CZECH TECHNICAL
UNIVERSITY
IN PRAGUE

F3

Faculty of Electrical Engineering
Department of Computer Science

Bachelor's Thesis

Contextual iOS app for MyICPC with context acquisition for IoT environments

Vladyslav Gorbunov
Open Informatics, Software Systems

May 2018
Supervisor: Ing. Tomáš Černý, MSc., Ph.D.

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Gorbunov** Jméno: **Vladyslav** Osobní číslo: **434997**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Studijní obor: **Softwarové systémy**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Kontextová iOS aplikace pro MyICPC se sběrem contextu v IoT prostředí

Název bakalářské práce anglicky:

Contextual iOS app for MyICPC with context acquisition for IoT environments

Pokyny pro vypracování:

Prozkoumejte možnosti sběru signálu a meta-informací o síti na WiFi a Bluetooth v mobilní platformě iOS.
Věnujte pozornost WiFi Channel State Information a meta-informacím o dalších zařízení na síti. Dále amplitudě a fázovému posunu [2,3].
- Prozkoumejte systém MyICPC [4] a interakci s mobilním zařízením pro účel interakce a účasti ve hře Quests s možností zaslání videa a obrázku.
- Prozkoumejte zabezpečení proti kompromitaci skrze Trusted Execution Environments (TrustZone) [5] či podobné.
- Implementujte prototyp mobilní aplikace komunikující s MyICPC a otestujte použití.

Seznam doporučené literatury:

[1] Michals Trnka, Martin Tomasek, and Tomas Cerny. Context-aware security using internet of things devices. In Kuinam Kim and Nikolai Joukov, editors, Information Science and Applications 2017: ICISA 2017, pages 706?713, Singapore, 2017. Springer Singapore.
[2] Cong Shi, Jian Liu, Hongbo Liu, and Yingying Chen. Smart user authentication through actuation of daily activities leveraging wifi-enabled iot. In Proceedings of the 18th ACM In- ternational Symposium on Mobile Ad Hoc Networking and Computing, Mobihoc ?17, pages 5:1?5:10, New York, NY, USA, 2017. ACM.
[3] Ioannis Agadakos, Per Hallgren, Dimitrios Damopoulos, Andrei Sabelfeld, and Georgios Por- tokalidis. Location-enhanced authentication using the iot: Because you cannot be in two places at once. In Proceedings of the 32Nd Annual Conference on Computer Security Applications, ACSAC ?16, pages 251?264, New York, NY, USA, 2016. ACM.
[4] Smetana Roman, Next generation of Second-Screen, Realtime application MyICPC, 2016, https://dspace.cvut.cz/handle/10467/62711
[5] Claudio Marforio, Nikolaos Karapanos, Claudio Soriente, Kari Kostiainen, and Srdjan Capkun. Smartphones as practical and secure location verification tokens for payments. In NDSS, 2014.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Tomáš Černý, MSc., Ph.D., laboratoř inteligentního testování softwaru FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **13.12.2017** Termín odevzdání bakalářské práce: **25.05.2018**

Platnost zadání bakalářské práce: **30.09.2019**

Ing. Tomáš Černý, MSc., Ph.D.		prof. Ing. Pavel Ripka, CSc.
podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

.	
Datum převzetí zadání	Podpis studenta

Acknowledgement

First and foremost, I would like to thank my supervisor Ing. Tomáš Černý, MSc., Ph.D. for interesting thesis topic, his guidance and advices. Furthermore, this thesis would not be possible without a support of my family and friends.

Declaration

Abstrakt / Abstract

Systém MyICPC byl navržen, aby podpořil informovanost a zájem účástníků a zaměstnanců během programovacích soutěží ACM-ICPC. Zatímco většina modulů MyICPC slouží primárně pro prezentaci soutěžních výsledků v reálném čase, existuji taktéž moduly pro podporu zájmu o sociální aktivity, sdílení fotek a textové zpětné vazby. Quest je jedním z několika interaktivních modulů navržených obzvláště, aby probudily zájem účástníků o ICPC, historii ICPC a pořádající univerzitu.

Cílem této bakalářské práce je zpřístupnit Quest širšímu publiku prostřednictvím nativní aplikace pro mobilní zařízení s operačním systémem iOS. Součástí implementace je taktéž podpora pro sběr kotextových informací o zařízení, které budou v budoucnosti použité pro výzkum ICPC zaměřený na bezpečnost autentizace.

Klíčová slova: MyICPC Quest; mobilní aplikace; iOS; Swift; IoT kontext.

Překlad titulu: Kotextová iOS aplikace pro MyICPC se sběrem kontextu v IoT prostředí

MyICPC system was developed to support awareness and interest of contestants and staff during the ACM-ICPC programming contests. While most MyICPC modules are responsible for real-time contest data presentation, there are several modules to support user social activities, sharing photos and user feedback. Quest is one of such interactive modules designed specifically to encourage participants to learn something more about ICPC, ICPC history and hosting university.

This thesis aims to make Quest accessible to wider audience via native application for mobile devices running on iOS operating system. Also, part of the implementation is support for contextual device data acquisition, which will be used in future ICPC research on Authentication Security.

Keywords: MyICPC Quest; mobile application; iOS; Swift; IoT context.

Contents /

Tables / Figures

Chapter 1
Introduction

ACM-ICPC is a multitier, team-based, programming contest. The contest involves a global network of universities hosting regional competitions whose teams are then being advanced to the ACM-ICPC World Finals. With an active involvement of worldwide known universities, ACM-ICPC is an event with a wide public interest.

1.1 MyICPC Quest, motivation for mobile clients

MyICPC was developed 4 years ago for the purpose of gathering large amounts of contest-related data and serving these data in real-time using an adaptive web interface [1]. However, MyICPC is not just about ICPC contest. Apart from real-time result presentation, there are also different modules to support social activity.

Quest is one of such modules. It is a unique way to encourage contest participants to get to know more information about ICPC, hosting university city or even other participants. All that is achieved through Quest challenges. Challenge is a task that the participant should be able to solve by posting a submission with either text, image or video. Quest game typically starts before contestants arrival to hosting university and ends after final results being announced. For each submission, if accepted, participant receives points which will affect participant positioning in a leaderboard.

Motivation for mobile clients is mainly influenced by vague user submission flow. First, user finds a specific Quest challenge to participate in. To make a Quest submission user is then required to post to Twitter with a unique challenge hashtag. Given that vast majority of challenges is an outdoor activity, navigating between multiple applications is rather cumbersome. With the emerging amount of mobile clients, a decision was made to elaborate on mobile application for both iOS and Android which will result in more pleasant user and ICPC contest experience. Android version of Quest, as well as REST API for mobile clients are part of Filip Ryšavý diploma thesis [2–3].

1.2 Thesis goals

This thesis aims to fulfill following tasks:

- Solve the problem of unnecessary navigation between multiple applications, while achieving same task, by introduction of mobile application prototype. Such prototype will facilitate navigation between contest timeline, leaderboards, list of challenges and will enable easy and straightforward media files submission.
- Comply with the requirement on device information and contextual network data acquisition for future research.
- Design and implement a prototype of mobile application for iOS with respect to current trends in iOS development. Place emphasis on user experience.
- Integrate mobile client with the provided REST API.

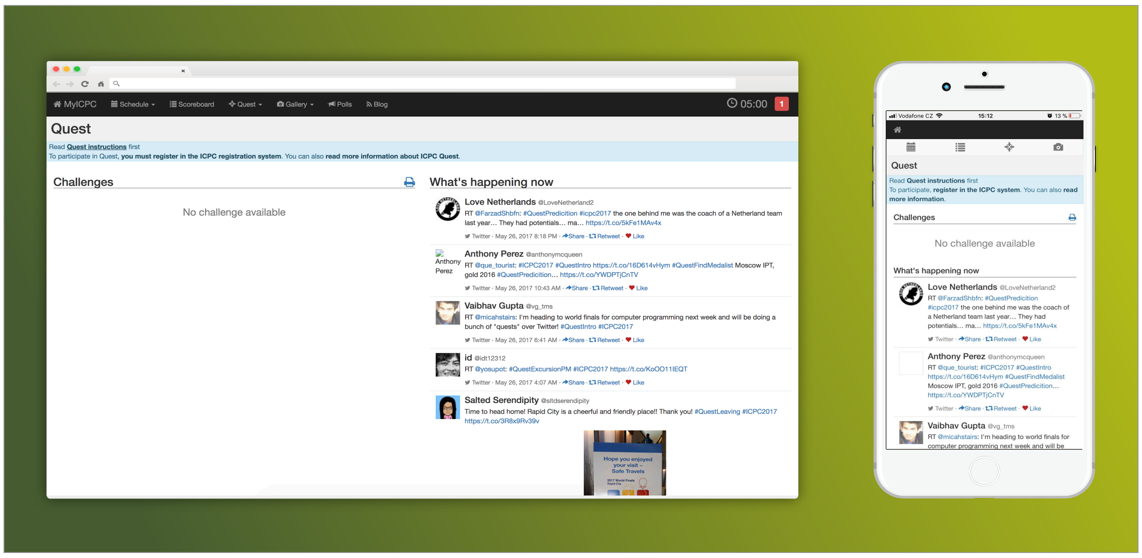
1.3 Structure of the thesis

The chapter **Related Work** presents current Quest web interface for mobile devices. Also, influences affected final graphical design proposal are explained. Next, the chapter **Analysis and Design** focuses on main design decisions made before implementation, such as selection of programming language, application architecture, synchronization techniques. This chapter also addresses restrictions on contextual data acquisition. The chapter **Implementation** takes a closer look at implementation specifics, topics such as navigation delegation, authentication state management and authentication state persistence across application relaunch are discussed. In the chapter **Testing** there are presented some user and unit tests. Finally, chapter **Conclusion** covers achieved goals and possible issues to be resolved in the future.

Chapter 2
Related Work

2.1 Current solution

Current Quest implementation uses adaptive web interface to present a list of challenges as well as latest submissions. An example from ACM-ICPC World Finals 2017[1] is depicted on Figure 2.1.



Read **Quest instructions** first
To participate in Quest, **you must register in the ICPC registration system**. You can also **read more information about ICPC Quest**.

No challenge available

Love Netherlands @LoveNetherland2
RT @FarzadShbfn: #QuestPredicition #icpc2017 the one behind me was the coach of a Netherland team last year… They had potentials… ma… https://t.co/5kFe1MAv4x
Twitter · May 26, 2017 8:18 PM · Share · Retweet · Like

Anthony Perez @anthonymcqueen
RT @que_tourist: #ICPC2017 #QuestIntro https://t.co/16D614vHym #QuestFindMedalist Moscow IPT, gold 2016 #QuestPredicition… https://t.co/YWDPTjCnTV
Twitter · May 26, 2017 10:43 AM · Share · Retweet · Like

Vaibhav Gupta @vg_tms
RT @micahstairs: I'm heading to world finals for computer programming next week and will be doing a bunch of "quests" over Twitter! #QuestIntro #ICPC2017
Twitter · May 26, 2017 6:41 AM · Share · Retweet · Like

id @idt12312
RT @yosupot: #QuestExcursionPM #ICPC2017 https://t.co/KoOO11IEQT
Twitter · May 26, 2017 4:07 AM · Share · Retweet · Like

Salted Serendipity @sltdserendipity
Time to head home! Rapid City is a cheerful and friendly place!! Thank you! #QuestLeaving #ICPC2017 https://t.co/3R8x9Rv39v

Figure 2.1. MyICPC Quest interface for web clients.

While following best practices on mobile web client design and displaying all necessary information, web client lacks an important feature: user submission support. Instead, Twitter is used to perform submissions, as shown on Figure 2.4. Submissions are then queried by MyICPC using contest specific hashtags.

2.2 Inspected applications

Here I list some examined mobile clients which influenced the final user interface design.

2.2.1 Medium

Medium[2] is a popular blogging platform with mobile clients playing crucial role in company's business. Medium home tab displays user stories with title, ellipsized story description and story thumbnail, see Figure 2.2. List item contents also have information about user and day posted. Unfortunately, this is not a suitable way to display

[1] http://myicpc.icpcnews.com/World-Finals-2017/quest
[2] https://itunes.apple.com/us/app/medium/id828256236?mt=8

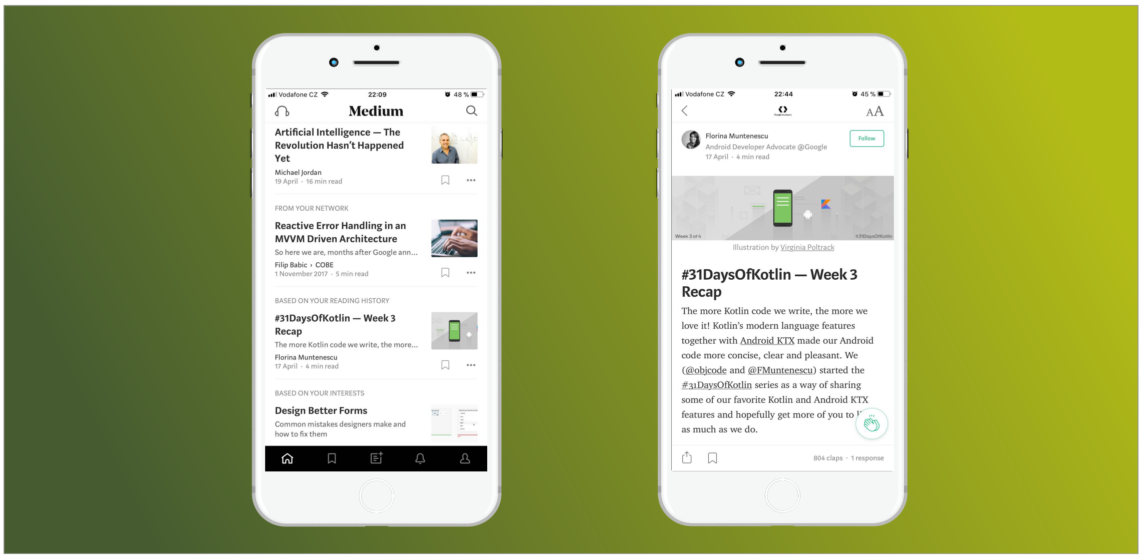

Medium
Artificial Intelligence — The Revolution Hasn't Happened Yet
Reactive Error Handling in an MVVM Driven Architecture
#31DaysOfKotlin — Week 3 Recap
Design Better Forms
Follow

Figure 2.2. Medium for iOS.

Quest submission as it may contain both image and a video. However, it may be a choice to go with for displaying a challenge list, as challenges may have a thumbnail picture.

2.2.2 Reddit

Reddit[1] is one of the most popular discussion websites. Reddit app is not particularly relevant for Quest, but it is still worth to examine since it displays small portion of information, often with a thumbnail image. As can be seen on Figure 2.3, application uses different layout for list items with expanded thumbnails. This, however, is not the best fit, since Quest application needs to display a lot of information about user and challenge.

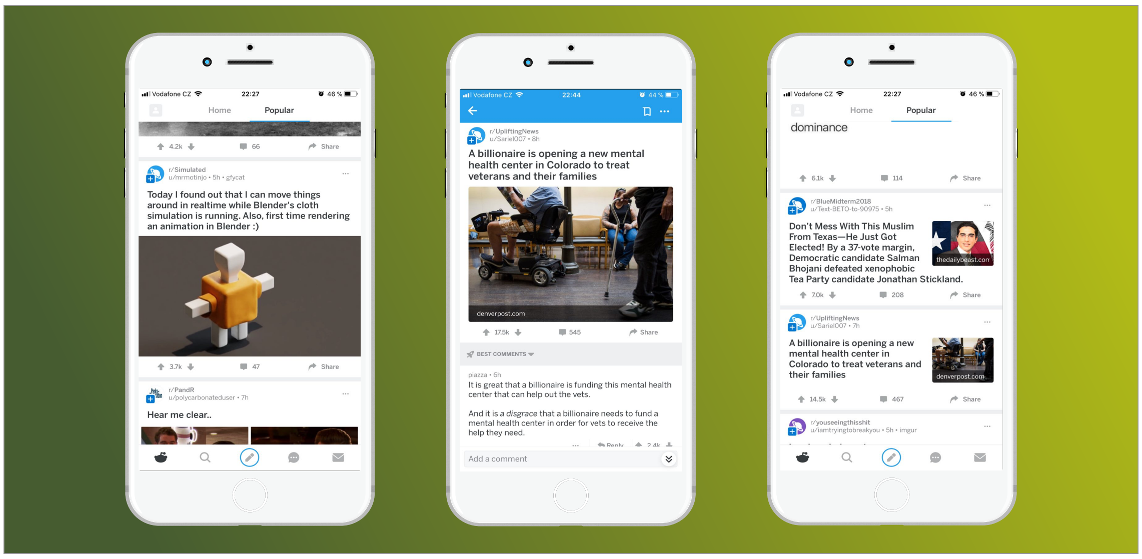
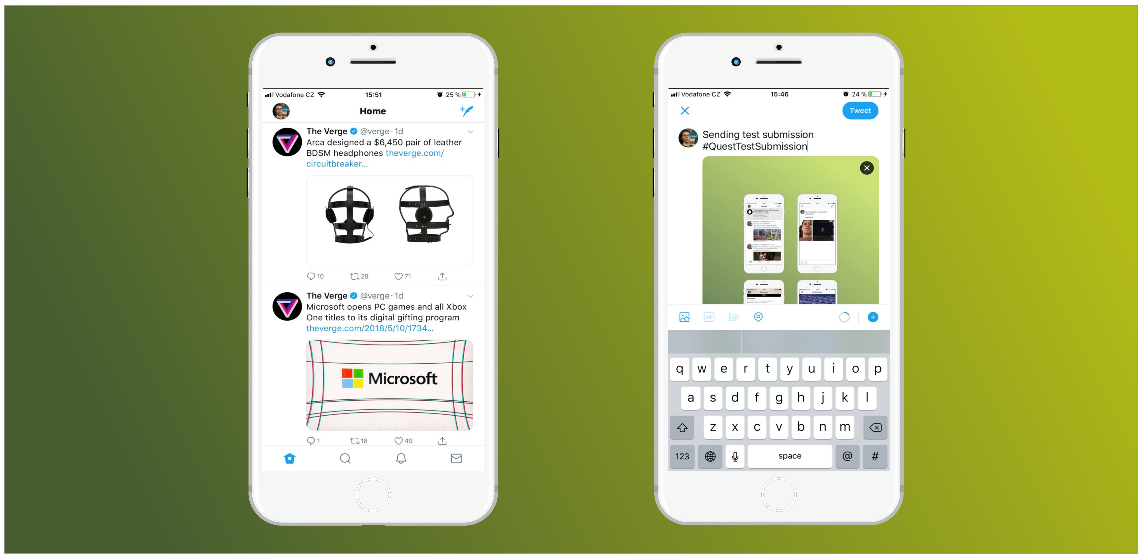

Home
Popular
Share
Today I found out that I can move things around in realtime while Blender's cloth simulation is running. Also, first time rendering an animation in Blender :)
A billionaire is opening a new mental health center in Colorado to treat veterans and their families
Add a comment
Don't Mess With This Muslim From Texas—He Just Got Elected!

Figure 2.3. Reddit for iOS.

[1] https://itunes.apple.com/us/app/reddit-official-trending-news/id1064216828?mt=8

2.2.3 Twitter

Twitter[1] is a social network Quest was originally built upon. The identifying application feature is the way how information is displayed, see Figure 2.4. A single list item contains a lot of information, which, being well arranged, are easily readable. Additionally, stacking text with image and video can allow multiple media file display in the future. With the current solution in mind, I have decided to adopt Twitter design concepts to preserve user experience.


Vodafone CZ 15:51 25 %

The Verge @verge · 1d
Arca designed a $6,450 pair of leather BDSM headphones theverge.com/circuitbreaker...
10 29 71

The Verge @verge · 1d
Microsoft opens PC games and all Xbox One titles to its digital gifting program theverge.com/2018/5/10/1734...
1 18 49

Vodafone CZ 15:46 24 %

Sending test submission #QuestTestSubmission

Figure 2.4. Twitter for iOS.

[1] https://itunes.apple.com/cz/app/twitter/id333903271?mt=8

Chapter 3
Analysis and Design

This chapter covers demands placed on application, analysis and design decisions which had to be made before implementation. Some common trends in iOS development are investigated, best practices and alternative approaches are examined before final decision whether specific technology, framework or architectural approach will be used in final project. Some decisions were influenced by prior experience with Android development.

3.1 Programming Language

There is a variety of languages available when it comes to iOS development. The most popular ones have proven to be ReactNative from Facebook, Xamarin from Microsoft, Swift and Objective-C from Apple. I am still very skeptical when it comes to cross-platform development, mainly due to maintenance, and, given the requirement to collect contextual device data, thus using system frameworks extensively, I have made a choice to go with native approach, meaning I had to choose either Swift or Objective-C. Objective-C is rather old language, based on C and Smalltalk. Its vague syntax made me quickly turn my attention to other, more modern alternative. Swift, unlike Objective-C, is a modern, multi-paradigm language developed with existing Apple frameworks compatibility in mind. It immediately seemed like a good candidate, given its popularity[1] and many attractive features.

The following list stresses the most significant ones:

- Protocol oriented. In most modern languages protocols are known as interfaces. One of the Swift's protocols most powerful features is protocol extension, which allows existing third party classes to conform to user-defined protocols.
- Powerful enums and pattern matching. This feature is especially handy when using the Result Pattern[2]. A user-defined enum with associated values can represent a stateful operation state, using pattern matching we then match the returned value against predefined conditions.
- Swift introduces an important concept of optional types, referred as Null Safety in other languages. This type-system feature aims to eliminate null-pointer errors by Optional enumerations which can be either `.none` or `.some` with backing value of original type.
- Introduction of variable scope (and immutability) with `var` and `let`.
- Functions, as well as objects and structs, are now first-class citizens.
- Swift is a very rich language in terms of syntax. With an implicit support for weak variables, lazy loading and concise semantic keyword meaning a well structured Swift code implies good code readability.

At the time of development used Swift versions are 4.0 and 4.1, more on Swift in [4–5].

[1] `http://redmonk.com/sogrady/2018/03/07/language-rankings-1-18/`
[2] `https:// fsharpforfunandprofit.com/posts/recipe-part2/`

3.2 Dependency Management

For third party libraries integration I have decided to use dependency management tools. Dependency management tool helps keeping specific dependency versions and updating them easily to newer ones. My attention was drawn by two most used ones, Carthage and CocoaPods, both of which are very similar, key difference is the method of integration.

3.2.1 CocoaPods

CocoaPods is an open-source command line tool highly managed by community [6]. Its task is to download specified dependencies and create a new Xcode (Apple's IDE for iOS developers) workspace with all linked dependencies. Created workspace then has to be used for further development. Main drawback of this approach is that all dependencies are linked directly into project and thus not only application code is compiled, but all dependencies.

This will be used as a secondary dependency management tool for dependencies unavailable with Carthage.

3.2.2 Carthage

Carthage is an open-source command line tool originally developed by Github [7]. Its task is to download specific dependency versions defined within Cartfile and compile dependency source code into frameworks. User is then responsible for linking compiled frameworks into project. This is a preferred approach as it does not require dependencies to recompile each time we run application code. To improve compile time user can specify preferred platform (macOS, iOS, tvOS, watchOS) dependency code to be compiled for. Dependency contributors may also include pre-compiled binaries, which are then downloaded and linked directly into project. Pre-compiled dependencies need be written in same Swift version, however. Otherwise, users will not be able to link frameworks into project. This typically happens only when migrating to newer Swift version, which in my case was Swift update from 4.0 to 4.1.

This will be used as a main dependency manager, largely because of faster build times and more straightforward usage.

3.3 App architecture, MVVM vs. Apple-MVC

3.3.1 MVC: Model-View-Controller

Since the very first release of iOS in 2007 Apple suggests developers to use a Model-View-Controller (MVC) design pattern [8].

Model represents objects which define application data-logic. This can be both Data Transfer Objects (DTOs) and objects containing logic responsible for data retrieval. This tier is typically represented by remote service interactor or local database access objects.

View contains all the necessary logic for intercepting user interaction and displaying data from *Model* tier. It is typically composed with hierarchy of other View objects.

Controller acts as a middleman between *View* and *Model*. Its job is to process incoming user interaction events, manipulate data using *Model* and present data to *View* to render the final output.

Despite MVC being a common way to organize code for reusability, support extensiveness and decoupling between business and UI logic, that is not the case with iOS projects. Apple's definition of MVC uses combined roles variation by coupling a View with a Controller class resulting in a so called ViewController, which is then supposed to handle both logic for user interactions as well as calls and result delivery from Model tier.

The diagram 3.1 shows the architecture.

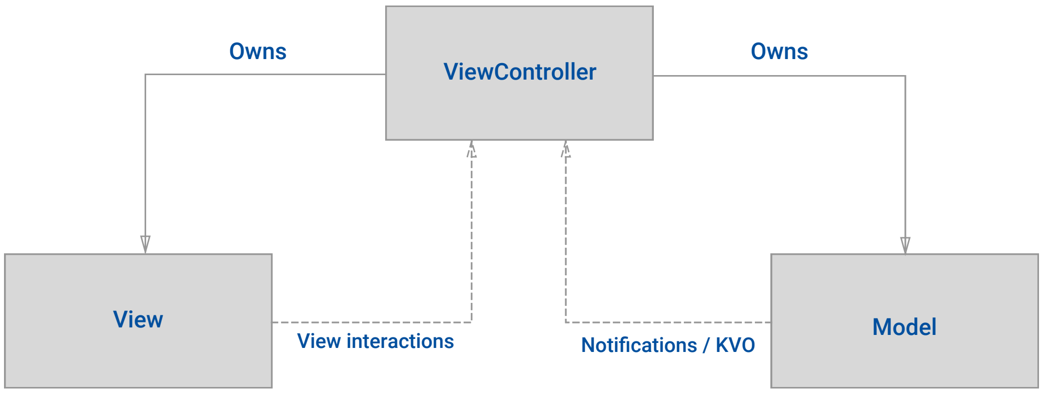


Figure 3.1. MVC architecture

The above implies tight coupling between View and Controller. Mobile applications often contain Views with complex behavior and animations which have to be managed by Controller, thus resulting in what is often called *Massive View Controller* [9], which is generally hard to test and mock.

3.3.2 MVVM: Model-View-ViewModel

The MVVM design pattern tries to solve the above drawbacks by introduction of *View-Model* tier. This design pattern was originally proposed by Microsoft for .NET frameworks specifically to simplify event-driven programming of user interfaces.

ViewModel is an object owned by ViewController, its task is to deal with persistence changes and to ensure data retrieval and preparation for View. In other words, View-Model's responsibility is performing actions initiated by user interactions intercepted by ViewController. Possible data updates triggered by such actions are then served again by ViewModel to ViewController for display. Architecture behavior can be seen on diagram 3.2.

ViewController responsibility is dynamically decreased with introduction of additional ViewModel layer when compared to MVC. At first glance, change might not seem like a big advantage, this change however allows easier testing as View layer does not have to be created, same behavior can be achieved by calling ViewModel interface methods directly.

Apart from MVC, there is an extra relation. ViewModel does not have reference to ViewController, thus we need to specify a mechanism of notifying View layer about data changes. This can be solved using Key-Value-Observable (KVO), a standard way supplied with core Apple frameworks. KVO allows to observe data changes of observed objects. Typically, ViewController intercepts user events to ViewModel, then gets notified about data change. Unfortunately, this approach does not allow synchronization

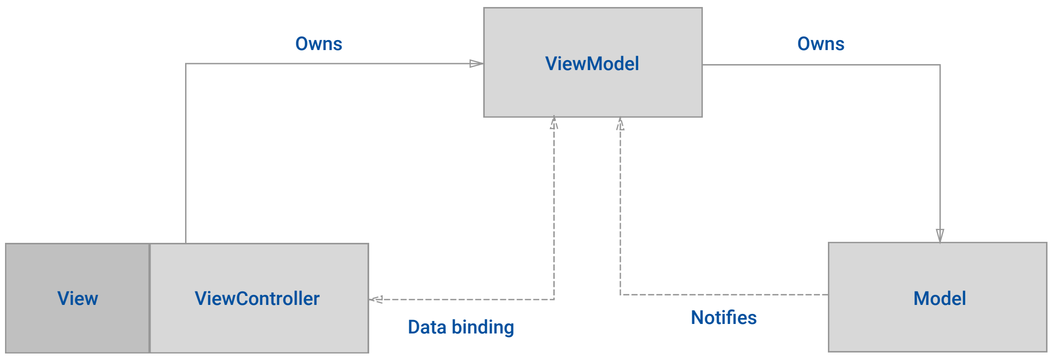


Figure 3.2. MVVM architecture

with main thread for view updates. Instead, reactive approach will be used, which is explained in the next section.

3.4 Threading and synchronization, motivation for reactive approach

Mobile applications differ a lot from desktop and web alternatives primarily in a variety of input methods. An input method can be a device location or even device orientation. One can easily imagine an application driven solely by accelerometer sensor: a compass application.

For this reason, mobile applications often need to process a lot if incoming events asynchronously to avoid main thread overtasking, which may result in irresponsive UI.

3.4.1 Grand Central Dispatch

Provided with iOS SDK, Grand Central Dispatch (GCD) is a standard approach to synchronize background operations with main thread. GCD was developed to optimize application support on multi-core processors. GCD, backed by a thread pool, manages available system resources for particular installation, completely abstracting the underlying manipulation with threads.

To distribute scheduling priority in a `DispatchQueue` instance user supplies an appropriate Quality of Service class (QoS), see Table 3.1 for further explanation.

While being very flexible, dispatch queues only work with first class functions (clojures), which cannot be chained. This functionality is available when using `OperationQueue`, where single work item is represented with an instance of `Operation` subclass. Individual tasks can be prioritized.

3.4.2 Reactive Programming

Reactive programming addresses a propagation of change using data streams. As it is most frequently used to implement user interfaces, a typical scenario is to react on user input events.

Functional reactive programming is paradigm of reactive programming which uses functional programming techniques. Similar to lists in functional programming, data streams are taken as immutable. If a stream of values needs to be transformed, then

QoS class	Work type	Work duration
User-interactive	Operations on the main thread, refreshing the user interface, or performing animations.	Instantaneous
User-initiated	Work, required to continue user interaction such as opening a photo or a document.	Nearly instantaneous, few seconds or less.
Utility	Work that may take some time to complete and doesn't require an immediate result, such as downloading or importing data.	A few seconds to a few minutes
Background	Work that operates in the background and isn't visible to the user, such as indexing, synchronizing, and backups.	Significant time, minutes or hours.

Table 3.1. Primary QoS classes summary, taken from [10].

a new stream must be created. Operators like *map*, *filter*, *flatMap* are also inspired by functional programming.

There are plenty of example use-cases from UI programming to performing network requests. For example, one would like to enable or disable a ”Send” button based on values contained in 3 distinct input fields. Another example is observing a stream of local database table changes for seamless UI update.

While there are several reactive Swift implementations available, chosen for this project is one from ReactiveX[1] called `RxSwift` [11]. This implementation of Swift reactive streams contains a lot of extensions for binding values to UIKit classes properties and has familiar API with another ReactiveX projects. A more detailed look at ReactiveX implementation of reactive streams in Swift refer to [12]. The rules on GCD scheduling priorities imply here as well.

Purpose of this section is not to justify the usage of a particular approach. `RxSwift` is used because it allows easier data mapping, filtering and general modifications. On the other hand, using `DispatchQueue` may be used in places where creating custom RxSwift extensions would not make sense due to simplicity of the given task and easier implementation using standard libraries.

3.5 View creation and graphic elements

Here I take a closer look at options available for defining user interfaces in Xcode projects. Additionally, I present tooling which was used to modify graphic elements.

3.5.1 Interface Builder vs Alternatives

A standard way to define user interfaces is the Interface Builder supplied with Xcode. Interface Builder editor allows user to compose full view hierarchy with defined navigation behavior between views. All standard ViewControllers which come with UIKit framework are supported, as well as creating support for user defined views. The whole project view hierarchy and navigation is typically stored within a *.storyboard* file. An example editor workspace can be seen on Figure 3.3.

[1] http://reactivex.io/

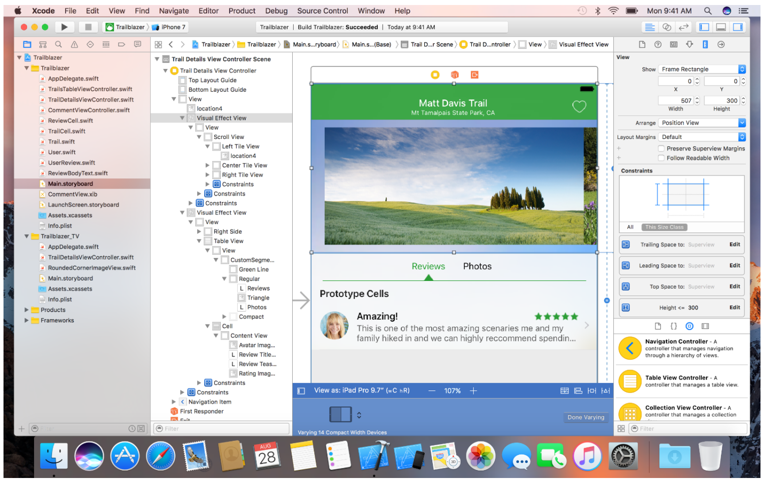

Matt Davis Trail
Mt Tamalpais State Park, CA
Reviews
Photos
Prototype Cells
Amazing!
This is one of the most amazing scenaries me and my family hiked in and we can highly reccommend spendin...

Figure 3.3. Interface Builder example, source: https://developer.apple.com/xcode/interface-builder/.

One of the most powerful components to define view positioning is Auto Layout[1]. Auto Layout is a mechanism of constraining UIView object descendant properties to properties of a container window or other UIView descendants. Interface Builder comes with direct support of Auto Layout, which can alert user when defined constraints result in a conflict.

It is worth noting that if a view defined with Auto Layout contains variable child view count, the defined constraints should be turned off for proper view stacking. This requires either storing a reference to Interface Builder defined constraint object or assembling whole constraints programmatically.

One of the common issues with Interface Builder defined views is that whole project UI definition is contained within a single *.storyboard* file. A *.storyboard* is essentially an XML file, containing almost non-human-readable Interface Builder data. This may not seem like a problem at first, however, one should imagine the source control change logs when multiple developers are working on a project. It is also worth mentioning that huge *.storyboard* files affect view inflation speed and build times.

For these reasons I have decided to use an Auto Layout Domain-Specific-Language (DSL) library. A DSL used in this project is a relatively popular open-source implementation called SnapKit [13]. SnapKit will allow us to manage dynamic data in single view component easily. A good example is a timeline list with each list item as a reusable view component where image and video thumbnail presence may vary.

3.5.2 Graphics

Application theming plays important role in mobile applications. For purposes of this project I have decided to use default system font to preserve native user interface look. All used graphics were downloaded freely and then exported using Sketch [2]. Sketch is powerful tool for prototyping and designing user interfaces. Additionally, can be used to modify imported graphics and export for specific mobile operating system to support different screen resolutions. An example Sketch workspace can be seen on figure 3.4.

[1] https://developer.apple.com/ library/content/documentation/UserExperience/Conceptual/AutolayoutPG/

[2] https://www.sketchapp.com/

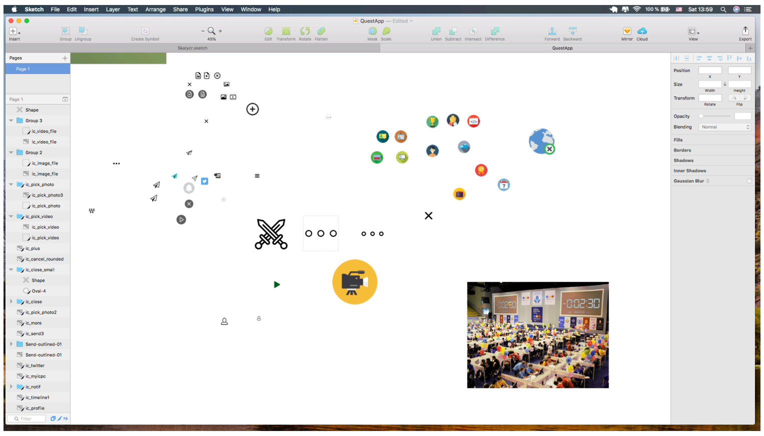


Figure 3.4. Sketch workspace with application graphics.

3.6 Functional and Non-Functional requirements

To ensure successful project completion I also had to analyse functional and non-functional requirements. Functional requirements are those directly reflected on application functionality. Non-functional requirements are application demands which are not necessary related to specific application functions.

3.6.1 Functional requirements

- Application can persist authorization state across application relaunch, as well as user profile, first retrieved making request to user endpoint.
- Application allows user to browse through contest timeline, latest Quest submissions, list of available challenges, Quest leaderboards.
- Application allows user to send media submissions. Either image, video or both can be sent with additional text.
- Every submission contains specific hashtags that uniquely identify contest and Quest challenge.
- Application has support for image display and video playback.
- Application has support for contextual device data acquisition.
- Application allows user to turn on or off device data retrieval.

3.6.2 Non-functional requirements

- Specific installation refers to specific contest, thus contest needs to be specified before application distribution.
- Application supports OpenID Connect credentials store provided by ICPC.
- Application supports Quest API 1.0 [2].
- Application user interface is based on present solution.
- Supported iOS versions are 10.3 and above.

3.7 OpenID Connect and Quest REST API

3.7.1 Quest API

Quest API for mobile clients has support for necessary data retrieval and submission. Deployed as a separate module it also manages retrieved contextual device data for later research. This module also handles verification of all incoming request using authorization framework.

3.7.2 OpenID Connect

For security reasons every request made on Quest back-end must be authorized. This is achieved using an authorization framework, which verifies user identity using uniquely assigned token. OpenID Connect (OIDC)[1] is an authentication layer on top of OAuth 2.0 authorization framework. From a mobile developer perspective all needed to be taken care of is:

- Authorization endpoint
- Token exchange endpoint
- Registered client id
- Redirect URL
- Logout endpoint

For the purpose of this project `AppAuth` is used, a library that implements OIDC specification [14]. This library follows modern practices for performing authorization requests in native clients, such as displaying request web page in *SFSafariViewController* instead of *UIWebView* [15]. Also, a big advantage of this library is automatic token exchange, meaning user only needs to react on authorization state changes.

3.8 Motivation behind Context Acquisition

Context acquisition is motivated by several studies addressing authentication security and identity verification by utilizing surrounding WiFi and BLE enabled devices. Article [16] discusses an approach to user authentication process by capturing user activities and behavioral patterns, which can be uniquely represented by surrounding WiFi-enabled devices meta-data. In [17] location-based authentication approach is described using information from nearby IoT devices. Potential treats of surrounding devices and services intervention is described in [18]. With respect to Apple restrictions and limitations of available API the most promising approach might be to make extensive use of surrounding devices conforming to Apple-developed iBeacon protocol as described in [17].

The purpose of context acquisition feature is to introduce a prototype of an iOS application capable of collecting surrounding device information and sending such data to MyICPC for analysis. Context acquisition is implemented with respect to data predefined and expected by MyICPC REST API for mobile clients [2].

3.9 Context Acquisition and Restrictions

For the purpose of future ICPC research Quest client has to support functional requirement on context data acquisition.

[1] `http://openid.net/connect/`

Contextual data are defined as:

- information about the device;
- device location;
- connected network;
- available networks;
- local network devices;
- Bluetooth devices;

Unfortunately, not everything is accessible, as Apple has always been strict about its security policy. Some of the data listed above may be gained using private libraries, these are intended to be used by Apple applications only. Given that the most convenient way to distribute this project would be the App Store, the only officially supported way, some of the requested data are ignored.

Next sections describe whether information can be retrieved.

3.9.1 Device-specific information

- source
- timestamp
- operating system
- device brand
- device model
- device serial number

Source indicates application the context is being sent from. Timestamp can be easily gained using Foundation framework classes, the later correspond to information easily retrievable using `UIDevice` class from UIKit framework [19].

3.9.2 Device location

Device location should include:

- latitude
- longitude
- timestamp

All of the listed above is easily retrievable using Core Location framework [20]. A user permission needs to be granted.

3.9.3 Connected network

Information about connected network:

- SSID
- BSSID
- RSSI
- link speed
- frequency
- IP address

This requires looping through supported system interfaces and using SystemConfiguration framework to copy interface data using `CNCopyCurrentNetworkInfo` [21]. However, only SSID and BSSID are accessible this way. There is an option to request entitlements and use `NEHotspotHelper` class, which is discussed in next subsection.

3.9.4 Available networks

This requirement violates Apple security policy. This functionality is primarily accessible to Apple applications only as it requires either private library or special entitlements. Entitlements need to be requested for NetworkExtension framework, specifically one would need to use `NEHotspotHelper` class, this option requires paid Apple Developer account [22].

3.9.5 Local network devices

Network device should contain:

- IP address
- MAC address

This task requires a classic network scanner which will ping every host available in network in order to build the ARP table. ARP table is then used to obtain MAC address and a host name. For this purpose an open-source library is used [23]. It should be taken into account that MAC address and host name retrieval will only work on devices running iOS 10.3 and below, as Apple restricts this functionality in iOS 11 due to API misuse, which enabled mobile clients to track users [24].

3.9.6 Bluetooth devices

Bluetooth device is supposed to contain:

- name
- MAC address

Apple devices running iOS support only discovery of Bluetooth Low-Energy (BLE) devices and only those, transmitting limited number of services [25]. Core Bluetooth framework does not guarantee proper name retrieval. A specific device name can be retrieved after device pairing, this may, however, drain device battery significantly. MAC address is unavailable, Core Bluetooth assigns specific device a UUID number, which is then used as unique device identifier.

Chapter 4
Implementation

For more detailed look at implementation refer to application sources at [26].

4.1 Project Environment

Project is set up to support 2 schemes: Development and Production. Each scheme has its own property list, a *.plist* file. A proporty list defines all application permissions as well as scheme-specific constants. In my case, these are: contest id, Authentication endpoints, API base URL, contest rules link and application version for context submissions. A convenience enum is declared as shown:

```
public enum Environment {

    fileprivate static let plist: [String: Any] = Bundle.main.infoDictionary!

    enum Contest {
        static var code: String { return plist["contestCode"] as! String }
    }

    enum Auth {
        fileprivate static let authDict = plist["auth"] as! [String: Any]

        static var authEndpoint: String { return authDict["AuthEndpoint"] as! String}
        /* other OIDC settings */
    }

    enum API {
        fileprivate static let apiDict = plist["api"] as! [String: Any]

        static var baseURL: String { return apiDict["BaseUrl"] as! String }
    }

    enum External {
        fileprivate static let linksDict = plist["external"] as! [String: Any]

        static var rulesLink: String { return linksDict["RulesLink"] as! String }
    }

    enum Context {
        static var appVersion: String { return plist["appVersion"] as! String }
    }
}
```

Next, project is preconfigured to support different localizations, the base one is English. To simplify work with localized strings, assets and colors I have opted to use SwiftGen which generates convenient enums based on resource declaration names [27].

With the dependency managers installed and build target specified, project is configured for development.

4.2 Application Design and Navigation use-cases

Having project set up, I have started designing a navigation map with screen prototypes. One of the functional requirements is to display 4 distinctive information types (timeline, recent submissions, challenges, leaderboards). For this reason, 4 views should be accessible form the top application level. To achieve such view composition, a `TabBarController` is used, as suggested by Apple Design Guidelines [28]. Implementation of specific screens is described in later sections.

Part of the navigation map for challenge list tab can be seen on Figure 4.1.

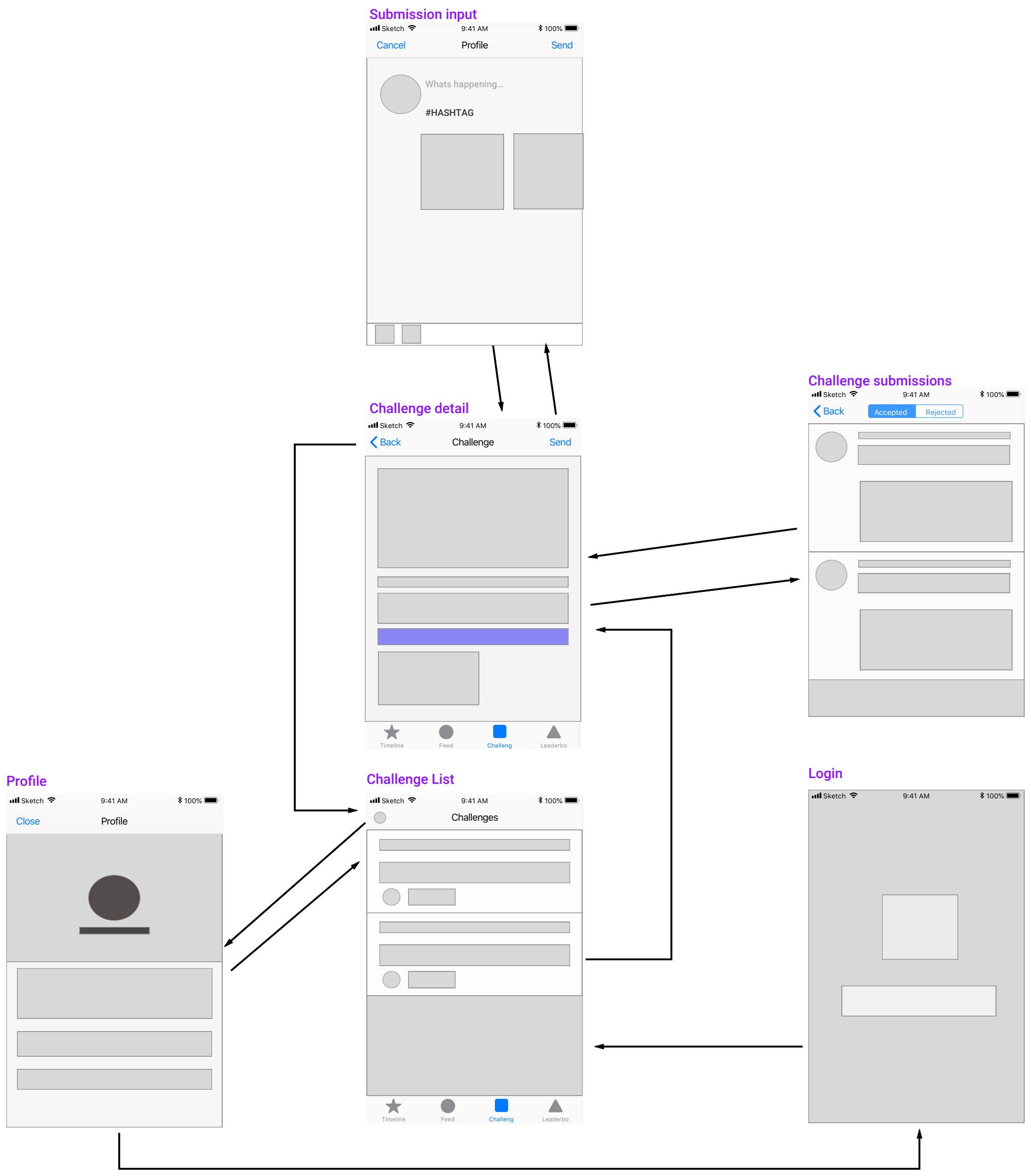


Figure 4.1. Navigation map for challenge list tab.

4.3 User managment, Keychain managment and Application settings

Components necessary for application state management (including state persistence across application relaunch) are decomposed to several *Manager* classes.

AppSettingsManager is a manager responsible for persisting application settings valid for particular installation. Later, we would like to display a dialog asking user for a permission to send contextual data on the very first application start. We will simply store a `Bool` property indicating whether permission was requested as well as whether context data fetch is allowed. These properties will be stored in `UserDefaults`, a transactional key-value store for saving application small data.

KeychainManager is an abstraction over generic wrapper [29] performing read/write operations in Keychain, a secure storage to persist application sensitive data, such as current authorization state with latest tokens. Data stored in Keychain are safely persisted even after application uninstall, *AppSettingsManager* is then used to check whether it is a very first run of a particular installation, appropriate action will then be taken: either delete everything on first run or use stored data for token exchange.

UserManager is a wrapper around *AppAuth* and abstracts token exchange and authentication requests. Instance of this class is then provided as a dependency to other application components, as I demonstrate in section **Dependency Injection** [ref].

4.4 API description and interactor

All available networking operations are defined within `ApiDescription` class. To perform API requests a popular HTTP networking library is used called *Alamofire* with community extensions for RxSwift [30–32]. `ApiDescription` class contains definition of all used endpoints and specific return type mapping. Following example shows endpoint for fetching timeline items.

```
func getTimeline(contestCode: String, token: String) -> Observable<[Notification]> {
        return RxAlamofire.requestData(
                            .get,
                            Environment.API.baseURL + "timeline/\(contestCode)",
                            headers: ["Authorization": "Bearer \(token)"]
                )
                .map(to: [Notification].self)
}
```

`ApiInteractor` class is then defined to tie `ApiDescription` with `UserManager` and perform authorized API requests. Following the reactive programming principles we observe a stream of exchanged tokens, which is a single emission, and another stream, based on original token stream, is created to perform the actual HTTP request to server and emit a value of a mapped type.

```
func getTimeline(contestCode: String) -> Observable<[Notification]> {
        return userManager.observeFreshToken()
                .flatMap { token in
                    self.apiDescription.getTimeline(contestCode: code, token: token)
                }
}
```

4.5 Repository Pattern

REST APIs are not always the only data source in mobile applications. A data source can be represented as a local storage which can be used to cache data for offline usage. Despite having single source, I mask data retrieval using Repository pattern[1].

Regardless of having single data source, here I am showing another benefit of using Repository on a very simple example.

Incoming data are not always defined in a way suitable for presenting in View layer, sometimes developer may want to alter incoming objects. Repositories are a perfect place for the job. One of the functional requirements is to display leaderboard types and a specific leaderboard rows. API defines an endpoint for retrieving dynamic list of leaderboard types and en endpoint for retrieving leaderboard rows with points [2]. However, there is also a separate endpoint to fetch team leaderboard rows, which is always available (does not have to be permitted by MyICPC administrator). Does it require us to adopt View layer to access same data is different ways? Well, turns out no. Instead, we use `LeaderboardRepository` to transform data to be later displayed in appropriate way.

```
enum LeaderboardType {
    case generic(Leaderboard)
    case team
}

class LeaderboardRepository: LeaderboardRepositoring {
    private let apiInteractor: ApiInteracting

    init(apiInteractor: ApiInteracting) {
        self.apiInteractor = apiInteractor
    }

    func observeLeaderboards() -> Observable<[LeaderboardType]> {
        return apiInteractor.getLeaderboardTypes()
            .map { $0.map { LeaderboardType.generic($0)} + [LeaderboardType.team] }
    }
}
```

This way, all available leaderboard types are displayed within a single list, no other special view needs to be created to navigate to team leaderboard. For future leaderboard rows loading, again a `LeaderboardType` enum is used to distinguish a requested type and which endpoint to use.

Following the single responsibility principle, each of `Repository` object will be responsible for particular resource type: `Notification`, `Leaderbord`, `Challenge` and `Context`.

4.6 Context Repository

In this section I describe the implementation of application's key feature: context acquisition. This Quest version is designed to fetch device and network data on every user submission. This way, context fetch is triggered after verifying submission is not empty. Network and BLE devices are then scanned for specified time interval.

Fetching different data types asynchronously and combining results may seem like a troublesome task. However, with reactive approach it is a matter of couple lines of code, the important thing is a concept, which I shall demonstrate next for each context data type.

[1] https://martinfowler.com/eaaCatalog/repository.html

4.6.1 App source, Timestamp, Device info, Connected Network Info

These are the easiest context types to retrieve, as they do not require heavy logic to produce the output. For each type I create a stream emitting a single entity.

An example is shown for App source:

```
private func observeAppSource() -> Observable<String> {
    return Observable.just(Environment.Context.appVersion)
}
```

4.6.2 Device location

To obtain device location *CoreLocation* framework is used with open-source reactive extensions on `CLLocationManager` class [33]. A separate `Provider` class is used to allow future system sensor functionality mock in unit tests.

```
class RxLocationProvider: RxLocationProviding {
    private let locationManager = CLLocationManager()

    func observeLocation() -> Observable<Location?> {
        locationManager.requestWhenInUseAuthorization()
        locationManager.startUpdatingLocation()
        return locationManager.rx.location
            .take(1)
            .map { $0?.toLocation() }
            .do(onDispose: { self.locationManager.stopUpdatingLocation() })
    }
}
```

4.6.3 Network devices

MMLanScan library is used to scan local network devices [23]. `RxLanScanner` class abstracts library functionality and provides a stream emitting every found device. An instance of `PublishSubject` is used to act both as an `Observer` and a cold `Observable` (emits only objects available after subscribing to the stream).

```
class RxLanScanner: NSObject, MMLANScannerDelegate, RxLanScanning {
    private let deviceSubject: PublishSubject<MMDevice> = PublishSubject<MMDevice>()
    private var scanner: MMLANScanner!

    override init() {
        super.init()
        scanner = MMLANScanner(delegate: self)
    }

    func lanScanDidFindNewDevice(_ device: MMDevice!) {
        deviceSubject.on(.next(device))
    }

    func lanScanDidFinishScanning(with status: MMLanScannerStatus) {
        deviceSubject.on(.completed)
    }

    func lanScanDidFailedToScan() {
        deviceSubject.on(.error(LanScanError.failedToScan))
    }

    // MARK: RxLanScanning
    func observeLANDevices() -> Observable<MMDevice> {
        return deviceSubject.asObservable()
                .do(onSubscribe: { self.scanner.start() })
                .do(onDispose: { self.scanner.stop() })
    }
}
```

With a stream of found devices, we would now want to satisfy a requirement to scan for certain amount of time. Plus, we need to create a new stream which will emit a *list* of all found devices, all this is easily achieved using reactive operators.

1. all emissions are mapped to defined DTO object accepted by Quest back end.
2. using *take(duration:, scheduler:)* a time interval is specified, as well as scheduler timeout event is to be triggered from. Stream is completed on timeout.
3. all emissions are mapped to a single emission of `LANDevice` array.
4. if `RxLanScanner` emits an error, a stream of single empty array is returned.

```
class RxLANDeviceProviderImpl: RxLANDeviceProvider {
    private let lanScanner: RxLanScanning = RxLanScanner()

    func observeLANDevices(timeInterval: RxTimeInterval) -> Observable<[LANDevice]> {
        return lanScanner.observeLANDevices()
/* 1 */    .map { LANDevice(ipAddress: $0.ipAddress, macAddress: $0.macAddress)}
/* 2 */    .take(
                timeInterval,
                scheduler: ConcurrentDispatchQueueScheduler(qos: .utility)
            )
/* 3 */    .toArray()
/* 4 */    .catchErrorJustReturn([LANDevice]())
    }
}
```

4.6.4 BLE devices

Scan for BLE devices is performed using *CoreBluetooth* framework and an open-source reactive abstraction [25, 34]. Again, strategy is same as for scanning network devices. First, observe the state of `CentralManager` of *CoreBluetooth* framework. Once Bluetooth sensor is powered on, we shall start the actual device scan. Device scan is performed within duration of specified time interval and then transformed into a stream of device array. Passing `nil` as parameter of *scanForPeripherals(withServices:)* indicates that scan should be performed for all supported BLE services. Empty array is returned on error.

```
class RxBLEDeviceProviderImpl: RxBLEDeviceProvider {
    private let centralManager = CentralManager(queue: .global())

    func observeBLEDevices(timeInterval: RxTimeInterval) -> Observable<[BTDevice]> {
        return centralManager.observeState()
            .startWith(centralManager.state)
            .filter { $0 == .poweredOn }
            .timeout(
                timeInterval,
                scheduler: ConcurrentDispatchQueueScheduler(qos: .utility)
            )
            .take(1)
            .flatMap { _ in
                self.centralManager.scanForPeripherals(withServices: nil)
                    .map { BTDevice(name: $0.peripheral.name) }
                    .take(
                        timeInterval,
                        scheduler: ConcurrentDispatchQueueScheduler(qos: .utility)
                    )
                    .toArray()
            }
            .catchErrorJustReturn([BTDevice]())
    }
}
```

4.6.5 Pack it all together

With streams for each individual context item, it is time to provide a mechanism blending all emissions into a `Context` DTO which is then sent to MyICPC. This is again done easily using reactive operator *combineLatest(source:, combiner:)* which combines latest emissions from all supplied streams and produces a stream of entities composed in *combiner* closure.

```
func observeContext() -> Observable<Context?> {
    guard appSettings.contextPermissionGranted else {
        return Observable.just(nil)
    }

    return Observable.combineLatest(
        observeAppSource(),
        observeTimestamp(),
        locationProvider.observeLocation(),
        deviceProvider.observeDevice(),
        networkProvider.observeConnectedNetwork(),
        bleDeviceProvider.observeBLEDevices(timeInterval: Values.bleScanTimeout),
        lanDeviceProvider.observeLANDevices(timeInterval: Values.lanScanTimeout)
    ) { source, timestamp, location , device, connectedNetwork,
        bleDevices, lanDevices in
        Context(
            source: source,
            timestamp: timestamp,
            device: device,
            location: location,
            connectedNetwork: connectedNetwork,
            bleDevices: bleDevices,
            lanDevices: lanDevices
        )
    }
}
```

We are now ready to observe context emissions and provide data to MyICPC. An example of data fetched during development is shown in Table 4.1.

Type	Data
Source	`MyICPC Quest 1.0 dev (iOS)`
Timestamp	2018-04-29 21:11:16 +0000
Device	operatingSystem: `iOS`, version: `11.2.6`, deviceBrand: `Apple`, deviceModel: `Vlad Gorbunov's iPhone`, serial: Optional(`1217A859-0D53-43A1-BBA5-F6C4B7246C6D`)
Location	Optional(timestamp: 2018-04-29 21:11:02 +0000, latitude: 50.053771412977994, longitude: 14.42745745183381)
Connected Network	ssid: Optional(`BOHEMIA`), bssid: Optional(`8:60:6e:5f:6a:44`)
BLE devices	name: Optional(`Vlad's MacBook Pro`) name: Optional(`Vlad's iPad`) name: Optional(`JBL Flip`)
LAN devices	ipAddress: `192.168.1.1`, macAddress: `02:00:00:00:00:00` ipAddress: `192.168.1.22`, macAddress: `02:00:00:00:00:00` ipAddress: `192.168.1.87`, macAddress: `02:00:00:00:00:00`

Table 4.1. Example of possible context submission.

4.7 A closer look at ViewModels with RxSwift

With data sources set up, it is time to tie a ViewController with ViewModel and describe data loading state propagation. For the sake of simplicity, an example is shown for user profile view.

To represent loading operation states a generic enum with associated values is used:

```
enum State<T> {
    case idle
    case empty
    case error(Error)
    case loading
    case loaded(T)
    case reloading
}
```

`ProfileViewModel`, in its simplified form, contains RxSwift `Variable` property producing stream of resource loading states and a computed property indicating whether context acquisition is allowed by user. During initialization step, occurance of stored `User` object is checked in application storage and passed immediately to *userVariable* if present, otherwise requested from MyICPC. Stream of `User` loading operation states is exposed by *observeUser()* method.

```
class ProfileViewModel: ViewModel {
    private let userVariable = Variable(State<User>.idle)
    /* -- omitted private properties -- */

    init(/* -- ViewModel dependencies -- */) {
        /* -- omitted sets -- */
        if let user = userManager.user {
            userVariable.value = .loaded(user)
        } else {
            loadUser()
        }
    }

    var isContextSettingEnabled: Bool {
        get { return appSettings.contextPermissionGranted }
        set { appSettings.contextPermissionGranted = newValue }
    }

    func loadUser() {
        userVariable.value = .loading
        repository.observeUser()
            .subscribeOn(ConcurrentDispatchQueueScheduler(qos: .utility))
            .subscribe { event in
                switch event {
                case .next(let user):
                    self.userManager.user = user
                    self.userVariable.value = .loaded(user)
                case .error(let error):
                    self.userVariable.value = .error(error)
                }
            }
            .disposed(by: disposeBag)
    }

    func logout() {
        userManager.logout()
    }

    func observeUser() -> Observable<State<User>> {
        return userVariable.asObservable()
    }
}
```

ViewModel is provided to ViewController using composition. Once ViewController descendant instance is created, it needs to be passed for presentation. Once passed for presentation, there are several lifecycle events needed to be reacted on. First, a *loadView()* method is called. Here we declare child view components and specify how they should be layouted. Next, we request data, prepare UI bindings and subscribe for data load states, all in *viewDidLoad* method. This method is called only once when ViewController's *view* property is lazily initialized. Once ViewController's *view* is deattached from `UIWindow` container, it and all its properties are deinitialized (unless strong reference to ViewController exists). Reactive stream subscriptions must be disposed to prevent potential memory leaks caused by reference cycles. In case of RxSwift, we provide a reference to a `DisposeBag` which disposes all subscriptions on deinitialization.

```swift
class ProfileViewController: UIViewController {
    // omitted private properties

    init(viewModel: ProfileViewModel) {
        self.viewModel = viewModel
        super.init(nibName: nil, bundle: nil)
    }

    // MARK: lifecycle methods
    override func viewDidLoad() {
        super.viewDidLoad()
        viewModel.observeUser()
            .observeOn(MainScheduler.instance)
            .subscribe(onNext: { state in
                 // display user and load profile pic
            })
            .disposed(by: disposeBag)

        contextOnSwitch.setOn(viewModel.isContextSettingEnabled, animated: false)

        contextOnSwitch.rx.value
            .skip(1)
            .subscribe(onNext: { value in
                self.viewModel.isContextSettingEnabled = value
            })
            .disposed(by: disposeBag)
    }


    // MARK: private methods
    @objc private func didSelectRules() {
        let controller = SFSafariViewController(url:
            URL(string: Environment.External.rulesLink)!)
        navigationController?.present(controller, animated: true)
    }

    // MARK: view definition
    override func loadView() {
        super.loadView()
        navigationItem.title = L10n.Navigation.profile
        // -- omitted view creation code --
    }
}
```

4.8 Assembling pieces, Dependency Injection and Coordinators

4.8.1 Dependency Injection

To prevent singletons and allow easier maintenance I have decided to adopt a concept of Dependency Injection[1]. The DI container in this project is represented by Swinject, a DI framework for Swift [35]. In the below explanation I am using terminology provided by Swinject documentation[2]:

- **Service**: A protocol defining an interface for a dependent type.
- **Component**: An actual type implementing a service.
- **Factory**: A function or closure instantiating a component.
- **Container**: A collection of component instances.

First, we start by registering a *Service*. Service is registered by providing a type of a defining `protocol` and providing protocol conforming implementation inside factory closure.

```
class AppContainer {

    static let container = Container() { container in

        container.register(KeychainManaging.self) { _ in
            KeychainManager()
        }.inObjectScope(.container)

        /* -- other services --*/

        container.register(UserManaging.self) { resolver in
            UserManager(
                keychainManager: resolver.resolve(KeychainManaging.self)!,
                settingsManager: resolver.resolve(AppSettingsManaging.self)!
            )
        }.inObjectScope(.container)

        container.register(ApiServicing.self) { resolver in
            ApiInteractor(
                userManager: resolver.resolve(UserManaging.self)!,
                apiDescription: ApiDescription()
            )
        }.inObjectScope(.container)

        /* -- other services -- */

        container.register(ChallengeListViewModel.self) { r in
           ChallengeListViewModel(repository: r.resolve(ChallengeRepositoring.self)!)
        }
    }
}
```

In the above example, concrete implementation may have its own dependencies, these have to be resolved using a `Resolver` provided by *.register(Service.Type, factory: (Resolver, args..))* method. Swinject then finds registered Services of requested type in dependency graph and provides an instance valid in assigned object scope. By specifying *.inObjectScope(.container)* we are telling Swinject to use a single instance for current container, as well as for child containers.

[1] https://www.martinfowler.com/articles/injection.html
[2] https://github.com/Swinject/Swinject/blob/master/Documentation/DIContainer.md

The tricky part comes when we need to assemble ViewController nested dependencies before presentation. Since I have opted not to use Storyboards for view creation and navigation, all navigation logic is handled manually. In the above example we assume every *Service* is provided using a *Component* – the actual implementation of a service. Suppose we have a ViewController displaying list of items and we want to display item detail by presenting an item detail ViewController. Supplying instance of detail ViewController will not do the trick, as it will require us to use some sort of method injection to specify list item we want to display and, based on lifecycle methods, we would then reload view for the up-to-date data. This is conceptually wrong as MVVM architecture suggests that displayed data should be owned by ViewModel. Instead, we provide a factory for the detail ViewController which will assemble all necessary dependencies managed within Swinject Container. In the below example, a specific challenge is passed as a factory argument and is then used to resolve `ChallengeViewModel` with underlying dependencies.

```
typealias ChallengeViewControllerFactory =
    (_ coordinator: ChallengeCoordinator, _ challenge: Challenge)
    -> ChallengeViewController


container.register(ChallengeViewControllerFactory.self) { r in
    return { coordinator, challenge in
        let controller = ChallengeViewController(
            viewModel: r.resolve(ChallengeViewModel.self, arguments: challenge)!
        )
        controller.navigationDelegate = coordinator
        return controller
    }
}
```

4.8.2 Coordinators

Coordinators are special types of controllers which mask navigation controllers (`UINavigationController`, `UITabBarController`, ...)[1] functionality.

It is an iOS-specific adoption of an *Application Controller*. In *Patterns of Enterprise Application Architecture* it is described as a *"centralized point for handling screen navigation and the flow of application"* [36], page 379. Coordinator's key responsibility is to allow easier navigation flow and perform tasks too general for the scope of particular ViewController. Navigation logic is delegated to responsible Coordinator by Coordinator conformation to protocols supported by ViewControllers, which are being "coordinated". Combination of this pattern with MVVM is often referred to as **MVVM-C**[2], however, with emerging cross-app functionality and adoption of deep linking in mobile applications, Coordinators have become an essential part of a good MVVM implementation.

In my implementation each coordinator is assembled with provided ViewController factories and is backed by supported UIKit navigation controller. Once navigation flow is started, Coordinator is supplied to ViewController before presentation using property injection. Coordinator is then stored as a weak reference to prevent reference cycle.

An example is shown for `AppCoordinator`, responsible for authorization state flow and managing top level `UITabBarController`.

[1] https://developer.apple.com/library/content/documentation/WindowsViews/Conceptual/ViewControllerCatalog/Chapters/NavigationControllers.html
[2] https://tech.trivago.com/2016/08/26/mvvm-c-a-simple-way-to-navigate/

```
// AppCoordinator is started once application finished launching
class AppDelegate: UIResponder, UIApplicationDelegate {

    func application(/* -- omitted params -- */) -> Bool {
        let appCoordinator = AppContainer.container.resolve(
                AppCoordinator.self, argument: UIWindow(frame: UIScreen.main.bounds)
            )!

        appCoordinator.start()
        return true
    }
}

final class AppCoordinator: NSObject {

    init(/* -- provided factories -- */) {
    }

    func start() {
        window.makeKeyAndVisible()
        if userManager.isAuthorized {
            showMain()
        } else {
            window.rootViewController = loginFactory(self)
        }

        userManager.observeAuthChanges()
            // -- subscribe to auth changes to switch main view to login view
            .disposed(by: disposeBag)
    }

    private func showLogin() {
        window.rootViewController = loginFactory(self)
    }

    private func showMain() {
        /*
            init tabBarController with child coordinators
        */
        tabbarController.viewControllers = [timelineCoordinator.rootViewController,
                                            feedCoordinator.rootViewController,
                                            questsCoordinator.rootViewController,
                                            leaderboardCoordinator.rootViewController]

        window.rootViewController = tabbarController

        // check if context dialog was shown
        if !appSettings.contextPermissionRequested {
            // show dialog asking to send context for research
        }
    }
}

// confirm to protocol accepted by LoginViewController
extension AppCoordinator: LoginNavigationDelegate {

    func didSelectLogin() {
        // perform auth request
    }
}
```

4.9 Notifications and Challenges

Notifications list is implemented using `UITableViewController` with support for different `UITableViewCell` types: `NotificationCell` and `SystemNotificationCell`. It is a generic view used with several display modes:

- **timeline** – all notifications produced by users and system notifications of MyICPC;
- **whatsHappeningNow** – latest Quest submissions;
- **accepted(Challenge)** – accepted Quest submissions for particular challenge;
- **pending(Challenge)** – pending submissions for specified challenge;
- **rejected(Challenge)** – list of rejected Quest submissions for challenge.

An important thing here worth pointing out is that both `NotificationCell` and `SystemNotificationCell` have a variable height due to multiline text, image and video presentation. This may result in a UI glitch on scroll, unless correctly implemented.

To prevent layouting problems, a property of `UITableView` indicating variable row height has to be set with accurate estimate.

```
tableView.rowHeight = UITableViewAutomaticDimension
// estimate is computed using defined cell insets and constraints
tableView.estimatedRowHeight = 300
```

UIKit does not provide an automatic mechanism for hiding views and switching off its constraints. This behavior is only available with `UIStackView`. `UIStackView` computes child view positioning attributes during the actual drawing, thus it is not applicable in our case. Instead, we need to switch constrains on and off and hide redundant views manually. This is where AutoLayout DSL comes into play. With SnapKit we can easily remake existing constraints to adopt a cell for specific data.

Each cell instance is a reusable view object contained withing the *object pool* it was registered in. To minimize possible glitch and achieve maximum efficiency I am registering each possible cell type data configuration with its own object pool.

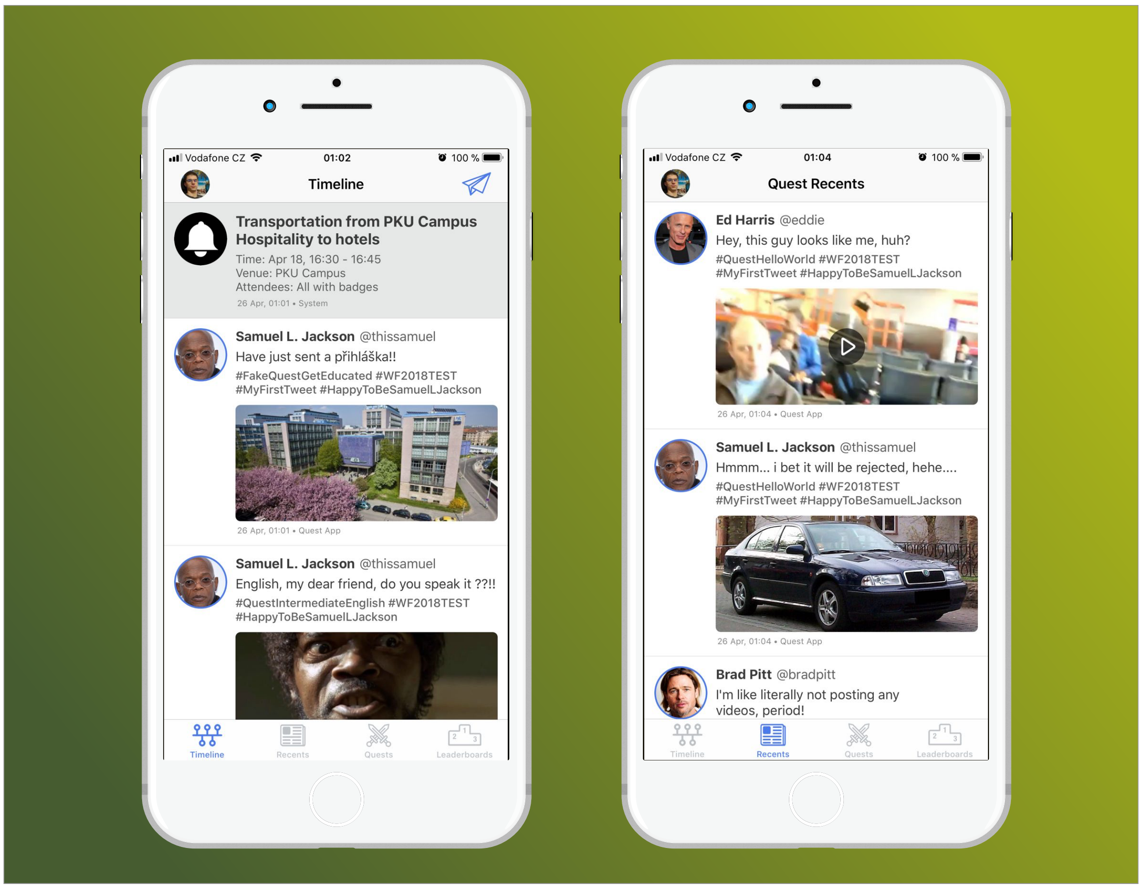
```
// register a cell configuration in a specific object pool
tableView.register(NotificationCell.self,
        forCellReuseIdentifier: NotificationCell.Identifiers.withText)

// deque cell later using identifier
tableView.dequeueReusableCell(withIdentifier: NotificationCell.Identifiers.withText,
        for: indexPath)
```

Notification image and video thumbnails are expanded on click for presentation using `SKPhotoBrowser` [37] and `AVPLayerViewController` from standard *AVFoundation* framework.

Same rules to achieve smooth scrolling are applied in challenge list view.

Final design of challenge and notification lists can be seen on Figures 4.2 4.3.

Application also comes with user-friendly error and empty states as depicted on Figure 4.4



Transportation from PKU Campus Hospitality to hotels
Time: Apr 18, 16:30 - 16:45
Venue: PKU Campus
Attendees: All with badges
26 Apr, 01:01 • System

Samuel L. Jackson @thissamuel
Have just sent a přihláška!!
#FakeQuestGetEducated #WF2018TEST
#MyFirstTweet #HappyToBeSamuelLJackson
26 Apr, 01:01 • Quest App

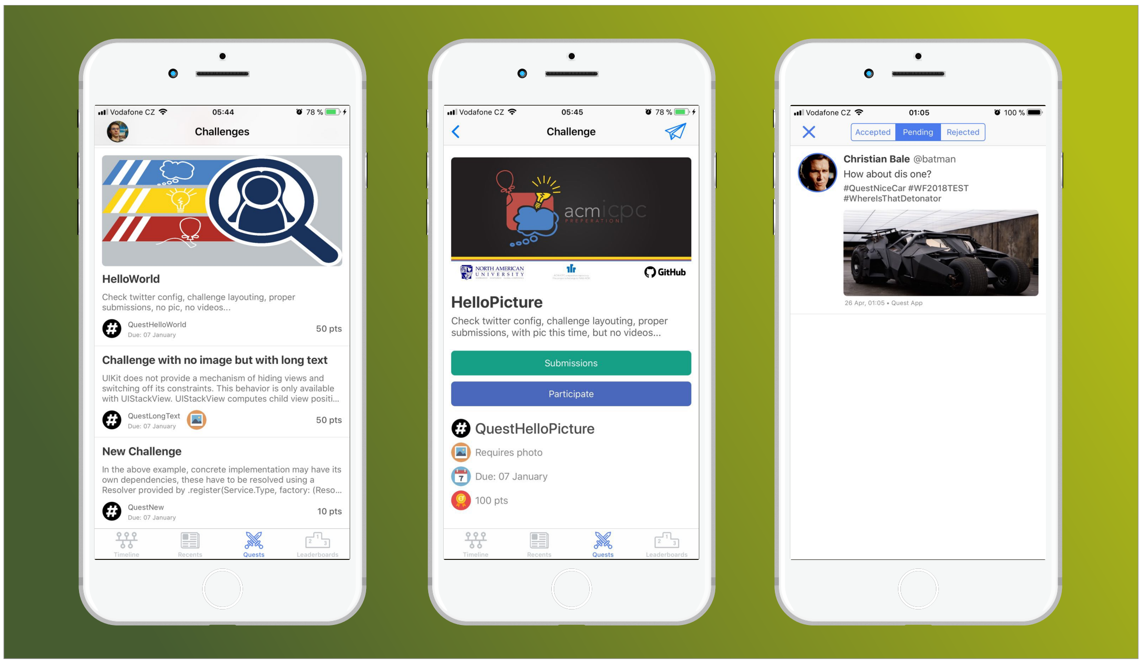
Samuel L. Jackson @thissamuel
English, my dear friend, do you speak it ??!!
#QuestIntermediateEnglish #WF2018TEST
#HappyToBeSamuelLJackson

Ed Harris @eddie
Hey, this guy looks like me, huh?
#QuestHelloWorld #WF2018TEST
#MyFirstTweet #HappyToBeSamuelLJackson
26 Apr, 01:04 • Quest App

Samuel L. Jackson @thissamuel
Hmmm... i bet it will be rejected, hehe....
#QuestHelloWorld #WF2018TEST
#MyFirstTweet #HappyToBeSamuelLJackson
26 Apr, 01:04 • Quest App

Brad Pitt @bradpitt
I'm like literally not posting any videos, period!

Figure 4.2. Final design of notification list.



HelloWorld
Check twitter config, challenge layouting, proper submissions, no pic, no videos...
QuestHelloWorld
Due: 07 January
50 pts

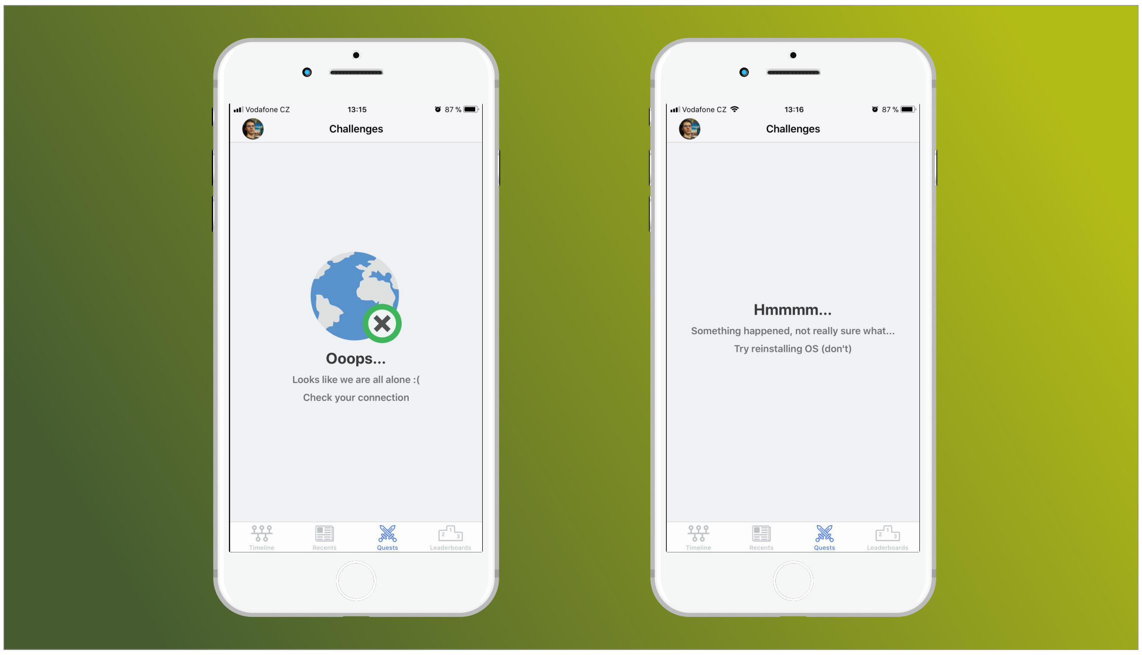
Challenge with no image but with long text
UIKit does not provide a mechanism of hiding views and switching off its constraints. This behavior is only available with UIStackView. UIStackView computes child view positi...
QuestLongText
Due: 07 January
50 pts

New Challenge
In the above example, concrete implementation may have its own dependencies, these have to be resolved using a Resolver provided by .register(Service.Type, factory: (Reso...
QuestNew
Due: 07 January
10 pts

HelloPicture
Check twitter config, challenge layouting, proper submissions, with pic this time, but no videos...
QuestHelloPicture
Requires photo
Due: 07 January
100 pts

Christian Bale @batman
How about dis one?
#QuestNiceCar #WF2018TEST
#WhereIsThatDetonator
26 Apr, 01:05 • Quest App

Figure 4.3. Final design of challenge list and challenge detail.


Vodafone CZ 13:15 87 %
Challenges
Ooops...
Looks like we are all alone :(
Check your connection

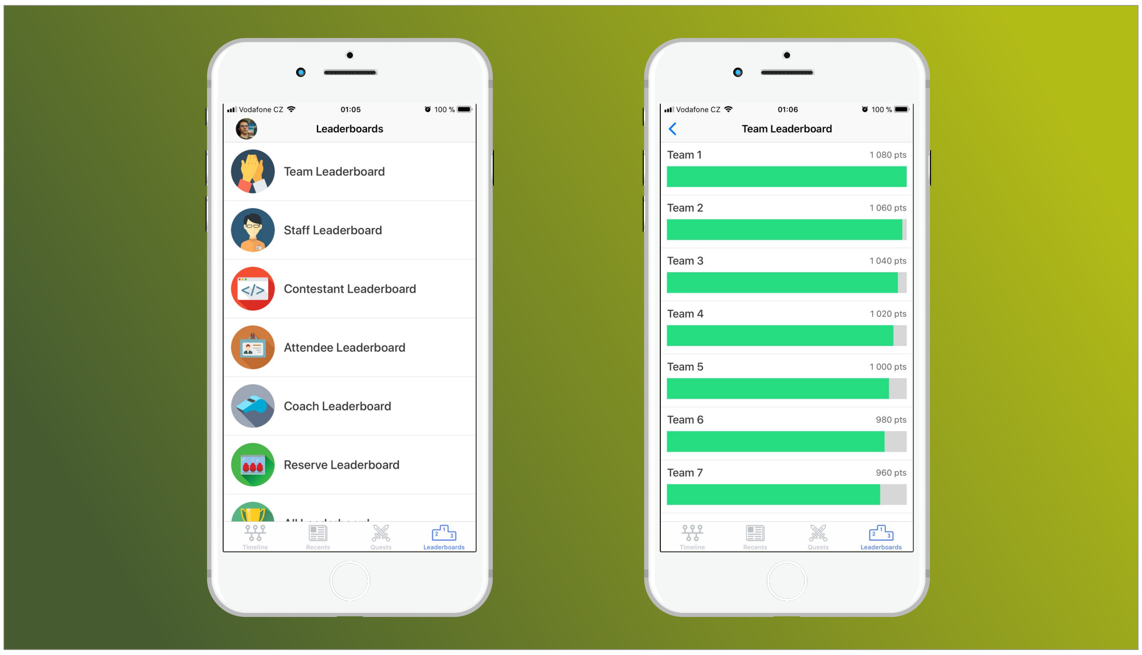
Vodafone CZ 13:16 87 %
Challenges
Hmmmm...
Something happened, not really sure what...
Try reinstalling OS (don't)

Figure 4.4. Error state mapping.

4.10 Leaderboards

Original `LeaderboardRowResponse` entity comes from API with 2 properties: name and points. To make an overall leaderboard appearance more attractive, a maximum of provided row points is found and is used to transform original row list into new `LeaderboardRow` entities containing additional *maxPoints* field. This property is used later to display a horizontal bar chart indicating current progress using Charts library [38].

Final leaderboards design can be seen on Figure 4.5.


Vodafone CZ 01:05 100 %
Leaderboards
Team Leaderboard
Staff Leaderboard
Contestant Leaderboard
Attendee Leaderboard
Coach Leaderboard
Reserve Leaderboard

Vodafone CZ 01:06 100 %
Team Leaderboard

Team 1	1 080 pts
Team 2	1 060 pts
Team 3	1 040 pts
Team 4	1 020 pts
Team 5	1 000 pts
Team 6	980 pts
Team 7	960 pts

Figure 4.5. Final design of leaderboard list.

4.11 Input view

Input view is a crucial part of application. As in every other app which is aimed at data retrieval, the better input method we provide, the better input data we get. In my case input view is presented as an `UITextView` for multiline text and a `UICollectionView` for horizontal collection of picked images and videos. `InputViewController` is designed to support any number of picked media files and in this version is explicitly restricted to exactly one of each media file type. There I also observe a keyboard opening event using KVO to adjust positioning of bottom bar from which image and video picker is accessed. Media picker used in this project is an open-source library called *Gallery* [39].

Interesting part here is also how user inputs are observed to enable/disable "Send" button using reactive extensions which come with *RxCocoa* (provided as part of RxSwift).

```
let isTextPresent = viewModel.message.asObservable()
    .map { !$0.isEmpty && self.messageTextView.textColor != self.textViewHintColor }
let isMediaPresent = viewModel.observeMedia()
    .map { !$0.isEmpty }
let isLoading = viewModel.observeInputState()
    .map { $0 == .uploading }
let isPostEnabled = Observable.combineLatest(isTextPresent,
                                            isMediaPresent,
                                            isLoading) {
    ($0 || $1) && !$2
}

isPostEnabled.bind(to: navigationItem.rightBarButtonItem!.rx.isEnabled)
    .disposed(by: disposeBag)
```

Same strategy is applied to observe media files submission limits and enable/disable media picker buttons.

Final input view design can be seen on Figure 4.6.

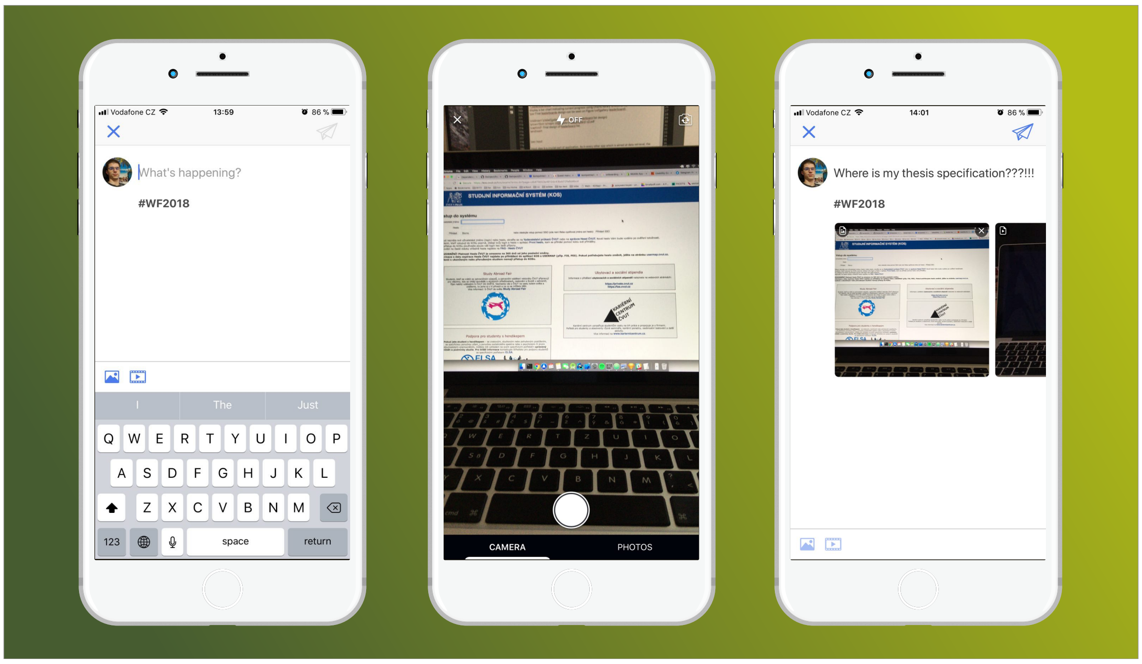


Figure 4.6. Final input view design.

4.12 Other views

Here I present some views which are not in scope of previous sections.

There is nothing particularly interesting that would be worth pointing out. Final designs are presented for completeness and can be seen on Figure 4.7.

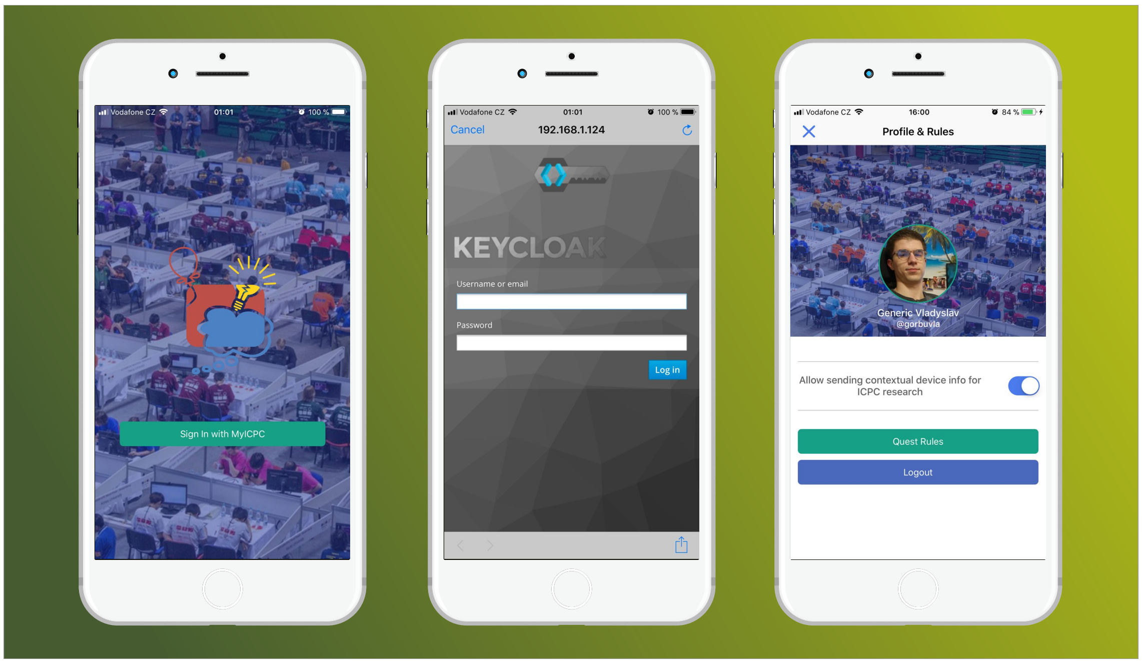


Figure 4.7. Final design of Login and Profile views.

Chapter 5
Testing

5.1 User Testing

5.1.1 Test progress

Apart from testing on different virtual devices to ensure application views are displayed correctly on all available screen resolutions, I have conducted a simplified user testing. User testing is particularly useful when it comes to mobile applications, particularly in situations when a developer is not within an app target group. A user target group and application use conditions dramatically affect interface design. That is why tested participants and conditions should be selected accordingly to provide relevant feedback.

Participants were selected to achieve best possible dispersion in user age and user familiarity with mobile devices. Task had to be performed on 2 devices: iPhone SE and iPhone 6s Plus. Below I present the table of key information about participants as well as received positive and negative feedback 5.1. Test task was performed with local MyICPC installation using fake data. Test assignment was defined as:

1. Sign in using provided credentials.
2. Open Quest rules.
3. In a timeline, find a submission with attached image.
4. Select same challenge in challenges tab.
5. Try to find out if there is some submission rejected for this challenge.
6. Post a submission for this challenge with attached image.
7. Check if your submission was posted.
8. Find your position in leaderboard, given that your role is Staff.

None of the participants struggled when executing the test task, all 3, however, pointed out inconvenient rules link placement in profile screen. Optionally, participants had a chance to explore application as they wished and give a feedback on overall appearance.

5.1.2 Evaluation

It can be seen that some negative points occur with several participants. While most of the negative feedback was resolved, there are still points missing due to lack of such functionality in current version of REST API.

Resolved points:

- Profile tab is removed from tab bar component, instead a button is placed in navigation bar.
- Submission usernames and names are inlined horizontally with profile image (original implementation displayed user profile image with name in a vertical stack with adaptive font size, thus long usernames could not be seen).
- Custom icons for video playback and media file types with adjusted background transparency were created.

Participants	Positives	Negatives
Participant 1 Age: 29 Work: Project Manager Experience: High	1. Nice, intuitive input design. 2. Nice challenge list. 3. User friendly error states.	1. Redundant 5th profile tab, use button in navbar. 2. Missing notification detail on tap. 3. Missing other user profiles. 4. Missing user management.
Participant 2 Age: 24 Work: Node.js Developer Experience: Middle	1. Nice error display. 2. Good input view. 3. Presence of reload.	1. Can't see long usernames 2. Can't see all my submissions 3. Can't delete a submission 4. Too big profile image on iPhone SE in notification list.
Participant 3 Age: 21 Work: FEE student Experience: Low	1. Nice icons. 2. Nice reload gesture. 3. Nice input view.	1. Bad video playback icon. 2. Can't click on notification to see detail. 3. Can't view other user profiles.

Table 5.1. User Testing. Participants feedback.

5.2 Unit tests

Unit tests in iOS projects play crucial role in potential bug reveal as well as in any other area of software development. This is a relatively small project and provided unit tests have more of informative character.

Unit tests for Xcode projects are defined using XCTest framework. To run the tests an individual build target needs to be created and all tests and test dependency classes need to be specified in compile sources.

A basic test is defined using *XCTestCase* class.

```
class ExampleTest: XCTestCase {
    override func setUp() {
        super.setUp()
        // Method is called before the invocation of each test method.
    }

    override class func tearDown() {
        // Method is called before the invocation of each test method.
        super.tearDown()
    }

    func exampleUnitTest() {
        XCTAssertEqual(3 + 1, 5, "Your math skills are not so great")
    }
}
```

There are several ways how to test asynchronous code. Given that majority of classes from Model tier return `RxSwift Observable<T>` types, I will adopt the simplest approach and convert asynchronous calls to blocking ones using operators supplied with `RxBlocking`. An example is shown for `LeaderboardRepository` tests.

```
class LeaderboardRepositoryTest: XCTestCase {
    var repository: LeaderboardRepositoring!

    // prepare
    override class func setUp() {
        super.setUp()
        let api = MockedApiService()
        repository = LeaderboardRepository(apiService: api)
    }

    // Expected: original types + team leaderboard
    func testLeaderboardTypes() {
        do {
            let types = try repository.observeLeaderboards()
                    .toBlocking().first()!!

            XCTAssertTrue {
                types.count == 3
                &&
                types.contains { $0 == LeaderboardType.team }
            }
        } catch {
            XCTFail(error.localizedDescription)
        }
    }
}
```

Chapter 6
Conclusion

Application prototype was successfully implemented. While current version of MyICPC Quest API was fully adopted, some of the contextual device and network data is missing. With an option to use private API, a preference was given to leave out a portion of data to prevent API misuse problems when distributing app in the future.

During the implementation phase I was trying to give as much attention to application architecture as possible. Some parts were completely rewritten several times. A non-standardized architectural approach made the resulting implementation easily adoptable for future enhancements. User tests have proven to be useful, even though none of the participants was involved in ICPC.

Applying reactive programming principles was also a challenge, since I did not have much of FRP background. However, I am sure that FRP have made a lot things easier, especially context data fetch.

As for future enhancements, I would like to point out some that seem necessary:

- extend REST API to provide more functionality;
- use onboarding[1] to present app functionality and usage instead of displaying a PDF file with Quest rules;
- make use of continuous delivery tools for easier development cycle, especially when single installation is provided for specific ICPC contest;
- make use of service like HockeyApp[2] which will allow pre-release app distribution for main MyICPC team;
- release functional version on AppStore.

[1] https://usabilitygeek.com/mobile-user-onboarding-examples/
[2] https://hockeyapp.net

References

[1] Roman Smetana. *Next Generation of Second-Screen - Realtime application MyICPC*. 2016.
https://dspace.cvut.cz/handle/10467/62711.

[2] Filip Ryšavý. *MyICPC 2.0 with API for Android App* .
https://bitbucket.org/frysavy/myicpc-2.0/src/myicpc-quest-app-backend/.

[3] Filip Ryšavý. *MyICPC Quest Android App* .
https://bitbucket.org/frysavy/myicpc-android-app/src/master/.

[4] Jon Hoffman. *Masering Swift 4 - Fourth Edition*. Packt Publishing, Ltd., 2017. ISBN 9781788477802.

[5] Apple Inc. *The Swift Programming Language (Swift 4.1)*. Apple Inc., 2018.
https://itunes.apple.com/us/book/the-swift-programming-language/id881256329?mt=11.

[6] *CocoaPods. The Cocoa Dependency Manager*.
https://cocoapods.org/.

[7] *Carthage by Github. A simple, decentralized dependency manager for Cocoa*.
https://github.com/Carthage/Carthage.

[8] *Model-View-Controller, Apple Developer documentation*.
https://developer.apple.com/library/content/documentation/General/Conceptual/CocoaEncyclopedia/Model-View-Controller/Model-View-Controller.html.

[9] Soroush Khanlou. *Massive View Controller*.
http://khanlou.com/2015/12/massive-view-controller/.

[10] *Apple Developer documentation, Prioritize Work with Quality of Service Classes*.
https://developer.apple.com/library/content/documentation/Performance/Conceptual/EnergyGuide-iOS/PrioritizeWorkWithQoS.html.

[11] *ReactiveX, Reactive Programming in Swift*.
https://github.com/ReactiveX/RxSwift.

[12] Navdeep Singh. *Reactive Programming with Swift 4* . Packt Publishing, Ltd., 2018. ISBN 9781787128781.

[13] *SnapKit, A Swift Autolayout DSL for iOS and OS X*.
http://snapkit.io/.

[14] *AppAuth. Native App SDK for OAuth 2.0 and OpenID Connect implementing modern best practices*.
https://appauth.io/.

[15] *OAuth 2.0 for Native Apps, RFC 8252*.
https://tools.ietf.org/html/rfc8252.

[16] Cong Shi, Jian Liu, Hongbo Liu, and Yingying Chen. Smart User Authentication through Actuation of Daily Activities Leveraging WiFi-enabled IoT.

[17] Ioannis Agadakos, Per Hallgren, Dimitrios Damopoulos, Andrei Sabelfeld, and Georgios Portokalidis. Location-enhanced authentication using the IoT: because you cannot be in two places at once.

[18] Claudio Marforio, Nikolaos Karapanos, Claudio Soriente, Kari Kostiainen, and Srdjan Capkun. Smartphones as Practical and Secure Location Verification Tokens for Payments.

[19] *UIDevice, Apple Developer documentation.* `https://developer.apple.com/documentation/uikit/uidevice`.

[20] *Core Location, Apple Developer documentation.* `https://developer.apple.com/documentation/corelocation`.

[21] *SystemConfiguration, Apple Developer deocumentation.* `https://developer.apple.com/documentation/systemconfiguration`.

[22] *NetworkExtension, Apple Developer documentation.* `https://developer.apple.com/documentation/networkextension`.

[23] *MMLanScan by Michael Mavris, An iOS LAN Network Scanner library.* `https://github.com/mavris/MMLanScan`.

[24] *The Mac Observer, Thanks to Misuse, Apps Can't View MAC Addresses on iOS 11.* `https://www.macobserver.com/news/product-news/apps-cant-view-mac-addresses-on-ios-11/`.

[25] *Core Bluetooth, Apple Developer documentation.* `https://developer.apple.com/documentation/corebluetooth`.

[26] Vladyslav Gorbunov. *MyICPC Quest iOS.* `https://bitbucket.org/kompotmalinovy/myicpc-quest-ios/`.

[27] *SwiftGen, The Swift code generator for your assets, storyboards, Localizable.strings,...* `https://github.com/SwiftGen/SwiftGen`.

[28] *Tab Bars, Human Interface Design, Apple Developer Documentation.* `https://developer.apple.com/ios/human-interface-guidelines/bars/tab-bars/`.

[29] *Locksmith, A powerful, protocol-oriented library for working with the keychain in Swift.* `https://github.com/matthewpalmer/Locksmith`.

[30] *Alamofire. Elegant HTTP Networking in Swift.* `https://github.com/Alamofire/Alamofire`.

[31] *AlamofireImage. An image component library for Alamofire.* `https://github.com/Alamofire/AlamofireImage`.

[32] *RxAlamofire. RxSwift wrapper around the elegant HTTP networking in Swift Alamofire.* `https://github.com/RxSwiftCommunity/RxAlamofire`.

[33] *RxCoreLocation, Reactive abstraction to manage Core Location.* `https://github.com/RxSwiftCommunity/RxCoreLocation`.

[34] *RxBluetoothKit, iOS and OSX Bluetooth library for RxSwift.* `https://github.com/Polidea/RxBluetoothKit`.

[35] *Swinject, Dependency injection framework for Swift with iOS/macOS/Linux.* `https://github.com/Swinject/Swinject`.

[36] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002. ISBN 0321127420.
https://www.martinfowler.com/books/eaa.html.

[37] *SKPhotoBrowser. Simple PhotoBrowser/Viewer inspired by facebook, twitter photo browsers written by swift.*
https://github.com/suzuki-0000/SKPhotoBrowser.

[38] *Charts. Beautiful charts for iOS/tvOS/OSX! The Apple side of the crossplatform MPAndroidChart.*
https://github.com/danielgindi/Charts.

[39] *Gallery. Your next favorite image and video picker.*
https://github.com/hyperoslo/Gallery.

Appendix A
Abbreviations

ICPC	International Collegiate Programming Contest
API	Application Programming Interface
IDE	Integrated Development Environment
DSL	Domain Specific Language
SDK	Software Development Kit
REST	Representational State Transfer
FRP	Functional Reactive Programming
MVC	Model-View-Controller
MVVM	Model-View-ViewModel
DTO	Data Transfer Object
KVO	Key-Value-Observable
UI	User Interface
GCD	Grand Central Dispatch
QoS	Quality of Service
FRP	Functional Reactive Programming
XML	Extensive Markup Language
SSID	Service Set Identifier
BSSID	Basic Service Set Identifier
RSSI	Received Signal Strength Indication
IP	Internet Protocol
MAC	Media Access Control
ARP	Address Resolution Protocol
UUID	Universally Unique Identifier
HTTP	Hypertext Transfer Protocol
BLE	Bluetooth Low Energy
URL	Uniform Resource Locator
DI	Dependency Injection

Appendix B
CD contents

root
/doc – TeX sources
/src – Project sources
/vids – Demo videos
/sketch – Sketch file with application icons and graphics used in thesis