulta essere l'uso di funzioni per lo svolgimento di attività ripetitive.

All'interno del progetto sono, quindi, state sostituite porzioni di codice uguale con funzioni che vengono chiamate solamente quando necessarie, evitando, così, l'esecuzione di codice non inerente al processo in atto. L'ottimizzazione appena descritta è stata effettuata molto velocemente grazie all'IDE. Infatti, Xcode offre lo strumento del "profiling", ovvero, la possibilità di simulare l'utilizzo dell'applicazione con l'intento di identificare i punti critici del prodotto, così da poter intervenire su di essi e ridurre le criticità dell'app.

Oltre a un'analisi approfondita delle prestazioni del proprio codice, è necessaria anche un'analisi delle prestazioni delle librerie di terze parti e delle API. Per far ciò è possibile utilizzare strumenti a pagamento specificamente creati per questo; in alternativa, è possibile testare manualmente l'efficienza, verificando i tempi di risposta con connessione a rete mobile o Wi-Fi e utilizzando dispositivi diversi.

4.6 Pianificazione del rilascio

I passi intrapresi precedentemente contribuiscono a portare l'applicazione a uno stato ottimale per il rilascio. In aggiunta a essi, per rispettare le linee guida imposte da Apple, è necessario eseguire delle modifiche finali unite a della documentazione riguardante l'utilizzo dei dati degli utenti. Apple impone agli sviluppatori di utilizzare il meno possibile le richieste di autorizzazione, specialmente se non necessarie. Inoltre, è necessario redigere un documento che attesti le motivazioni per cui si richiede l'utilizzo dei dati degli utenti e dei propri dispositivi, come si intende utilizzare e conservare questi dati e, infine, come si intende gestire i dati. Tutto ciò viene effettuato per garantire la privacy degli utenti e delle loro abitudini. Il passo successivo è, quindi, quello di identificare le richieste di autorizzazione e discernere quelle necessarie da quelle superflue. Una buona pianificazione al rilascio include, anche, la localizzazione della propria app, ovvero, la pianificazione di dove l'applicazione verrà resa disponibile e la traduzione dei suoi contenuti nella lingua locale, così da rendere l'esperienza simile in ogni paese.

Successivamente, è consigliabile verificare il corretto funzionamento dell'applicazione su ogni dispositivo per cui verrà rilasciata. Questo eviterà problemi di compatibilità e, di conseguenza, aumenterà la possibilità che l'applicazione venga pubblicata sull'App Store. Inoltre, è necessario preparare la descrizione dell'applicazione, unita alla grafica di presentazione per la pagina di download della propria. Essa deve risultare accurata e coinvolgente, deve elencare le caratteristiche della propria applicazione e l'utilità che essa genera.

Infine, per prepararsi all'invio di tutta la documentazione compresa di app ad Apple, è necessario preparare il pacchetto di distribuzione, ovvero una raccolta di tutto ciò che concerne l'applicazione, unito all'applicazione stessa. Se Apple riterrà soddisfatti i requisiti da essa imposti, l'applicazione verrà resa pubblica e sarà, quindi, rilasciata sull'App Store.

CAPITOLO 5

Manuale Utente

In questo capitolo verrà presentata l'applicazione, descrivendo i suoi obiettivi e i suoi punti di forza. Successivamente verranno descritte le sue funzionalità principali, iniziando dalla registrazione e dalla gestione dei dati utente, per poi proseguire con una descrizione dettagliata della struttura dell'applicazione. Infine, verranno mostrati i problemi comuni che potrebbero comparire durante l'uso e si vedrà come questi possono essere risolti.

5.1 Presentazione dell'app e dei suoi obiettivi

L'applicazione My Personal Budget è un moderno portafoglio digitale in grado di gestire le transazioni di un utente e fornire ulteriori informazioni. L'applicazione è stata creata utilizzando tecnologia Swift e seguendo i paradigmi del design imposti da Apple. La progettazione è stata svolta perseguendo gli obiettivi imposti da un'attenta analisi dei requisiti, dando rilevanza alla semplicità d'uso e a un design minimalista. Gli obiettivi principali dell'app sono la digitalizzazione di un processo comune, ovvero la gestione delle proprie spese, la velocizzazione dell'inserimento delle proprie transazioni e l'utilizzo di grafici per migliorare la percezione delle transazioni sostenute. Infatti, le due funzionalità principali dell'applicazione risultano essere l'inserimento di transazioni in entrata e uscita, unite all'utilizzo di grafici per fornire una sintesi dettagliata delle spese. L'applicazione si presenta, quindi, come un valido sostituto dei metodi classici per gestire le proprie transazioni, fornendo strumenti adatti a migliorare le proprie abitudini e, ad avere una maggiore percezione delle spese sostenute.

5.2 Registrazione e gestione del profilo utente

La sezione dedicata alla registrazione è accessibile dalla pagina mostrata all'avvio dell'applicazione, oppure mediante l'utilizzo del servizio "Sign in with Apple". Per accedere a questa pagina, come mostrato nella prima schermata a sinistra in Figura 5.1, è necessario premere sulla scritta blu al di sotto del pulsante "Sign in with Apple". Successivamente, l'utente verrà indirizzato alla pagina di registrazione. I valori che l'utente dovrà inserire sono l'email, il nome e il cognome e, infine, la password. Ognuno di questi deve rispettare delle specifiche impostate dal servizio Firebase; altrimenti, non verrà permessa la registrazione. L'email dovrà contenere il carattere "@", il nome e il cognome non dovranno contenere numeri o simboli e la password dovrà essere composta da un minimo di sei caratteri. Nel caso in cui l'utente inserisca dei valori errati, apparirà una scritta sottostante al pulsante "Registrati", indicante

l'errore da correggere. Nel caso l'utente scelga di procedere con una registrazione automatica utilizzando il sevizio "Sign in with Apple", i dati verranno compilati autonomamente dal servizio, per poi essere autorizzati usando il metodo di sicurezza biometrico adottato dal dispositivo. Una volta effettuata la registrazione, l'utente può accedere utilizzando la pagina di login, oppure, utilizzando nuovamente il servizio Apple che fornisce il riconoscimento degli utenti registrati e permette, quindi, di accedere per mezzo del sensore biometrico del dispositivo.

Figura 5.1: In ordine da sinistra: Splash Page, Login, Registrazione

5.3 Navigazione e struttura dell'applicazione

L'applicazione è strutturata in maniera lineare; una volta effettuato l'accesso, l'utente viene indirizzato alla pagina iniziale dell'applicazione, ovvero la sezione per l'inserimento e la visualizzazione delle transazioni. Una struttura lineare di questo genere viene gestita mediante una navigation bar che presenta al suo interno quattro opzioni selezionabili. Nella Figura 5.2 è mostrata la pagina delle transazioni e, nella parte inferiore dello schermo, la barra di navigazione.

L'applicazione ha, quindi, una struttura lineare a livelli: il livello iniziale, composto dalla registrazione e dal login, il secondo livello, composto dalle sezioni della navigation bar e, infine, il terzo livello, composto dalla sezione per l'inserimento delle transazioni. Utilizzando la barra di navigazione, l'utente può spostarsi tra le quattro sezioni principali dell'applicazione. Inoltre, in ogni sezione, è presente un pulsante di "logout", così da poter effettuare il logout dal proprio account in modo semplice. Alla pressione del pulsante, l'utente viene riportato alla splash page iniziale.

Figura 5.2: Barra di navigazione utilizzata per spostarsi tra le pagine dell'app

5.3.1 Utilizzo delle funzionalità principali

Le funzionalità principali dell'applicazione risiedono nel secondo livello sopra descritto. L'accesso a ogni sezione è, quindi, possibile grazie alla navigation bar raffigurata nella Figura 5.2. Partendo dalla pagina iniziale, viene presentata una lista delle transazioni, ordinate per data d'inserimento. Le righe sono colorate di rosso o verde, rispettivamente per le transazioni negative o positive. Ogni qualvolta l'utente inserisce una nuova transazione, essa verrà scaricata e inserita in automatico nella lista. L'inserimento delle transazioni avviene nelle pagine dedicate alle entrate e alle uscite. Nella Figura 5.3 sono mostrate tutte le pagine inerenti alle transazioni. Partendo dalla prima schermata da sinistra, l'utente potrà visualizzare le sue transazioni e un piccolo paragrafo con le istruzioni per l'inserimento di una nuova transazione. Utilizzando uno dei due pulsanti situati al di sopra della barra di navigazione, l'utente può accedere alle sezioni per l'inserimento delle entrate e delle uscite. Sono state utilizzate delle immagini stilizzate per rappresentare le categorie di appartenenza delle possibili transazioni, così da evitare confusione con l'inserimento di troppo testo. Una volta che l'utente avrà selezionato la categoria di appartenenza della transazione da inserire, al centro dello schermo verrà mostrata una finestra di dialogo. Al contempo, apparirà una tastiera numerica, con l'aggiunta del punto decimale. L'utente dovrà, quindi, inserire la somma da registrare e premere il pulsante "Inserisci", come raffigurato nella schermata a destra della Figura 5.3.

Passando, ora, alla sezione delle statistiche, l'utente potrà interagire con vari pulsanti situati poco al di sopra della barra di navigazione. Inizialmente non sarà presente alcun grafico, l'utente dovrà, quindi, premere uno dei pulsanti per visualizzare delle statistiche. Nella Figura 5.4 vengono mostrati i grafici disponibili, tutti di tipo a "torta", così da rendere il confronto tra le categorie più semplice. I tre grafici in questione rappresentano, in ordine: le entrate, il bilancio e le uscite. All'interno di ognuno di essi sono presenti i valori, in percentuale, per ogni categoria e una legenda. Il grafico del bilancio rappresenta un confronto tra le spese in uscita e quelle in entrata. Infine, come in ogni pagina dell'applicazione, l'utente può decidere di effettuare il logout utilizzando il pulsante nella sezione superiore dello schermo.

La terza opzione, selezionabile dalla barra di navigazione, risulta essere la più semplice ed utile. Infatti, essa è la sezione "Aiuto", ovvero, una pagina dove l'utente potrà semplicemente visualizzare un video. Il video in questione viene caricato sul momento e viene riprodotto in automatico. In questa sezione l'utente potrà visualizzare un breve video tutorial per imparare il funzionamento dell'applicazione o nel caso in cui si sia dimenticato come utilizzare alcune funzioni. Nella Figura 5.5 viene mostrata la sezione "Aiuto", dove l'utente visualizzerà il video tutorial sopra descritto.

Infine, l'ultima sezione con cui l'utente potrà interagire è quella per la gestione dei dati utente. In questa pagina, come mostrato in Figura 5.6, l'utente visualizzerà i dati inseriti

Figura 5.3: Sezione delle transazioni

Figura 5.4: Sezione delle statistiche

durante la registrazione e potrà interagire con gli stessi. Per modificare i dati, basterà premere sulle caselle di testo contenenti i dati e modificarli. Successivamente, selezionando il pulsante "Modifica", l'applicazione verificherà che i dati siano corretti e che rispettino i limiti imposti: nel caso in cui i dati siano errati li invierà al database. Al contrario, in caso di dati errati, verrà visualizzato un messaggio di colore grigio, indicante l'errore da correggere.

5.4 Risoluzione dei problemi comuni

In questa sezione, verrà affrontato l'argomento della risoluzione dei problemi comuni che possono insorgere durante l'utilizzo di un'applicazione mobile. Come ogni software, le app mobili possono riscontrare problemi di diversa natura a seconda del dispositivo utilizzato

Figura 5.5: Pagina della sezione aiuto

Figura 5.6: Pagine di modifica dei dati dell'utente

e altre condizioni. Analizzando in modo specifico ogni tipologia di problemi riscontrabili nell'applicazione in esame, troviamo i seguenti:

- *Errori d'interfaccia utente e compatibilità*: questa tipologia di errore può essere riscontrata in dispositivi non presi in considerazione durante lo sviluppo dell'applicazione, oppure a causa di un errore nel codice in esecuzione. Per una corretta risoluzione del problema è necessario verificare che l'applicazione supporti il dispositivo in questione; se il riscontro è positivo, il problema è stato causato dei constraint. In questo caso la risoluzione più efficace può essere imposta chiudendo e riaprendo l'applicazione.

- *Errori di connessione*: gli errori di questo genere dipendono dallo stato della connessione internet del dispositivo. Nell'applicazione è riscontrabile già in fase di login quando, dopo aver premuto il pulsante di login, non viene intrapresa nessuna azione da parte dell'app. Per risolverlo è necessario verificare che il dispositivo risulti connesso a internet.
- *Errori nell'inserimento dei dati*: essi sono riscontrabili ogniqualvolta un utente inserisce dati errati all'interno dell'applicazione. Ad esempio, in fase di registrazione, nel caso in cui l'utente abbia inserito una email non contenente il simbolo della chiocciola, l'applicazione presenterà un messaggio di errore indicante la risoluzione. Per problemi di questo genere è, quindi, necessario verificare i dati inseriti, visualizzando eventuali errori indicati dall'applicazione.

CAPITOLO 6

Discussione

Nei Capitoli 4 e 5 sono state trattate la progettazione e l'implementazione dell'app. In questo capitolo, per concludere, verrà applicata un'analisi SWOT, ovvero uno strumento di pianificazione strategica utilizzato per la valutazione dei punti di forza, delle debolezze, delle opportunità e delle minacce. Nella seconda sezione del capitolo, invece, verranno analizzate e messe a confronto altre applicazioni simili, utilizzando i criteri dell'analisi SWOT.

6.1 Analisi SWOT

L'analisi SWOT è uno strumento di pianificazione strategica utilizzato dalle aziende per valutare i punti di forza (Strengths), di debolezza (Weaknesses), le opportunità (Opportunities) e le minacce (Threats) di un progetto. L'analisi viene visualizzata graficamente mediante una matrice 2 X 2 mostrata nella Figura 6.1, dove è, inoltre, possibile osservare come la riga superiore sia riferita a un'analisi degli elementi interni, mentre quella inferiore all'analisi degli elementi esterni al progetto. Nelle successive sezioni verrà analizzata l'applicazione progettata utilizzando l'analisi SWOT.

6.1.1 Strengths - Punti di forza

I punti di forza risultano essere gli attributi interni positivi che possono influenzare la performance dell'app. Per poter analizzare questi attributi, è necessario porsi delle domande: Quali sono i principali vantaggi competitivi dell'app rispetto alle altre applicazioni simili sul mercato? Quali caratteristiche uniche e funzionalità avanzate offre l'app? Quali sono i punti di forza dell'app in termini di design, usabilità o esperienza utente? L'applicazione consente di gestire le proprie spese in maniera semplice ed elegante? Inoltre, mentre altre applicazioni hanno un costo, sotto forma di acquisto una tantum o abbonamento mensile, l'app progettata sarà gratuita.

Andando ad approfondire le funzionalità avanzate, troviamo la possibilità di eseguire il processo di registrazione e successivo login, mediante il servizio "Sign in with Apple". Un'altra funzionalità avanzata di cui tener conto è l'utilizzo di grafici moderni, utilizzati per fornire all'utente una rappresentazione dettagliata delle proprie spese. Inoltre, il design ideato rispecchia le linee guida imposte da Apple, fornendo, quindi, delle pagine semplici e di facile utilizzo. Questa attenzione nella creazione della UI (interfaccia utente) contribuisce a creare una UX (esperienza utente) ottimale.

Figura 6.1: Matrice 2 X 2 SWOT

Passando, ora, alla sicurezza, notiamo che tutti i dati dell'utente vengono gestiti dal database Firebase, noto per la sua sicurezza e l'utilizzo di protocolli di comunicazione criptati, così da evitare l'esposizione dei dati sensibili. Infine, la possibilità di tradurre in automatico l'applicazione in base alle impostazioni del dispositivo rende quest'ultima utilizzabile anche in altri paesi.

6.1.2 Weaknesses - Debolezze

I punti di debolezza sono gli attributi interni negativi che incidono sulla performance dell'app, mettendola in una condizione di svantaggio rispetto alle applicazioni concorrenti. Vi sono alcune domande da porsi per identificare i punti di debolezza, ovvero: ci sono limitazioni o carenze significative nell'app rispetto alle aspettative degli utenti o alle richieste di mercato? Sussistono problemi di prestazioni, stabilità o compatibilità dell'app su determinate piattaforme o dispositivi? Quali sono le principali difficoltà interne che potrebbero influire sul successo dell'app? L'applicazione, come ogni progetto, può beneficiare di ulteriori aggiunte e modifiche. Nel caso della tabella per la disposizione delle transazioni inserite, non è presente la possibilità di filtrare le transazioni secondo dei criteri preimpostati. L'applicazione, oltretutto, risulta non disponibile per dispositivi Android, ma solo per dispositivi Apple. Infine, nella sezione delle statistiche, i grafici non vengono inizialmente caricati, ma necessitano di un input da parte dell'utente.

6.1.3 Opportunities - Opportunità

Le opportunità sono i fattori esterni positivi che possono essere sfruttati per il successo dell'applicazione. Alcune domande da porsi sono: ci sono tendenze emergenti o cambiamenti nel comportamento degli utenti che possono favorire l'adozione dell'app? Esistono potenziali tecnologie o partner con cui intraprendere nuove opportunità e funzionalità per l'app? Quali sono le opportunità di espansione geografica e di apertura in nuovi mercati? Le opportunità

risultano essere il punto di analisi fondamentale per quest'applicazione. Infatti, essa può essere migliorata aggiungendo nuove tecnologie per la gestione delle transazioni. Ad esempio, le principali piattaforme per l'home banking offrono la possibilità di implementare le loro API per utilizzare i dati all'interno di altre piattaforme. Utilizzando un servizio di questo genere, l'applicazione diventerebbe un portafoglio automatizzato, in grado di gestire tutti i conti bancari dell'utente e, al contempo, le spese non tracciabili, come quelle svolte utilizzando il contante. Una ulteriore possibilità sarebbe l'introduzione dell'applicazione a nuovi mercati, rendendola disponibile in diversi paesi.

6.1.4 Threats - Minacce

Le minacce sono fattori esterni negativi che possono rappresentare un rischio per l'applicazione. Alcune domande da porsi per analizzare al meglio i rischi sono: vi sono barriere all'ingresso nel mercato che potrebbero rendere difficile la crescita o la sopravvivenza dell'app? Vi potrebbero essere cambiamenti normativi, legali o regolatori che potrebbero influenzare l'operatività dell'app? Quali sono le potenziali minacce in termini di sicurezza dei dati o violazioni della privacy che potrebbero danneggiare la reputazione dell'app? Quali sono i principali concorrenti o le principali alternative che potrebbero rappresentare una minaccia per l'app? Le minacce possibili per quest'applicazione risultano essere principalmente dipendenti dal mercato dei concorrenti, ovvero le app realizzate da altri sviluppatori potrebbero risultare migliori o più prestazionali, andando, così, a rimuovere utenti dall'app progettata. Inoltre, il lancio dell'app in un nuovo mercato potrebbe risultare ostico, in quanto la gestione delle proprie spese è un'abitudine che varia a seconda del luogo, della cultura e di altri fattori. Oltre alle minacce derivanti dal mercato dei concorrenti, potrebbero risultare problemi da un uso improprio dell'applicazione. Ad esempio, l'utilizzo errato del processo d'inserimento delle transazioni potrebbe generare un errore nella lettura dei dati e, di conseguenza, non fornire i servizi di gestione delle spese offerti dall'app.

6.2 Confronto con applicazioni correlate

In questa sezione verranno analizzate e confrontate applicazioni correlate a quella progettata. Queste app sono disponibili sull'App Store Apple e, quindi, sono concorrenti dell'applicazione "My Personal Budget". Ogni app verrà analizzata utilizzando l'analisi SWOT, presentata nella sezione precedente; verranno, inoltre, approfondite le caratteristiche e funzionalità che le contraddistinguono dall'app progettata.

6.2.1 Goodbudget

L'applicazione "Goodbudget", mostrata in Figura 6.2, mantiene una forma di navigazione alquanto simile a quella dell'applicazione progettata. Infatti, l'utilizzo di una navigation bar, unito a un design semplice e minimalista, rendono le due applicazioni concettualmente simili. Oltretutto, visualizzando l'app sull'App Store, essa risulta gratuita, con la possibilità di acquistare servizi aggiuntivi mediante acquisti in-app. Le modalità di registrazione e login dell'applicazione analizzata risultato minori in numero rispetto a quella progettata, infatti, l'unico metodo disponibile è l'utilizzo di email e password. Una conseguenza dell'utilizzo di un singolo metodo di registrazione è un ridotto numero di iscritti, in quanto le opzioni di registrazione sono spesso la prima barriera incontrata dall'utente medio.

Goodbudget permette di inserire transazioni mediante la compilazione di un breve documento. Una volta inserite, esse vengono categorizzate in automatico e visualizzate all'interno

di una sezione denominata "Transactions". A differenza dell'applicazione sviluppata, Goodbudget offre la possibilità d'inserire un massimale sulle spese. L'utilizzo di questo metodo di misura rende l'utente maggiormente in grado di gestire le proprie spese mensili e, di conseguenza, intuire quali categorie influiscono eccessivamente sul budget mensile impostato. Nella sezione delle statistiche, è evidente come Goodbudget fornisca una maggiore tipologia di dati, infatti, oltre alle normali statistiche presentate attraverso i grafici a torta, Goodbudget offre la possibilità di mostrare la riduzione dei debiti durante un determinato intervallo mediante l'utilizzo di grafici a barre. Oltre alle funzionalità elencate, l'applicazione non presenta nessun'altra opzione nella navigation bar, rendendola, quindi, estremamente simile all'applicazione sviluppata. Approfondendo le funzionalità presenti nelle impostazioni, è presente l'opzione di sincronizzazione dei dati e, di conseguenza, la possibilità di condividere i dati del conto con altri utenti. Questa funzionalità risulta di enorme importanza in un contesto familiare o aziendale e, quindi, una tecnologia potenzialmente applicabile all'interno dell'app progettata.

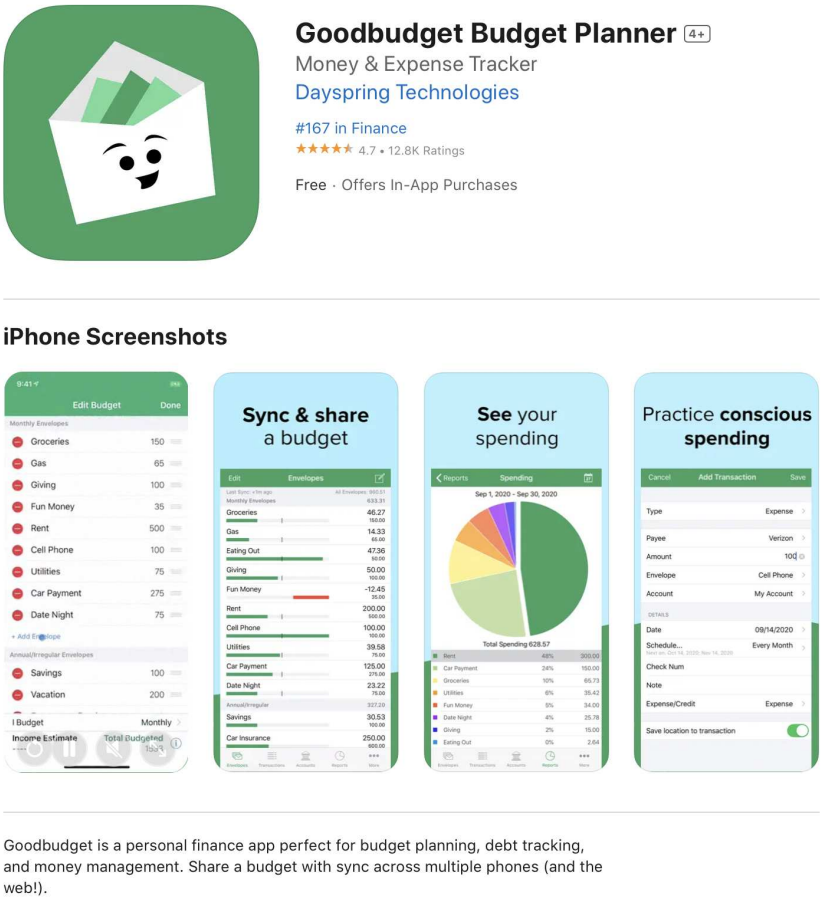

Figura 6.2: Goodbudget - Budget Planner

6.2.2 Wallet

L'app Wallet, mostrata nella Figura 6.3, si presenta con una grafica moderna e dettagliata. Allo stesso modo dell'app sviluppata, il design utilizza un numero ridotto di colori e grafiche vettoriali semplici. All'apertura dell'applicazione in analisi, l'utente visualizza le quattro opzioni di registrazione e login, ovvero: Facebook, Google, Apple e, infine, il metodo classico. Questa sezione dell'app risulta, quindi, maggiormente strutturata rispetto all'app

progettata, fornendo maggiori opzioni all'utente e, di conseguenza, favorendo una maggiore fidelizzazione degli utenti.

L'applicazione in analisi, una volta effettuato l'accesso, si presenta con una navigation bar contente cinque sezioni. Inizialmente, viene presentata una dashboard contenente la funzionalità per collegare tutti i servizi bancari all'app. La funzionalità in questione risulta essere di estrema utilità, in quanto centralizza tutti i conti all'interno di una singola piattaforma. Successivamente, viene proposta una seconda funzionalità denominata "Planning". Questa, formata da due categorie, permette di pianificare pagamenti ricorrenti, nonché di creare budget personalizzati. All'interno della terza e quarta sezione della navigation bar troviamo l'inserimento delle transazioni e le statistiche. Esse risultano speculari alle sezioni presenti nell'app sviluppata, con la sola differenza della possibilità di filtrare le statistiche in base a intervalli di tempo. Inoltre, è presente la funzionalità di sincronizzazione delle transazioni, con la possibilità di collegare fino a tre account contemporaneamente.

Oltre a questo limite, l'applicazione mette a disposizione una versione a pagamento mensile, annuale o mediante un singolo pagamento. Essa analizzata risulta, quindi, funzionalmente avanzata, sia per la sezione dell'accesso, che per la possibilità di personalizzazione e condivisione delle proprie spese. L'applicazione risulta, quindi, di ottima qualità e, come dimostrato dai numeri generati, estremamente popolare con oltre sei milioni di download e una media di recensioni tendenzialmente ottima.

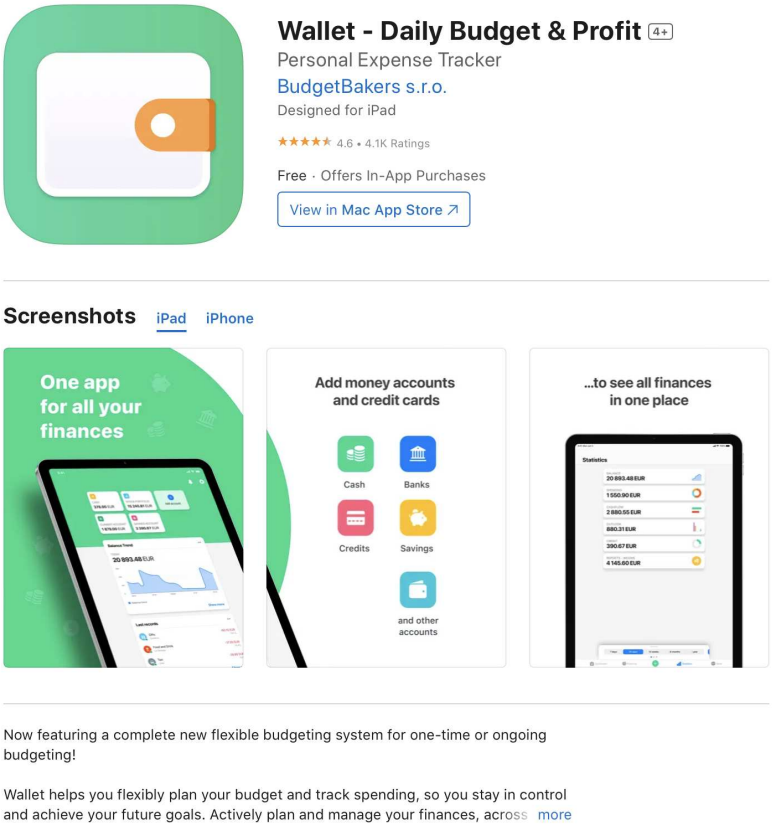

Figura 6.3: Wallet - Daily Budget and Profits

CAPITOLO 7

Conclusioni

Nel presente lavoro di ricerca sono stati esaminati in modo approfondito la progettazione e lo sviluppo, in linguaggio Swift, di un'applicazione mobile per la gestione e la visualizzazione delle spese.

L'approfondimento iniziale, riguardante il linguaggio utilizzato, è stato il primo passo per introdurre le motivazioni che hanno portato alla scelta delle tecnologie applicate. Successivamente, sono state descritte le specifiche del progetto, dando rilevanza ai requisiti funzionali e non funzionali, così da garantire un'analisi dettagliata delle funzionalità che sono state sviluppate in seguito. Inoltre, mediante l'impiego di diagrammi, sono stati identificati gli attori ed i casi d'uso principali.

In seguito alla fase iniziale, è stata intrapresa l'analisi dei requisiti dell'app. In questa sezione, inizialmente, è stata proposta una descrizione del progetto, per poi introdurre la descrizione dei requisiti funzionali e non funzionali. L'analisi dei requisiti ha permesso di identificare gli obiettivi dell'applicazione, facendo risaltare le funzionalità e le specifiche da implementare nelle fasi seguenti. Sono poi stati identificati gli attori ed i casi d'uso, implementando diagrammi volti a fornire una spiegazione dettagliata dei legami tra le componenti sopra citate.

Successivamente, si è passati alla fase di progettazione. Attraverso una mappa dell'applicazione, è stata descritta la struttura gerarchica del progetto e, mediante l'impiego di grafici e storyboard, sono state determinate le principali caratteristiche grafiche dell'app. Inoltre, sono state descritte le linee guida utilizzate per la creazione del design delle componenti, analizzando il comportamento degli utenti. Per concludere, è stata approfondita la componente dati, descrivendo il processo di sviluppo a partire dallo schema del database, per poi approfondire la gestione dell'autenticazione dell'utente e, inoltre, la gestione dei dati permanenti dello stesso.

Passando alla fase successiva, è stata svolta la fase di implementazione, ovvero lo sviluppo dell'applicazione. Inizialmente, sono state elencate le tecnologie e gli strumenti disponibili per lo sviluppo. In seguito sono state esposte le motivazioni per l'utilizzo dell'IDE Xcode. Lo step successivo è consistito nella la progettazione dell'interfaccia utente. In questa fase sono state applicate le scelte descritte nella sezione precedente, cercando di rendere semplici ed intuitive le componenti grafiche. Sono poi state descritte in modo approfondito le funzionalità principali, iniziando dalla scelta e dell'implementazione delle API esterne e dei servizi di terze parti. Dopo aver descritto le funzionalità introdotte da questi servizi, è stata analizzata l'implementazione della funzionalità di autenticazione, mostrando la gestione dei

dati e descrivendo le operazioni CRUD applicate all'interno dell'app. In questa fase sono state analizzate la gestione degli errori e, in seguito, la fase di test e debug, necessarie per ottimizzare l'applicazione e portarla ad uno stato ottimale per il rilascio.

Sono, poi, state descritte le modalità di utilizzo dell'app, approfondendo le funzionalità all'interno di un manuale utente. Infine, all'interno del manuale, sono stati descritti i problemi comuni e le diverse risoluzioni applicabili.

Infine, mediante l'impiego dell'analisi SWOT, l'applicazione è stata inizialmente analizzata, identificando i punti di forza, le debolezze, le opportunità e le minacce. Successivamente, è stato svolto un confronto, utilizzando il metodo SWOT, con applicazioni mobili correlate. All'interno di quest'ultima sezione, le funzionalità delle app a confronto sono state approfondite e descritte, evidenziando gli approcci e le soluzioni a vari problemi e identificando le differenza con l'app progettata.

Si presume che l'app sviluppata, descritta durante questa tesi, non sia conclusa. Essa, infatti, potrebbe essere considerata come una base su cui implementare gli sviluppi futuri. Le funzionalità descritte e implementate potrebbero sicuramente essere ampliate mediante l'impiego di aggiornamenti. Inoltre, come evidenziato dal confronto con le app correlate già presenti sul mercato, le possibilità di aggiungere nuove funzionalità maggiormente evolute sono molteplici. Per concludere, l'obiettivo di quest'app e dei suoi eventuali sviluppi futuri sarà di mantenere un livello di sicurezza dei dati elevato, unito ad un'attenzione nello sviluppo dell'interfaccia utente semplice, affinché l'esperienza utente rimanga ottimale e simile a quanto descritto in all'interno di questa tesi.

Bibliografia

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

FAROOK, F. e HOLLEMANS, M. (2020), *UIKit Apprentice, Firstr Edition*, Razeware LLC.

GRAY, A. (2015), *Swift Espresso - Programmazione per iOS e OS X*, Hoepli.

LAU, K. e NGO, V. (2021), *Data Structures and Algorithms in Swift*, Raywenderlich Tutorial Team.

NG, S. (2021), *Beginning iOS Programming with Swift and UIKit*, AppCoda Limited.

SAHAR, A. e CLAYTON, C. (2022), *iOS 16 Programming - Seventh Edition*, Packt.

SMYTH, N. (2022), *SwiftUI Essentials - iOS 16*, Payload Media.

VENNARO, E. (2023), *iOS Development at Scale: App Architecture and Design Patterns for Mobile Engineers*, Apress.

WANG, W. (2023), *Beginning iPhone Development with SwiftUI*, Apress.

Siti web consultati

- Apple Developer Documentation, Xcode – `https://developer.apple.com`
- Swift.org, Swift Documentation – `https://www.swift.org/documentation/`
- CocoaPods – `https://cocoapods.org`
- Swift Package Index – `https://swiftpackageindex.com`
- AppCoda – `https://www.appcoda.com`
- Kodeco – `https://www.kodeco.com/home`
- PaintCode – `https://www.paintcodeapp.com`