FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF BACHELOR'S THESIS

Title:	Mobile application Classification for iOS
Student:	Jiří Zdvomka
Supervisor:	Ing. Jakub Průša
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of summer semester 2019/20

Instructions

The purpose of this bachelor thesis is the implementation of Grades iOS mobile app for students and teachers on FIT CTU. Students can view grades for subjects they study and receive notifications about new grades. Teachers can manage the grades of students. The application will be available for download on the AppStore for free.

1. Read the documentation for existing Grades API and analyse it.

2. Analyse UI/UX of the existing Android app and map it to iOS design patterns.

3. Extend the notification system to support sending notifications to iOS platform.

4. Implement the application.

5. Test the application.

References

Will be provided by the supervisor.

Ing. Michal Valenta, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague November 7, 2018

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Mobile application Classification for iOS

Jiří Zdvomka

Department of Software Engineering

Supervisor: Ing. Jakub Průša

May 13, 2019

Acknowledgements

I would like to thank my supervisor Ing. Jakub Průša and his developer team in Quanti for consulting my work and sharing valuable advice. I am also grateful to Ing. Štěpán Plachý for helping me with the integration of the application with the university's services. Special thanks to my girlfriend Simona Pohlová and my family for overall support.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on May 13, 2019 .....

Czech Technical University in Prague

Faculty of Information Technology

Citation of this thesis

Zdvomka, Jiří. *Mobile application Classification for iOS*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2019.

Abstrakt

V této bakalářské práci se zabývám implementací mobilní aplikace pro platformu iOS, která komunikuje s existujícím klasifikačním systémem Grades Fakulty informačních technologií Českého vysokého učení technického v Praze. Studenti můžou v aplikaci prohlížet svojí klasifikaci a dostávat upozornění o změnách klasifikace. Učitelé můžou v aplikaci spravovat klasifikaci svých studentů. V práci jsem postupoval podle procesu softwarového inženýrství a vytvořil analýzu, navrhl řešení, implementoval a otestoval aplikaci. Výsledkem je funkční mobilní aplikace, která studentům a učitelům univerzity usnadní kalsifikační proces.

Klíčová slova mobilní aplikace, iOS, Swift, klasifikace, vývoj software, push notifikace

Abstract

This bachelor's thesis aims to implement a mobile application for iOS platform communicating with existing student evaluation system Grades at Faculty of Information Technologies of the Czech Technical University in Prague. The application allows students to view their evaluation and receive push notifications about new changes in their evaluation. Teachers can manage an evaluation of the students they teach. I proceeded by the software engineering process and created an analysis, designed a solution, implemented and tested the application. The result is working mobile application that makes the evaluation process for students and teachers at the university faster and more convenient.

Keywords mobile application, iOS, Swift, student evaluation, software development, push notifications

Contents

List of Figures

List of Tables

List of Listings

Introduction

Today's trend is migrating software solutions from the web to native mobile solutions as mobile platforms allow faster and easier interaction with a user. At the time of writing this work, there is a student evaluation web and Android application at Faculty of Information Technologies at Czech Technical University (FIT CTU) in Prague, but no mobile application for iOS platform.

My motivation is to simplify access to student evaluation at FIT CTU and allow teachers of the university to manage their students' evaluation in more convenient way.

In this bachelor thesis, I design, implement and test mobile iOS application Grades where students can view their grades of courses they study, receive notifications and teachers can manage the evaluation for subjects they teach either for a particular student or all students taking a test or doing a project. The application will communicate with the existing web application Grades.

In the first chapter, I research theoretical background and technologies required for creating the application – development, architecture and programming paradigm for iOS platform along with other patterns used in mobile development. In the second chapter, I do an analysis of existing solutions and services suitable for integration and define the application's requirements. In the following chapters, I apply previous principles and technologies and describe the software engineering process of the application - architecture, design, implementation and testing.

Even though the application is called Classification in the title of this thesis, in the remaining text the term Grades is used instead, because Grades was chosen by faculty management as the official name of the application.

Thesis aim

The primary aim of this bachelor thesis is to implement Grades iOS mobile application for students and teachers of FIT CTU in Prague integrated with existing web application service. Students can view detailed evaluation of courses and receive notifications about changes in it. Teachers can manage grades of their students in multiple ways.

In the research part, the aim is to summarize specifics of development for iOS platform and other technologies related to mobile application development. The second goal is to do an analysis of existing solutions - web application, Android application and notification server.

In practical part objectives are designing application requirements and user interface, compare user interface with existing Grades Android application, choose software architecture, extend the notification server, implement and test the application and describe release process.

Chapter 1

Technologies & solutions

In this chapter I describe the theoretical background, possible solutions and technologies used later in this work - specifics of development for iOS platform, functional reactive programming paradigm, REST API and it's usage in mobile development and OAuth 2.0 authentication protocol.

1.1 Development for iOS platform

In this section, I characterize essential parts of development for iOS. First, I describe programming languages and development tools of the platform. Then continue with two major architectural patterns *Model View Controller* (MVC) and *Model View ViewModel* (MVVM), following with explanation how iOS apps manage the view.

1.1.1 Swift and XCode

Primary programming language for iOS and other Apple platforms used to be Objective-C until June 2014, when the first version of Swift programming language was released [1] and since then it has been the first choice for most developers as Apple encourages them to use it. These two programming languages are compatible so both Objective-C and Swift can be used in an iOS project.

Swift is object-oriented strongly typed programming language developed under open source license. Some of the key features of it are:

- named parameters,

- optionals,
- memory management with Automatic Reference Counting (ACC),
- removal of unnecessary semicolon symbols and parentheses in expressions (not at all cases),
- low-level primitives like types, flow-control and operators,
- object-oriented features like classes, protocols and generics [2].

XCode is an integrated development environment (IDE) for Apple platforms and Swift language. It provides a full set of functionalities for development including code completion, debugging, testing or device simulators. It is possible to even run the app directly from XCode on a physical device connected via USB cable or network. The IDE supports building and running apps for diverse environments like development, staging or production.

1.1.2 UIKit

Part of the Swift language is `UIKit` framework, which is a set of classes providing window and view architecture for implementing a user interface in iOS applications. It also comes with handling events like touch or other types of input, animation support or drawing [3]. Below I describe critical parts of the framework.

1.1.2.1 ViewControllers

`ViewController` is a class for managing a user interface and navigation of an application. It contains a single root view which can itself contain other subviews [4]. Every app must have at least one `ViewController`.

There are four main types of ViewControllers:

- `UIViewController` – basic and most used ViewController managing a view hierarchy of the application [5]. One view controller often manages one app's screen.
- `UITableViewController` – "*A view controller that specializes in managing a table view*" [6].

- `UINavigationController` – "*A container view controller that defines a stack-based scheme for navigating hierarchical content*" [7].
- `UITabBarController` – a view that displays bar with containing subviews at the bottom of a screen allowing navigation between them [8].

1.1.2.2 Views and Controls

For presenting visual content on a screen, there are Views. For defining interaction, there are Controls. Below I list a couple of these essential classes [9].

- `UINavigationBar` – A class displaying navigation bar at the bottom of a screen, implementing underlying screen transition logic. Often used in conjunction with `UINavigationController` [10]. It contains subviews for a title text, back button and the right accessory item (e.g. submit button). It can be extended with other subviews like search a text field or easily configured to be larger with a bigger title.

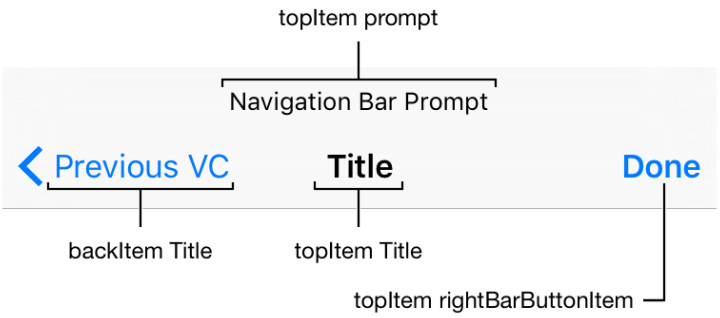


Figure 1.1: Layout of items in UINavigationBar [10]

- `UIView` – Fundamental View class managing content for a rectangular area on a screen [11].
- `UIStackView` – Handy View automatically managing autolayout and spacing of its arranged subviews, laying them out in either row or column [12].
- `UILabel` – View displaying a text often describing the purpose of other controls [14].
- `UITextField` – A Control, described at official Apple documentation as *"An object that displays editable text area in your interface"* [15].

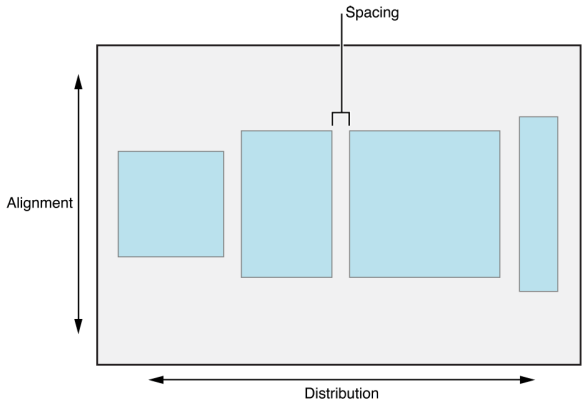


Figure 1.2: UIStackView subitems layout [13]

1.1.2.3 Building a user interface

There are two ways for building a user interface from `UIKit` components on the iOS platform - *Interface Builder* and programmatically. *Interface Builder* allows a developer to design UI without any code by dragging and dropping UIKit components from a list in the tool [16]. Programmatic approach involves creating components and setting their properties and layout in code.

1.1.3 Architecture

1.1.3.1 Model View Controller

Apple promoted architecture style on the iOS is *Model-View-Controller* (MVC). The architecture has three layers for handling different application's logic. The *Model* reads and writes data to persist application state. The *View* displays data and sends input to the Controller. Finally, the *Controller* plays a central role and is responsible for updating both model and view [17].

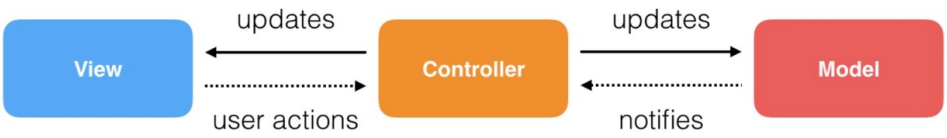


Figure 1.3: MVC architectural style diagram [17]

1.1.3.2 Model View ViewModel

Model-View-ViewModel (MVVM) is a newer pattern similar to MVC. Instead of the *Controller* layer, there is the *ViewModel* which takes care of business logic, talks to the *Model* and exposes changes to the View.
ViewController classes (the *Controller* layer in MVC) stays, but they are part of the *View* layer and their only responsibility is binding of user interface and input [17].

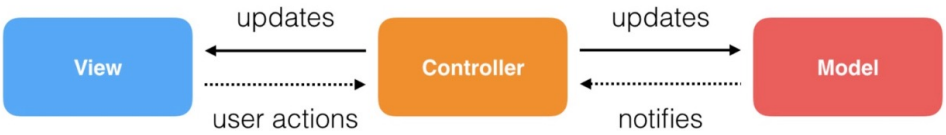


Figure 1.4: MVVM architectural style diagram [17]

1.2 Reactive programming

Mobile applications do not know their input in advance, thus they need to react to changes over time. Reactive programming paradigm solves this problem with asynchronous data streams and propagation of change to many parts of a software. This approach can be extended with functional paradigm (e.g. usage of filter functions like map, filter, reduce or avoidance of mutating data). That is called functional reactive programming (FRP) [18].

In Swift language, there are two main reactive libraries implementing the above principle - *RxSwift* and *ReactiveSwift*. I am going to compare these two libraries later in chapter about architecture.

1.2.1 RxSwift

In [19] the definition of RxSwift is following: "*RxSwift, in its essence, simplifies developing asynchronous programs by allowing your code to react to new data and process it in a sequential, isolated manner.*"

This library is Swift implementation of ReactiveX[1] API which is multiplatform FRP standard. It calls data streams observable sequences and parts of an application, consuming these sequences, observers [20].

[1] `http://reactivex.io`

1.2.2 ReactiveSwift

ReactiveSwift is Swift-only implementation of FRP and provides composable, flexible and declarative primitives over this concept. Data streams are called signals in the library [21].

1.3 REST API

Representational state transfer (REST) is communication standard and architectural style in software development. In relation with mobile or web development, REST or RESTful API is defined as an application programming interface based on REST standard which uses HTTP protocol for communication [22].

1.3.1 Related terms

Below I list terms used in this work related to REST API.

- *Endpoint* – URL address of REST API on which the interface accepts HTTP requests.
- *Controller* – In the context of REST API, it is a set of endpoints unifying operations over an entity.
- *CRUD* – create, read, update and delete operations over a data entity.

1.4 OAuth 2.0

OAuth 2.0 is standard authentication protocol which replaces original *OAuth* protocol. Its main improvement is easy implementation for client applications (web, desktop or mobile). It gives a client application access to protected user data from a web service without authorizing the user in the application and entering his secured credentials [23].

1.4.1 Terms

- *Protected resource* - secured private data.
- *Access token* – string with limited validity used for accessing a protected resource.

- *Refresh token* – string used for obtaining a new valid access token in case of expiration of a current one.
- *Authorization grant* – credential representing the resource owner's authorization, there are a few grant types – *Authorization Code, Implicit, Resource Owner Password Credentials* and *Client Credentials* [23].

1.4.2 Roles

- *Resource owner* – entity, usually end user, which is an owner of a protected resource.
- *Resource server* – provider and host of a protected resource, usually a web service.
- *Client* – an application requesting a protected resource with authorization of a resource owner.
- *Authorization server* – a server providing an access token to a client after successful authorization of a resource owner [23].

1.4.3 Authentication process

The authentication process for Authorization Code grant type is following:

1. A client requests authorization from a resource owner.
2. The client obtains authorization grant from the resource owner.
3. The client requests an access token from an authorization server with a permission of the resource owner.
4. The authorization server authenticates the client, validates the authorization grant and issues an access token.
5. The client requests resource owner's protected resource from the resource server signed with the obtained access token.
6. The resource server validates the access token and handles the request in case of a valid access token. If the access token is not valid, the server responds with an error [23].

The authentication flow is illustrated in the diagram 1.5.

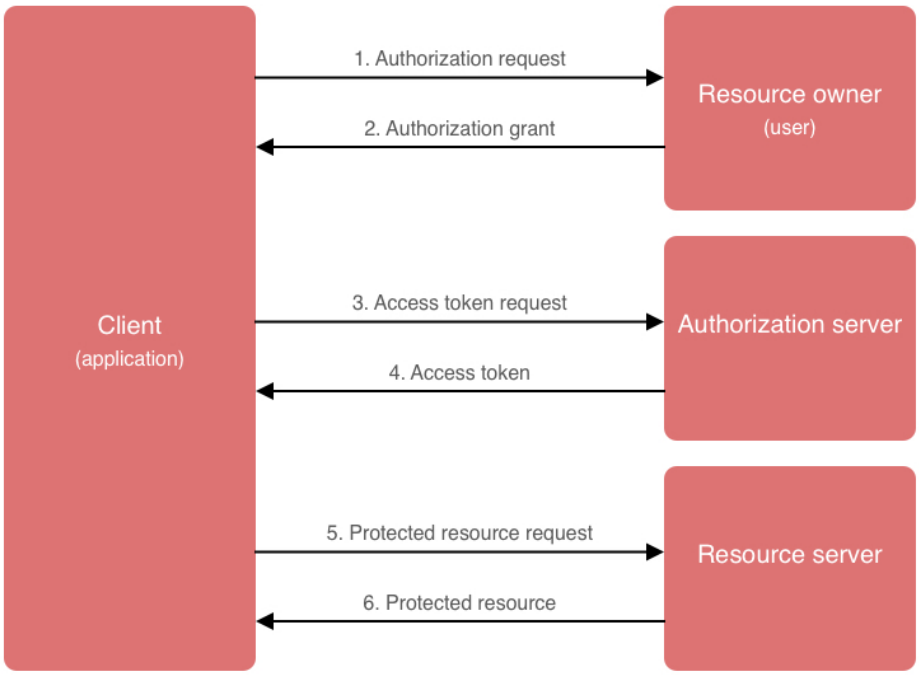


Figure 1.5: 0Auth 2.0 authentication flow diagram for Authorization Code grant type [24]

Chapter 2

Analysis

In this chapter I do an analysis of existing solutions, define requirements, use cases and analyze other services that the iOS application is going to be integrated with - Grades REST API, the authorization server, the notification server.

2.1 Existing solutions

2.1.1 Grades Android mobile application

Mobile application Grades for Android was implemented in diploma thesis by Tomáš Havlíček at CTU FIT in Prague [25]. Thus I do an analysis of his work rather than other mobile applications focused on school grades. Both Android and iOS applications should be unified in basic functionalities provided, usage and critical user interface (UI).

Android application is integrated with Grades REST API and has these functionalities:

- login and logout through CTU authorization server,
- list of courses for a given semester,
- display grades of a student in a course,
- entry and modification of grades for a student group or particular student,
- list of notifications and receiving of push notifications.

The application supports the operation system Android since version *4.4 Kitkat*. It is written in Kotlin programming language and uses MVVM (*Model-View-ViewModel*) architectural style. It uses principles of reactive programming with a help of library *RxJava* [25].

The user interface of the application conforms to Material Design[2] principle which is a standard on Android OS. I am going to further analyze the UI of Android application in following chapter 3 about design where I am also going to compare it with iOS design principles.

2.1.2 Grades web application

Apart from Android application there is Grades web application[3] also integrated with Grades REST API. The web application provides a richer set of functionalities, especially for teachers. For example the REST API allows definition of custom evaluation structure of a school course so teacher is not restricted by predefined one. Students can do more or less the same as in the Android application. These are some further functionalities of the web application teachers can do:

- define course with a custom evaluation structure,
- define grade value type (number, string, boolean),
- switch on and off sending notifications about grades change to students,
- import and export data in CSV format.

2.2 Grades REST API

Web application Grades provides REST API with a set of operations over the domain. This interface is public with university-wide authorization required. The REST API has still been developed in times of writing this work. Documentation of the interface can be found online[4] in the Grades web application.

Note that in the context if this work, the term classification means an evaluation item or a group of evaluation items (e.g. exam test or homework).

[2] `https://material.io/design`
[3] `https://grades.fit.cvut.cz`
[4] `https://grades.fit.cvut.cz/api/v1/swagger-ui.html`

The API is composed of many controllers form which these are important for this work:

- *Login-controller* – authentication state, user information and roles.
- *Semester-controller* – for obtaining current semester code.
- *Course-controller* – information about a course.
- *Classification-controller* – operations with course grades.
- *Student-classification-controller* – information about student grades or grades of students in a course group and possibility to edit it.
- *Student-group-controller* – information about a group of students.

The API is not very well designed for a client application and requires requests to multiple endpoints and further nontrivial transformations of received data. For example, to obtain list of student's courses with their names, client application needs to send one HTTP `GET` request to *login-controller* (to get courses for user role student) and then additional n `GET` requests to *course-controller* (to get name of each course) where n is number of courses from the first request. This is suboptimal to do on a client side. A simple addition of the name field to a response data of the first request on the server would reduce the overall complexity of this task. Moreover, response data from the server often contain fields logically unrelated to the entity or that can be easily obtained from different endpoint thus increasing data bandwidth.

2.3 Authentication

Grades REST API is public but requires authorization of a user. The user authorizes himself on the Czech Technical University authorization server[5] (OAAS) through OAuth 2.0 authentication protocol.

As stated in [26], a client application needs to be registered in university app manager[6] before using the authentication service. After registration, **authorization scopes** for the application must be chosen in order to work correctly with different services using OAAS. The iOS application needs these authorization scopes:

[5] `https://github.com/cvut/zuul-oaas`
[6] `https://auth.fit.cvut.cz/manager/index.jsf`

- *cvut:grades:user-write,*
- *cvut:grades:user-read,*
- *cvut:grades:course-restricted,*

Except for scopes, the client application must also use the appropriate **authorization grant**. *Authorization Code* is a grant type suitable for the iOS application. With this grant type, a user is authorized through OAAS which then provides authorization code to the client application. The client can then request access token for making signed API calls. The complete authentication process is described in chapter 1 *Technologies and solutions.*

An *access token* is valid for a limited amount of time. To prevent user authorization request every time the *access token* expires, a new one can be obtained with the *refresh token* received in athorization response. The *refresh token* is valid until new *access token* is requested. Thanks to this mechanism, a user can be kept authorized automatically until he explicitly logs out. It also allows receiving customized push notifications, because the application can make signed API requests on a background. When a push notification is received on a device, the application can fetch data and then present a customized notification to the user.

2.4 Notification server

Grades notification server is implemented in Java programming language and uses frameworks Spring and Hibernate [25]. It is a standalone service with public REST interface independent on Grades REST API. For sending notifications to Android platform the notification server uses *Firebase Cloud Messaging* which is a service that allows sending cross-platform push notifications (besides other services) [27].

2.4.1 Integration with the application

There are two options for integrating the notification server with the Grades iOS application to receive push notifications:

- Apple Push Notification service (APN).
- Firebase Cloud Messaging service (FCM).

APN is a native service for sending push notifications to iOS applications. It does not require any advanced setup in the application. A notification server that sends notifications to iOS platform must authenticate with APNs using a provider certificate. To obtain the certificate, a developer must have an Apple Developer account. If the notification server is authorized with the certificate it can start sending notification requests with defined payload to APNs [28]. Grades notification server would have to be extended to support APN service.

FCM allows sending push notification both to Android and iOS platform. These steps are required to integrate it with iOS application:

- upload provider certificate to Firebase server,
- install Firebase library to the application,
- configure the application to receive notifications,
- proccess notification payload and present it to a user.

Grades notification server would not have to be extended in case of integration with FCM, because it only sends one request to FCM that is processed and distributed to both platforms.

I have chosen to use APN service because it is a native way on the iOS and does not require to add any third party library. Even though it requires extending the Grades notification server, it is still simpler way than integration of FCM. In chapter 4, I describe how I extended the notification server.

2.5 Requirements

Functional and nonfunctional requirements' scope of iOS application is based on the Android application I analysed in the previous section.

2.5.1 Functional requirements

2.5.1.1 Authentication

User can log in to the applications through his account on the authorization server of CTU in Prague. The application supports two user roles - *student* and *teacher*. User can log out of the application.

2.5.1.2 Semester selection

User can choose a semester from options of previous semesters and display data for it. Semester selection stays persistent after application closing.

2.5.1.3 List of courses

User with role student can display a list of all courses he attends or attended in particular semester. In case the user has a teacher role the list is composed of teached courses in a particular semester. In case he has both student and teacher role he can see all courses he attends and teaches.

2.5.1.4 Course grades

User with student role can display detail of course and see grades for that course.

2.5.1.5 Grades of a student

User with role teacher can add or modify grades of a chosen student if he has rights for it.

2.5.1.6 Grades of a student group

User with role teacher can add or modify grades of student's group (class, parallel,..) for selected classification (e.g. exam test). He can choose both group and classification he wants to work with.

2.5.1.7 Notifications

User with student role can receive system-wide notifications about change in his evaluation and mark them as read.

2.5.2 Nonfunctional requirements

2.5.2.1 Operating system

The application supports operating system iOS 12 and is written in programming language Swift 5.

2.5.2.2 Grades API integration

The application is integrated with Grades REST API over the network and uses its resource endpoints for receiving and sending data.

2.5.2.3 OAuth 2.0

User is authorized using the OAuth 2.0 protocol through the authorization server of the CTU.

2.5.2.4 Localization

The application is localized to Czech and English. It supports extension with other languages. Language is based on device's system language. Default language is English.

2.5.2.5 Push notifications

The application receives push notification from the notification server through Apple Push Notification service.

2.5.2.6 Responsive user interface

User interface of application is adaptable to any device's screen size in portrait mode. Application does not support landscape mode. It only supports iPhone devices, not iPad.

2.6 Use cases

There are two user roles in the application - *student* and *teacher*. The use case diagram in 2.1 shows relations between user roles and requirements. In other words what activity can each user role do.

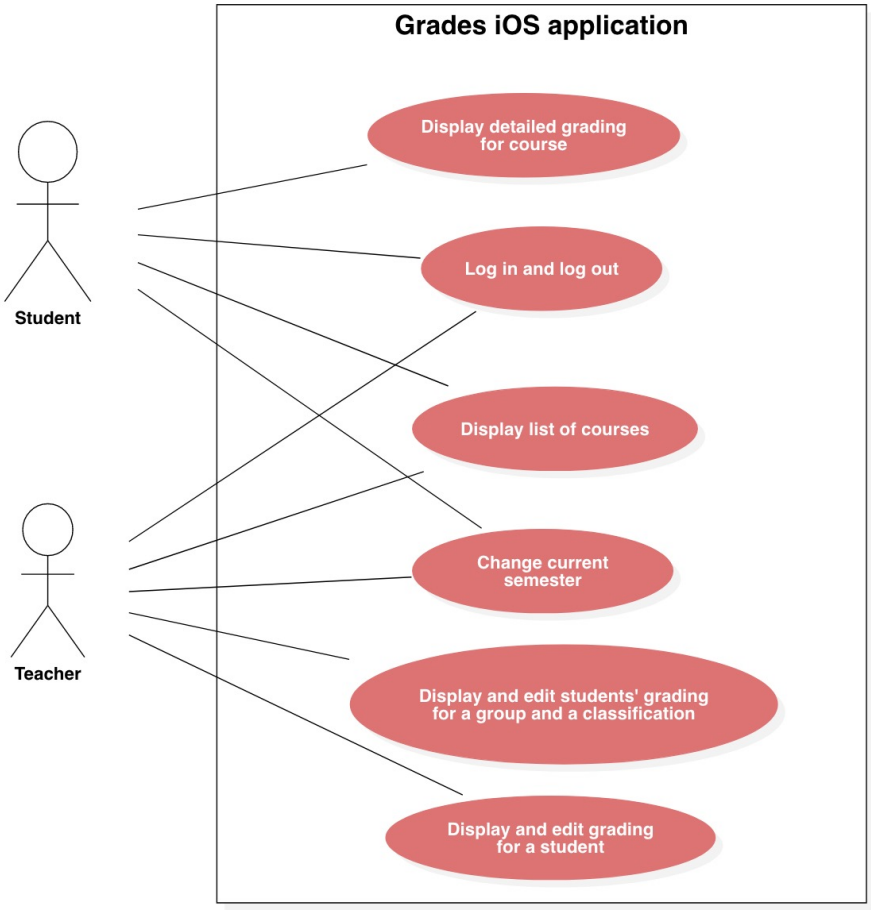


Figure 2.1: Use case diagram of the iOS application

Chapter 3

Architecture and design

This chapter is both about software and graphic design. In the first section of the chapter, I describe the architecture of the application and reasons for choosing it. Then in the second section, I outline main classes and their relationships. Finally, in the last section I compare designed user interface with the Android application, explain choices I made and list used iOS UI classes.

3.1 Architecture

I have chosen MVVM architectural style over MVC, because MVVM moves application logic from *ViewControllers* to *ViewModels* which leads to cleaner code in *ViewControllers* focused only to presenting view and interacting with a user. This isolated logic is also easier to test.

I am going to define the user interface programmaticaly with the help of library SnapKit[7] for autolayout. Even though *Interface Builder* is faster in prototyping, the custom code approach has a few advantages – customization, performance and reusability of UI elements. Also, *Interface Builder* makes it hard to collaborate on a project in a team.

ViewModels ensure all data and dependencies for *ViewControllers* and can contain part of business logic. They provide the data (possibly from more sources) and accept input through interface *ViewControllers* bind to.

[7] `https://github.com/SnapKit/SnapKit`

Thanks to the interface and encapsulation, these two layers are independent and replaceable (e.g. for mock classes in testing).

Model layer takes care of business logic, data transformation, networking or persistence. It contains three sublayers:

- *Data model* – all entities and types in the application.
- *Services* – classes responsible for networking, authentication or persistence.
- *Repositories* – encapsulate code for querying models in one place so that it can be reused in more parts of an application [29].

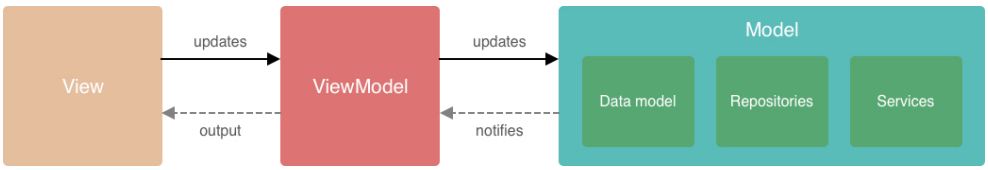


Figure 3.1: Extended MVVM pattern diagram

Often in development, there are other necessary classes (e.g. extensions, configuration, localization or setup code) that are not easily categorizable. Therefore the pattern is rather guid helping a developer than something that must be followed unconditionally.

3.1.1 Reactive framework

As a reactive framework, I have chosen **RxSwift** over ReactiveSwift because ReactiveX is multiplatform standard with a large community thus a developer has a great amount of materials to learn from which is my priority. On the other hand RxSwift learning curve is steep, while ReactiveSwift is not that complex and is also better suited for Swift.

The reactive approach goes in hand with MVVM pattern. Application layers bind to other layers reactive interfaces and provide their own. Such reactive interface consists of observable sequences that observer can subscribe to. This approach reduces problems with asynchronicity, error handling, data flow and it also simplifies the codebase.

3.2 Design

In this section, I list designed main classes and their responsibilities. I describe some of the designed classes like Services or Repositories more in-depth while only listing others like data model, because they are just holders of data and do not contain any business logic.

3.2.1 Scene

Scene is an abstraction over a screen in the app and couples ViewModel and View (`UIViewController`) for the screen. This abstraction allows transition flow between screens from the ViewModel layer instead of the View layer resulting in deeper independence between these two layers. Below I list scenes in the iOS application. Each scene has associated View and ViewModel.

- `Login`
- `CourseList`
- `CourseDetailStudent`
- `GroupClassification`
- `StudentClassification`
- `Settings`

3.2.2 Model layer

Below I list important designed classes with a description of their role. UML class diagram of entities in the application, demonstrating relationships between them, can be found in the attachment A.

- **Data model**
  - `Course`, `StudentCourse`, `TeacherCourse`
  - `User` – student or teacher.
  - `Classification` – grade item.
  - `StudentClassification` – grade item associated with a student.

  - `StudentGroup` – a group of students or parallel.
  - `OverviewItem` – overview of grades (e.g. total score).
  - `Settings` – for storing settings like selected semester or language.

- **Repositories**
  - `CoursesRepository` – querying of courses data.
  - `CourseRepository` – querying of course detail data, reused in both student course detail screen and student classification screen of a teacher.
  - `CourseTeacherRepository` – querying of group grades data for a user with teacher role.
  - `SettingsRepository` – loading, storing and persisting of user settings.

- **Services**
  - `AuthenticationService` – communication with the authentication server, authorization of user, storage of an access token, signed requests and refreshing of an expired access token.
  - `HttpService` – making HTTP requests to external service, has `AuthenticationService` as a dependency for making signed requests.
  - `GradesAPI` – reactive wrapper over Grades REST API endpoints providing clean interface.
  - `PushNotificationService` – service for configuring and handling push notifications from APN service.

3.3 User interface

In this section, I describe the application's user interface and compare designed screens with Android application's UI. The iOS UI is designed by Apple Human Interface Guidelines[8], standard on the platform. I used Sketch as a designing software. File with the design is availabel on the enclosed CD.

[8] `https://developer.apple.com/design/human-interface-guidelines/ios`

3.3.1 Login screen

The login screen is almost the same as in Android app. There is a big logo in the top part and a login button in the bottom part.

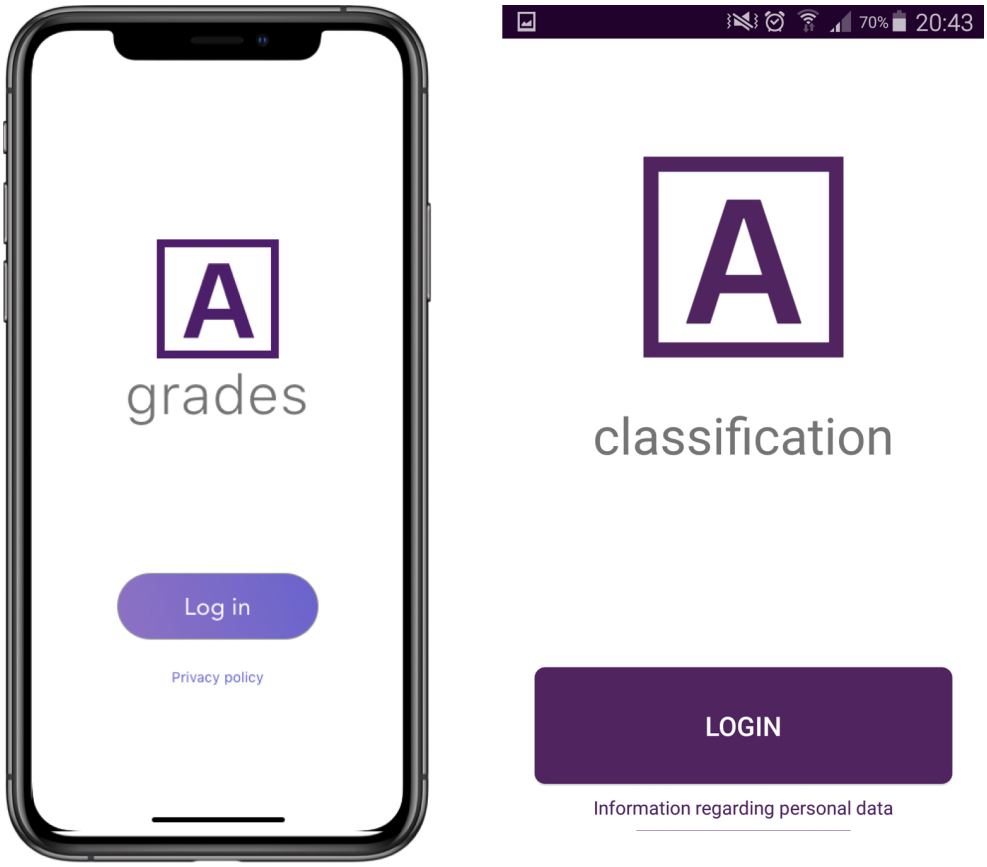


Figure 3.2: Login screens of the iOS and Android applications

3.3.2 Course list screen

This screen is composed of a navigation bar (`UINavigationBar`) and a table of courses (`UITableView`). The navigation bar has a title with a course name and settings button. The table consists of two groups of courses – for student, teacher or both in case a user has both roles. I decided not to show an icon with course shortcut as in the Android app, but a big bold title instead. There is no current semester displayed because a user can see it in settings.

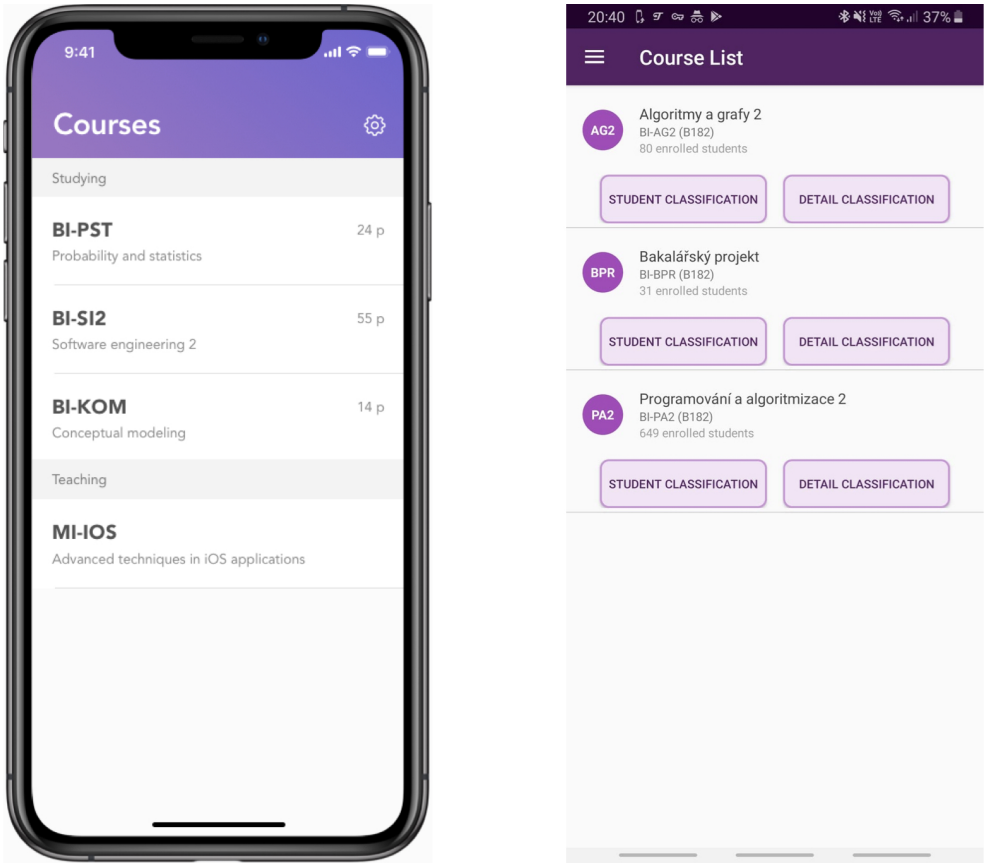


Figure 3.3: Course list screens of the iOS and Android applications

3.3.3 Course detail for a student screen

This screen consists of a navigation bar and a table with student's grades. There is a button for navigating back to course list screen in the top left corner of the screen. There is information about total points earned and final grade in the table header. Evaluation items are grouped by their type (e.g. test, attendance, homework,..) for better transparency. The composition of UI elements is similar to the Android app except for the grouping of table items.

3.3.4 Student grades screen

In this screen, there is a navigation bar with a title, subtitle, back button and save button for saving any changes in student's grades. Under navigation

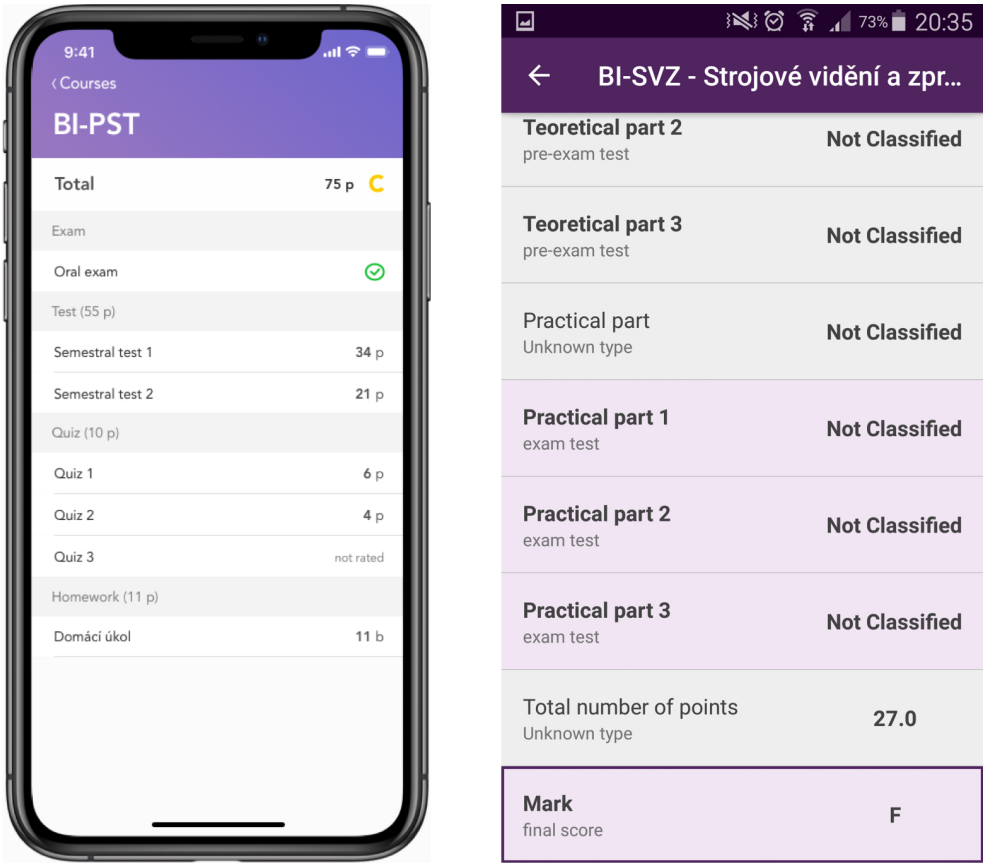


Figure 3.4: Course detail for student screens of the iOS and Android applications

bar, there is a control for switching group view (`UISegmentedControl`) with active student view. Android app does not have this switching control, but user navigates to this screen from the course list screen. Table header contains a selected student's name and total evaluation. When a user clicks on a "change" button he is navigated to a screen for student selection unlike in Android app where student searching is part of the screen. Table body contains evaluation items with text fields for changing the grades.

3.3.5 Student search screen

This screen has a text field in a navigation bar for searching students. The table under the navigation bar displays filtered student names and usernames.

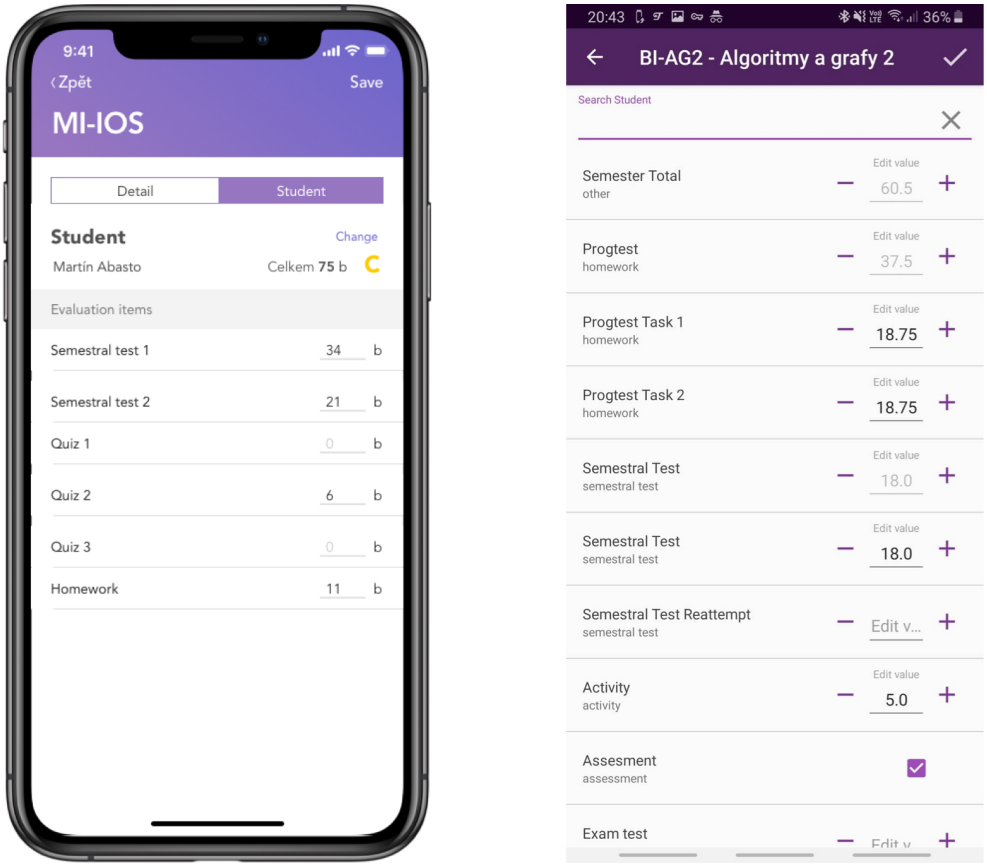


Figure 3.5: Student's grades screen of the iOS and Android application

Tapping on the table cell navigates back to the student grades screen with a new student selected. Android app does not have this screen.

3.3.6 Group grades screen

This screen is a view of grades for a student group. It is similar to the student grades screen. It has two filters in the table header - student group and evaluation item. There is a list of students for filtered selection in the table body. Each table cell has a student's name, username and text field with a value.

3.3.7 Settings screen

Settings screen's navigation bar contains a title, a back button and a logout button, unlike the Android app that has the logout button as part of a table

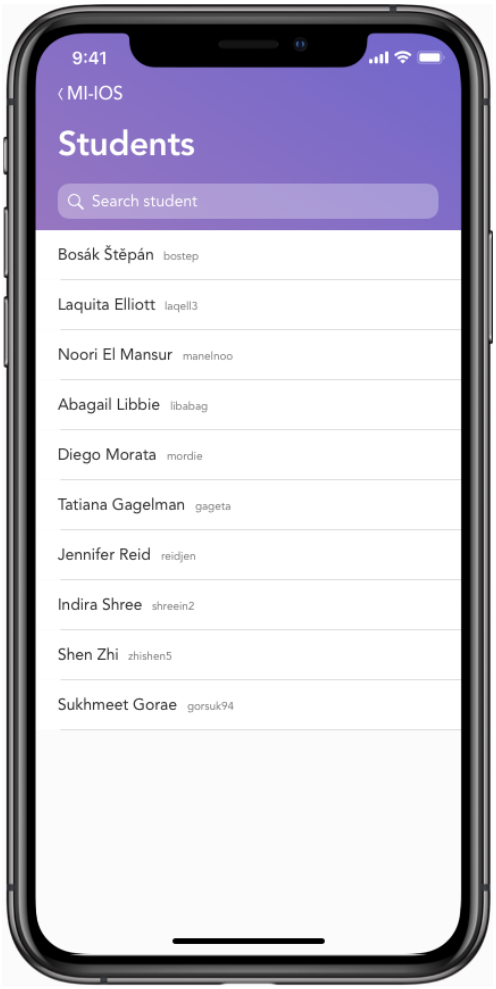


Figure 3.6: Student search screen of the iOS application

body. After logout, the login screen is presented. There are three sections in the settings table - *user* info (name, email, roles), *options* with a semester change and *other* with cells linking to info about the app and licence. There is no option for switching notifications on/off in the iOS app because this setting is located in system settings, which is native behaviour on iOS.

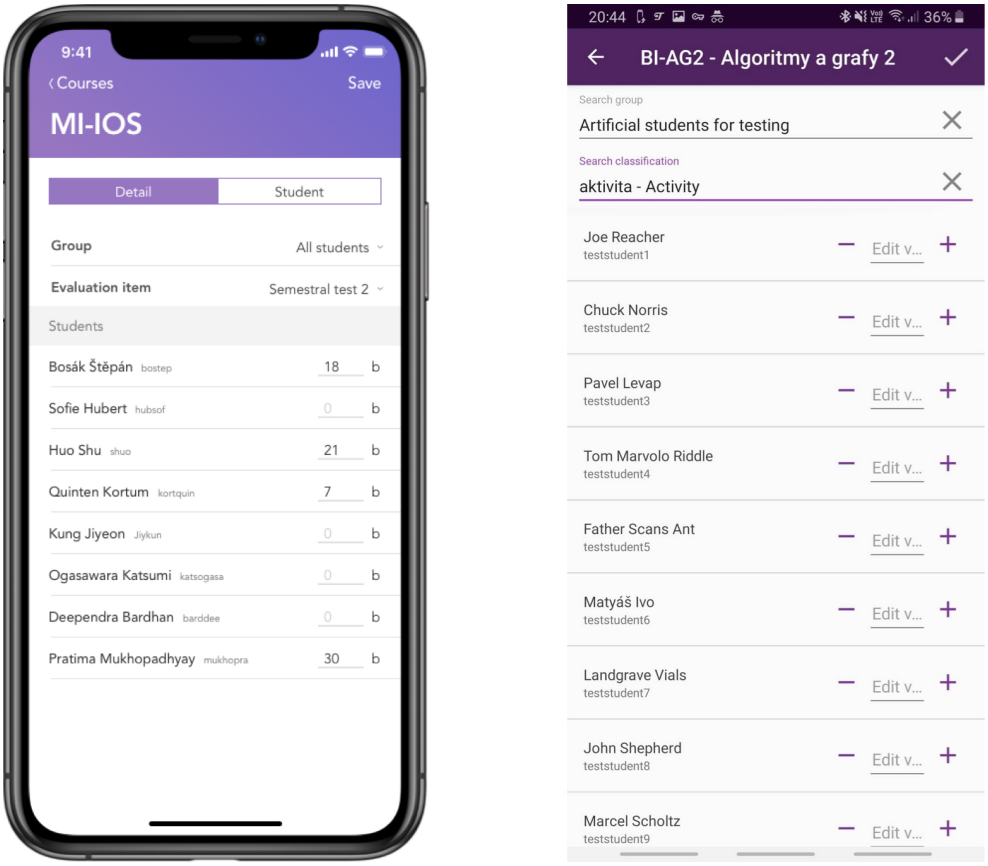


Figure 3.7: Group grades of the iOS and Android application

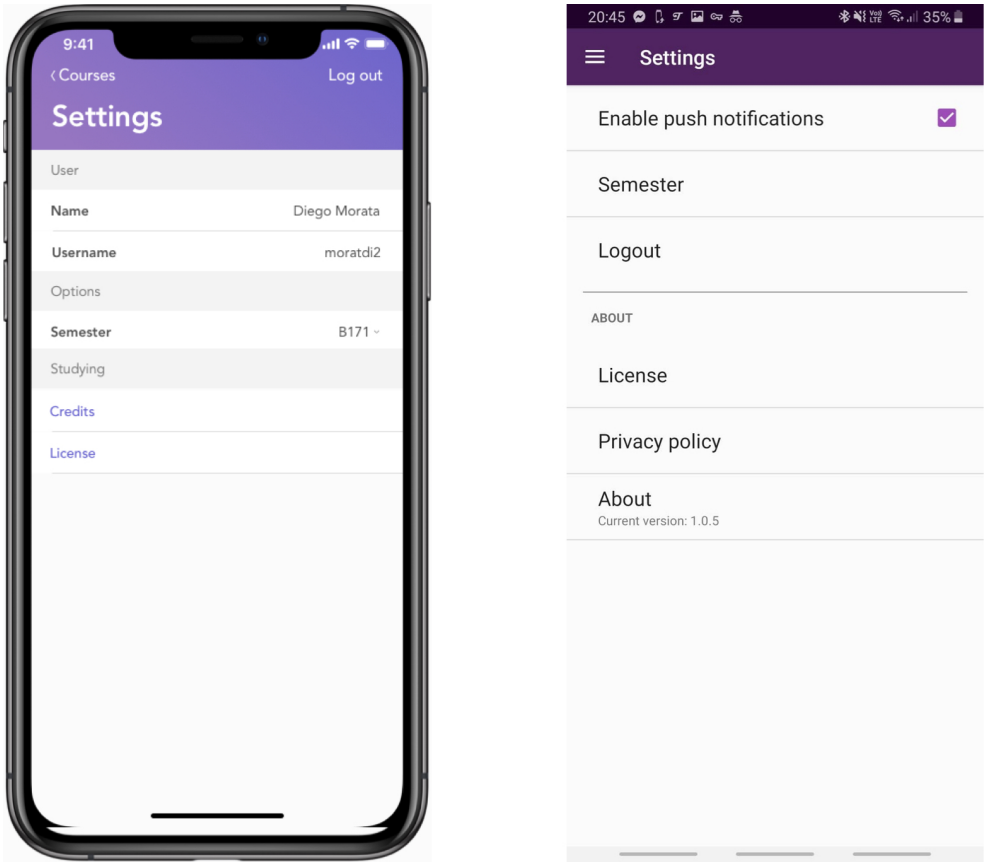


Figure 3.8: Settings screens of the iOS and Android applications

Chapter 4

Implementation

This chapter is about the implementation of the iOS application which I implemented by design and architecture described in the earlier chapter. I point out some interesting implementation parts and decisions I made like Configuration, Authentication, Dependency Injection and other. The XCode project with all source codes and setup guide can be found either in the public GitHub repository[9] or on the enclosed CD.

4.1 Dependency management

When choosing dependency manager for managing external libraries I considered two tools - CocoaPods[10] and Carthage[11]. These dependency management tools are a little bit different and can be also used both at the same project. CocoaPods are easier to use and has more libraries, but modifies the project file structure and build all external libraries every time before building project's source code. Carthage, on the other hand, keeps libraries built so it is faster in build time. Obviously, it takes more time on initialization. I chose **Carthage** as a dependency management tool.

4.2 Configuration

The application has two environments - *Debug* (development) and *Release* (production) which can have a different configuration (app secret, Grades API

[9] https://github.com/jstorm31/grades-ios
[10] https://cocoapods.org/
[11] https://github.com/Carthage/Carthage

URL, etc.). On iOS configuration can be put into Plist file which is basically a dictionary. To keep the configuration of the application easy to manage I put it into three Plist files - *Common*, *Debug*, *Release*. *Common* contains shared configuration for all environments while *Debug* and *Release* stores environment specific configuration. There is `EnvironmentConfiguration` class in the project that parses those Plist files into a Swift dictionary and merges configuration from different files and for the current build environment. Then I create strongly typed property for each record in the dictionary which I can access anywhere from code. With this approach, the configuration is at one place so adding a new property is as easy as adding it to corresponding Plist file and creating strongly typed property in the extension of `Environment-Configuration` class.

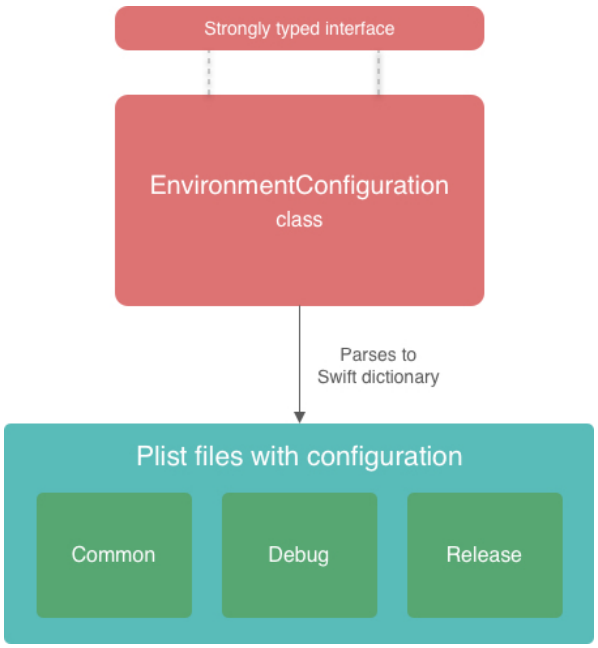


Figure 4.1: Configuration management diagram

4.3 Dependency injection

One of the most challenging problems in application development is dealing with dependencies. I needed some of the classes to have only one instance and to share the instance between other classes. I didn't want to use singleton

pattern[12], so I used following neat pattern which could be best explained with comments in the code example 4.1.

[12] https://matteomanferdini.com/swift-singleton/

```swift
// A dependency
class AuthenticationService {
  // ...
}
// Protocol saying that class implementing the protocol
// must have authService property
protocol HasAuthenticationService {
  var authService: AuthenticationService { get }
}
// AppDependency class holds all other dependencies in the project
// and is only singleton class
class AppDependency {
  // Prohibit initialization
  private init() {}

  // Property for accessing only one instance of the class
  static let shared = AppDependency()

  // lazy stored app dependencies - instantiated on first use
  lazy var authService: AuthenticationService =
    AuthenticationService()
  // ...
}
// AppDependency implements HasAuthenticationService protocol
// -> it has authService property of corresponding type
extension AppDependency: HasAuthenticationService {}
// Class that has AuthenticatinoService dependency
class LoginViewModel {
  // This ensures only AuthenticationService is taken from
    AppDependency class
  // Other dependencies could be chained with & character
  typealias Dependencies = HasAuthenticationService
  private let dependencies: Dependencies

  init(dependencies: Dependencies) {
    self.dependencies = dependencies
  }
  // ...
}
// Instantiation of LoginViewModel with AppDependency shared
    instance
let loginVM = LoginViewModel(dependencies: AppDependency.shared)
```

Listing 4.1: Dependency injection code example

4.4 Authentication

For OAuth 2.0 authentication I used OAuthSwift[13] library. This library helps with authentication flow, storing and refreshing access token and making authorized HTTP requests. It only needs to know a few things like URL address of the authentication server or application's secret before starting the authentication. When a user clicks Login button the OAuthSwift library the authentication process by opening Safari browser with URL of OAAS. After user's successful authentication, the server redirects him back to the application. The library than saves an access token and authentication is complete. I made a wrapper class `AuthenticationService` around this library to make the authentication interface more friendly and reactive.

4.5 Scene Coordinator

I described what Scene is in previous chapter. In the example, 4.2 I demonstrate the implementation of `SceneCoordinator` without implementational details. This implementation of `SceneCoordinator` was originally presented in [30].

```
1  // Enumeration of scenes with associated view models
2  enum Scene {
3    case login(LoginViewModel)
4    case courseList(CoursesViewModel)
5    //...
6  }
7
8  // Extension with function returning UIViewController for each
       screen
9  extension Scene {
10   func viewController() -> UIViewController {
11     switch self {
12     case let .login(viewModel):
13       var loginVC = LoginViewController()
14       loginVC.bindViewModel(to: viewModel)
15       return loginVC
16
17     case let .courseList(viewModel):
18       var courseListVC = CourseListViewController()
```

[13] https://github.com/OAuthSwift/OAuthSwift

```
19        let navController = UINavigationController(
      rootViewController: courseListVC)
20        courseListVC.bindViewModel(to: viewModel)
21        return navController
22      //...
23    }
24  }
25
26  // Different types of transition
27  enum SceneTransitionType {
28    case root // make view controller the root view controller
29    case push // push view controller to navigation stack
30    case modal // present view controller modally
31  }
32
33  // Protocol for SceneCoordinator with methods for managing the
      scene flow
34  protocol SceneCoordinatorType {
35    // Transition to another sceen
36    func transition(to scene: Scene,
37                    type: SceneTransitionType) -> Completable
38    // Pop current sceen
39    func pop(animated: Bool) -> Completable
40  }
41
42  // Usage
43  let coordinator = SceneCoordinator()
44  // Set root scene
45  coordinator.transition(to: Scene.login(LoginViewModel()),
46                         type: .root)
47  // Push new scene to navigation stack
48  coordinator.transition(to: Scene.courseList(CoursesViewModel()),
49                         type: .push)
50  // Pop from naviagtion stack
51  coordinator.pop()
```

Listing 4.2: Scene Coordinator code example

4.6 ViewModel

I described *ViewModel* application layer in the chapter 3 Architecture.
Here I want to demonstrate the relationship between the *ViewModel* and *View* layer. In the listing 4.3 `CourseListViewModel` provides RxSwift

interface and `CourseListViewController` binds it to adequate UI views and controls. Notice the `bindOutput()` function in `CourseListViewModel` which triggers the reactive chain.

```
class CourseListViewModel: BaseViewModel {
  // ViewModel's output, ViewController bind to these properties
  let courses = BehaviorRelay<CoursesByRoles>(value:
    CoursesByRoles(student: [], teacher: []))
  let coursesError = BehaviorSubject<Error?>(value: nil)
  // Initialization, other properties...

  // Triggers binding of output
  func bindOutput() {
    // Business logic for getting courses
    // from CourseRepository and binding the output
    // ...
  }
  // Other private helper methods...
}

// Binding to ViewModel's output in UIViewController
class CourseListViewController {
  var viewModel: CourseListViewModel!
  //...
  // Bind ViewModel's output when view did appear
  override func viewDidAppear(_ animated: Bool) {
    super.viewDidAppear(animated)
    viewModel.bindOutput()
  }
  // Binding to ViewModel's interface
  func bindViewModel() {
    // Bind data to UITableView to display it
    viewModel.courses.asDriver(onErrorJustReturn: [])
      .drive(tableView.rx.items(dataSource: dataSource))
      .disposed(by: bag)

    // Bind error subscription to Error UIView
    viewModel.coursesError.asDriver(onErrorJustReturn: nil)
      .drive(view.rx.errorMessage)
      .disposed(by: bag)
  }
}
```

Listing 4.3: Binding between ViewModel and View code exmaple

4.7 Tables

I used RxDataSources[14] library for managing data source of `UITableViews` in the project. The library simplifies `UITableView` data source setup, helps with the binding of RxSwift data sequences to a `UITableView` and allows easy definition of table sections and polymorphic items. The library supports both static and dynamic data source [31]. In listing 4.4 I present an example of `UITableView` data source setup with the library. To actually display data in the `UITableView`, `UIViewController` must bind to *ViewModel's* interface and connect it with `UITableView` through RxSwift extension on `UITableView` (demonstrated in the previous example).

```
class CourseListViewController {
  // Configure data source by static function from extension
  private let dataSource = CourseListViewController.dataSource()
  // ...
}

extension CourseListViewController {
  static func dataSource() -> RxTableViewSectionedReloadDataSource
    <CourseGroup> {
    return RxTableViewSectionedReloadDataSource<CourseGroup>(
      // Cell configuration
      // Multiple cell types could be used here
      configureCell: { _, tableView, indexPath, item in
        let cell = tableView.dequeueReusableCell(
          withIdentifier: type(of: item).reuseId,
          for: indexPath
        )
        item.configure(cell: cell)
        return cell
    },
      // Set header for each section
      titleForHeaderInSection: { dataSource, index in
        dataSource.sectionModels[index].header
    }
    )
  }
}
```

Listing 4.4: RxDataSources code example

[14] https://github.com/RxSwiftCommunity/RxDataSources

4.8 Push notifications

To make push notifications work on the server side, I extended the notification server. Sending requests to Apple Push Notification service requires the following:

- authenticate the notification server with Apple provider certificate,
- implement the `sendNotificationAPNs` method to send a notificaiton request.

With the help of open source Java APNs[15] library, I could implement it with no cost, because the library provides easy to use interface and the notification payload was already provided by the notification server. In listing 4.5 I present a method for sending the request to APNs written in Java. First I filter only target device tokens of iOS type, then build a notification payload from `notificationId` and `notificationType` and finally send it to APN service.

```
1  private void sendNotificationAPNs(
2    NotificationRequest notificationRequest,
3    List<TokenEntity> receiverTokens
4  ) {
5    // Get devices tokens
6    List<String> tokens = receiverTokens.stream()
7      .map(tokenEntity -> tokenEntity.getToken())
8      .collect(Collectors.toList());
9    // Build notification payload
10     String payload = APNS.newPayload()
11             .badge(1)
12             .localizedTitleKey("notification.title")
13             .localizedKey("notification." + notificationRequest.
       getType())
14             .customField("notificationId", notificationRequest.
       getNotidficationId())
15       .build();
16   // Send the notification
17     apnsService.push(tokens, payload);
18 }
```

Listing 4.5: APN service notification request code example

[15] https://github.com/notnoop/java-apns

On the client side, I had to implement `UNUserNotificationCenterDelegate` protocol and call method `registerUserForNotifications()` of `UIApplication` class to make push notifications work. Additionally, in `PushNotificationService` class, I implemented methods for requesting access to receive push notifications from a user and for redirecting to right screen in the application.

Chapter 5

Testing and Continues Integration

In this chapter, I present two types of automated tests on iOS platform - unit and UI tests. Then I explain my strategy used for testing the application and present test coverage. Finally, I describe Continues Integration process.

5.1 Unit tests

These are types of tests that test classes and methods (units of an application) in isolation from other parts of the application. They verify methods produce correct output from given input and conditions. In Swift, there is `XCTest` class that helps with creating tests and provides a rich set of asserting functions [32]. `XCTTestCase` is a test class for testing methods of an application class. It provides `setup()` and `teardown()` methods for setting and of a test environment [33].

For testing asynchronous tasks implemented in *RxSwift* framework, I use two additional testing frameworks - *RxBlocking*[16] and *RxTest*[17]. Both have a slightly different use case. *RxBlocking* is useful when an input of a method using observable sequences is uncontrolled (e.g. provided by other class). It can block the tested observable sequence until it emits an element or terminates. An output of the sequence can then be asserted. On the other hand, *RxTest* is good for providing custom input to a tested method and inspecting

[16] `https://github.com/ReactiveX/RxSwift/tree/master/RxBlocking`
[17] `https://github.com/ReactiveX/RxSwift/tree/master/RxTest`

the exact result [34]. With the framework you can even assert the exact time of element emission.

Tested classes often depend on other classes (dependencies). With Swift protocols (similar to Java interface) any class that conforms to a protocol can be replaced with a different implementation. To control dependencies of the tested class, I replace a dependency with mocked implementation (e.g. I create `GradesAPIMock` class to emit custom data as input into the tested class).

In listing 5.1 I demonstrate how Swift unit test can look like on real unit test from the application. The test asserts filtering of *ViewModel* data source returns the right amount of items. Note that property `viewModel.searchText` is an observable sequence that binds filtered data to `dataSource`.

```
1  func testStudentFilter() {
2    let dataSource = viewModel.dataSource.subscribeOn(scheduler)
3
4    // Search
5    viewModel.searchText.onNext("ond")
6    var result = try! dataSource.toBlocking(timeout: 2).first()
7    XCTAssertEqual(result![0].items.count, 1)
8
9    // Reset search
10   viewModel.searchText.onNext("")
11   result = try! dataSource.toBlocking(timeout: 2).first()
12   XCTAssertEqual(result![0].items.count, 2)
13 }
```

Listing 5.1: Swift unit test example

5.2 UI tests

Another type of tests on iOS platforms are User Interface tests. Compared to unit tests, they do not test isolated part of an application, but complete flow and interaction with a user interface from end to end. One disadvantage of UI tests is a slower execution speed because an application must run on a simulator during testing as it was used by a user. `XCTest` framework provides many useful classes for User Interface testing. One of them is `XCUIElementQuery` that locates UI elements within an app for testing. These queries return

`XCUIElement` instances which are representations of tested elements [35].

UI tests can be defined in two ways. Either programmatically create instances of `XCUIElement` by their identifiers and then assert or use *XCode* UI test recording feature - *XCode* runs an application, a developer creates flow (tapping buttons, filling forms,..) and the IDE automatically creates `XCUIElement` instances developer interacted with. The second way is convenient, because a developer does not have to write boilerplate code manually. It is a good practise though to first record test case and then update it manually to keep the code clean.

To control network requests in UI tests, I used framework *Swifter*[18], which creates tiny HTTP server within tested app and stubs network request (does not let the request through and replies with custom defined response). *Swifter* also works without any additional configuration in Continues Integration pipeline. With this, UI tests are independent on outer services. In listing 5.2, Ĩdemonstrate how such simple UI test case looks like.

```
1  func testCourseList() {
2    // Login to application
3    app.buttons["Login"].tap()
4    // After login, course list screen is presented
5
6    // Get XCUIElements with defined texts
7    let jsCell = app.staticTexts["BI-PJS.1"]
8    let iosCell = app.staticTexts["MI-IOS"]
9
10   // Wait for 5s to assert those elements are present on screen
11   let exists = NSPredicate(format: "exists == true")
12   expectation(for: exists, evaluatedWith: jsCell, handler: nil)
13   expectation(for: exists, evaluatedWith: iosCell, handler: nil)
14   waitForExpectations(timeout: 5, handler: nil)
15   XCTAssert(jsCell.exists)
16   XCTAssert(iosCell.exists)
17 }
```

Listing 5.2: Swift UI test example

[18] `https://github.com/httpswift/swifter`

This is the flow of the test:

1. `XCTest` runs the application on a simulator and sets up test environment.

2. The first screen of the application is presented when *Login* button is tapped and course list screen is presented (this is application logic).

3. The application makes an HTTP API request to fetch courses and *Swifter* HTTP server stubs the response.

4. The test case waits a maximum of 5 seconds to assert that course data are displayed on the screen of the simulator. Otherwise the test fails.

Video file named `UI_test_example.mp4` of this example test execution can be found on the enclosed CD.

5.3 Test strategy & coverage

Thanks to a separation of concept and layered MVVM architecture, most of the application's business logic is distributed between *ViewModel* and *Model (mainly Repository)* layer. I trust native (e.g. *UIKit*) and external (e.g. *RxSwift*) frameworks that they are tested and work. Thus my strategy for automated unit testing was to test *ViewModels*, *Repositories* and utility functions. With this approach, I maximized relative coverage of the code, because most of bugs are contained in the business logic code, and minimized the amount of written and maintained tests as they can become obsolete over time.

I tested user interaction and user interface in automated UI tests where I validated the main application flow:

- login,
- display list of courses,
- display course detail for students,
- change current semester,
- edit grades of a student group,

- edit grades of a student.

In table 5.1 I present test coverage of the application's code, which was taken from *XCode* test coverage reports. The reports are availabel on the enclosed CD. The overall test coverage 73,8 % is a satisfactory result.

Name	**Coverage**
Tests total	78,3 %
Unit tests	34,9 %
Unit tests ViewModels	74,57 %
Unit tests Repositories	86,28 %
UI tests	72,6 %

Table 5.1: Test coverage table

Apart from automated testing, I tested the application manually both on a simulator and a physical device, because not everything can be tested automatically. I also had to manually test push notifications and integration with the notification server. Thanks to this test strategy I had found many bugs I could fix and the application was validated for release.

5.4 Continues Integration

"*Continuous Integration (CI) is a development practice that requires developers to integrate code into a shared repository several times a day. Each check-in is then verified by an automated build, allowing teams to detect problems early*" [36].

In the development of the application, I used GIT versioning system and hosted the application code in public GitHub[19] repository under Apache 2.0[20] license. For Continues Integration, I used Travis[21] tool that is free and can be easily integrated with GitHub. CI pipeline is a sequence of operations done on a CI server and typically includes building of a project and testing, but can be also extended by deploying and releasing the project (then we are talking about Continues Delivery). My CI pipeline consists of the following tasks.

1. Cloning GitHub repository and installing external dependencies.

[19] `https://github.com/jstorm31/grades-ios`
[20] `https://www.apache.org/licenses/LICENSE-2.0`
[21] `https://travis-ci.org/`

2. Building the project.
3. Running unit tests.
4. Running UI tests.

In Travis one run of a pipeline is called a build. My builds are automatically triggered on merge requests before merging a feature or a bugfix branch into the *master* branch.

Conclusion

The goal of this thesis was to analyse, design, implement and test student evaluation management mobile iOS application for students and teachers of FIT CTU in Prague.

I proceeded by software engineering process and implemented the application integrated with existing web service where students can view detailed evaluation for courses in any semester (past or current), receive system push notifications about new changes and teachers can easily manage evaluation either for a student or a group of students. The application has a clean and intuitive user interface following iOS design guidelines. The integration with Grades web service was sometimes challenging, but I successfully connected it with the mobile application. I tested the application both with automated and manual tests. The result is working application satisfying all defined requirements. I hope it will be widely used at the university.

In chapters of this work, I described in detail the software engineering process of the application which can also serve as a manual for someone wanting to extend the application with other functionalities. One such possible extension could be grade definition section for teacher similar as in the web application. Teachers can define the structure of grade items (semesteral tests, exam tests, attendance) or use templates of pre-generated ones. Another, more technical extension, would be to implement data caching. Now data are fetched from the network. Storing them on locale device's storage and presenting immediately while new data are downloaded from network would result in better user experience. The application could also be extended to iPad and Apple Watch devices.

Bibliography

[1] Inc.", A. Swift Has Reached 1.0. [online], September 2014, [cit. 2019-03-23]. Available from: https://developer.apple.com/swift/blog/?id=14

[2] Apple Inc. Swift. [online], 2019, [cit. 2019-03-23]. Available from: https://developer.apple.com/swift

[3] Apple Inc. UIKit framework. [online], 2019, [cit. 2019-03-23]. Available from: https://developer.apple.com/documentation/uikit

[4] Apple Inc. View Controllers. [online], 2019, [cit. 2019-03-23]. Available from: https://developer.apple.com/documentation/uikit/view_controllers

[5] Apple Inc. UIViewController. [online], 2019, [cit. 2019-03-23]. Available from: https://developer.apple.com/documentation/uikit/uiviewcontroller

[6] Apple Inc. UITableViewController. [online], 2019, [cit. 2019-03-23]. Available from: https://developer.apple.com/documentation/uikit/uitableviewcontroller

[7] Apple Inc. UINavigationController. [online], 2019, [cit. 2019-03-23]. Available from: https://developer.apple.com/documentation/uikit/uinavigationcontroller

[8] Apple Inc. UITabBarController. [online], 2019, [cit. 2019-03-23]. Available from: https://developer.apple.com/documentation/uikit/uitabbarcontroller

[9] Apple Inc. Views and Controls. [online], 2019, [cit. 2019-03-24]. Available from: https://developer.apple.com/documentation/uikit/views_and_controls

[10] Apple Inc. UINavigationBar. [online], 2019, [cit. 2019-03-24]. Available from: https://developer.apple.com/documentation/uikit/uinavigationbar

[11] Apple Inc. UIView. [online], 2019, [cit. 2019-03-24]. Available from: https://developer.apple.com/documentation/uikit/uiview

[12] Apple Inc. UIStackView. [online], 2019, [cit. 2019-03-24]. Available from: https://developer.apple.com/documentation/uikit/uistackview

[13] Apple Inc. UIStackView subitems positioning. [online], 2019, [cit. 2019-03-24]. Available from: https://docs-assets.developer.apple.com/published/82128953f6/uistack_hero_2x_04e50947-5aa0-4403-825b-26ba4c1662bd.png

[14] Apple Inc. UILabel. [online], 2019, [cit. 2019-03-24]. Available from: https://developer.apple.com/documentation/uikit/uilabel

[15] Apple Inc. UITextfield. [online], 2019, [cit. 2019-03-24]. Available from: https://developer.apple.com/documentation/uikit/uitextfield

[16] Apple Inc. Interface Builder editor. [online], 2019, [cit. 2019-03-24]. Available from: https://developer.apple.com/xcode/interface-builder

[17] Pillet, F.; Bontognali, J.; et al. *RxSwift - Reactive programming with Swift*. Razeware LLC., 2017, 574 - 575 pp. Available from: https://store.raywenderlich.com/products/rxswift

[18] Singh, N. A quick introduction to Functional Reactive Programming (FRP). [online], March 2018, [cit. 2019-03-28]. Available from: https://medium.freecodecamp.org/functional-reactive-programming-frp-imperative-vs-declarative-vs-reactive-style-84878272c77f

[19] Pillet, F.; Bontognali, J.; et al. *RxSwift - Reactive programming with Swift*. Razeware LLC., 2017, 18 pp. Available from: https://store.raywenderlich.com/products/rxswift

[20] ReactiveX - asynchronous programming API with observable streams. [online], [cit. 2019-03-28]. Available from: http://reactivex.io

[21] ReactiveSwift - streams of values over time. [online], [cit. 2019-03-28]. Available from: https://github.com/ReactiveCocoa/ReactiveSwift

[22] Rouse, M. RESTful API. [online], March 2019, [cit. 2019-02-17]. Available from: https://searchmicroservices.techtarget.com/definition/RESTful-API

[23] D. Hardt, Microsoft. The OAuth 2.0 Authorization Framework. [online], October 2012, [cit. 2019-02-17]. Available from: https://tools.ietf.org/html/rfc6749

[24] Digital Ocean, L. OAuth 2.0 flow diagram. [online], 2019, [cit. 2019-03-24]. Available from: https://assets.digitalocean.com/articles/oauth/abstract_flow.png

[25] Havlíček, T. *Mobilní Aplikace - Klasifikace*. Master's thesis, Czech Technical University in Prague, Faculty of Information Technology, May 2018. Available from: https://alfresco.fit.cvut.cz/share/proxy/alfresco/api/node/content/workspace/SpacesStore/9f21f9bb-3ad3-4a51-b38e-c9127bbda5f2

[26] Jirůtka, J. OAuth 2.0. [online], January 2017, [cit. 2019-03-4]. Available from: https://rozvoj.fit.cvut.cz/Main/oauth2

[27] Inc., G. Firebase Cloud Messaging. [online], [cit. 2019-03-8]. Available from: https://firebase.google.com/docs/cloud-messaging

[28] Apple Inc. Apple Push Notification service. [online], June 2018, [cit. 2019-05-01]. Available from: https://developer.apple.com/library/archive/documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/APNSOverview.html

[29] Repository Design Pattern in Swift. [online], September 2017, [cit. 2019-03-18]. Available from: https://medium.com/@frederikjacques/repository-design-pattern-in-swift-952061485aa

[30] Pillet, F.; Bontognali, J.; et al. *RxSwift - Reactive programming with Swift*. Razeware LLC., 2017, 616 - 625 pp. Available from: https://store.raywenderlich.com/products/rxswift

[31] Table and Collection view data sources. [online], 2019, [cit. 2019-04-10]. Available from: https://github.com/RxSwiftCommunity/RxDataSources

[32] Apple Inc. XCTest. [online], June 2018, [cit. 2019-05-01]. Available from: https://developer.apple.com/documentation/xctest

[33] Apple Inc. XCTestCase. [online], June 2019, [cit. 2019-05-01]. Available from: https://developer.apple.com/documentation/xctest/xctestcase

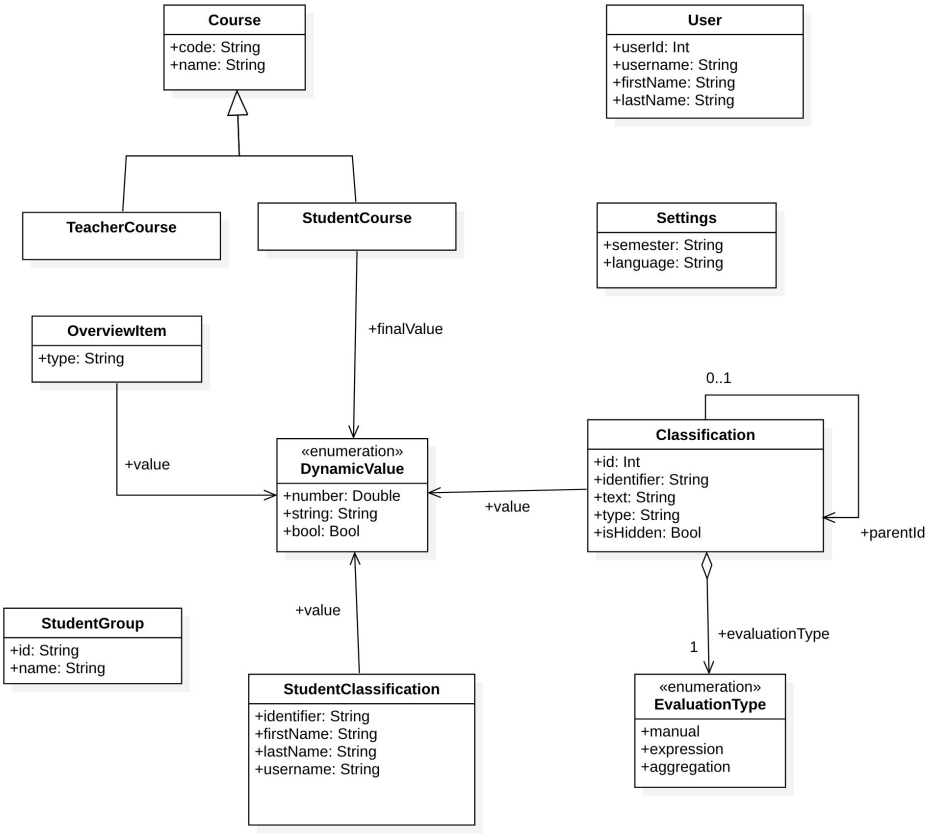
[34] Todorov, M. Testing with RxBlocking, part 1. [online], December 2017, [cit. 2019-05-01]. Available from: http://rx-marin.com/post/rxblocking-part1/

[35] Apple Inc. User Interface Tests. [online], 2019, [cit. 2019-05-02]. Available from: https://developer.apple.com/documentation/xctest/user_interface_tests

[36] ThoughtWorks, Inc. Continues Integration. [online], 2019, [cit. 2019-05-03]. Available from: https://www.thoughtworks.com/continuous-integration

Appendix A

Entity class diagram



Appendix B

Contents of enclosed CD

- `readme.md` ..... the file with CD contents description
- `design` ..... the directory with design files
- `build` ..... the directory with executables
- `src` ..... the directory of source codes
  - `grades-ios` ..... implementation sources
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `test` ..... test reports and video
  - `reports` ..... XCode test report files
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format
  - `thesis.ps` ..... the thesis text in PS format