

Bachelor Project Report



SportLook

A social network for sport events

Date: 1 June 2015

Alexandru-Cosmin Vasile, 168134@via.dk

Ibrahim Yildirim, 166802@via.dk

Juraj Petrik, 166740@via.dk

Supervisor: Jan Munch Pedersen, jpe@via.dk

Table of Contents

| | |
|--|----|
| Abstract | 5 |
| 1. Introduction..... | 5 |
| 1.1. Project Environment | 5 |
| 2. Analysis..... | 6 |
| 2.1. Requirements | 6 |
| 2.2. Domain Model..... | 6 |
| 2.3. Use Case Diagram | 7 |
| 2.3.1. Actor Descriptions..... | 8 |
| 2.3.2. Use Case Descriptions and Activity Diagrams | 8 |
| 3. Design | 9 |
| 3.1. System Architecture | 9 |
| 3.2. iOS Application | 10 |
| 3.2.1. Programming Language | 13 |
| 3.2.2. iOS SDK..... | 15 |
| 3.2.3. Application Architecture | 15 |
| 3.2.4. Design Patterns..... | 17 |
| 3.2.5. Persistent Data..... | 19 |
| 3.2.6. Graphical User Interface | 20 |
| 3.3. Web Service Design | 24 |
| 3.3.1. Messaging protocol | 24 |
| 3.3.2. Deployment | 25 |
| 3.3.3. Framework | 25 |
| 3.3.4. Database | 26 |
| 3.4. Third Party Services | 28 |
| 4. Implementation | 29 |
| 4.1. iOS Application Implementation..... | 29 |
| 4.1.1. BaseViewController & Supporting Different OS Versions | 29 |
| 4.1.2. Supporting different iPhone Screen sizes..... | 30 |
| 4.1.3. MapKit | 31 |
| 4.1.4. Reusable Views | 32 |
| 4.1.5. Open Source Components | 34 |
| 4.1.6. Chat implementation | 35 |
| 4.1.7. Push notifications implementation | 38 |

| | |
|---|----|
| 4.2. Web Service Implementation | 40 |
| 4.2.1. User Authentication..... | 40 |
| 4.2.2. Image Upload | 42 |
| 5. Testing..... | 44 |
| 5.1. iOS application Testing | 44 |
| 5.2. Web Service Testing..... | 45 |
| 6. Distribution | 47 |
| 6.1. Apple Member Centre, Certificates & Code Signing..... | 48 |
| 6.1.1. Provisioning Profiles & SSL Certificate for Push Notifications | 48 |
| 6.2. iTunes Connect | 49 |
| 6.3. Updates | 50 |
| 6.4. Sales and Trends | 51 |
| 7. Results..... | 51 |
| 8. Discussion | 52 |
| 8.1. Possible improvements | 52 |
| 8.2. Unexpected situations..... | 54 |
| 8.3. Monetization..... | 54 |
| 9. Conclusion | 55 |
| 10. References..... | 55 |
| 11. Appendices..... | 57 |

Table of Figures

| | |
|--|----|
| Figure 1. Domain model..... | 7 |
| Figure 2. Use case diagram | 7 |
| Figure 3. Activity diagram for the register workflow | 8 |
| Figure 4. Three tier architecture | 9 |
| Figure 5. System architecture overview | 9 |
| Figure 6. Mobile development dilemma | 12 |
| Figure 7. Common operations in Objective-C | 14 |
| Figure 8. Common operations in Swift | 14 |
| Figure 9. iOS layers..... | 15 |
| Figure 10. iOS application architecture..... | 16 |
| Figure 11. The MVC pattern | 17 |
| Figure 12. Apple's MVC pattern | 17 |
| Figure 13. Target-action pattern..... | 18 |
| Figure 14. UIViewController Lifecycle | 19 |
| Figure 15. User info JSON model..... | 20 |

| | |
|--|----|
| Figure 16. iOS distribution..... | 20 |
| Figure 17. iOS app anatomy..... | 21 |
| Figure 18. Views - Controllers relationship | 21 |
| Figure 19. Tab bar used..... | 22 |
| Figure 20. Example of UITextField customization..... | 22 |
| Figure 21. Wireframes UI | 23 |
| Figure 22. Digital design UI..... | 23 |
| Figure 23. Deploying on Heroku..... | 25 |
| Figure 24. Starting a web process | 25 |
| Figure 25. Code snippet for Event.js..... | 26 |
| Figure 26. Code snippet for User.js..... | 27 |
| Figure 27. ER diagram | 27 |
| Figure 28. BaseViewController implementation | 29 |
| Figure 29. Use of BaseViewController method | 30 |
| Figure 30. Size Classes overview..... | 30 |
| Figure 31. Size Classes example | 31 |
| Figure 32. MKMapView example | 31 |
| Figure 33. MKMapViewDelegate methods | 32 |
| Figure 34. mapView regionWillChangeAnimated..... | 32 |
| Figure 35. Create Event screen..... | 33 |
| Figure 36. SLInputView implementation..... | 33 |
| Figure 37. Setting a custom class in the Interface Builder | 34 |
| Figure 38. Using the setTextField () method | 34 |
| Figure 39. Chat UI..... | 35 |
| Figure 40. Initial Chat UI setup..... | 36 |
| Figure 41. Loading chat messages | 36 |
| Figure 42. Sending a push notification..... | 37 |
| Figure 43. Chat sequence diagram | 37 |
| Figure 44. APNs structure | 38 |
| Figure 45. Push notification sequence diagram | 39 |
| Figure 46. Handling a received push notification | 39 |
| Figure 47. Web Service structure | 40 |
| Figure 48. Registration sequence diagram | 41 |
| Figure 49. Authorized - Unauthorized flow | 42 |
| Figure 50. Sequence diagram for image upload resizing | 43 |
| Figure 51. Sequence diagram for direct image upload..... | 43 |
| Figure 52. Test case for logging in with email..... | 44 |
| Figure 53. DataProvider's use of console logs | 45 |
| Figure 54. A collection of POSTMAN requests | 46 |
| Figure 55. Testing a request with POSTMAN | 46 |
| Figure 56. Apple's distribution flow..... | 47 |
| Figure 57. Apple Member Centre | 47 |
| Figure 58. Example of provisioning profiles issue | 48 |
| Figure 59. iTunes Connect dashboard..... | 49 |
| Figure 60. Xcode's Application Uploader | 50 |
| Figure 61. Sales and trends | 51 |
| Figure 62. Scaling with Parse..... | 53 |

Abstract

The objective of the project was to develop a client-server system, where users can interact through the client (iOS application) and socialize around the topic of sport events.

Each component of the system was chosen to support the mobile applications environment. The client was developed using the iOS SDK and user interface components provided by the Cocoa Touch framework. This allows for a high performance and a favourable user experience.

The outcome of the project is a fully functional system, which allows users to interact as expected. All of the system components have been deployed successfully and the client is available for download in Apple's AppStore (digital distribution platform for mobile applications).

The result is a system that can support a moderate amount of users, further work being needed for a larger user base. The mobile application benefits from exposure to the public and it has the potential of becoming a popular and successful social application.

1. Introduction

The project will cover the development of the iOS application named *SportLook*, optimized for iPhone devices. The application is a renewed version of an already existing iOS application named [MoveBizz](#), which was available in Apple's AppStore. MoveBizz provides its users with the opportunity to meet other people who share the same sport interests through sport events. Unfortunately, the application lacks good functionality and good user experience. The concept of the application is that its users can create sport events and other users can view these events and all related information to the events such as name, date, location, participants etc.

The objective of this project is to build an iOS application, optimized for iPhone devices, that can share data between users. It builds on the concepts of MoveBizz, but will benefit from extra functionality which will be documented.

1.1. Project Environment

The outcome of this project will be an iOS application, which is a requirement from the company *Move Sport* (further addressed as the client). The client originally had the idea of this concept and hired a development team to create the first version - MoveBizz. Unfortunately, the end product is not a stable system and it cannot be used properly for its purpose due to poor performance.

The application will present users with sport events in their area, and then give them the opportunity to join the event and interact with the other participants. One of the requirements from the client is that the server can communicate with different mobile operative systems. The reason for that is because the client hired another development team ([AppUdvikleren.dk](#)) to build the Android version of this application. During the development of SportLook the current

development team had to coordinate with them in order to make the application have a similar user interface and similar functionality, as present in the iOS application.

The segment of users of this application are both genders in the age group of 12 to 65 and have an active lifestyle or want to socialize through a game of football, chess or different activities. A common person wants to meet other people that share the same sport interests as themselves.

2. Analysis

2.1. Requirements

The requirements have been divided into functional and non-functional, as follows:

Non-functional

- Application must be an iOS application, optimized for iPhone devices.
- Application must support iOS versions 7.1 - 8.3.
- Application must support all iPhone screen sizes (iPhone 4, 4S, 5, 5S, 6 & 6 plus).

Functional

- The user must be able to find sport events close to his location.
- The user must be able to view details about any sport event.
- The user must be able to create an event.
- The user must be able to join / leave an event.
- The user must be able to send messages to people that have joined the same event.
- The user must be able to invite users to events.
- The user must be able to change his profile information.
- The user must be able to find other users with similar interests close to his location.

2.2. Domain Model

The Domain Model diagram is a first approximation of the model that will be used for the application. Each user will be able to attend multiple events that have a sport associated with them. The user also has a specified sport preference to personalize his experience.

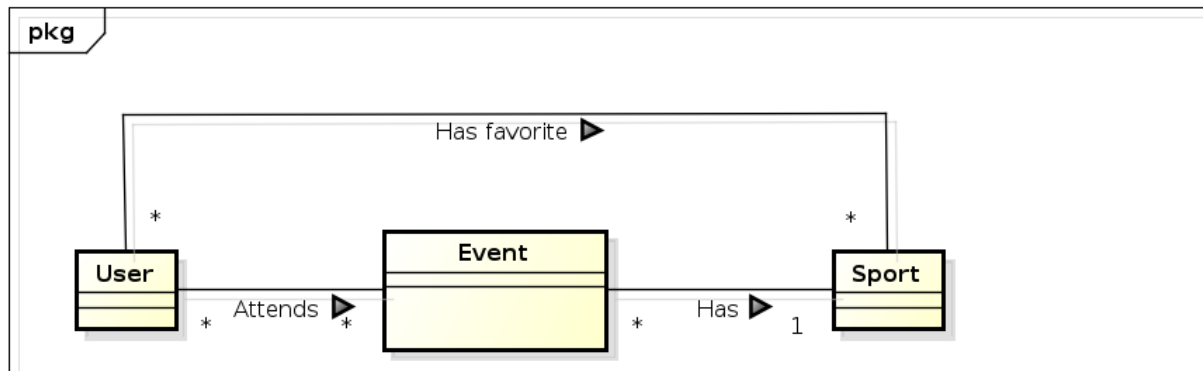


Figure 1. Domain model

2.3. Use Case Diagram

Similarly to the Domain Model, the Use Case diagram below accurately depicts the main use cases available to users of the app. As can be inferred from the diagram, the user first has to register or log in to perform any other use case in the app.



Figure 2. Use case diagram

2.3.1. Actor Descriptions

Name: Guest

Description: This actor represents a user that is not logged in the system. The only actions available to him are to enter the system either by logging in or registering

Name: User

Description: This actor represents a user that is logged in the system.

2.3.2. Use Case Descriptions and Activity Diagrams

Use case description and activity diagrams were generated for every Use Case. One is presented in the report, while others can be found in the Appendices section.

| | |
|------------------------|---|
| Name | Register |
| Purpose | The purpose of this use case is to allow new users to be created in the system |
| Preconditions | User has launched the application |
| Post conditions | New user is created and logged in |
| Limitations | Guest has to be connected to the Internet |
| Assumptions | Guest has an iPhone. Guest has downloaded and installed the application on his phone. |

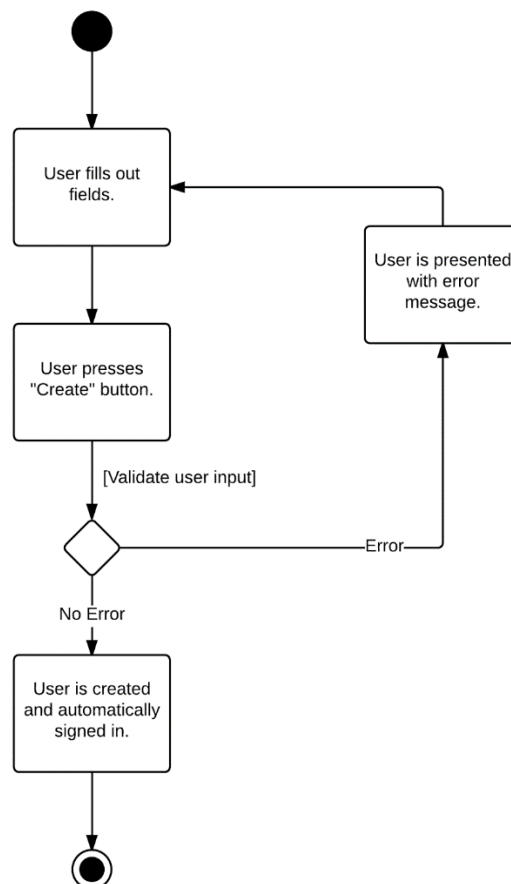


Figure 3. Activity diagram for the register workflow

3. Design

3.1. System Architecture

The system developed makes use of a client-server architecture, in which the presentation, application processing and data storage functions are separated. More specifically, a three-tier architecture is used, which is composed of a presentation tier, an application logic tier and a data storage tier. This approach allows for a flexible and reusable system.

The presentation tier is the topmost level of the system and it's responsible for displaying information to end-users. The application logic tier is responsible for performing processing and enforcing business logic rules. The data storage tier is responsible for storing and retrieval of persistent data.

The following is a diagram which illustrates this concept.



Figure 4. Three tier architecture

A full system architecture overview can be seen below, complemented with a short description of each of the system's components.

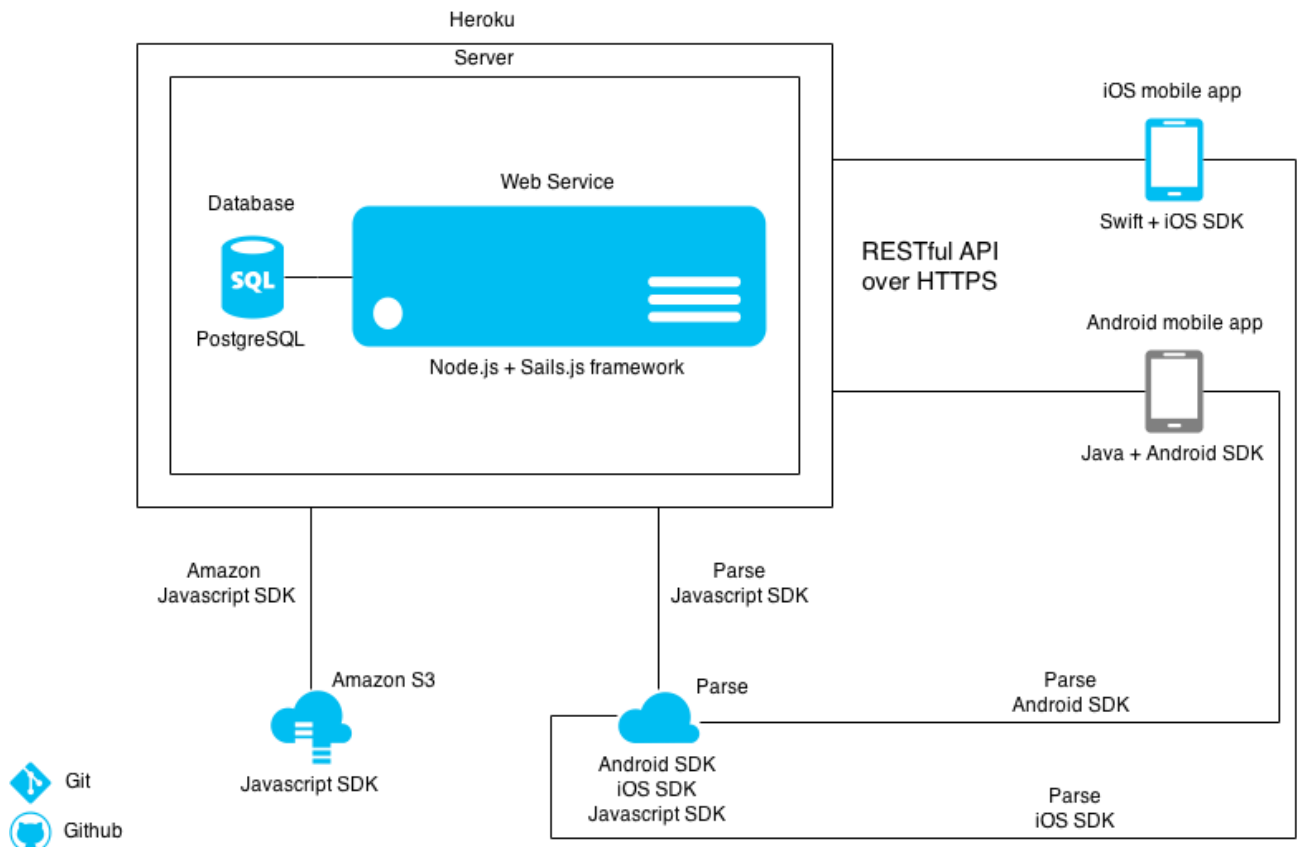


Figure 5. System architecture overview

- **iOS mobile app.** Developed using the Swift programming language and the iOS SDK.
- **Android mobile app.** In the diagram above, this component is greyed out because it was not developed by the current development team. It's mentioned because it has a significant importance in the development of the server and choice of the 3rd party services used. Those must be usable by both the Android and iOS applications.
- **Server.** The server's components are the Web Service and the database. The Web Service is written in Node.js while using the Sails.js framework. The database is PostgreSQL and the Web Service incorporates a database adapter for storing and retrieving data from the database. Also, the Web Service communicates with a RESTful API over HTTPS with the iOS and Android applications. Heroku (a Platform as a Service) is used for hosting the server.
- **Amazon S3.** Amazon S3 is an Infrastructure as a Service (IaaS) solution. Amazon's Simple Storage Service is used for object storage. It communicates with the server via Amazon Web Service's JavaScript SDK.
- **Parse.** Parse is a Backend as a Service (BaaS) solution, which is used for data storage and its push notification services. It communicates with the server via Parse's JavaScript SDK. Additionally, it also communicates with the iOS mobile app via Parse's iOS SDK and with the Android mobile app via the Parse's Android SDK.
- **Git.** Git is a distributed Version Control System used for code collaboration and code version control. Although not directly a component of the system, it's mentioned due to its heavy use.
- **GitHub.** GitHub is a web-based Git repository hosting service used for code hosting. Although not directly a component of the system, it's mentioned due to its heavy use.

3.2. iOS Application

A major decision that needs to be taken at the start of developing an iOS mobile application is to decide which technology should be used.

There are 4 ways that emerged for iOS mobile development, as explained below:

1. **Web Application.** Developed using JavaScript, HTML & CSS. This approach makes use of web development skills. Essentially a mobile website.
2. **Hybrid application.** Developed using JavaScript, HTML & CSS, but hosted inside a native application that utilizes the mobile platform's WebView. Essentially a chromeless browser window that's configured to run full screen. Can access device hardware. Most of the times also making use of a framework or platform such as Apache Cordova.
3. **Cross-platform application.** Developed using a platform (such as Xamarin) and the same code base (written in either C# or Ruby). The code written is then compiled to native ARM assembly code, which can run on iOS devices.
4. **Native application.** Developed using programming languages, frameworks and tools provided by Apple. The code written in either Objective-C or Swift accesses Apple's native APIs.

Each approach has its advantages and disadvantages. The following table addresses a few advantages and disadvantages:

Table 1. Mobile development approaches

| | Native | Cross-platform | Hybrid | Web |
|--|---------------|-----------------------|---------------|-------------|
| Cross-Platform | | ✓ | ✓ | ✓ |
| Low-Cost | | | ✓ | ✓ |
| Easy Maintenance | | | ✓ | ✓ |
| Content Restrictions | ✓ | ✓ | ✓ | |
| High performance | ✓ | | | |
| Access Device Hardware | ✓ | ✓ | ✓ | |
| GPS Location Access | ✓ | ✓ | ✓ | ✓ |
| Touchscreen Gestures | ✓ | ✓ | ✓ | Swipes only |
| Clean Graphics & Animation | ✓ | ✓ | ✓ | |
| Offline Connectivity | ✓ | ✓ | ✓ | |
| App Store visibility | ✓ | ✓ | ✓ | |
| Need to pay for Apple Developer program | ✓ | ✓ | ✓ | |
| Access Apple's API directly | ✓ | | | |

After comparing the available options, the decision taken was to build a native iOS application. The decision was taken due to the following reasons:

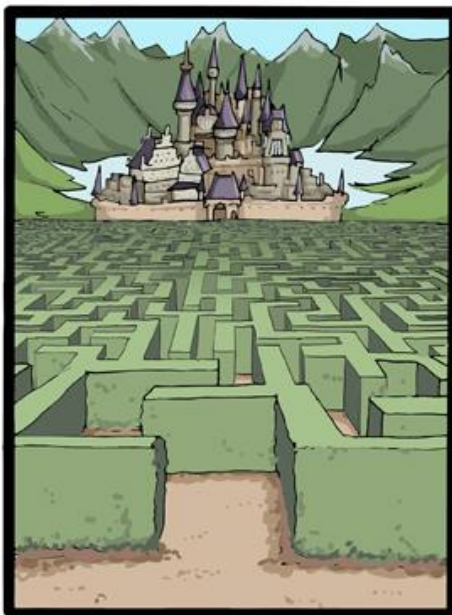
- Apple's iOS API is used by direct access. This minimizes possible bugs and eliminates the need of an extra layer between the code written and the access of Apple's iOS API. Also, it makes debugging more straight-forward.
- Only the enrolment fee into the Apple Developer Program is a cost. Most of the cross-platform and hybrid application solutions are based on a subscription model. As one of the project's constraints is to keep the development cost low, the native approach is the logical choice.
- Exposure into Apple's AppStore is needed. Therefore the web application approach needs to be discarded, because it doesn't allow this exposure.
- Same-day access to the newest Apple iOS APIs. New software is often shipped with an amount of bugs. Having an extra layer between accessing the newest native iOS APIs and the code written on a cross-platform solution can significantly increase the difficulty in debugging. Therefore the native approach would be a good decision, in order to avoid this problem.

- High performance is needed. One of the project's non-functional requirements is that the application should have high performance. Unfortunately, this cannot be achieved with the web or hybrid application approach on an equal level with a native approach. Native applications communicate directly with the operating system, while the web or hybrid applications communicate with the browser, which communicates with the operating system. Therefore there is an extra layer that needs to be performed by those applications, making them slower.
- The development team doesn't have a strong background in web development. One of the main benefits of using a web or hybrid approach is the re-use of web development skills. Since this is not the case, this benefit does not apply.
- The application is platform specific. One of the client's requirements was that the application should be iOS only, therefore the benefits of using a cross-platform approach with the same code base across multiple platforms (Android, iOS, Windows Phone) do not apply.

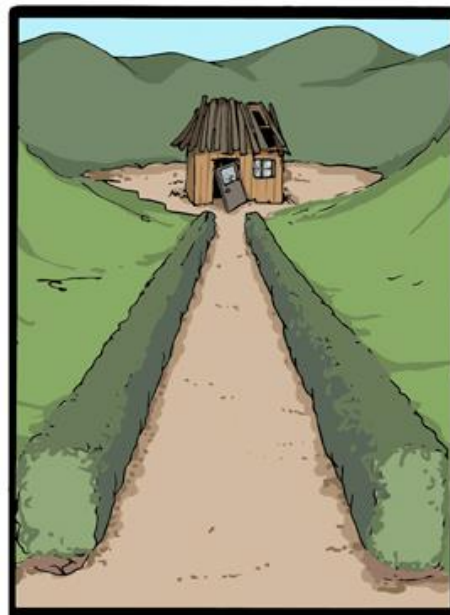
The development approach for mobile applications was always a big debate in the mobile community, but in the end the performance is always better while using the native approach.

The dilemma of mobile apps development

Develop a native app for each device and maintain several projects



Use a unique framework (Phonegap, Adobe Air, Appcelerator) and maintain only one project



CommitStrip.com

Figure 6. Mobile development dilemma

3.2.1. Programming Language

There are 2 choices in the programming language to be used when developing native iOS apps: Objective-C and Swift.

Objective-C is Apple's initial programming language for the OS X and iOS operating systems, originally developed in the early 1980s. This is Apple's brief description of the language:

“Objective-C is the primary programming language you use when writing software for OS X and iOS. It’s a superset of the C programming language and provides object-oriented capabilities and a dynamic runtime. Objective-C inherits the syntax, primitive types, and flow control statements of C and adds syntax for defining classes and methods. It also adds language-level support for object graph management and object literals while providing dynamic typing and binding, deferring many responsibilities until runtime.”

On the other hand, Swift is a new programming language released by Apple in June 2014. Swift is designed to work with Apple's Cocoa and Cocoa Touch frameworks and the existing Objective-C code written for Apple products. This is Apple's brief description of the language:

“Swift is a new programming language for iOS and OS X apps that builds on the best of C and Objective-C, without the constraints of C compatibility. Swift adopts safe programming patterns and adds modern features to make programming easier, more flexible, and more fun. Swift’s clean slate, backed by the mature and much-loved Cocoa and Cocoa Touch frameworks, is an opportunity to re-imagine how software development works.”

The decision to use Swift was taken due to several factors:

- Swift performs faster than Objective-C in several benchmarks. Swift objects can call one another without needing to perform the message sending which is a bottleneck on Objective-C performance.
- Swift is designed to provide seamless compatibility with Cocoa and Objective-C. The Objective-C APIs can be used in Swift and the same project can make use of both languages via a bridging header.
- A Swift file has only one declaration, while an Objective-C file would have both a header and an implementation file. Having to read separate header files slows down compilation.
- Swift makes use of modern programming language features such as optionals, control flow statements, type safety, tuples support, type inference, optional semicolons, dictionaries, closures, generics, string interpolation.
- Apple focuses on Swift when developing their newest APIs and providing code snippets for it.
- The Swift syntax is more concise and has improved readability compared to the Objective-C one.

Objective-C syntax

```
//Non-Mutable Dictionary
NSDictionary * myFixedDictionary = @{@"key1":@"This is value1",@"key2":@"This is value2"};

// Mutable Dictionary
NSMutableDictionary * myFlexibleDictionary = [[NSMutableDictionary alloc] init];
//Set Object using Old Syntax
[myFlexibleDictionary setObject:@"This is value1" forKey:@"key1"];
//Set Object using New Syntax
[myFlexibleDictionary setObject:@"This is value2" forKey:@"key2"];

NSLog(@"myFixedDictionary: %@",myFixedDictionary);
NSLog(@"myFlexibleDictionary: %@",myFlexibleDictionary);

//Non-Mutable Array
NSArray * myFixedArray = [[NSArray alloc] initWithObjects:@"Object1", @"Object2",nil];

//Mutable Array
NSMutableArray * myFlexibleArray = [[NSMutableArray alloc] init];
//Add Object using Old Syntax
[myFlexibleArray addObject:@"Object1"];
//Add Object using New Syntax
myFlexibleArray[1] = @"Object2";

//No String interpolation
NSLog(@"myFixedArray: %@",myFixedArray);
NSLog(@"myFlexibleArray: %@",myFlexibleArray);
```

Figure 7. Common operations in Objective-C

Swift syntax

```
//Constant Dictionary (Almost similar with Non-Mutable Dictionary)
let myFixedDictionary = ["key1":"This is the value1","key2":"This is value2"]

//Variable Dictionary (Almost similar with Mutable Dictionary)
var myFlexibleDictionary = [String : String]()
myFlexibleDictionary["key1"] = "This is the value1"
myFlexibleDictionary["key2"] = "This is the value2"

println("myFixedDictionary: \(myFixedDictionary)")
println("myFlexibleDictionary: \(myFlexibleDictionary)")

//Constant Array (Almost similar with Non-Mutable Array)
let myFixedArray: [String] = ["Object1", "Object1"]

//Variable Array (Almost similar with Mutable Array)
var myFlexibleArray = [String]()
myFlexibleArray.append("Object1")
myFlexibleArray.append("Object2")

//String interpolation
println("myFixedArray: \(myFixedArray)")
println("myFlexibleArray: \(myFlexibleArray)")
```

Figure 8. Common operations in Swift

3.2.2. iOS SDK

The development tool used is Apple's IDE, Xcode. The IDE has helpful features such as an interface builder, debugging and source editor. The iOS SDK is included in Xcode.

At the highest level, iOS acts as an intermediary between the underlying hardware and the application created. The application does not talk to the underlying hardware directly. Instead, it communicates with the hardware through a set of well-defined system interfaces. These interfaces make it easy for the application to run consistently on devices having different hardware capabilities.

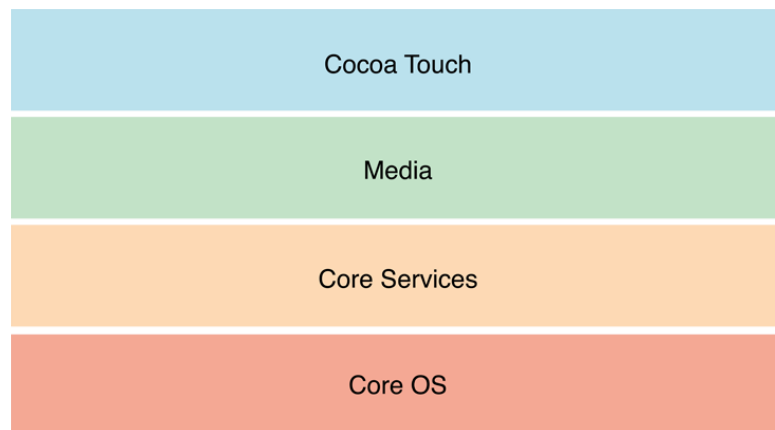


Figure 9. iOS layers

The use of higher-level frameworks over lower-level frameworks is preferred whenever possible. The higher-level frameworks provide object-oriented abstractions for lower-level constructs. Elements from all 4 layers were used in the app development, as follows:

- **Cocoa Touch:** multi-touch, core motion, view hierarchy, controls, alerts, map kit, image picker.
- **Media:** JPEG, PNG, core animation.
- **Core Services:** collections, networking, core location, preferences.
- **Core OS:** Keychain access.

3.2.3. Application Architecture

The application makes use of the following architecture:

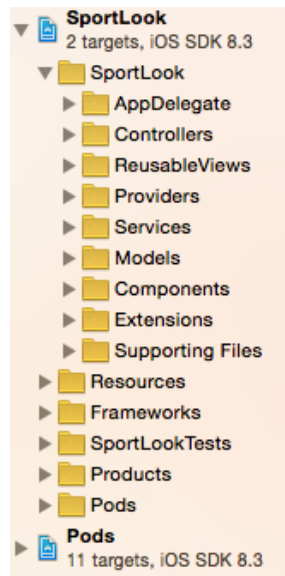


Figure 10. iOS application architecture

- **The project file.** Contains information about deployment, supported device orientation, assets, Apple frameworks and libraries used, capabilities, build settings etc.
- **AppDelegate.** This is the entry point of the application containing methods that are called when the application opens in various situations: app is not running and is launched, app is launched from a push notification, app is running in the background etc. Global app customization takes place after the app is launched.
- **Controllers.** An iOS application is a set of Controllers linked together. Each controller has its own .xib file, which is the user interface for that controller.
- **ReusableViews.** Those are user interface components that are used in more than one place. In order to keep the code clean and reusable, the use of such components is preferred whenever possible.
- **Providers.** The providers communicate with the server and return the appropriate data, to be used in the appropriate screen. The HTTPS requests are asynchronous.
- **Services.** The services perform operations in a non-object oriented manner, operations that need to be performed more than once and can be isolated from the Controllers.
- **Models.** All the data returned from the server is mapped to a model.
- **Components.** Contains app constants, a bridge header and user interface components.
- **Extensions.** Contains new functionality added to Apple frameworks (such as UIKit).
- **Supporting files.** Contains a preference list with information about the application.
- **Resources.** Contains assets as images, icons and fonts.
- **Frameworks.** Contains Apple frameworks and open source frameworks used for development.
- **Pods.** Contains open source libraries used for development.

3.2.4. Design Patterns

The Model-View-Controller (MVC) pattern is the skeleton of every iOS app. MVC categorizes the objects of an app in one of the 3 roles: model, view or controller. The models keep track of the app's data, views display the user interface and make up the content of an app, and controllers manage the views. By responding to user actions and populating views with content from the data model, controllers serve as a gateway for communication between the model and views.

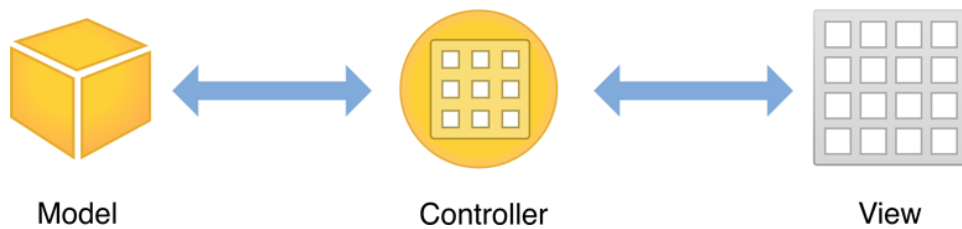


Figure 11. The MVC pattern

Apple uses a slightly more complex version of the MVC pattern, as the following diagram shows:

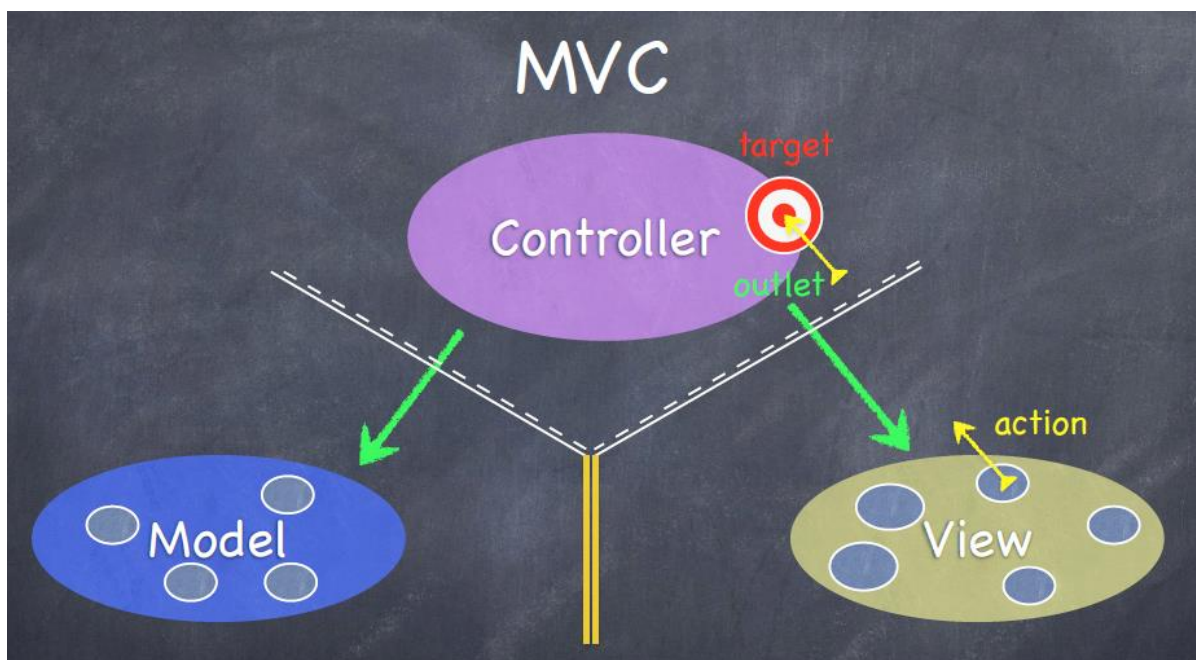


Figure 12. Apple's MVC pattern

- The Controller can communicate with its Model and View.
- The Model and View cannot communicate.
- The View can communicate with its Controller via the target-action design pattern, which is explained next.

The Target-action pattern is a simple design in which one object sends a message to another object when a specific event occurs. The action message is a selector defined in source code and the target (the object that receives the message) is an object capable of performing the action, typically a Controller. The object that sends the action message is usually a UI control (such as a button or switch) that can trigger an event in response to user interaction such as tap or value change.

An example in the Login screen of the app is used to illustrate this pattern:

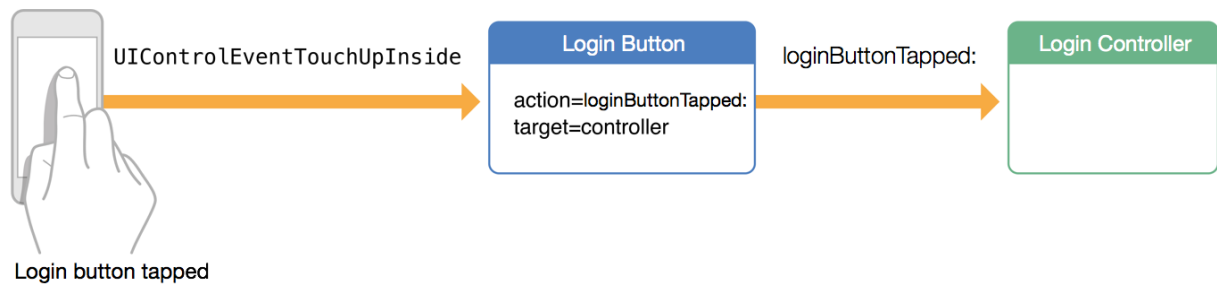


Figure 13. Target-action pattern

The user wants to login and taps the login button (created in the user interface). First, the action '*loginButtonTapped:*' is implemented to perform the logic of logging in. Next, the button's Touch Inside event is registered to send the '*loginButtonTapped:*' action message to the Login Controller that implements the method. Target-action is a powerful mechanism for defining interaction and sending information between different parts the app.

Another important concept is that each Controller managing a View has life cycle methods, which are called automatically at the appropriate times by the operating system when the Controller is loaded/shown/hidden. The following diagram illustrates what are those methods and when are they being called:

- **viewDidLoad.** Called when the Controller's view is loaded from .xib (the user interface). Used for initial setup in all of the Controllers.
- **viewWillAppear.** Called right before the view appears. Also, if another view is shown and then the user returns to this view, the method is called again, every time before it becomes visible.
- **viewDidAppear.** Used for customization after the view appeared, such as triggering animations.
- **viewWillDisappear.** Called right before the view disappears. Used when logic needs to be performed before switching to another view.

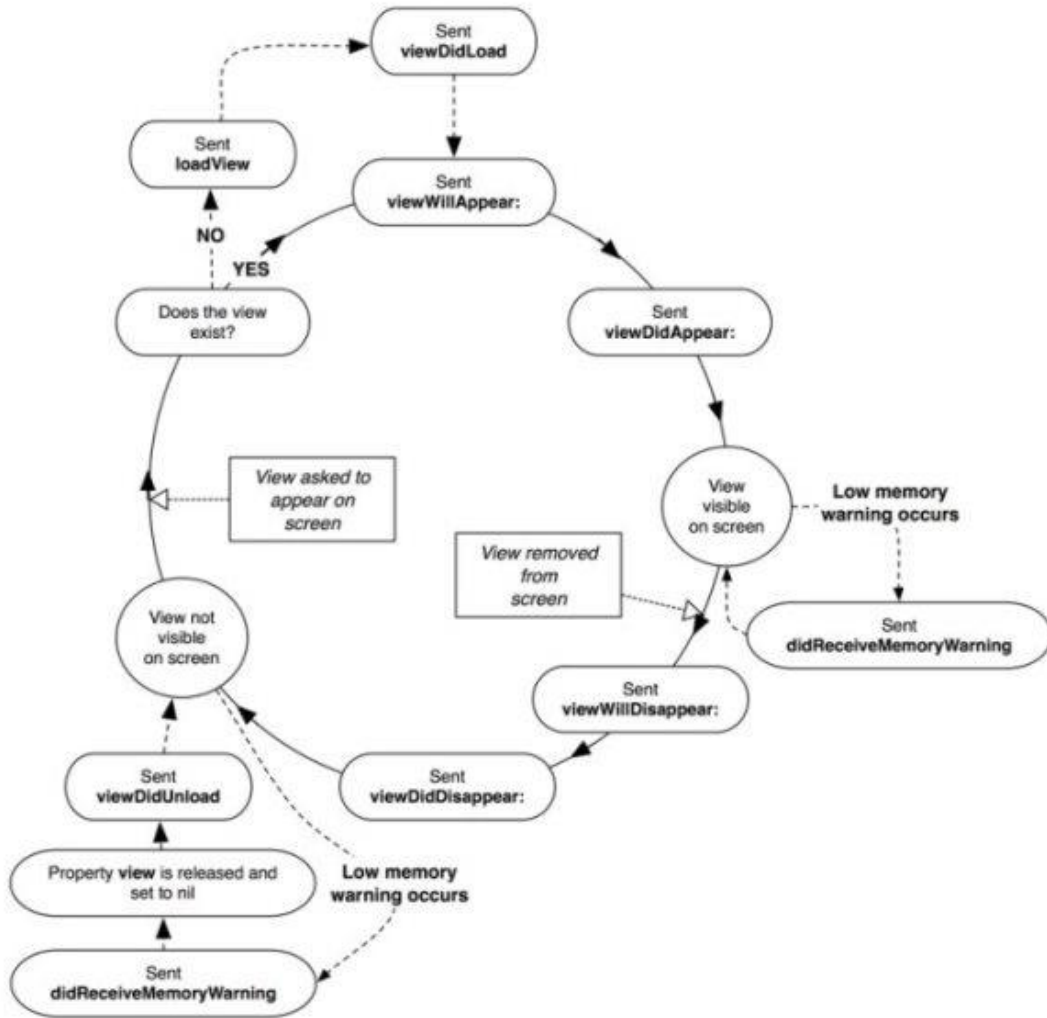


Figure 14. UIViewController Lifecycle

3.2.5. Persistent Data

It was defined in the project's limitations that there will be no persistent data saved in the iOS application: *There will be no cache support in the mobile app, the app relying on an Internet connection for being functional.*

Therefore, in order to acquire data for the view that needs to be displayed, each Controller will request the data from the server at the appropriate time. However, a minimum amount of information is stored in the app, which is necessary for the app's functionality. The data stored is the current logged user information: the user name, id and token.

The user id and the user name need to be persistent in the app, due to their use for the Chat functionality (discussed in the Implementation section). The user token needs to be persistent due to the necessity of sending it as a parameter to every HTTPS request when communicating with the server. This happens after the login or sign up requests.

Those 3 pieces of information are saved in Apple's Keychain when the user logs in or sign ups. Keychain is a password management system available on iOS devices. Although this information is not exactly what the Keychain is designed to hold, it is stored in the Keychain due to being user sensitive data.

The following is an example of user information that is persistent in the iOS app, as a JSON model received from the server.

```
{
  "name": "Alexandru Vasile",
  "id": 5,
  "access_token": "28dfba2e9820cbf5e071fdf234410b7d"
}
```

Figure 15. User info JSON model

3.2.6. Graphical User Interface

The graphical user interface was built to support the following iOS devices: iPhone 4(640x960 px), iPhone 4S (640x960 px), iPhone 5(640x1136 px), iPhone 5S (640x1136 px), iPhone 6(750x1334 px), iPhone 6 Plus (1242x2208 px). There is no support for other iOS devices, as stated in the scope of the project. Also, only the portrait device orientation is supported. This minimizes the development work, due to removing the need to adapt the user interface for the landscape orientation.

Another important issue is deciding which iOS versions to support, since the UI components (provided by the Cocoa Touch framework) can have design differences in different iOS versions. The following figure (statistics provided by Apple) shows that only 2% of the devices are running an iOS version lower than 7, therefore the minimum supported version is iOS 7.0 and the latest is 8.3. As there are no major differences between versions 7 and 8, it was decided to support version 7, even though only 16% of the devices are running it.

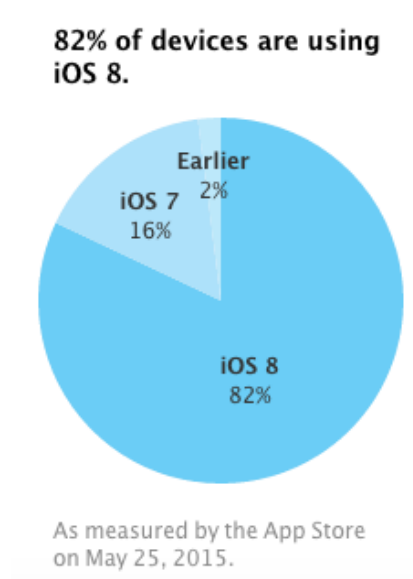


Figure 16. iOS distribution

The user interface was designed according to Apple's guide called '*iOS Human Interface Guidelines*'. Patterns, best practices and recommendations can be found in this guide, along with UI components provided by the Cocoa Touch framework. The anatomy of an iOS app is a gathering of UI components, as it can be seen in the following illustration:

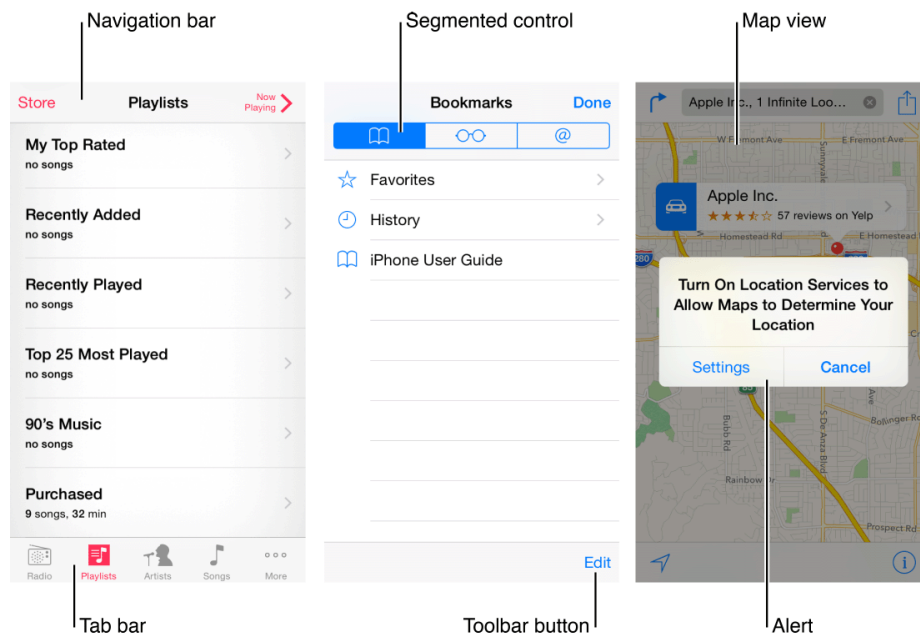


Figure 17. iOS app anatomy

The basic UI component provided by the Cocoa Touch framework is 'UIView'. All other UI elements inherit from it and each controller manages a container view that holds all the views that need to be visible at a given time. The relationship between the views and the controllers can be observed in the following image:

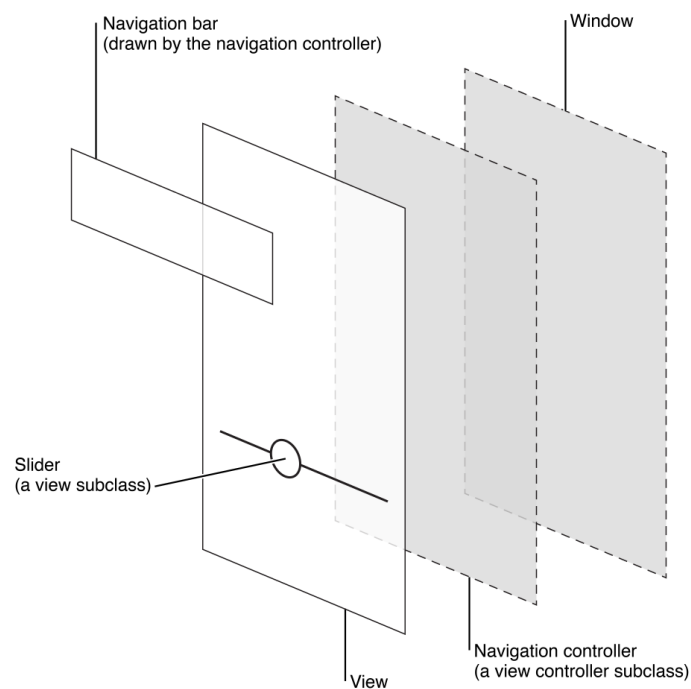


Figure 18. Views - Controllers relationship

In order to support all the required screen sizes, the concept of 'Auto Layout' is used. Apple defines this concept as following:

'Auto Layout is a system that lets you lay out your app's user interface by creating a mathematical description of the relationships between the elements. You define these relationships in terms of constraints either on individual elements, or between sets of elements. Using Auto Layout, you can create a dynamic and versatile interface that responds appropriately to changes in screen size, device orientation, and localization.'

Constraints are used as relationships between UI elements and they are enforced at run time, making the user interface adaptive to the screen size of the device.

One of the main design decisions was to use a 'Tab Bar' (UI component provided by the Cocoa Touch framework). This component is used for the main navigation of the application. It's defined by Apple in the following way:

'A tab bar gives people the ability to switch between different subtasks, views, or modes in an app. Use a tab bar to organize information at the app level. A tab bar is well suited for use in the main app view because it's a good way to flatten your information hierarchy and provide access to several peer information categories or modes at one time.'

The following is an illustration of the tab bar used. Each item lets the user access another navigation flow.



Figure 19. Tab bar used

All of the UI components used are components provided by the Cocoa Touch framework, with a layer of customization.

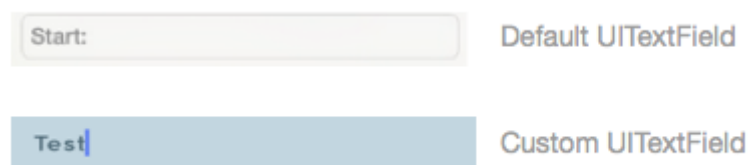


Figure 20. Example of UITextField customization

The design process of the user interface is composed of 3 steps: wireframes, digital design and implemented design. Those 3 steps were repeated for all of the screens of the UI.

The wireframes are paper sketches used for defining navigation, arrange UI elements and defining content. The digital design was performed in Sketch (design tool similar to Photoshop) and its purpose is to create an interface that is consistent through the application, without actually spending time implementing it. The implementation of the design is the last step, performed by following the output of the step 2. The implemented UI is often identical with the digital design.

The following is an example of the first 3 screen of the application, from the wireframes to the digital designs.

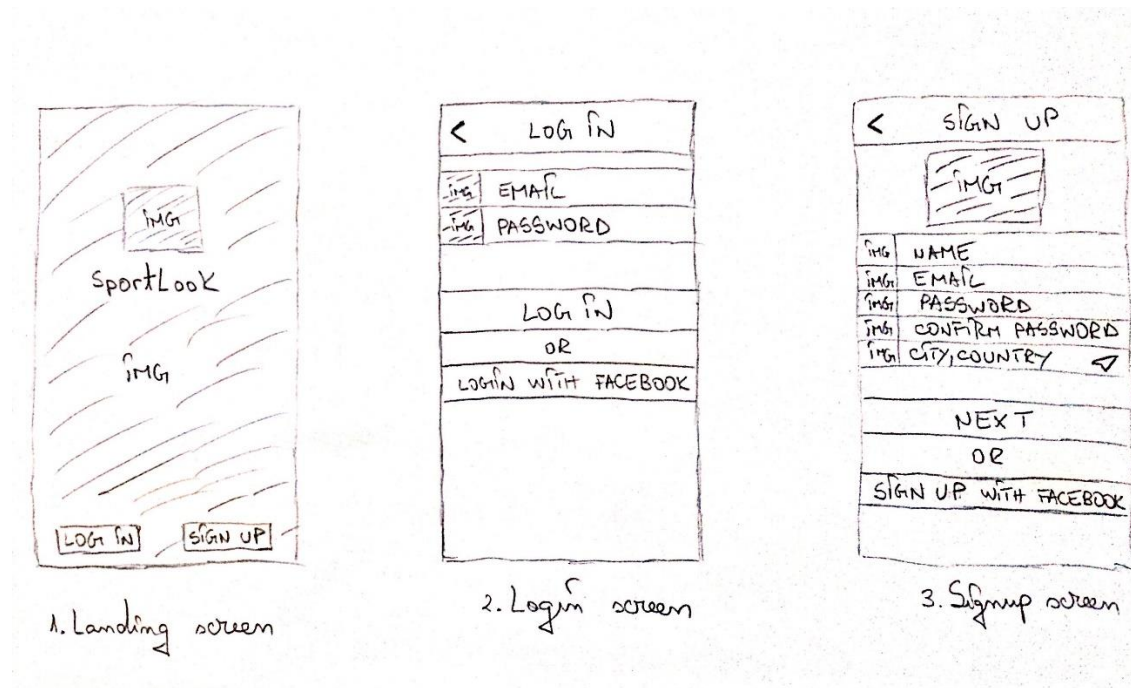


Figure 21. Wireframes UI

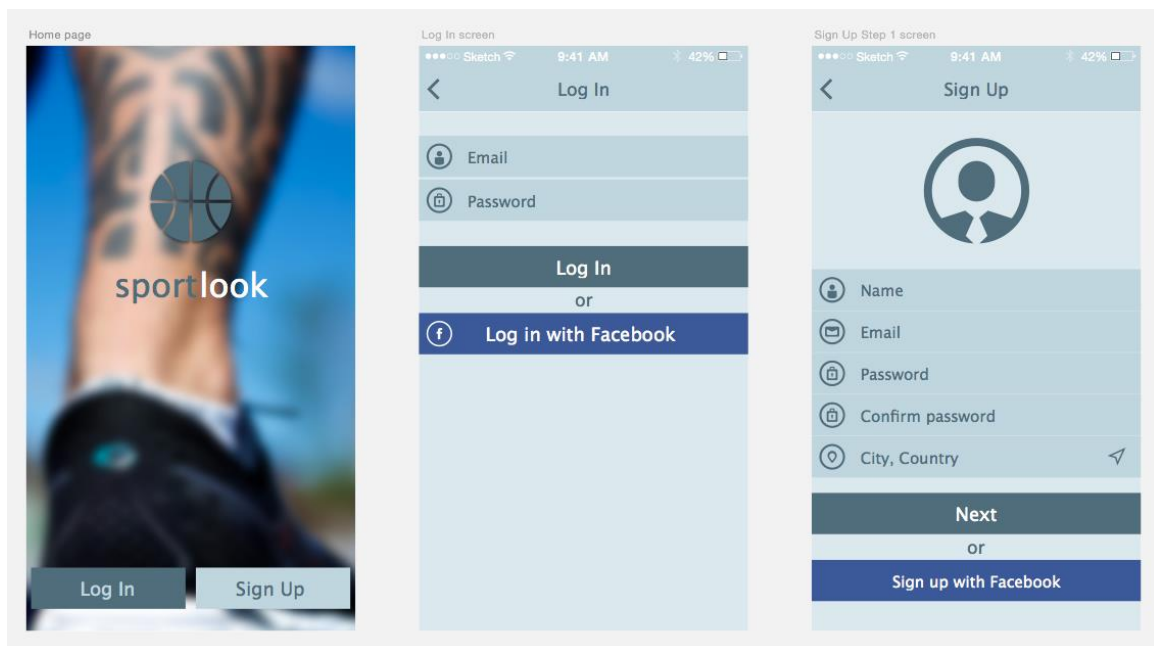


Figure 22. Digital design UI

Storyboards, views and code

A very important decision at the beginning of implementing an iOS application is how the user interface should be implemented. There are 3 ways of achieving this task:

- **By using Storyboards.** A storyboard is a visual tool for laying out multiple application views and the transitions between them. This is the newest way to implement an UI, but

unfortunately it's not mature enough to be used in complex projects. Storyboards were created to help developers visualize the views and flow of an application.

- **By using .xib files.** Each .xib file corresponds to a single view element (that can have subviews) and can be laid out in the Interface Builder, making it a visual tool.
- **In code.** Without using any GUI tool, handling all custom positioning, animations and constraints programmatically.

Each approach has its pros and cons, but in this project the interface is built by using .xib files due to several reasons:

- Storyboards make working in a team harder. Even if source control software is used, 2 developers cannot work on the storyboard at the same time, since there will be almost unsolvable conflicts every time this happens.
- Storyboards hinder code reuse. Setting values in a storyboard has to happen in all needed elements, therefore duplicating the setup. In case of change needed, all the occurrences need to be modified.
- Reusable views cannot be used in storyboards. If there is one UI component that needs to be used more than once, then this component needs to be created and customized each time where it is used.

Building the UI in code would substantially increase the project time, since there would be no visual interface of the application, unless it's build and run.

3.3. Web Service Design

3.3.1. Messaging protocol

The application requires sharing of data between the server and the client application. Additionally the server has to support both iOS and Android. The communication between the server and the client application has to be agnostic. The standard way to accomplish that is to create SOAP or RESTful API.

SOAP (Simple object access protocol) is XML based protocol consisting of three parts:

- an envelope, which defines the message structure and how to process it
- a set of encoding rules for expressing instances of application-defined data types
- a convention for representing procedure calls and responses

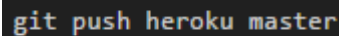
REST (Representational state transfer) Web Services are defined with the following aspects:

- base URI, such as <http://ourapp.com/api/>
- An Internet media type for the data exchanged. This is usually JSON
- standard HTTP methods (GET, PUT, POST, or DELETE)

It was decided to go with REST because it is more lightweight, more suited for the basic create, read, update and delete (CRUD) use case that is necessary for this Web Service.

3.3.2. Deployment

To deploy the Web Service on a server with public URL it was decided to use Heroku. Heroku is a platform as a service that supports code written in Node.js among other languages. It allows to deploy the Web Service easily by simply pushing the source code through the chosen version control software (Git):

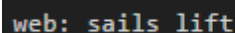


```
git push heroku master
```

Figure 23. Deploying on Heroku

Heroku then locates the dependency file *package.json* which is part of every Node.js application, installs the dependencies using Node Package Manager (npm) and builds the binary. The only separate instruction we need to provide to Heroku is file named *Procfile* which specifies how to run our application.

This is how the *Procfile* looks. It specifies to run the command "sails lift" to start a web process.



```
web: sails lift
```

Figure 24. Starting a web process

Heroku thus handles compiling and building the source code, installing the dependencies, choosing the hardware to run it on, starting the application, exposing it publicly and restarts it if it crashes. It also makes it easy to scale the application up in the case it starts to get too many requests or using too many resources. All the developers need to focus on is the application logic.

The downside is that as the application grows it will be required to pay a monthly fee for using Heroku. It was consulted this with the client and was deemed acceptable, opting to focus developers' time on application logic not deployment infrastructure.

Another downside is that Heroku does not allow storing files directly on their file system, which are not part of the git repository. The Web Service however required an image uploading functionality. It was decided to use a third party service for that (more information available in the 3.4 section).

3.3.3. Framework

The Web Service is implemented in the Sails.js framework running on top of Node.js platform. Node.js was chosen for the following reasons:

- Fast performance and data processing.
- Relatively easy to set up load balancing, clusters and scale up if the need arises.
- Open source and a healthy ecosystem of open source packages.
- Developers were already familiar with it.

Sails.js makes it easy to build custom, enterprise-grade Node.js apps. It is designed to emulate the familiar MVC pattern of frameworks like Ruby on Rails, but with support for the requirements of modern apps: data-driven APIs with a scalable, service-oriented architecture.

The decision to use Sails.js was made to abstract away most of the boiler-plate code that would otherwise be required. It also has a very steep learning curve.

3.3.4. Database

Sails.js allows developers to utilize powerful data access layer with Object Relation Mapping (ORM) called Waterline which works with multiple databases. To use Waterline properly we define model entities together with relationships and database-adapter for it to use such as in Figure 25. Waterline constructs the underlying table/document structure from the model in case it doesn't exist yet.

```
attributes: {  
  name: {  
    type: 'string'  
  },  
  image: {  
    type: 'string'  
  },  
  startTime: {  
    type: 'datetime'  
  },  
  endTime: {  
    type: 'datetime'  
  },  
  address: {  
    type: 'string'  
  },  
  latitude: {  
    type: 'float'  
  },  
  longitude: {  
    type: 'float'  
  },  
  description: {  
    type: 'text'  
  },  
  category: {  
    model: 'sport'  
  },  
  createdBy: {  
    model: 'user'  
  },  
  attending: {  
    collection: 'user',  
    via: 'eventsAttending',  
  },  
}
```

Figure 25. Code snippet for Event.js

```

attributes: {
  name: {
    type: 'string',
    required: true
  },
  eventsCreated: {
    collection: 'Event',
    via: 'createdBy'
  },
  eventsAttending: {
    collection: 'Event',
    via: 'attending',
    dominant: true
  },
  email: {
    type: 'email',
    unique: true,
    required: true
  },
  image: {
    type: 'string',
    defaultsTo: "/image/1e6ef38c-dbb4-4c20-abd6-61f724b5c4d6.jpg"
  },
  score: {
    type: 'integer',
    defaultsTo: 0
  },
  address: {
    type: 'string',
    required: true
  },
  lattitude: {
    type: 'float'
  },
  longtitude: {
    type: 'float'
  },
},

```

Figure 26.Code snippet for User.js

The ER diagram for the database is illustrated as follows:

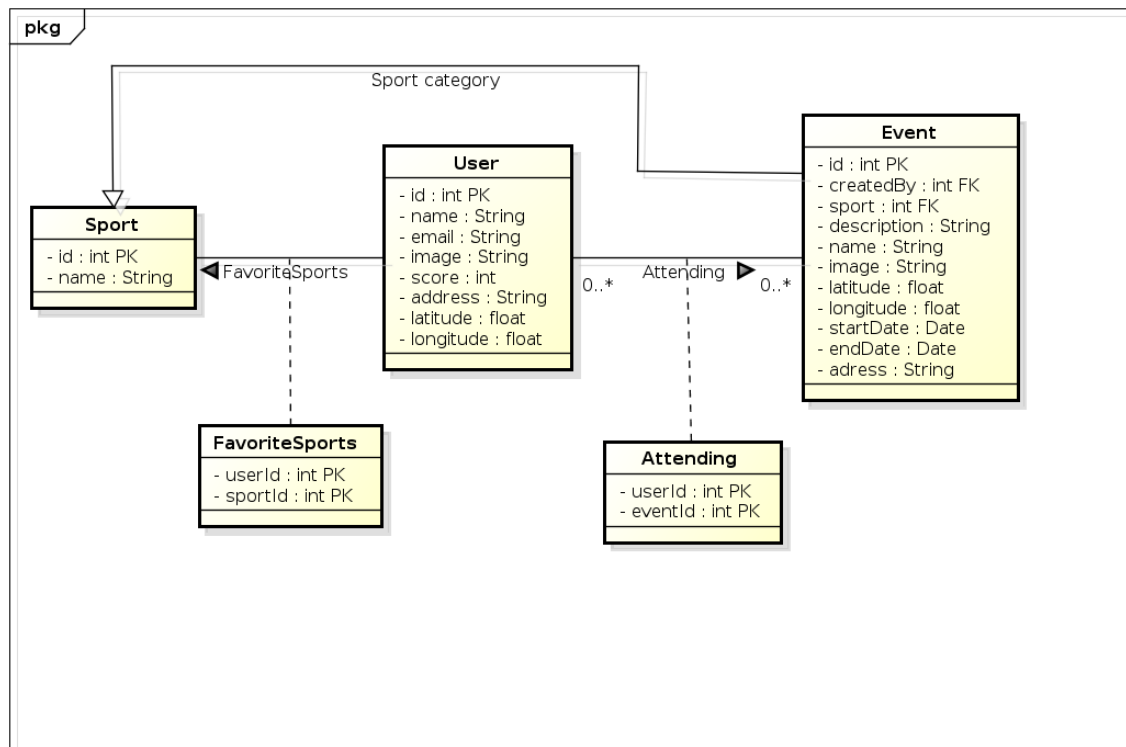


Figure 27. ER diagram

Waterline allows to specify which database to use simply by specifying a database adapter. The ORM interface stays the same regardless of the chosen database.

The biggest choice concerning database choice was whether to use a traditional relational database such as MySQL or PostgreSQL or an object database (NoSQL) such as MongoDB. After analysing the data model it was decided to go with a relational database. Some other important requirements for the database were:

- It is easy to set up with Node.js.
- It is open source and free to use.
- It has good community behind it to help solving any occurring issues.
- It is supported by Heroku.

Both MySQL and PostgreSQL were seriously considered. Both of them fulfilled requirements 1-3 but in the end PostgreSQL was chosen because of it is supported by Heroku.

3.4. Third Party Services

As any software developer knows, it is not worth the time to reinvent the wheel. If there is an acceptable out of the box solution for a problem then it should be at least seriously considered. Two such problems were encountered: the need for push notifications and image uploading.

Push notifications - Parse

The system has to be able to send push notifications to the mobile applications in the event a user is invited to an event or receives a chat message. There are many services that provide this functionality so there is no need to implement it all over again.

Parse was chosen because it has Node.JS, iOS and Android APIs as well as offering a freemium model. It also allows to store chat messages directly in Parse eliminating the need to store chat messages in the Web Service database. The decision to go with Parse was also business decision as the pricing and freemium model were suitable for the client.

Image uploading - Amazon S3

As mentioned earlier Heroku does not allow to write to the file system directly as it flushes the file system with each new build.

Enter Amazon Simple Storage Service (S3). Instead of storing images in the Heroku file system we store them in the Amazon S3. This is hidden from the client and image uploads are simply piped to S3 and downloads are piped from S3 to the client's response socket. From the client side there is no difference.

4. Implementation

4.1. iOS Application Implementation

The implementation of the application has been made in Xcode (Apple's IDE for developing MacOS / iOS applications). This section will not cover the entire implementation of the app, but the parts that would require specialized knowledge. It has been a priority to keep the code clean and easy to read, modularized, reusable and above all efficient.

4.1.1. BaseViewController & Supporting Different OS Versions

One of the non-functional requirements was to support iOS 7+.

In order to not rewrite the same code over and over, it was devised that every View Controller that was used in the app should inherit from `BaseViewController`. This class contains methods for OS version handling.

An example is the loading screen that looks different in iOS 7 & iOS 8. The component used for the loading screen uses a blur effect that was introduced with iOS 8. The `BaseViewController` would have a method called `showLoading (message: delay :)` and then in the View Controllers the method could simply be called.

```
class BaseViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()

        self.edgesForExtendedLayout = UIRectEdge.None
        self.navigationController?.navigationBar.translucent = false
    }

    func showLoading(message: String, delay: Double) {
        if SYSTEM_VERSION_GREATER_THAN_OR_EQUAL_TO("8.0"){
            Utils.performActionWithDelay(seconds: delay) { () -> () in
                SwiftSpinner.show(message, animated: true)
            }
        }
        else {
            MRProgressOverlayView.showOverlayAddedTo(self.navigationController?.view, title: message, mode:
                MRProgressOverlayViewMode.Indeterminate, animated: true, stopBlock: nil)
        }
    }

    func hideSpinner(delay: Double) {
        if SYSTEM_VERSION_GREATER_THAN_OR_EQUAL_TO("8.0"){
            Utils.performActionWithDelay(seconds: delay) { () -> () in
                SwiftSpinner.hide()
            }
        }
        else {
            MRProgressOverlayView.dismissAllOverlaysForView(self.navigationController?.view, animated: true)
        }
    }
}
```

Figure 28. BaseViewController implementation

Below is an example of how this is used in ProfileViewController. The `self.showSpinner` method is called if the user is not loaded, it only takes a single line to show or hide the loading.

```

if !userLoaded || Static.userChanged || Constants.Load.shouldLoad("profile") {
    self.showLoading("Loading Profile", delay: 0)

    ProfileProvider.getProfileWithToken(KeychainHandler.getUserLoggedToken()!, success: { (user) -> Void in
        self.user = user,
        self.updateUser()
        self.userLoaded = true
        Static.userChanged = false

        self.hideSpinner(0)
    }) { (error) -> Void in
        println(error)
        println("Error Loading Profile")

        self.hideSpinner(0) //Remove loading screen
        let alert = SCLAlertView()
        alert.showError("Oops!", subTitle: "Error loading Profile. Try again later.", closeButtonTitle:
            "Ok")
    }
}

```

Figure 29. Use of BaseViewController method

4.1.2. Supporting different iPhone Screen sizes

In September 2014, Apple announced iPhone 6 & iPhone 6 Plus. For developers this meant that suddenly there was twice the amount of screen sizes the apps need to support. But at the same time they introduced *Size Classes*.

Size Classes are Apple's solution to the question "How can I easily work with so many different screen sizes and device orientations?" They also enable the condensing of universal apps into one storyboard file. Combined with the new adaptive view controllers, it's easier than ever to rely on Interface Builder, rather than fight with it, to simplify the creation of the app layouts.

Every view controller in the app gets a trait collection object. This trait collection object has 2 size classes, a horizontal size class and a vertical size class. And each of these classes has 3 possible values: compact, regular, or any. These values can change based on the device and its orientation. The app will layout its interface for each View Controller based on the current size classes. Apple uses a grid to let the user choose which configuration to work in, so here's that same grid but with each device + orientation highlighted in its corresponding size class combo.

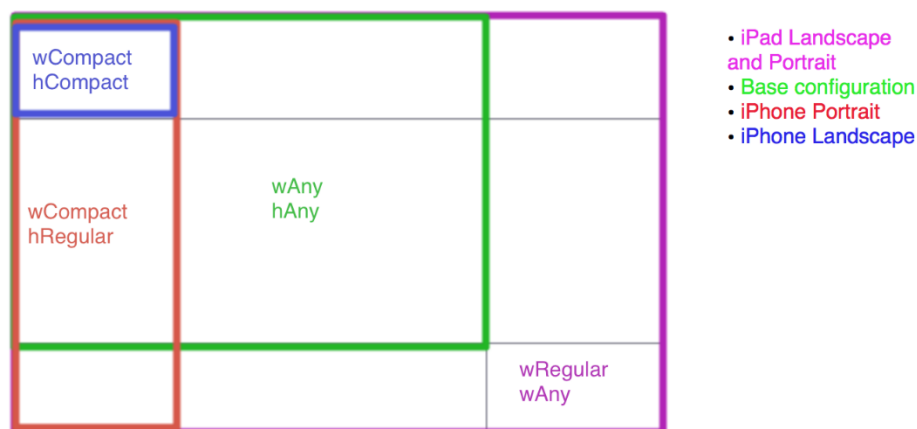


Figure 30. Size Classes overview

At the bottom of the Interface Builder window, there's now a control that allows switching between each combination.

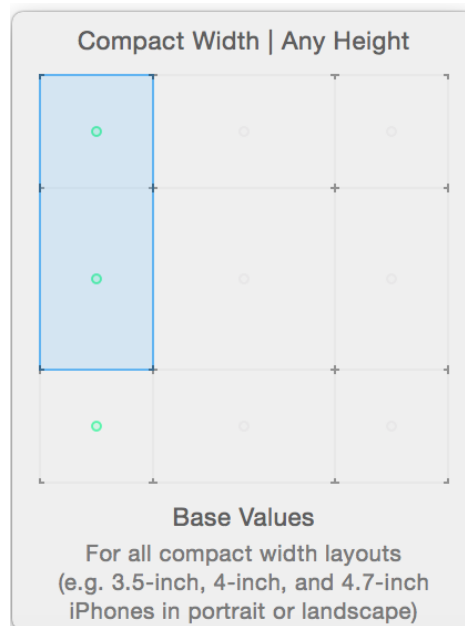


Figure 31. Size Classes example

In the project the size class Compact Width X Any Height was used. This made sure that the design made in Interface Builder could be supported with all iPhones in Portrait mode (landscape orientation will not be supported) as long as the views were made with constraints.

4.1.3. MapKit

The map was used in the "Search Events" tab, where it shows pins on locations where users have created events and in the "Select on Map" feature where one can select the location in the map. In order to work with the MapKit, it is necessary to set the delegate of the MapView to self in the View Controller where the map is used.

```
import UIKit
import MapKit

class SearchEventsViewController: UIViewController, MKMapViewDelegate {

    @IBOutlet weak var mapView: MKMapView!

    //MARK: Utilities
    override func viewDidLoad() {
        super.viewDidLoad()

        mapView.delegate = self
    }
}
```

Figure 32. MKMapView example

The following MKMapViewDelegate methods were used in "Search Events".

```

//MARK: MapView Delegate Methods
func mapView(mapView: MKMapView!, didUpdateUserLocation userLocation: MKUserLocation!) {
}

func mapView(mapView: MKMapView!, viewForAnnotation annotation: MKAnnotation!) -> MKAnnotationView!
{
    return MKAnnotationView()
}

func mapView(mapView: MKMapView!, regionDidChangeAnimated animated: Bool) {
}

func mapView(mapView: MKMapView!, regionWillChangeAnimated animated: Bool) {
}

func mapView(mapView: MKMapView!, didSelectAnnotationView view: MKAnnotationView!) {
}

```

Figure 33. MKMapViewDelegate methods

The following figure shows the example of use of ‘*mapView (: regionWillChangeAnimated :)*’.

```

func mapView(mapView: MKMapView!, regionWillChangeAnimated animated: Bool) {
    if showEventInfo {
        self.fadeViewAway(vwEventInfo)
        showEventInfo = false
    }
    if selectedAnnotation != nil {
        self.mapView.deselectAnnotation(self.selectedAnnotation, animated: true)
    }
    if !vwClusterInfo.hidden {
        self.fadeViewAway(vwClusterInfo)
    }
}

```

Figure 34. mapView regionWillChangeAnimated

This methods gets fired every time the view region of the map changes. Whether it is by the interaction of the user or when programmatically moving the region. In this case the code checks if the event info view is shown to the user. If it is, it will fade it away. This is made to provide a good UX and not keep showing event info (after you click on a pin) when the user moves the map. The ‘vwClusterInfo’ is basically the same, but is a view that gets shown instead of ‘vwEventInfo’ when there are more events at the same location (show a list of events instead of just one).

4.1.4. Reusable Views

In order to provide a good UX it is a common rule to make different views look the same. In this case the text fields for inputting information from the user were used multiple times in the project flow. A bad case would be to design all those text fields over and over again. A good practice would be to make a view component that is reusable. So in the future if there would be a need to change the design, it would be easier to change it only in one place instead of all the places the view was used.

Xcode and iOS doesn't come with an out of the box way to make custom views. One way to do it is to make the view programmatically, the other way is to create a view (.xib) file, and

associate it with an implementation file. It is then possible to add custom controls through the view's implementation file.

Below is an example of a screen where the same components were used multiple times.

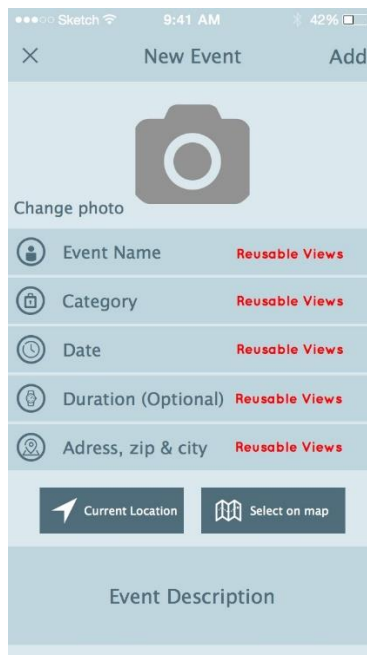


Figure 35. Create Event screen

The view consist of a text field and an image view. So when implementing this view, it had to be somehow generic.

```
class SLInputview: UIView, UITextFieldDelegate {

    @IBOutlet var view: UIView!
    @IBOutlet weak var txtField : UITextField!
    @IBOutlet weak var imgvIcon : UIImageView!

    required init(coder aDecoder: NSCoder) {
        super.init(coder: aDecoder)

        //Load Interface
        NSBundle.mainBundle().loadNibNamed("SLInputView", owner: self, options: nil)
        self.addSubview(self.view)
    }

    override func awakeFromNib() {
        super.awakeFromNib()

        self.setup()
    }

    func setup(){

        self.backgroundColor = UIColor.COLOR_TEXT_FIELD_BACKGROUND
        self.txtField.font = UIFont(name: UIFont.QUICKSAND_BOLD, size: 15)
        self.txtField.textColor = UIColor.COLOR_TEXT_FIELD_TEXT

        self.txtField.delegate = self
    }

    func setTextField(placeholder: String, keyboardType: UIKeyboardType!, image: UIImage!, returnKey:
        UIReturnKeyType!)
    {
        self.txtField.placeholder = placeholder

        if keyboardType != nil {
            self.txtField.keyboardType = keyboardType
        }

        if returnKey != nil {
            self.txtField.returnKeyType = returnKey
        }

        self.imgvIcon.image = image
    }
}
```

Figure 36. SLInputView implementation

The *'init'* method loads a view from a .xib file named *SLInputView*. Afterwards, the method *'awakeFromNib ()'* is called, which setups the colours used in the view.

In the Interface Builder (IB) it is now possible to draw a UIView and set its class to *'SLInputView'*. When creating this instance of the view in the implementation (.swift) file, the method *'setTextField (placeholder: keyboardType: image: returnKey :)'* can be called and therefore customize the view according to the placeholder, image keyboard type etc.

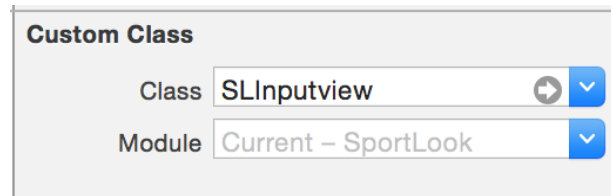


Figure 37. Setting a custom class in the Interface Builder

Now it is easy to setup the image view and placeholder for each of the custom input fields. In the *'viewDidLoad ()'* method the *'setLayout ()'* function is called.

```
func setupLayout()
{
    slEventName.setTextField("Event Name", keyboardType: UIKeyboardType.Default, image: UIImage(named:
        "ic_name"), returnKey: UIReturnKeyType.Next)

    slEventCategory.setTextField("Category", keyboardType: nil, image: UIImage(named: "ic_name"), returnKey:
        UIReturnKeyType.Next)

    slEventDate.setTextField("Date", keyboardType: nil, image: UIImage(named: "ic_date"), returnKey:
        UIReturnKeyType.Next)

    slEventDuration.setTextField("Duration (Optional)", keyboardType: UIKeyboardType.Default, image: UIImage
        (named: "ic_date"), returnKey: UIReturnKeyType.Next)

    slEventAddress.setTextField("Address, city, zip", keyboardType: nil, image: UIImage(named: "ic_location"),
        returnKey: UIReturnKeyType.Done)
}
```

Figure 38. Using the *setTextField ()* method

4.1.5. Open Source Components

The following is a complete list of open source components used for speeding up the development of the iOS application:

- **Pop.** Pop is an extensible animation engine for iOS and OS X. In addition to basic static animations, it supports spring and decay dynamic animations, making it useful for building realistic, physics-based interactions.
- **IQKeyboardManager.** Prevents the situation when a UITextField would not be visible to the keyboard overlaying it.
- **MRProgress.** The loading view used for iOS 7.
- **SwiftSpinner.** The loading view used for iOS 8.
- **KingPin.** Clustering engine for MKAnnotation in MapKit. When zooming out in the map view kingpin clusters the pins. By overwriting the delegate methods of kingpin it was possible to show different images for the pins depending on how many pins the cluster had included.

- **HUMSlider**. The slider used in the discovery page when the user selects the radius of discover. Provides a nice animation when the slider is used.
- **SDWebImage**. Loads images from URL links asynchronously.
- **JSQMessagesViewController**. A View Controller which has predefined UI elements suitable for building a chat user interface.
- **Facebook SDK**. Used for the implementation of the features “Login with Facebook” and “Sign up with Facebook”.
- **SwiftJSON**. Makes it easy to deal with JSON data in Swift. It becomes really handy when parsing JSON data to Swift objects.
- **KeychainAccess**. It’s a simple Swift wrapper for Keychain that works on iOS and MacOS. It makes using the Keychain API very straight-forward in Swift. Used for storing sensitive user information.
- **Alamofire**. HTTP(S) networking library written in Swift.
- **SCLAlertView**. Animated Alert View written in Swift. Used instead of the default UIAlertView.
- **Net**. HTTP request wrapper written in Swift. Used to upload images to the Web Service.

4.1.6. Chat implementation

The UI implementation for the Chat feature can be seen in the following illustration:

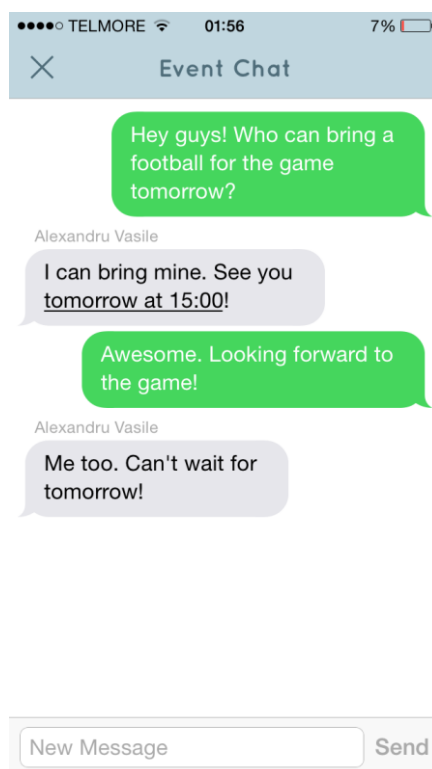


Figure 39. Chat UI

For this feature, 2 components were used: an open source UI library called '*JSQMessagesViewController*' and Parse (BaaS) for storing the chat messages and sending push notifications.

JSQMessagesViewController provides already customized UI components that can be combined together for building the interface. Further customization is also possible, the library offering flexibility. The base of JSQMessagesViewController is a 'UIViewController' (class provided by the Cocoa Touch framework) that manages a *UICollectionView* (also provided by the Cocoa Touch framework). Therefore, *UICollectionViewDataSource* protocol methods are used to customize the layout in addition to JSQMessagesViewController protocol methods.

The following is the basic layout customization, which takes place after a Chat controller is visible on the device screen:

```
func setupChat(){
    setupChatLayout()
    //Mandatory setting senderId and displayName
    self.senderId = KeychainHandler.getUserLoggedId()
    self.senderDisplayName = KeychainHandler.getUserLoggedName()
}
func setupChatLayout(){
    //No accessory left button, next to the 'Send' button
    self.inputToolBar.contentView.leftBarButtonItem = nil
    //Prevent the avatars from taking size
    self.collectionView.collectionViewLayout.incomingAvatarViewSize = CGSizeZero
    self.collectionView.collectionViewLayout.outgoingAvatarViewSize = CGSizeZero
    //Create outgoing and incoming message bubbles
    let bubbleFactory = JSQMessagesBubbleImageFactory()
    outgoingBubbleImageData = bubbleFactory.outgoingMessagesBubbleImageWithColor(UIColor.jsq_messageBubbleGreenColor())
    incomingBubbleImageData = bubbleFactory.incomingMessagesBubbleImageWithColor(UIColor.jsq_messageBubbleLightGrayColor())
}
```

Figure 40. Initial Chat UI setup

The second part is the implementation of the chat logic and communication between multiple iOS apps. Parse (BaaS) is used for this purpose, with its iOS SDK. Unfortunately, the chat is not real-time, but a simulation of this is achieved by using a timer, which loads the messages for a specific event chat every 5 seconds. This is performed only when a Chat controller is visible on the screen.

```
override func viewWillAppear(animated: Bool) {
    super.viewWillAppear(animated)
    timer = NSTimer()
    timer = NSTimer.scheduledTimerWithTimeInterval(5.0, target: self, selector: "loadMessages", userInfo: nil, repeats: true)
}
func loadMessages() {
    if self.isLoading == false {
        self.isLoading = true
        var lastMessage = messages.last
        var query = PFQuery(className: Constants.Parse.PF_CHAT_CLASS_NAME)
        query.whereKey(Constants.Parse.PF_CHAT_EVENT_ID, equalTo: eventId!)
        if lastMessage != nil {
            query.whereKey(Constants.Parse.PF_CHAT_CREATED_AT, greaterThan: lastMessage!.date!)
        }
        query.orderByDescending(Constants.Parse.PF_CHAT_CREATED_AT)
        query.limit = 50
        query.findObjectsInBackgroundWithBlock({ (objects, error) -> Void in
            if error == nil {
                self.automaticallyScrollsToMostRecentMessage = false
                for object in (objects as! [PFObject]).reverse() {
                    self.addMessage(object)
                }
                if objects!.count > 0 {
                    self.finishReceivingMessageAnimated(true)
                    self.scrollToBottomAnimated(false)
                }
                self.automaticallyScrollsToMostRecentMessage = true
            } else {
                let alert = UIAlertController()
                alert.showError("Oops!", subTitle: "Couldn't load messages! Please check your internet connection and try again!")
            }
            self.isLoading = false
        })
    }
}
```

Figure 41. Loading chat messages

In order to provide a good user experience, push notifications are used for the purpose of informing the users that are attending an event that there are new messages in the event chat. The Parse iOS SDK is used for sending the push notification to all other iOS apps, except the sending one. The receiving iOS apps can open the push notification, which triggers handling that opens the Chat controller for the specific event corresponding to the push notification received.

```
func sendPushNotification(groupId: String, text: String) {
    //Sending to everyone except myself
    let installationQuery = PFInstallation.query()
    let currentUserId = KeychainHandler.getUserLoggedId()
    installationQuery!.whereKey(Constants.Parse.PF_INSTALLATION_CURRENT_USER_ID, notEqualTo: currentUserId!)
    //Who is subscribed to the event channel
    let eventId = Constants.Parse.CHANNEL_EVENT + self.eventId!
    installationQuery!.whereKey(Constants.Parse.CHANNELS, containedIn: [eventId])
    var push = PFPush()
    push.setQuery(installationQuery)
    //Message format: User_name to event_name : Message
    var completeMessage = self.senderDisplayName + " to " + self.eventName! + " : " + text
    let data : [NSObject : AnyObject] = [
        "title" : "New chat message received!",
        "alert" : completeMessage,
        "badge" : "Increment",
        "eventId" : self.eventId!,
        "isForInvite": 0,
        "isForChat": 1
    ]
    push.setData(data)
    push.sendPushInBackgroundWithBlock { (succeeded, error) -> Void in
        if error != nil {
            println("sendPushNotification error")
        }
    }
}
```

Figure 42. Sending a push notification

The handling of opening the push notification with the correct Chat screen is based on the receiving 'eventId', which is set by the push notification sender. A complete flow of sending push notifications is illustrated in the following sequence diagram:

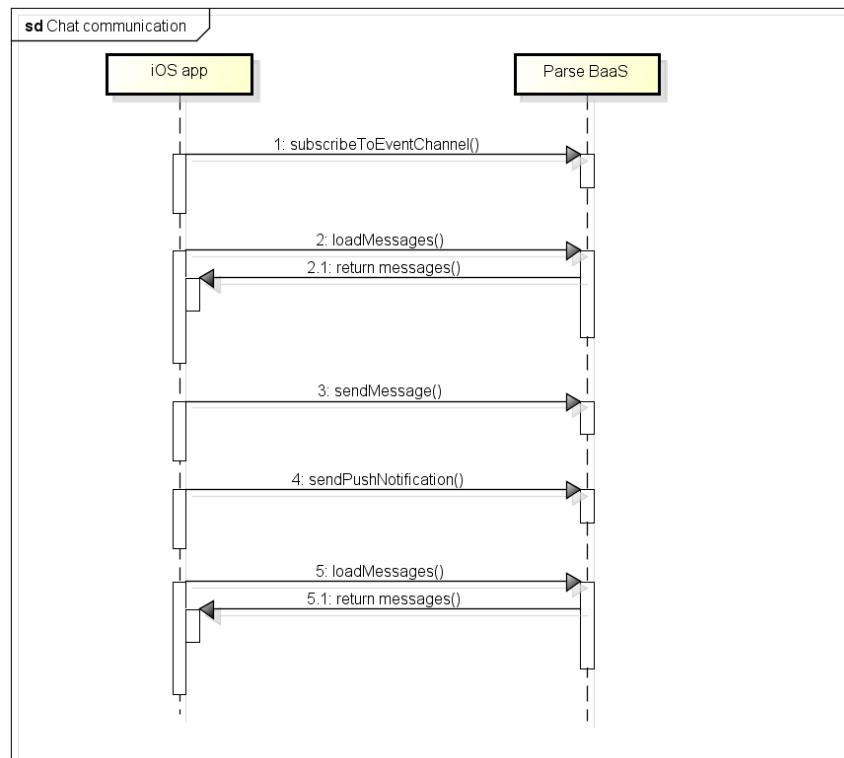


Figure 43. Chat sequence diagram

4.1.7. Push notifications implementation

The push notifications are implemented using Parse, a high level cloud solution that provides abstraction for sending push notifications to iOS applications.

The base of the concept is the Apple Push Notification Service (APNs), which Apple defines as follows:

'Apple Push Notification service transports and routes a remote notification from a given provider to a given device. A notification is a short message consisting of two major pieces of data: the device token and the payload. The device token is analogous to a phone number; it contains information that enables APNs to locate the device on which the client app is installed. APNs also uses it to authenticate the routing of a notification. The payload is a JSON-defined property list that specifies how the user of an app on a device is to be alerted.'

In the following diagram, Parse is the provider, providing an out of the box setup.

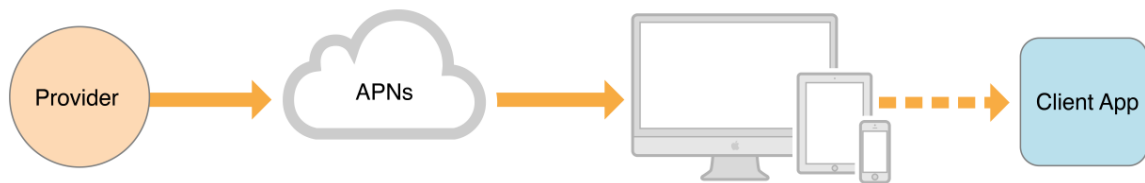


Figure 44. APNs structure

Push notifications are used in 2 situations: for sending information to users attending an event that a new message was sent to the event chat and for sending an invitation to an event from a user to another user.

The event chat notification is sent from an iOS device to other iOS devices, by using the Parse iOS SDK as an intermediate layer, which handles all the low level communication.

The user invitation notification is sent from an iOS device to another iOS device, by using both the Web Service and the Parse layers. A HTTPS request is made to the Web Service, which then uses the Parse JavaScript SDK for sending the notification to the corresponding iOS device. This flow can be observed in the following sequence diagram:

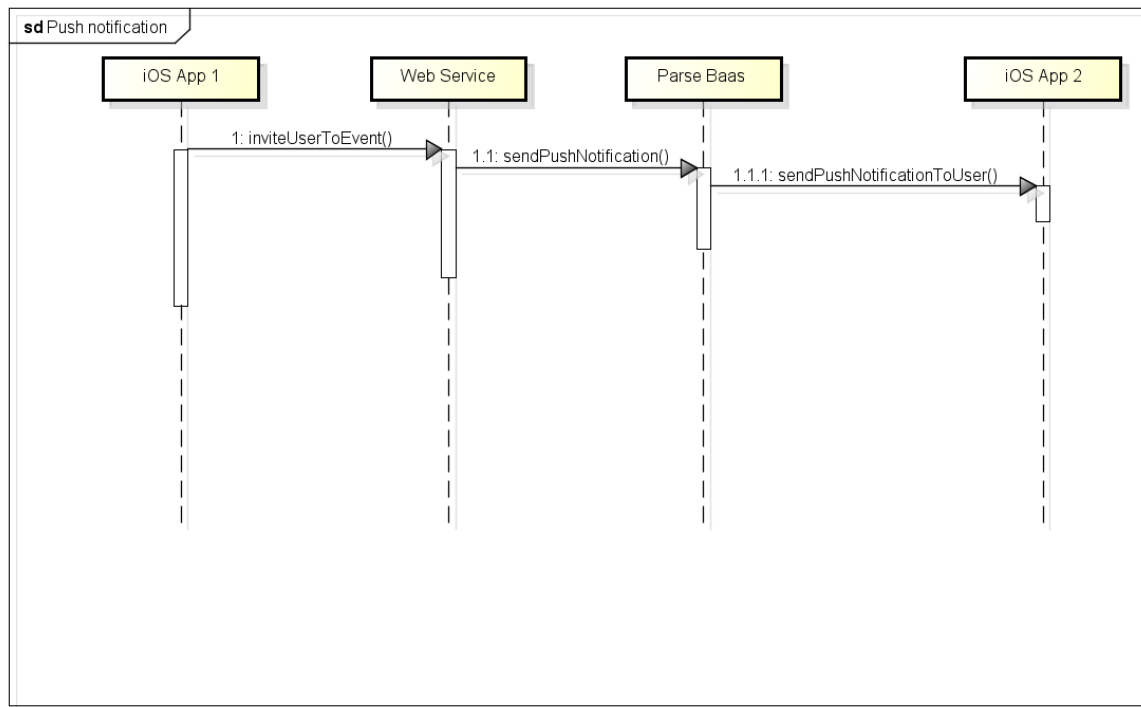


Figure 45. Push notification sequence diagram

The opening of the push notification is handled in the iOS devices in 3 application states: application is running in background, application is running in foreground, application is not running.

The following handles the state where the app is not running and the user taps a received push notification:

```

if let notification = launchOptions?[UIApplicationLaunchOptionsRemoteNotificationKey] as? NSDictionary{
    //Received a push notification
    println("Push notification content:\n\(notification)\n")
    if isNotificationAnInvite(notification as [NSObject : AnyObject]){
        //Show Event Info
        if let eventId: String = notification["eventId"] as? String{
            showEventInfo(eventId)
        } else{
            //This means an 'eventId' was not sent
        }
    } else {
        //Show Chat
        if let eventId: String = notification["eventId"] as? String{
            StructEventInfo.shouldPresentChat = true
            showEventInfo(eventId)
        }
    }
} else{
    //No push notification received
    handleFirstScreen()
}
  
```

Figure 46. Handling a received push notification

4.2. Web Service Implementation

As mentioned in Section 4.3 the Web Service is implemented in the Sails.js framework. It follows the Model View Controller design pattern.

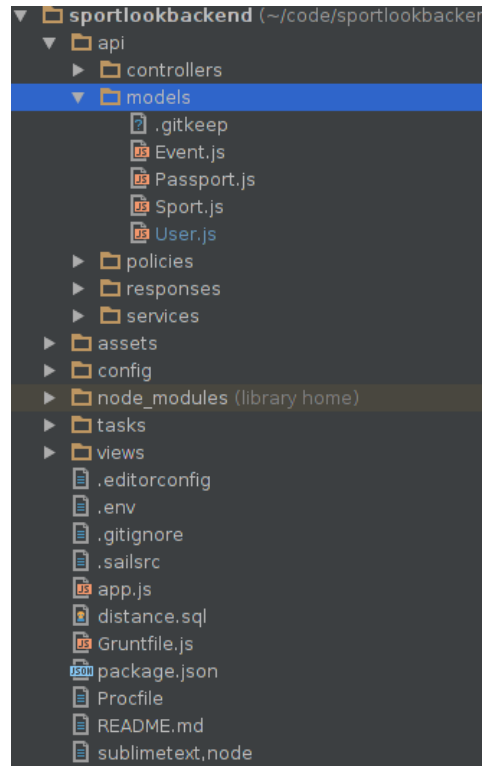


Figure 47. Web Service structure

As shown in Domain Model, each model is added in the project solution under the Model folder and default CRUD functions are added into separate Controllers under the Controller folder e.g. *User.js* model will have corresponding *UserController.js* which will provide the default CRUD actions and any custom action that is required. The URL mapping from requests to controller actions is part of the Sails.js framework.

A HTTP GET request to *https://<api_url>/event/1* will be automatically bound to the function *findOne ()* in *EventController* with the request parameter *id* set to 1.

A HTTP request to *https://<api_url>/AppleController/juiced* will execute the function *juiced* in *AppleController*. Custom mapping in the configuration file *config/routes.js* can also be added to map requests like *https://<api_url>/register* to the function *register* in *Authcontroller*.

4.2.1. User Authentication

As it can be seen from the Use Case diagram, the user has to be logged in to perform any action. The only exceptions are login and registering. Furthermore there are two ways for both. The user can either register with a user name and a password or his Facebook account. The same holds true for logging in.

For checking the user's state (logged in/logged out) on a stateless protocol like HTTP we utilize the concept of a web Token. When a user is registered a token is generated, which simply a string is guaranteed to be unique. This token is saved to the database together with the rest of the user information. It is also returned to the client. It is a responsibility of the client to maintain the token and include it with the header or as an extra parameter in any authenticated request.

For every request that a user has to be logged in, the user is fetched from the database using the token as an identifier. If no user is found or if no token is provided a HTTP response with 401(Unauthorized) error code is returned. Otherwise, the request is routed to the correct function with the user variable now set. Sails.js makes it possible to abstract this behaviour by using policies. The above logic is simply defined in a policy file and controllers which are affected by this policy are specified.

The mechanics can be seen in the following diagrams.

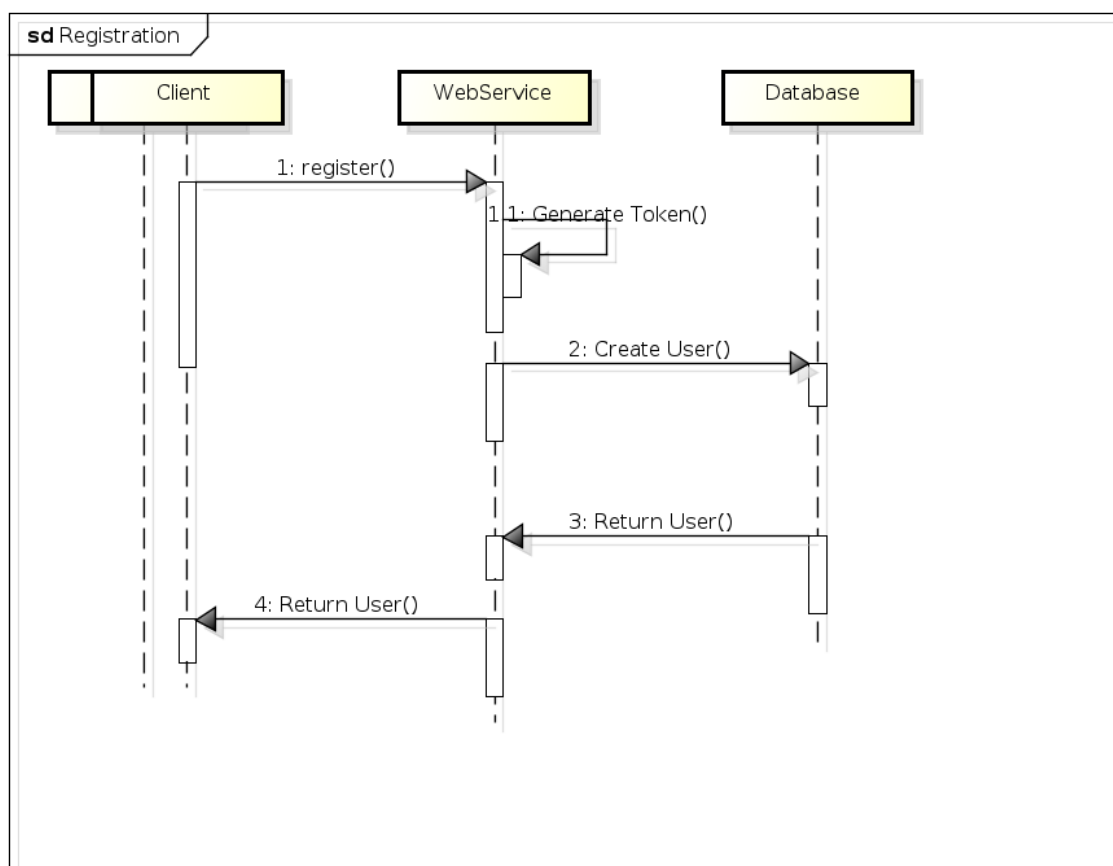


Figure 48. Registration sequence diagram

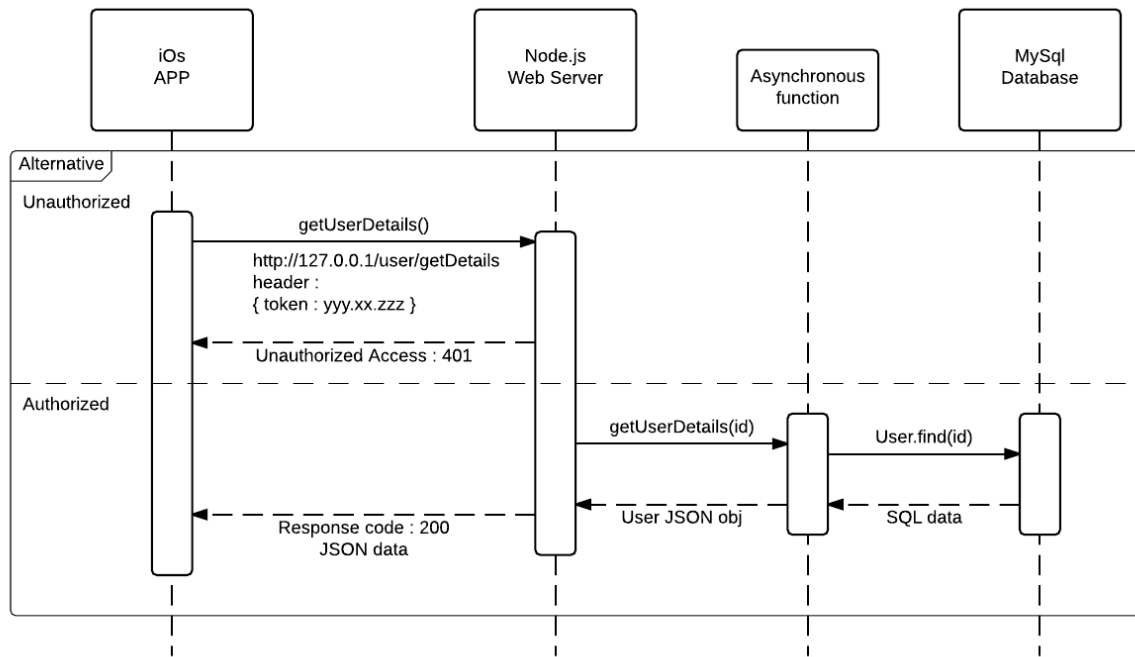


Figure 49. Authorized - Unauthorized flow

A login functionality is also provided. Although a user token is issued directly when a user is registered, thus the user is considered logged in right away, in case the user switches to a new device there has to be a way to connect to his old account. Thus the login functionality.

For logging in the client sends along the user's name and password and the token is fetched from the database and returned as long as the passwords match.

Of course the user's password is not stored in plain text. This is generally considered very insecure, if the database were to be compromised the attacker would get a list of all the user passwords that he could use to log in as them and if they reuse their password across different websites possibly gain access to them as well.

Thus the user password is hashed using the *bcrypt* algorithm before it is stored in the database. Naturally a third party library is used as it is considered wrong to implement your own cryptography algorithm.

4.2.2. Image Upload

Image uploading is usually a very simple thing on any web server. However because of the way Heroku works, it does not allow to simply store any files on their file system. The only files that can be stored are those that are included in version control, dependency packages that are automatically installed and a temporary folder that is deleted between each app restarts. It is therefore impossible to directly and persistently store uploaded files inside Heroku.

However, luckily it is very easy to integrate Amazon Web Services Simple Storage Service (AWS S3) to store the uploaded files. There are 2 use cases where a user needs to upload an image in the requirements and they are slightly different. There is a profile image uploading,

and event cover image uploading. The difference is that for a profile image a miniature version also must be generated. For an event image only the image itself is needed.

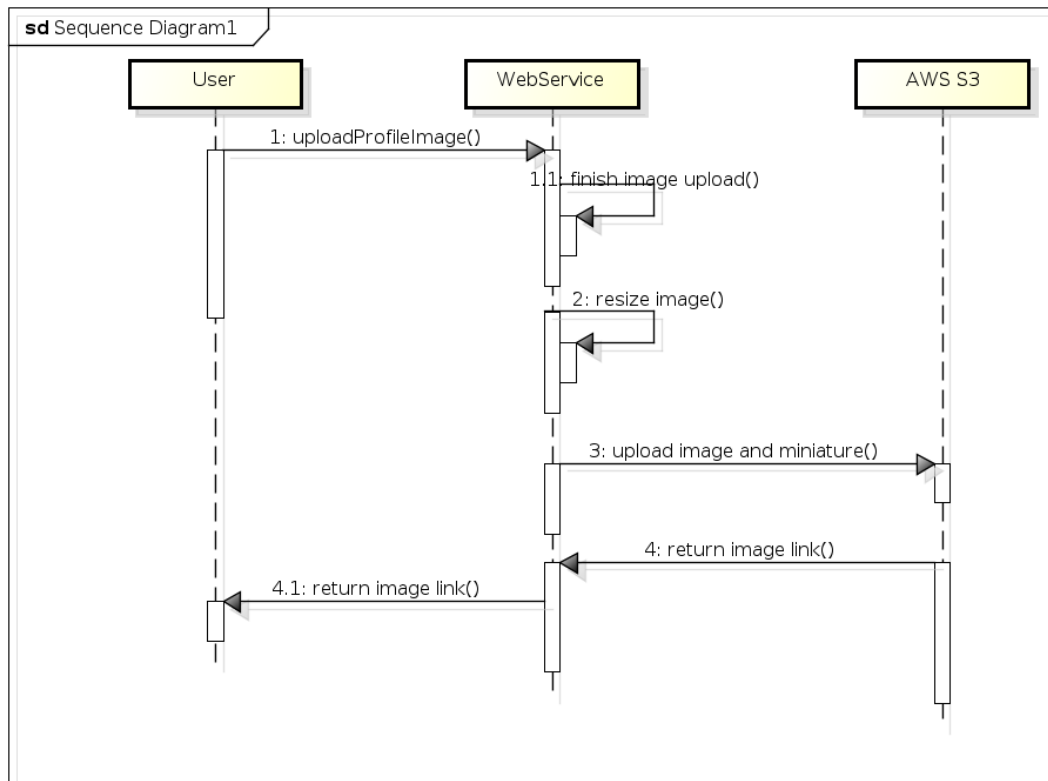


Figure 50. Sequence diagram for image upload resizing

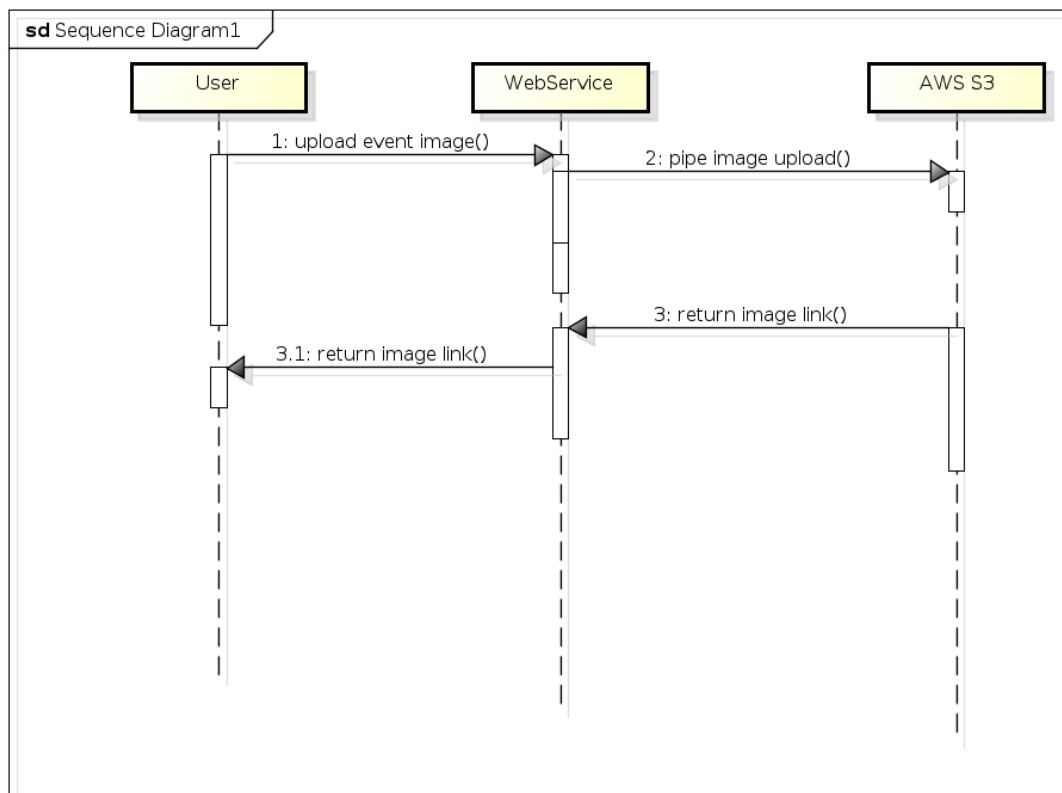


Figure 51. Sequence diagram for direct image upload

5. Testing

5.1. iOS application Testing

Black-Box testing

The main part of the testing happened as black box testing. This means that the tester is aware that a particular input returns a certain, invariable output but is not aware of how the software produces the output in the first place. In order to test the application test cases with specifications & procedures were created. An example of a test case is stated below (the rest can be found in the Appendices section).

Test Name: Test Case 1: Login test case with email
Description: The user will be able to login with an existing account.
Requirement(s): BR-03
Prerequisites: Active internet connection, user has created the account on beforehand.
Setup: The user downloaded the app, but haven't logged in.

Steps:

| Step | Operator Action | Expected Results | Observed Results | Pass/Fail |
|-------|------------------------|---|------------------------------------|-----------|
| 1. 1. | Open the app | Display the home screen | | Pass |
| 2. 2. | Click login | Display the login screen | | Pass |
| 3. 3. | Enter email & password | Show the input values in the text fields. Password is secure text entry | | Pass |
| 4. 4. | Tap login | Loading is displayed Success: User logged in show My Events screen Failure: Error message shown to the user | Successfully logged in. | Pass |

Figure 52. Test case for logging in with email

The purpose of these test cases was mostly to provide a good user experience and make sure the app worked as expected for an end-user.

White-Box Testing

During development of the app, white-box testing was used by testing the different implemented methods. An example is listed below in the *DataProvider*. As per White-Box testing, the method was called when loading the app and output can be observed in the console logs.

```

func requestData(success: (response: JSON) -> Void, failure: (error: Error) -> Void){
    Alamofire.manager.request(getHttpMethod(), DataProvider.getBaseUrl() + path, parameters: params)
        .responseJSON {(_, response, JSONObj, error) in
            var responseStatusCode: Int = -1
            if(error != nil){
                //Error from Alamofire or non-JSON response from the backend
                let error = Error(code: String(responseStatusCode), message: "Network couldn't be reached. Try again!")
                println("Error from Alamofire or not JSON response from the backend")
                failure(error: error)
            } else if(JSONObj != nil && response != nil){
                if let statusCode = response?.statusCode {
                    responseStatusCode = statusCode
                }
                let json = JSON(JSONObj!)
                println(json)
                //Backend returned an 'error' field
                if let serverError = json["error"].string {
                    let error = Error(code: String(responseStatusCode), message: serverError)
                    println("Server error: " + serverError)
                    failure(error: error)
                } else if(responseStatusCode != 200) {
                    //Backend returned status code != 200
                    let error = Error(code: String(responseStatusCode), message: "We're experiencing a technical difficulty current. Try again later!")
                    println("Either request was wrong, either there's a problem with the backend response")
                    failure(error: error)
                } else {
                    //Actually got the response we want
                    println("Success!")
                    success(response: json)
                }
            } else {
                println("Error in requestData")
                let error = Error(code: "To be set...", message: "Something wrong happened in requestData()")
                failure(error: error)
            }
        }
}

```

Figure 53. DataProvider's use of console logs

Debugging

The Xcode debugging tool was also widely used in order to test the app and make sure everything was working as it was expected during development. The code execution can be stopped with the usage of a break-point and variables can be inspected.

5.2. Web Service Testing

The full functionality of the system relies on data transmission between the clients and the Web Service. The HTTP requests have to be tested. The tests should prove that the Web Service is implemented correctly.

To test the HTTP requests a tool called POSTMAN is employed. POSTMAN is designed to create custom HTTP requests and view the HTTP responses. The testing has to be done during the Implementation phase and must be performed manually to verify that the Web Service is sending back the responses as expected.

All of the POSTMAN HTTP request names can be seen below, plus a sample request.

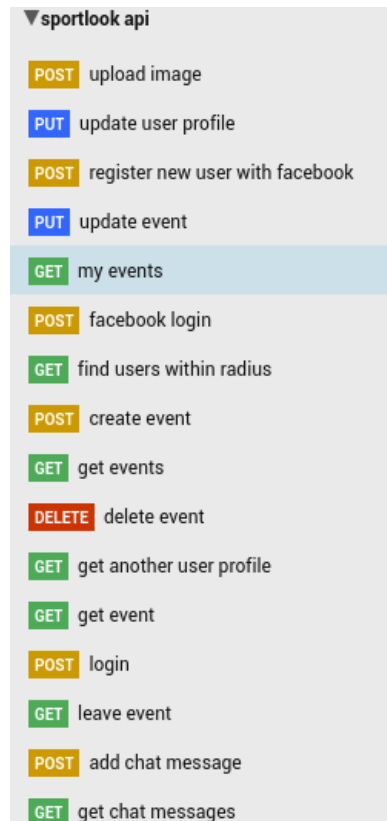


Figure 54. A collection of POSTMAN requests

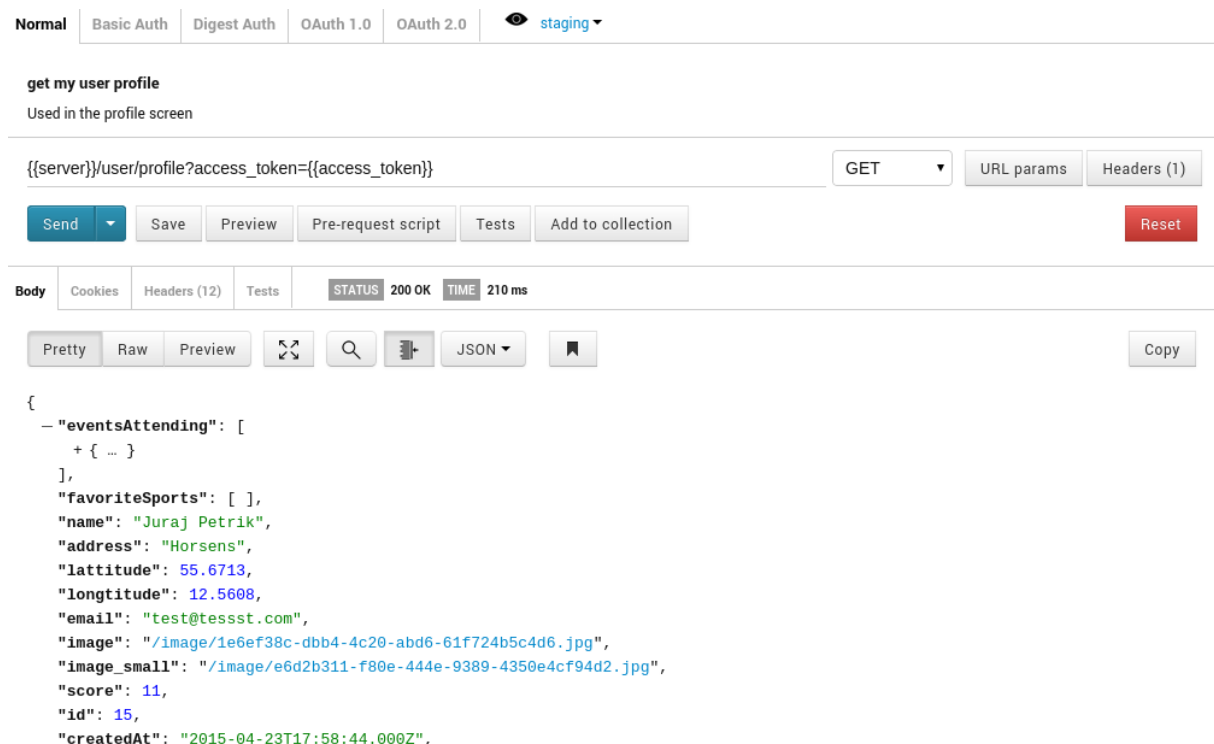


Figure 55. Testing a request with POSTMAN

6. Distribution

iOS devices users download and update their apps through Apple's AppStore. In order to publish apps in the AppStore, an enrolment in an Apple developer program is needed, which costs 99\$ / year. In this project, the client provided the development team with an existing enterprise developer account, which was used for publishing.

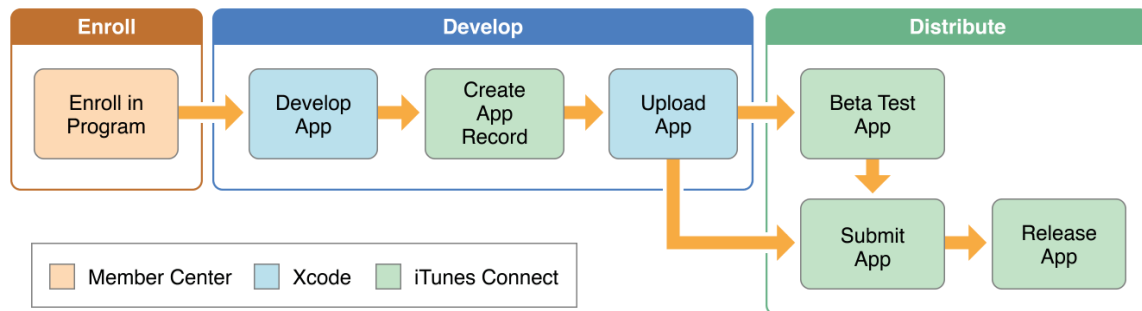


Figure 56. Apple's distribution flow

Access to Apple's Member Centre is needed. This is central point for handling app signing and creating an App ID which is required for publishing. An overview of devices, developers and apps registered to the account is available. A preview of the Member Centre can be seen below:

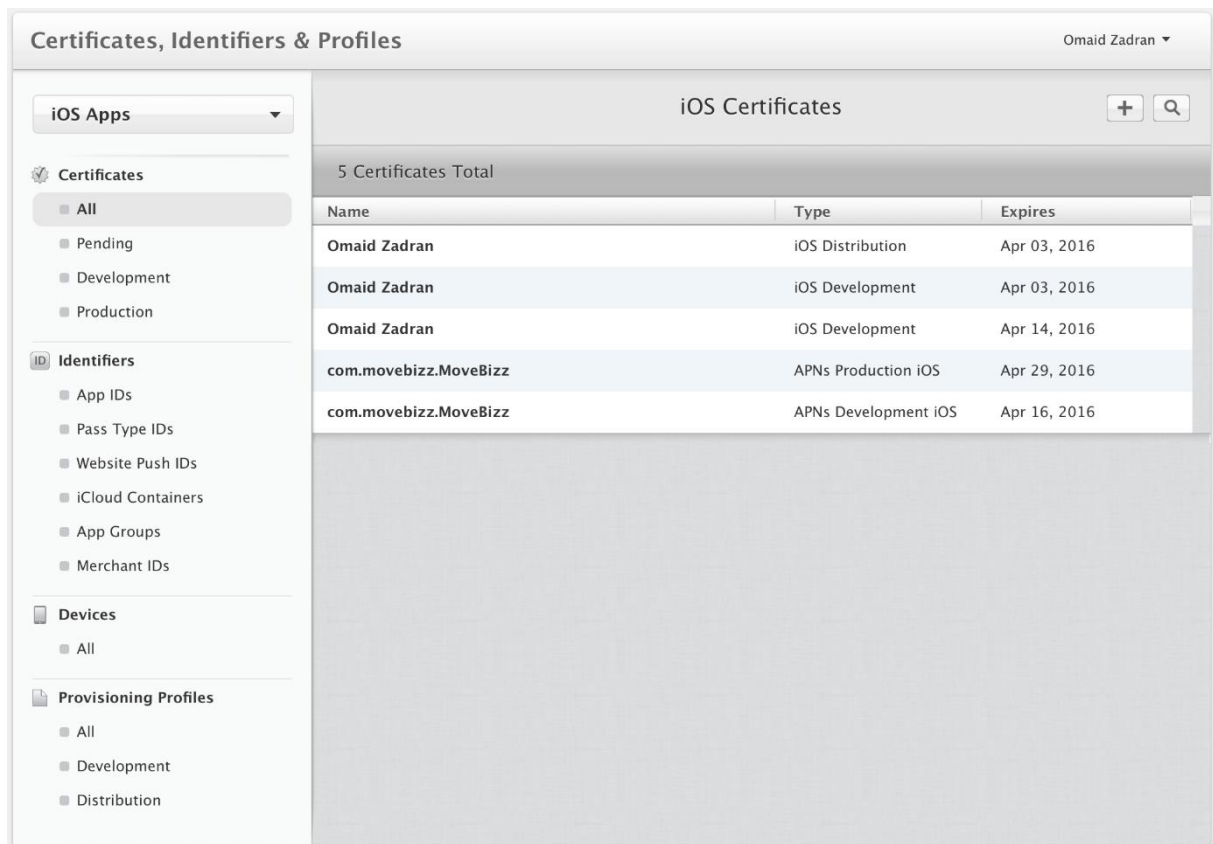


Figure 57. Apple Member Centre

6.1. Apple Member Centre, Certificates & Code Signing

Code signing is a sensitive issue in Apple's ecosystem. Unlike the Android ecosystem, it is required to perform code signing in order to run an application on a physical device. In case code signing is not used, the application will run only on the iOS simulator or a Jailbroken device, which does not allow for a wide audience.

An important issue is testing on a physical device. The simulator (Xcode comes with a built-in simulator) provides excellent initial testing, but there are 3 main reasons of why testing on a physical device is important, before deploying: a physical device is usually slower, a physical device has far less memory and there are some APIs that function correctly only on a physical device.

In Xcode versions prior to 5.0, it was necessary to enter Member Centre, register a device, create a development certificate and provisioning profile, then download that certificate, and copy it into Xcode before testing on a physical device would be possible.

The newer versions of Xcode (5.0+) made it simpler. If a project doesn't have a certificate, or the device is not added to the device list, it suggests to "fix issue" by automatically adding the device - and create a certificate if none is existing - as long as you are logged in with a valid account in Xcode.

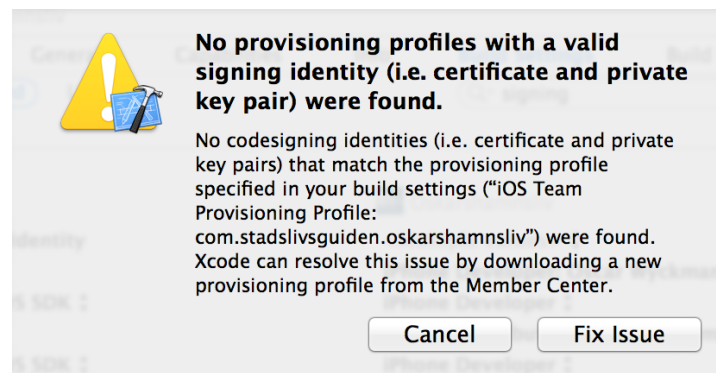


Figure 58. Example of provisioning profiles issue

It is possible to develop apps without being enrolled in a developer program, but the only way to run and test the app would be with the iOS simulator.

6.1.1. Provisioning Profiles & SSL Certificate for Push Notifications

A provisioning profile is a combination of: the unique App ID, the different certificates signed by a developer and a list of restricted devices it can be used for.

A provisioning profile accompanies the device by being copied into Xcode and from there onto the device. While in development Xcode can generate a development provisioning profile automatically - as described earlier by "fix issue" - but at the publishing stage, it is necessary to create a production provisioning profile for the app.

In order to enable push notifications in an app, it needs to be signed with a provisioning profile that is configured for push. In addition, the server needs to sign its communications to Apple's Push Notification Server (APNS) with a SSL certificate. In this project this is automatically handled by Parse by uploading the generated certificate to the Parse dashboard.

The provisioning profile and SSL certificate are closely tied together and only valid for a single App ID. This is a protection that ensures only signed servers can send push notifications to instances of an app that matches that profile.

6.2. iTunes Connect

When all the code signing and provisioning profile issues are resolved, the app is ready to be published in the AppStore. Every developer or enterprise enrolled in the developer programme gain access to an iTunes account at registration.

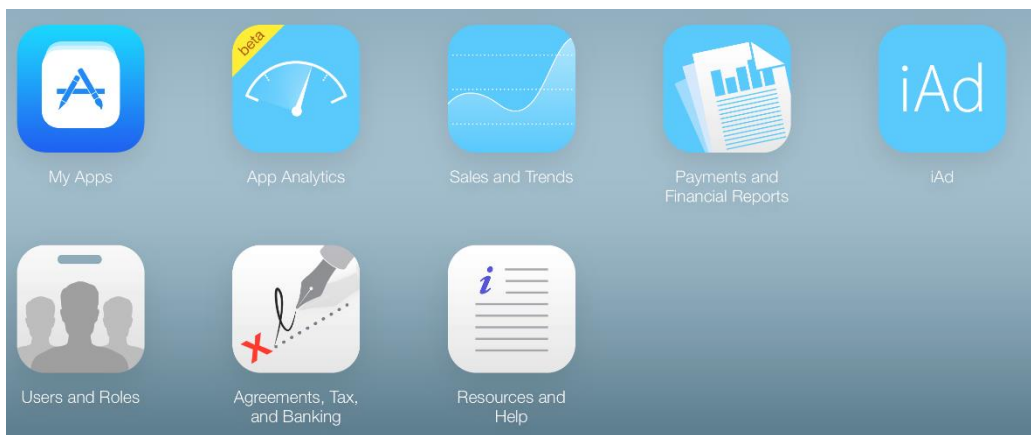


Figure 59. iTunes Connect dashboard

The iTunes Connect page handles all communication with the AppStore. Once the App ID is created in the Member Centre, it is possible to create the app record and all information of the app such as description, screenshots, price tier, company information, iTunes logo, geographic targeting, upload build from Xcode.

When the "Create new App" option is selected, it is only possible to select the app IDs that are created in Member Centre. This step must be performed before creating the app record in iTunes Connect.

To summarize, the Member Centre is the centralized point for handling the technical part of the app, while iTunes Connect is the portal that connects the app to the AppStore.

When developing apps for iOS it's not possible to get the app in a file and run it on a phone as it is with Android (unless the iPhone is Jailbroken). To upload the app to the app record, the build have to be uploaded to iTunes connect from Xcode. That is done by logging in with the same account, as the app is being created by, and then you can archive the product and submit it to the app store. When that is done, it is possible to see all uploaded builds in iTunes Connect, and then it's possible to select the one to submit.

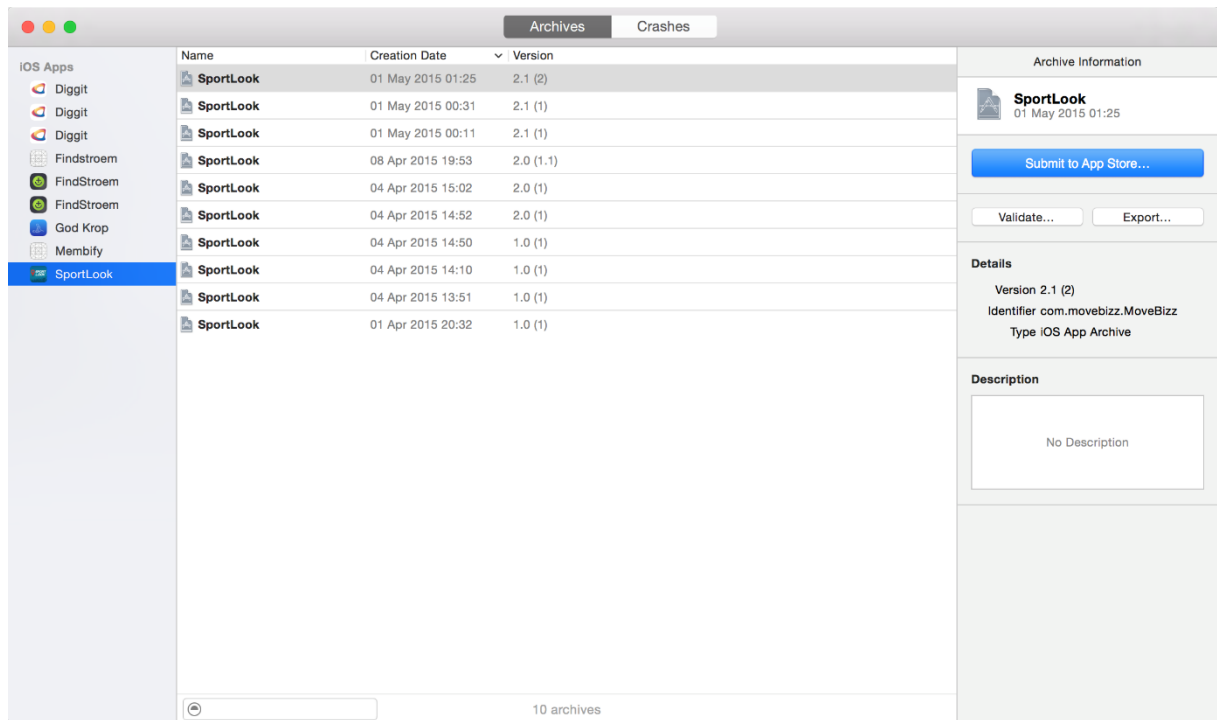


Figure 60. Xcode's Application Uploader

6.3. Updates

The first version 2.0 (1.0 was MoveBizz, the app name changed in the update) of this app was released 1st of April 2015, date chosen by the client. The app is to be released in two versions. The first was a basic one, where the user could:

- Create profile & login with an existing one
- Create / join / delete events
- View events and event information
- View own profile
- Edit profile

The second version (2.1) was released one month after (1st of May 2015) and it benefits from extra functionality:

- Push notifications
- Chat with other participants in the same event (only if you attending the event)
- Discover Function: Find other people by radius, and see their favourite sport categories & attending events
- Invite other people to events
- Public Profile

Unless there are new features added such as Push Notifications, In-App Purchases, Game-Centre, etc. the update can be made only in iTunes connect. If there is a change to the app's functions (as mentioned above), it would be necessary to create the certificates in Member Centre. For the first version it was necessary to create a new provisioning profile, because the

"signer" of the app was not the same one as version 1.0. In the second one it was necessary to create certificates for Push-Notifications. During development, development profiles were used, and before publishing, production profiles were created.

6.4. Sales and Trends

Since the application is publicly available in the AppStore for a period of 60 days, there is a limited analytics data available. The figure below shows number of downloads from 1st February to 25th May 2015.

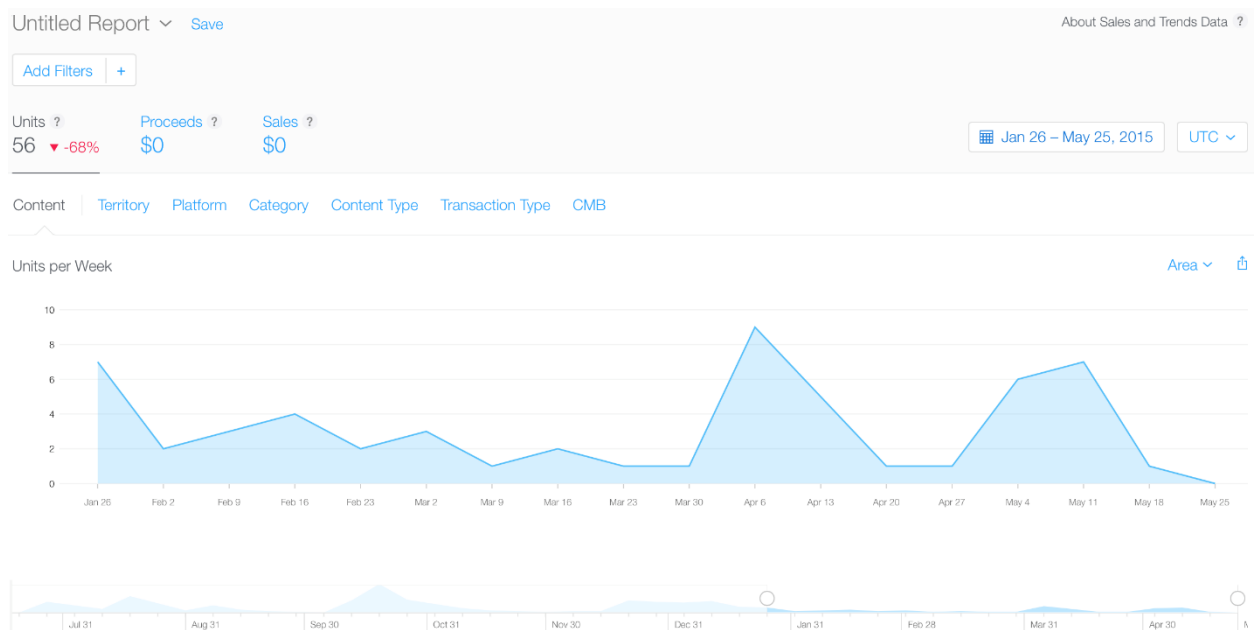


Figure 61. Sales and trends

7. Results

The results of the project is a client-server system, which functions as expected. The system successfully represents a sport events social network. Multiple clients on multiple platforms can interact through a server and attend sport events. All of the listed requirements were implemented and tested in various scenarios.

The client (iOS application) behaves as expected in all of the targeted iOS devices and all of the iOS versions.

The server has a robust architecture, being able to scale up in case of need. The Web Service is able to communicate over HTTP(S) via a RESTful API, which can be called in the targeted platforms (Android and iOS) plus others if needed. The API defines functionalities that are independent of their according implementations. The server also successfully communicates with the 3rd party services used, Parse and AWS S3. Data can be stored and

retrieved from the AWS S3 and chat messages can be stored and retrieved from Parse, as well as sending and receiving push notifications.

Every component of the system is functional and the system as a whole behaves as expected.

8. Discussion

Even though the system is fully functional and behaving as expected, it cannot be assumed that the system is flawless. Due to time constraints, several testing methods and scenarios were not performed. Therefore, it's possible that the performance may not be up to expectations in some situations.

The outcome of the project is a system that can sustain a moderate amount of active users (roughly approximated up to 1000). As stated in the delimitations of the project, scalability was not taken into account. Therefore, the system could benefit from scalability adjustments when the number of active users increases, otherwise it might behave poorly.

The full list of requirements was met and all of the problems that presented themselves during the development were resolved to a reasonable acceptance. The system did not have any known reproducible errors at the time of deployment.

8.1. Possible improvements

Mainly all of the possible improvements presented were not addressed due to time constraints.

Scalability and low cost

The system could benefit from scalability improvements in order to support an increased amount of users. There was no testing performed in the situation of high traffic, therefore there is no estimate of how would the system perform.

Also, when the system would acquire a large user base, scaling up while using the current 3rd party services dependencies would become very costly. Only the AWS S3 should be kept, while the other services can be replaced by an own implementation. Parse could be replaced with the lower-cost Amazon Simple Notification Service. Heroku could be replaced with the lower-cost Amazon Elastic Compute Cloud.

The following diagram illustrates the cost of scaling up from 30 requests per second to 50 requests per second, in Parse. 200\$ per month may not correspond to the project's objective of keeping a low cost.

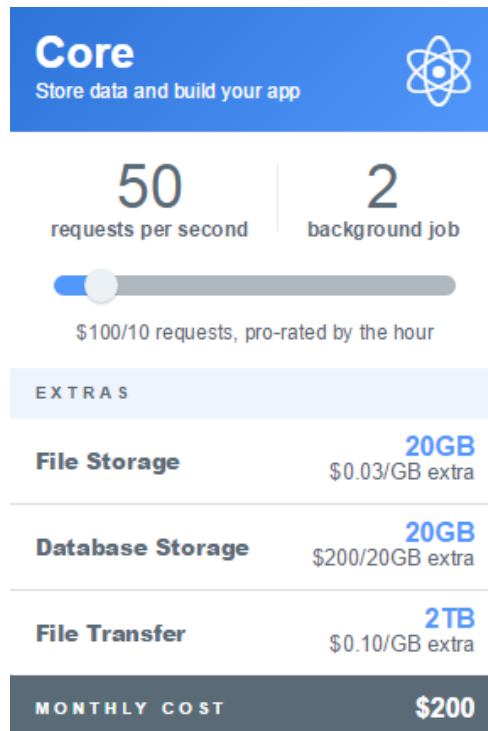


Figure 62. Scaling with Parse

Multiple types of testing

User interface testing, manual functional testing, usability testing and offline navigation testing were performed on the system. In order to ensure high performance and stability, more types of testing should be performed. A testing framework such as XCTest (embedded in the Xcode IDE) could be used for assertion tests or unit tests. Test automation frameworks such as Appium could be used for speeding up the testing. Also, a tool such as Travis Continuous Integration could be added to the development work flow, which would run automated unit tests to each new commit.

The system could also benefit from further usability testing, where a user would be given a certain task to perform and the results would be observed, respectively how fast and easily can the user achieve the task.

Complete testing on device and OS version

The iOS application was fully manual functional tested on a limited number of iOS devices and iOS versions, since it is a time consuming task. Also, the development team only had 2 physical devices for testing: iPhone 5 (running iOS 7.1) and iPhone 5S (running iOS 8.3). The rest of the testing was performed with the iOS simulator.

Facebook SDK integration

Basic testing was performed for the integration of the Facebook iOS SDK. However, the iOS application could benefit from further testing of the situation when the requested Facebook permissions are not granted by the user.

Offline support

Currently, the client (iOS application) does not store persistent data when it communicates with the Web Service. A cache system could be implemented with custom business logic, making the application usable even when there is no network connectivity.

HTTP request time-out

The open source library used for performing HTTPS requests from the iOS application has a default time-out of 60 seconds. The time-out period could be reduced to 20-30 seconds for a better user experience.

Improved security

The system accounts for security through using HTTPS requests (instead of HTTP) for client-server communication and storing sensitive user information in the iOS application in Keychain Access (encrypted storage). This does not mean that the system is safe from attacks which could compromise user data. The security enforced should be tested, for example through attacks simulations.

8.2. Unexpected situations

Swift 1.2

During the development of the iOS application, Apple released a new version of the Swift syntax, Swift 1.2. The new syntax was embedded in the Xcode 6.3 IDE version. The problem was that testing the application on the iOS 8.3 version was not possible, unless the new Swift syntax was used. The upgrade produced more than 60 compilation errors in the project. Extra-work was put into converting the project to the new syntax.

Heroku's pricing change

Towards the end of the development period, Heroku (PaaS used for hosting) changed its plan pricing. This affected the allowed uptime of the server per day, from 24 hours to 18 hours, according to the Heroku plan used. In order to keep the server up 24 hours per day, a plan change is necessary which comes with a cost of 7\$ monthly. The cost could be lowered if the server would be moved to an instance of Amazon Elastic Compute Cloud.

8.3. Monetization

Currently, the iOS application is free. This business approach would work with a small user base, but as this increases, another business approach should be used, since increased traffic results in higher costs for maintaining the infrastructure of the system.

Studies have shown that currently the dominant business model in mobile applications are free applications with in-app purchases for premium features. Research would need to be done

to come up with relevant in-app purchases that users would be willing to pay for. Also development time needs to be invested in implementing this potentially new feature plus the premium features.

Therefore, a key issue in the future of the system would be to find a way to generate revenue, in order to sustain the infrastructure.

9. Conclusion

The objective of the project was achieved. A client-server system is fully functional, allowing users to interact through an iOS application via a Server. The system is deployed and available to the public.

The current version would support a roughly estimated amount of 1000 users, but scalability improvements are needed to support a larger user base. The client user interface was designed for improved usability and a rich user experience. The technology chosen for implementing the iOS application clearly supports the needs of the project.

All the problems defined in the beginning of the project were addressed to an acceptable extent, with positive outcomes. The delimitations set were useful due to offering control of the development time.

The product is a medium size project, with roughly 14500 lines of code written just for the iOS application. The project is considered a success and a satisfactory result.

10. References

- Multitier architecture [May 2015] http://en.wikipedia.org/wiki/Multitier_architecture
- Building a hybrid app [May 2015] <http://www.smashingmagazine.com/2014/02/11/four-ways-to-build-a-mobile-app-part3-phonegap/>
- Building a cross-platform app [May 2015] <http://www.smashingmagazine.com/2014/03/10/4-ways-build-mobile-application-part4-appcelerator-titanium/>
- Building a native iOS app [May 2015] <http://www.smashingmagazine.com/2013/11/22/four-ways-to-build-a-mobile-app-part1-native-ios/>
- Mobile development approaches [May 2015] <http://www.appmakr.com/blog/hybrid-apps-mobile-market/>
- Native vs web [May 2015] <http://readwrite.com/2015/02/27/native-vs-web-apps-ceasefire>
- Hybrid app explanation [April 2015] <http://developer.telerik.com/featured/what-is-a-hybrid-mobile-app/>
- Cross-platform solution [April 2015] http://developer.xamarin.com/guides/cross-platform/getting_started/introduction_to_mobile_development/

- Web vs native [April 2015] http://www.quirksmode.org/blog/archives/2015/05/web_vs_native_1.html
- Facebook Reactive Native [May 2015] <https://facebook.github.io/react-native/>
- Swift introduction [March 2015] https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/
- Swift vs Objective-C [March 2015] <http://www.archer-soft.com/en/blog/technology-swift-vs-objective-c-pros-and-cons>
- Swift vs Objective-C performance [March 2015] <http://www.quora.com/How-is-Swift-faster-than-Objective-C>
- Swift and Objective-C comparison [March 2015] <http://thenewstack.io/some-reasons-why-swift-is-better-than-objective-c/>
- Swift influences [March 2015] <http://www.infoworld.com/article/2606431/application-development/155797-10-prominent-features-stolen-by-Apple-s-Swift-and-where-they-came-fro.html>
- Swift analysis [March 2015] <http://www.toptal.com/swift/from-objective-c-to-swift>
- Swift advantages [March 2015] <http://www.infoworld.com/article/2920333/mobile-development/swift-vs-objective-c-10-reasons-the-future-favors-swift.html>
- Swift and Objective-C 2nd comparison [March 2015] <http://mobileoop.com/the-comparison-between-swift-and-objective-c-programming-language>
- iOS layers [April 2015] <https://developer.apple.com/library/prerelease/ios/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/Introduction/Introduction.html>
- iOS development course [March 2015] <https://itunes.apple.com/bj/course/developing-ios-8-apps-swift/id961180099>
- iOS design patterns [April 2015] <https://developer.apple.com/library/ios/referencelibrary/GettingStarted/RoadMapiOS/DesignPatterns.html>
- UIViewController life cycle methods [April 2015] <http://rdkw.wordpress.com/2013/02/24/ios-uiviewcontroller-lifecycle/>
- iPhone resolutions [March 2015] <http://www.paintcodeapp.com/news/ultimate-guide-to-iphone-resolutions>
- iOS Human Interface Guidelines [April 2015] <https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/MobileHIG/>
- Autolayout [May 2015] <https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/AutolayoutPG/Introduction/Introduction.html>
- Sketch design tool [March 2015] <http://bohemiancoding.com/sketch/>
- Designing an iOS UI [April 2015] <http://www.toptal.com/ios/ios-user-interfaces-storyboards-vs-nibs-vs-custom-code>
- Wireframe [April 2015] http://en.wikipedia.org/wiki/Website_wireframe
- Open source component for Chat [May 2015] <https://github.com/jessesquires/JSQMessagesViewController>

- Apple Push notifications [May 2015] <https://developer.apple.com/library/ios/documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/Chapters/ApplePushService.html>
- Parse push notifications [May 2015] <https://parse.com/docs/ios/guide#push-notifications>
- Testing with Xcode [May 2015] https://developer.apple.com/library/mac/documentation/DeveloperTools/Conceptual/testing_with_xcode/Introduction/Introduction.html
- Test automation tool [May 2015] <http://appium.io/>
- Continuous integration [May 2015] http://en.wikipedia.org/wiki/Continuous_integration
- Monetize apps [May 2015] <http://thinkapps.com/blog/post-launch/monetize-apps-paid-apps-vs-app-purchases-vs-freemium-vs-subscription/>
- Heroku file system [May 2015] <https://devcenter.heroku.com/articles/read-only-filesystem>
- Encryption algorithm [May 2015] <http://en.wikipedia.org/wiki/Bcrypt>

11. Appendices

The Appendices are located under the Appendices folder.

Activity diagrams

Under *Appendices/Activity diagrams*

Astah files

Under *Appendices/Astah files*

Figures

Under *Appendices/Figures*

List of requirements

Under *Appendices/List of requirements*

Project description

Under *Appendices/Project description*

Source code

Under *Appendices/Source code*

Test specifications

Under *Appendices/Test specifications*

Use case descriptions

Under *Appendices/Use case descriptions*

User interface design

Under *Appendices/User interface design*