TESINA FINAL DE MASTER

Design and implementation of a contributive photo-sharing mobile application for iOS.

(Diseño e implementación de una Aplicación móvil para compartir fotos de manera contributiva).

Studies: Master in Science of Information and Communication Technologies (MINT)

Author: Roberto Arreaza

Tutor: Josep Pegueroles

Year: 2015

General Index

Collaborations

The following thesis project has come together in collaboration with:

- Escola Tècnica Superior d'Enginyeria de Telecomunicació de Barcelona (ETSETB), a faculty that belongs to the "Universitat Politècnica de Catalunya" (UPC). Website: www.etsetb.upc.edu.

Acknowledgements

I would first like to take this opportunity to thank the UPC Telecommunications School, as well as mi project director, professor Josep Pegueroles, with whom I shared a nice work experience during the development of this project.

I would also like to give a special mention to all the friends, colleagues and acquaintances who have helped me, may that be with bits of good judgment, advice, technical help, motivational words or any other forms of support during this endeavor: Pierluigi Cifani, Alan Osers, Juan José Coello, Omar Alayón, Kebby Belisario.

Finally, to the person that has helped and shaped me to become who I am today, and to whom I owe in one way or another all of my success stories: my mom. She who has showed me the way through the path of virtue and excellence at all times, always leading by example. I also need to mention the rest of my family: siblings, grandparents, aunts, uncles and cousins, as well as my godmother, who in spite of the geographical distance that separates me from them, have always been there for me, both in my personal life as well as my projects. I could not finish writing this without leaving a special mention for them as well.

As a final note, to all people that have directly or indirectly contributed to the realization of this project and have thus been a part of it. Thank you!

Abstract

Throughout the following document, the concept for a photo-sharing mobile Application will be explained. Within its pages, the document demonstrates the development of an Application to create "social Events", where users can create digital collaborative spaces to share their pictures, with other people who take part in the same event.

The proposal is to be achieved by using the latest available technological tools for iOS development, as well as server Backend development.

The fundamental requirements and specifications will be presented, as will all the considerations needed from the final user's perspective, and all the technical issues regarding the design and development of the software Application.

Parting from the initial presentation of the concept, going through the research phase and inspection of the required technologies, getting to the final design and implementation of the App, each stage is approached throughout the document in a way to give the reader a clear idea of the underlying motivation and philosophy adapted in the Project. In this manner, the reader is able to keep in mind this information, and will therefore be able to understand and justify the more practical and concrete sections of the Project.

Finally it –the document- will present an overall evaluation including the initial goals set in contrast with the resulting software product. This is one in order to offer a list of possible future improvements that could in turn result in a better user experience, always in accordance to the initially stated philosophy and general purpose for the Application.

1. Introduction

The Mobile Apps market remains in continuous growth (Gartner, 2013). Communication and interactions via mobile electronic media has experienced a big "boom" due to the proliferation of Applications such as Whatsapp, Facebook, Instagram, Twitter, Snapchat or Google+, among many others. This phenomena has had such a deep impact, that in a study conducted by comscore (www.comscore.com), referenced in a Tech Crunch article (Perez, 2014) it states the following:

"…Combined with mobile web, mobile usage as a whole accounts for 60% of time spent, while desktop-based digital media consumption makes up the remaining 40%…"

Since mobile phones are now the main tools of engagement for most internet users worldwide, and it also happens to be the de-facto instrument for taking photos as well (due to an increase in the quality and miniaturization of cameras in mobile devices), it is no surprise that this combination of circumstances has also changed the way people share their photos, which in turn has led to the emergence of big market for Photo Applications.

The current photo Applications market has been dominated by Apps such as Instagram, Snapchat, Facebook and some others. Every Application came to serve a specific need or purpose and consolidated into its own niche of the global market. For example, Instagram came along with its photo sharing broadcast model, based on users following other users and getting the content shared, pretty much like tuning up to a specific channel composed of a user's content, as well as bringing filters to make photos look very artistic, with very little effort.

Snapchat on the other hand, came along as a "visual communication" tool, based on communicating with other users via photos, which are not saved or persisted by default, thus making it a very useful tool and gave the phrase "a picture's worth more than a thousand words" a very practical sense in what seems to have become the "internet era".

Every platform ends up being very useful for curtains kinds of tasks to address a variety of needs. There is a problem, however, when people who come together in a real-life social event, and every person takes their own photos and videos to post them into their own social media

channels or accounts. As it happens, many people participating in these events may like -or want to have even- some of the content generated by other participants, and because of the individual-centric nature of the existing social photo sharing applications, this task is not easily achieved by conventional Apps. This is why many people end up asking others for "that nice picture from the other day", or creating Whatsapp groups for everyone to put some of the pictures for everyone in the group to have, or other alternative solutions that may end up being very cumbersome for many people.

This is why a need has been detected for creating an Application that solves this predicament. The Application proposal of this project is named "inJoin", and it basically consists on a platform specifically designed for creating a shared space for people participating in social events in real life (called Moments), in which they can share all the photos derived from this event in a single place, and available only to everyone who participates in such event. This way the photos that everyone takes are easily accessible and available for all participants, thus removing the need for looking into external cumbersome ways of sharing this content with the right people. With this goal in mind, this project aims to design and implement such Application, considering the best and easiest possible way to make this tasks as intuitive and easy as possible.

1.1 Objectives

i) General objective

As the general objective for this project, it is proposed to perform the design, implementation and real-life testing of a mobile Application concept, meant for the iOS platform from Apple Inc., which consists on an event-based crowd-sourcing platform for gathering and sharing photos among friends, to be accessible from anywhere and anytime, thus creating a reflection in the digital world of people's real-life experiences.

In addition to the main objective of this project, there are also a series of **specific objectives**, which contemplate each of the different stages to go through, in order to achieve the creation of the Application concept and are therefore important to enunciate:

1. As an initial phase, there's the evaluation stage. Here, the technologies available by Apple Inc. intended for iOS (Apple's mobile operating system) development are studied in detail to get well acquainted with the system and tools to be used, as well as any additional tools and technologies of relevance.

2. Following next, and taking into account the fact that the App concept requires an underlying distributed system to support the core functionalities, it will be proceeded to design the appropriate system architecture that will make possible the implementation Application concept, as well as its scalability.

3. Once the appropriate architecture is defined, any additional technologies, tools and services derived from it, which are also necessary for its implementation, must be studied in order to become familiar with their appropriate usage for the specific purpose of the project.

4. Afterwards, the definition of the basic functionalities of the application must be specified, and a first draft of the App design be done, depicting how the final result should look and behave like. From this point, it is then safe to establish an appropriate

Data model for the Application data that adapts to the final functionalities and design. Furthermore, this stage allows for defining the required API services for the Application to interact with the system in order to perform its functionalities properly.

5. The next important task is the development of the "backend" supported by the system architecture that has been defined, and using the API services definition as guidelines to what is to be programmed at this stage.

6. Once the backend is finished, the Application itself must be developed. The guidelines for this endeavor are the design and functionalities defined before, and it must then integrate and interact with the backend to perform some of the tasks required. The development of the core concept is proposed to be done in different stages: in the first place, the client backend API integration must be coded, followed by the development of the main or home view of the App. Then building the "viewing" (read-only) of existing collaborative Events and their photos in a list, as well as the ability to inspect the details of events. Afterwards creating the possibility of adding photos to Events, followed by coding the capability of creating new events. Then the ability to add other users as connections for participating in Events, and finally any other specifications defined in the App's functionality and design that are outside the core functionalities of the project.

7. Deploying the system architecture to be accessed via the Internet, to be able to use the Application from anywhere by using an iOS mobile phone.

8. Performing several Beta tests of the Application in real-life events, and fix bugs and perform any test-derived improvements to it.

1.2 Context for the project

As he had been doing since a couple of years back, the author of this project decided that, in order to celebrate his birthday for the current year, he would set out to inviting all of his close friends, colleagues and acquaintances to celebrate it at the beach, in a manner of a picnic where everybody would bring some food and drinks, while some music, volleyball, cushions and blankets would be already there, welcoming them into the event.

The birthday event went great, and everybody had a good time. Nevertheless, something curious was happening: every attendee –as it's the normal thing to do- was posting the pictures he or her had taken on Instagram, Facebook, Snapchat, and other social media platforms independently. The result from the sum of this individual actions was that a lot of pictures from the same social event were being posted scattered in the various social networks for each participant of the birthday, and event though all of those photos had a shared context and were linked by the real-life event where they were taken, this fact is not reflected in the digital world when they're posted online individually. The result is that even though everybody shared this life experience together, they did not share it in the same way in their social platforms, thus creating one of the most common –and somewhat annoying- problems that many people facing the same situation encounter: the "hey, can you send me the pictures you have from the other day?" problem. As it turns out, it is not uncommon for others to want to have access to other friends and participant's photos, maybe simply because they "took a good picture that I did not capture with my own phone", or maybe as the organizer I'm interested in as many pictures from an event as I can get, to keep as memories of that birthday, wedding, party or holiday.

This is how the main motivation and idea for creating inJoin came along, because creating the "birthday pictures" Whatsapp group simply seemed unnecessary nowadays that we have the technology to do something better about this problem, and after an unsatisfactory search for an Application that would satisfy that specific need.

1.3 Structure of this document

From the introduction, this document explains the motives behind the development of the Mobile App project and clarifies the development environment in which the proposal is intended to come true. Additionally, it defines the main goals and objectives set out to fulfill in order to get to the final result: the development of the Application concept.

As a next step, this document seeks to provide some context to the project, by presenting a bit of the background behind the initially detected customer needs that were not properly addressed by existing Applications.

Following next, it sets out to provide an explanation of the employed technologies, thus explaining all the tools that play an important role throughout the implementation of the project, with the aim of providing readers with the information needed for a better comprehension of the more technical sections of the project.

Right afterwards, in the section named "development environment"), lies the explanation regarding the work environment where the most important steps of development take place, as well as related tasks such as registering and distributing the final product for beta testing purposes, among other processes of practical relevance.

Next up is the "User Manual" section, where the Application concept is explained from the perspective of the final user, thus making a thorough navigation of all use cases and sections of the App.

Once readers get a good idea of what the final product should do, look like, and how it should work, in section "System Architecture", this document explains the main architecture design and implementation that is required to build such a working product.

Following next, in the "Developer's Manual" section of this document, it proceeds to perform the more technical analysis of the Application itself. It lays out functional requirements in contrast with the inner design, modeling of data structures and final code implementation that characterize the resulting product.

To finalize, this document presents the conclusions and a roadmap to future implementations about the work done, in contrast with the initial objectives, to define a possible future direction for this project. This is then followed by the Appendix and bibliographical references.

2. Photo sharing iOS Apps

2.1 Background

As a regular Facebook and Instagram user, the author of this document is very accustomed to sharing pictures and creating albums for safekeeping in the cloud. This kinds of services make it really simple to accomplish tasks such as capturing specific moments people would like to keep and show the world, or simply for a very personal use and just keep all pictures well organized in a single place, without significant fear of loosing data, which may be accessed anywhere there's an internet connection.

Although existing apps work greatly for these very generic purposes, they fall short when it comes to a –surprisingly common- usage case.

Most people who participate in some form of social event or gathering tend to take pictures of their experiences during this event, so they may then share it via social media. And sometimes that's enough, but more often than not, when the people participating in such events are well acquainted with each other (friends, colleagues, family, etc.) they tend to find themselves in a situation where one person takes one or more pictures that are interesting for others, and so they get asked "can you send me the pictures of yesterday?" which usually happens via email, Whatsapp or just an old-school pen drive containing the media desired by many.

This exact same situation occurred to the author of this document, during his birthday, and in the process of trying to figure out a simple and efficient way of being able to share pictures of social events with the people who actually participated in them, there was no satisfactory service that would accomplish that without some form of unnecessary hassle.

This is the very reason why the inJoin App project started: providing people with a hassle-free and simple way of sharing common experiences with other people, in a single place with only this fundamental purpose in mind.

2.2 State of the Art

When it comes to similar mobile Applications (updated until 2015), there a many Applications intended to share photos for multiple purposes in the iOS App Store. Such great diversity makes it harder to find exactly what one is looking for, but Many similar apps can be found with very interesting functionalities.

The most obvious App choices with similar features and purpose, due to popularity would be Facebook, Instagram, Snapchat or maybe event Whatsapp, but there's another interesting Application called Banjo that has increased its popularity and looks also promising for its intended purpose as well.

Everybody knows Facebook, and so it seems like a good place to start. It is the ultimate content sharing social network and one could state it set the standards for what a social network in the age of the Internet should be. It has so many features that people may sometimes be confused by what they're offered. In it's core, it allows people to connect with other users and share content (most importantly pictures, for the sake of this comparison) and organize them into albums visible to others. It is not only a website, but a mobile Application available for any platform. If one wanted to use it to solve the problem presented in this document, one should know that it also allows for multiple users to make a collaborative album (Southern, 2013). The key problem here lies on the fact that since Facebook is such a general-purpose application, its focus cannot really be on this simple feature that one might find in it, and therefore it's usability for this purpose becomes quite cumbersome, and lacks some other interesting relevant information about the concept of an event, that does not necessarily applies to albums (such as start and end times, for example). This makes Facebook a less than ideal choice for the solution to the problem.

Instagram is all about sharing specific moments, making pictures beautiful by using image filters, and posting it to followers on the tap of a button. It cannot be used as a contributive tool since it's mainly a "broadcast channel" others can tune to, and just recently is that they added support for one-to-one picture sharing (Instagram Inc., 2012) Nonetheless, it has a very interesting feature called photo-map, where

geo-tagged pictures can be seen in the context of location, which could also be interesting for an events application.

Snapchat came to be highly popular with it's spontaneous picture-based Communication model, where users send other users pictures that have a limited lifespan, with the purpose of conveying a message in a graphical way. The fact that pictures are not meant to last and stay there forever, and the focus on communication, makes it unsuitable for solving the kind of problem that this document tries to solve.

Whatsapp is more that just a chat Application, it has become the de-facto communication tool for millions of users worldwide, and the fact that it allows for both group chats and sending pictures, makes it into a suitable choice for trying to share pictures with a closed set of people. This may very well be in fact the most common alternative people usually rely upon when they want to share privately pictures with a group of people, but the problem is that this is done under the context of a "chat", which means there's no contextual information determining that those pictures were shared as part of a social event, which makes it not a good solution for the problem at hand, although it can get the job done.

This is why an Application with a focus centered on the concept of "events" is necessary in order to suit the particular needs explained.

An increasingly popular "Banjo" Application can be defined as a "location-based social aggregator" which means it's not really an independent social network in of itself, but an App that allows people to connect many of their already existing social networks, and if some location information (such as the Instagram geo-tagged pictures) is posted, it automatically gives the aggregate of content, the context on location and time, as their internal algorithm considers it appropriate. The result you get is an aggregate of information of twitter, Instagram, Facebook users with some location and time context. They even auto-generate what they consider are "relevant" events from the information they handle, suited for specific users. As an example, I've received notifications about fires in Spain, and recently about an armed robbery at the famous Rambla in Barcelona, which Banjo generates based on tweets and Instagram feeds regarding this events. It seems like a great tool for automatically and intelligent events, but it lacks the ability to be able to create your own events privately among a specific set of users. For this

reason, it is not really suitable to solve the problem depicted in this document in an appropriate way.

When one navigates through the Photo- sharing Apps world, it is then possible to realize that even though there are many Applications with several interesting features, no single App really addresses the problem in a satisfactory way, and so a possible need to supply this kind of utility into the iOS Apps market.

3. Employed Technologies

A great diversity of technological tools is used all throughout this project. Such tools allow for the development to come true, and facilitate many of the implementation tasks that are key in implementing certain features. The most important of these technologies are presented next:

3.1 Objective-C

Objective-C consists a programming language that, as it is very well insinuated in its name, is "object-oriented", in the same way most modern programming languages such as Java, C++ and others are designed.

In a book regarding the fundamental principles of Objective-C, Rich Warren (Warren, 2011) defines the language as:

"…Objective-C is a small, elegant, object-Oriented extension of the C language…"

By being an extension of the C language it's implied that there is total compatibility with any C written code, due to the fact that Objective-C was constructed *on top of* C. Consequently, this allows for the usage of any C or C++ written library, with the additional features that Apple's own Cocoa Framework has to offer for iOS development, thus granting a vast number of advantages to any developer that wishes to be integrated in to the world of iOS mobile development.

As it has been well mentioned before, it consists on an object-oriented language, although when contrasting it with other traditional languages such as C++ or Java, it differs on several aspects. For this reason, any given Java or C++ developer could have expectations or misconceptions when coming to Objective-C that may not be present in the language. This is mainly due to the fact that Objective-C was designed based on *Smalltalk*, which according to the official Objective-C reference from Apple (Apple Inc., 2012), was "…one of the first object-oriented programming languages…".

This section does not intend on being a complete Objective-C reference; instead it aims towards highlighting the most significant and peculiar differences and features available, that were useful in the

development of the project. For a more complete and detailed reference, readers are encouraged to consult the official document on the website of Apple, Inc. "The Objective-C programming language" (Apple Inc., 2012).

The first relevant concept to keep in mind is the general design philosophy for programming in Objective-C, better known as MVC (Model-View-Controller). This paradigm indicates that classes created for an Application may be classified into three different categories: models, views and controllers.

Models conform the classes and mechanisms used for the abstraction of the data that's handled by the Application (App), and they're the basic building stones over which the rest of the App is constructed.

Views on the other hand, are the classes and elements displayed to the final user, thus conforming the UI (User Interface) of the Application. They compose the "visual output" or final representation of the data that's handled at a given time, following also a specific logic of behavior.

In regards to the "specific logic of behavior" mentioned above, this is what Controllers are meant to accomplish. They are classes that serve as mediators or managers handling models and views, and they *control* what's being displayed on screen (views) and the manner in which it is shown.

This fundamental paradigm involves a variety of important advantages, among which is found the separation of the programming logic and the Graphical User Interface (GUI), thus facilitating code writing; without this, code readability would rapidly become cumbersome. In a similar manner, it facilitates the re-utilization of code of classes by reducing their inter-dependence and encapsulation, which makes code easily reusable. Furthermore, it commands for the separation of the data model from the final UI, thus allowing for reusing these elements and simplifies greatly their maintenance.

The first thing that stands out to a programmer coming from languages such as Java or C++ when looking at some Objective-C code, are some radical syntactical differences to which developers must get acquainted.

To exemplify the previous statement, one can observe the declaration of a class in Objective-C (Apple Inc., 2010):

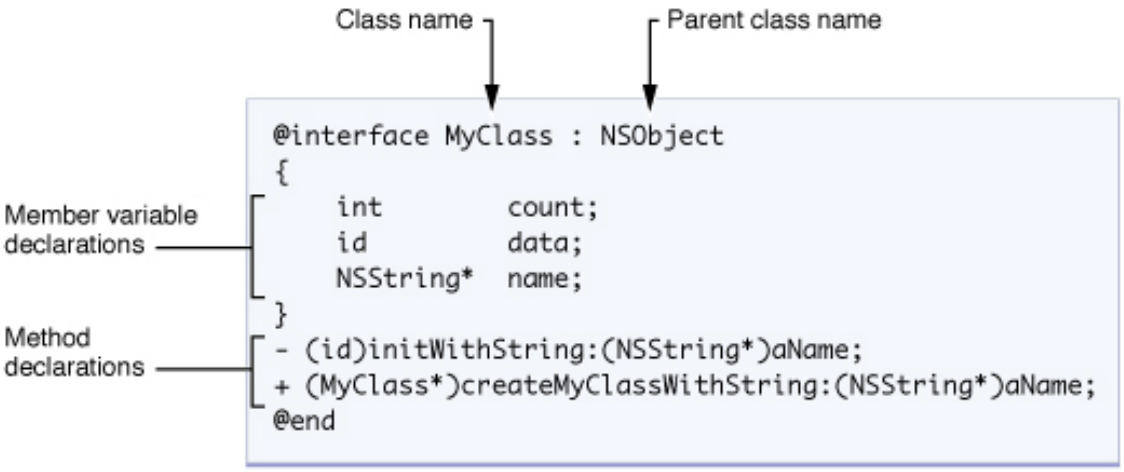

Figure 3.1: Class declaration in Objective-C. Source: https://developer.apple.com/library/mac/#referencelibrary/GettingStarted/Learning_Objective-C_A_Primer/_index.html

On the previous example, some relevant aspects are worth noticing. Firstly, the class is declared (or rather its interface, normally declared in a .h file) via the *@interface* and ends in *@end* statements as opposed to the traditional "{}", as it could be expected. Curly brackets are reserved for the declaration of *member variables*, and right after them one usually finds the methods of the class (more on that subject shortly). Additionally, the class name MyClass : NSObject can be observed. The notation would be the Java equivalent of MyClass extends Object, so the ":" are employed for specifying the parent class.

On the other hand, the declaration of the methods inside a class follows the following format:

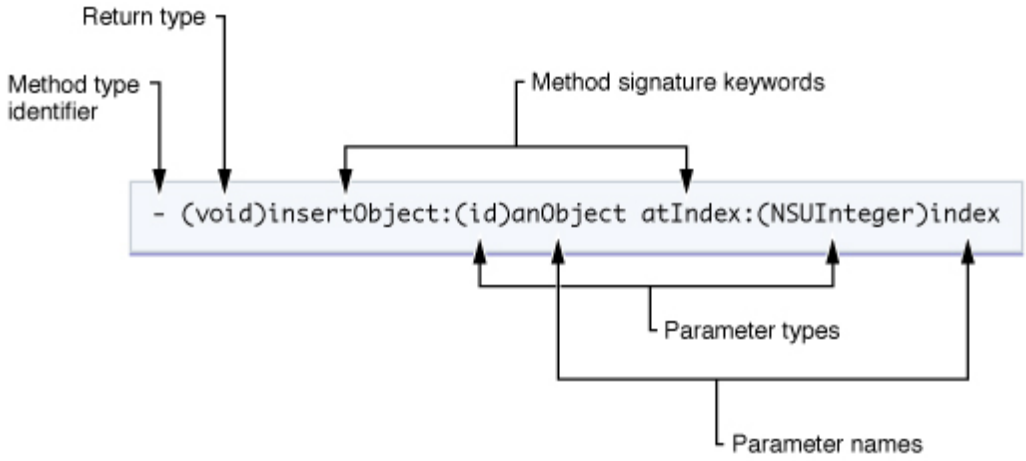

Figure 3.2: Declaration of a method in Objective-C.

It can be observed that the method name is preceded by a symbol, which can be either "+" or "-", and determines whether the method is a class method or an instance method, respectively. Similarly, it's possible to verify that the return type of the method is written among parenthesis right before the method name.

Method names are not as they're conceived traditionally, meaning that instead of MethodName(type1 param1, type2, param2, …) parts of the name are alternated with the parameters; this format facilitates the description of every parameter inside the name of the method itself, which represents an advantage on legibility and code comprehension in most cases. In this manner, the entire method name results (for the example provided in figure 3.2) insertObject: atIndex: where it is immediately known by just from inspecting the name, that after *insertObject:* follows an object (*anObject*), and after *atIndex:* should then follow an index parameter (*index*).

The so-called ***messages*** are the method execution calls sent to specific objects, and conform to the same format used in the declaration of the method itself. Nevertheless, the syntax for calling them varies, so invoking the example method upon an object *myObjectCollection* (assuming anObject and index belong to the right types), would result in something like:

```
[myObjectCollection insertObject: anObject atIndex: index];
```

The call is contained in brackets and the object where the method is invoked goes at the beginning, followed by the method + parameters pairs required for the call.

On another note, a fundamental concept in the Objective-C development for iOS, which is used in virtually any App, is the concept of ***delegation***.

Delegation basically consists on commissioning or assigning the fulfillment and execution of a specific task to an external class. This external class may also take any kinds of parameters in order to properly perform its duties and is (most likely) unknown by the delegating class. The task must therefore be sent in a standardized format known by any given class, in order to properly interpret it, and perform its execution correctly. Such well-known format is what Objective-C defines as a ***protocol***.

A protocol fundamentally consists on a series of methods, that may very well be optional o mandatory, and which implementation is left empty by the time of their declaration.

An protocol example could be:

```
@protocol myProtocol <NSObject>
- (void)firstTaskToDelegate: (id) anObject;
- (void)secondTaskToDelegate: (id) anObject;
@end
```

The important thing to note on the example is that inside the protocol by the name MyProtocol (inheriting from NSObject, which is also a protocol), two tasks are defined to be delegated.

In this manner, any class implementing the MyProtocol protocol (implying the implementation of the methods specified in MyProtocol), can be called from any other class to execute the tasks (methods) solicited.

It is for this reason that delegation consists on making calls to an external class that implements a determined protocol which is well-known by the class making the call, in order to provide the external it with the opportunity to manage a certain specified task.

The key advantage of this is immensely important, due to the fact that it allows for decoupling classes and encapsulating functionalities, thus allowing to get further away from the common issue of class inter-dependence, which in turn translates into better object-oriented code and more and better class reusability.

Another important subject in iOS is related to Memory Management, which involves the mechanisms Objective-C has to offer to control the usage of this relevant hardware resource which, for the specific case of mobile devices, it's usually very limited and a subject that must be addressed always in the creation of any App. Objective-C offers 2 mechanisms for memory management (Apple Inc., 2012) ARC and MRC (or MRR). Before there used to be Garbage Collection for OSX, but it was later deprecated is OSX Mountain Lion in favor of ARC (Apple Inc., 2015). The first mechanism (ARC) is known as *Automatic Reference Counting*, and was added from iOS 5 onwards. It basically consists on delegating the task of keeping track of all retain/release calls for all objects where memory has been allocated onto the compiler, and allows it to manage such tasks automatically. The MRC acronym stands for Manual Reference Counting, and it's based on the idea of retaining and releasing objects manually, so these memory management tasks are then delegated to the developer. This makes the resulting code a lot denser and more prone to memory access bugs. Even though Garbage Collection is one of the most popular and comfortable methods for the programmer, since it does all memory management automatically, and it is present in most modern languages such as Java, Apple has taken the decision (mostly based on hardware and performance constraints) to deprecate it in order to favor the ARC approach for both iOS and OSX (the latter was the only system where Garbage Collection existed).

This project is entirely written under ARC, as the most logical way to keep code clean, minimize memory-related bugs and to have as a result a more concise and readable code.

Lastly, it is important to mention, due to its frequent use throughout the project, the concept of ***Categories***. Categories are a mechanism used by Objective-C developers to "extend" the functionalities of a particular class, without modifying the original class, or having to write a *subclass* either. This means that categories may implement additional methods

that will be added to the original class and be available during runtime. The syntax of a Category is similar to:

```
#import "myClassName.h"

@interface myClassName ( myCategoryName )

// method declarations to be added to the class

@end
```

The slight difference in notation lays in the fact that right after the name of the class, instead of typing the inheritance information, developers must write (between parenthesis) the name of the category. Any method added after this declaration belonging to the category will then belong to the original class, thus achieving the extension of a the functionalities of a class, without any need for subclassing it. The only disadvantage revolves around the fact that Categories do not allow for additional member variables, which is perfectly valid when constructing a subclass. Nonetheless, in many cases, member variables are not required, and this is where Categories become very useful.

Objective-C is a dynamic, practical and efficient language that may not converge meet in many criterions with traditional programming languages, but its usefulness and features are undeniable. Along with the Cocoa Framework from Apple Inc., it provides developers with some very valuable tools that allow them to undertake tasks that would normally be complicated, and solve them in simple ways, thus making it easier to code mobile Apps of great value and quality, such as the one presented in this project.

3.2 Ruby on Rails

"A dynamic, open source programming language with a focus on simplicity and productivity. It has an elegant syntax that is natural to read and easy to write"

That's how the official website (https://www.ruby-lang.org/en/) defines Ruby.

It is an interpreted programming language that was designed and developed around the 90's by Yukihiro Matsumoto in Japan (Wikimedia Foundation Inc., 2015).

Matsumoto combined different principles from existing languages, such as Perl, Smalltalk, Eiffel, Ada, and Lisp, to give form to a completely new language, whose main focus revolves around the principles of conciseness, consistency and flexibility (Matsumoto, 2000) resulting in a language that is very programmer-friendly, and highly readable and understandable code (Gillette, 2007), when compared to other language alternatives.

Since its creation, Ruby has become a highly popular programming language for many other reasons. For instance, it is completely free of charge, free to use and to copy, modify and distribute.

On the other hand, it is true for modern languages that any programming language is only as strong and popular as its frameworks, and Ruby is no exception. As stated in Ruby's official page (Ruby Community, 2015), the language's increasing fame is highly due to the popularity of its web framework "Ruby on Rails", that allows Ruby programmers to develop web applications.

Ruby on Rails (or just *Rails*) is the most popular web framework available to this date for Ruby. It is open-source, and was created by David Heinemeier Hansson, who extracted Rails from the web application project called Basecamp, a SaaS (software as a service) project collaboration tool (Grimmer, 2006) conceived as a web application that was developed on top of Rails.

Rails follows the design philosophy of Model View Controller (MVC) in the same way that iPhone Applications do (for further information refer to

the section on Objective-C), thus making it easier and more compatible to use Rails as the Backend technology for an iPhone Application, as it is an "easier" fit. This implies that for a Rails Application, developers usually create Models, which are typically associated to a model class (Ruby source file) and a Database table. The model class determines the behavior of the model defined, whilst the database table linked to this class holds all the data that provides instances of this class with an object state, and makes model instances "query-able" in a very convenient way, and with a powerful and full-featured engine behind it.

Controllers in Rails are defined as ruby source files in charge of processing web requests (HTTP requests for the Web application) along with any provided data embedded into the request, and return and configure a specific response, typically in the form of a View (resulting HTML file, whose behavior and configuration is ruled by the View source file).

Views are not required for the scope of the present project, as Rails will be used to build a REST API backend, which is not intended to output any end-user visible documents (such as HTML pages), but instead return petition results in the form of JSON-formatted data, intended for the client app to handle. Therefore the most important pieces on the Backend REST API are the models and Controllers.

Nevertheless, there is still one key piece of the Rails framework that is vital for the Rails architecture to work properly: the routing engine. The routing engine of Rails is configured by developers by means of the *routes.rb* ruby source file, and contains all routes for REST resources available for access from the backend. It essentially specifies the REST API endpoints that clients may reach when sending requests, and is also in charge of mapping this requests with the appropriate controller that must process them (to finally generate and send a response back to the client).

By default, Rails creates some basic API endpoint for the RESTful resources defined, as a raw CRUD (Create, Read, Update, Delete) interface. For some cases, this is enough, but for the majority of the API services definition, this had to be customized to the project's needs. All of the customization lies also within the *routes.rb* source file, as would be expected.

For further information on how to get started with a new rails (installation, initial steps for creating a sample application, and some customization of models, views, controllers and routes), readers may refer to one of the Getting Started guides (Rails Guides, 2015), which holds the official and comprehensive guide to dive into the Rails, and was consulted and used by the author of this thesis project to get the basics of using the Ruby on Rails framework.

3.3 Cocoapods

The official website for Cocoapods (https://cocoapods.org/) states the following:

"Cocoapods is the dependency manager for Swift and Objective-C Cocoa projects. It has thousands of libraries and can help you scale your projects elegantly."

In short, Cocoapods manages an entire set of libraries and external code that can be used and easily integrated with any iOS projects, without the very common problems of handling all configuration, set-up and version managing tasks that developers who choose to handle libraries and external code by themselves tend to suffer.

It is very similar to dependency managers such as the *gem* utility for Ruby, or the *pip* utility for Python. Developers usually find a specific Cocoapods project library (known as a *pod*), and use a special file called Podfile (similar to Gemfile in Ruby), which defines the names and version of the libraries to use, in a format similar to:

```
pod 'SomePodName', '~> 1.2.0'
```

Where *SomePodName* is the name of the project to integrate and use from within the iOS App, and '1.2.0' defines the version to use from this pod. In this way third-party code may be easily integrated into any iOS project.

For this project, for the iOS App client, some tasks usually require third-party pods to perform specific tasks, and Cocoapods is thus used to facilitate such endeavors.

For a more detailed explanation on how to integrate pods into an iOS project, the reader may refer to the official documentation (Cocoapods, 2015).

4. Development environment

In addition to the variety of technologies use throughout the lifespan of this project, it must be noticed that the development tasks themselves are carried out under very specific development environments. For the case of iOS and OSX, the main development environment is called XCode. Additionally, other environments are used for all Application management tasks, as well as Beta Testing and App Store deployment endeavors.

4.1 XCode

Any software developer, at any level of expertise, must have at his or her disposal a variety of tools that make easier the coding tasks at hand, independently of the language in which development takes place.

Modern programmers take for granted high level features, such as syntax highlighting, auto-completion, one-click navigation for inspecting specific pieces of code, text search, among others that dramatically improve task resolution efficiency of programmers nowadays.

This idea applies to iOS and Mac OSX all the same. Programmers who develop for Apple Inc., are used to having a set of top-of-the-line technological tools, with features that may sometimes exceed the normal expectations for other environments. This is greatly due to the go-to development tool for iOS and OSX: XCode.

XCode consists of on an IDE (Integrated Development Environment), specially designed for developing applications in languages such as C, C++, Objective-C, Objective-C++, and most recently, Swift, for the iOS and OSX platforms. XCode is composed by a set of tools that facilitate the development, testing, maintenance and UI design processes, as well as many other additional tasks. Regarding a complete list of XCode features, one may find too many of them to get into details, although there are some key functionalities that are very popular, and have been widely used in the implementation of this project (Wentk, 2011):

- A source code text editor, which includes static code verification, context auto-completion, and a fast dynamic clues and tips engine.
- New project templates with base source code already in place for common application design structures.
- A powerful UI design tool known as IB (Interface Builder), with a wide variety of classes with the specific purpose of building user interfaces, which allows for linking UI components to pieces of code logic such as objects, methods and actions.
- A full-featured set of debugging tools.
- Simulators for both iPhone and iPad Applications, for different OS and hardware versions, facilitating much of the development without the need for the physical devices at hand.
- Visual tools for the design of model classes and integration with Apple's own data persistency framework: Core Data.
- Instruments, a series of software utilities that allow developers to measure several app performance metrics, such as speed, memory usage or CPU, that may be actively measuring performance while some live code is running.

For a more detailed and complete list of XCode features, readers may refer to the book by the name: XCode 4 (Wentk, 2011).

In order to get a more practical idea about XCode's usage, the typical basic steps to create a new Application for iOS in XCode 6 will be described next. This will allow for the introduction to explaining some important concepts about XCode itself, and iOS applications in general.

The first thing to do is to download XCode at the official Apple developers website: https://developer.apple.com/xcode/. This may also redirect Mac users to the Mac App Store for the download to take place, depending on what OSX version is installed on the Mac computer.

Once installed, XCode is ready for development. An important thing to note, consists on the fact that in order to publish an app to the App Store, or for the development and testing to take place in physical devices (instead of just using the simulators) with iOS, a set of "Provisioning Profiles" and "Developer Certificates" are required, and they can only be obtained through the acquisition of official OSX or iOS developer accounts at https://developer.apple.com/. These certificates and provisioning

profiles are used to sign the apps, and validate and identify developers to official Apple accounts.

As soon as XCode opens, the first screen shown displays a series of options guiding developers about what kind of project to start. For the case where creating a new App is desired, developers should choose "create new project", as follows:

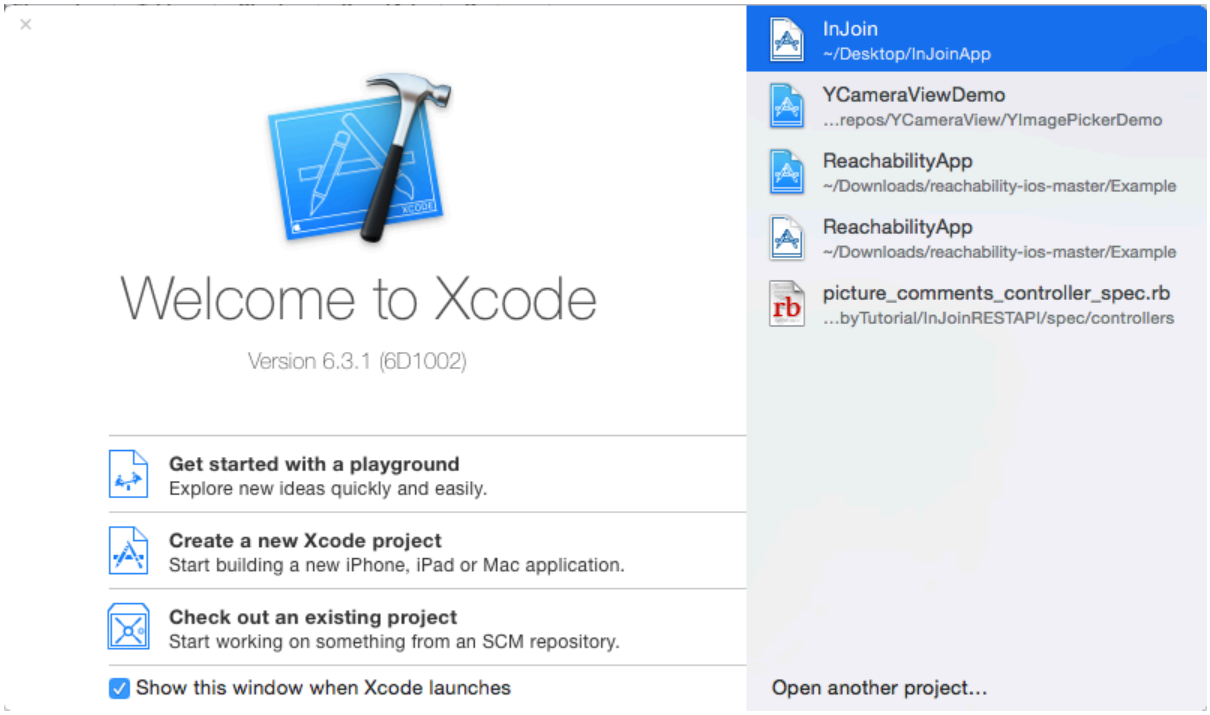

Figure 4.1: Initial screen when opening XCode

Once this is done, XCode redirects to a screen where the type of project desired must be selected. A variety of categories and types of projects are available for choosing, depending on factors such as if OSX or iOS is the desired platform, and what kind of project will be developed (application, library, among others). This may be observed on the following figure:

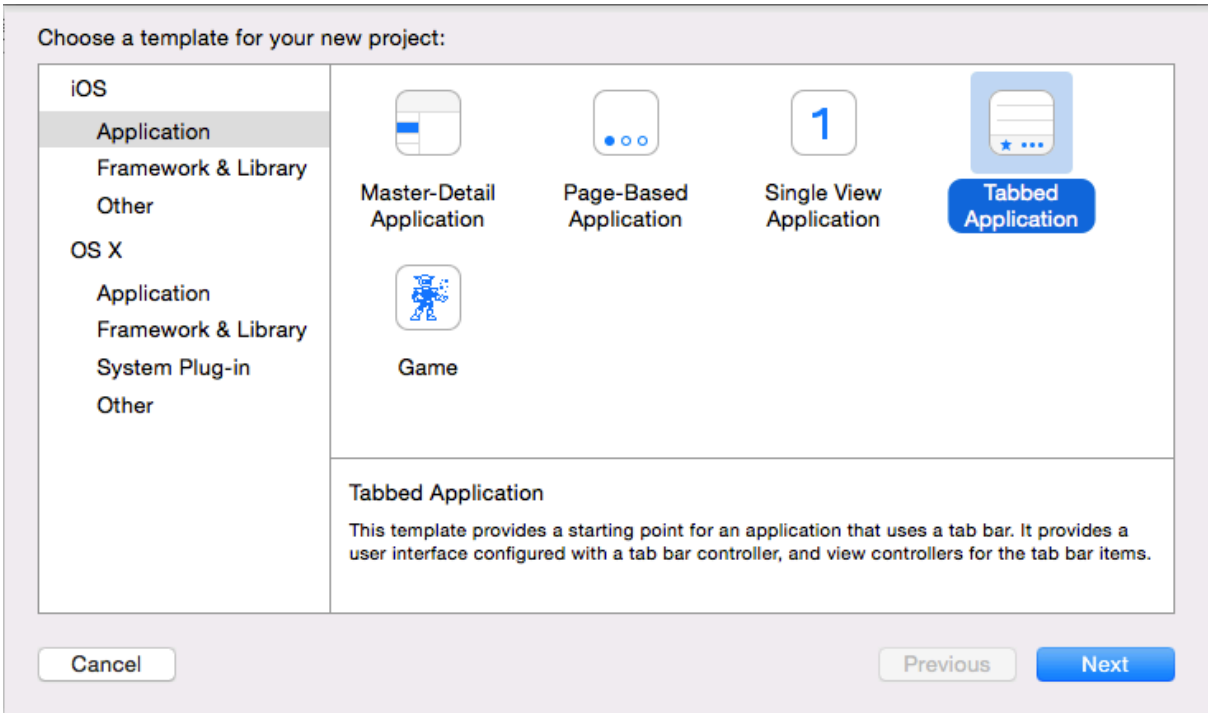

Figure 4.2: Choosing a Project base template.

As readers may appreciate on the figure, different template types are available for selecting under iOS ->Application. Among them options for Master-Detail Application, Page-based application, Single-view application, Tabbed Application and Game can be found, along with a brief explanation on each one of them. Regarding the decision of which template to choose, it would depend directly on what kind of app will be developed, and what set of requirements are desired for it, and it varies according to these needs. When one is chosen, XCode will redirect to the Application information form.

At the Application information form screen, XCode requires the name, the organization identifier, the language in which it will be developed (Objective-C or Swift), and some additional information. This can be verified in the following figure:

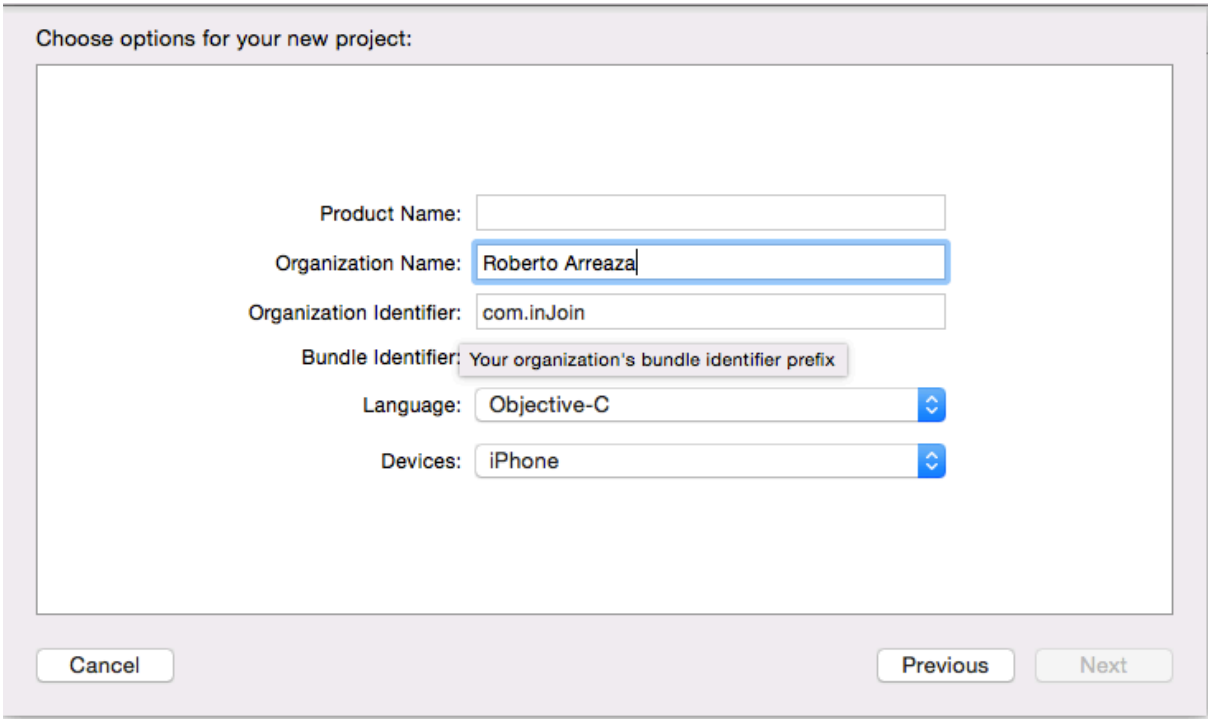

Figure 4.3: New Project information form.

It is important to note that there's an option to determine if the new project is intended for iPhone, iPad or universal (any device). The "bundle Identifier" of an application is composed of the "company Identifier" plus the application name, and it has reverse-URL notation, which is a consensus and allows for uniquely identifying Apps. This also proves useful when registering Apps on iTunes Connect, for the creation of provisioning profiles specific for each App that are necessary for installing them into physical devices with iOS, and for publishing Apps into the App Store.

Afterwards, the main development screen of XCode will open. From within this screen, the presence of some pre-loaded files can be noted, which depends on which template was chosen on previous steps. This is illustrated on the following figure:

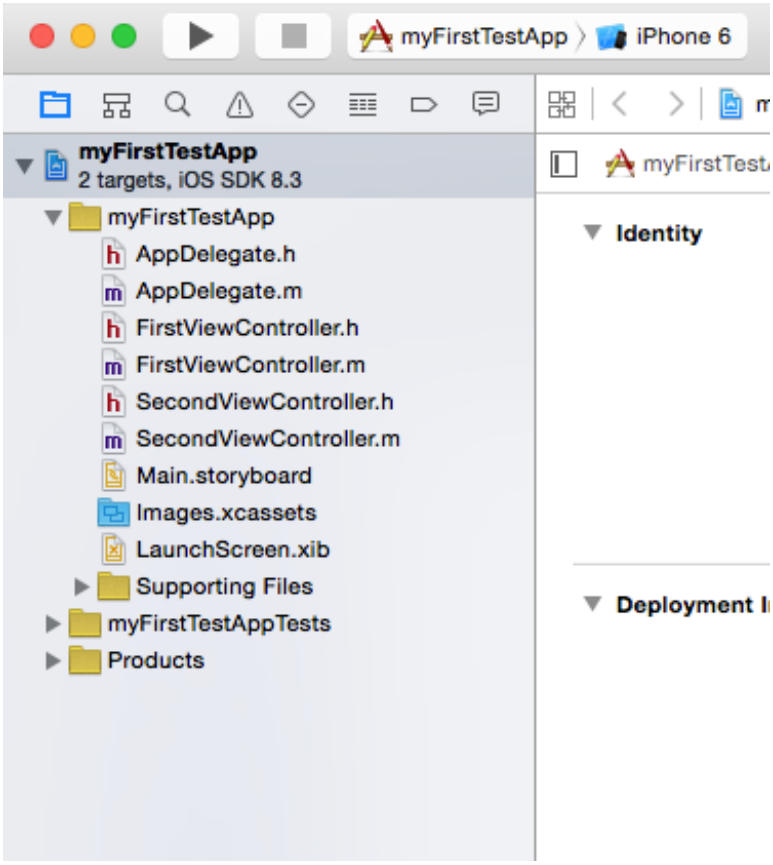

Figure 4.4: XCode project navigator with the default files for the "Tabbed-Application" template (XCode 6).

The chosen template for the example is that of a Tabbed Application, and therefore the default files shown are those corresponding to a generic app of this kind. The "myTestApp" file in blue corresponds to the new Project, and within it the files/directories hierarchy is located. A full demonstration of all of the XCode interface, or for each type of files created is out of the scope of the present section, although there are still a couple of clarifications to make. Firstly, any iOS application, no matter which template is chosen, requires an "Application Delegate" file, better known as the "app delegate". Such file constitutes the fundamental element that will receive relevant events at the **Application level**, such as the event indicating the App has completed the initialization process (launch), determined by the method:

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
```

Other events, such as the ones indicating the App will be terminated, or that it will enter in background/foreground mode, or entering in activity/inactivity are received in a similar way. In any case, the fundamental events is that of the app launch, due to the fact that it represents the point of entry or the beginning of the Application, and is widely used for initialization purposes, as well as other important tasks that may be required when the App initializes.

Another fundamental point to make from XCode's main view, revolves around the ability to directly create/edit UI files. To create a new UI file go to File->new->File. Immediately afterwards choose the option ***View***. "Views" are the mobile equivalent to a "screen" or a "window" in other environments. They constitute a set of related UI elements presented in a single interface, which aims to fulfill certain objectives from within the App. Each View is usually related and linked to a "View Controller" o ***ViewController***, as determined in the MVC (Model-View-Controller) philosophy. For further information regarding MVC, readers may refer to the section explaining Objective-C of this document.

Storyboards are special files (with the *.storyboard* file extension), where many Views are related among each other, giving a better sense of the lifecycle and sensation of the Application, from a navigation perspective. This means they hold different Views and many of the sequences that relate them to each other, once the user or the code performs a certain action at the application level.

When opening the storyboard file for the example, on the XCode editor can be seen something like the following:

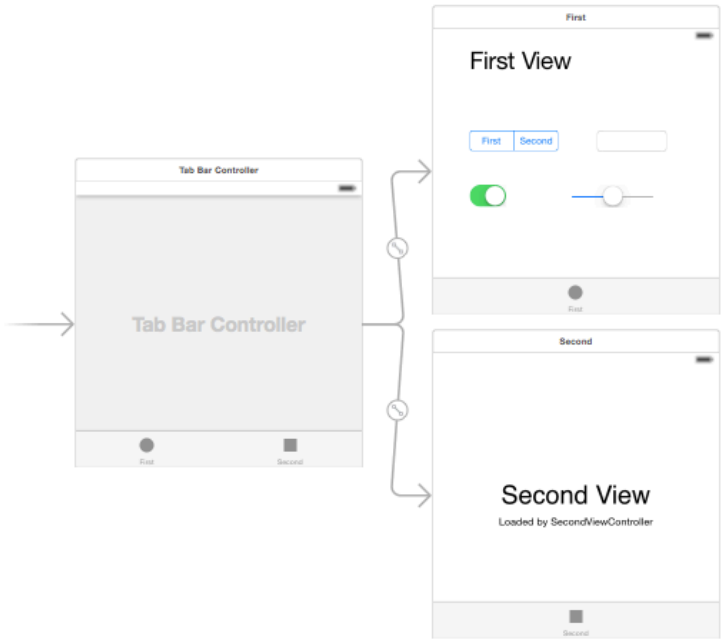

Figure 4.5: Opening a .storyboard file from XCode's editor.

There are also single View files (identified by the extension .xib) specifying only the information contained in a single independent View. Opening a .xib file opens a special UI editor in XCode, as shown next:

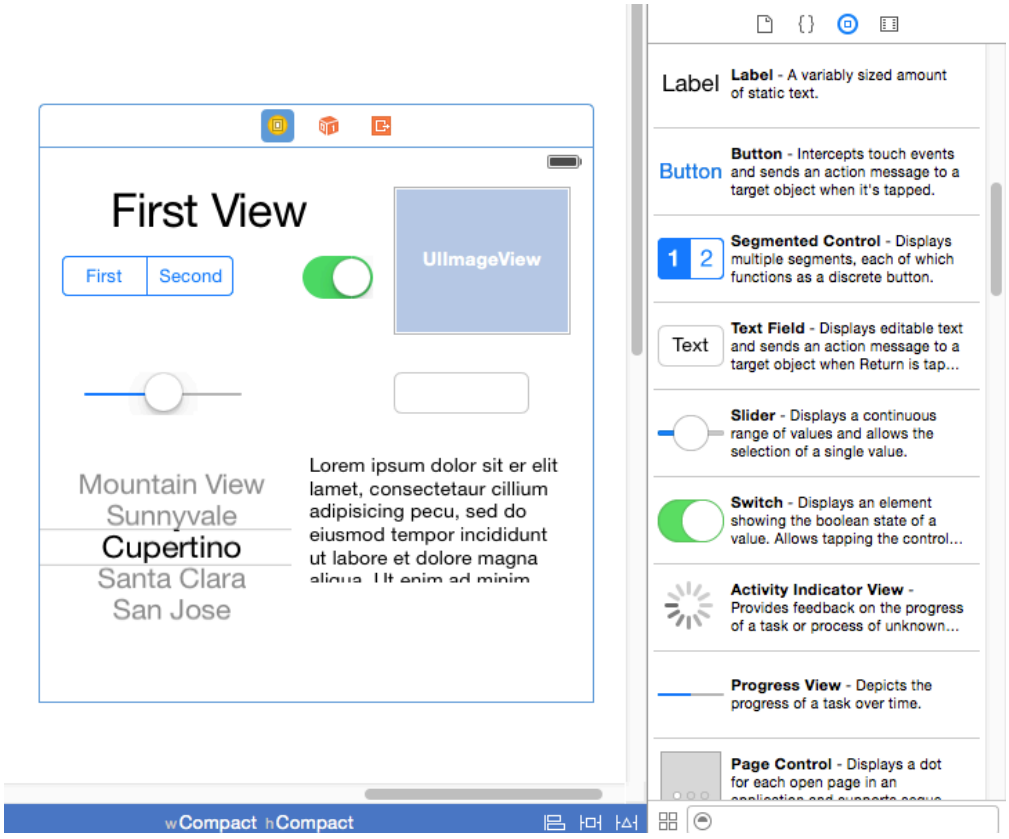

Figure 4.6: Opening a .xib file from XCode's editor with a glance into the UI elements toolbar (XCode 6).

As it may be appreciated, it's an iPhone view with several standard UI elements belonging to the iOS framework. Usually all classes with the purpose of being a UI element, have a name with the prefix "UI". Parting from the most simple Labels (UILabel), going through text fields (UITextField), image fields (UIImageView), to more complex elements such as web views (UIWebView), a great variety of very useful elements can be found, and with a simple "drag&drop" they'll be at the disposal of developers to build their user interfaces.

The right side toolbar is where all the options for UI components are presented from Interface Builder (XCode component in charge of the creation/edition of interface files), and it shows all sorts of components with many different purposes. Another fundamental feature of Interface Builder is that it allows to "connect" or link UI elements to different corresponding pieces of code (from within ViewControllers generally) to associate the code and behavior defined in Controllers with the appropriate elements on the user interface. Furthermore, it allows to accomplish this just by doing ctrl+drag from any UI element, into the appropriate ViewController file, as it is shown in the following figure:

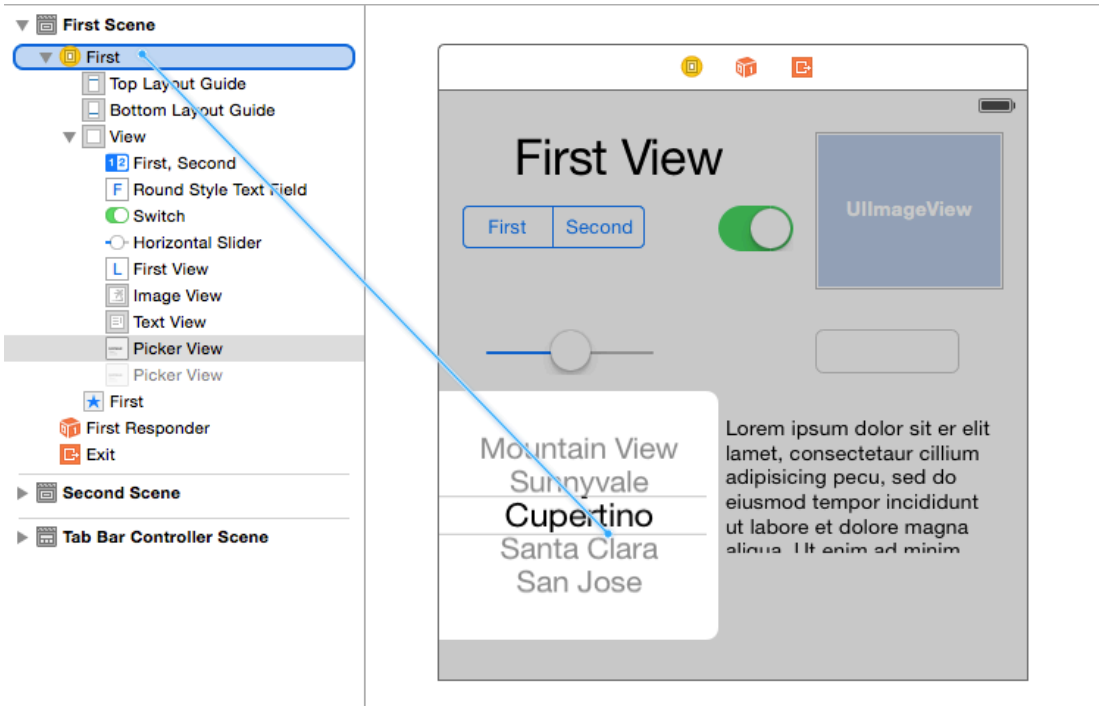

Figure 4.7: Linking UI elements to source code interfaces in ViewControllers (XCode 6).

In the figure, "First" (on the left panel) represents the Controller assigned to the created View, and once it's selected by crt+drag from any of the View's UI elements, Interface Builder will display the connection options available, as it can be verified in the next figure:

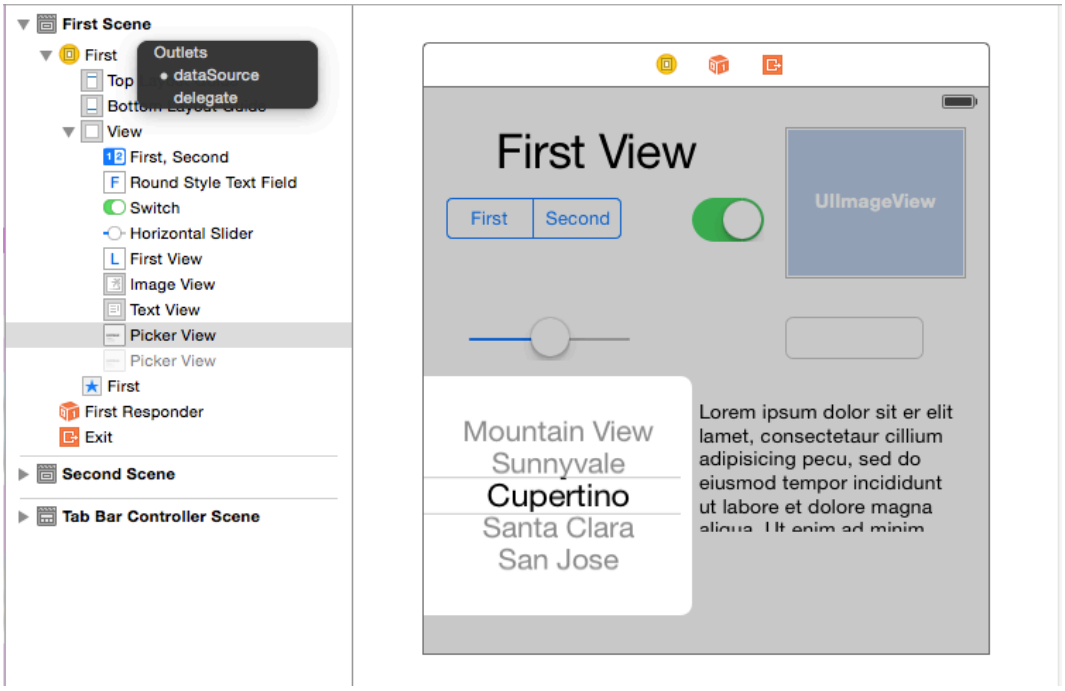

Figure 4.8: Possible connection options displayed from Interface Builder (XCode 6).

Reader may observe the possibilities for connections among the UI element selected, and the Controller file, to be able to link them.

Both Interface Builder and XCode can easily be the subjects of a book of great depth, but the purpose set for this section is to provide a glimpse to the main and most important sections and concepts that have the greatest relevance when developing for iOS or OSX. If getting in-depth knowledge about these topics is desired, it's recommended to refer to the official Apple guides. For instance, it is recommended to take a look into the XCode user guide, which can be found on the official apple developers website https://developer.apple.com (Apple Inc., 2012a) or in the diverse literature with focus on the subject (Wentk, 2011).

4.2 iTunes Connect y iOS Dev Center

The iOS Dev Center is Apple's portal dedicated to developers. This website goes hand and hand with the iTunes Connect website (which is the subject of a deeper examination further ahead on the present section) for some additional complementary tasks, although its main focus is on being the portal where the latest development tools (beta versions of iOS before their official release, latest XCode versions, OSX versions, among other resources) are made available to developers worldwide. In addition to this, the iOS Dev Center hosts a fundamental section that allows for installing, signing and validating code on physical devices, or to upload Apps to publish them on the App Store: the Certificates, Identifiers & Profiles section (Apple Inc., 2012c).

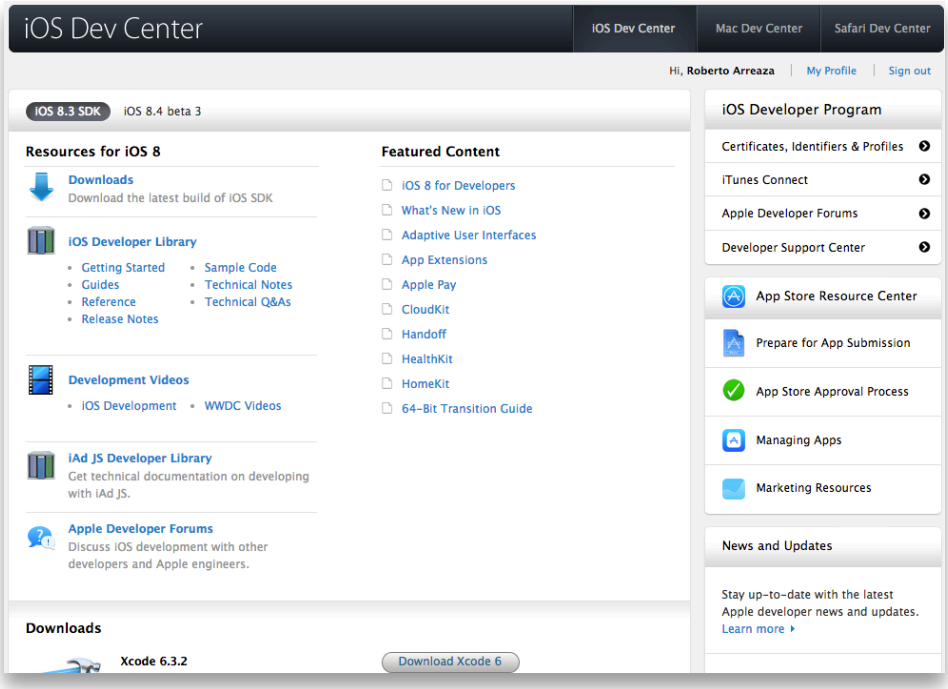

Figure 4.9: iOS Dev Center Portal (the link for accessing Certificates, Identifiers & Profiles section is found on the right) . Source (Apple, Inc. 2012c)

From within the *Certificates, Identifiers & Profiles* section of the iOS Dev Center website, which is accessible from the right side, developers can access a wide variety of important sections. Among them can be

found the Certificates, Devices, App IDs, Provisioning and Distribution sections. They can be observed in the following figure:

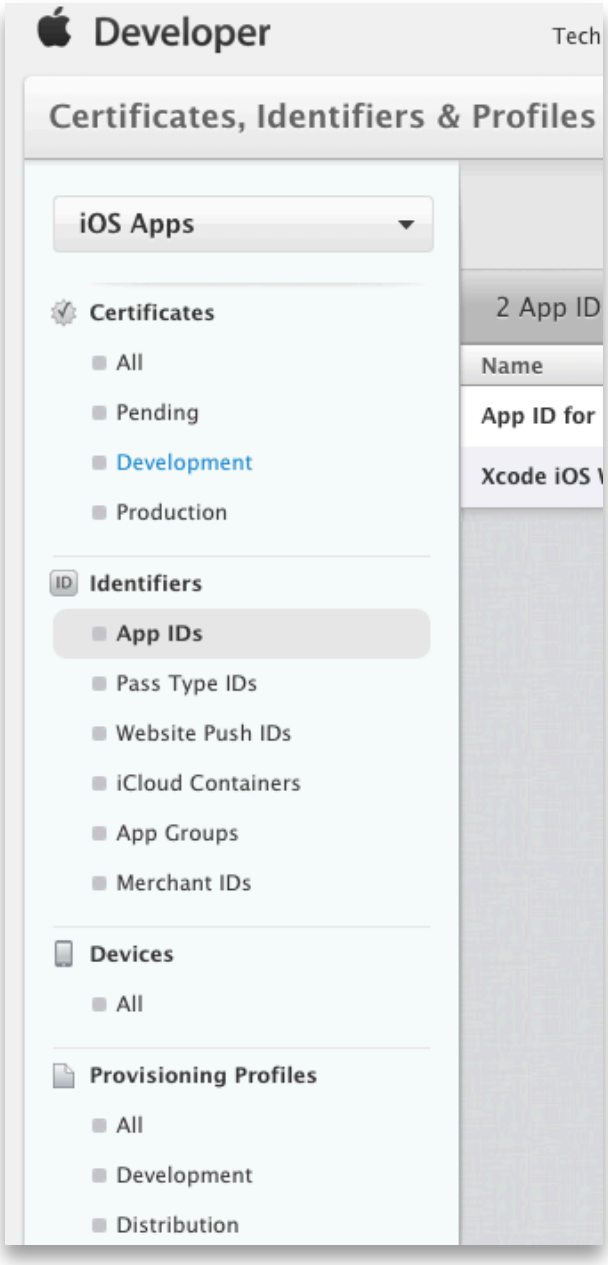

Figure 4.10: iOS Provisioning Portal sections.

Before creating Provisioning Profiles for installing and running Apps in physical devices and outside of the iOS simulators, the first section of interest is the Certificates section. This section allows for the creation and management of the developer certificates, which compose a digital

identifier with real-life information about developers, entities or companies, such as an email, name and other data. Such certificates are employed when digitally signing Apps, employing a system based on public and private key pairs. The process of creating a developer certificate is performed locally in the computer via a program called "Keychain Access", which is preinstalled on macs for OSX. From there, developers can access the certificates section and proceed to creating a new petition for a certificate with the email address linked to the official iOS developer account, and with a 2048 bit RSA encryption key. This will generate a file that is then loaded into the certificates version of the *Certificates, Identifiers & Profiles* page on the web. Once processed and validated by the web tool, the certificate may be downloaded to the Computer to be used for digitally signing Apps. It's enough to double-click the downloaded file (.cer) so that Keychain Access opens it and installs it automatically for its use and availability in the computer.

Once this essential step is performed, it is possible to operate on other sections of the *Certificates, Identifiers & Profiles* section of the iOS dev center website. The second relevant task to perform corresponds to the "App IDs" section, from which developers manage unique App identifiers used for accessing different parts of the keychain. These identifiers are necessary to be binded to specific Applications. When clicking on the button to create a new App ID, a form such as the following is presented:

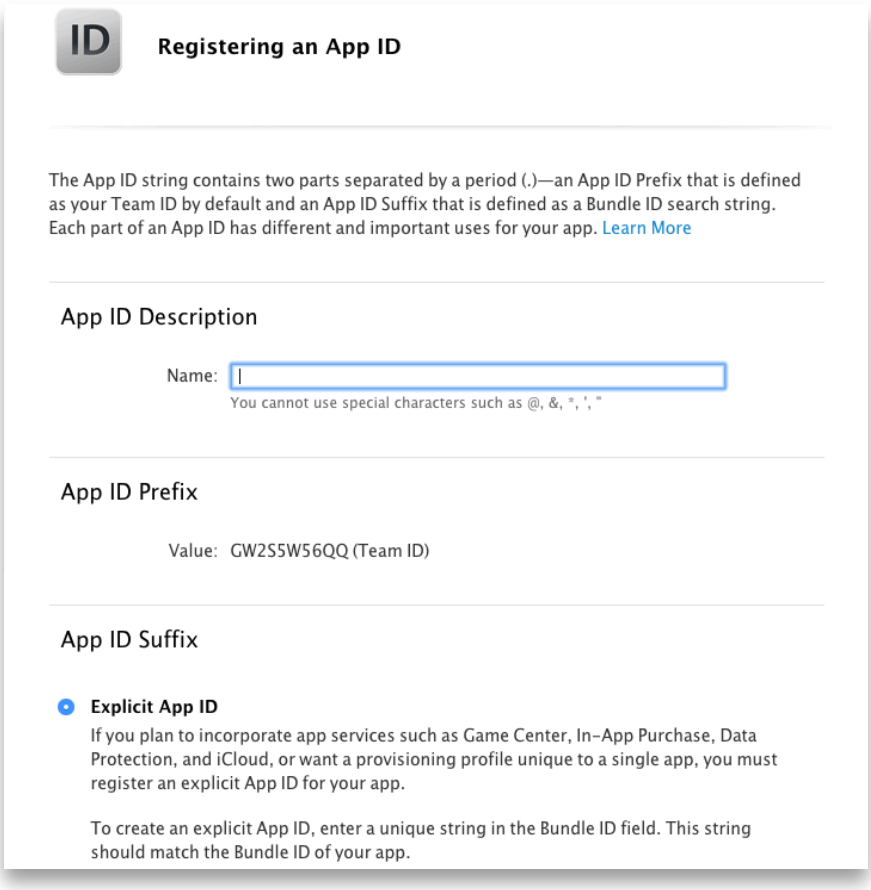

Figure 4.11: App IDs creation form (iOS Provisioning Portal).

This form asks for a descriptive name for the new App ID, and uses a "Bundle Identifier", with a reverse URL format, such as com.CompanyName.AppName, and this must constitute a unique identifier. The final App ID will be composed by the Bundle Identifier and a prefix denominated "Bundle Seed ID", which is generated by Apple, and is 10 characters long.

Another important part of the iOS *Certificates, Identifiers & Profiles* section, is the one named "devices". Here developers manage everything related to iOS physical devices to be used in any stage of the development. A single official developer account can link to up to 100 devices for this purpose. Adding a device intended for development tasks is pretty simple. When clicking on the "Add" button in the appropriate section, a form requesting a name for the device and the UDID code linked to it will appear. The UDID can be obtained by plugging in the device to the computer, and from XCode going to Window->Devices, and

the code will be under "identifier" in the "device information" section, as shown in the next figure:

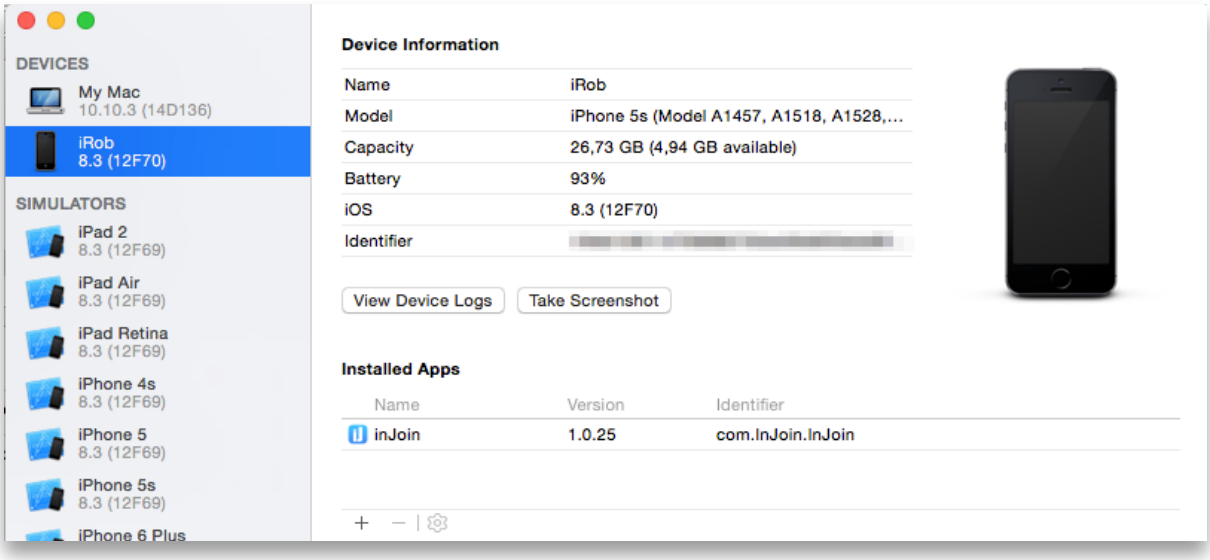

Figure 4.12: Finding the UDID from XCode (XCode 6).

It is also possible to retrieve the UDID directly from iTunes, by entering in the device information, under "summary", and clicking on "serial number", which will reveal the "UDID" as it is observed in the following figure:

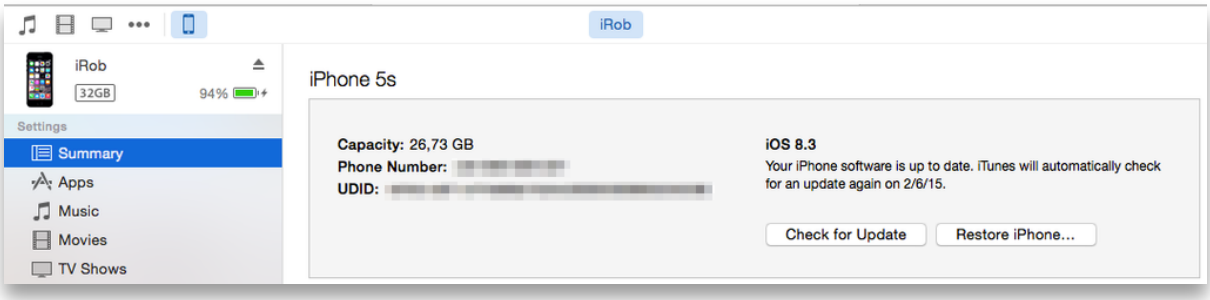

Figure 4.13: Finding the UUID of a device from iTunes (Apple Inc. 2012b).

Once the form is completed with the UDID, it is then possible to associate the added device to certain "special files" that will allow for the installation of development Apps in such device, and use it freely for this purpose.

The referred "special files" that allow for the use of development devices are known as "Provisioning Profiles". This are composed by certificates, App IDs and UDIDs related amongst each other. Inside the "Provisioning Profiles" section of the *Certificates, Identifiers & Profiles* part of the iOS Dev Center, is where developers manage everything related to these files. There are different types of Provisioning Profiles: development and distribution profiles, and their usage depends on whether they'll be employed to install Apps on devices for development tests, or if they're meant for publishing Apps on the App Store, respectively.

Creating Provisioning Profiles is fairly simple. By clicking on "new profile", a form is displayed, which allows to choose a name for the profile, the certificates, App ID and the Devices to which it will be linked. Once all elements are selected, by finishing the creation the Provisioning Profile is ready to be downloaded. Once the file is in the local computer, it is enough to drag it onto the XCode application icon and it will be automatically added into the appropriate place to facilitate development using this profile. From now on, when opening from XCode an Application project using the App ID that is associated to the Provisioning Profile generated, and when plugging in a device that is also associated with the profile, this device will then be displayed under the available devices for installation list that allows the App to run on the device.

Once everything regarding the iOS Dev Center is clarified, the next subject to approach would have to be iTunes Connect. Reachable on the website https://itunesconnect.apple.com, iTunes Connect is a web from Apple that's destined for developers and development companies for iOS and OSX. It is through this site that official Apple developers can manage their Applications as well as creating new Applications, or simply edit and change their characteristics and general information. It also allows for accessing sales information, taxes and banking data necessary for managing the business built around the Applications.

The main functionalities of iTunes Connect revolve around App management tasks, and even though it is also well known for the presence of other kinds of information (sales, statistics, among others), the emphasis of this section focuses precisely on these management topics.

From the "My Apps" section of iTunes Connect there is a list of all the existing Applications that have been created from this account. From this

list, one has the possibility of editing them (adding new versions, modify some information, etc.). In addition to this, it allows for the creation of a new App; clicking on the "add new App" button will suffice to perform the operation. When doing this, there will be a form asking for information to identify the App, define the default language to use (for the purpose of the web and App Store) and the bundle ID (which is obtained from the list of App IDs created previously from the iOS Dev Center). This way developers can create an ID for each App to be developed, and associate them to each other directly in this first step of the App's creation. Right afterwards iTunes Connect will require will require other kinds of information such as screenshots of the App, a description and other sets of data that are beyond the scope of this brief introduction. The process explained is illustrated in the following figures (Apple, Inc. f12d):

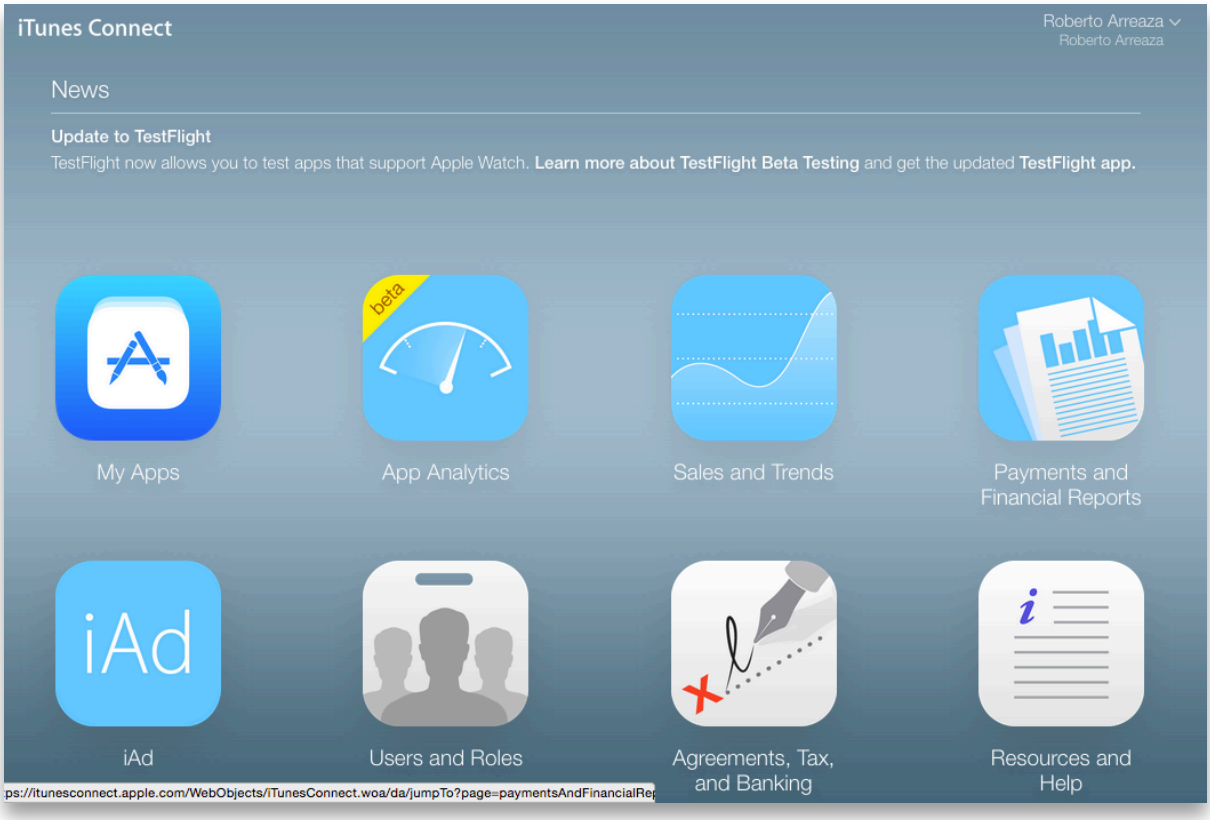

Figure 4.14: Main menu for iTunes Connect (notice the "My Apps" section as the top left option).

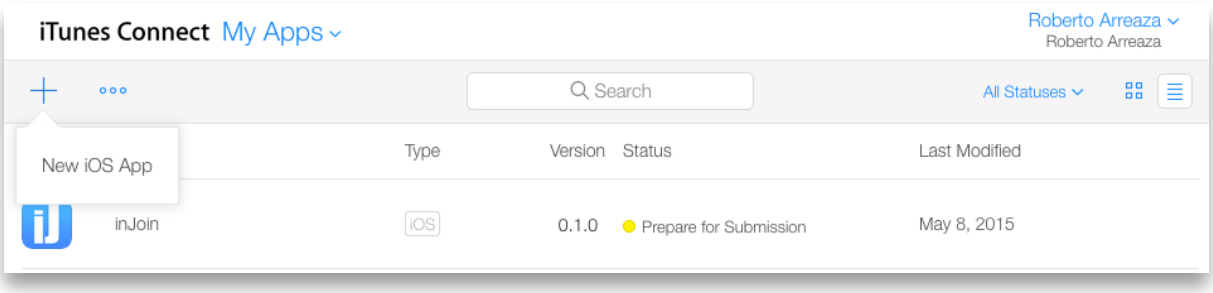

Figure 4.15: "Add new App" option from the "My Apps" section from iTunes Connect.

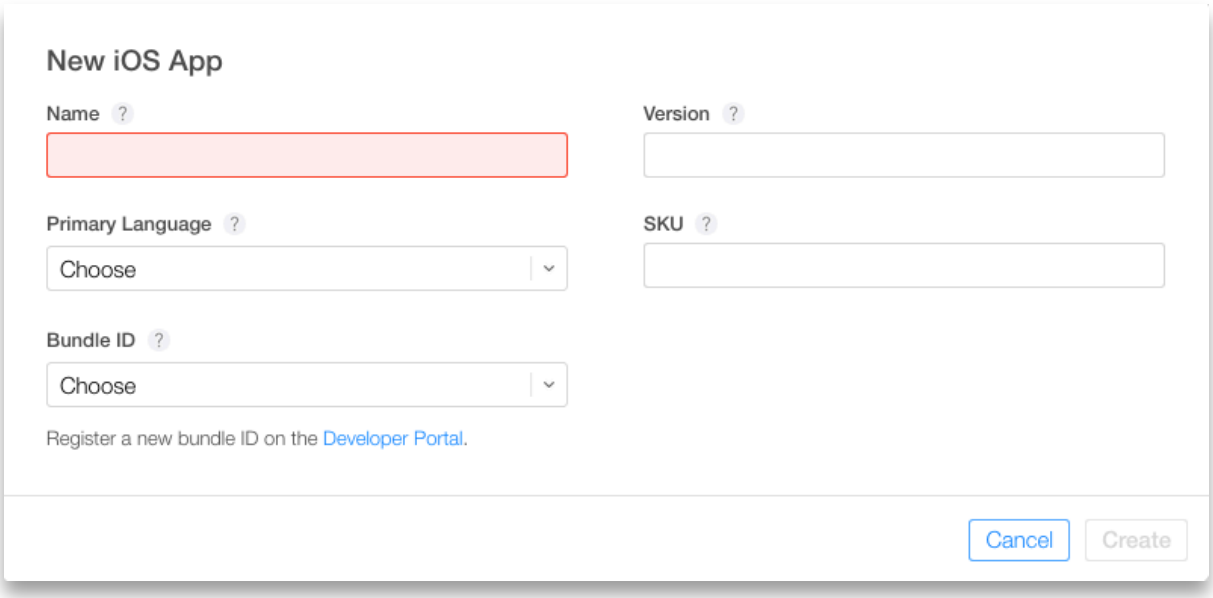

Figure 4.16: Creating a new iOS Application in iTunes Connect.

Another very important process in the making of this project, and in general for any modern iOS Application, is the process of beta testing.

Beta testing Apps consists on the process of releasing a Beta (stable but prior to release) version of the Application to a set of early-stage tester users, in order to be able to detect, track, analyze and fix any encountered bugs, User Experience (UX) problems or improvements, and get additional useful feedback from a contained sample of the Application's target market users. It is not only useful for the early-development stages of an App before its release, but also for the continuous improvement of the App, to validate new features, experiment

with certain functionalities to see how customers react, and many other useful tasks, without having to risk publishing some code into the App Store with a higher degree of uncertainty about its performance, user-perception and usability.

Before the official release of iOS8, Beta testing for iOS Applications was traditionally done by means of mostly "manual installations". This process implied that in order to install an App in one or many devices, one would have to ask for the devices UDIDs, and manually add them into the Devices section of the iOS Dev Center (as it is explained on the section regarding the iOS Dev Center).

Once all UDIDs are registered (one for every device to be used for beta testing), from the iOS Dev Center Provisioning, developers must create a new distribution provisioning profile of type "ad hoc" and send a packaged version of the compiled app (signed with this provisioning profile) to all Beta tester users, who could then install it from iTunes or XCode.

This process seems like a lot of unnecessary work for the simple task of sending testers a Beta version of the App to install it, and taking into account that there may be hundreds or even a thousand Beta users (depending on the scope and scale of the Application), it easily becomes a tedious and cumbersome task to perform.

So from iOS8 onwards, Apple incorporated a new service called *TestFlight*, which dramatically improves the efficiency, scope, and ease of use of the process for delivering Beta testing Versions of iOS Apps. They incorporated it from an existing web service that simplified some of the tasks when delivering Beta versions manually, but still most of the cumbersome devices and profiles work was unavoidable.

On the official website (Apple, Inc. 2015a) this is what Apple has to say about TestFlight:

"TestFlight Beta Testing makes it easy to invite users to test your iOS 8 apps before you release them on the App Store. You can invite up to 1,000 external testers using just their email address."

Thus, TestFlight is a platform where iOS Application developers are able to distribute Beta Testing versions of an App to up to 1.000 test users almost effortlessly, by only using an email address to where an email

invitation will be sent. This makes App distribution tasks as simple as they can possibly get.

This is accomplished by following a process roughly summarized as:

- Uploading an App version into iTunes Connect.
- Enabling and setting-up the App version for TestFlight Beta testing.
- Adding Beta tester users by their emails.
- Associating Beta tester users to the uploaded App version.
- Initiating Beta App validation (performed by Apple).
- Once validated for Beta testing by Apple, send the invitations to all associated users.
- Upon the reception of the invitation, users download the TestFlight App.
- From the TestFlight App, users will then be able to download the Beta version of the uploaded Application.
- Users may send feedback about the version (optional).

This process can then be segmented into two parts: setting up the App version for TestFlight Beta testing (performed by the developer), and afterwards installing the App from TestFlight (performed by the test users).

i) Setting up an App for TestFlight Beta testing

The first task is to upload the Beta version to be installed and used by the beta testers. In order to do this, the first thing to do, is to make sure that all the App's required Application icons are added into the project and properly set-up. For the case of this project, the resulting icons set should look as follows:

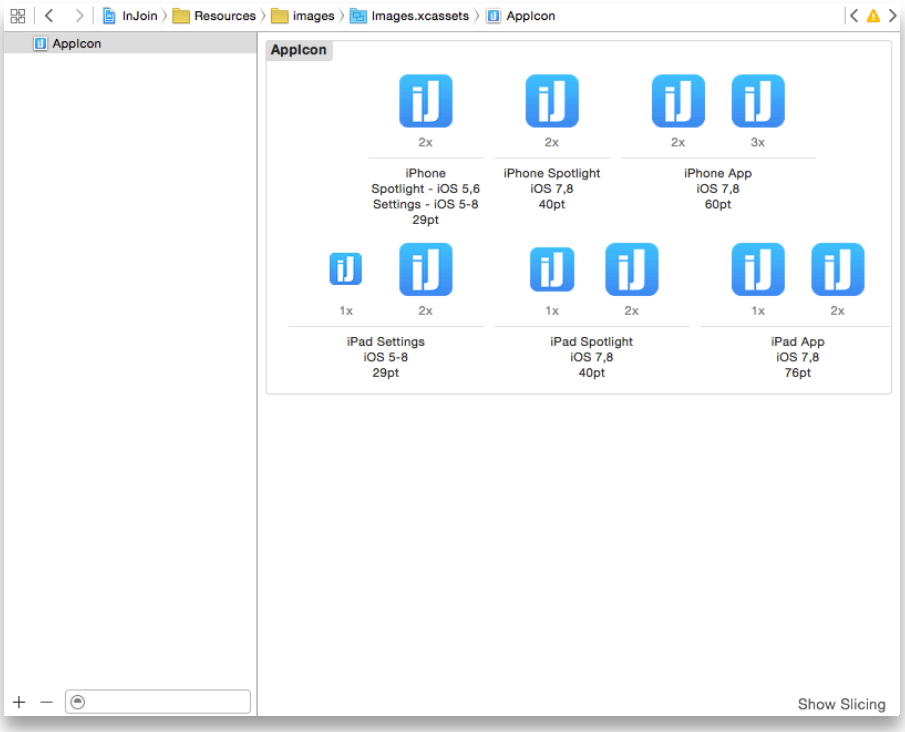

Figure 4.17: App icons set viewed from XCode (XCode 6).

Since there are several kinds of devices with different screen sizes and a variety of resolutions, the bundled Application requires this set of icons before uploading the App for TestFlight (or App Store) distribution.

After this is properly set-up, the project is ready to be bundled and uploaded into iTunes Connect. With a physical device connected into XCode, select from the main menu Product->Archive. This causes the project to compile and archive for uploading into iTunes Connect. When the archiving finishes successfully, XCode will display a list of all archives that have been bundled up to this moment, with the one on the top, being the most recent, as shown in the following figure:

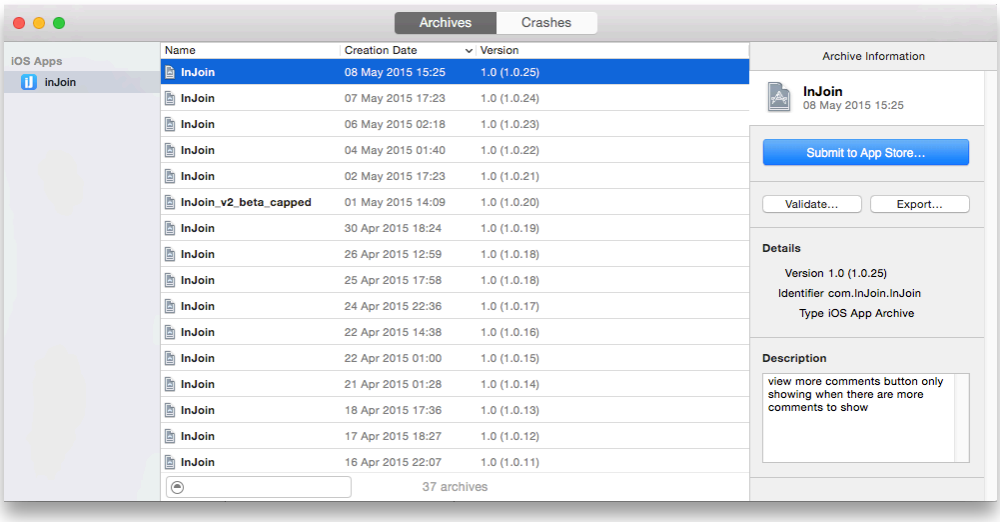

Figure 4.18: List of archives from XCode (most recent highlighted in blue) (XCode 6).

As mentioned earlier, readers may observe a chronological list of all archives performed up to the current date, and on top the one that has just been created. Each archive contains a description field, in order to identify any differences in all the different versions created (optional).

Notice the blue button enunciating "Submit to App Store"; this constitutes the next step to take. By clicking on "Submit to App Store", anew dialogue appears, requesting for a developer account certificate (with its corresponding provisioning profiles) to use in order to bundle the app and prepare it for either App Store upload or (for the case at hand) TestFlight upload.

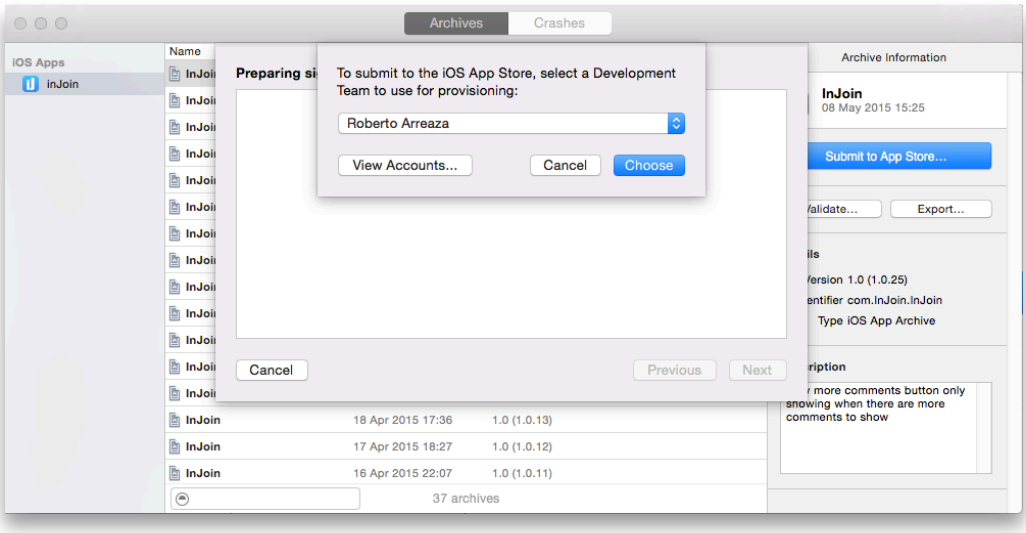

Figure 4.19: "Submit to App Store" selection of provisioning to use for App Store / TestFlight upload (XCode 6).

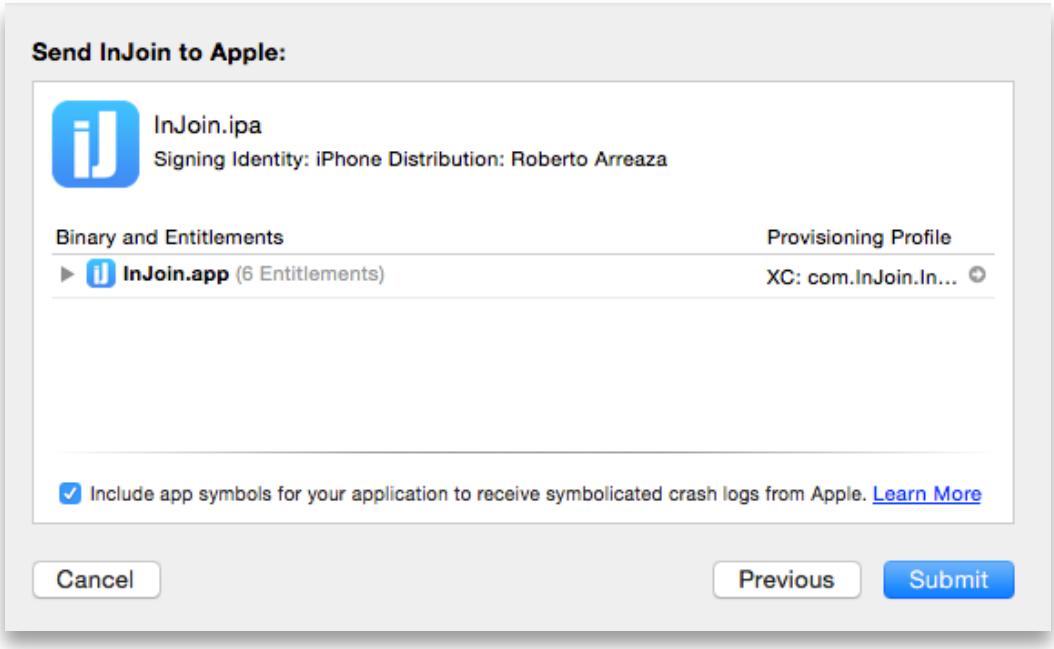

Figure 4.20: App submission into iTunes Connect (XCode 6).

Once the bundle process is finished, and the right provisioning profiles are used to accomplish this process, only then is the App bundle (.ipa file) ready for submission into iTunes Connect. This process is initiated by

simply clicking on "submit", which triggers the following upload progress dialogue:

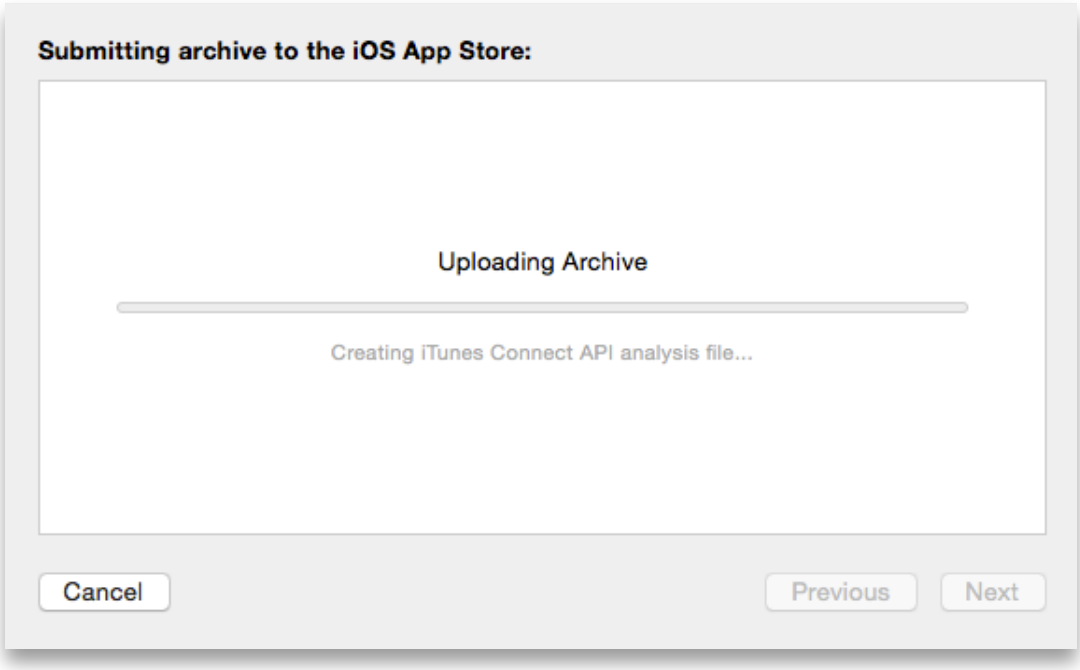

Figure 4.21: App submission upload and validation process (XCode 6).

Once the bundled App finishes uploading (and provided there are no errors during this process) the "submission successful dialogue is shown, indicating that now there's an available binary uploaded and accessible to/via iTunes Connect.

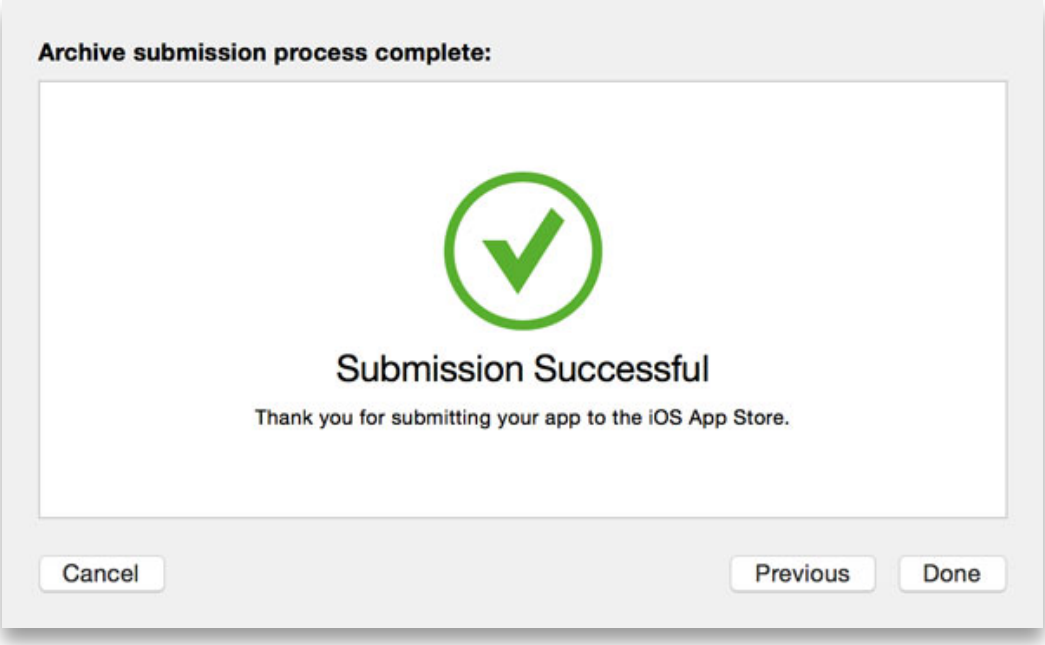

Figure 4.22: Successful App submission to iTunes Connect (XCode 6).

In order to setup the new binary for TestFlight testing, some important steps must now be taken from the iTunes Connect website.

A list of all the uploaded binaries done to iTunes Connect is found by accessing the App, under the tab "Prerelease".

From here, the "TestFlight Beta Testing" button must be switched ON, as displayed in the following figure:

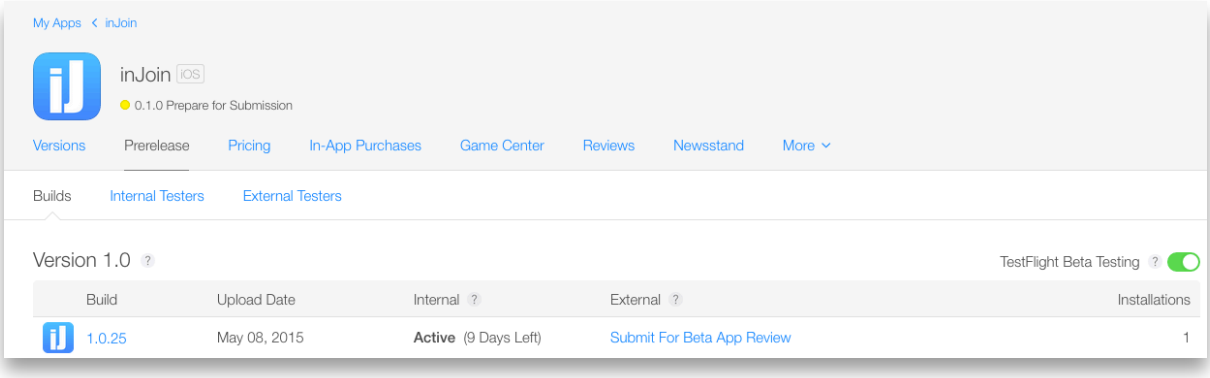

Figure 4.23: list of Prerelease builds for an App from iTunes Connect.

This activates TestFlight capabilities to the new App version and allows for Beta Testing any uploaded binary desired.

Since Beta testing requires, Beta testers, then a list of users that will become Beta testers must be created. This is done under Prerelease -> External Testers. This section shows all external testers available for the Application

To create new testers, it is necessary to click on the Add button and then "Add new Testers" (there's also an option for associating existing users as the current App's testers), as shown next:

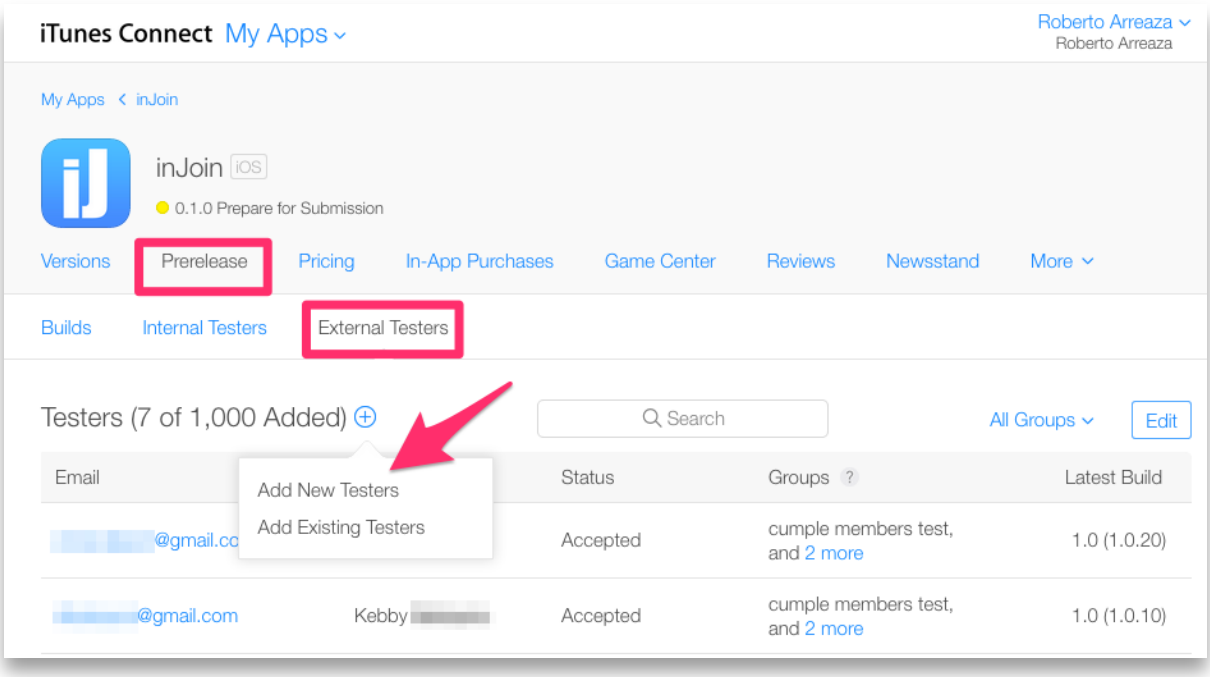

Figure 4.24: Associating external (Beta) testers to an App from iTunes Connect.

The previous step displays a new form, where one or many users can be created as testers for the Application. Since TestFlight invitations are sent via email, only the email is required, but a name and last name can also be added. This can be observed on the next figure:

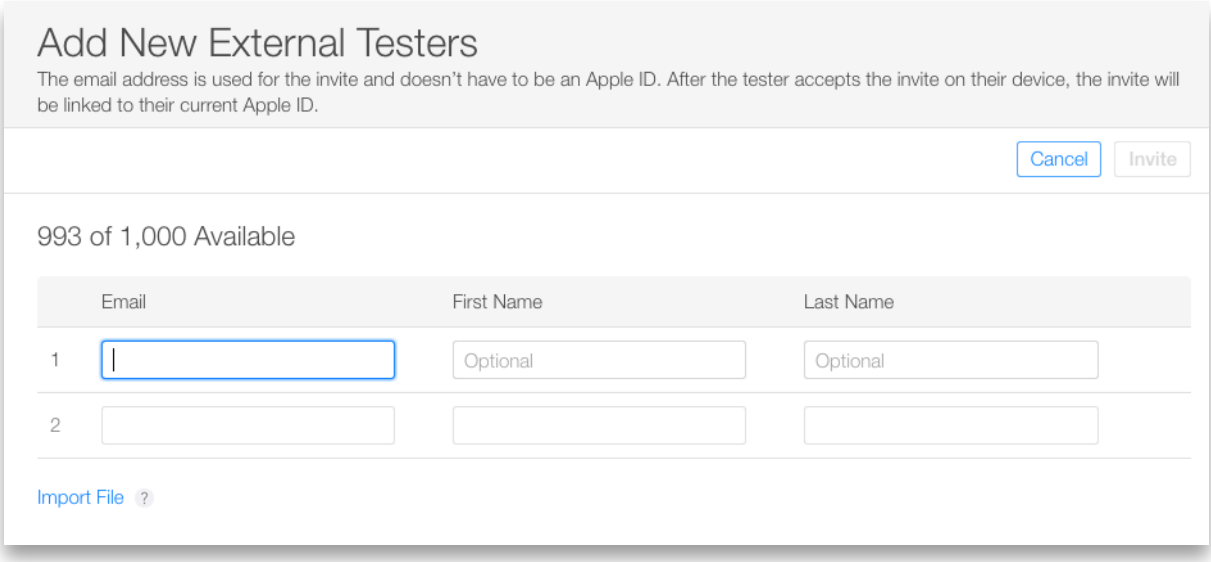

Figure 4.25: Adding new external (Beta) testers iTunes Connect (XCode 6).

Once all testers are created, all that's necessary is to submit the App for Beta App review and send out invitations.

Beta App Review is similar to the App Store review (although faster, and less meticulous naturally), and it consists on an Apple team testing the App and making sure it's ready to be Beta tested by any user, by checking some basic usage standards.

In order to initiate this process, it's enough to click on "Submit for Beta App Review" for the build version that's wished to be tested. This is depicted in the following figure:

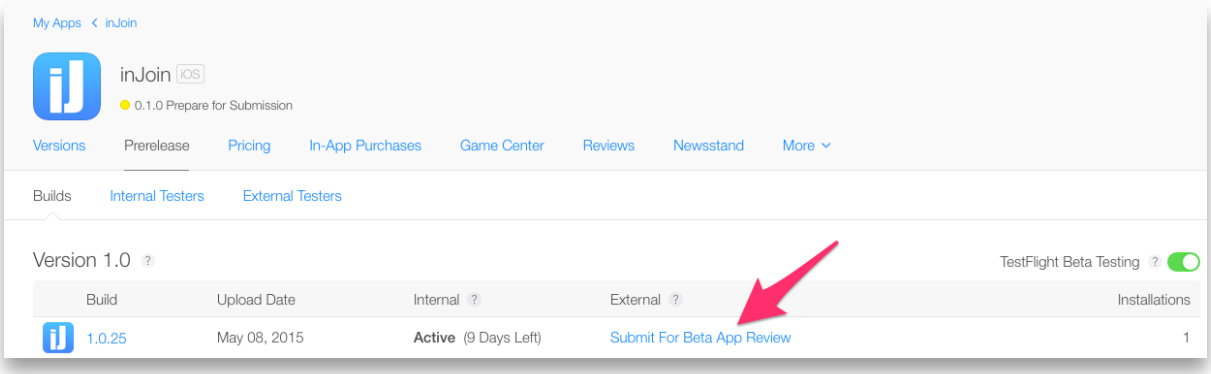

Figure 4.26: Submit an uploaded build for Beta App Review.

The Beta Review process may take a couple of working days to finish, so in the meantime it's a good idea to fill out some basic description and

information on the test version (intended for the Beta testers). By getting into a specific build, developers can access its details, and from there two fields stand out: "App description" and "What to test". They're both self-explanatory, and it's recommended to fill both, so that testers have a better idea on what the have in their hands, and what they're supposed to test.

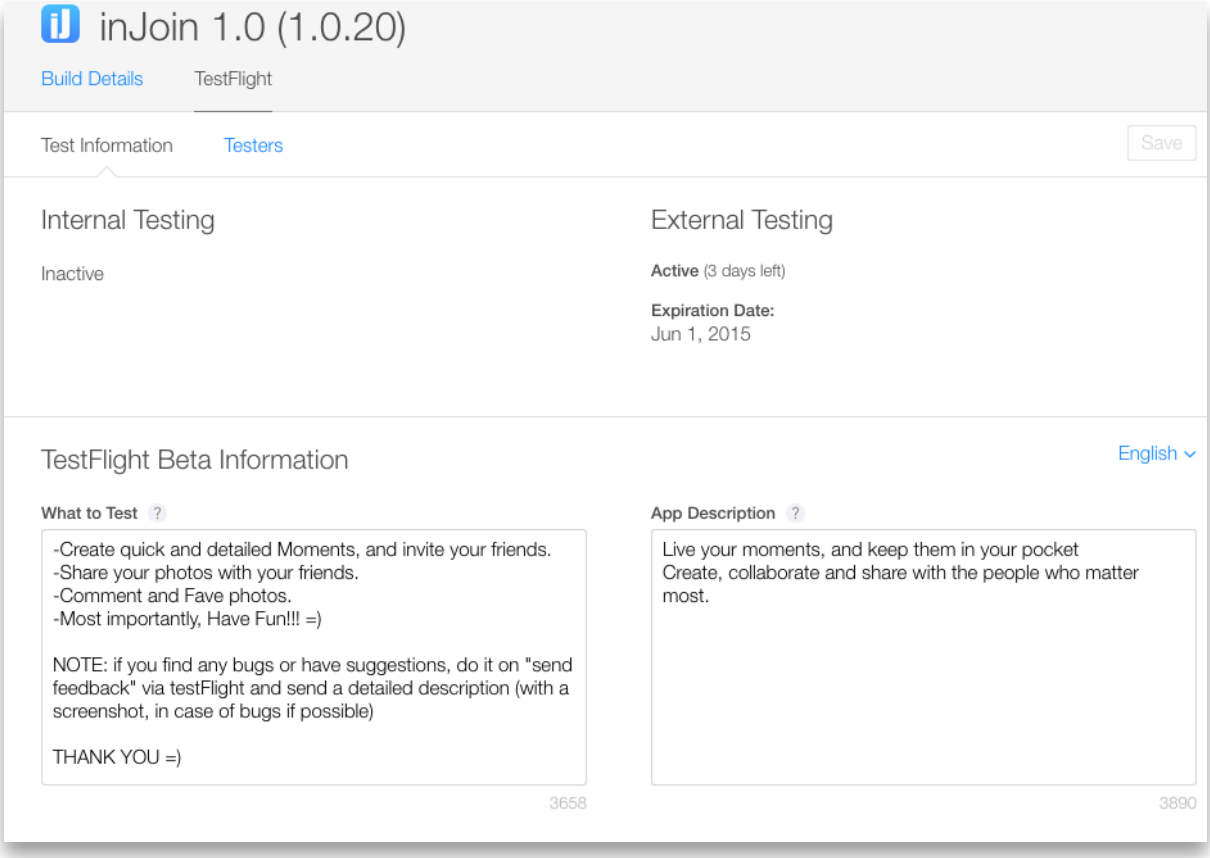

Figure 4.27: Basic test information and description of the Beta version from iTunes Connect.

Once the Beta Review process is finished, then the TestFlight invitations can be sent to the Beta testers (external testers), and they can download and install the app to their iOS devices.

ii) Installing a Beta version App via TestFlight

Beta testers receive an invitation via email to test the Apps sent by developers. By opening this email, it will have a link to The TestFlight iOS Application. If the TestFlight App is not already installed on the device, it will lead to the App Store for downloading it right away:

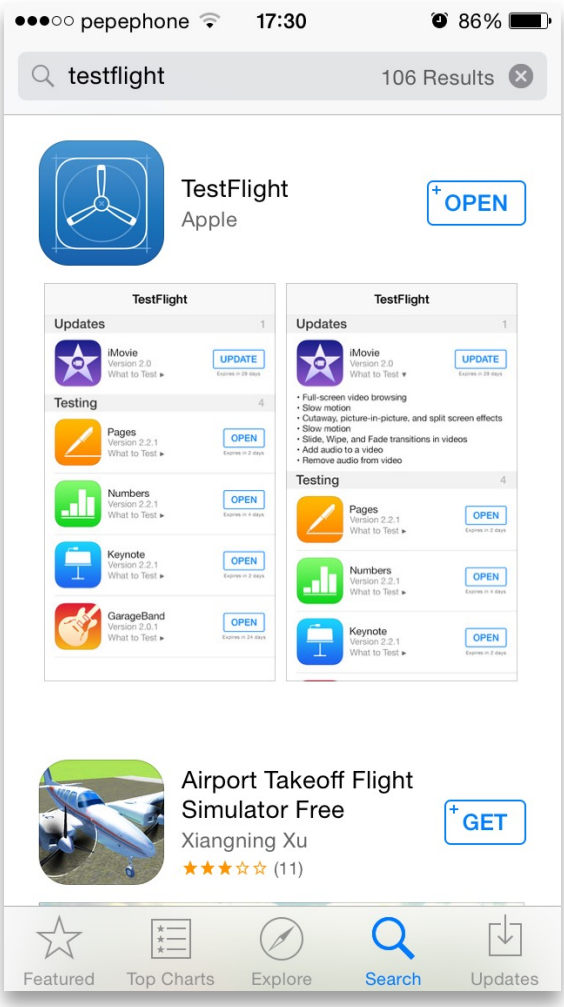

Figure 4.28: downloading TestFlight from the App Store (iOS 8).

Once the TestFlight App is installed, then the email link can be opened again, and it will take users straight to the TestFlight App, noticing that now testers will find upon the list of Beta apps, the Application to which they've been invited. This looks similar to the following image:

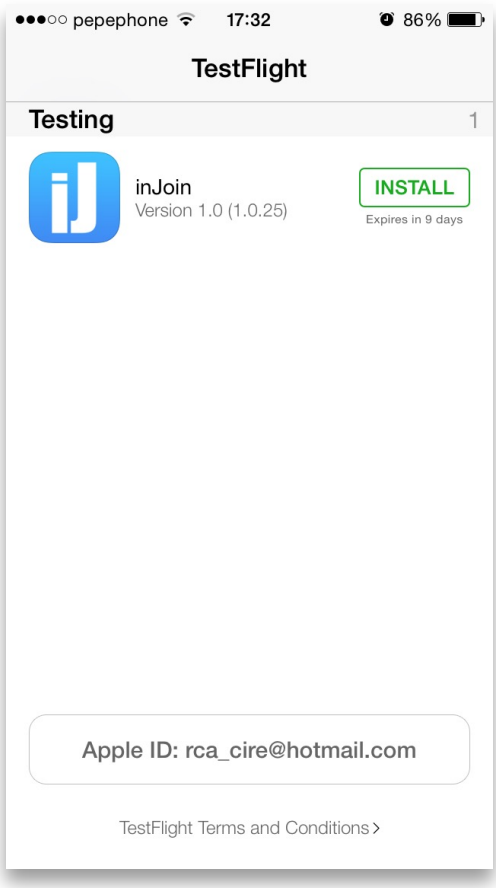

Figure 4.29: TestFlight Apps list viewed from the TestFlight App (iOS 8).

At it can be observed, the App icon prompt users to install the Beta version of the App resembling the mechanisms used when installing Apps from the App Store. By proceeding to installing the Application, users will end up with the App showing up in the home screen, as such:

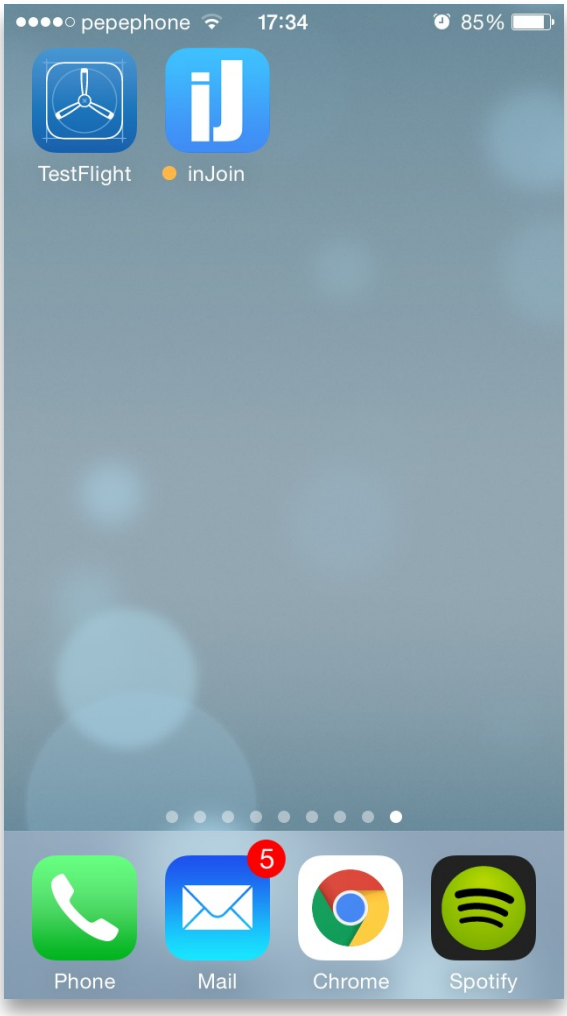

Figure 4.30: Successful Beta App installation via TestFlight (the orange dot identifies TestFlight-installed Apps) (iOS 8).

Notice the orange dot next to the App's name under the icon. This is an indicator that identifies Applications installed via TestFlight, from normal Application installed from the App Store.

An important Feature of TestFlight, and one very widely used to improve and fix the App developed, is the capacity to send feedback from the Beta testers to the developers, so they can make any pertinent changes or fixes that are considered appropriate. By opening the TestFlight App, users can access the details of the Application being tested, and form there, the "description" and "what to test" information is visible to users, but also a button labeled "Send Feedback", as it can be observed in the following figure:

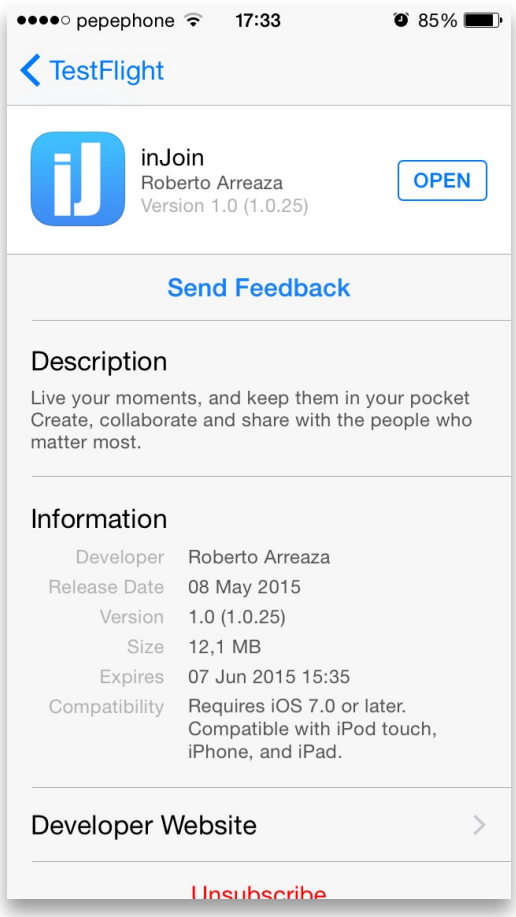

Figure 4.31: App details viewed from within the TestFlight App (iOS 8).

When a user taps on "send Feedback", their action triggers an email form with relevant device information that users may leverage to write and describe any suggestions or bugs found during their testing experience, as well as any attachments (very useful to send screenshots for UI problems, among other issues). This is depicted in the next image:

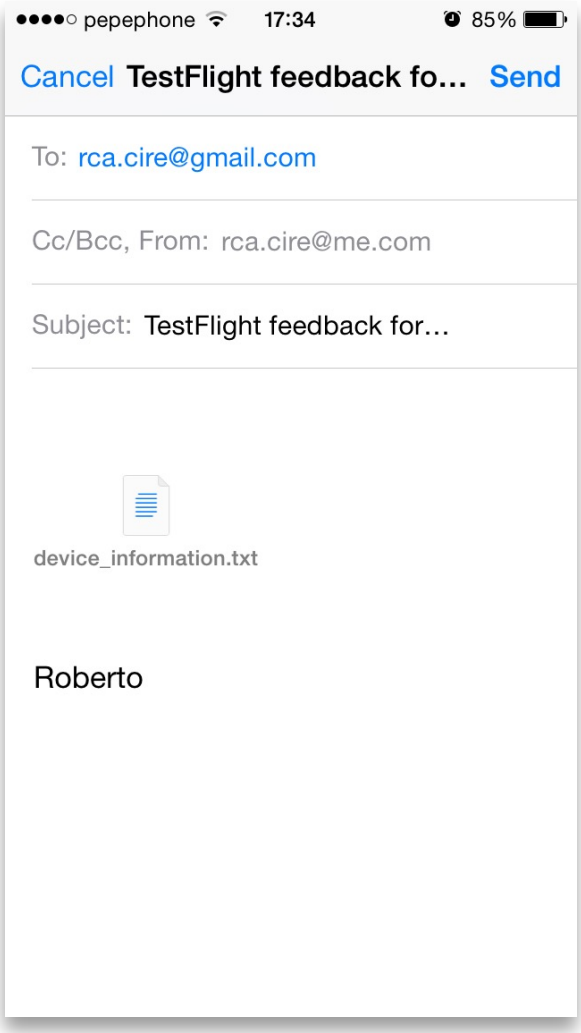

Figure 4.32: TestFlight feedback sample (sent via email) (iOS 8).

The TestFlight tool is a great advancement and it promotes beta testing at the right scales so that developers can then continuously improve and fix their applications based on real user feedback. This contributes to setting a good minimum level of quality for any Apps aimed to target the iOS market, and allows developers to work on the right feature sets as well as discovering and solving bugs that could otherwise be overlooked.

In this project it constituted an essential tool for validating everything that was being developed, and getting confidence about where to go next.

5. User manual

At this point, readers have an idea about the concept of Mobile Application to be implemented, as well as the main goals to accomplish, and the various sets of technological tools and concepts that are harnessed in the development of the project. The next step is then to present use cases and usage flow of the Application. For this very reason, this section intends to present the Application from the end user's perspective, including both the design, and its usage.

The first step should then consist on defining the basic functional specifications required to define precisely what the project should and should not do.

5.1 Required features

Based on the specific objectives that are outlined, as well as the basic necessities and requisites demanded by the core concept of the project, this section aims to define the specific desired functionalities that will be implemented, described below:

- There must be user accounts in order to uniquely identify users, and provide the ability of inviting people to Events, as well as the ability to be able to access from any device that has the Application installed. User accounts should at least include an email address, full name and a profile picture.
- For ease of use, facilitate the possibility of logging in with Facebook, to make registrations quick and painless, but still keeping the possibility of performing conventional user registration.
- Users should be able to form "connections" to other users, which conform the available networks of users that participate in the events that are created. A user may then invite to events of his own creation, any users that are part of his connections.
- The previous point implies that the App must also contain a "connections" section, where user connections can be managed, and where users can add/accept connections requests to/from other users.
- A "Me" section, where user profiles can be inspected by other users, and also conveniently being able to add them as

connections right from their profile. Here is where any available editions to the user profile should take place.

- The presence of a "news" section, as a form of news-feed, where any available relevant content is shown to the user, in a timely manner.
- An "inVites" section, where all invitations to specific events arrive, and users can then decide if they wish to participate in them.
- A "Check-ins" section, which should contain a timely ordered list of the Events where the user has participated in the past, as a form of historic record of what the user "has been up to".
- An obvious but fundamental feature is the capacity to create events in a very usable and simple way.
- The core functionality for the App compels to allow for accessing the details of an event and see the pictures posted to it, as well as being able to publish pictures directly from the app: this should be done in an intuitive, easy and straightforward way. Additionally, it's important for events creators to be able to invite people to any events created, as well as editing any detail information related to such events.
- Events should contain some relevant information such as the time and place where it will happen, a name, description, and any other information considered relevant for other functionalities.
- Finally, and with the purpose of making social interactions around pictures feel more familiar, people should be able to like events, as well as liking pictures and commenting on them as well, in a similar way Apps like Facebook or Instagram have their users accustomed to doing this.

5.2 App Terminology

Throughout this entire document, and within the context of the App, a variety of terms will be used to refer to very specific concepts. In order to facilitate a clear understanding of these terms and to what they refer, they are defined and explained next:

Moment:

A *Moment* consists of a specific event. It is an entity shared by the users that participate in it, and it's bounded by a geographical and temporal context. Temporal, due to the fact that moments are defined as

finite events with specific starting and ending times. The geographical context is due to the fact that events not only occur at a certain time, but are also confined to a specific place or location.

Another important concept regarding Moments is their visibility. ***Public moments*** are those open for anyone to see and participate, while ***Private moments*** are those where an invitation is required to have access to the content and participate in them.

In a nutshell, a Moment is a group of people participating in a social function at a certain place, at a certain time, where they can share their experience by contributing with their content into the Application. Thus, they constitute the central and fundamental concept of the Application.

Connections:

A Connection is the most fundamental way of linking users to each other. Each user has a list of connections, which are the people he or she can invite to Moments he or she creates. In order to establish Connections, one user must send a Connection Request to another, and the other user must confirm or deny such request. If the request is confirmed, the Connection is established, and both users can start inviting each other to Moments.

Unlike Applications such as Facebook, Connections do not necessarily imply that both users are friends, or that they'll have access or visibility to everything each other does, but only the ability to invite one another to Moments. Moment visibility is independent from Connection status, and the rule is very simple: only those who participate in a moment have access to it.

inVites:

An ***inVite*** is defined simply as an invitation to a specific Moment. When a user creates a Moment, he or she invites other people to participate in them, and those people will in turn receive inVites to that Moment.

Joins:

Whenever users get invited to Moments, they gain the possibility to become participants for those moments. Similarly to real life social events, *inVites* to moments can be RSVP-ed. This means that the user receiving the *inVite* may decide to confirm the invitation, essentially saying "I will attend this Moment", or simply to ignore the invitation

altogether. The process of confirming a Moment *inVite* is what is referred to as a ***Join***, or (used as a verb) ***Joining*** a Moment.

Check-ins:

Another fundamental step in participating inside Moments is the process of ***Checking-in*** into a Moment. The term ***Check-in*** is used in a very similar fashion as it is for Apps such as Foursquare or Facebook, with some slight differences. It's similar in the sense that it implies indicating that the user who Checks-in is currently at a certain location. Nevertheless, when taking this to the domain of Moments, it indicates that a user is at that Moment's location at the appropriate time. This means that if a user Checks-in into a Moment, he or she is saying "I'm at the venue (location) **where** this Moment takes place, and at the time **when** it takes place".

Check-ins are then the way in which users actually are enabled to participate in Moments, since it validates that they are at the right place, at the right time, and are therefore part of the Moment that goes on in that context.

Faves:

A Fave is a very common and simple idea: it indicates that a user likes a certain entity from within the App.

Users may Fave Moments, indicating they like them, or content from those Moments, such as photos taken. This is a similar concept to the very common "Likes" of Facebook, and to which mobile application users have grown very accustomed.

Comments:

A comment is also a very common concept: a piece of text that is linked to an entity from the Application.

The most fundamental elements that users are accustomed to commenting (on Facebook, Instagram, and basically most content social Apps) are photos. This allows for users to comment on them and say whatever they wish to say about these photos.

5.3 Login

The first task an inJoin user has to go through, is the login or registration process. It consists on creating or accessing an inJoin account, which holds all the necessary user information for the App to work properly. The first thing inJoin presents is the Login view, which can be observed next:

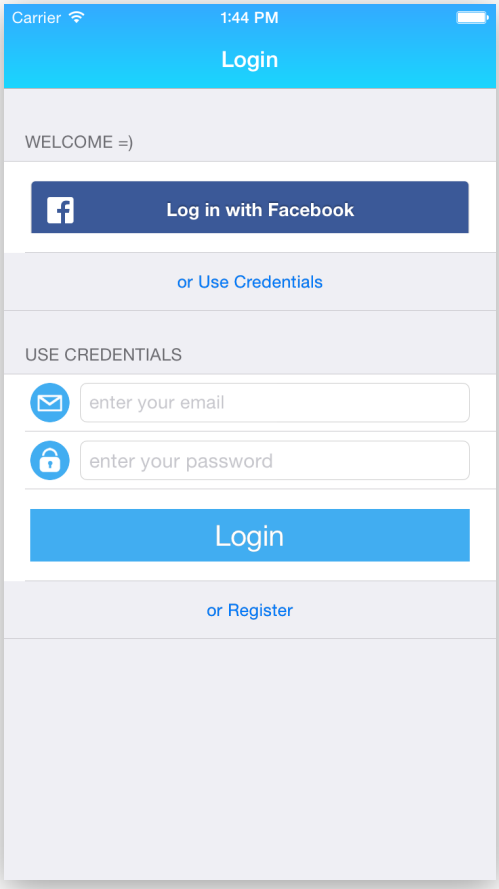

Figure 5.1: login view (iPhone 6).

As it can be appreciated from the figure above, the Login view is very straightforward. It offers two possibilities to log in: Facebook authentication, or regular authentication.

Facebook authentication requires a valid and active Facebook account, and it's used for both the Login or Registration process: if the user is detected to be a new user, then it registers using the required Facebook information (name, last name, email, profile picture). Otherwise, the user

is simply authenticated, and logged in immediately after. Facebook authentication is performed by either using the Facebook App to login (if installed on the device), or by opening a new web browser view, and asking to authenticate using the Facebook credentials, which will then redirect back to inJoin.

Regular authentication is as would be expected: it requires a unique email and a password to login (for already registered users), so there's not any further information required on how to perform this type of login.

It can also be observed that below the Login button, the user is presented with the option to ***Register***, so when tapping there, the App will present the registration form, which looks as follows:

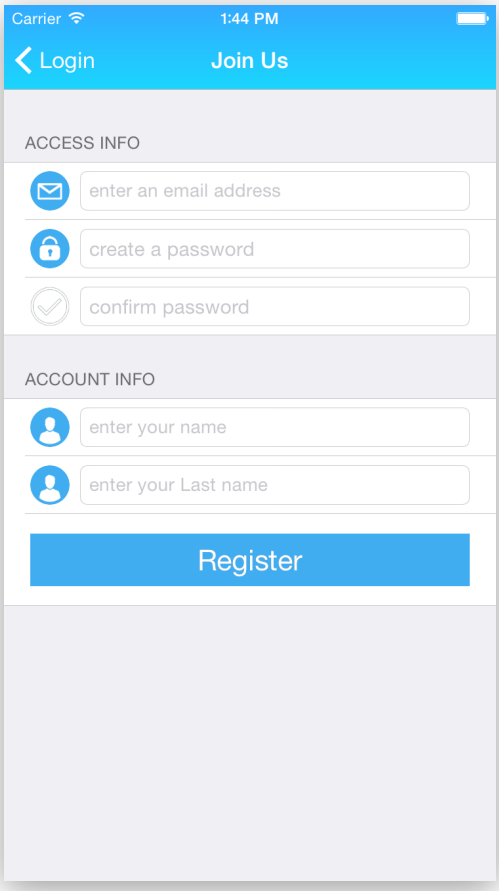

Figure 5.2: registration form (iPhone 6).

It is easy to observe the information required for registering a new account:

- An email, which is used as the identifier for the login process.
- A password, and a password confirmation.
- The person's name and last name, since this project aims for a social network of real people, in real life scenarios, it's encouraged to use the person's real name, so that other people may connect with him or her.

Once the registration process succeeds, the App will automatically perform a login, and present the main view of inJoin to the user, who's now ready to create and participate in all kinds of Moments.

5.4 Navigation

As with any Application that presents a variety of well-defined sections for different tasks, inJoin utilizes a navigation strategy in order to allow users to better move through the App in the least intrusive manner.

For this purpose, inJoin presents a sliding left panel menu that, when visible, it is shown as follows:

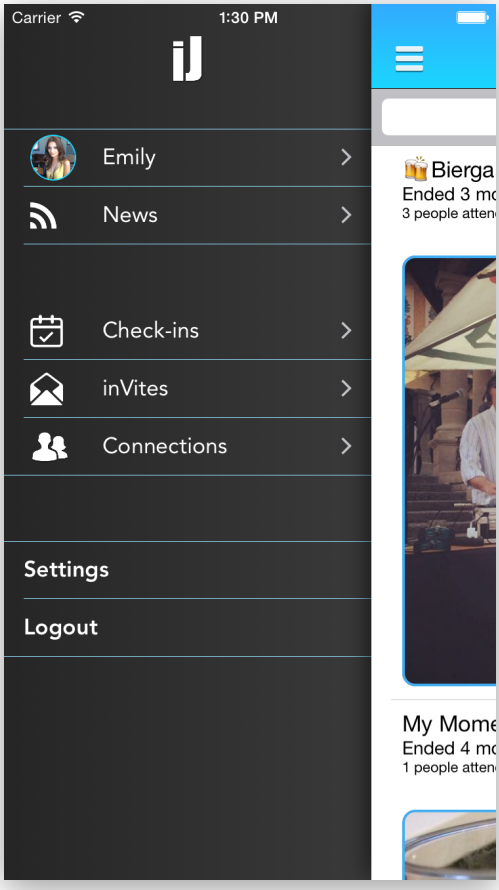

Figure 5.3: Main navigation menu panel opened (iPhone 6).

Notice at the top right corner of the screen, the button with the three white lines indicating the availability of the left panel menu, is presented to the user. By tapping it, the left menu panel will slide into visibility from the left side of the screen. An alternative and convenient way of presenting it, requires the user to swipe from left to right, and the menu

will also be presented. Similarly, as it would be expected, swiping right to left will hide it.

The menu will display several options to the user, in the order (from top to bottom) of: Me, News, Check-ins, inVites, and Connections sections (the main sections of the App) and finally a Logout option.

This strategy is the one employed to navigate easily throughout the entire application, and provide users with an easy way of getting to where they want to go inside inJoin. Next, readers will be provided with a more in-depth look into how each section works and how to use its features.

5.5 "Me" section

The section named "Me" consists on the public profile of a inJoin user inside the Application.

It displays some basic information, such as the current profile picture for the user and email address used for registration. This can be observed next:

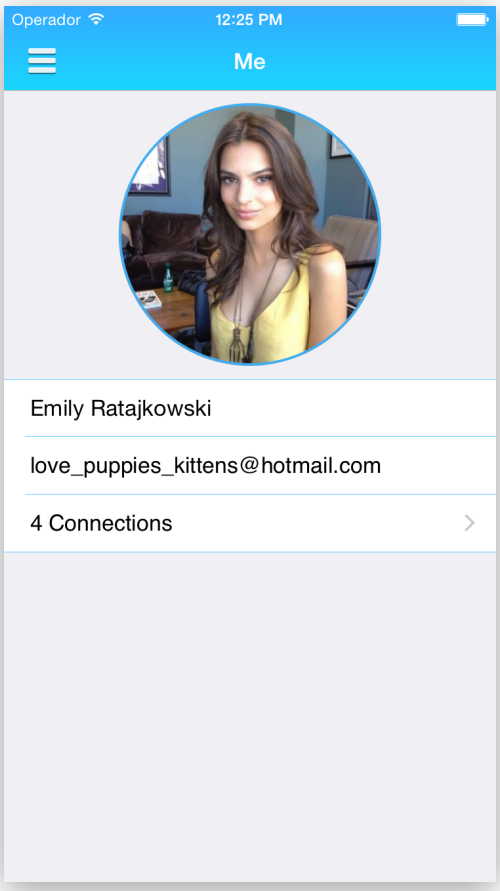

Figure 5.4: User profile displayed from the "Me" section (iPhone 6).

By tapping on the profile picture, the App navigates into a list of all the available User Profile pictures that have been uploaded into the inJoin platform:

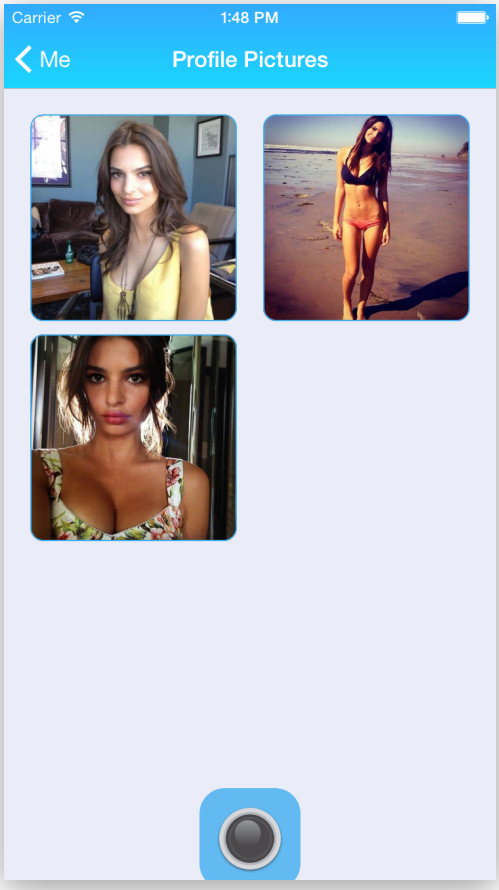

Figure 5.5: User Profile pictures list (iPhone 6).

From here, users may upload new User Profile pictures, by tapping the camera button below the list, which allows for taking or selecting a new picture from the phone's camera or camera roll, respectively.

Furthermore, User Profile pictures are available in full screen mode, whenever users tap on any of the pictures on the list. Pictures in full screen mode look as follows:

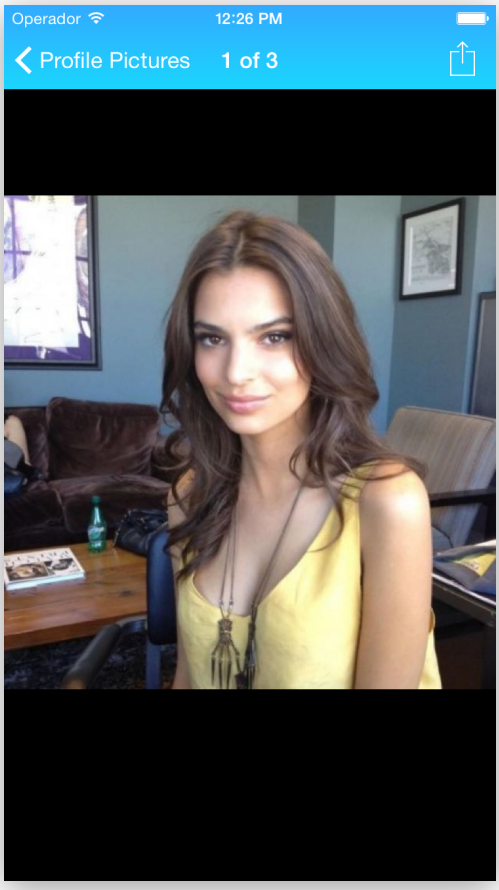

Figure 5.6: Full-screen User Profile picture (iPhone 6).

Additionally, there's a quick access to inspect all Connections, by tapping on the "connections" button. When inspecting the profile of other users, there is a bit more information to observe. First, users can see common connections with other users, by tapping on "Common Connections", as well as the "Common Check-ins", which displays the Moments where both users have attended and participated, in a way indicating when and where 2 users have "crossed paths".

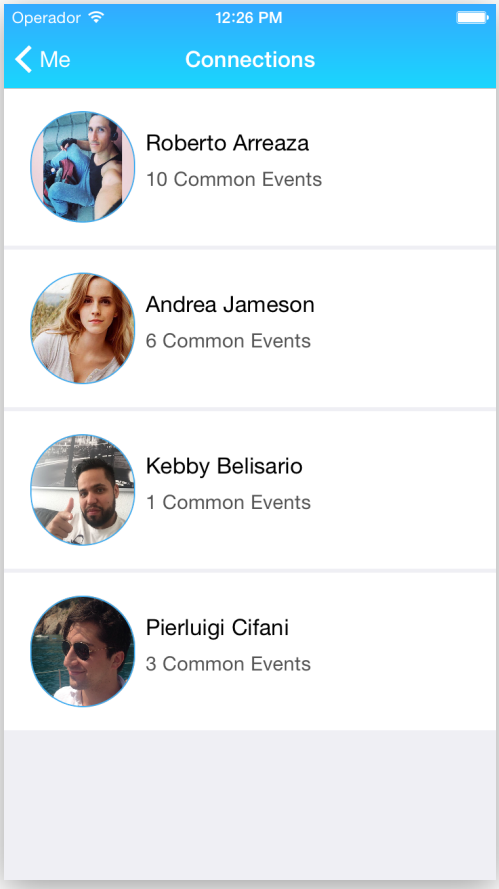

Figure 5.7: Connections (iPhone 6).

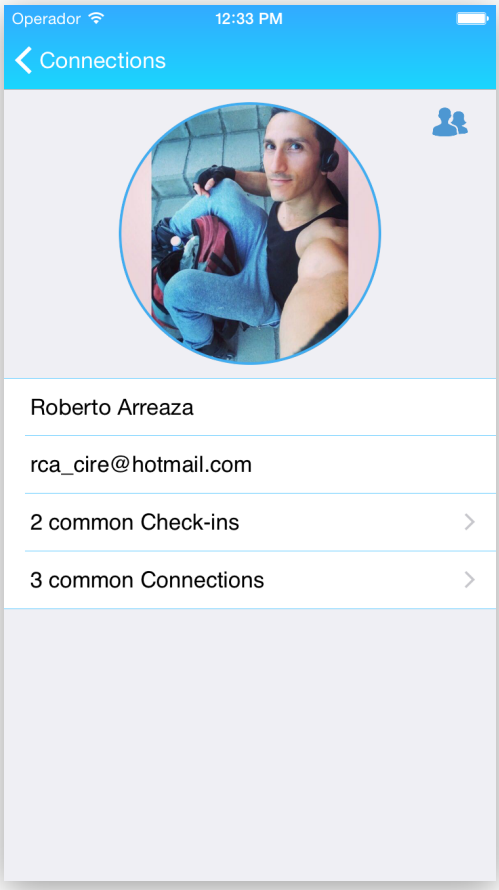

Figure 5.8: User profile of a Connection (iPhone 6).

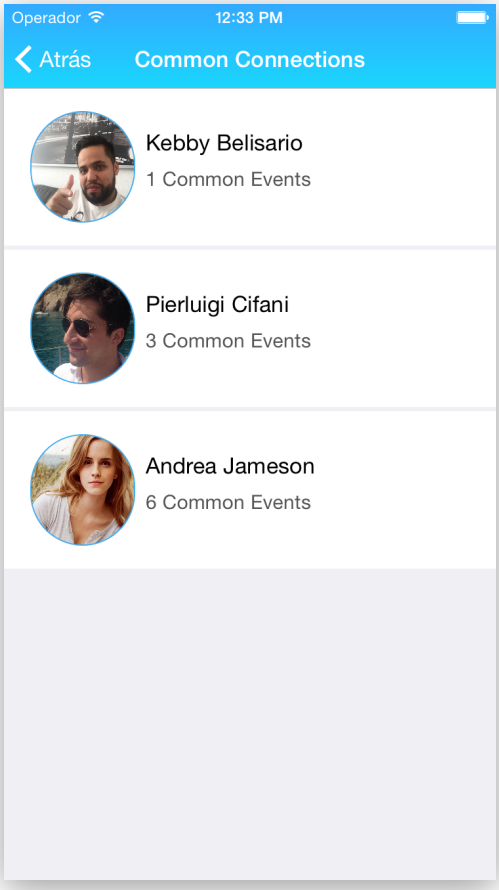

Figure 5.9: Common Connections (iPhone 6).

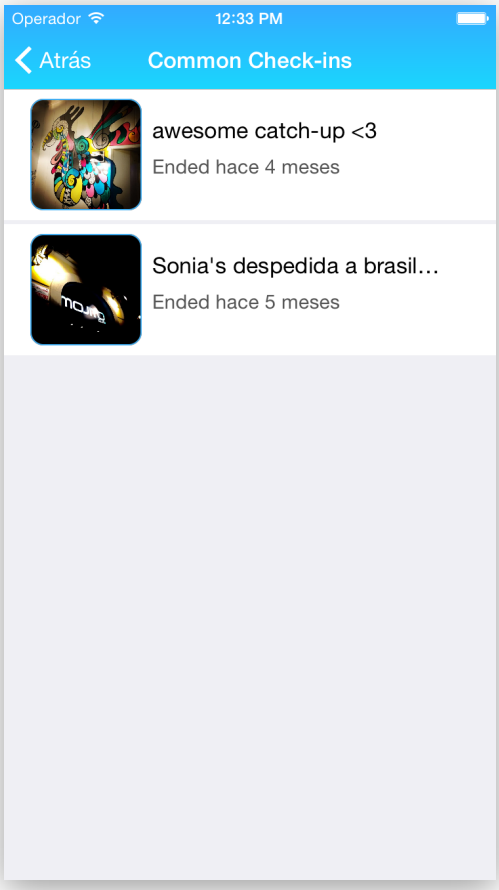

Figure 5.10: Common Check-ins (iPhone 6).

This sums up the functionalities and usage of the "Me" section showing user profiles and general information about inJoin users. Coming up next, is one of the core sections of the app: the News section.

5.6 "News" section

A variety of Photo Applications in the current Apps market have some sort of photos Feed. The concept of News Feed, widely popularized by Facebook, and now also present in most of the popular Applications (Instagram and Google+ are another common examples). The reason why it has become such an important part of these Applications is due to the fact that it allows users to discover and find out about new content in real time, as it's being created and shared in the Application.

This same principle applies to inJoin, since users find appealing knowing about relevant Moments happening recently around contexts of their interest. Examples of this might be Moments happening soon to which a user has been recently invited, or public Moments trending right now, or Moments where users are participating right now, displaying live content, as well as Moments of a certain type which are interesting for the user.

This is the reason behind the definition of the ***News*** section of the App. As stated before, it displays a Feed of Moments of relevance to the user, in a timely manner.

This constitutes the main Section of inJoin, as it is the starting point of the App, as soon as the login is performed. It displays a vertically scrollable list of relevant Moments, as shown next:

Figure 5.11: News main screen (iPhone 6).

There are a couple of things to note here. Firstly, some basic information about the Moment is presented, with the Cover Picture as the main element to display, but also the Moment's title, when it starts/ends/ended (depending on when it took place in reference to current date), as well as how many people are participating in it. In the upper-right corner a lock or world icon displays as well, indicating whether the Moment is private or public, respectively.

From here, users also have the ability of searching for Moments by name (title), in case the users wishes to look up a specific event.

In addition to the main purpose of *Discovering* Moments, the News section allows for a few other fundamental tasks in the Application, explained next.

i) Creating and managing Moments

It can be easily observed from the News screen, the presence of a button located at the lower part of the screen. This allows for creating new Moments. When tapping on it, a dialogue will present two options: quick Moment and Detailed Moment.

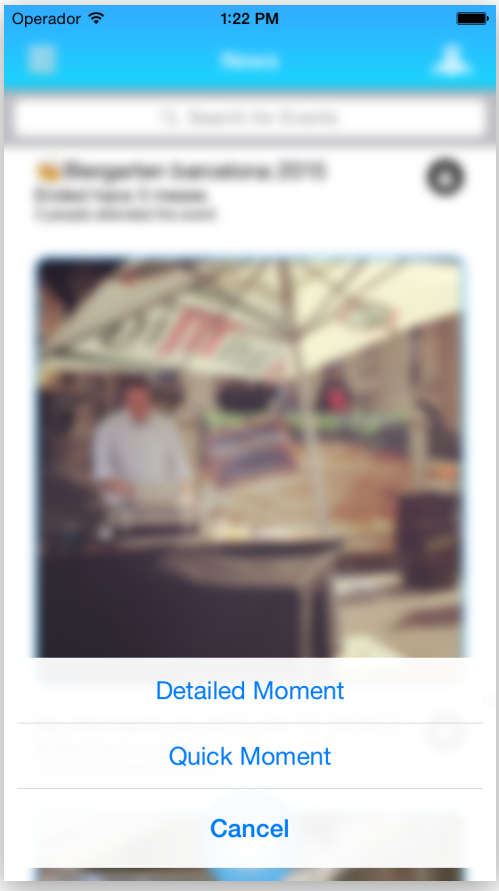

Figure 5.12: New Moment creation dialogue (iPhone 6).

Detailed Moment:

This constitutes the conventional way of creating Moments. A Form will be presented to the user, asking for all the information needed to create the Moment. It is presented as follows:

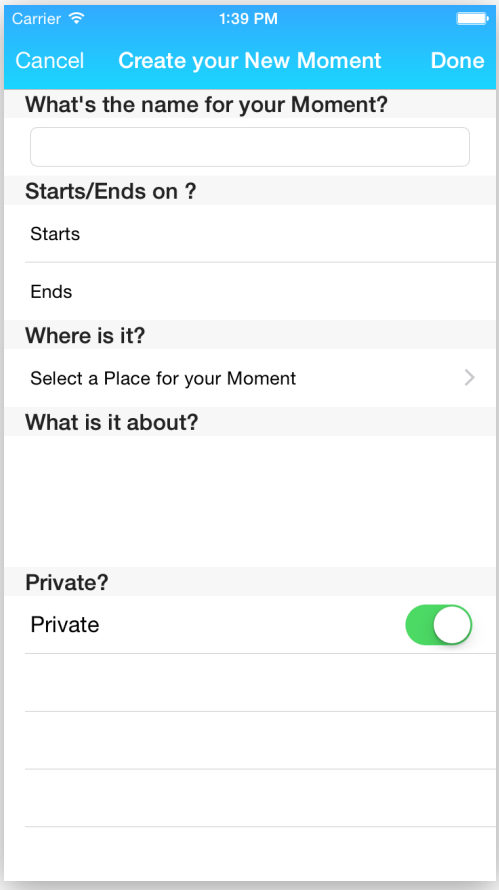

Figure 5.13: Create new detailed Moment view (iPhone 6).

Here can be observed all the fields available to fill out when creating a Moment. The title for the moment, a description, the start/end times, as well as the location for it and a switch indicating whether it's private (invitation only) or public (open to anyone). Only the description is optional. Pretty much all fields are self-explanatory, with the exception of the location.

When tapping on "Select a place for the event", the users is taken to a new view, which looks as follows:

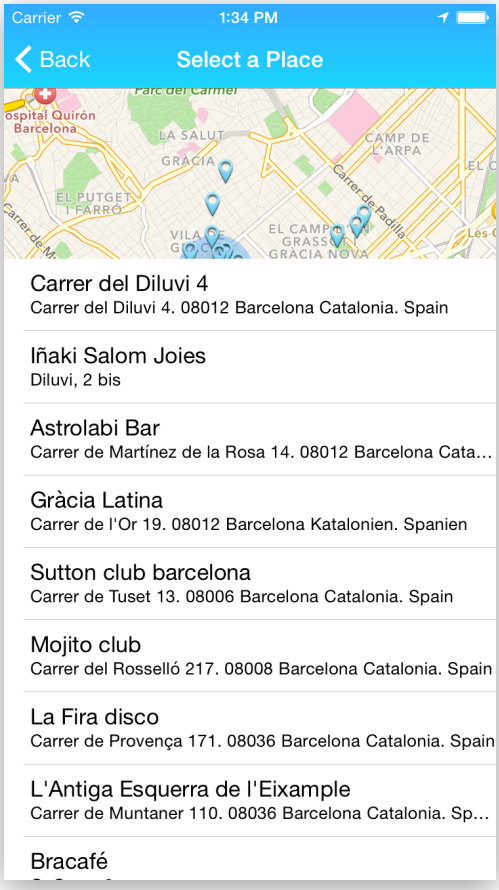

Figure 5.14: Selecting a venue for the Moment (iPhone 6).

It shows a list of places with a Map View on top of it (centered on the current location of the device). The map can be expanded to perform the search for a place on the map, or if the place desired is already listed below, it can be selected directly. An example of the expanded map is shown below:

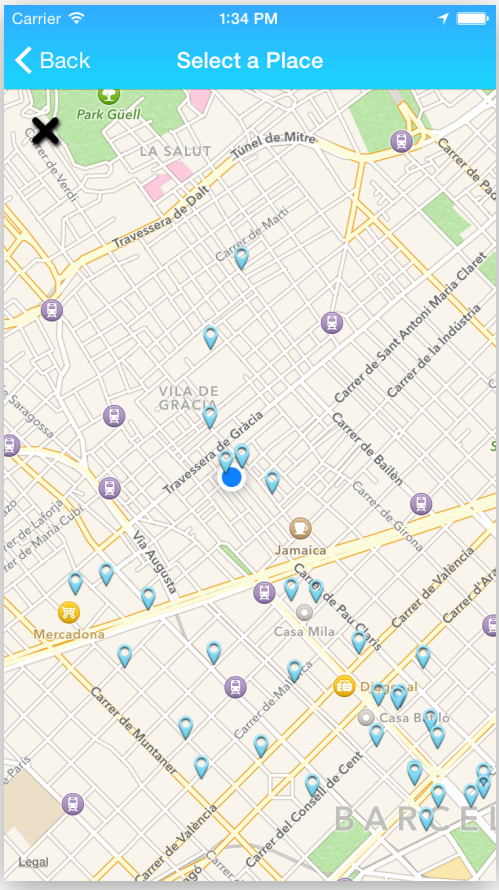

Figure 5.15: Expanded map on venue selection for a Moment (iPhone 6).

From the map, by tapping on any on the pins, the details about the place will be shown, and the "+" (add) icon shown is used to add this place to be used as the location for the Moment being created, as shown in the following figure:

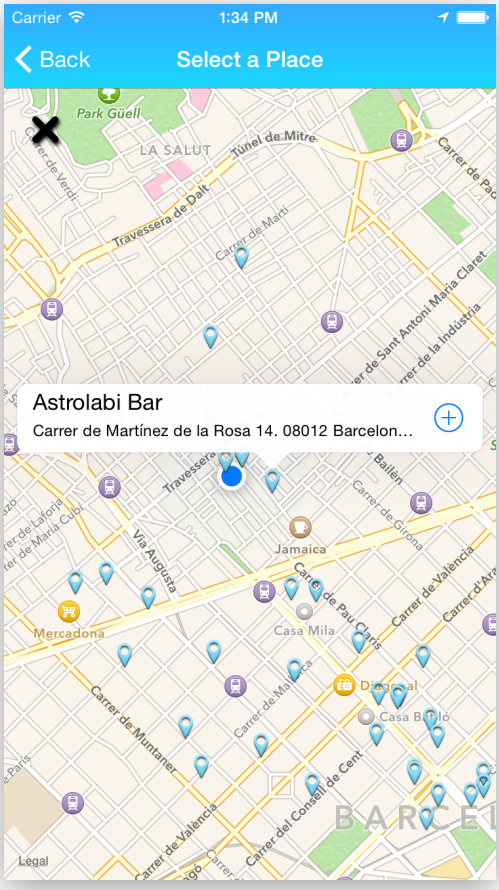

Figure 5.16: Selecting a venue for a Moment from a pin on the map (iPhone 6).

Places are listed from the inJoin Database, as well as from the Foursquare Database, which means many well-known places will be already available to use. However, users might not find the exact place they wish to use. For this case, inJoin allows users to create new places.

In order to create a new place, and from the map, users just have to find the coordinate by navigating through the map, and once found, they must tap&hold for a couple of seconds, and a dialogue for place creation will appear. It requires both a name for the place, as well as the radius (in meters) for it, to determine how big it is. Once filled out, the place s created in the map by tapping "done". Then from the map, it can be selected for using as the Moment's location in the same way done for any other existing place (tapping on the pin, and then on the "+" button).

After selecting a location for the Moment, the App will take the user back to the Moment creation form, and once everything is ready, it will create the moment after tapping on "done" at the top right corner. If successful, this will take the user directly to the newly created Moment.

Quick Moment:

Detailed Moments are a great way to create events planned ahead of time, for maybe something happening tomorrow, in a week, or even a year from now. There are times, however, when Moments are very spontaneous, and asking the user for so much information related to it can be very cumbersome, when all they wish is to share a picture quickly with other participants: this is why inJoin defines Quick Moments.

When selecting to create a Quick Moment, the only thing required from the user, is the location for it, so it shows a dialogue similar to this:

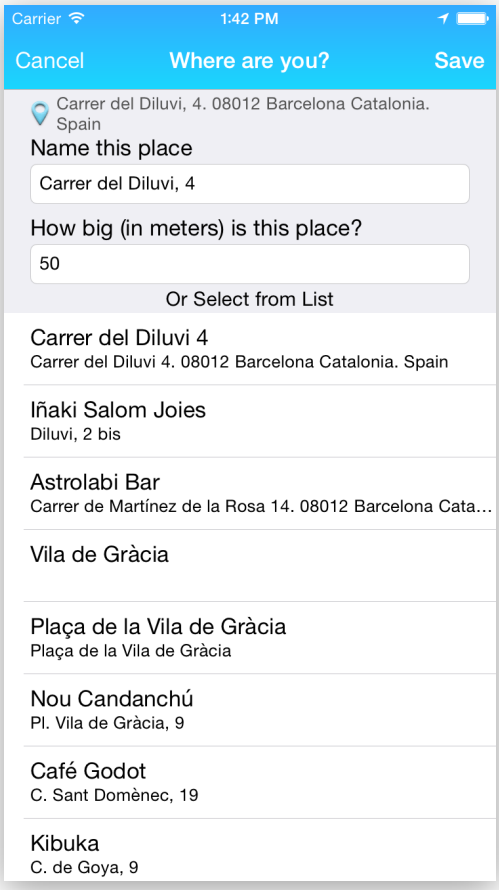

Figure 5.17: Quick Moment creation view (iPhone 6).

As it can be verified, the device's current location is used to show an approximate address for a place, and sets a default radius, as well as listing all places nearby that location. All the user has to do is look through the list and find the right place, or if it's not there, to provide a name (if the default address does not suffice) and the radius (if the default is not accurate), and by taping "done" the moment will automatically be created.

The way this works is by providing default values to all other fields, so it gives a default title to the moment, a default description, it sets it to be private, uses the current time to set the start time, and a default span of 2 hours from now for the end time. All of which can be edited by the creator later on, thus focusing on the spontaneity of the Quick Moment, in order to empower users with the ability of sharing their pictures as quickly and as painless as possible.

ii) Participating in Moments

As important as discovering Moments and creating one's own, probably the most important part of sing the App consists on actually participating in them.

A user participates in a Moment by posting and sharing photos into that Moment. This implies that in order to participate, a user must have visibility of that Moment. Users have visibility to all public Moments, and to private Moments to which they've been invited, or in case they created the Moment in question.

Another important factor that plays an important part in Moment participation is the ***status*** of the Moment. A Moment is considered as "not started" when the start date for it has not yet been reached. A moment is considered as "ongoing" when the start date has passed but it still has not ended yet (still taking place). Lastly, a Moment is considered "ended" when the end date has passed.

Users can then participate -under certain conditions- differently for each status of the Moment.

Once an Event has been created or accessed via the News section, users can inspect such Moment by getting into its details (when creating a

new Moment, the redirection to its details happens automatically). The details of a moment look as follows:

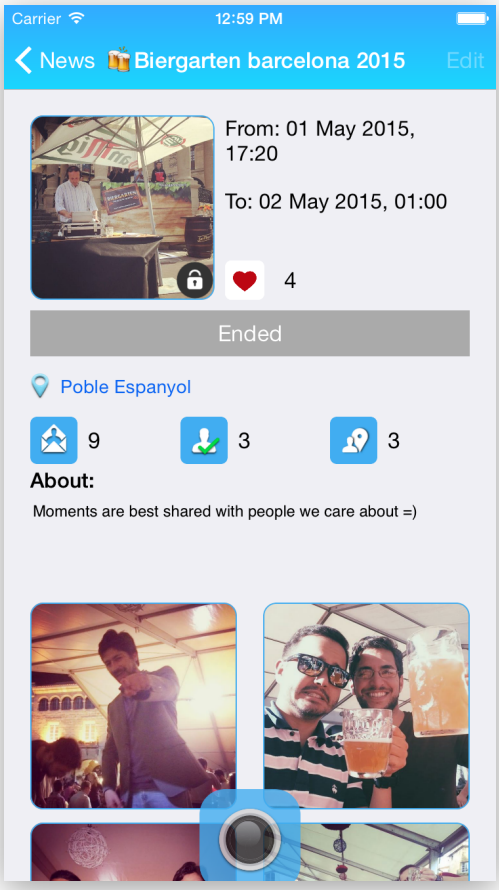

Figure 5.18: Moment details view (iPhone 6).

The first thing to observe is the main information about the event. Users can see the title, description, when it starts/ends, as well as the "cover picture", which is the main picture used when presenting the Moment in the news, and in general anywhere needed to represent a Moment in the App. Right below this information is the collection of photos available inside the Moment, as all participants post their pictures.

Additionally, there's a main action button, which allows users to participate in the Moment by doing a ***Join*** or a ***Check-in***. Since only a single action button is required and used, its state will depend on whether the user has performed a Join or a Check-in, as well as the state of the

event itself (not started, ongoing or ended). The state of the button can be depicted in the following table, for a better comprehension of this:

	Not Started	**Ongoing**	**Ended**
Not Joined	Join	Join	Ended
Joined	Joined	Check-in	Ended
Checked-in	N/A	Checked-in	Ended

Figure 5.19: table showing the state displayed by the action button depending on Event status (Not started / Ongoing / Ended) and user participation status (not Joined / Joined / Checked-in).

This basically means that when the user has not yet Joined the Moment, the button allows to Join, and only if the event is Ongoing the button will allow users to Check-in as well. It also indicates when the event has ended. Some examples of these button states can be observed next:

Figure 5.20: status samples for the main action button to Join or Check-in into Moments.

A side note on the Check-in operation: check-ins can only be done if the User is at the location of the Moment, and once it is ongoing. Check-ins are validated by using the Device's location, thus ensuring that only real participants are able to take part in them.

The explanation above holds relevance due to the fact that a user can only participate (share photos) in a Moment, if:

- The Moment has not started, but the user has Joined it: this allows for posting some content as preview for the Moment, but only for those who have stated they mean to be there when it starts.
- The Moment is ongoing, and the user is Checked-in into it: this is the obvious way of participating, indicating that the user is

physically present at the time it takes place, and is therefore allowed to share.

- After the Moment ends, and the user did Check-in: most people do not publish all pictures of an event in a timely manner. By allowing users to share pictures after the Moment ends, but only for people who attended, inJoin permits people who attended to post some after-the-fact pictures.

This how users are able to participate in any kind of Moment they choose. In order to share a photo in a Moment, from the details of the Moment, it's just necessary to tap  at the bottom of the view. This will show the camera, which allows users to either take a picture straight from there, or to choose a pre-existing one from the camera roll of the device. The camera view presented be observed next:

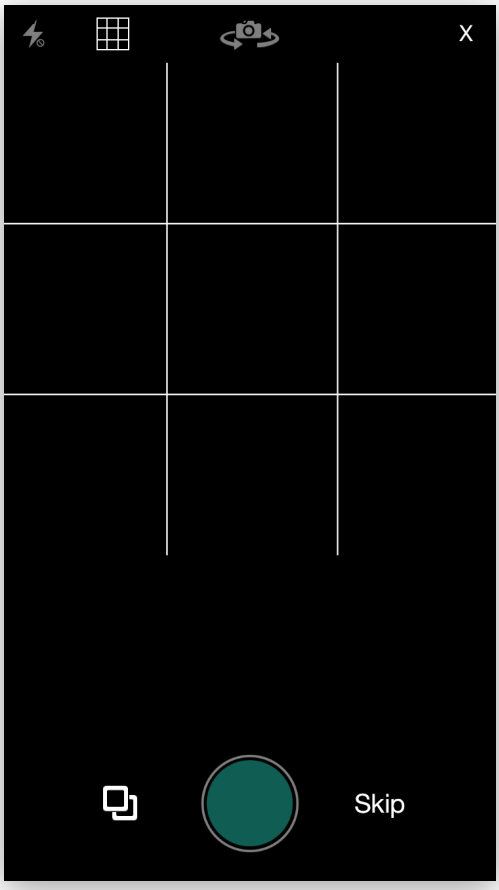

Figure 5.21: Camera view (iPhone 6).

Apart from participating in a Moment, and reviewing the Moment's information and shared content, there a couple more interesting things users can do from this Moment details view. One of them –naturally- is to find out who else is participating in it.

By tapping on the "invited" icon  , users can see who has been invited to this Moment, so users are presented with a dialogue such as the following:

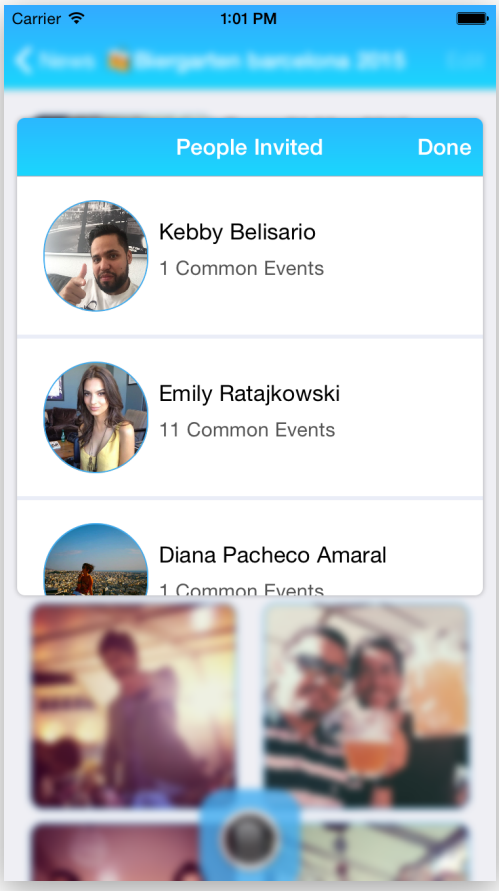

Figure 5.22: Invited people view (iPhone 6).

Here, it can be observed a list of the users that have been invited. It can be navigated to inspect users profiles, and even add them as connections straight away.

Similarly, by tapping on the joined icon  , users have the ability to see which of the invited users have Joined the Moment, indicating they'll attend it by the time it starts. A Similar navigable list of users is presented, which looks as follows:

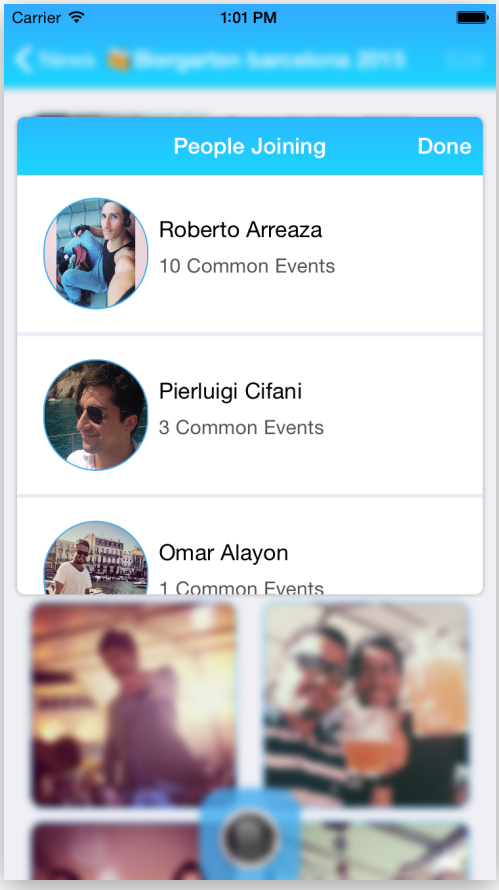

Figure 5.23: Joined people view (iPhone 6).

Furthermore, when users tap on the "Checked-in" icon  , they're presented with the list of the users that Checked-in into the Moment, meaning they have been present by the time it took place. The dialogue of Checked-in users is shown below:

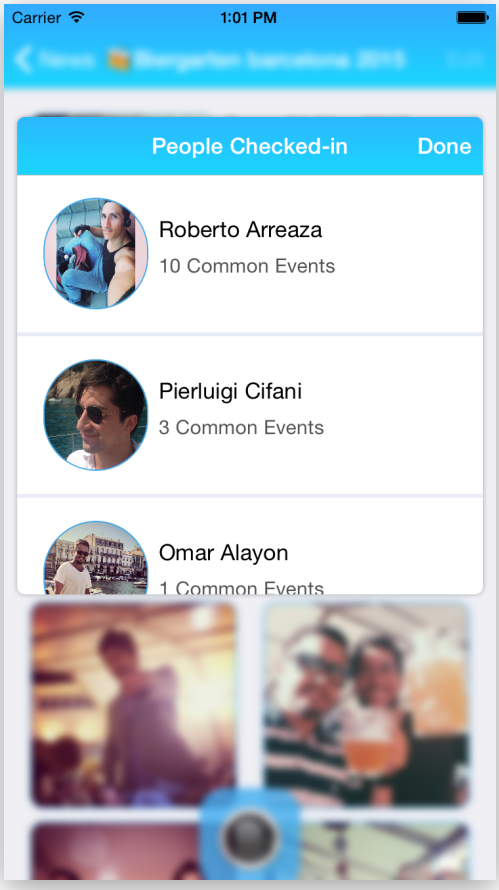

Figure 5.24: Checked-in people view (iPhone 6).

All of this functionality constitutes the standard way of getting involved in moments as normal participants. Nevertheless, creators of a Moment also play a key roll in some operations, which are also available from the details of a Moment. The first thing to note, is that creators posses the ability to edit the details of a Moment. This is done by tapping on the "edit" button at the top right corner of the view, which in turn leads to an edition form that looks very much like the one used for creating Moments:

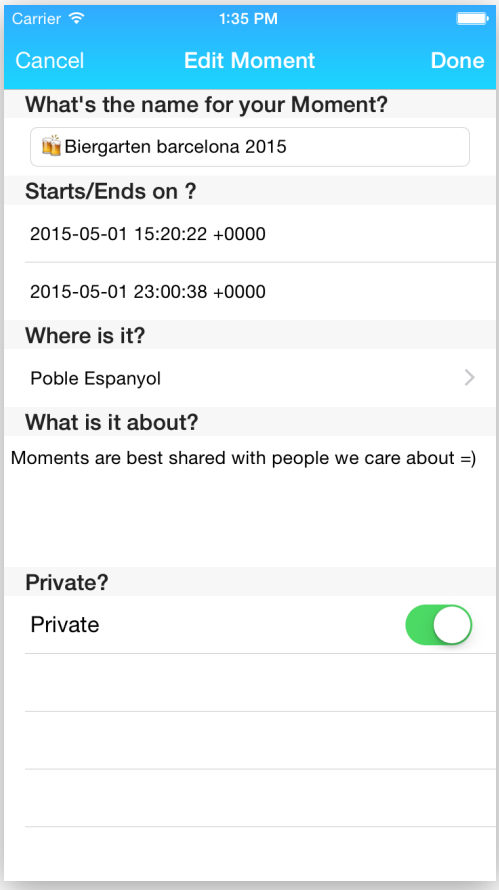

Figure 5.25: Editing a Moment (iPhone 6).

As it is verifiable, creators may edit any details about the event, which allows for a lot of flexibility, and is a key aspect of how "Quick Moments" –discussed in "Creating and managing Moments"- work.

Another operation reserved only to creators consists on "setting the cover picture" of a Moment. As mentioned in "Creating and managing Moments" each Moment has a main picture, which is presented in the News of other places to represent that Moment throughout the App. This main picture is known as the "cover picture". Creators have the ability to set and change the cover picture for the Moment whenever they wish, although there's a feature that auto-sets the cover picture to the first picture uploaded into the Moment, if no cover picture has yet been set. If no cover picture has been set, creators may tap on the cover picture icon,

and it will lead to the camera view of the App, which in turn is used to upload and set it.

The last (but highly relevant) creator task is to invite people to Moments. After all, this is key for collaborating. When a creator taps on the "invited" icon  , a different form shows up. This time, it not only lists the people who have been invited, but lists all of the creator's connections. In this manner, they can select which people they will invite to the new Moment. The form sheet shown looks as follows:

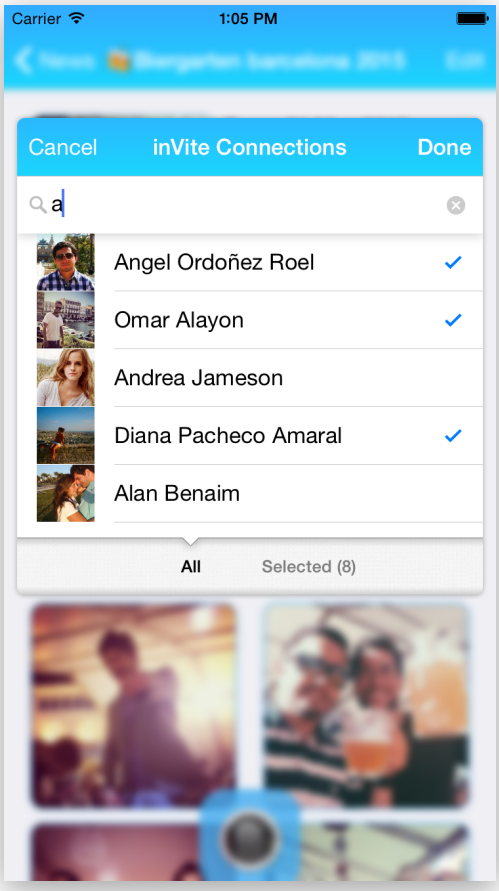

Figure 5.26: Inviting people to a Moment (iPhone 6).

In the figure above, it can be appreciated that it not only lists all connections, but also allows for a text search to quickly find any connection's name quickly and efficiently. It also has the option to only view the people selected (meaning, the people to invite, or the ones that

are already invited). By tapping "done" any newly-added user will get an invitation to the Moment, enabling them to participate in it.

iii) Interacting with Photos

Since Facebook incorporated "likes" and "comments" into pictures, and due to its wide acceptance by their users, most social photo sharing apps already incorporate similar mechanisms to interact with photos. This created a sort of de-facto way of making photos social in the digital domain. The Application presented here incorporates those and a few more ways to interact and operate on photos.

From the details of a Moment, users can enter the details of a photo by tapping on it. The result is shown next:

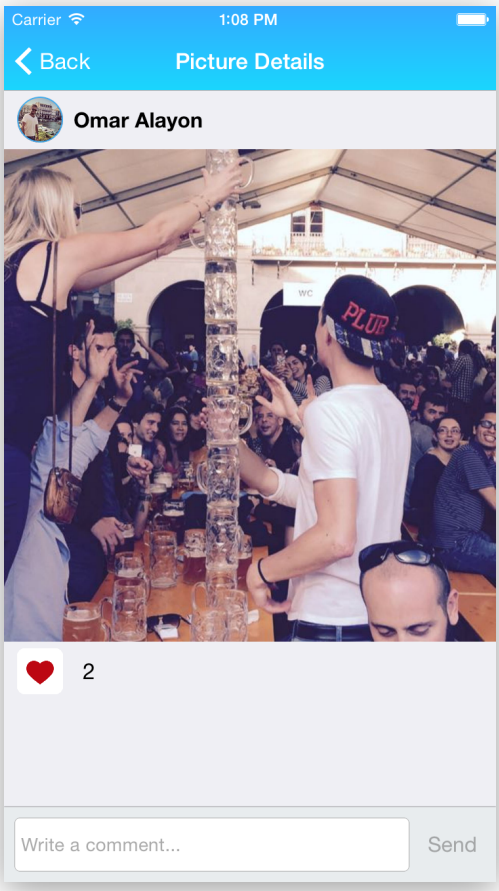

Figure 5.27: Picture details view (iPhone 6).

As expected, users have the ability to "fave" photos by tapping on the button right below the image, as well as to comment about the photo by simply typing on the text box sticking to the bottom of the view, and the comments will load below the photo. Additionally, users may also double-tap on the picture to fave/unfave it. An example of how a photo with faves and comments looks is presented below:

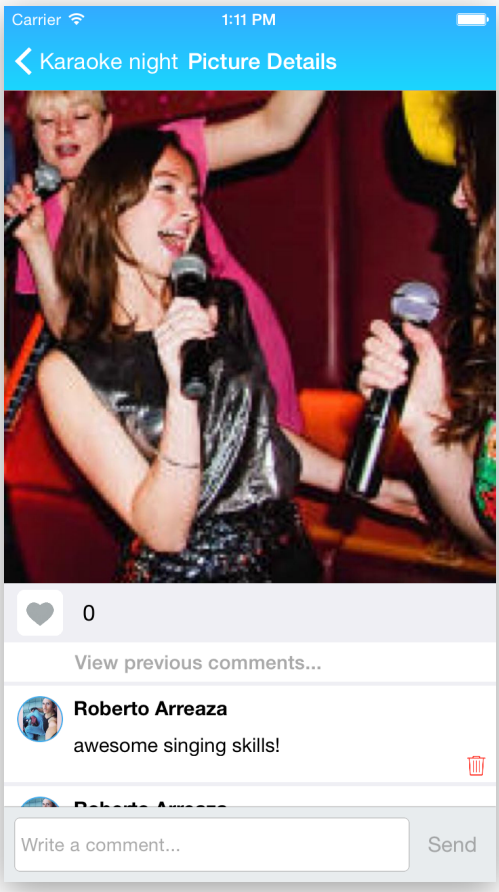

Figure 5.28: Picture example with paginated comments (iPhone 6).

It's worth noticing that for photos with many comments, users may tap on "view previous comments" to view more of the existing comments for the photo.

For an application with the intent of making collaboration easier, it is essential to allow users to download and save any photo shared in a Moment into their device. By doing tap&hold over the photo, a options dialogue is presented to the user, which looks like this:

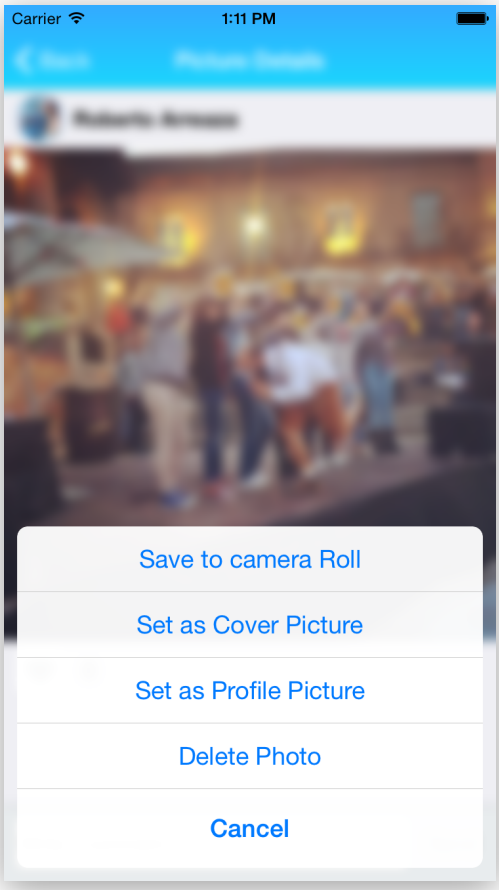

Figure 5.29: available Picture actions (iPhone 6).

As it can be observed, there are a few tasks that may be performed:

- Save to camera roll: as mentioned above, users may save the picture into the camera roll of the device, in the original resolution held by the server.
- Set as cover picture: this is only available for the creator of the Moment, and it allows this user to set this picture as the cover picture for that Moment.
- Set as profile picture: an extra bit of functionality, it allows a user to set this picture as his own profile picture, which might be useful in some cases.
- Delete picture: this options is only available to the user that created and posted the picture into the Moment, thus allowing

people to remove any undesired or maybe accidentally posted photos from a Moment.

This concludes the main tasks that Participants of Moments have available to them inside inJoin, once they have access to the details of a moment, may that be by invitation, or for publicly-available Moments. It constitutes the core of the Application, but there are still some relevant sections that are key to making everything work properly, thus contributing to a more pleasant usage experience for people.

5.7 "Connections" section

Generally speaking, social Applications will always have one thing in common: they generate a community around a certain topic, and facilitate people's connections and interactions that nurtures and cultivates this community. This is why there is not social Application without the ability to make users connect.

And inJoin is no different. This is why the ***Connections*** section is such a fundamental part of the Application. From within this section, a user has the ability to both inspect his or her existing connections and access their profiles, or to discover new people and connect with the as well.

The Connections section view looks as displayed next:

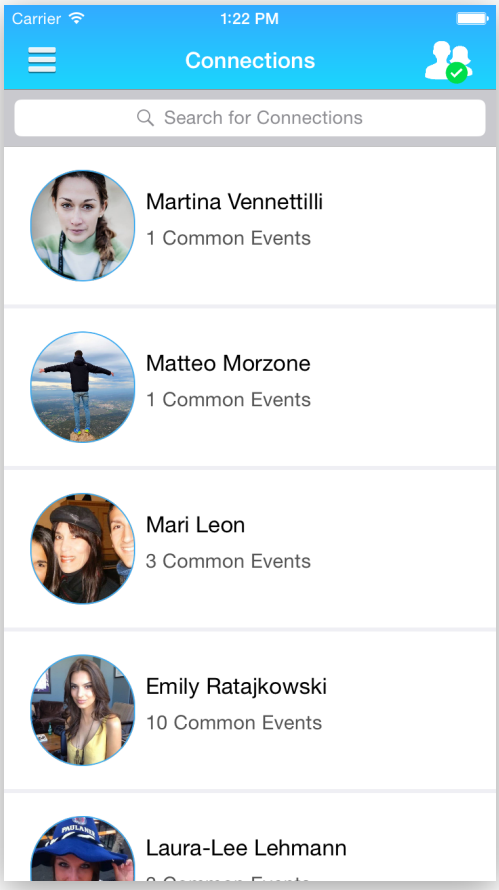

Figure 5.30: Connections section (iPhone 6).

i) Browsing through connections

It is easy to observe that all the existing connections are laid out on a list, thus making it easy for users to browse through their connections and inspect their profiles. This is achieved by tapping on a connection from the list, which leads to the user's profile, as depicted below:

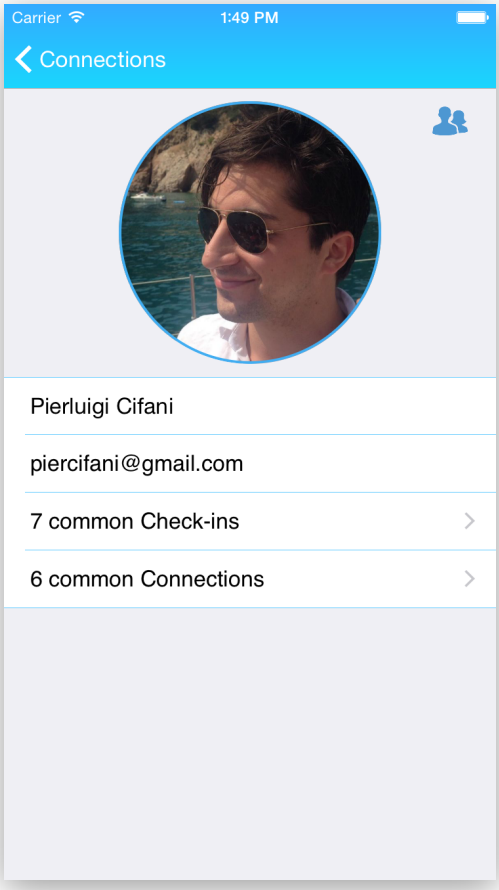

Figure 5.31: Inspecting other user's profiles (iPhone 6).

As it can be appreciated, a user's profile looks very similar to the logged user profile, with a few exceptions:

- When tapping on the user's profile picture, it also leads to the list of all available profile pictures for that user, but –naturally- there's no camera button allowing for uploading a new one, since the profile does not belong to the logged user.
- At the top right corner, the icon indicates that the user being inspected is already a connection. When inspecting user profiles that are not currently connections, the icon shown is , indicating that if the user taps on it, it will send a Connection Request. When a Connection Request has been sent already, but not yet confirmed by the receiving user, the icon indicating such state will then be .

- It's also easy to appreciate the number of "common Check-ins", which directs the App to a list of Moments where the logged user and the user whose profile being inspected, have both Checked-in, thus being a way to find out where two users have "crossed paths" in real life. The list displayed looks as follows:

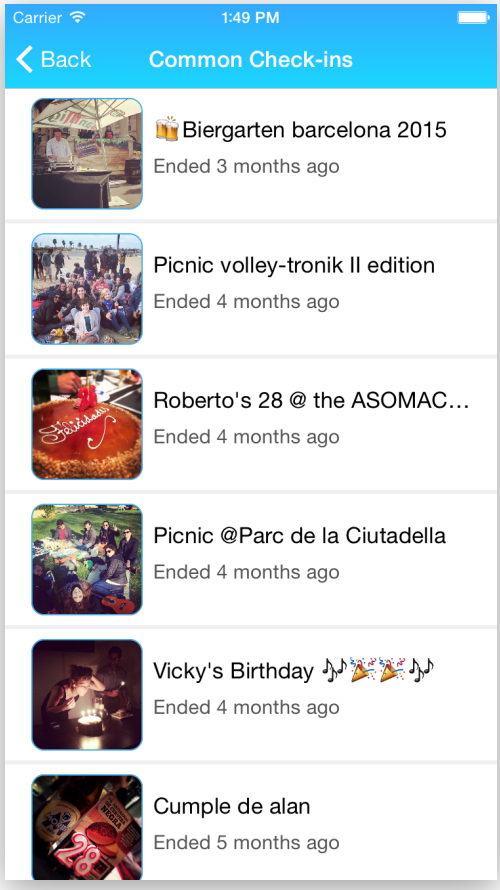

Figure 5.32: common Check-ins example (iPhone 6).

- Finally, the list of "common Connections" which how the name indicates, lays out all the users which are connections to both the logged user and the user whose profile is being inspected. An example of this list is presented next:

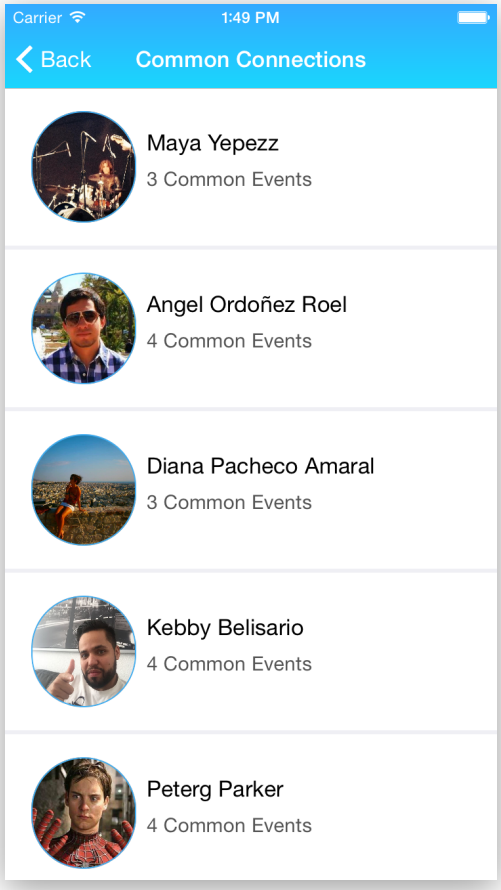

Figure 5.33: common Connections list (iPhone 6).

ii) Searching for users

At the top of the Connections view, it's easy to spot a search bar prompting "search for connections". It can be used to search on the entire inJoin database of available users, in order to find any user by name or last name. This is very useful when searching for people to add as connections. An example search is presented in the following figure:

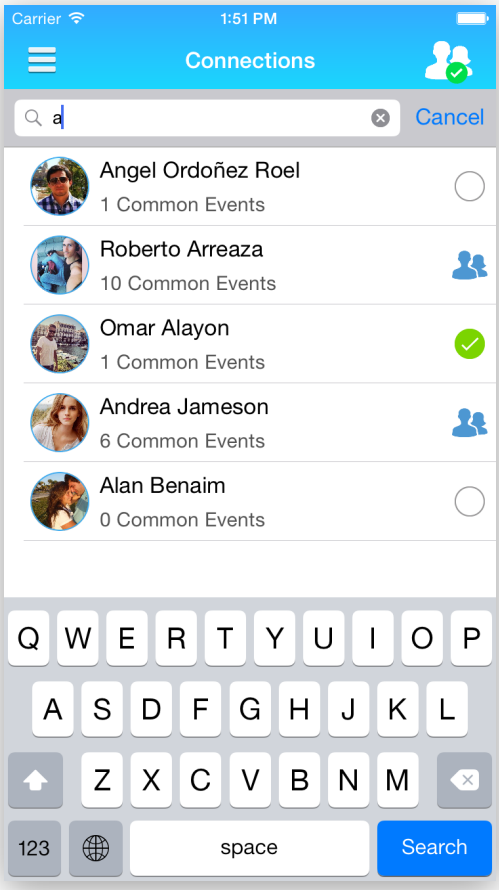

Figure 5.34: search for inJoin users example (iPhone 6).

As it can be appreciated, the search lays out all results of users found under a certain name or last name matching the characters typed, and it also shows some additional information:

- The name of the user and his or her profile picture.
- The number of common Moments between the result user and the logged user (indicating the number of Moments for which they've both been invited)
- The connection status icon, such as the one displayed on each user's profiles, allowing to add connections easily or to determine the connection request status (user's already a connection or the request is pending for confirmation). Once again, the icon (Connection) indicates the user is already a Connection. On the

other hand the icon (send Connection Request) indicates is the user taps on it, a new connection Request will be created for the resulting user, and while the request remains pending confirmation, it displays as (Connection Request sent).

Apart form this, if the user taps on any of the resulting users from the search, it will take the App to the user's profile, and it can be explored in the same way explained above in "Browsing Through Connections".

iii) Confirming Connection Requests

As mentioned previously, when a users sends a Connection Request to another user, the latter must confirm it in order for them to become Connections.

As it would be expected, this confirmation process is also done from the Connections section. Tapping at the top right corner, presents the "Confirm Connections" view, which may be inspected in the following figure:

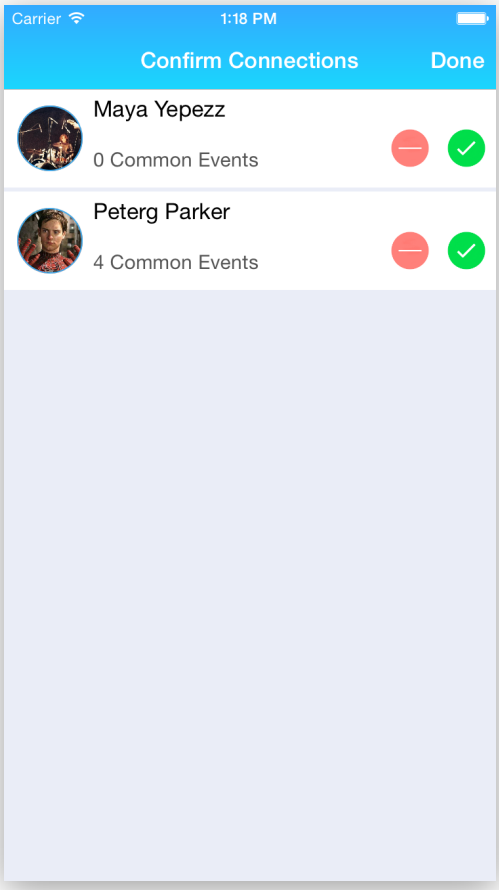

Figure 5.35: confirm Connections view (iPhone 6).

If there are any pending Connection Requests, they'll be listed in the view, and by tapping on ✓ or —, users can confirm or deny the Connection request, respectively. Users that are confirmed, will then appear listed as Connections for the logged user from this moment onwards. An example confirmation/denial of Connection Requests is presented next:

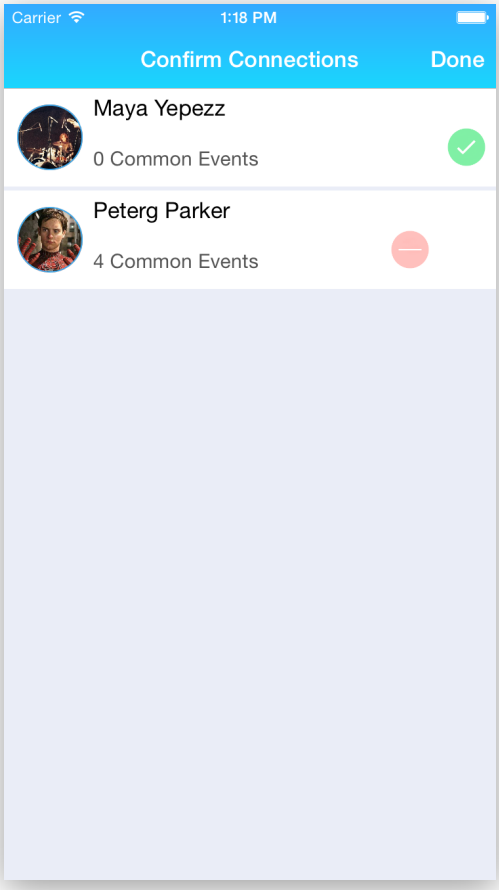

Figure 5.36: confirming/denying Connection Requests (iPhone 6).

As a final remark on the relevance of Connections, it's important to highlight that when a user creates a Moment, he only has the ability of inviting Connections to it, thus making it an essential part of inJoin. The invitation process is covered in **Participating in Moments**, as part of the **News section** of this document.

5.8 "inVites" section

As it has been stated in previous sections of this document, the concept of inviting people to Moments is of the outmost relevance, and Moment visibility is built upon this intuitive strategy: a user only has visibility to a private Moment if he or she has been invited to it, by receiving what in inJoin terms is known as an ***inVite***.

For this very reason, inJoin defines an inVites section, where users are able to review a historic list of all Moments to which they've been invited. The list is chronologically ordered, and is depicted in the following figure:

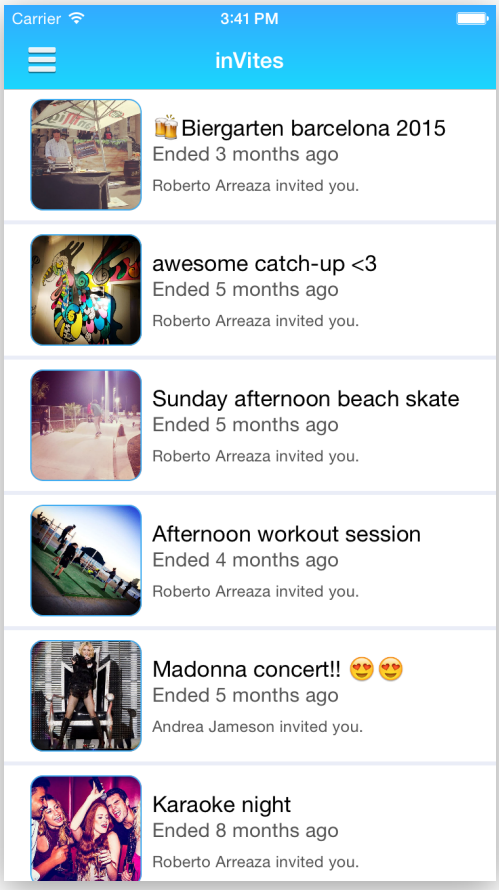

Figure 5.37: invites section view (iPhone 6).

It is fairly easy to observe the view consists of a list of all moments for which the logged user has been invited, ordered from most recent (at the top) to the oldest Moment at the bottom. Whenever a user gets invited to a new Moment, it will then show up at the top, so it's a pretty convenient place to see all the possible activities to available for a user to participate in.

When inspecting the rows for each Moment, it can be observed that it shows its title, when it starts or how long ago it ended, and an extra bit of useful information displaying who was the one that sent the inVite.

Additionally, and as it would be expected, when tapping on any of the Moments listed, the App will present the details of that Moment, and thus users may participate straight from here if they wish to do so. For more information on how to participate in Moments, readers should refer to "participating in Moments" from "News section".

Although this consists of a very simple part of the Application, its usefulness is key for users to be able to find out about all the different Moments to which they're being invited, so that they always keep aware of all the possibilities opened up for them by other people. This fact makes the inVites section a very relevant part of inJoin.

5.9 "Check-ins" section

As people make use of inJoin, and keep participating and contributing to a higher and higher number of Moments, such as Birthdays, concerts, get-togethers, parties, dinners, weddings and all kinds of social events, they might soon realize that it would be nice to have a place where they can just come and re-visit those memories later on, or simply just see and remember what they've been up to lately. This is the main motivation behind the existence of the *Check-ins section*.

Very much like the name suggests, and similarly to the inVites section, the Check-ins section consists of a list of Moments to which the user has Checked-in into. An example of the Check-ins section is presented next:

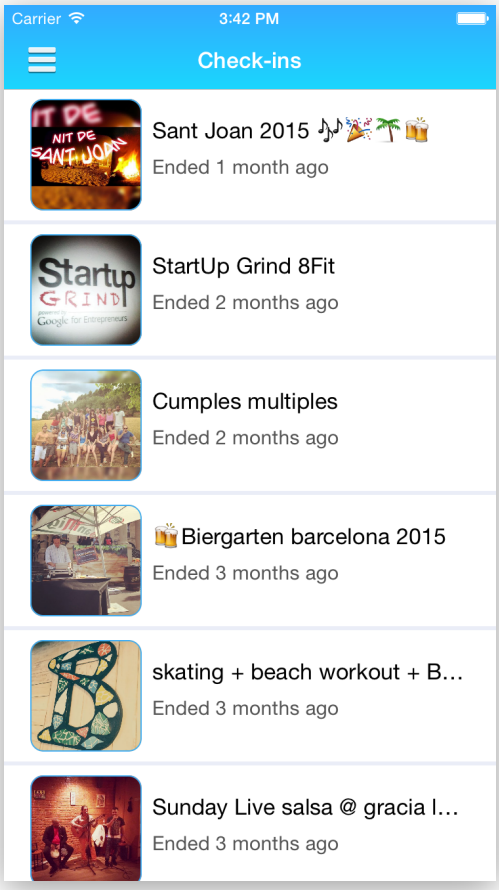

Figure 5.38: Check-ins section view (iPhone 6).

It looks very similar to the Moments presented in the inVites section, by displaying a chronologically ordered list of all Moments where the logged user has Checked-in. The cover picture of each Moment, along with its title and when it took place constitute the main information displayed to the user, who can tap on any of the Moments to inspect the details view of that Moment if he or she wishes to do so.

A very important thing to notice is that since Check-ins are the inJoin way of saying "the user is participating in the moment when and where it takes place" or, simply put, it's the way for users to say "I'm here" once they're taking part in a social event, then this means that the check-ins section displays a historical list of what the user has been doing, thus making it easy to revisit one's own steps and experiences, in a

chronological way, which ends up being a very nice personal experience, as it becomes an actual “trip down memory lane”.

6. System Architecture

In order to be able to build a platform capable of supporting the core functionality of the project, a set of components and technologies will be required. Each piece of the system plays a fundamental part in making the application modular, reliable and scalable as needed. The basic architecture design chosen for the implementation is then as follows:

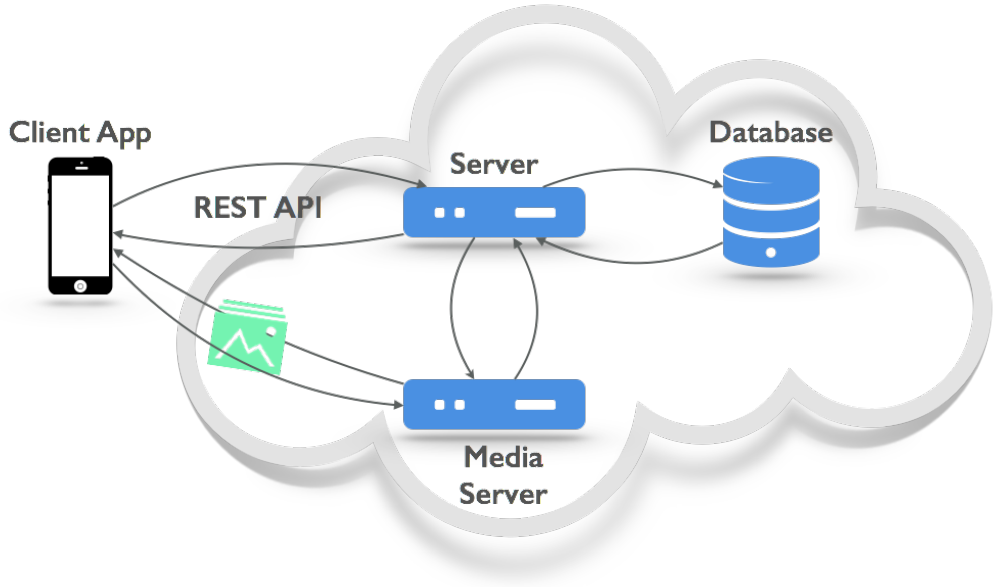

Figure 6.1: basic system architecture design.

The most obvious and fundamental part takes place on the mobile phone itself. This is the App that the final users have access to and from which they perform interactions by posting and getting any content that the App is meant to hold and make use of. It will be developed for the iOS platform and targeted for iPhones only (although it may still run in scale

on iPads). This App will then run the client code that will display everything the users needs and handle all communications (sending any relevant information, as well a retrieving all data needed for displaying the App to the final user) directly to a server over HTTP GET, POST, UPDATE and DELETE requests.

This then takes the following explanation towards another piece of the system architecture: the server. The server is in charge of holding the *backend* Application that is in charge of interacting with client App instances by receiving all HTTP requests, process them according to application-specific logic, and return an appropriate HTTP response as a result back to the client instances. This interaction is done by means of the Representational State Transfer (REST) philosophy, which takes advantage of the HTTP protocol and *stateless* interactions mostly focused on read/write/delete access to web *Resources* (Fielding, 2000) that allows for ease of scalability when compared to other web service solutions, which is a key factor in regards to the performance of the project's implementation.

The Database is necessary for storing any application-specific data (which is later mapped as the RESTful resources) and it's queried directly by the main server in order to process queries and generate appropriate responses accordingly.

Readers might find a critical piece of the puzzle still missing: storage. For the scope of this application project, storage mostly refers to the space required by the Media content (in this case, the pictures shared on the App) that's hosted by the App. Since the main server is in charge of processing the requests, and sometimes these requests may need some non-trivial processing, added to the design priority of scalability, then the role of storage is better implemented in a completely separate server instance. This way, exceeding loads to the backend main server are avoided by separating and encapsulating each functionality (processing and storage) independently. Only then may the client perform requests to the main server (application server) and receive resources that contain URLs to files (pictures in this case) located in a completely different server (media server).

A typical request for this architecture would start at the client (iOS mobile App) once it performs an appropriate RESTful HTTP request for a

backend resource. The request would reach the main application server, and start processing, which involves querying the Database in order to fetch any necessary data to process, as well as writing (is necessary) any picture files into the media server, so it can finally shape all processed data into the form of an appropriate RESTful HTTP server response and send it back to the client App.

Upon the reception of the response from the server, the client App must then parse the response obtained from the application server, and use the data to display any information necessary to the end user. This process involves also direct communications with the media server to fetch any picture files content it may require to display to the user.

6.1 Cloud hosting services

When developing this sort of distributed applications that rely heavily on an external web Server backend to properly function, it is common for developers to face a certain decision about the web application backend: to get it hosted *in-house* or to host it *in the cloud*.

In-house hosting refers to developers or companies getting all hardware and software components to set-up everything needed to run the web service and have it accessible over the Internet. This implies buying the required physical hardware servers, hard-disks, installing any support software, as well as taking on the tasks of server maintenance (Operating System updates, software updates, etc.), server fault handling, among other important tasks to keep everything up and running as desired.

One advantage of in-house hosting is the fact that the developer has full and absolute control over all components of the system, and customization then becomes fairly easy and straightforward. Another possible advantage would be that this requires a one-time investment and control over the hardware used, which depending on the scale and requirements may or may not be something positive.

On the other hand, nowadays there are very well established and known IaaS (Infrastructure as a Service) and PaaS (Platform as a Service) that offer solutions *in the cloud* for common hosting problems, without having to tackle many hardware and software problems that are highly common to an *in-house* hosting solution.

IaaS (Amies, Ning Liu, Sluiman, Guo Tong. 2012) solutions consist on access and availability of physical and/or virtualized hardware components, that developers may use in order to deploy and use to their own discretion. This then allows developers to buy some basic resources (usually paid by rates that depend on an on-demand amount of resources consumed) and not have to worry about the virtualization platform and hardware issues that would normally arise (Amies et al. 2012)

"… the cloud service provider is concerned about the virtualization platform; you do not need to worry about it."

Some common examples of the rates to pay for these services may be found for Amazon AWS (Amazon Web Services Inc., 2015), Microsoft Azure (Microsoft, 2015), or on Google Cloud Platform (Google Inc., 2015).

An IaaS then saves you normal Hardware-related issues, but does not resolve server-maintenance tasks, or other very common software related issues as well. These extra requirements can be approached by means of using another cloud-based alternative: PaaS (Platform as a Service).

PaaS solutions can be thought of as a higher-level stack built upon an IaaS components with a particular purpose in mind. PaaS offers a complete and customizable platform for delivering software on the web. This means it offers a complete solutions stack for supporting and running a web application, which normally requires the configuration and set-up of components such as:

- Server instances
- Load balancers for dynamic traffic handling
- Databases (Postgresql, MySQL, MongoDB, among others)
- DNS set-ups
- Storage components

That sounds a lot like IaaS, but the main difference is that PaaS already handles most of the work to interconnect and use all those components automatically, and the developer only has to worry about answering questions such as:

- "What server? What software stack should be preinstalled in it?"
- "What kind of database is needed?"
- "Under what URL should the Application be available?"
- "One or more servers? Should a Load balancer be used?"

And the PaaS handles all implementation details for those tasks automatically, whereas in a IaaS the developer would simply get all the virtualized hardware and do all the set-up work (Y Chang, W., Abu-Amara, H., & Feng Sanford, J., 2010).

Since the current project requires a great deal of development effort on the Client and Web Application side of things, the most convenient solution would then appear to be the use of a PaaS for hosting the Web Application in the cloud. In this manner, the developer may focus on the App-inherent development and tasks, and not so greatly on the underlying infrastructure that supports it and makes it scalable.

6.2 PaaS provider

There are several commercial PaaS that provide out-of-the-box support for a Ruby on Rails (the chosen technology for the backend) web application (from now on, *Rails App*), and among them two of the most important and renowned names are: Heroku and Amazon AWS.

Heroku may integrate any Rails App fairly easily via a set of CLI(command-line Interface) set of software tools called the "Heroku Toolbelt" (Heroku, 2015). It relies upon the Git version-control software (http://git-scm.com/) to upload and deploy any new version of a Rails App into the Heroku platform.

Heroku names the basic unit of functionality for applications as *Dynos*, and defines them as follows (Heroku, 2015a):

"Dynos are isolated, virtualized Unix containers, that provide the environment required to run an application."

These are abstract virtualized computational units where an Application may run, and an app may have multiple Dynos dedicated to serving requests.

While inspecting the Heroku platform, however, it was unclear on how to be able to customize or evaluate any Dyno at any given time, and that is one fundamental problem inherent to PaaS providers will often have.

On the Other hand, we have the widely used Amazon AWS (Amazon Web Services) platform, which started as a very complete and robust IaaS, and later matured into providing PaaS services as well.

Amazon *Elastic Beanstalk* is the name of the go-to PaaS for deploying web applications supported by AWS. It uses the IaaS resources already existing for AWS, but handles all the PaaS features for developers as well.

They key advantage revolves around the fact that, being also a IaaS provider, developers have also access to each individual resource allocated from the IaaS into the PaaS system, which gives a lot more flexibility, and gives a better sense of how things are working behind the scenes. In a way, this allows developers to avoid the hassle of dealing with common IaaS problems, while being able to inspect and have access to all the IaaS parts that construct the entire PaaS system.

Furthermore, AWS seems to integrate also very easily via its own CLI called the Elastic Beanstalk Command-line Interface (*EB CLI)* (Amazon Web Services Inc., 2015a) which similarly to Heroku, makes use of Git to deploy a new application version to the cloud infrastructure.

Finally, AWS is one of the most widely used and actively supported PaaS platforms in the market, with the backing of a very big and experiences cloud-based company behind it, which additionally allows a 1 year Free-Tier account for limited use of their services. This made amazon AWS a good choice as a cloud provider for the deployment and hosting of the Backend components for the Application.

6.3 AWS services used

Referencing figure 6.1, readers are able to see the system architecture proposed in this chapter. When this is applied to the Amazon AWS services, it ends up being mapped in the following way:

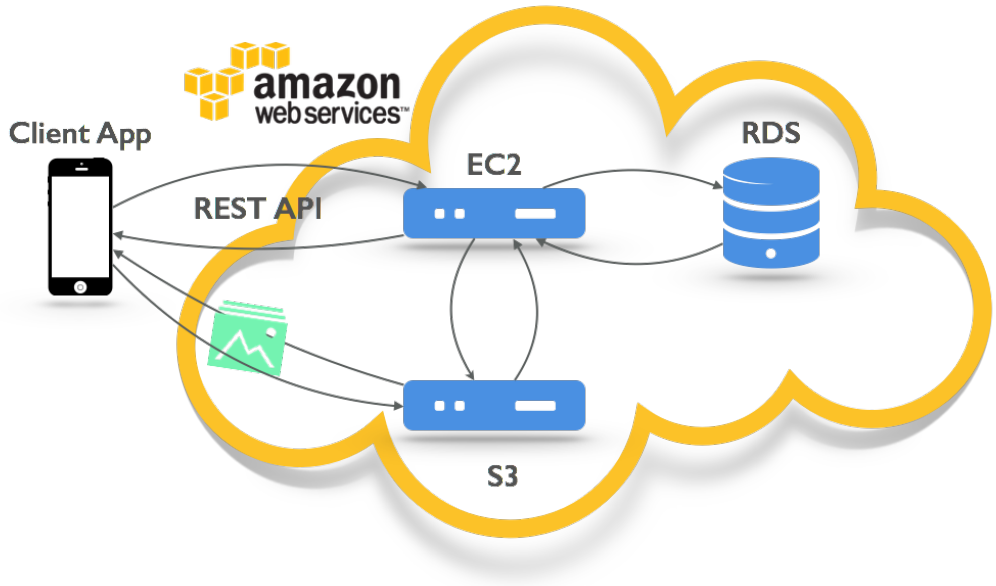

Figure 6.2: system architecture hosted in AWS (Amazon Web Services).

i) EC2

As observed in the figure above, the Server instance is mapped to an amazon EC2 instance, which is the virtualized hardware of a server machine, with a specific computational power and minimum storage available to it, and it is pre-loaded with the appropriate software bundle that the developer desires.

For the case of this project a Unix machine with pre-installed Ruby on Rails 1.9.3 capabilities will suffice for the tasks required, and this is available among the AWS choices provided.

The Rails App will then be deployed and run on one or more EC2 instances accordingly, and they'll be the ones serving any client requests.

ii) RDS

Amazon RDS is the cloud Database solution provided by AWS for many platforms, such as (MySQL, Postgresql, MongoDB, among others).

This may provide a Database instance that can automatically be accessed from the EC2 instances associated with the application, without depending on whether the EC2 instance(s) are down or not, and allowing (with the proper configuration) for external access to the Database as well, which is very useful when wanting to inspect the Database's data from a Postgresql client, for example.

For this project, an RDS Postgresql database instance was used in order to store all application data.

iii) S3

The Amazon S3 service's primary focus is on storage. They are servers dedicated solely to this purpose, and storage operations are abstracted via an API, which uses amazon authentication credentials to grant access to the right resources. This is why this fits perfectly to constitute the Media Server implementation.

S3 defines "buckets" as a virtual location within S3, with an associated URL that may hold directories and/or files of any kind. Bucket access may be configured and customized for public or private access, so resources are protected accordingly.

There are many publicly available Ruby gems (libraries) that perform the implementation of a client for accessing AWS S3 storage (Amazon Web Services Inc., 2015b), which can be leveraged from within the source code of the Rails app, to manipulate any files in any S3 accessible locations.

iv) Load balancer

When Scalability is an important design consideration in a distributed web application, as it is for this project, a key component (although optional for basic functionality) is a load balancer.

This component is in charge of distributing intelligently all traffic among -typically- many Server instances (EC2), that run the Application,

thus sharing the load of client requests taking advantage of the available computing power more efficiently.

Another important feature to note about load balancers in AWS, is that Developers are able to configure a dynamic number of running instances that adapts to current traffic demand. This means that an appropriate number of EC2 instances will be activated and configured to serve requests, in the same proportion in which traffic levels reach the currently available servers (i.e., if there's a peak of traffic, more servers will be configured and used to serve requests, and for lo traffic less servers will be used).

v) Elastic Beanstalk

As mentioned previously, Amazon Elastic Beanstalk is the name for the PaaS platform provided by Amazon, and it therefore integrates seamlessly with all previous components (EC2, RDS, S3, Load Balancers), in order to provide exactly the infrastructure required to deploy, run and scale the backend Application.

It can be configured to use any of those components (some are optional) and it automatically sets-up everything needed for the resources to work together and support the deployed application.

For the purpose of this project and the Beta version of the application, it is set to only use a single EC2 instance (without load balancing), a Postgresql RDS database with 10GB of capacity, and an amazon S3 bucket. Readers will immediately wonder why the lack of a load balancer: the answer is simply due to the fact that for the initial number of users this element is not critical, and the fact that load balancers don't fall under the AWS Free usage Tier. Nevertheless, for a production-scale application, a load balancer is definitely needed to support any unforeseen traffic peaks and user growth, without disrupting the Quality of Service experienced by Users.

When configuring an Elastic Beanstalk environment, developers must make a series of choices necessary for the environment's configuration:

- A name for the environment (many different environments holding different applications may be created for a single AWS account).
- Which kind of EC2 hardware will be needed?

- Part of the URL where the API will be reached (a prefix is added automatically by AWS, followed by the chosen name to conform the base URL).
- Should RDS be used for the project? If so, basic database configuration parameters, such as: database engine to use (Postgresql for this project), name of the database, root user name, root user password, storage capacity of the database (10GB for this project)

This comprises the most basic configuration decisions for an Elastic Beanstalk Application Environment set-up.

7. Developer's manual

In the same manner that the inJoin Application is presented from the final user's perspective, it becomes equally important to explain the how, meaning the functional description and some implementation details of the Application aimed at developers. The goal is then for readers to get a better idea about the development process and final implementation of inJoin.

7.1 Class structures

Once all the functionalities required for the Application are defined, it is then possible to design the Class structures and diagrams to implement in order to satisfy all the product guidelines, always following the fundamental iOS design paradigms and principles, as well as aiming to provide the best suited implementation to achieve the desired final user experience.

Throughout the project, there is a very large amount of classes implemented in order to solve specific problems and in charge of several tasks, and explaining all of them is out of the scope of this project, since it can get too repetitive, cumbersome, and sometimes unnecessary to understand the basic pillars of functionality. This is why readers will find next an explanation on the most important classes (with the exception of model classes, since readers may refer to a more extensive explanation on the Data Model in section "Data Model" of this document) that compose the basis of the Application.

IJWebAPIManager

At the core of this project, there is an inherent dependency that comes from the architecture implemented. Since the system is based on a REST API backend, then it should be easy to consider that this should have an impact on the class hierarchy design.

The class IJWebAPIManager is in charge of abstracting out all of the REST API complexity, into single, contained class that offers an Objective-C API mapped from the backend REST API. Its input/output is in the form of Model Objects mostly, and so it must then not only perform all web requests and communications to the backend, but also perform the data

translation (parsing) from Object Model instances into JSON representations and vice-versa. This is the reason why it's such a fundamental piece of the puzzle, and widely used all throughout inJoin. A small diagram shown next depicts the class and some of its relations to other classes:

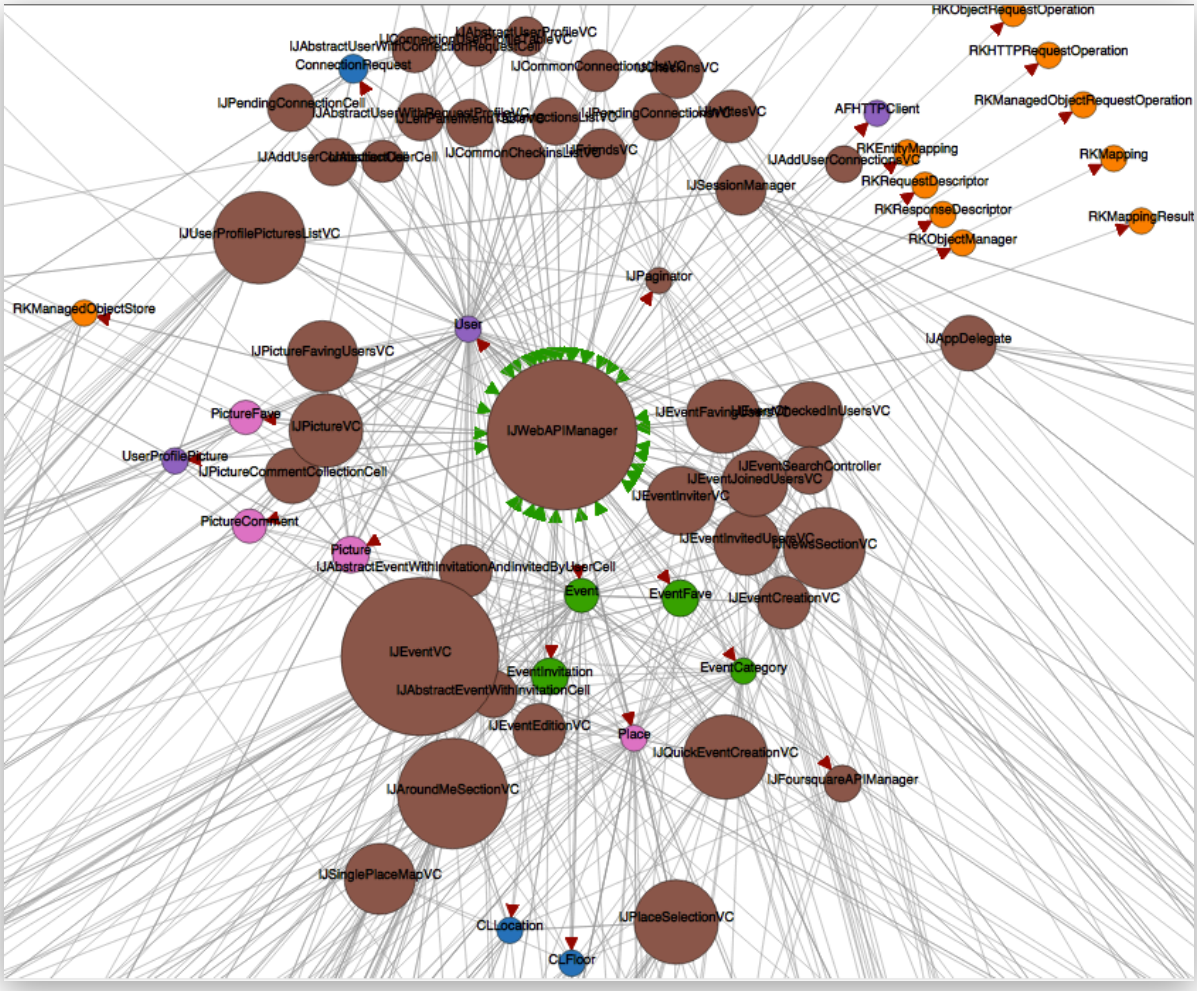

Figure 7.1: IJWebAPIManager class dependency diagram. Bubble size reflects how referenced the class is in other classes (the bigger the bubble, the more referenced it is). Green tip arrows represent incoming references from other classes, while red arrows represent other classes being used by this class.

Readers may observe from the figure above that IJWebAPIManager is quite a popular class, being used by many others such IJEventVC,

IJPictureVC, IJSessionManager, among many others. This is due to the fact that all these classes require interaction with the backend in one way or another to function properly, and thus make use of IJWebAPIManager in order to perform these interactions to send/receive data to/from the backend.

It is also possible to verify that IJWebAPIManager also makes use of classes such as AFHTTPClient or all of the Model classes, in order to perform the HTTP REST operations and encode/decode Model instances as input/output of these operations.

IJSessionManager

IJSessionManager consists on a single class used very frequently for backend session state consulting and session state handling operations. It is most evidently used in the IJLoginVC and IJRegistrationVC controllers, which as their names hint, are in charge of the user login and registration process, respectively.

Additionally, IJSessionManager is used whenever the App launches, to verify that the session is still valid and the app can continue functioning in a normal manner. In case a problem with the session is detected, the Login dialogue is presented to the user to activate a new session. Session validity is implemented by acquiring an authentication token from the backend once a successful login or registration process is finished.

The IJSessionManager class dependency diagram is presented in the following figure:

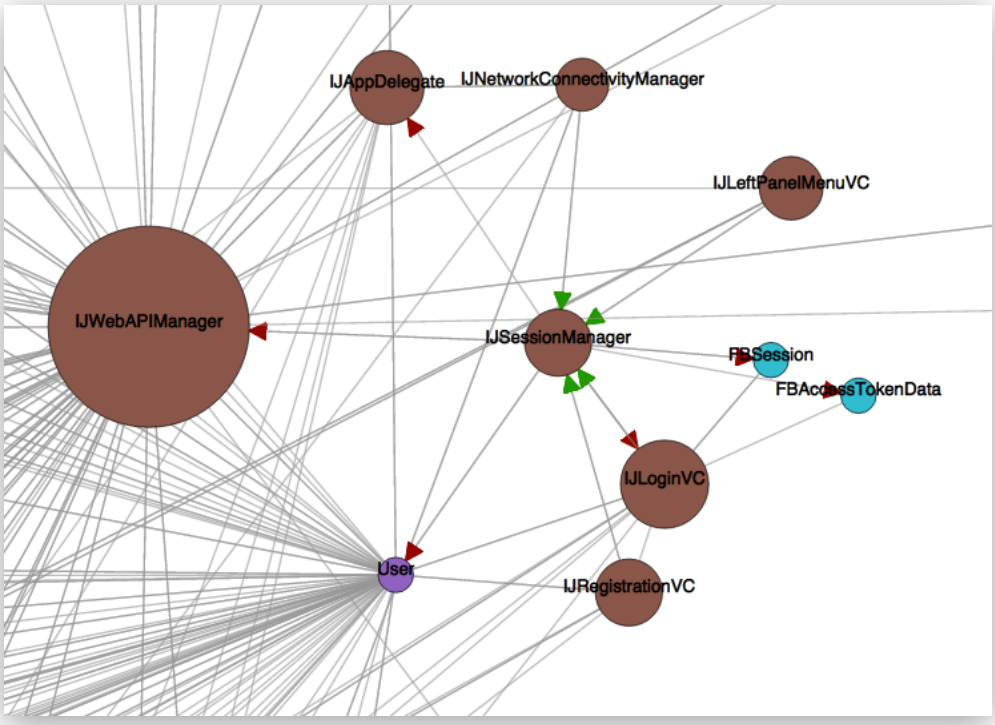

Figure 7.2: IJSessionManager class dependency diagram. Bubble size reflects how referenced the class is in other classes (the bigger the bubble, the more referenced it is). Green tip arrows represent incoming references from other classes, while red arrows represent other classes being used by this class.

Readers will realize the dependencies here are much less complicated than the ones in IJWebAPIManager, since session validation is a very specific and contained operation that is only required in certain classes.

One more important note on IJSessionManager is that it also handles the login operation via Facebook authentication, which is why in the diagram classes such as FBSession are present, since they are part of the Facebook API SDK that is used for this purpose. In regards to Data Model classes, Session operations only require information that is related to the User model, and thus this is the only Model class present in the diagram, as being used by the IJSessionManager class.

IJUIManager

This class is intended for more general purposes throughout the App, and as the prefix entails, this general purpose is mostly related to

handling User Interface (UI) common operations. For this reason, it will be used in various classes that require some UI functionalities, which in turn means it should only be found in View Controllers and View classes.

Some examples of the common operations available in IJUIManager are:

- Button or Image cropping.
- View round corners.
- Shadows and blur effects.
- Making Views rounded (for example, user profile pictures)
- Borders and color functionalities.
- Showing standard notifications.
- Main App color palette.
- Placing Spinner indicators for busy operations.

The class dependency diagram for IJUIManager is as follows:

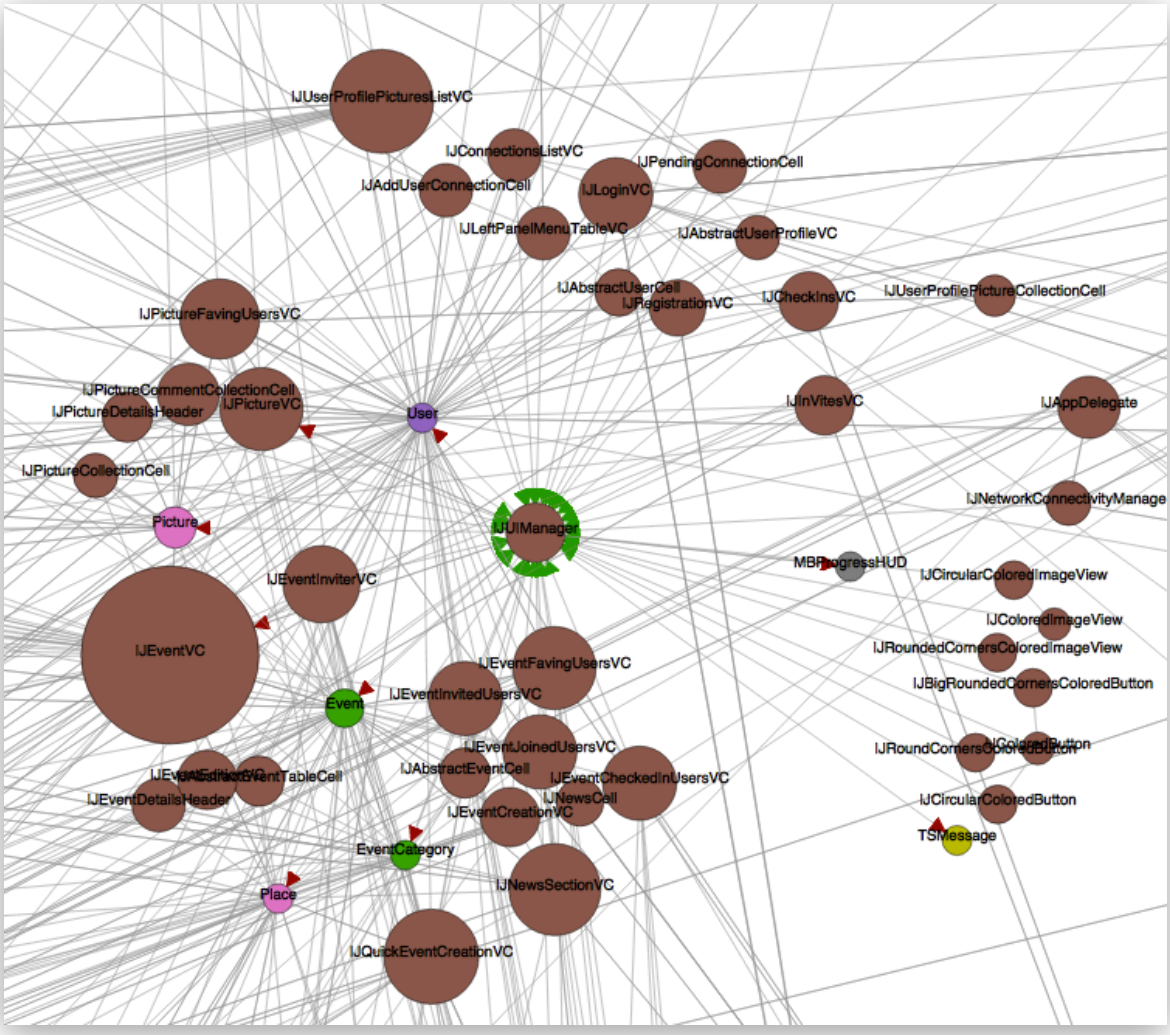

Figure 7.3: IJUIManager class dependency diagram. Bubble size reflects how referenced the class is in other classes (the bigger the bubble, the more referenced it is). Green tip arrows represent incoming references from other classes, while red arrows represent other classes being used by this class.

As it may be appreciated, it is widely used throughout most ViewControllers and several View classes, such as buttons, table cells and several other elements.

IJNavigationManager

Most of the time, user actions while interacting with an Application are the main drivers for App navigation. Users tap on buttons to trigger the presentation of new Views, they tap as well to go back, they perform a gesture to bring up a menu and navigate through it, and many other common operations that are pretty standard within the iOS world.

There are also times however, when an App's navigation requisites demand for an automatic navigation approach for different scenarios.

Automatic navigation implies that the App should navigate to a point in the navigation hierarchy, without the need for the user to perform all the sequence of actions that it normally takes to get to that particular point in the Application. This is particularly true and mostly common for the case where navigation is triggered by opening a push notification, that requires the App to open up directly at a certain point to display the notification's information appropriately.

This is the main purpose that IJNavigationManager serves in inJoin. The class dependency diagram for it as follows:

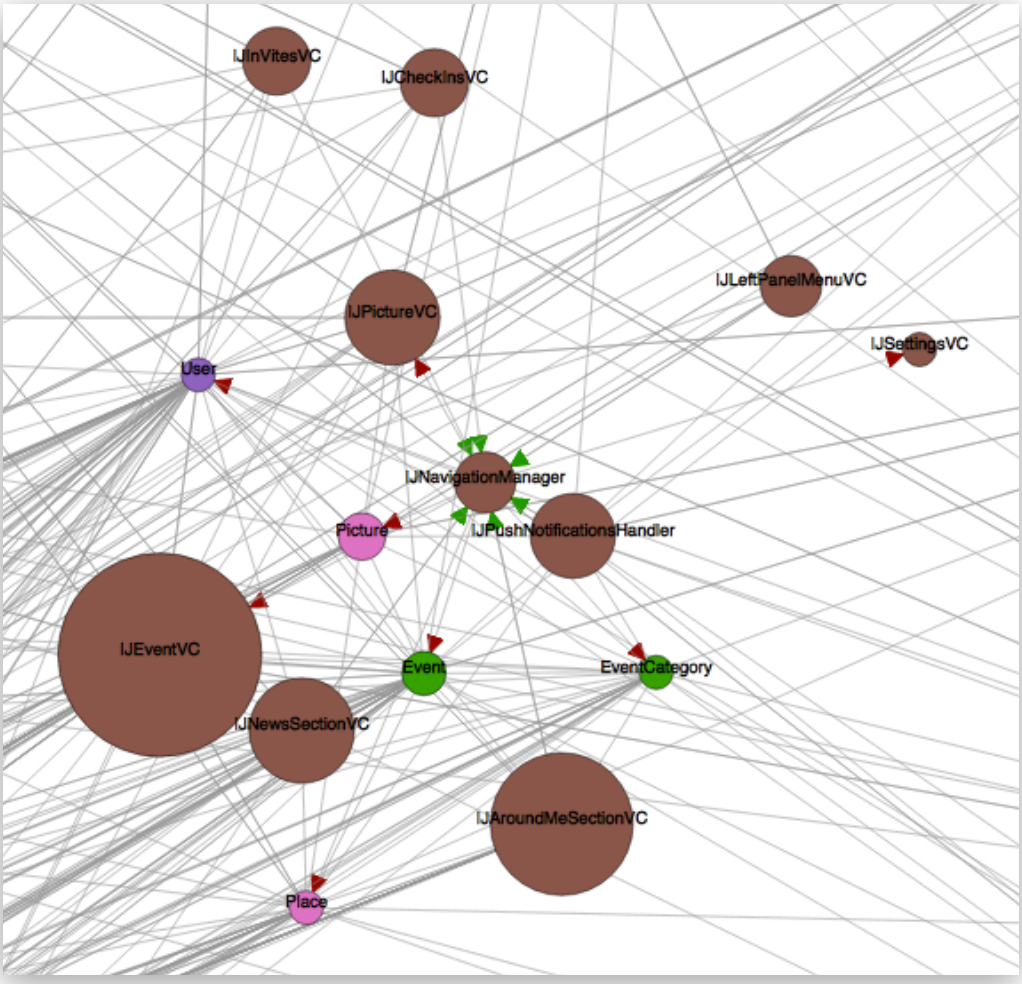

Figure 7.4: IJNavigationManager class dependency diagram. Bubble size reflects how referenced the class is in other classes (the bigger the bubble, the more referenced it is). Green tip arrows represent incoming references from

other classes, while red arrows represent other classes being used by this class.

As it can be seen, it relates Mostly to ViewControllers that will be presented by means of an automatic navigation sequence, as well as classes such as IJPushNotificationHandler, in charge of handling and routing push notification reception throughout the App. This approach allows for Push notifications to be received by the device, and if users select them, the App automatically takes them to the appropriate piece of content indicated by the notification.

IJSidePanelsVC

As the application was designed, there appeared to be some very well defined sections that constituted the most fundamental parts of the functionalities and, as such, they would have to be accessed in a very simple, intuitive and quick way in order to make user experience as good as possible. For this very reason, the design of the App was made with a main lateral menu, composed of a sliding panel that appears/disappears from the left side of the device screen, which could be accessed via a main menu button or via a convenient lateral swipe gesture. This kind of menu has become more and more common due to its usage in several popular iOS Applications worldwide, and this made it a suitable choice for the main menu implementation.

The IJSidePanelsVC class creates and appropriately configures precisely such functionality, implementing the sliding left panel main menu that is found on the App. It is configured with a table with cells corresponding to each one of the main menu sections of the Application, and its class dependency diagram is depicted next:

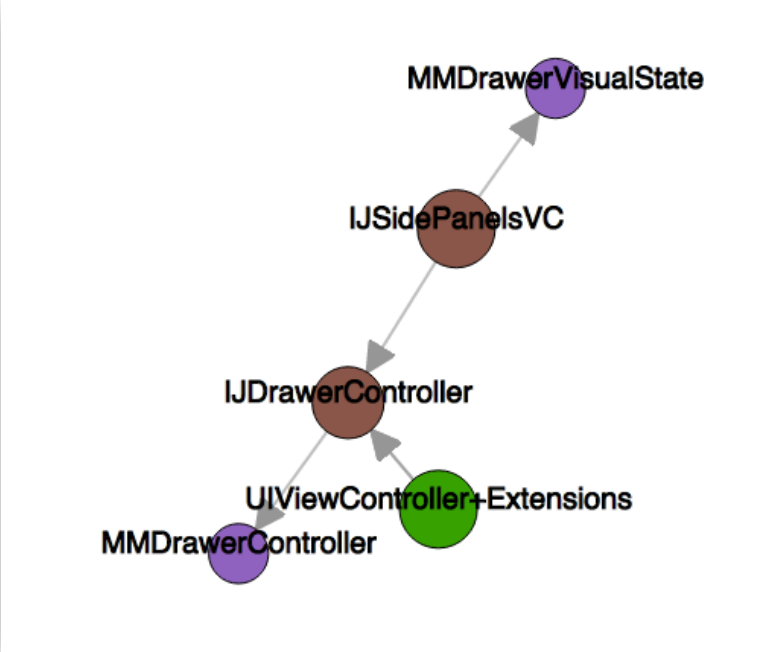

Figure 7.5: IJSidePanelsVC class dependency diagram. Bubble size reflects how referenced the class is in other classes (the bigger the bubble, the more referenced it is). Green tip arrows represent incoming references from other classes, while red arrows represent other classes being used by this class.

Notice that IJSidePanelsVC uses a DrawerController, which in turn uses a class named MMDrawerController (https://github.com/mutualmobile/MMDrawerController), a class publicly-available on GitHub that performs most of the heavy lifting in regards to the sliding panel implementation.

IJMeSectionVC

As explained in the "Me section" of the "User Manual" chapter of this document, the Me section of the App consists on the profile information of the inJoin user currently logged in. It's basically composed of another ViewController named IJCurrentUserProfileTableVC, which is in charge of showing the user's profile. The class dependency diagram for IJCurrentUserProfileTableVC is as follows:

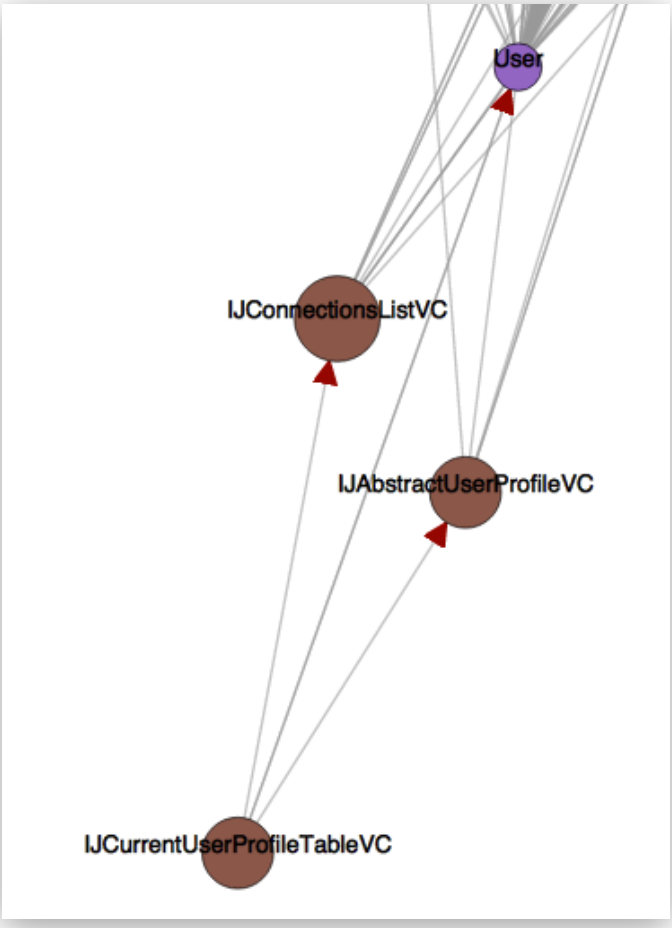

Figure 7.6: IJCurrentUserProfileTableVC class dependency diagram. Bubble size reflects how referenced the class is in other classes (the bigger the bubble, the more referenced it is). Green tip arrows represent incoming references from other classes, while red arrows represent other classes being used by this class.

IJCurrentUserProfile is a subclass of IJAbstractUserProfile, and it adds some extra information such as the connections list to the base class. The class dependency diagram for IJAbstractUserProfileVC is as follows:

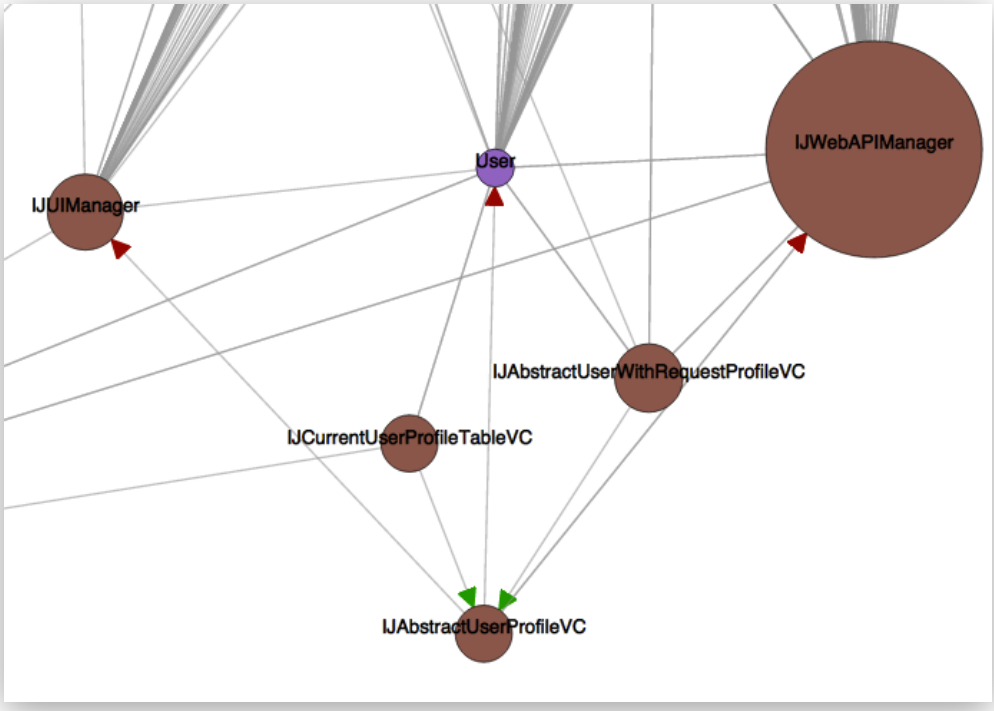

Figure 7.7: IJAbstractUserProfileVC class dependency diagram. Bubble size reflects how referenced the class is in other classes (the bigger the bubble, the more referenced it is). Green tip arrows represent incoming references from other classes, while red arrows represent other classes being used by this class.

As it is depicted by the above figure, the IJAsbtractUserProfileVC is the main class for defining user profiles, and uses IJWebAPIManager to obtain any required information from the backend such as the user's profile picture and basic information to be displayed inside the profile, as well as making use of UIManager for styling up some of the UI displayed to the user.

IJFriendsSectionVC

In regards to the IJFriendsSectionVC, this is what is referred to the Connections section of the application, and basically contains the information about the connections for the currently logged in inJoin user, as well as the mechanisms for finding other inJoin users, sending connection requests, as well as confirming/denying any connection requests received by the current logged user. In simple terms, it's a centralized place to manage all connection-related tasks. The class dependency diagram for this class is as follows:

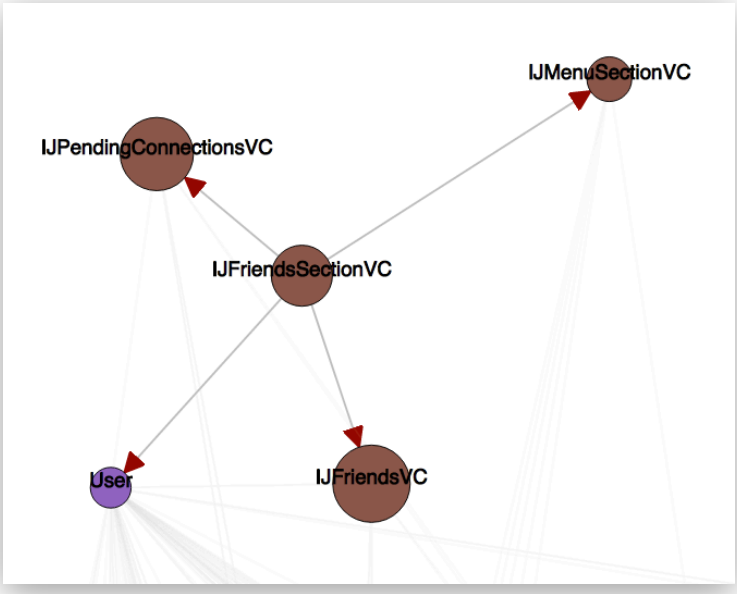

Figure 7.8: IJFriendsSectionVC class dependency diagram. Bubble size reflects how referenced the class is in other classes (the bigger the bubble, the more referenced it is). Green tip arrows represent incoming references from other classes, while red arrows represent other classes being used by this class.

As it's possible to observe, IJFriendsSectionVC uses IJPendingConnectionsVC as the controller for confirming/denying any pending connection requests. The IJFriendsVC class is the main component of the IJFriendsSectionVC, and its corresponding diagram is presented next:

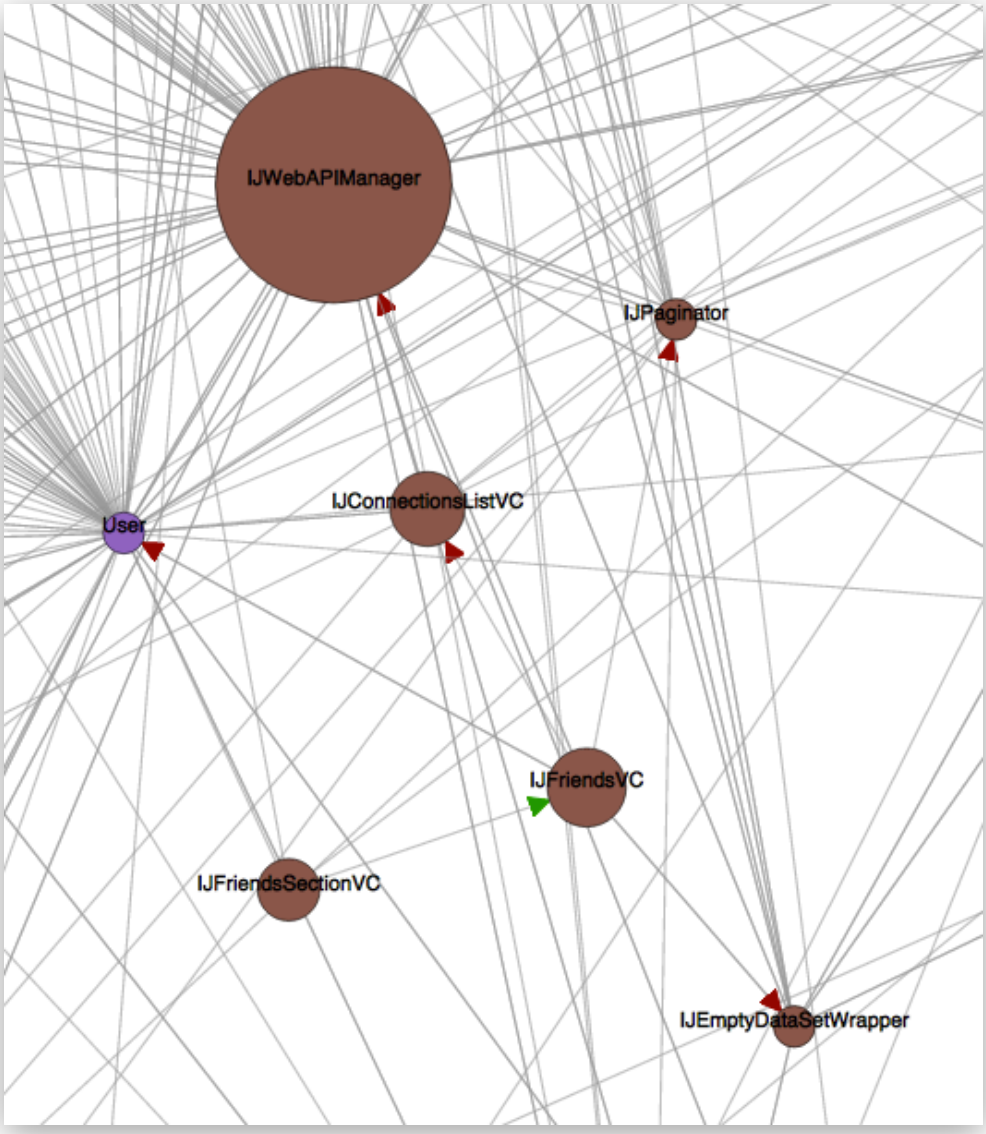

Figure 7.9: IJFriendsVC class dependency diagram. Bubble size reflects how referenced the class is in other classes (the bigger the bubble, the more referenced it is). Green tip arrows represent incoming references from other classes, while red arrows represent other classes being used by this class.

By observing the figure it's possible to derive that IJConnectionsListVC represents the controller in charge of handling the presentation of a list of al the existing user connections, as well as the fact that IJFriendsVC employs IJWebAPIManager for network requests, as all other main section will also do.

IJNewsSectionVC

This corresponds to the "home" view of the Application, since it holds the updated list of news about Moments of interest to the user. It also holds protagonism due to the fact that within this section is where new Events (Moments) can be created, thus constituting a fundamental part of inJoin.

The class dependency diagram is shown next:

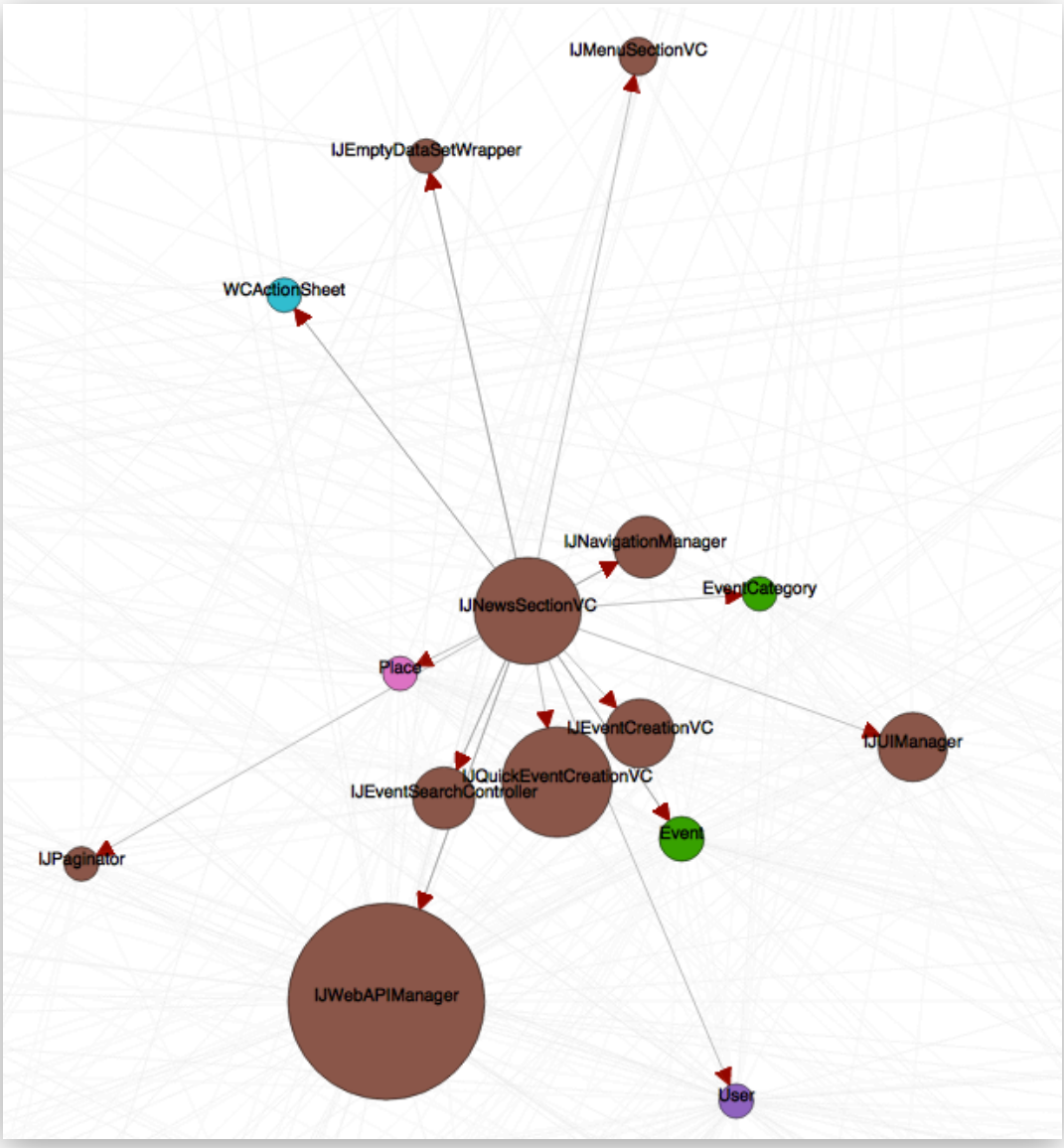

Figure 7.10: IJNewsSectionVC class dependency diagram. Bubble size reflects how referenced the class is in other classes (the bigger the bubble, the more referenced it is). Green tip arrows represent incoming references from

other classes, while red arrows represent other classes being used by this class.

Readers may infer from the diagram some important things about the IJNewsSectionVC. It naturally makes use of IJWebAPIManager for fetching all the news information from the backend, as well as using IJUIManager to format most of its list UI. More interestingly, however, is observing classes such as IJEventCreationVC and IJQuickEventCreationVC, both in charge of the creation process of new Moments (events). Furthermore, it also employs IJEventSearchController as the means for performing global text search operations for Moments throughout the entire News content, thus enabling users to find a specific Moment by performing name searches. From the news it's also possible to access the details of specific Moments (events), which are represented by IJEventVC, which is explained below.

IJEventVC

An Event is the codebase abstraction for an inJoin Moment, and IJEventVC is in charge of controlling an Event's details view. From this View Controller, users get access to the pictures posted, as well as to the basic information of the Event. Furthermore, it also centralizes several tasks, such as inviting people to an Event, participating in it, Joining/Checking-in, posting photos, among several other actions. Due to this, it can be said that IJEventVC is the most fundamental controller in the entire Application.

The class dependency diagram for IJEventVC is depicted below:

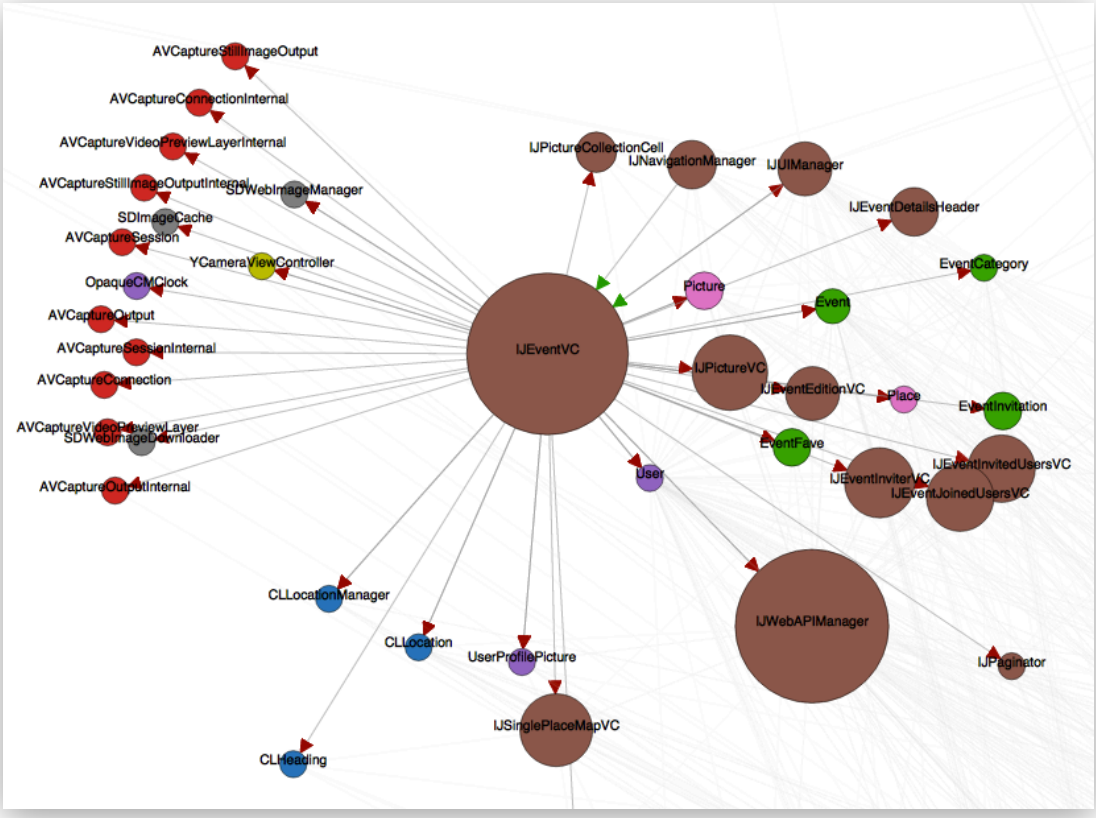

Figure 7.11: IJEventVC class dependency diagram. Bubble size reflects how referenced the class is in other classes (the bigger the bubble, the more referenced it is). Green tip arrows represent incoming references from other classes, while red arrows represent other classes being used by this class.

As it may be expected, IJEventVC makes use of IJWebAPIManager to fetch all event-related data for presenting this information to the user, as well as executing actions such as Check-ins, Joins or Faves. It also employs classes like IJPictureVC for presenting the details of specific photos that have been posted into the Event, or IJEventEditionVC, which as the name entails, is presented to edit the Event's content. Additionally it takes advantage of IJEventInviterVC for allowing creators to invite connections to the Event, and several other classes intended for a variety of additional purposes, among which readers may find IJNavigationManager, because it is sometimes necessary to present a specific Event's details or a specific Photo posted into a particular Event, directly after receiving a push notification, for instance.

IJPictureVC

The fundamental component of an Event (Moment) is –naturally- the content posted into it. Currently for inJoin, this content is represented by the photos posted into that Moment, and the presentation of this important structure is managed by IJPictureVC.

The class dependency diagram for IJPictureVC is the following:

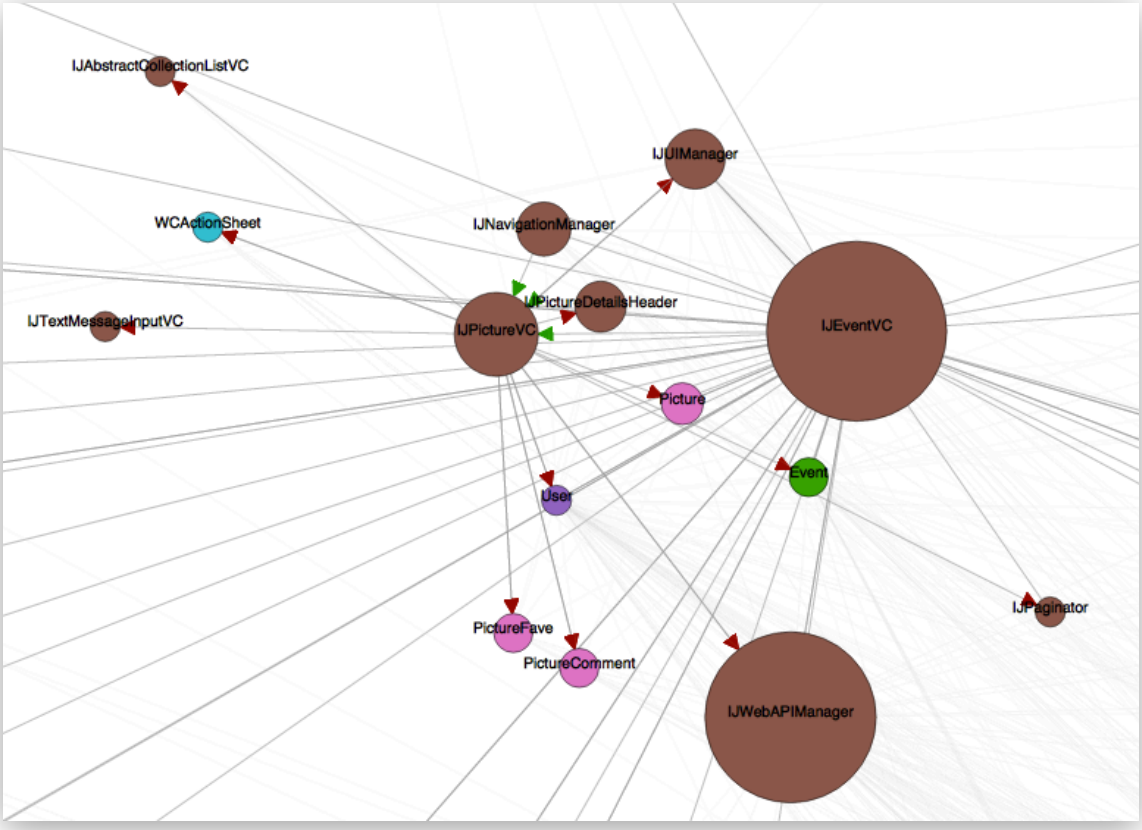

Figure 7.12: IJPictureVC class dependency diagram. Bubble size reflects how referenced the class is in other classes (the bigger the bubble, the more referenced it is). Green tip arrows represent incoming references from other classes, while red arrows represent other classes being used by this class.

This view Controller class is in charge of presenting a View with the photo image, as well as the mechanisms for interacting with the photo it represents. These "interacting" mechanisms basically consist on commenting photos and faving photos, but it also includes operations such as saving the image into the device's "camera roll", or deleting the

picture from its containing Moment, as well as setting the image as the main user profile picture or cover picture for the Moment.

By using IJWebAPIManager it obtains all the information about the image data and the comments content, as well as performing fave and new comment operations. The class IJTextMessageInputVC is the one delegated to present the text input box for commenting a picture directly from IJPictureVC, thus making picture comment operations fairly straightforward.

IJCheckInsVC

Check-ins in the inJoin Application seek to ensure that participants of Moments (Events) are actually –i.e. in "real life"- taking part in the Moment they choose to contribute to, by using the App. This is validated by using the device's location, and make Check-ins a big part of the idea of taking real shared experiences into the digital world. As a sort of "side-effect" from implementing Check-ins, having this kind of information then allows for creating a form of historical record, where essentially any user is able to see what he or her himself or herself has been up to, in a chronological order. This is achieved by the set of classes presented next:

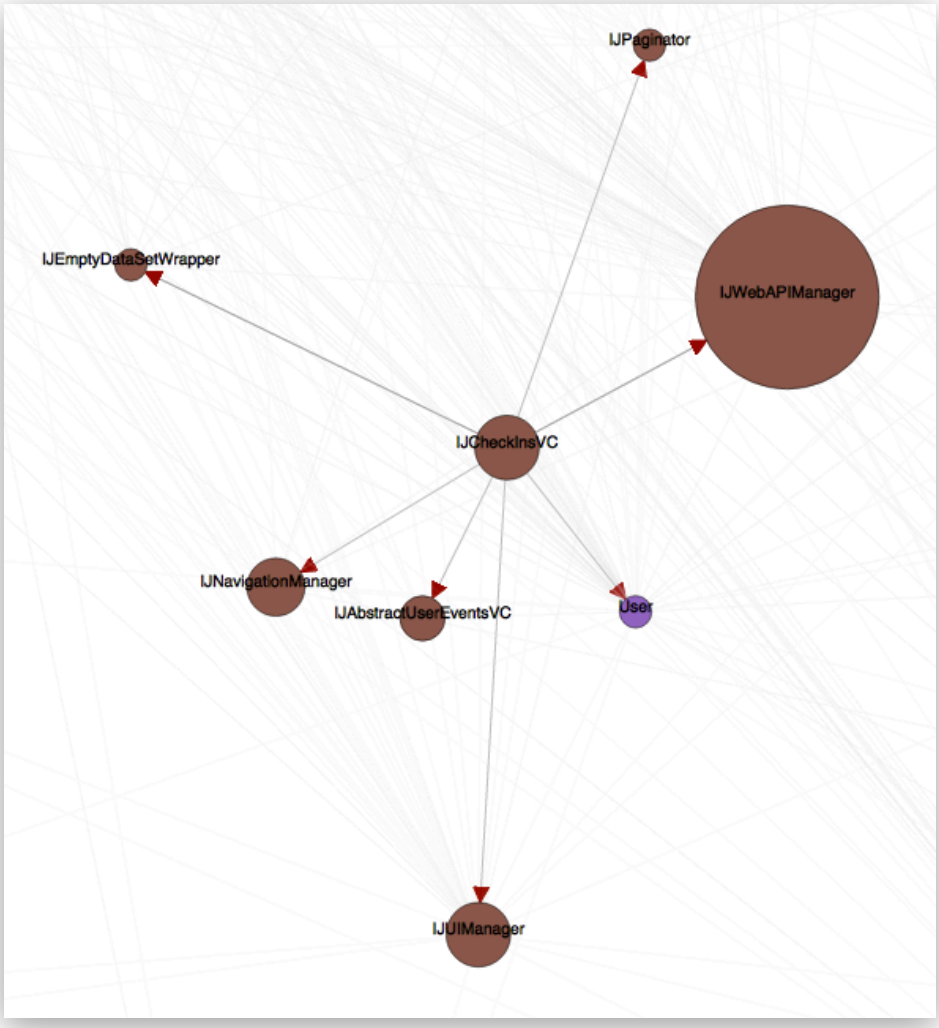

Figure 7.13: IJCheckInsVC class dependency diagram. Bubble size reflects how referenced the class is in other classes (the bigger the bubble, the more referenced it is). Green tip arrows represent incoming references from other classes, while red arrows represent other classes being used by this class.

IJAbstractUserEventsVC is employed by IJCheckInsVC in order to list all the events where the uses has Checked-in in the past. For the purpose of retrieving the Check-ins information, IJWebAPIManager is used for accessing the API, and IJPaginator to paginate the list so that the user can perform an almost seamless "infinite scrolling down" throughout his or her list of Checked-in Moments. IJUIManager is used to format the cells representing each particular Moment, providing borders and rounded corners to the cover picture, and other look and feel tasks required by design.

IJInVitesVC

Similarly to IJCheckInsVC, IJInVitesVC presents the user with a list of all the event invitations sent by others via inJoin, so that the user does not miss out or overlooks a particular Moment potentially interesting for him or her.

In the above point lies the importance of IJInVitesVC, which takes central part in the following class dependency diagram:

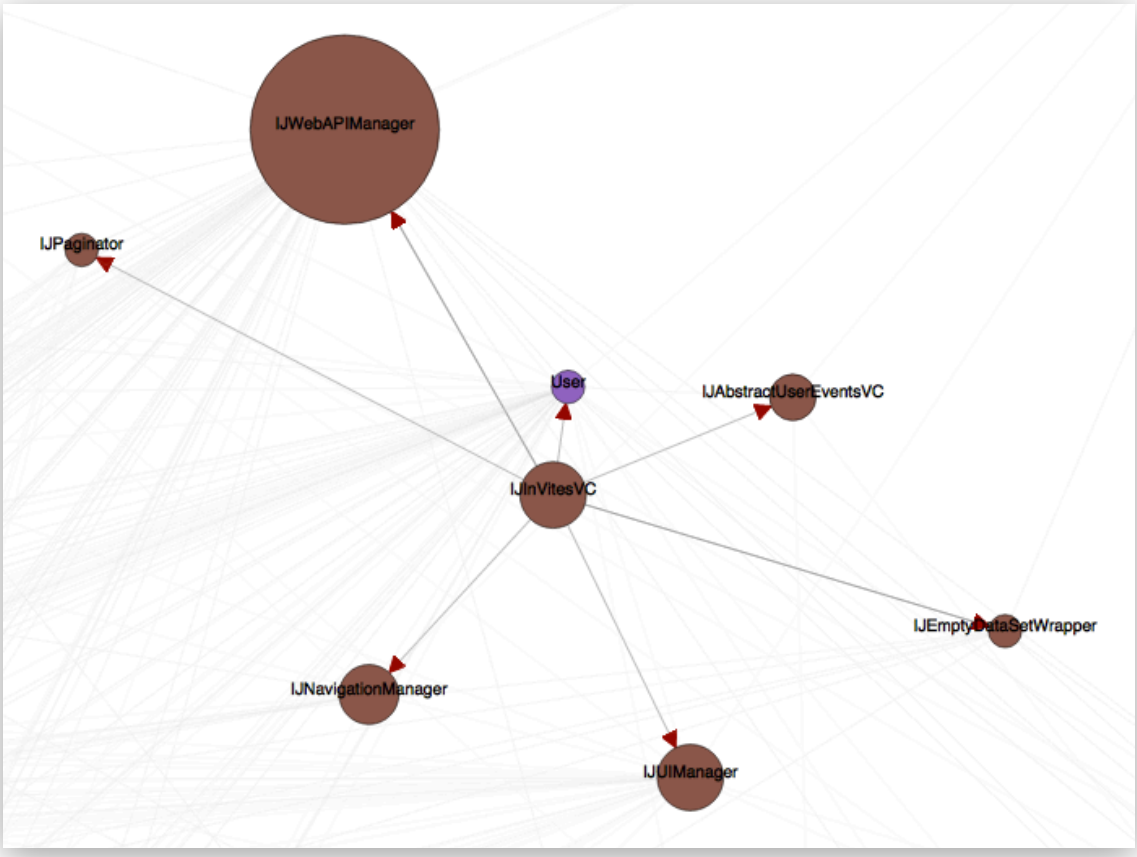

Figure 7.14: IJInVitesVC class dependency diagram. Bubble size reflects how referenced the class is in other classes (the bigger the bubble, the more referenced it is). Green tip arrows represent incoming references from other classes, while red arrows represent other classes being used by this class.

Readers will find the class diagram quite similar to the one for IJCheckInsVC, due to the fact that it's essentially the same kind of information, namely "Moments" being represented. The difference lays on

the fact that in this case the Moments being listed are the ones for which the logged user has been invited, as opposed to IJCheckInsVC where the common feature was the Moments where Check-ins have been performed by the logged user. The rest of the inner workings for both classes apart form this point are basically the same.

7.2 Data Model

For any kind of Application in Modern software, a crucial step, which lays out the basis for the entire functionality, is the definition of the Data Model to use. This is why the Data Model structure has to be coherent, complete and adapted to the kinds of functionalities that will be required by the Final application itself. It is not uncommon for developers to realize minor improvements have to be made over time when implementing a specific feature or functionality, but the basic structure of the Data Model should be so that it should not change, unless some radical changes in the App concept are implemented.

The Data Model for inJoin is the defined keeping in mind all of these factors, and the resulting data structure can be represented as follows:

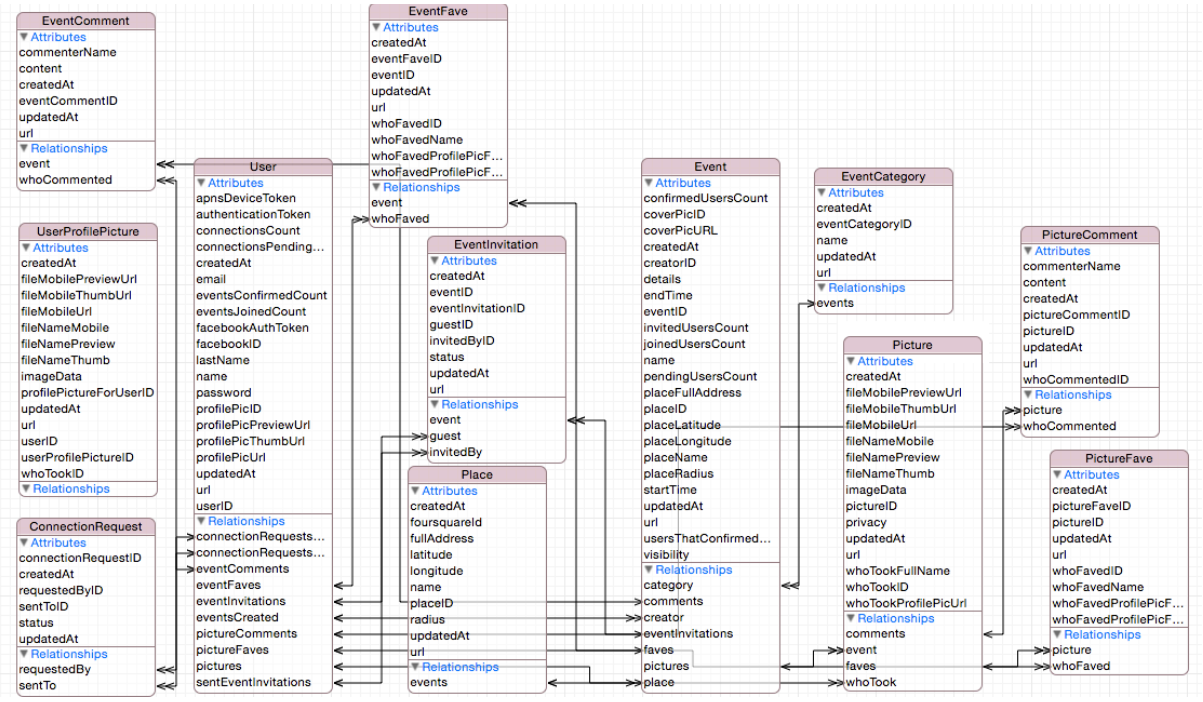

Figure 7.15: inJoin Object Model map. "to-many" relationships are shown with a double-arrow connector. Source: XCode 6.

At first sight it can be a bit overwhelming from looking at so many data structures with many relationships, but when breaking it down to each entity, and keeping in mind the concept to achieve, it becomes easy to grasp.

It is evident that a ***User*** entity is a primordial need for the Application, so it was then defined with this same name. It holds basic user information, such as a userID, name, lastname, email, password, or profilePicUrl (URL for the profile picture of a user) as well as information that is not so intuitive or very straightforward, such as the Facebook Authentication token (facebookAuthToken), the local authentication token (authenticationToken), the Push services Token (apnsDeviceToken), and other relevant information that is useful for various sections of the Application.

In order for users to be able to connect to each other, inJoin defines the ***ConnectionRequest*** entity as the abstraction of a petition for connecting two users. This entity contains all the information related to a user's petition to another to become *Connections*. Therefore, it can be observed to be a very simple data structure, composed mainly of an identifier field (connectionRequestID), the identifier of the user who sent the request (requestedByID) as well as the identifier for the user who receives the request (sentToID), and the status of the request, to indicate if it's pending confirmation, confirmed or denied by the user that receives it.

Users also may posses one or many profile pictures. This is abstracted in the ***UserProfilePicture*** model entity. It contains information about to which user it belongs (userID), whether or not it is the currently active profile picture for the user (profilePictureForUserID), as well as the URL from which the image data is available (fileMobileUrl), as well as the thumbnail (fileMobileThumbUrl) and preview (fileMobilePreviewUrl) versions of the image.

As it can be appreciated, Users are present all throughout the App, since they're the main actors of any petitions and actions, and this makes them one of the main entities in inJoin. However, there are many other model entities that deserve an explanation as well. The most vital –and evident- entity for the inJoin Data Model, is the abstraction for Moments, which is contained within the ***Event*** entity. Here all data regarding moments is contained and all associations are held here as well.

An Event's instance basic information consists on the name of the event (name), description for it (details), the starting date and time (startTime), the user who created it (creatorID), as well as the ending date and time (endTime) defining when it takes place, as well as information regarding the Event's privacy value (visibility) –determining if it's a public or private Moment-, its cover picture (coverPicUrl), the location set as the venue for the Moment (placeID), as well as other complimentary convenience data, which might be useful for specific parts of the App.

The way in which users get invited to Moments, requires the presence of another important model structure called ***EventInvitation***. EventInvitations then have to contain the information about which user is inviting another (invitedByID), which user is being invited (guestID), as well as which is the event for which the invitation has effect (eventID), as well as a status (status) field indicating if the invitation has been accepted, ignored or if the user is in checked-in state, once the guest performs each operation via the Application.

As the Event entity was explained, the concept of the venue, representing the location where the Moment takes place, was brought up to reader's attention. This also consists on a model entity named ***Place***. As it may be inferred, a Place requires some basic geo-information, such as the latitude (latitude), longitude (longitude), and a radius (radius) indicating not only the exact gps coordinates of the location, but also a sense of the physical dimension of its area size. This radius is particularly useful and key in the Check-in process, since the user must be in the area determined by this field in order to perform the Check-in successfully. Additionally, some extra information, such as a provided name for the place (name), and a foursquare identifier (foursquareID) are also present in the Place entity. The foursquareID field is used because inJoin integrates with the Foursquare Database of places, as a convenience for the user to show already existing and well-known places from that service. If one of the Foursquare places is used when creating a inJoin Place, then it will have the foursquareID indicating its source.

On the other hand, Events can also be "faved" by users. This Faving operation creates an entity called ***EventFave***, which abstracts the faving process by holding a reference to the user who performs the faving (whoFaved), the Event that is being faved (eventID) as well as other convenience information that may be used in specific parts of the Application.

A final note on Event entities, they also are designed with the feature of defining a category for them, thus allowing for future improvements based on event categorization. This is why the entity ***EventCategory*** exists, which basically hold a name for the category (name), and a reference to the events that belong to this category. This feature, however, is left for future implementation, as it is unused throughout the App, event though it exists on the data Model.

As a final piece of the Data Model puzzle, the abstraction for the photos that users share on Moments is contained inside the ***Picture*** entity. Similarly to UserProfilePicture, the Picture entity holds references to the source of the image data in the various formats (fileMobileUrl, fileMobilePreviewUrl, fileMobileThumbUrl), but it also holds information of who is the author (whoTookID) of the picture –i.e. who posted it on a Moment- as well as a privacy field, which is currently unused, and its purpose is for generating a fine-grain level of privacy into specific pictures for moments.

Very much like an Event, a picture may also be "faved" by users, and thus the entity ***PictureFave*** exists, containing a reference back to who has faved the picture (whoFavedID), what picture was faved (pictureID) and some additional convenience information.

Finally, pictures may not only be faved, but commented as well, and so there is a ***PictureComment*** entity which contains the identity of who commented it (whoCommented), the picture that is commented (pictureID), as well as the comment's content itself, holding the text to be displayed as the comment on the Application.

7.3 REST API Services

The last relevant piece of the Development mechanism consists on the backend REST API services offered. This API is designed in perfect harmony with the iOS Client, taking into account what kind of information is required by the App in order to properly display the information it need to present to the user, as well as allowing users to perform operations that should persist on the backend, such as posting a new photo into a Moment (Event).

As the data model design should fundamentally be the same in both the Client Application and the backend, many of the API endpoint names will result familiar to readers who have gone through previous sections of this document. One important thing to note is the common notion that most model classes are represented by API endpoints such as base-url.com/**events** or base-url.com/**users**, and endpoints with a format such as base-url.com/events/**:id**, represent a single instance of an event (Moment), providing an id by means of "**:id**" to reference it accurately. There are also cases where endpoint resources are nested inside other "parent" resources. Such is the case, for example, of a Comment belonging to a Picture. Since every comment has to belong to a given Picture, this means the nesting association is a good fit for the API. An endpoint for this example would be something like:

base-url.com/pictures/**:picture_id**/comments/**:comment_id**

Another thing to keep in mind -for the sake of simplification- is also the fact that any API resource that represents a Data Model class will have a basic CRUD (Create Read Update Delete) interface, which means, for instance, that a Picture will have the following basic API available:

HTTP method	URL	Function
POST	base-url.com/pictures/	creating a new Picture (**C**reate)
GET	base-url.com/pictures/**:id**	fetching a specific Picture by its **:id** value (**R**ead)
PUT	base-url.com/pictures/**:id**	editing an existing Picture (**U**pdate)
DELETE	base-url.com/pictures/**:id**	deleting an existing Picture (**D**elete)

Figure 7.16: common CRUD API example for the Picture Model class.

In the same way, all Data Model classes will have available a CRUD API for themselves, and thus in the following explanations the CRUD part of each API will be omitted.

In order to provide a better-organized API, the explanation will be grouped into different categories, following a coherent and structured order.

Users API

As expected, the users API has its basic CRUD API, and additionally some interesting extra services:

GET base-url.com/**users/:id/news**

This endpoint is used to retrieve the news for a specific user, by providing its **:id**. This is due to the fact that news are very personalized requests, and it returns all the relevant Moments (events) that the algorithm determines to send back.

GET base-url.com/**users/:id/connections**

This petition returns all the connections (in the form of user instances) for a specific user with a provided :**id**.

GET base-url.com/**users/:id/connections_pending**

This petition returns all the connections (in the form of user instances) that are in state pending (meaning that the connection request linking the users has not yet been confirmed) for a specific user with a provided :**id**.

GET base-url.com/**users/:id/events_joined**

This petition returns all Events that have been Checked-in (unlike in the client, in the backend the word "join" implies Check-in operations) by a specific user with a provided :**id**.

GET base-url.com/**users/:id/events_confirmed**

This petition returns all Events that have been Joined (confirmation in the backend maps to "joining" operations in the Client) by a specific user with a provided :**id**.

GET base-url.com/**users/:id/events_pending**

This petition returns all Events that are still pending (meaning the user has been invited but not yet RSVPed by Joining) by a specific user with a provided :**id**.

GET base-url.com/**users/:id/common_connections**

This petition returns all Connections (in the form of User instances) that are connections to both the user making the request and the user determined by the provided :**id** parameter.

GET base-url.com/**users/:id/common_events_confirmed**

This petition returns all Events that have been confirmed ("Joined", in Client terminology) by both the user making the request and the user determined with the provided :**id** parameter.

GET base-url.com/**users/:id/common_events_joined**

This petition returns all Events that have been joined ("Checked-in", in Client terminology) by both the user making the request and the user determined with the provided :**id** parameter.

GET base-url.com/**users/:id/common_events_pending**

This petition returns all Events that are still pending (not yet RSVPed to it) by both the user making the request and the user determined with the provided :**id** parameter.

GET base-url.com/**users/:id/common_events**

This petition returns all common Events where both the user making the request and the user determined with the provided :**id** parameter have been invited.

GET base-url.com/**users/:id/last_seen_on**

This petition returns the last Event where a user has performed a Check-in ("join" in backend language) operation -visible to the user performing the request- by the user determined with the provided :**id** parameter.

GET base-url.com/**users/:id/user_profile_pictures**

This petition returns all the UserProfilePicture instances for the user determined with the provided :**id** parameter.

GET base-url.com/**users/:id/associated_connection_request**

This petition returns a single instance of a ConnectionRequest –if any exists- linking both the user making the request and the user determined with the provided :**id** parameter.

GET base-url.com/**users/:id/connection_requests_pending**

This petition returns all ConnectionRequest instances that are pending for confirmation by the user determined with the provided :**id** parameter.

GET base-url.com/**users/name_lookup?string=LOOKUP_STRING**

This petition returns all results (in the form of User instances) deriving from performing a global user name lookup in inJoin database, by matching the provided LOOKUP_STRING to the names and last names of inJoin users.

Events API

Unlike the Users API, Events holds some nested resources, namely event_comments and event_faves, which also provide their own CRUD API at the endpoints (respectively):

base-url.com/**events/:id/comments**

and

base-url.com/**events/:id/faves**

On the other hand –and in similarity to the Users API-, the Events API also has the basic CRUD API, plus several other services of interest, among which are:

GET base-url.com/**events/:id/pictures**

This petition returns all Pictures belonging to the Event instance determined by the provided :**id** parameter.

GET base-url.com/**events/:id/joined_users**

This petition returns all User instances that have Joined ("Checked-in" in Client terminology) the Event instance determined by the provided :**id** parameter.

GET base-url.com/**events/:id/confirmed_users**

This petition returns all User instances that have confirmed ("Joined" in Client terminology) the Event instance determined by the provided :**id** parameter.

GET base-url.com/**events/:id/invited_users**

This petition returns all User instances that have been invited to the Event instance determined by the provided :**id** parameter.

GET base-url.com/**events/:id/associated_invitation**

This petition returns a single EventInvitation instance –if any exists- that invites the user performing the request to participate in the Event instance determined by the provided :**id** parameter.

PUT base-url.com/**events/:id/join**

This performs a backend Join ("Join" in Client terminology) operation by the user performing the request, to the Event instance determined by the provided :**id** parameter.

PUT base-url.com/**events/:id/check-in**

This performs a backend Check-in ("Check-in" in Client terminology) operation by the user performing the request, to the Event instance determined by the provided :**id** parameter.

GET base-url.com/**events/:id/associated_fave**

This petition returns a single EventFave instance –if any exists- in case the user performing the request has previously Faved the Event instance determined by the provided :**id** parameter.

Places API

Another important resource is the Place Model class, which presents a CRUD API as well as other classes, and additionally offers the following services:

GET base-url.com/**places/around_location?lat=LATITUDE&long=LONGITUDE&rad=RADIUS**

This petition returns all places found in the inJoin database around a specific circular geographical region, determined by the provided LATITUDE, LONGITUDE and RADIUS parameters.

Pictures API

As it was the case for the Event resources, Pictures also provides a basic CRUD API, and contains a couple of nested resources, namely the PictureComments and PictureFaves related to each Picture instance, and they also possess their own CRUD API at the endpoints:

base-url.com/**pictures/:id/comments**

and

base-url.com/**pictures/:id/faves**

Additionally, it also facilitates some extra API functionality, via:

GET base-url.com/**pictures/:id/associated_fave**

This petition returns a single PictureFave instance –if any exists- in case the user performing the request has previously Faved the Picture instance determined by the provided :**id** parameter.

UserProfilePictures API

UserProfilePictures defines the resource endpoint for accessing the Model instance for the profile pictures of users, and its basic functionality is limited to only the basic CRUD interface.

EventInvitations API

Similarly to UserProfilePictures, the EventInvitations API is reduced to the basic CRUD interface allowing the basic operations for Event invitations in inJoin.

ConnectionRequests API

ConnectionRequests also requires a very basic functionality, and has therefore availability for the CRUD API without the need for any other more complex interface services.

8. Conclusions

At the beginning of this document, all the motives and rationale behind the development of the project were explained, and this allowed for the definition and shaping of the Application concept to develop.

Afterwards, the preliminary guidelines for the project were established, and based on these guidelines, the main objectives of the execution of the project were depicted, in order to steer the development process in an appropriate and coherent manner.

In the first stage, all the technologies required for developing a mobile application for the iOS platform were carefully studied, in order to get familiarized with the Objective-C language, the XCode IDE and other useful development tools.

Then next step taken was defining the system architecture as a Client/Server model, with a backend REST API used as means of communication. The system consists on a database, some servers running the backend code on the Ruby language and the final iOS Application Client. After this was done, the main functionalities and requirements for the application were specified, which in turn allowed the UI design of the Client application, thus making it possible to define the Data Model and all the specific petitions required from the REST API by the Client, in order to show all necessary data.

The next logical step was then to develop the backend API in Ruby to implement all required Client services, which was then followed by the development of the Client Application itself. Once all initial test work was done, the backend was deployed into the cloud, more specifically, into Amazon Web Services (AWS) in order to fulfill easy scalability and wide availability of the services, which then allowed for deploying Beta testing versions of the App into Apple's TestFlight platform, to validate and get feedback from a bit less than 50 users at the time of writing this document.

As any process as complex as this, made by different parts and a wide variety of technologies in use, many challenges are always present in all stages of design and development. From going through the learning curves of diving into new languages, frameworks and inner workings of certain platforms, to the limitations inherent to those platforms and

sometimes even the more "bureaucratic" tasks, such as Apple's code signing, development profiles and certificates creation, management and maintenance, all of which add complexity that goes beyond the "development process" itself, and are always worth considering when incurring into these types of projects for any developers out there deciding to do so. Additionally, when dealing with an application architecture that is "split" between more than one element (in the case of inJoin one element is the Client and the other is the backend API Ruby application), many bugs and problems may arise from this "splitting" process, relating to the required communication between all the elements, which in turn also adds a layer of complexity and possibility of failures that can be harder to detect or debug.

As it has been mentioned before, inJoin is currently available and undergoing a Beta Testing process, done by means of using Apple's TestFlight platform, and it's set to be improved until its release into the iOS App Store in the near future.

In order to conclude it's also important to provide some possible future directions for inJoin. The most basic things to improve are related to usability. Since it has been available via TestFlight, users have been very helpful in determining a few things that can be implemented in a better way. For example, the ability to navigate by swiping sideways among the details of pictures inside a Moment, which is currently not possible, and has been requested repeatedly by testers. In another note, it would also be interesting to be able to integrate the App with social platforms in a deeper way. For example, being able to post a specific picture, or a selected group of them, or even an entire moment into a Facebook album, or a single picture into Instagram. This would allow people to post exactly what they want into each of their most used social networks, while keeping everything else privately in inJoin.

Developing inJoin has been a very interesting and fulfilling learning experience, and has given a sense of global perspective of what it takes to put a rather simple idea and make it into a reality, while doing everything that a final product with the standards that the current user's market demands.

9. Appendix

9.1 Mobile Application downloads growth

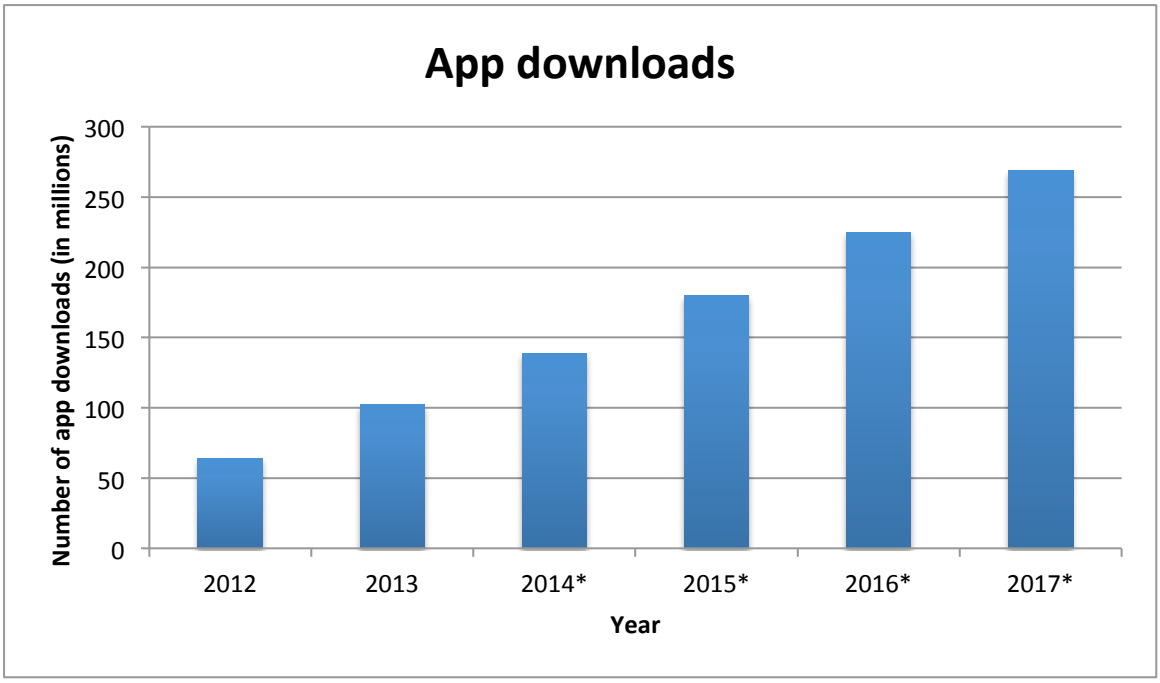

Figure 9.1: Projection for global Mobile Application downloads growth trend. Source: http://www.gartner.com/newsroom/id/2592315.

10. References

10.1 Electronic references

Amazon Web Services Inc. (2015). Amazon EC2 Pricing. Available: http://aws.amazon.com/ec2/pricing/. Last accessed September 7, 2015.

Amazon Web Services Inc. (2015a). EB Command Line Interface. Available: http://docs.aws.amazon.com/elasticbeanstalk/latest/dg/eb-cli3.html. Last accessed September 7, 2015.

Amazon Web Services Inc. (2015b). AWS SDK for Ruby. Available: http://aws.amazon.com/sdk-for-ruby/. Last accessed September 7, 2015.

Apple Inc. (2010). Learning Objective-C: A Primer. Available: https://developer.apple.com/library/mac/#referencelibrary/GettingStarted/Learning_Objective-C_A_Primer/_index.html. Last accessed January 23, 2014.

Apple Inc. (2012). The Objective-C Programming Language. Available: https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ObjectiveC/ObjC.pdf. Last accessed January 23, 2014.

Apple Inc. (2012a). XCode Overview. Available: https://developer.apple.com/library/mac/documentation/ToolsLanguages/Conceptual/Xcode_Overview/Xcode_Overview.pdf. Last accessed September 7, 2015.

Apple Inc. (2012b). iOS Provisioning Portal. Available: https://developer.apple.com/ios/manage/overview/index.action. Last accessed January 23, 2014.

Apple Inc. (2012c). iOS Dev Center. Available: https://developer.apple.com/devcenter/ios/index.action. Last accessed January 23, 2014.

Apple Inc. (2012d). iTunes Connect. Available: https://itunesconnect.apple.com/. Last accessed January 23, 2014.

Apple Inc. (2015). Mac Apps That Use Garbage Collection Must Move to ARC. Available: https://developer.apple.com/news/?id=02202015a. Last accessed September 7, 2015.

Apple Inc. (2015a). TestFlight Beta Testing. Available: https://developer.apple.com/testflight/index.html. Last accessed September 7, 2015.

Cocoapods. (2015). Getting started. Available: https://guides.cocoapods.org/using/getting-started.html. Last accessed on September 7, 2015.

Fielding, R. (2000). Representational State Transfer (REST). In *Architectural Styles and the Design of Network-based Software Architectures*. Available: http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm. Last access on September 7, 2015.

Gartner (2013). Gartner Says Mobile App Stores Will See Annual Downloads Reach 102 Billion in 2013. Available: http://www.gartner.com/newsroom/id/2592315. Last accessed September 7, 2015.

Google Inc. (2015). Compute Engine Pricing. Available: https://cloud.google.com/compute/#pricing. Last accessed September 7, 2015.

Grimmer, L. (2006). Interview with David Heinemeier Hansson from Ruby on Rails. Available: https://web.archive.org/web/20130225091835/http://dev.mysql.com/tech-resources/interviews/david-heinemeier-hansson-rails.html. Last access September 7, 2015.

Heroku. (2015). Heroku Toolbelt. Available: https://toolbelt.heroku.com/. Last accessed September 7, 2015.

Heroku. (2015a). How Heroku works. Available: https://devcenter.heroku.com/articles/how-heroku-works. Last accessed September 7, 2015.

Instagram Inc. (2012). Introducing Instagram Direct. Available: http://blog.instagram.com/post/69789416311/instagram-direct. Last access on September 7 2015.

Matsumoto, Y. (2000). The Ruby Programming Language. Available: http://www.informit.com/articles/article.aspx?p=18225. Last access on September 7, 2015.

Microsoft. (2015). Virtual Machines Pricing. Available: http://azure.microsoft.com/en-us/pricing/details/virtual-machines/. Last accessed September 7, 2015.

Perez, Sarah. (2014). Majority Of Digital Media Consumption Now Takes Place In Mobile Apps. Available: http://techcrunch.com/2014/08/21/majority-of-digital-media-consumption-now-takes-place-in-mobile-apps/. Last access on August 25 2015.

Rails Guides (2015). Getting started with Rails. Available: http://guides.rubyonrails.org/getting_started.html. Last access on September 7, 2015.

Ruby Community (2015). About Ruby. Available: https://www.ruby-lang.org/en/about/. Last access on September 7, 2015.

Southern, Matt. (2013). Facebook Allows Multiple Users To Contribute To Shared Photo Albums. Available: http://www.searchenginejournal.com/facebook-allows-multiple-users-contribute-shared-photo-albums/68069/. Last access on September 7 2015.

Wikimedia Foundation, Inc. (2015). Ruby (programming language). Available: https://en.wikipedia.org/wiki/Ruby_(programming_language). Last access on September 7 2015.

10.2 Bibliographical references

Isted T., Harrington, T. (2011). An Overview of Core Data on iOS devices. In: Taub, M. *Core Data for IOS: Developing Data-Driven Applications for the IPad, IPhone and iPod touch*. Boston, MA: Addison-Wesley Professional. 3-4.

Warren, R. (2011). Objective-C Overview. In: Foster, D. *Objective-C Boot Camp: Foundation and Patterns for iOS Development*. Berkeley, CA: Peachpit Press . 1.

Wentk, R. (2011). Introducing XCode 4. In: Minner, M. ; Miller, B. ; Black, A. *XCode 4*. Indianapolis, In: Wiley Publishing, Inc. (pp. 7-10).

Amies, A., Ning Liu, G., Sluiman, H., & Guo Tong, Q. (2012). Infrastructure as a Service Cloud Concepts. In *Developing and Hosting Applications on the Cloud* (pp. 22-24).

Y Chang, W., Abu-Amara, H., & Feng Sanford, J. (2010). General Information Technologies. In *Transforming Enterprise Cloud Services* (pp. 55-56). Springer Science & Business Media.

Gillette, J. (2007). Language and I MEAN Language. In *Why's (poignant) Guide to Ruby*.