MASTER'S THESIS

CZECH
TECHNICAL
UNIVERSITY
IN PRAGUE

F3

Faculty of Electrical Engineering
Department of Computer Science

Reactive GitLab API library for Apple platforms

Bc. Anh Duc Tran

Supervisor: Ing. Jakub Průša
Field of study: Software Engineering
January 2019

ČVUT
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení:	**Tran**	Jméno: **Anh Duc**	Osobní číslo: **406442**
Fakulta/ústav:	**Fakulta elektrotechnická**		
Katedra/ústav:	**Katedra počítačů**		
Studijní program:	**Otevřená informatika**		
Studijní obor:	**Softwarové inženýrství**		

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Reaktivní GitLab API knihovna pro Apple platformy

Název diplomové práce anglicky:

Reactive GitLab API library for Apple platforms

Pokyny pro vypracování:

1. Proveďte rešerši knihoven pro komunikaci s GitLab API. Zaměřte se na následující majoritní Apple platformy: iOS, macOS, watchOS, tvOS.
2. Navrhněte vlastní knihovnu, která bude fungovat na zmíněných Apple platformách. Návrh musí být modulární, aby bylo možné jednoduše doplnit
ostatní části API. (např. CI, CD, Award Emoji, ?)
3. Vzhledem k rozsáhlosti Gitlab API není cílem implementovat celé API, ale pouze část API (tzv. endpointy). Implementujte endpointy, které souvisí s
autentizací, repozitáři, revizí a autory. Při implementaci použijte reaktivní přístup programování.
4. Funkčnost knihovny demonstrujte použitím ve vámi nově vytvořené aplikaci na platormě iOS. Aplikace bude zobrazovat všechny repozitáře, do kterých má uživatel přístup. Zároveň bude u každého repozitáře zobrazovat všechny jeho revize a detail každé revize. Uživatel bude mít možnost v nastavení aplikace vyplnit své jméno a heslo pro použití API.
5. Knihovnu otestujte pomocí unit testů. Dále proveďte měření rychlosti a srovnejte naměřené výsledky s existujícími knihovnami pro Apple platformy

Seznam doporučené literatury:

[1] iOS Apprentice: Beginning iOS development with Swift 4 - Matthijs Hollemans,
Fahim Farook
[2] RxSwift: Reactive Programming with Swift - Florent Pillet, Junior Bontognali,
Marin Todorov, Scott Gardner
[3] Design Patterns by Tutorials: Learning design patterns in Swift 4 - Joshua
Greene, Jay Strawn
[4] Apple Developer Documentation - https://developer.apple.com/documentation/

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Jakub Průša, katedra softwarového inženýrství FIT

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

Datum zadání diplomové práce: **25.06.2018** Termín odevzdání diplomo [redacted]

Platnost zadání diplomové práce: **30.09.201** [redacted]

[redacted]

Ing. Jakub Průša
podpis vedoucí(ho) práce

podpis vedoucí(ho) ústavu/katedry

prof. Ing. Pavel Ripka, CSc.
podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

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

19/12/2018

Datum převzetí zadání

[redacted]

Podpis studenta

Acknowledgements

I would like to show a deep gratitude and many thanks to Ing. Jakub Průša and mobile development team from Quanti s. r. o. for all suggestions, consultations, and feedback during the creation of this thesis. Furthermore, I want to thank my parents and all people that supported and motivated me during my studies. Thank you very much.

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 particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on 7st January 2019

Abstract

This master's thesis aims at the creation process of a reactive library for communication with GitLab API for Apple platforms, demonstrating the functionality in an iOS demo application and comparison of the current solutions. This library for communication simplifies future use in applications that need to communicate with GitLab API. This library and the demo application are available as open source for the community for usage or for adding new functionalities.

Keywords: GitLab, Reactive Extensions, RxSwift, API, Networking, Apple platforms, iOS, macOS, tvOS, watchOS

Supervisor: Ing. Jakub Průša

Abstrakt

Tato práce se zaměřuje na proces vytváření reaktivní knihovny pro komunikaci s GitLab API pro Apple platformy, následné na použití této knihovny ve zkušební iOS aplikaci a poté na porovnání se stávajícími řešeními. Knihovna pro komunikaci usnadní budoucí použití v aplikacích, které potřebují komunikovat s GitLab API. Tato knihovna i demo aplikace je dostupná jako open source komunitě pro použití či případné rozšíření o další funkcionality.

Klíčová slova: GitLab, Reactive Extensions, RxSwift, API, Networking, Apple platforms, iOS, macOS, tvOS, watchOS

Překlad názvu: Reaktivní GitLab API knihovna pro Apple platformy

Contents

Figures

Tables

Chapter 1

Introduction

Version control (also known as source control or revision control) systems are software tools that help a software team manage source code changes over time. Every modification is saved in this system so that it is possible to recall specific versions later. It is also a way how to collaborate with other team members and work on the same files without the members rewriting each other's files. The main benefits of VCS are a long-term change history of every file with information, a possibility to work concurrently on the same code and traceability of changes. Nowadays some of the most used VCS are Apache Subversion (SVN), Mercurial and Git[1].

Git is an open-source system for distributed version control and nowadays is by far the most widely used modern VCS. One of the best hosted open-source Git repositories is GitLab [2]. GitLab hosts user accounts like GitHub, but it also offers software to be used on third-party servers. Users can communicate with this application using a web application or an REST API.

The web application allows users to do some tasks easier, but when it comes to automation, it is not very fast and effective. Therefore these applications provide a REST API which allows developers to communicate with the application using commands. These commands can be used to automate tasks and therefore save time and money.

The communication uses HTTP requests and responses. The requests must be specially created and sent to a correct API URL in order to receive the desired response. Manual creation of these requests is a complex, repetitive and time-consuming process. Therefore using a library, which makes this process faster and more straightforward is beneficial. After the request is created, the communication takes place. The communication is asynchronous and requires advanced techniques to handle this problem. One of the technique is to use a Functional Reactive Programming (FRP), which many developers nowadays prefer to use when developing new applications. At this moment, there is no library for communication with GitLab API created with support on all Apple platforms and which uses FRP. Therefore the primary goal of this master's thesis is to create a library for communication with GitLab API which is supported on all Apple platforms and is implemented using FRP. An iOS demo application must also be created to demonstrate the functionality of the library. The subgoal of this master's thesis is to use software engineering principles and processes during the creation of this library and the application.

Chapter 2

Requirements

As stated in the introduction, the goal of this thesis is to create a library for communication with GitLab API with the support of all Apple platforms such as iOS, macOS, watchOS and tvOS. The whole GitLab API provides in total hundreds of endpoints. The goal is not to implement the communication with all endpoints. The goal is to create a modular library so that adding components for communication with new endpoints is simple and doesn't require many changes in the existing code. The created library must contain endpoints related to authentication, projects, repositories, commits, and authors.

2.1 Library requirements

The functional and nonfunctional requirements are mostly related to the specification of this master's thesis. Functional requirements define the internal workings of the software and its functionality. Nonfunctional requirements are related to the constraints on the design or implementation [3]. The functional requirements for this library are the followings:

1. Communication with different GitLab hosts
2. User authenticate using username and password or a private key or an OAuth token

The goal of this library is not to implement all endpoints but to be open for new endpoint implementation. Therefore a modular approach must be used. The main endpoints, that need to be implemented are

1. Authentication
2. Repositories
3. Commits
4. Authors/Users

The non-functional requirements for this library are the followings:

1. Use a functional reactive approach
2. Covered by unit tests

3. Work on these Apple platforms: iOS, macOS, watchOS and tvOS.
4. The API of the library should be intuitive
5. Design must be modular

2.2 Demo application requirements

The functional requirements for this application are the followings:

1. Communication with different GitLab hosts
2. The user can log in using a username and a password
3. Show all repositories the user has access to
4. In each repository, the user can view all commits the user has access to
5. The user can show a detail of each commit he has access to

The non-functional requirements for this application are the followings:

1. Works on iOS 12 and newer
2. Works on iPhone and iPads
3. Must use RxGitLabKit to demonstrate its functionality
4. Use a functional reactive approach

2.3 Additional requirements

1. The code must be documented
2. The code is open source

Chapter 3

Analysis

The aim of this chapter is to analyse the areas that help reaching the goal of this thesis. The main topics discussed in this chapter are GitLab API with available API clients, programming languages for Apple platforms, programming paradigms and architecture patterns.

3.1 GitLab API

GitLab API is a REST API and is divided into Community Edition (CE) and Enterprise Edition (EE). Nowadays only version 4 of the API is available. The version v3 was removed in GitLab 11.0. In the future, the API will start moving to GraphQL which will bring many benefits. For example, avoiding the maintenance of two different APIs, callers can request only for data which is needed, and it is versioned by default [4]. The Community Edition API has hundreds of endpoints which are divided 68 groups which are shown in table E.1 included in appendix section. An endpoint group is a set of endpoints related to a certain functionality category. For example a Commits endpoint group includes endpoints which work with related operations to a commit. The implemented endpoint groups in this thesis are highlighted with **bold** font.

The Enterprise edition has additional endpoint groups to the Community edition. These endpoint groups are shown in the table 3.1:

Epics
Epic Issues
Geo Nodes
Issue Links
License
Managed licences
Merge Request Approvals

Table 3.1: GitLab API Enterprise Edition Endpoint Groups

3.1.1 Endpoint groups to be implemented

Because the goal is not to implement whole GitLab API (because it contains hundreds of endpoints), five endpoint groups were chosen to be implemented. This subsection summarizes the information about the endpoint groups regarding commits, projects, repositories, users, and authentication.

Authentication

There are three ways to authenticate with the GitLab API:

1. OAuth2 tokens
2. Personal access tokens
3. Session cookie
4. username and password

For admins who want to authenticate with the API as a specific user, or who want to build applications or scripts that do so, two options are available:

1. Impersonation tokens
2. Sudo

How to obtain the tokens is not covered in this thesis.

Projects

Projects API provides endpoints allowing developers to work with projects. GitLab offers three visibility options [5]:

- **private**: Project access must be granted explicitly for each user.
- **internal**: The project can be cloned by any logged in user.
- **public**: The project can be accessed without any authentication.

This API group offers 23 endpoints which allow the developers to do these actions: [1]

- List all projects
- List user projects
- Create, read, update and delete a single project
- Get project users
- Create read, update and delete a single project for user
- Get project events
- Fork project
- List forks of a project
- Star and unstar a project

---

[1] `https://docs.gitlab.com/ee/api/projects.html`

- Get languages
- Archive and unarchive a project
- Upload a file
- Share project with a group
- Delete a shared project link within a group
- Hooks
- Fork relationship
- Search for projects by name
- Start the Housekeeping task for a Project
- Push Rules
- Transfer a project to a new namespace
- Branches
- Project Import/Export
- Project members
- Start the pull mirroring process for a Project
- Project badges
- Issue and merge request description templates

Repositories

Repositories API provides endpoints allow developers to work with repositories. This API group offers 7 endpoints which allow the developer to do these actions: [2]

- List repository tree
- Get a blob from repository
- Get a file archive
- Compare branches, tags or commits
- Contributors
- Merge base

---

[2] `https://docs.gitlab.com/ee/api/repositories.html`

Commits

Commits API provides endpoints allow developers to work with commits in repositories. This API group offers 10 endpoints which allow the developer to do these actions: [3]

- List repository commits
- Create a commit with multiple files and actions
- Get a single commit
- Get references a commit is pushed to
- Cherry pick a commit
- Revert a commit
- Get the diff of a commit
- Add and read the comments of a commit
- Commit status
- List Merge Requests associated with a commit

Users

Users API provides endpoints allow developers to work with users. Some endpoints allow to change the state of another user, but it requires admin privileges. This API group offers 21 endpoints which allow the developer to do these actions: [4]

- List users
- Creation, update, read, delete of a user
- Get and set a status of a user
- List user projects
- List SSH keys of a user
- Create, delete and read an SSH key a user
- Create, delete and read a GPG key a user
- List emails for the current or given user
- Add or delete an email of the current or given user
- Block or Unblock user
- Read all impersonation tokens of a user
- Create, delete and read an impersonation token of a user

---

[3] `https://docs.gitlab.com/ee/api/commits.html`

[4] `https://docs.gitlab.com/ee/api/users.html`

3.1.2 Additional aspects to consider

Pagination

Some endpoints return a list of objects. Sometimes the number of objects is too large to be returned at once - for example a list of all projects on the server. GitLab deals with this problem using *pagination*. It returns objects in *pages* which contain up to a specific number of the desired objects. For this purpose the query parameters `page` and `per_page` are used. The default page number is 1 and the default number of objects per page is 20 and maximum 100. Each response to a request to endpoints which paginate the result includes a pagination header. The header contains data about the total number of items, total number of pages and more useful information. The number of pages and number of items is a useful information, that can be used for downloading all items if needed. The list of parameters contained in the header is shown in table 3.2.

Header	**Description**
`X-Total`	The total number of items
`X-Total-Pages`	The total number of pages
`X-Per-Page`	The number of items per page
`X-Page`	The index of the current page (starting at 1)
`X-Next-Page`	The index of the next page
`X-Prev-Page`	The index of the previous page

Table 3.2: Pagination Headers [6]

3.2 Programming languages for Apple platforms

Nowadays the applications for Apple products are written in two main programming languages: *Objective-C* and *Swift*. The focus of this section is to analyze and decide which programming language (or both) will be used in the GitLab API library.

3.2.1 Objective-C

Objective-C, also known as ObjC is an object-oriented programming language created as an extension of C to which a messaging system from Smalltalk programming language was added. The development of this language began in 1986 and it is used in Mac OS X, iOS and GNU.

This language isn't a fast language because it uses the runtime code compilation. That involves an extra level of indirection when calling another object from an object which when performing many times can slow down the execution. The language also uses null pointers which can cause a security vulnerability. Maintenance of the code is also complicated, because the developers must manage two files (header files and implementation files) for each class. The syntax of the language uses many symbols like `@`, semicolons, parentheses. [7]

Overall Objective-C has many downsides and this language is nowadays not considered to be used in a new project anymore. There is a newer and better language for Apple platforms called *Swift* which is described in the next subsection 3.2.2.

3.2.2 Swift Language

Swift is an open-source programming language developed by Apple Inc. for iOS, macOS, watchOS, tvOS and Linux which was announced at the Developer conference WWDC 2014. It is a multi-paradigm programming language, which took ideas from Objective-C, Rust, Haskell, Ruby, Python, C#, CLU, D [8] and more languages.

Swift offers many features [9]:

- Support for functional programming (`filter`, `map`, `reduce`)
- Native error handling using `try`, `catch`, `throw`
- Generics
- Tuples and multiple return values
- Protocols and extensions

Swift is a relatively new programming language and the latest release version is Swift 4.2. Swift is in comparison to Objective-C faster, safer, more readable and open-source [10]. These are the reasons, why this language was chosen for the implementation of the library developed during this thesis.

Further subsubsections describe some of the functionalities of Swift, which was used in the development phase.

Optionals

Swift is a *type-safe* programming language, which ensures the data type of the variable doesn't change. For example if the variable is of type `Bool` it is not possible to change it to `Int`. The variables are declared using the keyword `var` and constants using `let`. The variables must be initialized before usage and they can not be `nil`. For oparation with the `nil` values in Swift, a type `Optional` is used.

An `Optional` is a wrapper around object types (`Float`, `Int`, `String` ...), which represents two states - the optional has a value set or it is `nil`. If the value of the `Optional` is set, the operation *unwrap optional* can be performed to retrieve this value. If the value is not set, then it behaves like `nil`.

The code example 1 shows the initialization of the `Optional` value with `nil`. The question mark (`?`) indicates an `Optional` value inside the variable. First the code prints `"text is nil"`. Then after setting the value, it performs an unwrap using `optional binding` which saves the optional value into another constant (`unwrappedText` in the example 1), which can be then used in the block scope (inside the curly braces) as a value which is not `nil`. A declaration without using the question mark (`var text:String = nil`) leads to a compilation error.

```
var text:String? = nil

if text != nil {
   print(text)
} else {
   print("text is nil")
}
OUTPUT:
text is nil

text = "I have a value now."
if let unwrappedText = text {
  // unwrappedText has the text value and is not nil
   print(unwrappedText)
} else {
   print("text is nil")
}
OUTPUT:
I have a value now.
```

Listing 1: An `Optional` example

Extensions

`Extensions` allow adding new functionality and attributes to existing classes, structures and protocols. They can also be used on the classes, in which the source code cannot be changed as shown in example 2, where the `Date` from Apple's `Foundation` was extended by a computed type property and an initializer. This feature can reduce and make the code cleaner to use.

Extensions enable adding the followings: [11]

- Adding computed type properties
- Adding new initializer
- Definition of subscripts
- Definition of new methods
- Making an existing type conform to a protocol

```
extension Date {
  public init?(from string: String, using formatter: DateFormatter) {
    if let date =  formatter.date(from: string) {
      self = date
    }  else {
      return nil
    }
  }

  var asISO8601String: String {
    let formatter = ISO8601DateFormatter()
    return formatter.string(from: self)
  }
}
```

Listing 2: An `Extension` of `Date` class example

Protocols

`Protocols` in Swift represents a blueprint set of methods, properties and other requirements which are necessary for the functionality. The `protocol` can be *adopted* by classes, structures and enumerations by implementing the requirements. When a type implements these requirements, it is said that the type *conforms* to the protocol [12]. Essentially a protocol is very similar to an *interface* in Java, but a protocol can be extended by an implementation even on source code which cannot be changed as described in 3.2.2. The code listing 3 illustrates an example of the protocol and protocol extension.

Subscripts

`Subscripts` are shortcuts for accessing member elements of a collection, list or a sequence. An example usage is to access an `Array` element on a certain `index` like this: `arrayOfData[index]` [13]. They can be defined on classes, structures and enumerations and the interesting part is that they can be manually defined to do whatever the developer desires. That means that the `subscript` doesn't have to operate on a collection, list or a sequence, it can for example return a computed value as shown in code listing 4.

3.3 Available GitLab API clients

There are many API clients for GitLab in many different languages such as Swift, Ruby, R, Pearl, Python, Go, PHP, Clojure, Java, and technologies such as Backbone, Node.js, .NET and PowerShell. It is useful to examine these clients to get the inspiration for designing the new library. In this section, firstly the available libraries for Swift are shown, and then a summary of the maintained libraries for other languages is shown. In this thesis, a library

```
protocol APIRequesting {
  var method: HTTPMethod { get }
  var path: String? { get }
  var parameters: QueryParameters { get }
  var jsonDictionary: JSONDictionary? {get}
  var data: Data? { get }

  func buildRequest(with hostURL: URL,
    header: Header?,
    apiVersion: String?,
    page: Int?,
    perPage: Int?) -> URLRequest?
  }

extension APIRequesting {
  public func buildRequest(with hostURL: URL,
   header: Header?,
   apiVersion: String?,
   page: Int?,
   perPage: Int?) -> URLRequest? {
    // Implementation of the method
  }
}
```

Listing 3: Protocol method implementation using an extension

```
struct Power {
  let base: Double
  subscript(index: Int) -> Double {
    return pow(base, Double(index))
  }
}

let base = Power(base: 2)
print("The 3rd power is \(base[3])")
print("The 10th power is \(base[10])")

OUTPUT:
The 3rd power is 8.0
The 10th power is 1024.0
```

Listing 4: Definition of `subscript` example

that supports the latest GitLab API version 4, and the latest release was in the year 2018 is considered to be a maintained library. At the end of this section, there is a summary of the key findings, which can be used in the design phase. Note, that the information is up to date in time of writing this thesis, which is December 2018 and there may be new updates in the future.

3.3.1 Swift

Here is a summary of available Swift clients and a deeper comparison with *RxGitLabKit* can be found in chapter 7.

GitLabKit

GitLabKit is an API client library for GitLab API, written in Swift.

Project link:	`https://github.com/toricls/GitLabKit`
Language:	Swift 3.0
Platforms:	macOS
Latest release:	-
GitLab API v4 support:	Yes
Last commit date:	18 Jun 2017

TanukiKit

A Swift 2.0 API Client for the GitLab API.

Project link:	`https://github.com/nerdishbynature/TanukiKit`
Language:	Swift 2.0
Platforms:	iOS, macOS, tvOS, watchOS
Latest release:	v0.5.2 (4 Aug 2017)
GitLab API v4 support:	No
Last commit date:	4 Aug 2017

3.3.2 Other languages/technologies

There are many clients implemented in other technologies [5]. This is a summary of the most maintained libraries.

NARKOZ/Gitlab

Ruby wrapper and CLI for the GitLab REST API `https://narkoz.github.io/gitlab`

---

[5] `https://about.gitlab.com/partners/#api-clients`

Project link:	`https://github.com/toricls/GitLabKit`
Language:	Ruby 2.0+
Latest release:	v4.7.0 (7 Nov 2018)
GitLab API v4 support:	Yes

GitLab-API-v4

A complete GitLab API v4 client. `https://metacpan.org/pod/GitLab::API::v4`

Project link:	`https://github.com/bluefeet/GitLab-API-v4`
Language:	Pearl
Latest release:	v0.14 (6 Dec 2018)
GitLab API v4 support:	Yes

python-gitlab

Python wrapper for the GitLab API

Project link:	`https://github.com/gpocentek/python-gitlab`
Language:	Python
Latest release:	v1.6.0 (25 Aug 2018)
GitLab API v4 support:	Yes

go-gitlab

A GitLab API client enabling Go programs to interact with GitLab in a simple and uniform way

Project link:	`https://github.com/xanzy/go-gitlab`
Language:	Go
Latest release:	v0.11.7 (15 Nov 2018)
GitLab API v4 support:	Yes

php-gitlab-api

GitLab API client for PHP

Project link:	`https://github.com/m4tthumphrey/php-gitlab-api`
Language:	PHP
Latest release:	v9.9.0 (16 Nov 2018)
GitLab API v4 support:	Yes

Gitlab Java API Wrapper

A wrapper for the Gitlab API written in Java.

Project link:	`https://github.com/timols/java-gitlab-api`
Language:	Java
Latest release:	v4.1.0 (5 Oct 2018)
GitLab API v4 support:	Yes

GitlLab API for Java (gitlab4j-api)

GitLab API for Java (gitlab4j-api) provides a full featured and easy to consume Java API for working with GitLab repositories via the GitLab REST API.

Project link:	`https://github.com/gmessner/gitlab4j-api`
Language:	Java
Latest release:	v4.9.1 (5 Oct 2018)
GitLab API v4 support:	Yes

GitLabApiClient

GitLabApiClient is a .NET rest client for GitLab API v4 (`https://docs.gitlab.com/ce/api/README.html`).

Project link:	`https://github.com/nmklotas/GitLabApiClient`
Language:	.NET Standard 2.0.
Latest release:	v1.0.2 (4 Nov 2018)
GitLab API v4 support:	Yes

PSGitLab

An interface for administering GitLab from the PowerShell command line.

Project link:	`https://github.com/ngetchell/PSGitLab`
Language:	PowerShell
Latest release:	v3.0.1 (3 Oct 2018)
GitLab API v4 support:	Yes

3.3.3 Key findings

The main focus when looking for key findings was how the API of the clients looks like for the developers. The implementation details were not analyzed, because the clients were mostly written in other languages and the ideas are not always transferable between the languages.

Most of the approaches to using the client is to create an instance of the client and providing the host URL with some sort of authorization. This instance was then used to establish the communication with the GitLab API. If rewritten to Swift, the code would look like this:

```swift
let client = GitLabAPIClient("https://example.gitlab.com",
  "PRIVATE_TOKEN")
// or
let client = GitLabAPIClient("https://example.gitlab.com")
client.login("USERNAME", "PASSWORD")
```

The approach the libraries took for reaching the endpoints were mainly separated in two directions. In some of the libraries (3.3.2, 3.3.2, 3.3.2), all the API calls are directly in the client, making the client contain many functions which communicate with the API. The other set of libraries (3.3.2, 3.3.2, 3.3.2, 3.3.2) have the functions grouped by the API endpoint groups as discussed in section 3.1. The second approach is more preferable because of the modular nature of the approach. The code of these ideas rewritten to Swift:

```swift
// The first approach
let project: Project = client.getProject(1)

// The second approach - modular
let project: Project = client.projects.get(1)
```

Some GitLab API endpoints contain a large number of objects which can potentially overload processing. To deal with this problem, pagination is used as introduced in 3.1.2. The libraries also implement a class which handles pagination. The instance of this class is returned instead of an array of objects when requesting endpoints containing a list of those objects. An example of this behavior is shown in this code:

```swift
// The paginator
let projectsPaginator: Paginator = client.projects.getAll()
```

```
// Getting the project
let projects: [Project] = projectsPaginator.load(page: 2, perPage: 100)
```

The key takeaway from this analysis is to create one instance of the client with authorization and the host URL, group endpoints into modules and use paginators when accessing a list of objects.

3.4 Programming paradigms

Before the design phase takes place, it is worth considering which approach will be used in the development, because it can save time while making the implementation simpler and cleaner. One of the considerations to be made is which programming paradigm to use for the problem.

A programming paradigm is a philosophy, style, or general approach to programming. Each paradigm is composed of a set of concepts that makes it the best for a certain kind of problem [14].

Two common programming paradigms are imperative and declarative. Imperative programming describes computation as a list of instructions which change the computers state. On the other hand, declarative programming describes the logic of a computation without describing its control flow.

These programming paradigms include: [15] [16] [17]

- **Imperative**
  - **Procedural** - groups of instructions are grouped into procedures
  - **Object-oriented** - groups instructions with part of the state
- **Declarative**
  - **Functional** - the desired result is declared as the output of a series of function applications
  - **Reactive** - the desired result is declared as a composition of data streams and the propagation of change which update the result when the values change
  - **Logical** - the desired result is declared as the answer to a question about a system of facts and rules
  - **Mathematical** - the desired result is declared as the solution of an optimization problem

In this thesis, only Functional, Reactive paradigms are described because the others are not relevant for this thesis or are well known.

3.4.1 Functional programming

Functional programming (also known as FP) is a declarative programming paradigm that describes computation as an evaluation of a mathematical function. It is based on lambda-calculus, and many functional programming languages can be considered as an extension of lambda-calculus. The keystone of this approach is that using *pure functions* prevents side-effects which makes reasoning about the code easier. [18]

The main concepts of FP

- **Higher-order functions and first class**

  In mathematics and computer science, a *higher-order function* is a function that does at least of the following:

  - takes one or more functions as arguments (i.e., procedural parameters)
  - returns a function as its result

  Functions in functional programming languages are *first class* citizens, which means functions can be used as an argument and return another function as an output. This inherently means that they can also be higher-order. The difference between higher-order and first-class citizen is that higher-order describes a mathematical function applied on another function and first class in a given programming language is a computer science term describing an entity which supports all the operations generally available to other entities. The typical operations include being passed as an argument, modified, assigned to a variable and being returned from a function. [19]

- **Pure functional and referential transparency** Pure functional programs don't have any *side-effects*. This makes the behavior simpler for understanding and to write the code. The output of a pure function on a pure argument doesn't depend on the order of evaluation.

  Because pure functions don't mutate the shared variables of the program, the variables can be parallelly accessed without being influenced by each other. This means that pure functions are thread-safe.

  Pure functional programming languages typically require *referential transparency*. Referential transparency means that if two expressions have the same value, one can be input as the other one in any other expression without influencing the result.

- **Recursion** Looping (iteration) in functional programming languages is usually achieved using recursion. Recursive functions invoke themselves, which lets the program repeat itself. Tail recursion can be detected and optimized by the compiler into the same code used to implement loops in imperative languages.

- **Strict and non-strict evaluation** Functional programming languages can be categorized based on the evaluation strategy. In *strict (eager)* evaluation the arguments of the function are processed before the function is invoked. On the other hand, *non-strict (lazy)* evaluation leaves the arguments in a function unevaluated, and the outer function invocation decides when the values will be computed. Let's consider the next example of functions `f` and `g`:

```
f:= x^2 - x + 5
g:= x * y

// Evaluate expression
f(g(2, 4))
```

The strict evaluation of the expression looks like this:

```
f(g(2, 4)) -> f(2 * 4) -> f(8) -> 8^2 - 8 + 5 -> 61
```

On the otherhand the non-strict evaluation the inner function are computed when they are needed.

```
f(g(2, 4)) -> g(2, 4)^2 - g(2,4) + 5
-> (2 * 4)^2 - (2 * 4) + 5 -> 8^2 - 8 + 5 -> 61
```

From the example above, it is noticeable that in poorly implemented non-strict evaluation a the argument `g(2, 4)` is computed multiple times and in strict evaluation only one time. Strict evaluation is, therefore, more efficient.

Although non-strict evaluation is not very efficient, it is also used and mostly in definition languages. These languages support infinite data structures like an array of all negative numbers of type integer or an array of all prime numbers. Non-strict evaluation is then used only on context with non-infinite length. This lead to the development of lazy evaluation, which is a type of non-strict evaluation, where the result of the initial evaluation of any argument can be shared across an evaluation sequence so that the arguments are evaluated at most only once.

Comparison of functional and imperative programming

Imperative functions can have side-effects, which change the global state of the program. This means that they lack *referential transparency* - the same input can have a different output because of a different state. On the other hand, functional programming prevents side-effects by using *pure functions*. A pure function is a function which given the same inputs, always returns the same output, and has no side-effects. This fact supports referential transparency.

In conclusion, referential transparency of pure functions leads to the elimination of side-effects, which can lead to better understanding and reasoning about the code and makes verification, optimization and parallel programs easier to do. These are the key motivations for using functional programming.

3.4.2 Reactive programming

Reactive programming is a programming paradigm that describes programming with asynchronous data streams or event streams which propagate the changes. Using this paradigm expressing static or dynamic data streams is possible, and the changes are automatically propagated to the execution model.

For example, in an imperative programming `x := y + z` means that `x` is set to the result of `y + z` in the moment the expression is evaluated. After this moment `y` and `z` can be changed but the change doesn't propagate to `x`. Hoewer in reactive programming, whenever `y` or `z` is updated, the value `x` is also automatically updated without the need of new execution of `x := y + z`.

Reactive programming was designed as a way to make the creation of interactive user interfaces and real-time animations easier. For example, in MVC architecture, reactive programming can allow changes to the underlying model and the changes are automatically reflected in the view and vice versa. [20]

More examples and concrete implementation can be found in section 3.6 RxSwift.

3.4.3 Functional reactive programming

Functional reactive programming is a combination of functional and reactive programming. The basic usage is that functions are applied to event streams or data streams, and the result is observed and reacted upon.

3.4.4 Conclusion

The core functionality of the developed library is to send and receive data from GitLab API using the network. The data usually must be transformed from one format to another (for example from JSON to an object).

Network communication in its core requires asynchronous data handling. For asynchronous streams of data, the reactive programming approach is suitable. Also because the sent/received data from the API must be transformed, the functional approach is appropriate. In conclusion, the best paradigm to use for the library of this thesis is functional reactive.

3.5 FRP in Swift

Swift offers native functions like `filter`, `map` and `reduce` and also can use `NSNotificationCenter` or `Key-Value Observing (KVO)` to make the code functional and reactive. `NSNotificationCenter` is a singleton and can make the code hardly traceable when debugging because it is globally accessible and can notify or be notified anywhere from code [21]. KVO in Swift has an API which is not easy to use and can bring much boilerplate code to observe one variable. Using the native Swift components to implement FRP is a

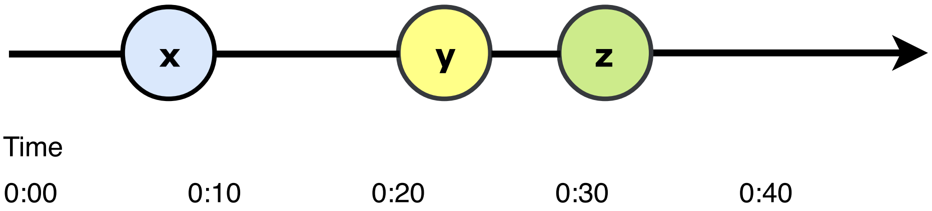


Figure 3.1: An example marble diagram

process which can take much time, and the maintainability of the code can be difficult.

Using FRP frameworks for implementation is therefore a better idea. Most popular frameworks for FRP are *ReactiveSwift* and *RxSwift* [22]. The *RxSwift* framework is a part of *ReactiveX* family. ReactiveX is a family of libraries for composing asynchronous and event-based programs by using observable sequences. It extends the observer pattern to support sequences of data or events and adds operators that allow you to compose sequences together declaratively while abstracting away concerns about things like low-level threading, synchronization, thread-safety, concurrent data structures, and non-blocking I/O. It was developed by Microsoft Corp. and nowadays is open-source.This API is implemented in many languages such as Swift(RxSwift), Java (RxJava), JavaScript(RxJS), C#(Rx.NET), Python(RxPY). For this thesis, the library *RxSwift* is the most important because out of the ReactiveX family, it is the only one used for developing on Apple platforms. **RxSwift** was, therefore, was chosen over ReactiveSwift for the implementation because the knowledge of RxSwift API can be transferable to other programming languages in which the ReactiveX is supported. The framework is described in the next section 3.6.

3.6 RxSwift

In this section, the main building components of RxSwift are described. A basic knowledge of these components can give an idea, how FRP works. The main components are *Observables*, *Operators* and *Schedulers*. One of the best ways of visualizing the behaviour is using marble diagrams. As illustrated in figure 3.1, a marble diagram shows values plotted on a timeline. The left to right arrow represents time, and the circles with values represent elements of a sequence. Element `x` is emitted and after some time elements `y` and `z` will be emitted.

The time between emission of the values can vary, and it could be at any point in the life of the observable. Every observable has a life-cycle which is further described in the subsection below.

3.6.1 Observables and Subjects

An *Observable* (also know as *observable sequence* or *sequence*) is an object that asynchronously emits a sequence of events that carry values of a given type. [23] An instance of `Observable<T>` allows one or more observers to listen to the events and react on these events in real time. In RxSwift, an *Event* is an enumeration type of 3 possible states:

1. `.next(value: T)` - An event that contains the latest data value. This is how observers can receive the actual data.
2. `.error(error: Error)` - If an `Error` has occured, the `Observable` will emit an *error event* and terminate the sequence. No other `next` events will be emitted after the termination.
3. `.completed` - This event occurs when a sequence terminates sucessfully. It means the `Observable` completed its life-cycle normally and will not emit any other events.

These states determine the life-cycle of an `Observable`. The `.next` event can be emitted any time before the `Observable` is terminated. An `Observable` can terminate in 2 ways `.error` or `.completed`. After the termination of the `Observable` no events can be emitted anymore. The life-cycle is depicted in the figure 3.2

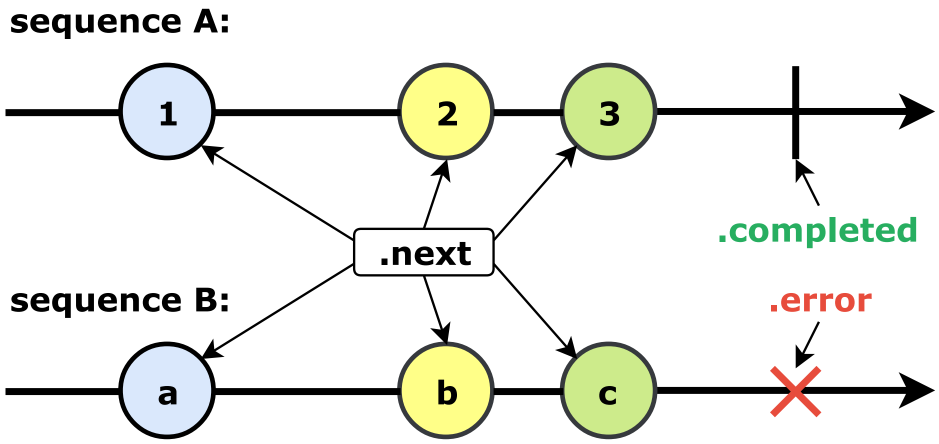


Figure 3.2: Life-cycle of an observable

The events of an `Observable` can be observed by observers using `subscribe(on: (Event<T>)->())` method. After the subscription the observer can then react on the values of the sequence. An example of this action is shown in the code snippet 5.

The `Observable` doesn't emit data until it receives a subscription. The first subscription triggers the sequence to begin emitting events until it is terminated. The subscription can manually be canceled by calling `dispose()` on it or adding the subscription to a `DisposeBag` which cancels the subscription automatically on its deinitialization. If there are no subscriptions on the

```
let sequence = Observable.from(["T", "E", "S", "T", "!"])

let subscription = sequence.subscribe { event in
  switch event {
    case .next(let value):
      print(value)
    case .error(let error):
      print(error)
    case .completed:
        print("completed")
  }
}

OUTPUT:
T
E
S
T
!
completed
```

Listing 5: A subscription example

`Observable`, it terminates automatically. The subscriptions also live until the `Observable` has terminated or until the subscription has been disposed. If the subscription is not disposed and not used anymore, it remains in the memory, and that can lead to *memory leaks*. Therefore adding a subscription to a `DisposeBag` or disposing it manually using `dispose()` is very important to prevent this unwanted effect.

3.6.2 Subjects

Subjects can act as an observable of an observer. [24] That means that it is possible to subscribe to a subject and also dynamically add events to it. There are four diffirent types of `Subjects` in RxSwift:

- `PublishSubject:` When an observer subscribes to this subject, only the events **after** the subscription occurred are received by the observer. The behavior is shown in figure 3.3.

- `BehaviourSubject:` This subject gives any subscriber the last recent element and every event that is emitted by this sequence after the subscription happened. The behaviour can be seen in the figure 3.4.

- `ReplaySubject:` This subject gives the option to replay more than one recent element to new subscribers. The behaviour can be seen in the figure 3.5.

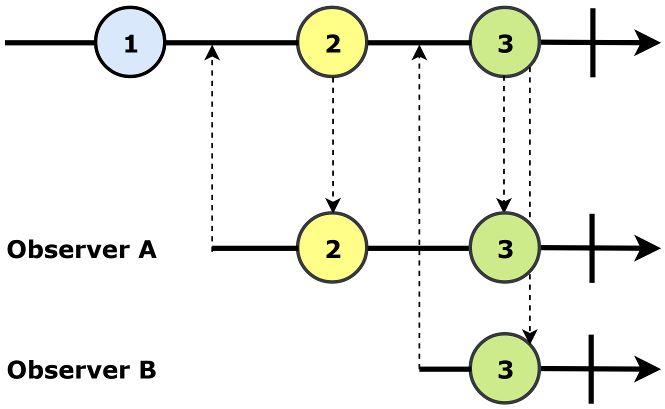


Figure 3.3: Observing a `PublishSubject`

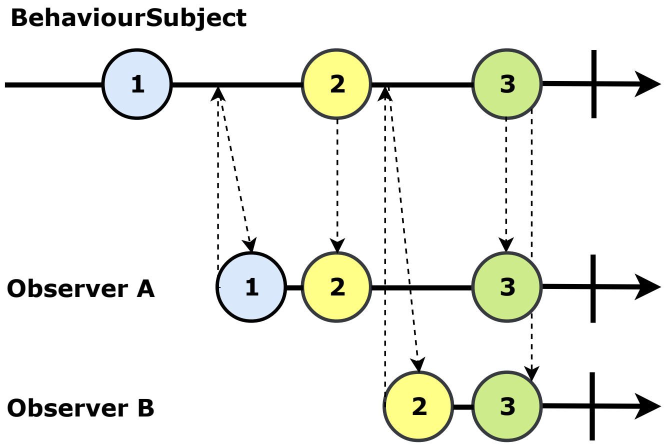


Figure 3.4: Observing a `BehaviourSubject`

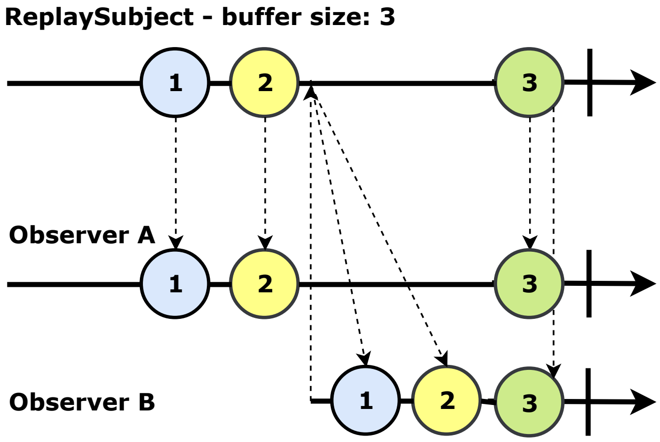


Figure 3.5: Observing a `ReplaySubject`

- `Variable:` A `Variable` only wraps a `BehaviourSubject`, it preserves its current value and replays this value to new subscribers.

3.6.3 Operators

Operators can be used to transform, filter, combine, process and react to events emitted by observables [23]. They don't change the values in the sequence they are applied to, they create a new observable which contains changed values. The operators can be composed together in a chain to express a complex app logic. Currently, there are 74 operators which are not the focus of this thesis therefore only some basic transforming, filtering and combining operators are described. The description of all operators can be found in the ReactiveX online documentation [6] To better understand the output observable after using an operator, an extended marble diagram is used. The observable on the top is the original observable, below this observable is an operator and below the operator is a new observable with the operator applied. If an operator has a function, usually the `$0` refers to the value, that is passed into the function. The extended marble diagram is illustrated in 3.6.

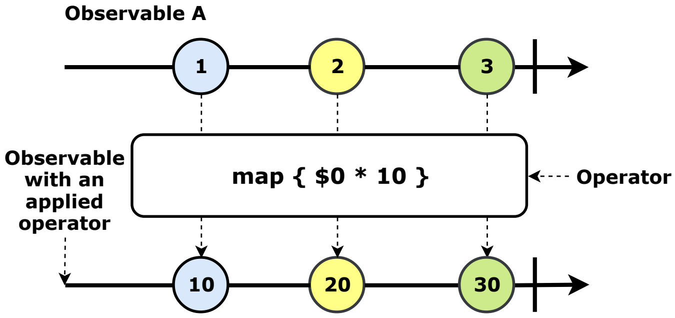


Figure 3.6: A marble diagram with an operator

Transforming Operators

The values coming from the observables may not always be in the format that is needed. By using a transformation operator, a new observable with the needed output data can be created. Two of the most used transforming operators are `map` and `flatMap` which work like Swifts standard `map` and `flatMap` except they operate on observables.

- `map` - this operator takes each emitted event and transforms its value using a *transform function* . An example of the transformation is shown in figure 3.7.
- `flatMap` - this operator can be used when the observable emit other observables, and the values of those observables are needed. The `flatMap` operator merges the emission of these resulting observables and merges them into one sequence. This operator a little bit difficult to understand by reading what it does. The example marble diagram 3.8 and the code

---

[6] `http://reactivex.io/documentation/operators.html`

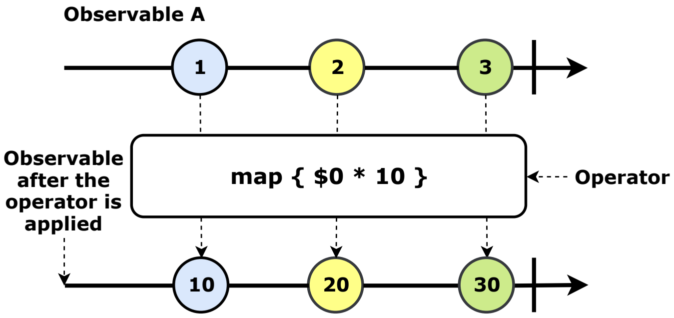


Figure 3.7: An example of `map` behavior

snippet 6 should make the understanding of this operator a little bit clearer.

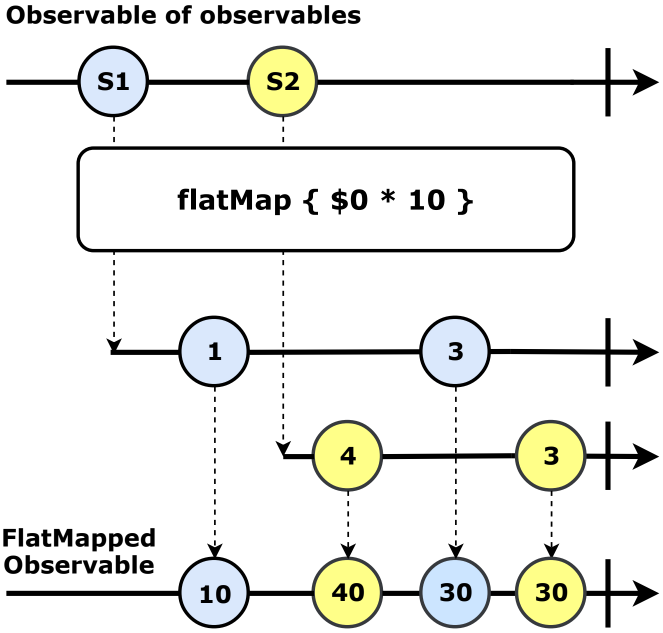


Figure 3.8: An example of `flatMap` behavior

Filtering Operators

Not every eve coming from the observable is useful for the subscriber. Filtering operators are used for passing through only the values that pass through certain criteria to the subscriber. In this part, the main filtering operator `filter` and operator `distinctUntilChanged` are illustrated.

- `filter` - this operator passes through only elements, that fulfill a condition (the result of condition is true). An example of the `filter` operator is shown in figure 3.9.
- `distinctUntilChanged` - this operator passes through an element only

```
let sequence1 = Observable<Int>.of(1, 3)
let sequence2 = Observable<Int>.of(2, 4)

let sequenceOfSequences = Observable.of(sequence1, sequence2)

sequenceOfSequences
  .flatMap { $0.value * 10}
  .subscribe (onNext: {
    print($0)
  })

OUTPUT:
10
20
30
40
```

Listing 6: An example code of `flatMap`

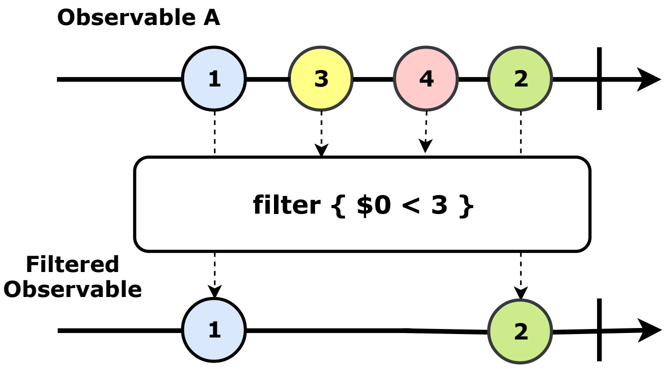


Figure 3.9: An example of `filter` behavior

if the value changed from the previous one. An example of the `filter` operator is shown in figure 3.10.

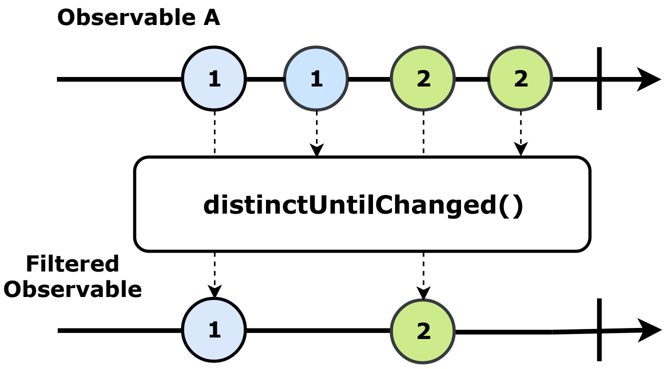


Figure 3.10: An example of `distinctUntilChanged` behavior

Combining Operators

- `merge` - this operator merges the output of multiple observables into a single observable with all emitted events from the individual observables. An example of the `merge` operator is shown in figure 3.11.

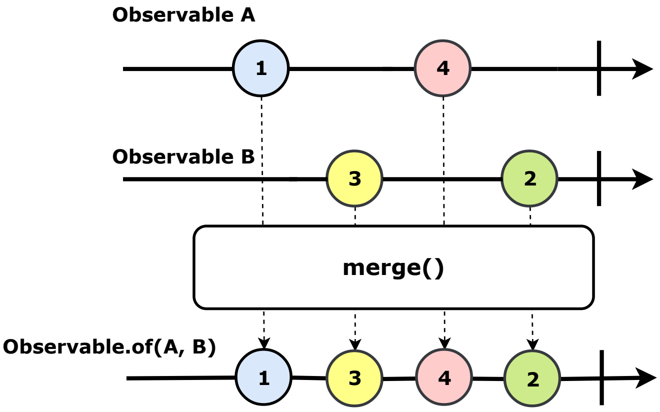


Figure 3.11: An example of `merge` behavior

- `combineLatest` - this operator combines the latest values from multiple observables into a single observable. Each time one of the observables emitts an event, a new combined value is also emitted from the resulting observable. An example of the `combineLatest` operator is shown in figure 3.12.

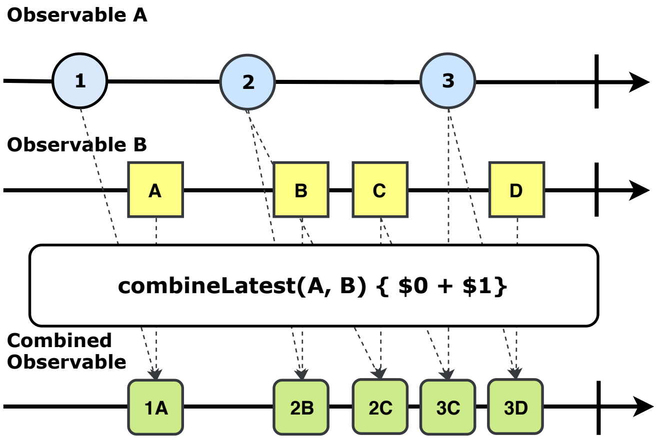


Figure 3.12: An example of `combineLatest` behavior

- `zip` - this operator combines the latest values from multiple observables into a single observable. It operates in strict sequence, meaning that the first combined value emitted by `zip` is emitted after all of the observables emit the first element. Each time a new value is emitted from an observable, `zip` waits until all the observables emit a new value until it emits the combined value. This means that this operator emits as many elements as the number of elements of the source observable with fewest values. An example of the `zip` operator is shown in figure 3.13.

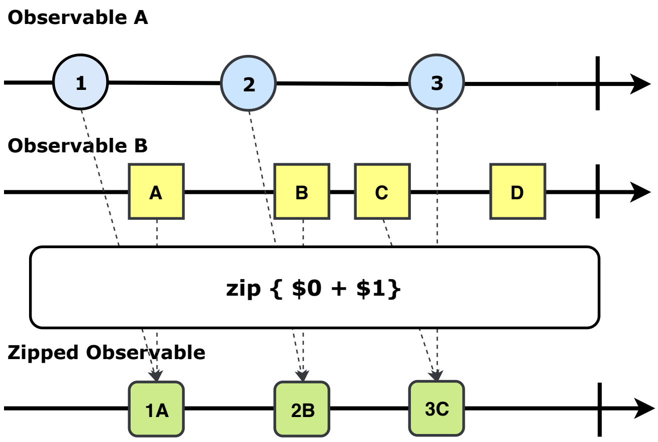


Figure 3.13: An example of `zip` behavior

3.6.4 Schedulers

A scheduler is a context where a process takes place. This context can be a thread, dispatch queue or similar entities [23]. Operators work on the same thread on which the subscription is created unless this behavior is changed. In RxSwift to force operators to do their work on a specific queue, the schedulers are used. The thread of a subscription can also be forced using schedulers. The two main operators for doing this are `observeOn` and `subscribeOn`.

Serial and concurrent schedulers

Because a scheduler is a context, which could be anything (thread, dispatch queue, custom context), and all operators which transform sequences must preserve implicit guarantees, it is necessary to use the right scheduler. There are two types of schedulers - serial or concurrent:

- **serial scheduler** - using this scheduler, RxSwift does the computations serially. When a serial dispatch queue is used, the schedulers perform some optimizations underneath.
- **concurrent scheduler** - RxSwift tries to run the jobs simultaneously. The operators `observeOn` and `subscribeOn` preserve the order in which the tasks need to be performed in order to ensure that the subscription is on a correct scheduler.

Built-in schedulers

These are the 5 built-in schedulers in RxSwift [25]:

- **Serial**

  - `MainScheduler` - this scheduler abstracts the work that needs to be executed on `MainThread`. UI work is usually performed by this scheduler.
  - `CurrentThreadScheduler` - this scheduler schedules units of work on the current thread and is the default scheduler for operators generating elements.
  - `SerialDispatchQueueScheduler` - this scheduler abstracts the work on a serial `DispatchQueue` and is suitable for processing background jobs which are better scheduled serially. This scheduler has several optimizations when using `observeOn`.
- **Concurrent**
  - `ConcurrentDispatchQueueScheduler` - this scheduler abstracts the work on a concurrent `DispatchQueue` and is suitable for multiple, long-running tasks that are performed in the background and need to finish at the same time.
  - `OperationQueueScheduler` - this scheduler abstracts the work on a `NSOperationQueue` and is used when more control over the concurrent jobs. A maximum number of concurrent jobs cah be defined by setting `maxConcurrentOperationCount`.

3.7 Architecture patterns for iOS applications

One part of this thesis is to create an iOS demo application showing the functionality of the library. Nowadays mobile applications are getting more complex and more significant hence architecture patterns are needed for maintainability and reusability of the code. There are more architecture patterns to choose from [26], in this section, only the popular patterns [27] MVC, MVVM and VIPER architecture patterns are described.

3.7.1 MVC

The architecture pattern MVC is based on three components: Model, View, Controller. This architecture pattern is very often used for developing applications with user interfaces [28].

- *Model* defines the data which the application contains and if the model data changes, it notifies the *Controller* or the *View*.
- *View* is presented to the user. It presents the application data and observes the user interaction and notifies the `Controller`
- *Controller* is a layer between the *View* and the *Model*. It is in charge of the logic of the application. It manages the state updates from *Model* to *View* and updates the *Model* based on the interaction of the user on *View* layer.

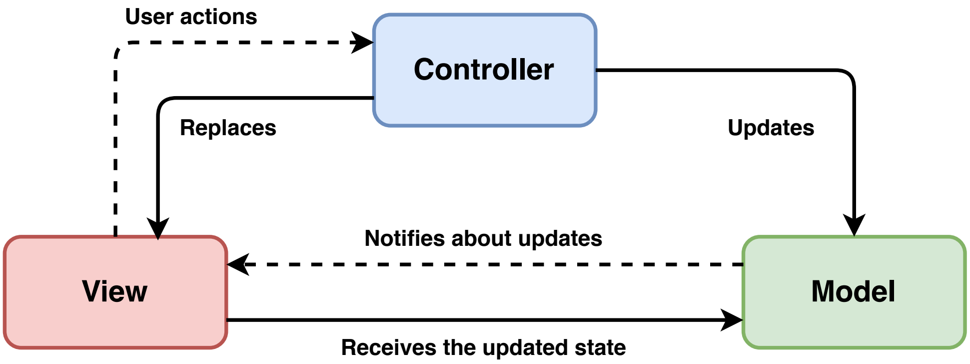


Figure 3.14: Original MVC (originaly taken and recreated from [7])

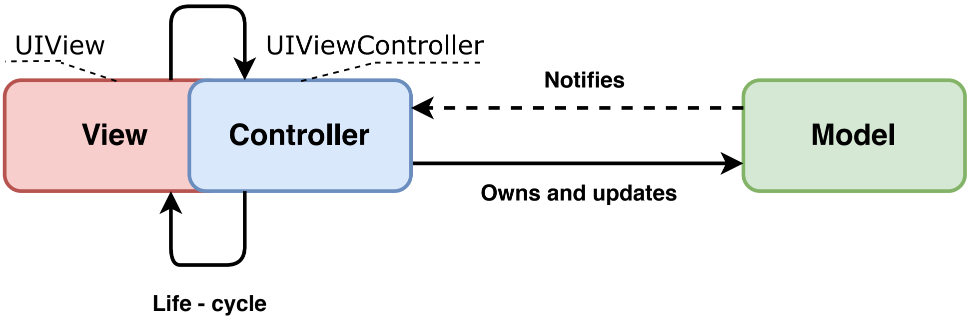


Figure 3.15: Apples MVC (originaly taken and recreated from [8])

Apples form of MVC is a little bit different from the original which can be seen on figures 3.14 and 3.15. The difference is that the *View* and *Model* never communicate with each other directly. This enables the reusability of the *View* without the coupling with the *Model*. On the other hand, *View* and *Controller* are very tightly coupled which brings more code to *Controller* [29] and therefore fails to separate the concerns [26]. This architecture is however good for building small projects because it is easy to learn and doesn't bring much boilerplate code.

3.7.2 MVVM

The MVVM architecture pattern has a similar concept to MVC. This pattern is composed of three components: *Model*, *View* and *ViewModel*, therefore the abbreviation MVVM stands for Model-View-ViewModel. [30]

- *Model* has the same function as in MVC, hence it defines the data the application contains.
- *View* presents the data to the user and forwards user inputs to *ViewModel*. This component contains a minimum amount of application logic and reacts mainly on *ViewModel*. [31]
- *ViewModel* connects view and model and contains the main logic of the

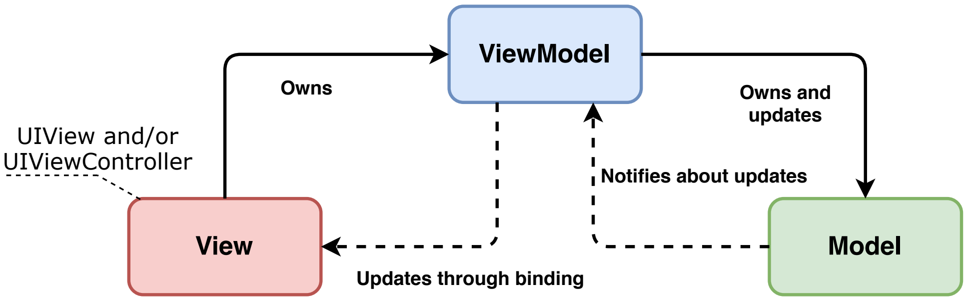


Figure 3.16: MVVM (originaly taken and recreated from [9])

application. It communicates with *Model* and prepares data for *View*. It also reacts on user interaction forwarded from *View*.

When comparing MVC and MVVM architecture patterns used in iOS it is necessary to note, that the implementation of iOS MVC practically has only two components - *View/Controller* and *Model*. A lot of the application **and** presentation logic is contained in the *View/Controller* component, which lead to view controllers with a lot of code (also known as Massive View Controller [32]). The MVVM pattern, the component *View/Controller* is considered to be as one *View* and between this component and *Model* a new component *ViewModel* is added. The *ViewModel* connects the two components and the most of the appliation logic. The architecture pattern is depicted in figure 3.16.

The main benefits of MVVM are followings:

- **Separation of concerns** - The view just presents the data
- **Avoiding Massive View Controllers**
- **Better testability** of the code improves due separation of code into smaller pieces
- **Reusable code**

3.7.3 VIPER

This architecture has a different approach from MVC and MVVM architecture. It is composed of five layers: *View*, *Interactor*, *Presenter*, *Entity* and *Router* which makes the separation of responsibilities more granular.

- *View* presents the data to the user and forwards user inputs to *Presenter*.
- *Interactor* contains the application logic related to the data *Entities*.
- *Presenter* contains the *View* related logic. It reacts on user inputs and communicates with *Interactor* from which it receives updated data.
- *Entities* are plain data structures which can be accessed only by *Interactor*.

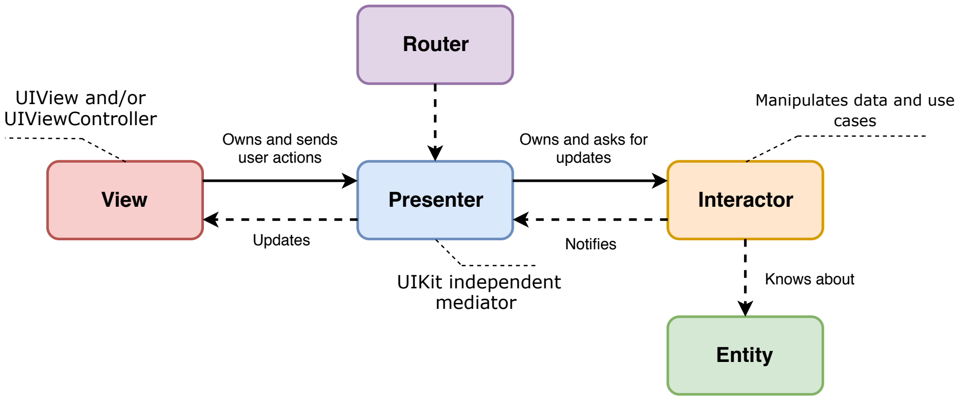


Figure 3.17: VIPER (originaly taken and recreated from [10])

- *Router* is responsible for interaction between the VIPER modules.

The main benefits of VIPER are followings:

- **Separation of concerns**
- **Better testability**
- **Reusable code**

The downsides of VIPER are that it bring a lot of boilerplate code and a lot of complexity to the code. More complex architectures can lead to large development and maintenance overhead in the beginning in the project but it can save time in the future if the application gets bigger. VIPER is not suitable for smaller applications.

3.7.4 Conclusion

The architecture pattern MVVM was chosen for the development of the demo application because it brings separation of concerns and better testability while not being too complex as VIPER architecture. Also MVVM architecture is often used in conjunction with FRP frameworks, because they bring bindings, which is one of the main aspects of MVVM.

3.8 Chapter summary

To sum up this chapter, some key ideas were extracted from the examination of the GitLab API and existing API clients. After reviewing the programming paradigms, the FRP approach was chosen for the implementation because it suits the final products needs. Swift 4.2 was selected as the primary programming language in which the library will be implemented because of many benefits over Objective-C such as performance, safety, and simpler cleaner syntax. As FRP framework, RxSwift was favored over other frameworks and solutions because of the knowledge transferability to other programming languages. This framework was also examined and described to show an example of the FRP concept implementation and to get familiar with the API before the implementation phase. The demo application will follow the MVVM architecture pattern due to its synergy with `RxSwift`.

Chapter 4

Design

This chapter describes the library and demo application design. The section Library design describes how the library is structured and what are the main entities. Application design section covers the architecture design and shows wireframes of the demo application. The library developed during this thesis is further referred to as `RxGitLabKit` and the demo application as `RxGitLabKitDemoApp`. The library is named `RxGitLabKit` because it is created for GitLab, uses reactive extensions (Rx) and the suffix 'Kit' is widely used in iOS framework naming (to name few of them: `UIKit`, `ARKit`, `MapKitCallKit`).

4.1 Library Design

As stated in the analysis part, the GitLab API is divided into groups E.1. The design follows this division and is created with modularity and extensibility in mind.

4.1.1 Structure

The main class of the library is a class called `RxGitLabAPIClient`, which represents the main entry point for using this library. The client provides child classes of *EndpointGroup*. These child classes then offer concrete methods for the communication with GitLab API group. The implemented child classes are the following:

- `AuthenticationEndpointGroup`
- `RepositoriesEndpointGroup`
- `UsersEndpointGroup`
- `ProjectsEndpointGroup`
- `CommitsEndpointGroup`

`HostCommunicator` is used for the underlying HTTP communication with the GitLab API Server. Some of the of the methods of the `EndpointGroup` child classes return a `Paginator` which is used when there is a larger amount of returned objects in a list. For a clearer picture a simplified class diagram is presented in figure 4.1 and the extended class diagram can be seen in the appendix section D.1.

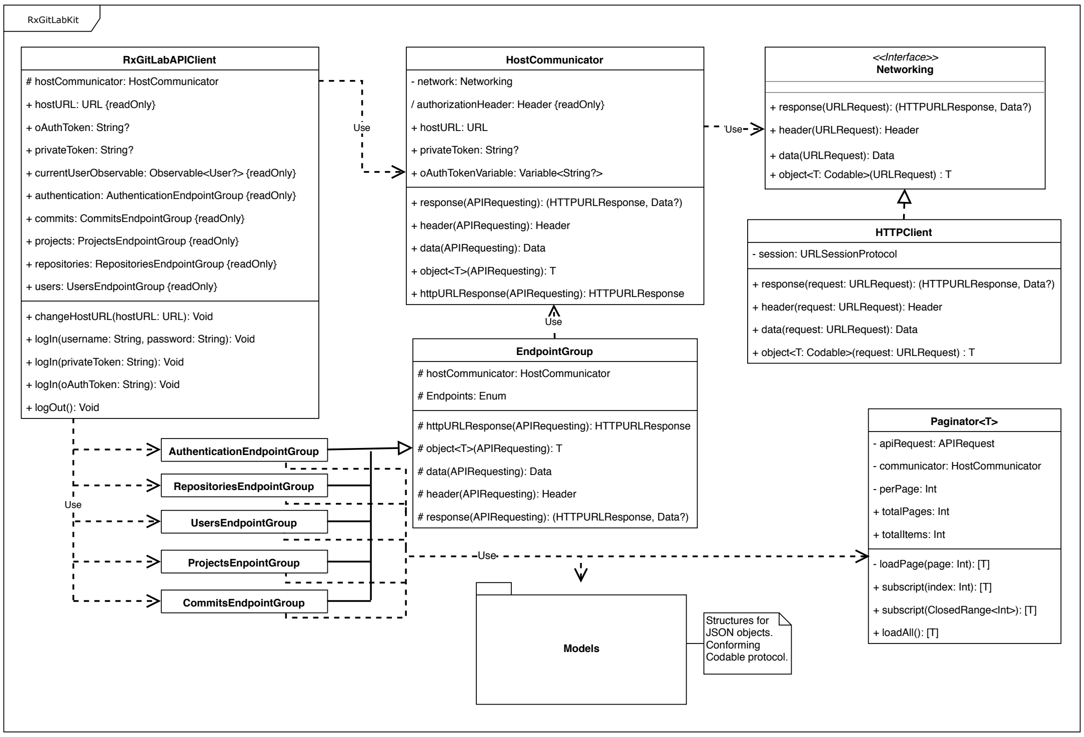


Figure 4.1: Simplified diagram of `RxGitLabKit`

HostCommunicator

All HTTP communication with GitLab API host goes through this class. It uses a `HTTPClient` for HTTP communication and contains the host URL, private and OAuth token for authenticated communication.

HTTPClient

This class provides the following basic networking functions:

- `response(for request: URLRequest)`
- `header(for request: URLRequest)`
- `data(for request: URLRequest)`
- `object(for request: URLRequest)`

These functions are wrapped by a Rx extension so that the functions return an `Observable` to which can be then subscribed to.

RxGitLabAPIClient

This is the root class of `RxGitLabKit` library. This class acts as a hub for all `EndpointGroup` classes and is responsible for authentication. An instance of this class can be created with a GitLab host URL and a private or OAuth token. If no token is provided, a manual authorization using an username and password (`func logIn(username: String, password: String)`) must be then called which results in acquiring an OAuth token from the server. This OAuth token is then used for the authorized communication between the host and the library.

EndpointGroup

An `EndpointGroup` is a superclass for all endpoint groups. The children of this class provide the endpoint URLs for a given endpoint group and related methods for communication with those API endpoints. For the communication with GitLab API server, an instance of `HostCommunicator` is used. Most of the functions return an `Observable` of the desired objects, allowing further asynchronous processing. Some of the functions return a `Paginator` which deals with pagination of endpoints that can return a large amount of items.

Paginator

It communicates with a concrete endpoint and uses parameters `page` and `perPage` to retrieve the desired page from the server. Paginator is used when the concrete endpoint can provide a large amount of objects and only a part of the objects are needed. `Paginator` also provides a function `loadAllItems` which concurrently loads all items from the given endpoint.

4.1.2 API Definition

As found in the analysis chapter 3.3.3, the basic usage of this library should look like this:

```
let hostURL = URL(string: "gitlab.test.com")!
let client = RxGitLabAPIClient(with: hostURL,
  privateToken : "mockprivtkn12345")
let commitObservable = client.commits.getCommits(projectID: 10)
commitObservable
    .subscribe(onNext: { commits in
        // do something with commits
    },
    onError: { error in
        // do something with the error if it occures
    })
```

4.2 Demo application design

In this section, the demo application UI and functionality is described. The design of the screens is following the common practices of MVVM architecture discussed in section 3.7.2, and for this reason, it is not described in this part.

The design follows requirements stated in section 2.2 and common practices of UI development taken from Apples Human Interface Guidelines [33].

4.2.1 Screens

First, it must be considered which screens should this application be consisted of and then decide how to transition between these screens. If two or more screens can transition between each other, they are further referred to be in one *"navigation stack"*. In summary, the user must be able to log in, see a table of repositories, in those repositories, see a list of commits and then be able to see a detail of a commit.

In total the application must have these screens:

- Log In
- User detail with log out option
- List of repositories
- List of commits
- Commit detail

The screen navigation stacks of the application can be divided into two:

1. List of repositories <-> List of commits <-> Commit detail
2. Log In <-> User Detail

For multiple navigation stacks using a Tab navigation stack is a suitable choice because it organizes information at the app level and the navigation stacks are then easily accessible [34].

The first tab consists of two lists and then one detail. For this usage, there is a suitable component called `UISplitViewController` which shows a list of objects on one part of the screen and the detail on the second part. This works only on devices with higher resolution such as iPads and on smaller devices (iPhones) it works like a normal `UINavigationController`. A table component is fitting for showing a list of objects or a list of information. All the screens in the first tab therefore include a table to show the information.

The second tab manages the logged in user and the GitLab API server to be used. If the user is not logged in, the Log In screen is shown first, User Detail screen is shown otherwise.

List of repositories

This screen shows a list of repositories the user has access to. It can also show a list of public repositories. The user has a choice to show the list of public repositories or to show his repositories. The user can also use a search bar to filter the results.

After taping on a repository name a new screen with a list of commits of that repository is shown.

List of commits

This screen shows a list of commits of the previously selected repository. The user must have privileges in order to see the commits.

After taping on a commit name a new screen with a commit detail is shown.

Commit detail

Commit detail shows the information about the commit.

Log In

On this screen, there is an input for username, password, and a GitLab URL. The user can also input a private token or OAuth token for authorization. The last component is a button which initiates the login process.

The logic of this screen is to take take the user input, try to log in with the information given by the user, and if successful, transition to the User Detail screen. If unsuccessful, show an Error message.

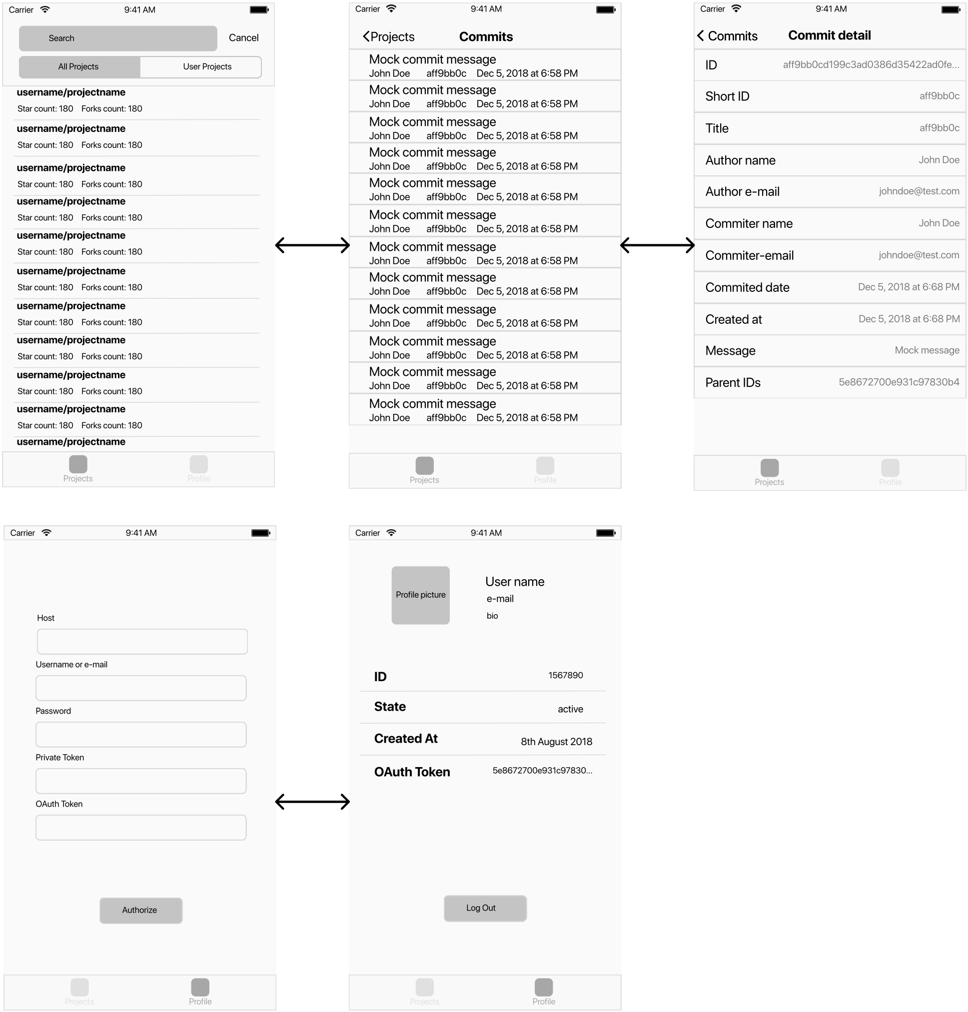
User Detail

This screen shows the most important user information like username, OAuth or Private Token, e-mail address. The user details are shown in a table component.

Furthermore, it shows a log out button which initiates a logout process when pressed. After logging out, this screen transitions to Log In screen.

4.2.2 Application Transitions

The figure 4.2 shows wireframes for iPhone size and also shows all transitions in the application. The wireframes of the iPad version is shown in figure 4.3. The wireframes for iPad don’t include the Login / Profile section because the layout is the same, only it is displayed on a bigger screen.



username/projectname
Star count: 180 Forks count: 180

username/projectname
Star count: 180 Forks count: 180

username/projectname
Star count: 180 Forks count: 180

username/projectname
Star count: 180 Forks count: 180

username/projectname
Star count: 180 Forks count: 180

username/projectname
Star count: 180 Forks count: 180

username/projectname
Star count: 180 Forks count: 180

username/projectname
Star count: 180 Forks count: 180

username/projectname
Star count: 180 Forks count: 180

username/projectname
Star count: 180 Forks count: 180

username/projectname

Mock commit message
John Doe aff9bb0c Dec 5, 2018 at 6:58 PM

Mock commit message
John Doe aff9bb0c Dec 5, 2018 at 6:58 PM

Mock commit message
John Doe aff9bb0c Dec 5, 2018 at 6:58 PM

Mock commit message
John Doe aff9bb0c Dec 5, 2018 at 6:58 PM

Mock commit message
John Doe aff9bb0c Dec 5, 2018 at 6:58 PM

Mock commit message
John Doe aff9bb0c Dec 5, 2018 at 6:58 PM

Mock commit message
John Doe aff9bb0c Dec 5, 2018 at 6:58 PM

Mock commit message
John Doe aff9bb0c Dec 5, 2018 at 6:58 PM

Mock commit message
John Doe aff9bb0c Dec 5, 2018 at 6:58 PM

Mock commit message
John Doe aff9bb0c Dec 5, 2018 at 6:58 PM

Mock commit message
John Doe aff9bb0c Dec 5, 2018 at 6:58 PM

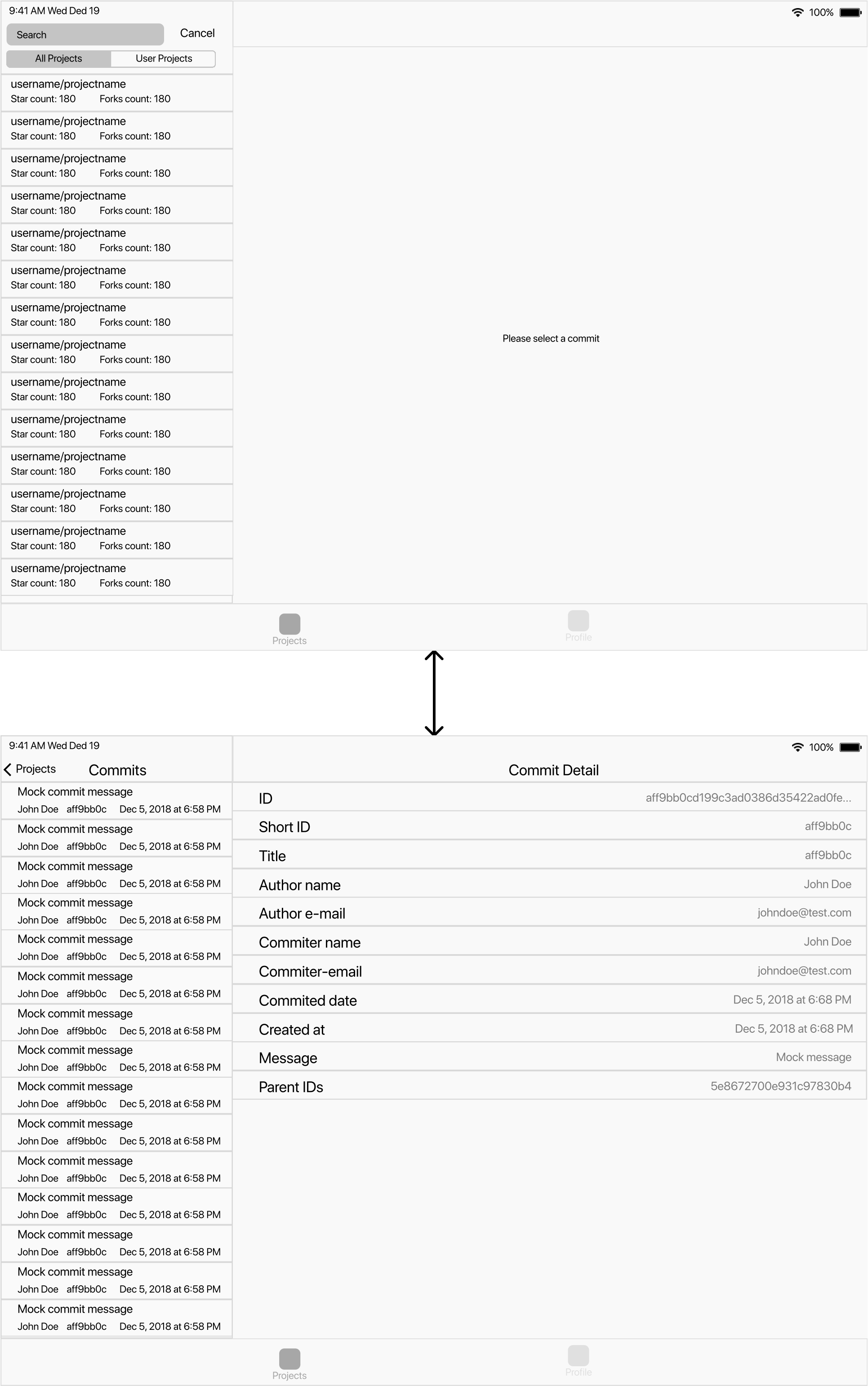
Mock commit message
John Doe aff9bb0c Dec 5, 2018 at 6:58 PM

ID	aff9bb0cd199c3ad0386d35422ad0fe...
Short ID	aff9bb0c
Title	aff9bb0c
Author name	John Doe
Author e-mail	johndoe@test.com
Commiter name	John Doe
Commiter-email	johndoe@test.com
Commited date	Dec 5, 2018 at 6:68 PM
Created at	Dec 5, 2018 at 6:68 PM
Message	Mock message
Parent IDs	5e8672700e931c97830b4

User name
e-mail
bio

ID	1567890
State	active
Created At	8th August 2018
OAuth Token	5e8672700e931c97830...

Figure 4.2: iPhone wireframes



9:41 AM Wed Ded 19

100%

Search

Cancel

All Projects

User Projects

username/projectname
Star count: 180 Forks count: 180

username/projectname
Star count: 180 Forks count: 180

username/projectname
Star count: 180 Forks count: 180

username/projectname
Star count: 180 Forks count: 180

username/projectname
Star count: 180 Forks count: 180

username/projectname
Star count: 180 Forks count: 180

username/projectname
Star count: 180 Forks count: 180

username/projectname
Star count: 180 Forks count: 180

username/projectname
Star count: 180 Forks count: 180

username/projectname
Star count: 180 Forks count: 180

username/projectname
Star count: 180 Forks count: 180

username/projectname
Star count: 180 Forks count: 180

username/projectname
Star count: 180 Forks count: 180

username/projectname
Star count: 180 Forks count: 180

Please select a commit

Projects

Profile

9:41 AM Wed Ded 19

100%

Projects

Commits

Mock commit message
John Doe aff9bb0c Dec 5, 2018 at 6:58 PM

Mock commit message
John Doe aff9bb0c Dec 5, 2018 at 6:58 PM

Mock commit message
John Doe aff9bb0c Dec 5, 2018 at 6:58 PM

Mock commit message
John Doe aff9bb0c Dec 5, 2018 at 6:58 PM

Mock commit message
John Doe aff9bb0c Dec 5, 2018 at 6:58 PM

Mock commit message
John Doe aff9bb0c Dec 5, 2018 at 6:58 PM

Mock commit message
John Doe aff9bb0c Dec 5, 2018 at 6:58 PM

Mock commit message
John Doe aff9bb0c Dec 5, 2018 at 6:58 PM

Mock commit message
John Doe aff9bb0c Dec 5, 2018 at 6:58 PM

Mock commit message
John Doe aff9bb0c Dec 5, 2018 at 6:58 PM

Mock commit message
John Doe aff9bb0c Dec 5, 2018 at 6:58 PM

Mock commit message
John Doe aff9bb0c Dec 5, 2018 at 6:58 PM

Mock commit message
John Doe aff9bb0c Dec 5, 2018 at 6:58 PM

Mock commit message
John Doe aff9bb0c Dec 5, 2018 at 6:58 PM

Mock commit message
John Doe aff9bb0c Dec 5, 2018 at 6:58 PM

Commit Detail

ID	aff9bb0cd199c3ad0386d35422ad0fe...
Short ID	aff9bb0c
Title	aff9bb0c
Author name	John Doe
Author e-mail	johndoe@test.com
Commiter name	John Doe
Commiter-email	johndoe@test.com
Commited date	Dec 5, 2018 at 6:68 PM
Created at	Dec 5, 2018 at 6:68 PM
Message	Mock message
Parent IDs	5e8672700e931c97830b4

Projects

Profile

Figure 4.3: iPad wireframes

Chapter 5

Implementation

In this chapter, the implementation of the library and the demo application is described, and some interesting parts of the code are shown. The development of the *RxGitLabKit* followed the TDD principles. Networking, object parsing, and pagination are shown as the interesting parts of the code.

The *RxGitLabKitDemoApp* describes the implementation of the MVVM architecture with the *RxGitLabKit* and the usage of `RxSwift` and `RxCocoa`.

The code was developed in Xcode 10.1 which is the latest version of Xcode at the time of writing this thesis. Xcode is the main Integrated Development Environment (IDE) for developing software for macOS, iOS, watchOS, and tvOS.

5.1 RxGitLabKit implementation

5.1.1 Networking

One of the most popular library for networking in Swift is `Alamofire` with over 29 thousand stars and over 5 thousand forks on GitHub. This library provides a lot of features like chainable request/response methods, URL / JSON/ plist parameter encoding, authentication with URLCredential, HTTP Response Validation and many more.[35]. This library does not support Reactive extensions in its core, but there is a library called *RxAlamofire* that wraps *Alamofire* in order to use the benefits of `RxSwift`. These libraries provide some of the functionalities needed for GitLab API library, however, adding these libraries increases the number of dependencies and brings much unused code to this project.

Therefore a lightweight custom networking layer was created using native Cocoa library components and wrapped with `RxSwift` in order to minimize dependencies and provide a reactive networking component for this project.

Custom networking layer

The custom networking layer was created using Cocoa core libraries and `RxSwift`. That makes this layer lightweight and depends only on `RxSwift` which is used in most parts of this project. Cocoa networking components allow the code to communicate over the network and `RxSwift` makes this communication reactive. The main classes for networking in Cocoa library are `URL`, `URLRequest/HTTPURLResponse`, `URLSession`, `HTTPURLResponse` and `Data`.

- `URL`

  The `URL` class represents a local or remote URI. It can be anything from a local file to an HTML webpage. An `URL` in Swift is specified like this:

```
let httpURL: URL? = URL(string: "https://gitlab.test.com")
```

  Note that the initializer returns an optional value, because the string provided to the initializer can be in a non-compatible format such as `URL(string:"5-a.b.c")`, which results in returning `nil`.

- `URLRequest`

  The `URLRequest` represents a request for an URL, and its instance has an HTTP method (GET, POST, DELETE ...), a body and a header. A definition of an URLRequest for a POST request sending and asking for a JSON response is depicted in the following example:

```
let postURL = URL(string: "https://gitlab.com/oauth/token")!
var postRequest = URLRequest(url: postURL)
postRequest.httpMethod = "POST"
postRequest.setValue("application/json",
  forHTTPHeaderField: "Content-Type")
postRequest.httpBody = try? JSONSerialization
  .data(withJSONObject:
    ["grant_type" : "password",
     "username" : "mockUserName",
     "password" : "mockPassword123"])
```

- `URLSession`

  The `URLSession` is a class that performs an `URLRequest`. In most cases a singleton `URLSession.shared` is used. The most important method is `func dataTask(with:completionHandler:)`. This method takes the `URLRequest` and handles it by a defined completion handler. The basic usage of getting the content of a website looks like the followings:

```
let httpURL = URL(string: "https://gitlab.com")!
let httpTask = URLSession.shared.dataTask(with: httpURL) {
  (data, response, error) in
  guard let validData = data,
   let results = String(data: validData, encoding: .utf8),
   error == nil else {
    print("Error getting GitLab website")
    return
  }
}
httpTask.resume()
```

  First an `URL` is defined, then an `URLSessionDataTask` is created. The completion handler has response parameters `(Data?, URLResponse?,` `Error?)` as input which represent an actual response from the server. All inputs are optional, meaning that the data, response or error can be `nil`, which is needed when for example an valid response is returned, there should be no error returned. The `URLSessionDataTask` is not invoked upon creation, for running the task the function `resume()` must be called.

- `Response`

  The response from server is represented by 3 variables of these types: `Data`, `URLResponse` and `Error`. The `Data` represents body of the response. Usually, the data is transformed into a string or being used for JSON parsing. The `URLResponse` represents a response header from server. It contains response headers, status code and the URL of the request. The `Error` represents an error response, for example when the server does not respond.

- `Reactive wrapper`

  As seen in the code of 5.1.1, the code is not reactive. The program state can be only changed in the `dataTask` completion handler. Reactive wrapper allows the developer to observe the response and when it comes, react upon it and change the state. One of the benefits is that there can be more observers that can react to the same action without changing the completion handler.

  The main idea is to create an `Observable` which pushes `URLResponse` and `Data` from `dataTask` completion handler. The wrapped function looks like this:

```
func response(for request: URLRequest)
  -> Observable<(response: HTTPURLResponse, data: Data?)> {
  return Observable.create { observer in
    let task = URLSession.shared.dataTask(with: request)
    { (data, response, error) in
      guard let response = response else {
        observer.on(.error(error ?? HTTPError.noResponse))
        return
      }
      observer.on(.next((httpResponse, data)))
      observer.on(.completed)
    }

    task.resume()
    return Disposables.create(with: task.cancel)
  }
}
```

5.1.2 Data models and parsing

Data models were created using the provided JSON examples from GitLab API docs and instead of manually creating the models from JSON format, quicktype.io and json4swift.com were used. These applications allow the user to insert a JSON and it creates a Swift struct. The generated code is not perfect and needed to be refactored, but overall using these tools saved a lot of time when creating the models. The data models are structs and conform to Codable protocol. Codable protocol allows easy JSON encoding and decoding. Let's assume that an object of type `Commit` is needed to be decoded and encoded.

```
// Decoding
let commitData =  ... // some commit data
let decoder = JSONDecoder()
let commit = try? decoder.decode(Commit.self, from: commitData)

// Encoding
let commit = Commit(....)
let encoder = JSONEncoder()
let commitData = encoder.encode(commit)
```

5.1.3 Paginator

Paginator is a class that handles pagination of the GitLab API results. It uses an instance of `APIRequest` to create the desired `URLRequest` and uses `HostCommunicator` as a communicator with GitLab API. It provides variables to determine the total number of pages and the total number of items. These values must be fetched from the GitLab API and are included in the response header. The interesting part of this class is using a `subscript` (explained in the analysis chapter 3.2.2) to fetch a page or multiple pages. Fetching pages from 2 to 5 is shown in the following code:

```
let paginator = client.users.getUsers()
let usersObservable = p[2...5]
usersObservable.subscribe(onNext: { users in
  // do something with users
})
```

The implementation of the subscript uses the potential of the `RxSwift` and its operators. The basic idea is to take a range of numbers and for each number create a request for that page. Then merge all results into one array. Fetching the responses for each page is asynchronous, and the order of the responses is not guaranteed. Therefore a page number for the request is added to later be used for sorting of the result. The final stage of the pipe is to get the values from observables. This is done using `.flatMap`. The code of the implementation is shown here:

```
public subscript(range: Range<Int>) -> Observable<[T]> {
  let arrayOfObservables: [Observable<(Int, [T])>] = range
    .map { page in self.loadPage(page: page).map {(page, $0)}}

  let mergedObjects: Observable<[T]> = Observable
    .zip(arrayOfObservables)
    .map { arrayOfTuples -> [(Int, [T])] in
      arrayOfTuples.sorted(by: { (lhs, rhs) -> Bool in
        return lhs.0 < rhs.0
      })}
    .map { arrayOfTuples -> [T] in
      arrayOfTuples.flatMap {$0.1}
    }

  return mergedObjects
}
```

Loading all pages is then simple to implement. From total pages create a range, and then the subscript is used. The code is depicted here:

```
public func loadAll() -> Observable<[T]> {
    return totalPages.flatMap { $0 > 1 ? self[1...$0] : self[1] }
}
```

5.2 Demo application implementation

The demo application implementation follows the MVVM architecture chosen in the analysis part 3.7.4 and is based on the Demo application design section 4.2. It also uses `RxSwift` and `RxCocoa` for data binding between view models into views/view controllers. The UI of the application can be done in a GUI using *Storyboard* or to write the UI in code. Although the *Storyboard* approach gives a faster visual feedback when designing the screens, it is not very flexible. Because of the lack of flexibility, the UI is created programmatically in code.

5.2.1 UI element positioning

In iOS, the UI elements can be positioned using auto layout. Auto layout dynamically computes the dimensions and the positions of the views in the view hierarchy. These computations are based on constraints that are placed on the views [36]. For example, a constraint on an image can be placed so that the image is centered with its parent view and the edges of the images are inset by 16 points. If the parent view of the image changes size, the image size also automatically adjusts in order to meet the constraints. This is also very beneficial when creating a UI for more screen sizes. If the constraints are set correctly, the application will look the same on devices with different screen sizes. This option reduces the development time when creating applications for iPhone or iPad. However, the work with native `UIKit` constraints is not intuitive a library called *SnapKit* was used. SnapKit is a DSL to make Auto Layout on iOS and macOS easy. [37]. It provides an intuitive API and makes creating UI much faster.

5.2.2 Screen description

The screens are represented by a base `UIViewController` class included in `UIKit` (`UIKit` is a library for creating UI for iOS applications [38]). Because the application has two navigation stacks as discussed in the design section 4.2 a `UITabBarController` was chosen as the root view controller. Each navigation stack has its base component. The navigation stack which includes repositories, commits and commit detail, has a `UISplitViewController` as a base component. `UISplitViewController` is a container view controller that handles two child view controllers as master-detail. Two view controllers can be arranged side-by-side. Usually, the master view controller contains a list of objects and the detail view controller details about the object. The object can be selected in the master view controller, and the detail is shown in the detail view controller. The benefit of using `UISplitViewController` is that the user does not need to navigate back from the detail and then select another object. On smaller devices, only one view controller is visible [39]. The difference is depicted in the figures 4.2 and 4.3. The second navigation stack has its first view controller wrapped in a `UINavigationController` because it enables easy transitions between the screens in a navigation stack.

Each screen is implemented as a view controller with its view model. Only Projects screen has a model class, the rest of screens don't have a separate model, because it consists of one object or one list of objects. Therefore the model is directly included in the view model. The implemented screens are the followings:

- `Projects`

  The `ProjectsViewController` is composed from `UITableView` and a `UISearchBar`. The `UITableView` shows a list of the projects and the `UISearchBar` takes text as an input and forwards the input into the `ProjectsViewModel`. The `UISearchBar` also provides two scopes - *All Projects* and *User Projects*. Switching between these two scopes enables the user to select whether he wants to show all projects or just the projects he has on his profile. Based on the input, the `ProjectsViewModel` fetches the projects with the search term and scope and shows the projects names in the table view. Fetching is done using `RxGitLabAPIClient`. After taping on the project table cell, a `CommitsViewController` is shown.

- `Commits`

  The `CommitsViewController` is composed from just `UITableView` which shows a list of the commits. The commits are fetched in the `CommitsViewModel` and shown in this table view. When the commit table cell is tapped, a `CommitDetailViewController` is shown.

- `CommitDetail`

  The `CommitDetailViewController` is composed just from `UITableView` which shows the information about the commit. The `CommitsDetailViewModel` fetches more details about the commit, prepares them into a presentable format and forwards the data into the table view.

- `Login`

  The `LoginViewController` contains input fields and a log in button. After the button is tapped, the data from input fields are forwarded to `LoginViewModel` which then initiates the authorization process. If the login went successfully, a `ProfileViewController` with user details is shown. An alert is shown otherwise.

- `Profile`

  The `ProfileViewController` is similar to `CommitsViewController` because it is also composed from `UITableView` which shows the information about the user. There is a log out button which when tapped, forwards the action to `ProfileViewModel` which clears the user details and sends the signal back to `ProfileViewController` which then closes while showing the `LoginViewController`.

5.2.3 RxSwift in MVVM

This subsection shows an example of MVVM architecture and `RxSwift` usage on Projects screen. This screen was selected as an example because from all screens, it is the most complex one. `RxSwift` helps with data binding and the propagation of change. The object structure of this screen is illustrated in the figure 5.1.

For communication between the `UITableView`/`UISearchBar` and the view controller uses the delegate pattern [40]. The setup of the delegate pattern requires the view controller to be expanded by delegate methods and then be assigned to the respective views. On the other hand `RxSwift` allows to use the methods directly without the need of this setup. A sample code showing the difference of setting up connection between a `UITableView` and a `ProjectsViewController` is illustrated in code 8 and 9. The difference of the whole setup is shown in code listings 14 and 13 in the appendix section. As can be observed from the code samples, using `RxSwift` for data binding removes much boilerplate code and makes the code easier to write and read. It is also worth noting, that when the data source is changed, in the version without `RxSwift` the table view must be reloaded by calling `tableView.reloadData()` but with the usage of `RxSwift` binding, the table view updates automatically.

An another example of `RxSwift` usage is using operators. The search bar sends a new text every time the value has changed, which triggers a request to the server. This behavior can overflow the server with requests which is undesired. An operator `throttle(dueTime:)` can be used to take the

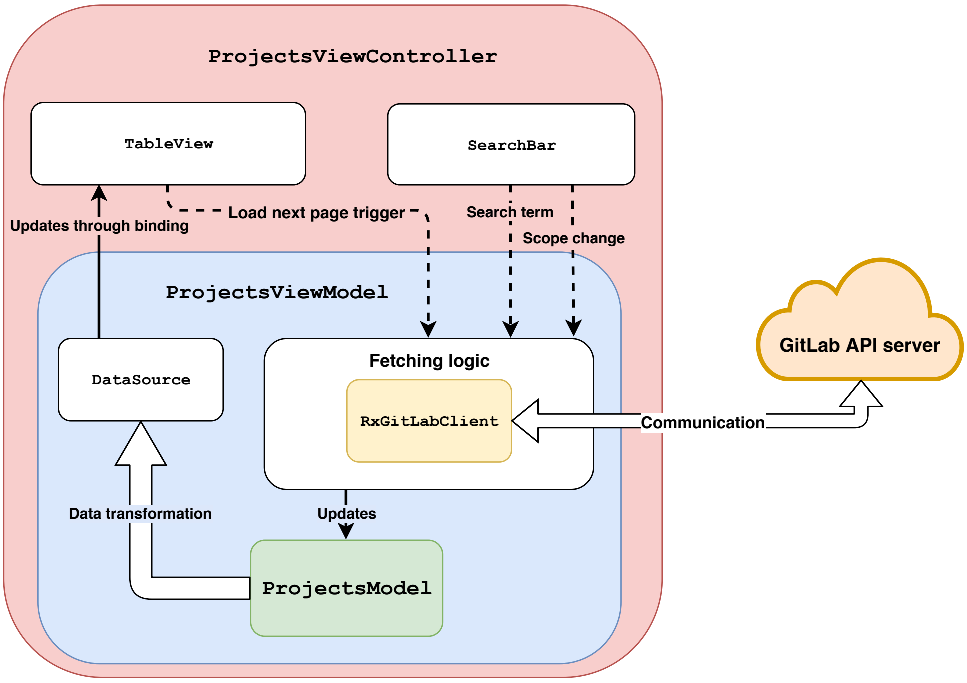


Figure 5.1: Projects screen objects structure

change in the interval specified by the `dueTime` argument which can reduce the number of requests. The sample code is shown in listing 7

```
// Text search
let searchTextObservable = searchTextVariable.asObservable()
  .throttle(0.2, scheduler: MainScheduler.instance)
  .subscribe(onNext: { searchText in
  // Fetch data using textSearch
  }
```

Listing 7: Example usage of `throttle` operator

5.3 Dependency management

Using third party libraries and frameworks can speed up the development and reduce the costs. When the application requires more libraries and each library has other dependencies, it can be to maintain the dependencies up to date. For this reason, using a dependency manager is nowadays a part of development. A dependency manager is a tool for automated declaration and resolution of dependencies required by the project [41]. The three dependency managers for Swift used today are *CocoaPods*, *Carthage* and *Swift Package Manager (SPM)*. Because the main focus of this thesis is to create a library, which can be used in other projects, allowing other developers to integrate this library into their application using a dependency manager is essential. Support for mentioned dependency managers was added. This section introduces the three dependency managers, briefly describes how they work and then shows how to integrate the `RxGitLabKit` can be integrated into other projects using the manager.

5.3.1 CocoaPods

CocoaPods[1] is a third-party centralized dependency manager for Swift and Objective-C projects. It was the first dependency manager for iOS, and at at the time of writing this thesis, it is most widely used. [42] CocoaPods can simply be set up by creating a *Podfile* which contains the list of the dependencies and then run `pod install` in the Terminal and it creates a new *Xcode Workspace* file that contains the project and all other dependencies linked and ready for usage. Although the set up is uncomplicated, the disadvantage is that it modifies the project files in a non-transparent manner, which can make future changes to the project structure difficult. These changes are however not very common, that is why this con is acceptable.

To integrate the RxGitLabKit using CocoaPods, these steps must be taken:

1. Adding the following code to Podfile

```
# Podfile
use_frameworks!

target 'YOUR_TARGET_NAME' do
  pod 'RxGitLabKit'
end
```

2. Replacing `YOUR_TARGET_NAME` with the target name
3. Run `pod install` in the terminal

5.3.2 Carthage

Carthage [2] is a third-party decentralized dependency manager and currently supports these dependency sources: Git public open source repositories and binary links using public HTTPS [43]. The set up consists of creating a *Cartfile* with a list of dependencies and running `carthage update`. Carthage clones the repository and builds the code locally or downloads the binary if it is provided. The binaries must be then manually added to the project. In comparison to CocoaPods, Carthage does not change the project structure, but a manual binary linking is needed. This dependency manager was used when creating the RxGitLabKit library because of the cleaner approach to project structure.

---

[1] https://cocoapods.org/

[2] https://github.com/Carthage/Carthage

Adding Carthage support for RxGitLabKit is simple. The code must be publicly accessible on a git repository, and the developers need to the following line add into the Cartfile and run `carthage update` in a terminal.

```
git "https://gitlab.com/dagytran/RxGitLabKit.git"
```

5.3.3 Swift Package Manager

SPM [3] is a tool for managing the distribution of Swift code. It is an official dependency manager, but currently supports only macOS platform [44]. The installation of the RxGitLabKit using SPM follows these steps:

1. Creating a `Package.swift` file with the following code:

```
// swift-tools-version:4.2
import PackageDescription

let package = Package(
  name: "YOUR_PROJECT_NAME",
  dependencies: [
    .package(url: "https://gitlab.com/dagytran/RxGitLabKit.git",
      from: "1.0.0")
  ],
  targets: [
    .target(name: "YOUR_PROJECT_NAME",
      dependencies: ["RxGitLabKit"],
      path: "SOURCE_PATH")
  ]
)
```

2. Replacing `YOUR_PROJECT_NAME` with the target name and `SOURCE_PATH` with the sources path name
3. Run `swift build` in the terminal
4. Run `swift package generate-xcodeproj` in the terminal

5.4 Documentation

The code was documented using the official recommendations. Also a `README.MD` file containing basic information was added. The documention in HTML format is included on the CD in folder `documentation` and is also provided online on `https://dagytran.gitlab.io/RxGitLabKit/`. The documentation was generated using Jazzy [4].

---

[3] https://swift.org/package-manager/

[4] https://github.com/realm/jazzy

5.5 Chapter summary

In summary, the implementation followed the ideas stated in the design chapter 4 and used TDD principles. The main components of `RxGitLabKit` were thoroughly described, and some sample code was shown. The components of screens in `RxGitLabKitDemoApp` were described and an example usage of `RxSwift` in MVVM architecture was shown. After the implementation, the support for dependency managers was added, and the HTML documentation generated using Jazzy was released online and is also included on the CD in the `documentation` folder.

```
class ProjectsViewController: UIViewController, UITableViewDelegate,
  UITableViewDataSource, UISearchBarDelegate {
  // ... ViewController setup ...
  override func viewDidLoad() {
    super.viewDidLoad()
    // ... view setup code - adding, layout etc. ...
    searchBar.delegate = self
    tableView.dataSource = self
    tableView.delegate = self
    tableView.register(ProjectsTableViewCell.self,
      forCellReuseIdentifier: "ProjectsCell")
  }

  // UITableViewDataSource delegate functions
  func numberOfSections(in tableView: UITableView) -> Int {
    return 1
  }

  func tableView(_ tableView: UITableView,
   numberOfRowsInSection section: Int) -> Int {
    return viewModel.dataSource.count
  }

  func tableView(_ tableView: UITableView,
    cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier: "ProjectsCell",
      for: indexPath)
    // ... configure the cell using viewModel.dataSource
    return cell
  }
}
```

Listing 8: Setting up a `UITableView` data source without `RxSwift`

```
class ProjectsViewController: UIViewController {
  // ... ViewController setup ...
  override func viewDidLoad() {
    super.viewDidLoad()
    // ... view setup code - adding, layout etc. ...
    tableView.register(ProjectsTableViewCell.self,
      forCellReuseIdentifier: "ProjectsCell")

    // ViewModel binding to the tableView
    viewModel.dataSource
      .bind(to: tableView.rx.items(cellIdentifier: "ProjectsCell",
        cellType: ProjectsTableViewCell.self)) { row, element, cell in
        // ... configure the cell using the element directly
      }
      .disposed(by: disposeBag)
  }
}
```

Listing 9: Setting up a `UITableView` data source with `RxSwift`

Chapter 6
Testing

Testing is the process of evaluating a system or its components with the intent to find whether it satisfies the specified requirements or not. Testing is executing a system to identify any gaps, errors, or missing requirements in contrary to the actual requirements. [45] The demo application `RxGitLabKitDemoApp` was tested manually by the developer on a real GitLab server such as `gitlab.com` and `gitlab.fel.cvut.cz`. Because the demo application was not complex, automated tests were not necessary. Therefore this chapter only describes how the `RxGitLabKit` was tested to ensure the quality of the software. As a testing technique, unit testing and integration testing were chosen and they were used to support the development using the TDD principles.

6.1 Test driven development

TDD stands for Test Driven Development. The basic idea is to write a failing test, then write the code so that the test passes and then refactor the code. Repeat until the code meets certain standards.

This approach takes more time at the beginning of development, but in the long run, the tests can help to identify bugs created by modifying some parts of the code.

This library contains a large amount of data serialization from JSON format to Swift objects and vice versa. This example is perfect for using TDD. If the objects or JSON data from the server is incompatibly modified, the failed tests point out to the error.

6.2 Unit Testing

The main functionality of `RxGitLabKit` is to create and send a HTTP request to the GitLab API server and from the response of the server create a Swift structure containing the recived data. This chain is composed of multiple functions which can be tested separately using unit tests. First the networking layer was tested using a mocked `URLSession` and then tests to ensure the correct data transformation from JSON to Swift data structures and vice-versa were conducted.

6.2.1 Mocking

Because unit tests should be isolated from other systems, mocked data instead of live data from server were created and used. Mocking is creating objects that simulate the behaviour of real objects. These objects are then used to test the rest of the tested code.

Networking layer mocking

Networking layer was mocked by using a custom-made `MockURLSession` class. `MockURLSession` enables providing the mocked data which should be returned in the response before a request is called. This enables testing the rest of the code without having to communicate with a real server. The testing code also tests whether the initial request is in the correct format.

Data Mocking

The mocking data (JSON Objects) was copied from the GitLab API examples and used as mocking data. The JSON string was serialized to Data format using UTF-8 standard. These mocked objects were then used for parsing and decoding JSON objects into actual Swift structures.

6.2.2 XCTest

The main framework for testing is XCTest and the tested classes are usually child classes of `XCTestCase` class. Two of the main functions are `setUp` and `tearDown`. The `setUp` function is called before every test method in the class is called and the `tearDown` is called at the end of every test method.

Unit testing often needs to compare an expected and actual values, testing whether a condition is true or false. For these tests, the framework `XCTest` offers functions like `XCTAssertEqual(expression1: T, expression2: T)`, `XCTAssert(expression: Bool)`. `XCTAssert` has many variations.

Because of the nature of the library, a lot of code is asynchronous. This made testing a little bit tricky, because the testing function usually finished sooner before the response arrived. Fortunately `RxSwift` provides a `.toBlocking()` method, which makes working with `Observables` synchronous. This is perfect for testing asynchronous code.

An example of a unit test is depicted in listing 10.

6.3 Integration Testing

`RxGitLabKit` provides a client for GitLab API server, therefore the whole functionality depends on the server. During the development of `RxGitLabKit` the written code depended on the GitLab documentation [4]. It can happen that the real system behaves differently in comparison to the documentation. To ensure correct behavior on the client side according to the real system, integration tests must be conducted and the found bugs must be fixed.

6.3.1 Creating a GitLab instance

For integration tests, a local GitLab instance was used. A private local GitLab server instance provides full control over the access privileges and the data stored on the server. GitLab state can be backed up and recovered. This is useful when executing integration tests with destructive instances (for example deleting a project). The GitLab server instance was created using Docker [1]. The integration tests depend on a concrete mocked state of the GitLab server. For the purpose of the integration tests recreation, the server state is stored in form of a backup on the provided CD with this thesis. The backup file was done on GitLab version 11.4 EE ais named `gitlab_backup.tar` and the server state can be restored using a restoration process described on [2].

6.3.2 Creating mock data

To be able to simulate a GitLab server, some mock data was needed. 12 mock users were created in admin mode and then 7 projects cloned from GitHub.com and added manually. Additional 60 randomly generated projects with random commits were created to show more projects in the `RxGitLabKitDemoApp`.

6.3.3 Testing code

The testing code was very similar to unit tests. In unit tests, the networking layer was mocked, in integration tests, the real data from the server were returned.

6.4 Chapter summary

The `RxGitLabKit` was tested using unit tests and integration tests on a local GitLab server. Unit tests were mainly used for data transformation validation and URLRequest validation and integration tests for networking and also data transformation validation. These tests helped revealing many bugs which were then fixed.

The `RxGitLabKitDemoApp` was tested only manually by the developer because the application was not complex and was sufficient enough.

---

[1] `https://docs.gitlab.com/omnibus/docker/`

[2] `https://docs.gitlab.com/ee/raketasks/backup_restore.html#restore`

```
func testAuthenticate() {
  // Mocking the response dataAhoj,
  mockSession.nextData = AuthenticationMocks.oAuthResponseData

  // Calling the request
  let result = client.authentication
    .authenticate(username: "root", password: "admin12345")
    .toBlocking()
    .materialize()

  // Asserting results
  switch result {
  case .completed(elements: let elements):

    // Request asserts
    if let body = mockSession.lastRequest?.httpBody,
    let dict = try? JSONSerialization.jsonObject(with: body,
      options: .mutableContainers) as! [String: String]
    {
      XCTAssertNotNil(dict["grant_type"])
      XCTAssertNotNil(dict["username"])
      XCTAssertNotNil(dict["password"])
    } else {
      XCTFail("Body data is corrupted")
    }
    if let lastURL = mockSession.lastURL,
      lastURL.pathComponents.count == 3 {
      XCTAssertEqual(lastURL.pathComponents[0], "/")
      XCTAssertEqual(lastURL.pathComponents[1], "oauth")
      XCTAssertEqual(lastURL.pathComponents[2], "token")
    } else {
      XCTFail("Number of path components doesn't match.")
    }

    // Response asserts
    XCTAssertEqual(elements.count, 1)
    if let authentication = elements.first {
      XCTAssertNotNil(authentication.oAuthToken)
      XCTAssertEqual(authentication.tokenType, "bearer")
      XCTAssertNotNil(authentication.refreshToken)
      XCTAssertEqual(authentication.scope, "api")
      XCTAssertNotNil(authentication.createdAt)
    } else {
      XCTFail("Authentication is nil.")
    }
  case .failed(elements: _, error: let error):
    XCTFail(error.localizedDescription)
  }
}
```

Listing 10: Authentication unit testing

Chapter 7

Comparison with other GitLabAPI clients written in Swift

This chapter compares *RxGitLabKit* with other available Swift solutions for communication with GitLab API. As listed in 3.3.1 there are only two available libraries namely *GitLabKit* [1] and *TanukiKit*[2]. The comparison was done on these parameters: technologies used, support of technologies and performance.

7.1 Technologies comparison

RxGitLabKit was written in the latest Swift 4.2 supporting the latest version of GitLab API v4 and all Apple platforms (iOS, macOS, tvOS, watchOS). It provides a reactive API using `RxSwift`. No other dependencies (besides SnapKit, which is needed in the demo application) are used.

TanukiKit is a library written in Swift 2.0 and the last release v0.5.2 was on August 4th 2017 and supports only GitLab API v3, which is no longer supported by GitLab. The supported platforms are not described in the documentation. No more in-depth analysis of this library was conducted, because of lack of maintenance and no future usability.

GitLabKit was written in Swift 3.0, supports GitLabAPI v4 and only macOS platform. There is no release, and time of writing this thesis, the last commit date is June 18th 2017. It dependens on Alamofire[3] and Mantle [4]. It has implemented almost all communication with GET API endpoints but has no implementation for POST, DELETE and PUT endpoints. Although this library is not maintained anymore, it can be used in an application if few changes are made and no other API endpoints than GET are needed. Therefore this library can be used in a performance comparison.

7.2 Performance comparison

Because *TanukiKit* is written in an older version of Swift and because it doesn't support GitLab API v4, it was excluded from this comparison. Therefore only the performance of *RxGitLabKit* and *GitLabKit* is compared in this

---

[1] `https://github.com/toricls/GitLabKit`

[2] `https://github.com/nerdishbynature/TanukiKit`

[3] `https://github.com/Alamofire/Alamofire`

[4] `https://github.com/Mantle/Mantle`

section. Although `RxSwift` used in *RxGitLabKit* brings a cleaner and easier to understand codebase, it can bring some overhead because of the reactive implementation which may negatively influence the performance. The goal is to determine whether `RxGitLabKit` is not significantly slower in performance in comparison to other usable Swift clients.

7.2.1 Potential limitations

When comparing the performance of the clients, some potential limitations arise. The limitation might be the fact, that the measured code **includes** the time which it takes the server to get the request, process it, and return a response. The times measured can be heavily dependant on the server performance instead of the client performance. Therefore the same requests should be requested by both clients and the server should have as stable performance as possible. These two focus areas are discussed later in this section.

7.2.2 Benchmarking scenarios

The benchmarking scenarios consist of fetching all commits from a project from the GitLab server. For this scenario, a project containing 3112 commits is used. GitLab limits the number of elements using pagination as described in 3.1.2. One scenario is fetching all commits using the default GitLab API `per_page` 20 and the other scenario is to use the maximum `per_page` 100. Using lower `per_page` results in more requests which negatively impact the performance, but it can show the performance differences of the tested implementations of the clients. This scenario was specifically chosen because it includes using multiple asynchronous requests to the server which can be run concurrently and the result must be then merged. The benchmarking code for `per_page = 20` *RxGitLabKit* can be seen in the listing 12 and *GitLabKit* in the listing. The code for `per_page = 100` has only parameter per page set to 100 and the number of iterations in 12 is 32 instead of 156.

```
func testPerformancePerPage20() {
  self.measure {
    let commitsPaginator = self.client.commits
              .getCommits(projectID: 3, perPage: 20)
    let commits = commitsPaginator.loadAll()
      .toBlocking()
      .single()
  }
}
```

Listing 11: RxGitLabKit measured block of code (`per_page = 100`)

```
func testPerformancePerPage20() {
    let totalCommitCount = 3112
    self.measure {
      let expectation = XCTestExpectation(description: "load")
      let params = ProjectCommitQueryParamBuilder(projectId: 3)
      _ = params.perPage(100)
      var allCommits = [Commit]()
      for i in 1...156 {
        _ = params.page(UInt(i))
        self.client
          .get(params, handler:
            { (response: GitLabResponse<Commit>?, error: NSError?) in
            guard let commits = response?.result else { return }
            allCommits.append(contentsOf: commits)
            if allCommits.count == totalCommitCount {
              expectation.fulfill()
            }
        })
      }
      self.wait(for: [expectation], timeout: 1000)
    }
  }
```

Listing 12: GitLabKit measured block of code (`per_page = 100`)

7.2.3 Experiment circumstances

To minimize random variables of the experiment, they were carried out under the same or very similar circumstances (the same HW, same current load on the machine). The client and GitLab API communicate over the network which can heavily influence the results of the experiment. Therefore a local instance of GitLab server has been created using Docker to minimize the dependency on a stable network connection. The same instance was used in integration testing 6.3.

The measurements were executed on the same machine as the running GitLab server instance. For these measurements, the platform macOS was chosen because it is the native operating system of the machine on which the measurements were done, and no simulators needed to be used, which minimized some random variables. Also *GitLabKit* is only supported on macOS. To minimize the hardware load differences, the machine was rebooted to clear all unnecessary programs and allowed only Docker and XCode to run on this machine. The full hardware specification of the computer is shown in table 7.1 and the software used for performance measuring is in 7.2.

Model Name:	MacBook Pro 2016 (13-inch)
Operating system:	macOS High Sierra Version 10.13.3
Processor:	2,9 GHz Intel Core i5
Number of Processors:	1
Total Number of Cores:	2
L2 Cache (per Core):	256 KB
L3 Cache:	4 MB
Memory:	8 GB 2133 MHz LPDDR
Graphics:	Intel Iris Graphics 550 1536 MB

Table 7.1: Hardware specification

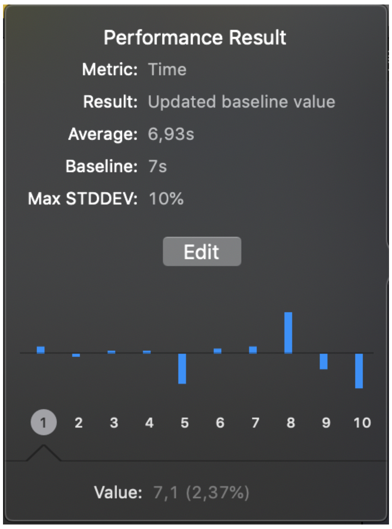
XCode:	Version 10.1 (10B61)
Docker:	Docker Engine - Community version 18.09.0
GitLab:	GitLab Enterprise Edition version 11.4.0-ee
Swift:	Apple Swift version 4.2.1 (swiftlang-1000.11.42 clang-1000.11.45.1)

Table 7.2: Software specification

7.2.4 Measurements and comparison

The performance was measured by execution time of a block of code. As stated before, the experiment had many random variables, therefore the execution time was measured repeatedly to acquire enough measurements to obtain a narrow confidence interval from the measured data.

The execution times were measured on two levels. In the first level, the program ran in a loop and on the second level, the program itself was executed several times. For this measurement, a method `measure(_:)` the class `XCTest` was used. This method measures the performance of a block of code. By default, the method measures the number of seconds the block of code takes to execute [46]. It runs the block of code 10 times and each time it measures the execution time of each iteration and generates a small graph as can be seen in 7.1. This measurement was executed 20 times, and in each measurement, the code was measured 10 times and the average execution time from these 10 iterations was taken for further analysis. The running of the measurements was executed on each client alternatively to minimize the difference of the state of the machine during each measurement. From the measured data, means, averages and variance were used to determine, whether the performance of the clients was significantly different. From the results of the measurements presented in table 7.3 (the full measurements table is shown in the appendix E.2) and using the two-sided 95% confidence level of Student's t-distribution for 20 values $q_{t(20)}(.975) = 2.086$ was calculated that for `per_page = 100` RxGitLabKits execution times were lower by $0.31 \pm 0.04s$ ($10.39\% \pm 1.46\%$) lower and for `per_page = 0` the RxGitLabKits execution times were lower



Performance Result

Metric: Time

Result: Updated baseline value

Average: 6,93s

Baseline: 7s

Max STDDEV: 10%

Value: 7,1 (2,37%)

Figure 7.1: Measurement result example

Library (per_page	Median	Average	Std. dev.	Confidence
GitLabKit `(100)`	2.94	2.83	0.12	2.83 ± 0.12
RxGitLabKit `(100)`	2.585	2.60	0.09	2.60 ± 0.09
GitLabKit `(20)`	6.875	6.97	0.022	6.97 ± 0.22
RxGitLabKit `(20)`	6.58	6.64	0.20	6.64 ± 0.20

Table 7.3: Summary for executions times. Times in seconds.

by $0.329 \pm 0.13s$ ($4.72\% \pm 1.95\%$).

7.3 Chapter summary

There are two other libraries for GitLab API communication written in Swift, and both are not maintained anymore. The latest contribution to these libraries is in the year 2017. Only *GitLabKit* supports GitLab API v4 and can still be used but only on macOS platform. On the other hand *RxGitLabKit* was developed in the latest version of Swift 4.2 with the support of all Apple platforms.

The goal of performance comparison was to conclude whether the performance of *RxGitLabKit* library is not significantly worse than the existing *GitLabKit* library because of the concern with the overhead reactive extensions might bring. The execution time measurements were conducted under the same hardware and software circumstances using two-level executions: 20 executions with 10 iterations each. Thus the measurement results are considered to be representable. The executed block of code differed in the implementation, while *RxGitLabKit* provides a function to load all objects from all pages, *GitLabKit* doesn't provide any. The measured times and calculated average and standard deviation values cannot be used to determine the relative speed between the libraries because of the limitations stated in subsection 7.2.1. However, it can be concluded that the *RxGitLabKit* performed better in the tested cases with 95% confidence.

A clean reactive API of *RxGitLabKit* makes this library more natural to use, the modular design is easy to expand and the performance is even slightly better than the current solutions. These variables make *RxGitLabKit* a better choice for future use in applications.

Chapter 8

Conclusion

The goal was to create a library for communication with GitLab API supported on all latest Apple platforms and compare it with existing solutions. The subgoal was to follow the software engineering principles and procedures when approaching the goal.

Firstly the functional and nonfunctional requirements were stated and then the analysis chapter introduced the necessary information about the GitLab API and available libraries. These libraries were summarized, and some key ideas were extracted for the design phase. Furthermore, after the summary of programming paradigms and programming languages for Apple platforms, a functional reactive paradigm with Swift language was chosen for the development because of the asynchronous nature of the communication between GitLab API server and the library and because Swift is a more modern programming language than Objective-C. In the second half of the analysis, possibilities of using FRP in Swift were described, and in the end, `RxSwift` was chosen as the FRP framework for the created library and demo application. To illustrate the FRP concepts, essential elements of `RxSwift` were described. After this section, the widely used architecture patterns for iOS app creation were described resulting in choosing the MVVM design pattern as the most suitable pattern for the demo application, because it synergizes well with the FRP approach.

The requirements were kept in mind while designing the library. The main focus was on modularity, intuitive API and using FRP approach. The central decomposition of the problem was to separate the networking part and the encoding and decoding part. In the implementation part, a detailed approach and some of the interesting parts of the code was shown. During the development part, a TDD approach has been used because a large part of the library deals with encoding/decoding objects to JSON and vice versa. The encoding/decoding unit tests assured that encoding and decoding behave as expected when code modifications have been made, which in return saved much time searching for bugs. The whole library was then tested using integration tests on a local GitLab server instance with manually created mock data.

The implementation of the demo application followed a standard pattern for iOS application development. Using MVVM architecture with `RxSwift` and `RxCocoa` enabled fast and intuitive implementation that shows the basic functionality of `RxGitLabKit`. The functionality was tested manually by the developer.

The result of this master's thesis is a GitLab API client library called `RxGitlabKit` written in Swift that supports all Apple platforms - iOS, macOS, tvOS and watchOS. This library provides a reactive API for communication with a GitLab API server. The user can choose a GitLab server using an URL and can log in using a username and password, a private token or an OAuth token. Because the number of all endpoints is too large, only endpoint groups containing the endpoints concerning Projects, Repositories, Commits, Users, and Authentication were implemented. An application called `RxGitlabKitDemoApp` was created to demonstrate the functionality of the library. Future integration of this library into other projects was simplified by adding support for the three most used dependency managers (CocoaPods, Carthage and SPM. The source code of `RxGitlabKit` and `RxGitlabKitDemoApp` is provided on the CD in the folder `sources/implementation` and is also uploaded on GitLab (`https://gitlab.com/dagytran/RxGitLabKit`) as a public open-source project under MIT license. Documentation to accompany the source code is available in the folder `documentation` and online `https://dagytran.gitlab.io/RxGitLabKit/`. All functional and nonfunctional requirements for both the library and the demo application are thus met.

The library was created because there wasn't any reactive library for communication with GitLab API with the support for all Apple platforms. The closest solution is `GitLabKit` which is not maintained anymore, has only a few endpoints implemented, supports only macOS and doesn't provide a reactive API. In comparison to `GitLabKit`, `RxGitLabKit` is written in the latest Swift 4.2 with the support of all Apple platforms, supports the latest GitLab API v4 and provides a nice and clean reactive API while according to the performance tests, `RxGitLabKit` performs slightly better. The only downside may be a dependency on `RxSwift`. However, although it is not a native framework, it has a large supporting community which makes the framework reliable. This `RxGitLabKit` library can be a cornerstone for a GitLabAPI Client with an GUI. Alternatively, it can be used for example in a repository monitoring application, which monitors the information about builds, commits, contributors and other information about the repositories.

Because the library has 5 of the total 68 endpoint groups implemented, the future work can be directed towards expanding the support for other API endpoint groups. Because of the modular nature of this library, implementing other endpoint groups and incorporating it into the client is straightforward. Another future work can be creating a similar library for Android, which could also be beneficial for Android developers.

Appendix A

Bibliography

[1] R. Rawson. *2018 Version Control Software Comparison: SVN, Git, Mercurial.* URL: `https://biz30.timedoctor.com/git-mecurial-and-cvs-comparison-of-svn-software/` (visited on Nov. 13, 2018).

[2] slant.co. *What are the best hosted version control services?* URL: `https://www.slant.co/topics/153/~best-hosted-version-control-services` (visited on Sept. 18, 2018).

[3] J. G. Andrew Stellman. *Applied Software Project Management.* O'Reilly Media, 2005. Chap. Chapter 6: Software requirements, p. 110. ISBN: 978-0596009489.

[4] GitLab Developers. *GitLab API.* URL: `https://docs.gitlab.com/ee/api/README.html` (visited on Nov. 13, 2018).

[5] GitLab Developers. *Project visibility level.* URL: `https://docs.gitlab.com/ee/api/projects.html#project-visibility-level` (visited on Nov. 13, 2018).

[6] GitLab Developers. *Pagination.* URL: `https://docs.gitlab.com/ee/api/#pagination` (visited on Nov. 13, 2018).

[7] altexsoft. *Swift vs Objective-C: Out with the Old, In with the New.* June 2018. URL: `https://www.altexsoft.com/blog/engineering/swift-vs-objective-c-out-with-the-old-in-with-the-nw/` (visited on Dec. 18, 2018).

[8] Apple Developers. *Building assert() in Swift, Part 2: FILE and LINE.* Sept. 2014. URL: `https://developer.apple.com/swift/blog/?id=15` (visited on Oct. 25, 2018).

[9] Apple Developers. *Swift 4.* URL: `https://developer.apple.com/swift/` (visited on Dec. 20, 2018).

[10] A. Lastovetska. *Swift vs Objective-C. Which iOS Language To Choose.* Oct. 2018. URL: `https://mlsdev.com/blog/51-7-advantages-of-using-swift-over-objective-c` (visited on Dec. 18, 2018).

[11] Apple Developers. *Extensions.* URL: `https://docs.swift.org/swift-book/LanguageGuide/Extensions.html` (visited on Dec. 20, 2018).

[12] Apple Developers. *Protocols.* URL: `https://docs.swift.org/swift-book/LanguageGuide/Protocols.html` (visited on Dec. 20, 2018).

[13] Apple Developers. *Subscripts.* URL: `https://docs.swift.org/swift-book/LanguageGuide/Subscripts.html` (visited on Dec. 20, 2018).

[14] P. Smyth. *An Introduction to Programming Paradigms.* URL: `https://digitalfellows.commons.gc.cuny.edu/2018/03/12/an-introduction-to-programmig-paradigms#orgheadline2` (visited on Nov. 5, 2018).

[15] K. Nørmark. *Overview of the four main programming paradigms.* Sept. 2012. URL: `http://people.cs.aau.dk/~normark/prog3-03/html/notes/paradigms_themes-paradigm-overviw-section.html` (visited on Nov. 5, 2018).

[16] F. Coenen. *Characteristics of declarative programming languages.* Oct. 1999. URL: `http://cgi.csc.liv.ac.uk/~frans/OldLectures/2CS24/declarative.html#detail` (visited on Nov. 5, 2018).

[17] M. A. Covington. *CSCI/ARTI 4540/6540: First Lecture on Symbolic Programming and LISP.* Aug. 2010. URL: `https://web.archive.org/web/20120307124013/http://www.ai.uga.edu/mc/LispNots/FirstLectureOnSymbolicProgramming.pdf` (visited on Nov. 5, 2018).

[18] W. P. Bird Richard. *An Introduction to Functional Programming.* Hertfordshire, UK, UK: Prentice Hall International (UK) Ltd., 1988, p. 1. ISBN: 978-0134841892.

[19] M. L. Scott. *Programming Language Pragmatics.* CA: Morgan Kaufmann Publishers, 2006, p. 140. ISBN: 978-8131222560.

[20] Tellis, Tele community. *Model-View-Controller and the "Observer" Pattern.* URL: `http://peak.telecommunity.com/DevCenter/Trellis#model-view-controller-and-the-observer-pattern` (visited on Dec. 16, 2018).

[21] A. Naumov. *Callbacks, Part 1: Delegation, NotificationCenter, and KVO.* Jan. 2017. URL: `https://nalexn.github.io/blog/2017/01/28/callbacks-part-1-delegation-notificationcenter-kvo/` (visited on Dec. 22, 2018).

[22] C. Eberhardt. *ReactiveSwift.* Apr. 2016. URL: `https://www.raywenderlich.com/1190-reactivecocoa-vs-rxswift` (visited on Dec. 18, 2018).

[23] F. Pillet, J. Bontognali, M. Todorov, S. Gardner. *RxSwift: Reactive Programming with Swift, Second Edition.* 2017. ISBN: 978-1942878469.

[24] N. Singh. *Reactive Programming with Swift 4: Build asynchronous reactive applications with easy-to-maintain and clean code using RxSwift and Xcode 9.* Feb. 2018. Chap. Chapter 2. FRP Fundamentals, Terminology, and Basic Building Blocks, p. 56. ISBN: 978-1787120211.

[25] RxSwift Developers. *Schedulers.* URL: `https://github.com/ReactiveX/RxSwift/blob/master/Documentation/Schedulers.md` (visited on Dec. 10, 2018).

[26] A. Mugo. *Architecture patterns in iOS.* May 2018. URL: `https://medium.com/the-andela-way/architecture-patterns-in-ios-a01780e271e8` (visited on Dec. 16, 2018).

[27] K. Zabłocki. *Good iOS Application Architecture: MVVM vs. MVC vs. VIPER.* May 2017. URL: https://academy.realm.io/posts/krzysztof-zablocki-mDevCamp-ios-architecture-mvvm-mvc-viper/ (visited on Dec. 16, 2018).

[28] Mozilla and individual contributors. *MVC architecture.* URL: https://developer.mozilla.org/en-US/docs/Web/Apps/Fundamentals/Modern_web_app_architectue/MVC_architecture (visited on Dec. 16, 2018).

[29] B. Orlov. *iOS Architecture Patterns - Demystifying MVC, MVP, MVVM and VIPER.* Nov. 2015. URL: https://medium.com/ios-os-x-development/ios-architecture-patterns-ecba4c38de52 (visited on Dec. 16, 2018).

[30] M. D. Network. *The MVVM Pattern.* URL: https://docs.microsoft.com/en-us/previous-versions/msp-n-p/hh848246(v=pandp.10) (visited on Dec. 16, 2018).

[31] M. S. John Morrison. *iOS Design Patterns: MVC and MVVM.* URL: https://www.captechconsulting.com/blogs/ios-design-patterns-mvc-and-mvvm (visited on Dec. 16, 2018).

[32] B. Jacobs. *Three Strategies to Keep View Controllers Skinny.* Aug. 2017. URL: https://cocoacasts.com/three-strategies-to-keep-view-controllers-skinny (visited on Dec. 16, 2018).

[33] Apple Developers. *Human Interface Guidelines.* URL: https://developer.apple.com/design/human-interface-guidelines/ (visited on Dec. 18, 2018).

[34] Apple Developers. *Tab Bars.* URL: https://developer.apple.com/design/human-interface-guidelines/ios/bars/tab-bars/ (visited on Dec. 18, 2018).

[35] Alamofire Software Foundation. *Alamofire Docs.* URL: https://alamofire.github.io/Alamofire/.

[36] Apple Developers. *Understanding Auto Layout.* URL: https://developer.apple.com/library/archive/documentation/UserExperience/Conceptual/AutolayoutG/index.html (visited on Dec. 18, 2018).

[37] SnapKit contributors. *SnapKit.* URL: https://github.com/SnapKit/SnapKit (visited on Dec. 18, 2018).

[38] Apple Developers. *UIKit.* URL: https://developer.apple.com/documentation/uikit (visited on Dec. 18, 2018).

[39] Apple Developers. *UISplitViewController.* URL: https://developer.apple.com/documentation/uikit/uisplitviewcontroller (visited on Dec. 18, 2018).

[40] Apple Developers. *Using Delegates to Customize Object Behavior.* URL: https://developer.apple.com/documentation/swift/cocoa_design_patters/using_delegates_to_customize_object_behavior.

[41] Gradle developers. *Introduction to Dependency Management.* URL: `https://docs.gradle.org/current/userguide/introduction_dependency_management.html` (visited on Dec. 24, 2018).

[42] S. Jagtap. *Carthage or CocoaPods: That is the question.* Mar. 2018. URL: `http://shashikantjagtap.net/carthage-cocoapods-question/` (visited on Dec. 23, 2018).

[43] Y. Brigance. *Choosing the Right iOS Dependency Manager.* Oct. 2017. URL: `https://aimconsulting.com/insights/blog/choosing-the-right-ios-dependency-manager/` (visited on Dec. 23, 2018).

[44] Apple developers. *Swift Package Manager Project.* URL: `https://github.com/apple/swift-package-manager` (visited on Dec. 23, 2018).

[45] T. point. *Software Testing Tutorial.* URL: `https://www.tutorialspoint.com/software_testing/` (visited on Dec. 18, 2018).

[46] Apple Developers. *Developer Documentation - measure(_:)* URL: `https://developer.apple.com/documentation/xctest/xctestcase/1496290-measure` (visited on Dec. 11, 2018).

Appendix B

Acronyms and Abbreviations

API Application Programming Interface. 1, 5, 6, 17, 21, 22, 34, 35, 39, 50, 63, 69, 70

CD Compact Disc. 54, 55, 61, 70

DSL Domain Specific Language. 50

FP Functional Programming. 18
FRP Functional Reactive Programming. 1, 21, 22, 34, 35, 69

GUI Graphical User Interface. 49, 70

HTML Hypertext Markup Language. 46, 54, 55
HTTP Hypertext Transfer Protocol. 1, 37, 38, 45, 46, 59
HTTPS Hypertext Transfer Protocol Secure. 53

IDE Integrated Development Environment. 45

JSON JavaScript Object Notation. 45–48, 59, 60, 69

KVO Key-Value Observing. 21

MVC Model-View-Controller. 31–33
MVVM Model-View-ViewModel. 31–35, 39, 45, 49, 51, 55, 69

REST Representational State Transfer. 1, 5

SPM Swift Package Manager. 52, 54, 70
SVN Apache Subversion. 1

TDD Test Driven Development. 45, 55, 59, 69

UI User Interface. 49

URI Uniform Resource Identifier. 46

URL Uniform Resource Locator. 1, 17, 18, 38, 39, 45–47

VCS Version Control Systems. 1

VIPER View-Interactor-Presenter-Entity-Router. 31, 34

Appendix C

CD Contents

.
- `documentation` .... source code HTML documentation folder
- `gitlab_backup.tar` .... GitLab Server instance backup
- `readme.txt` .... brief summary of the CD Contents
- `sources`
  - `implementation` .... source code folder of the implementation
  - `thesis` .... source code folder of the thesis in LaTeX format
- `tran_anh_duc_masters_thesis.pdf` .... thesis in PDF format

Appendix D

Figures

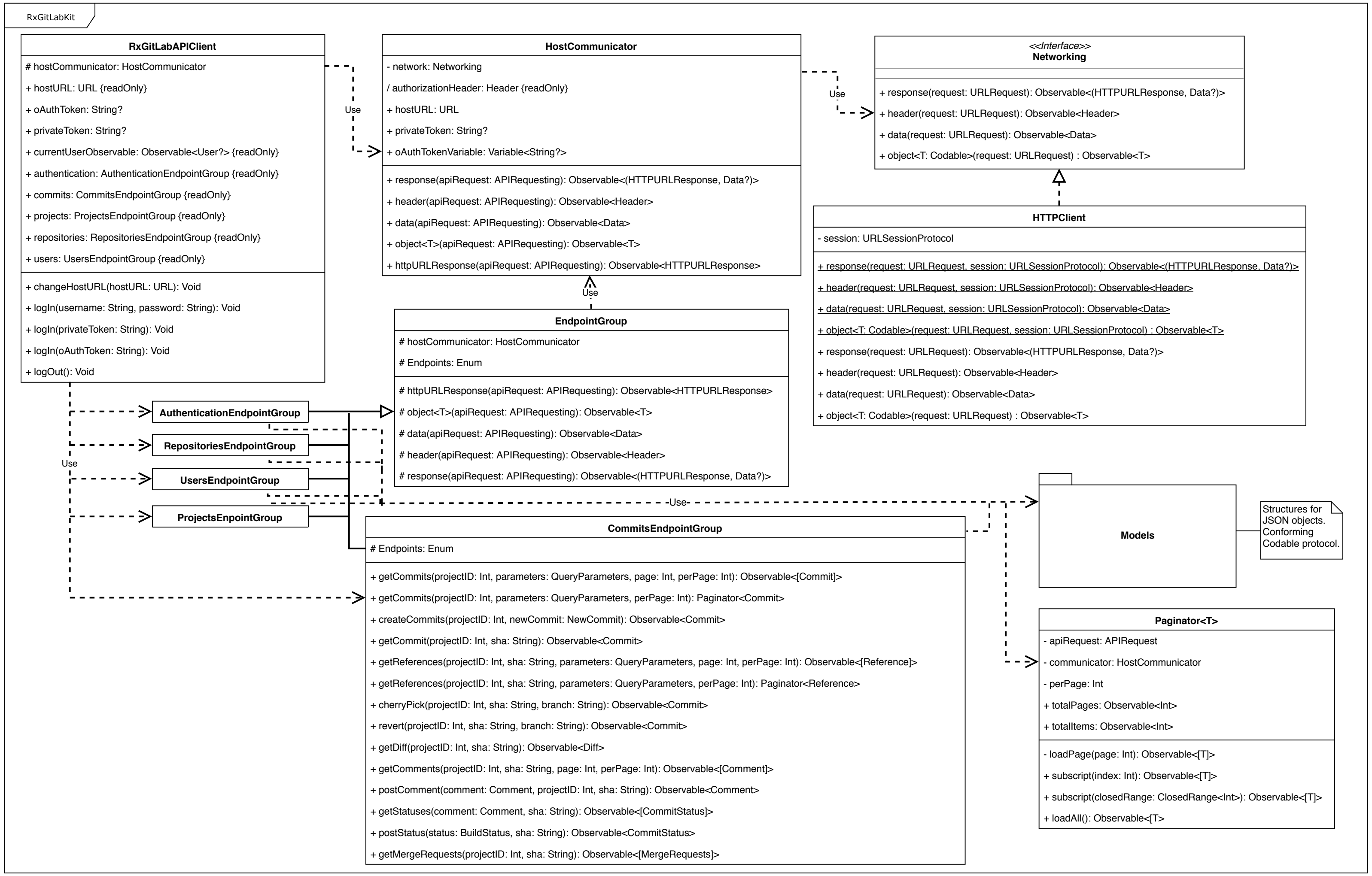


Figure D.1: Extended diagram of `RxGitLabKit`.

Appendix E

Tables

Authorization
Award Emoji
Branches
Broadcast Messages
Project-level Variables
Group-level Variables
Code Snippets
Commits
Custom Attributes
Deployments
Deploy Keys
Dockerfile templates
Environments
Epics
Epic Issues
Events
Feature flags
Geo Nodes
Gitignore templates
GitLab CI Config templates
Groups
Group Access Requests
Group Badges
Group Members
Issues
Issue Boards
Issue Links
Group Issue Boards
Jobs
Keys
Labels
License
Managed licenses
Markdown
Merge Requests
Merge Request Approvals
Project milestones
Group milestones
Namespaces
Notes (comments)
Discussions (threaded comments)
Resource Label Events
Notification settings
Open source license templates
Pages Domains
Pipelines
Pipeline Triggers
Pipeline Schedules
Projects including setting Webhooks
Project Access Requests
Project Badges
Project import/export
Project Members
Project Snippets
Project Templates
Protected Branches
Protected Tags
Repositories
Repository Files
Runners
Search
Services
Settings
Sidekiq metrics
System Hooks
Tags
Todos
Users
Validate CI configuration
V3 to V4
Version
Wikis

Table E.1: GitLab API Endpoint Groups

Execution Number	**GitLabKit** `(per_page = 100)`	**GitLabKit** `(per_page = 20)`	**RxGitLabKit** `(per_page = 100)`	**GitLabKit** `(per_page = 20)`
1	2.83	7.46	2.51	6.93
2	2.88	6.86	2.75	6.9
3	2.76	6.87	2.68	6.71
4	2.83	7.24	2.73	6.72
5	2.74	6.87	2.6	6.55
6	2.73	7.18	2.61	6.97
7	2.78	6.84	2.71	6.75
8	3.29	7.34	2.68	6.92
9	2.73	7.00	2.56	6.91
10	2.8	7.24	2.51	6.42
11	2.86	6.88	2.47	6.57
12	2.85	6.87	2.49	6.43
13	2.85	6.72	2.72	6.53
14	2.88	7.01	2.53	6.37
15	2.8	6.81	2.66	6.58
16	2.9	7.02	2.64	6.36
17	2.86	6.66	2.57	6.55
18	2.68	6.71	2.52	6.40
19	2.83	7.03	2.56	6.63
20	2.81	6.75	2.57	6.58
Min	2.68	6.66	2.47	6.36
Max	3.29	7.46	2.75	6.97
Median	2.83	6.87	2.585	6.58
Average	2.83	6.96	2.60	6.63
Std. dev.	0.12	0.22	0.09	0.20

Table E.2: Time measurements from performance testing. Times in seconds.

Appendix F

Code samples

```swift
class RxViewController: UIViewController {
  let searchBar = UISearchBar()
  let tableView = UITableView(frame: .zero)
  var viewModel: ViewModel!
  let disposeBag = DisposeBag()

  override func viewDidLoad() {
    super.viewDidLoad()
    // ... view setup code - adding, layout etc.

    tableView.register(CustomTableViewCell.self, forCellReuseIdentifier: "Cell")

    viewModel.dataSource
      .bind(to: tableView.rx.items(cellIdentifier: "Cell",
        cellType: CustomTableViewCell.self)) { row, element, cell in
        cell.project = element
      }
      .disposed(by: disposeBag)

    tableView.rx.itemSelected
      .subscribe(onNext: { [unowned self] indexPath in
        // do something with the selected item
      })
      .disposed(by: disposeBag)

    searchBar.rx.selectedScopeButtonIndex
      .bind(to: viewModel.scopeIndex)
      .disposed(by: disposeBag)

    searchBar.rx.text
      .bind(to: viewModel.searchText)
      .disposed(by: disposeBag)
  }
}
```

Listing 13: Setting up a `UITableView` and `UISearchBar` with RxSwift

```
class NoRxViewController: UIViewController, UITableViewDelegate,
 UITableViewDataSource, UISearchBarDelegate {
 let searchBar = UISearchBar()
 let tableView = UITableView(frame: .zero)
 var viewModel: ViewModel!

 override func viewDidLoad() {
  super.viewDidLoad()
  // ... view setup code - adding, layout etc.
  searchBar.delegate = self
  tableView.dataSource = self
  tableView.delegate = self
  tableView.register(CustomTableViewCell.self,
   forCellReuseIdentifier: "Cell")
 }

 // UITableViewDataSource delegate functions
 func numberOfSections(in tableView: UITableView) -> Int {
  return 1
 }

 func tableView(_ tableView: UITableView,
  numberOfRowsInSection section: Int) -> Int {
  return viewModel.dataSource.count
 }

 func tableView(_ tableView: UITableView,
  cellForRowAt indexPath: IndexPath) -> UITableViewCell {
  let cell = tableView.dequeueReusableCell(withIdentifier: "Cell",
   for: indexPath)
  // ... configure the cell using viewModel.dataSource
  return cell
 }

 // UITableViewDelegate delegate function
 func tableView(_ tableView: UITableView,
  didSelectRowAt indexPath: IndexPath) {
  // do something with the selected item
 }

 // UISearchBarDelegate functions
 func searchBar(_ searchBar: UISearchBar,
  textDidChange searchText: String) {
  // send the search text to viewModel
 }

 func searchBar(_ searchBar: UISearchBar,
  selectedScopeButtonIndexDidChange selectedScope: Int) {
  // send the search scope to viewModel
 }
}
```

Listing 14: Setting up a `UITableView` and `UISearchBar` without RxSwift