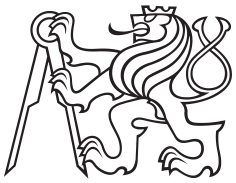
Master Thesis



Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

Mobile Application for Memsource 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

Figures

Tables

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.

[1] https://www.memsource.com/en/features

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 for Translation Agencies adds more features on top of the freelancer version, notably the possibility to work with users, set up user roles and workflows. This allows project managers within the agency to distribute translation jobs among translators and specify the workflow through which multiple versions of a translated document can be kept in a project. A typical workflow may consist of translation, editing and proofreading.

The ultimate editions also support advanced features and, more importantly, access to Memsource Cloud API. Lastly, the version for Translation Buyers is intended for corporate customers who need to have various texts translated for their business and through Memsource, they're connected to the translation agencies or freelancers who will do the job for them.

1.1 Motivation

Memsource operates on the market of CAT (computer-aided translation) tools since 2010 an it is its best effort to provide modern and innovative solutions for translators based on its SaaS model. This effort is fulfilled by a set of described web and desktop-based applications. The current tools developed by Memsource are designed for use on computers mostly with a large screen, i.e. laptops, desktops, or large tablets through a web browser (with the exception of desktop editor which can be installed on Windows, OS X and Ubuntu).

The sector of mobile devices, however, remains largely uncovered by Memsource. In today's world, mobile devices play an ever important role, allowing people to access online resources from virtually anywhere and at any time. There are a number of studies that show the increasing presence of mobile devices on the internet and in both our professional and personal lives. Statista offers an overview of Smartphone share of visits to websites in the United States in 2014 and 2015, by industry [14]. This statistics shows that the shares vary greatly between industry, with technology websites having 11.7% share of visits from mobile, while for media and entertainment, mobile accounts for 36.6%. Comscore goes even further and in its study from 2014 [9], it claims that more than a half of time spent with digital media (social networks, videos, magazines, etc.) in the U.S. is spent on mobile devices. An

interesting blogpost by Google AdWords Vice President from 2015 [8] reads that "more Google searches take place on mobile devices than on computers in 10 countries including the US and Japan.". It clearly follows we have to design our software products to play well with mobile devices and the limitations that are inherent to them.

While Memsource can be accessed from a mobile device through its internet browser, the web browser cannot take full advantage of the features of the platform it runs 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 inquires are made. Making the project and job creation from email as smooth as possible is one the most important goals that the application should allow and that are overly complicated or impossible with a standard web browser.

Creating a mobile application therefore gives us more flexibility. Apart from that, on the Memsource Cloud web, there are a number of options and settings available (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 or do not have a computer at their disposal. For the aforementioned reasons, the existing solution is not suitable for such needs and therefore, the goal of this thesis is to develop an client application specifically tailored for use on mobile devices. This app should contain a subset of the features that are currently available and make the 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 Memsource 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 Memsource Cloud

Each of the target groups has a partially different way of working with the Memsource 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 Memsource Cloud (which currently supports translation of about 50 file formats). In Memsource, 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 Memsource 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.

An important concept here is segment, which usually consists of a single sentence of the translated document. Translation is done one segment after another. Translation memory is essentially a database of previously translated segments. When translating files that are similar in their content, for example contracts, official documents or different version of the same document, it often happens that there are partial or full matches in the translated text. Translation memory can spot these similarities and offer the translator a previously confirmed translation. It can also smartly replace small differences such as numbers or names. Term base , similarly, is a database of specific terms (single words) and their translations into multiple languages. It can include additional information such as a definition, subject area or industry and etc. Its job is to assure that a term is used consistently throughout the translated document. The Quality Assurance tools can be used to check if the document meets criteria such as no trailing spaces, no repeated words in close proximity, correct spelling and more.

While the document is being translated, it can optionally go through multiple stages of processing. This feature is called workflow and it allows to keep multiple versions of the same translation job in a project. Workflow may, for example, consist of translation, editing and proofreading. Therefore, once a segment is confirmed in translation, it is propagated to the next workflow level where other agency employer can continue working on it. Once the document is translated, it is delivered to the customer. Figure 2.1 shows a translation project in Memsource Cloud.

2.2 Requirements

The requirements for what the application has to support were determined gradually by a discussion with the Memsource developer team and CEO, followed by discussions with members of the support and product team who understand the customer use scenarios. The outcome of conducted discussions is described in this section.

First I present a summary of requirements, a more detailed description is also included.

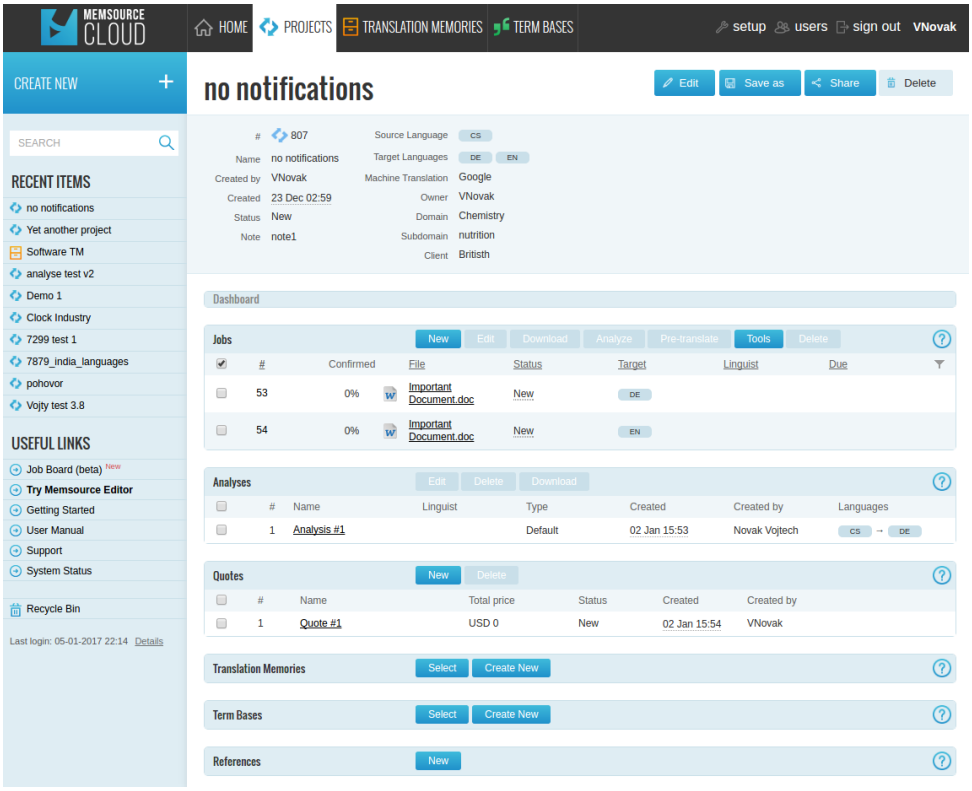


Figure 2.1: A translation project in Memsource Cloud

Summary of requirements

Non-functional requirements

1. app has to run on iOS and Android and be stable
2. app should be written using technologies already used in Memsource
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 Memsource 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 Memsource project manager and translator roles
2. support multiple users logged in at the same time and allow to switch between accounts
3. list projects and allow to filter them
4. list jobs and allow to filter them
5. support adding jobs and projects
6. allow to edit projects and jobs
7. store user credentials in a secure manner
8. support error reporting
9. use Memsource API to acquire the content
10. present its UI in English only

In some cases, the tools developed by Memsource are complex so that they support different user needs (for example lots of import options for translation jobs). We do not want to bring all this complexity to the mobile application and thus needed to find the features we wish the mobile app to include.

The context in which the app will be used is a user needing to create, access or modify translation job and project information while being on the move or without access to a computer.

Follows a detailed description of requirements:

1. User requirements
   1.1. The app will target handheld devices and a narrow group of users—professional users of Memsource Cloud who understand its features.
   1.2. The app has to support the project manager and translator user roles

1.2.1. App has to support multiple users being logged into the app

1.2.2. App will display content for only one user at a time and there will be a simple way to switch between users (i.e. set the active user).

1.2.3. The number of added accounts will be limited to a maximum of four.

The situation when a single Memsource user has several accounts is not common, but it is not unheard of (This does happen to users who work for several translation agencies). This will make working with several user accounts easier, without the need to log in and out or use several browser windows as is the case when using a computer. Currently, the API uses a token for authentication and this token is sent 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 Memsource editions.

2. Content requirements

2.1. The app will load all its content through a public HTTP API which is provided by Memsource. 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 5 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 Memsource 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 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

3.1.3.1. App will provide means to show a list of jobs contained in the project, showing relevant information for each job—job name, linguist name, due date, target languages and status.

3.1.3.2. PM has to be able to create (upload) new translation jobs from document providers available on the platform.

3.1.3.3. Adding jobs to project from an email attachment also has to be supported.

3.1.3.4. When adding a new job, user will be able to enter the target languages, select due date, enter linguists for the job and have the option to notify them of the new job.

3.1.3.5. After a job is uploaded, user has to be informed about it. Similarly, user has to be informed about unsuccessful file uploads and requests.

3.1.3.6. App should only support creating jobs from the most common file types (MS Word, Excel, Powerpoint and HTML) and cover their file import options to the same extent as Memsource Cloud.

3.1.3.7. User needs to have the ability to select individual job, as well as several jobs, to download, edit and delete them. Editing consists of changing the linguist, status and due date.

3.1.3.8. Similarly to searching and filtering of projects, the app will allow to filter jobs by name, status, target language, linguist and due date.

3.1.4. The application will not cover support for translating, i.e. Memsource Editor will not be a part of it since we found that it is overly complex to be used on a small screen of a mobile device.

Note: The advanced project creation options for machine translation, analysis and other which are visible in Memsource Cloud, will not be included.

3.2. Functionality for Linguist users

3.2.1. The app will support listing projects that are visible to her, filtered based on status (new, accepted, completed). The list will contain relevant information—name, date created, owner, and source and target languages.

3.2.2. User will be able to open a project, download or preview selected jobs and change their status (accepting or rejecting and marking them as completed).

3.2.3. Same as with a PM, translator will have the ability to search based on job filename, status, target languages and due date.

4. Other requirements

4.1. It is important the app be developed for at least the two major mobile phone platforms, that is Android and iOS.

4.2. The app must be developed using programming languages that Memsource developers are already familiar with, that is one or more of: Java, Groovy, C++, Javascript or any other technology if it brings substantial benefits.

4.3. For the user, the app should look and feel as close to a native app as possible.

4.4. The application has to support error reporting to allow Memsource continually improve the user experience, app stability and addition of new features.

2.3 Analysis of Platforms and Development Tools

Today's market of mobile devices is largely divided between two major 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 [11]. 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 [13]. 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 market share: 86.8%, 12.5% and 0.3% for Android, iOS and Windows Phone, respectively according to IDC [12] and a report from Gartner [10] shows very similar numbers.

Since we are looking for a multiplatform solution, 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" [25] 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 [24]. Xamarin.Forms is an attempt to bring code reuse event higher, usually more than 90% [24] 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 [23].

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 defined in C# code or XAML (an XML-like sytax for UI description) which can be done in a graphical UI designer.

[1] https://github.com/xamarin

Using Xamarin.Forms is best for apps that require little platform-specific functionality and apps where code sharing is more important than custom UI. Xamarin.Forms UI can be done either in C# or in XAML but without the support of UI designer.

Development with Xamarin is done under Windows or OSX and the language of the framework is C#. The code must be built and sent onto a device or emulator, which can take a considerable amount of time. Debugging is done in Visual Studio. Xamarin allows to write the code shared by all platforms in form of a Shared Project or a Portable Class Library [21] which are described in the following sections.

2.4.1 Shared Project

Shared Project is the simplest way to share source code between platforms. This way, a cross-platform app that supports Android, iOS and Windows Phone would require an application project for each platform. Additionally, there would be a Shared Project for the code common to all projects.

The code within a Shared Project can be branched into platform-specific parts using compiler directives (e.g. using #if ___ANDROID___). The application projects can also include platform-specific references that the shared code can utilize. The downside to this approach is that a Shared Project has no output assembly. During compilation, the files are treated as part of the referencing project and compiled into that project's DLL. This does not allow to distribute the code from Shared Project as an independent library.

2.4.2 Portable Class Library

Portable Class Library addresses the fact that Shared Project cannot be distributed as a standalone library. Portable Class Library offers the possibility to distribute it independently of the mobile app.

The disadvantages are that it is not possible to use compiler directives to reference platform-specific features and the fact that different platforms often use a different subset of the .NET Base Class Library (BCL) and therefore only such subset is available to use. This, however, can be to some extent

circumvented by the Provider pattern or 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 framework 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 frameworks. 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 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. 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.

RN application code is written in JavaScript which runs in JavaScriptCore engine on the device [17]. RN features what is called a *bridge* [5]. 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-

[2] https://facebook.github.io/react-native

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 view, important features of RN are its live and hot reloading [16]. Live reloading enables the developer to apply code changes to the app running on a device or in an emulator quickly. Live reload in fact takes about five seconds on my development machine. This is a tremendous improvement over traditional native development, which until recently—with the introduction of Instant Run into Android Studio—suffered from the slow process of building an app and loading it into a device or emulator.

The other feature called Hot Reloading offers essentially the same functionality as Live Reloading with the advantage of being faster and preserving the application state—the screen displayed before and after the Hot Reload is the same—thanks to which the developer does not need to navigate through the app to the screen where the change is being done. This is especially helpful for making changes to the UI layout and styles because Hot Reload needs about one second to take effect. These features can significantly accelerate app development and improve the developer experience. The downside of Hot Reloading is that it mostly works for simple changes in the UI but fails when modifications are of larger extent.

Debugging RN is accomplished through running the code in Chrome browser or external debugging tools (such as those included with Visual Studio Code or Webstorm) where user can set breakpoints and work with code similarly to working in web development. In this case, all the JavaScript code runs within Chrome's V8 engine itself and communicates with the phone or emulator via WebSockets.

2.5.1 Native Modules

When a developer needs native functionality which is not already provided by RN, they will often find such functionality already implemented by the community which surrounds RN. In such case, the component is available through the Node package manager (npm).

In case the functionality is not yet implemented, the developer can create a

[3] http://flowtype.org/
[4] https://www.typescriptlang.org/

native module [19] for it. Native module consists of code written in Java (for Android) and Objective-C (or Swift) for iOS which implements the desired functionality and of JavaScript code that will expose the native functionality to the app's JavaScript code. The native code is invoked from JavaScript through the RN bridge and results (if any) can be passed back by a Promise or callback. Native modules give developer the freedom to implement any functionality desired, as long as it is available on the platform, but have the downside of needing to code both for iOS and Android.

2.6 Ionic

Currently in version 2, RC 4, 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 is integrates with Angular 2, a framework for web development from Google. Ionic has put o 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 quite 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, a mobile application development frame-

[5] http://ionicframework.com

work which provides access to native platform features such as camera, sensors, filesystem or contacts. 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 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 an even export it as an Ionic app.

Also, with live reloading, the iteration process is a lot faster than traditional procedures involving compilation.

2.7 Conclusions

There is currently a very strong competition in the area of multiplatform mobile app frameworks and choosing the right one for one's needs is difficult. All of the researched solutions are very capable. After developing simple proof-of-concept apps using the three described frameworks I first ruled out Xamarin. Although C# can be considered a very mature and powerful language, the reasons for ruling Xamarin out were the need to write UI twice - which would involve learning the specifics of Android as well as iOS (we did not want to use Xamarin.Forms because we were uncertain about whether it would not limit us and the available demonstrations apps written using Xamarin.Forms were not stable) and its slow development iteration cycle which for me felt like a big drawback.

Ionic 2 is a popular framework. After creating simple application in it, I could not help but notice very slow startup times: between 6 and 9 seconds for a very simple app. This issue was confirmed by posts in the community forum and according to an Ionic representative, the startup time will be improved. Moreover, after installing several apps developed in Ionic we noticed not all of them work on all devices (probably due to cpu family), which left me with mixed feelings. The advantage of Ionic 2 surely lays in its maturity, strong community and lots of readymade UI widgets styled differently for each platform.

React Native was chosen after difficult comparisons. Its advantages are that it is backed and used by Facebook, it is being developed at a quick pace and has a growing community with lots of components available or in development. It also provides greater UI flexibility than Ionic, better performance, and the ability to easily communicate with native code is a need for our use case since the app will have to deal a lot with files (background file uploads, downloads, opening the app with them). Most importantly, RN does a very good job at transforming the React code to the native views. Obvious disadvantage is its immaturity and probably the need to work harder to get eye-appealing designs because unlike Ionic, RN does not offer styled UI components ready to be used but only the essential components with no multiplatform styling.

Table 2.1 provides a quick comparison of the described libraries or frameworks against selected criteria. Plus sign (+) denotes positive result, while zero (0) and minus (-) denote neutral and negative result.

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).

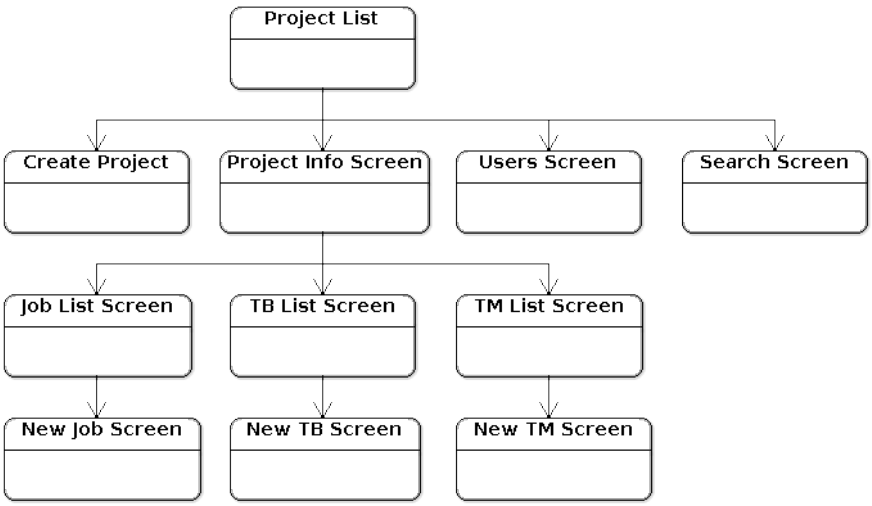


Figure 3.1: Structure of the app's screens.

3.2 Prototyping

Based on the collected requirements, a set of mockups was constructed. Some of these were later used to construct a prototype for an Android device that was used for tests with users. Both the mockups and the prototype consider the project manager role because its feature set is a superset of the one of the linguist role.

The mockups were discussed with employees of Memsource support team. Different ideas of presenting information to the user were brought up and consulted. Memsource support members provided feedback and deeper insight on how different features are used and whether they are needed. Understanding the workflow was important to designing the final mockups.

In the end, the prototype largely follows the structure of Memsource cloud, but it is simpler in terms of the number of supported options and gives off Android platform feel. The following pages contain several figures that present selected mockup screens.

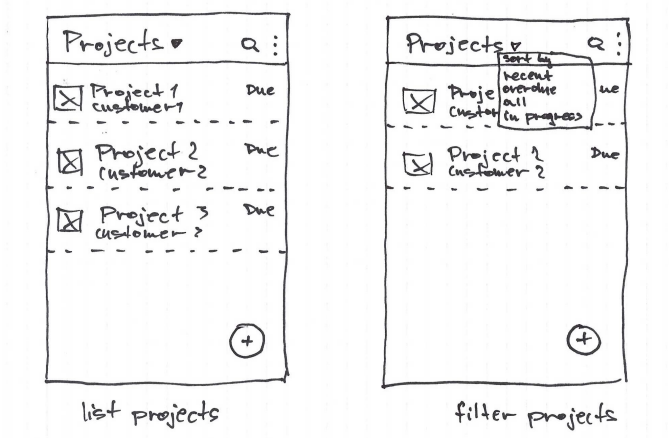


Figure 3.2: Mockups showing the project list screens. Second screen shows the chevron active, where user can filter displayed projects.

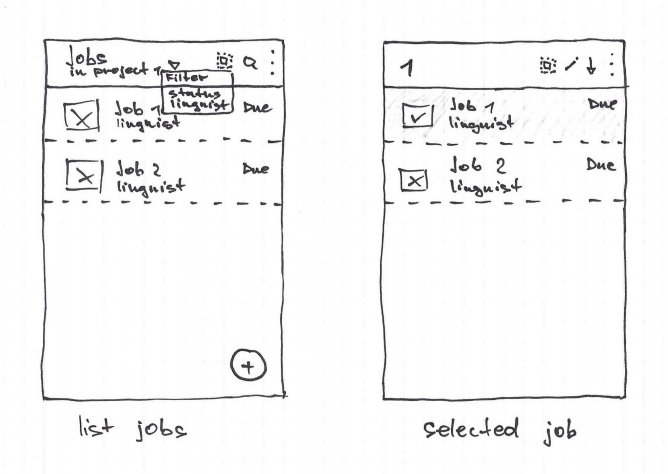


Figure 3.3: Job list (left) and a mockup where "Job 1" is selected and different actions are available for it (right).

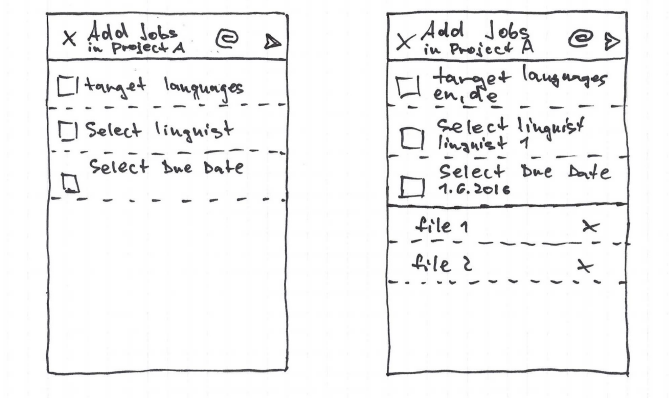


Figure 3.4: Adding a new job to a project.

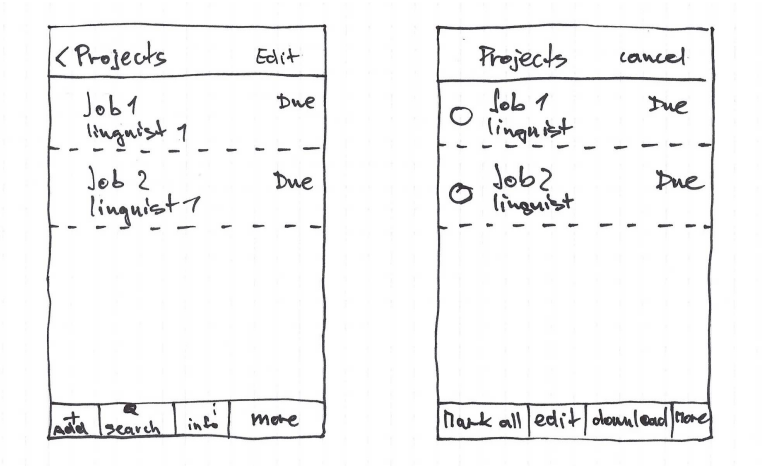


Figure 3.5: Different way of listing jobs within a project on iOS. The second screen displays the state of the first after clicking on the 'edit' button.

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 patters 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.

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 and 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:

You are a Memsource 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 Memsource 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

[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

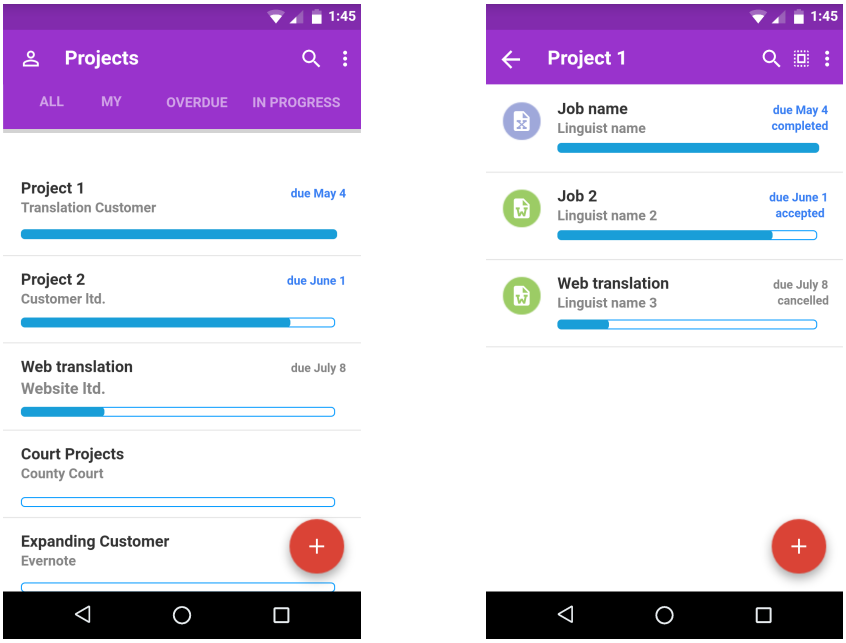


Figure 3.6: Listing projects (left) and jobs in a project (right).

(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 Memsource 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 [7].

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 [6] (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." [22]. 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

```
let appState = observable({
    timer: 0
})

setInterval(() => {
    appState.timer += 1
}, 1000)

let TimerView = observer((props)=>
    return <span>Seconds passed: {props.appState.timer}</span>
)

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 Memsource 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 Memsource API.

3.4 Client-server Communication

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

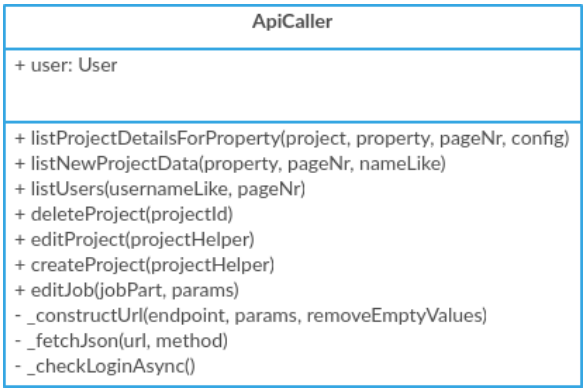


Figure 3.7: ApiCaller class

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.

3.5 Domain Objects and Stores

There are several core domain objects the application needs to work with. These represent the corresponding entities in Memsource 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.

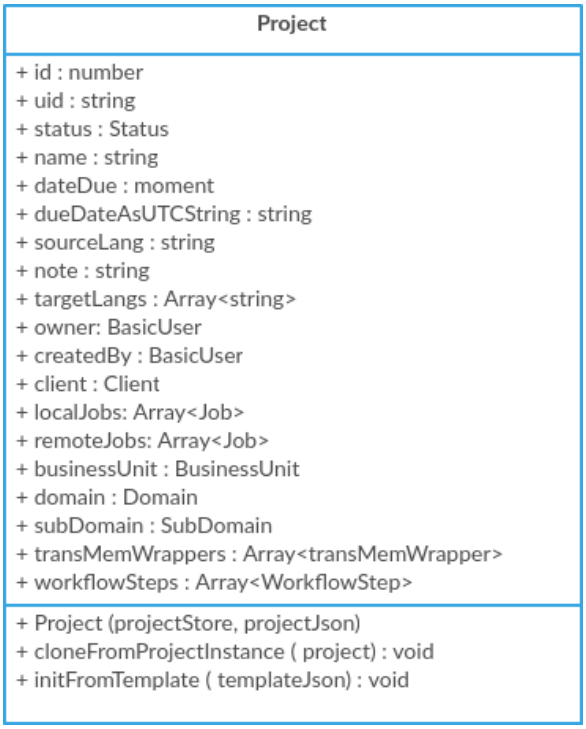


Figure 3.8: Project class

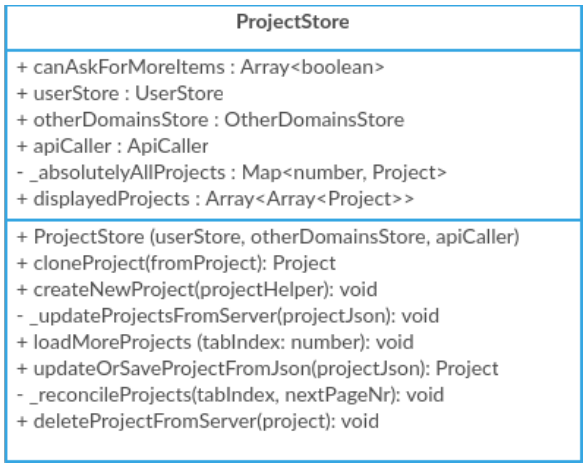


Figure 3.9: ProjectStore class

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.

3.5.1 Representing Users

Since the application has to support multiple users being logged in (with only one user being active at a time), we need objects for representing individual users as well as the collection of users who are logged in.

For this purpose, the User and UserStore classes are used. User instance contains data such as user name and id, and also user password, token, role and other. It is also the place where the user's search history is kept.

UserStore will contain an array of all users who are currently logged into the app, and a reference to the user which is currently active. Furthermore it will contain methods for creating new user instances and persisting the user information so that it is available upon application startup.

3.5.2 Representing Jobs

3.5.3 Platform-specific Look and Feel

React Native does not aim to provide developers with a way to run the same code on both platforms, instead it promotes the "learn once, write anywhere" paradigm and allows to create apps for both platforms while writing code using the same syntax.

Due to the nature of how both platforms are interacted with, we need to have an ability to make the user experience different per platform. As an example, take the Datepicker on Android versus the iOS Datepicker, or the Android navigation bar which often offers several actions (some with icons) versus iOS navigation bar which usually contains the title and no more than 2 actions. Actions that, on Android, would be included in the navbar, are often presented in a toolbar on the bottom of the screen on iOS, or hidden in an action sheet. Also note how Android works with long presses for item selection (for example in the Gmail app or the Downloads app), while this is usually done by an edit button in iOS navbar. The edit button then switches to an edit mode where user can perform their edits. If we want to follow these customs, this requires us to write separate code that would make the UI look and react to user actions differently on each platform, as well as custom code for the item selections and Android back button handling — when in edit

mode, the first back button tap disables the edit mode and only the second tap navigates to the previous screen.

React Native offers two ways how to go about platform-specific behavior. First is through the Platform module which gives information about the platform and its version. Another method is to use different file extensions (ie. `android.js` or `ios.js`) for components. The appropriate file will then be packed for the JavaScript bundle of each platform. Specifying what component to render by using different file extension is a 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 Memsource 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 Memsource 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 use 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 Memsource 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.

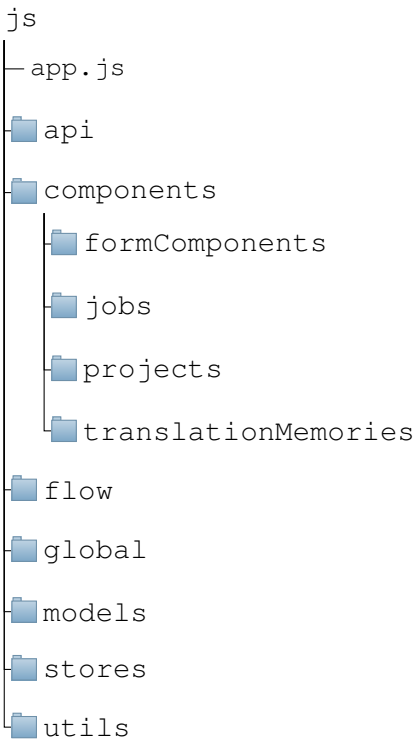


Figure 4.1: Simplified folder structure of the project.

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.

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 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.

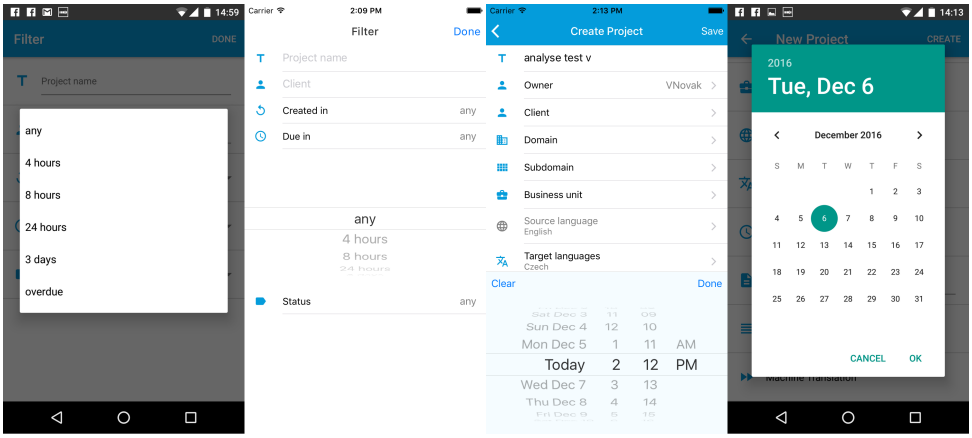


Figure 4.2: Platform-customized behavior of list and date picker components

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.

4.2 Stores

As explained in section 3.5, the information about projects, jobs, and other entities in 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 Memsource 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 [15]. 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 Memsource 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) 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 {
2   render() {
3     return <Provider color="red">
```

```
4           <App/>
5       </Provider>;
6     }
7   }
8   ...
9   @inject("color")
10  class Paper extends React.Component {
11    render() {
12      return
13        <div style={{backgroundColor: this.props.color}}>
14          the selected color is ${this.props.color}
15        </div>
16    }
17  }
```

I have used the `Provider` component extensively, but almost exclusively for the application's screens (e.g. job list screen) to get access to the stores, and for any components used within the screens, I pass the stores explicitly as a prop.

4.3 Upload Module

Upload module is a native module made to allow users to upload files to Memsource Cloud and create jobs from them. It offers different ways to select the files and allows to set up various import options. After the upload is triggered, a new item (a *temporary job*) representing the ongoing upload is added to the job list. A notification and the network activity indicator on Android and on iOS, respectively, inform the user of the ongoing upload.

There are important differences in the iOS and Android implementations, but both expose the same interfaces to the JavaScript layer. When talking about file handling in the native module, I will use the singular form for simplicity, but note the upload module has capabilities for uploading multiple files.

4.3.1 Android

To upload a file as a job, user has to either start the app and navigate to the "add job" screen and select the files for upload from a file picker, or start

the app "externally" by opening a file using the Memsource app, for example from the Gmail app or a file browser. 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 Memsource 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 user switches to a different app or even "kills" the app by swiping it away from the screen. The upload service also stores the results of uploads in SharedPreferences so that if the application is killed and the job creation finishes in background, it can be updated the next time it starts. The token is not persisted. If the app is still running in the foreground at the time when the job creation is confirmed, an event is sent to JavaScript and the views are updated. If that is not the case, the response of job creation is processed the next time the app is started or switched from background to foreground. If creation was successful the item (temporary job) which was previously added to the top of the job list is removed and the job list is re-fetched. In case of an error, the item is not removed but instead gives user an option to repeat the upload or remove the item manually. The native module is written in Java.

4.3.2 iOS

On iOS, the module provides the same functionality but behaves very differently internally. iOS is much stricter about how background tasks are handled. There are fewer things that needs to be taken care of by the developer and more that are taken care of by iOS. This gives the developer a lot less flexibility (which also resulted in a problem with the API) but also results in less coding. The job creation on iOS works in the following way: similarly to Android, files can be selected within the app or sent to the app from another application such as Mail. If a file is selected using the Document Picker, iOS automatically downloads (if needed) and saves it to the app's sandbox temporary folder from where it is available until the application exits [3]. In the case of importing the file from other application, the file is copied to app's sandbox and the application receives the url of the file. In the latter case, removing the file is the developer's responsibility.

Originally, I wanted to use the same approach for creating job as on Android, but it turned out not to be suitable because it is not possible to create a job from a file using the current `Create New Job` API in the background.

Let me expand on this claim: to upload a file in the background I need to use the `NSURLSession`'s BackgroundSession which uses a separate process [4] independent of the application to handle the background upload or download tasks. Its upload task only supports uploading from a file [4], i.e. the request body only contains the file's content. The `multipart/form-data` message which is used by the `Create New Job` API, however, contains a series of

parts describing the content of the request, embedded in the request body [18].

Therefore, a file uploaded by iOS's background upload task is sent directly in the body of the request and having a custom request body is impossible unless we write it directly into the file (which cannot be considered a good practice). Possible workaround here is to use another Memsource 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 of 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 awaken too often in response to its background requests, the interval in which it is be awaken 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 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 an 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 itself doesn't come with a mechanism for state persistence, and therefore another library, Serializr was used. Serializr provides a variety of functions for serializing data stored in different data structures and also custom objects. The data that needs to be serialized and the data structure used are described using decorators placed on the member variables of selected classes. The application also stores search history for all of its users.

While the implemented (de)serialization works well, it poses extra layer of complexity; implementing it was a task more lengthy than expected, partly due to some hard-to-find unexpected behaviors and unhelpful error messages. The (de)serialization, however, is implemented in such way that when an error happens (which is more likely to happen during deserialization), the app falls back to to not deserializing any data and instead loads the data only from the API.

State is serialized upon switching the app into the background. The number of items that are serialized is limited so that the set of (de)serialized data doesn't grow indefinitely. Serializr outputs a JSON object which is persisted using React Native's AsyncStorage as a string. Note that storage of user credentials is handled differently and is described in the next section. Upon app start, the objects that hold state are created empty, the JSON string is deserialized and all of the information is inserted back into the state objects.

4.4.1 Storing User Credentials

Communication with the Memsource API requires the user to enter their username and password. The app then asks for a token which is used for the requests to follow. The token validity is limited to 24 hours and the app therefore needs to request a new one once the current token's validity is approaching its expiration date. To be able to ask for a new token, the app needs to have the user credentials at its disposal, and persist them so that it doesn't need to repeatedly ask the user to enter them. Such storage, obviously, needs to be safe and the AsyncStorage used for state persistence does not

meet the safety criteria. To store the user credentials, I used a package which internally uses Keychain on iOS and an encrypted SharedPreferences entry on Android. I authored the Android part of the package which is now available as react-native-keychain on npm.

4.5 Data Fetching

Application fetches all data through the "Memsource REST API". The data is provided in JSON which makes it easy to work with, given that the app is written in JavaScript. 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.

If we want to have some universal fetching mechanism, it needs to account for all of these requirements. For this purpose I implemented the ProjectDetailsFetcher module. The most important function it exposes is the fetchProjectDetail function which accepts a project for which it fetches the detail (jobs, translation memories and term bases or other items potentially added in future). The other parameters include the field name (eg. jobs), a boolean denoting whether the request is a full reload request (one triggered by the pull down gesture), configuration object (eg. to specify a filter) and page limit that will not allow fetching more than specified number of pages. ProjectDetailsFetcher internally handles the number of the next page that should be requested for a particular project and property as well as tracking which requests are allowed or not. The fetchProjectDetail function returns a Promise which contains the response data. This data is usually requested from stores. ProjectDetailsFetcher does not handle displaying or hiding the loading indicators, as I have found it to be better to control this from the places where fetching is being requested because it offers more flexibility.

4.5.1 Handling Internet Connection Outage

The application's functionality is dependent on Internet connection since acquiring all of its data and possible user actions need access to the internet. However, once the application fetches its data, it, of course, stays in the memory and is available for reading. Moreover, the app serializes data which is needed to display the list of projects and the project info screen, which includes projects, clients, domains, subdomains, business units and other details. That way the data is available for reading even if the user starts the

app without internet connection.

When the app is offline, there is a bar displayed at the top of the screen, which informs the user that there is no internet access. Also, when a user's request times out, they are informed about it via a toast. This behavior however, may change if it is found to be too intrusive.

4.6 Multi-stage Deployment and Testing

One of the advantages of using React Native or hybrid application frameworks is the ability to use services such as Code Push [1] that enable the developer to update the application without going through Apple AppStore or Google Play Store submission process. This is achieved through being able to switch the JavaScript bundle which contains the app's logic for another one. When a developer wants to publish a new 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 Memsource'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 Memsource Cloud version which is the next to be deployed to

[1] https://microsoft.github.io/code-push/

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 Memsource'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.

4.7.1 Flow - Static Type Checker

Flow [2] is a static type checker for JavaScript developed by Facebook. It works by using type inference on JavaScript code even without any additional information provided by the developer. It tracks the type of variables as they are used throughout a program and allows to catch bugs before it is executed, without changing the existing code.

Flow attempts to infer the types whenever possible, but JavaScript code can be very dynamic and hard to analyze statically. Flow therefore offers ways to specify types explicitly.

Flow supports standard primitive types such as number or string, as well as custom types eg. for application-specific objects. It guards common bugs such as null dereferencing, silent type conversions and many more potential sources of bugs. An example of how flow-typed code can look like, see listing

[2] https://flowtype.org/

4.2. In this example, Flow would report that the annotated return type of `string` is incompatible with the return type of the `length` function, which is a `number`. The listing also shows how to enable Flow checking for a JavaScript module — this is done by putting `@flow` in a comment at the top of the file.

Listing 4.2 Flow-annotated JavaScript code

```
1 // @flow
2 function foo(s): string {
3   return s.length;
4 }
5 foo('Hello, world!');
```

I have used Flow extensively throughout the project and annotated the code regularly, because apart from the type checking, the annotations work very well as documentation for the developer and also for an IDE which can offer a better autocomplete. One of the features of Flow I found very useful are maybe types which are denoted by a question mark (eg. `?string`). When accessing a function or property on an object which is of maybe type, Flow will issue a warning that `Property cannot be accessed on possibly null or undefined value`. This greatly helps avoiding the "Undefined is not an object / function" error which is one of the most common ones in JavaScript development. Other handy features I have used include interfaces or guarding against a function receiving too few or too many parameters. My estimate, based on running `git grep` for several of the most common types, is that there are more than 700 Flow annotations placed in the app's files.

4.7.2 ESLint

ESLint [3] 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

[3] http://eslint.org/

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 [20] 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 JavaScript has its benefits - most importantly it allows for complete control of the navbar, the animations and gestures, and the ability to run the same code on multiple platforms [20]. These were the main reasons why Facebook decided to favor the Navigator [20].

Some would say that not using the native navigation breaks the promise of React Native - that is to be able to create apps that are indistinguishable from the native ones. This is why Wix (an Israeli mobile and web development company) is working on a native navigation for react native. At the time of

writing, there is not a stable release of this package available which is why I didn't use it.

Over the course of time new issues with using Navigator emerged and there was a need to come up with a better way of managing the navigation state. Among other issues, the original Navigator becomes hard to work with in case we use several instances of it. For example, the application I have developed uses three different navigation components. One StackNavigation is used as the root navigator. The root navigator contains another StackNavigation where the vast majority of the app's content is pushed. In some places, however, I wanted to display modal windows. The modals are pushed onto the master navigator so that they are displayed in the foreground. The modal window contains another StackNavigation of its own, which yields a total of three navigators, but for applications that use a tab-based navigation or drawer navigation, that number is likely to be higher.

At the time when I started working on the project, the new solution to navigation was already present in the React Native core and was named NavigationExperimental. However, since NavigationExperimental is only a set of low-level components, people started writing libraries around them to provide a more feature-rich experience for the developers. In the end, I have settled with a library called ex-navigation. It offers some needed functionality out of the box and allows to customize the behavior for iOS and Android such as handling the navigation bar, animations and back gesture on iOS as well as the back button handling on Android. Most importantly it works well with several navigators which can be managed independently and offers better animation performance. Since navigation is one of the longer-term major problems present in React Native, the React Native team also addressed the issue and a new navigation solution is scheduled for beta release in January 2017. This solution is based on ex-navigation so I should not have problems migrating to it once it becomes stable.

4.9 Issues

There were a number of issues encountered throughout the development, caused by different factors such as React Native's immaturity and frequent releases where new 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 include:

- react-native-keychain
- mobx-utils
- react-native-scrollable-tab-view
- ex-navigation

- react-native-router-flux
- serializr
- mobx
- react-native
- react-native-android-kit
- react-native-radio-button-android
- react-native-android-checkbox

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 Memsource 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 actively developed by Facebook and is open source.

It offers essential functionality similar to other popular Javascript test runners (e.g. AVA or Mocha), such as making assertions upon the results of tested code, creating mocks and also offers snapshot testing, which is a React-specific feature for testing the structure of React components without directly rendering them. Snapshot testing is a very useful feature especially in React Native as it allows to test component appearance without the need for rendering the UI on a device or emulator.

Jest creates a snapshot that captures the necessary information for component rendering. When the component changes, the snapshot changes as well, and we're notified of this fact during testing and also by version control when the change is being merged since the snapshot files live alongside the code.

Snapshot testing currently has the drawback of not being able to trigger and capture possible changes in the inner state of the component (if there is any state), ie. snapshot testing only considers the component's props. This, however, is a subject to change in one of the future releases of Jest, which is being developed at a quick pace.

Since a React Native app is a native application, we can use the same testing frameworks that we would use for testing any other native app on ios or Android.

todo popsat kolik jich ej jak jsem je pouzil atd - vystupy

[1] http://facebook.github.io/jest/

5.2 Testing With Users

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 binding for React Native is called Fabric and is provided by Twitter free of charge for both ios and Android. I have incorporated this service into both ios and Android versions of the app.

While it does report the application crashes, I am not completely happy with how the reported issues are presented in the Fabric 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 place and crashlytics considers all these bugs to be a single bug. This makes working with crashlytics uncomfortable. I will continue investigating further possibilities in this area.

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 there was a desire to change it. In the scope of this thesis 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 such as search or multi-user support.

React Native, despite its young age proved to be a valuable and functional tool for multiplatform mobile app development. Also MobX is a library that works very well with React and I have enjoyed working with.

The future work will involve further development of the application within Memsource. There is a severe bug in the Android version of the app caused by a bug in React Native core. Rather than waiting for the bug to be fixed in React Native, I have decided for a workaround (to replace the Android navigation bar component) which will also have the pleasant side effect of improving navigation bar code structure in the application. The downside is that the app will not be using the native navigation bar provided by Android. I will, of course, continue adding more features to the application, and improve the styling of some components. Other important future step is to introduce a continuous integration tool.

Appendix A

Memsource mobile app walkthrough

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

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 7pm
- Workflow steps: select translation and revision

Verify the created project and try to list jobs of the project. The list should be empty at this time.

Now let's create a new job in the project. The job should be created from a file named "important document.doc" which is present in the device's Google Drive found under "mobile app/testfiles". Keep the job's target languages se to "cs" and "de". Set the job's due date to 11th of January, 10 am and select the "VNovak23" user as the linguist for both languages. Make sure that comments and hidden text from the word document are imported and then create the job.

Wait for the job to be imported and then list the jobs and switch to the "Revision" workflow step. Select the job part whose target language is German and change its due date to 17.1.2017 11am and save. Verify the due date is updated.

The next task is to add existing translation memories to the project. If you list the translation memories, you will see that none are assigned yet. Attach the "Software TM" translation memory to the German target language. Use the filter if you're having trouble finding it. Attach it with read and write modes enabled, a penalty of 10

The next step is to add term bases. Attach the "Clock Industry" term base with read, write and QA enabled. Also add the "jim" term base in read mode.

After the term bases are attached, go back to the project list screen and try to use the search feature to find the project you just created. Verify it shows up in the search results and that you can open it.

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

Appendix B

Index

T

Appendix C

Bibliography

[1] Dan Abramov. *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/FileManagement/Conceptual/DocumentPickerProgrammin AccessingDocuments/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] *Bridging in React Native*. 2015. URL: https://tadeuzagallo.com/blog/react-native-bridge/ (visited on 01/03/2017).

[6] Lee Byron. *React Conf 2015 - Immutable Data and React*. 2015. URL: https://www.youtube.com/watch?v=I7IdS-PbEgI.

[7] Tom Coleman. *The State of JavaScript 2016 - State Management*. 2016. URL: http://stateofjs.com/2016/statemanagement/ (visited on 01/04/2017).

[8] Jerry Dischler. *Building for the next moment*. 2015. URL: http://adwords.blogspot.com/2015/05/building-for-next-moment.html.

[9] 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.

[10] 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).

[11] 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).

[12] IDC Inc. *Smartphone OS Market Share, 2016 Q3*. URL: http://www.idc.com/promo/smartphone-market-share/os (visited on 01/04/2017).

[13] Kantar Inc. *Smartphone OS sales market share*. 2016. URL: http://www.kantarworldpanel.com/global/smartphone-os-market-share/ (visited on 06/01/2016).

[14] 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/.

[15] Ecma International. *Map Objects*. 2015. URL: http://www.ecma-international.org/ecma-262/6.0/#sec-map-objects (visited on 01/05/2017).

[16] *Introducing Hot Reloading*. 2016. URL: https://facebook.github.io/react-native/blog/2016/03/24/introducing-hot-reloading.html (visited on 03/24/2016).

[17] *JavaScript Environment*. 2016. URL: https://facebook.github.io/react-native/docs/javascript-environment.html (visited on 10/13/2016).

[18] L. Masinter. *Returning Values from Forms: multipart/form-data*. 1998. URL: http://www.ietf.org/rfc/rfc2388.txt (visited on 11/06/2016).

[19] *Native Modules*. 2016. URL: https://facebook.github.io/react-native/docs/native-modules-ios.html (visited on 11/03/2016).

[20] 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).

[21] *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).

[22] Michel Weststrate. *MobX - Simple, scalable state management*. 2016. URL: http://mobxjs.github.io/mobx/.

[23] *Xamarin Forms*. 2016. URL: https://www.xamarin.com/forms/ (visited on 04/13/2016).

[24] *Xamarin Platform*. 2016. URL: https://www.xamarin.com/platform (visited on 04/13/2016).

[25] *Xamarin.com*. 2016. URL: https://www.xamarin.com (visited on 06/02/2016).