

Master Thesis



**Czech
Technical
University
in Prague**

F3

**Faculty of Electrical Engineering
Department of Computer Graphics and Interaction**

Mobile Application for Memsources Cloud Translation Platform

Vojtěch Novák

**Supervisor: ing. Ivo Malý, Ph.D.
Field of study: Software Engineering
January 2017**

Acknowledgements

I would like to thank my parents for their continuous support throughout my studies. I would also like to express my thanks to Memsource, s.r.o. for the thesis topic and the opportunity to work with cutting-edge technologies. Last but not least, I thank my supervisor ing. Ivo Malý, Ph.D. for his guidance.

Declaration

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze, 9. ledna 2017

Abstract

Memsource Cloud is an online translation platform that helps individuals as well as large translation agencies to manage their translation projects. Memsource Cloud offers tools for the entire translation workflow from document import to final review. Features as Translation memories and Term Bases ensure translation consistency.

This document describes development of an mobile application which will enable the users of Memsource Cloud to access its features through public APIs while—compared to the current web-based solution—offering the user an experience specifically tailored for a mobile application.

Keywords: mobile application development, React Native, Android, iOS, Memsource Cloud, Memsource

Supervisor: ing. Ivo Malý, Ph.D.

Abstrakt

Memsource Cloud je online platforma pro překlady, která jak jednotlivcům, tak i velkým překladatelským agenturám pomáhá spravovat jejich překladatelské projekty. Memsource Cloud nabízí sadu nástrojů pokrývajících kompletní průběh práce od importu dokumentu, až po závěrečnou revizi. Funkce jako Paměti překladů a Báze termínů zajišťují konzistenci překladu.

Tato práce popisuje vývoj mobilní aplikace, která uživatelům Memsource Cloud umožní přistupovat k funkcionalitám platformy skrze její veřejná API a oproti stávajícímu webovému řešení bude mít výhodu vyššího uživatelského komfortu díky tomu, že bude navržena přímo pro mobilní zařízení.

Klíčová slova: vývoj mobilních aplikací, React Native, Android, iOS, Memsource Cloud, Memsource

Překlad názvu: Mobilní aplikace pro překladatelskou platformu Memsource

Contents

1 Introduction	1	3.2.1 Testing with users.....	23
1.1 Motivation	2	3.2.2 Test results	25
2 Analysis	5	3.3 Application Architecture.....	25
2.1 Working with Memsources Cloud .	5	3.3.1 Redux	25
2.2 Requirements	6	3.3.2 MobX	26
2.3 Analysis of Platforms and Development Tools.....	11	3.4 Client-server Communication...	28
2.4 Xamarin	11	3.5 Domain Objects and Stores	28
2.4.1 Shared Project	12	3.5.1 Representing Users	30
2.4.2 Portable Class Library.....	12	3.5.2 Platform-specific Look and Feel.....	30
2.5 React Native	13	4 Implementation	33
2.5.1 Native Modules	14	4.1 UI with React Components	34
2.6 Ionic	15	4.2 Stores	39
2.7 Conclusions	16	4.2.1 Connecting Stores with Views	39
3 Design	19	4.3 Upload Module	40
3.1 Application Structure	19	4.3.1 Android	40
3.2 Prototyping	20	4.3.2 iOS	42
		4.4 State Persistence	43

4.4.1 Storing User Credentials	44	6 Conclusions and Future Work	59
4.5 Data Fetching	44	A Tasks for User Testing	61
4.5.1 Handling Internet Connection Outage	46	B Installation Instructions	65
4.6 Multi-stage Deployment and Testing	46	C Attachment Contents	67
4.7 Code Quality Tools	47	D Index	69
4.7.1 Flow - Static Type Checker .	48	E Bibliography	71
4.7.2 ESLint	49		
4.8 Navigation	49		
4.9 Issues	50		
4.10 Open Source Software Contributions	51		
5 Testing and Crash Reporting	53		
5.1 Unit Testing	53		
5.2 Testing With Users	54		
5.2.1 Test Conclusions	57		
5.3 Crash Reporting	57		

Figures

2.1 A translation project in Memsources Cloud	7	4.3 Platform-customized behavior of selecting jobs and providing actions over the selection.....	36
3.1 Structure of the app's screens...	20	4.2 Platform-customized behavior of list and date picker components ..	36
3.2 Mockups showing the project list screens. Second screen shows the chevron active, where user can filter displayed projects.	21	4.4 Search screen along with displayed search history entry.....	37
3.3 Job list (left) and a mockup where "Job 1" is selected and different actions are available for it (right)..	21	4.5 Project info screen.	38
3.4 Adding a new job to a project. .	22		
3.5 Different way of listing jobs within a project on iOS. The second screen displays the state of the first after clicking on the 'edit' button.	23		
3.6 Listing projects (left) and jobs in a project (right).	24		
3.7 ApiCaller class	28		
3.8 Project class	29		
3.9 ProjectStore class.....	29		
4.1 Simplified folder structure of the project.....	35		

Tables

2.1 Comparison of the considered multiplatform development tools . .	17
---	----



Chapter 1

Introduction

Memsource is a Prague-based company that develops an online platform for translation, translation management and analytics called Memsource Cloud ¹. The platform is provided as software as a service (SaaS) and uses a freemium business model. It is used by translation agencies as well as freelancers to administer their projects, the documents they need to translate and provides an editor tailored specifically for translation needs. It enables its customers to keep all important information in one place and increase their productivity. In the document, I will use the names Memsource and Memsource Cloud interchangeably and will make a note in case we need to distinguish between them.

To start using Memsource, user has to sign up and choose one of the offered plans. After completing the registration process, they can start using the project management and translation tools. The management part consists of a web-based interface where the user can administer their translation projects, jobs (translated documents), translation memories and term bases (these terms will be explained later on), users and other features. Closely related is the Memsource editor which is available both for major web browsers and as a standalone application. The editor serves as a specialized tool for performing the translation and includes features for improving the speed and quality of translation. The changes made in either of the editors are synchronized. One of the newer features is the analytics bundle which allows the user to see translation progress and performance and thus get a deeper insight into their business.

Memsource is designed to support three kinds of customer groups: individual translators, translation agencies and translation buyers and offers corresponding versions, sub-editions and services to each of those. The Translator edition is meant for individual translators or freelancers and has only a basic set of features offered free of charge. The edition

¹<https://www.memsource.com/en/features>

on. More specifically, with just the web browser, it is not possible to upload files for translation directly from email inbox, which is one of the main channels through which translation inquiries are made. Making the project and job creation from a phone's e-mail inbox as smooth as possible is an important functionality that Memsource does not have covered and that is overly complicated or impossible with a standard web browser. A mobile application, on the other hand, offers much more flexibility.

Apart from that, features of Memsource Cloud often need to provide a number of options and settings available to the user (some of which are not used frequently because they support advanced functionality) and in some cases, this makes the UI quite complex. Rendering such UI on a mobile device's browser results in inconsistencies as well as limited usability even though it is programmed responsively.

Memsource, as a leading translation platform provider wants to be able to provide its core features accessible to customers who are on the go, do not have a computer at their disposal or simply do not want to be tied to one — especially at times such as Friday evening. For the aforementioned reasons, the existing solution is not sufficient and therefore, the goal of this thesis is to develop an client application for mobile devices. This app should contain a subset of the features that are currently available and make them simple to use, keeping in mind the specifics of development for mobile devices.



Chapter 2

Analysis

In this chapter, I shortly introduce the reader to Memsourse Cloud and its features, and move onto the requirements for the mobile application. Next, I cover the market shares of Android, iOS and Windows Phone, followed by an analysis of today's major cross-platform mobile app development tools. I discuss their features and conclude by picking React Native as the library of choice.



2.1 Working with Memsourse Cloud

Each of the target groups has a partially different way of working with the Memsourse platform. Here, I will cover a usual workflow of a translation agency. Such agency typically has two main types of employees—project managers and translators (linguists). Project managers communicate with customers and receive documents that need to be translated into one or more target languages from them. For the purpose of managing several documents relating to one customer and/or topic, the manager can set up a project. Following project creation, the manager uploads the documents to Memsourse Cloud (which currently supports translation of about 50 file formats). In Memsourse, such documents are referred to as translation jobs. Jobs have a plethora of properties bound to them, such as due date, linguist who is assigned the job and other settings.

Typically, it is then the translator's job to actually translate the document, using the Memsourse web or desktop editor. The editor helps to accomplish this task by allowing preprocessing using machine translation algorithms, through term bases and translation memories and quality assurance features.

The screenshot displays the Memsources Cloud web application interface. The top navigation bar includes the Memsources Cloud logo, a home icon, and links for PROJECTS, TRANSLATION MEMORIES, and TERM BASES. User information (setup, users, sign out, VNovak) is on the right. The left sidebar contains a 'CREATE NEW' button, a search bar, and sections for 'RECENT ITEMS' (listing various test projects) and 'USEFUL LINKS' (including Job Board, Memsources Editor, Getting Started, User Manual, Support, and System Status). The main content area shows a project titled 'no notifications' with details: #807, Source Language CS, Target Languages DE and EN, Created by VNovak, Machine Translation Google, Owner VNovak, Status New, Domain Chemistry, Subdomain nutrition, Client British, and Note note1. Below this is a 'Dashboard' section with four tables: 'Jobs' (showing two items with 0% confirmed), 'Analyses' (showing one analysis), 'Quotes' (showing one quote), and 'Translation Memories' (with a 'Select' button). There are also buttons for 'Create New' for Term Bases and References.

Figure 2.1: A translation project in Memsources Cloud

3. UI components should have a platform-native look
4. UI transitions have to feel smooth
5. UI loading times not greater than 1.5 second on high-speed internet connections
6. UI follows the design guidelines of both platforms
7. app has to be intuitive to Memsources users
8. app should keep number of API request minimal, to save resources
9. the software has to be maintainable and testable
10. use external software packages only if their license is suited for commercial use

Functional requirements—the app has to:

1. support Memsources project manager and translator roles

together with all requests. It is thus possible to make such requests for multiple users from the same device. Note that the API is available to only some of the Memsources editions.

2. Content requirements

- 2.1. The app will load all its content through a public HTTP API which is provided by Memsources. In cases where a new API is needed, it will be developed.
- 2.2. The app has to keep the number of API requests as low as possible so that the phone's and the servers' resources are not overused.
- 2.3. The first screen should render within 4 seconds after starting the app.

It follows that we assume the user will have internet connection available on their device at all times when the app is used to create new content and fetch the up-to-date data.

Compared to the web-based service, the app will only support selected features, to keep its UI simple and easy to work with. At the same time, it should follow the patterns users know from the web version to avoid confusion. More specifically:

3. Functionality requirements

3.1. Functionality for PM users

- 3.1.1. The app has to support logging in for Memsources users whose role is project manager (PM).
- 3.1.2. Project-related requirements
 - 3.1.2.1. App has to support listing projects based on their status (my projects, all, overdue, in progress). The list of projects should contain relevant information (name, customer, due date) for each project.
 - 3.1.2.2. App has to offer project search using the project name, when the project was created (last 24 hours, last 3 days, last 7 days, last 30 days), the client, the project owner and due date.
 - 3.1.2.3. PM has to be able to create a new project where they have to be able to enter a predefined template from which project can be created, project name, client, domain and subdomain, business unit, source and target languages, due date and note.
 - 3.1.2.4. App will support deleting projects.
 - 3.1.2.5. App will support editing projects. The same project properties which are supported by the app when creating a new project will be supported for editing, with the addition of project owner.
 - 3.1.2.6. The app will allow the PM see project details (name, user who created the project, date when created, status, due date, source and target languages and project owner).
 - 3.1.2.7. App will support adding existing Translation memory and existing Term base to a project.
- 3.1.3. Job-related requirements

■ 2.3 Analysis of Platforms and Development Tools

Today's market of mobile devices is largely divided between two platforms—iOS and Android. With 80.7% for Android and 17.7% for iOS, these two alone made up more than 98% of worldwide sales in 4Q15, according to Gartner [13]. Windows Phone comes third with 1.1% of sales. Interestingly, in its March 2016 report, Kantar Worldpanel shows the sales of vary greatly among different states [15]. For example, while iOS has a sales share of 56% in Japan, it only has 17.8% in Germany. As of Q1 2016, Android is growing in Europe, at the expense of iOS and Windows Phone, whose sales have been dropping in the UK and France where Windows Phone was historically relatively successful. The data from 3Q2016 shows growing Android sales: 86.8%, 12.5% and 0.3% for Android, iOS and Windows Phone, respectively according to IDC [14] and a report from Gartner [12] shows very similar numbers.

Since we are looking for a solution for creating apps for more than just one platform, I continue with describing the state of the art of multiplatform development tools.

■ 2.4 Xamarin

Xamarin is a framework for developing native apps for iOS, Android and Windows Phone. The company was founded in 2011 and was acquired by Microsoft in 2016. All of Xamarin is now open-sourced on GitHub ¹. Xamarin divides into four main parts: Xamarin.iOS, Xamarin.Android, Xamarin.Windows and Xamarin.Forms. The first three offer access to native APIs for the particular platforms, where Xamarin stresses that “Anything you can do in Objective-C, Swift, or Java you can do in C# with Xamarin” [28] When using these, the developed solution consists of one project per supported platform (which contains the platform-specific code — mainly the UI) plus single project whose code is shared and contains the business logic. Reportedly, this approach can result into about 75% code reuse [27]. Xamarin.Forms is an attempt to bring code reuse even higher, usually more than 90% [27] by sharing also the UI code.

As explained, UI in Xamarin can be either defined specifically for each platform within the platform's application project or cross-platform using Xamarin.Forms [26].

Taking the first way is recommended for apps with interactions that require native behavior, apps which use many platform-specific APIs or cases when custom-tailored UI has higher priority than code sharing. With this approach, each app will have its own UI

¹<https://github.com/xamarin>

Dependency Injection. That way, the actual implementation is coded in the platform projects against an interface defined within the Portable Class Library.

■ 2.5 React Native

React Native (RN) is a counterpart of the popular web development library React and is also developed by Facebook which uses in several production apps and “will continue to invest in it” ². It was first released in 2015, which makes it the youngest among the covered solutions. React is popularized under the slogan “Learn once, write anywhere.”, as opposed to e.g. Java whose goal is that one codebase runs anywhere, this means that once a developer learns React, she can use her skills to write apps for multiple platforms (web, Android, iOS, etc.) using just React, but not necessarily with a single codebase.

React originally started as a tool for describing user interfaces for the web, and rapidly became popular within the web development community. However, it was recognized that React’s usage was not limited only to web.

React describes the user interface through reusable components which tell what the UI is supposed to look like. It is then the matter of transforming the description into a user-facing UI. On the web, this is the task of React-DOM [9] which uses the Virtual DOM tree as a layer of abstraction between the developer’s code and what is rendered in the browser. When programming mobile apps, this abstraction is handled by React Native [9]. At this point, it is important to state that the UI rendered with React Native is not running in a WebView but is built from the native UI elements of the platform in question (i.e. View on Android and UIView on iOS), which makes it different from the longer-established hybrid development environments which also use JavaScript.

RN application code is written in JavaScript which runs in JavaScriptCore engine on the device [20]. RN features what is called a *bridge* [29]. The bridge is in turn responsible for bridging the calls onto the native platform APIs and back. This way, user can access any native functionality and get information back in a callback or promise payload.

When developing with RN, the developer creates or makes use of ready-made UI components and uses them to compose the application UI. The components are written in a combination of JavaScript and XML tags, called JSX. Optionally, Flow ³, a static type checker for JavaScript can be used. Also supported although not so frequently used are languages that transpile to JavaScript, such as TypeScript ⁴. From developer’s point of

²<https://facebook.github.io/react-native>

³<http://flowtype.org/>

⁴<https://www.typescriptlang.org/>

2.6 Ionic

In version 2.RC4 as of 1/2017, Ionic ⁵ is another successful framework for developing cross-platform mobile apps which was initially released in 2013. Ionic is a hybrid framework, meaning an app created with it runs in a WebView, same as a website would—with the important difference that it can also use the native device APIs. It supports iOS, Android and Windows Phone.

Just like the aforementioned frameworks, Ionic is open source and offers a set of mobile-optimized components written in HTML, CSS and JavaScript. Ionic 2 integrates with Angular 2, a framework for web development from Google. Ionic has put a lot of work into providing components that are styled according to each supported platform, thus saving the developer's time by not having to spend valuable time by styling the UI. Compared to React Native and Xamarin, Ionic gives less flexibility in customizing the app for different platform. With Xamarin and RN, developer can make the app look and behave completely different (at the expense of writing more code) while this is limited on Ionic. Depending on the particular app context, this can be both a downside or a benefit.

Ionic 2 developer can optionally choose to develop in TypeScript, which is a language that compiles to plain JavaScript. As its name reveals, the most important TypeScript's feature is addition of types to JavaScript. This can reveal errors before they happen, and gives extra information to both the developer and IDE (Integrated Development Environment) which therefore can offer better code completion. TypeScript is basically a competitor of Flow.

Ionic runs inside Apache Cordova [19], a mobile application development framework which provides access to native platform features such as camera, sensors, filesystem or contacts [7]. Access to arbitrary features can be allowed through plugins, which are composed of a single JavaScript interface used on all platforms, and platform-specific code which is called from JavaScript. In this respect, Cordova Plugins are similar to native modules in RN.

Ionic does more than only mobile app development. It offers features like ionic lab, which allows to run iOS and Android apps one next to the other in browser as well as in a GUI application for Windows and Mac. There is also the Ionic Market, which contains lots of starter templates and themes. Ionic's View app allows developer to easily share apps with customers and testers. Ionic Creator is a prototyping tool where developer can drag and drop components to create a simple app and even export it as an Ionic app.

⁵<http://ionicframework.com>

denote neutral and negative rating.

Platform	Xamarin	Xamarin Forms	Ionic 2	React Native
development speed	-	-	+	+
performance	+	+	-	0
maturity	+	0	+	-
platform-specific behavior	+	-	-	+

Table 2.1: Comparison of the considered multiplatform development tools



Chapter 3

Design

After analyzing the task and collecting the requirements, this chapter will go through the design of initial paper prototypes as well as higher fidelity software prototypes. I will also describe the architecture of the mobile application which is crucial for successful development, future extensions and maintenance. I will talk about different components that the application is composed of and how they cooperate to achieve desired functionality. This chapter contains several UML diagrams, all of which are simplified.



3.1 Application Structure

The application requirements give a thorough description of the features the app will offer. Upon the requirements I have designed the application navigation structure that I'd like to follow. Figure 3.1 shows the hierarchy of the app's screens with somewhat simplified screen transitions. The root of the navigation is the Project List screen which will show lists of projects. For each project, the hierarchy then goes deeper to allow user to view further project information and to work with jobs, translation memories and term bases. Another application entry point is the screen for adding a job from an external application (e.g. from Mail or from a filesystem browser application).

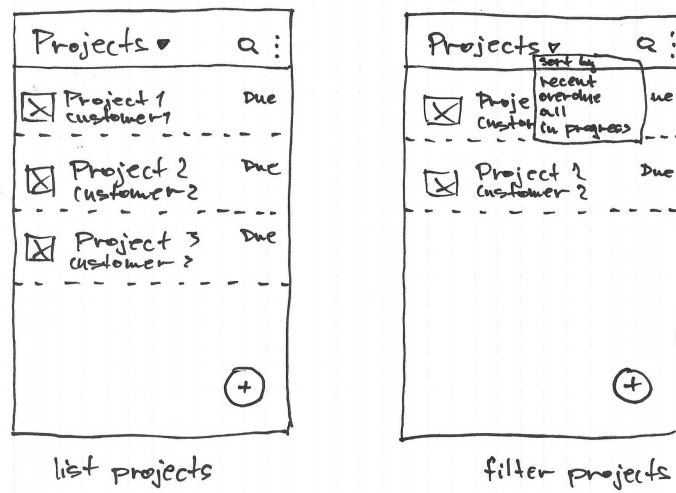


Figure 3.2: Mockups showing the project list screens. Second screen shows the chevron active, where user can filter displayed projects.

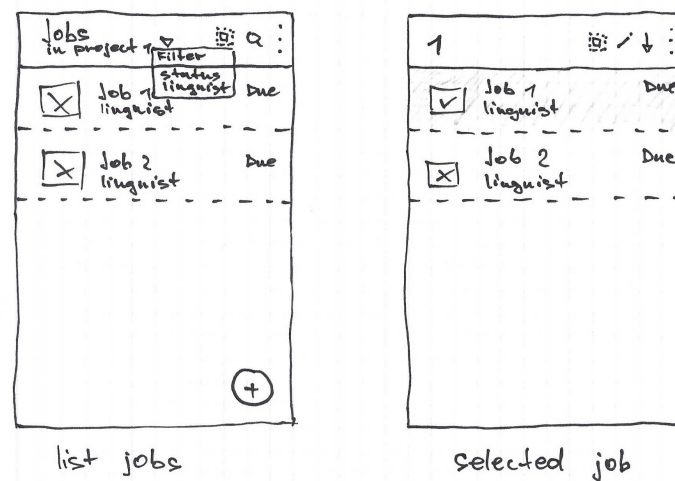


Figure 3.3: Job list (left) and a mockup where “Job 1” is selected and different actions are available for it (right).

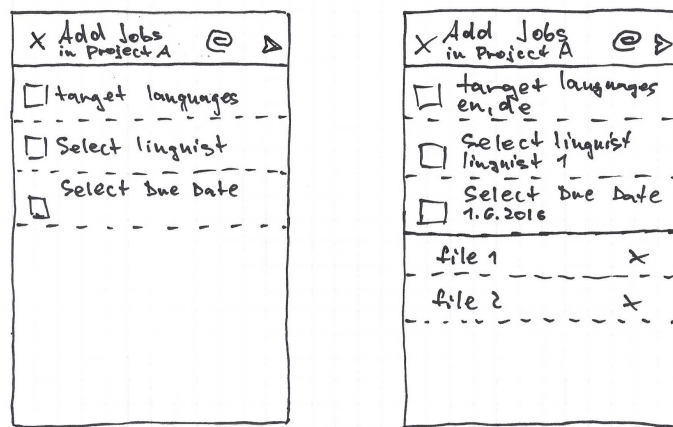


Figure 3.4: Adding a new job to a project.

Figure 3.2 shows design of the project listing. There are controls for creating a project, searching, filtering (using the chevron) and getting the important information about user's projects. In figure 3.6 you can see how filtering the projects changed in the final prototype.

Figure 3.3 shows a mockup of an opened project with translation jobs listed (first screen). The second screen shows a job being selected and the consequent changes in the navbar: the user can choose to edit, select all or download job.

Figure 3.4 contains the screens for adding a new job to a project from within the application. The figure shows two states of the same screen, first screen is waiting for user to enter needed information, the second displays the state when the information is entered, along with two files chosen for upload. In the prototype, the layout of these screens was preserved, except for division into two tabs: one for uploaded files and second for import settings.

For iOS, which has different interaction patterns compared to Android, I created separate mockups of what the UI could look like. The figure (3.5) shows the mockup of job listing and handling.

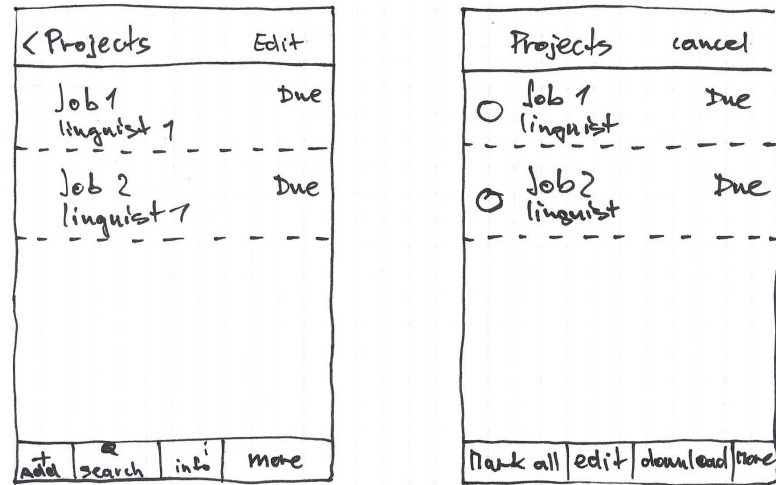


Figure 3.5: Different way of listing jobs within a project on iOS. The second screen displays the state of the first after clicking on the 'edit' button.

3.2.1 Testing with users

From the mockups I created a software prototype for Android. The prototype was created using Axure RP 8.0 ¹ which is a software for creating different kinds of UI prototypes. There are widget libraries available, which contain ready-to-use Android and iOS UI elements. Figure 3.6 show selected prototype screenshots.

To verify the created prototype, I conducted two informal tests with users. Axure provides Axure Share service to share projects, with an Android app ² available on the Google Play Store, but this app proved not to be suitable for testing because it does not scale the UI well. Instead, I took advantage of Axure's ability to export created project as HTML which can be viewed directly on the device, for which I used the Kiosk Browser ³ app which allows to display content full-screen.

To help keep the users relaxed, I explained the purpose of the application we were about to test and that we were testing only an initial prototype to catch its flaws, and not testing their abilities of working with Android.

The users were given a list of tasks corresponding to a possible walkthrough of the app. The text is following:

¹<http://www.axure.com/>

²<https://play.google.com/store/apps/details?id=com.axure.axshare>

³<https://play.google.com/store/apps/details?id=it.automated.android.browser.kiosk>

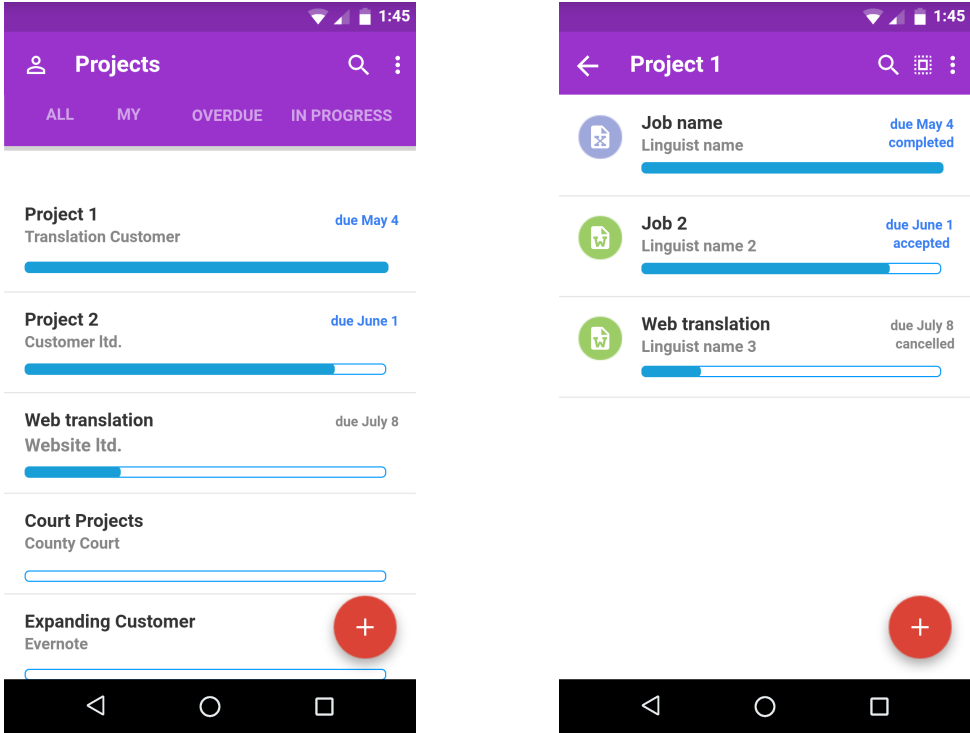


Figure 3.6: Listing projects (left) and jobs in a project (right)

You are a Memsources Cloud user and your role is project manager. Log in using the username “user” and password “pass”. View translation jobs in Project 1 and download job whose name is “Job name” onto your device. Then create a new job from the “document.docx” which is available on Google drive. For the job, select English and German as target languages, due date as 2nd January 2016, 11:00 am and enter linguist name. You need the file be imported with comments and hidden text. Then, create a new project as new project name, enter “test project”, select “client 3” as the client and select arbitrary parameters for the other options.

The informal testing was conducted in the company offices with two members of Memsources support team who both were owner of a mobile phone running Android. Test was conducted using LG Nexus 5 with the prototype running in the aforementioned Kiosk Browser. The downside of this setup was that the back button of the prototype was not available and in one instance (before adding a new project), this required intervention into the test process and consequent finding that a back button should be added to the navigation bar. Also, the generated html prototype seemed to have issues with entering text into textfields—they were accessible only after a long press instead of a simple tap. I have not found the root cause of this and needed to inform users of this issue before starting the test.

3.2.2 Test results

The test was completed by both users. However, during the test, two mistakes present in the UI were reported (problem with import options and adding new project). Both users complained about unintuitive icons, which was especially true of the white cloud icons in the upper right corner of the screen. After filling in all information for a new project, one user asked if that was the icon they were supposed to tap.

For the second iteration of testing, I used an improved prototype which fixed the flaws we found in the first iteration. Also, I stopped using the kiosk app and opted for the Axure Share Android app after fixing the scaling issue. The second test was successful and users reported they were satisfied. One tended to play with the prototype beyond the extend of what it was made for and complained some buttons were not functional. I do not consider that a problem, since this was mainly a horizontal prototype with only a particular interaction path implemented.

3.3 Application Architecture

In this section I describe the process of designing the inner workings of the application with respect to the fact that React Native was chosen as the library for implementing the application. This first involves finding a solution for app's state management which may fundamentally influence the architecture.

React Native allows to create user interfaces from the fundamental building blocks of the platform it runs on (View on Android and UIView on iOS) and it also provides means for communicating between the JavaScript and native layers. What remains to be chosen is a library that will be used for storing the application state - ie. all of the data fetched from the Memsources API and displayed in the app such as project data, app user information and other. There are several libraries that help solve the problem. The most popular at the time of writing is called Redux [6].

3.3.1 Redux

Redux is built around several core principles: The entire app state is stored in a single object called the store [1]. The state can be modified by actions which are plain JavaScript objects describing the name and payload of the action [1]. Actions are dispatched to

reducers. Reducer is a pure function that takes two arguments: previous state and action, and returns the new state.

Pure function is a function that always returns the same result given the same parameters and produces no side effects. It is important that the reducer calculates the next state and returns it, without modifying the previous one. As the application grows, the root reducer function is split into more reducer functions responsible for reducing different subparts of the state. Designing the shape of the state object is therefore key part of using Redux in any application.

Redux requires that the state is not modified in the reducer, which works well with the use of persistent immutable data structures [5] (PIDS). The most popular implementation of PIDS in JavaScript is Immutable.js. Immutable.js offers data structures that present an mutable interface (such as `add()` method for an array) but instead of mutating the original object, a new object is returned, so using the reference equality operator will return `false`.

When changing an object in PIDS, the new object is essentially a copy of the previous one but as much content as possible is recycled from the previous object. Other objects that pointed on the old object need to be copied as well, but objects that do not need to be copied stay unchanged. Implementations of PIDs use Trie data structure to represent common data structures such as arrays using a tree. This is shown in the following figure [todo](#).

Lastly, Redux provides the `redux-react` package which allows the React components to "connect" themselves to relevant parts of the state tree and receive the data from them through props.

3.3.2 MobX

MobX is another state management library with growing popularity that has React bindings. MobX uses observable data structures that, as opposed to Redux, are mutable. Its key philosophy is that "Anything that can be derived from the application state, should be derived." [25]. Compared to Redux, MobX requires writing less code and while its internals are much more complex because of the change tracking, it offers synchronized state and views out of the box and its API surface is small.

With MobX, the first step is to declare the state and make the relevant parts of it observable. This is usually done in ES6 classes. The next part is observing the changes in observable data. This is done through tracked functions. MobX tracks the observable

data used during the execution of tracked functions and invokes them upon change in that data.

One of the most important tracked function is `autorun`. If an observable data used in `autorun` changes, `autorun` is re-run. This is the function that is responsible for keeping the React views in sync with the observable state. MobX provides the `mobx-react` package which includes an `@observer` decorator which can be used for React component to make them react to changes in observable data, and the decorator makes use of `autorun` internally. MobX also provides many other reactive utility functions for more fine-grained reactions. An example of how MobX can be used with React is shown in figure 3.1 where a simple React component shows the number of seconds since the code was executed.

Listing 3.1 Using MobX with React

```

1 let appState = observable({
2   timer: 0
3 })
4
5 setInterval(() => {
6   appState.timer += 1
7 }, 1000)
8
9 let TimerView = observer((props)=>
10   return <span>Seconds passed: {props.appState.timer}</span>
11 )
12
13 React.render(<TimerView appState={appState} />, document.body)
```

After developing a small part of the application with Redux and also MobX, I chose to use MobX for storing the app state. The reasons for choosing MobX were its simplicity and proximity to object-oriented software design which I'm more experienced with. Using Redux requires writing boilerplate code to describe the actions and writing the reducers. Also, it is not always possible to dispatch actions that are plain objects - communication with Memsorce API, for example, would require dispatching functions that would change the state after receiving data from the API. Working with hierarchies of objects also requires normalizing the application state, similar to how it is done in databases, with ids used as keys for retrieving a referenced object from other parts of the state tree. Apart from Redux itself, this then involves understanding several other libraries such as `normalizr`, `redux-thunk` (or other), `reselect` (optional) and `Immutable.js` (optional but much recommended).

With MobX, the app will consist of the views which are handled by React, domain objects, which are JavaScript objects and some of their properties will be marked as observable or computed (ie. derived from other observables). The domain objects will be stored in domain stores. Stores are objects that instantiate new domain objects, delete the existing ones and provide other necessary functions with regard to domain objects. The last piece of the puzzle is handling communication with the Memsorce API.

Communication with the Memsources API will be facilitated through an ApiCaller object whose simplified class diagram is shown in figure 3.7. The documentation of Memsources APIs is publicly available at the "Memsources wiki".

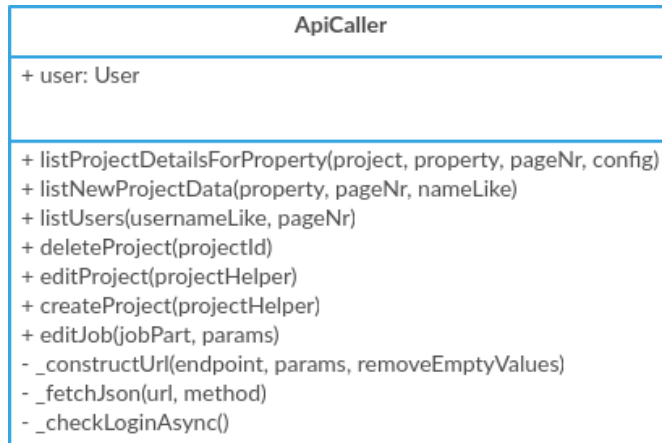


Figure 3.7: ApiCaller class

ApiCaller will expose methods that offer CRUD operations over various resources such as project or jobs. All of its methods will return a Promise object which returns the fetched data upon resolving. ApiCaller will be used mostly from the stores and other functions needing API access. The advantage of having an object that will hide the fetching logic inside is easy testability and maintenance - if fetching is done in one place, it is easy to mock and change the implementation if needed. ApiCaller also contains a reference the currently active user, so that it fetches data in the user's name.

There are several core domain objects the application needs to work with. These represent the corresponding entities in Memsources Cloud and will be fetched through the API. These objects are mostly simple data holders, with little logic included in them.

Domain objects will be stored in stores. Every type of object will have its own store that takes care of saving, editing and deleting the objects and may contain further logic needed to fulfill these tasks. Figure 3.8 shows the Project class and figure 3.9 shows the store for projects.

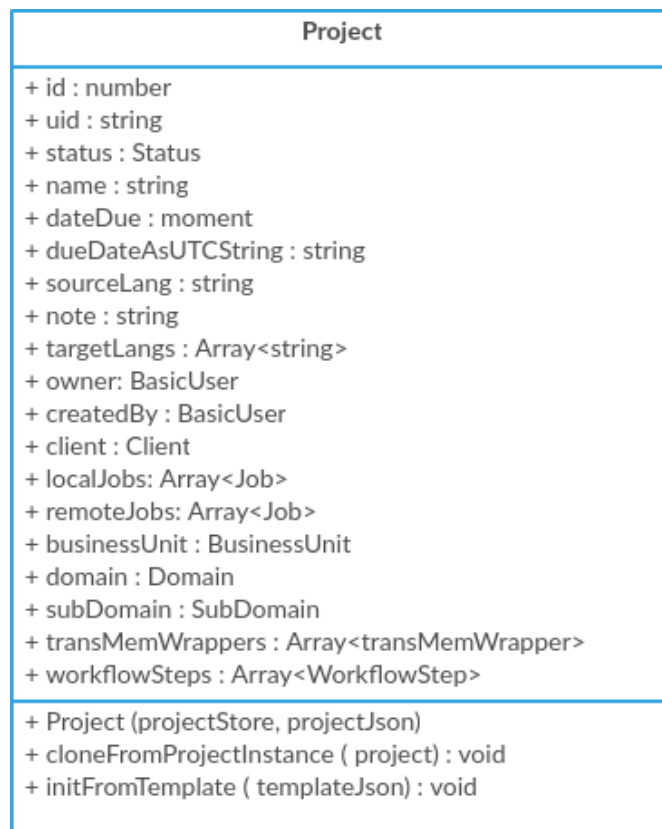


Figure 3.8: Project class

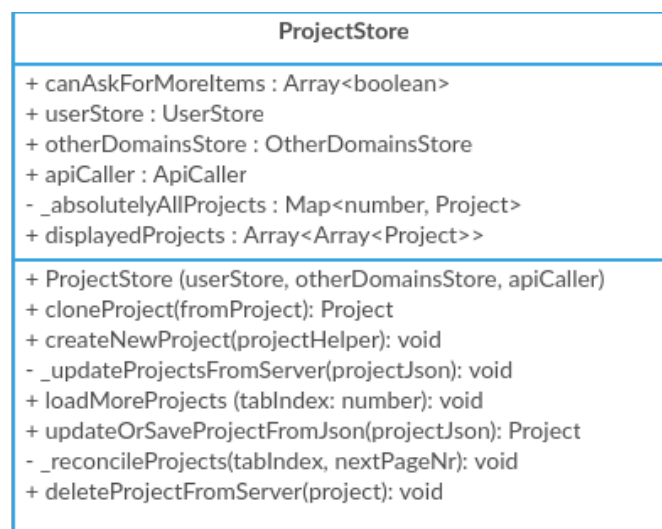


Figure 3.9: ProjectStore class

For tasks such as creating projects or jobs, dedicated objects will be created, with their life span being limited by the sole task they need to fulfill.

powerful concept: typically a developer would use this approach for components that will serve the same purpose but need to look differently on each platform. Both files then have the same interface which abstracts away the inner differences. Such approach can be used in a number of components such as buttons, pickers or even a non-component code. I will take advantage of these features to improve user experience and to follow the design guidelines.



Chapter 4

Implementation

This chapter describes the implementation of the features that were identified in the analysis. I will focus on the interesting parts of the implementation.

From a higher-level perspective, there are several subparts of the project that need to cooperate for the client app to work well. Firstly, there are the domain objects and stores implemented with the help of MobX. Secondly, there is the view layer created with React Native. The third piece is handling API communication and lastly, there are the native modules for handling job upload (ie. uploading files to Memsources and creating jobs from them) which are the only custom code of the application that is not written in JavaScript. Native code is also needed for the case when the Memsources application is opened from eg. the Mail application for job import. This chapter explains in depth the solutions to these tasks and issues encountered along the way.

In the following text, I will be using the terms JavaScript (JS) layer, native layer and asynchronous bridge. When talking about the JavaScript layer, I refer to the JavaScript code which runs in JavaScriptCore engine. By native layer I mean the code that is being executed in the Android or iOS runtime environments. To explain the importance of these terms, let us quickly take a look at an example of how these affects us.

When React Native (RN) communicates between the JavaScript and native layers, it uses batched messages that it sends through the asynchronous bridge. This means that if you want (or are forced to, in some cases) to access some value synchronously from one layer or another, it has to be available in that particular layer. In some cases this means a value has to be duplicated on both sides of the bridge and has to be synchronized using a call through the bridge. However, when a value exists on both sides of the bridge, we need some mechanism that will be able to determine which of the two values is the

most-up-to-date one.

As an example, let us consider the `TextInput` component (a text field) on iOS. Because of the design of the platform, the value entered in the text field has to live in `UIKit`. We would, however, like to be able to control and read the text input's value from JavaScript synchronously. Imagine a text field into which we start typing "ABC" as input. At time 1, the native UI thread sends the first letter "A" to the JS thread. The JS thread, however, does not pick up the value until it is picked up from the event loop queue. At time 2, user enters "B" and the UI thread sends the value "AB" to the JS queue. At the same time, the text field value is set to "A" in response to the update sent from the JS thread. At time 3, user enters "C" and so the UI thread sends "AC" to the JS queue. At the same time the value "AB" that JS just received is sent through the bridge to the native layer. In the end, the JS thread receives "AC" and sends it again through the bridge. Both the native and JavaScript layers now contain the same value - "AC" but the character "B" has been dropped! The solution implemented in RN involves a counter of input events of the text field which is sent through the bridge and the value of the text field is not changed unless the counter number received from JS is higher than the one stored in the native layer.

This gives us understanding of the JavaScript layer, native layer and the asynchronous bridge. This also explains some unexpected effects that a developer may meet when working with native UI controls.

Figure 4.1 shows simplified schema of the project structure. Since React Native makes no assumptions about the rest of the development stack, developer has the freedom to structure the project as they find fit. In my case, I first created dedicated folders for domain objects (models) and stores. All components live in the `components` directory which is further divided based on where in the application the components are used or what purpose they serve. The `api` folder contains the objects related to connectivity and communication with Memsorce API and the remaining folders provide supporting utilities such as global styles. Native code and modules are placed higher in the directory structure so they are not visible in the figure.

4.1 UI with React Components

User interfaces made with React consist of components, which are independent and reusable pieces of code. The complete application UI consists of a tree of components. In the case of this project, the root component is defined in `app.js` in the `js` directory as can be seen in figure 4.1.

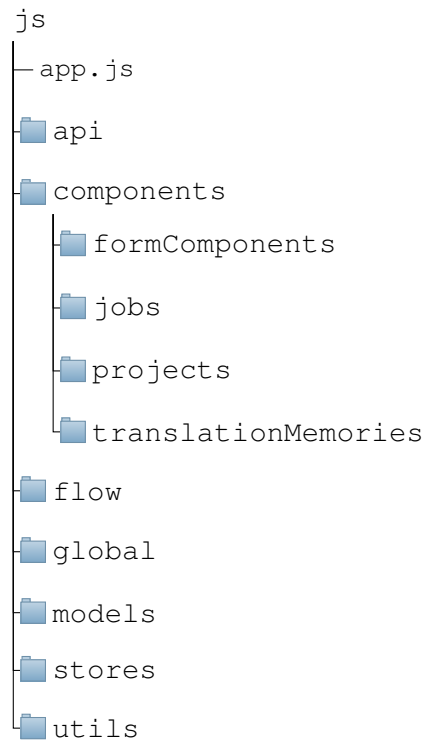


Figure 4.1: Simplified folder structure of the project.

The main way of modifying component behavior is composition - by wrapping a component and adding some functionality, we create a new one. Since the app needs to support lots of CRUD operations we need components for choosing date, choosing one or more items from a small as well as large lists and more. One of the first issues I have encountered is that finding form components which would look according to what is customary on both platforms is hard. While there are UI toolkits for React Native such as Shoutem UI Toolkit or NativeBase and community-developed components, none of them offered quite the functionality that I needed at the time when I evaluated them.

I therefore created wrappers around the basic form components provided by React Native and gave them default styling which is overridable (for example a label or an icon can be added) and mode (for example modal or inline pickers on iOS). All of these options are available under a unified interface of the component, which allows it to be easily used throughout the app. The figure 4.2 shows an example of different look of ListPicker and DatePicker components on Android and iOS. Note that the pickers on iOS can be displayed both inline and in a modal at the bottom of the screen. Android ListPicker on the other hand, can be displayed in a dialog or as a dropdown.

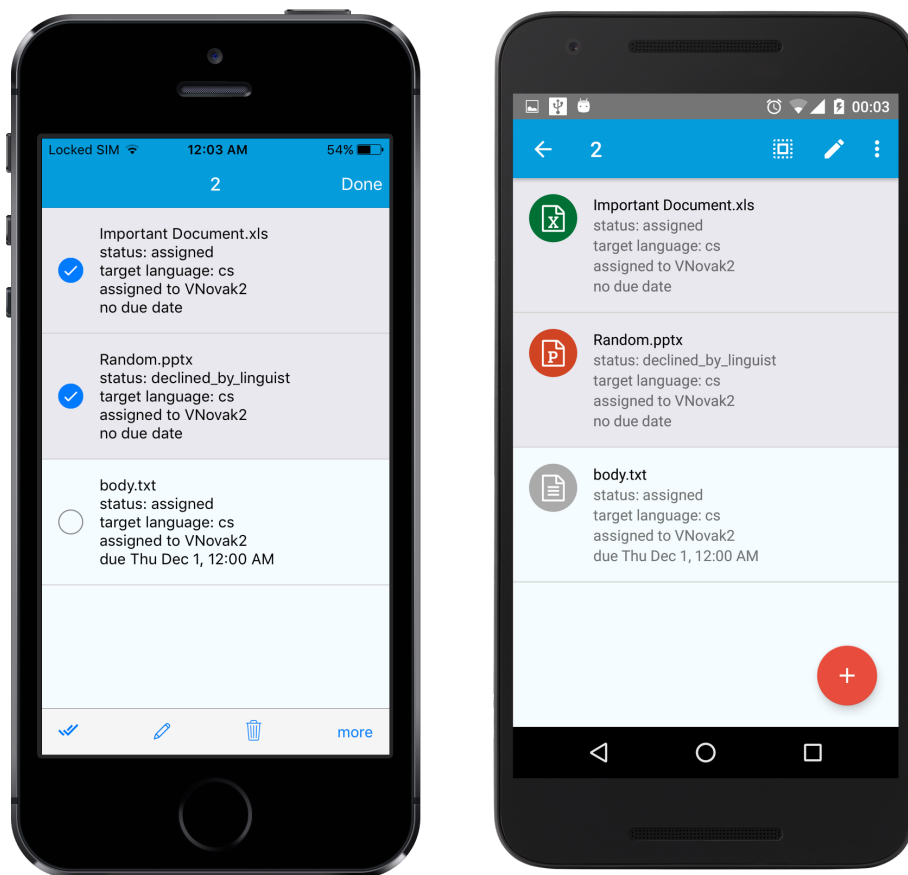


Figure 4.3: Platform-customized behavior of selecting jobs and providing actions over the selection.

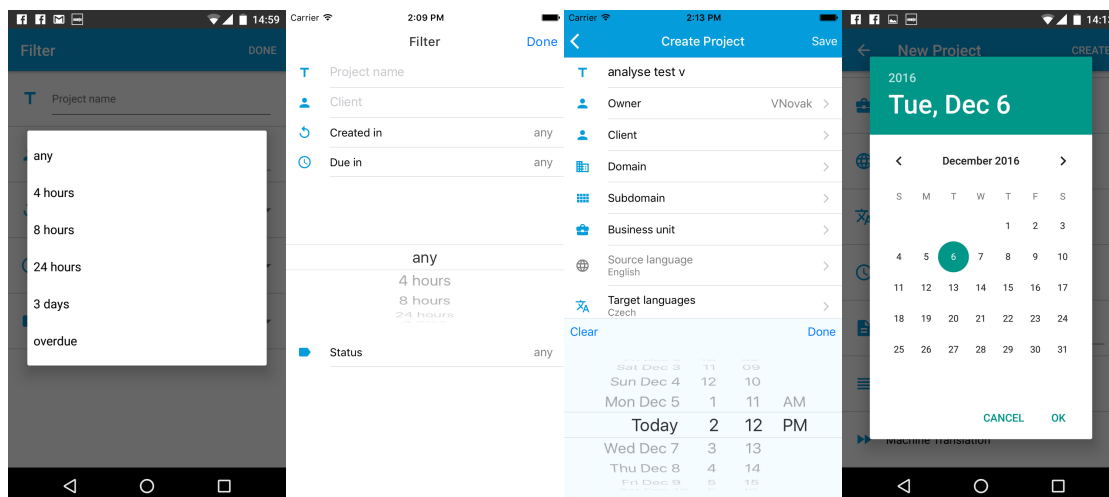


Figure 4.2: Platform-customized behavior of list and date picker components

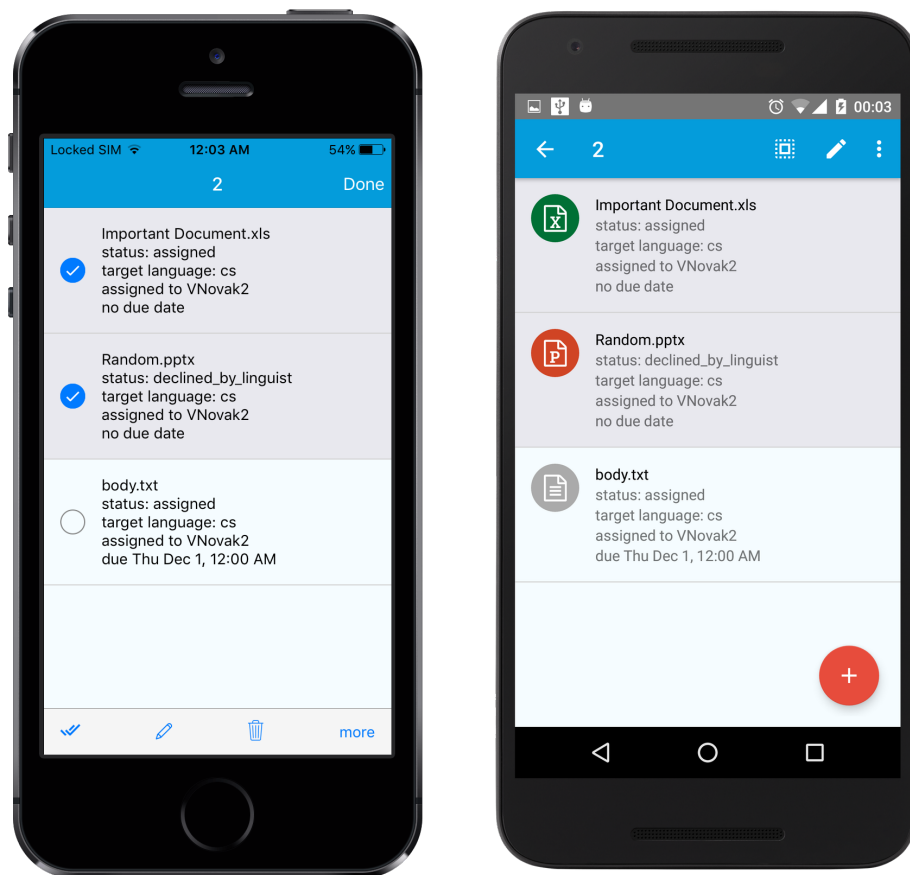
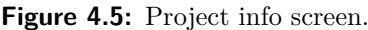


Figure 4.4: Search screen along with displayed search history entry.

When using such components in code, one only has to provide them with the information about the icon, the values they need to show, the mode and how they should respond to user action. The logic of how the component should be displayed and styled is hidden inside of it, with the styling being overridable. This way we can construct reusable components with platform-specific behavior. This, in my opinion is a very powerful concept, since as the community evolves it will create components that serve the same purpose and have the same interface, but may look even completely different per platform. The code responsible for the different behavior, however, will be hidden inside those libraries and we will just plug the components into our projects. This will allow to focus more on the business logic of the app while enabling to develop it so that looks as is customary for the platform it runs on.

The job list screen shown in figure 4.3 is a good example of where Android and iOS usage patterns deviate and how I captured this difference in code to provide different user experience for each platform. The figure displays the job list screen in selection mode. To get into this mode on Android user would have to tap a file icon on the left, or long-press an item. On iOS, the selection mode is enabled by tapping "Edit" button in the right of the navbar. To get out of the mode on Android, user can tap the arrow



Figures 4.4 and 4.5 show other selected screenshots with only small platform-specific differences.

4.2 Stores

As explained in section 3.5, the information about projects, jobs, and other entities is stored in domain objects, which in turn are kept in stores. It is important that there is always only one instance of a particular domain object in memory, and only one instance of a store.

As an example of how a store is implemented, let's consider the project store whose class diagram is in figure 3.9 on page 29. In its constructor it accepts (among other) the user store - that way we can reactively clean and refresh the project list in case the active user changes. This is done through the aforementioned `autorun` function from MobX. It also receives `ApiCaller` instance which is used for communication with Memsources API. Projects are stored in a Map data structure, which is typically backed by a hash table that offers constant access time, or by other mechanism that provides sublinear access time [17]. The keys of the Map are projects ids and the values are instances of Project class. That way we can access the stored project instances quickly, which is especially convenient because the projects are displayed in multiple tabs that correspond to filters in Memsources Cloud (all, in progress, overdue, my). One project may be displayed in any number of tabs and even when a project is displayed in all four of them, internally this refers to the same project instance. Also, when the app receives a JSON (JavaScript Object Notation - lightweight, human-readable data-interchange format) containing projects, it can quickly see whether or not a project is already stored.

4.2.1 Connecting Stores with Views

Clearly, stores need to be made available to React, so that components can visualize the content of the domain objects. For this purpose, the `mobx-react` package offers the `Provider` component and `@inject` decorator, thanks to which arbitrary objects can be passed to React components as props. Internally this uses React's context to function.

As I explained earlier, React app is composed of a tree of components where props are passed from top to bottom. You can either pass the stores as props explicitly through the entire tree which can get tedious, or use `Provider`, and grant store access for them by using `inject`. This makes it simple and transparent to "connect" components with relevant parts of the state. Listing 4.1 shows a simple example of how color prop can be injected. We can still pass the prop explicitly from the parent component, in which case the explicit prop takes precedence. This can be taken advantage of in testing.

Listing 4.1 Using MobX Provider and inject

```
1 class Main extends React.Component {
```


In both cases, the app receives a uri which points to the file. Note that the file doesn't necessarily need to be on the device, it may as well come from a cloud storage such as OneDrive or Google Drive. The user then sets up various options for the import and taps the "send" button. Upon pressing the button, the necessary information is passed to the upload module and upload is started. The information includes the following fields.

- token
- upload url
- upload id
- project id
- user id
- whether the job should be pre-translated
- file information: uri, file name, id of temporary job

The module runs a background service whose responsibilities are issuing a notification when the upload starts, when the job is imported or if there was an error. The notification also shows different stages of the job creation - uploading of the file, file import and pre-translation, if selected.

To upload a file, we first need to check whether it is present on the device. If not, file is downloaded. For working with the uri object, the service uses the Storage Access Framework API introduced in Android 4.4. The next step is uploading the file to Memsources and creating a job from it. This is done through the asynchronous `Create New Job` API call which handles the upload and puts the file into a queue where it waits for import. A server backend service dequeues the file and creates a job from it. Because dequeuing and job creation can potentially be long-running operations, the API call returns an ID of the enqueued object immediately after the upload is finished. The Android service then repeatedly polls the `Get Asynchronous Request` API to check if the job was already imported.

Once done, it issues a final notification or, if the user checked the pre-translation checkbox at job upload screen, continues with pre-translation based on project settings. This is another long-running operation and the native module keeps polling the `Asynchronous API` to check the pre-translation until it is confirmed. At that point it issues the final notification.

The service repeats its download and upload requests if there was an error and is also made to run in the background so that its actions are not disrupted in case the

to use another Memsources API, the File API which allows to upload a file in the request body and returns a file ID which can be used in other API calls. The problem with this approach is that we need to make two API calls to make the job import happen: first call to upload the file and second to the `Create Job` API. This poses an issue since iOS may decide to not perform our background request.

iOS uses several pieces of information to decide whether or not a request will be carried out. The decision involves eg. how often the app is used by the phone's owner or what the battery level is. The exact algorithm is not publicly available. It may therefore happen that the first request for file upload is honored but the second request for actually creating the job is ignored. Also, if the user terminates (swipes away) the application during file upload, the upload will finish but the call to the `Create Job` API will not be carried out since the system cancels any pending tasks [2]. Lastly, there is another situation when the call may not happen - if the application is awoken too often in response to its background requests, the interval in which it is be awoken will prolong, and it may thus happen that the request for job import is carried out long after the job was uploaded.

The best possible solution to this is implementing another API which would accept the file in the request body and the numerous parameters sent to the `Create Job` API would be sent as a JSON string in a special request header. For the time being, I have implemented the described workaround and a better solution on the server side will be implemented later.

Similarly to Android, as the upload starts, all of its information except the token is serialized and stored to UserPreferences entry and is updated whenever the status of the upload or import changes. This way the information is not lost when the app is killed during or after the upload or import and the information about whether an upload was successful can be processed the next time the app starts or resumes: if the file was uploaded and import was requested on the server, the job import is considered successful and the temporary job entry is removed from the job list. If the upload did not finish or if the import wasn't requested, the item is kept in the list and flagged it as a failed import. The iOS native module is written in Swift 3.

4.4 State Persistence

One of the implemented features is having parts of the app's data stored on the device so that it is available right after the app's startup. This involves all project-related information, so that when the app starts, the user sees their projects immediately, along with a loading indicator which denotes that the projects are being refreshed. MobX

The API uses standard HTTP response codes to denote operation result and provides error description when a request is incorrect.

There are some common patterns related to data fetching arising throughout the app. In many places we need to display some data, be able to reload it (using the well known pull down gesture), and be able to load more of the content and append it to the existing data (informally known as infinite loading).

Many of the used APIs use paging, ie. they deliver results in batches of 50 items per request (or less if more aren't available). The app uses this fact to find out if more items can be fetched since the number of the next page to be fetched can be calculated as $next = number\ of\ received\ items / 50$. If a response contains less than 50 items we know there are no more items to be fetched. However, we need to keep in mind that items can be both added and removed to the lists, for example when projects are added or deleted. That would give us a page number which is not an integer. In that case we perform a request for a page whose number is the closest lower integer. This may give us items that are already stored in the list, in which case we remove items at indices from $next \cdot 50$ to the end. That way we display the correct data and do not need to make any additional requests.

Also, in some cases we want to limit the number of pages that we fetch so that we do not allow the app to keep too many objects in memory which could cause undesired behavior.

In some places where data is fetched we want to give the user a possibility to refresh the loaded list (such as in project or job lists) while in other we only offer listing without refreshing. This means we need to control up to two loading indicators that will denote refreshing (that would be the pull down indicator) or loading more content (loading indicator at the bottom of a list). We also need a means for blocking a request if it is already in progress or if it is forbidden (because of reaching the limit of how many items can be fetched or because no more items are available).

Blocking a request if it already in progress is needed for cases when we eg. scroll down a `ListView` which has the infinite loading implemented. Infinite loading is implemented using `ListView`'s `onEndReached` function. This function is invoked when a user scrolls down the `ListView` and arrives at some pre-defined distance from the end of its content. Invocation of this function triggers fetching more items. In case of a poor network connection, fetching might take several seconds during which the user may scroll through the already rendered items and trigger another fetch. We need to prevent this second fetch from happening, otherwise when the returned promise resolves, it would append the results to the end of the list two or more times, causing duplicate entries.

version of the app they create a new JavaScript bundle and upload it to a Code Push server. When a user starts the app, it downloads the new bundle (if available) and stores it. In a typical scenario, the bundle would be applied upon the next app start but this is configurable. This way the user receives updates without any interruption on their side. This does not only give us the ability to publish updates at an arbitrary frequency but also offers greater control over the updates, since the user does not influence them.

In the app, I have used the Code Push service which is being developed by Microsoft and currently offered free of charge.

Other possibility this offers and that I have implemented is multi-stage deployment and testing. For the purposes of our app, three build configurations were set up: debug configuration where Code Push is not being used; this configuration is used for everyday development and runs in React Native's Dev mode. In Dev mode, React Native reports warnings and errors to the developer directly on-screen and performance is decreased.

The second configuration is Staging, which is set up to request the staging version of the JavaScript bundle from Code Push and uses Memsources's pre-release server at `cloud9.memsource.com` to serve its requests. This version is made for testing the application's new features and also its compatibility with the Memsources Cloud version which is the next to be deployed to production. The development mode is not enabled in this configuration and thus the app behaves like a production application.

Finally, the third setup is for release. This configuration uses the corresponding JavaScript bundle from Code Push to get its updates. This is the version that runs on the phones of the Memsources's customers. When suitable, the updates made in the staging version can be easily promoted to the release build of the application by a single Code Push CLI (Command Line Interface) command.

4.7 Code Quality Tools

Due to JavaScript's dynamic nature and the absence of any transformation that would try to verify code correctness before it is run, it is relatively easy to introduce bugs that only come to light during runtime. There are, however, tools for code quality assurance that help developers find potential bugs before the code is executed. In this project, I have used two such tools which this section shortly describes.

■ 4.7.2 ESLint

ESLint³ is a linter – a tool that flags potential problems in source code. ESLint takes the form of a set of rules that the developer specifies and ESLint warns her when the code violates a particular rule. Rules may describe a potential bug in source code (such as calling a function that is not defined) or a desired coding style (such as using semicolons at the end of a line).

ESLint itself does not force any rules onto the developer. Instead, different rule sets can be obtained online and plugged into the project. Choosing such rule set is often a matter of personal preference or the technology that the project uses. For example, there are React Native-specific rules that eg. warn about having unused style definitions in the component code.

During development, I have used ESLint rules assembled by the React community, available on npm under `eslint-plugin-react`, `eslint-plugin-react-native` and `eslint-plugin-flowtype`.

■ 4.8 Navigation

In the context of React Native, by navigation I mean transitioning between different screens of the app. Navigation integrates with the components and also stores very closely, because many parts of the code will want to trigger navigation a different component as a response to user input or network event, so the navigation solution is of great importance.

React Native started off with two solutions for navigation - the Navigator and NavigatorIOS. Navigator is implemented entirely in JavaScript, runs on both platforms and tries to mimic the appearance of native navigation, while NavigatorIOS leverages the native navigation of iOS. They originally started as two competing implementations solving the same problem [23] with the goal of assessing which of the two solutions should be supported further on. Ultimately, the Navigator solution was found to be better for reasons described later on, and Facebook used it in the F8 and Facebook ads applications.

There is one drawback to Navigator - it is only trying to mimic the native navigation. This includes navigation bar with its animations, as well as transitions to and from different scenes of the app and implementation of the swipe back gesture. This, to a certain extent, can negatively affect the user experience. Having navigation controlled by

³<http://eslint.org/>

version is shipped every other week and updating is not always straightforward due to breaking changes (although this is changing to one release per month as of January 2017), lack of quality documentation, or my effort to create partially different UIs on each platform so that the user experience is on par with what the user would get in an application designed specifically for iOS or Android. This sometimes led to dead ends such as when I had to replace the entire navigation solution for a new one. Navigation as a whole is an interesting topic, which is why I devoted an entire section to it.

Another issue is minor differences in behavior on each platform – the component appearance on iOS may not always be the same as the one on Android. This involves borders, border radius or animations. However, given React Native’s complexity — the library uses JavaScript, Java, Objective-C, C++ and also C — it does a very good job in abstracting the platform away.

Other very painful issue, are animations which are controlled from the JavaScript thread. This means that the JavaScript thread has to periodically send commands to the native layer for the animation to run. If the JavaScript thread has too much work on it, issuing the command is delayed and the result is a laggy animation. This issue, however, is already partially solved and offloading the animations from the JavaScript thready will probably be fully functional in the first months of 2017.

Other difficulties were met in working with files (uploading and downloading jobs), since this needed to be done once for each platform and involved coding in Java as well as Objective C and Swift. Working with files has many hidden culprits given by different behavior on older Android versions, file permission issues and problems resulting from different file sources - ie. files imported from other applications versus files from `UIDocumentPicker` on iOS.

■ 4.10 Open Source Software Contributions

Throughout the development of the application, I have contributed to different projects by bug fixes or code and documentation improvements. The projects, all open source and hosted on GitHub, include:

- `react-native-keychain`
- `mobx-utils`
- `react-native-scrollable-tab-view`



Chapter 5

Testing and Crash Reporting

The Android and iOS versions of the application were implemented in parallel, mostly on my personal machine with Ubuntu 16.04 and my Nexus 5 phone with Android 6.0.1, following a verification and fine-tuning for iOS. The development phone for iOS was iPhone 5C running the latest version of iOS which was available at the time.

I have also used the iPhone and Android emulators for development and to see how the application looks with different screen resolutions and OS versions (the latter is especially relevant for Android). However, having access to real devices was crucial, since only a real device can give a feeling of how well touchable elements respond to touches, how the software keyboard influences the displayed content and how the application functions in terms of performance. A real iPhone device was also needed for implementing the native module for creating jobs, since the background capabilities of NSURLSession of an iPhone emulator do not fully correspond to the behavior of a real device.

All throughout the development, the application was receiving its data from Memsources servers, ie. I have not used any server implementation specifically for the development.



5.1 Unit Testing

The lowest level upon which the application is tested is unit testing. I have chosen Jest ¹ for implementing the unit tests. Jest, like several other tools I have used, is a library

¹<http://facebook.github.io/jest/>

■ First Test (knowledgeable user)

In the first test, all tasks were completed. The test revealed some inconsistencies in how project metadata is presented in the project info screen. User also reported that the control for selecting linguist (in the screen for creating new jobs) doesn't seem to be a control that would react to touch (button), plus the heading which reads "select linguists" is not intuitive.

Next, I observed that the user expected something (item selection) to happen when they tapped an item in the job list. The user got around it by long-pressing the item, which is the intended way of selecting it. The selection logic was implemented at a time when there was another action expected to happen upon short tap (a preview of the imported job). This functionality was decided not to be needed in the end. I have later implemented a fix for this - the short as well as long taps are now used for item selection.

When selecting Term Bases (TBs) and Translation Memories (TMs), the user was slightly confused by the term "Selected TMs". I agree a better term would be "Attached TMs" as was suggested, but chose the term "Selected TMs" to keep consistency with Memsources Cloud. The user also found the screen for adding TMs and TBs generally confusing. This is due to too many controls being displayed on one screen (a control for filtering the results, control for selecting the target language and workflow step for the assignment, along with checkboxes and dialogs for selecting different parameters). This problem was later addressed by splitting the adding of TMs and TBs into two screens, so it now involves an extra step, but is much clearer.

■ Second Test (power user)

In the second test, all tasks were completed as well. The second user performed the project and job tasks successfully. They also found an unexpected text reading "owner from different organization" on some project items displayed in the project list. This is an older issue caused by the API not returning the project owner. It remained in the application since I forgot to investigate the root cause. At any rate, the message looks confusing and will be removed and I will look more into the cause and possible solutions. The user also requested one feature and noticed one bug, both relating to content displayed in the project info screen.

Similar to the first test, I observed the user was confused when adding TMs because of they expected that the TMs in the screen that lists them to react to touches, while the listing is intended as a read-only preview. I later addressed this problem by adding

5.2.1 Test Conclusions

The tests of the application for Android were completed by all users. Unfortunately I didn't manage to run a user test with the iOS version of the application. The tasks related to projects and jobs were completed by all users without problems. Issues emerged when attaching Translation Memories and Term Bases to the project. This is due to the way items are presented for selection. I have already addressed some of these issues and will continue working on them.

The way TMs and TBs are selected can be further improved. Now, while the mobile app is consistent with Memsource Cloud, I find it somewhat confusing (and the tests confirmed it): when the user selects eg. a TM in read more, write mode is automatically selected too and user is not always in complete control. Not all checkboxes that seem to be possible to select can be selected due to selection constraints.

I believe a better solution to the problem is a single button reading something like “select TM (TB) mode”, which would open a list with all possible modes for the TM (TB) and only one mode could be selected. The list would show all modes: “Read & Write”, “Read only” in case of TMs and in case of TBs “Read & Write”, “Read & write & QA”, “Read only”, “Read & QA”. If applicable, the modes would be displayed along with an explanation of why a particular mode cannot be selected, or that selecting the mode will overwrite the mode of some other TM (TB). This would give user complete control and no unexpected automatic selecting or de-selecting happening as it is now.

5.3 Crash Reporting

One of the requirements is the ability to collect crash reports from users running the application so that we can observe how it functions on their devices and react to potential issues.

There are several services which provide crash reporting as well as means for collecting information about how the app is used, similar to analytics as it is on the web. One of such solutions which is widely used and also has community-developed bindings for React Native is called Fabric ² and is provided by Twitter free of charge for both iOS and Android. Fabric's crash reporting service is called Crashlytics, and the analytics is marketed under the name Answers. I have chosen Fabric because of its popularity, the ability to use Crashlytics as well as Answers and the fact it is available for both platforms. I have incorporated Fabric into both Android and iOS versions of the app.

²<https://get.fabric.io/>

While Fabric with the community-developed module for React Native does report the application crashes, I am not completely happy with how the reported issues are presented in the Crashlytics dashboard. In particular, when a crash happens in the JavaScript layer of the application, the information is merely passed to the native module which backs the JavaScript module. In the native module, an exception is thrown and its information is collected and recorded by Crashlytics. The problem with this approach is that all exceptions are thrown from exactly the same line in code and Crashlytics considers all of them to relate to a single bug. Then, in the Crashlytics dashboard (which is best accessible through web browser) all of the exceptions are grouped together which makes working with Crashlytics uncomfortable. I have investigated other possible crash reporting solutions and found Bugsnag ³ which directly supports React Native. It, however, does not provide any analytics service and is not offered free of charge. I will evaluate the situation around bug reporting after we gain more experience with Crashlytics and might make a switch.

³<https://bugsnag.com/>



Chapter 6

Conclusions and Future Work

Mobile devices are ubiquitous and people use them extensively in their everyday lives. As a consequence, many online services are accessible not only through the web, but also via applications made specifically for mobile devices. Indeed, the presence of mobile devices is so strong that many products are even built solely for mobile and do not exist on the desktop.

Memsource Cloud is an online service that had no mobile application and the goal of this thesis was to change it. In its scope I have introduced Memsource to the reader and explained the motivation. Since it was clear from the beginning that the application must be multiplatform, I have performed an analysis and comparison of the tools for multiplatform development. Prior to starting the development I have collected requirements for the application and created mockups at different fidelity levels. The application was implemented mostly in JavaScript and React, with Objective C, Swift and Java being used in the functionality implemented natively.

I have laid the foundations for an application that will allow Memsource users access the most important features of Memsource Cloud. The current functionality includes CRUD operations on projects and jobs, as well as working with term bases and translation memories and other features.

React Native, despite its young age proved to be a valuable and functional library for multiplatform mobile app development. Its easy integration with the underlying platform was important for working with files in the native modules. Also MobX is a library that works very well with React and I have enjoyed working with.

6. Conclusions and Future Work

The future work will involve further development of the application within Memsourse. I will continue adding more features to the application, and improve the styling of some components. Another important future step is to introduce a continuous integration tool.



Appendix A

Tasks for User Testing

■ Log In

Log in with the following credentials: username “VNovak” and password (removed). You will be presented with the list of all your projects which are shown in different tabs.

■ Projects

Take a look around and familiarize yourself with the screen. When you're ready, create a new translation project with the following parameters:

- project name: “yet another project”
- Client: Slavia
- Domain: Machinery
- Source language: English
- Target languages: Czech, German
- Due date: 19th of January at 7 pm
- Workflow steps: select translation and revision

That's it for today, thank you for your participation!



Appendix B

Installation Instructions

The simplest way to install the application is using the attached apk file (Android) or ipa (iOS).

Alternatively, you can build the application from source which requires to set up react native according to the guide found at React Native docs ¹. In that case, copy the `memsourceApp` folder to your computer and make sure to fetch all git submodules. The next step is to run `npm i` to install the dependencies from npm. Then run `react-native run-android` to have the android app installed on your device or emulator. Alternatively, use `react-native run-ios` to install the app on iOS device or emulator. This will install the app in debug mode. Before running it, you will need to run the command `npm start` to start the React Native packager.

¹<https://facebook.github.io/react-native/docs/getting-started.html>



Appendix C

Attachment Contents

The attached data contains the text of this thesis both in pdf and in its \LaTeX sources. Furthermore, it contains the project source files and the apk and ipa files for installation.

The structure of data is following:

```
/
├── memsourceApp.....application sources
├── install.....apk and ipa files
├── text.....the thesis text directory
│   └── latex.....the  $\text{\LaTeX}$  sources
```




Appendix D

Index



E

ESLint, 47



F

Flowtype, 46



H

hot reloading, 15



J

JSON, 37



L

live reloading, 15



N

native module, 16

P

pure function, 25

I T

term base, 6

translation memory, 6

Appendix E

Bibliography

- [1] Dan Abramov. *Redux - Three Principles*. 2017. URL: <http://redux.js.org/docs/introduction/ThreePrinciples.html> (visited on 01/04/2017).
- [2] Inc. Apple. *App Programming Guide for iOS - Background Execution*. 2016. URL: https://developer.apple.com/library/content/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/BackgroundExecution/BackgroundExecution.html#//apple_ref/doc/uid/TP40007072-CH4-SW5 (visited on 01/06/2017).
- [3] Inc. Apple. *Document Picker Programming Guide - Accessing Documents*. 2016. URL: <https://developer.apple.com/library/content/documentation/FileManager/Conceptual/DocumentPickerProgrammingGuide/AccessingDocuments.html> (visited on 11/06/2016).
- [4] Inc. Apple. *URL Session Programming Guide - Using NSURLSession*. 2016. URL: https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/URLLoadingSystem/Articles/UsingNSURLSession.html#//apple_ref/doc/uid/TP40013509-SW44 (visited on 11/06/2016).
- [5] Lee Byron. *React Conf 2015 - Immutable Data and React*. 2015. URL: <https://www.youtube.com/watch?v=I7IdS-PbEgI>.
- [6] Tom Coleman. *The State of JavaScript 2016 - State Management*. 2016. URL: <http://stateofjs.com/2016/statemanagement/> (visited on 01/04/2017).
- [7] Cordova. *Overview*. 2017. URL: <https://cordova.apache.org/docs/en/latest/guide/overview/> (visited on 01/08/2017).
- [8] Jerry Dischler. *Building for the next moment*. 2015. URL: <http://adwords.blogspot.com/2015/05/building-for-next-moment.html>.
- [9] Bonnie Eisenman. *Learning React Native*. Edition 1. O'Reilly, Dec. 2015, p. 254. ISBN: 978-1-491-92900-1.

- [10] ExponentJS. *ExNavigation*. 2017. URL: <https://github.com/exponent/ex-navigation#an-important-note-about-the-future> (visited on 01/07/2017).
- [11] Comscore Inc. *Major Mobile Milestones in May: Apps Now Drive Half of All Time Spent on Digital*. May 2014. URL: <https://www.comscore.com/Insights/Blog/Major-Mobile-Milestones-in-May-Apps-Now-Drive-Half-of-All-Time-Spent-on-Digital>.
- [12] Gartner Inc. *Gartner Says Chinese Smartphone Vendors Were Only Vendors in the Global Top Five to Increase Sales in the Third Quarter of 2016*. URL: <http://www.gartner.com/newsroom/id/3516317> (visited on 01/04/2017).
- [13] Gartner Inc. *Gartner Says Worldwide Smartphone Sales Grew 9.7 Percent in Fourth Quarter of 2015*. 2016. URL: <http://www.gartner.com/newsroom/id/3215217> (visited on 05/18/2016).
- [14] IDC Inc. *Smartphone OS Market Share, 2016 Q3*. URL: <http://www.idc.com/promo/smartphone-market-share/os> (visited on 01/04/2017).
- [15] Kantar Inc. *Smartphone OS sales market share*. 2016. URL: <http://www.kantarworldpanel.com/global/smartphone-os-market-share/> (visited on 06/01/2016).
- [16] Statista Inc. *Smartphone share of visits to websites in the United States in 2014 and 2015, by industry*. 2015. URL: <http://www.statista.com/statistics/412971/us-mobile-website-industry-visits-share/>.
- [17] Ecma International. *Map Objects*. 2015. URL: <http://www.ecma-international.org/ecma-262/6.0/#sec-map-objects> (visited on 01/05/2017).
- [18] *Introducing Hot Reloading*. 2016. URL: <https://facebook.github.io/react-native/blog/2016/03/24/introducing-hot-reloading.html> (visited on 03/24/2016).
- [19] Ionic. *Installing Ionic and its Dependencies*. 2017. URL: <http://ionicframework.com/docs/guide/installation.html> (visited on 01/08/2017).
- [20] *JavaScript Environment*. 2016. URL: <https://facebook.github.io/react-native/docs/javascript-environment.html> (visited on 10/13/2016).
- [21] L. Masinter. *Returning Values from Forms: multipart/form-data*. 1998. URL: <http://www.ietf.org/rfc/rfc2388.txt> (visited on 11/06/2016).
- [22] *Native Modules*. 2016. URL: <https://facebook.github.io/react-native/docs/native-modules-ios.html> (visited on 11/03/2016).
- [23] React Native Radio. *episode 40 Navigation in React Native with Eric Vicenti of Facebook*. 2016. URL: <https://devchat.tv/react-native-radio/40-navigation-in-react-native-with-eric-vicenti> (visited on 01/05/2017).
- [24] *Sharing Code Options in Xamarin*. 2016. URL: https://docs.xamarin.com/guides/cross-platform/application_fundamentals/building_cross_platform_applications/sharing_code_options/ (visited on 04/13/2016).

- [25] Michel Weststrate. *MobX - Simple, scalable state management*. 2016. URL: <http://mobxjs.github.io/mobx/>.
- [26] *Xamarin Forms*. 2016. URL: <https://www.xamarin.com/forms/> (visited on 04/13/2016).
- [27] *Xamarin Platform*. 2016. URL: <https://www.xamarin.com/platform> (visited on 04/13/2016).
- [28] *Xamarin.com*. 2016. URL: <https://www.xamarin.com> (visited on 06/02/2016).
- [29] Tadeu Zagallo. *Bridging in React Native*. 2015. URL: <https://tadeuzagallo.com/blog/react-native-bridge/> (visited on 01/03/2017).