Università Politecnica delle Marche

Facoltà di Ingegneria

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea Magistrale in Ingegneria Informatica e dell'Automazione

Tesi di Laurea

Progettazione e implementazione di un'app iOS per il supporto alle attività di marketing sui clienti di una nota azienda del settore alimentare

Design and implementation of an iOS app to support marketing activities on the customers of a well known food company

Relatore

Prof. Domenico Ursino

Candidato

Alessandro Zechini

Anno Accademico 2019-2020

Indice

1 L'ecosistema iOS e il linguaggio SWIFT 5
1.1 Il sistema operativo iOS 5
1.2 Objective-C 6
1.3 Da Objective-C a Swift 6
1.4 L'architettura di iOS 6
1.5 Xcode 9
1.5.1 Interface Builder 9
1.5.2 Assistant Editor 10
1.5.3 XCTest e strumenti per il testing 10
1.6 AutoLayout 11

2 Descrizione generale dell'app 13
2.1 Macro-architettura del sistema 13
2.2 Progetto iniziale 14
2.2.1 Registrazione e Login 14
2.2.2 Onboarding 14
2.2.3 Home 14
2.2.4 Store Locator 15
2.2.5 Il tuo profilo 15
2.2.6 Questionario 16
2.2.7 Notifiche push 16
2.2.8 Assistenza 17
2.3 Struttura finale dell'applicazione 18
2.3.1 Home 18
2.3.2 Store Locator 18
2.3.3 Punti e Premi 19
2.3.4 Altre sezioni 19
2.4 Scelta delle tecnologie 19
2.5 Concetti preliminari 20
2.5.1 Protocol 20
2.5.2 UIViewController 20
2.6 Architettura dell'applicazione 21
2.6.1 Il pattern MVC 21

2.6.2 MVVM 23
2.7 Flussi di navigazione e Flow Coordinator 24

3 Progettazione e implementazione del componente "Magazine" . . 27
3.1 Progettazione 27
3.2 Realizzazione grafica 28
3.2.1 Card 28
3.2.2 Magazine 28
3.3 Implementazione 31
3.3.1 Schermata principale e UITablieView 31
3.3.2 Delegation Pattern 32
3.3.3 Filtro articoli 33
3.3.4 Dettaglio articolo 34
3.3.5 Riproduzione dei video 35

4 Progettazione e implementazione del componente "Impostazioni" 37
4.1 Progettazione 37
4.2 Realizzazione grafica 37
4.3 Implementazione 38
4.3.1 La schermata principale 38
4.3.2 Cambio dell'immagine di profilo 40
4.3.3 Autorizzazioni per l'accesso a fotocamera, geolocalizzazione e notifiche push 41
4.3.4 Observer pattern 43

5 Progettazione e implementazione del componente "Punti e premi" 45
5.1 Progettazione 45
5.2 Realizzazione grafica 45
5.3 Implementazione 46
5.3.1 Promises e gestione avanzata di richieste asincrone 46
5.3.2 Immagini delle card e libreria KingFisher 48
5.3.3 "Catalogo premi", filtri e RangeSlider 49

6 Progettazione e implementazione del componente "Profilo" 53
6.1 Progettazione 53
6.2 Realizzazione grafica 53
6.3 Implementazione 54
6.3.1 "Prossime missioni" 54
6.3.2 Gestione della memoria in Swift 54
6.3.3 "Lista di tutte le attività", "Premi richiesti", "Le tue macchine" 55
6.3.4 Enumeration con valori associati 57

7 Manuale utente dell'app realizzata 59
7.1 Registrazione e login 59
7.2 Primo accesso e procedura di "Onboarding" 61
7.3 Home 63
7.4 Store 64
7.5 Punti e premi 66
7.6 Profilo 67
7.7 Impostazioni 68

8 Analisi e discussione del lavoro realizzato 73
8.1 Analisi SWOT 73
8.1.1 Punti di forza 73
8.1.2 Debolezze 74
8.1.3 Opportunità 75
8.1.4 Minacce 75
8.2 Sistemi affini e altre app con programmi di loyalty 75
8.2.1 Aliexpress 75
8.2.2 Gearbest 77
8.2.3 Mulino per me 78
8.2.4 Coccole Pampers 79
8.2.5 Confronto con l'app realizzata 79

9 Conclusioni e sviluppi futuri 83

Riferimenti bibliografici 85

Elenco delle figure

1.1 I livelli dell'architettura di iOS 7
1.2 Un esempio di storyboard; pur non conoscendo l'applicazione è possibile osservarne il flusso di navigazione. 10
1.3 Sulla sinistra il Test Navigator di Xcode. Sulla destra un report di Code Coverage. 11

2.1 La macro architettura del sistema Applicazione - Digital Platform. . . 13
2.2 Mockup della sezione Home; nella parte inferiore della schermata viene mostrata la barra di navigazione. 15
2.3 Mockup della sezione relativa al profilo. 16
2.4 La procedura di registrazione alla piattaforma Firebase Cloud Messaging 17
2.5 Un mockup della funzione di assistenza 18
2.6 Il "ciclo di vita" di uno *UIViewController*. 21
2.7 L'evoluzione di MVC proposta da Apple. 22
2.8 L'architettura effettivamente ottenuta applicando alle app Swift il modello di MVC introdotto da Apple. 22
2.9 Il pattern MVVM. 23

3.1 La sezione "Magazine". 29
3.2 La schermata filtro della sezione "Magazine". 30
3.3 A sinistra è riportata una galleria di immagini all'interno di un articolo, a destra una *UICollectionView* che mostra gli articoli correlati. 31
3.4 Un diagramma che illustra il funzionamento del Delegation Pattern. . 32
3.5 Un video riprodotto all'interno dell'applicazione. 36

4.1 La schermata principale di "Impostazioni". 38
4.2 Le proprietà del componente *UISTackView* 39
4.3 Un diagramma che rappresenta il funzionamento dell' "observer pattern". 43

5.1 Benchmark di alcune librerie che implementano le "promise". 47
5.2 Il modale "filtri" della sezione "Catalogo premi". 50

6.1 Le card contenenti le sottosezioni di “Profilo”. 56

7.1 La schermata di benvenuto. 59
7.2 La schermata di registrazione. 60
7.3 La schermata di “Email inviata”, mostrata alla fine del processo di registrazione. 60
7.4 La schermata di login. 61
7.5 Il flusso di “Onboarding”. 62
7.6 La sezione “Home”. 63
7.7 La schermata di dettaglio di una promozione. 63
7.8 La sezione “Store”. 64
7.9 Il catalogo prodotti. 64
7.10 La schermata principale di “Magazine”. 65
7.11 La schermata di dettaglio di un articolo. 65
7.12 La schermata principale di “Punti e premi”, seguita dall’informativa sulla racolta punti. 66
7.13 A sinistra, la schermata “catalogo premi”; a destra, la schermata dei filtri. 67
7.14 La procedura di riscatto premi. 67
7.15 La sezione “Profilo”. 68
7.16 La schermata principale di “Impostazioni”. 69
7.17 La schermata per il cambio di nome e cognome. 69
7.18 Il cambio dell’immagine di profilo. 70
7.19 Il reset della password. 70
7.20 La schermata per il cambio dei consensi. 71
7.21 “Domande frequenti”. 71

8.1 Analisi SWOT dell’app realizzata. 74
8.2 La schermata principale della sezione loyalty di Aliexpress. 76
8.3 Il gioco “Lucky tree” di Aliexpress. 77
8.4 La schermata “VIP” di Gearbest. 78
8.5 La schermata principale di “Mulino per me”. 79
8.6 Una schermata di “Mulino per me” relativa a un concorso. 80
8.7 Alcune schermate dell’applicazione “Coccole Pampers”. 80

Introduzione

Negli ultimi anni, la diffusione degli smartphone è cresciuta ad un ritmo sempre maggiore, tanto che, seguendo un trend ormai stabile, il traffico Internet proveniente dai dispositivi mobili continua a crescere a scapito di quello proveniente dai personal computer. Secondo un'indagine del portale tedesco "Statista", a maggio 2020 il numero di utenti di smartphone è pari a circa 3,5 miliardi, ovvero il 45% della popolazione mondiale. L'ingresso degli smartphone nel mercato di massa ha plasmato sempre più la vita delle persone, portando alla comparsa di abitudini precedentemente inesistenti, e cambiando profondamente il modo in cui ognuno di noi comunica, paga, fa acquisti o prenota servizi.

La possibilità di sviluppare app di terze parti per questi dispositivi, nonchè la disponibilità di strumenti sempre più potenti e complessi hanno fatto sì che nascessero business mai visti prima, e che il mercato delle app esplodesse fino a raggiungere le proporzioni attuali. Sempre secondo il portale Statista, infatti, alla fine del primo quadrimestre del 2020 si possono contare circa 2560000 app sul Play Store di Google e 1847000 sull App Store di Apple, oltre a svariate migliaia di app presenti su store meno diffusi.

Grazie agli smartphone e alle loro app, moltissime aziende sono state in grado di trasformare il proprio business, sia dal punto di vista dell'organizzazione interna, sia per quanto riguarda la gestione del rapporto con i clienti e le iniziative di marketing. Lo stato di connessione ad Internet permanente da parte delle persone rappresenta una grande opportunità, in quanto permette un flusso continuo di dati sui clienti che utilizzano quotidianamente e in modo intensivo i propri dispositivi. Le imprese sono così in grado, in tale contesto, di ricevere un riscontro immediato circa l'efficacia della propria comunicazione. Un altro grande grande vantaggio nella raccolta e nell'analisi di questi dati consiste nel fatto che, basandosi sui feedback da parte dei clienti, le imprese possono rimodulare le proprie attività produttive, e non solo, allocando in modo più efficiente le risorse di cui dispongono e adattandosi più prontamente alle richieste del mercato.

In alcuni casi le app per smartphone sono divenute veri e propri prodotti di punta per le imprese; basti pensare a tutte quelle case del settore videoludico che, negli ultimi anni, hanno conseguito ricavi maggiori tramite giochi per smartphone,

piuttosto che mediante la vendita di prodotti dedicati a piattaforme tradizionali, come personal computer o console. Anche il settore dei software per la produttività ha visto un incremento dell'utilizzo delle app in versione mobile e, per lo svolgimento di una grande quantità di operazioni, si è passati dall'utilizzare i pc ad optare per gli smartphone. Tutto ciò è stato possibile anche grazie alla crescente potenza di questi dispositivi, che sono ormai in grado di offrire un grado di fluidità impensabile fino a pochi anni fa.

Questo lavoro di tesi si basa sull'app richiesta da una nota compagnia italiana operante nel settore alimentare. Tale azienda ha dimostrato, nel corso della sua storia, una forte propensione all'innovazione, ed ha effettuato importanti investimenti nel campo del marketing, maturando ottime competenze in tale ambito. L'azienda ha praticato le sue attività di promozione sia nelle sue forme più tradizionali, sia in quelle di più recente diffusione, come, ad esempio, le campagne sui social network, coinvolgendo numerosi testimonial di prestigio. Essa è solita promuovere, periodicamente, dei concorsi a premi, indirizzati a coloro che acquistano i suoi prodotti. Tali iniziative sono state gestite, negli ultimi anni, mediante dei codici che dovevano essere inviati via SMS o inseriti su un sito web dedicato. Queste modalità di partecipazione sono ormai divenute obsolete e non sono più in grado di coinvolgere i clienti in modo massiccio; infatti, si stima che il grado di coinvolgimento nell'utilizzo di un'app sia pari a circa quattro volte quello derivante dalla navigazione di un sito web. Per questa ragione, l'azienda ha richiesto la realizzazione di un'app per smartphone, che fosse compatibile con i sistemi operativi più diffusi, ovvero Android ed iOS, e che permettesse di rinnovare le sue iniziative di loyalty, di introdurne di nuove e migliorare le modalità di gestione delle stesse.

Il compito principale dell'applicazione realizzata è, quindi, quello di consentire all'utente di raccogliere punti per il programma di loyalty, registrando dei codici riportati sulle confezioni dei prodotti acquistati. Una volta accumulati i punti necessari, egli può spostarsi in una sezione apposita per richiedere dei premi, che potrà scegliere tra quelli elencati all'interno di un catalogo. La promozione di questo tipo di iniziative mediante un'app per smartphone, vista la diffusione di questi dispositivi, può consentire all'azienda di conseguire una maggiore partecipazione al programma fedeltà, e, di conseguenza, una fidelizzazione più efficace dei suoi clienti. Tramite l'applicazione, l'azienda si propone, anche, di far conoscere tutti i suoi prodotti, incentivando l'acquisto di quelli più recenti. Mediante degli strumenti per il tracciamento si potranno conoscere le inserzioni più visualizzate dai clienti, valutando, al contempo, l'efficacia delle proprie campagne pubblicitarie. La raccolta di informazioni non si limita ai dati di utilizzo dell'app; infatti, periodicamente, verranno proposti all'utente dei questionari mirati ad individuare la fascia di mercato a cui egli appartiene. Spingendo l'utente a registrare i prodotti acquistati, mediante l'immissione dei loro codici, il sistema di profilazione raggiunge un'efficacia ancora maggiore, ed è in grado, in questo modo, di mostrargli i contenuti di maggior interesse.

Un ulteriore obiettivo dell'app è quello di rafforzare l'immagine del marchio dell'azienda cliente, mettendo a disposizione di quest'ultima una sorta di rivista virtuale, mediante la quale potranno essere messi a disposizione dell'utente degli articoli riguardo arte, cucina e iniziative umanitarie. In questo modo sarà possibile diffondere, allo stesso tempo, i valori di sostenibilità, diritti umani e sviluppo, che

costituiscono i punti fondamentali della mission dell'azienda.

La tesi si articola in nove capitoli. Il primo di questi fornisce una panoramica sull'ecosistema Apple, il sistema operativo iOS e il linguaggio Swift. Nel secondo capitolo vengono descritte l'architettura e le caratteristiche principali dell'applicazione realizzata. I capitoli dal terzo al sesto sono dedicati alle diverse sezioni dell'app. Per ciascuna di esse sono stati approfonditi gli aspetti riguardanti l'implementazione e le caratteristiche peculiari dell'interfaccia grafica. Nel settimo capitolo è riportato il manuale utente dell'app, che illustra i diversi flussi di navigazione della stessa. L'ottavo capitolo è dedicato all'analisi SWOT del progetto e al confronto con sistemi affini a quello realizzato, infine nel nono capitolo, sono esposte le conclusioni ed alcune ipotesi di sviluppi futuri dell'applicazione.

1

L'ecosistema iOS e il linguaggio SWIFT

In questo capitolo vengono descritti l'ecosistema dei dispositivi mobili Apple, il sistema operativo iOS e il relativo SDK.

1.1 Il sistema operativo iOS

La potenza e la versatilità degli smartphone che vengono ormai date per scontata sono il frutto di un'importante rivoluzione lanciata da Apple nel 2007, quando, al Macworld Conference & Expo di San Francisco, viene presentata, ancora priva di nome, la Versione 1 del sistema operativo che oggi conosciamo come iOS.

Il primo iPhone entra in commercio il 29 giugno dello stesso anno, sebbene sprovvisto di tutte le caratteristiche che oggi lo contraddistinguono, tra cui il supporto per le applicazioni di terze parti. Tra le app installate comparivano Calendario, Foto, Fotocamera, Note, Safari, Mail, Telefono, e iPod.

Il 6 marzo 2008, in concomitanza con la pubblicazione della prima versione beta dell'SDK, il sistema operativo è stato denominato ufficialmente come "iPhone OS". L'11 luglio 2008 viene pubblicato l'aggiornamento a iPhone OS 2 che aggiunge, tra le altre funzioni, l'App Store e la possibilità di installare applicazioni di terze parti tramite quest'ultimo.

Il quarto aggiornamento del sistema operativo, pubblicato con iPhone 4 il 21 giugno 2010, ha aggiunto numerose funzioni, quali il multitasking per le applicazioni di terze parti, FaceTime e iBooks.

Il 7 giugno 2010, Apple rinomina il suo sistema operativo da iPhone OS a iOS, e unifica iPhone, iPodTouch e iPad con una Versione comune, la 4.2.1.

Nel corso degli anni sono state aggiunte importanti migliorie ad iOS, fino a giungere all'ultima versione, la 13.3.1, rilasciata in data 10 Dicembre 2019. A partire da questa versione Apple ha deciso di separare nuovamente iPad da iPhone e iPodTouch, rilasciando un sistema dedicato ai suoi tablet denominato iPadOS.

1.2 Objective-C

Quando iPhoneOS 2 è stato lanciato gli sviluppatori avevano a disposizione il linguaggio che in ambiente Apple era già utilizzato per applicazioni desktop, ovvero Objective-C.

Objective-C fu creato da Brad Cox e Tom Love nei primi anni '80 presso la Productivity Products International(PPI). L'obiettivo principale del linguaggio era quello di integrare i meccanismi di messaging tipici dello Smalltalk senza perdere la compatibilità con il linguaggio C.

Love e Cox fondarono la PPI per commercializzare Objective-C. Nel 1986 Cox pubblicò una descrizione di Objective-C nel libro "Object-Oriented Programming, An Evolutionary Approach", mentre nel 1988 NeXT acquisì la licenza per Objective-C ed estese il compilatore GCC rendendolo in grado di supportare Objective-C.

La Apple acquisì NeXT nel 1996, ed iniziò ad utilizzare OpenStep, la piattaforma di programmazione di NeXT, nel suo sistema operativo Mac OS X. Il nuovo sistema includeva Objective-C e tutti i relativi strumenti di NeXT, riuniti in un unico tool denominato Xcode. La maggior parte delle API Cocoa di Apple è tuttora basata sulle interfacce di OneStep.

1.3 Da Objective-C a Swift

Per molti anni Objective-C è stato il linguaggio standard per lo sviluppo in ambiente iOS ed Apple in generale, finchè nel 2014, durante il WorldWide Developers Conference (WWDC), fu annunciato un linguaggio più moderno, sicuro e performante di Objective-C: si trattava di Swift. Grazie a Swift il mondo dello sviluppo iOS è riuscito ad aggiornarsi rimanendo totalmente compatibile con la code-base già esistente. Swift è stato infatti progettato per essere integrato con il codice Objective-C e poter essere utilizzato assieme a quest'ultimo all'interno dello stesso programma.

Per permettere ciò viene utilizzato il compilatore open source Low Level Virtual Machine (LLVM), incluso in Xcode, e il run time di Objective C.

Il 3 Dicembre 2015 venne pubblicato il codice sorgente di Swift con licenza Apache 2.0, avviando il linguaggio verso una diffusione sempre più ampia ed aprendolo ai preziosi contributi della comunità di sviluppatori. Il 25 marzo 2019 è stata pubblicata la Versione 5.0 per sistemi Apple e Linux, mentre, a settembre dello stesso anno, è stata rilasciata la Versione 5.1.

1.4 L'architettura di iOS

L'architettura di iOS è di tipo stratificato. Il sistema operativo fa da intermediario tra le applicazioni e l'hardware del dispositivo utilizzato. Le applicazioni utilizzano una serie di interfacce di sistema che permetttono la semplificazione della scrittura delle app stesse. Gli strati inferiori espongono una serie di servizi di base su cui tutte le applicazioni si appoggiano, mentre quelli superiori forniscono interfacce più complesse. Solitamente Apple fornisce queste interfacce attraverso i suoi framework

che includono librerie dinamiche, file di header e applicazioni di supporto. Nella figura 1.1 possiamo osservare gli strati principali dell'architettura di iOS.

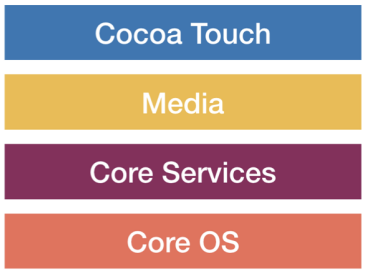

Figura 1.1. I livelli dell'architettura di iOS

Più specificamente possiamo considerare i seguenti livelli:

- *Core OS*. Questo strato include tutti i servizi di più basso livello tra cui thread, crittografia e servizi per il calcolo. I componenti facenti parte di Core OS sono:
  - *Accelerate*: permette l'esecuzione di calcoli complessi e l'elaborazione di segnali digitali.
  - *Core Bluetooth*: fornisce la connettività per dispositivi bluetooth low-energy.
  - *External Accessory*: consente di interfacciarsi con gli accessori connessi al telefono tramite dock o bluetooth.
  - *Local Authentication*: è un framework per l'autenticazione tramite passphrase o touch-ID.
  - *Security*: fornisce funzioni per la crittografia.
  - *System*: include una serie di metodi per le operazioni di sistema di basso livello.
- *Core Services*. Lo strato Core Services viene utilizzato per accedere a servizi come la gestione dei file, lo storage iCloud e i servizi di rete. I suoi componenti principali sono:
  - *Accounts*: utilizzato per facilitare la memorizzazione e l'autenticazione degli utenti.
  - *Address Book*: permette di manipolare gli indirizzi e le informazioni di contatto.
  - *CFNetwork*: fornisce l'accesso ai socket BSD e permette di effettuare richieste HTTP e FTP.
  - *Core Data*: fornisce un modello dati basato su SQLite.
  - *Core Foundation*: offre una serie di funzionalità di base, come memorizzazione e persistenza dei dati, elaborazione di test, ordinamento e filtraggio.
  - *Core Location*: utilizzato per ottenere latitudine e longitudine dal GPS del telefono.
  - *Core Motion*: gestisce gli eventi relativi all'uso di accelerometro e giroscopio.
  - *Event Kit*: utilizzato per accedere e modificare le informazioni memorizzate nel calendario.

  - *Foundation*: fornisce dei wrapper per l'utilizzo dei servizi di Core Foundation mediante il paradigma ad oggetti. Permette la manipolazione di stringhe, array e dizionari, nonchè la gestione dei thread.
  - *HealthKit*: memorizza e legge dati relativi alla salute dell'utente.
  - *HomeKit*: è un framework per l'interazione con l'hardware relativo alla domotica.
  - *NewsStand*: utilizzato per la creazione di contenuti come articoli di giornale e riviste.
  - *Pass Kit*: fornisce all'utente informazioni sulle sue transazioni, coupon e biglietti.
  - Quick Look: consente di aprire file direttamente all'interno di un'app, separando da quest'ultima la logica che riguarda la lettura e scrittura degli stessi.
  - *Social*: facilita l'integrazione di social come Facebook e Twitter all'interno della propria app.
  - *Store Kit*: facilita l'esecuzione di transazioni in-app, utilizzando l'App Store di Apple per gestirle.
  - *System Configuration*: permette di determinare lo stato della configurazione di rete del telefono.
- *Media*. Lo strato Media si occupa di gestire tutto ciò che riguarda audio, video e grafica 3D. I componenti che fanno parte di questo layer sono:
  - *AV Foundation*: utilizzato per gestire la riproduzione e l'editing di video e suoni.
  - *Core Audio*: espone metodi per riprodurre e registrare audio, nonchè riprodurre suoni di notifica e vibrazioni.
  - *Core Image*: permette operazioni avanzate nell'ambito di elaborazione di immagini e video, come ad esempio, riconoscimento facciale o filtraggio di immagini.
  - *Core Graphics*: fornisce componenti per il disegno 2D.
  - *Core Text*: consente la manipolazione del testo.
  - *Image I/O*: utilizzato per importare ed esportare dati e metadati di tutti i formati di immagine supportati da iOS.
  - *Media Player*: fornisce un media player che può essere utilizzato facilmente nelle proprie applicazioni.
  - *Metal*: consente di accedere all'hardware grafico a basso livello.
  - *OpenGL ES*: un sottoinsieme della libreria grafica OpenGL.
  - *Photos*: permette di accedere alle foto presenti nella galleria iOS e allo storage iCloud
  - *Quartz Core*: utilizzato per creare animazioni che sfruttano l'hardware del telefono
- *Cocoa Touch*. Lo strato di livello più alto, comprende diversi framework con funzionalità avanzate. Questi sono:
  - *UIKit*: fornisce numerose funzionalità, dal lancio dell'applicazione, alla gestione degli eventi multitouch fino all'accesso all'accelerometro.
  - *Map Kit*: consente di aggiungere le funzionalità di Apple Maps alla propria applicazione.

- *Game Kit*: fornisce meccanismi per creare e usare reti peer-to-peer, chat vocali e altre funzionalità di rete.
- *Message UI/ AddressBook UI/ EventKit UI*: consentono di accedere a messaggi, contatti ed eventi del calendario.
- *Notification Center*: consente di interagire con il centro notifiche di iOS.
- *PhotosUI*: consente di estendere le funzionalità dell'editor di immagini di iOS.
- *iAd*: assiste lo sviluppatore nell'integrazione delle pubblicità nella propria applicazione.

1.5 Xcode

Xcode è l'IDE fornito da Apple per lo sviluppo delle applicazioni per macOS e iOS. Esso include numerosi strumenti avanzati per migliorare la produttività e la qualità dei prodotti degli sviluppatori. Di seguito ne vengono descritti alcuni.

1.5.1 Interface Builder

Xcode mette a disposizione un editor che consente di realizzarne interfacce utente per la propria app senza scrivere codice. Nell'interface builder sono disponibili tutti i componenti dei framework Cocoa e Cocoa Touch. Poichè questi sono progettati tramite il pattern Model-View-Controller è immediato realizzare le interfacce in modo totalmente indipendente dalla loro implementazione.

Le UI vengono memorizzate in file ".xib" e connesse dinamicamente al codice da macOS o iOS quando le app vengono eseguite. Su Xcode il sistema di riferimento degli elementi grafici è quello cartesiano, con le coordinate espresse in pt. L'origine del piano, cioè le coordinate x:0 e y:0, coincidono con il lato superiore sinistro del display.

Nell'Interface Builder, premendo il bottone "View as:" è possibile visualizzare le view sui diversi dispositivi Apple e scegliere l'orientamento del display. Questo consente di adattare le proprie schermate alle diverse dimensioni degli schermi.

A partire dalla Versione 5 di iOS sono stati introdotti gli Storyboard, dei particolari file che rendono lo sviluppo delle interfacce ancora più rapido e agevole. Gli storyboard consentono infatti di disegnare e gestire più schermate contemporaneamente, controllandone anche il flusso di navigazione. Un esempio di storyboard viene riportato in Figura 1.2.

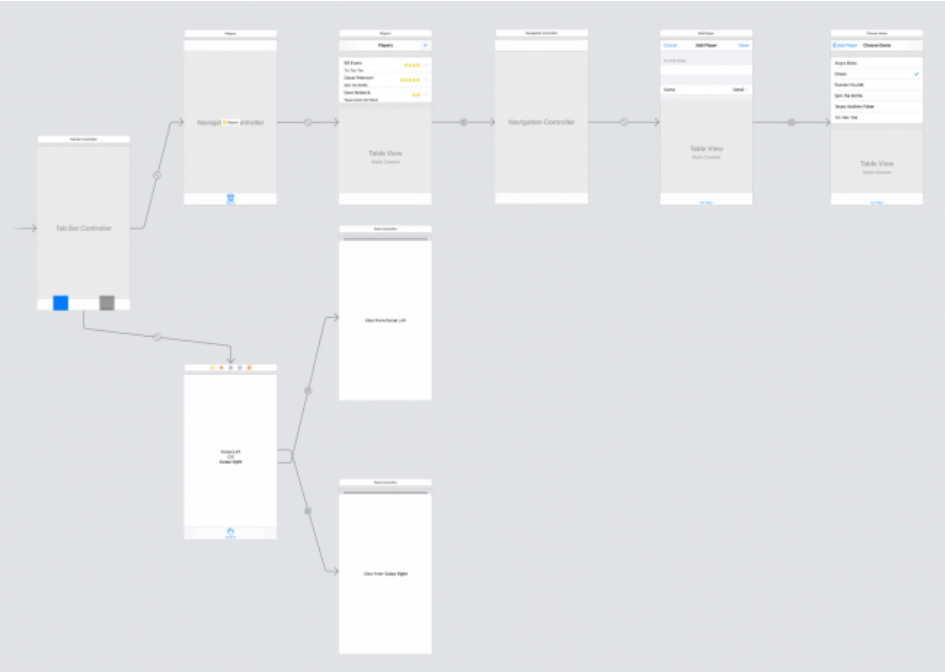

Figura 1.2. Un esempio di storyboard; pur non conoscendo l'applicazione è possibile osservarne il flusso di navigazione.

1.5.2 Assistant Editor

Altro strumento fondamentale presente in Xcode è l'Assistant Editor. Nel momento in cui si ha la necessità di modificare diversi file tra loro correlati, l'Assistant Editor consente di visualizzarli all'interno della stessa finestra, affiancandoli e permettendone la modifica simultanea. Questa funzione è particolarmente utile nel momento in cui si vuole modificare una classe legata ad una view, collegando rapidamente i componenti e gli eventi della UI al codice eseguibile.

1.5.3 XCTest e strumenti per il testing

Xcode fornisce numerosi strumenti per il testing che aiutano lo sviluppatore a conseguire una maggiore stabilità del proprio software. Tra questi vi è XCTest, framework per Unit Testing e UI Testing, attualmente considerato una delle migliori opzioni per il testing di applicazioni scritte in Swift o Objective-C.

I test scritti con XCTest possono essere eseguiti indifferentemente su simulatore o su dispositivi fisici e sono metodi di istanza; di conseguenza non richiedono il passaggio di parametri e devono avere un nome che inizia con "test". Tutti i test sono visibli nel Test Navigator di Xcode, un'interfaccia che consente di visualizzarne rapidamente la classe di appartenenza e l'esito in caso di esecuzione. Inoltre dal Test Navigator si può lanciare singolarmente una classe di test o un particolare test.

È altresì disponibile un tool per verificare la Code Coverage, ovvero la percentuale di codice che i test scritti sono in grado di verificare. Il report di Code Coverage mostra la percentuale di codice coperto a livello di applicazione, classe o metodo, comunicando allo sviluppatore quali parti dell'applicazione necessitano di nuovi test. La figura 1.3 riporta, sulla sinistra, il Test Navigator di Xcode. Sulla destra un report di Code Coverage.

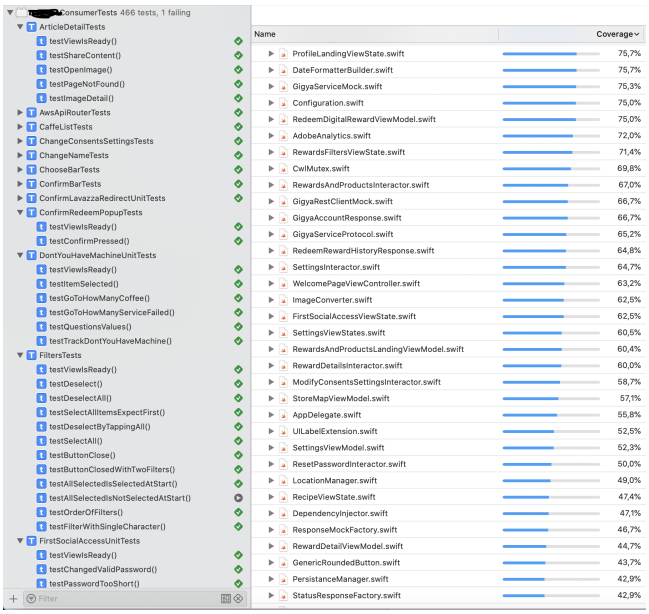

Figura 1.3. Sulla sinistra il Test Navigator di Xcode. Sulla destra un report di Code Coverage.

1.6 AutoLayout

Con la commercializzazione di numerosi modelli di iPhone è emersa la necessità di adattare le interfacce grafiche delle app alle diverse dimensioni e ai diversi orientamenti degli schermi dei dispositivi. A tal proposito nel 2012 è stato rilasciato, in concomitanza con iOS 6, Auto Layout. Auto Layout fornisce un sistema potente e versatile che descrive il modo in cui le view, i loro contenitori e il loro contenuto sono legati gli uni agli altri, offrendo un maggior controllo sul layout delle interfacce.

Il sistema si basa su toolkit di risoluzione dei vincoli Cassowary, sviluppato presso la University of Washington da Greg J. Badros e Alan Borning per affrontare i problemi legati alle interfacce utente.

Cassowary è stato pensato tenendo presente che, nelle interfacce utente, vi sono sempre relazioni di uguaglianza e disuguaglianza tra i componenti, per cui il toolkit utilizza una serie di regole che permettono allo sviluppatore di descrivere tali relazioni tra le view.

Le relazioni sono descritte mediante dei vincoli che limitano gli spazi che ciascuna view può occupare (ad esempio un determinato componente può occupare solo la metà sinistra dello schermo, o due view devono essere sempre allineate tra loro). Cassowary offre un risolutore automatico che trasforma i vincoli in regole geometriche che li esprimono.

Il toolkit ebbe grande successo, tanto che, dal suo debutto, è stato adattato a molti linguaggi, tra cui JavaScript, .NET/Java, Python, Smalltalk, C++ e, tramite Auto Layout, a Cocoa e Cocoa Touch. In iOS e OS X gli elementi grafici sono appunto organizzati mediante vincoli, i quali possono essere espressi sia tramite Interface Builder che da codice.

2

Descrizione generale dell'app

In questo capitolo vengono trattati il contesto dell'app e la sua descrizione generale. L'applicazione analizzata in questa tesi è nata per supportare le operazioni di marketing verso i clienti di un'importante azienda che opera nel settore alimentare.

2.1 Macro-architettura del sistema

L'applicazione è stata progettata per integrarsi perfettamente con le piattaforme digitali preestistenti dell'azienda. La macro-architettura del sistema è mostrata nella Figura 2.1.

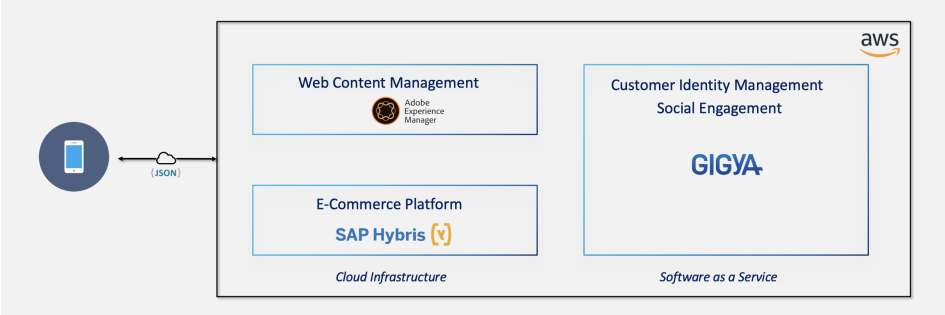

Figura 2.1. La macro architettura del sistema Applicazione - Digital Platform.

La piattaforma dell'azienda cliente sfrutta l'infrastruttura in cloud di Amazon AWS ed è composta da:

- *SAP* Hybris: software per la gestione di e-commerce e CRM facente parte della suite SAP Customer Experience, una serie di tool per la gestione delle relazioni con il cliente.
- *Gigya*: altra soluzione software facente parte del portafoglio SAP. Tramite Gigya vengono gestiti i dati sui clienti nonchè le operazioni di engagement e fidelizzazione. Gigya consente, inoltre, la gestione dei dati raccolti ed elaborati in

conformità con il GDPR, facilitando le operazioni di trasparenza verso l'utente sulla politica di memorizzazione dei suoi dati. Su questa piattaforma viene, di conseguenza, memorizzato l'account che l'utente utilizza per accedere all'app.
- *Adobe Experience Manager*: un insieme di strumenti per analytics, gestione dei social media, advertising, targeting e gestione dei contenuti interamente basata su cloud. Tramite Adobe Experience Manager è possibile creare, adattare e distribuire i propri contenuti attraverso diversi canali in modo mirato ed automatizzato.

La comunicazione tra l'applicazione e la piattaforma digitale avviene mediante servizi RESTful con payload in formato JSON; non è previsto il salvataggio di dati persistenti.

2.2 Progetto iniziale

In questa sezione vengono esposte le funzionalità previste dal progetto iniziale dell'app. Tali funzionalità sono legate principalmente al programma di loyalty dell'azienda e, in alcuni casi, sono state ampiamente ripensate, ampliate o rimosse.

2.2.1 Registrazione e Login

Dopo lo splashscreen dell'applicazione viene presentata all'utente una schermata da cui è possibile effettuare l'accesso o avviare il flusso di registrazione.
L'utente può scegliere se effettuare quest'ultima tramite la piattaforma Gigya o tramite un identity provider a scelta tra Amazon, Facebook e Google.
Per il corretto funzionamento dell'app è fondamentale che l'utente fornisca il consenso per la raccolta e l'elaborazione dei dati, per cui nella procedura di registrazione viene richiesta l'accettazione della privacy policy; in caso di rifiuto, la stessa viene riproposta dopo il login.

2.2.2 Onboarding

La funzionalità di onboarding ha lo scopo di proporre all'utente una serie di semplici domande tramite cui è possibile eseguire una profilazione dello stesso, consentendo all'applicazione di adattare l'esperienza, i contenuti e i prodotti proposti in base alle risposte date.

Durante la procedura si chiede all'utente, ad esempio, di selezionare i prodotti dell'azienda in suo possesso e il suo punto vendita preferito. Alla fine dell'onboarding vengono assegnati all'utente i suoi primi punti con cui potrà riscattare i premi del programma di loyalty.

2.2.3 Home

Dopo aver effettuato l'accesso viene presentata una schermata caratterizzata dalla presenza, nella parte bassa dell'app, di una barra di navigazione, tramite

la quale l'utente può spostarsi tra le varie sezioni dell'app selezionando l'icona corrispondente.

La prima sezione che viene mostrata è "Home". Qui l'utente può visualizzare offerte e promozioni a lui dedicate e sfogliare i premi che è possibile riscattare tramite i punti del programma di loyalty. Dalla home è possibile, inoltre, visualizzare notizie riguardanti iniziative, eventi e progetti dell'azienda cliente. Nella Figura 2.2 viene riportato un mockup della sezione.

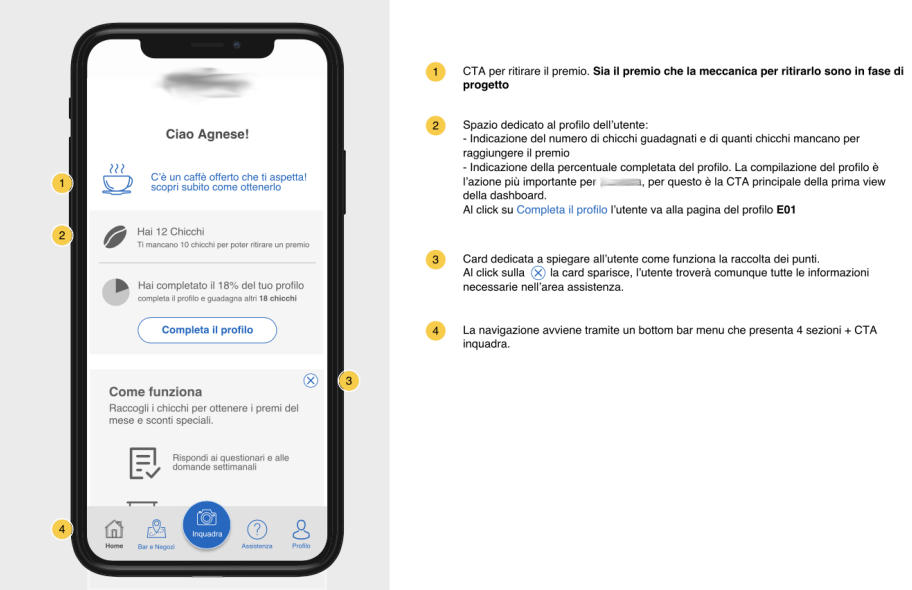

Figura 2.2. Mockup della sezione Home; nella parte inferiore della schermata viene mostrata la barra di navigazione.

2.2.4 Store Locator

Attraverso l'applicazione l'utente deve essere in grado di cercare i punti vendita associati all'azienda cliente, visualizzarne i contatti e impostare il proprio negozio preferito (se non ha fatto ciò durante la procedura di onboarding); egli può, inoltre, modificare la propria scelta.

L'applicazione è, altresì, in grado di fornire all'utente le indicazioni per raggiungere il punto vendita cercato.

2.2.5 Il tuo profilo

In questa sezione dell'app vengono visualizzate le informazioni relative al profilo dell'utente, ai premi riscattati e alle missioni del programma di loyalty. Nello specifico vengono mostrati i punti accumulati, i prodotti registrati dall'utente, le sue attività relative alla loyalty e, se disponibili, vengono proposte nuove missioni tramite cui è possibile guadagnare punti.

Nella Figura 2.3 è presente un mockup della sezione relativa al profilo

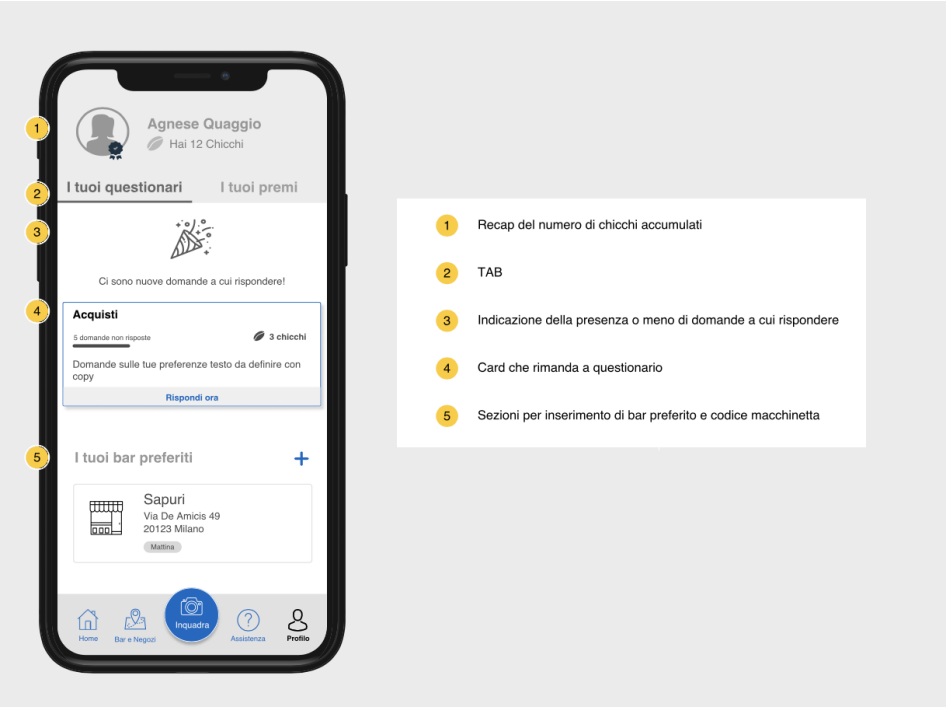

Figura 2.3. Mockup della sezione relativa al profilo.

2.2.6 Questionario

Periodicamente l'app propone la compilazione di questionari. Quando questi sono particolarmente corposi vengono suddivisi in diverse fasi al termine delle quali l'utente riceve un compenso, così da essere incentivato a rispondere a ogni domanda proposta.

2.2.7 Notifiche push

L'applicazione è in grado di ricevere notifiche push con cui viene segnalata all'utente la presenza di nuovi questionari o di missioni.

L'implementazione di questa funzionalità si basa sull'utilizzo della piattaforma Firebase Cloud Messaging come strumento per l'inoltro delle notifiche sui dispositivi prevenitvamente registrati.

Nella Figura 2.4 viene mostrato il flusso di registrazione al servizio.

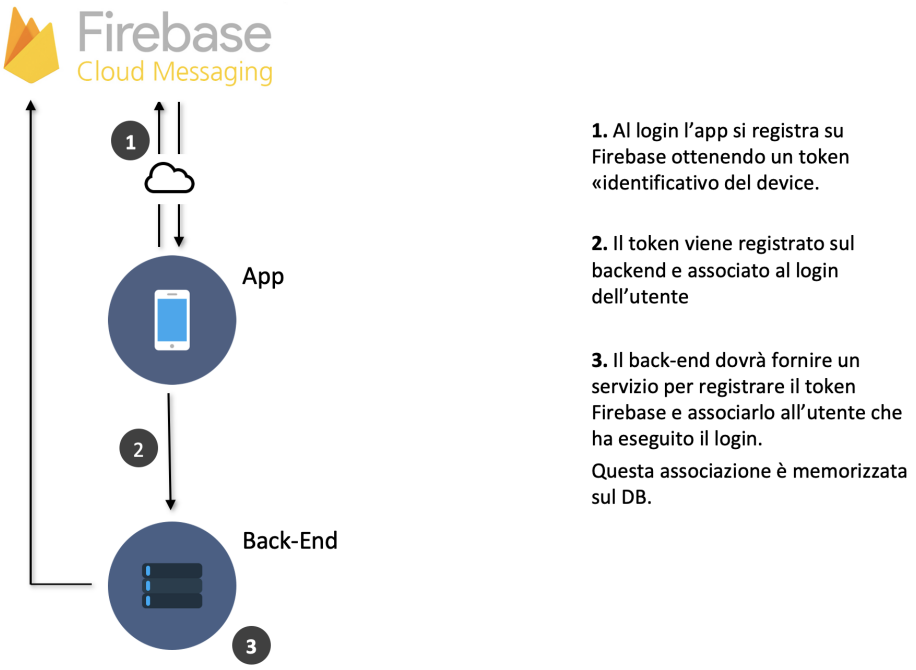

Figura 2.4. La procedura di registrazione alla piattaforma Firebase Cloud Messaging

Per inviare notifiche agli utenti il backend inoltra, tramite API, le richieste verso Firebase, che si occupa della consegna effettiva ai dispostivi.

2.2.8 Assistenza

L'app prevede una sezione dedicata all'assistenza clienti.

Attraverso un'interfaccia simile a quella di una chat l'utente viene guidato passo passo nella consultazione delle FAQ selezionando delle risposte predefinite ai messaggi di un assistente virtuale.
Nella Figura 2.5 viene mostrato un mockup della sezione.

Figura 2.5. Un mockup della funzione di assistenza

2.3 Struttura finale dell'applicazione

In questa sezione verrà esposta la struttura finale dell'app, enfatizzando le differenze con il progetto originale.

2.3.1 Home

Rispetto al progetto originale, "Home" è stata riorganizzata in modo tale da presentare immediatamente una card che collega al catalogo dei premi, i quali sono stati rimossi dalla pagina. Gli altri contenuti della sezione, ovvero le offerte, i prodotti in primo piano e le notizie importanti riguardanti gli eventi dell'azienda, sono stati organizzati in slider appositamente dedicati. In ciascuno di questi sono presenti le card che consentono di visualizzare i dettagli del relativo contenuto.

2.3.2 Store Locator

Lo Store Locator è stato inserito all'interno di una sezione più generica. In essa vi sono due card collegate ai suoi flussi. La prima indica il bar preferito e consente di cambiarlo, l'altra avvia la ricerca dei punti vendita.

Subito dopo le card relative allo Store Locator, ve ne sono altre cinque. Tre portano ad un elenco di prodotti di diverse categorie, le altre due sono dedicate rispettivamente alle notizie in primo piano e alla navigazione verso l'area "Magazine", che verrà approfondita nel Capitolo 3.

2.3.3 Punti e Premi

"Punti e premi" mostra, in primo piano, il saldo punti dell'utente e una guida sulle modalità di raccolta punti. In uno slider sono indicati i premi che è possibile ritirare con i punti che si hanno a disposizione, mentre, in fondo alla pagina, una card per la navigazione verso il catalogo premi completo.

2.3.4 Altre sezioni

La sezione "Profilo" non ha subito cambiamenti da un punto di vista concettuale; tuttavia l'interfaccia è stata ampiamente riprogettata. Tale sezione verrà trattata approfonditamente nel capitolo XXX.

Similmente, le altre sezioni sono andate incontro unicamente a una riprogettazione grafica, mentre lo sviluppo della funzionalità riguardante i questionari è stata rimandata ad una fase successiva del progetto.

2.4 Scelta delle tecnologie

La prima scelta cruciale nello sviluppo dell'applicazione è stata quella riguardante il linguaggio da adottare.

L'obiettivo era quello di raggiungere sia gli utenti di dispositivi Android, sia quelli di dispositivi Apple. Al giorno d'oggi è possibile scegliere diverse tecnologie per perseguire questo scopo; sono, infatti, disponibili molti framework per lo sviluppo di applicazioni cross-platform, che consentono di ridurre costi e tempi di sviluppo. Tuttavia, le performance delle app prodotte non sono affatto paragonabili a quelle ottenute mediante l'utilizzo dei linguaggi nativi. Sebbene vi siano diversi framework che traducono il codice scritto nel proprio linguaggio in linguaggio nativo, compensando, quindi, la scarsità di prestazioni, essi presentano spesso importanti limitazioni sull'utilizzo dei componenti messi a disposizione dagli SDK Android e iOS. Altra importante considerazione da fare in merito è quella riguardante l'integrazione di librerie di terze parti. Le librerie che supportano i framework per lo sviluppo cross-platform sono spesso meno soddisfacenti in termini di stabilità; inoltre, dovendosi adattare alle unicità dell'uno o dell'altro sistema, non riescono a sfruttare le feature in cui uno dei due sistemi eccelle rispetto all'altro.

Per tutte queste ragioni l'applicazione è stata sviluppata in linguaggio nativo, garantendo la compatibilità a partire dalla Versione 10 di iOS e dalla Versione 6 per il sistema operativo Android.

Questa scelta ha consentito di rispettare i più alti standard qualitativi in termini di performance, fluidità delle animazioni e grafica. Gli altri benefici ottenuti grazie all'adozione dei linguaggi nativi sono rappresentati dalla grande disponibilità

e facilità di integrazione delle librerie di terze parti, dalla minimizzazione della dimensione dell'applicazione sugli store, nonchè dalla capacità di sfruttare al meglio le feature uniche di entrambi i sistemi.

2.5 Concetti preliminari

In questa sezione verranno analizzati alcuni concetti preliminari che ci consentiranno di comprendere al meglio le spiegazioni tecnologiche, relative all'app realizzata, che verranno trattate nei prossimi capitoli.

2.5.1 Protocol

I *protocol* sono dei costrutti messi a disposizione da Swift mediante cui è possibile definire le proprietà e le firme dei metodi che una classe deve implementare. Si tratta di uno strumento molto versatile, infatti, oltre all'utilizzo appena esposto, può essere sfruttato come tipo nelle variabili per gestire la visibilità di determinate caratteristiche di una classe. Altro utilizzo interessante è costituito dalla possibilità di richiamare i metodi di una classe indipendentemente dall'implementazione della stessa; infine, è possibile estendere i *protocol* in modo da fornire l'implementazione di alcuni dei metodi di una classe e rendere quest'ultima disponibile in tutte le classi conformi ai *protocol* stessi.

2.5.2 UIViewController

Uno dei componenti chiave di *UIKit* è sicuramente *UIViewController*. Gli *UIViewController* sono oggetti che controllano una gerarchia di componenti grafici della propria app. Solitamente la classe *UIViewController* non viene utilizzata così com'è, ma viene estesa per adattarsi meglio alla gestione delle "View" che si desidera controllare, aggiungendo proprietà e metodi per facilitare il processo. Le principali responsabilità di uno *UIViewController* sono le seguenti:

- aggiornare le proprietà delle "View";
- rispondere alle interazioni dell'utente;
- ridimensionare le "View" e gestirne il layout.

Quando la visibilità delle "View" di uno *UIViewController* cambia, quest'ultimo chiama alcuni metodi che consentono alle sottoclassi di gestire i propri componenti grafici. Nella Figura 2.6 viene mostrato il cosiddetto "ciclo di vita" di uno *UIViewController*, con i metodi che vengono chiamati al verificarsi dei diversi eventi.

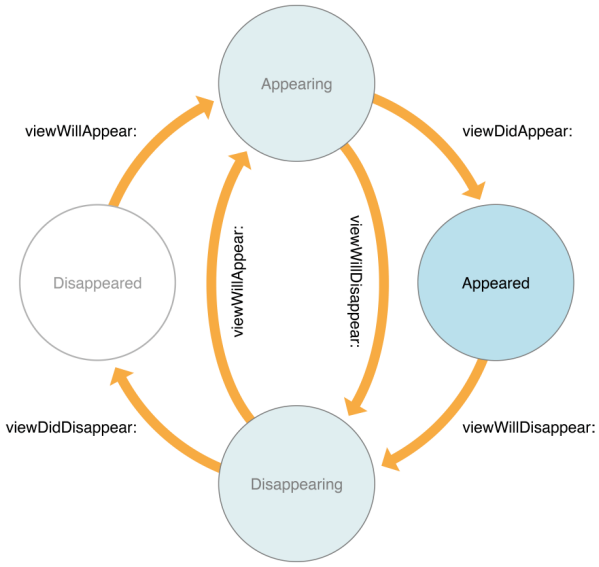

Figura 2.6. Il "ciclo di vita" di uno *UIViewController*.

Nel momento in cui una "View" sta per apparire o scomparire dallo schermo vengono chiamati, rispettivamente, i metodi *ViewWillAppear* e *ViewWillDisappear*; in modo analogo, quando una view termina l'apparizione o la scomparsa viene effettuata la chiamata a *ViewDidAppear* e *ViewDidDisappear*. Nel ciclo di vita di una "View" è incluso, anche, un altro metodo, non riportato in Figura 2.6, denominato "ViewDidLoad" e richiamato solo una volta in seguito alla creazione della "View".

2.6 Architettura dell'applicazione

Sin dal rilascio dell'SDK per Swift, Apple ha incenvitvato l'utilizzo del pattern architetturale Model-View-Controller (MVC) nelle applicazioni di terze parti. La sua semplicità di implementazione ha contribuito all'ampia adozione di questo pattern che, tuttavia, ha rivelato rapidamente i suoi limiti. Questa architettura offre, infatti, se applicata ai componenti dell'SDK iOS, un livello poco soddisfacente di disaccopiamento del codice, compromettendo la manutenibilità, la leggibilità e la testabilità. Tali ragioni hanno portato all'adozione di un pattern più elaborato, facente parte della famiglia MVX. Tale pattern è il Model-View-ViewModel (MVVM). Nelle prossime sottosezioni illustreremo in dettaglio questi due pattern.

2.6.1 Il pattern MVC

Dopo aver riflasciato Swift, Apple ha incoraggiato l'utilizzo di MVC per lo sviluppo delle applicazioni, proponendo una propria visione del pattern. Nel pattern Model-View-Controller tradizionale vi sono i seguenti componenti:

- *Model*: fornisce i metodi per l'accesso ai dati utilizzati nell'applicazione;
- *View*: si occupa della visualizzazione dei dati;
- *Controller*: renderizza la *View* e la aggiorna in base al cambiamento di stato del *Model*.

Questa implementazione di MVC non si adatta allo sviluppo di applicazioni iOS in quanto i tre componenti sono strettamente accoppiati, ragione per cui Apple ha introdotto un'evoluzione del pattern che si adattasse meglio al proprio linguaggio. Come mostrato nella Figura 2.7, il "Controller" fa da intermediario tra "View" e "Model", in modo che questi ultimi non abbiano conoscenza l'uno dell'altro.

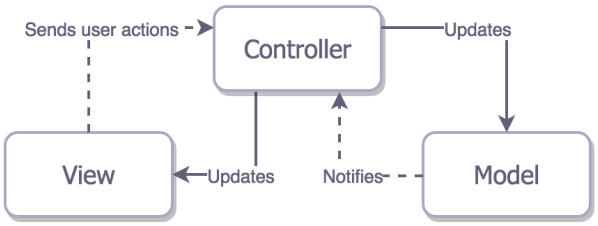

Figura 2.7. L'evoluzione di MVC proposta da Apple.

Il modello proposto da Apple ha, però, portato a un disaccoppiamento solo parziale dei componenti; infatti, la classe *UIViewController* è strettamente legata al ciclo di vita delle "View", e il risultato ottenuto è più simile a quello mostrato nel diagramma in Figura 2.8.

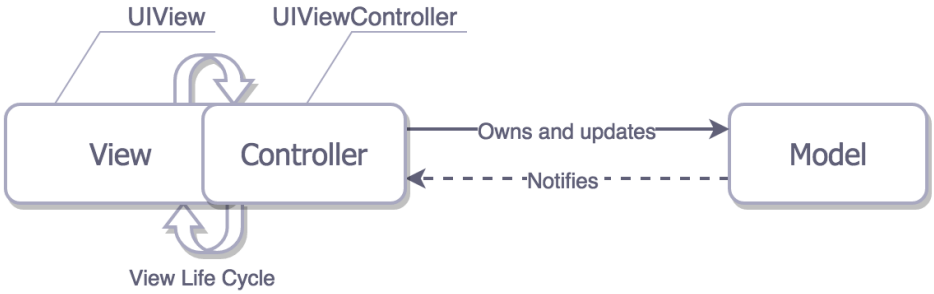

Figura 2.8. L'architettura effettivamente ottenuta applicando alle app Swift il modello di MVC introdotto da Apple.

Le implementazioni dei "ViewController" risultano di dimensioni molto elevate, tanto che nella comunità degli sviluppatori iOS questo pattern è conosciuto come "Massive-View-Controller". L'accoppiamento tra "View" e "ViewController" rende, inoltre, questi ultimi meno agevoli da testare, in quanto la "business logic" è fortemente dipendente dall'interfaccia, e, quindi, difficilmente riproducibile negli "Unit test".

2.6.2 MVVM

Il pattern architetturale utilizzato nella realizzazione dell'applicazione in analisi rappresenta una delle più recenti evoluzioni del pattern Model-View-Controller, ovvero il Model-View-ViewModel. Nella Figura 2.9 è riportato un diagramma che mostra il funzionamento di questo pattern.

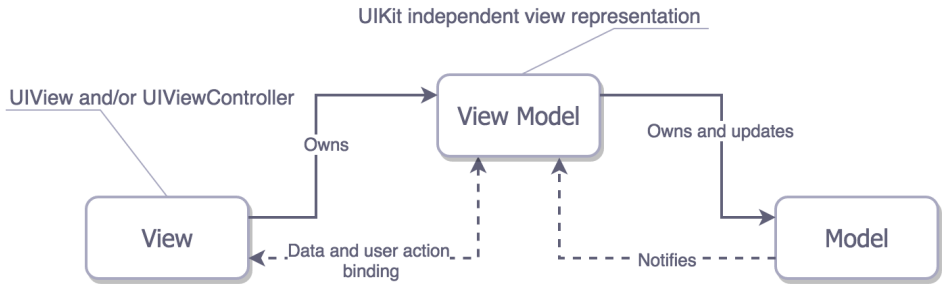

Figura 2.9. Il pattern MVVM.

Il "ViewModel" si pone come un mediatore tra "View" e "Model"; questi ultimi risultano disaccoppiati in quanto non conoscono nulla l'uno dell'altro. Nel contesto dello sviluppo iOS la caratteristica fondamentale di questo pattern risiede nel fatto che il "ViewModel" è una reppresentazione dello strato "View" totalmente indipendente da *UIKit*, mentre gli oggetti di tipo *UIViewController* sono considerati parte integrante della "View". Il "Model" viene aggiornato dal "ViewModel", il quale contiene la logica di business e mantiene una copia dei dati di interesse per la "View". Quando i dati memorizzati nel "ViewModel" si modificano i cambiamenti si propagano alla "View" mediante un meccanismo di "binding" dei dati. L'adozione del "MVVM pattern" ha apportato notevoli benefici a livello di:

- *Distribuzione*: le responsabilità di ciascun componente sono ben delineate, le classi sono più snelle e più facilmente leggibili e manutenibili. L'accoppiamento tra gli strati "Controller" e "View" è stato totalmente rimosso.
- *Testabilità*: la totale indipendenza del "ViewModel" da *UIKit* e l'assenza di conoscenza della "View" consente di riprodurre agevolmente i flussi logici dell'applicazione, velocizzando e facilitando la scrittura di "unit test".

L'implementazione di MVVM utilizzata nell'app introduce un ulteriore livello di separazione. La comunicazione con i servizi REST del backend è, infatti, affidata a degli oggetti denominati "Interactor", i quali hanno il compito di gestire le chiamate ai servizi e passare le risposte di questi ultimi ai "ViewModel", che elaborano i dati o gestiscono gli errori ricevuti.I rusiltati di queste elaborazioni vengono poi memorizzati nella proprietà *viewState*.

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        if segue.identifier == "HistorySegue" {
            if let viewController = segue.destination as? HistoryController {
              if(barcodeInt != nil){
                viewController.detailItem = barcodeInt as AnyObject
              }
            }
        }
    }
```

Listing 2.1. Un esempio di "Segue" istanziato in un "ViewController" che passa un dato al "ViewController" di destinazione.

2.7 Flussi di navigazione e Flow Coordinator

Uno degli aspetti fondamentali dello sviluppo di applicazioni per smartphone è rappresentato dalla navigazione tra le diverse schermate di un'app. I pattern architetturali presentati nella sezione precedente forniscono una struttura per la propria applicazione; tuttavia non coprono l'aspetto della navigazione. Pertanto, è stato necessario studiare e scegliere una soluzione che consentisse una transizioni agevoli tra una schermata e quella successiva.

Il sistema di navigazione standard fornito da Swift fa affidamento sugli "Storyboard" e su due componenti di *UIKit* denominati *UINavigationController* e *UIStoryboardSegue*. Per determinare la transizione da un "ViewController" ad un altro è necessario collegarli sullo Storyboard mediante i cosiddetti "Segue" ,e definire la sorgente e la destinazione di un oggetto di tipo *UIStoryboardSegue* all'interno di un "ViewController". La navigazione vera e propria avviene mediante lo *UINavigationController*, il quale conserva uno stack di "ViewController" che verranno mostrati all'esecuzione di operazioni di "push" o "pop" su quest'ultimo. Tali operazioni vengono chiamate quando viene eseguito il metodo *perform* di uno *UIStoryboardSegue*.

Il sistema dei "Segue" presenta due grossi limiti. Il primo è quello di rendere difficoltoso il riutilizzo, poichè lo sviluppatore è costretto a collegare graficamente all'interno di uno "Storyboard" i controller tra cui navigare. Questa situazione scoraggia i riferimenti a "Storyboard" esterni per non rendere troppo complessa la gestione dei flussi di navigazione. Il secondo limite consiste nell'obbligo di dover istanziare il "ViewController" di destinazione all'interno di un altro "ViewController", che, quindi, è costretto a gestire anche il passaggio dei dati tra una schermata e l'altra. Nel Listato 2.1 viene mostrata una porzione di codice che istanzia un "Segue".

Per superare questi limiti è stato adottato un pattern più innovativo, denominato "Flow Coordinator". Con questo pattern, la gestione della navigazione viene affidata a degli oggetti, denominati "Coordinator", che si occupano istanziare i "ViewController" e inserirli nello stack di uno *UINavigationController*. Un oggetto di tipo "Coordinator" è in grado di gestire un intero flusso di navigazione in maniera totalmente indipendente dagli "Storyboard" e fa in modo che un "ViewController" non conosca nulla degli altri. Nel Listato 2.2 viene mostrato il "protocol" che definisce i metodi che ogni "Coordinator" deve implementare.

```
1   protocol Coordinator: AnyObject {
2       var childCoordinators: [Coordinator] { get set }
3       var navigationController: UINavigationController? { get set }
4
5       var readyToBeRemoved: (() -> Void)? { get set }
6       func start()
7   }
```

Listing 2.2. Il protocol “Coordinator”.

Ogni “Coordinator” memorizza, a sua volta, una serie di oggetti dello stesso tipo, così da poter gestire unicamente un caso d’uso e affidare gli altri ai “Coordinator” figli. Questo consente non solo di snellire il codice di ciascuna classe, ma anche di richiamare un intero flusso di navigazione semplicemente istanziando il relativo “Coordinator”. L’attributo navigationController non è altro che uno *UINavigationController* che viene passato tra tutti i “Coordinator” e che mantiene l’intero stack di navigazione dell’applicazione. La closure *readyToBeRemoved* viene valorizzata dal “Coordinator” padre, che memorizza il nuovo “Coordinator” figlio, ed è utilizzata per rimuoverlo dai “childCoordinators” quando necessario. Infine, il metodo *start* viene utilizzato per lanciare il “Coordinator”, istanziando il suo primo “ViewController” e inserendolo nello stack dello *UINavigationController*.

3

Progettazione e implementazione del componente “Magazine”

Nel progetto finale dell’app si è deciso di riunire in un’unica sezione lo store locator, il catalogo prodotti e la funzionalità “Magazine”. Tale sezione è stata chiamata “Store”.

In questo capitolo verranno approfonditi i contenuti e la struttura della funzionalità Magazine.

3.1 Progettazione

Si vuole fornire all’utente dell’app una sezione, denominata “Magazine”, che gli permetta di conoscere le ultime novità sulle iniziative e gli eventi riguardanti l’azienda cliente.

All’interno di “Magazine” potranno essere pubblicati, sotto forma di articoli, contenuti di diversa natura. Gli articoli da mostrare saranno caricati sulla piattaforma Hybris dal personale dell’azienda cliente e, successivamente, esposti nell’app che li recupererà mediante i servizi REST.

Poichè per la stesura degli articoli verrà utilizzato l’editor What You See Is What You Get (WYSIWYG) incorporato in Hybris, l’applicazione dovrà essere in grado di riconoscere i tag HTML presenti nei documenti ricevuti e renderizzare un contenuto che sia coerente con gli stessi. Gli articoli potranno includere contenuti multimediali, quali, ad esempio, video, immagini singole o raccolte in una gallery. In quest’ultimo caso dovrà essere possibile aprire una schermata dedicata alla visualizzazione in sequenza di tutte le immagini della gallery, mentre i video dovranno essere gestiti presentando un player integrato per poterli riprodurre.

A ciascun articolo dovrà essere assegnata una categoria, in base alla quale i contenuti potranno essere raggruppati e riconosciuti come correlati tra loro. Questa classificazione consentirà all’utente di filtrare i contenuti di Magazine, visualizzando quelli di maggior interesse al momento della consultazione; inoltre, alla fine di ogni articolo, ne verranno suggeriti altri della medesima categoria.

Sarà possibile filtrare il contenuto di “Magazine” selezionando o escludendo una o più categorie contemporaneamente. In questa fase dello sviluppo si è deciso di adottare tre categorie, denominate rispettivamente “Coffee Culture”, “Art and Culture” e “Inspiring Cooking”. Magazine sarà divisa in più sottosezioni, ognuna delle

quali includerà gli articoli di una determinata categoria. La prima sottosezione, denominata “News in primo piano”, mostrerà gli articoli più rilevanti e sarà seguita da “Scopri nuove ricette” in cui verranno visualizzati i contenuti della categoria “Inspiring Cooking”. L’ultima sezione, chiamata “Altre news”, ospiterà, invece, tutti gli altri articoli disponibili.

Per incentivare ulteriormente la fruizione dei contenuti di Magazine, dovrà essere mostrato un articolo nella sezione “Store”. Tale news verrà contrassegnata come “In primo piano”.

3.2 Realizzazione grafica

L’interfaccia dell’applicazione presenta un elemento grafico di cui è stato fatto ampio uso. Tale elemento è la card e verrà introdotto nella prossima sottosezione prima di illustrare il suo utilizzo.

3.2.1 Card

Le card sono un elemento fondamentale nel design delle applicazioni moderne. Sono semplici, intuitive e molto apprezzate da un punto di vista estetico.

Il principio su cui si basano è quello di presentare contenuti all’utente evitando di mostrare lunghi testi, in modo simile a quello che avviene tradizionalmente mediante i biglietti da visita. Questa somiglianza contribuisce molto all’intuitività delle card, infatti ogni utente sa già come utilizzarle, in quanto già familiare con l’utilizzo dei biglietti da visita, comunemente usati per trasmettere informazioni puntuali come indirizzi, numeri di telefono o statistiche.

Ogni card ha, di norma, dei bordi in rilievo che la delimitano, così da avere una somiglianza con le card fisiche anche dal punto di vista grafico.

Il vantaggio offerto dalle card non è solo di carattere estetico, ma anche pratico. Queste risultano infatti molto efficaci nel guidare l’utente attraverso i flussi di navigazione, poichè permettono di isolare immediatamente delle informazioni specifiche, aiutando a trovare più velocemente i contenuti di proprio interesse.

Un’altra caratteristica fondamentale delle card è la loro versatilità. Vi sono, ad esempio, applicazioni che le utilizzano per mostrare i prodotti in vendita in un sito e-commerce, informazioni riepilogative su un particolare argomento o contenuti multimediali. Altre app sfruttano le card come Call To Action (CTA), ovvero elementi grafici che invitano l’utente a intraprendere azioni quali visitare una pagina, effettuare un acquisto o richiedere informazioni. Esempi importanti di applicazioni che hanno deciso di adottare le card sono AliExpress, Trello, SkyNews e Pinterest.

3.2.2 Magazine

Come da specifiche la sezione Magazine è stata suddivisa in tre sottosezioni, contrassegnate con un diverso colore di sfondo. Tali sottosezioni sono:

- News in primo piano;

- Scopri nuove ricette;
- Altre News.

Ognuna di queste sezioni ospita delle card con un'anteprima degli articoli. Nella parte alta delle card è presente una label con la categoria dell'articolo seguita dal titolo e da un'immagine di sfondo.

Le sottosezioni sono poste verticalmente una dopo l'altra e l'utente può scorrerle in sequenza per visualizzarne i contenuti.

Per la funzione di filtraggio degli articoli è presente un pulsante premendo il quale viene mostrata una schermata in cui è possibile selezionare le categorie degli articoli che si vogliono visualizzare. La lista dei filtri viene mostrata in modale, ovvero con un'animazione a comparsa dal basso verso l'alto e in sovrapposizione alla schermata precedente.

Figura 3.1. La sezione "Magazine".

Figura 3.2. La schermata filtro della sezione “Magazine”.

Nelle figure 3.1 e 3.2 sono mostrate rispettivamente la schermata di Magazine e il modale per filtrare gli articoli.

Al tap dell’utente sulla card di un articolo viene visualizzata, ancora una volta in modale, la schermata con il testo integrale dell’articolo, le immagini e, se prevista, una galleria, mostrata come una collezione di card a scorrimento orizzontale. In fondo sono proposti gli articoli correlati. Anche questi ultimi sono raggruppati in una collezione di card dall’aspetto differente a quello delle gallery, come visibile in Figura 3.3.

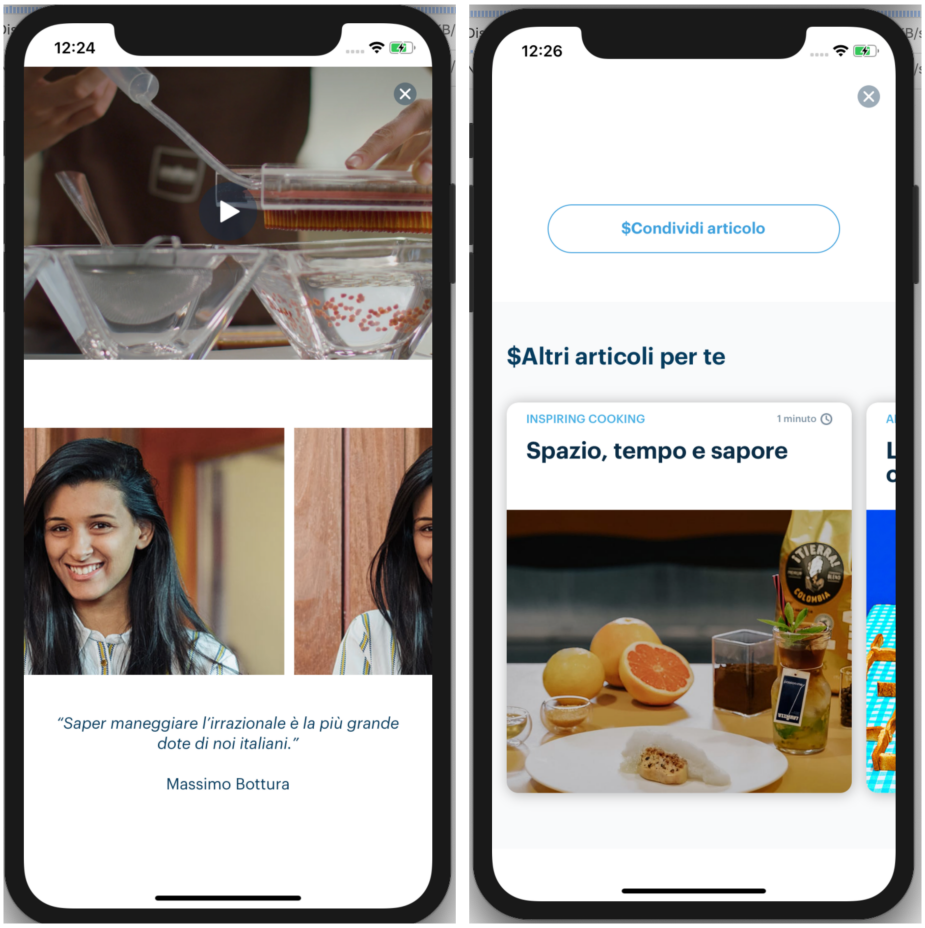

Figura 3.3. A sinistra è riportata una galleria di immagini all'interno di un articolo, a destra una *UICollectionView* che mostra gli articoli correlati.

3.3 Implementazione

Verranno ora approfonditi gli aspetti implementativi, i componenti e le tecniche utilizzate per la realizzazione della sezione e il soddisfacimento delle specifiche.

3.3.1 Schermata principale e UITablieView

Come già spiegato, la schermata principale di Magazine è composta da tre sezioni verticali, ciascuna contenente diverse card, a loro volta disposte verticalmente.

La presenza di questo tipo di layout e di collezioni di elementi simili tra loro suggerisce, in modo naturale, l'utilizzo di un componente di UIKit chiamato UITablewView. UITableView presenta una singola colonna di contenuti suddivisa in righe

che scorrono verticalmente. Le righe correlate tra loro possono essere raggruppate in sezioni.

Caratteristica molto importante delle UITableView è quella di “riciclare” gli oggetti utilizzati per immagazzinare le celle della tabella, migliorando notevolmente sia la fluidità della view che l’utilizzo della memoria del dispositivo.

Ad ogni cella viene assegnata una classe o un file *.xib* che ne indicano le caratteristiche. Successivamente, per rendere la cella disponibile al riutilizzo e per comunicare alla *UITableView* come creare nuove celle, viene chiamato il metodo *dequeueReusableCell*, a cui viene passato un indentificatore assegnato alla classe della cella. Una volta chiamato il metodo, gli oggetti cella creati vengono inseriti all’interno di una coda i cui elementi vengono usati dalla *UITableView* ogni volta che sarà necessaria una nuova cella. Solo nel caso in cui non siano disponibili elementi del tipo richiesto, ne verrà creato uno nuovo.

Come molti componenti di *UIKit* la *UITableView* fa uso del delegation pattern; questo verrà illustrato nella prossima sezione.

3.3.2 Delegation Pattern

Un delegato è qualcuno designato per agire al posto di altri. Il Delegation Pattern riprende questo concetto adattandolo al contesto dell’ingegneria del software. In questo pattern un oggetto espone un comportamento, delegando la responsabilità di implementarlo ad un altro oggetto ad esso associato. Nella figura 3.4 è possibile osservare un diagramma che schematizza il funzionamento del Delegation Pattern.

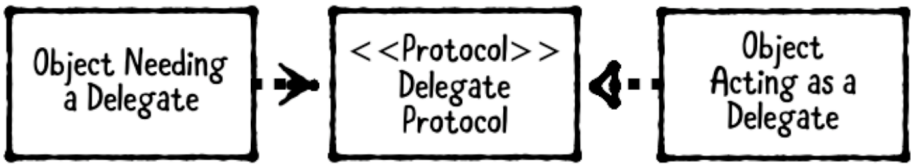

Figura 3.4. Un diagramma che illustra il funzionamento del Delegation Pattern.

Come si evince dalla figura, nel Delegation Patterm sono presenti tre componenti:

- un oggetto che ha bisogno di un delegate;
- un’interfaccia o protocol, dove si definiscono i metodi che un delegate deve, o dovrebbe, implementare;
- un delegate, ovvero l’oggetto che implementa i metodi definiti nel protoccolo del punto precedente.

L’utilizzo del Delegation Pattern consente di ottenere, oltre ad una maggiore separazione delle responsabilità, un più alto grado di flessibilità e riutilizzo degli oggetti. Facendo affidamento su un protocol, infatti, si può utilizzare come delegate qualunque oggetto che implementi i metodi necessari. Per incrementare ulteriormente la separazione delle responsabilità i componenti dei framework di Swift, in

```
1   extension MagazineLandingViewController: UITableViewDelegate, UITableViewDataSource, CellDequeuer {
2
3       func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
4           return viewState.listOfSections[section].sectionItems.count
5       }
6
7       func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
8
9           let cell: MagazineTableViewCell = dequeueCell(tableView: tableView, indexPath: indexPath)
10
11          cell.selectionStyle = .none
12
13          let data = viewState.listOfSections
14
15          let item = data[indexPath.section]
16          let magazineItem = item.sectionItems[indexPath.row]
17          cell.setFromMagazineItem(item: magazineItem)
18
19          return cell
20      }
21  }
```

Listing 3.1. Alcuni metodi del delegate per la UITableView della sezione Magazine

special modo UIKit, sono soliti utilizzare due oggetti che implementano il Delegation Pattern: uno chiamato "DataSource", che ha il compito di fornire dati, l'altro chiamato "Delegate", che, solitamente, definisce dei comportamenti specifici per un oggetto. È, ad esempio, compito del DataSource generare le sezioni della tabella e le relative celle con il loro contenuto, mentre il Delegate definisce le loro caratteristiche grafiche, nonchè le operazioni da eseguire all'occorrenza di eventi, quali il tap su una cella, lo scorrimento, la comparsa o la scomparsa delle stesse.

Nel Listato 3.1 possiamo vedere l'implementazione di due metodi del delegate per la UITableView presente nella sezione magazine.

Il primo restituisce alla tabella il numero delle celle da creare per la sezione corrispondente all'indice passato come valore del parametro section, mentre il secondo crea la cella e la popola con i dati contenuti nella variabile "data". Questa non è altro che una lista di articoli, a sua volta contenuta nella proprietà *viewState* di *MagazineLandingViewController*, contenente la lista delle sezioni del magazine e i relativi articoli. Questo delegate implementa, inoltre, il protocollo *CellDequeuer*, il quale come è possibile osservare nel Listato 3.2, offre un metodo per ottenere l'oggetto cella di una UITableView utilizzando una sintassi molto più snella, facendo uso dei generics.

Il metodo *dequeueCell* riceve il tipo T dell'oggetto cella, la *UITableView* a cui questa appartiene e il suo indice all'interno della tabella. Dal tipo T viene ricavato il nome della classe della cella e, come di consueto, viene chiamato il metodo *dequeueReusableCell* e restituito, infine, l'oggetto creato da quest'ultimo.

3.3.3 Filtro articoli

Nella schermata modale per il filtraggio degli articoli di Magazine vengono elencate le categorie in base a cui è possibile filtrare la lista degli articoli. In questa schermata la UITableView è stata utilizzata per contenere le categorie, e le sue celle sono state disegnate all'interno di un file *.xib*. Ogni cella contiene una checkbox e un'etichetta con il nome della relativa categoria. Per ottenere il comportamento desiderato per la checkbox è stato utilizzato il componente *UIImageView*, che consente di visualizzare

```
protocol CellDequeuer {
    func dequeueCell<T: UITableViewCell> (tableView: UITableView, indexPath: IndexPath) -> T
}
extension CellDequeuer{
func dequeueCell<T: UITableViewCell> (tableView: UITableView, indexPath: IndexPath) -> T {
        // this pulls out "MyApp.MyViewController"
        let fullName = NSStringFromClass(T.self)

        // this splits by the dot and uses everything after, giving "MyViewController"
        let className = fullName.components(separatedBy: ".")[1]

        guard let cell = tableView.dequeueReusableCell(withIdentifier: className, for: indexPath) as? T
            else {
                fatalError("\(className) must have the right identifer")
        }

        return cell
    }
}
```

Listing 3.2. Il protocollo *CellDequeuer*

un’immagine. Quest’ultima può cambiare a seconda dello stato del componente stesso.

In questo caso, quando la *UIImageView* viene selezionata mediante il tap da parte dell’utente, si passa dal visualizzare l’immagine di una checkbox vuota al mostrare una spunta. Gli stati di ogni singola checkbox vengono memorizzati all’interno di un oggetto che, alla chiusura del modale, viene passato alla schermata precedente, la quale si occuperà di nascondere o mostrare i contenuti desiderati.

3.3.4 Dettaglio articolo

La schermata di dettaglio degli articoli viene presentata come modale della sezione Magazine. Per prima cosa viene mostrata l’intestazione dell’articolo, che comprende una label con la categoria, il titolo e il sottotitolo dell’articolo, seguiti da un’immagine o un video e un testo introduttivo.

Per quanto riguarda il corpo dell’articolo si è deciso di utilizzare nuovamente una UITableView, in modo tale da offrire la possibilità di dividere l’articolo in paragrafi e superare i limiti del component *UITextView*, il quale non offre la possibilità di mostrare immagini. Le celle della tabella che rappresenta il corpo dell’articolo sono composte da tre elementi:

- un testo superiore;
- un’immagine;
- un testo inferiore;

Ciascuno di questi può essere nullo, in modo tale da offrire maggiore flessiblità nella struttura dell’articolo.

Per supportare la visualizzazione di testi HTML formattati con i relativi tag non sarebbe stato possibile utilizzare la classe *String*, poichè gli oggetti costruiti mediante quest’ultima sono uniformi a livello di font. Per questa ragione è stata scelta la classe *NSAttributedString*, che consente di creare stringhe in cui ogni singolo carattere può essere formattato in maniera indipendente dal resto della stringa. Per snellire il codice e poter creare rapidamente le stringhe a partire dal testo HTML,

```
1   extension String {
2       init?(htmlEncodedString: String) {
3           guard let data = htmlEncodedString.data(using: .utf8) else {
4               return nil
5           }
6           let options: [NSAttributedString.DocumentReadingOptionKey: Any] = [
7               .documentType: NSAttributedString.DocumentType.html,
8               .characterEncoding: String.Encoding.utf8.rawValue
9           ]
10          guard let attributedString = try? NSAttributedString(data: data,
11                                                               options: options,
12                                                               documentAttributes: nil) else {
13              return nil
14          }
15          self.init(attributedString.string)
16      }
17  }
```

Listing 3.3. L'estensione della classe *String* che consente di creare stringhe formattate in HTML

ricevuto dalle chiamate al servizio di backend, è stata creata un'estensione della classe *String* che possiamo osservare nel Listato 3.3.

L'estensione consiste in un nuovo costruttore che riceve come parametro la stringa codificata in HTML e che, specificando il formato HTML e l'encoding UTF8, richiama il costruttore di *NSAttributedString*. Il risultato di questa chiamata viene passato ad un altro costruttore della classe *String*, in grado di restituire il testo desiderato.

Terminato il corpo dell'articolo viene mostrata, se l'articolo lo prevede, una galleria di immagini. Per implementare questo elemento è stato scelto un componente di *UIKit* chiamato *UICollectionView*. Questo è concettualmente molto simile alla *UITableView*, tuttavia, esso presenta una maggiore flessibilità dal punto di vista del layout, che può essere personalizzato a seconda delle esigenze dello sviluppatore e non è più limitato ad una sola colonna a scorrimento verticale. Le *UICollectionView* sono ad esempio utilizzate nell'app foto di Apple. L'ultimo elemento della schermata di dettaglio degli articoli è un'altra UICollectionView, questa volta contenente un elenco di articoli correlati a quello appena letto. Sia nel caso delle gallery che in quello degli articoli correlati gli elementi della Collection sono costituiti da card.

3.3.5 Riproduzione dei video

Per la riproduzione dei video presenti nell'intestazione degli articoli è stata creata una schermata apposita che sfrutta il componente *WKWebView*. Questo è in grado di mostrare un URL all'interno di un applicazione e riceve come parametro del suo costruttore un oggetto di tipo URL. In questo sono memorizzati l'host, il path e gli eventuali parametri dell'indirizzo da visualizzare.

Figura 3.5. Un video riprodotto all’interno dell’applicazione.

Nel nostro caso passeremo l’indirizzo di video caricati sulla piattaforma YouTube, sfruttando la modalità embedded che consente di riprodurre i filmati direttamente da una pagina esterna, offrendo anche i controlli per lo scorrimento e la modalità schermo intero. Nella Figura 3.5 viene mostrata la “webview” per la riproduzione dei video.

4

Progettazione e implementazione del componente “Impostazioni”

In questo capitolo verranno approfonditi i dettagli più rilevanti del componente “Impostazioni”.

4.1 Progettazione

Si vuole consentire all’utente di modificare i dati del proprio account *Gigya*. Questi dati sono, nello specifico, il nome e il cognome dell’utente, la sua foto profilo e la sua password. La modifica avverrà in una sezione dedicata, denominata “Impostazioni”. Oltre alle informazioni di base, dovrà essere possibile, per soddisfare i requisiti del “GDPR”, modificare il consenso alla raccolta, memorizzazione ed elaborazione dei dati dell’utente per fini pubblicitari e di profilazione. La sezione comprenderà tre voci, indicanti, rispettivamente i permessi concessi dall’utente relativamente all’accesso ai servizi di geolocalizzazione del telefono, alla fotocamera e al servizio di notifiche push. Dovrà essere possibile modificare l’autorizzazione per ciascuna delle tre voci. Da “Impostazioni” sarà, infine, possibile effettuare il logout, aprire ad una pagina contenente i contatti dell’azienda cliente e accedere all’a ssistenza.

4.2 Realizzazione grafica

La sezione “Impostazioni” è stata realizzata dando massima importanza, dal punto di vista grafico, a semplicità ed intuitività. Per questa ragione l’aspetto della schermata principale è stato reso il più simile possibile a quello delle impostazioni di sistema di iOS. Queste sono caratterizzate da una serie di righe contenenti un’etichetta, separate tra loro da una linea molto sottile di colore grigio. Cliccando sulla riga desiderata si accede ad una schermata più specifica in cui è possibile modificare i parametri dell’impostazione selezionata.

L’accesso alla sezione “Impostazioni” dell’app avviene mediante tap su un’icona a forma di ingranaggio presente nella sezione “Profilo”. L’aspetto della schermata principale di “Impostazioni” è nostrato nella Figura 4.1.

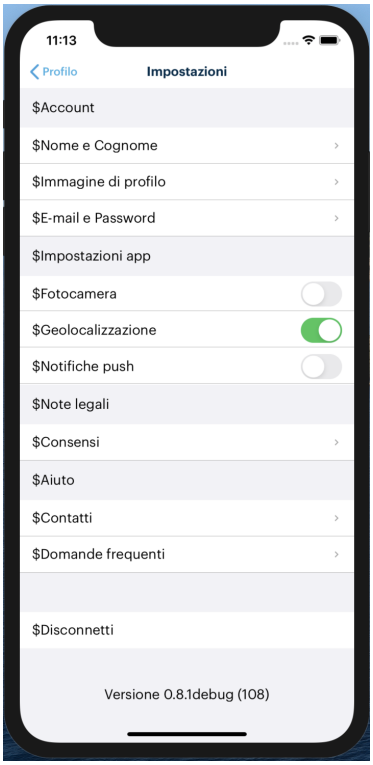

Figura 4.1. La schermata principale di “Impostazioni”.

Nel gruppo “Impostazioni app” è possibile notare le tre voci relative ai permessi di “Fotocamera”, “Geolocalizzazione” e “Notifiche push”. Queste sono state rese differenti dalle altre opzioni in modo da mostrare immediatamente lo stato dell’autorizzazione.

4.3 Implementazione

Verranno, ora, discusse le scelte implementative approfondendo la descrizione dei i componenti di cui si è fatto uso nella sezione “Impostazioni”.

4.3.1 La schermata principale

Sebbene il layout mostrato nella Figura 4.1 potrebbe far pensare, ancora una volta, all’utilizzo di *UITableView*, si è optato per un’alternativa diversa, ovvero il componente *UIStackView*. Questo consente di disporre diversi elementi all’interno di una “view”, gestendo i vincoli di posizionamento richiesti da *Autolayout*. I contenuti di una *UIStackView* verranno, per questa ragione, adattati automaticamente alle dimensioni e all’orientamento dello schermo del dispositivo.

La classe *UIStackView* possiede le seguenti proprietà:

- *axis*, che determina la disposizione verticale o orizzontale degli elementi che contiene.
- *distribution*, che specifica il modo in cui gli elementi occupano lo spazio della "UIStackView". Ad esempio si può scegliere la modalità "fill", che ridimensionerà gli elementi in modo da occupare l'intero spazio della "UIStackView" che li contiene.
- *spacing*, per regolare la distanza, lungo l'asse specificato in *axis*, tra un elemento e quello successivo.
- *alignment*, per impostare i margini lungo l'asse perpendicolare a *axis*.

Nella Figura 4.2 è riportata una rappresentazione grafica dell'effetto che queste proprietà hanno sulla *UIStackView*.

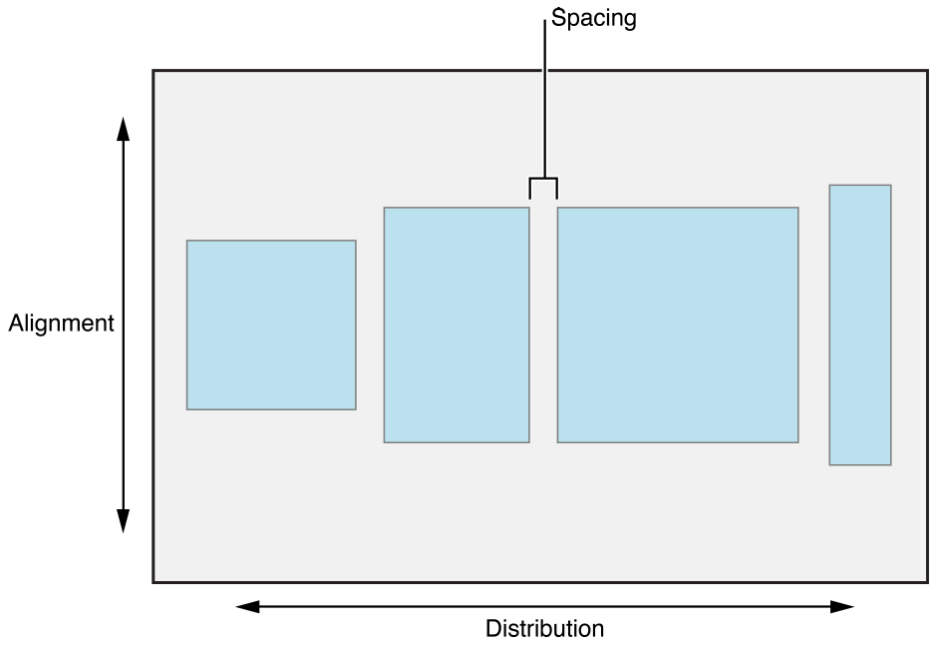

Figura 4.2. Le proprietà del componente *UISTackView*

La *UIStackView* è stata preferita alla *UITableView* in quanto gli elementi da mostrare sono ben definiti a priori e il loro contenuto non viene recuperato mediante chiamata REST ai servizi del backend; inoltre c'è una bassa probabilità che le voci dell'elenco delle impostazioni cambino in futuro. Un'ulteriore ragione per cui è stata adottata la *UIStackView* è l'eterogeneità delle operazioni da eseguire al tap sui suoi elementi. Se fosse stata utilizzata una *UITableView*, la gestione di tali operazioni sarebbe stata affidata a un "delegate". Questo si sarebbe basato su numerosi controlli condizionali sugli indici degli elementi nella tabella, il che avrebbe portato ad avere un codice molto meno leggibile e più difficile da mantenere.

Gli elementi della *UIStackView* sono stati disegnati all’interno di due file “.xib” differenti, uno per le voci relative alle autorizzazioni per fotocamera, geolocalizzazione e notifiche e l’altro per le restanti impostazioni. Questi file sono denominati, rispettivamente, “SettingsRowSwitch” e “SettingsRow”. Essi sono molto simili; tuttavia, nel primo, è stato inserito uno *UISwitch*, ovvero un componente che simula un interruttore, i cui stati “acceso” e “spento” sono stati mappati sulle autorizzazioni concesse dall’utente. La classe dei due file è caratterizzata dagli “outlet” degli elementi grafici e da una “closure”, da eseguire in caso di tap sull’elemento associato. Le singole righe vengono controllate dallo *UIViewController* della schermata principale. Questo, tramite un metodo apposito, assegna un testo all’etichetta e valorizza la closure delle “SettingsRow” e “SettingsRowSwitch”.

4.3.2 Cambio dell’immagine di profilo

L’utente ha la possibilità di caricare o cambiare la propria immagine del profilo. Per farlo dovrà cliccare sulla voce corrispondente nella schermata principale di “Impostazioni”. A questo punto viene mostrato un menù da cui è possibile scegliere se caricare un’immagine dalla galleria o scattare una foto mediante fotocamera. Il menù contestuale che propone all’utente questa scelta è stato realizzato mediante il componente nativo *UIAlertController*.

Tale componente viene utilizzato nelle applicazioni iOS per avere feedback dall’utente, chiedere conferma, mostrare avvisi o proporre delle scelte. Le proprietà di *UIAlertController* sono le seguenti:

- *title*: rappresenta il titolo dell’avviso.
- *message*: è un testo descrittivo che fornisce maggiori informazioni sull’avviso.
- *style*: indica lo stile grafico dell’avviso da mostrare. Ci sono due possibili valori, ovvero “alert”, che mostra l’avviso in modale e “actionSheet”, il quale propone una serie di azioni in un menù posto nella parte inferiore dello schermo.
- *actions*: è un array di azioni che è possibile richiamare mediante l’avviso.

Nel nostro caso è stato utilizzato un “actionSheet”, la cui inizializzazione è mostrata nel Listato 4.1; in esso è possibile osservare anche il componente che gestisce la scelta vera e propria dell’immagine. Tale componente è denominato *UIImagePickerController*.

UIImagePickerController è un “ViewController” che gestisce le interfacce di sistema per la cattura di foto e video, oltre che la selezione di oggetti dalla libreria multimediale dell’utente. Il funzionamento di questo componente si basa sul “Delegation Pattern”. Nello specifico le interazioni dell’utente vengono gestite direttamente dal “ViewController” e i risultati vengono forniti al “delegate”. Il comportamento e l’aspetto di *UIImagePickerController* dipendono dal valore assegnato alla sua proprietà *sourceType*; questa può assumere i seguenti valori:

- *camera*: viene aperta la fotocamera e la foto scattata viene passata al “delegate”;
- *savedPhotosAlbum*: l’immagine da passare al “delegate” può essere selezionata dagli album dell’utente;
- *photoLibrary*: l’immagine viene selezionata tra tutte quelle della galleria multimediale dell’utente.

```
1   @objc
2       func showChangePicSheet() {
3           let changePicActionSheet = UIAlertController(title: "Cambia immagine profilo",
4                                                        message: "",
5                                                        preferredStyle: UIAlertController.Style.actionSheet)
6
7           let cameraAction = UIAlertAction(title: readFrom(.setting_change_profile_image_from_camera),
8                                            style: UIAlertAction.Style.default) { (_) in
9               self.imagePicker.sourceType = .camera
10              self.present(self.imagePicker, animated: true, completion: nil)
11          }
12          let galleryAction = UIAlertAction(title: readFrom(.setting_change_profile_image_from_gallery),
13                                            style: UIAlertAction.Style.default) { (_) in
14              self.imagePicker.sourceType = .savedPhotosAlbum
15              self.present(self.imagePicker, animated: true, completion: nil)
16          }
17          let cancelAction = UIAlertAction(title: "Annulla", style: UIAlertAction.Style.destructive)
18
19          changePicActionSheet.addAction(cameraAction)
20          changePicActionSheet.addAction(galleryAction)
21          changePicActionSheet.addAction(cancelAction)
22
23          self.present(changePicActionSheet, animated: true, completion: nil)
24      }
```

Listing 4.1. L'inizializzazione dello *UIAlertController* per il cambio dell'immagine del profilo

Poichè la piattaforma *Gigya* non permette di caricare immagini di dimensioni superiori ai 5MB, è stato necessario gestire la compressione delle foto scelte dall'utente. Questa operazione viene eseguita nel "delegate" del nostro *UIImagePickerController*. L'immagine deve essere trasmessa, mediante chiamata REST ai servizi del backend *Gigya*, facendo uso della codifica "Base64", la quale è caratterizzata da una dimensione pari al 133% di quella del file originale, pertanto la dimensione massima delle immagini compresse è stata limitata a circa 3.5MB.

Per la compressione delle immagini che vengono passate al "delegate" è stato sfruttato il metodo *jpegData* della classe *UIImage*; questo restituisce un "buffer di byte" contenente i dati dell'immagine memorizzata nell'istanza del suo oggetto. *jpegData* riceve in input una percentuale che rappresenta la qualità dell'immagine restituita alla fine della sua esecuzione rispetto alla foto originale. Mediante questo parametro si è scelto di dimezzare iterativamente la dimensione dell'immagine, fino a renderla inferiore al limite consentito.

Una volta terminato il processo di compressione, il "buffer di byte" viene convertito in una stringa "Base64", che viene successivamente passata poi al metodo del "SettingsInteractor", il quale provvederà a chiamare il servizio *Gigya* per il caricamento.

4.3.3 Autorizzazioni per l'accesso a fotocamera, geolocalizzazione e notifiche push

Una richiesta del cliente prevede la possibilità di modificare sull'app le autorizzazioni per l'accesso alla fotocamera, ai servizi di geolocalizzazione e alle notifiche push. Queste autorizzazioni ricadono sotto la categoria speciale di impostazioni per la privacy. In iOS, è possibile scegliere le app abilitate ad accedere alle informazioni sensibili archiviate sul dispositivo. Le impostazioni per la privacy regolano l'accesso a:

- Localizzazione;
- Contatti;
- Calendari;
- Promemoria;
- Foto;
- Bluetooth;
- Microfono;
- Riconoscimento vocale;
- Fotocamera;
- Salute;
- HomeKit;
- Libreria e Apple Music;
- Ricerca;
- File e cartelle;
- Movimento e fitness.

Un’app può usare questi dati solo dopo che l’utente ha acconsentito. La prima volta che l’app tenta di accedervi viene mostrato un avviso in cui si richiede all’utente un’autorizzazione. Questi può accettare o rifiutare; una volta effettuata la sua scelta, potrà visualizzare l’app con i relativi permessi nella sezione “Impostazioni per la privacy” di iOS ed, eventualmente, modificarli.

La richiesta del cliente ha dovuto subire dei cambiamenti per poter essere soddisfatta, poichè, una volta concessa o negata un’autorizzazione, non è possibile modificare tale scelta direttamente da app a causa delle limitazioni di iOS. Per questa ragione, la soluzione adottata comprende degli “switch” grafici che mostrano lo stato delle autorizzazioni. Al tap sugli stessi viene generato un avviso che permette all’utente di aprire la schermata di sistema “Impostazioni per la privacy”, da cui scegliere se concedere o bloccare l’accesso ai dati desiderati.

Per mostrare l’avviso è stato utilizzato, come di consueto, *UIAlertController*, mentre, per l’apertura vera e propria delle impostazioni di sistema, si è sfruttato il metodo condiviso *open* della classe *UIApplication*, il quale riceve come parametro un URL, che verrà aperto al termine della sua esecuzione. Nel nostro caso tale URL è una costante, memorizzata nella classe *UIApplication*, che rappresenta l’indirizzo delle impostazioni di sistema.

L’utilizzo dell’approccio appena descritto ha portato a due nuove problematiche. La prima riguardava la gestione dello stato degli “switch”, che, al tap dell’utente, veniva cambiato indipendentemente dalla modifica delle autorizzazioni. Questo generava un’inconsistenza tra lo stato mostrato e l’effettiva scelta dell’utente. La problematica è stata gestita come evidenziato nel Listato 4.2, ovvero mediante un metodo in grado di intercettare il componente di cui è avvenuto il tap. All’interno di tale metodo viene, poi, eseguita una closure, che, nel nostro caso, è valorizzata con le istruzioni per l’apertura delle impostazioni di sistema.

La seconda problematica dell’approccio utilizzato per la mdofica delle autorizzazioni si presenta nel momento in cui l’utente apre effettivamente le impostazioni di sistema, modifica le autorizzazioni concesse e ripristina l’app. Lo stato degli switch veniva infatti aggiornato dall’apposito “ViewController” all’interno del metodo *viewDidLoad*; questo a sua volta richiama un metodo del “ViewModel”. Poichè

```
1    @IBAction func switchChanged(_ sender: UISwitch) {
2            DispatchQueue.main.async {
3                self.switchView.isOn = !sender.isOn
4            }
5            switchPressedClosure?()
6        }
```

Listing 4.2. Il metodo per la gestione dell'animazione degli switch e dell'apertura delle impostazioni di sistema.

quando si passa da un'applicazione all'altra in iOS non vengono richiamati i metodi del ciclo di vita degli *UIViewController*, si è deciso di sfruttare il componente *NotifcationCenter*. Questo è in grado di gestire diversi tipi di eventi, tra cui la riattivazione di una determinata applicazione, mandando informazioni agli "observer" registrati. Nel nostro caso, alla riattivazione dell'applicazione, viene richiamato il metodo del "ViewModel" per l'aggiornamento del "ViewState".

4.3.4 Observer pattern

L' "observer pattern" è un pattern caratterizzato dalla presenza di un oggetto che cambia il suo stato ed altri che ricevono delle notifiche quando questo avviene. Nella Figura 4.3 viene mostrato un diagramma che illustra il funzionamento dell' "observer pattern".

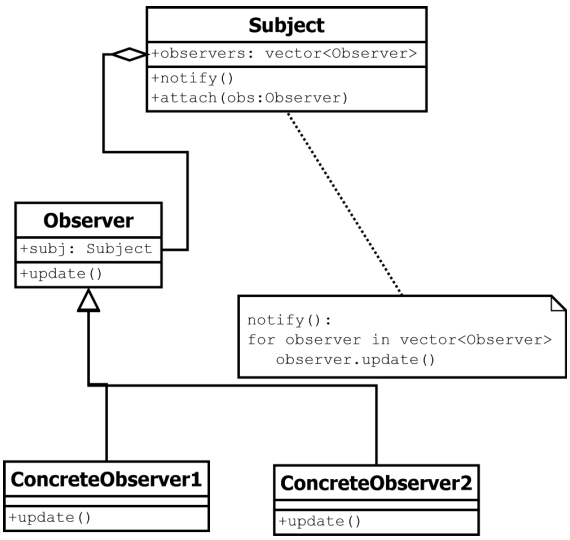

Figura 4.3. Un diagramma che rappresenta il funzionamento dell' "observer pattern".

L'oggetto di cui si vuole osservare il cambiamento di stato, denominato "Subject", presenta una collezione di oggetti di tipo "Observer". All'occorrenza di uno specifico evento, "Subject" chiama il metodo "update" di ciascun "Observer" che si

sarà precedentemente registrato presso “Subject”. Ogni “Observer” mantiene un’istanza di “Subject” e aggiorna il proprio stato indipendentemente dalle altre istanze della sua stessa classe.

5

Progettazione e implementazione del componente “Punti e premi”

In questo capitolo verranno approfonditi gli aspetti riguardanti la progettazione e l’implementazione del componente “Punti e premi”.

5.1 Progettazione

L’applicazione dovrà includere una sezione dedicata unicamente al programma di loyalty e alla raccolta punti. La schermata principale della sezione dovrà mostrare, in primo piano, il saldo punti dell’utente. Dovrà, inoltre, essere inserita una schermata informativa per spiegare all’utente le modalità di raccolta punti. Dovranno essere mostrati tutti i premi che l’utente è in grado di richiedere con i punti a sua disposizione. Verrà proposto, periodicamente, un premio in evidenza, che verrà mostrato all’utente ogni volta che questi accede alla sezione “Punti e premi”, ma solo se egli dispone di punti sufficienti a richiederlo. Infine, dovrà essere possibile consultare il catalogo completo dei premi, con la possibilità di filtrarlo in base al saldo disponibile, alla data di scadenza dei premi e ai punti necessari per richiederli.

5.2 Realizzazione grafica

Come richiesto, la sezione mostra, in primo piano, il saldo punti. L’informativa sulla raccolta punti è mostrata in modale in seguito al tap su un pulsante dedicato, posto immediatamente dopo il saldo punti e un’etichetta di saluto con il nome dell’utente. A questo punto viene mostrata una *UICollectionView* con i premi che l’utente può richiedere, mediante delle card. La schermata principale termina con una card che consente di aprire il catalogo completo dei premi. Quest’ultimo è organizzato come una colonna di card, che mostrano un’anteprima degli articoli e consentono di accedere ad informazioni dettagliate sugli stessi. Il filtraggio del catalogo avviene con modalità del tutto simili a quelle già adottate nella sezione Magazine, con l’aggiunta di un componente di tipo “slider” che permette la selezione del range di punti richiesti per il riscatto dei premi.

5.3 Implementazione

Verranno, ora, discussi gli aspetti implementativi del componente “Punti e premi”, con particolare attenzione alle librerie che si è deciso di integrare.

5.3.1 Promises e gestione avanzata di richieste asincrone

Le chiamate ai servizi di backend, utilizzate per recuperare i dati elaborati dall’applicazione, necessitano di una gestione asincrona delle risposte, poichè avvengono attraverso la rete. L’approccio tradizionale al problema prevede l’utilizzo delle cosiddette “funzioni di callback”, ovvero funzioni passate come argomento ad un’altra funzione e richiamate al termine dell’esecuzione di quest’ultima.

Quando si vogliono effettuare più elaborazioni ascinrone, l’una dipendente dai risultati dell’altra, la gestione tramite callback diventa piuttosto complessa, in quanto si è costretti a innestare su più livelli le istruzioni da eseguire. Questa situazione porta alla scrittura di codice poco comprensibile.

Un espediente migliore dal punto di vista della chiarezza è costituito dalle cosiddette “promise”. La differenza principale tra funzioni di callback e “promise” sta nel fatto che, con le callback, si comunicano alla funzione eseguita le operazioni da compiere una volta che le elaborazioni asincrone sono state completate, mentre, utilizzando le “promise”, la funzione da eseguire restituisce un particolare oggetto, a cui vengono comunicate le operazioni da effettuare in modo asincrono. In generale, una “promise” rappresenta il risultato finale di un’operazione asincrona o, rispettivamente, l’errore che spiega il motivo del fallimento di un’operazione. Si possono avere tre stati per questo costrutto, ovvero:

- textitpending: la “promise” è in attesa del risultato, ancora non disponibile;
- *fullfilled*: la “promise” contiene il risultato del “task” asincrono;
- *rejected*: la “promise” contiene un errore e una spiegazione delle ragioni del fallimento.

Una volta “mantenuta” o “infranta”, non è possibile più cambiare lo stato di una “promise”, inoltre, il risultato ottenuto può essere trasmesso a un numero illimitato di *observer*, i quali, a loro volta, restituiscono una “promise”. Questo fa sì che la creazione di una “pipeline” di operazioni asincrone risulti immediata. Mediante questo costrutto, è possibile effettuare in modo triviale operazioni come:

- eseguire una serie di elaborazioni asincrone, scrivendo un unico blocco di istruzioni post completamento;
- gestire gli errori in cascata;
- lanciare diverse operazioni asincrone e restituire il risultato della prima che termina;
- effettuare più tentativi per un’elaborazione asincrona che fallisce.

Per utilizzare le “promise” nell’applicazione realizzata, è stata integrata *Promises*, una libreria sviluppata da Google che offre compatibilità sia con Objective-C che con Swift, e prestazioni superiori rispetto alle librerie alternative. Nella Figura 5.1 è riportato il tempo medio impiegato da *Promises* ed altre librerie per creare

```
let promise = Promise<String> { fulfill, reject in
  // serie di operazioni asincrone
  if success {
    fulfill("Hello world.")
  } else {
    reject(someError)
  }
}

promise.then(on: backgroundQueue) { string in
  return string
}

promise.catch { error in
  print("Couldn't retreive string: \(error)")
}
```

Listing 5.1. Creazione di una "promise" e registrazione di due *observer* mediante i metodi *then* e *catch*.

una "promise" mantenuta, concatenare due blocchi di codice ed entrare nell'ultimo di questi.

Framework	Objective-C	Swift
Promises	0.000035536	0.000048412
PromiseKit	0.000071271	0.000061765
BrightFutures	N/A	0.000044416
Hydra	N/A	0.000086497
RxSwift	N/A	0.000060675

Figura 5.1. Benchmark di alcune librerie che implementano le "promise".

La creazione di una "promise" mediante *Promises* è un'operazione molto semplice; ne viene mostrato un esempio nel Listato 5.1.

I metodi *fulfill* e *reject* servono a decretare il risultato delle operazioni in caso di successo o fallimento, rispettivamente. I metodi *then* e *catch* registrano, invece, degli observer che vengono notificati del cambio di stato della "promise".

Promises mette a disposizione anche diversi operatori di alto livello, che consentono di eseguire operazioni avanzate o combinare gli esiti di diverse "promise", generando una risultante il cui stato di "fulfillment" o "rejection" dipende da quello delle sue componenti. Tali operatori sono:

- *all*: tutte le sue componenti devono essere in stato di "fulfillment" per lanciare il proprio *fulfill*, altrimenti viene lanciato il *reject*.
- *any*: è sufficiente che una sola delle sue componenti sia "fulfilled" per attivare il suo *fulfill*.
- *always*: utilizzato per eseguire un blocco di codice a prescindere dal risultato delle "promise" che lo compongono.
- *await*: consente di attendere, in modalità sincrona, il cambio di stato di una "promise".

```
let userOverview = userOverviewPromise()
        let rewardsCatalog = getRewardsCatalogPromise(fields: "FULL")

        any(
            userOverview,
            rewardsCatalog
        ).then({ (overview: Maybe<GetUserOverviewResponse>, catalog: Maybe<GetRewardsCatalogResponse>) in
            var userPoints = -20
            if let overview = overview.value, let points = overview.pointsBalance {
                userPoints = points
            }
            if let catalog = catalog.value {
                let tuple = PointsAndRewards(points: userPoints, rewards: catalog)
                completion(.success(tuple))
            } else if catalog.error != nil && userPoints > 0 {
                let tuple = PointsAndRewards(points: userPoints, rewards: nil)
                completion(.success(tuple))
            } else if let error = catalog.error, let _ = overview.error {
                completion(.failure(Error.genericError(description: "\(error)")))
            } else {
                completion(.failure(Error.genericError(description: "Something really went wrong")))
            }
        }).catch({ error in
            completion(.failure(Error.genericError(description: " \(error)")))
        })
```

Listing 5.2. Creazione e gestione delle promise per la generazione dell’elenco dei premi che l’utente può richiedere.

- *delay*: ritarda il completamento di una “promise”.
- *race*: simile ad *all*, ma la “promise” risultante avrà lo stesso risultato della prima componente che completa le sue operazioni.
- *retry*: consente di effettuare un secondo tentativo se la “promise” su cui viene chiamato fallisce.

Un esempio di utilizzo di “promise” all’interno dell’applicazione realizzata lo si può trovare nel processo di costruzione del *viewState* della schermata principale di “Punti e premi”. Per generare l’elenco dei premi che l’utente può richiedere, è necessario recuperare dal backend i dati del suo account, estrarre il saldo punti e, infine, eseguire la chiamata per la ricezione dell’elenco di tutti i premi. Combinando i risultati delle due chiamate, si ottiene la lista desiderata. Nel Listato 5.2 è riportato il metodo della classe *RewardsLandingInteractor* per la gestione del risultato delle “promise” relative al recupero dell’account utente e dei premi.

Si è scelto, in questo caso, l’utilizzo dell’operatore *any* per rendere la sezione più tollerante al malfunzionamento di una delle due chiamate. Le variabili *userOverview* e *rewardsCatalog* sono gli oggetti di tipo “promise” che si occupano di chiamare il backend. Una volta ottenuta la “promise” combinata, si procede a creare una tupla che contiene i dati ricevuti, i quali vengono passati come argomento alla funzione di callback fornita dal *ViewModel* all’*Interactor*. L’utilizzo delle “promise”, in questo caso, ha permesso di semplificare notevolmente il codice, ed evitare l’innesto di funzioni di callback, nonchè la scrittura di diversi blocchi condizionali per la gestione delle situazioni di errore.

5.3.2 Immagini delle card e libreria KingFisher

Nella sezione “Punti e premi” viene fatto un utilizzo massiccio di card per mostrare l’anteprima dei premi del programma di loyalty. Le card sono caratterizzate

dalla presenza di un titolo, di un'etichetta indicante i punti richiesti per riscattare l'articolo e di un'immagine.

Tutte le immagini delle card mostrate devono essere recuperate dalla rete; pertanto, prima viene letto un URL restituito dai servizi di backend e poi si procede con lo scaricamento dell'immagine vera e propria da tale indirizzo. Per questa ragione si è resa necessaria la gestione del processo di scaricamento in maniera semplice e con modalità asincrona, al fine di evitare il blocco dell'interfaccia grafica durante il recupero dei dati, nonchè il conseguente peggioramento dell'esperienza utente. Tale esigenza è stata soddisfatta tramite l'utilizzo di una libreria open source, denominata *Kingfisher*. Essa è caratterizzata da una notevole facilità di utilizzo e di integrazione, in quanto è scritta interamente in *Swift* ed estende i componenti nativi *UIImageView* e *UIButton*, per facilitare l'impostazione di un'immagine di sfondo recuperata dalla rete. Tra le feature offerte da *Kingfisher* troviamo le seguenti:

- scaricamento asincrono delle immagini e sistema di caching;
- lettura delle immagini da URL o da "buffer di dati" locali;
- metodi per l'elaborazione delle immagini e l'applicazione di filtri alle stesse;
- gestione avanzata della cache, sia in memoria che su disco;
- personalizzazione del comportamento della cache, con possibilità di impostare dimensioni e tempo di validità;
- funzione di annullamento del download delle immagini e possibilità di riutilizzo di contenuti precedentemente scaricati;
- componenti totalmente indipendenti l'uno dall'altro, con la possibilità di utilizzare unicamente quelli utili al proprio contesto;
- precaricamento delle immagini;
- possibilità di impostare un placeholder per le immagini il cui download è ancora in corso.

Le feature di *KingFisher* sfruttate dall'app sviluppata sono quelle di download asincrono delle immagini, impostazione di un placeholder e utilizzo di caching per evitare lo scaricamento multiplo delle stesse immagini, che avrebbe appesantito l'ecosistema sia lato backend che lato applicazione. Nel Listato 5.3 è riportato un esempio di utilizzo di *Kingfisher*. Per impostare l'immagine di una card è sufficiente chiamare il metodo *kf.setImage* sulle *UIImageView*, passando l'url del file da utilizzare sotto forma di oggetto di tipo *URL*. Per assegnare un placeholder occorre istanziare una *UIImage*, mentre per scegliere le opzioni riguardanti caching ed elaborazioni post scaricamento è sufficiente passare il parametro *options*; al termine del download si può lanciare una funzione di callback, passando una closure come parametro *completionHandler*.

5.3.3 "Catalogo premi", filtri e RangeSlider

La schermata "Catalogo premi" presenta una *UICollectionView* le cui celle sono costituite da card identiche a quelle presenti nella schermata principale di "Punti e premi". Come già accennato, la peculiarità di "Catalogo premi" è quella di consentire la visualizzazione della lista completa dei premi e l'applicazione di filtri in base a diversi criteri.

```
let url = URL(string: item.imageUrl)
let placeholder = UIImage(named: "placeholder")
            iconImageView.kf.setImage(with: url, placeholder: placeholder, options: nil, progressBlock: nil,
        completionHandler: { _ in
                DispatchQueue.main.async {
                    self.resetImageViewLayout()
                }
            })
```

Listing 5.3. Scaricamento di un’immagine da URL con utilizzo di un placeholder mediante la libreria *Kingfisher*.

L’implementazione dei filtri è molto simile a quella descritta per la sezione “Magazine”, ovvero, si utilizza una *UITableView*, posta in una schermata mostrata in modale. Ciò che contraddistingue i filtri del catalogo è la presenza del componente *RangeSlider*, il quale consente all’utente di selezionare, in modo immediato, il limite superiore e quello inferiore dell’intervallo di punti richiesti per riscattare i premi. In questo modo vengono mostrati solo quelli che ricadono nel range prescelto.

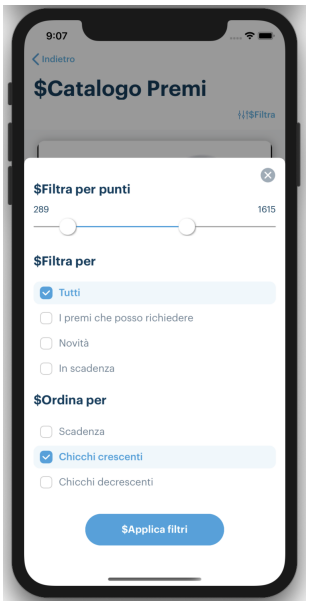

Figura 5.2. Il modale “filtri” della sezione “Catalogo premi”.

Come si può intuire dalla Figura 5.2, *RangeSlider* si presenta all’utente come una barra orizzontale colorata, con un delimitatore superiore e uno inferiore. Questi possono essere trascinati per modificare gli estremi dell’ intervallo, inoltre, i valori scelti vengono indicati da due etichette. La modifica degli estremi avviene per incrementi o decrementi discreti, la cui ampiezza è determinata da un parametro impostato da codice. Lo “slider” è realizzato facendo uso di componenti di base di *Swift*, quali *UIBezierPath*, che consente di definire delle curve per disegnare forme

```
1   // Invoked when the user first touches the control. The return value for the method informs the UIControl
        superclass whether subsequent touches should be tracked.
2       override func beginTracking(_ touch: UITouch, with event: UIEvent?) -> Bool {
3           previousLocation = touch.location(in: self)
4
5           // Hit test the thumb layers
6           if lowerThumbLayer.frame.contains(previousLocation) {
7               lowerThumbLayer.highlighted = true
8           } else if upperThumbLayer.frame.contains(previousLocation) {
9               upperThumbLayer.highlighted = true
10          }
11          if hapticsOn {
12              haptic.prepare()
13          }
14          return lowerThumbLayer.highlighted || upperThumbLayer.highlighted
15      }
```

Listing 5.4. Il metodo *beginTracking* con il quale inizia il tracciamento del movimento di scorrimento.

```
1       override func endTracking(_ touch: UITouch?, with event: UIEvent?) {
2           lowerThumbLayer.highlighted = false
3           upperThumbLayer.highlighted = false
4       }
```

Listing 5.5. Il metodo *endTracking* con il quale si resetta la selezione degli estremi dello slider.

geometriche, e *CALayer*, utilizzato per la creazione di animazioni e, nel nostro caso specifico, per tracciare il trascinamento dei componenti da parte dell'utente sullo schermo.

Per quanto riguarda l'implementazione dal punto di vista del codice, nei Listati 5.4, 5.6, e 5.5 sono riportati i metodi per la gestione di inizio trascinamento, fase intermedia e fase finale del movimento. La configurazione delle proprietà grafiche non viene riportata per semplicità.

Il metodo *beginTracking* presenta due parametri, *touch* e *event*; il primo memorizza le coordinate, il movimento e la forza di pressione di un singolo tocco dell'utente sullo schermo, mentre l'ultimo rappresenta un input fornito all'applicazione. Lo scopo di questo metodo è, essenzialmente, quello di individuare le coordinate del tocco appena avvenuto, per discriminare se l'utente ha premuto un punto corrispondente all'estremo inferiore o a quello superiore dello "slider".

In *continueTracking*, facendo affidamento di nuovo a *UITouch* e *UIEvent*, si individua la posizione finale dello scorrimento. Le coordinate sullo schermo vengono utilizzate per calcolare la differenza tra posizione iniziale e finale. Se il movimento è abbastanza ampio da eguagliare lo spostamento necessario a ricoprire uno step discreto, allora viene calcolato il nuovo valore estremo e viene dato un feedback sonoro all'utente chiamando il metodo *haptic.selectionChanged()*. A questo punto, viene memorizzato il valore dell'estremo modificato e, infine, viene aggiornata l'interfaccia.

Il metodo *endTracking*, il più semplice dei tre, ha il compito di resettare lo stato dei componenti grafici dello slider che rappresentano gli estremi dell'intervallo, in modo da farli risultare nuovamente come non selezionati dall'utente.

```
override func continueTracking(_ touch: UITouch, with event: UIEvent?) -> Bool {
        let location = touch.location(in: self)
        var deltaValue: Double

        // 1. Determine by how much the user has dragged
        let deltaLocation = Double(location.x - previousLocation.x)
        deltaValue = round((maximumValue - minimumValue) * deltaLocation / Double(bounds.width - thumbWidth))

        if discreteSteps > 0 {
            if abs(deltaLocation) >= Double(bounds.width - thumbWidth) / Double(discreteSteps) {
                deltaValue = round((maximumValue - minimumValue) * deltaLocation / Double(bounds.width -
        thumbWidth))
                previousLocation = location
                if hapticsOn {
                    haptic.selectionChanged()
                }
            } else {
                return true
            }
        } else {
            deltaValue = (maximumValue - minimumValue) * deltaLocation / Double(bounds.width - thumbWidth)
            previousLocation = location
        }

        // 2. Update the values
        if lowerThumbLayer.highlighted {
            lowerValue = min(lowerValue + deltaValue, upperValue - Double(thumbWidth / 3))
            lowerValue = boundValue(value: lowerValue, toLowerValue: minimumValue, upperValue: upperValue)
        } else if upperThumbLayer.highlighted {
            upperValue = max(upperValue + deltaValue, lowerValue + Double(thumbWidth / 3))
            upperValue = boundValue(value: upperValue, toLowerValue: lowerValue, upperValue: maximumValue)
        }

        // 3. Update the UI
        sendActions(for: .valueChanged)     // notify any subscribed targets of the changes
        return true
    }
```

Listing 5.6. Il metodo *continueTracking* con il quale si individua lo scostamento tra posizione iniziale e finale dello scorrimento eseguito dall’utente.

6

Progettazione e implementazione del componente “Profilo”

In questo capitolo verrà trattata nel dettaglio la sezione “Profilo”, progettata per fornire un riepilogo delle caratteristiche dell’account utente, nonchè promuovere la compilazione dei questionari.

6.1 Progettazione

La sezione “Profilo” dovrà fornire all’utente un resoconto rapido dei suoi dati e di tutte le sue attività all’interno dell’app. Verranno mostrati, per prima cosa, un’immagine del profilo, il nome dell’utente e il suo saldo punti per il programma di loyalty; inoltre, dovrà essere possibile accedere alla sezione “Impostazioni”. Quando dispoinibili, i questionari dovranno essere mostrati in questa sezione; per ciascuno di essi dovrà essere mostrata la ricompensa che si ottiene in seguito alla loro compilazione. Oltre ai questionari dovranno essere presenti tre sottosezioni. La prima di queste riporterà tutte le azioni dell’utente che comportano una variazione del saldo punti, come, ad esempio, il completamento di questionari o il riscatto di premi. La seconda sottosezione sarà invece dedicata ai premi richiesti. La terza mostrerà, infine, la lista delle macchine per caffè acquistate dall’utente e registrate tramite l’app; inoltre, dovrà includere una “CTA” per la registrazione di nuovi prodotti.

6.2 Realizzazione grafica

La sezione “Profilo” presenta, come richiesto, l’immagine del profilo impostata dall’utente; questa viene mostrata in una “view” di forma circolare ed è seguita da due label, indicanti, rispettivamente, il nome dell’utente e il suo saldo punti. Nella parte superiore della schermata è stata posizionata un’icona a forma di ingranaggio per permettere all’utente di aprire la sezione “Impostazioni”.

Le sottosezioni richieste sono state organizzate in una colonna verticale, ciascuna con una propria intestazione e con una card che ne mostra il contenuto vero e proprio. Nelle card vengono visualizzati solo alcuni degli elementi delle sottosezioni, in modo da rendere più agevole lo spostamento tra una sottosezione e l’altra, ed avere una navigazione più rapida del profilo.

Per visualizzare l’elenco completo di “attività”, “premi richiesti” o “macchine registrate” è sufficiente eseguire un tap sulle relative card; verrà mostrata, in modale, una schermata con tutte le voci relative all’utente.

6.3 Implementazione

La sezione “profilo” rappresenta un esempio concreto di utilizzo avanzato del componente *UITableView*, infatti, la schermata principale del profilo utente è realizzata mediante una *UITableView*; ciascuna cella di questa tabella è associata ad una delle sottosezioni richieste. Per ognuna di esse è stata implementata una classe apposita, e ne è stata disegnata l’interfaccia grafica mediante *InterfaceBuilder*. Procediamo, ora, ad analizzare nel dettaglio la sezione.

6.3.1 “Prossime missioni”

La sottosezione “Prossime missioni” è costituita da una *UICollectionView*, che presenta una serie di card aventi una struttura già utilizzata nell’applicazione, caratterizzata da tag, titolo, descrizione e immagine.

“Prossime missioni” viene gestita mediante la classe *MissionSliderTableViewCell*, ed è istanziata come cella della *UITableView* principale di “Profilo”. Essa è caratterizzata da un riferimento alla *UICollectionView* contenente i questionari, da una proprietà che memorizza tutti gli elementi della collezione, da una closure, chiamata al tap dell’utente su questi ultimi, e, infine, da un oggetto che implementa il “delegate” della *UICollectionView*. I metodi di *MissionSliderTableViewCell* hanno il compito di impostare il “delegate” della collezione, nonchè la closure da eseguire ad ogni aggiornamento dei dati.

Nel “delegate” di *MissionSliderTableViewCell*, denominato *MissionsCollectionViewSourceDelegate*, sono presenti metodi per istanziare le celle, configurarne le proprietà, regolarne la dimesione e gestire gli eventi inerenti l’aggiornamento dati e le interazioni dell’utente. Le celle della *UICollectionView* che ospita le missioni sono definite nella classe *MissionSliderCollectionViewCell*. Al tap dell’utente su di esse viene chiamato un apposito metodo del “delegate”, mediante il quale si esegue una closure, definita nella classe *MissionSliderTableViewCell*. Nella closure viene chiamato, a sua volta, un altro metodo del “delegate”. La situazione appena descritta porta ad una problematica che, per essere compresa, necessita di un approfondimento del meccanismo di gestione della memoria utilizzato da *Swift*. Tale meccanismo verrà illustrato in dettaglio nella prossima sottosezione.

6.3.2 Gestione della memoria in Swift

Swift utilizza un sistema automatico di gestione della memoria, chiamato *Automatic Reference Counting* (ARC). Come intuibile dal suo nome, ARC sfrutta il numero di riferimenti a un blocco di memoria per decidere se deallocarlo o meno. Quando un oggetto viene creato, il suo contatore dei riferimenti è impostato a 1, e può aumentare o diminuire durante il suo ciclo di vita. Nel momento in cui il contatore scende a 0, allora l’oggetto può essere deallocato.

```
1    let cell: MissionSlideCollectionViewCell = dequeueCell(collectionView: collectionView, indexPath: indexPath)
2
3    weak var weakCollection = collectionView
4            cell.cardTapped = { [weak self] cell in
5                guard let self = self, let collectionView = weakCollection else { return }
6                self.cardTapped(cell: cell, collectionView: collectionView)
7            }
```

Listing 6.1. Valorizzazione di una closure con riferimento weak all'oggetto che effettua l'assegnazione.

In *Swift*, quando viene dichiarata una variabile, si sceglie se questa dovrà essere *weak* o *strong*. Le variabili "deboli" non fanno variare il contatore dei riferimenti, mentre quelle forti lo fanno incrementare. Questo significa che, finchè una variabile *strong* è attiva, rimarrà sicuramente in memoria, mentre ciò non è garantito per le variabili *weak*. Questo meccanismo porta ad un problema molto noto tra gli sviluppatori di *Swift*, ovvero quello dei *retain cycle*, riferimenti ciclici tra oggetti, causa dei cosiddetti "memory leak", o perdite di memoria, che si verificano quando una variabile rimane allocata ma non viene più utilizzata dall'app.

La situazione descritta nella sottosezione predecente prevede un oggetto che valorizza la closure, proprietà di un altro oggetto; questa, a sua volta, richiama un altro metodo del primo oggetto. Per evitare un *retain cycle*, si rivela fondamentale adottare la soluzione mostrata nel Listato 6.1.

Nel momento in cui si valorizza la closure con la chiamata a un metodo del "delegate" che aveva effettuato l'assegnazione, si definisce quest'ultimo come *weak*. In questo modo, una volta deallocato, esso non viene più tenuto vivo dal riferimento interno alla closure.

6.3.3 "Lista di tutte le attività", "Premi richiesti", "Le tue macchine"

Le altre sottosezioni di "Profilo" sono costituite ciascuna da una card, all'interno della quale sono mostrati elementi di diverso tipo, a seconda della sezione di appartenenza: questi possono rappresentare attività di riscossione premi, acquisizione punti o macchine registrate. Ogni card presenta un tag e un titolo, ottenuti mediante il componente *UILabel*, e seguiti dall'elenco vero e proprio degli elementi; nel caso delle macchine si ha anche una CTA per la registrazione di una nuova macchina. L'elenco degli elementi, interno a ciascuna card, viene gestito mediante una *UITableView*; in particolare, si tratta di una *UITableView* innestata all'interno di un'altra tabella. Nel "ViewState" della schermata principale di "Profilo", vengono memorizzati i contenuti delle sottosezioni all'interno di un array di oggetti di tipo *ProfileLandingSections*. Questi, a loro volta, comprendono due prorietà, ovvero *title* e *items*. La prima rappresenta l'intestazione della sottosezione mentre l'ultima il suo contenuto. *items* è un array di *ProfileLandingSectionsType*, variabili enumerative definite mediante il costrutto *enum*. Questo concetto verrà approfondito nella Sezione 6.3.4. Ogni elemento di *items*, a seconda del suo tipo, conterrà oggetti di natura diversa, che verranno gestiti in maniera opportuna dal "delegate" della tabella principale, il quale genererà celle di tipo *TransactionsCardTableViewCell*. Le

```
1
2   private func createTransactionItemsForActivities(response: GetUserDetailResponse) -> [TransactionItem] {
3           let filtered = response.filter({ !$0.transactionEvent.isNullOrEmpty && $0.point != nil && $0.point != 0
            }).sorted(by: { $0.date > $1.date })
4
5           let items = zip(filtered, filtered.indices).map({
6               return self.mapActivityResponseToTransaction(response: $0.0, separatorHidden: ($0.1 == filtered.count
            - 1))
7           })
8
9           return items
10      }
```

Listing 6.2. Il metodo *createTransactionItemsForActivities* che filtra gli elementi non validi della risposta ricevuta dal backend li ordina dal più recente al più vecchio e infine li utilizza per istanziare deglli oggetti di tipo *Transactionitem*.

tabelle contenute in queste ultime presentano righe molto simili tra loro, con alcune differenze grafiche in base alla sezione in cui vengono utilizzate. In Figura 6.1 sono riportate le tre card che rappresentano le sottosezioni di “Profilo”.

Figura 6.1. Le card contenenti le sottosezioni di “Profilo”.

Prima di essere utilizzati dai diversi “delegate” per istanziare le righe delle *UITableView*, i dati ottenuti dalle chiamate al backend vengono elaborati dal “viewModel”, che, a seconda della sezione di destinazione dei dati stessi, utilizza dei metodi opportuni per convertirli in un formato più idoneo alla manipolazione da parte dei “delegate”. Nel Listato 6.2 è riportato il metodo *createTransactionItemsForActivities*, il quale genera oggetti di tipo *TransactionItem*, utilizzati nella tabella della sezione “Lista di tutte le attività”.

Per prima cosa, *createTransactionItemsForActivities* filtra gli elementi della risposta passata ad esso come parametro, eliminando quelli che non presentano i campi *points* e *transactionEvent*. Il primo rappresenta i punti guadagnati o spesi mediante l’attività a cui l’oggetto si riferisce, mentre l’ultimo indica il tipo di attività eseguita. Dopo aver filtrato la risposta, *createTransactionItemsForActivities* ordina gli elementi dal più recente al più vecchio; infine, questi ultimi vengono inseriti in un array di oggetti *TransactionItem*; questi contengono le stringhe for-

```
1   enum ProfileLandingSectionsTypes {
2       case missions(missions: ProfileLandingMissions)
3       case activities(bottomType: ProfileBottomType)
4       case machines(bottomType: ProfileBottomType)
5   }
6   enum ProfileBottomType: Equatable {
7       case transactionCard(card: TransactionCardItem)
8       case button(title: String)
9   }
```

Listing 6.3. I tipi enumerativi *ProfileLandingSectionsTypes* e *ProfileBottomType*.

mattate come mostrato nella terza card riportata in Figura 6.1. Inoltre, l'attributo *separatorHidden* permette al "delegate" della tabella in cui verrà mostrato il *TransactionItem*, di nascondere il separatore degli elementi della lista relativo all'ultima voce della stessa. Il metodo *createTransactionItemsForActivities* utilizza tre metodi per la manipolazione delle collezioni di oggetti forniti da *Swift*. Questi sono:

- *filter*: restituisce solo gli oggetti della collezione di partenza che rispettano il predicato logico passato al metodo;
- *zip*: riceve in input due oggetti e crea una collezione di tuple, formate da coppie di elementi contenute nelle collezioni di partenza;
- *sorted*: restituisce una collezione ordinata secondo il criterio specificato in input;
- *map*: restituisce una collezione ottenuta applicando il predicato passato in input ad ogni singolo elemento della collezione di partenza.

I metodi appena descritti fanno uso della libreria *NSPredicate*, inclusa in *Swift* e utilizzata per esprimere condizioni logiche. Nel Listato 6.2 sono stati creati dei predicati logici "inline", ovvero istanziati direttamente nel loro contesto di utilizzo; i termini su cui la condizione logica viene verificata di volta in volta sono indicati mediante l'espressione *$0*.

6.3.4 Enumeration con valori associati

La "Enumeration" è un costrutto che definisce un gruppo di valori tra loro correlati, abbassando la probabilità di commettere errori quando si lavora con essi. Rispetto alle enumerazioni di linguaggi più tradizionali, quelle offerte da *Swift* presentano una maggiore flessibilità. Innanzitutto, non è necessario fornire esplicitamente un valore per ogni voce definita; inoltre, se tale valore viene fornito, questo può essere una stringa, un intero o un *Float*; in alternativa, si può specificare il tipo di valore da associare ai casi dell'enumerazione.

In *Swift*, le enumerazioni sono veri e propri tipi di dato e presentano molte caratteristiche che, tradizionalmente, sono possedute solo dalle classi; ad esempio, si possono definire delle "computed property", ovvero delle proprietà calcolate mediante l'esecuzione di un metodo, definito, a sua volta, nell'enumerazione.

Le enumerazioni possono associare anche dei valori alle loro voci. In questo caso, tali valori non vengono inizializzati, ma si definisce una struttura che, successivamente, permette di accedervi in modo agevole. Nel Listato 6.3 sono riportate le

definizioni di *ProfileLandingSectionsTypes* e *ProfileBottomType*, utilizzati, rispettivamente, per distinguere le sottosezioni di “Profilo” e i tipi possibili di righe che compaiono nelle loro tabelle.

La parola chiave *enum* è utilizzata per definire un’enumerazione, mentre con *case* si specifica uno dei valori che quest’ultima include. In entrambi gli esempi presentati, sono stati associati dei valori a ciascun caso dichiarato. Le variabili di tipo *ProfileBottomType*, nello specifico, possono contenere i dati di una transazione o il testo da mostrare su un pulsante.

7

Manuale utente dell'app realizzata

Questo capitolo è dedicato al manuale utente dell'applicazione. Verranno descritte nel dettaglio tutte le sue funzionalità e verranno mostrate le relative schermate.

7.1 Registrazione e login

All'avvio dell'applicazione viene presentata la schermata di benvenuto, riportata in Figura 7.1.

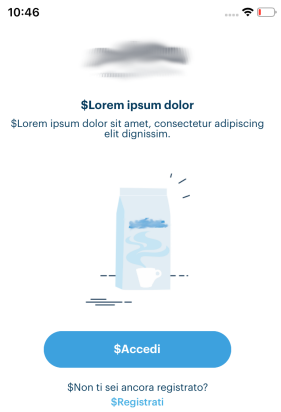

Figura 7.1. La schermata di benvenuto.

In essa si invita l'utente ad accedere o, se non possiede un account, a registrarsi. Cliccando sul pulsante "Registrati" si giunge alla schermata mostrata in Figura 7.2.

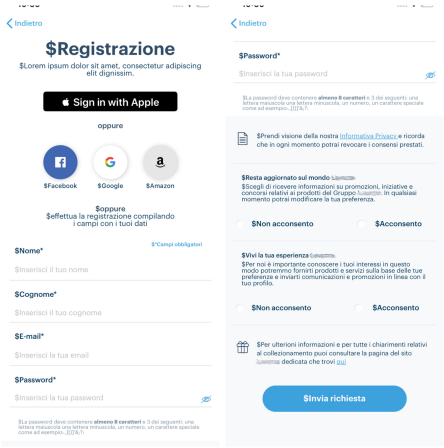

Figura 7.2. La schermata di registrazione.

A questo punto l'utente compilerà i campi, seguendo le indicazioni riportate sotto di essi. In caso vengano commessi errori, verranno mostrati dei suggerimenti per correggerli. Se la compilazione viene eseguita correttamente, al tap sul pulsante "Invia richiesta", viene inviata una mail all'indirizzo specificato e si passa alla schermata riportata in Figura 7.3. L'utente dovrà seguire le istruzioni riportate nel messaggio e l'attivazione dell'account sarà completata.

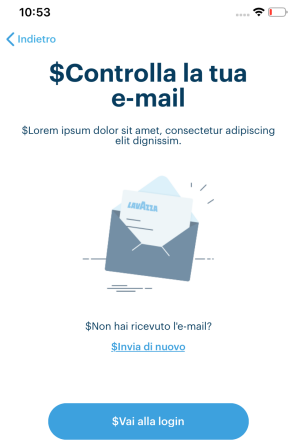

Figura 7.3. La schermata di "Email inviata", mostrata alla fine del processo di registrazione.

È possibile registrarsi anche mediante un social network; in questo caso basterà cliccare, dalla schermata di registrazione, sul logo del social scelto e autenticarsi.

Per completare la registrazione all’app basterà, in questo caso, fornire il consenso per la raccolta dei dati.

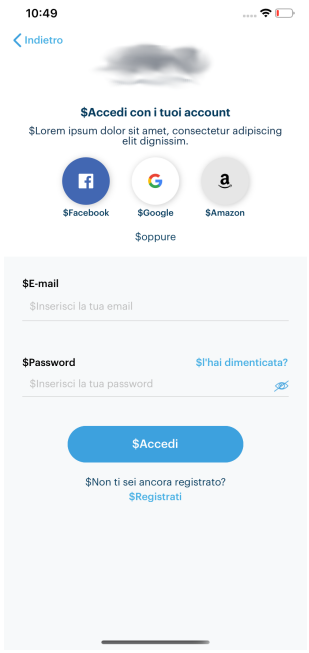

Figura 7.4. La schermata di login.

Dalla schermata di “Login”, mostrata in Figura 7.4, basterà inserire le proprie credenziali o cliccare sul logo del social network con cui ci si è registrati, e si entrerà nella sezione “Home” dell’applicazione. Se l’utente ha dimenticato le proprie credenziali può cliccare sul pulsante “L’hai dimenticata”, e riceverà un messaggio con le istruzioni per impostare una nuova password.

7.2 Primo accesso e procedura di “Onboarding”

Quando l’utente accede per la prima volta con il suo account, gli viene mostrata una schermata in cui lo si invita a rispondere ad alcune domande. Cliccando sul pulsante “Conosciamoci meglio” viene lanciato il processo di “Onboarding”, le cui schermate sono riportate nella Figura 7.5.

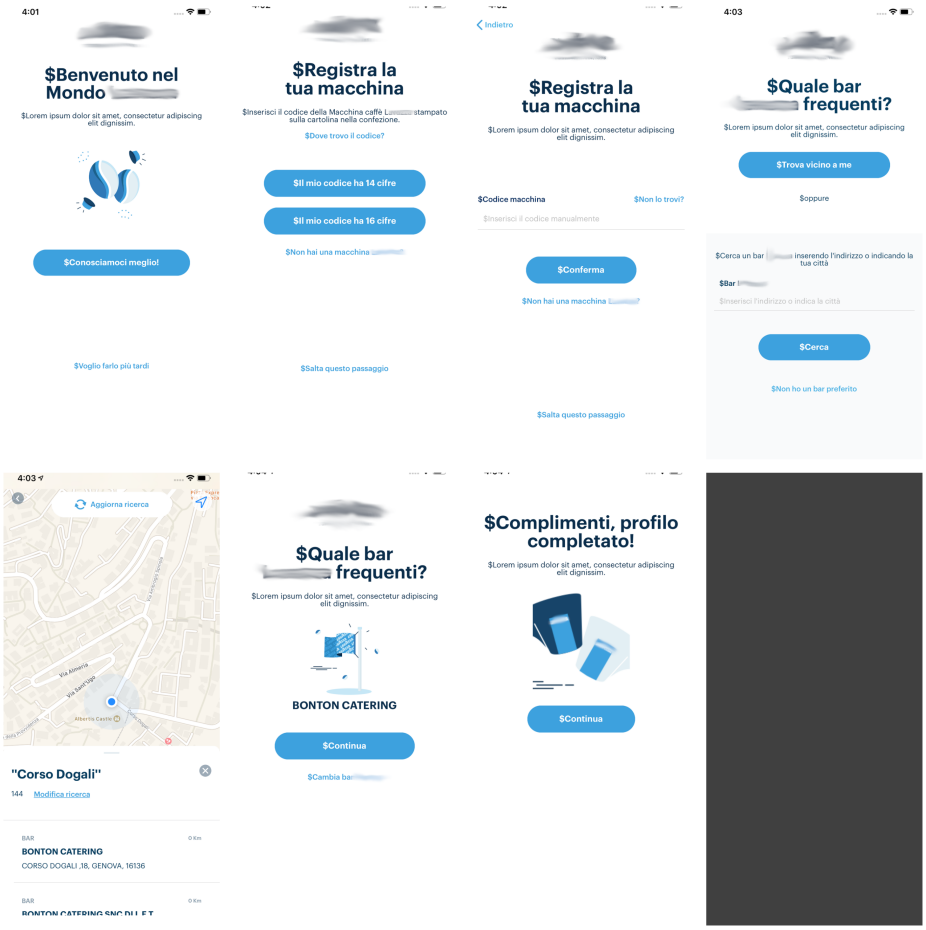

Figura 7.5. Il flusso di "Onboarding".

La prima fase prevede la registrazione, mediante numero di serie, di eventuali macchine da caffè prodotte dall'azienda cliente. Queste potranno avere 14 o 16 cifre. Dopo aver cliccato sul pulsante opportuno, verrà mostrata una schermata in cui inserire tale numero. Nella seconda fase dell'"Onboarding" l'utente può scegliere il suo bar preferito, digitando un indirizzo o utilizzando la posizione del dispositivo. A questo punto verranno aperti una mappa e un elenco da cui scegliere il bar.

Nell'ultima fase dell'"Onboarding" si chiede all'utente di comunicare la quantità di caffè bevuto giornalmente. Egli potrà selezionare la risposta che lo rispecchia mediante i pulsanti a forma di freccia posti nell'apposita schermata.

7.3 Home

Una volta completato l'"Onboarding" si può accedere ai contenuti dell'app veri e propri. Nella parte inferiore dello schermo è presente una barra di controllo, mediante la quale è possibile spostarsi tra le diverse sezioni dell'applicazione.

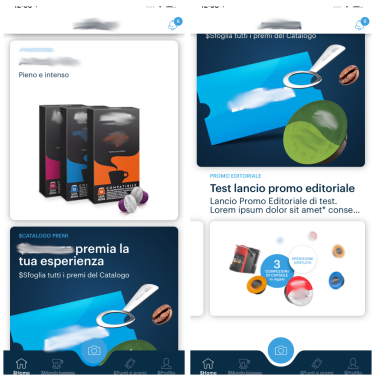

Figura 7.6. La sezione "Home".

La prima sezione mostrata è "Home", riportata nella Figura 7.6; in questa possiamo notare due card seguite da una collezione. La prima card conduce al catalogo dei prodotti, la seconda a quello dei premi, mentre la collezione mostra una serie di card dedicate alle promozioni offerte dall'azienda cliente. Cliccando sugli elementi della collezione viene aperta la schermata riportata in Figura 7.7.

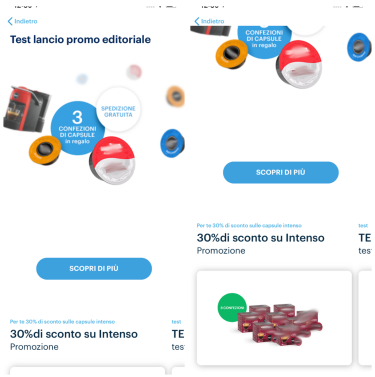

Figura 7.7. La schermata di dettaglio di una promozione.

In essa sono descritti i dettagli delle promozioni, ed è presente un pulsante che permette di aprire, in un browser esterno, una pagina dedicata all'offerta che si sta visualizzando. L'ultimo elemento mostrato nella schermata di dettaglio promozioni è una collezione di card contenenti elementi correlati all'offerta visualizzata.

7.4 Store

La seconda macrosezione dell'app, riportata in Figura 7.8, è denominata "Store" e presenta diverse funzionalità.

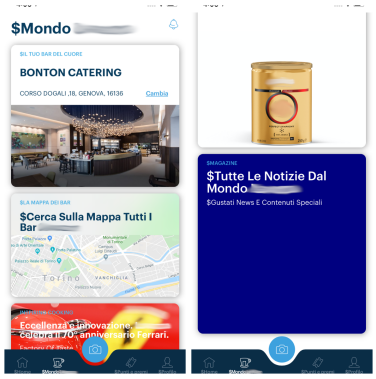

Figura 7.8. La sezione "Store".

Le prime due card mostrate consentono, rispettivamente, di selezionare il proprio bar preferito o di cercare attività convenzionate con l'azienda cliente, con le modalità già descritte nella Sezione 7.2.

La terza card ospita un articolo della sezione "Magazine", contrassegnato come "in primo piano". Abbiamo a questo punto tre card, che conducono al catalogo prodotti; ognuna di esse permette di visualizzare una diversa categoria di prodotto e, al tap su di esse, verrà mostrata la schermata riportata in Figura 7.9. Qui si può osservare una serie di prodotti e, cliccando sulle card, viene aperta la schermata di dettaglio prodotto, del tutto simile a quella già descritta per le promozioni.

Figura 7.9. Il catalogo prodotti.

L'ultima card presente in "Store" conduce alla sezione "Magazine". Questa è suddivisa in tre sottosezioni, caratterizzate, ciascuna, da un'intestazione e da una diversa categoria di articoli ospitata. In base a queste categorie, "Magazine" può essere filtrata, cliccando sul pulsante "Filtra". Viene, così, aperta la schermata riportata in Figura 7.10, che permette di selezionare le categorie a cui si è interessati, tramite un semplice tap sulla relativa etichetta.

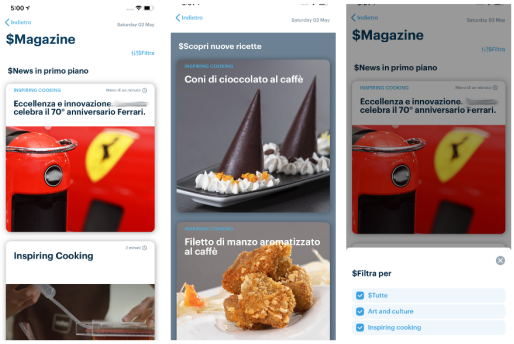

Figura 7.10. La schermata principale di "Magazine".

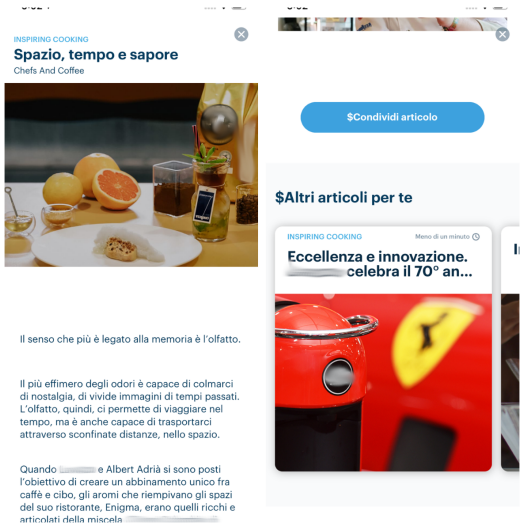

Figura 7.11. La schermata di dettaglio di un articolo.

Al tap sulla card di un articolo viene aperta la schermata di dettaglio ad esso relativa, riportata in Figura 7.11.

Da qui si può leggere il contenuto completo dell'articolo stesso e, se previsto, viene mostrata l'anteprima di un video; cliccando su di essa appare un player in

grado di riprodurre il filmato. Scorrendo l'articolo è possibile trovare una galleria di immagini, seguita da un pulsante, che permette di condividere un link all'articolo mediante applicazioni esterne. L'ultimo elemento della schermata di dettaglio degli articoli è una collezione di card, contenenti elementi correlati a quello visionato.

7.5 Punti e premi

La terza sezione dell'app, denominata "Punti e premi", è dedicata al programma di loyalty. Nel momento in cui si accede alla sezione, se si possiedono abbastanza punti per richiederlo, viene mostrata una schermata con un premio consigliato, la sua descrizione e un pulsante che permette di riscattarlo. Nella schermata principale di "Punti e premi" vengono mostrati il saldo punti dell'utente e un pulsante che consente di aprire una guida sulle modalità di collezionamento dei punti. Seguono una sottosezione con l'elenco dei premi che l'utente può riscattare con i punti a sua disposizione, e, infine, una card per la visualizzazione del catalogo premi, analoga a quella mostrata in "Home". Nella Figura 7.12 sono riportate la schermata principale della sezione e l'informativa sulla raccolta punti.

Figura 7.12. La schermata principale di "Punti e premi", seguita dall'informativa sulla racolta punti.

Il catalogo dei premi è costituito da una collezione di card, che è possibile filtrare secondo diversi criteri. Nella Figura 7.13 è riportata, a sinistra, la schermata con il catalogo, mentre, a destra, quella dei filtri.

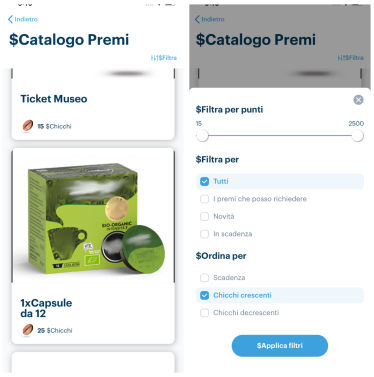

Figura 7.13. A sinistra, la schermata "catalogo premi"; a destra, la schermata dei filtri.

Cliccando su un premio viene mostrata una sua descrizione ed è possibile richiederlo mediante un apposito pulsante. Se il premio scelto è di tipo digitale come, ad esempio, un codice sconto, allora viene inviata all'utente una e-mail con il codice.

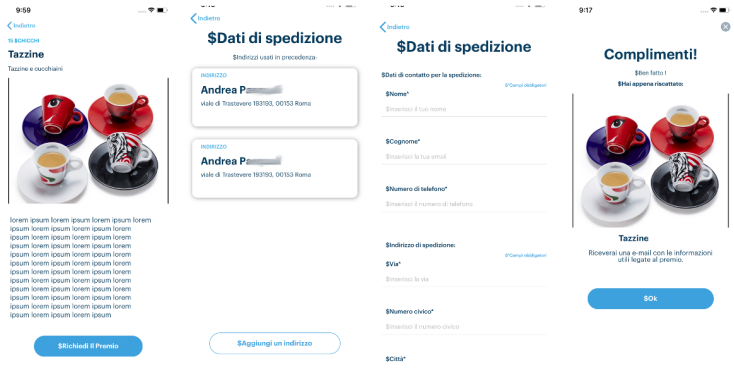

Figura 7.14. La procedura di riscatto premi.

Se il premio è di tipo fisico, invece, si passa alla schermata mostrata in Figura 7.14, da cui è possibile selezionare un indirizzo di spedizione o aggiungerne uno nuovo, compilando il relativo form. Dopo la selezione dell'indirizzo viene aperta una schermata che segnala il completamento della procedura.

7.6 Profilo

La quarta sezione dell'applicazione, denominata "Profilo", offre all'utente un riepilogo delle attività eseguite sull'app. Alcune schermate della sezione sono riportate in Figura 7.15.

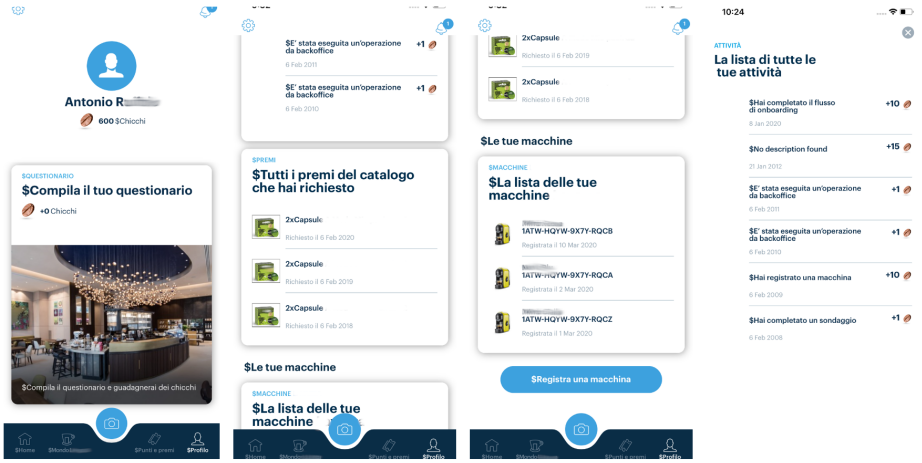

Figura 7.15. La sezione "Profilo".

In primo piano viene visualizzato il nome dell'utente, seguito dal suo saldo punti. Nel profilo vengono visualizzati i questionari non ancora compilati, le azioni effettuate che hanno portato a una variazione del saldo punti, i premi riscattati e, infine, le macchine da caffè registrate.

Cliccando su uno dei questionari disponibili viene visualizzata una schermata introduttiva, in cui sono riportate alcune informazioni di base, tra cui, il numero di punti che il questionario permette di guadagnare. Premendo il pulsante "Inizia", si passa alla schermata con le domande. Queste sono mirate a profilare l'utente e, per ciascuna di esse, egli può scegliere una risposta da 1 a 5, a seconda del grado di pertinenza con la propria persona. Una volta fornite tutte le risposte il questionario può essere inviato; verrà quindi visualizzata una "thank you page" in cui si ringrazia l'utente per aver completato la missione.

La seconda sottosezione di "Profilo" presenta le ultime attività all'interno dell'app, ciascuna con una propria descrizione e la variazione di punti che ha comportato. Cliccando su un elemento della lista viene aperta una modale con l'elenco delle attività completo. Analogo è il funzionamento della terza sottosezione, che riporta l'elenco dei premi richiesti dall'utente.

L'ultima sottosezione di "Profilo" è dedicata alle macchine per caffè che l'utente ha registrato; inoltre, è possibile registrarne di nuove, cliccando sul pulsante "Registra una macchina" e seguendo una procedura analoga a quella mostrata per l'"Onboarding".

7.7 Impostazioni

Cliccando sull'icona a forma di ingranaggio nella sezione "Profilo" viene aperta la schermata delle impostazioni, riportata in Figura 7.16.

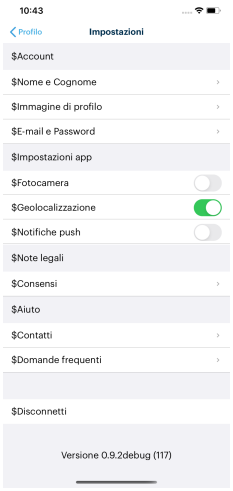

Figura 7.16. La schermata principale di "Impostazioni".

Le prime tre opzioni delle impostazioni permettono la modifica del proprio account *Gigya*. Per cambiare il proprio nome e il proprio cognome l'utente ha a disposizione una semplice schermata con due campi di testo, come mostrato in Figura 7.17.

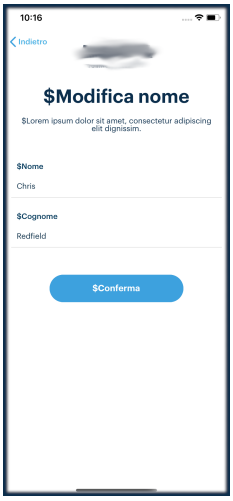

Figura 7.17. La schermata per il cambio di nome e cognome.

Cliccando su "Immagine di profilo" verrà aperto un "ActionSheet" in basso, grazie al quale è possibile scegliere se scattare una foto o selezionarla da galleria, come mostrato in Figura 7.18.

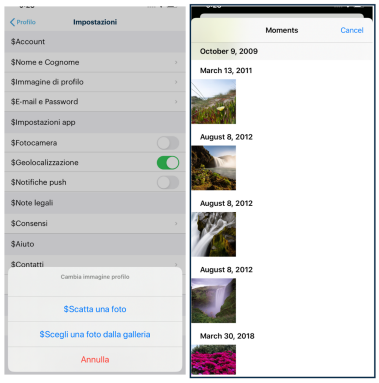

Figura 7.18. Il cambio dell'immagine di profilo.

Il cambio della password può essere effettuato cliccando sull'apposita opzione. Verrà a questo punto visualizzata la schermata riportata in Figura 7.19 a sinistra. Il proprio indirizzo email sarà mostrato in un campo di testo non modificabile, e, cliccando sul pulsante "Richiedi", verrà inviato un messaggio con le istruzioni per il completamento della procedura.

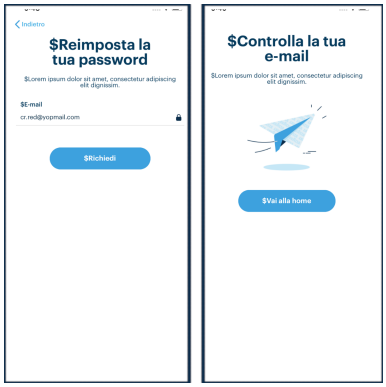

Figura 7.19. Il reset della password.

Le opzioni "Fotocamera", "Geolocalizzazione" e "Notifiche push" riguardano la concessione delle autorizzazioni per accedere a queste componenti del sistema. Lo stato delle stesse è indicato da un interruttore acceso o spento e, cliccando su di esso, viene mostrato un avviso che reindirizza l'utente alle impostazioni di sistema, in modo da poterle modificare.

La voce "Consensi" aprirà, in modale, la schermata riportata in Figura 7.20, da cui potranno essere concessi o revocati i consensi alla raccolta dati a fini pubblicitari e di profilazione.

Figura 7.20. La schermata per il cambio dei consensi.

La voce "Contatti" aprirà una pagina del sito web dell'azienda cliente, in cui è possibile visualizzare i contatti di quest'ultima, mentre, cliccando su "Domande frequenti", verrà visualizzata la schermata mostrata in Figura 7.21, che fornirà all'utente le risposte alle domande più frequenti, simulando una chat.

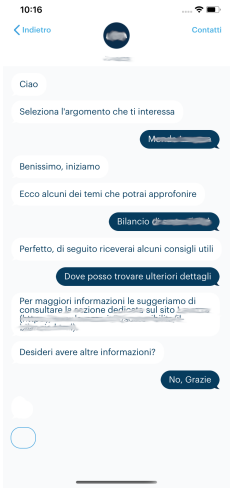

Figura 7.21. "Domande frequenti".

Cliccando su "Disconnetti", infine, verrà eseguito il logout e l'utente verrà portato alla schermata di login.

8

Analisi e discussione del lavoro realizzato

In questo capitolo verrà effettuata un'analisi SWOT del sistema realizzato, valutando punti di forza, debolezze, opportunità e rischi ad esso relativi. Verrà poi confrontata l'app con alcuni sistemi affini.

8.1 Analisi SWOT

L'analisi SWOT è uno strumento utilizzato per identificare e valutare i fattori, interni ed esterni ad un'impresa, che possono avere un impatto su un'attività o sull'impresa stessa. L'analisi si divide in quattro punti:

- Strenghts, ovvero i punti di forza interni all'azienda;
- Weaknesses, ovvero i punti di debolezze, e, quindi, gli aspetti da migliorare;
- Opportunities, le opportunità esterne all'azienda che questa può sfruttare mediante nuovi sviluppi;
- Threats, le minacce esterne che non dipendono dall'impresa ma che questa potrebbe dover fronteggiare.

Nella Figura 8.1 è riportato un diagramma che schematizza l'analisi SWOT del sistema oggetto della presente tesi. Ciascun punto dell'analisi verrà approfondito nelle prossime sottosezioni.

8.1.1 Punti di forza

L'applicazione è stata realizzata utilizzando le architetture e i pattern di successo più recenti, in modo da garantire un alto grado di manutenibilità e consentire di ampliare facilmente le funzionalità del prodotto, adattandosi alle esigenze future del cliente. Si è garantita, inoltre, una buona stabilità dell'app mediante unit test.

L'interfaccia utente risulta molto intuitiva e accattivante, e la user experience è studiata per rendere piacevole l'utilizzo dell'applicazione; infatti, ogni sezione è raggiungibile mediante pochi tap.

L'azienda cliente possiede ottime competenze nell'ambito del marketing e dell'advertising, che possono essere utilizzate per aumentare la diffusione dell'applicazione, contribuendo, in tal modo, ad accrescere la brand loyalty.

Figura 8.1. Analisi SWOT dell'app realizzata.

Il sistema di notifiche consente un coinvolgimento continuo dell'utente, che, in questo modo, sarà aggiornato più prontamente e più frequentemente sulle ultime promozioni e sul lancio di nuovi prodotti da parte dell'azienda cliente.

8.1.2 Debolezze

I dati raccolti tramite l'app devono essere integrati con quelli provenienti dagli altri punti di contatto con gli utenti; inoltre, in molti casi, tali dati sono relativi all'utilizzo specifico delle funzionalità dell'app. Ciò comporta una maggiore complessità dei processi di aggregazione e analisi dei dati.

Per garantire l'utilizzo dell'app da parte degli utenti, nonchè l'afflusso costante di dati, è necessario creare continuamente nuovi contenuti che spingano alla consultazione dell'applicazione.

A causa dei meccanismi di funzionamento di *SAP Hybris* l'app non gestisce tutte le operazioni e le transazioni eseguite dall'utente, che è costretto a consultare la propria casella email per recuperare eventuali codici sconto o altri premi digitali richiesti.

Per l'acquisto dei prodotti i clienti vengono rimandati ai rivenditori fisici o al sito di e-commerce dell'azienda, il che rende l'esperienza utente meno omogenea. Per questa ragione, in futuro, sarebbe consigliato integrare il processo di vendita direttamente all'interno dell'app.

8.1.3 Opportunità

I dati che verranno raccolti tramite l'applicazione rappresentano un'opportunità per poter conoscere meglio i gusti degli utenti, orientare la produzione e le campagne di vendita, nonchè far conoscere ai clienti i prodotti che saranno più propensi ad acquistare.

Oltre a migliorare la conoscenza delle abitudini dei clienti attuali, l'app fornisce un'ulteriore opportunità. Grazie all'ampia diffusione degli smartphone, infatti, si potranno individuare e raggiungere nuovi segmenti di mercato, che l'azienda può curare sviluppando prodotti e promozioni ad hoc.

La possibilità di configurare tutti i contenuti dell'app consentirà di adattare la stessa ai diversi contesti nazionali in cui l'azienda cliente opera, offrendo ad essa l'opportunità di lanciare programmi di loyalty differenziati per rafforzare ulteriormente la presenza del marchio all'estero.

8.1.4 Minacce

L'attuale situazione di pandemia rappresenta una minaccia per il progetto. La forte contrazione del mercato potrebbe portare a una diminuzione delle vendite per l'azienda, con una conseguente riduzione dell'utilizzo dell'app. Infatti, è evidente che, se verranno venduti meno prodotti, ne verrà registrato un numero inferiore nell'applicazione.

Sempre a causa della pandemia, gli eventi promossi mediante l'app, che spesso prevedono un'utilizzo della stessa presso i rivenditori, potranno essere fortemente impattati.

Iniziative di loyalty simili da parte di aziende concorrenti potrebbero ostacolare la fidelizzazione dei clienti e diminuire il senso di unicità percepito grazie alle promozioni pubblicizzate mediante l'app e ai premi messi in palio.

L'eventuale scarsa diffusione dell'app comporterebbe, infine, un annullamento di tutti i benefici del progetto per l'azienda cliente.

8.2 Sistemi affini e altre app con programmi di loyalty

Verranno ora analizzate le caratteristiche di alcuni programmi di loyalty lanciati tramite app.

8.2.1 Aliexpress

Aliexpress è un marketplace creato dal colosso cinese Alibaba Group, ed ha lo scopo di mettere in contatto piccole imprese cinesi con acquirenti di tutto il mondo per la vendita al dettaglio o all'ingrosso.

Per il suo programma di loyalty, Aliexpress utilizza una valuta virtuale, che l'utente può scambiare con buoni sconto o prodotti veri e propri. Ogni premio del programma di loyalty prevede, a monte, un accordo con i rivenditori iscritti alla

piattaforma, pertanto, un certo buono sconto sarà valido solo per gli articoli di un determinato venditore.

L'app di Aliexpress prevede una sezione dedicata esclusivamente alla loyalty, riportata nella Figura 8.2; da tale sezione è possibile accedere a tutte le iniziative del programma.

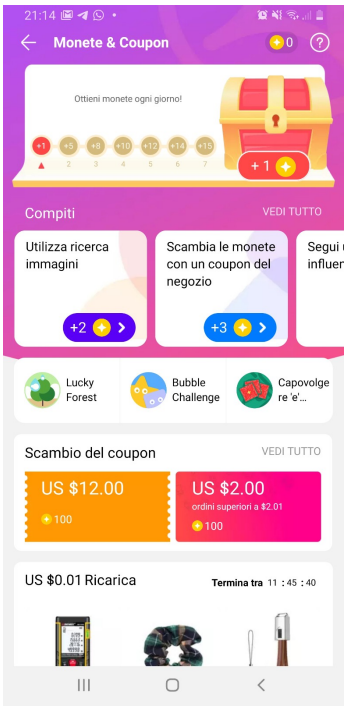

Figura 8.2. La schermata principale della sezione loyalty di Aliexpress.

I metodi per accumulare punti sono svariati. Il più semplice è quello di effettuare giornalmente l'accesso all'app; con accessi consecutivi, è possibile ottenere ricompense sempre più sostanziose. Abbiamo, poi, una serie di "Compiti" che prevedono interazioni con i venditori, l'utilizzo di particolari funzionalità dell'app o l'acquisto di determinati prodotti.

Una parte molto interessante della sezione è quella dedicata ai giochi; è stata sviluppata, infatti, una serie di giochi interni all'app, che permette di accumulare punti validi per la loyalty. Una particolarità dei giochi disponibili è quella di avere degli obiettivi che l'utente può condividere con i propri amici, guadagnando ancora più punti. In questo modo viene aumentata la diffusione dell'app tra i conoscenti dell'utente.

La sezione giochi, così come quella dei "Compiti" viene aggiornata periodicamente in modo da aumentare sempre più le interazioni dell'utente con l'app. Nella Figura 8.3 è riportata la schermata di uno dei giochi disponibili.

Anche Aliexpress possiede un sistema di questionari a cui si può rispondere per accumulare punti; inoltre, ogni acquirente viene premiato per recensire i prodotti

Figura 8.3. Il gioco "Lucky tree" di Aliexpress.

acquistati tramite il marketplace.

Il programma di loyalty di Aliexpress rappresenta un esempio molto ben riuscito di campagna di fidelizzazione, ricco di contenuti molto vari e coinvolgenti, in grado di mantenere un engagement costante degli utenti.

8.2.2 Gearbest

Gearbest è una piattaforma per la vendita al dettaglio di elettronica e abbigliamento. A differenza di Aliexpress, in questo caso, abbiamo una holding, denominata Globalegrow, che possiede numerosi negozi i cui prodotti vengono venduti tramite Gearbest. Anche questa piattaforma prevede un programma di loyalty a punti, che possono essere utilizzati per ottenere uno sconto sui propri ordini. Ad ogni ordine completato l'utente guadagna punti; inoltre, più si effettuano acquisti più il proprio account sale di livello nel programma loyalty, così da beneficiare di maggiori sconti e poter partecipare a lotterie ed estrazioni a premi. Anche su Gearbest viene conferito un bonus agli utenti che effettuano giornalmente l'accesso all'app. Nella Figura 8.4 è riprotata la sezione "VIP", ovvero schermata di riepilogo del livello loyalty dell'utente; in essa vengono mostrati i bonus di livello, gli eventi a cui si può partecipare e le missioni per aumentare il proprio "livello VIP".

Un ulteriore modo per ottenere punti consiste, anche in questo caso, nel recensire i prodotti acquistati. Un'ulteriore caratteristica del programma loyalty di Gearbest

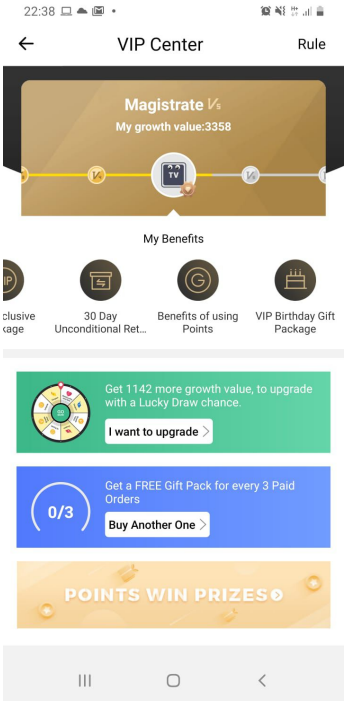

Figura 8.4. La schermata "VIP" di Gearbest.

è la scadenza periodica dei punti che, se da un lato potrebbe spingere i clienti ad acquistare più spesso, dall'altro rischia di fidelizzarli in modo meno duraturo.

8.2.3 Mulino per me

"Mulino per me" è l'applicazione lanciata da Barilla per gestire le iniziative di loyalty relative alla linea di prodotti "Mulino Bianco". Questa app presenta una struttura simile a quella realizzata e discussa nella presente tesi, trattandosi di app per una raccolta punti relativa a prodotti alimentari. Dopo aver effettuato la registrazione, l'app invita a rispondere ad un questionario che ha lo scopo di individuare i prodotti preferiti dell'utente. Successivamente si viene portati alla schermata mostrata in Figura 8.5, in cui sono elencate tutte le iniziative in corso.

Cliccando su uno dei concorsi viene mostrata la schermata riportata in Figura 8.6, con il saldo punti e i premi che è possibile richiedere. Per accumulare punti è richiesta la registrazione in app dei codici presenti sulle confezioni dei prodotti acquistati.

Analogamente all'app realizzata, "Mulino per me" presenta anche una sezione con le domande frequenti inerenti al programma di loyalty e ai prodotti "Mulino Bianco" in generale.

Figura 8.5. La schermata principale di "Mulino per me".

8.2.4 Coccole Pampers

Pampers è un marchio di pannolini posseduto dalla multinazionale Procter & Gamble e commercializzato in tutto il mondo. Il programma di loyalty Pampers prevede l'utilizzo dell'app per la raccolta di punti, con i quali è possibile richiedere premi. I premi del catalogo non sono a marchio Pampers, bensì giocattoli, ingressi a parchi divertimenti e sconti presso rivenditori che offrono articoli per neonati e per la prima infanzia.

Anche in questo caso l'utente riceve punti in seguito al caricamento di codici contenuti nelle confezioni dei prodotti acquistati. Un altro modo per ottenere punti consiste nel rispondere periodicamente a dei questionari proposti dall'app.

Nella Figura 8.7 sono riportate alcune schermate dell'applicazione "Coccole Pampers"

8.2.5 Confronto con l'app realizzata

In questo capitolo sono state descritte diverse app di supporto a iniziative di loyalty. Rispetto all'app realizzata queste presentano differenze e affinità. Tra i prodotti analizzati, Aliexpress offre sicuramente l'esperienza più "engaging", grazie alla varietà delle modalità con cui si possono raccogliere punti, che coinvolgono in modo continuo sia l'utente che i suoi amici. Per quanto riguarda Gearbest, anch'esso offre un buon grado di engagement grazie ai numerosi eventi in app promossi; inoltre,

Figura 8.6. Una schermata di "Mulino per me" relativa a un concorso.

Figura 8.7. Alcune schermate dell'applicazione "Coccole Pampers".

i vantaggi ottenuti con i punti fedeltà sono molto sostanziosi in termini di risparmio sugli acquisti effettuati. Rispetto all'app realizzata, Aliexpress e Gearbest sono caratterizzati da un migliore grado di integrazione con le rispettive piattaforme di vendita. Questo è stato possibile, in primo luogo, perchè le loro iniziative di loyalty sono nate come feature delle piattaforme stesse; tuttavia, queste ultime sono gli unici canali di vendita a cui i suddetti programmi di loyalty si riferiscono. La presenza di giochi ed eventi sempre nuovi, nonchè l'integrazione in app di una piattaforma per e-commerce sono caratteristiche che apporterebbero sicuramente all'applicazione maggior valore.

Bisogna considerare, tuttavia, che le raccolte punti di Aliexpress e Gearbest si riferiscono ad utenti che acquistano esclusivamente online e utilizzando l'apposita piattaforma, che è rivolta ad acquirenti provenienti da tutto il mondo. Al contrario, l'app realizzata è studiata per un contesto di tipo domestico e consente l'adesione alle iniziative loyalty anche da parte di quegli utenti che acquistano i prodotti dell'azienda cliente tramite canali di vendita più tradizionali, quali, ad esempio, i supermercati. In questo modo l'applicazione si pone come una versione digitale di quelle raccolte punti a cui chi acquista i prodotti dell'azienda è già abituato da molti anni.

A tal proposito sono state considerate per il confronto anche le app "Mulino per me" e "Coccole pampers", relative a programmi di loyalty concettualmente molto più simili a quello dell'app realizzata, anche nelle modalità di collezionamento punti. L'app analizzata in questa tesi risulta, rispetto ai prodotti affini considerati, molto valida. Essa comprende tutte funzionalità presenti in "Mulino per me" e "Coccole pampers", inoltre, tramite la sezione magazine e gli eventi pubblicizzati, offre dei contenuti che vanno oltre il semplice premiare il cliente per l'acquisto di un prodotto e, contemporaneamente, contribuisce a diffondere i valori caratterizzanti la mission dell'azienda cliente.

9

Conclusioni e sviluppi futuri

In questa tesi è stato introdotto lo sviluppo di applicazioni per smartphone in ambiente iOS, e sono state presentate le principali librerie native fornite da Apple, nonchè le evoluzioni del sistema operativo nel corso degli anni. Si è passati poi ad analizzare l'architettura MVC nella sua implementazione proposta da Apple, descrivendone i punti di forza e le criticità, superate tramite l'adozione del pattern architetturale MVVM. Anche per quanto riguarda il sistema di navigazione tra le schermate delle app sono stati presentati parallelamente il metodo standard proposto da Apple ed il pattern Flow Coordinator, enfatizzando le differenze tra i due approcci e le potenzialità di riutilizzo del codice offerte dall'ultimo.

È stata, poi, approfondita l'app realizzata. Quest'ultima è stata richiesta da un'azienda italiana, leader del settore alimentare, ed ha la funzione di supportare l'azienda stessa nelle sue iniziative di loyalty, nonchè nella promozione di nuovi prodotti e nella raccolta di dati e informazioni sui propri clienti. Per ciascuna sezione dell'app sono state descritte le principali funzionalità sia dal punto di vista grafico che implementativo. È stata data particolare importanza alle librerie utilizzate, e ci si è soffermati in modo specifico sui design pattern e le best practice adottati, elementi che hanno consentito di ottenere un prodotto di altà qualità.

È stata effettuata, successivamente, un'analisi SWOT del progetto, al fine di individuare le potenzialità, i limiti e le opportunità di crescita da esso introdotti. Infine, sono state analizzate le caratteristiche principali di altre applicazioni per smartphone dedicate ad iniziative di loyalty. Queste erano indirizzate, in alcuni casi, a contesti domestici, in altri ad un contesto mondiale. Tali applicazioni sono state, poi, confrontate con l'app realizzata, evidenziando analogie e differenze.

L'app si presta a numerosi sviluppi futuri. Il più importante di questi è, sicuramente, l'integrazione di funzioni per l'e-commerce, che consentirebbe un'esperienza utente più omogenea e coinvolgente per coloro che, dopo aver scoperto un prodotto tramite l'app, sono intenzionati ad acquistarlo. In questo modo i contenuti promozionali diverrebbero ancora più efficaci, a sarebbe sufficiente, per i clienti, un numero di click inferiore per concludere un ordine. Oltre alla possibilità di acquistare prodotti direttamente nell'app potrebbero essere introdotti il controllo dello stato dei propri ordini, nonchè la gestione dei codici promozionali riservati al singolo utente. Quest'ultima, al momento, è legata alla piattaforma SAP Hybris; pertanto, per tenere traccia degli sconti che gli sono stati riservati, l'utente può fare affidamento al

proprio indirizzo e-mail. La funzionalità FAQ, infine, potrebbe essere ampliata con la possibilità di parlare con un operatore, trasformandosi in un sistema di assistenza clienti vero e proprio.

Riferimenti bibliografici

1. UICollectionView - UIKit `https://developer.apple.com/documentation/uikit/uicollectionview`.
2. Avoiding retain cycles in Swift `https://medium.com/mackmobile/avoiding-retain-cycles-in-swift-7b08d50fe3ef`.
3. R. Cacheaux and J. Berlin. *Advanced iOS App Architecture Real-World App Architecture in Swift*. Razeware LLC, 2019.
4. Cards UI design: fundamentals and examples `https://www.justinmind.com/blog/cards-ui-design/`.
5. J. Clayton, A. Gallagher, M. Galloway, E. Ganem, and C. Pupăză S. Van Impen E. Kerber, B. Morrow. *Swift Apprentice*. Razeware LLC, 2019.
6. Gigya's documentation `https://developers.gigya.com/`.
7. J. Greene and J. Strawn. *Design Patterns by Tutorials*. Razeware LLC, 2018.
8. J. Hoffman. *Mastering Swift*. Packt Publishing, 2015.
9. Apple Inc. *The Swift Programming Language(Swift 5.2)*. Apple Inc., 2018.
10. C. Keur and A. Hillegass. *Ios Programming, The Big Nerd Ranch Guide*. Big Nerd Ranch, 2020.
11. GitHub - Kingfisher. `https://github.com/onevcat/Kingfisher`.
12. UIAlertController in Swift `https://medium.com/swift-india/uialertcontroller-in-swift-22f3c5b1dd68`.
13. M. Neuburg. *Ios 13 Programming Fundamentals With Swift: Swift, Xcode, and Cocoa Basics*. Oreilly Associates, 2019.
14. M. Neuburg. *Programming iOS 13: Dive Deep into Views, View Controllers, and Frameworks*. O'Reilly Media, 2019.
15. NotificationCenter - Foundation `https://developer.apple.com/documentation/foundation/notificationcenter`.
16. Design Patterns — A quick guide to Observer pattern. `https://medium.com/datadriveninvestor/design-patterns-a-quick-guide-to-observer-pattern-d0622145d6c2`.
17. Promises `https://github.com/google/promises`.
18. J. Ray. *iOS 9 Application Development in 24 Hours*. Sams, 2016.
19. iOS Storyboards: Getting Started `https://www.raywenderlich.com/5055364-ios-storyboards-getting-started`.
20. J. Reid. *iOS Unit Testing by Example XCTest Tips and Techniques Using Swift*. Pragmatic Bookshelf, 2019.
21. E. Sadun. *iOS Auto Layout Demystified (2nd Edition) (Mobile Programming)*. Addison-Wesley Professional, 2013.

22. How To Make a Custom Control Tutorial: A Reusable Slider `https://www.raywenderlich.com/2297-how-to-make-a-custom-control-tutorial-a-reusable-slider#toc-anchor-007`.
23. N. Smyth. *SwiftUI Essentials iOS Edition: Learn to Develop iOS Apps using SwiftUI, Swift 5 and Xcode 11*. Payload Media, 2018.
24. UIStackView - UIKit `https://developer.apple.com/documentation/uikit/uistackview`.
25. UITableView - UIKit `https://developer.apple.com/documentation/uikit/uitableview`.