TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Oleksandr Shabala 177229 IASM

DESIGN AND DEVELOPMENT OF MOBILE-ORIENTED SOCIAL NETWORK. CLIENT SIDE

Master's thesis

Supervisor: Eduard Petlenkov
Professor

Tallinn 2019

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Oleksandr Shabala 177229 IASM

MOBIILSE SOTSIAALVÕRGUSTIKU PROJEKTEERIMINE JA ARENDAMINE. KLIENDI POOL

Magistritöö

Juhendaja: Eduard Petlenkov
Professor

Tallinn 2019

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Oleksandr Shabala

06.05.2019

Abstract

This thesis is written in English language and is 77 pages long, including 7 chapters, 62 figures and 2 tables.

Annotatsioon

Mobiilse sotsiaalvõrgustiku projekteerimine ja arendamine. Kliendi pool

Lõputöö on kirjutatud Inglise keeles ning sisaldab teksti 77 leheküljel, 7 peatükki, 62 joonist, 2 tabelit.

List of abbreviations and terms

UI	User Interface
UX	User Experience
RN	React Native
JS	JavaScript
JSX	JavaScript-XML
MVP	Minimum Viable Product
OS	Operating System
NPM	Node Package Manager
SDK	Software Developer Kit
IDE	Integrated Development Environment

Table of Contents

List of Figures

List of Tables

1. Introduction

Technical advances of the Internet and mobile technologies have promoted new forms of social communication, allowing the maintenance of large distributed networks of contacts. Under these “new forms” we mean Social networks, mainly. Such tools can either complement or replace face–to–face meetings.

Initially, the main goal of social networks was to make communication much easier and more accessible for people. With appearance of the very first social networks people switched from sending letters to each other by post to sending electronic messages, which were delivered instantly to the recipient. No doubts – that is a great progress.

However, with the time social networks, gaining more and more popularity among the society shown the other side of the medal – instead of complementing people’s real communication – they are quit often tend to replace face-to-face meetings.
There is quite vast spectrum of factors influencing this, which even can be considered in a separate thesis, however – we got an obvious need for social network to be the one that besides virtual communication also encourages people’s real communication.

That’s what this thesis is about – Developing a social network, which will gather people by a common interests – films and will effectively encourage real communication, thus making virtual communication actually a useful one.

Since nowadays, because of the most people’s lifestyle and temp of living, the most popular and effectively used social networks are the mobile ones – the goal is to develop, first of all, a mobile-oriented social network.

2. Analysis of platforms for mobile application development

2. 1. Analysis of Android and iOS platforms for mobile application development.

When starting developing any mobile application – first thing that shall be paid attention to is the platform that will be used for development. Most widely used mobile platforms are iOS and Android, so usually we're choosing between them.
This choice depends on several major factors:

- The ratio of usage of both platforms among the target audience;
A well known fact – the only producer of iOS devices is Apple. But when talking about Android phones – there are numerous different companies that produce these devices. As a result, it creates high competition on the market, which leads to the lowing the price of Android devices down, which, in it's turn, gives Android the major part of the market share. Here's a chart that reflects this situation [1]:

Fig 1. Worldwide Operating System Market for Android and iOS

As we see – the major part of mobile users worldwide prefer Android – 75.66% against iOS 19.23%.

In Europe situation is a bit different - 72.77% for Android and 26.21% for iOS users [2] – but still Android takes the lion part of the market share:

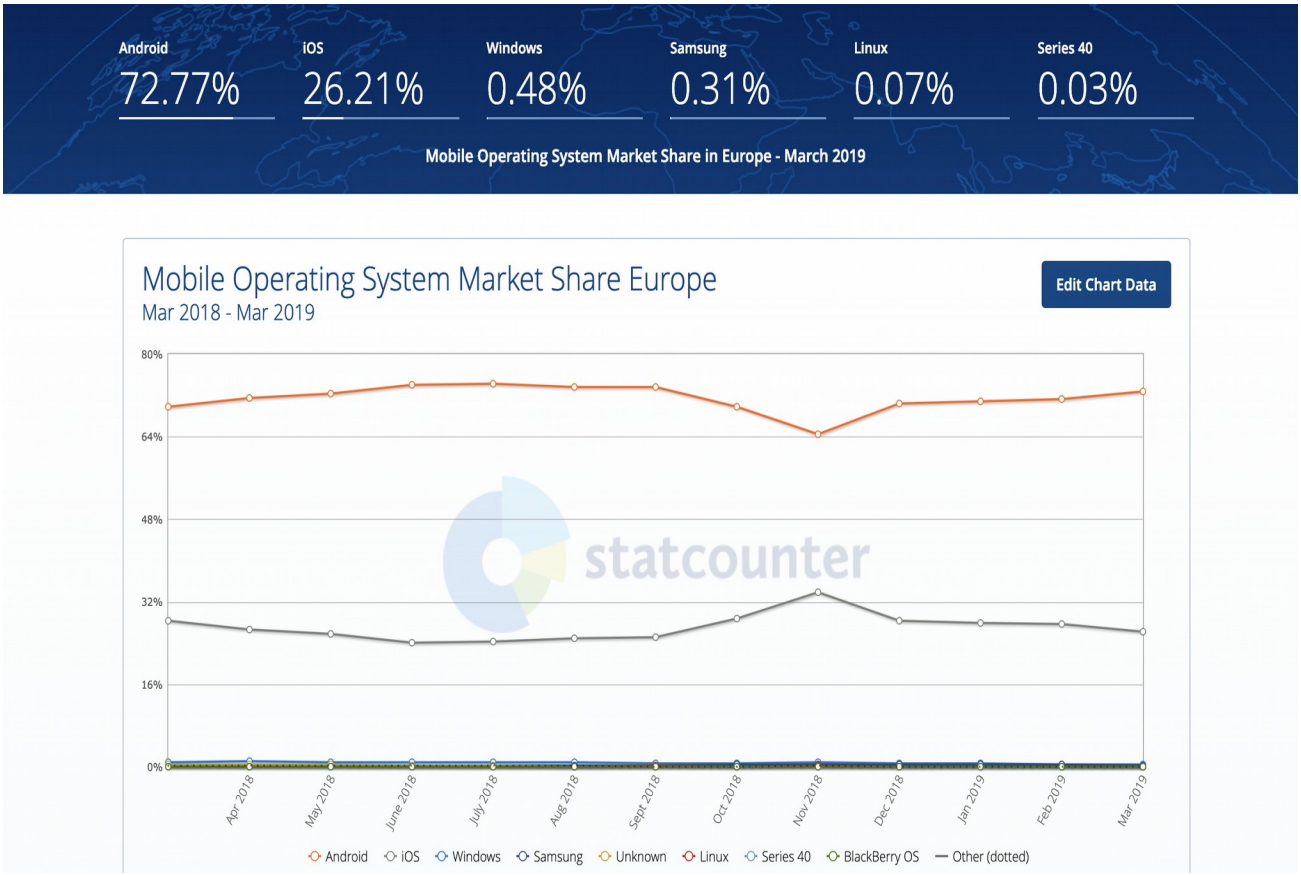


Fig 2. European Operating System Market for Android and iOS

- User payment capacity;

Statistics shows that the iOS users spend 45% more money on in-app purchases than the Android users[3]. This fact shall be certainly taken into account in case if the app's main revenue model is build mainly on the in-app purchases, even though that Android holds larger part of market share.

- Brand loyalty;

92% of iPhone users say that their next device will be an iPhone too [4]. Among the vendors of mobile phones running on Android this number is significantly smaller

(Samsung – 72%, LG – 59%, others much less). That also increases the possibility of selecting an iPhone as a new phone for these users which will, obviously, increase the number of iOS users more and more each year.

So, today it is clear that if you're building a free app, without any in-app purchases the most critical parameter will be the area where you want to launch the application and percentage of people using certain platform. For Europe – makes sense to go with Android.

If the revenue model of the application is build mainly on the in-app purchases and/or the application itself is paid, then obviously iOS users would be a better target audience.

However, there are many other points described in the following paragraphs that influence the choice of the platform for the application.

2.2 Development complexity

Of course, the crucial point on which the app's complexity depends on – is the desirable functionality set. Another important parameter – is the platform. The main factors that determines development complexity are next:

- Hardware

In order to build an Android application – almost any computer can be good for that (Windows, MacOS, Linux – all are fine). But only Mac computer can be used for developing an iOS app.

- Compatibility

When dealing with the modern projects, it is a better idea to choose newer technologies for this purpose. So makes sense to go with Swift and Kotlin programming languages, rather then sticking to older Objective C.

But here, the main questions that rises up is the compatibility of these languages. For example, Kotlin is completely compatible with Java, which means that you can use all of the countless libraries, frameworks for Java in the Kotlin project. Moreover, in fact even switching from one programming language to another is possible.

That cannot be said about Swift. Newer Swift is not 100% compatible with Objective-C. Here's where a lot of compatibility issues arise, which creates more headache for the developer. Along with that, the same framework can be more compatible with one code version (e.g. Swift 2) but less compatible with another (e.g. Swift 3) or vice versa, since the compatibility of every code version is different.

- Code

Also, it should be taken into account that there's constant development and, thus, changes of the programming languages. Newer versions are getting more and more advanced. New, updated versions are more universal and advanced. This is good, however it leads to another point – when the updated version of the programming language is way different from the previous one - as in the case with Swift, which makes the development process harder, when dealing with projects, written in the previous versions, and even when writing new project – it takes time to get used to the new version.

- Device Fragmentation

Today there are 14 iPhone models in regular use. With Android devices – situation is quite different. Nobody even know exactly what is the number of existing Android phones, all different in sizes and shapes.

This is what we call "device fragmentation". When building an app specifically for Android – it is reasonable to select some certain devices and screens, which will be supported by the application, which definitely has it's influence on the costs and makes development more complicated.

Beside that, the number of different OS systems that are currently in use is much greater than in the case with iOS, and that rises one more fragmentation problem.

With all these points – it is obviously very hard to build an app, which will be compat-

ible with all devices running on all OS versions and all devices, since it eats a lot of time and money.

- **Interface peculiarities**

Both Google and Apple want the apps in PlayStore and AppStores respectively to look like native ones, as they sell those (or distribute) and want their users to get some quality guaranties.

For that reasons, they both created guidelines for the app's design – to give an app a recognizable look among countless number of applications in the stores. These guidelines can be different but in any case they are quite easy to follow.

The major difference between Google and Apple apps is that the second ones are very strict in their requirements, which are precisely checked when the app is under review. Google also prefer to go along with the main design principles, but still – gives an opportunity for innovation.

Conclusion:

Table 1 below shows both platforms comparison in regards of target users in a simple way [5]:

19 %	**Market share**	75%
76 %	**% of users running latest OS version**	5.7 %
More loyal users	**User loyalty**	Less loyal users
North America, Europe, Australia	**User demography**	Europe, Asia, Africa

Table 1. Infographic of the most significant metrics about iOS and Android platforms

In order to check which one (or both) would be more reasonable to start with, let's check the requirements of the MovieGo application:

User demography	Globally
Application cost/fee for users	Application is free
In-app purchases	Yes
OS version updates needed	Yes

Table 2. Requirements of the MovieGo application

Since the plan is to make the application available globally (even-though starting with Europe) – it's certainly not an option to stick only to one platform.

So, if starting simultaneously with both iOS and Android platforms - we need to choose the way of development:

- Native development separately for each platform (using Java/Kotlin for Android and Objective C/Swift for iOS);

- Cross-platform development (using React/React Native).

Pros and Cons of each method is described in Part 3.

3. Analysis of React Native development and Platform-Native development and their comparison.

3.1 React Native Development

3.1.1 React Native Description

React Native is the framework for cross-platform development, which is based on ReactJS and JSX.

When React Native has just entered the market it was extremely popular – framework developed by such a giant company was causing trust among a lot of companies, who started developing their projects with the help of React Native.

One of the RN's advantages is that it is capable to facilitate the design system's use and maintain also atomic-oriented components among mobile platforms and across the web as well. React Native has libraries which are strengthening this ability.

Beside that, it respects both iOS and Android design peculiarities – RN's components compile to the native ones and it is shown as a native component on each of those.

Since React Native uses native layout components – it gives a possibility to make compelling UIs that are basically identical to the native iOS or Android application. It's code is shared through both of these code bases, however for example for the design elements (styles, views and so on), JSX uses native for the appropriate platform components for creating the layout for the corresponding platform (iOS or Android). This is actually the reason of the name of the framework – React **Native.**

On the other hand – it's only possible with either default components or some another ones, which are implemented in different ways on each platform, thus, it's casing a kind of snag since because of that many applications have to work out a solution which will help to prevent breaking of some features on one or another platform.

It is also so that it gives a possibility to use platform specific code and makes use of the native modules for both platforms.

So React native has definitely found it's place and probably holds the leading position when considering community and production experience.

3.1.2 Advantages of React Native Development

- Development speed

The biggest advantage of React Native is that development time is much shorter. It provides a lot of ready components, which can speed up the process. Actually, RN is not ideal and sometimes happens that some solutions should be built from the scratch, however React Native is based on JavaScript, meaning that it has access to the NPM, which is biggest package ecosystem. Since you can access so large package base – it saves a lot of time already now, and, most probably, it will even be better in the future. As the number of products built with RN is growing and the new updates are coming out regularly, in most cases it is possible to find needed components for most cases.

Investigation shows that to built the very same app with Swift takes as much as 33% more time than with React Native and still it will be working only on iOS – with no Android version [5].

Also, RN allows to share the codebase between OS in a matter of few hours, so it helps to save both time and money when developing apps for both platforms.

- Usage of one framework for several platforms

React Native allows to share the bigger part of the codebase between both OS in a matter of few hours. Actually, full cross-platform development is doable in certain extents, depending on how many native modules are used in the app.[6].

In MovieGo application 2 modules are currently used:

- react-native-splash-screen [7,8]. (See paragraph 6.1 for the splash screen picture and Appendix 1 for code example).

- react-native-gesture-handler [9]- for routing between multiple screens.
More details see in paragraph 6.2.1

Some features are present in NPM packages, some others may need to be written from the scratch, though. The good thing is that the framework is constantly improving, as it is supported by community – more and more tools are being added to the open source.

JS Development also makes it possible to share the codebase between React Web apps (not only between mobile platforms). That means that it is even possible to work on web applications and mobile ones both – because of the pretty similar technologies. From one side – it is a question how stable such solutions are, but in any case – it is still quite beneficial to be able to share non-UI dependent code. And in addition to shortening the time of development, it also helps improving the consistency of business logic of the application between the supported platforms.

- Hot Reloading

Because of the hot reloading you can run the application while implementing updates and setting the UI. Thanks to that the changes are applied without rebuilding the application, which is surely a great plus, because firstly, it saves the compilation time and secondly – it doesn't loose any app's states while applying these changes. It raises productivity and shortens the time of the development process.

- Single Team

Another major advantage of RN, in my case in particular, is that I don't have to develop iOS and Android app separately, which simplifies the development itself and also – doesn't slow down the process.

- Speedy Apps

There are a lot of debates that React Native code may obstruct the performance of the app.

In fact that's true, since JS is not that fast as native code, however this difference is so small that can be neglected – the average end user won't even see it in most cases. Indeed, when developing some more complex applications, I assume that RN's efficiency will inferior to native code, but even in those cases it is possible to switch to a native module and transfer some part of the code there, so it's not such a big issue.

- Simple User Interface

React Native is very convincing when dealing with mobile UI creation. In the case with native development, you have to make a sequence of actions in your app, and React Native engages declarative programming, where such implementation of actions is outdated. So it's way easier to detect errors and bugs on the user's paths [11].

3.1.3 Disadvantages of React Native Development

- Navigation issues

React Native is already widely used in production by numerous companies, however we can't say that it's the best solution in any case. Pretty often some changes in tools and dependencies appear between different versions and this causes issues – for example, failures of hot reloading, not compatible packages and so on. In it's turn it slows down the development.

- Absence of Some Custom Modules

Despite being so advanced platform, React Native still lacks some components and some are kind of "underdeveloped". In fact, there weren't such situations with MovieGo app, as the custom modules that were needed were found well-documented and they properly work. Still, this probability needs to be pointed out. It can also be so that it will be needed to create own solution or somehow modify the one that exists. And if you develop a custom module – instead of one codebase you can eventually have 3 codebases (iOS, Android and React Native), in each of which there may be obtained some appearance and behaviour differences. Luckily, there's a very low probability of such occasion.

- Performance Issues

When making rather simple apps, then React native is, without doubts, the best solution because of all the advantages mentioned above.

However, when some more advanced functionality is required, then performance goes down in comparison to applications created with platform-specific logic.

3.2 Native Development

3.2.1 Native Development Description

Native applications are the ones developed for some specific platform using the respective programming language – Java or Objective C/Swift for Android or iOS applications respectively.

Native applications work on the OS of the device. They do require full access to all the device's functionality and hardware and they are placed on the device.

Both Android and iOS provide developers with standard SDK – a kit with a set of needed tools, code examples and documentation, libraries and guides for developing an app on the platform.

These SDKs, together with the set of tools native app development gives great performance and nice UX.

Apple's XCode and Google's Android Studio are specially created IDE - software suites, which consist of editor, compiler and debugger of the code. These suites are very comfortable to use when to write and test the software.

IDE's raise up the application development efficiency, as it fixes bugs and shortens the time of development process.

3.2.2 Advantages of Native Development

- Higher Speed

As these are native applications – they work faster, because they work with the existent features of the device and it receives the user's data from the web, not the whole app.

- Common well-known look and UX

Native apps are kind of modified versions of the default applications of the device. When some function is performed by the user – it gives him a possibility to figure out the way how this application works, since it's quite similar to they way how all the other apps on that device work. Native apps have an advantage here, because mobile apps that want to play an upper hand here, as mobile applications that try to imitate the appearance and usage peculiarities of native ones often fail.

3.2.3 Disadvantages of Native Development

- Flexibility lack

You have to write code separately for each platform.

- Development and Maintenance Peculiarities

Because of the above mentioned statement - more people are needed not only to develop an app but also for post development maintenance, again, for each platform, iOS and Android, separately. Which also leads to more expenses.

- Development process consumes more time

Since you can't develop an app for both platforms simultaneously, but need to do it separately for Android and iOS, then in case if you can't hire people to work separately on iOS and Android development – it will take much more time to develop the application for each platform, comparing to RN development.

3.3 Conclusion and selection of application development method

So, taking into account all the pros and cons in this certain situation, more suitable solution will be to go along with React Native development, as it allows to simultaneously develop application for bot Android and iOS platforms, and thus doesn't require more people for separate development for iOS and Android. As for the running speed of the app – indeed it will be slower as in the case of native development, however, in this case the difference will be minor and can only be visible when applying some performance test, which is not a crucial point for the user.

4. Product Overview and MVP

4.1 Product Overview

4.1.1 Existent problems/targets that MovieGo tends to solve/reach

The pain point for the target user is a great improving of a social aspects of the everyday life. With the help of the MovieGO app it's possible to just grab your phone – see what are the most relevant movies in the nearby cinemas, choose the one you like, buy tickets and GO! But, in contradiction to other similar services – the user doesn't have to go alone (in case if no friends are available right now). It's possible to see other people who are interested to see this particular movie at the same time which is suitable for you and in the same cinema. With just clicking a button you already have a good plan for this evening, and, it's not ending there at the cinema – you've just made a new friend – why not to prolong your good time? Have a walk, discuss the movie you've just watched or do whatever you want – it's up to you – you will definitely have a great time. Before, when people wanted to visit a cinema – they used some kind of checking service, buying tickets, watching the movie and that's it. Of course, there're a plenty of incredible movies nowadays and another plenty of them are going to see the world soon, but it is much more cool to watch the movie with someone else – it's becoming more interesting and it's not only up to a movie then. Meet new people, make cool acquaintances, have fun and see the life from a different, much more brighter angle – that what's the user gets from the MovieGO! Absolutely for free.

Another important point that MovieGo encourages people to do charity. And, it is not only about the customers but also rather Cinemas as well. It's possible for Cinemas to give the customers an opportunity to buy a ticket a bit cheaper and spend the difference on charity. Thus, it will encourage users to visit the cinemas more and more often for various reasons: tickets are cheaper, now it's much more entertaining than before, and, along with all these – they can help others. And everybody wins!

4.1.2 The importance of the problem

The significance of these points is obvious. It is brand new page of the people's social life. Starting from improving the esthetic aspects of watching the movie, improving the social integration and interaction of the society, differing the daily routine and ending up with charity. Moreover, it's not only related to the actual private users of the application, but also the cinemas as well – as it is the additional source of promoting their activity.

- Using MovieGo users can solve all the problems and gain the opportunities mentioned above, mainly:
- Immediately find a buddy you'd LIKE to go for a movie with and book/buy a ticket at once, using one application.
- Make a new acquaintance with a person/people you like or with somebody new without any fears of seem weird as you are sure that these people are pursuing the same goal.
- Not making any efforts to do the above mentioned (e.g. with thinking how to compose a message to be liked or to attract) as it possible just to press a button to invite somebody.
- MovieGo has more simple interface than other movie checking services, so you won't spend time figuring out how to choose an appropriate movie in an appropriate cinema. Just scroll the main feed page with all the movies sorted by relevantness.
- Involves you into charity projects. You don't need to do or even spend anything (unless you want it), but anyway will help those who needs your help.
- Allows more organizations (cinemas in particular) into charity projects. Provides the cinemas with an additional wide source of promotion of their activity.

4.1.3 Alternatives

Currently there are no alternatives which allows to get all these benefits (you're basically help people in need by having fun and making your life better) – in just one app. There are different booking services where you can book/buy a movie ticket, also there're different charity platforms as well (but only minority of people cares about that)

and for sure there're a plenty of social networks, dating services, etc, but the first ones are used in more wide purposes and are not strictly related only to people's spontaneous communication (in other words: it may look weird to invite a stranger for a movie in an original social network) thus in many cases it stops possible communication (especially between unknown people). MovieGo gives people some common purpose, uniting them together.

4.2. Solutions

Using MovieGo users can solve all the problems and gain the opportunities mentioned above, mainly:

- Immediately find a buddy you'd LIKE to go for a movie with and book/buy a ticket at once, using one application.

- Make a new acquaintance with a person/people you like or with somebody new without any fears of seem weird as you are sure that these people are pursuing the same goal.

- Not making any efforts to do the above mentioned (e.g. with thinking how to compose a message to be liked or to attract) as it possible just to press a button to invite somebody.

- MovieGo has more simple interface than other movie checking services, so you won't spend time figuring out how to choose an appropriate movie in an appropriate cinema. Just scroll the main feed page with all the movies sorted by relevantness.

- Involves you into charity projects. You don't need to do or even spend anything (unless you want it), but anyway will help those who needs your help.

- Allows more organizations (cinemas in particular) into charity projects. Provides the cinemas with an additional wide source of promotion of their activity.

4.3 Minimum viable product.

4.3.1 Requirements for MVP

So, minimum viable product is the application, where you can:

1. Register with the help of email or Facebook
2. Log in
3. See the list of movies in a certain area
4. Go to the Movie page, see the description of the film.
5. “Like” a film and see other’s who liked it in the same area.
6. Send and receive invitations for a movie.
7. Follow users.
8. Track history (Likes, and visited movies)
9. See my plans (movies marked as the ones you’d like to go and all the info about it)
10. Chat with users (1 to 1 chat).

4.3.2 MVP's design

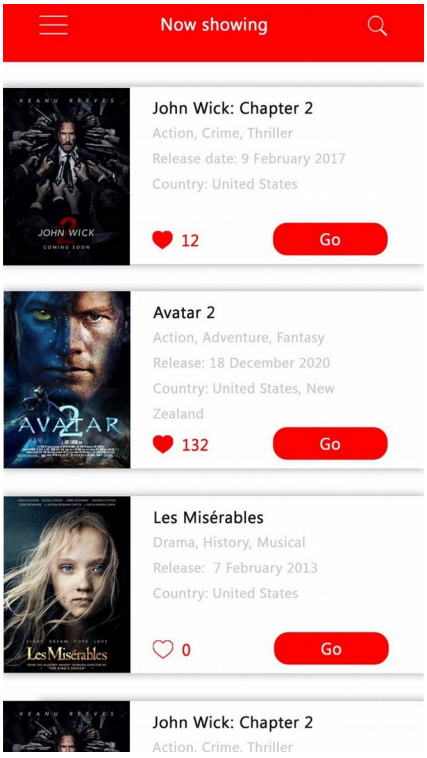


Fig. 3 Initial Screen

Fig. 3 Shows the initial screen of the app (home page), which user can see once signed in. Here you can see the list of movies which are currently showing at cinemas. You can scroll down to see the whole list of movies, click "Go" button and after that users that are visiting this movie page will see you in the list of people who'd like to go for that movie (See. Fig. 2) or "like" a certain movies and then check it from the Bookmarks page (which will be implemented later on).

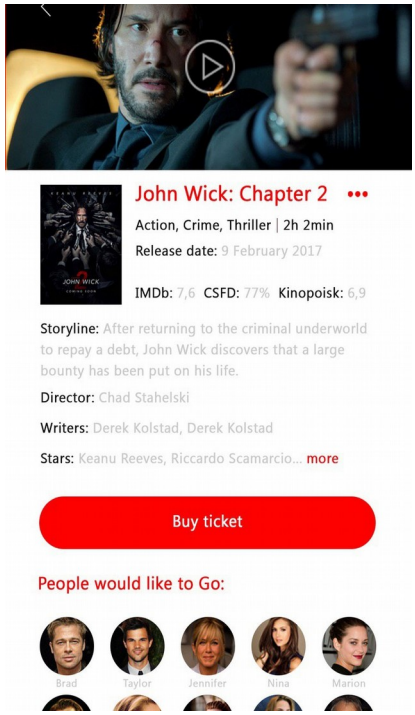


Fig. 4 Movie description

Fig. 4 shows the page that you reach when clicking on any movie card presented on the previous screen Fig. 4). Here you will see the description of the movie and People, who'd like to watch it.

You can also click on users avatar and see their profile, where you can also see the list of movies this user is interested in and invite a user to watch it together, like shown on Fig. 5

Fig. 5 User's Avatar that lead's to their profile

Invited user will be able to see his/her invitations and confirm/decline it on the invitations page Fig. 6

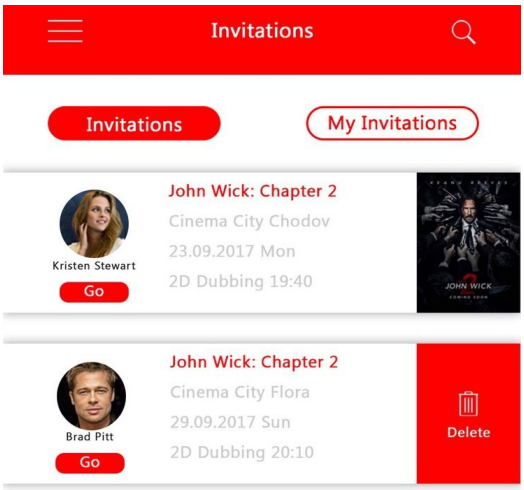


Fig. 6 Invitations list with an ability to accept/decline

And the one who invites, can see all his/her sent invitations and their statuses as well (Fig.7).

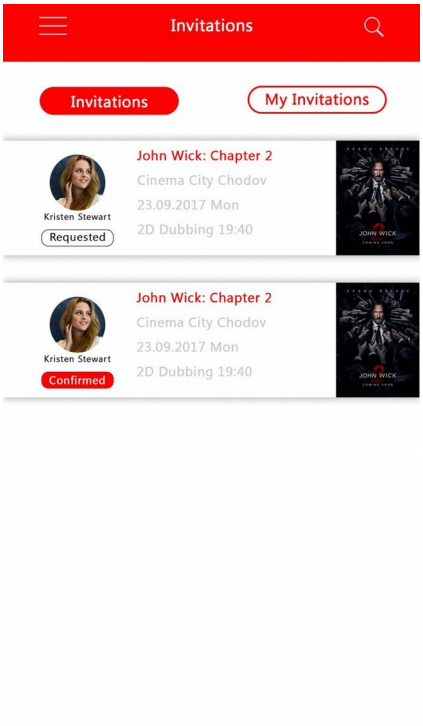


Fig. 7 List of self - sent invitations

Also, a user can follow other users to see what movies are they interested in, and any other info they post on their page (this feature will be implemented later). User can always check his/her list of followers as shown on Fig. 8

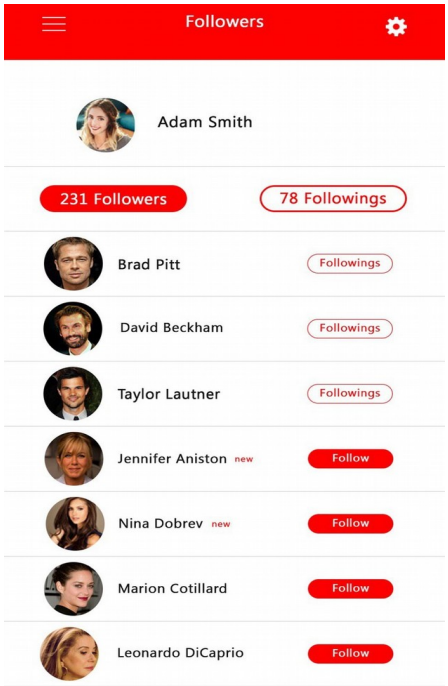


Fig. 8 List of followers and their interests posted on their page

Also, users can chat with each other(Fig 9, 10). Currently only the chat between two people is considered for MVP, however it is possible to add a group chat option later on.

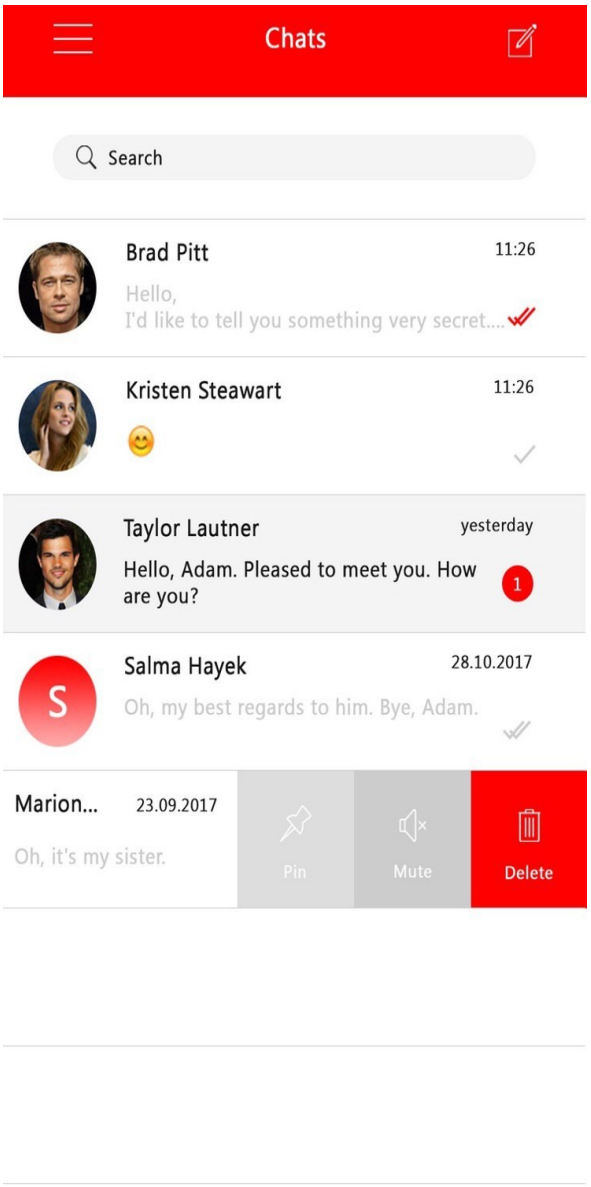


Fig. 9 Chats with other users

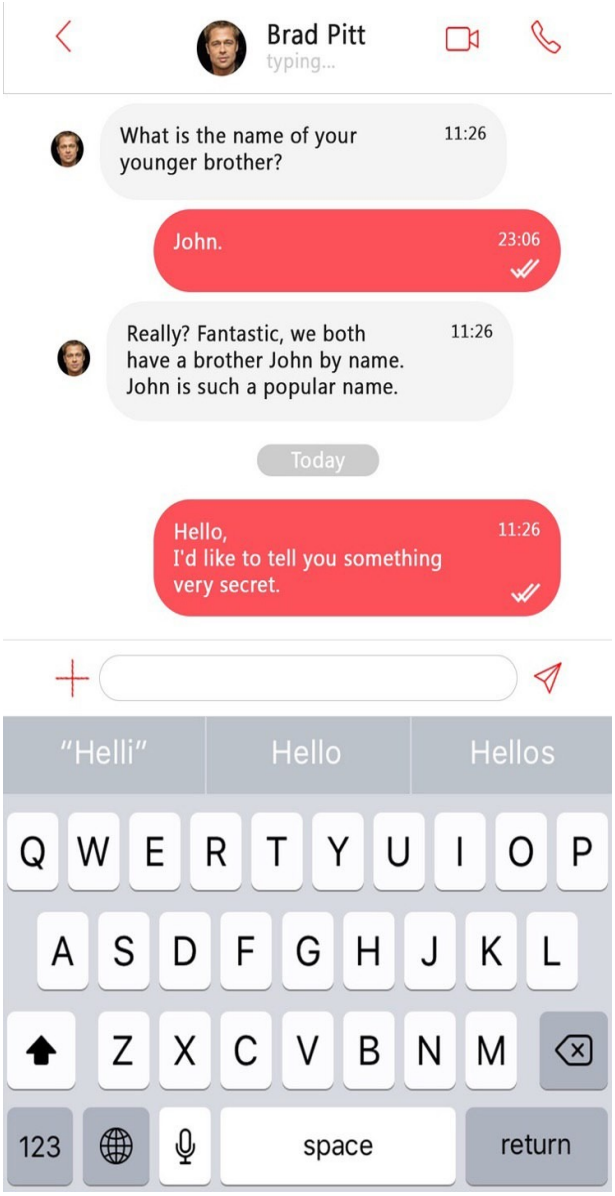


Fig. 10 Chat conversation between 2 users

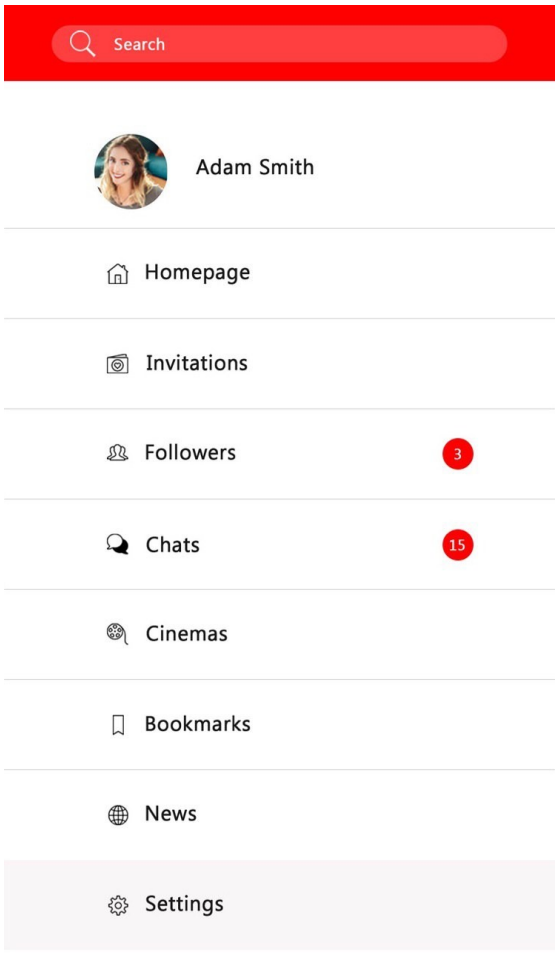


Fig. 11 Side bar menu

When you press on the “burger” on the top left corner of the main page (as shown on fig.3) a side bar menu will appear (Fig.11). From here user can go to all the other pages of the app: Homepage (Fig.3), Invitations (Fig. 6,7), Followers (Fig. 8), Chats (Fig 9, 10), Settings page where you can set and edit your personal information (Fig. 12).

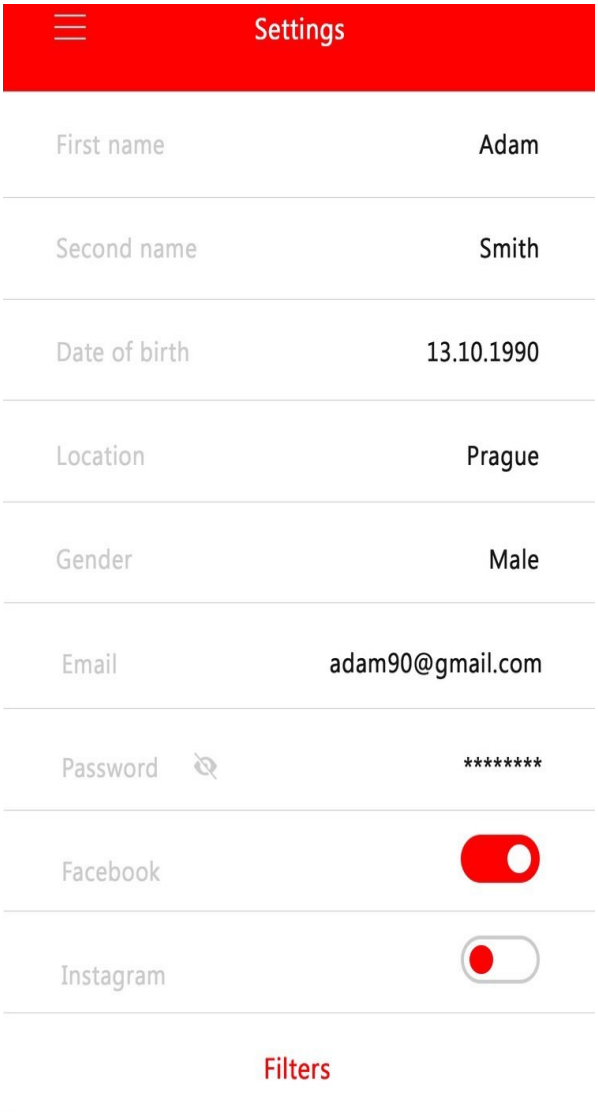
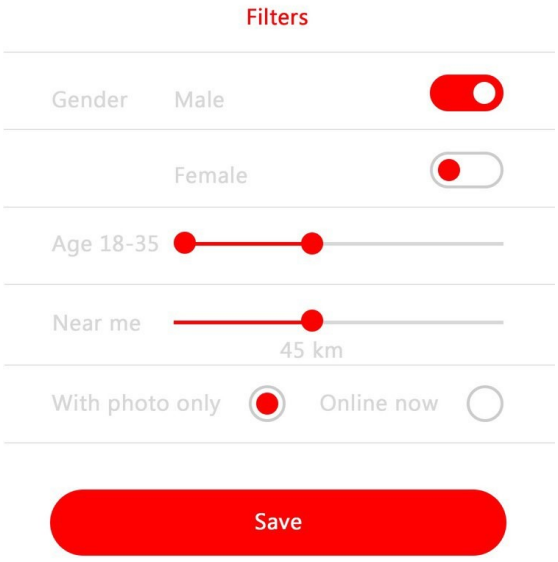


Fig. 12 Manage personal information



and also set the filters (Fig. 13) of users you'd like to see under the list of people that are interested in going for some certain movie (Like shown on Fig. 4).

Fig. 13 Setting filters

All these pages are necessary for MVP. Also, it is planned to add Bookmarks page, where user could see all the posts he liked, Cinemas page – with the list of cinemas in your location and News page – just for some interesting info, facts, announces about movies, actors, etc.

4.4 Areas for improvement

1. Enlarge the area of service.
2. Provide a possibility to buy a ticket directly through the app.
3. Create a Group chat option.
4. Create a Feed line – interesting news from the film industry (kind of blog).
5. Usage of micro services backend architecture instead of monolithic architecture

5. Developing UI toolkit for React Native

avatar/
Avatar.js #Avatar component. See Figures 1-6. in Appendix 1 for code example. It works as a placeholder for the user's photo. With it help user uploads image to the server.

```
<Avatar
              size="xlarge"
              onPress={this.openImagePicker}
              backgroundIcon={icons.camera}
              showAddPhotoButton
              iconAddPhoto={icons.add}
              source={avatar}
            />
```

Figure 14. Usage of Avatar component example

Example of usage of Avatar component on the client:

Properties of Avatar component:

size => *string* or *number*: Size of the main View.

- OnPress => function: Handler to be called when the user taps on the avatar.
- BackgroundIcon => number: Path to the icon that will be located inside the avatar (instead of the image).
- IconAddPhoto => number: Path to the icon that displays information to add photo.
- ContainerStyle => object: Custom styles to the container.
- OverlayContainerStyle => object: Overlay container styles.
- Source => image: Avatar image.
- AvatarStyle => object: Image styles.
- ActiveOpacity => number: Active opacity number for TouchableOpacity.
- ShowAddPhotoButton => boolean: Trigger to show icon from "iconAddPhoto".
- EditButton => object: Props that could be applied to the showAddPhotoButton.
- ImageProps => object: Props that could be applied to Image component.
- ...attributes => all the other props inherited from TouchableOpacity

buttons/
DefaultButton/ # Button component. See Figure 7 in Appendix 1 for code samples. This components is responsible for rendering of buttons with pre-defined styles (e.g. “Sign in”, “Sign up” or any other default (red oval) button).
index.js
styled_components/ # Styled components of Button component
StyleTouchableOpacity.js
Title.js
index.js

```
<DefaultButton
                    title="Sign Up"
                    disabled={avatarUploadProgress !== 100}
                    onPress={() => formikProps.handleSubmit()}
                  />
```

Figure 15. Usage of DefaultButton component example

Properties of DefaultButton component:

- backgroundColor => string: Background color of the button. (not-required)
- title => string: Text that displays inside of the button. (required)
- onPress => function: Handler to be called when the user taps on the button. (required)
- disabled => boolean: Disables all interactions with the element if 'true'. (not-required)
- ...rest => all the other props inherited from TouchableOpacity

HeaderBackButton/ # Button component. Left Arrow button that leads the user to the previous page. See Figure 8 Appendix 1 for code samples. The purpose of this button is usage for backwards navigation. And as it’s going to be used several times, it makes sense to create a separate component. In this certain case an icon is used instead of a title (when you press on the button).
index.js
styled_components/ # Styled components of Header Back Button component
StyledAllText.js
StyleTouchableOpacity.js
Wrapper.js
index.js

```
<HeaderBackButton
        imagePath={icons.back}
        onPress={() => navigation.goBack()}
      />
```

Figure 16. Usage of HeaderBackButton component example

Properties of HeaderBackButton component:

- imagePath => number: Path to the image that will be served as a view for button. (required)
- onPress => function: Handler to be called when the user press the button. (required)
- ...rest => all the other props inherited from TouchableOpacity

PrivacyButton/ # Button component. See Figure 9 in Appendix 1 for code samples. This button is a text buttons (“Terms of Use” “Privacy Policy”), which are marked in red color.

```
    index.js
    styled_components/          # Styled components of Privacy Button component
      StyledText.js
      Wrapper.js
      index.js
```

```
<PrivacyButton
          emphasizeColor={Theme.primary_red}
          onPress={() => console.log('pressed')}
        />
```

Figure 17. Usage of PrivacyButton component example

Properties of PrivacyButton component:

- emphasizeColor => string: A color that will emphasize important words. (required)
- onPress => function: Handler to be called when the user press the button. (required)
- ...rest => all the other props inherited from TouchableOpacity

TextButton/ # Button component. Clickable text (e.g. 'Forgot password button') See Figure 10 in Appendix 1 for code samples. The functionality of this component is similar to the previous one, but this component can be used for absolutely any text button.

index.js
styled_components/ # Styled components of Text Button component
StyledText.js
StyledTouchableOpacity.js
index.js

```
<TextButton
        greyText="Forgot your"
        redText="password?"
        onPress={() => navigation.navigate('ForgotPassword')}
       />
```

Figure 18. Usage of TextButton component example

Properties of TextButton component:

- greyText => string: Text that will be displayed grey-colored. required
- redText => string: Text that will be displayed red-colored. required
- onPress => function: Handler to be called when the user press the button. Required
- ...rest => all the other props inherited from TouchableOpacity

checkbox/ # Check box component. See Figure 11 in Appendix 1 for code samples.

CheckBoxIcon.js
index.js
styled_components/ # Styled components of Check box component
StyledContainer.js
StyledTitle.js
index.js

```
<CheckBox
      checked={this.state.checked}
      title="Check me"
      titleColor="#fff"
      onPress={this.handleCheck}
      checkedColor="#00ff00"
      uncheckedColor="#0000ff"
     />
```

Figure 19. Usage of CheckBox component example

Properties of CheckBox component:

- checked => *boolean*: Flag for checking the icon. required
- title => *string*: Title of checkbox. (not-required)
- *Default*: 'Check Me'.
- TitleColor => *string*: Color of the checkbox title. (not-required)
- *Default*: Black color.
- OnPress => *function*: Function for the checkbox. (required)
- checkedColor => *string*: Color the checkbox icon when 'checked' is 'true'. (not-required)
- *Default*: Red color.
- UncheckedColor => *string*: Color the checkbox icon when 'checked' is 'false'. (not-required)
- *Default*: Grey'ish color.

date-picker/ # Date picker component. See Figure 12 -14 in Appendix 1 for code samples. This component vidpovidae for birth date selection. You won't be ablt to select invalid date (less then 14 years from now), since users have to be at least 14 years old.

```
index.js
styled_components/          # Styled components of Date picker component
  DatePickerInnerView.js
  DatePickerOuterView.js
  StyledText.js
  index.js
```

```
<DatePicker
            iconPosition="left"
            icon={icons.calendar}
            text="Date of Birth"
            minimumDate={
              new Date(
                new Date().getFullYear() - 110,
                new Date().getMonth()
              )
            }
            maximumDate={
              new Date(
                new Date().getFullYear() - 14,
                new Date().getMonth()
              )
            }
            getDate={date =>
              formikProps.setFieldValue('birthDate', date, true)
            }
          />
```

Figure 20. Usage of Date-Picker component example

Properties for Date-Picker component:

- icon => number: Path to the local icon allocation. (not-required)
- iconPosition => string: String either 'right' or 'left' that points on the position of icon. (not-required)
- text => string: Field name. (required)
- minimumDate => date: Minimum date. (required)
- maximumDate => date: Maximum date. (required)
- getDate => function: Helper that sends selected date to the parent component, so it is possible to use date further. (required)

gender-picker/ # Gender picker component. See Figure 15-17 in Appendix 1 for code samples. Is used for user's gender selection

index.js

```
<GenderPicker
            iconPosition="left"
            icon={icons.gender}
            value={gender}
            onValueChange={value => this.setState({ gender: value })}
          />
```

Figure 21. Usage of Gender-Picker component example

Properties for Gender-Picker component:

- icon => number: Path to the local icon allocation. (not-required)
- iconPosition => string: String either 'right' or 'left' that points on the position of icon. (not-required)
- value => string: Picker value. (required)
- onValueChange => function: Function that handles change of the value. (required)

input-fields/ # Component of input fields. See Figure 18 Appendix 1 for code samples. This component is used for entering data to input fiels (e.g email, password, username, etc)

```
TextInput.js
styled_components/          # Styled components of input field component
  Icon.js
   StyledTextInput.js
    StyledView.js
     index.js
```

```
<TextInput
            icon={icons.email}
            iconPosition="left"
            onChangeText={formikProps.handleChange('email')}
            onBlur={formikProps.handleBlur('email')}
            placeholder="email@example.com"
            autoFocus
           />
```

Figure 22. Usage of Input-fields component example

Properties for Input-fields component:

- icon => number:
- Path to the local icon allocation. (not-required)
- iconPosition => string:
- String either 'right' or 'left' that points on the position of icon. (not-required)
- onChangeText => function:
- Handler to be called when the user enters text. not-required
- onBlur => function:
- Callback that is called when the text input is blurred. (not-required)
- ...rest => all the other props inherited from TextInput.

```
<LocationPicker
                  iconPosition="left"
                  icon={icons.marker}
                  text={`${
                    latitude && longitude ? 'Your location is set!' :
'Location'
                  }`}
                  onPress={this.getLocation}
                 />
```

Figure 23. Usage of LocationPicker component example

locatin-picker/ # Component of location picker. See Figure 19 in Appendix 1 for code samples. This component gets the vales of latitude and longitude of user's location in case if use gives his permission
index.js

- Properties for LocationPicker component:
- icon => *number*: Path to the local icon allocation. (not-required)
- iconPosition => *string*: String either 'right' or 'left' that points on the position of icon. (not-required)
- text => *string*: Field name. (required)
- onPress => *function*: Handler to be called when user will press on a button. (required)

text/ # Text Component. See Figure 20 in Appendix 1 for code samples. This component is responsible for Header's text on the pages.
TextElement/
index.js
styled_components/ # Styled components of text component

```
<TextElement header title="Sign Up" />
```

Figure 24. Usage of Text Element component example

Properties for TextElement component:

- header => *boolean*: Either 'true' or 'false' that points whether component header or not. (required)
- color => *string*: Color of the text. (not-required)
- title => *string*: Displayed text. (required)

6. Implementing client-side of application

6.1 Screens example and description

Fig. 25 Splash Screen

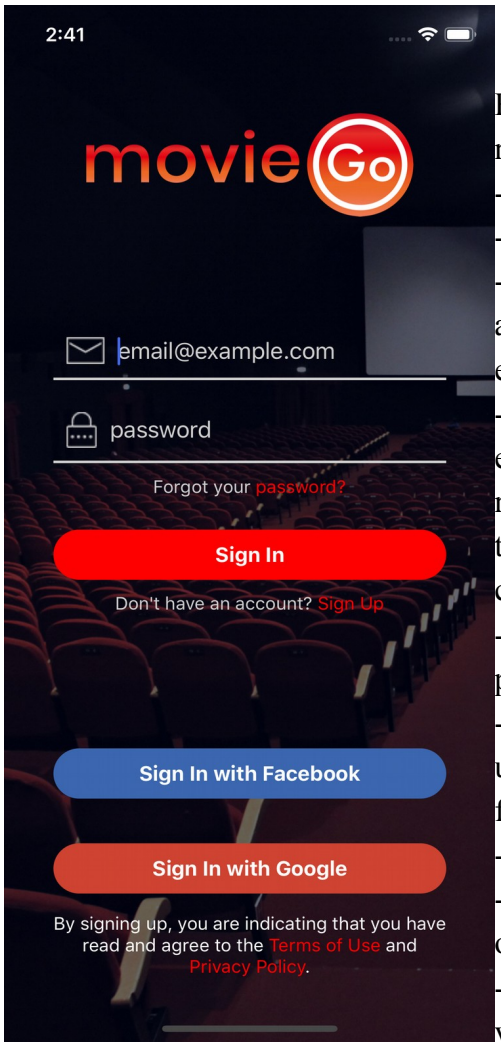


Fig.26 Initial Screen

Fig. 26 Shows the initial screen which user see right after the splash screen. It contains:
- the MovieGO logo on the top;
- 2 fields for email and password;
- "Forgot your password" button, which leads to another screen (Shown in Fig 27.) to enter your email to get a password recovery link;
- "Sign in" button, which lets the user in, ones he enters the correct email and password (Fig. 28) or recognizes that the fields are empty (Fig. 29) or that the email or/and password were not entered correctly (Fig. 30).
-"Sign Up" button, which leads to the Sing Up page (Fig. 32)
- Sign In with Facebook button, which redirects users to the Facebook app, where they can confirm their sign in.
- Sign In with Google button
- "Terms of Use" button, which opens the Terms of Use of the application.
- "Privacy Policy" button, which opens the Privacy Policy of the application.

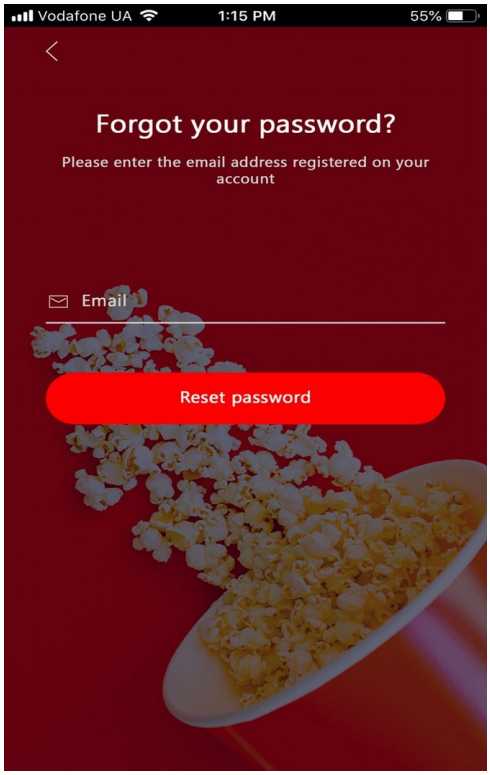


Fig. 28 Right email, valid length of the password

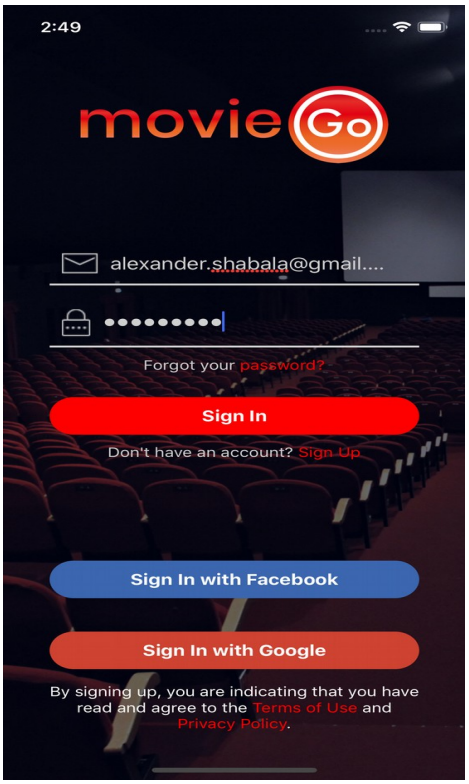


Fig. 27 "Forgot your password" screen

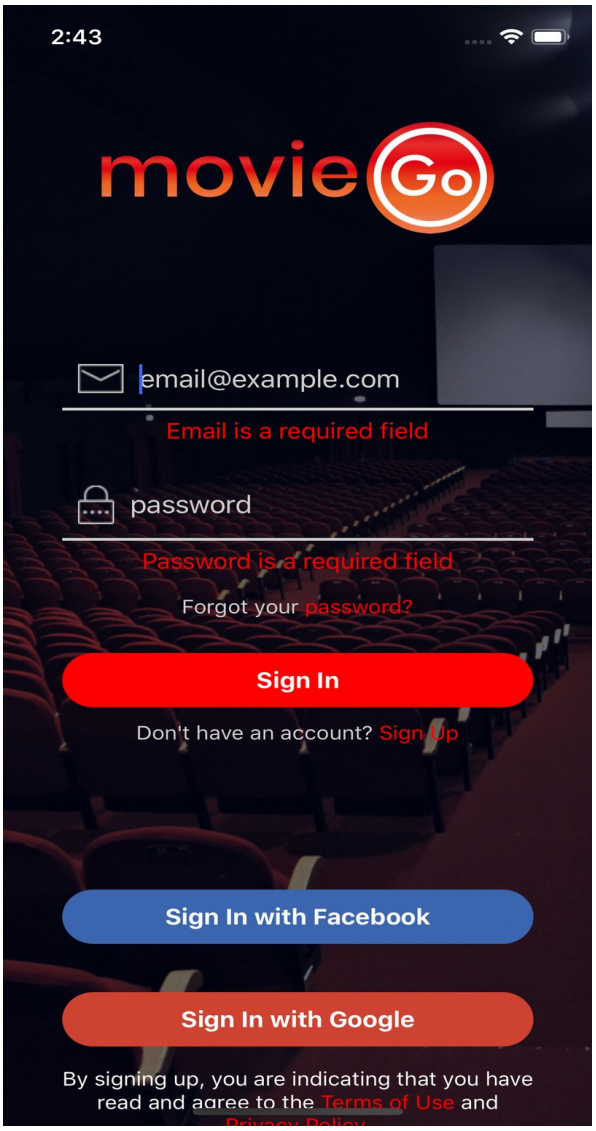


Fig. 29 Sing In pressed when the fields are empty

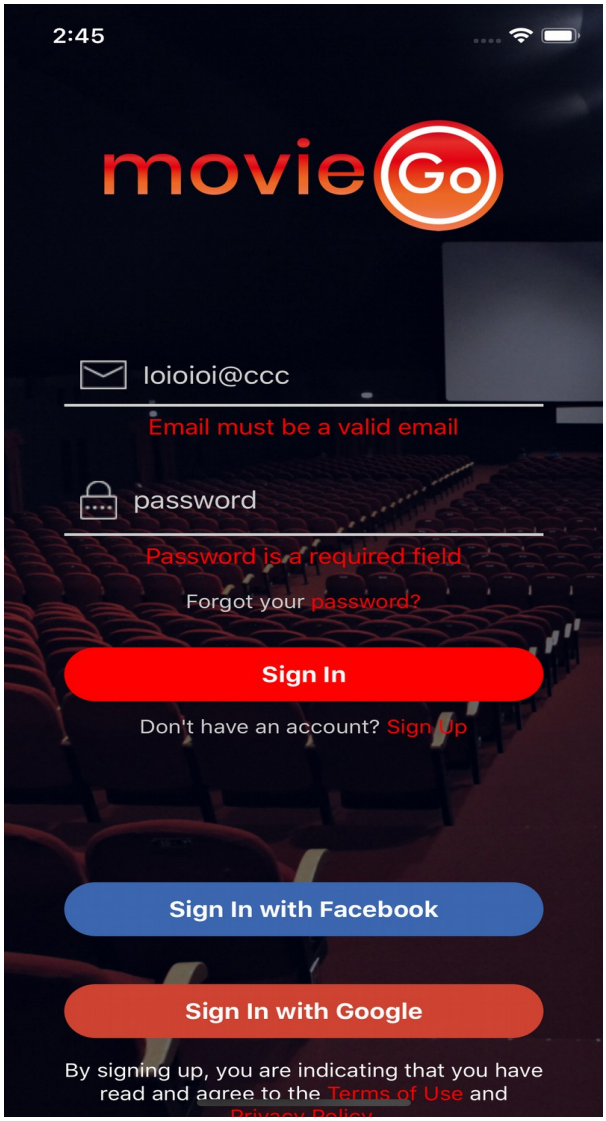


Fig. 30 Sing In pressed when email is not valid

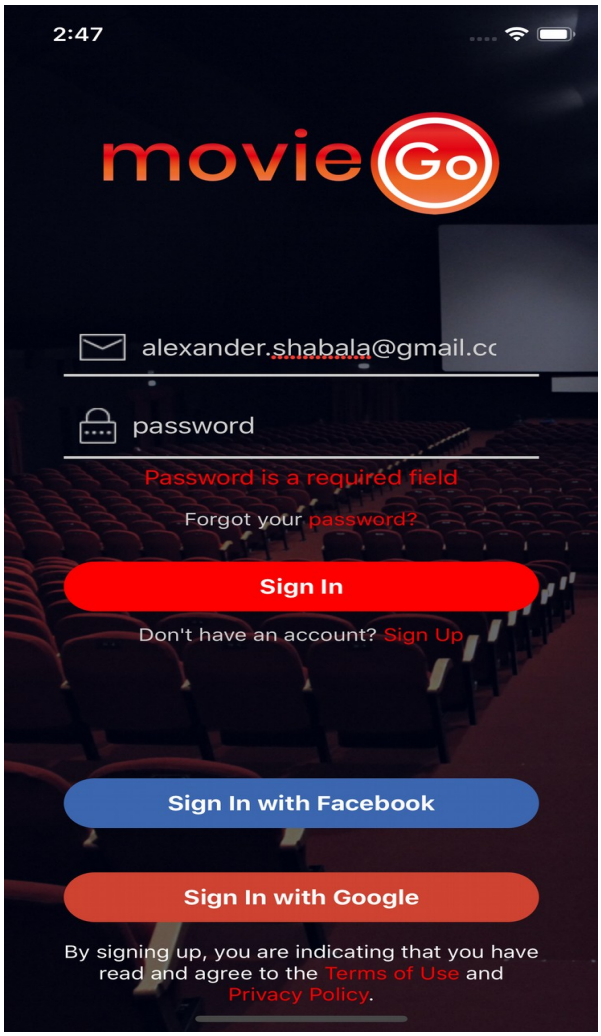


Fig. 31 Right Email, no password

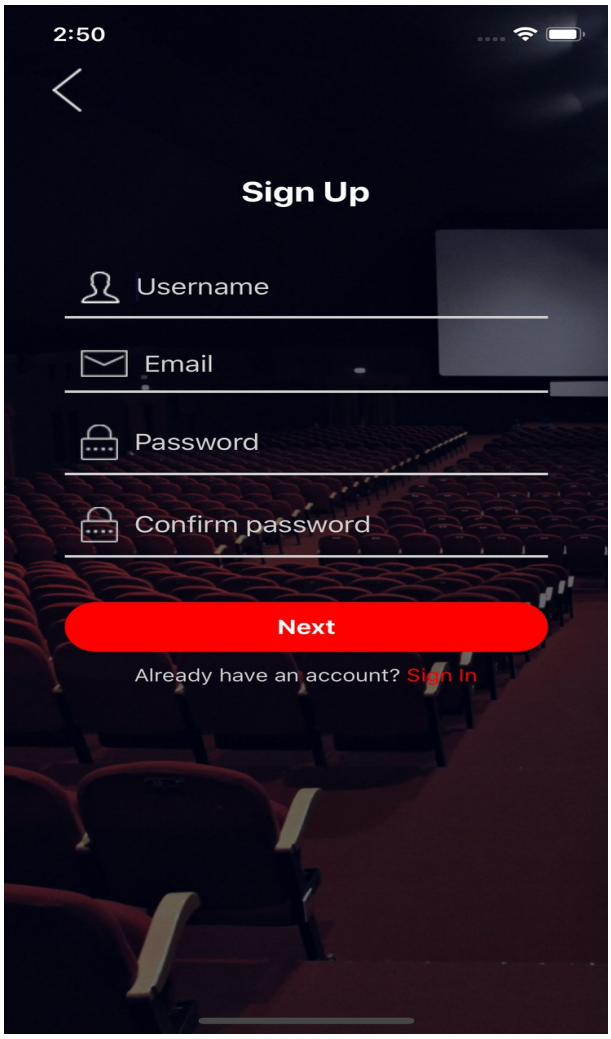


Fig. 32 “Sign Up” screen

- When the email is entered correctly, but the password field is still empty – it will accept the email and will not show any error message under email field, but will still show “Password is a required field” under the Password field, as shown in Fig. 31.

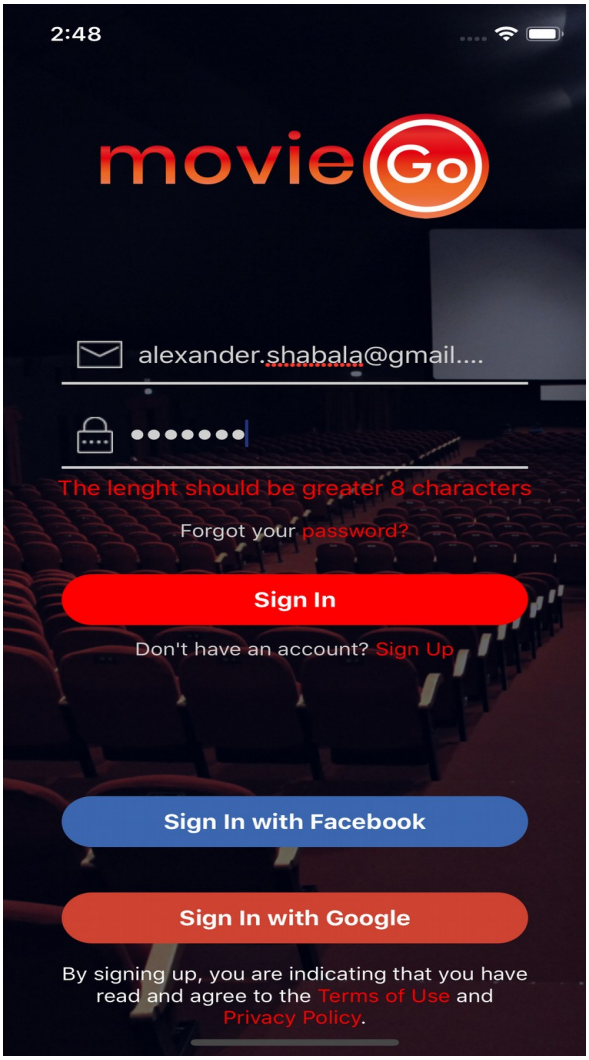


Fig. 33 Not valid length of the password

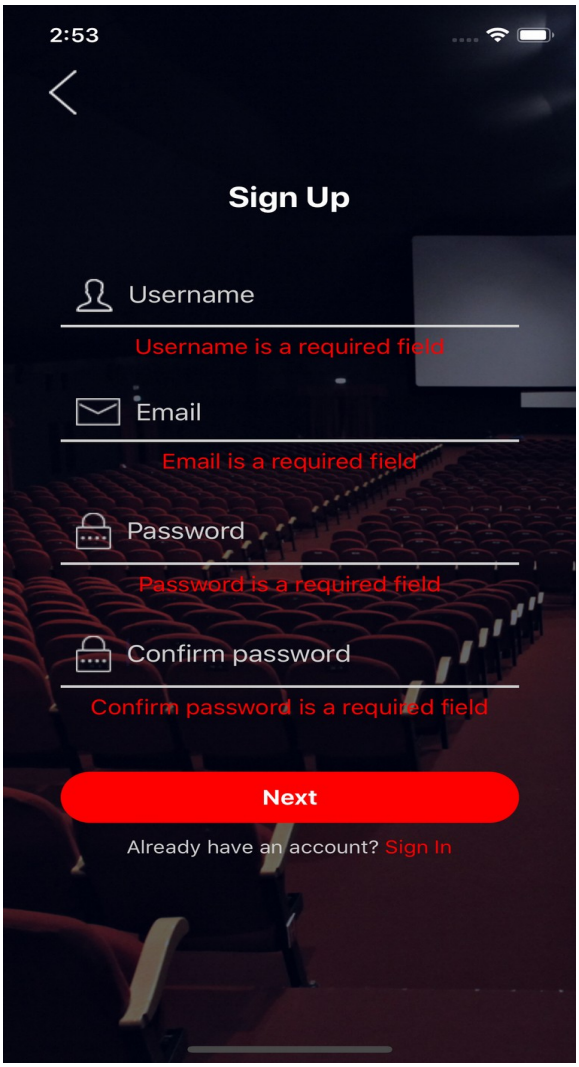


Fig. 34 SingUp Screen Error, everything is empty

- When the email is entered correctly and the written password is shorter than 8 characters – the message “The length should be greater 8 characters” displays, as shown in Fig. 33.

- When the user is on the Sign Up page but hasn’t filled any info – it will show error messages under each field, like shown on Fig. 34.

If user has filled all the information on the sign up page – it will not show any errors and user will be able to click next and proceed to the next page (Fig. 35), where he/she also will need to fill all the fields (Fig. 36), and after that user will be able to access the application.

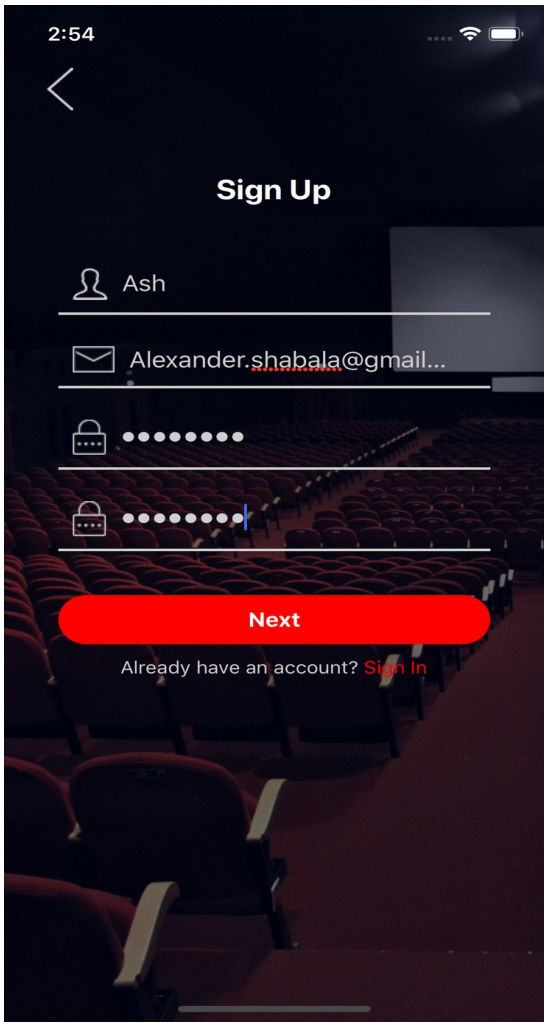


Fig. 35 All the information filled in on the Sign Up page

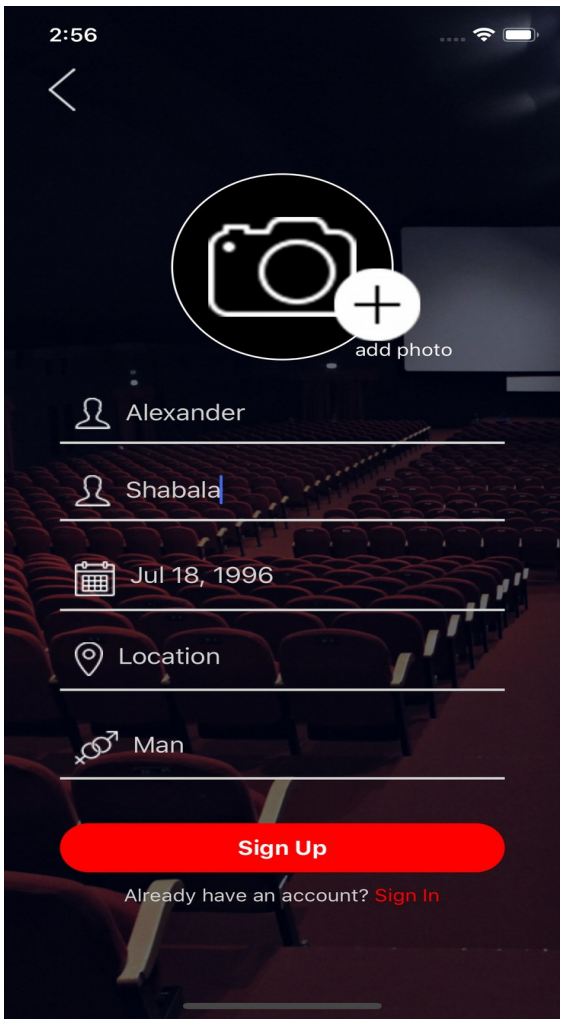


Fig. 36 Personal information fields are filled in

6.2 CLIENT DESCRIPTION

6.2.1 Navigation:

Auth Flow and Auth Navigation behaves in the following way:

In App.js file (which is a root one), with the help of react-navigation package, the Navigation Stack is formed:

```
createSwitchNavigator(
    {
      AuthLoading: AuthLoadingScreen,
      App: AppStack,
      Auth: AuthStack
    },
    {
      initialRouteName: 'AuthLoading'
    }
  )
```

Figure 37. Navigation Stack

Here 3 main Stacks are created:

1) AuthLoading - Which is the starting point for the application.

2) App is the internal navigation stack (Profile, Movies List and all other routes that should be protected).

3) Auth is the Authentication stack that checks and handles all the stuff related to the User permissions, logging in and out.

AuthLoadingScreen is the starting point for the mobile application:

When the app is loaded, inside of the AuthLoadingScreen constructor, the following check is executed:

```
const token = await AsyncStorage.getItem('x-auth');
this.props.navigation.navigate(token ? 'App' : 'Auth');
```

Figure 38. AuthLoadingScreen

I am checking there whether I have a token in the local device storage. If there is a token, navigation redirects the user to the Profile Screen, where token is passed to the server and server validates it. If everything is OK, the access granted for user and he can navigate inside of the app, otherwise the navigation redirects user to the Auth Stack (it is a SignIn Screen) and user has to log in.
The token is saved to the local storage using the _handleLogin() function from SignIn.js file:

```
  const { navigation } = this.props;
        try {
          // get token from header
          const token = res.headers['x-auth'];
          if (token) {
            // save token to the local storage
            AsyncStorage.setItem('x-auth', token)
              .then(() => {
                navigation.navigate('App');
              })
              .catch(() => {
                Alert.alert('Error', 'There was a problem with logging in.');
              });
          }
        } catch {
          Alert.alert('Error', 'There was a problem with logging in.');
        }
```

Figure 39. HandleLogin function

The function above takes token from header and tries to save it into the device local storage. If that was successful, the user will be redirected to the ProfileScreen where I do the check whether token is valid. Otherwise, app will redirect the user to the Auth screen.

```
_checkToken = () => {
    const { navigation } = this.props;
    AsyncStorage.getItem('x-auth')
      .then(token =>
        Axios.get('http://localhost:4444/user/account', {
          headers: {
            'x-auth': token
          }
        })
          .then(res => {
            this.setState({
              res: JSON.stringify(res.status)
            });
          })
          .catch(() => {
            Alert.alert('Error', 'Credentials are not valid');
            navigation.navigate('Auth');
          })
      )
      .catch(() => {
        Alert.alert('Error', 'Credentials are not valid');
        navigation.navigate('Auth');
      });
  };
```

Figure 40. Auth screen

6.2.2 Validation Process

Validation is needed for checking the correctness of input information (email, password, username).

- Email should be written in one string and have the following form: <example@email.com>
In case if it is written in a different form – it will show an error “Incorrect email address”.

- Password should be minimum 8 characters long.

- Username shall be minimum 4 characters long. Also it should not contain symbols like “-,<>, @” and system usernames cannot be used (admin, etc). Also, in case if user choose a username that already exists – an error “User already exists” will be shown.

The most popular components for validation are yup, formik and redux. Redux is mainly used for global state management, which is currently not an option for this application at the moment. Currently we are sticked to yup and formik.

YUP & Formik

- A **Yup schema** is an immutable object responsible for validating an object and defines the validation schema (shown in Figure 41 below),:

Sign in screen example:

```
// schema for validation fields
const validationSchema = yup.object().shape({
  email: yup   - schema with predefined functions for validation of email field is created.
    .string() - defines that email should be written in a string.
    .label('Email') – defines the name of the field that is validated.
    .email() - instruction that checks this field in order to ensure that current string is written as an email (example@gmail.com)
```

.required(), - shows that current field is required for filling.

password: yup - schema with predefined functions for validation of password field is created.

.string() -defines that password should be written in a string.

.label('Password') - defines the name of the field that is validated.

.required() - shows that current field is required for filling.

.min(8, 'The lenght should be greater 8 characters')

});

Figure 41. Yup schema example

- <Formik/> is a component that helps you with building forms. It uses a render props pattern made popular by libraries like React Motion and React Router.

Pros of Formik:

- Declarative

Formik takes care of the repetitive and annoying stuff - keeping track of values/errors/ visited fields, orchestrating validation, and handling submission - so you don't have to. This means you spend less time wiring up state and change handlers and more time focusing on your business logic.

- Intuitive

No fancy subscriptions or observables under the hood, just plain React state and props. By staying within the core React framework and away from magic, Formik makes debugging, testing, and reasoning about your forms a breeze. If you know React, and you know a bit about forms, you know Formik.

- Adoptable

Since form state is inherently local and ephemeral, Formik does not use external state

management libraries like Redux or MobX. This also makes Formik easy to adopt incrementally and keeps bundle size to a minimum.

Formik serves as a component, which uses a yup schema for validation purposes:
Example:

```
<Formik
        initialValues={{
          email: '',
          password: ''
        }}
        onSubmit={(values, actions) => {                    // accepts validated values
          this.handleLogin(values.email, values.password);  // makes a request to server
and post data
          actions.setSubmitting(false);                     // stop the data transferring
        }}
        validationSchema={validationSchema}                 // schema defined by yup (see
Figure 41)
      >
{formikProps => (
  ... // custom components
{formikProps.isSubmitting ? (                               // if submitting values are sent
then activity indicator is shown
            <ActivityIndicator />
          ) : (
            <DefaultButton
              title="Sign In"
              onPress={formikProps.handleSubmit}            // triggers form submission
            />
          )}
        </View>
      )}
    </Formik>
```

Figure 42. Formik component

7. Summary

During this work I reviewed different ways of building an application for iOS and Android platforms and performed the following tasks:

1. Analyzed both platforms for mobile application development (iOS and Android) and defined different scenarios in which a certain platform would be more efficient to go with and why.
2. Analyzed the React Native framework for mobile applications development, it's advantages and disadvantages.
3. Analyzed the case with specific platform native development, it's advantages and disadvanteges.
4. Compared both of the above mentioned ways, peculiarities of development in each of these methods and found out alternatives which could applied in particular solutions. Selected React Native as more efficient way of building the mobile application exactly for this case.
5. Created Minimum Viable Product, based on the product's specifications and requirements and on the existent problems that MovieGo application offers solution to. Defined the ways of implementing these solutions, why is it important to do that. Analyzed alternatives that do exist today and the ones that can be expected in the future.
6. Defined the ways of improving the MVP, ways of developing future updates and possible redesigning of the application.
7. Described the structure of the application, it's peculiarities and tools for developing UI toolkit for the React Native.
8. Described the client side of the application.

References

[1] "iOS vs Android Development: Which One is Best for Your App'?, Daria R. [Internet Source] - 20.06.2018

[2] "Mobile Operating System Market Share Europe", Statcounter, LLC [Internet Source] - 01.04.2019

[3] "Despite Android's growing market share, Apple users continue to spend twice as much money on apps as Android users", Prachi Bhardwaj and Shayanne Gal, [Internet Source] - 06.07.2018

[4] "Mobile Vendor Market Share Worldwide", Statcounter, GlobalStats [Internet Source] - 01.04.2019

[5] "How I built my first React Native app for my first freelance client", Charlie Jeppsson [Internet Source] - 02.04.2019

[6] ""What do you dislike about React Native", Christoph Nakazawa [Internet Source] - 27.02.2019

[7] "Build native mobile apps using JavaScript and React", GitHub [Internet Source] - 02.01.2019

[8] "How to Add a Splash Screen to a React Native App (iOS and Android)", Spencer Carli [Internet Source] - 27.02.2019

[9] "Routing and navigation for your React Native apps", React Navigation [Internet Source] – 15.01.2019

[10] "React Native Pros and Cons - Facebook’s Framework in 2019 (update)", Natalia Chrzanowska [Internet Source] - 07.03.2019

Appendix – [Program Code Examples]

```
import React from 'react';
import {
View,
Text,
Image,
Platform,
StyleSheet,
TouchableOpacity
} from 'react-native';
import PropTypes from 'prop-types';

import Theme from '../_configs/Theme';
import ViewPropTypes from '../_configs/ViewPropTypes';

const DEFAULT_SIZES = {
small: 34,
medium: 50,
large: 75,
xlarge: 150
};

const Avatar = props => {
const {
size,
onPress,
backgroundIcon,
iconAddPhoto,
containerStyle,
overlayContainerStyle,
source,
avatarStyle,
activeOpacity,
showAddPhotoButton,
editButton,
imageProps,
...attributes
} = props;
```

Figure 43. Code example of Avatar Component.

```
const iconDimension =
typeof size === 'number'
? size
: DEFAULT_SIZES[size] || DEFAULT_SIZES.small;

let height;
const width = (height = iconDimension);

const renderUtils = () => {
if (showAddPhotoButton) {
const editButtonProps = { ...editButton };

const defaultEditButtonSize = (width + height) / 2 / 3;
const editButtonSize = editButton.size || defaultEditButtonSize;
const editButtonSizeStyle = {
width: editButtonSize,
height: editButtonSize,
borderRadius: editButtonSize / 2
};

return (
<View
style={{
position: 'absolute',
bottom: 0,
right: 0,
alignItems: 'center',
justifyContent: 'center'
}}
>
<View
style={[
styles.editButton,
editButtonSizeStyle,
editButtonProps.style
]}
underlayColor={editButtonProps.underlayColor}
>
<View>
<Image source={iconAddPhoto} />
</View>
</View>
<Text style={{ color: '#fff', left: 40, bottom: 0 }}>add photo</Text>
</View>
);
}
};
```

Figure 44. Code example of Avatar Component - Prod

```
const iconDimension =
typeof size === 'number'
? size
: DEFAULT_SIZES[size] || DEFAULT_SIZES.small;

let height;
const width = (height = iconDimension);

const renderUtils = () => {
if (showAddPhotoButton) {
const editButtonProps = { ...editButton };

const defaultEditButtonSize = (width + height) / 2 / 3;
const editButtonSize = editButton.size || defaultEditButtonSize;
const editButtonSizeStyle = {
width: editButtonSize,
height: editButtonSize,
borderRadius: editButtonSize / 2
};

return (
<View
style={{
position: 'absolute',
bottom: 0,
right: 0,
alignItems: 'center',
justifyContent: 'center'
}}
>
<View
style={[
styles.editButton,
editButtonSizeStyle,
editButtonProps.style
]}
underlayColor={editButtonProps.underlayColor}
>
<View>
<Image source={iconAddPhoto} />
</View>
</View>
<Text style={{ color: '#fff', left: 40, bottom: 0 }}>add photo</Text>
</View>
);
}
};
```

Figure 45. Code example of Avatar Component – Continuation

```
const renderContent = () => {
if (source) {
return (
<Image
style={[
styles.avatar,
{ borderRadius: width / 2 },
avatarStyle && avatarStyle
]}
source={source}
{...imageProps}
/>
);
}
if (backgroundIcon) {
return <Image source={backgroundIcon} />;
}
};
const styles = StyleSheet.create({
container: {
backgroundColor: 'transparent',
width,
height
},
avatar: {
width,
height
},
overlayContainer: {
flex: 1,
alignItems: 'center',
borderWidth: 1.2,
borderColor: '#fff',
backgroundColor: Theme.avatar_default_color,
alignSelf: 'stretch',
justifyContent: 'center',
position: 'absolute',
top: 0,
left: 0,
right: 0,
bottom: 0
},
editButton: {
position: 'absolute',
bottom: 20,
left: 30,
alignItems: 'center',
justifyContent: 'center',
backgroundColor: Theme.avatar_default_color,
...Platform.select({
ios: {
shadowColor: Theme.primary_red,
shadowOffset: { width: 1, height: 1 },
shadowRadius: 2,
shadowOpacity: 0.5
},
android: {
elevation: 1
}
})
}
});
```

Figure. 46 Code example of Avatar Component – Continuation

```
return (
<TouchableOpacity
onPress={onPress}
activeOpacity={activeOpacity}
style={[
styles.container,
{ borderRadius: width / 2 },
containerStyle && containerStyle
]}
{...attributes}
>
<View
style={[
styles.overlayContainer,
{ borderRadius: width / 2 },
overlayContainerStyle && overlayContainerStyle
]}
>
{renderContent()}
</View>
{renderUtils()}
</TouchableOpacity>
);
};
Avatar.propTypes = {
size: PropTypes.oneOfType([
PropTypes.oneOf(['small', 'medium', 'large', 'xlarge']),
PropTypes.number
]),
onPress: PropTypes.func,
backgroundIcon: PropTypes.number,
showAddPhotoButton: PropTypes.bool,
iconAddPhoto: PropTypes.number,
containerStyle: PropTypes.objectOf(PropTypes.any),
overlayContainerStyle: PropTypes.objectOf(PropTypes.any),
source: Image.propTypes.source,
avatarStyle: PropTypes.objectOf(PropTypes.any),
activeOpacity: PropTypes.number,
editButton: PropTypes.shape({
size: PropTypes.number,
iconName: PropTypes.string,
iconType: PropTypes.string,
iconColor: PropTypes.string,
underlayColor: PropTypes.string,
style: ViewPropTypes.style
}),
imageProps: PropTypes.objectOf(PropTypes.object)
};
```

Figure 47. Code example of Avatar Component – Continuation

```
Avatar.defaultProps = {
size: 'small',
onPress: () => {},
backgroundIcon: null,
showAddPhotoButton: false,
iconAddPhoto: null,
containerStyle: {},
overlayContainerStyle: {},
source: null,
avatarStyle: {},
activeOpacity: 0.5,
editButton: {
size: null,
iconName: 'mode-edit',
iconType: 'material',
iconColor: '#fff',
underlayColor: Theme.primary_red,
style: null
},
imageProps: {}
};
export default Avatar;
```

Figure 48. Code example of Avatar Component – Continuation

```
import React from 'react';
import Wrapper from '../../_common-styled-components/Wrapper';
import { Title, StyledTouchableOpacity } from './styled-components/index';

const DefaultButton = ({
backgroundColor,
title,
onPress,
disabled,
...rest
}) => (
<Wrapper>
<StyledTouchableOpacity
backgroundColor={backgroundColor}
onPress={onPress}
disabled={disabled}
{...rest}
>
<Title>{title}</Title>
</StyledTouchableOpacity>
</Wrapper>
);

DefaultButton.defaultProps = {
backgroundColor: null,
onPress: null,
disabled: false
};

DefaultButton.propTypes = {
backgroundColor: PropTypes.string,
title: PropTypes.string.isRequired,
onPress: PropTypes.func,
disabled: PropTypes.bool
};

export default DefaultButton;
```

Figure 49. Code example of DefaultButton Component

```
 import React from 'react';
import { Image } from 'react-native';
import PropTypes from 'prop-types';


import {
Wrapper,
StyledTouchableOpacity,
StyledAltText
} from './styled-components/index';


const HeaderBackButton = ({ imagePath, onPress, ...rest }) => (
<Wrapper>
<StyledTouchableOpacity onPress={onPress} {...rest}>
{imagePath ? (
<Image source={imagePath} />
) : (
<StyledAltText>{'< Back'}</StyledAltText>
)}
</StyledTouchableOpacity>
</Wrapper>
);


HeaderBackButton.defaultProps = {
imagePath: null
};


HeaderBackButton.propTypes = {
onPress: PropTypes.func.isRequired,
imagePath: PropTypes.number
};


export default HeaderBackButton;
```

Figure 50. Code example of HeaderBackButton Component

```
 import React from 'react';
import { TouchableOpacity } from 'react-native';
import PropTypes from 'prop-types';

import { Wrapper, StyledText } from './styled-components/index';

// --- Text as constants for easier manipulation
const BLANK_SPACE = ' ';
const FULL_STOP = '.';
const AND = `and${BLANK_SPACE}`;
const TERMS_OF_USE = `Terms of Use${BLANK_SPACE}`;
const PRIVACY_POLICY = 'Privacy Policy';
const PRIVACY_BTN_TITLE = `By signing up, you are indicating that you have read and agree to the$
{BLANK_SPACE}`;
// ---

const PrivacyButton = ({ emphasizeColor, onPress, ...rest }) => (
<Wrapper>
<TouchableOpacity onPress={onPress} {...rest}>
<StyledText>
{PRIVACY_BTN_TITLE}
<StyledText emphasizeColor={emphasizeColor}>{TERMS_OF_USE}</StyledText>
<StyledText>{AND}</StyledText>
<StyledText emphasizeColor={emphasizeColor}>
{PRIVACY_POLICY}
</StyledText>
<StyledText>{FULL_STOP}</StyledText>
</StyledText>
</TouchableOpacity>
</Wrapper>
);

PrivacyButton.propTypes = {
emphasizeColor: PropTypes.string.isRequired,
onPress: PropTypes.func.isRequired
};

export default PrivacyButton;
```

Figure 51. Code example of Privacy Button Component

```
 import React, {Fragment} from 'react';
import PropTypes from 'prop-types';

import Theme from '../../_configs/Theme';
import { StyledTouchableOpacity, StyledText } from './styled-components/index';

const TextButton = ({ greyText, redText, onPress, ...rest }) => (
<Fragment>
<StyledTouchableOpacity onPress={onPress} {...rest}>
<StyledText color={Theme.primary_grey}>
{greyText}
<StyledText color={Theme.primary_red}>{` ${redText}`}</StyledText>
</StyledText>
</StyledTouchableOpacity>
</Fragment>
);

TextButton.propTypes = {
greyText: PropTypes.string.isRequired,
redText: PropTypes.string.isRequired,
onPress: PropTypes.func.isRequired
};

export default TextButton;
```

Figure 52. Code example of TextButton Component

```
 import React from 'react';
import PropTypes from 'prop-types';
import { TouchableOpacity } from 'react-native';

import Theme from '../_configs/Theme';

import CheckBoxIcon from './CheckBoxIcon';
import { StyledContainer, StyledTitle } from './styled-components/index';

const CheckBox = props => {
const { ...rest } = props;

const {
checked,
title,
titleColor,
onPress,
checkedColor,
...attributes
} = rest;

return (
<TouchableOpacity {...attributes} onPress={onPress}>
<StyledContainer>
<CheckBoxIcon {...props} checkedColor={checkedColor} />
<StyledTitle checked={checked} titleColor={titleColor}>
{title}
</StyledTitle>
</StyledContainer>
</TouchableOpacity>
);
};

CheckBox.defaultProps = {
center: false,
title: 'Check Me',
titleColor: `${Theme.primary_black}`
};

CheckBox.propTypes = {
...CheckBoxIcon.propTypes,
title: PropTypes.string,
titleColor: PropTypes.string,
center: PropTypes.bool,
onPress: PropTypes.func.isRequired
};

export default CheckBox;
```

Figure 53. Code example of CheckBox Component

```
 import React, { Component } from 'react';
import {
Modal,
View,
Alert,
Platform,
TouchableOpacity,
DatePickerIOS,
DatePickerAndroid
} from 'react-native';
import PropTypes from 'prop-types';
import moment from 'moment'; // prettify time from Date() object

import Wrapper from '../_common-styled-components/Wrapper'; // common wrapper
// styled components from 'input-field'
import { Icon } from '../input-fields/styled-components/index';
import { TextElement, DefaultButton } from '../../index';
import Theme from '../_configs/Theme';
// styled components for the 'date-picker'
import {
DatePickerOuterViewIOS,
DatePickerInnerViewIOS,
StyledText
} from './styled-components/index';

class DatePicker extends Component {
state = {
isVisible: false,
chosenDate: new Date(),
isChosenDate: false // triggers when chosenDate is specified
};

openIOSDatePicker = visible => {
const { isVisible } = this.state;
this.setState({
isVisible: visible
});
if (isVisible === false) {
this.setState({
isChosenDate: false
});
} else {
this.setState({
isChosenDate: true
});
}
};

setIOSDate = newDate => {
this.setState({ chosenDate: newDate });
};
```

Figure 54. Code example of date-picker component

```
 openAndroidDatePicker = async (minimumDate, maximumDate) => {
try {
DatePickerAndroid.open({
date: minimumDate,
minDate: maximumDate,
maxDate: minimumDate
}).then(date => {
if (date.action !== DatePickerAndroid.dismissedAction) {

this.setState({
chosenDate: date,
isChosenDate: true
});
}
});
} catch ({ code, message }) {
this.setState({
isChosenDate: false
});
Alert.alert(`Error ${code}`, `${message}`);
}
};

render() {
const { isVisible, chosenDate, isChosenDate } = this.state;
const { iconPosition, icon, text, minimumDate, maximumDate } = this.props;
return (
<Wrapper>
<View style={{ marginTop: 22 }}>
<Modal
animationType="slide"
transparent
visible={isVisible}
onRequestClose={() => {
Alert.alert('Modal has been closed.');
}}
>
<DatePickerOuterViewIOS>
<DatePickerInnerViewIOS>
<TextElement
header
title="Choose the Birth Date"
color={Theme.primary_grey}
/>
<DatePickerIOS
mode="date"
date={chosenDate}
minimumDate={minimumDate}
maximumDate={maximumDate}
onDateChange={this.setIOSDate}
/>
<DefaultButton
backgroundColor={Theme.primary_red}
title="Confirm"
onPress={() => {
this.openIOSDatePicker(false);
}}
/>
</DatePickerInnerViewIOS>
</DatePickerOuterViewIOS>
</Modal>
</View>
```

Figure 55. Code example of date-picker component – continuation

```
 <TouchableOpacity
onPress={() =>
Platform.OS === 'android'
? this.openAndroidDatePicker(maximumDate)
: this.openIOSDatePicker(true)
}
style={{ width: '80%' }}
>
<View
style={{
flexDirection: 'row',
padding: 10,
borderBottomWidth: 2,
borderBottomColor: Theme.primary_grey
}}
>
{iconPosition === 'left' && (
<Icon iconPosition={iconPosition} source={icon} />
)}
<StyledText icon={icon}>
{isChosenDate ? moment(chosenDate).format('ll') : text}
</StyledText>
{iconPosition === 'right' && (
<Icon iconPosition={iconPosition} source={icon} />
)}
</View>
</TouchableOpacity>
</Wrapper>
);
}
}

DatePicker.defaultProps = {
icon: null,
iconPosition: null
};

DatePicker.propTypes = {
icon: PropTypes.number,
iconPosition: PropTypes.string,
text: PropTypes.string.isRequired,
minimumDate: PropTypes.instanceOf(Date).isRequired,
maximumDate: PropTypes.instanceOf(Date).isRequired
};

export default DatePicker;
```

Figure 56. Code example of date-picker component - continuation

```
 import React, { Component } from 'react';
import {
Platform,
Modal,
View,
Alert,
TouchableOpacity,
Picker
} from 'react-native';
import PropTypes from 'prop-types';

import Wrapper from '../_common-styled-components/Wrapper'; // common wrapper
// styled components from 'input-field'
import { Icon } from '../input-fields/styled-components/index';
import { TextElement, DefaultButton } from '../../index';
import Theme from '../_configs/Theme';
// styled components for the 'date-picker'
import {
StyledText,
DatePickerOuterViewIOS,
DatePickerInnerViewIOS
} from '../date-picker/styled-components/index';

class GenderPicker extends Component {
state = {
isVisible: false
};

openModal = visible => {
this.setState({
isVisible: visible
});
};
```

Figure 57. Code example of gender-picker component

```
 render() {
const { isVisible } = this.state;
const { iconPosition, icon, value, onValueChange } = this.props;
return (
<Wrapper>
<View style={{ marginTop: 22 }}>
<Modal
animationType="slide"
transparent
visible={isVisible}
onRequestClose={() => {
Alert.alert('Modal has been closed.');
}}
>
<DatePickerOuterViewIOS>
{Platform.OS === 'android' ? (
<View
style={{ backgroundColor: 'white', width: 300, height: 150 }}
>
<TextElement
header
title="Please select gender..."
color="black"
/>
<Picker selectedValue={value} onValueChange={onValueChange}>
<Picker.Item label="Please select gender..." value="" />
<Picker.Item label="Man" value="Man" />
<Picker.Item label="Woman" value="Woman" />
</Picker>
<DefaultButton
backgroundColor={Theme.primary_red}
title="Confirm"
onPress={() => {
this.openModal(false);
}}
/>
</View>
) : (
<DatePickerInnerViewIOS>
<Picker selectedValue={value} onValueChange={onValueChange}>
<Picker.Item label="Please select gender..." value="" />
<Picker.Item label="Man" value="Man" />
<Picker.Item label="Woman" value="Woman" />
</Picker>
<DefaultButton
backgroundColor={Theme.primary_red}
title="Confirm"
onPress={() => {
this.openModal(false);
}}
/>
</DatePickerInnerViewIOS>
)}
</DatePickerOuterViewIOS>
</Modal>
</View>
```

Figure 58. Code example of gender-picker component - continuation

```
 <TouchableOpacity
style={{ width: '80%' }}
onPress={() => this.openModal(true)}
>
<View
style={{
flexDirection: 'row',
padding: 10,
borderBottomWidth: 2,
borderBottomColor: Theme.primary_grey
}}
>
{iconPosition === 'left' && (
<Icon iconPosition={iconPosition} source={icon} />
)}
<StyledText icon={icon}>
{value !== '' ? value : 'Please select gender...'}
</StyledText>
{iconPosition === 'right' && (
<Icon iconPosition={iconPosition} source={icon} />
)}
</View>
</TouchableOpacity>
</Wrapper>
);
}
}

GenderPicker.defaultProps = {
icon: null,
iconPosition: null
};

GenderPicker.propTypes = {
icon: PropTypes.number,
iconPosition: PropTypes.string,
value: PropTypes.string.isRequired,
onValueChange: PropTypes.func.isRequired
};

export default GenderPicker;
```

Figure 59. Code example of gender-picker component - continuation

```
 import React from 'react';
import PropTypes from 'prop-types';

import Theme from '../_configs/Theme';
import Wrapper from '../_common-styled-components/Wrapper';
import { StyledTextInput, StyledView, Icon } from './styled-components/index';

const TextInput = ({ icon, iconPosition, onChangeText, onBlur, ...rest }) => (
<Wrapper>
<StyledView>
{iconPosition === 'left' && (
<Icon iconPosition={iconPosition} source={icon} />
)}
<StyledTextInput
icon={icon}
onChangeText={onChangeText}
onBlur={onBlur}
placeholderTextColor={Theme.primary_grey}
{...rest}
/>
{iconPosition === 'right' && (
<Icon iconPosition={iconPosition} source={icon} />
)}
</StyledView>
</Wrapper>
);

TextInput.defaultProps = {
icon: null,
iconPosition: null,
onChangeText: null,
onBlur: null
};

TextInput.propTypes = {
icon: PropTypes.number,
iconPosition: PropTypes.string,
onChangeText: PropTypes.func,
onBlur: PropTypes.func
};

export default TextInput;
```

Figure 60. Code example of input-fields component

```
import React from 'react';
import { TouchableOpacity, View } from 'react-native';
import PropTypes from 'prop-types';

import Theme from '../_configs/Theme';

import Wrapper from '../_common-styled-components/Wrapper'; // common wrapper
// styled components from 'input-field'
import { Icon } from '../input-fields/styled-components/index';
// styled components for the 'date-picker'
import { StyledText } from '../date-picker/styled-components/index';

const LocationPicker = ({ iconPosition, icon, text, onPress }) => (
<Wrapper>
<TouchableOpacity style={{ width: '80%' }} onPress={onPress}>
<View
style={{
flexDirection: 'row',
padding: 10,
borderBottomWidth: 2,
borderBottomColor: Theme.primary_grey
}}
>
{iconPosition === 'left' && (
<Icon iconPosition={iconPosition} source={icon} />
)}
<StyledText icon={icon}>{text}</StyledText>
{iconPosition === 'right' && (
<Icon iconPosition={iconPosition} source={icon} />
)}
</View>
</TouchableOpacity>
</Wrapper>
);

LocationPicker.defaultProps = {
icon: null,
iconPosition: null
};

LocationPicker.propTypes = {
icon: PropTypes.number,
iconPosition: PropTypes.string,
text: PropTypes.string.isRequired,
onPress: PropTypes.func.isRequired
};

export default LocationPicker;
```

Figure 61. Code example of location-picker component

```
 import React from 'react';
import PropTypes from 'prop-types';

import Wrapper from '../../_common-styled-components/Wrapper';
import StyledText from './styled-components/StyledText';
import Theme from '../../_configs/Theme';

const TextElement = ({ header, color, title }) => (
<Wrapper>
<StyledText header={header} color={color}>
{title}
</StyledText>
</Wrapper>
);

TextElement.defaultProps = {
color: Theme.primary_white
};

TextElement.propTypes = {
header: PropTypes.bool.isRequired,
color: PropTypes.string,
title: PropTypes.string.isRequired
};

export default TextElement;
```

Figure 62. Code example of TextElement component