



UNIVERSITÀ DEGLI STUDI DI UDINE

Dipartimento di Scienze Matematiche, Informatiche e Fisiche

Corso di laurea in Tecnologie Web Multimediali

**PROGETTAZIONE E SVILUPPO DI
UN'APPLICAZIONE PER LA
FIDELIZZAZIONE E IL REWARDING
DEGLI UTENTI**

RELATORE
Prof. STEFANO BURIGAT

LAUREANDO
RICCARDO CARANFIL

ANNO ACCADEMICO 2019/2020

*Ringrazio la mia famiglia e tutti i miei amici per avermi
accompagnato lungo questo percorso*

Indice

Introduzione	1
1 Progettazione iniziale	3
1.1 Obiettivo	3
1.2 Requisiti essenziali	3
2 Componenti base di iOS	5
2.1 Navigazione standard	5
2.1.1 Tipologie di navigazione	6
2.2 UIKit Dynamics	7
2.2.1 Comportamenti base	8
2.3 UIWindow	9
2.4 Delegation pattern	10
2.4.1 Esempio Delegation	10
2.5 UIGestureRecognizer	12
3 Navigazione dinamica	15
3.1 Il Coordinator Pattern	15
3.2 Implementazione	17
3.2.1 Protocolli base	17
3.2.2 Coordinator modale	20
3.2.3 Esempio di utilizzo	20
3.3 Aggiunta del Navigator	20

4	QIX Shake	21
4.1	Cos'è il QIX Shake?	21
4.2	Implementazione sfruttando ereditarietà	22
4.3	Implementazione con estensione	22
5	Animazioni interattive	25
5.1	Scomposizione del requisito	25
5.1.1	1. Presentazione delle animazioni	26
5.1.2	Aggiunta del ViewController nella UIWindow	26
5.1.3	2. Aggiunta di una gesture	27
5.1.4	3. Aggiunta della gravità	29
5.1.5	4. La CardView modulare	30
5.1.6	5. Paginazione delle animazioni	33
5.1.7	5. Il Collider	33
6	Autenticazione	35
6.1	Firebase Authentication	35
6.2	Tipologie di autenticazione	35
7	Dynamic Links	37
	Conclusione	39

Introduzione

L'obiettivo principale del tirocinio presso **Urbana Smart Solutions srl** [1] è stata la creazione di un'applicazione iOS denominata **QIX** atta alla fidelizzazione e al rewarding di un target di utenti specifico. Il principali target di clienti a cui mira l'applicazione sono delle FMCG¹, ossia delle compagnie che vendono beni di consumo a basso costo e molto velocemente.

Tali compagnie attraverso i loro prodotti possono creare diverse tipologie di eventi e gli utenti dell'applicazione possono accedervi e vincere dei punti denominati **QIX coins** con cui comprare o avere degli sconti sui beni venduti.

L'elemento cardine dell'app è il **QIX Shake** ossia l'attivazione di un particolare servizio che dipende dal contesto attuale dell'utente e il tipo di offerta che viene selezionata. Tale servizio dà agli la possibilità di vincere dei QIX coins agitando lo smartphone. I principali servizi di shake dell'applicazione sono:

1. **TV Shake**: agitando lo smartphone inizierà un'analisi dei dati del microfono allo scopo di trovare un particolare **watermark** inserito in campagne pubblicitarie televisive;
2. **Read Shake**: all'utente viene proposta la lettura di contenuti o questionari in cambio di QIX coins;
3. **Video Shake**: a seguito della visualizzazione di uno video l'utente viene premiato con dei punti;

¹Fast Moving Consumer Goods

4. **Scan shake:** dopo aver scannerizzato il barcode di un prodotto (Es. al supermercato) all'utente verranno assegnati dei punti;
5. **Receipt Shake:** dopo aver scannerizzato uno scontrino di acquisto di prodotti partner delle FCMG;
6. **Stadium Shake:** inierà anche in questo caso l'analisi del suono esterno per la ricerca di eventuali watermark generati in un'audio durante delle partite allo stadio;

Capitolo 1

Progettazione iniziale

1.1 Obiettivo

Il mio compito nello sviluppo dell'applicazione è stato quello di creare un **prototipo** iniziale avendo a disposizione un mock up creato con proto.io [2] e una serie di requisiti essenziali.

1.2 Requisiti essenziali

La base di partenza di QIX sono state delle funzionalità essenziali e sostanzialmente molto difficili da inserire in una versione dell'app già avanzata. È stato quindi deciso di creare un prototipo di partenza avente i seguenti requisiti:

- **Navigazione dinamica:** L'applicazione deve gestire dei cambiamenti di contesto dinamici: dev'essere possibile mostrare all'utente contenuti dinamici indipendentemente dal contesto in cui si trova.
- **QIX Shake:** L'utente deve poter agitare lo smartphone in qualsiasi sezione dell'applicazione e il risultato deve essere basato sul contesto attuale o su delle direttive dettate dalle Rest API;
- **Animazioni interattive:** L'intera applicazione dev'essere progettata in modo tale da presentare all'utente delle **animazioni interattive**

in stile CardView [3] disponibili in qualunque sezione o vista in cui si trovi l'utente e definite dal contesto attuale;

Le animazioni in questione devono essere progettate in pagine, in cui ogni pagina può contenere più CardView. L'utente vedrà in un determinato momento una e soltanto una pagina.

Ogni CardView deve essere trascinabile dall'utente e deve interagire con le altre CardView della pagina. Quando l'utente usa una forza di trascinamento superiore a un valore di soglia tutte le viste devono cadere per gravità;

Tale gravità finirà con la fine dell'animazione o l'apparizione di una nuova pagina se presente;

- **Autenticazione:** L'applicazione deve supportare tre diversi stati o modalità di autenticazione:
 1. **Trial Mode:** l'utente è anonimo, esiste solo un id per tenere traccia dei suoi QIX coins.
 2. **Signed Mode:** l'utente ha inserito il numero di telefono e il suo genere;
 3. **Pro Mode:** l'utente aggiunge dei dati su se stesso o collega il suo account a dei social media come Facebook, Google o Instagram;

Si nota facilmente che non esiste uno stato in cui l'utente non è registrato: questo perché per tenere traccia dei suoi QIX coins e di altri dati utili è necessario avere un riferimento all'utente;

- **DeepLinks:** L'applicazione deve poter essere avviata dinamicamente attraverso dei **Deep Links** [4]; E deve essere in grado di gestirli in base al contesto dell'utente;

Capitolo 2

Componenti base di iOS

Prima di entrare nel merito delle soluzioni e problemi affrontati, elenco brevemente gli strumenti base di iOS che mi sono stati utili nella progettazione e implementazione finale.

2.1 Navigazione standard

Un'applicazione iOS è un insieme di **UIViewController** [5] diversi che regolano ogni aspetto della loro vista;

Ogni applicazione può avere degli **UINavigationController** [6], ossia dei contenitori di **UIViewController** che vengono utilizzati per mantenere lo stack di navigazione e gestire le transizioni tra due **UIViewController**.

Nella figura 2.1 si nota facilmente come un **NavigationController** gestisce un'array di **ViewController** e una sola **navigation bar**.

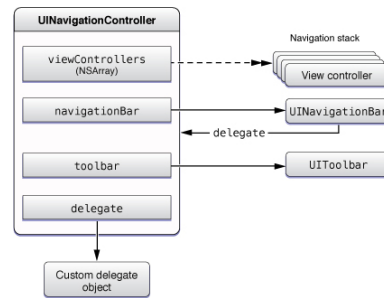


Figura 2.1: Navigation controller scheme

2.1.1 Tipologie di navigazione

Esistono tre tipologie base di navigazione:

1. **Push:** un UIViewController figlio di un UINavigationController può rendere visibile un altro ViewController attraverso la funzione "push-ViewController" di un Navigation controller. Ogni UIViewController ha infatti un puntatore al UINavigationController genitore, che gestisce la navigazione.

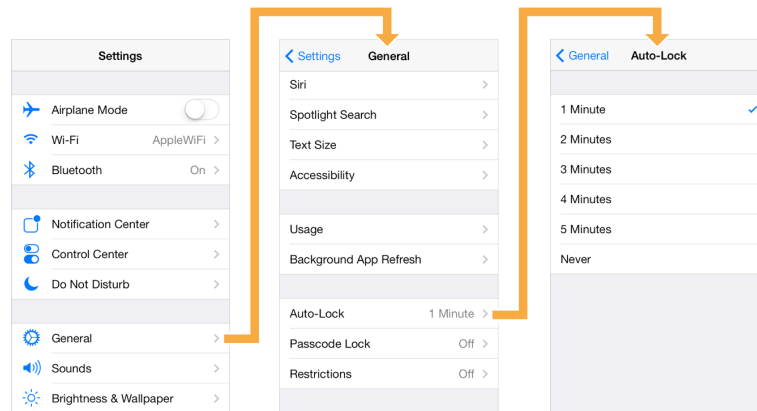


Figura 2.2: Presentazione di un ViewController tramite push

2. **Modal:** un ViewController può presentare un altro ViewController senza necessariamente avere un Navigation Controller, l'animazione standard è dal basso verso l'alto come in figura 2.3. La navigazione

modale viene usata solitamente per la presentazione di task che richiedono una sola operazione oppure per rendere evidente un cambio del contesto precedente.

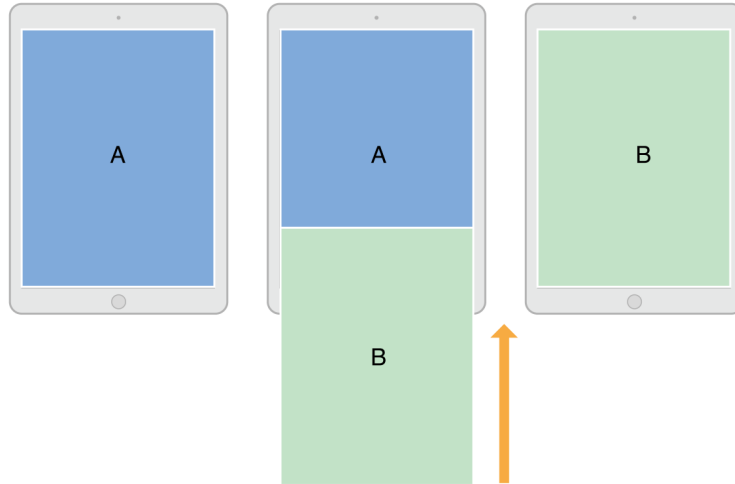


Figura 2.3: Presentazione di un ViewController tramite modal

3. **Segue:** Una segue non è altro che un link tra due view controller attraverso un'interfaccia grafica. In base alla tipologia cambia il tipo di navigazione (Modal o Push). Le segue vengono utilizzate programmaticamente o nei file **storyboard**, in cui ogni ViewController viene "disegnato", e poi linkato ad altri ViewController.

2.2 UIKit Dynamics

Per la progettazione iniziale delle animazioni è stato fatto un attento studio a metodologie e frameworks atti a creare animazioni interattive fluide.

Alla fine è stato deciso di utilizzare un pacchetto nativo di iOS incluso nello UIKit [7], chiamato UIKit Dynamics [8]: questo framework, con una serie di API, offre delle funzioni di animazione base che donano alle viste i comportamenti fisici del mondo reale.

Il framework si basa su degli oggetti **UIDynamicAnimator** in cui ogni animator è responsabile delle animazioni che avvengono sulla sua **referenceView** e si inizializza attraverso la seguente funzione:

```
UIDynamicAnimator.init(referenceView view: UIView)
```

2.2.1 Comportamenti base

Definiamo **DynamicItem** qualsiasi vista a cui viene assegnato un **UIDynamicBehavior**. Ogni animator attraverso il metodo `addBehavior` può assegnare a determinate viste comportamenti fisici elencati sei seguito:

- **UIAttachmentBehavior**: crea una relazione o legame tra due **DynamicItem** o tra un **DynamicItem** e un punto di ancoraggio;
- **UICollisionBehavior**: un oggetto che conferisce a un array di **DynamicItems** la possibilità di impegnarsi in collisioni tra loro e con i limiti specificati del comportamento
- **UIFieldBehavior**: un oggetto che conferisce delle proprietà magnetiche, elettriche a un **DynamicItem**;
- **UIGravityBehavior**: aggiunge all'oggetto una forza di gravità;
- **UIPushBehavior**: Aggiunge all'oggetto una forza continua o istantanea in una direzione specifica;
- **UISnapBehavior**: Un comportamento simile a una molla il cui movimento iniziale viene smorzato nel tempo in modo che l'oggetto si stabilizzi in un punto specifico.

Di seguito un esempio di implementazione degli strumenti **UIDynamics** in QIX

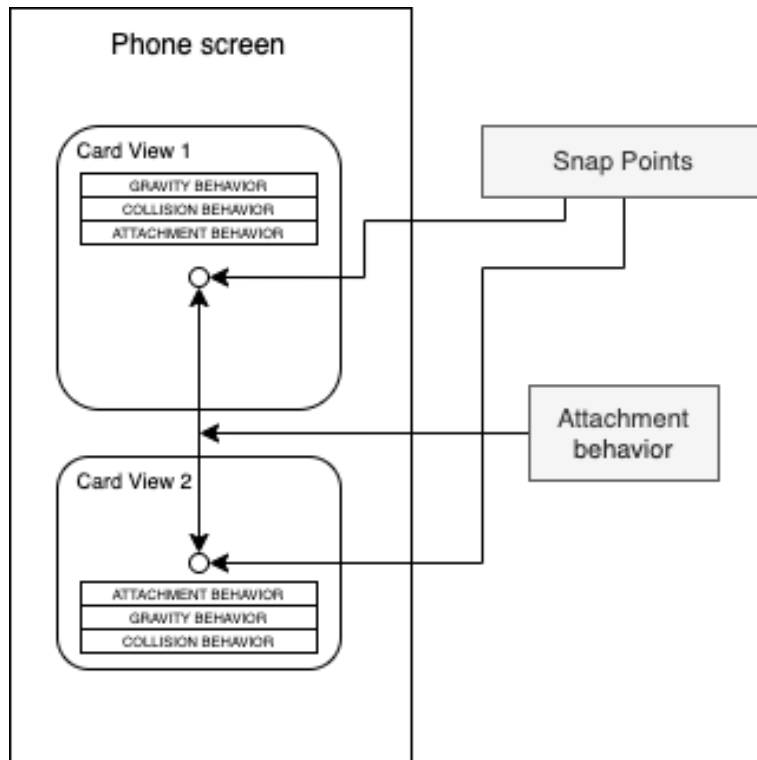


Figura 2.4: Schema dell'implementazione di UIDynamics

2.3 UIWindow

Il contenuto di ogni applicazione iOS è inserito all'interno di un oggetto denominato **UIWindow** [9]. Questa finestra è disponibile in ogni UI-ViewController e permette di aggiungere contenuti come viste o interi UI-ViewController al di sopra di tutto il contesto dell'app. Questo rende essenzialmente ogni contenuto presentato indipendente per esempio da un stack di navigazione.

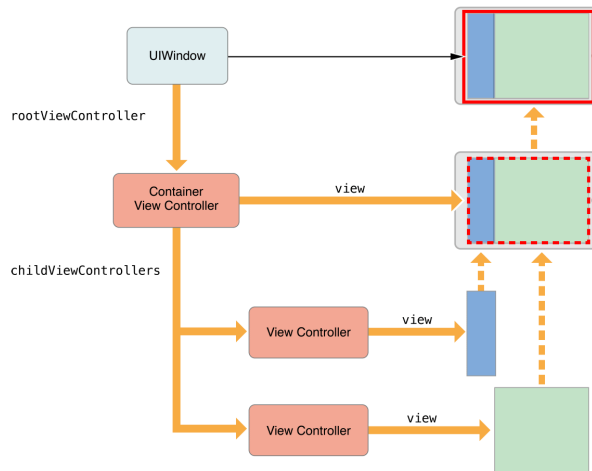


Figura 2.5: Rappresentazione dell UIWindow

2.4 Delegation pattern

Il **Delegation pattern** è una delle strutture più usate nelle app iOS in particolare in QIX è stato molto sfruttato per la comunicazione tra ViewController e Coordinator.

Tale pattern non è altro che un insieme di interfacce e metodi che vengono progettate per pubblicizzare con il resto dell'applicazione degli eventi che avvengono in uno specifico ViewController

In Swift le interfacce si utilizzano attraverso la Keyword **protocol**, questi oggetti possono contenere metodi, variabili, closure e sono molto utili per la definizione di strutture dinamiche tipizzate.

2.4.1 Esempio Delegation

Un semplice esempio potrebbe essere un ViewController che vuole pubblicizzare la pressione di un bottone e dall'altra parte il coordinator che viene notificato e agirà di conseguenza:

```
// ----- View controller -----  
// Interfaccia per la definizione delle API
```



```
protocol ExampleViewControllerDelegate {
    func exampleViewController(_ controller: UIViewController,
                               didPressButton button: Any)
}

class ExampleViewController: UIViewController {

    // Istanza opzionale del delegate
    weak var delegate: ExampleViewControllerDelegate?

    // Evento di pressione del bottone
    @IBAction func buttonPressed(_ sender: Any) {
        // Se istanziato il delegate lancia la funzione
        self.delegate?.exampleViewController(self,
                                              didPressButton: sender)
    }
}

// ----- Coordinator -----
class ExampleCoordinator: Coordinator {

    private var exampleViewController: ExampleViewController

    func init() {
        let exampleViewController = ExampleViewController()
        self.exampleViewController = exampleViewController

        // Presentazione modale dell'exampleViewController
        self.present(exampleViewController, animated: true)
    }

    func start() {
        // Impostazione del delegate
        exampleViewController.delegate = self
        // ...
    }
}
```

```
    }  
}  
  
extension ParentViewController: ExampleViewControllerDelegate {  
    func exampleViewController(_ controller: UIViewController,  
        didPressButton button: Any){  
  
        // Handling della pressione del bottone  
    }  
}
```

Nell'esempio è in parte utilizzato anche il Coordinator Pattern che spiegherò al capitolo 3.

2.5 UIGestureRecognizer

In iOS per interagire con le viste attraverso il display touch screen si utilizzano delle UIGestureRecognizer. Tali strumenti sono nativi e offrono diverse tipologie per l'iterazione:

- UITapGestureRecognizer: responsabile della gestione dei tap
- UIPinchGestureRecognizer: responsabile della gestione del pitch ossia la gesture spesso usata per lo zoom
- UIRotationGestureRecognizer: responsabile della gestione delle rotazioni
- UISwipeGestureRecognizer: responsabile di uno swipe ossia un trasciamento in una direzione molto breve
- UIPanGestureRecognizer: responsabile del drag and drop
- UIScreenEdgePanGestureRecognizer: responsabile di uno swipe ossia un trasciamento nei bordi dello schermo

- `UILongPressGestureRecognizer`: responsabile di una pressione prolungata nel tempo

In QIX è stata usata una `UIPanGestureRecognizer` per la gestione del trascinamento delle `CardView` animate. L'implementazione è mostrata nella sezione 5.1.3

Capitolo 3

Navigazione dinamica

Avendo definito i principali metodi di navigazione tra ViewController al capitolo 2 torniamo al problema iniziale: *Come possiamo rendere dinamica la navigazione?*

A seguito di uno studio approfondito di varie tecniche di navigazione iOS ho scelto di utilizzare il **Coordinator Pattern** [10].

3.1 Il Coordinator Pattern

Generalmente in iOS l'intera logica di un ViewController viene scritta nel controller stesso, creando spesso file di grosse dimensioni e disordine generale. Il Coordinator Pattern è nato proprio per rendere le applicazioni più scalabili e leggere.

Ogni ViewController infatti delega tutte le decisioni al suo Coordinator usando il Delegation Pattern (vedere sezione 2.4) che in base alle logiche della vista in questione deciderà i passi successivi.

Ogni Coordinator può controllare un ViewController o più Coordinator, questo rende le viste indipendenti tra di loro e rende ogni ViewController totalmente invisibile agli altri.

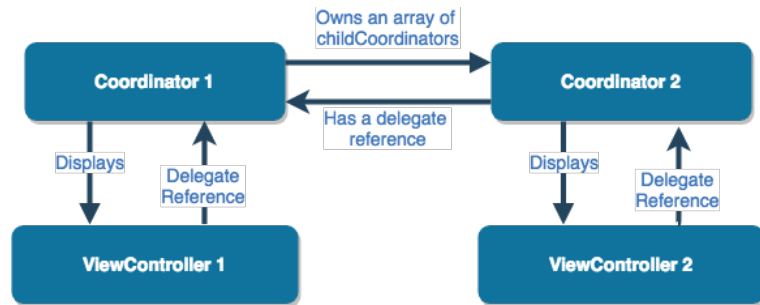


Figura 3.1: Il Coordinator Pattern

La responsabilità dei coordinator è infatti la navigazione, come un navigation controller gestisce i suoi View Controller, un coordinator gestisce i suoi figli e questo rende ogni vista o flow di navigazione totalmente indipendente dal resto dell'applicazione.

Per navigare tra i view controller vengono generalmente usate le tipologie di navigazione descritte nella sezione 2.1.1, tranne le segue, che essendo definite da vista grafica renderebbero la navigazione statica e fissata su determinati ViewController.

Di seguito in figura 3.2 presento uno schema dell'utilizzo di due coordinator per la gestione di una lista di prodotti e il carrello.

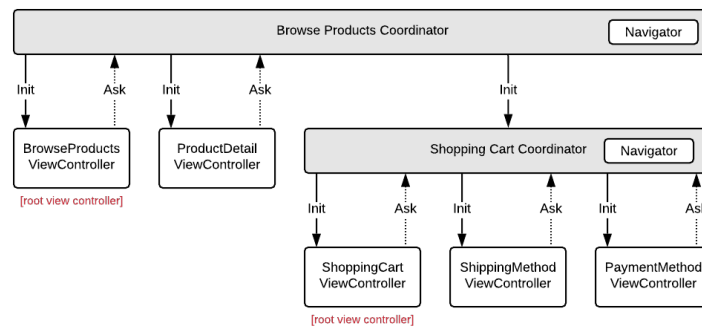


Figura 3.2: Esempio di coordinator pattern

Come si evince dalla figura 3.2 è presente in entrambi i coordinator un oggetto **navigator** che sarà il proxy di un generico UINavigationController

3.2 Implementazione

Ogni coordinator ha degli elementi fissi che sono:

- Una funzione di partenza denominata **start**;
- Un array di coordinator figli;

Per questo ho iniziato implementando dei protocolli:

3.2.1 Protocolli base

Il primo protollo implementato è quello che defisce un qualunque coordinator e ne tiene in memoria i figli in modo che non vengano deallocati automaticamente dal sistema operativo.

```
protocol Coordinator: class {  
  
    var childCoordinators: [Coordinator] { get set }  
  
    /** Utilizzato task di inizializzazione */  
    func start()  
  
}  
  
extension Coordinator {  
  
    // Aggiunge un figlio al coordinator padre  
    func add(childCoordinator: Coordinator) {  
        childCoordinators.append(childCoordinator)  
    }  
}
```

```

    // Rimuove un Coordinator figlio dal parent
    func remove(childCoordinator: Coordinator) {
        childCoordinators = childCoordinators.filter {
            $0 !== childCoordinator
        }
    }
}

```

Successivamente è stato implementato un `BaseCoordinatorPresentable`, che estende `Coordinator` e aggiunge delle funzionalità di presentazione modale.

```

protocol BaseCoordinatorPresentable: Coordinator {

    // Il view controller principale del coordinator
    var _rootViewController: UIViewController { get }
}

// MARK: - Presentation Methods

extension BaseCoordinatorPresentable {

    /**
     Inizia un coordinator figlio e presenta
     il suo rootViewController modalmente

     - Parametri:
       - childCoordinator: Il coordinator da presentare
       - animated: Specifica se con o senza animazione modale
     */

    func presentCoordinator(_ childCoordinator: BaseCoordinatorPresentable,
        animated: Bool) {

```



```

        add(childCoordinator: childCoordinator)
        childCoordinator.start()
        _rootViewController.present(childCoordinator._rootViewController, animated:
    }

    /**
     Inizia un viewController senza coordinator e
     lo presenta modalmente

     - Parametri:
       - controller: Il controller da presentare
       - animated: Specifica se con o senza animazione modale
    */

    func present(_ controller: UIViewController, animated: Bool) {
        _rootViewController.present(controller, animated: animated)
    }

    /**
     Interrompe la presentazione di un child Coordinator
     eliminandolo anche dall'array in memoria

     - Parameters:
       - childCoordinator: Il coordinator da chiudere e rilasciare
       - animated: Specifica se con o senza animazione modale
       - completion: closure che viene eseguita alla fine del dismiss
    */

    func dismissCoordinator(_ childCoordinator: BaseCoordinatorPresentable,
        animated: Bool, completion: (() -> Void)? = nil) {

        childCoordinator._rootViewController.dismiss(animated: animated,
            completion: completion)
        self.remove(childCoordinator: childCoordinator)
    }

```

```
}
```

3.2.2 Coordinator modale

Il `BaseCoordinatorPresentable` è stato definito per evitare errori di **associatedtype**, infatti questo protocollo non deve mai essere implementato direttamente, ma soltanto con il protocollo successivo

```
protocol CoordinatorPresentable: BaseCoordinatorPresentable {

    /**
     * Questo protocollo utilizza la tipizzazione per
     * permettere di ottenere Coordinator con view
     * controller tipizzati.
     */
    associatedtype ViewController: UIViewController

    // Il rootViewController del coordinator
    var rootViewController: ViewController { get }

}

extension CoordinatorPresentable {

    // Ritorna rootViewController
    var _rootViewController: UIViewController {
        return rootViewController
    }

}
```

3.2.3 Esempio di utilizzo

3.3 Aggiunta del Navigator

Capitolo 4

QIX Shake

4.1 Cos'è il QIX Shake?

In iOS ogni UIViewController risponde a degli eventi. L'evento designato per lo shake è

```
func motionEnded(_ motion: UIEvent.EventSubtype,  
with event: UIEvent?)
```

In caso di shake infatti motion sarà uguale a `.motionShake`.

Inizialmente per rendere disponibile l'evento "shake" in un qualunque ViewController la soluzione è stata abbastanza semplice: è bastato l'utilizzo di un ViewController genitore e attraverso l'ereditarietà ogni view controller è stato in grado di eseguire la stessa funzionalità.

Successivamente sono stati riscontrati dei problemi, dato che con il sempre più alto grado di complessità ogni vista avrebbe dovuto ereditare una classe base. In alcuni casi questo non è stato possibile, ho quindi optato per un'**estensione** della classe UIViewController e un semplice protocollo che aggiunge agli elementi che lo eridatono una closure opzionale.

Per notificare l'avvenuta del QIX Shake ho utilizzato delle notifiche locali: quando avviene uno shake il ViewController interessato invia una notifica globale e, nel mio caso, solamente l'AppCoordinator riceverà tale notifica. Si nota anche che questa notifica contiene un campo **sender** che

identifica il ViewController da cui viene lanciata e quindi il **contesto attuale** dell'utente

4.2 Implementazione sfruttando ereditarietà

```
class BaseViewController: UIViewController {

    override open func motionBegan(_ motion: UIEvent.EventSubtype,
        with event: UIEvent?) {

        if motion == .motionShake {
            let notification = Notification(
                name: Notification.Name.UserDidShake,
                userInfo: ["sender": self]
            )
            NotificationQueue.default.enqueue(notification,
                postingStyle: .asap)
        }
    }
}
```

Sebbene questo metodo sia semplice e funzioni, quello mostrato di seguita risulta migliore per gli scopi voluti.

4.3 Implementazione con estensione

```
protocol Shakerable {
    var shakeAction: ShakeClousure? { get }
}

extension UIViewController {

    override open func motionBegan(_ motion: UIEvent.EventSubtype,
        with event: UIEvent?) {
```

```
    if motion == .motionShake {
        let notification = Notification(
            name: Notification.Name.UserDidShake,
            userInfo: ["sender": self]
        )

        NotificationQueue.default.enqueue(notification,
            postingStyle: .asap)

        if let shakerable = self as? Shakerable {
            shakerable.shakeAction?(self)
        }
    }
}
```

L'esempio riportato qui sopra è molto simile a quello precedente, ma utilizzando le estensioni ogni UIViewController erediterà questo metodo sovrascritto senza dover far eridatere a tutti i ViewController una classe aggiuntiva, anche perchè, come abbiamo definito in precedenza, il QIX Shake deve essere disponibile ovunque.

Un altro fattore interessante è il protocollo **Shakerable** dato che tutti i ViewController che lo implementano possono, attraverso la `shakeAction` closure, eseguire ciò che devono a seguito di uno shake.

Capitolo 5

Animazioni interattive

5.1 Scomposizione del requisito

Per semplicità divido il requisito in diversi punti e per ognuno ne spiego la soluzione o metodologia utilizzata:

1. Le animazioni devono essere disponibili in qualunque sezione o vista in cui si trovi l'utente e definite dal contesto attuale;
2. Ogni CardView deve essere trascinabile dall'utente;
3. Nel momento in cui l'utente usa una forza di trascinamento superiore a un valore di soglia tutte le viste devono cadere per gravità;
4. Ogni CardView mostrerà un contenuto dinamico differente e definito da dei componenti limitati specifici
5. Le animazioni in questione devono essere progettate in pagine, in cui ogni pagina può contenere più CardView. L'utente vedrà in un determinato momento una e soltanto una pagina. Una volta che le CardView acquisiscono una gravità e cadono, finirà l'animazione o apparirà una nuova pagina, se presente;
6. Ogni CardView deve interagire con le altre della stessa pagina, come se si toccassero;

5.1.1 1. Presentazione delle animazioni

Per creare animazioni definite dal contesto usiamo quindi un semplice `UIViewController` che gestirà tutte le animazioni, ma invece di presentarlo attraverso i metodi base visti alla sezione 2.1.1, lo presentiamo al di sopra della **`UIWindow`** (vedere sezione 2.3), in modo da non essere vincolati dal contesto dell'utente quando l'animazione finirà, ma allo stesso tempo consentire il controllo dell'inizializzazione del `UIViewController` in base al contesto attuale.

5.1.2 Aggiunta del `ViewController` nella `UIWindow`

L'animazione viene inizializzata nel `ViewController`, in questo caso **`main-ViewController`**, genitore di tutto il contesto, in modo da avere il controllo su tutto il contesto. Questo perchè tutti i `ViewController` nello stack di navigazione possono essere chiusi a seguito di un qualunque eventi dell'animazione, avendo così un controllo molto accurato del contenuto che l'utente visualizzerà alla fine dell'animazione.

```
let keyWindow: UIWindow?

if #available(iOS 13.0, *) {
    keyWindow = UIApplication.shared.connectedScenes
        .filter({$0.activationState == .foregroundActive})
        .map({$0 as? UIWindowScene})
        .compactMap({$0})
        .first?.windows
        .filter({$0.isKeyWindow}).first
} else {
    keyWindow = UIApplication.shared.keyWindow
}

mainViewController.addChild(currentAnimationViewController!)
```



```
// Aggiungiamo la vista del nostro viewController alla UIWindow
keyWindow?.addSubview(currentAnimationViewController!.view)

// Rimuoviamo il viewController
currentAnimationViewController!.didMove(toParent: mainViewController)
```

5.1.3 2. Aggiunta di una gesture

Per un'animazione che ha necessità di muoversi come se l'utente la stesse spostando trascinando sullo schermo occorre una **UIPanGestureRecognizer**. Dalla documentazione delle `UIGestureRecognizer` viene spiegato che ogni vista "draggabile" necessita una e solo una gesture, per questo ogni card view dovrà averne una diversa. Di seguito il codice di implementazione

```
// Durante il settaggio delle cardview
// aggiungo la vista alla parentView
parentView.addSubview(view)

// Creo la gesture
let panGesture = UIPanGestureRecognizer(target: target, action: action)

// Assegno un nome che verrà utilizzato come lock
// per impedire la concorrenza di due gesture differenti
panGesture.name = "gesture-\(index)"

// Aggiungo la gesture alla cardView
cardView.addGestureRecognizer(panGesture)
```

I parametri `target` e `action` sono rispettivamente il viewController che gestisce la gesture e il metodo che deve essere eseguito quando avviene una `PanGesture`. Di seguito l'implementazione del metodo `handlePan` che gestisce tutte le gesture di una stessa pagina.

```
private let activeGesture = nil

// Metodo chiamato dalla UIPanGesture
```

```
@objc private func handlePan(_ recognizer: UIPanGestureRecognizer) {  
    // Lock per evitare che due gesture funzionino in contemporanea  
    if let gestureName = activeGesture, recognizer.name != gestureName {  
        return  
    } else {  
        activeGesture = recognizer.name  
    }  
  
    // CardView che viene mossa dall'utente  
    let animatedView = recognizer.view!  
  
    // Coordinate del punto di tocco rispetto alla vista principale  
    let touchPointInView = recognizer.location(in: self.view)  
  
    // Coordinate del punto di tocco rispetto alla cardView  
    let touchPointInAlert = recognizer.location(in: animatedView)  
    let velocity = recognizer.velocity(in: self.view)  
  
    // Ottengo l'offset rispetto al centro della cardView  
    let offset = UIOffset(  
        horizontal: touchPointInAlert.x - animatedView.bounds.midX,  
        vertical: touchPointInAlert.y - animatedView.bounds.midY  
    )  
  
    // Aggingo i comportamenti iniziali  
    self.createPageDynamicBehaviours()  
    self.addCheckForDismiss()  
  
    // Attivo e disattivo i comportamenti del UIDynamicAnimator  
    // in base agli stati delle PanGesture  
    switch recognizer.state {  
    case .began:  
        self.configureForStartDragging(animatedView,  
            with: offset,  
            touchPointInView: touchPointInView,
```

```

        touchPointInAlert: touchPointInAlert
    )
    case .changed:
        // Sposto la cardView attraverso un attachmentBehavior
        dragAttachmentBehavior.anchorPoint = touchPointInView
    case .cancelled, .ended, .failed:
        self.configureForFinishDragging(animatedView,
            with: velocity,
            offset: offset
        )
        // Disattivo il lock
        activeGesture = nil
    default:
        break
}
}

```

5.1.4 3. Aggiunta della gravità

Attraverso la UIPanGesture è possibile calcolare il vettore della velocità del trascinamento, con questo semplice dato e ciò che abbiamo studiato dello UIKit Dynamics è possibile aggiungere una gravità solo se il vettore della velocità è superiore a una soglia prestabilita.

Per implementarlo è stato utilizzato un UIGravityBehavior, inizializzato in questo modo:

```

let gravityBehavior = UIGravityBehavior(items: page.views)
gravityBehavior.magnitude = Constants.gravity

```

Il valore **magnitude** è la forza di gravità che vogliamo assegnare alle viste in questione.

Nella figura 2.4 si vede lo schermo di uno smartphone, che presenta l'animazione voluta in questo caso con due CardView. Ognuna di esse ha un **UIAttachmentBehavior** al centro per fare in modo che sia sempre

centrata in esso (o nel punto di drag dell'utente), un **UICollisionBehavior** per permettere che durante il drag le CardView possa scontrarsi e non si accavallino e un **UIGravityBehavior**, il quale viene utilizzato per la caduta delle viste alla fine dell'animazione o al cambiamento di pagina.

In più esiste uno speciale **UIAttachmentBehavior** tra i centri delle due CardView per permettere che il movimento di una sposti anche l'altra, è una sorta di corda che le lega.

Per l'ingresso invece è stato inserito un **UISnapBehavior**, al centro per ogni oggetto, che anima gli oggetti dando un effetto a molla.

Tutte le animazione sono attivate e disattive in specifici momenti, questo dipende dalla **UIPanGestureRecognizer** e dai movimenti dell'utente.

5.1.5 4. La CardView modulare

L'intero **UIViewControllor** che regola le CardView animate è stato progettato per presentare animazioni con contenuti dinamici, per questo sono state progettate delle strutture dati specifiche per consentire la personalizzazione del contenuto di ogni CardView.

Inizialmente sono stati definiti i possibili componenti che ogni CardView può adottare:

- **Solo testo:** Il componente include un testo, il suo colore e il colore di sfondo
- **Solo Immagine:** Il componente include un'immagine e il colore di sfondo
- **Row:** Il componente include un'icona e due testi in quest'ordine;
- **Animazione Lottie [11]:** Il componente include una semplice **UIView** in cui viene mostrata un'animazione Lottie
- **Footer:** Il componente include del testo e un'icona sulla destra;

Una volta definiti tutti i componenti per ogni CardView questa viene assemblata "incollando" un componente dopo l'altro attraverso una vista chiamata **ModularCardView**, di seguito, in figura 5.1 il risultato finale

ottenuto.

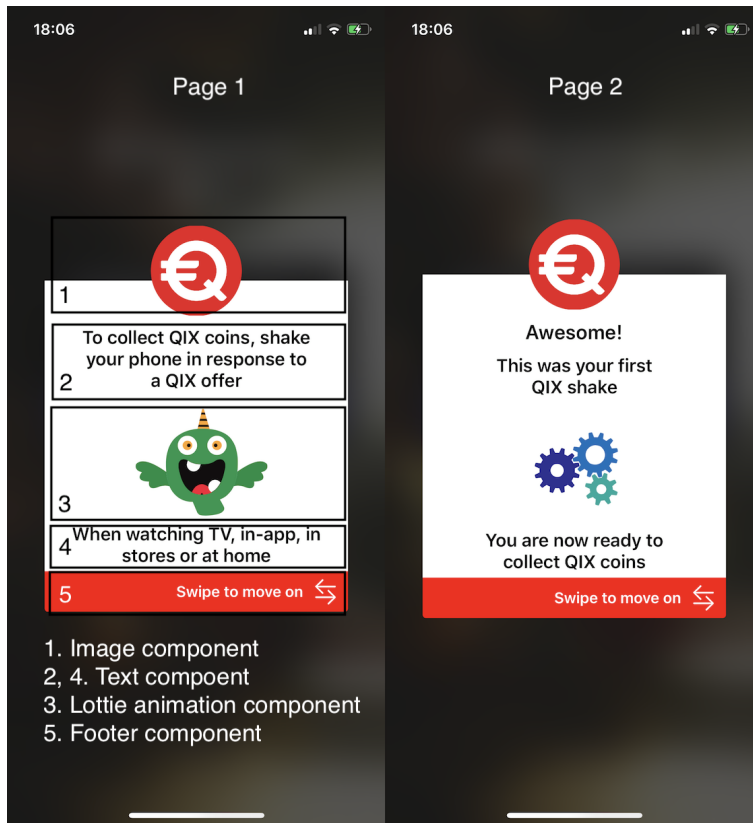


Figura 5.1: CardViews modulari

Dopo aver creato ogni componente separatamente ho creato la **Modular-CardView**, questa vista attraverso un array di `UIView`, ossia i componenti, utilizza dei constraints per assemblarli insieme. Vediamo di seguito il metodo **build**:

```
// Creo un array di constraints
var constraints = [NSLayoutConstraint]()

// Creo un loop enumerato sull'array dei componenti
for (index, view) in components.enumerated() {
```

```
// Disabilito l'autoresizing mask automatico
view.translatesAutoresizingMaskIntoConstraints = false

// Aggiungo il componente alla ModulaCardView
self.addSubview(view)

// Aggiungo i constraint a destra e sinistra
constraints.append(contentsOf: [
    view.leadingAnchor.constraint(equalTo: self.leadingAnchor),
    view.trailingAnchor.constraint(equalTo: self.trailingAnchor)
])

if components.count > 1 {

    if index == 0 {
        // Al primo componente setto il top della ModulaView
        constraints.append(
            view.topAnchor.constraint(equalTo: self.topAnchor)
        )
    } else {
        // Ai successivi setto top anchor il bottom
        // anchor dell'elemento precedente
        let bottomAnchor = components[index - 1].bottomAnchor
        constraints.append(
            view.topAnchor.constraint(equalTo: bottomAnchor)
        )
    }

    // Se è l'ultimo componente gli aggiungo un constraint
    // alla fine della ModulaView
    if index == components.count - 1 {
        constraints.append(
            view.bottomAnchor.constraint(equalTo: self.bottomAnchor)
        )
    }
}
```

```

    } else {
        // Esiste un solo componente,
        // gli setto i constraint sopra e sotto
        constraints.append(contentsOf: [
            view.topAnchor.constraint(equalTo: self.topAnchor),
            view.bottomAnchor.constraint(equalTo: self.bottomAnchor)
        ])
    }
}

// Attivo tutto i constraints
NSLayoutConstraint.activate(constraints)

```

5.1.6 5. Paginazione delle animazioni

Avendo definito al punto 1 l'utilizzo di un solo UIViewController per le animazioni da un lato viene semplificata la presentazione delle stesse, dato che basterà presentare un solo UIViewController, ma dall'altro verrà delegata l'intera paginazione al view controller.

In particolare questo UIViewController accetta come variabile di inizializzazione un array di pagine e in base allo stato della UIPanGestureRecognizer deciderà se passare alla pagina successiva o interrompere l'animazione;

5.1.7 5. Il Collider

Come accennato al punto 3 ogni le CardView hanno dei comportamenti fisici definiti da UIKit Dynamics, in particolare per conferire un effetto di collisione viene usato un **UICollisionBehavior** e implementato come segue:

```
let itemsCollisionBehavior = UICollisionBehavior(items: page.views)
```

In questo modo ogni ogni oggetto con questo comportamento riuscirà a scontrarsi con gli altri oggetti simili.

Capitolo 6

Autenticazione

Per la progettazione dell'autenticazione si è voluto ricorrere a qualcosa di già pronto per velocizzare i tempi di sviluppo e quindi evitare di progettare e implementare un sistema complesso come la gestione di utenti non autenticati (Guest). Cercando in rete abbiamo optato per l'utilizzo di **Firebase** [12]

6.1 Firebase Authentication

Firebase è una piattaforma di sviluppo per applicazione mobili e web acquisita di Google nel 2014. Uno dei motivi fondamentali per cui è stata scelta è la grande quantità di servizi utili come analisi dei Crash, tracking della navigazione degli utenti, DynamicLinks (si veda sezione ??) e ovviamente l'autenticazione.

6.2 Tipologie di autenticazione

Esistono 3 tipologie di autenticazione come definito nel requisito, ognuna della quali è un'estensione di quella precedente: un utente che apre per la prima volta l'applicazione verrà subito registrato come utente anonimo e quindi in **Trial Mode**, se poi decide di effettuare la registrazione può

inserire il suo numero di telefono o la sua email e attraverso Firebase l'utente anonimo evolverà in un utente con più informazioni, in **Signed Mode**.

Successivamente ci sarà nell'applicazione una sezione specifica dove l'utente potrà inserire nuovi dati come nome, cognome e data di nascita o semplicemente potrà linkare un social media come Facebook, Google o Instagram. In questo caso l'utente evolverà nuovamente e arriverà nella **Pro Mode**.

Il pacchetto Firebase Auth infatti include tutte queste funzionalità e attraverso delle semplici API è possibile utilizzarlo come segue:

Inizializzazione di Firebase nel file di inizializzazione dell'applicazione ossia l'AppDelegate.swift

```
FirebaseApp.configure()
```

Creazione di un utente anonimo

```
Auth.auth().signInAnonymously() { (authResult, error) in
    // ...
}
```

Collegamento di un account anonimo con numero di telefono o email

```
// L'oggetto user è sempre disponibile in firebase e
// salvato nel keychain di iOS
let user = Auth.auth().currentUser

// Creazione dell'oggetto credenzial da collegare
// all'utente attuale (anonimo o non)
let credential = EmailAuthProvider.credential(withEmail: email,
    password: password)

// Linking dei due account
user?.link(with: credential) { (authResult, error) in
    // ...
}
```

Il linking con un qualunque social media è simile all'esempio riportato qui sopra, cambia soltanto il Provider delle credenziali

Capitolo 7

Dynamic Links

Un'altra interessante funzione di Firebase sono i **Dynamic Links** ossia dei semplici URL generati dalla console o generati direttamente dall'applicazione che consentono l'iniezione di un altro url a cui l'utente verrà reindirizzato.

Questo permette agli utenti di aprire direttamente l'applicazione in questione attraverso lo smartphone o il Computer attraverso il browser, ma trasportando dei dati utili all'azienda.

In particolare nel mio caso mi è stato chiesto di implementare un sistema di inviti, per cui un utente può invitare un altro attraverso questi links.

Per risolvere il problema abbiamo prima implementato i Dynamic Links

Conclusione

Bibliografia

- [1] Urbana smart solutions srl. <https://www.urbanasolutions.net>.
- [2] Proto.io. <https://proto.io>.
- [3] *CardView*. Definiamo CardView una vista rettangolo con un border radius e del contenuto di testo e immagini variabile.
- [4] Apple Inc. Universal links. <https://developer.apple.com/ios/universal-links/>.
- [5] Apple Inc. Uiviewcontroller. <https://developer.apple.com/documentation/uikit/uiviewcontroller>.
- [6] Apple Inc. UINavigationController. <https://developer.apple.com/documentation/uikit/uINavigationController>.
- [7] Apple Inc. UIKit. <https://developer.apple.com/documentation/uikit>.
- [8] Apple Inc. Uikitdynamics. https://developer.apple.com/documentation/uikit/animation_and_haptics/uikit_dynamics.
- [9] Apple Inc. Uiwindow. <https://developer.apple.com/documentation/uikit/uiwindow>.
- [10] Soroush Khanlou. Introdotto nel 2015 alla nsspain conference.
- [11] Airbnb Inc. Lottie. <https://github.com/airbnb/lottie-ios>.
- [12] Google LLC. Firebase platform. <https://firebase.google.com/>.

Elenco delle figure

2.1	Navigation controller scheme	6
2.2	Presentazione di un ViewController tramite push	6
2.3	Presentazione di un ViewController tramite modal	7
2.4	Schema dell'implementazione di UIDynamics	9
2.5	Rappresentazione dell UIWindow	10
3.1	Il Coordinator Pattern	16
3.2	Esempio di coordinator pattern	16
5.1	CardViews modulari	31