UNIVERSITÀ DEGLI STUDI DI CAMERINO

SCUOLA DI SCIENZE E TECNOLOGIE

Corso di Laurea in Informatica Classe L-31

STUDIO E SVILUPPO IN LINGUAGGIO SWIFT DI UN'APPLICAZIONE MOBILE

Tesi di Laurea di:

Lorenzo Brancaleoni

Relatore:

Ch.mo Prof. Fausto Marcantoni

Correlatore:

Dott.ssa Manuela Benèdetti

Anno accademico 2019/2020

ABSTRACT

La tesi di Laurea descrive il lavoro svolto durante il periodo di stage dal laureando Lorenzo Brancaleoni presso l'azienda "Be Ready Software" di Corridonia, in provincia di Macerata. Lo stage è stato svolto come conclusione del percorso di studi della Laurea Triennale in Informatica, richiedendo una quantità di lavoro pari a trecento ore. Grazie a questa esperienza è stato possibile redigere questa tesi di Laurea. L'obiettivo della tesi è quello di approfondire le nozioni teoriche apprese durante lo stage, esponendo anche un esempio di applicazione sviluppata in linguaggio *Swift* durante il periodo di stage. In ordine viene descritta l'esperienza personale dello stage e presentato brevemente il contesto aziendale. Successivamente viene introdotto l'ambiente di lavoro *Xcode* utilizzato per sviluppare applicazioni native *iOS*, in seguito viene descritto approfonditamente il linguaggio *Swift*. In conclusione viene presentata l'applicazione svolta durante lo stage.

Ringraziamenti

Arrivare a questo obiettivo non è stato per nulla semplice, durante il percorso ho incontrato diverse difficoltà che inevitabilmente hanno prolungato i tempi di questa Laurea, prima su tutte il terremoto del 2016; fortunatamente l'università di Camerino ha saputo reagire ed ha permesso a noi studenti di continuare questo percorso nel migliore dei modi e per questo ci tengo a ringraziare UNICAM e tutti i suoi collaboratori.

Successivamente vorrei ringraziare l'azienda "Be Ready Software" che mi ha ospitato nel corso dello stage grazie al quale ho potuto realizzare questa tesi di Laurea, supervisionata dal professor Fausto Marcantoni, relatore di questo documento, il quale si è dimostrato sempre gentile, disponibile ed esaustivo nonostante i tempi siano stati stringenti, la tutor aziendale e correlatrice del presente documento, Manuela Benedetti, che mi ha seguito costantemente in questo percorso.

Desidero ringraziare con affetto la mia famiglia, i miei genitori, mio fratello e la mia ragazza per il sostegno fornitomi e per essermi stati vicini durante gli anni di studio, tutti i miei amici e compagni di università che hanno collaborato con me nei vari progetti ed in particolare voglio rivolgere un ringraziamento speciale al mio amico Gioele. Senza tutti loro probabilmente non avrei raggiunto questo traguardo.

Ringrazio tutti di cuore.

Con soddisfazione ed orgoglio.

Lorenzo Brancaleoni

STUDIO E SVILUPPO IN LINGUAGGIO SWIFT DI UN'APPLICAZIONE MOBILE

Lorenzo Brancaleoni

INDICE

INDICE DELLE FIGURE

CAPITOLO 1

Introduzione

La multinazionale americana ***Apple Inc.***, fondata nel 1976 da Steve Jobs, Steve Wozniak e Ronald Wayne, da sempre leader nella produzione di computer con interfacce grafiche innovative, è sempre stata un'azienda all'avanguardia per quanto riguarda le tecnologie sviluppate al suo interno, affermandosi sui mercati, nazionali ed internazionali, tramite l'elettronica di consumo e prodotti di lusso come iPhone, iPad, iPod e Mac ecc. Nonostante il colosso statunitense faccia parlare costantemente di sé grazie alle nuove tecnologie che ogni anno rilascia, la vera e propria rivoluzione nel mondo dei programmatori Apple è stata introdotta con il rilascio, nel settembre 2014, di un nuovo linguaggio di programmazione: ***Swift.*** Già dalla prima release, è fino a 8,4 volte più veloce di *Python* e fino a 2,6 volte più veloce di *Objective-C* in alcuni tipi di algoritmi. *Swift* si è subito conquistato il primo posto tra i programmatori a tempo pieno di applicazioni per iOS e OS X, abituati alla sintassi dell'*Objective-C*, riuscendo anche ad entrare nella Top 10 dei linguaggi più utilizzati nel 2015 e tutt'ora ne continua a far parte [1]. Si è pertanto deciso, attraverso questa tesi di Laurea, di approfondire lo studio del nuovo linguaggio di programmazione concludendo con lo sviluppo di un'applicazione iOS.

1.1 Struttura del documento

Il seguente documento è strutturato in 6 capitoli: il primo capitolo si occupa di introdurre e spiegare in linea generale i dettagli principali della tesi; nel secondo capitolo si analizza l'esperienza dello stage introducendo le basi dell'applicazione portata nella tesi; il terzo e quarto capitolo si occupa di spiegare la parte teorica approfondendo gli argomenti trattati durante lo stage. Nel quinto capitolo viene introdotto e presentato il progetto realizzato sulla base di quanto appreso in precedenza. Nell'ultimo capitolo, si discute la conclusione della tesi, i risultati ottenuti ed una valutazione personale dell'esperienza avuta.

1.2 Obiettivi della tesi

Lo scopo di questa tesi di Laurea è quello di approfondire le nozioni teoriche sul linguaggio *Swift* apprese durante lo Stage, effettuato presso l'azienda "Be Ready Software", presentando anche un'applicazione mobile iOS sviluppata in linguaggio *Swift* denominata "*ARTILS*", realizzata durante lo stage.

1.3 Problemi affrontati

Durante lo stage e quindi durante lo sviluppo dell'applicazione la grande maggioranza dei problemi riscontrati sono stati di natura tecnica, specialmente riguardo al nuovo linguaggio di programmazione per lo sviluppo di applicazioni iOS in cui sono stato inserito dall'azienda ospitante, affrontando perciò tutte le varie problematiche che ne conseguono.

CAPITOLO 2

Lo stage

È stato possibile redigere il seguente documento di Laurea grazie allo stage universitario presso l'azienda *"*Be Ready Software*",* una piccola realtà imprenditoriale situata nella città di Corridonia, in provincia di Macerata che si occupa principalmente di programmazione di applicazioni mobile native. Per tutto il progetto sono stato supervisionato dalla tutor aziendale, nonchè correlatrice del presente documento di Laurea, la dottoressa Manuela Benedetti.

2.1 Illustrazione del progetto

Dopo aver effettuato uno studio individuale che mi ha permesso di avvicinarmi al mondo dello sviluppo di applicazioni mobile, la tutor aziendale ha scelto di inserirmi in un progetto in corso d'opera. Sostanzialmente si tratta di sviluppare un'applicazione mobile nativa per iOS denominata *"ARTILS"* (*Augmented Real Time Learning for Secure workspace*). L'idea del progetto consiste nella realizzazione di una piattaforma per il supporto alla formazione per la sicurezza negli ambienti di lavoro, garantendo l'accesso sicuro a banche dati e contenuti riservati. I compiti che mi venivano assegnati dalla tutor aumentavano progressivamente permettendomi così di poter procedere gradualmente con lo sviluppo dell'applicazione.

2.2 Vincoli metodologici

Di comune accordo è stato deciso con la tutor aziendale che il lavoro fosse interamente svolto presso la sede dell'azienda, vista la necessità di un confronto continuo permettendomi così di avere giornalmente direttive su come procedere con il lavoro e creando un ambiente dove il dialogo potesse giovare ad entrambe le parti.

2.3 Vincoli temporali

Lo stage si è svolto per un totale di 300 ore complessive di lavoro. Queste ore sono state distribuite in modo uniforme nell'arco di due mesi circa in settimane da 40 ore lavorative ciascuna. L'orario di lavoro accordato tra me e la tutor aziendale è stato dal lunedì al venerdì

dalle 09:00 alle 18:00, con un'ora di pausa pranzo. La scansione temporale delle attività svolte è avvenuta *in itinere* con base giornaliera, in quanto correlata anche alle nuove richieste da parte del cliente. Nello specifico la prima settimana è stata interamente dedicata allo studio individuale del linguaggio *Swift*, in seguito il lavoro è proseguito con lo sviluppo dell'interfaccia grafica dei moduli e l'implementazione delle relative funzionalità.

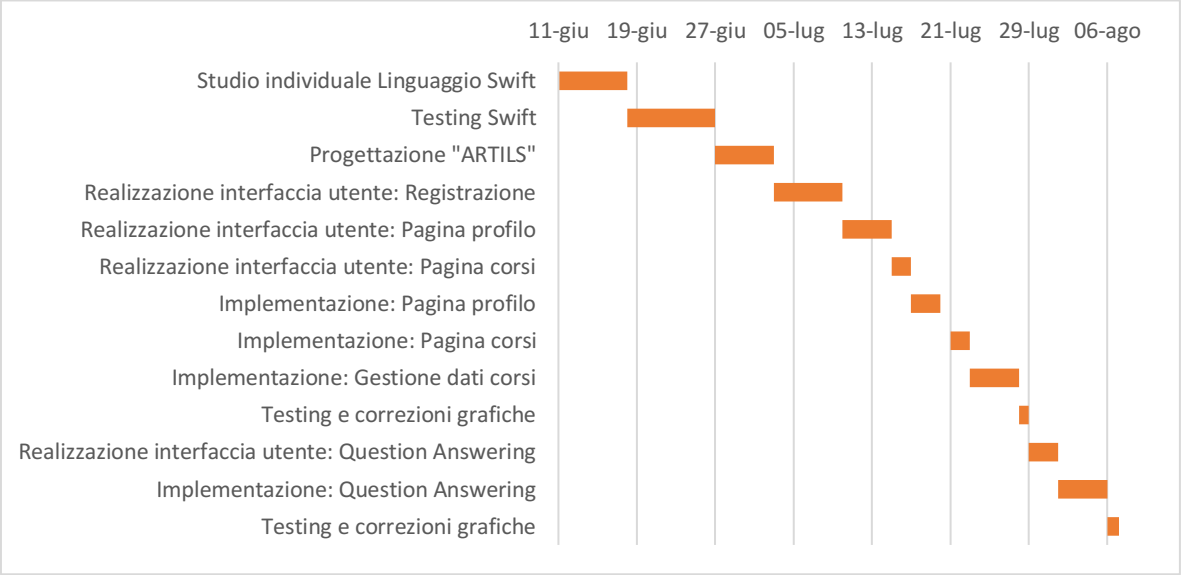

***Figura 2.1**: Diagramma di Gantt del lavoro svolto*

2.4 Ambiente di lavoro

Per quanto riguarda l'ambiente di lavoro ho dovuto fare utilizzo di un nuovo software ovvero il sistema operativo di casa *Apple* macOS dato che per sviluppare applicazione in linguaggio *Swift* è necessario lavorare con il software prodotto da *Apple*. Quindi, dovendo utilizzare strumenti a me sconosciuti, ho dovuto all'inizio impiegare una porzione del tempo lavorativo anche per familiarizzare con i nuovi strumenti di lavoro.

2.4.1 Hardware

Durante tutto lo stage ho utilizzato un computer messo gentilmente a disposizione dall'azienda, un *MacBook Pro*, mentre per analizzare e testare l'applicazione durante lo sviluppo ho utilizzato un iPhone 6s e un iPad con versione del sistema operativo iOS 13 entrambi messi a disposizione dall'azienda

2.4.2 Software

Per lo sviluppo ho utilizzato l'IDE *Xcode*, un ambiente di lavoro integrato completamente sviluppato e mantenuto da *Apple* tramite cui è possibile compilare il linguaggio di programmazione *Swift* per lo sviluppo di applicazioni mobile native. Per il controllo di versione l'azienda si affida a *Git* sfruttando nello specifico *Bitbucket* ovvero un servizio di hosting web-based per progetti che usano i sistemi di controllo versione Mercurial o Git, nel mio caso per interfacciarmi con Bitbucket aziendale sono ricorso a linea di comando con Git, i repository sono organizzati sfruttando nella maniera corretta la suddivisione dei progetti in *branch*.

CAPITOLO 3

Xcode

Xcode è un ambiente di sviluppo integrato (*Integrated development environment*, *IDE*), completamente sviluppato e mantenuto da *Apple*, contenente una suite di strumenti utili allo sviluppo di software per i sistemi macOS, iOS, watchOS e tvOS.

Figura 3.1: *Logo Xcode*

Precedentemente era fornito gratuitamente in *bundle* con il sistema operativo, di recente invece non è più così ma è possibile scaricarlo gratuitamente dal *Mac App Store*. Esso estende e rimpiazza il precedente tool di sviluppo della *Apple*, *Project Builder*, che era stato ereditato dalla *NeXT,* e lavora in congiunzione con *Interface Builder* (proveniente da *NeXT*), un tool grafico per realizzare interfacce grafiche.[2]
Per gli sviluppatori iOS, *Xcode* è uno strumento indispensabile per la compilazione e il debug del codice, la creazione di interfacce utente, la lettura della documentazione, l'invio di app all'App Store e molto altro ancora.

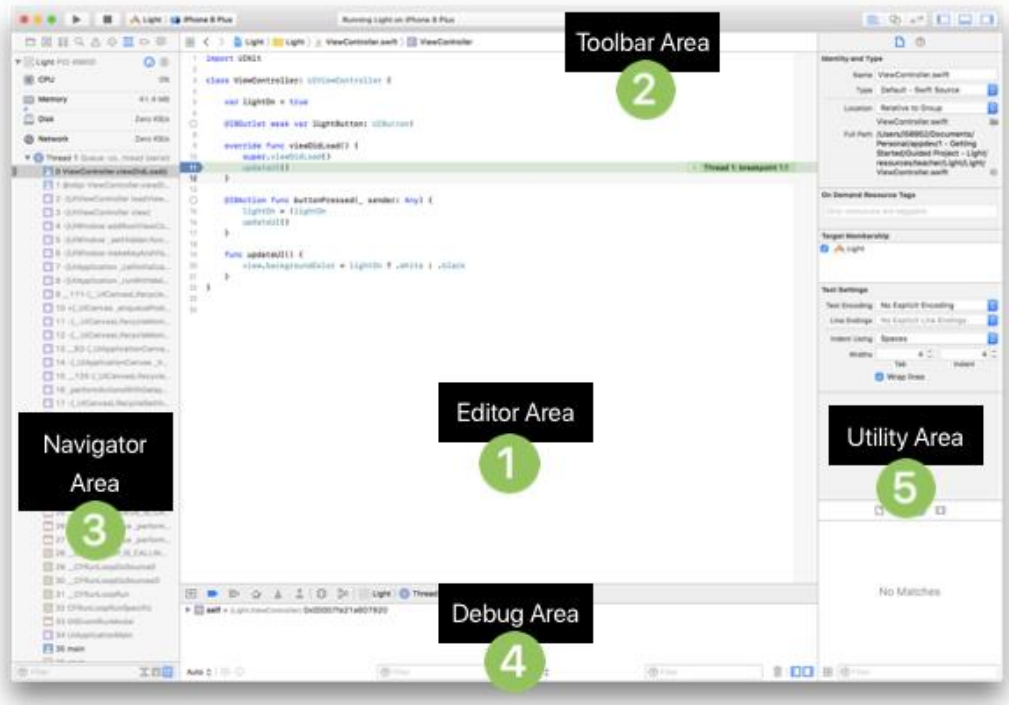

Figura 3.2 : *Esempio Schermata Xcode*

Xcode permette di testare le applicazioni tramite due modalità: utilizzando un simulatore oppure utilizzando un dispositivo fisico (iPhone, iPad ecc..). Se si utilizza il simulatore non occorre creare un account sviluppatore, però ciò comporta alcuni svantaggi, per esempio utilizzando il simulatore si ha un'incompatibilità con le notifiche "*push*", per questo è necessario testare le applicazioni su un dispositivo fisico, per fare ciò però come detto in precedenza occorre disporre di un account per sviluppatore. Fortunatamente questo passaggio è totalmente gratuito, mentre se si ha necessità di distribuire le app su più dispositivi o pubblicarle sull'App Store, sarà necessario sottostare ad un abbonamento chiamato *Apple Developer Program*.

Le funzionalità previste dal software sono molteplici tra cui il *Debug*, infatti se si esegue l'app sul simulatore o sul dispositivo, *Xcode* collegherà l'app al suo *debugger*. Ciò consente di osservare l'esecuzione del codice in tempo reale, interrompere l'esecuzione del codice utilizzando i punti di interruzione, stampare le informazioni dal codice alla console e molto altro. Un'altra funzionalità molto utile è quella del *Assistant Editor* che consiste nel completamento automatico durante la scrittura prevedendo anche le signature delle funzioni.

3.1 Interfaccia grafica

Xcode ha uno strumento integrato chiamato ***Interface Builder*** che semplifica la creazione visiva delle interfacce. *Interface Builder* si apre ogni volta che si seleziona un file *xib* (*.xib*) o un file *storyboard* (*.storyboard*) dal navigatore del progetto. Un file *xib* contiene l'interfaccia utente per un singolo elemento visivo, come una visualizzazione a schermo intero, una cella o un controllo dell'interfaccia utente personalizzato. Gli xib venivano utilizzati maggiormente prima dell'introduzione degli *storyboard*. Sono però ancora molto utili in determinate situazioni, per esempio se si vuole creare una cella e renderla utilizzabile in più contesti senza dover reiterarne la creazione. A differenza di uno xib, un file *storyboard* include molti pezzi dell'interfaccia, definendo il *layout* di uno o più schermi e la progressione da uno schermo all'altro.

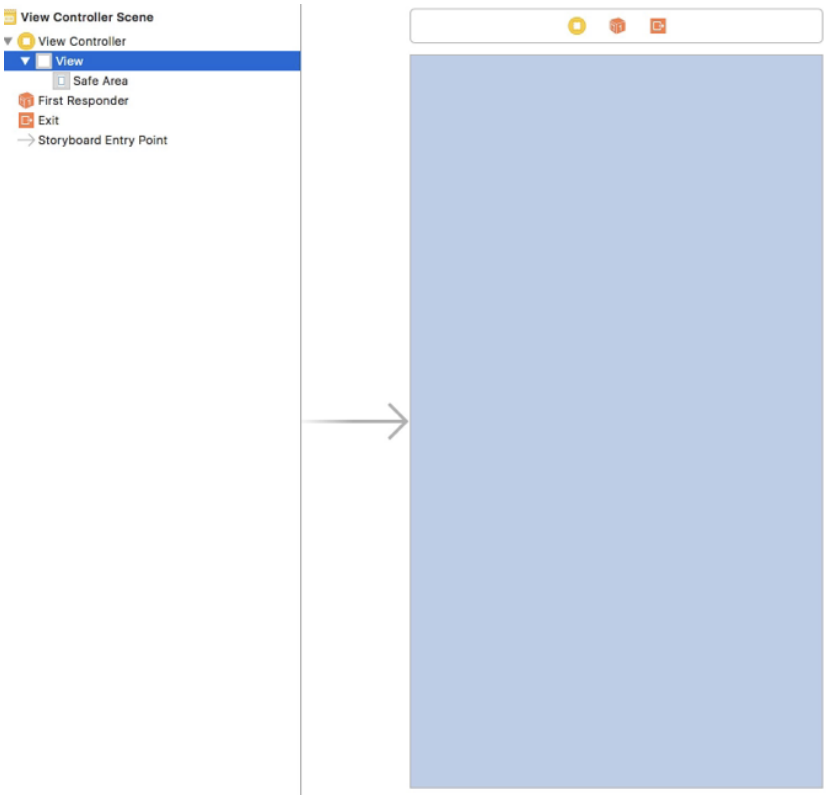

Figura 3.3: *View Controller Scene*

La figura 3.3 mostra un esempio di *View Controller* che contiene al suo interno una *View* vuota che possiamo solitamente incontrare alla prima creazione di un file *.storyboards.* La freccia che troviamo in grigio sta ad indicare lo *Storyboard Entry Point* ovvero la prima schermata che troveremo all'avvio della nostra applicazione.

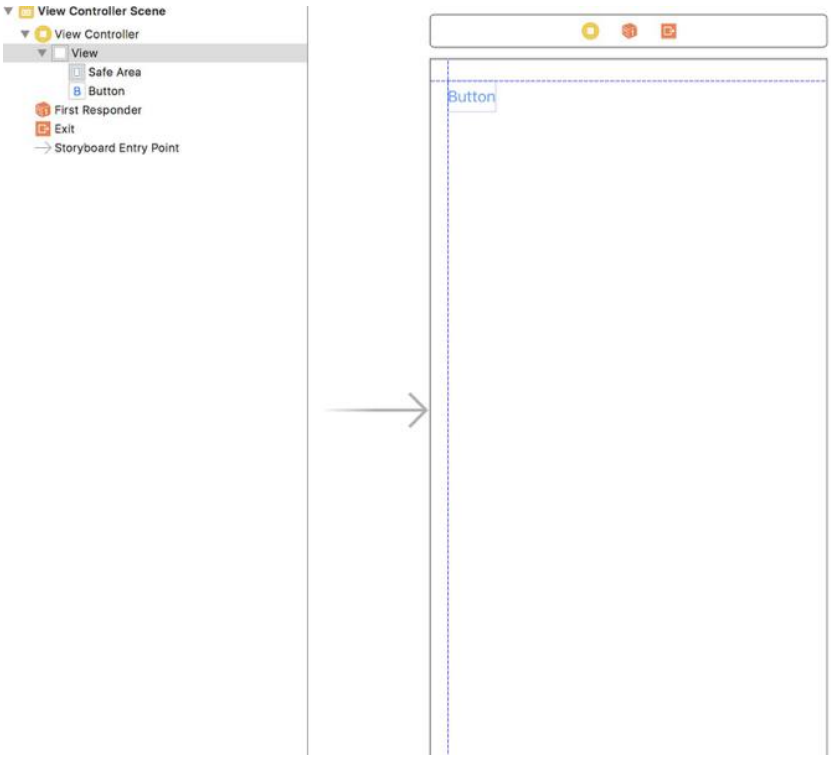

Figura 3.4: *Inserimento di un bottone nella View*

Per inserire elementi all'interno della *View* occorre solamente scegliere l'oggetto desiderato all'interno della libreria e trascinarlo nella *View,* mentre effettuiamo questa operazione ci viene in aiuto la guida di Layout prevista da *Xcode* permettendo, mediante l'utilizzo di margini, di poter centrare il contenuto.

Un aspetto fondamentale che *Xcode* mette a disposizione per l'interfaccia grafica sono i ***constraints*** che l'utente inserirà manualmente per ogni oggetto presente all'interno del *View Controller Scene.* Grazie ai *constraint* sarà possibile rendere il contenuto grafico dell'applicazione totalmente *responsive* ovvero ci sarà un adattamento automatico delle dimensioni degli oggetti in base al dispositivo e alla sua grandezza così da far risparmiare tempo prezioso al programmatore.

Per ogni oggetto che viene inserito si ha la possibilità, cliccando l'elemento desiderato, di poter accedere a quattro diversi campi *dall'Utility area*:

- *Identity Inspector*
- *Attributes inspector*
- *Size inspector*
- *Connection Inspector*

Identity Inspector consente di modificare le proprietà relative all'identità di un oggetto, come la classe a cui appartiene. In questo esempio, il pulsante appartiene alla classe *UIButton*. Se avessimo definito una classe di pulsante personalizzata altrove, sarebbe stato possibile utilizzare *l'identity inspector* per modificare la classe del pulsante appena aggiunto.

Attributes inspector fornisce un elenco di proprietà visive regolabili per l'elemento selezionato. Nel caso di un pulsante *UIButton*, è possibile modificare attributi come il testo del pulsante, il colore del testo, lo sfondo e l'allineamento.

Size Inspector permette di regolare le dimensioni e la posizione dell'elemento selezionato all'interno della *View Controller Scene.* Verranno utilizzati i campi X e Y. Il valore X aumenta man mano che viene spostato l'elemento più a destra e Y aumenta man mano che ci si sposta sullo schermo verso l'alto. Per *UIButton*, *Size Inspector* include campi aggiuntivi per regolare la spaziatura interna intorno al titolo o per l'immagine del pulsante.

Connection inspector elenca tutte le funzioni e i nomi delle variabili relativi all'oggetto. Al momento, il pulsante non è connesso a nulla, quindi non si avrà alcuna connessione.

Outlets e Azioni

Durante lo sviluppo di un'applicazione sarà necessario fare riferimento agli elementi dell'*Interface builder* all'interno del codice, in modo che tali elementi possano essere modificati in fase di esecuzione o quando l'applicazione è già in esecuzione. Questo riferimento da *Interface Builder* al codice è chiamato ***outlet***. Quando si dispone di un oggetto con cui si desidera che l'utente interagisca, si crea un'azione, un riferimento a un pezzo di codice che verrà eseguito quando avrà luogo l'interazione.

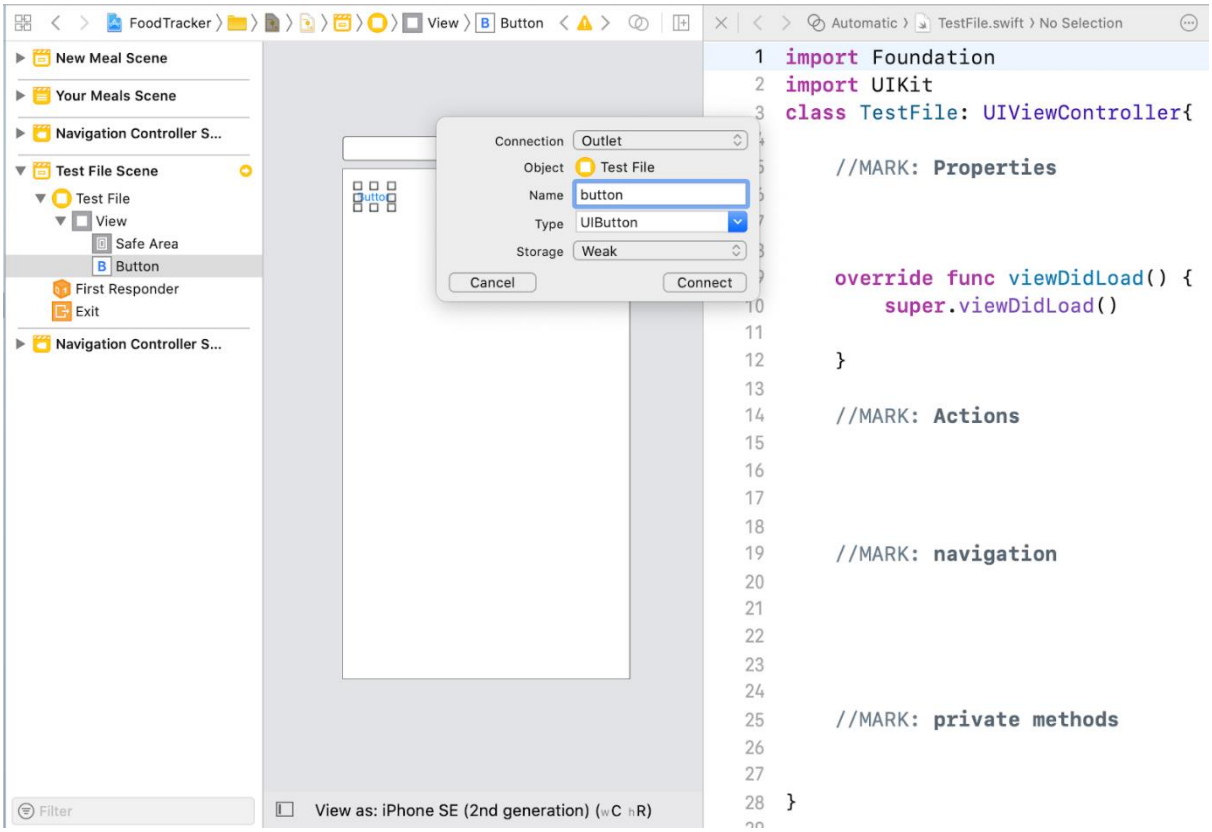

Figura 3.5: *Creazione Outlet*

Una volta creato *l'Outlet* occorrerà però creare anche l'azione mostrata in figura 3.6 che permetterà di eseguire appunto una specifica azione al click del pulsante durante l'esecuzione dell'applicazione.

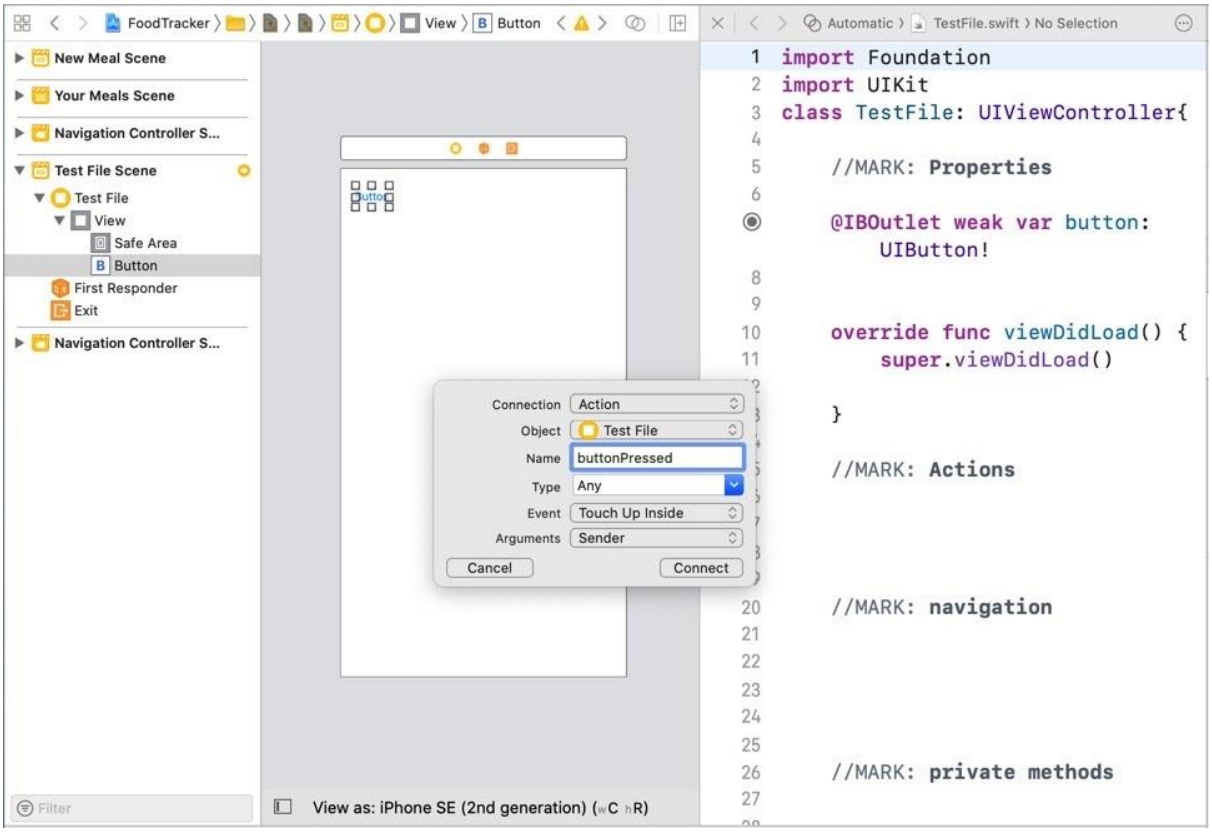

Figura 3.6: *Creazione di un'azione*

CAPITOLO 4

Swift

Durante *l'Apple Worldwide Developers Conference* 2014, Apple ha introdotto ***Swift*** come linguaggio moderno per scrivere app per iOS e macOS.

Apple ora ha nuove piattaforme, tra cui watchOS e tvOS, anch'esse utilizzano *Swift* come linguaggio di programmazione principale. Dagli anni '90, la maggior parte degli sviluppatori ha scritto applicazioni per piattaforme *Apple* in *Objective-C*, un linguaggio basato sul linguaggio di programmazione C, *Objective-C* ha più di 30 anni e C ha più di 40 anni, Entrambi forse i linguaggi più utilizzati nella storia e non sono ancora destinati ad andarsene nel prossimo futuro. Tuttavia, *Objective-C* può essere difficile da imparare, poiché la tecnologia è progredita velocemente negli ultimi anni, *Apple* ha visto l'opportunità di creare un linguaggio più moderno che fosse più facile da imparare e più facile da leggere, scrivere e mantenere. Sarà infatti possibile notare l'influenza di *C* ed *Objective-C* nel linguaggio *Swift[3]*.

Figura 4.1: *Logo Swift*

Swift è stato progettato da *Apple* con il preciso obiettivo di essere un linguaggio moderno, alcune delle caratetteristiche che lo rendono tale sono:

- Sintassi pulita, che rende il codice leggibile e più facile da lavorare,
- *Optionals* ovvero un nuovo modo di esprimere quando un valore potrebbe non esistere
- *Generics*, che aiutano gli sviluppatori a scrivere codice che può essere utilizzato in più scenari

Nei paragrafi successivi vengono introdotti i principali aspetti che caratterizzano il linguaggio ed approfonditi i punti precedentemente elencati partendo dalle variabili, i tipi di dato, fino ad

arrivare ad osservare il ciclo di vita di un'applicazione sviluppata in *Swift.* Per ogni paragrafo sarà presente un frammento di codice per chiarire al meglio i concetti introdotti.

4.1 Introduzione al linguaggio

Swift è considerato un linguaggio ***Type-Safe*** e **Type-Inference**:

- ***Type-Safe*** Questi linguaggi incoraggiano o richiedono al programmatore di essere chiaro sui tipi di valori con cui il codice può lavorare. Ad esempio, se una parte del codice prevede un *Int*, non è possibile passargli un *Double* o una *String*. Durante la compilazione del codice, *Swift* esegue questi controlli su tutte le costanti e le variabili e contrassegna come errori eventuali tipi non corrispondenti.
- **Type-Inference** *Swift* utilizza la *type-inference* per formulare ipotesi sul tipo di dato in base al valore assegnato alla costante o alla variabile.

4.1.1 Control flow

Nel seguente paragrafo saranno analizzati i tre operatori logici principali che permettono di usare le istruzioni del control flow per poter scegliere la porzione di codice da dover eseguire.

Condizione if

L'istruzione condizionale più semplice è l'istruzione *if*. Un'istruzione *if* fondamentalmente dice "Se questa condizione è vera, esegui questo blocco di codice". Se la condizione non è vera, il programma salterà il blocco di codice.

Condizione if-else

L'istruzione *if-else* sostanzialmente è uguale alla condizione *if* con la semplice differenza che se la condizione *if* non è soddisfatta entra in gioco la clausola *else* che permette l'esecuzione del relativo blocco di codice.

```
if testVariabile == "test"{
        testVariabile = "condizione vera"
}
else{
        testVariabile = "condizione falsa"

}
```

Guard
La maggior parte dei bug risiede in un codice complesso. Più semplice è il codice da leggere, più facile sarà individuare potenziali bug, in questo caso l'operatore ***guard*** (letteralmente guardia) permette di gestire al meglio il *control flow*.

```
//controllo che la variabile "testCostante" non sia nil
guard let testCostante = testCostante else {return}
```

Condizione switch
Il costrutto *switch* è ottimo per lavorare con molte condizioni potenziali. Un'istruzione *switch* di base assume un valore con più opzioni e consente di eseguire codice separato in base a ciascuna opzione o caso. È inoltre possibile fornire un caso predefinito per specificare un blocco di codice che verrà eseguito in tutti i casi non specificatamente definiti.

```
switch provaEnum {
case .Domenica :
        print("Domenica")
case .Lunedi :
        print("Lunedi")
default:
        print("error")
}
```

4.1.2 Gestione della memoria

Swift utilizza ***Automatic Reference Counting (ARC)*** per monitorare e gestire l'utilizzo della memoria nell'app. Nella maggior parte dei casi, questo significa che la gestione della memoria "funziona" in *Swift* e non è necessario pensare alla gestione della memoria da soli come accade invece nel linguaggio *C*. *ARC* libera automaticamente la memoria utilizzata dalle istanze di classe quando tali istanze non sono più necessarie. Tuttavia, in alcuni casi, *ARC* richiede maggiori informazioni sulle relazioni tra le parti del codice per poter gestire la memoria. Il *reference counting* è molto veloce a differenza del *garbage collector* utilizzato da altri linguaggi di programmazione quali *Java* e *Python*.
Ogni volta che viene creata una nuova istanza di una classe, *ARC* alloca un pezzo di memoria per salvare le informazioni su quell'istanza. Inoltre, quando un'istanza non è più necessaria, *ARC* libera la memoria utilizzata da quell'istanza in modo che la memoria possa essere utilizzata

per altri scopi. Tuttavia, se *ARC* dovesse deallocare un'istanza che era ancora in uso, non sarebbe più possibile accedere alle proprietà di quell'istanza e se provassimo ad accedere a tale istanza l'applicazione andrebbe in *crash.*

```
class Person {
    let name: String
    var city: City?

    init(name: String) {
        self.name = name

    }
    deinit { print("\(name) is being deinitialized") }
}

class City {
    var abitante: Person?
    var name: String
    init(name: String){
        self.name = name

    }

    deinit {
          print("\(name) is being deinitialized")

    }

    // Dichiariamo due variabili opzionali
    var person: Person?
    var macerata: City?
    // Creiamo due istanze delle rispettive classi
    person = Person(name: "lorenzo")
    macerata = City(name: "macerata")

    person!.city = macerata
    macerata!.name = person
    // Proviamo a deallocare la memoria senza successo
    person = nil
    macerata = nil
```

Come possiamo notare dal codice qui sopra viene creato un riferimento forte tra l'oggetto *city* e l'attributo *city* dell'oggetto *Person*; il simbolo " *!* " serve per forzare il compilatore al recupero dell'attributo all'interno della classe *Person*, *person* è infatti dichiarato come variabile opzionale, senza il punto esclamativo il compilatore proverebbe a ritornare l'istanza della classe

Person e non gli attributi al suo interno proprio perché l'istanza è di tipo opzionale. Quando poi si perde il riferimento della variabile *person,* il *reference counter* dell'oggetto puntato dalla variabile non scende a zero, bensì passa da 2 a 1 e pertanto l'oggetto non può essere deallocato, infatti dall'ARC non viene eseguita la funzione *deinit*. Lo stesso avviene se si cerca di deallocare l'oggetto *city* mettendo a *nil* la variabile *macerata.* Swift fornisce due modi per risolvere uno *strong reference cycle*: *weak references e unowed references*. I *weak references* e *unowed references* consentono a un'istanza in un ciclo di riferimento di fare riferimento all'altra istanza senza creare una referenziazione forte.
Perciò se nel nostro esempio cambiamo la dichiarazione di **var** abitante: Person? anteponendo la parola chiave *weak* cambiando così la dichiarazione: **weak var** abitante: Person? verrà risolto il problema.

4.2 UIKit

Il framework ***UIKit*** fornisce gli oggetti principali necessari per creare app per iOS e tvOS. Questi oggetti vengono utilizzati per visualizzare il contenuto sullo schermo, per interagire con quel contenuto e per gestire le interazioni con il sistema. Le app si affidano a *UIKit* per il loro comportamento di base e *UIKit* offre molti modi per personalizzare tale comportamento in base alle esigenze specifiche del programmatore[4].

4.2.1 Classi ed Ereditarietà

Per definire una classe occorre utilizzare la parola chiave ***class*** anteposta ad un nome univoco. Quindi si definiscono le proprietà come parte della classe elencando le dichiarazioni di costanti o variabili con l'annotazione di tipo appropriata. È possibile aggiungere funzionalità a una classe aggiungendo funzioni alla definizione della classe. Ogni classe può avere classi padre e figlio. Una classe genitore è chiamata superclasse e una classe figlia è chiamata sottoclasse. Le sottoclassi ereditano proprietà e metodi della superclasse, ma possono aumentare o sostituire l'implementazione dei metodi delle superclassi.

```
import Foundation
import UIKit

class TestFile: UIViewController {
}
```

In questo esempio la nostra classe *TestFile* eredità la classe *UIViewController*

Funzioni

L'idea di prendere qualcosa che è complesso e definire un modo più semplice per fare riferimento ad esso è un'astrazione e una funzione è uno dei modi fondamentali per creare un'astrazione nel codice[5].

Una funzione è composta da tre elementi: il suo nome, un elenco di parametri opzionali e un tipo di dato restituito opzionale. È possibile trasmettere informazioni mediante i parametri ma è anche possibile ottenere informazioni indietro ovvero un valore di ritorno. Una funzione in *Swift* può accettare zero, uno o più parametri, ma può anche restituire zero, uno o più valori. Quando una funzione restituisce un valore, si ha la possibilità di ignorarlo.

```
func test(parameter: String) -> String{

        return parameter
}
```

Come possiamo notare la dichiarazione di una funzione è simile alla dichiarazione di una variabile con la differenza che si utilizza la parola chiave "***func***" anteposta al nome che si vuole assegnare alla funzione. In questo esempio la funzione avrà come parametro(opzionale) una Stringa, e come valore di ritorno(opzionale) nuovamente una Stringa.

Override

Un vantaggio delle sottoclassi è che ogni sottoclasse può fornire la propria implementazione personalizzata di una proprietà o di un metodo. Per sovrascrivere una caratteristica che altrimenti verrebbe ereditata, come scrivere una nuova implementazione per una funzione, occorre anteporre alla nuova definizione la parola chiave ***override***.

```
override func viewDidLoad() {
    super.viewDidLoad()

}
```

In questo esempio si sta facendo l'override di un metodo molto importante, infatti è chiamato ogni volta che la *view* viene caricata in memoria. Perciò se bisogna aggiungere controlli che dovrebbero apparire insieme alla *view*, il codice per la creazione va inserito nel metodo viewDidLoad().

References
Una caratteristica speciale delle classi è la loro capacità di fare riferimento a valori assegnati a una costante o variabile. Quando si crea un'istanza di una classe, *Swift* seleziona una regione nella memoria del dispositivo per memorizzare quell'istanza. Quella regione in memoria ha un indirizzo. Le costanti o le variabili assegnate a quell'istanza memorizzano quell'indirizzo per fare riferimento all'istanza. Perciò la costante o la variabile non contiene il valore stesso, punta al valore in memoria.

4.2.2 Collections

Swift definisce due principali tipologie di ***Collections***: ***Array*** e ***Dictionary***. Ogni tipo fornisce un metodo univoco per interagire con più oggetti. Entrambi condividono determinate funzionalità, per esempio l'aggiunta e la rimozione di elementi e l'accesso a singoli elementi.

Array
Il tipo di *collection* più comune in *Swift* è l'***array***, che memorizza un elenco ordinato di valori dello stesso tipo. Quando si dichiara un *array*, è possibile specificare il tipo di valori che verrà conservato nella *collection* oppure lasciare che il *type inference system* scopra il tipo.

```
var array = ["1", "2", "3", "4"]
```

Una volta definito un *array* potrebbe essere necessario aggiungere nuovi valori all'interno dell'*array* appena creato, ciò può essere semplicemente fatto mediante l'utilizzo degli operatori " + " e " - ", oppure se si vuole aggiungere o rimuovere un elemento situato in una specifica posizione è possibile utilizzare la funzione ***insert(_: at:)*** o ***remove(_: at:)***.

```
//inserimento elemento alla posizione zero
array.insert("inserimento", at: 0)

//rimozione elemento alla posizione zero
array.remove(at: 0)
```

Dictionaries
Un ***Dictionary*** in *Swift* è un elenco di chiavi, ciascuna con un valore associato. Ogni chiave deve essere unica, proprio come ogni parola nel dizionario è unica. Puoi impostare un *dictionary* utilizzando un elenco di coppie chiave / valore separate da virgole racchiuse tra parentesi quadre. I due punti separano ciascuna chiave e il valore risultante. Come con gli *array*, un *dictionary* ha una proprietà *count* per determinare il numero di coppie chiave / valore e la

proprietà ***isEmpty*** restituisce *true* se non ci sono coppie chiave / valore nel dizionario e *false* in caso contrario. I *dictionary* in *Swift* posseggono due proprietà uniche: è possibile utilizzare la funzione "***keys"*** per ritornare la lista di tutte le chiavi presenti nel *dictionary*, mentre la parola chiave "***values"*** per ritornare la lista di tutti i valori.

```
//dictionary chiave : valore
var dictionary = ["key":2]
```

4.2.3 Cicli

Gli scenari che richiedono il completamento e la ripetizione di un'attività possono essere eseguiti nel codice utilizzando i ***Loops****, Swift* offre diversi modi per scorrere o ripetere blocchi di codice.

Ciclo For

Il ciclo *For* noto anche più specificamente come ciclo ***for-in*** è utile per ripetere qualcosa un determinato numero di volte. Un ciclo *for-in* esegue una serie di istruzioni per ogni elemento all'interno di un intervallo, sequenza o *collection*.

```
// ciclo for
for value in array{
    print(value)
}
```

Nel codice sopra, l'indice "*value*" è una costante disponibile per personalizzare il lavoro svolto all'interno delle parentesi graffe (il ciclo *for-in*).

Ciclo While

La particolarità del ciclo ***while*** è che continuerà a ripetersi finché la condizione specificata non sarà più verificata. *Swift* controlla la condizione prima che ogni ciclo venga eseguito, il che significa che è possibile saltare completamente il ciclo se la condizione non è mai soddisfatta. il corpo del ciclo *while* dovrebbe eseguire un determinato lavoro che porterà a non soddisfare più la condizione.

```
while array.count < 6 {

    array.insert("new value", at: array.count)
    print(array)
}
```

Break

Potrebbero verificarsi situazioni in cui si desidera interrompere l'esecuzione di un ciclo dall'interno del corpo del ciclo. Tramite l'utilizzo della parola chiave ***break*** *Swift* interromperà l'esecuzione del codice all'interno del ciclo e inizierà a eseguire qualsiasi codice definito immediatamente dopo.

4.3 Navigazione e WorkFlow

4.3.1 Tipi di dato opzionali

Uno dei maggiori punti di forza di *Swift* è la sua capacità di leggere il codice e comprendere rapidamente i dati. Quando una funzione può o meno restituire dati, *Swift* obbliga a gestire correttamente entrambe le condizioni. *Swift* utilizza una sintassi unica, chiamata ***optionals***, per rappresentare quando un valore può o meno contenere dati. Gli *optionals* forniscono un *wrapper* attorno a un valore che conterrà un'istanza del tipo previsto o niente (*nil*). Ogni tipo ha un tipo *optional* corrispondente, che si dichiara aggiungendo un " ***?*** " dopo il nome del tipo di dato, c'è anche la possibilità di aggiungere " ***!*** " dopo il nome per forzare l'***unwrapping*** del suo valore. La differenza principale è che utilizzando la prima tipologia vista l'*optional chaining* fallisce normalmente quando l'optional è *nil*, mentre effettuando un *unwrapping* forzato si innesca un errore di *runtime* quando l'optional è *nil[5]*.

4.3.2 Enumerazioni

Un'enumerazione, o ***enum***, è un tipo speciale di *Swift* che ti consente di rappresentare un insieme di opzioni. Per definire una nuova enumerazione viene utilizzata la parola chiave *enum*. Il codice seguente definisce un'enumerazione:

```
enum GiorniDellaSettimana {
    case Lunedi
    case Martedi
    case Mercoledi
    case Giovedi
    case Venerdi
    case Sabato
    case Domenica
}
```

Spesso le enumerazioni vengono utilizzate in uno *Switch* per controllare il flusso, come possiamo vedere dal codice seguente:

```
switch provaEnum {
case .Domenica :
        print("Domenica")
case .Lunedi :
        print("Lunedi")
default:
        print("error")
}
```

4.3.3 View Controller

Le classi del ***View Controller (UIViewController)*** sono responsabili della visualizzazione dei dati dell'utente e della gestione delle interazioni da parte dell'utente.

Ciclo di vita

In iOS, i *View Controller* possono trovarsi in diversi stati:

- *View not loaded*
- *View appearing*
- *View appeared*
- *View disappearing*
- *View disappeared*

Quando l'applicazione passa da uno stato all'altro, iOS chiama metodi definiti dall'*SDK* che possono essere implementati nel codice[5].

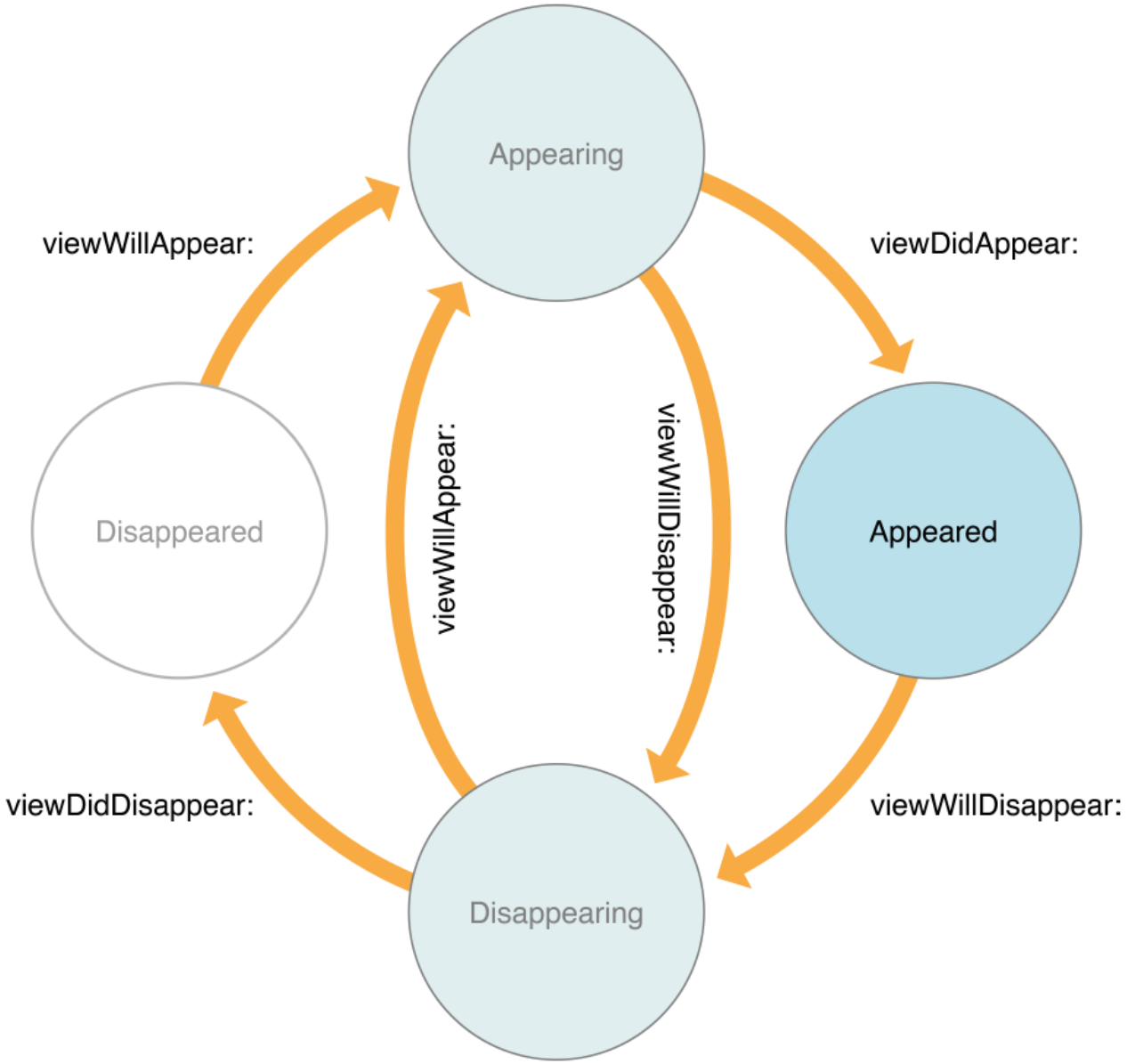

Figura 4.2: *ciclo di vita del View Controller*

Una volta caricata la visualizzazione, i metodi sono disponibili in coppia: "*will*" e "*did*". Questo modello di progettazione *Apple* standard consente di scrivere codice prima e dopo che si verifica l'evento denominato.

4.4 Persistenza dati

In informatica, il concetto di persistenza si riferisce alla caratteristica dei dati di un programma di sopravvivere all'esecuzione del programma stesso che li ha creati: senza questa capacità i

dati verrebbero salvati solo in memoria *Ram* venendo dunque persi con la terminazione del programma. Nei seguenti paragrafi verrà analizzato il concetto di persistenza dati in *Swift* e in che modo i dati vengono salvati.

4.4.1 Protocolli

Un ***protocollo*** è un insieme di regole formalmente descritte, definite al fine di favorire la comunicazione tra uno o più entità[6]. I computer comunicano tra loro utilizzando protocolli come *HTTP* (*Hyper Text Transfer Protocol*). *HTTP* è un protocollo di rete appartenente al livello di applicazione al modello ISO/OSI su cui è basato il Web. Esso utilizza il modello *client/server*, il *client* riceve e "mostra" oggetti Web, il *server* invia oggetti in risposta alle richieste[7].

Nella programmazione, un protocollo definisce le proprietà o i metodi che un oggetto deve avere per completare un'attività. La libreria standard di *Swift* definisce molti protocolli, per esempio: ***Equatable***, che consente di definire come le istanze dello stesso tipo sono uguali tra loro; ***Comparable***, che consente di definire come vengono ordinate le istanze dello stesso tipo; e ***Codable***, che consente di codificare facilmente le proprietà del tipo di dato come coppie chiave / valore che possono quindi essere salvate al lancio dell'app. [5]

Codable

Molte app salvano i dati dell'utente in modo che tali dati esistano ancora ogni volta che l'utente lancia l'app. Per salvare i dati, i valori che risiedono nella memoria devono essere codificati in una forma che può essere scritta in un file. Il protocollo ***Codable*** rende questo compito molto semplice creando coppie chiave / valore dai nomi e dai valori delle proprietà dell'oggetto che possono quindi essere utilizzati da un oggetto ***Encoder*** o ***Decoder***.

Delegation

Delegation è un ***Design Pattern***,che consente a una classe o una *struct* di trasferire, o delegare, alcune delle proprie responsabilità a un'istanza di un altro tipo. La cosa importante è che ci sono due parti: la persona che delega il compito e il delegato il cui scopo è svolgere effettivamente il compito. Questa relazione vale nella programmazione, potrà esserci un oggetto che necessita di un'attività, ma potrebbe non essere l'oggetto che implementa effettivamente il codice per realizzarlo. I tipi che delegano l'implementazione ad altri tipi in genere lo fanno definendo un protocollo. Il protocollo definisce le responsabilità che possono essere delegate e il delegato adotta il protocollo per eseguire l'attività effettiva. Il pattern *delegate* è ampiamente utilizzato in *UIKit* e in altri framework iOS.[5]

4.4.2 Ciclo di vita dell'applicazione

La maggior parte delle app iOS viene eseguita quando l'utente le apre e si interrompe quando quest'ultimo preme il pulsante Home, o passa ad un'app diversa. Esistono metodi *delegate* in cui è possibile eseguire codice in uno di questi eventi nel ciclo di vita dell'app. Nello specifico verrà esaminato il file *AppDelegate.swift* che *Xcode* crea ad ogni nuovo progetto.

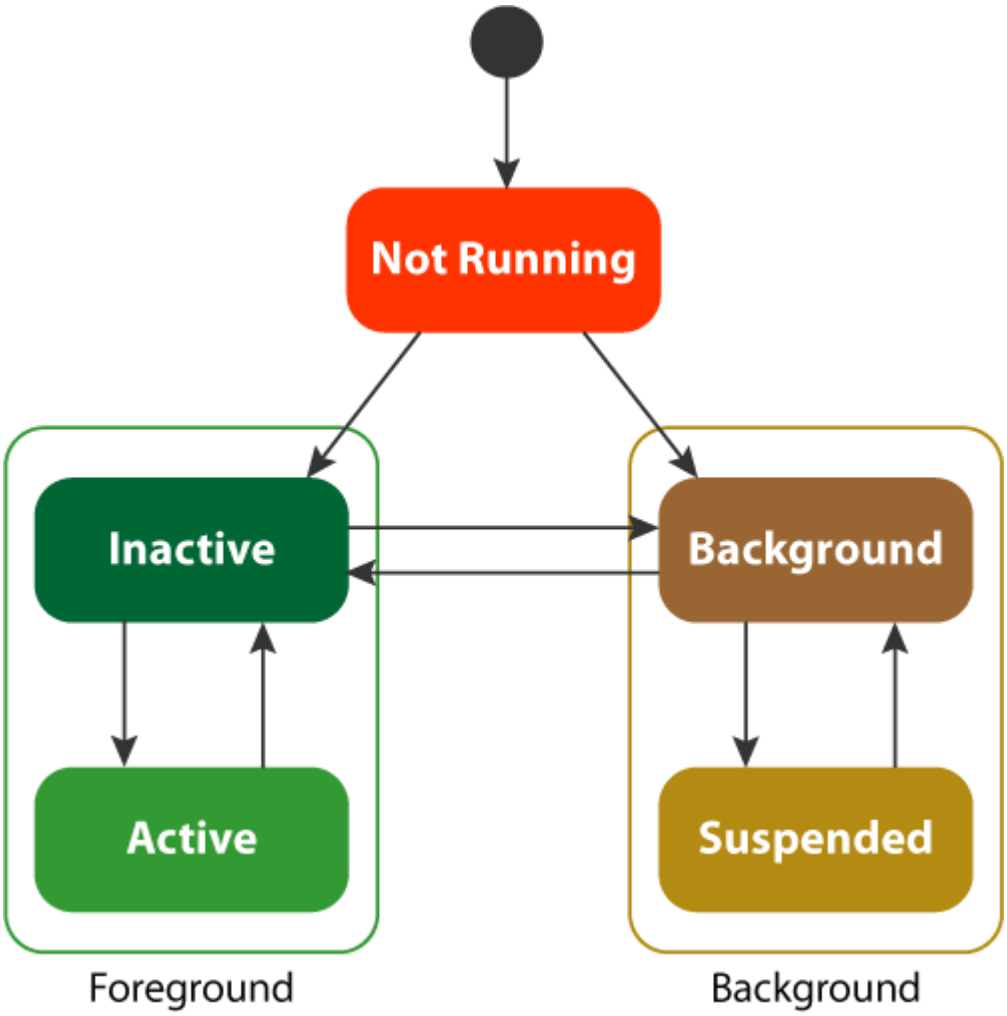

Figura 4.3: *ciclo di vita dell'applicazione*

Tabella 4.1: *Descrizione delle fasi del ciclo di vita dell'applicazione*

Stato dell'applicazione	Descrizione
Not Running	L'app non è stata lanciata o è terminata
Inactive	L'app è attiva ma non riceve azioni da parte dell'utente. Solitamente questo è uno stato transitorio
Active	L'app è attiva normalmente e riceve azioni da parte dell'utente
Background	Quando l'utente esce dall'app il sistema sposta lo stato in background per poi sospenderla definitivamente, Tuttavia il sistema può risvegliare l'app in background per eseguire specifici tasks, per esempio uno dei più comuni, può essere quello di ricevere notifiche "push".

I sei metodi principali sono:

- **DidFinishLaunching:** consiste nella prima funzione chiamata una volta terminato il lancio dell'applicazione
- **WillResignActive:** quando l'utente chiude l'app o quando la normale esecuzione viene interrotta temporaneamente
- **DidEnterBackground:** quando l'app entra nello stato di *background*, viene chiamato immediatamente dopo il precedente metodo
- **WillEnterForeground:** è chiamato nel momento di transizione che intercorre dallo stato di background allo stato attivo
- **DidBecomeActive:** viene chiamato nel momento in cui avviene il passaggio dallo stato inattivo a quello attivo dell'app
- **WillTerminate:** è l'ultimo metodo chiamato, comunica all'app che è terminata e ne ripulisce la memoria

4.4.3 Model View Controller

Uno dei modelli più comuni per la progettazione software è il ***Model-View-Controller*** o *MVC*. *MVC* assegna agli oggetti uno dei tre ruoli (*medel, view o controller*) e aiuta a definire il modo in cui i diversi oggetti comunicano tra loro.

Il diagramma seguente fornisce una panoramica del funzionamento di ciascuno di questi oggetti in relazione l'uno con l'altro. Ogni livello, o tipo, ha ruoli e linee guida specifici per la comunicazione con gli altri livelli.

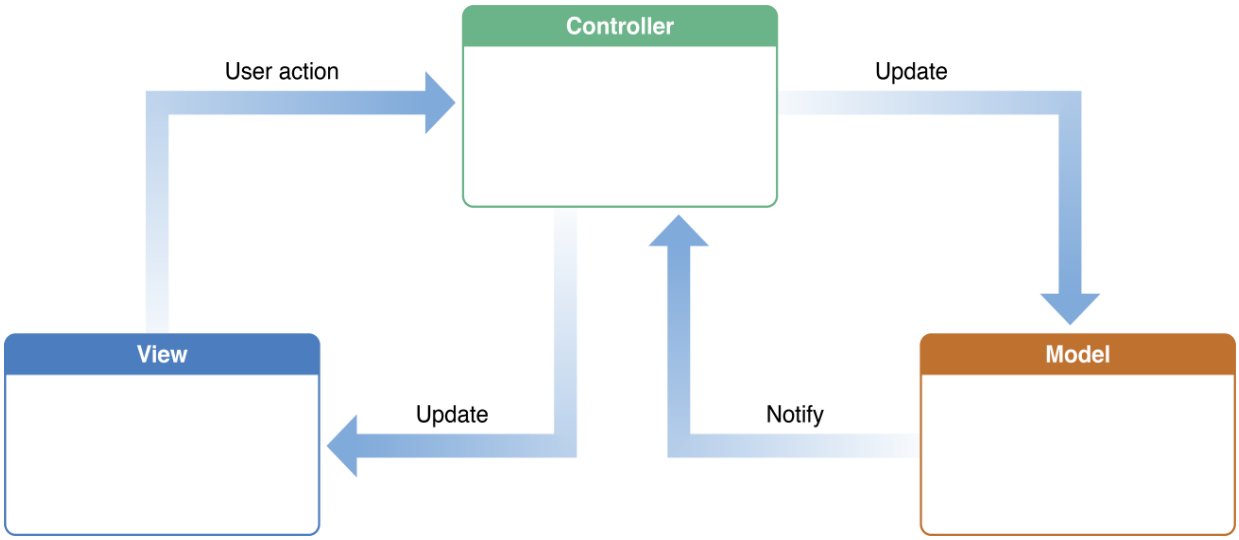

Figura 4.4: *Pattern Model-View-Controller*

Model

Un oggetto *model* raggruppa i dati necessari per uno specifico tipo di soluzione che si sta cercando di creare.

```
class Studente: Object, Codable{

    var nome: String!
    var eta: Int!
    var scuola: Scuola!


    internal init(nome: String, eta: Int, scuola: Scuola) {
        self.nome = nome
        self.eta = eta
        self.scuola = scuola
    }
```

Nell'esempio sopra riportato, un oggetto model rappresenta uno studente, il *model* **Studente** è collegato ad un altro *model* di tipo **Scuola**. Gli oggetti del *model* sono costituiti da proprietà che rappresentano attributi del tipo e talvolta dispongono di metodi per aggiornare e modificare le proprie proprietà.

Views
Uno degli scopi principali degli oggetti di *view* è visualizzare i dati sugli oggetti del *model* di un'applicazione e consentire all'utente di modificare tali dati. Le *view* possono essere riutilizzate o riconfigurate per mostrare diverse istanze dei dati del *model*.

Controllers
Un oggetto *Controller* funge da nesso tra le *view* e gli oggetti del *model*. Ad esempio, quando l'utente interagisce con la *view*, la *view* invia un messaggio al *controller* della *view* e il *controller* della *view* può quindi aggiornare l'oggetto del *model*. In alternativa, quando un oggetto del *model* viene creato o aggiornato, un *controller* della *view* può dire alla sua *view* di ridisegnarsi o aggiornarsi con i nuovi dati. Il *View Controller* è il tipo di *controller* più comune per i nuovi sviluppatori iOS. Come visto nei paragrafi precedenti, un *controller* della *view* controlla una *view* insieme alle sue *sottoview* e di solito mostra le informazioni su uno o più oggetti del *model*. Quando l'utente interagisce con una *view* o un *controller*, viene attivata un'azione o un blocco di codice sul *controller* della *view*, che può quindi aggiornare l'oggetto del *model*.

4.4.4 Libreria Realm

La piattaforma ***Realm*** è una combinazione di componenti *server* e *client* basati su *noSQL* collegati tramite un protocollo di sincronizzazione veloce ed efficiente per abilitare app e servizi connessi in tempo reale che siano reattivi e dalle prestazioni indipendentemente dallo stato della rete. La piattaforma *Realm* ha due componenti principali: il ***database Realm*** e il ***Realm Object Server***. Questi due componenti lavorano insieme per sincronizzare automaticamente i dati consentendo un gran numero di casi d'uso che vanno dalle prime app *offline*, app di assistenza sul campo e di raccolta dati, servizi mobili in cui la disponibilità dei dati e la reattività dell'utente sono fondamentali. È disponibile per i principali linguaggi mobili, come *Swift* e *Objective-C* (*iOS*), *Java* (*Android*), *C #* (*Xamarin, .NET*) e *JavaScript* (*React Native e Node.js*). [8]

***Figura 4.5**: Logo Realm*

Realm Data Models
I ***Realm data models*** sono definiti come classi *Swift*. Gli oggetti del model *Realm* funzionano principalmente come qualsiasi altro oggetto *Swift*. È possibile collegare insieme due oggetti

Realm qualsiasi, i collegamenti non sono costosi in termini di velocità o memoria, i tipi di relazioni che *Realm* consente sono: ***Many-to-one*** e ***Many-to-many***.[8]

Queries

Le *query* restituiscono un'istanza di risultati, che contiene una raccolta di *Object*. I risultati sono molto simili ad *Array*. I risultati di una *query* non sono copie dei dati: la modifica dei risultati di una *query* modificherà direttamente i dati in memoria. Una volta che la *query* è stata eseguita, i risultati vengono aggiornati con le modifiche apportate.

```
class func get() -> Studente? {
    let realm = try! Realm()
    let studente = realm.objects(Studente.self).first
    return studente
}
```

Nell'esempio sopra riportato possiamo notare una semplice *query* che restituisce come valore di ritorno un oggetto di tipo Studente (*model* precedentemente creato).

4.5 API

Con il termine ***API* (*application programming interface)*** si intende insieme di codice di programmazione che consente la trasmissione di dati tra un prodotto software e un altro. La finalità delle *API* è ottenere un'astrazione a più alto livello, di solito tra l'*hardware* e il programmatore o tra *software* a basso e quello ad alto livello semplificando così il lavoro di programmazione. Le API permettono infatti di evitare ai programmatori di riscrivere ogni volta tutte le funzioni necessarie al programma dal nulla, ovvero dal basso livello, rientrando quindi nel più vasto concetto di riuso di codice. Le *API* stesse rappresentano quindi un livello di astrazione intermedio: il *software* che fornisce una certa API è detto **implementazione dell'API**.

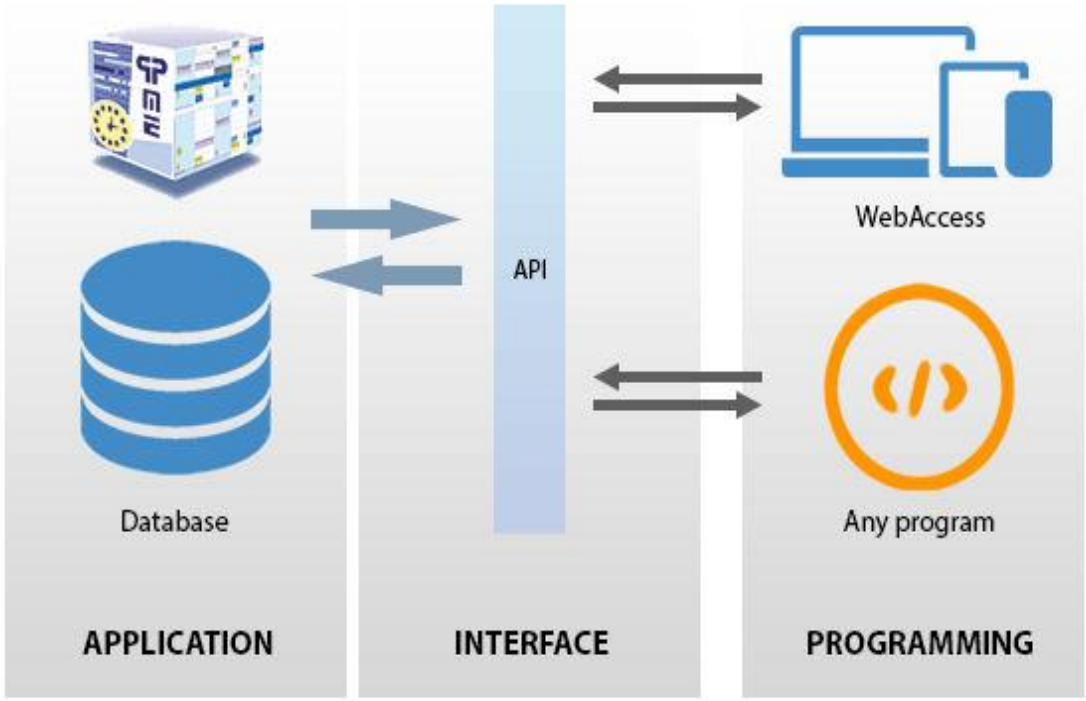

Figura 4.6: API

4.5.1 Richieste HTTP

I metodi ***GET*** e ***POST*** sono i due metodi *HTTP* più comuni. *GET* viene utilizzato per richiedere informazioni da un server e *POST* viene utilizzato per inviare informazioni al *server*. Le richieste di rete possono avere un ***header*** e un ***body***.

- **Header**: funzionano allo stesso modo di un *dictionary*: con chiavi e valori che indicano al server come gestire la richiesta.
- **Body**: include i dati che sono stati inviati o ricevuti. Quando si riceve una risposta a una richiesta *GET*, anche questa avrà un corpo. Ad esempio, quando si richiede informazioni per un sito *Web*, le risorse che compongono il sito (HTML, CSS e immagini) sono tutte incluse come *body* della richiesta di rete.

4.5.2 JSON e Decodifica

JSON (JavaScript Object Notation) è un formato adatto all'interscambio di dati fra applicazioni *client/server*. È basato sul linguaggio *JavaScript* Standard, ma ne è indipendente. Il JSON è tipo di formato dati utilizzato per restituire informazioni da parte del *server*

```
{
   "nome":"Lorenzo",
   "eta":"22",
   "scuola":[
              "Informatica"
   ]
}
```

Le parentesi graffe {} racchiudono un dizionario di chiavi e valori, le parentesi quadre [] incapsulano un *array*, Il testo tra virgolette "" sono valori stringa, gli interi e booleani vengono scritti senza virgolette.

Decoding

Le istanze di ***JSONDecoder*** hanno un metodo di decodifica ***(_: from :)*** che accetta un parametro di tipo *Type* e un parametro di tipo *Data*. Il metodo tenta quindi di decodificare l'oggetto passato al metodo nel tipo desiderato. Affinché questo metodo funzioni, il tipo passato al metodo deve essere conforme al protocollo *Codable* e i dati devono essere in un formato tale da poter essere decodificati in questo tipo *Codable*. Se il tuo tipo non è conforme a *Codable*, il codice produrrà un errore.

Il metodo *decode* (_: from :) di *JSONDecoder* è considerato una funzione di *throwing*, perciò associando la sintassi ***do-try-catch***, è possibile catturare e risolvere ogni errore che ne consegue. Come visto nei paragrafi precedenti il protocollo *Codable* definisce un insieme di regole che possono essere applicate a una classe o struttura che consentono di convertire facilmente i dati da un tipo a un altro. In questo caso, la classe *JSONDecoder* può essere utilizzata per tradurre le informazioni da *JSON* negli oggetti del model personalizzato. Per impostazione predefinita, *coder* e *decoder* funzionano con i tipi *codable* utilizzando due metodi generati automaticamente.[5]

Il primo è ***init (with :)***, che viene utilizzato per inizializzare il nuovo tipo di dati dai vecchi dati. Il secondo è ***decode (to :)***, che viene utilizzato per convertire il tipo corrente in un nuovo tipo di dati. Per impostazione predefinita, il protocollo *Codable* abbina i nomi e i valori delle proprietà alle chiavi e ai valori del tipo codificato.[5] Di seguito è riportato un semplice esempio di decodifica JSON:

```
func JSONDecoder(){
    let json = """
    {
        "nome": "Lorenzo",
        "eta": "22",
        "scuola": ["Informatica"]
    }
    """.data(using: .utf8)!
        let decoder = JSONDecoder()
        let studente = try decoder.decode(Studente.self, from: json)

    func print(studente.nome) //  print: Lorenzo
    }
}
```

4.5.3 Libreria Alamofire

Alamofire è una libreria di rete *HTTP* scritta in *Swift*. Si basa sul sistema di caricamento degli URL di *Apple* fornito dal *framework Foundation*. Al centro del sistema ci sono *URLSession* e le *URLSessionTask*. *Alamofire* racchiude queste *API*, e molte altre, in un'interfaccia più facile da usare e fornisce una varietà di funzionalità necessarie per lo sviluppo di applicazioni moderne utilizzando i metodi *http*.[9]

***Figura 4.7**: Logo Alamofire*

Per gli esempi seguenti verrà utilizzato **httpbin.org** per simulare le chiamate *http*.

Get request

Di seguito è riportato un esempio di una semplice richiesta GET:

```
//simple GET REQUEST
AF.request("https://httpbin.org/get").response{ response in
    debugPrint("Response: \(response)")
}
```

Autenticazione

Esistono diversi modi per gestire l'autenticazione utilizzando Alamofire, nel seguente esempio viene riportato un modo molto semplice per autenticarsi utilizzando ***.authenticate*** seguito da *username* e password.

```
let user = "test@email.com"
let password = "testpassword"

AF.request("https://httpbin.org/basic-auth/\(user)/\(password)").authenticate(username:
user, password: password).responseJSON{  response in
      debugPrint(response)
}
```

Nella figura 4.8 viene riportata la risposta che viene stampata dal *server* nella console di *debug*, come possiamo notare l'autenticazione è andata a buon fine, infatti otteniamo come status code: **200**

```
[Request]: GET https://httpbin.org/basic-auth/test@email.com/testpassword
[Request Body]:
None
[Response]:
[Status Code]: 200
[Headers]:
Access-Control-Allow-Origin: *
Content-Length: 57
Content-Type: application/json
Date: Fri, 11 Sep 2020 16:16:18 GMT
Server: gunicorn/19.9.0
access-control-allow-credentials: true
[Response Body]:
{
  "authenticated": true,
  "user": "test@email.com"
}

[Data]: 57 bytes
[Network Duration]: 0.9585509300231934s
[Serialization Duration]: 0.00011243799963267 52s
[Result]: success({
    authenticated = 1;
    user = "test@email.com";
})
All Output ⌃                                                    Filter
```

Figura 4.8: *Risposta del server nella console di debug*

Download locale

Alamofire supporta diversi modi di gestire i dati, per set di dati più grandi sono supportati anche: ***Session.download***, ***DownloadRequest*** e ***DownloadResponse***. Di seguito viene mostrato un semplice esempio di *download:*

```
let destination: DownloadRequest.Destination = { _, _ in
        let documentsURL = FileManager.default.urls(for: .picturesDirectory, in:
        .userDomainMask)[0]
        let fileURL = documentsURL.appendingPathComponent("image.png")

        return (fileURL, [.removePreviousFile, .createIntermediateDirectories])
}

AF.download("https://httpbin.org/image/png", to: destination) .downloadProgress {
progress in
        print("Download Progress: \(progress.fractionCompleted)")
}.response { response in
        debugPrint(response)

        if response.error == nil, let imagePath = response.fileURL?.path {
        let image = UIImage(contentsOfFile: imagePath)
        }
}
```

Grazie al comando "***.downloadProgress***", è possibile monitorare lo stato del nostro *download.*

CAPITOLO 5

L'applicazione

Questo progetto presenta molte delle conoscenze trattate nel corso di questo documento di tesi, infatti prevede di sviluppare un'applicazione mobile nativa in linguaggio Swift, più precisamente, come anticipato nei capitoli precedenti, ho realizzato alcuni moduli dell'applicazione *"ARTILS"* (*Augmented Real Time Learning for Secure workspace*) ovvero una piattaforma per il supporto alla formazione per la sicurezza negli ambienti di lavoro**.** L'applicazione è fruibile da qualsiasi dispositivo mobile e tablet, può essere utilizzata da chiunque abbia bisogno di svolgere attività didattiche a distanza orientate alla sicurezza sul lavoro; inoltre può risultare molto utile in questo periodo in quanto le attività a distanza stanno diventando sempre più diffuse. Nei paragrafi successivi verrà mostrato il funzionamento dell'applicazione tramite delle foto ricavate dal simulatore; successivamente verranno analizzate nel dettaglio alcune delle funzionalità da me implementate fornendo delle righe di codice.

5.1 Tecnologie utilizzate

Il progetto è stato sviluppato all'interno dell'ambiente *Xcode, p*er la scrittura del codice è stata utilizzata la versione 4.2 di *Swift.* Durante la scrittura del codice si sono eseguiti test sul simulatore dell'iPhone 11 Pro Max e precedenti. Nel progetto sono stati utilizzati principalmente frameworks nativi quali *Foundation* e *UIKit* mentre per la gestione delle *API* e del database ci si è affidati a delle librerie esterne quali *Alamofire* e *RealmSwift* mentre per gestire i file zip è stata utilizzata la libreria *SSUnzipeArchive*; importando i pacchetti all'interno del progetto tramite *CocoaPods,* all'interno del *podfile* così da permetterne l'utilizzo all'interno di *Xcode.*

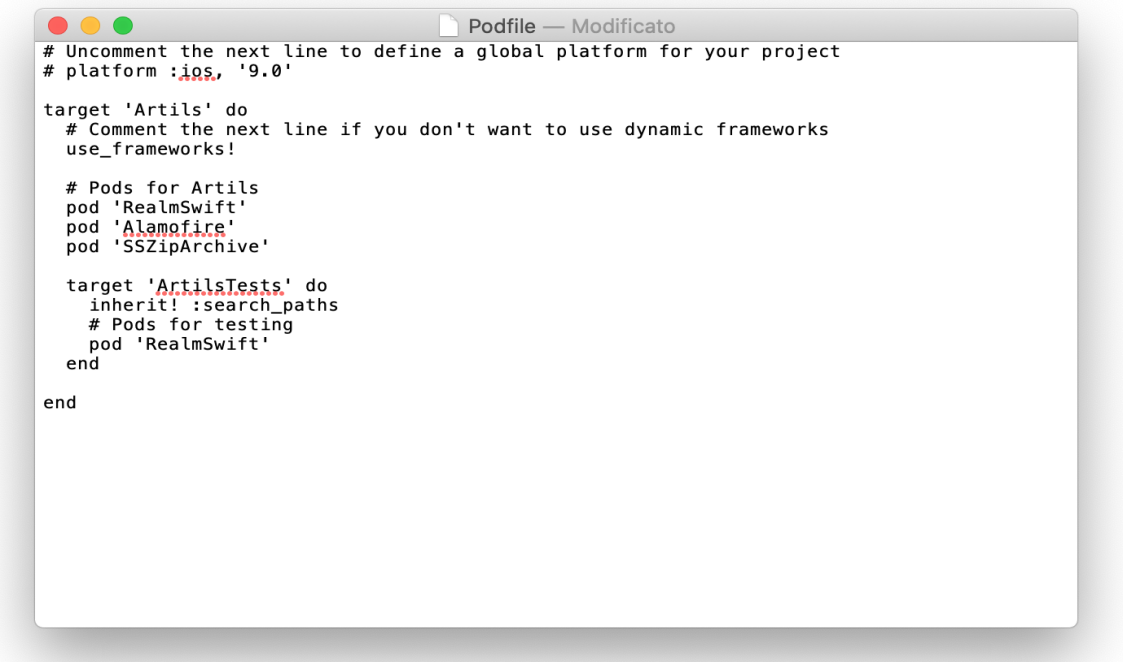

Figura 5.1: *Podfile del progetto*

Nella figura 5.1 è mostrato il contenuto del *Podfile* utilizzato per richiedere l'installazione del *pod* delle tre librerie. Una volta salvato il file, si esegue da terminale, dopo essersi posizionati nella *directory* del progetto contenente il *Podfile*, il commando:

```
$ pod install
```

Installato il *pod* desiderato, con le rispettive dipendenze, viene creato all'interno della *directory* un nuovo file per lanciare il progetto *Xcode* con estensione *.xcworkspace* invece della classica *.xcodeproj*. D'ora in avanti si dovrà utilizzare l'ultimo creato per lanciare il progetto.

5.2 Presentazione del progetto

Durante lo stage la tutor aziendale ha optato per inserirmi nello sviluppo di questo progetto. All'inizio ho realizzato principalmente l'interfaccia utente, in quanto non ero ancora pronto per addentrarmi in funzionalità più complesse, progressivamente ho acquisito familiarità con l'ambiente di lavoro così da poter iniziare con l'implementazione di funzionalità di più difficile realizzazione.

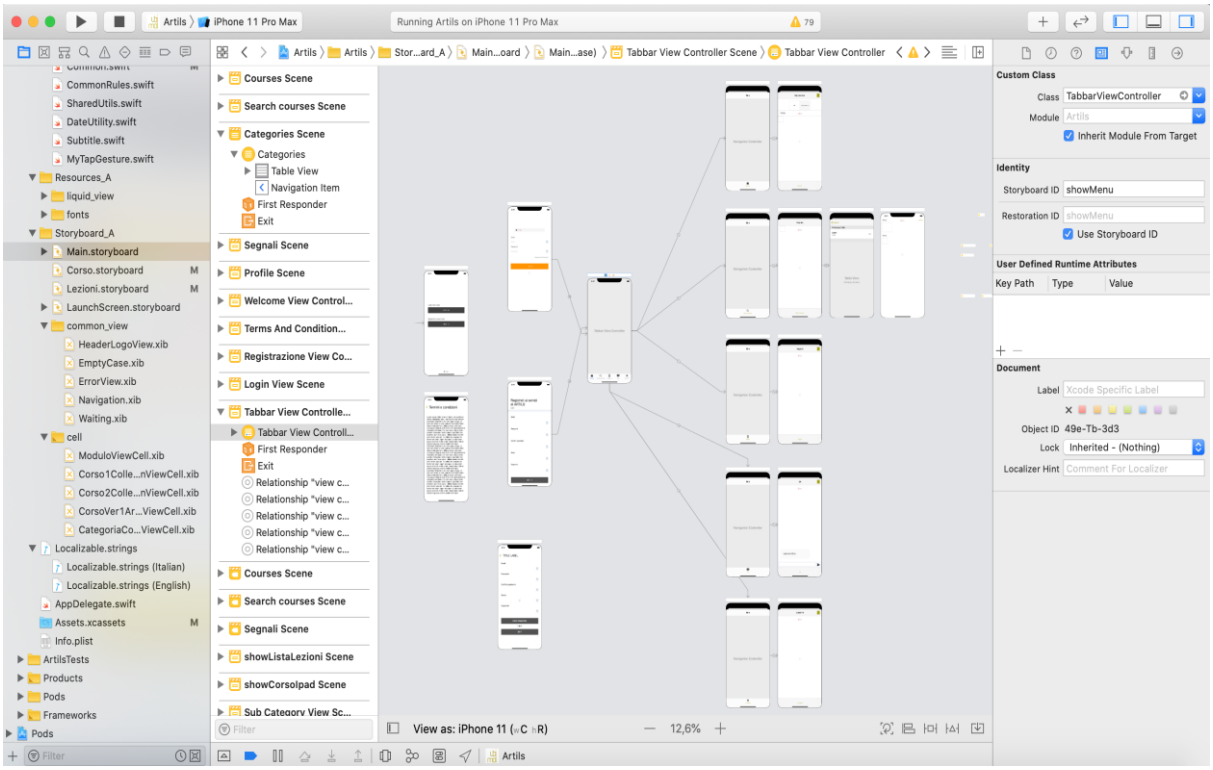

Figura 5.2: *Scheletro dell'interfaccia grafica del progetto*

La figura 5.2 rappresenta lo scheletro di quella che è l'interfaccia grafica dell'applicazione partendo dalle pagine iniziali dell'app (registrazione e login) collegate ad un *UITabBarController* ovvero un *container view controller* che gestisce un'interfaccia di selezione multipla, in cui in base alla selezione viene determinato quale *view controller* mostrare.[10] Dall'immagine notiamo che alla TabBarController sono collegate le cinque sezioni dell'applicazione (due delle quali non verranno mostrate perché ancora in fase di sviluppo).

Di seguito è mostrata una sequenza di immagini ricavate dal simulatore di un iPhone 11 Pro Max, partendo dall'avvio dell'applicazione per poi proseguire mostrandone il vero e proprio funzionamento.

Primo avvio dell'applicazione

Figura 5.3: Pagina di login e registrazione

Durante il primo avvio l'utente sarà indirizzato verso la pagina di accesso, infatti per poter utilizzare l'app sarà necessario effettuare la registrazione nell'apposita pagina, invece nella pagina di login sarà possibile, inserendo le credenziali, accedere all'interno dell'applicazione.

Corsi

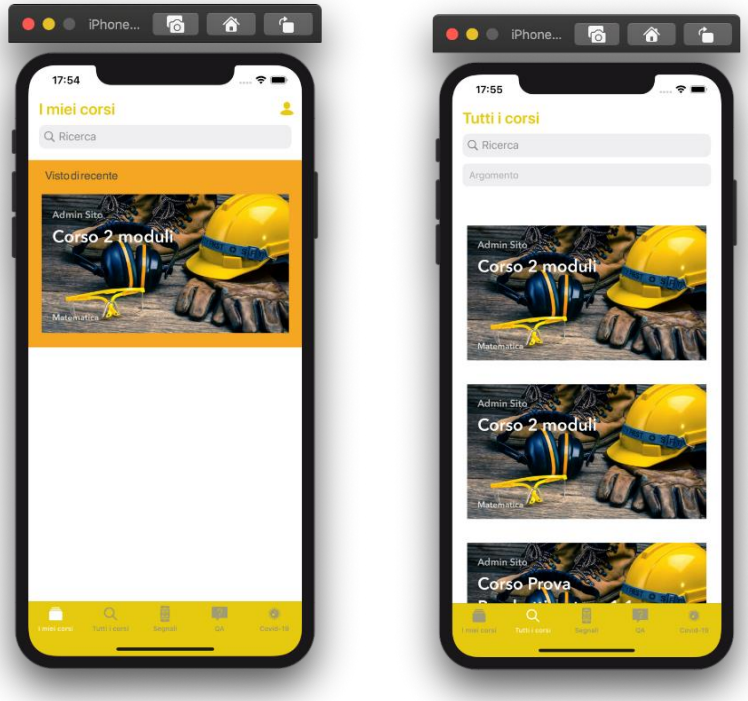

Figura 5.4: *Pagina dei corsi*

Una volta effettuato il login l'utente sarà reindirizzato all'interno dell'applicazione, in particolare nella pagina dei corsi, la figura 5.4 a sinistra mostra i corsi recenti ovvero gli ultimi corsi in cui si è fatto l'accesso, mentre nell'immagine di destra vengono visualizzati tutti i corsi a cui l'utente può accedere.

Una volta scelto un corso si potrà accedere alla relativa pagina di dettaglio:

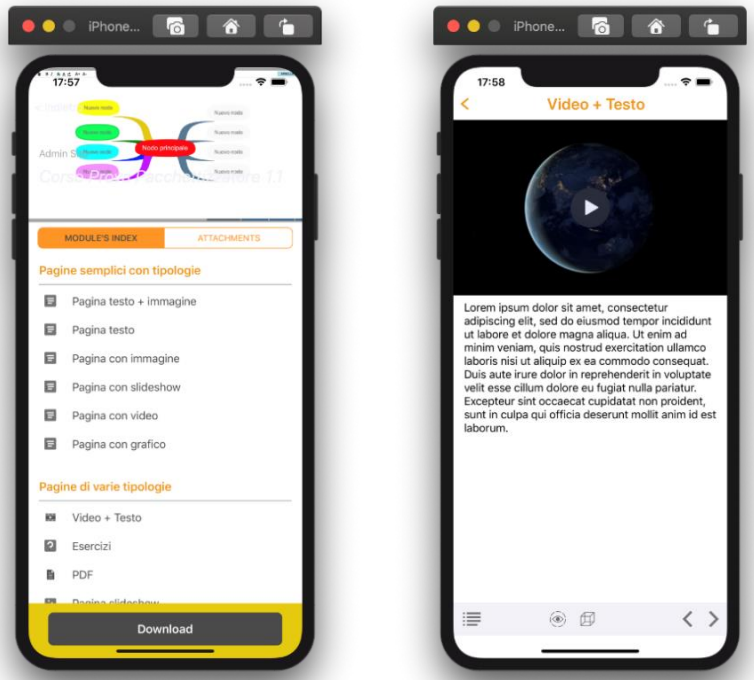

Figura 5.5: *Pagina dettaglio del corso*

Ogni corso è composto da una serie di lezioni, vedi figura 5.5, ci sono diversi tipi di lezione ed in base alla tipologia scelta il contenuto della lezione carica una grafica differente, ad esempio nella figura 5.5 nell'immagine di destra viene mostrata la tipologia "video_text" per cui la grafica carica un video in streaming da un *url* online ed un testo formattato html. Inoltre nella pagina del corso è possibile effettuare una serie di esercizi per testare le conoscenze acquisite:

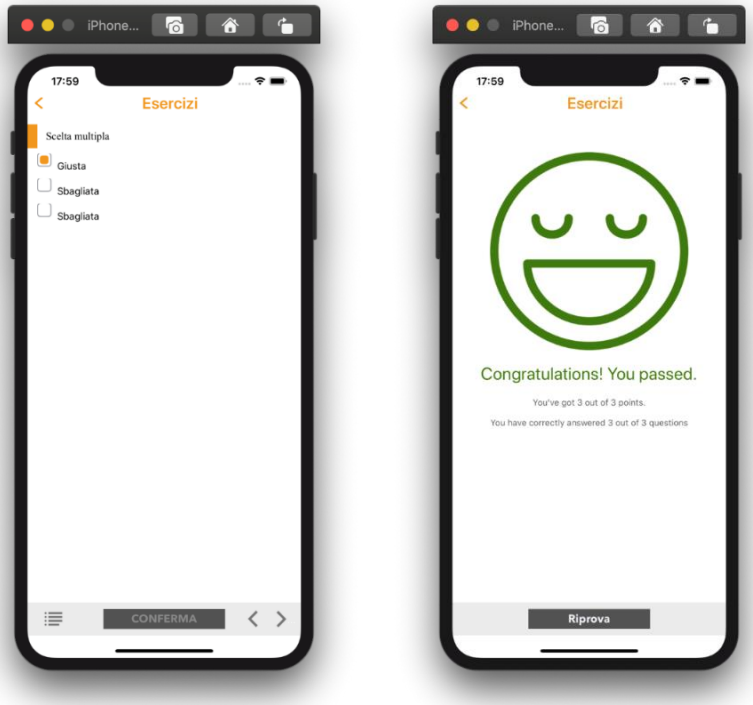

Figura5.6: *Pagina esercizi del corso*

5.3 Interfaccia utente: corsi

Xcode, come già anticipato nei precedenti capitoli, mette a disposizione uno strumento molto utile per la creazione delle scene relative al pattern architetturale *Model-View-Controller*, si tratta dello *storyboard*. Tramite questo *tool* si vuole fornire un modo grafico e il più semplice possibile per disegnare le interfacce delle applicazioni, per navigare e trasferire dati ed informazioni tra i *view controller* che la compongono.

Il primo passo per costruire l'interfaccia grafica di una pagina, solitamente, è quello di creare un nuovo file *storyboard* ed al suo interno un *view controller* che permette di gestire l'interazione degli utenti con quella pagina che si coordina con gli altri oggetti dell'applicazione. In base alle esigenze, al suo interno saranno inseriti tutti gli elementi che andranno a comporre la pagina finale. Infine il *view controller* verrà collegato con un file Swift che dovrà necessariamente estenderà la classe *UIViewController*.

Nei seguenti paragrafi osserveremo la struttura dell'interfaccia utente per la pagina "i miei corsi", osservando dettagliatamente come essa è stata pensata e costruita mediante i mezzi messi a disposizione da *Xcode* e *Swift.*

5.3.1 .Storyboard

I file analizzati sono i seguenti: *iMieiCorsiView.Swift* e *Corso.Storyboard*. Nello *storyboard* in figura 5.7 è presente un *navigation controller,* un oggetto molto comune nelle applicazioni mobile, permette di gestire la parte di navigazione tra i *view controller,* in particolare permette di mostrare un singolo *view controller* alla volta nascondendo il *view controller* precedente e mostrando il successivo, gestendo così la gerarchia mediante l'utilizzo di un *array* ordinato noto come *navigation stack* al cui interno in fondo allo *stack* è presente il *view controller* principale, mentre l'elemento più in alto rappresenta il *view controller* attualmente visualizzato. Sostanzialmente ci sono due modalità per navigare all'interno dei *view controller*: *pushViewController e presentViewController.* Il primo permette di accodare il *view controller* nello *stack* così da lasciare ad iOS la gestione del *back button* che permetterà automaticamente di tornare al view controller precedente; mentre con il secondo metodo viene mostrato il nuovo *view controller* azzerando la gerarchia della *navigation stack.*

Nel *view controller* in figura 5.7 sono presenti gli elementi che andranno a comporre l'interfaccia con cui l'utente interagirà. In particolare, come elemento principale che andrà a comporre la pagina, è stata utilizzata una *collection view.*

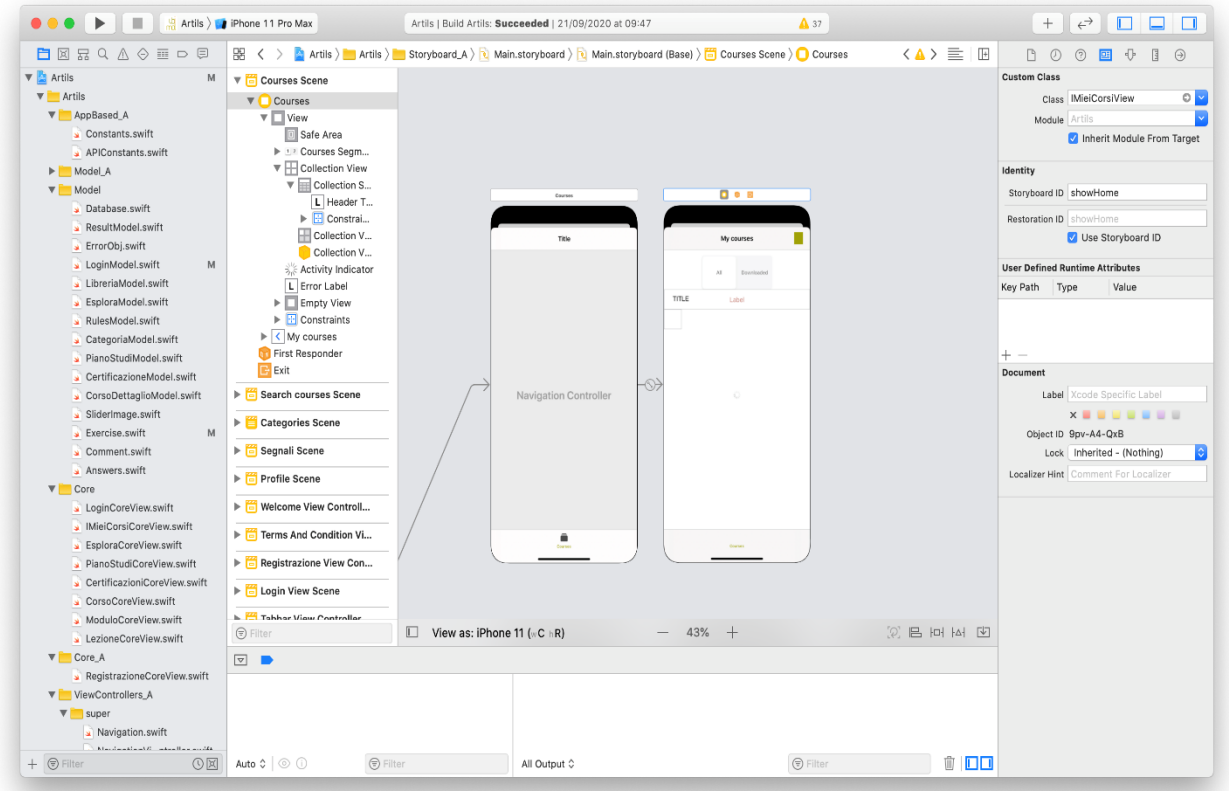

Figura5.7: *Storyboard del corso*

Una *collection view*, è un sistema per rappresentare degli elementi sotto forma di una "griglia", composta da righe e colonne. Possiamo dire che la *collection view* è simile alla *table view*, un componente grafico che utilizza delle celle che vengono impilate tutte una sotto l'altra, la *collection view,* invece, utilizza una cella, che viene collocata in riga e colonna in base allo spazio messo a disposizione dal display.
In questo caso è stato scelto l'utilizzo della *collection view* piuttosto che della *table view* per permettere a dispositivi più grandi quali per esempio iPad di visualizzare il contenuto su più righe così da ottimizzare l'applicazione per ogni tipologia di dispositivo. Per semplicità verrà analizzato solamente il caso in cui il dispositivo utilizzato sia un iPhone. Di seguito vengono mostrate diverse righe di codice per osservare come viene costruita una *collection view.*

```
    func numberOfSections(in collectionView: UICollectionView) -> Int{
    return 1  // numero di sezioni
    }

    func collectionView(_ collectionView: UICollectionView, numberOfItemsInSection
section: Int) -> Int {
    var returnValue = 0

        if recentArray.isEmpty {
            returnValue = libraryArray.count
        }
        else {
            returnValue = section==0 ? recentArray.count : libraryArray.count
        }
    return returnValue // numero di elementi per sezione
}
```

Figura 5.8

La *collection view* viene costruita assegnando il numero di sezioni e il numero di elementi all'interno della sezione (figura 5.8), utilizzando come riferimento da cui prendere i dati un *array* in cui all'interno sono presenti i corsi.

```
/*il metodo principale che permette di riempire la collection view utilizzando un file
.xib */
    func collectionView(_ collectionView: UICollectionView, cellForItemAt indexPath:
IndexPath) -> UICollectionViewCell {
        let cell = collectionView.dequeueReusableCell(withReuseIdentifier:
"corsoVer1ArtilsViewCell", for: indexPath) as! CorsoVer1ArtilsViewCell
    let indexSection = indexPath.section
    let indexRow = indexPath.row

    var course = Libreria()

        if recentArray.isEmpty {
            course = libraryArray[indexRow]
        }
        else {
            course = indexSection==0 ? recentArray[indexRow] : libraryArray[indexRow]
            cell.backgroundColor = indexSection==0 ? orangeColor : UIColor.white
            }

    cell.authorLabel.text = course.teacher
    cell.titleOfTheCourseLabel.text = course.title
        cell.argumentOfTheCourseLabel.text =
Array(course.categoriesData.map({$0.level == 1 ? $0.title :
nil})).compactMap({$0}).first
    cell.setBackgroundImage(with: course)

    return cell
}
```

Figura 5.9: *Metodo per il riempimento della collection view*

All'interno del metodo (figura 5.9), il passaggio fondamentale è il seguente:

```
        let cell = collectionView.dequeueReusableCell(withReuseIdentifier:
"corsoVer1ArtilsViewCell", for: indexPath) as! CorsoVer1ArtilsViewCell
```

in cui viene dichiarata la variabile *cell* che assumerà tutte le proprietà della classe *CorsoVer1ArtilsViewCell* ovvero il file che verrà analizzato nel successivo paragrafo, infatti utilizzando la variabile *cell* è possibile accedere ad una serie di campi dell'interfaccia grafica presente nel file *xib*, rispettivamente: autore del corso, titolo del corso , argomento ed immagine di background. Ad essi viene assegnato il corretto valore partendo dal *model* di riferimento che in questo caso è denominato: Libreria().

```
/*Il seguente metodo permette di gestire l'interazione dell'utente nel momento in cui
seleziona un elemento all'interno della collection view */
   func collectionView(_ collectionView: UICollectionView, didSelectItemAt indexPath:
IndexPath) {
      let indexSection = indexPath.section
      let indexRow = indexPath.row
      var course = Libreria()
         if recentArray.isEmpty {
            course = libraryArray[indexRow]
         }
         else {
            course = indexSection==0 ? recentArray[indexRow] : libraryArray[indexRow]
         }

      CorsoCoreView.askCorsoDettaglio(idCorso: course.uuid, completion: { result in
         self.activityIndicator.isHidden = true
         self.result = result.elem
         let modulo = course.modulesNumber
         if result.success {
               self.gotoLezioneView(with: result.elem as! Corso)
               CorsoCoreView.setCorsoAsRecente(idCorso: course.uuid)
         }
         else {
            switch result.statusCode {
               case 400:
                  self.showMessage(errorLabel: self.errorLabel, message:
Constants.Constant.COMMON_ERRORE_400, error: true, numLines: 0, isAutoShrink:
false)
               case 401:
                  self.showLoginRequest()
               default:
                  self.showMessage(errorLabel: self.errorLabel, message:
Constants.Constant.COMMON_ERRORE_CONNESSIONE, error: true, numLines: 0,
isAutoShrink: false)
            }
         }
      })
```

Figura 5.10: *funzione per la gestione dell'interazione dell'utente con uno degli oggetti presenti nella* collection view

Infine viene gestita l'interazione dell'utente con uno degli oggetti presenti nella *collection view(figura 5.10),* in particolar modo è possibile notare che viene effettuata una chiamata API

in GET per permettere al sistema di mostrare la pagina del corso corretta in base alla selezione effettuata dall'utente. Se la chiamata va a buon fine allora tramite il comando:

```
self.gotoLezioneView(with: result.elem as! Corso)
```

l'utente viene reindirizzato nella pagina del dettaglio del corso. Una volta compresa la costruzione di una *collection view* è ora possibile analizzare dettagliatamente quali elementi grafici si trovano al suo interno.

5.3.2 .Xib

Lo *Xib* (XML Interface Builder) come lo *storyboard* è un tipo di *interface builder* ma la cosa che rende unico lo *xib* è la sua versatilità, infatti la sua peculiarità sta nel poter essere riutilizzato in più di un *view controller*. Ciò toglie una mole di lavoro non indifferente al programmatore, infatti riallacciando l'esempio del paragrafo precedente all'interno della *collection view* viene utilizzato il file *xib* reiterandolo in base alle esigenze, inoltre è possibile riutilizzare lo stesso file *xib* per altre *collection view* o *table view* presenti in altre pagine, infatti nell'applicazione lo stesso file *xib* viene riutilizzato all'interno di diverse pagine permettendo così un considerevole risparmio di tempo.

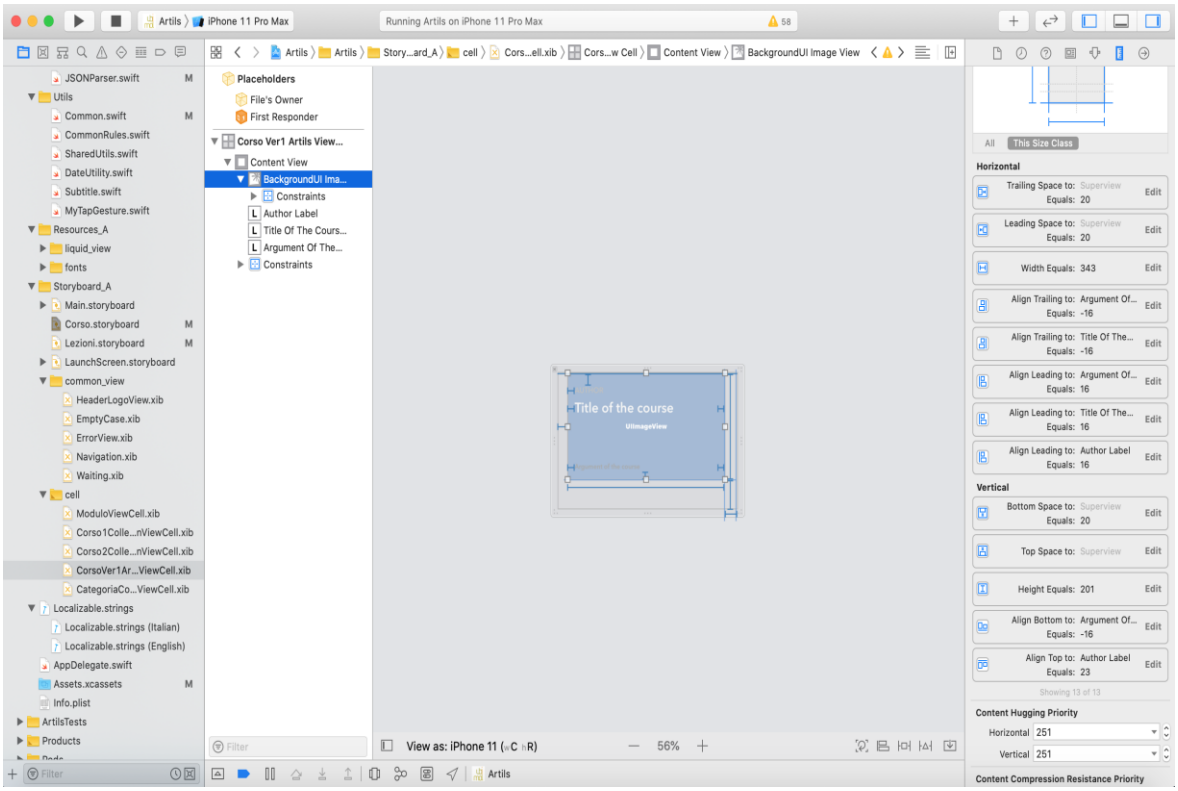

Figura 5.11: *Xib del corso*

La figura 5.11 mostra lo scheletro del file *xib* in questione, esso rappresenta la cella del corso. Una volta creato il file *xib* viene costruito come fosse un *view controller*, includendo gli elementi desiderati al suo interno. Una parte da non sottovalutare che ha causato parecchi problemi soprattutto agli inizi dello sviluppo è stata la costruzione grafica degli elementi e soprattutto l'inserimento dei *constraint* necessari per permettere all'interfaccia grafica di adattarsi a qualsiasi *display*, Infatti proprio quest'ultimi sono stati oggetto di molteplici test tramite il simulatore utilizzando diversi dispositivi per cercare di renderlo *responsive* ottenendo così il risultato desiderato.

```
import UIKit

class CorsoVer1ArtilsViewCell: UICollectionViewCell {

    @IBOutlet weak var authorLabel: UILabel!
    @IBOutlet weak var titleOfTheCourseLabel: UILabel!
    @IBOutlet weak var argumentOfTheCourseLabel: UILabel!

    @IBOutlet weak var backgroundUIImageView: UIImageView!

    override func awakeFromNib() {
        super.awakeFromNib()
        // Initialization code
        authorLabel.textColor =
Constants.Colors.COLOR_ARTILS_CELL_AUTHOR_LABEL
        titleOfTheCourseLabel.textColor =
Constants.Colors.COLOR_ARTILS_CELL_COURSE_TITLE_LABEL
        argumentOfTheCourseLabel.textColor =
Constants.Colors.COLOR_ARTILS_CELL_ARGUMENT_LABEL

    }
}
```

Figura 5.12

Esaminando la classe *CorsoVer1ArtilsViewCell,* ovvero la classe collegata al file *xib,* al suo interno vengono dichiarati gli oggetti presenti nel relativo file: autore, titolo del corso, argomento del corso e immagine di *background*. Nella classe vengono solamente impostati gli elementi statici dell'interfaccia, questo perché come già anticipato gli elementi dinamici verranno assegnati nel *view controller* in cui viene utilizzato il file, proprio questo passaggio rende possibile utilizzare lo stesso file *xib* più volte.

5.4 Gestione corsi

Quando l'utente entra all'interno della pagina del dettaglio del corso, potrà usufruire delle lezioni e degli esercizi messi a disposizione però solamente dopo aver effettuato il download del corso. Per scaricare il corso è stato utilizzato un metodo messo a disposizione dalla libreria *Alamofire*. L'interfaccia utente della pagina dei corsi si presenta in questo modo, come già anticipato in precedenza, con la lista delle lezioni disponibili e con un bottone in fondo alla pagina per permettere il download. Se si clicca sul bottone scarica comparirà una nuova *view* completamente bianca con un *activity indicator* che simula il download in corso. Una volta terminato ritornerà la *view* precedente con il bottone scarica nascosto ed al suo posto il bottone per annullare il download.

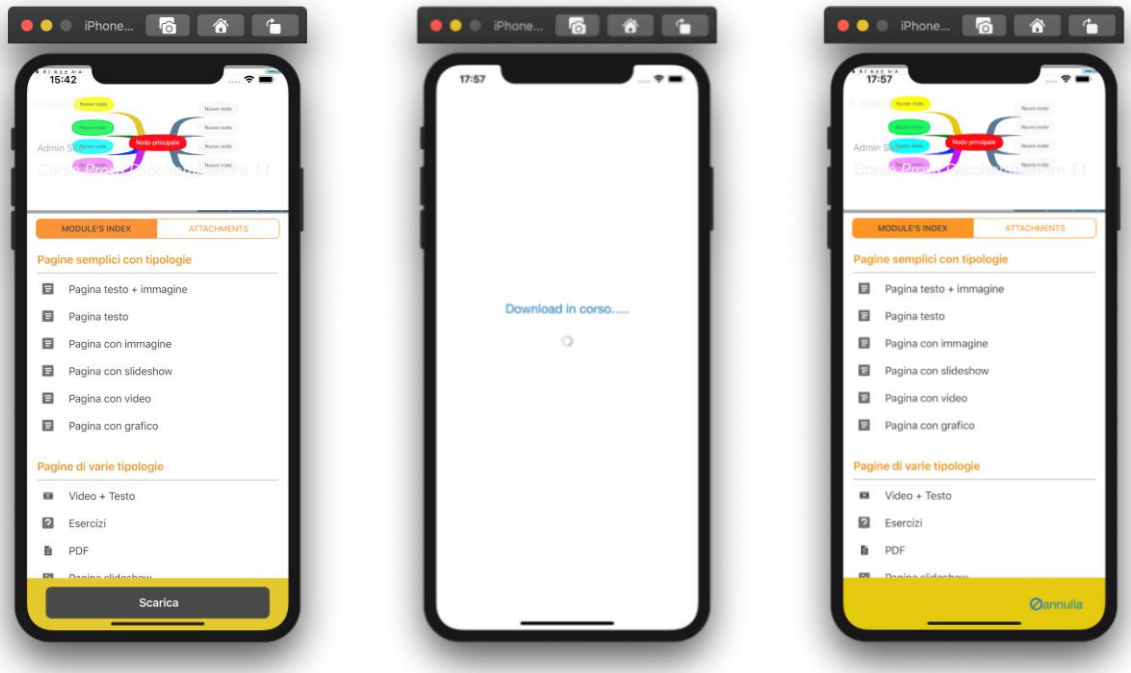

Figura 5.13: *Pagina dettaglio del corso*

5.4.1 Salvataggio dati

Le seguenti righe di codice mostrano il metodo standard di download presente in *Alamofire*. Alla funzione vengono passati come parametri in input l'url del download, il tipo di chiamata

ovvero la *.get*, l'header che equivale al *bearer authentication* ovvero la stringa generata dal server per permettere l'accesso ed infine la destinazione dove salvare il file.

```
Alamofire.download(downloadUrl, method: .get, parameters: nil, encoding:
URLEncoding.default, headers: header, to: destination)
        .downloadProgress(closure: { (progress) in
            let payload: [String:Any] = [Constants.Notification_Keys.downloadedValue:
Float(progress.fractionCompleted),
                                         Constants.Notification_Keys.downLoadingUuid:localUrl,
                                         Constants.Notification_Keys.indexRow:indexRow]
            NotificationCenter.default.post(name: .downloading, object: self, userInfo:
payload)
        })
        .responseData(completionHandler: { (response) in
            completion(response)
        })
```

Figura 5.14

Una volta creata la funzione basterà solamente applicarla all'interno del *view controller* desiderato; poiché il file scaricato è fornito in archivio .zip, occorre effettuare l'unzip, per fare questo si è scelto di utilizzare una libreria chiamata *SSZipArchive*.

```
SSZipArchive.unzipFile(atPath: sourceURLString, toDestination:
destinationURLString)
```

Figura 5.15

Nella figura 5.15 viene mostrato il codice utilizzato per effettuare l'unzip indicando l'url del file e l'url destinazione dove salvarlo.

5.5 Question Answering

La pagina *Question Answering* permette all'utente di interrogare il sistema, il quale in base alla domanda posta restituirà una risposta coerente grazie all'assegnazione di un punteggio. Lo storyboard si presenta nel seguente modo:

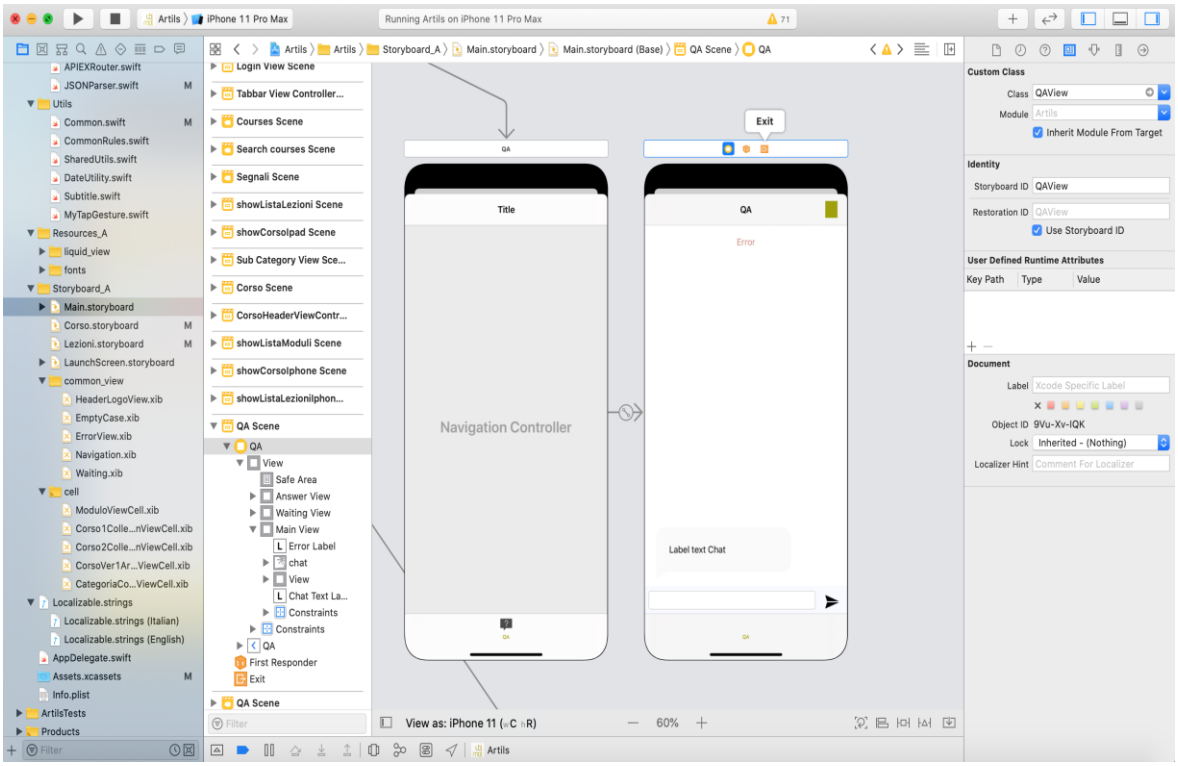

Figura 5.16: *Storyboard della pagina QA*

Un semplice *navigation controller* collegato ad un *view controller* al cui interno sono state inserite tre diverse *view*:

1. *Main View*: rappresenta ciò che l'utente vede nel momento in cui dovrà inserire una domanda
2. *Waiting View*: rappresenta una semplice *view* di caricamento, è uno stato transitorio tra le due *view* principali
3. *Answer View*: rappresenta ciò che l'utente vede nel momento in cui il sistema ha restituito una risposta.

Il risultato finale ottenuto è il seguente:

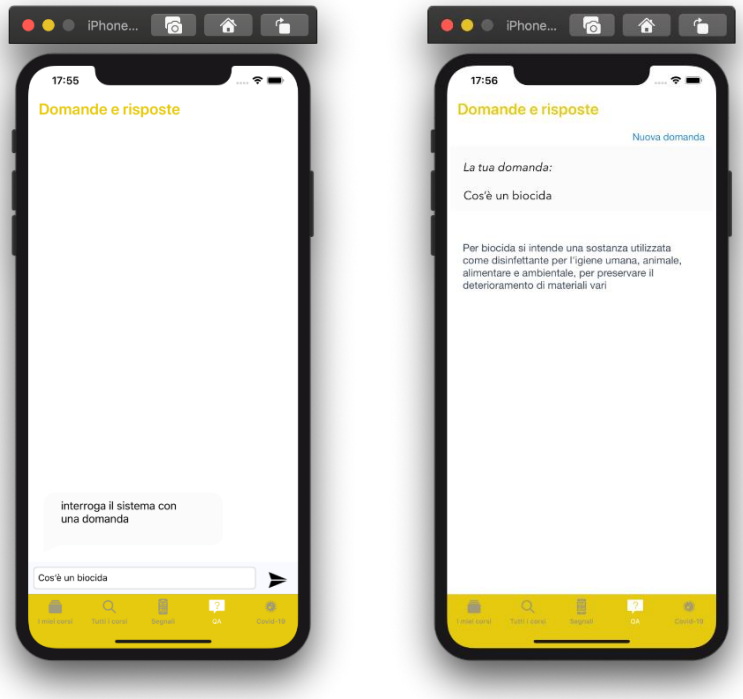

Figura 5.17: *Pagina QA*

5.5.1 Chiamata API GET

Quando l'utente effettua una domanda il sistema recupera le informazioni nel *server* tramite una chiamata *API GET*. Solitamente il codice riguardante le chiamate *API* all'interno del progetto viene scritto in un file a parte dichiarando la funzione statica così da poter essere richiamata nella classe di utilizzo.

```
static func getQA(question: String, completion:@escaping (DataResponse<Any>)
->Void) {
      Alamofire.request(APIRouter.getQA(question:
question)).responseJSON(completionHandler: { (response) in
            print(response)
            completion(response)
        })
  }
```

Figura 5.18

La funzione mostrata in figura 5.18 richiede un singolo parametro in *input*, ovvero la domanda dell'utente, mentre restituisce come risultato un *JSON* composto da un *array i*n cui sono presenti le risposte ordinate in base al punteggio assegnatogli dal sistema così da permettere di comprendere quale risposta sia più coerente. Il problema presentatosi inizialmente era quello di gestire la domanda dell'utente per evitare che inserisse caratteri che potessero portare al fallimento della chiamata *http*, questo perché all'interno dell'*url* della chiamata al server: *http://dev-artils-api.gruppometa.it/qa/cosa%20è%20il%20covid* è compresa la domanda dell'utente. Per far fronte a questo problema è stato utilizzato un metodo di Swift: *replacingOccurrences* per sostituire tutte le lettere accentate con lettere non accentate e sostituire gli spazi con il "%20". Una volta risolto questo problema si è continuato con la costruzione della chiamata *API*. Nella figura 5.19 viene mostrato un esempio di risposta della chiamata in *GET* ottenuto tramite il software *CocaRestClient.*


```
      {
"answer":"Per biocida si intende una sostanza utilizzata come
disinfettante per l'igiene umana, animale, alimentare e
ambientale, per preservare il deterioramento di materiali vari ",
"score":"0.9759892141651425"
      },
      {
"answer":"Il presidio medico chirurgico è un prodotto o un
dispositivo che contiene una o più sostanze disinfettanti,
germicide, battericide, fungicide, insetticide, topicide da usare
contro i corrispondenti organismi nocivi.",
"score":0.96911941385318334
      },
      {
"answer":"Va precisato che i vari prodotti per la disinfezione
(con specifiche proprietà nei confronti dei microrganismi), sono
diversi dai detergenti e dagli igienizzanti con i quali,
pertanto, non vanno confusi. ",
"score":0.95894244272624285
      }
```


Figura 5.19: *risposta del server in JSON*

Per agevolare l'importazione e il recupero dati nel database locale, sfruttato in caso di mancata connesione internet, è stato creato un *model* denominato Answer all'interno sono state create due variabili che dovranno avere necessariamente lo stesso nome e lo stesso tipo di quelle restituite dalla chiamata così da permettere la decodifica automatica da *JSON* ad *Object.*

```
import Foundation
import RealmSwift


class Answers: Object {
   @objc dynamic var answer: String = ""
   @objc dynamic var score: Double = 0.0

}
```

Figura 5.20

Una volta acquisito il contenuto del *JSON*, molto simile ad un *dictionary* chiave/valore in *Swift*, per proseguire occorre necessariamente convertire questi dati in oggetti *Swift*. Per ovviare a questa necessità è stata creata una funzione che avrà come parametro di input un array composto dal *JSON* restituito dalla chiamata GET e come parametro di output un array di tipo Answer precedentemente creato. All'interno di tale metodo verrà creato un nuovo oggetto di tipo Answer, successivamente tramite un ciclo *for* verrà "iterato" l'*array* del *JSON* così da poter assegnare la risposta della chiamata *API* ai due oggetti score ed answer del model Answer il quale sarà poi restituito dalla funzione. Una volta completati questi passaggi occorrerà solamente chiamare la funzione statica getQA precedentemente creata, al suo interno verranno gestiti i possibili risultati: in caso di fallimento verrà mostrato un messaggio di errore mentre in caso di successo (*status code 200*) verrà utilizzata la funzione precedentemente creata per convertire il contenuto del *JSON* in oggetti *Swift* ed ottenere così l'array di tipo Answer. Ora non resterà altro che salvare nel database i dati ottenuti mediante una query in *Realm*:

```
  class func addAll(_ elems: [Answers]) {
      let realm = try! Realm()
      try! realm.write {
          realm.add(elems)
      }
  }
```

Una volta salvati i dati basterà recuperarli mediante un ulteriore query:

```
  class func get() -> Answers? {
      let realm = try! Realm()
      let elem =
realm.objects(Answers.self).first
      return elem
  }
```

Figura 5.21: *Query Realm*

La funzione non farà altro che recuperare dal *database* locale il primo elemento dall'*array* che corrisponde al primo elemento restituitoci dalla chiamata *http* ovvero la risposta corretta alla domanda posta dall'utente.

5.6 Profilo utente

La pagina del profilo permette all'utente registrato di gestire le sue credenziali ed eventualmente apportare alcune modifiche. Per gestire questa funzione è necessario implementare una chiamata *API POST*. Lo *storyboard* della pagina si presenta in questo modo: è presente un *view controller* al cui interno si trova una singola *view*. Per questa pagina sostanzialmente sono state utilizzate cinque *text field* e due pulsanti. All'accesso della pagina le *text field* si riempiranno automaticamente con i dati del profilo utente, salvati ad ogni accesso

dell'utente, in una variabile globale. Il primo pulsante è stato creato per salvare i dati mentre il secondo per effettuare il *logout*.

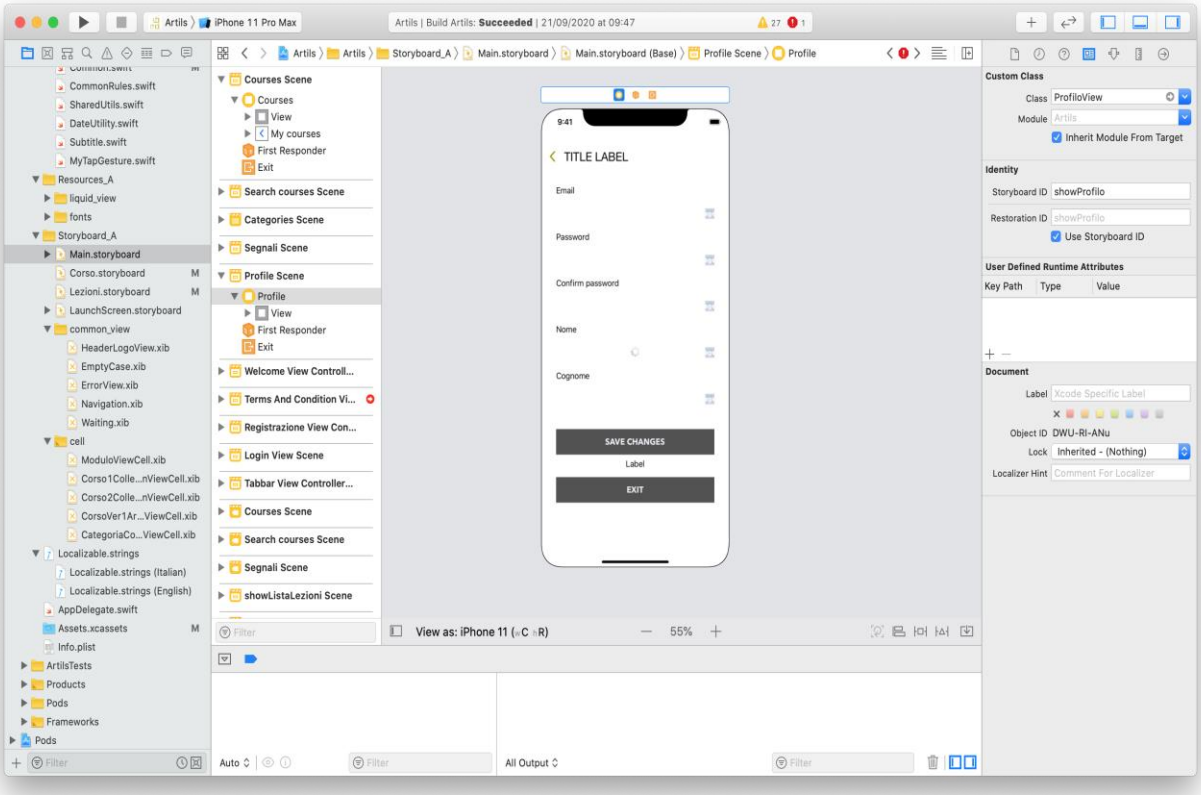

Figura 5.22: *Storyboard della pagina profilo utente*

Il risultato finale è il seguente:

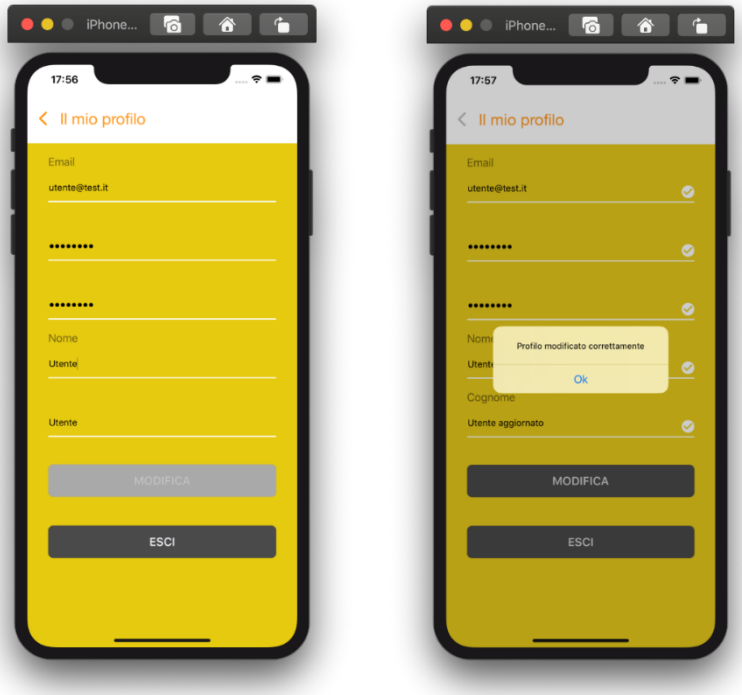

Figura 5.23: *Pagina profilo utente*

5.6.1 Chiamata API POST

Per permettere all'utente di modificare i campi del profilo è necessario effettuare una chiamata *API POST* utilizzata per inviare dati al server, infatti a differenza della chiamata GET la POST richiede un *body* ovvero dei dati che possono essere utilizzati per modificare il *database* remoto. In questa circostanza è stato preferito suddividere la chiamata *API* in due: una per la modifica dei campi nome, cognome, e-mail ed username mentre la seconda per permettere la modifica della password. È stata effettuata questa scelta di implementazione perché il campo password difficilmente viene modificato dall'utente, perciò per evitare che ad ogni modifica venisse aggiornata anche la password sono state suddivise. Dato che entrambe le chiamate sono praticamente uguali verrà analizzata solamente quella di modifica password. Per gestire quale delle due chiamate API effettuare sono stati effettuati dei controlli nelle *text field* più precisamente è stato controllato tramite un *if* se la *text field* della password inserita dall'utente è diversa dalla password già presente. In caso non siano diverse allora verrà effettuata la chiamata API che modificherà tutti i campi meno che la password. Mentre nel caso in cui il

controllo restituisca un risultato positivo allora si procede con la chiamata *API* per la modifica della password.

É stata creata una funzione statica (figura 5.24) che avrà come parametri la vecchia password da modificare, la nuova password digitata dall'utente, l'username e l'e-mail che eventualmente potranno essere cambiati o meno.

```
   static func updatePassword(oldPassword: String, newPassword: String,
username: String, email:String, completion:@escaping (DataResponse<Any>)-
>Void) {
      Alamofire.request(APIRouter.updatePassword(oldPassword: oldPassword,
newPassword: newPassword, username: username, email: email))
         .responseJSON(completionHandler: { (response) in
            print(response)
            completion(response)
         })
   }
```

Figura 5.24

La funzione restituisce un JSON ma in questo caso sarà semplicemente un intero (0-1) che comunicherà se la chiamata è andata a buon fine oppure no.

L*'url* della chiamata API è il seguente*:* *http://dev-artils-api.gruppometa.it/user/change-password*, mentre il *body*, che questa volta a differenza della chiamata in GET è richiesto, è il seguente:

```
{
         "oldPassword":"password",
         "newPassword":"password1",
         "username":"utente@test.it",
         "email":"utente@test.it"
}
```

Come è possibile notare corrisponde ai parametri della funzione, mentre il risultato della risposta se va a buon fine è il seguente:

```
{
         "status":1;
}
```

Per utilizzare la funzione all'interno del *view controller* del profilo, non sarà necessario creare un metodo per convertire il *JSON* in oggetti *Swift* in quanto, in questo caso, basterà semplicemente la seguente riga di codice:

```
let status = json["status"] as? Bool
```

Se la chiamata va a buon fine (*status code: 200*) viene controllato che lo status restituitoci sotto forma di *JSON* precedentemente convertito sia *true(1)*, se è così si procede con la modifica dei dati all'interno del database mediante la seguente *query*:

```
func updatePassword(password: String, username:String, email: String) {
    let realm = try! Realm()
    try! realm.write {
        self.user?.username = username
        self.user?.email = email
        self.user?.password = password
        realm.add(self, update: .modified)
    }
}
```

Figura 5.25

In input vengono passati come parametri la nuova password inserita, l'username e l'e-mail così da permetterne la modifica con in campi già presenti nel database. Invece se lo status restituitoci dalla chiamata è *false(0)* verrà mostrato un messaggio di errore per comunicare che la chiamata *API* non è andata a buon fine e la password non è stata modificata.

CAPITOLO 6

Conclusione

In quest'ultimo capitolo riporto una valutazione degli obiettivi conseguiti, ponendo in luce come questa prima esperienza avuta con il mondo lavorativo sia stata per me formativa.

6.1 Obiettivi raggiunti

L'obiettivo posto dalla tutor aziendale nel corso delle 300 ore di stage, era quello di insegnarmi e consentirmi di aiutare l'azienda con lo sviluppo di un'applicazione. Come riportato nel capitolo 5 sono riuscito a completare gli obiettivi prefissati implementando diverse funzionalità per l'applicazione *ARTILS.*

In definitiva l'azienda si è ritenuta soddisfatta del lavoro svolto e delle capacità di apprendimento da me dimostrate.

6.2 Conoscenze acquisite

Durante l'esperienza di stage posso dire di aver ampliato enormemente il mio bagaglio formativo, in particolare ho appreso le basi per lo sviluppo di applicazioni mobile native iOS, utilizzando tecnologie che non erano previste durante il corso di studi. Durante la mia carriera accademica ero stato abituato a lavorare a progetti con vincoli ben definiti. Invece, in questa esperienza lavorativa, il fatto di dover ricercare e scegliere personalmente il miglior approccio per sviluppare una soluzione, mi ha aiutato ad ottenere delle competenze sicuramente molto utili in un ambiente lavorativo altamente dinamico come quello dell'informatica.

Infine l'esperienza di lavoro in azienda mi ha arricchito permettendomi di acquisire maggiore autonomia in ambito lavorativo, grazie a ricerche e sviluppo individuali, che rendendomi consapevole di quanto nello sviluppo del software, avendo interagito con sviluppatori molto più esperti di me, sia importante la collaborazione e il confronto con gli altri, che diviene, oltre che utile, praticamente indispensabile.

6.3 Valutazione personale

In conclusione mi ritengo pienamente soddisfatto del lavoro svolto durante lo stage. Come già esposto, l'esperienza mi ha arricchito moltissimo a livello formativo ma soprattutto sono davvero soddisfatto di come mi abbia arricchito a livello umano. Infatti l'accoglienza in azienda e il clima di serenità presente al suo interno mi ha permesso di svolgere lo stage in armonia facendomi ancor più appassionare a questo lavoro. Sono altresì riconoscente nei confronti della Facoltà di Informatica, la quale mi ha permesso di svolgere questo stage. Ritengo che questa tipologia di tirocinio presso un'azienda sia essenziale per un laureando in informatica, in quanto permette di acquisire delle nuove competenze e la capacità di affrontare degli ostacoli che di solito il mondo accademico non riesce a far emergere.

Ritengo doveroso, in conclusione, ringraziare la tutor aziendale, nonché correlatrice di questo documento di Laurea, la cortesissima Dr.ssa Manuela Benedetti, che è sempre riuscita a valorizzare il lavoro da me svolto insieme a suo fratello, Dr. Daniele Benedetti, entrambi molto disponibili e gentili, che con determinazione e dedizione portano avanti un grande progetto lavorativo che hanno scelto di condividere con me, proponendomi di continuare a lavorare nella loro azienda.

Ho scelto con piacere di accettare la loro proposta, vista la bellissima esperienza avuta ed il forte interesse lavorativo maturato nel corso del tirocinio.

Bibliografia

[1] https://it.wikipedia.org/wiki/Apple

[2] https://it.wikipedia.org/wiki/Xcode

[3] https://it.wikipedia.org/wiki/Swift_(linguaggio_di_programmazione)#Caratteristiche_e_curiosit%C3%A

[4] https://developer.apple.com/documentation/uikit

[5] *Apple inc. – Education, App Devolopment with Swift, 2019*

[6] https://computerscience.unicam.it/marcantoni/reti/ch01%20-%20Internet%20e%20Reti%20di%20Calcolatori.pdf

[7] https://computerscience.unicam.it/marcantoni/reti/ch02%20-%20Protocolli%20Applicativi.pdf

[8] https://realm.io/docs/swift/latest/

[9] http://cocoadocs.org/docsets/Alamofire/4.5.1/

[10] https://developer.apple.com/documentation/uikit/uitabbarcontroller