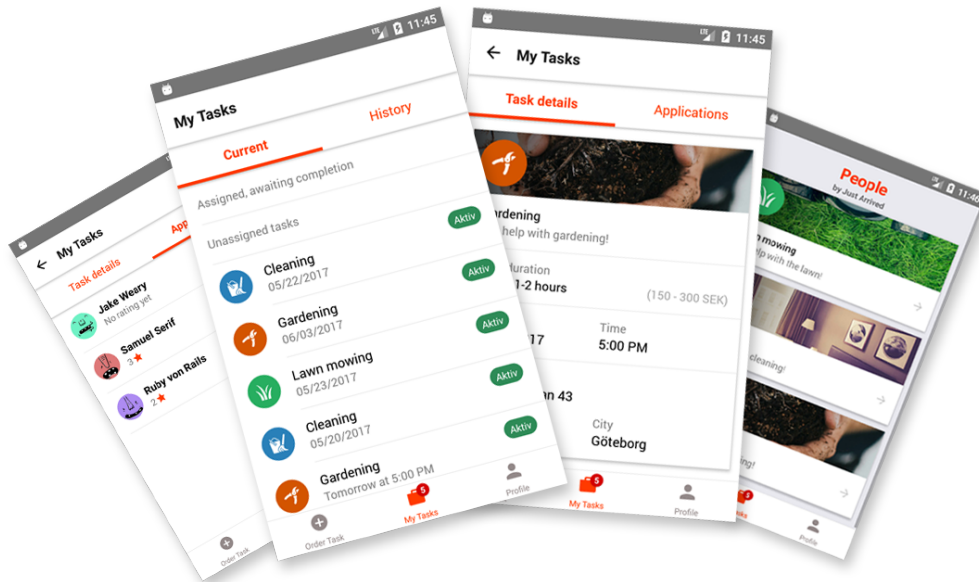




CHALMERS
UNIVERSITY OF TECHNOLOGY



Developing a Cross-Platform Service Ordering Mobile Application for Social Integration

Bachelor of Science Thesis in Computer Science and Engineering

Emina Cindrak
Anton Ingvarsson
Carl Jansson
Bassel Kanaan
Christoffer Karlsson
Martin Sigvardsson

BACHELOR OF SCIENCE THESIS

Developing a Cross-Platform Service Ordering Mobile Application for Social Integration

Emina Cindrak
Anton Ingvarsson
Carl Jansson
Bassel Kanaan
Christoffer Karlsson
Martin Sigvardsson



Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG

Göteborg, Sweden 2017

Developing a Cross-Platform Service Ordering Mobile Application for Social Integration

Emina Cindrak
Anton Ingvarsson
Carl Jansson
Bassel Kanaan
Christoffer Karlsson
Martin Sigvardsson

© Emina Cindrak, Anton Ingvarsson, Carl Jansson, Bassel Kanaan, Christoffer Karlsson, Martin Sigvardsson, 2017

Examiner: Morten Fjeld, Professor, Department of Applied IT

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
SE-412 96 Göteborg
Sweden
Telephone: +46 (0)31-772 1000

The Author grants to Chalmers University of Technology and University of Gothenburg the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet. The Author warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he/she has obtained any necessary permission from this third party to let Chalmers University of Technology and University of Gothenburg store the Work electronically and make it accessible on the Internet.

Cover: Screenshots from the client part of the application.

Department of Computer Science and Engineering
Göteborg 2017

Developing a Cross-Platform Service Ordering Mobile Application for Social Integration

Emina Cindrak
Anton Ingvarsson
Carl Jansson
Bassel Kanaan
Christoffer Karlsson
Martin Sigvardsson

*Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg*

Bachelor of Science Thesis

Abstract

The aim of this project was to design and develop a cross-platform service ordering mobile application for Android and iOS. The application should enable clients to order services, and in turn, provide an easy way for newly arrived immigrants to find suitable tasks to perform, in order to enter the Swedish labour market. The application was developed on behalf of the company Just Arrived.

The construction and development was focused on implementing the client part of the application and designing a visual prototype for the worker part. The client part enables clients to order simple services, such as lawn mowing or cleaning. These tasks are then published in the worker part of the application, used by newly arrived immigrants, to easily be found and applied for.

This thesis describes the methodologies used, the theory behind the development and the design, the challenges confronted during the development process, and the final result of this project.

Keywords: React Native, i18n, Internationalization, JavaScript, Cross-Platform, Android, iOS

Sammanfattning

Syftet med projektet var att designa och utveckla en plattformsoberoende mobilapplikation för Android och iOS. Mobilapplikationen ska erbjuda kunder möjligheten att beställa tjänster, men även erbjuda ett enkelt sätt för nyanlända immigranter att hitta jobb att utföra, för att kunna ta sig in på den svenska arbetsmarknaden. Applikationen utvecklades för, och i samarbete med, företaget Just Arrived.

Uppbyggnaden och utvecklingsarbetet var fokuserat på att utveckla beställardelen av applikationen, samt att designa en visuell prototyp utan funktionallitet för arbetstagare. Beställardelen av mobilapplikationen gör det möjligt för kunder att beställa enklare tjänster, såsom städning eller gräsklippning. Tjänsterna publiceras sedan till arbetstagare, för att de enkelt ska kunna hitta och ansöka till uppdrag.

Denna projektrapport beskriver de metoder som använts, teorin bakom utvecklingen och designarbetet, de utmaningar som uppkommit under utvecklingsprocessen, samt det slutgiltiga resultatet av projektet.

Keywords: React Native, i18n, Internationalization, JavaScript, Cross-Platform, Android, iOS

Vocabulary

- Actions - Redux methods handled by reducers
- Android - Mobile phone operating system
- API - Application Programming Interface
- Back end - Code handling data displayed in the front end
- Cross-platform software - Software working across multiple platforms from a single codebase
- ESLint - Linting utility for JavaScript
- Fetch API - JavaScript interface for fetching resources
- FIFO - First In, First Out
- Git - Version control system
- GitHub - Web-based Git repository hosting service
- GUI - Graphical User Interface
- HTTP - Hypertext Transfer Protocol
- i18n - Internationalization (used to support multiple languages)
- iOS - Mobile phone operating system
- Java - Programming language used by Android apps
- JavaScript - Programming language (used by React Native applications)
- JSON - JavaScript Object Notation, used to store and retrieve information
- Just Arrived - The company that proposed this thesis project
- Kanban - Method for managing work (used within agile software development)
- Linting - Static code analysis to detect potential programmatic and stylistic errors
- MVC - Model View Controller
- npm - Node Package Manager
- Objective-C - Programming language (used by iOS applications)
- P2P - Peer-to-peer
- React - A JavaScript library for building user interfaces
- React Native - Framework for building native applications using React
- Reducer - Redux state storage and action handler
- Redux - Predictable state container for JavaScript applications
- REST - Representational state transfer, a type of API
- Swift - Programming language (used by iOS applications)
- URL - Uniform Resource Locator
- UX - User Experience

Project Specific Definitions

- Client - Customer ordering a service using Just Arrived
- Worker - Newly arrived immigrant working through Just Arrived

Contents

List of Figures	xii
1 Introduction	1
1.1 Purpose	1
1.2 Background	1
1.3 Challenges	2
1.3.1 Graphical User Interface	2
1.3.2 Target Group and User Testing	2
1.3.3 Language Barrier	2
1.4 Scope	3
1.5 Outline	3
2 Methods	4
2.1 Agile Software Development	4
2.1.1 Kanban	4
2.1.2 Lean UX	5
2.2 Interaction Design Methods	5
2.2.1 Paper Sketching	5
2.2.2 Wireframing	6
2.3 Testing	6
2.3.1 Expert Testing	6
2.3.2 User Testing	6
3 Tools and Theory	7
3.1 Interaction Design	7
3.1.1 Designing for Touch	7
3.2 Frameworks	8
3.2.1 React Native	8
3.2.2 Redux	8
3.2.3 React Navigation	10
3.2.4 NativeBase	10
3.2.5 Internationalization i18n	11
3.2.6 React Native Fetch	12
3.3 Just Arrived API	13
4 Process	14
4.1 Pre-study	14

4.1.1	Requirements from Just Arrived	14
4.1.2	Functionality and Use Cases	15
4.1.3	Brief Market Analysis	16
4.2	Interaction Design Process	17
4.2.1	Low Fidelity Sketching	17
4.2.2	Combining and Refining Sketches	18
4.2.3	Digital Wireframing	18
4.3	Design Testing	19
4.3.1	Expert Tests	20
4.3.2	User Tests	20
4.3.3	Client Wireframe Tests	21
4.3.4	Worker Wireframe Tests	21
4.4	Programming Development Process	22
4.4.1	Programming	22
4.4.2	Pull Request	23
4.4.3	Code Review	23
4.4.4	Application Testing	23
5	Results	25
5.1	Application Flow	25
5.1.1	Task Ordering - <i>Client</i>	25
5.1.2	Find Tasks - <i>Worker</i>	26
5.2	Design	27
5.2.1	Usability	27
5.2.2	Language Selection	28
5.2.3	Design for the Client Part of the Application	28
5.2.4	Worker Prototype Design	33
5.2.5	Shared Design	34
5.3	System Architecture	37
5.3.1	Project Folder Structure	37
5.3.2	Application Navigation	38
5.3.3	Redux Integration	39
5.3.4	Networking	40
5.3.5	Networking and Redux	40
6	Discussion	41
6.1	Limitations	41
6.1.1	The Language Barrier	42
6.2	Retrospective	42
6.3	Usability Evaluation	43
6.4	Design Decisions	44
6.4.1	Login Screen Placement	44
6.4.2	Navigation Menu Choice	45
6.4.3	Unifying the Design for All Users	45
6.5	Ethical Choices	46
6.5.1	Worker Selection	46
6.5.2	Rating	46

Contents

6.6	Framework Evaluation	46
6.6.1	React Native	47
6.6.2	Networking Component	47
6.6.3	Redux	48
6.6.4	NativeBase	48
6.6.5	React Navigation	49
7	Conclusion	50
	Bibliography	51
A	All Views of the Application	I

List of Figures

2.1	An example of a kanban board using post-it notes.	5
3.1	Simplified overview of frameworks used in the project and their hierarchy. Dotted lines indicate communication, i.e. function calls.	8
3.2	Illustrative comparison of component communication without and with Redux.	9
4.1	Screenshots from TaskRabbit, showing the <i>Home</i> tab and the <i>Tasks</i> tab.	16
4.2	Flowchart visualising the programming development process.	17
4.3	Example of low-fidelity paper sketches during development.	18
4.4	Early iteration of combined and refined whiteboard sketches, created together as a group.	18
4.5	Early iteration of wireframes for the client part of the application, showing the list of owned tasks, as well as the option to choose a task type during task ordering.	19
4.6	Printed wireframes used during testing.	20
4.7	Flowchart visualising the programming development process.	22
5.1	Interaction flow when ordering a task.	25
5.2	Interaction flow when choosing a worker.	26
5.3	Interaction flow when confirming that a task has been completed.	26
5.4	Interaction flow when finding and applying for a task.	26
5.5	Interaction flow when receiving a notification about being assigned the task.	27
5.6	Interaction flow when confirming that a task has been completed.	27
5.7	Screenshots from the client part of the application, showing screens involved when ordering a task.	29
5.8	Screenshots from the client part of the application, showing screens involved when ordering a task.	29
5.9	Card with image header displaying the lawn mowing task.	30
5.10	Screenshots from the client part of the application, showing screens involved when choosing a worker.	31
5.11	Screenshots from the client part of the application, showing screens involved when paying for a task.	31
5.12	Screenshots from the client part of the application, showing screens involved when marking a task as completed.	32

5.13	Screenshots from the client part of the application, showing screens involved when rating the worker.	32
5.14	Wireframes from the worker part of the application, showing screens involved when finding tasks.	33
5.15	The notification received by the worker when accepted for a task. . .	34
5.16	Confirm task completed and rate experience.	34
5.17	The introduction wizard.	35
5.18	Screenshot of the shared login screen.	35
5.19	Main navigational tab bar shown throughout the application.	36
5.20	Task icons representing the different tasks in the application.	37
5.21	Tree data structure visualising an overview of the folder structure. . .	37
5.22	Visualisation of how the views are grouped in tabs, and the navigation between them. Black arrow head indicates the ability to navigate, and white arrow head indicates the ability to navigate backwards (i.e. by using the return button). The grey section with dotted border indicates the main content of the application, requiring the user to be logged in. Each tab of the navigation tab bar is indicated by a package with screens in them: <i>CreateJobTab</i> , <i>MyJobsTab</i> , and <i>MyProfileTab</i> . While inside the main part of the application, navigation is always possible between tabs.	39
5.23	Overview of GUI, Redux, networking, and how they are connected. .	40
A.1	Login screen and choose task type screen.	I
A.2	Create and inspect task information.	II
A.3	View owned tasks and receive a push notification.	II
A.4	Views listing and displaying applicant information.	III
A.5	Views displaying applicant references and listing credit cards.	III
A.6	Payment confirmation and payment success screens.	IV
A.7	Task completion notification and confirmation screen.	IV
A.8	Rate worker and task completed screens.	V
A.9	User profile and account creation screens.	VI

1

Introduction

This chapter will give an introduction to the project as well as provide background information and defining the purpose of the work that has been done.

1.1 Purpose

The purpose of this bachelor's thesis was to, together with Just Arrived, develop a mobile application where clients can order services to be performed by newly arrived immigrants and they, in turn, can find suitable tasks. The aim of the application is to help newly arrived immigrants enter and become a part of the Swedish labour market. Furthermore, the project focused on how a modern interface should be designed to overcome language barriers on touch devices.

1.2 Background

More and more people are fleeing their home countries to Europe, and in 2015 over 160 000 people were in search of asylum in Sweden [1]. This number dropped drastically in 2016, after severe border controls were introduced, but the integration of those who come to Sweden is still a relevant question. The company Just Arrived consists of 150 volunteers [2] who have been working for two years on reaching their goal - to make it easier for newly arrived immigrants in Sweden. They believe that access to the labour market is one of the main barriers that must be overcome to create better and safer living conditions for all affected.

Just Arrived's concept is to match clients, who have tasks they want to be performed, with newly arrived, who are in need of work and experience. These tasks can be anything from lawn mowing to engineering services.

To further facilitate the process and increase the amount of clients, Just Arrived reached out to Chalmers University of Technology for help with developing a mobile application for the company. This application was aimed at helping potential clients to order services. The project also included designing a prototype for the worker part of the application.

The application had to be designed in such a way that it helps users overcome language barriers in a visually appealing and efficient way. This was the main interest of the project, since many potential users neither understand Swedish nor English.

1.3 Challenges

The project, which might be perceived as quite extensive, was split up into several more manageable sub-problems. The combination of solutions to these sub-problems together formed the finished product.

1.3.1 Graphical User Interface

There were a number of interesting challenges related to developing the design of the application. Several questions had to be answered, including, *How can a user interface be optimised for touch devices?* and *How does culture and background impact our perception of interfaces?*

There were also many challenges related to different languages, and how they affect the design. As the worker part of the application was aimed at a wide audience, with people from different cultures and countries, it was vital that the application is multilingual and provides the user with a way of choosing different languages. In this project we needed to explore many questions within the field of graphical user interface (GUI) design. A few of them were which technical approach would be preferable for supporting multiple languages, and how a right-to-left script could affect components in the graphical user interface.

As a side note, the theme of the application could follow the current style guide of Just Arrived, with their brand colours and so forth. This was however not a requirement by Just Arrived.

1.3.2 Target Group and User Testing

Through Just Arrived's wide network of contacts, we were provided with access to people within the target group for the worker part of the application. We were also provided with a description of the target group for the client part of the application, so we could reach out to people within that group and perform user tests on them.

There were various challenges related to user testing - what should be tested, in which way, and how should the tests be evaluated? Would communication between the test subject and the tester be an issue, as they might not have a language in common?

1.3.3 Language Barrier

Since the clients and workers might not share a language, a big challenge consisted of designing and implementing GUI that is easily understandable by both types of users, and aid in their communication with each other.

A challenge faced during development was to guarantee that the client would feel secure in that the published task would be understood by the worker. Another challenge was to clearly communicate to the worker what task is expected to be performed, and details about the particular task.

1.4 Scope

Just Arrived prioritised the development of the client part of the application. This was due to the fact that Just Arrived had more workers than tasks available on their web page. Therefore, the development for the clients was always the first objective during the implementation.

Another goal was to let the application handle multiple languages. But due to internal language knowledge this functionality was decided to be limited to Swedish, English, and maybe Arabic if time constraints were to allow for it. Eventual users of the application are assumed to have some basic understanding of one of the available languages and some experience using touch screens.

Implementation of a payment system is out of scope for this project. The application should however provide a GUI for credit card payments, that might be utilised in the future, but back end functionality is not necessary for this project.

The academic interest of the project was focused on GUI and usability, hence front end development was prioritised over back end functionality. Usability and application flow was prioritised over designing a pixel perfect user interface, unless these parts negatively impacted the usability.

1.5 Outline

The next chapter, chapter 2, explains the implementation methods used during the project. Thereafter, chapter 3 introduces the theory behind the tools, frameworks, libraries, and evaluation techniques used. Chapter 4 describes the realisation of the application while chapter 5 contains the results produced. In chapter 6 a discussion regarding various parts of the project is conducted, whereat a conclusion is formed in chapter 7.

2

Methods

Various methods that were used during the project will be covered in this chapter, including methods related to agile software development and methods for testing.

2.1 Agile Software Development

Agile software development, is a widely used development methodology, where work is done iteratively [3]. Key concepts that define the agile process is the iterative way of working, where software is developed and delivered continuously, and constantly improved in a evolutionary way. As the name suggests, teams following the agile process must be ready to act with agility, and be flexible and responsive when it comes to making changes. Work is done in close collaboration with the customer, to make sure both the customer and developers have a mutual idea of where development is heading. To succeed, agile methodology states the following [4]:

- **Individuals and Interactions** over processes and tools
- **Working Software** over comprehensive documentation
- **Customer Collaboration** over contract negotiation
- **Responding to Change** over following a plan

2.1.1 Kanban

Kanban [5] is an agile process-management system, originally used to schedule work in lean manufacturing. The name comes from the Japanese word *kanban*, which roughly translates to billboard. While the system is widely used in industry, it has lately also gained a lot of popularity and recognition within software development [6]. When working with kanban, scheduling is done and visualised on a kanban board. The most basic and traditional starting point of a kanban board consists of three columns: *To Do*, *Doing*, and *Done*. The kanban board is later filled with cards, each corresponding to a task.

Traditionally kanban boards were physical, with post-it notes commonly used as cards. Today there are more alternatives, specially within the software development community. One common web-based service is called Trello [8], which provides an online tool for project management following the kanban paradigm.



Figure 2.1: An example of a kanban board using post-it notes.

[7]

2.1.2 Lean UX

Lean UX [9] is another design process methodology, inspired by The Lean Startup [10] and Agile UX [11]. In Lean UX, there is more focus on producing the right thing, that is, what the customer wants. The biggest waste is designing something that the customer does not want. Work in Lean UX is hypothesis driven, where a design idea is quickly turned into a working prototype, often of quite low fidelity. The prototypes are tested on customers, and later evaluated and improved in an iterative approach as inspired by Agile UX.

Lean UX, as described by Gothelf (2013):

"Lean UX is the practice of bringing the true nature of a product to light faster, in a collaborative, cross-functional way that reduces the emphasis on thorough documentation while increasing the focus on building a shared understanding of the actual product experience being designed." [9]

2.2 Interaction Design Methods

There are several different methods used to draft testable prototypes before spending valuable time to implement them. Here is a listing and short description of the different methods used during the project.

2.2.1 Paper Sketching

Paper sketching or paper prototyping [12] is a widely used method to quickly visualise ideas and put them to test. It is often used as a way to test different design ideas and improve them through many quick iterations [13].

2.2.2 Wireframing

Wireframes [14] are used to illustrate a screen's possible interface. Wireframes can differ in level of detail and in whether they are made digitally or not, but wireframes are typically more detailed than paper sketches, and should be a somewhat accurate depiction of how the final interface could be implemented.

2.3 Testing

Usability Testing is a method used to observe an individual's experience with an application, as the individual goes through several given tasks [15]. *Think Aloud Testing* is a method that encourages individuals to verbalise their thoughts and actions during the test [15].

2.3.1 Expert Testing

An expert test, also known as *heuristic assessment* [16], is a method in which an expert tests an interface against generally accepted usability principles. An expert is expected to already be familiar with proper heuristics, which makes the test format rather informal since no general instructions should be necessary.

2.3.2 User Testing

User testing is a method that is used to evaluate a certain product, by testing the product on potential end users of the product. The main purpose of the user testing method is examining how the product works in practice. User testing often combines other methods used for testing, such as *Usability Testing*, and *Think Aloud Testing*.

3

Tools and Theory

Different tools and techniques have been used throughout the project. This chapter provides knowledge and contextual information about the theory of those tools and principles.

3.1 Interaction Design

Designing a mobile application involves a lot more than just creating a visually appealing design. Creating an efficient navigation, simplifying user interaction, and minimising so called *excise*. Excise being the extra work the user do that do not directly contribute towards the user goal. These are just a few of many other important concepts involved in application design. Today's mobile phone applications often include numerous views, offering a wide variety of user interaction. This makes the behaviour of the application, as well as the experience for the user, possibly more important than ever before. Within the field of interaction design, the *behaviour* of a product is in focus, rather than the visuals [17].

3.1.1 Designing for Touch

When designing for mobile devices, and touch devices in particular, there are a number of things to keep in mind. Perhaps the most noticeable difference in comparison to other devices, is the drastically smaller screen size. This naturally makes it harder, or even impossible, to fit all content that would normally be displayed on a desktop screen, for example.

In general, mobile applications are transient [18], meaning that they are used for a relatively short amount of time, where user interaction is brief, and the functionality often is focused on a small but well-defined set of tasks.

Even though technology is advancing, with the resolution of touchscreens increasing, the human anatomy stays the same. The human finger is clumsy and inaccurate in comparison to a digital mouse pointer, which can achieve accuracy to hit a single pixel on a screen [19]. A common approach to achieving an accurate touch interface is to make the onscreen objects large enough so that they can be triggered easily with a finger [18].

Another problem the human finger introduces is that it blocks part of the user's vision when covering the display. In mouse-driven desktop interfaces we are used to receiving feedback from the interface upon hovering interactive components (e.g.

buttons). It is not possible to hover components in a touch interface, therefore it is more complicated to use traditional interface components such as tool-tips and other contextual hints [18]. The lack of hover state feedback persuades designers into communicating these hints in another way.

3.2 Frameworks

A number of frameworks and tools were used to create the application. This section describes the main functionality of these frameworks and tools, to provide contextual information about how and why they can be used.

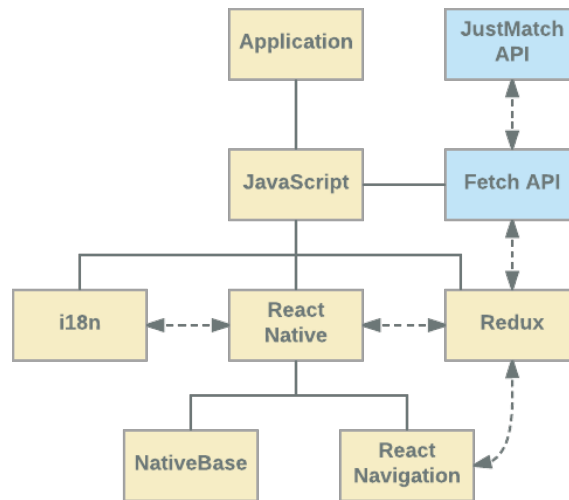


Figure 3.1: Simplified overview of frameworks used in the project and their hierarchy. Dotted lines indicate communication, i.e. function calls.

3.2.1 React Native

React was originally developed by Facebook as a framework for website development [20]. React is component-based, meaning independent components are created and combined into views [21]. This facilitates code refactoring and reuse, thus minimising the need to rewrite code.

Facebook later developed the framework React Native, based on JavaScript and React, to take the principles they successfully used in web development into mobile development [22]. React Native enables the programmer to create native cross-platform applications [23].

3.2.2 Redux

As applications are becoming more complex, the code must handle a lot of states. The states the applications need to keep track of include server responses, local data not yet sent to a server, cached data, GUI states, such as navigation routes.

Keeping track of constantly changing states is a difficult task, but if various components can manipulate each others states it will be extremely difficult to predict when, why, and how a state changes. It is hard for developers to find bugs and add new features, when it is difficult to predict the behaviour of the application.

Redux is a state container for JavaScript applications, attempting to solve this issue with three core principles [24]:

- **Single source of truth**

All states of the application are stored and made accessible from a single source. The states are no longer spread out through multiple components. This means changes to states are not passed between views or components, but sent to the store, and the components that are interested in a particular state can get the information about the state themselves.

- **State is read only**

The only way to change a state is to dispatch an action [25]. This guarantees that components cannot directly manipulate the state. Instead the action declares the intent to change the state. The actual change is handled centrally by the, so called, reducers. Changes made centrally are executed separately, thus avoiding obscure bugs that could occur if the components handled the manipulation of the state themselves.

- **Changes are made in pure reducer functions**

A pure reducer function specifies how a state is changed by actions. A reducer takes the previous state, an action, and then returns a new state depending on the action provided [26]. The state is immutable, therefore a new state (which is a manipulated copy of the previous state) is returned, instead of changing the old state.

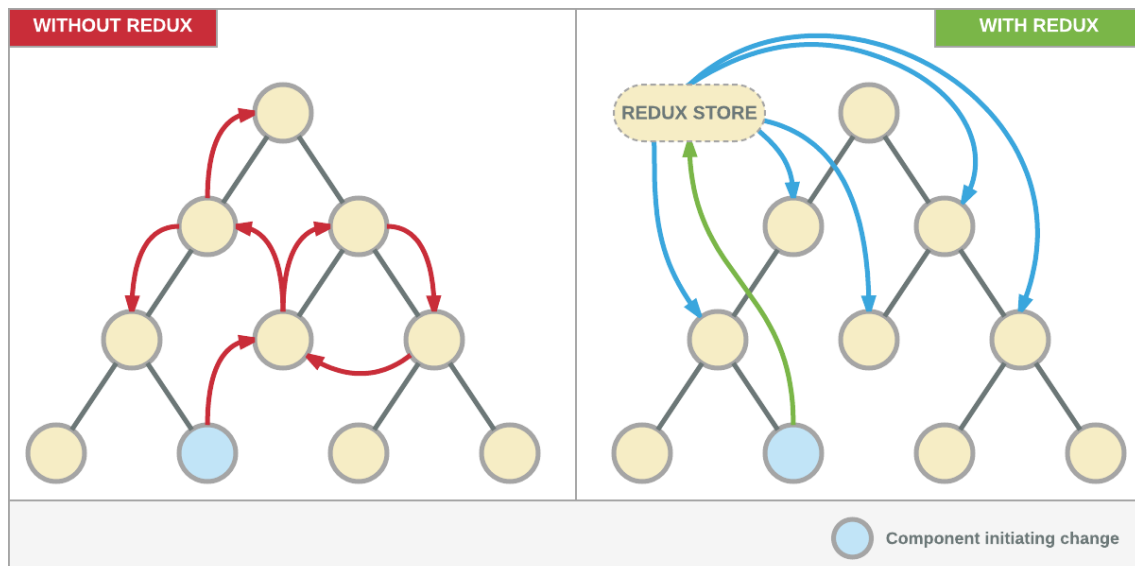


Figure 3.2: Illustrative comparison of component communication without and with Redux.

3.2.3 React Navigation

React Navigation is an external library for navigation in React Native [27]. It offers a set of navigators, which provides a way of defining the navigational structure of an application.

React Navigation has three different navigators [28]: `StackNavigator`, `DrawerNavigator`, and `TabNavigator`. Each view in a `StackNavigator` is placed on top of a stack. This is useful when the user wants to navigate backward through the history of views previously visited, i.e. by pressing the back button. The `TabNavigator` facilitates the creation of navigational tab bars, while the `DrawerNavigator` is used to create drawer menus. React Navigation also offers the possibility of nesting navigators. For example, an application with a tab menu and multiple views, will usually contain a `TabNavigator` with a `StackNavigator` for each tab. Each view in a navigator is identified by a unique key. This key can later be used in the code to programmatically navigate to a particular view.

This is a simple example showing how application navigation can be defined. In this example the application consists of a `TabNavigator` known as `BasicApp`, with two tabs, `MainScreen` (with key `Main`), and `SetupScreen` (with key `Setup`):

```
// Simple example of an application with two screens in a tab menu
const BasicApp = TabNavigator({
  Main: {screen: MainScreen},    //key: Main
  Setup: {screen: SetupScreen}, //key: Setup
});
```

Below is another simple example of how navigation from one view to another can be implemented. The example shows how to implement navigation to the setup screen by clicking on a button, which can be located in an arbitrary view. In the `onPress` method of the `Button`, a call to `navigate('Setup')` is done, which tells the application to display the view with key `Setup`, which in the example is `SetupScreen`:

```
// Simple navigation example
<Button title="Go to Setup Tab" onPress={() => navigate('Setup')} />
```

3.2.4 NativeBase

NativeBase [29] is an open source library for React Native, providing essential cross-platform GUI components. It is built as a layer on top of React Native and provides components with platform specific design through the same code.

Below is an example of how to add a button to an application, using React Native without external libraries, and using React Native with NativeBase. Using NativeBase with React Native often enables developers to build interfaces using much less code, as can be seen in the example below.

```
// React Native / Without NativeBase
var style = StyleSheet.create({
  button: {
```

```
    backgroundColor: '#99AAFF',
    borderRadius: 5,
    borderWidth: 1,
    borderColor: '#000033'
  }
});

<TouchableOpacity style={style.button}>
  <Text style={{color: 'white'}}>
    Click Me!
  </Text>
</TouchableOpacity>

// With NativeBase
<Button>
  <Text>Click Me!</Text>
</Button>
```

3.2.5 Internationalization i18n

Internationalization, or i18n [30] (where 18 is the amount of letters between the I and n [31]), is a term for adapting software to different languages, regional differences, and specific technical requirements of a particular market.

There are various i18n frameworks for different programming languages, e.g. i18n-js for React Native [32]. The i18n frameworks provide various functionality, where some of the most common are translation look-ups, interpolation, and date formatting.

The internationalization frameworks read strings from an i18n file, where translations are saved in as key-value pairs, usually in a JSON-formatted file. The following example shows a simple i18n file, containing translations for a greeting in English and Swedish:

```
// Simple i18n translation file example
I18n.translations = {
  en: {
    greeting: 'Hello!'
  },
  sv: {
    greeting: 'Hallå!'
  },
};
```

To access i18n translations, a function is called with the key of the key-value pair translation as argument.

```
// Translation lookup using i18n-js
I18n.t('greeting') // --> Hello (if English locale is selected)
```

The call to the lookup function returns a translation, that depends on the current locale (i.e. the chosen language in an application).

i18n frameworks also provide the functionality of interpolation, which means that variables can be passed to the translations.

```
// Interpolation example in i18n-js
I18n.translations = {
  en: {
    greeting: 'Hello {{name}}!'
  },
};
```

```
// Call to the translation lookup function using interpolation
I18n.t('greeting', { name: John }) // --> Hello John
```

i18n frameworks also offer the functionality of displaying localised dates, as the formatting of dates differs between different locales. An example of such a framework is `moment.js`[33], which offers functionality for handling and displaying dates appropriately for the current locale. It also offers the possibility of displaying dates in a humanized format (e.g. Last Monday at 2:30 AM).

3.2.6 React Native Fetch

React Native implements the Fetch API to provide a simple and easy to use interface for communication with network resources [34]. In order to retrieve a resource from a publicly available uniform resource locator (URL), one can use the global method `fetch` and retrieve the resource using for example `fetch('justarrived.se')`. For more complicated requests, the `fetch` method can also be passed an object, containing the hypertext transfer protocol (HTTP) method, various headers, and body, e.g. `fetch('justarrived.se', {method: 'POST'})`.

The global `fetch` method returns a JavaScript promise, representing a response stream that might be available at some point in time [35]. In order to handle the data, once provided by the promise, method calls can be chained as they are needed. Each method chained after a promise is also considered to be a promise. At the end of a promise method chain, it is also recommended to use some way to ensure no open promises are left behind as memory leaks. [36].

```
// Example fetching tasks and handling promises
fetch('https://api.justarrived.se/api/v1/jobs/')
  .then((response) => {
    if (response.status === 200) {
      // If status code is 200 convert stream to JSON
      return response.json();
    }
    throw new Error('Response was not 200 ok');
  })
  .then((responseJson) => {
```

```
// Successfully fetched JSON response
console.log(responseJson);
})
.catch((error) => {
  // There was an error
  console.warn(error);
})
.done();
```

3.3 Just Arrived API

Just Arrived provides an open source API named JustMatch API [37]. This API was created with their web application in mind, where the newly arrived can apply for jobs advertised by companies. In order for the JustMatch API to be usable with the mobile application, functionality is being implemented to allow regular users to post and manage tasks, in addition to the previously exclusive companies.

The API is a JSON representational state transfer (REST) API that follows the JSON API 1.0 standard [38]. This means that the API, by default, returns JSON objects from relatively simple URL requests.

The main difference from regular HTTP requests are some of the headers. The Content-Type header, is defined as `application/vnd.api+json`, which differs from the usual `application/json`, by informing the client that the content provided is vendor specified and can be parsed as JSON. Other headers are

X-API-KEY-TRANSFORM: `underscore`, which ensures that all response data use underscore as the separator, and

Authorization: `Token token=XXXXYYZZZ`, which is how authentication is included in requests [39].

As specified in the JSON API 1.0 standard, the API response also provides links to further resources that might be of interest based on the request. If the API returns a subset of the total result, and pagination is necessary to retrieve additional resources, the response should contain links to what was actually fetched, what should be fetched next, and the last page that is possible to fetch. If the response contains a variable that points towards some other endpoint with an ID, then that URL should also be included in the response. It is also possible to add the `include` variable in requests, in order to fetch additional resources directly without doing additional request, thus minimising the amount of requests necessary to fetch additional data not part of the standard response [38].

4

Process

This chapter covers the process involved in realising the project, beginning with describing how the pre-study was conducted, followed by the interaction design process, and ending with the programming development process.

4.1 Pre-study

Pre-studies were made for various factors of the application, but the initial part was to establish Just Arrived's expectations. This was done by frequent meetings with Just Arrived, and by inspecting their already existing design guidelines and web application, which gave us guidance and a clearer understanding of what the application we were developing should offer.

Other existing and similar mobile applications that are on the market were also inspected, such as TaskRabbit [40], Freelancer [41], and Welcome [42]. These provided a greater understanding of what the mobile application should be capable of. The digital service platform TaskRabbit [40] gave us insight in how similar problems to ours have been solved, such as methods for scheduling and rating.

The main goal of the pre-study was to define and conclude the ideal scope of the project, and to gather fundamental information and requirements needed for the development of the application.

React Native was recommended by Just Arrived as a means to ease cross-platform development. The group initially had limited JavaScript experience, and no one had prior knowledge of React Native. During the pre-study phase of the project, React Native was evaluated by us, by individually completing various tutorials online, after which the framework was approved by common consensus.

4.1.1 Requirements from Just Arrived

The company Just Arrived specified some expected functionality for the client part of the application. These were that a client should be able to order and pay for tasks, and provide feedback on the worker who performed the task.

The company specified additional functionality that could be developed if time would permit, such as subscribing to a task and ordering a task to be performed a certain date. When subscribing to a task, the client is given the possibility to arrange so the task is ordered regularly. When ordering a task, the client should be able to choose a certain date for the task to be performed.

Other design specifications Just Arrived had for the client part of the application, were uncomplicated solutions to the functionality mentioned above, and an intuitive design. Uncomplicated solutions imply fewer steps before accomplishing a goal with the help of the functionality that the application provides. Intuitive design implies a self-explanatory application, where the user can use the application independently and efficiently. Altogether, the functionality provided by the application should be easily accessible, rather than time-consuming.

Aside from the company's expectations, the group elaborated additional functionality that improved the application. This functionality, together with the company's expectations, is covered in the next chapter.

4.1.2 Functionality and Use Cases

A functionality is a property of an application, while a use case is a goal that can be accomplished by using the functionality. Initially, the functionality of the application was elaborated by following the expectations from the company. Additional functionality was defined by the group, some of it by looking into other existing applications, such as the option to sign in.

Shared Functionality

- Sign up, sign in, and sign out
- View and edit user profile
- Support for multiple languages
- Push notifications through application, email, or phone
- View for current tasks
- View for history of tasks
- Application wizard
- Map view for locations
- Confirm that task is completed

Client Functionality

- Easy creation and management of tasks
- Rate worker

Worker Functionality

- Filter and sort available tasks
- Apply for tasks
- Rate client

Use Cases of the Shared Functionality

The first shared functionality is the ability to sign up, sign in, and log out. The user should also have the ability to view and edit the user profile, providing vital and necessary information about oneself. The application should provide support for multiple languages, where the user can choose which language to use. The application should also offer the functionality of sending reminders and messages to the user in terms of push notifications, emails, or text messages. When in need of finding an address or tasks at a specific location, the application should offer a map view for locations to be found. A view for displaying current tasks, and a history of

tasks, should also be implemented, where the client can review all created, pending and completed task, and the worker can review all ongoing and performed tasks.

Use Cases of the Client Functionality

The client can easily order a task and manage it, by choosing date and time, or even creating a repeatable subscription event. The client can choose a suitable worker after receiving applicants. After the task is done, the client can confirm that the task has been done, and then rate the worker. These use cases are characteristic for the client part of the application.

Use Cases of the Worker Functionality

The worker has the ability to filter and sort available tasks, in order to find relevant tasks in an efficient way. The worker can apply for several tasks, and when a task has been performed, also submit that the task has been completed. Additionally, the worker can rate the client for whom the tasks was performed. These use cases are characteristic for the worker part of the application.

4.1.3 Brief Market Analysis

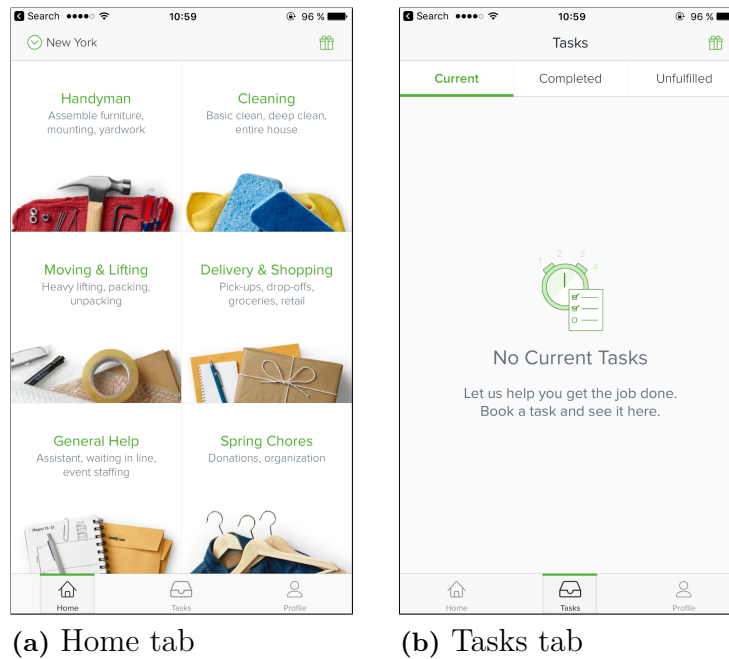


Figure 4.1: Screenshots from TaskRabbit, showing the *Home* tab and the *Tasks* tab.

Researching similar applications gave basic design inspiration and an idea of what the application should be capable of. The first application that was studied was TaskRabbit’s mobile application. The main view of their application contains navigational tab bar, leading to *Home*, *Tasks*, and *Profile*, as can be seen in Figure 4.1. The setup of using a tab bar, with tabs leading to pages similar to this example, also

seemed suitable for the application that we were designing. A similar design is also used in many other popular applications, such as Spotify’s [43] mobile application for example. Another interesting aspect was the task view found in TaskRabbit’s application, where all current, completed and unfulfilled tasks could be found, as can be seen in Figure 4.1.

4.2 Interaction Design Process

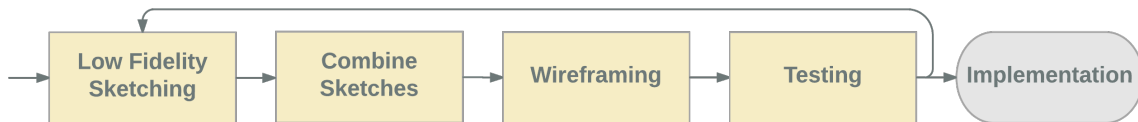


Figure 4.2: Flowchart visualising the programming development process.

The GUI implementation process used, was inspired by Agile UX [11] and Lean UX [9]. Our work was hypothesis driven, as suggested by Lean UX, and done in short iterations as suggested by Agile UX. More concretely, the implementation process (which was iterative and repeated) used in the project, consisted of the following steps:

1. Low fidelity sketching
2. Combining and refining sketches
3. Digital wireframing
4. User testing
5. Test evaluation

4.2.1 Low Fidelity Sketching

Initially, we started the design process with brainstorming, where we took the functionality previously defined in Section 4.1.2, and turned them into low-fidelity sketches of application views. We did this by visualising different design solutions by creating quick sketches on paper, to ensure a low fidelity. The choice of using paper sketches, which can be seen as a relatively primitive method, was made consciously, with the intention to enable us to create prototypes quickly, without risking being too attached to one’s own work, due to spending a lot of time on a particular design proposal. The paper sketches of application views were done individually, to promote maximum creativity and variety within the group. If this had been done together as a whole group at once, the group might have risked being influenced by, and mentally tied to, the ideas of a single individual, which could potentially strangle the combined creativity of the group. During the early stages of designing, it was valuable to get a large variety of design proposals, and these took advantage of being unbiased, to achieve the best outcome possible from the sketching phase.

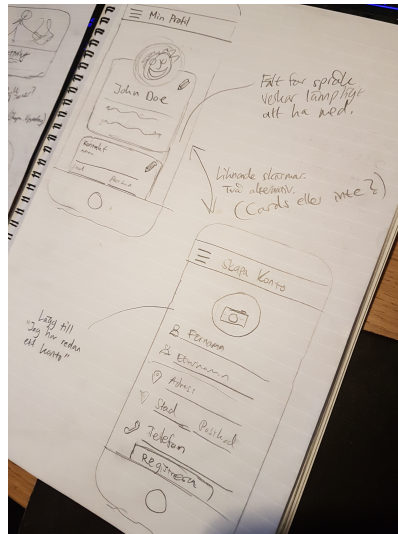


Figure 4.3: Example of low-fidelity paper sketches during development.

4.2.2 Combining and Refining Sketches

Once the individual low fidelity sketches had been created, the sketches were presented and discussed together as a group. The different design approaches were weighed against each other, and the strengths from each sketch were combined to create a common, refined version. Figure 4.4 shows an early iteration of combined and refined whiteboard sketches, which were created together as a group, based on individual low fidelity paper sketches.

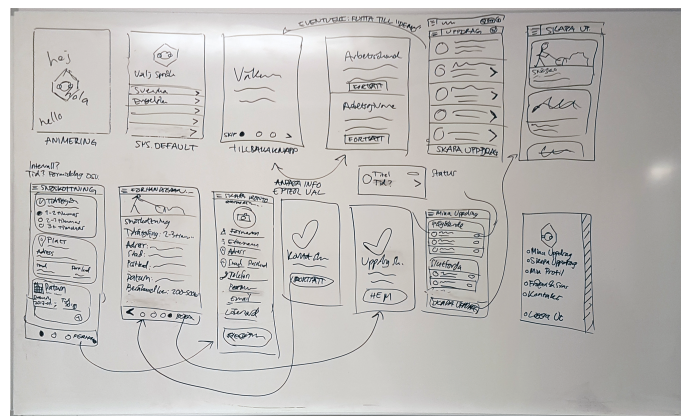


Figure 4.4: Early iteration of combined and refined whiteboard sketches, created together as a group.

4.2.3 Digital Wireframing

Using the common combined and refined sketches, a higher fidelity version of the sketched views was created in the form of digital wireframes [44]. These wireframes were slightly more detailed and advanced than the refined sketches, but they were not pixel perfect. Figure 4.5, shows two different preliminary views created as wireframes. Figure 4.5a displays the list of owned tasks, where all tasks owned by

the user are available for an overview. Figure 4.5b shows a view displayed during task ordering. These wireframes were later used during user testing.

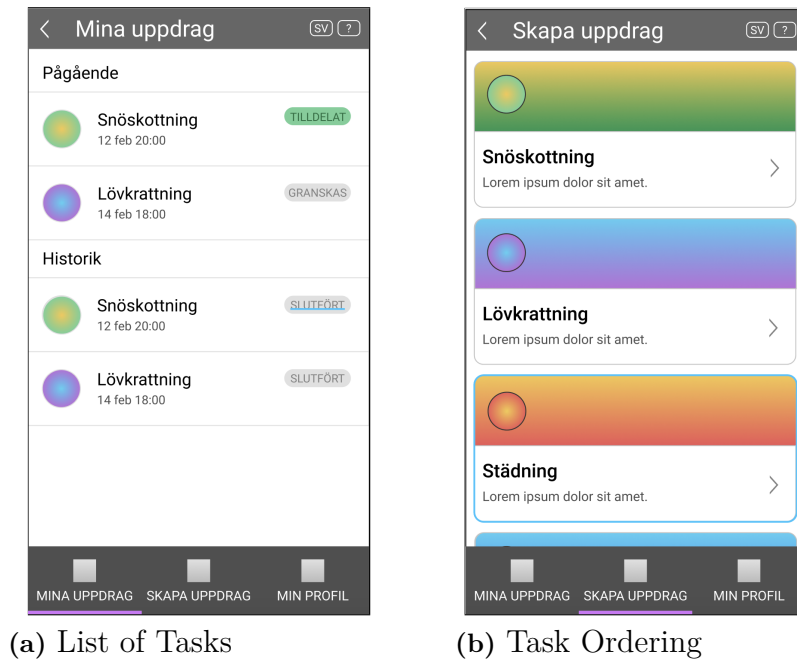


Figure 4.5: Early iteration of wireframes for the client part of the application, showing the list of owned tasks, as well as the option to choose a task type during task ordering.

4.3 Design Testing

The first stage of the design process, consisting of brainstorming and paper sketches, was tested and evaluated within the project group itself before moving on to testing on different target groups and an expert. The external tests were performed with printed wireframes of the application views, to enable us to quickly switch between views during our interactive test scenarios. The wireframes were printed to match the size of an average smartphone display to give a realistic impression when seeing them. The usage of wireframes enables the demonstration of interactive prototypes in an effective and more realistic way.

Methods used when testing were, as mentioned in 2.3, *Usability Testing*, *Think Aloud Testing*, and *Heuristic Assessment*. These methods are helpful when identifying elements of the interface that could be improved with further developed.

The tests focused mostly on design flows, icons, and formulations. The group supervisor, Sus Lyckvi, was consulted as an expert, before and after the tests were performed, in order to optimise the gathering and interpretation of relevant data.

After analysis of the collected data, a new design with improvements was proposed. This design was evaluated, implemented and tested again to see if the change gave the desired result.



Figure 4.6: Printed wireframes used during testing.

4.3.1 Expert Tests

The expert tests were performed using wireframes in several iterations throughout the design phase. The expert tests were always performed on our supervisor, Sus Lyckvi [45], and were prepared by printing the wireframes on paper. No specific tasks, or use cases, were tested, since *heuristic assessment* does not require this. Instead the method focus on identifying both major and minor problems, by looking at several parts of the design separately.

During the tests notes were taken as the expert went through the wireframes, so her comments could be discussed within the group when improvements were suggested and implemented.

4.3.2 User Tests

Two user tests were conducted for this application. The first user test was targeting the clients who order various tasks, while the second user test was focused on workers who were going to apply for those tasks.

A number of test scenarios were prepared ahead of the tests, such as *Order a service* and *Change language of the application*. These use cases were deemed to cover important functionality and should therefore be covered by the tests, as any problems had to be caught quickly and corrected.

The think-aloud protocol was used when conducting the test scenarios. One person administered the test and interacted with the person going through the scenarios, while another person handled the paper prototypes, and the last person took notes of the test subjects comments and interaction with the prototype. Afterwards, the test subject was asked a series of questions regarding their overall experience, for example regarding any issues they had not articulated during the test, and if they would be interested in using the service in the future.

The content of the tests varied depending on the target audience, however the methods used during the tests remained the same.

4.3.3 Client Wireframe Tests

The description of the possible task ordering clients, as provided by Just Arrived, was a middle-aged person with good experience of using touch devices, preferably living in a house. The client tests were performed on four subjects, randomly selected but all judged as potential users of the application, within the description of the provided persona.

The tests were based upon two use cases, where the first one was attempting to open the application for the first time, and to order an arbitrary service. The second test case was to change the language of the application, by interacting with the application wireframes.

The first use case gave a lot of feedback. Arbitrary issues brought up included an inconsistent use of words, such as the inconsistency between *account* and *profile*. Another person noted that the wireframe views did not include any terms and conditions. Issues hindering progress for the use case included a clear payment confirmation, and some people were confused when they had to create an account, and wanted that step moved to a different part of the application. One person did not want to create an account at all, preferring to get a confirmation number or email instead.

The use case of changing the application language appeared to be a clear and straightforward task. The test persons found the language icon at the top of the wireframes quickly. However, one person wanted flags to be used to represent languages, something the group consciously had decided not to do to avoid confusion, and to avoid alienating people from possibly conflicted areas.

4.3.4 Worker Wireframe Tests

The worker part of the application was tested using two newly arrived immigrants, who were potential workers. They were contacted and selected by Just Arrived.

This user test involved two use cases, where the first one was to create an account, and the second test case was to find a suitable task and try apply for it. For this test the wireframes were in Swedish, making some translation necessary. Here we could benefit from having a member fluent in Arabic, as the test subjects had limited knowledge of Swedish and English.

The Swedish user interface was the main obstacle for the test participants. This brought up issues with translation as well as suggestions for how Swedish could be simplified. This also contributed to testing how well the icons and general flow of the application performed. One of the users had some issues finding the option to change the language and the other user further suggested the option to upload documents, such as old recommendations and qualifications, to the user profile. Other than this the general flow of the application seemed to work well.

4.4 Programming Development Process

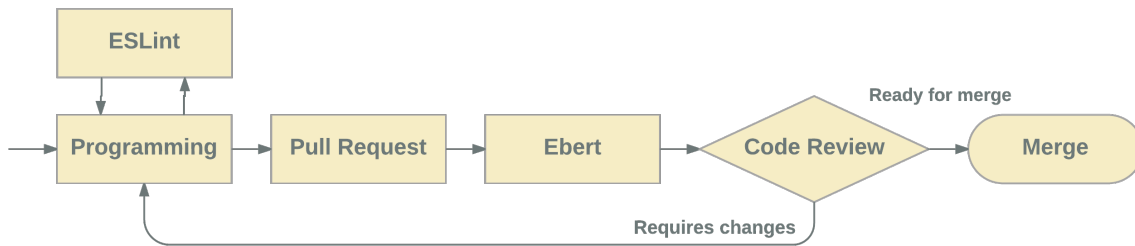


Figure 4.7: Flowchart visualising the programming development process.

The programming development process began with splitting the project into smaller tasks to populate our Kanban backlog as described in **2.1.1**. As soon as that had been done, the backlog was sorted with the most valuable cards at the top prioritising their development. A couple of the tasks were then assigned and work commenced. New cards were added to the backlog throughout the process, as soon as new tasks were discovered, often after a meeting or a test.

4.4.1 Programming

Initially, the programming tasks mainly consisted of implementing views from wireframes. At this stage the components used static data, to illustrate what they would look like in the future, once they were populated with actual data from the Just-Match API (see Section **3.3**). Hence the static temporary data tried to follow the structure of actual API responses. In this way the components were constructed in a way such that they were able to render static data following the structure of data from actual API responses. This made it very simple to later introduce back end functionality, which would replace the static demonstrative data with actual data from API responses, and without changing anything but the data source in the components, the data could be displayed instantly.

When most views had been created with static data, a basic navigation flow between the views was introduced. Up to this point all static views had been reached through a single view containing buttons to each view, for quick access during development. Now the static views were placed in a navigational tab bar, and navigation between views was introduced. Back-end functionality was developed in parallel during this time. As API functionality was realised, the previously static demonstrative data in the views were replaced with data from real API responses.

The JavaScript linting utility ESLint [46] was used to follow best practises, avoid bugs, and follow a consistent code style. Having a consistent code style makes it easier for other developers to be able to read and understand the code. When using ESLint and other linting utilities, a rule set explaining how various things should be treated is defined. These rules can for example define whether to use tabs or spaces in the code. If a developer is not following the rules defined in the rule set, ESLint notifies the developer with either a warning or an error. Airbnb has released

a popular and widely used JavaScript linting guide, with numerous rules for best practises defined [47]. The ESLint configuration file used in the project was based on the Airbnb configuration, with some additional React Native specific rules applied.

4.4.2 Pull Request

The version control system Git [48], and the web-based Git repository hosting service GitHub [49], were used throughout the project. These tools facilitated development, as everyone could stay up-to-date with the latest version of the application, and make changes in the same code without facing numerous merge conflicts. Techniques from the well-known Git branching model GitFlow [50] were adopted, in order to standardise the workflow. Changes in the code were done in separate feature branches, and never pushed or merged directly to the develop branch of our Git repository, but instead all changes wanting to be merged to the develop branch had to be turned into pull requests [51]. This allowed us to inspect and review each others code before accepting and merging it with the develop branch. The pull requests contained a concise summary of the changes made, as well as a screenshot if the changes affected the user interface. This made it easier to get an overview of the changes made in the pull request, as opposed to having to go through the code changes manually, in order to try to figure out what the changes entailed.

4.4.3 Code Review

Using version control through GitHub, with pull requests that got reviewed by some other developer before merging, in combination with warnings and tips from ESLint during development, the code was constantly evaluated to ensure a high code quality.

Many unnecessary merge conflicts could be avoided, since all team members shared the same development setup, with ESLint giving warnings and hints, to automatically standardise the code.

In addition to the ESLint reviews, we also used the service Ebert [52] to review the code as soon as it was turned into a pull request. This made it easier for the pull request reviewer to find lesser errors, such as indentation issues, and also worked as a compliment to ESLint for the submitter by showing potential errors in all submitted files.

4.4.4 Application Testing

The application tests were performed several times with the application in its state at the current time. Mainly by having other group members review the implemented views or functionality added while doing the code review, before merging them with the develop branch. A heuristic evaluation was also performed together with the group's supervisor, Sus Lyckvi.

Expert Testing

The expert tests performed on the application were very similar to the tests using wireframes, mentioned in **4.3.1**, with the expert moving through the application and identifying possible flaws.

5

Results

This chapter will show the complete results of the project. The first part of this chapter will go through the different flows of the application. The second part of this chapter will go deeper into the visual design of the application, and the design related choices that have been made. Lastly, this chapter will cover the system architecture of the application, regarding folder structure, navigation, and networking.

5.1 Application Flow

This section demonstrates the most essential application flows, both from a client's and a worker's perspective.

5.1.1 Task Ordering - *Client*

This is the main application flow a user goes through when ordering a task.

Order Task

The first part of ordering a task consists of choosing a task, and specifying details, such as date and location. As soon as that is done, a preview and confirmation screen is shown before the task is published. Figure 5.1 shows the first steps involved in ordering a task.

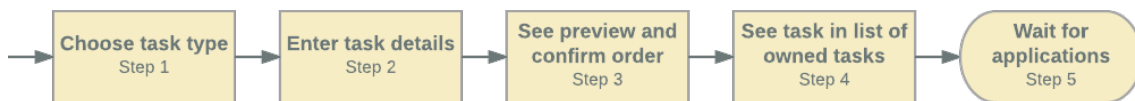


Figure 5.1: Interaction flow when ordering a task.

Choose Worker

When the task receives applicants the client gets notified through a push notification, whereat a worker from the list of applicants can be chosen. After paying for the task, a confirmation message is displayed. When the task has been completed by the worker the client is notified (see Section 5.1.1).

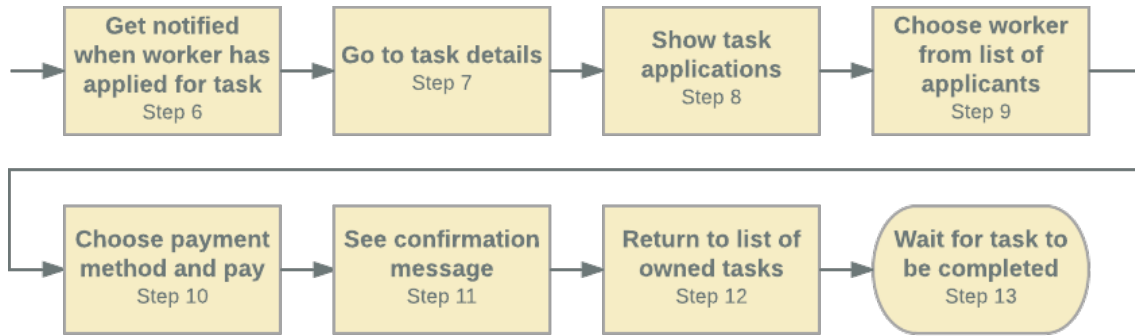


Figure 5.2: Interaction flow when choosing a worker.

Confirm Task Completed

When the task has been completed, it is time to confirm its finish, and rate the worker. The confirmation that the task has been completed is either initiated by the client or the worker. If the worker states that the task has been completed through the worker application, the client receives a notification about the claim, and can then proceed accordingly. The client can also take the initiative and mark the task as completed. Figure 5.3 shows the steps involved in confirming that a task has been completed.

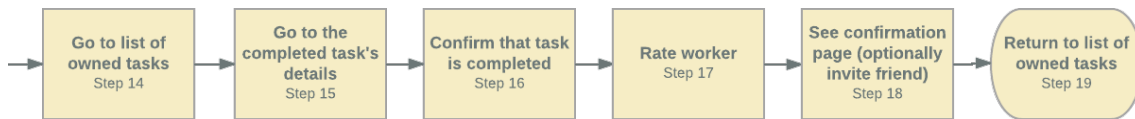


Figure 5.3: Interaction flow when confirming that a task has been completed.

5.1.2 Find Tasks - *Worker*

This section will display and describe the flow through the application when a worker applies for a task.

Find and Apply for Task

The tab for finding tasks lets the worker find tasks in two alternative ways - either by using a compact list view, or by using a map view. The user then proceeds by pressing on a task in order to view more details about it, such as location and date. The user can apply for a specific task from the detailed task view. A confirmation screen with information about the task is then displayed to give feedback to the user. The user is then redirected back to the list of tasks, located in the *My Tasks* tab. After applying for a task, the worker waits for a response from the client, to know if the task was assigned to the worker or not. Figure 5.4 shows the steps involved in finding and applying for a task.

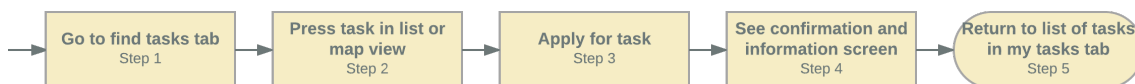


Figure 5.4: Interaction flow when finding and applying for a task.

Task Status Notification

After applying for a task and waiting for the clients response, the worker is notified by an email and a push notification, to know whether or not the task was assigned to them. If the task was assigned to the worker, the option to navigate to the detailed task screen is included in the push notification. Otherwise, a notification is still received, but without navigation to the task in question.

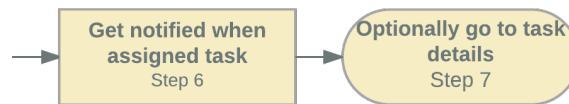


Figure 5.5: Interaction flow when receiving a notification about being assigned the task.

Confirm Task Completed

When the task has been completed, it is time to confirm that it is finished, and also let the worker rate the client, who ordered the task. This use case is very similar to the steps in 5.1.1, with the difference that the worker is rating the client, and not the other way around. Figure 5.6 shows the steps involved in confirming that a task has been completed.

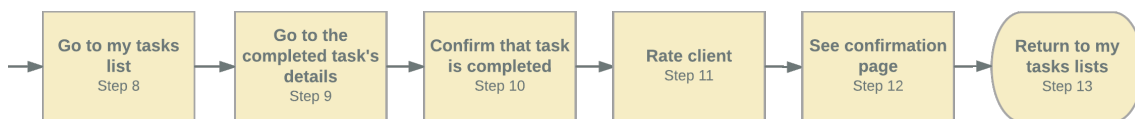


Figure 5.6: Interaction flow when confirming that a task has been completed.

5.2 Design

An effort was made to follow the design guidelines of Just Arrived when designing the mobile application. The user interface is designed in a simplistic way with minimal use of text and with numerous graphical images that clearly explains different steps. For example a picture or illustration of someone mowing a lawn, in addition to only having the text *Lawn Mowing*, helps visualise the lawn mowing option in the set of task categories.

5.2.1 Usability

The mobile application is designed in a way that should make it easy and intuitive for newly arrived immigrants to comprehend and to use. The user interface of the application is designed in a simplistic way with illustrative images and minimal use of text.

There are also no hidden views (e.g. content in a drawer menu), that a user has to swipe, or do some other obscure action, to access. If there is a view the user needs to access, there should be a straightforward way to get there.

The worker part of the application is designed for newly arrived to quickly get an initial basic understanding of each task there is to apply to. Each published task has an associated image representing the type of the task as well as a brief introduction of the task and how it should be performed. For example the lawn mowing task displays a picture of a lawn being mowed, as well as an icon depicting grass.

5.2.2 Language Selection

Currently the language implemented in the application is English and all text is retrieved using `il8n` (see Section 3.2.5), making it simple to add new languages. If the application supports the phone's default language, that language will be displayed by default. This drastically lowered the priority of implementing another way to select language, and therefore it has not yet been implemented. In the case where the phone's default language is not supported, the application will fallback to a preset language which currently is set to English. The application does also not support right to left adjustment, meaning that icons remain in the same position for all languages, independent of the reading direction of the chosen language.

5.2.3 Design for the Client Part of the Application

This section will show screenshots from, and describe the design of, the client part of the application. *All additional screenshots can be found in Appendix A*

Order Task

In the use case when a client wants to order a task, images displaying the different tasks are shown. Figure 5.7 shows screenshots of the application, from the *Order Task* tab. Here the different tasks are visualised with a title text, a short description, an image, and an icon. All these four elements are connected to the task, and used in the application in multiple ways.

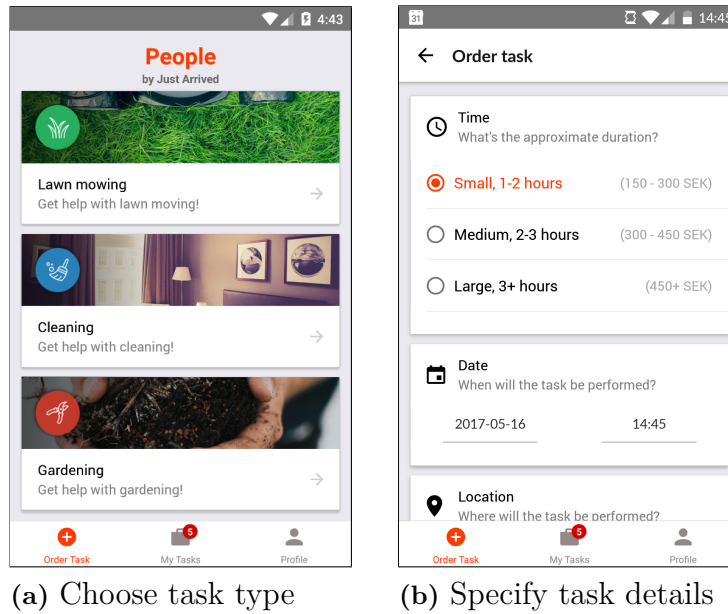


Figure 5.7: Screenshots from the client part of the application, showing screens involved when ordering a task.

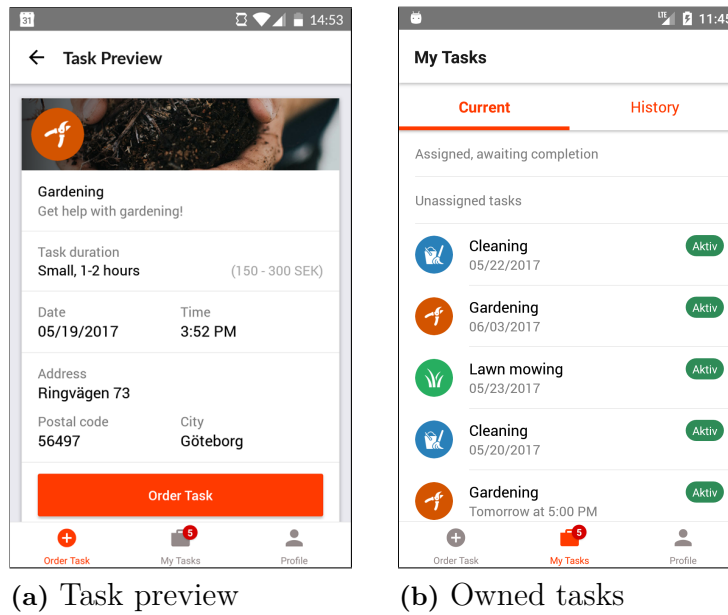


Figure 5.8: Screenshots from the client part of the application, showing screens involved when ordering a task.

In Figure 5.8b, the *My Tasks* tab is selected. Here the icons shown in 5.8a can be found again. However, the header image is omitted in this screen, in order to allow for a more compact view. The reason for wanting to achieve a more compact view of the list of tasks is that it might contain a relatively large amount of tasks. A more compact list makes it easier to get an overview of the data. The use of icons makes

it even easier, than opposed to reading the title of each task in the list. Reusing components makes it easier for the user to get a feel of how different screens and behaviour of the application are connected [53].

Figure 5.9 displays a component displaying the lawn mowing task. It is built using the standard component Card, coming from Material Design [54]. It consists of a title, a description, a header image, and an icon. The information the user gathers from these elements strive to make it intuitive and clear what the component is showing. The small grey arrow pointing to the right indicates that the card is clickable, and that it will take the user forward, to the next stage, upon clicking. The grey arrow can also be seen in other components in the application, once again to indicate interactivity and navigational manoeuvrability. The descriptive text has a smaller font size, and lower contrast, than the title text above it. This is used for typographic purposes [55], to indicate hierarchy, and to show what information is supposed to draw the attention of the user, and what information is secondary.

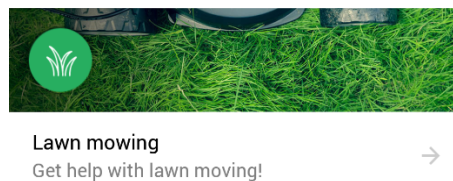


Figure 5.9: Card with image header displaying the lawn mowing task.

Choose Worker

Figure 5.10 shows the three steps in the choose worker flow. The main screen (see Figure 5.10b) being the list of applications to easily get an overview of the workers different ratings and prices. And the worker profile screen (see Figure 5.10c) where you get to see more detailed information about the worker and also get the choice to view references before either moving back to view other profiles or pressing the assign button.

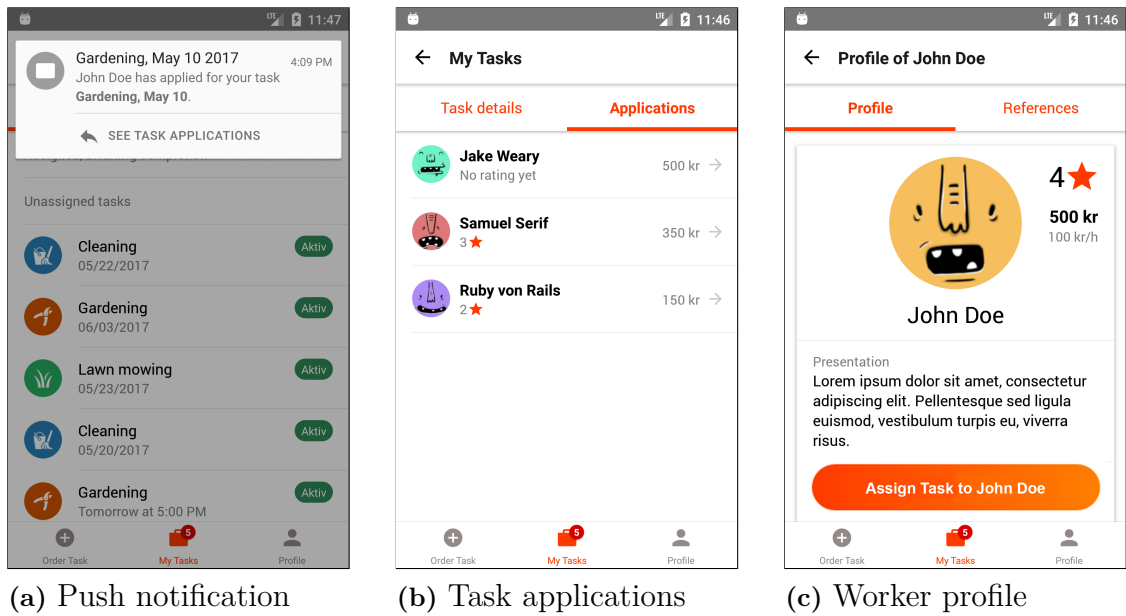


Figure 5.10: Screenshots from the client part of the application, showing screens involved when choosing a worker.

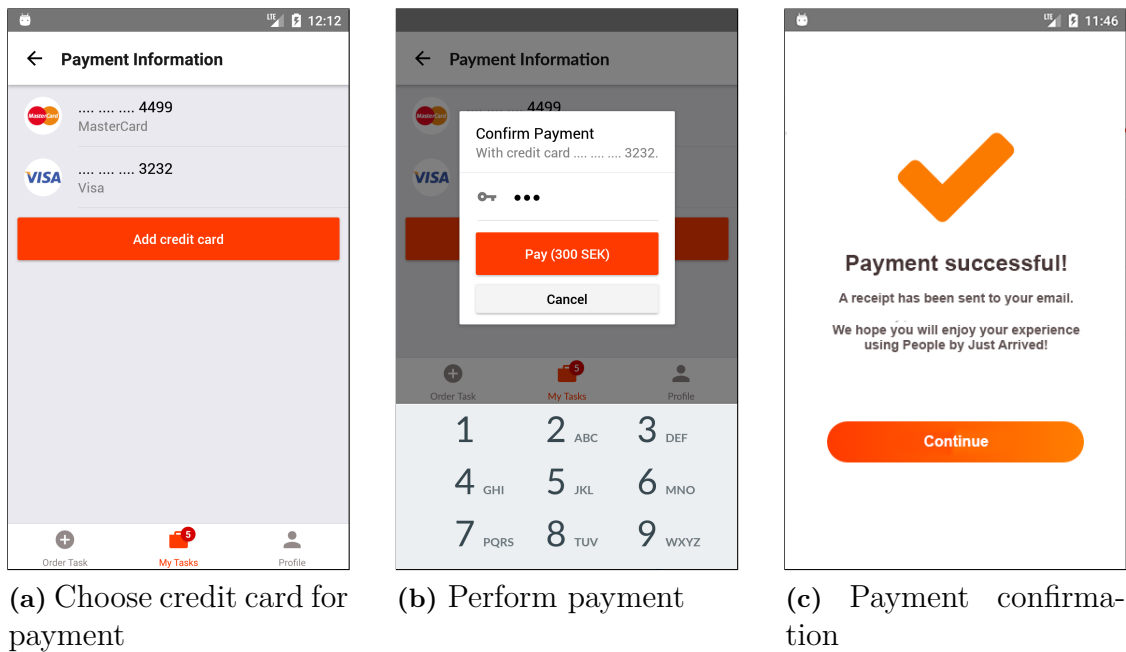


Figure 5.11: Screenshots from the client part of the application, showing screens involved when paying for a task.

Confirm Task Completed

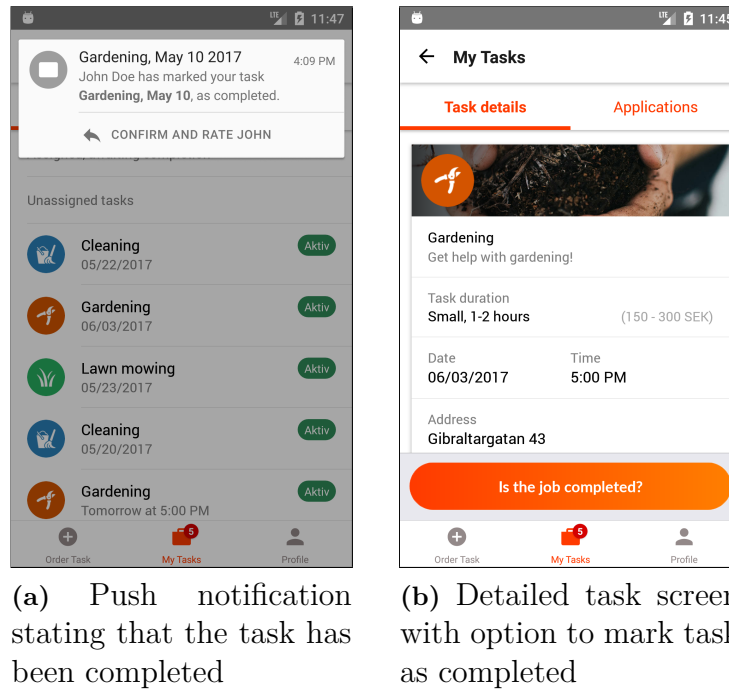


Figure 5.12: Screenshots from the client part of the application, showing screens involved when marking a task as completed.

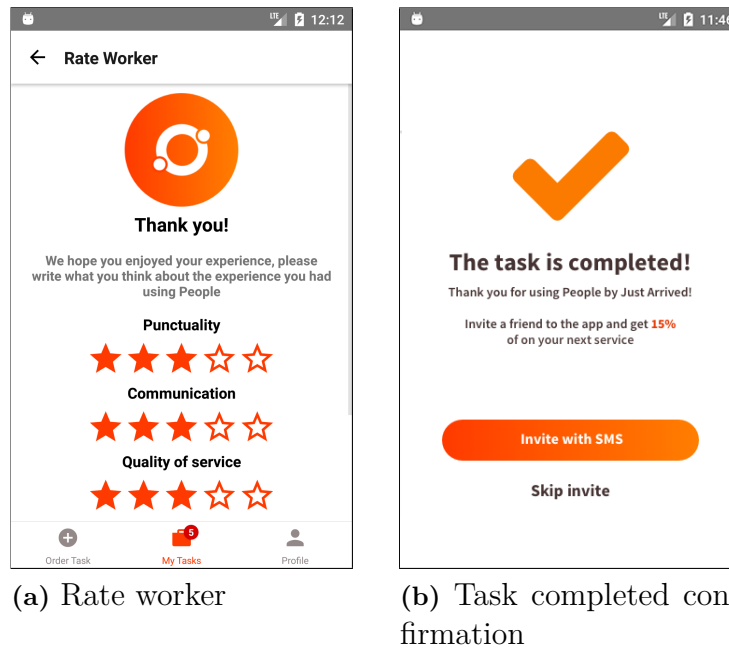


Figure 5.13: Screenshots from the client part of the application, showing screens involved when rating the worker.

5.2.4 Worker Prototype Design

This section will show images from the visual prototype and describe the design of the worker part of the application.

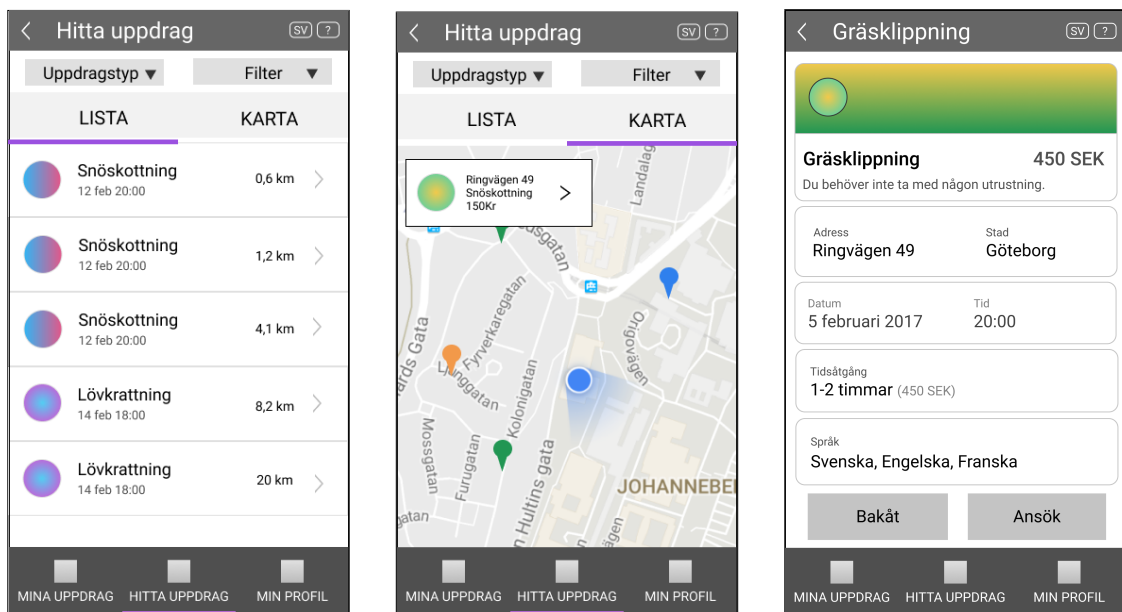
Find Task and Apply

There are two alternative views for finding tasks as a worker. In the default view, tasks are displayed in a compact list to get a quick overview of the tasks available. The information for each task is the task type, date, time, and distance from the home address of the worker.

There is also a map view, used to see the location of the tasks in the vicinity of the user. The different pins on the maps have different colours to symbolise the different task type. This will make it possible for the worker to get an idea of the task at each pin easier.

The worker is able to filter the results by task type and only show tasks within a certain distance, between certain dates and times. This is to enable the worker to just look at the tasks that is interesting to them.

Both the list and the map utilise the task type symbols to enable the worker to quickly differentiate between all task types.



(a) Find tasks (List)

(b) Find tasks (Map)

(c) Selected task to apply for

Figure 5.14: Wireframes from the worker part of the application, showing screens involved when finding tasks.

Task Status Notification

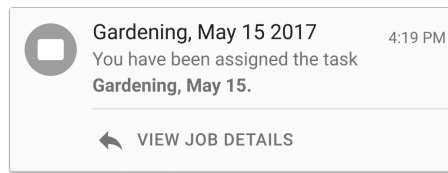
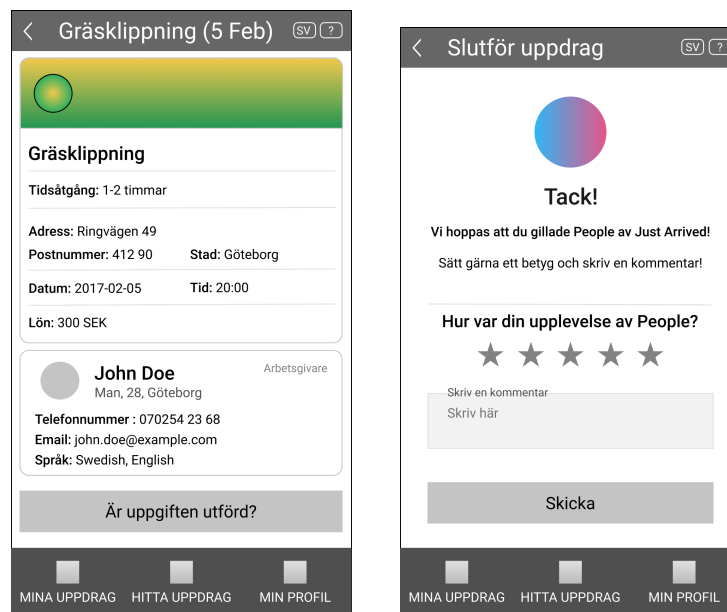


Figure 5.15: The notification received by the worker when accepted for a task.

If the worker gets accepted for a task they receive a push notification notifying the worker of the task and provides a link to the task details. The worker also receives an email with the task details. If the task would be assigned to someone else, the worker is still notified through a push notification.

Confirm Task Completed



(a) Confirm completion

(b) Review experience

Figure 5.16: Confirm task completed and rate experience.

5.2.5 Shared Design

Many aspects of the design is shared between the client part and the worker part of the application. Below some of our shared views and components will be described.

Introduction Wizard

An introduction wizard [56] is displayed the first time the application is started. It contains brief information about Just Arrived, and the application specifically.

5. Results

The information is displayed across different screens, and provides the user with the ability to navigate to the next screen, or completely skip the introduction.

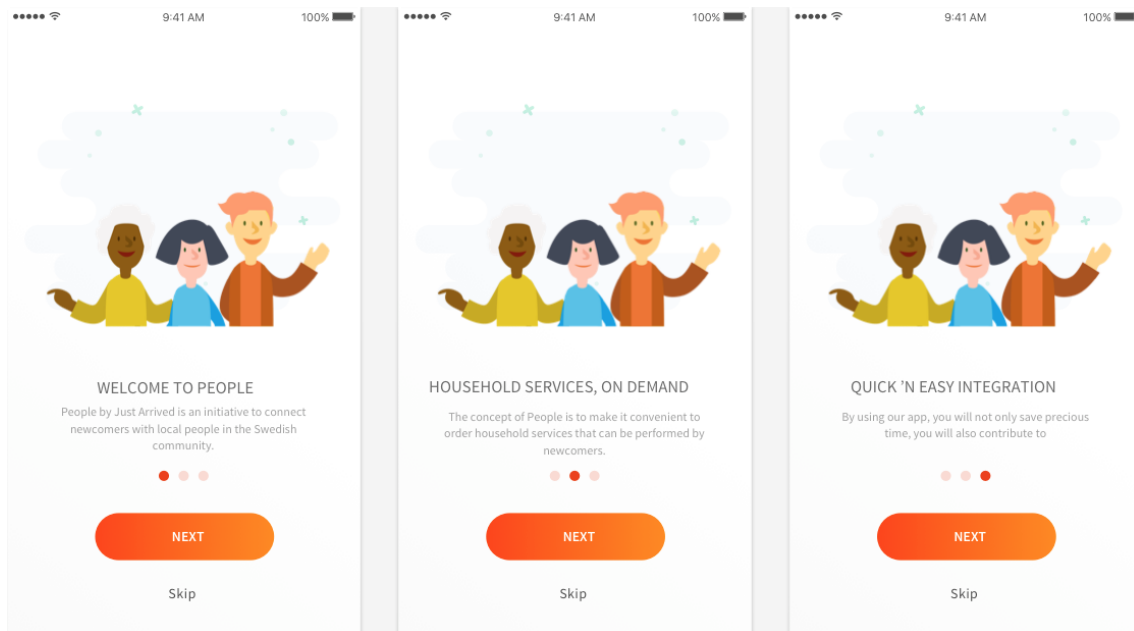


Figure 5.17: The introduction wizard.

Login Screen

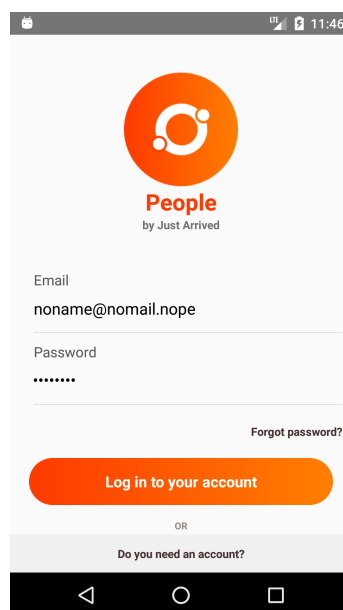


Figure 5.18: Screenshot of the shared login screen.

Figure 5.18 shows the shared login screen. It follows a fairly common layout for login screens, and contains the elements a user is expecting to see. There are however some interesting design decisions that have been made here. For example the input

fields in the form are using floating labels. When using a floating label, the label is always displayed, even after text has been entered in the input field. If a normal placeholder had been used, the label would be hidden after text input by the user. Floating labels have been used in other components in the application as well, for the same reasons.

The login screen contains navigation to three different locations, in three different ways: *Log in to your account*, *Do you need an account?*, and *Forgot password?*. The button with text *Log in to your account* is the most prominent, as it is the biggest button, as well as the most prominent when it comes to colour. It has the accent colour of the application, which instantly catches the user's attention. *Do you need an account?*, and *Forgot password?*, are less salient, with smaller font sizes, and no outstanding backgrounds. This way of indicating prominence and primary actions has been used in other places of the application as well.

The login screen is placed at the beginning of the application flow, after the splash screen and potential wizard screens. This means the user has to login before being granted access to the main part of the application.

Main Navigation

The navigation is built as a tab bar, with three tabs: *Order Task*, *My Tasks*, and *Profile*. The tab titles are written using title case capitalisation to give them more visual prominence and symmetry [57]. The tabs also have corresponding icons, to further make the tabs more intuitive. The tabs used in the tab bar provide access to the most used features of the application. According to Material Design Guidelines [58] and iOS Human Interface Guidelines [59], there should be between three and five tabs in a bottom navigation tab bar. The orange accent colour (which can be seen in the My Tasks tab in Figure 5.19) indicates which tab is selected. Badges are used to indicate that new information related to a tab exists, and is yet to be discovered by the user. A badge can be seen in Figure 5.19, as the red circle with the number five in it.



Figure 5.19: Main navigational tab bar shown throughout the application.

Task Icons

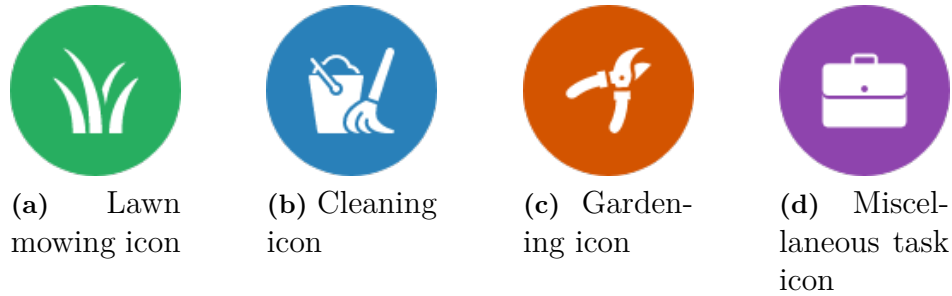


Figure 5.20: Task icons representing the different tasks in the application.

The motives for the icons in Figure 5.20 were found on the website Icons8 [60]. The motive and colour schema for the icons were selected with the goal of making them easily distinguishable and related to reality. For example the task of lawn mowing has a green icon with the silhouette of grass, and the gardening task has an icon of gardening shears, on an earth tone coloured background. The same idea of trying to replicate reality is applied to the other icons, with the generic briefcase icon as an exception.

5.3 System Architecture

The application is initialised directly in the main repository folder and the system architecture for the application can be divided into two head folders, `assets` and `js`, which will be discussed below.

5.3.1 Project Folder Structure

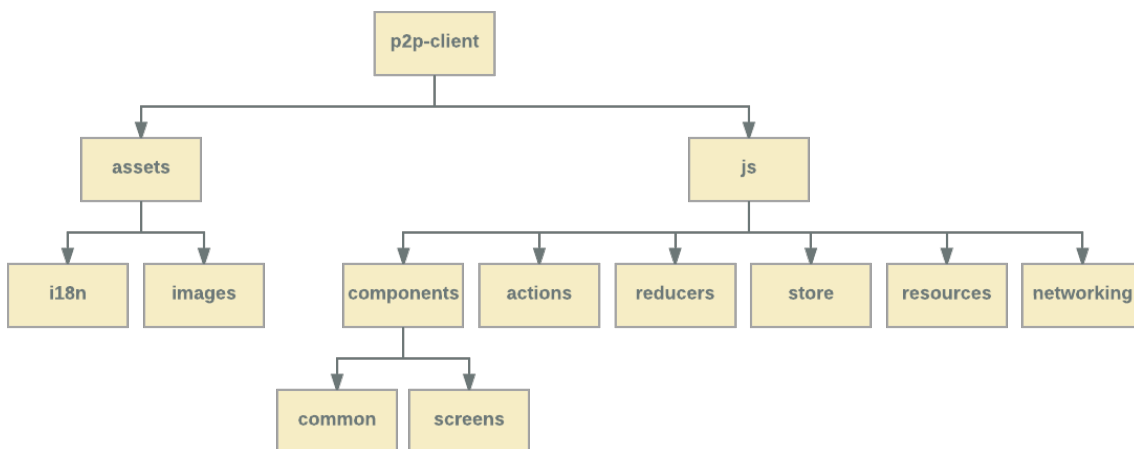


Figure 5.21: Tree data structure visualising an overview of the folder structure.

The `p2p-client` folder has the executable files for Android and iOS, Node Package Manager (npm) package information and the dependencies necessary to run the application.

`p2p-client` has a `js` folder, short for JavaScript, where the vast majority of the application specific code is found. When the platform specific executables from the main folder are started all they do is to instantiate `app.js` and `setup.js` found in the `js` folder. `setup.js` instantiate the state of the application through Redux and `app.js` start `appNavigator` found in the same folder. `appNavigator` keeps track of what view is displayed at any specific moment.

In the `component` folder there are two sub folders. The `screens` folders contains the screens used in the application, e.g the login screen. The screens are built partly with components from the `common` folder. Each screen has a folder for the components only used for the particular screen, style sheets, etc.

The `common` folder contains small components that are used in multiple screens. Keeping these files separate from the screen code makes it easier to find and reuse.

The top level `assets` folder contains two sub folders. The `i18n` folder contains the i18n JSON-formatted translation files. There is one file for each language, e.g. `en.json` for all English strings. The `images` folder contains all local images (i.e. images that are not directly fetched through a URL), such as the application logo etc.

In the distinction between `assets` and `resources`, `assets` contains things that have no code in them, while the `resources` folder contains assets that might contain code, e.g. the theme file, which contains rules for the colour scheme and component sizes. What makes the theme file a resource and not an asset in this distinction, is that it contains some JavaScript code (e.g. some theme properties depend on the device platform (Android/iOS), and this is determined using JavaScript).

Everything that has a graphical representation is considered to be a React Component, and can thus be found inside the `component` folder.

The `js` folder further has folders for Redux actions, reducers, and store (found under the folders with respective names: `actions`, `reducers`, `store`). The `js` folder also contains a folder called `networking`, with files related to API communication.

5.3.2 Application Navigation

Navigation between views in the application is realised using React Navigation as described in **3.2.3**. The root navigation of the application is a `TabNavigator`, with three different tabs. Each tab is a separate `StackNavigator` (which makes it possible to keep track of a stack of views while navigating in one of the tabs (i.e. the back button makes sure that you return to the previously shown view in that tab)).

The root `TabNavigator` is called `appNavigator`, and the three `StackNavigator`s that it contains are called `CreateTaskTab`, `MyTasksTab`, and `MyProfileTab`.

A wrapper class called `AppNavigatorWithNavigationHelpers` creates an `AppNavigator` component and passes in the `appNavigationHelpers` prop. This makes it possible

(but not mandatory) to expose navigational functionality to Redux. Previously you would need access to the navigation prop (which gets passed down to all components which are defined as views in the navigators), in order to use the navigate method. When using Redux it is no longer necessary for a component to have the navigation prop provided, as it can use Redux actions in order to navigate freely. In other words this makes it possible to navigate to any view, from any component.

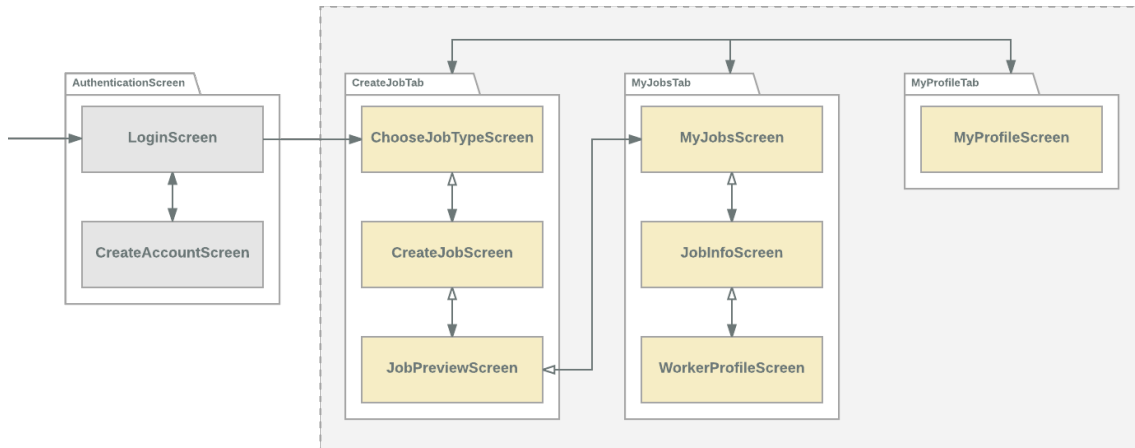


Figure 5.22: Visualisation of how the views are grouped in tabs, and the navigation between them. Black arrow head indicates the ability to navigate, and white arrow head indicates the ability to navigate backwards (i.e. by using the return button). The grey section with dotted border indicates the main content of the application, requiring the user to be logged in. Each tab of the navigation tab bar is indicated by a package with screens in them: *CreateJobTab*, *MyJobsTab*, and *MyProfileTab*. While inside the main part of the application, navigation is always possible between tabs.

5.3.3 Redux Integration

Redux (see Section 3.2.2) is used to manipulate and maintain the state of the application. The Redux storage is initialised when the application starts, and the storage is manipulated using actions and reducers, which are separated into two corresponding folders. Each folder contains files with the same names. For instance `actions/session.js` provides functions used when calling reducers, while `reducers/session.js` handles incoming actions and perform the manipulation of the state of the application.

The Redux state and actions are connected to the GUI through props. Props can be defined to point towards certain parts of the storage, and every time the store changes, the view is also updated with new props that contain the updated data. Actions are also dispatched to the reducers through the use of props. A prop is set to point towards a function that dispatch some action and when it is called the store is updated. This can be considered to follow the Model View Controller (MVC) pattern where the store (model) contain some data that is displayed in a view. When something happens in the view an action is dispatched to the corresponding reducer (controller), and the model is updated, which triggers a change in the view.

5.3.4 Networking

The networking folder contain methods used to communicate with the API. The only class that actively sends fetch requests (see Section 3.2.6) is `networking.js` which defines methods for getting, putting, patching, and deleting network data towards some URL. The request method, headers and body is created in `request.js`. Further there are separate classes to communicate with each API endpoint. To set up and remove user authentication against `/api/v1/users/sessions` the application would use `session.js` to communicate with the API.

5.3.5 Networking and Redux

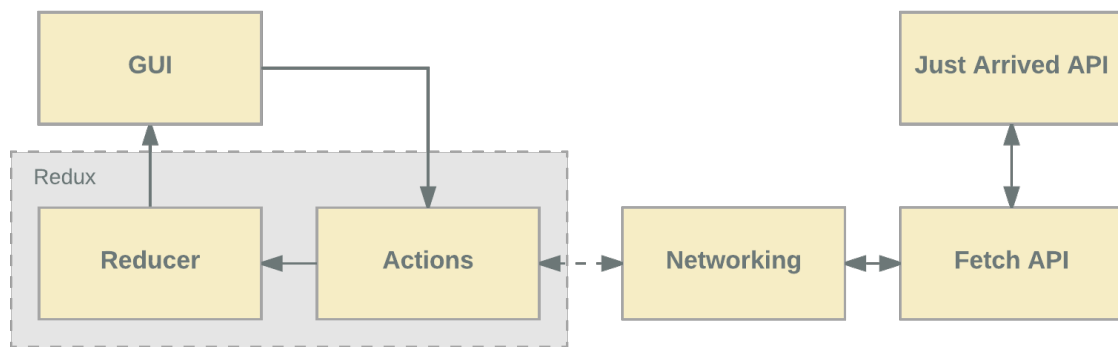


Figure 5.23: Overview of GUI, Redux, networking, and how they are connected.

In order for networking to be seamlessly integrated with the application it had to be tied closely together with Redux. To achieve this all networking methods are only accessed through Redux actions. When some action that require networking functionality is called from the GUI it first dispatch to store that a request has been sent before sending the request. While the request is being sent, the GUI is simultaneously set to display a spinner due to the state change. Once data is received the action dispatches the received data to the store, and the GUI is updated yet again to display the data.

6

Discussion

In this chapter, the entire project will be evaluated and discussed. Some of the main points that will be covered are limitations, methodology, design decisions, frameworks, and ethical choices.

6.1 Limitations

As mentioned in section 1.4, the intention was to implement both the client and the worker parts of the application, and test different languages within the applications. But due to time constraints these goals had to be dropped. An overly optimistic project plan, a slow initiation, and uncertain project goals, resulted in a struggle with following the timetable. The lack of guidelines also contributed to the slow initiation, where the group for a couple of weeks received different requirements from different actors of the project.

Due to the setbacks given above, we had to limit ourselves to developing the code for the client part of the application, with support for English only. This affected the project both negatively and positively. Negatively, because the worker application was left in the prototype stage, at the same time as the client part of the application was developed supporting only one language. On the other hand, as the implementation of the worker application was decided to be out of scope for the project, we could focus on the client part of the application. Therefore, we could implement a system for later adding more languages in the client part of the application, which was one of our solutions to the obstacle we had regarding supporting multiple languages. We also had more time to improve the code for the client part of the application, making a lot of it reusable for Just Arrived when implementing the worker application in the future.

Some other areas that were discussed thoroughly but we never had the time to implement, are proper language selection, a map view for locations, and the option to subscribe to recurring tasks. The lack of language support also led to right-to-left GUI components never being properly explored.

Looking back, wondering what could have been done differently, we still would have had struggles both with time estimation, and seeing the project in its entirety. This due to the fact that a timetable with reasonable margins for possible setbacks is difficult to plan, when unfamiliar tools and techniques are explored.

6.1.1 The Language Barrier

As presented in the section above, the overcoming of the language barrier between the clients and the workers did not go as expected. During the initial phase of the project, the first thought of a solution to the language barrier was to design the applications in a way that is easy to comprehend, and with support for multiple languages. We also attached great importance to the use of graphical images and animations, which together with the text description would provide a clear demonstration of the task.

As time passed by, we realised the limitation of time. Our main goal for preventing language confusions, the support of multiple languages, as well as languages written from right-to-left, had to be eliminated from the project. Instead, we looked into an simplification of words, graphical images, and an implementation of the i18n functionality, which contributes with an uncomplicated way of adding more languages in the future. The graphical image is a vital part of the task description, since it is associated with the task. For instance, the task type of cleaning has an image with cleaning tools.

An additional solution to the language barrier was the decision to have prearranged tasks to choose between when ordering a service. As the task is ordered by the client, the client is not required to type text describing the task. The information needed to order a task is kept to a minimum and has specified input fields, for details such as time and location.

The simplified task ordering ensures that the same type of information is provided to the worker each time. It gets easier for the workers, as they do not have to translate the page each time they view a task information page. Having mostly predetermined text also makes it easier to translate the application into multiple languages, as no text will need automated translations, which would rely some external service like Google Translate. Staying away from automated translations will also prevent mistranslations. The predetermined text is mostly for task types, descriptions, and labels for client input fields. This makes it easier for the workers to understand the information associated with the labels.

The translation into other languages is handled by i18n, as previously discussed. If the translation is handled this way, it can be professionally taken care of, providing high quality translation for the necessary text.

6.2 Retrospective

About halfway through the project we realised that the goals set up during the project did not always match the expectations held by the different group members. To discuss these issues we decided to have a retrospective meeting. The meeting was composed of the two following questions as well as an discussion part, leading to the conclusions below.

What are we great at?

- Helping each other out whenever help is needed.

- Sharing knowledge on a regular basis.
- Writing clean code.
- GitHub discipline, making pull requests and reviewing as soon as possible.
- Giving good feedback.
- Communication.

What can we do better?

- Work at the same location as much as possible, being able to ask questions to someone in the same room greatly increases productivity. It also allows for easy sharing of knowledge.
- Set up firmer goals and follow them as close as possible, if you realise you are about to be delayed make sure to announce it as soon as possible. Whatever reason it may be, as long as we are aware we can adapt.

The retrospective meeting was a break for us to reflect on how the project had been going, and from these reflections make improvements. As within most project groups, our group occasionally faced problems, which were not always discussed openly. This due to the fact that successes are easier to point out, than failures. Especially since we had not worked together before, there was an ice that needed to be broken. By openly discussing both good and bad aspects, the trust among the group members grew. When comparing the time before and after the meeting, our productivity increased and the group was brought closer together.

Because of the positive effect the retrospective meeting had on the group, we all agreed on that it should have been done earlier and more often. If this had been the case, the increased productivity and the improvement of the group spirit would have evolved already in the initiation phase. However, we are pleased that it happened, because the tension we had within the group before the meeting has been replaced with good friendship.

6.3 Usability Evaluation

To make sure the project was always on the right track regarding usability, many evaluations were done throughout the project. Since we found it important to get feedback as early as possible, setbacks could be avoided.

The user testing helped us find flaws and collect useful thoughts on what actual users would want to see in this kind of application. The user tests were scheduled around specific dates when we expected to have reached different stages of the design, before the implementation of the application.

During the development of the client part of the application, we had many opinions about how the hiring process should be implemented. Even though we had many opinions, we did not know clearly how to solve this problem. So we decided to go for the method First In, First Out, where the task is assigned to the first person who applies for it. This choice was made temporarily, so the development could set off, as we were open for improvements and other suggestions. As a result of the user testing, the First In, First Out method was removed and another one was introduced. Most test subjects were not comfortable nor pleased with the idea of

not knowing who the worker performing the task will be. Instead, they preferred to choose the worker from a list of applicants, all applying for the same task. To avoid that only workers with the highest rating would be assigned all tasks, we decided to lower the price for newly registered workers, and workers with lower ratings in general.

When performing user tests for the worker application, we were limited by the number of test subjects provided to us by Just Arrived. Since we only tested on two test subjects, the extent of valuable feedback we received was limited. But some feedback included notifications for the application. The test subjects would prefer an SMS sent to their phone number, in addition to the push notification sent to the application.

6.4 Design Decisions

During the process a number of design decisions had to be made. Overall the goal was to improve usability, ease adaptation, and to minimise confusion. This chapter covers some of these motivations, which will be explained and discussed.

6.4.1 Login Screen Placement

As stated in 5.2.5, the login screen is currently placed at the beginning of the flow, when the user starts the application. At the earlier stages of development, the login screen was placed at a later stage in the application flow. The reason for this was the idea to let the user try the application before being prompted with a login screen, as requiring the creation of an account can be seen as a hurdle the user has to overcome.

The initial thought was to place the authentication before checkout of the order, or when attempting to view profile information. This way the client would be able to choose a desired task, fill in all necessary information, and then be prompted with an authentication screen just before finalising the order. However during user testing some subjects expressed that the sudden appearance of the login screen felt out of place when it blocked the completion of ordering a task. This impression, and discussion with Just Arrived, prompted the move of the authentication to the beginning of the application.

The discussion about authentication placement concluded with the idea that placing the login screen at the start of the application would not result in too much of a threshold, as the user had already downloaded the application, and thus already had enough interest and incentive to register for an account and log in. Another solution that was discussed was to skip the login requirement entirely and instead give the option of order numbers. But this was deemed to be unsafe as it would make it harder for users to manage and keep track of existing tasks. If further testing show that the initial idea was the correct one, it should not be to much of a challenge to adjust the current application to the old flow.

Additionally, the idea was to give the user the option to sign in with their pre-existing Facebook or Google accounts. This would completely eliminate the hurdle

of filling in information when creating their account, eliminating many of the points about why authentication placement matters. However, due to limited time, this was never implemented.

6.4.2 Navigation Menu Choice

Deciding on the best way to navigate through the application was an important decision with many discussions, whether to use a navigation drawer or tab menu, for the main navigation of the application. Weighing the advantages and disadvantages for each type of navigational menu helped during the development process, and eventually we settled on using a navigational tab bar menu bar, which should be located at the bottom of the screen.

Choosing the tab menu over the navigation drawer was done for several reasons. The main motivation was that a navigation drawer might not be very intuitive to use, as it requires the user to either use a swipe gesture, or press a, so called, hamburger menu icon, at the top of the screen, in order to make the menu appear. Just Arrived also confirmed our suspicion that the newly arrived can have problems with these kind of options, specifically that a hamburger icon might not be very easy to understand. The navigation drawer[61] is an Android design pattern, which would make the platform implementations less cohesive. Another factor was the realisation that the application simply did not have enough navigation requirements to justify having an entire drawer. Placing the tabs at the bottom of the screen also makes it easier to navigate when the option to change view is always available and easily accessible.

A possible downside to tabs is that it could be harder to expand on application functionality with the tab navigation. But at the moment there are no indications that more than three tabs will ever be necessary. In the worst case more tabs could also be added. As stated in 5.2.5, when using a tab menu, three to five different tabs are alright.

6.4.3 Unifying the Design for All Users

During the design process it was decided to use the same pattern and general style for both parts of the application. The motivation behind this was that it should be easy for the respective kinds of users to imagine how the other part of the application functions. When someone orders a task, the flow and preview should also be presented in such a way that the person placing the order should be confident that the applicants can easily understand what the task entails.

While there could be some benefits to optimising the task ordering flow for power users, these options rarely seem to be appreciated by the market at large. Instead the application design is an attempt to appeal to a larger market and to have as many shared components as possible. Reusing components gives the user familiarity with the functions, and makes it easier to overhaul the look and feel of the application. As a bonus, it is generally also convenient for developers to reuse components in multiple parts of an application.

Another consideration was that while the main market might be young professionals and busy house owners, the application should also be usable by older people who might need some help with various tasks. Their technical knowledge could be in doubt, giving them many of the same requirements as those defined for the newly arrived, making a shared design something that greatly eased all parts of the project.

6.5 Ethical Choices

During the course of the project an ethical stance had to be decided upon regarding a couple of design decisions. In the following subsections we will discuss them further.

6.5.1 Worker Selection

One of the ethical choices made was regarding the selection of workers for a task. At the very beginning we used a First In, First Out system, assigning the task to the first applicant, mainly because it was easy to implement, but also since it was fair and focused on the task being performed rather than who would perform it. During our first user tests we however received feedback on this approach of assigning workers. The test persons asked for the possibility to choose who would come and help them with their tasks, mainly in the case of tasks that were to be performed inside their homes. Essentially changing the focus from just having the tasks be completed, into giving the client the power to choose from all the workers that applied to perform the task, is something that perhaps is not entirely in line of the main interest of the application, in helping refugees without previous work experience into the labour market. This version of the selection more closely assembles how a company would proceed when hiring staff. However, if you were in need of a plumber or an electrician, you would rarely take part of that information before letting them into your house, perhaps simply because there is no easy way to find that information.

6.5.2 Rating

When we implemented the rating system for the clients, a thought about people possibly giving bad ratings was raised. Are we supposed to simply trust the client ordering the task giving a fair rating? What if everything is based on a misunderstanding due to communication, or if the client simply did not try co-operate? To be able to find such issues we decided to add the possibility for the worker to give a rating of the client, and the general experience of using the service, as well.

6.6 Framework Evaluation

As mentioned in 3.2, a number of various frameworks and tools were used during the project. In this section we evaluate and share our thoughts about these frameworks and tools.

6.6.1 React Native

In the beginning of the project, a decision had to be made regarding how to implement the application. The alternatives considered were to either write native code separately for Android and iOS, or to use a cross-platform approach, such as React Native as recommended by Just Arrived, in which code would be written in React and JavaScript. The group had previous experience of coding in Java (which is used for Android development), but little experience of JavaScript, and developing iOS applications using Objective-C or Swift.

After evaluating React Native during the pre-study, as described in 4.1, a decision was made to use React Native for the development of the application. This choice was made due to the ease with which a simple project could be started, the convenient cross-platform compatibility with iOS and Android, and the abundance of available information regarding JavaScript online. Just Arrived's preference for React Native also influenced the decision, considering that they might use our code in the future, in a continuation after this project has been completed.

There are other cross-platform development tools in addition to React Native, some of which are considered more stable than React Native. If React Native had been deemed as unsuitable, these tools would have been evaluated before settling for developing using native code. We simply found the cross-platform development too useful to pass up on. Sticking to one set of code saves time as the same functionality does not have to be written twice in different languages. This is also an advantage when it comes to maintaining the code. If a change is made in the code, it always affects both platforms, saving time and making sure no version gets ahead of the other.

After working with React Native for a while, the group is now fairly proficient in JavaScript. Jacob Burenstam, the CTO of Just Arrived, voiced the opinion that we should not hesitate to apply for jobs that require two years of JavaScript experience.

6.6.2 Networking Component

The initial idea for networking was to use Firebase, an easily integrated and expandable bucket service as a placeholder for a dedicated API [62]. Firebase differs from a traditional API in that instead of providing endpoints that have to be configured, they offer a bucket with storage space, where developers can create security rules as to who is allowed to access which folders. Their SDK also offers authentication straight out of the box, enabling developers to concentrate on creating the client rather than spending time on implementing API functionality.

On suggestion from Just Arrived, Firebase was discarded, and networking was realised straight against Just Arrived's development API [63], while Jacob Burenstam, CTO of Just Arrived, took responsibility for modifying API functionality as it became necessary. This has limited the implementation in various ways due to only being able to use existing functions most of the time, and the existing functions sometimes not being very clear about their purpose. But it has also been a relief not having to work on creating server, or Firebase functionality, simultaneously while creating the application.

The resulting networking component works well, and the integration between fetching data and Redux is fairly seamless, but the process of getting there did not go entirely smooth. After discarding Firebase, the second obstacle was the integration between networking functionality and Redux. The main issue was that Redux mainly functions as a state container, where logic is handled in actions and in the views (see Section 3.2.2). A way had to be found for an action to trigger a loading screen, download data, and then update the view with the received data. The way it has been implemented is powerful and easy to expand (see Section 5.3.5), but it might unfortunately not be entirely trivial to understand everything that is going on. It did not help that for the majority of the time there was only one developer dedicated to implementing this part of the application. Due to views being created before implementing networking and Redux functionality, there was also a large amount of refactoring necessary for most views.

6.6.3 Redux

The fact that Redux uses a single repository for storing all the states of the application, facilitates a lot of work for us as developers. We did not have to keep track of which component has which state at what time, or where the state is supposed to be sent. All changes to the states of the application are sent to the store, and components and views that use a particular state automatically receive the state by themselves. Not having to figure out how to pass state information between components in a proper fashion, and guaranteeing that every component that rely on the state is passed the new value, and instead just have each component work against the store, has made it an almost painless experience.

Redux has also made it easy to build new components for the application. We did not need to figure out what information had to be passed from certain view into the new component. We could just build a view and get the information from the store. This also made it easy to test out a new module without having to modify the already existing views, as the views operate independently, and only communicate states with the store through Redux, and never directly with each other. In this way we could test something new and delete it without affecting other components.

Although we found that Redux is a good tool for most situations regarding handling states, it is not perfect for everything. In some cases we have used normal React local states for components. If the state is used strictly in a specific component it is a lot more work to send it to the store and request the information, instead of just keeping it as a local component state. This has usually not been used for storage of regular data, but instead for some GUI related states, such as toggling what type of button should be displayed in a certain view.

6.6.4 NativeBase

When we first started coding we found using the components of NativeBase very useful and easy to use. It became surprisingly easy to create views for the application quickly, and try different design ideas.

However, as the project progressed we ran into more and more problems. The documentation was occasionally poor, and multiple components did not behave in the way they were described. In several cases we had to read the source code for NativeBase, instead of the documentation, to figure out how certain components actually worked.

We encountered especially many problems regarding input fields that used icons or labels. These were reported as bugs by users to NativeBase, but not yet fixed. Not being able to do anything about the problems, or finding out how components work from the documentation, made us feel like we were not in control. It also made the application worse as it seemed to contain bugs that we could not do anything about it.

If we had to make a choice today we would still have used NativeBase. In the case of bugs that were not fixed, we would have created our own components to replace the broken ones.

Someone with more experience might want to create all the components by themselves to have complete control, but for a smaller project with little to no prior experience of React Native, it was a good library. Especially if the documentation is improved.

6.6.5 React Navigation

React Navigation was introduced in January 2017 [64], making it a relatively young framework. Even though it is one of the more popular and commonly used frameworks for navigation in React Native, it is still in beta, and a stable version is yet to be released. This can unfortunately be noticed when using the framework, in some cases. During development we encountered some minor bugs, but more noticeably documentation that sometimes was invalid or not up-to-date. Generally speaking, the life-span and stability is something to examine and consider before committing to a framework. In retrospect, however, it is not guaranteed that we would have chosen another navigation framework due to these reasons. This is partially due to React Native in general being a relatively young framework, so it is hard to expect all frameworks for React Native (e.g. React Navigation) to be fully stable.

7

Conclusion

While all the goals of the project could not be realised, there are still conclusions that can be drawn from what has been done.

Optimisation for touch devices is something that should mostly be done by following existing design guidelines for the platforms used. The result might not always be the most intuitive, but design compliance ensures that the user will recognise design patterns, and from habit will be able to use the components.

The user's experience with technology, touch units, and the specific platform affects what value can be drawn from platform specific design guidelines. As the user's familiarity with the platform increases, the value of the design guidelines should increase as well. Respectively, if the user has no or little experience with the platform, the value of the guidelines decreases.

Based on the variety in expected technical background for end users, the value of user tests increases. The developers of an application typically all have fairly advanced technical knowledge, and thus do not represent the average user. Based on this it is also valuable to have both broad and narrow use cases tested, since they can result in different kinds of issues depending on who performs the tests.

Having a unified simplified design could go a long way in making the users feel secure in the knowledge that the correct message is transferred through the application. The use of standard input also helps both the client and the worker to be confident that the correct task will be performed, and it also facilitates translations and support for multiple languages.

Based on the experience gained during the execution of this project, some more subjective conclusions about how to maintain a collaborative project can be drawn. When scheduling a project plan, leave a margin for errors. If nothing else it would ensure time for a final test which would ensure more objective results. Working together and continuous feedback through retrospectives are crucial aspects of the agile process.

React Native and the packages used during development performed satisfactorily despite some version issues. As the framework matures these issues should decrease and the value gained from immediate cross platform support should increase.

The development of functional and intuitive user interfaces is an area of research that will remain relevant as new techniques are invented and the market continues to grow. Companies' desire to continuously renew their image in order to remain relevant also ensures that there will always be a constant demand for new solutions.

Bibliography

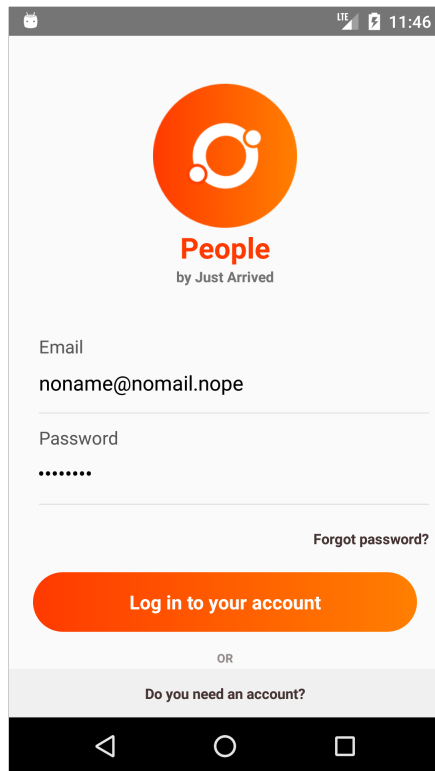
- [1] Migrationsverket, “Migrationsverket Statistik.” [Online]. Available: <https://www.migrationsverket.se/Om-Migrationsverket/Statistik.html>
- [2] Just Arrived, “About Just Arrived.” [Online]. Available: <https://justarrived.se/about-us/>
- [3] Agile Alliance, “Agile 101.” [Online]. Available: <https://www.agilealliance.org/agile101/>
- [4] K. Beck, M. Beedle, A. Van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas, “Agile Manifesto,” p. 28–35, 2001. [Online]. Available: <http://agilemanifesto.org/>
- [5] T. Ohno, “Toyota Production System: Beyond Large-Scale Production,” *Productivity Press*, p. 152, 1988.
- [6] Dan Radigan, “Atlassian Kanban.” [Online]. Available: <https://www.atlassian.com/agile/kanban>
- [7] Jeff Iasovski, “Kanban Board.” [Online]. Available: <https://commons.wikimedia.org/wiki/File:Simple-kanban-board-.jpg>
- [8] Trello, “Trello.” [Online]. Available: <https://trello.com/about>
- [9] J. Gothelf and J. Seiden, *Lean UX*. O’Reilly Media, 2013. [Online]. Available: <http://ebooks.cambridge.org/ref/id/CBO9781107415324A009>
- [10] E. Ries, *The Lean Startup*. Crown Publishing Group, Division of Random House Inc, 2011. [Online]. Available: <http://theleanstartup.com/book>
- [11] L. Ratcliffe and M. McNeill, *Agile Experience Design*. New Riders, 2012.
- [12] C. Snyder, *Paper Prototyping: The Fast and Easy Way to Design and Refine User Interfaces*. Morgan Kaufmann Publishers, 2003.
- [13] S. Farrell, “Mozilla Case Study,” 2015. [Online]. Available: <https://www.nngroup.com/articles/mozilla-paper-prototype/>
- [14] “Wireframing.” [Online]. Available: <https://www.usability.gov/how-to-and-tools/methods/wireframing.html>
- [15] B. Hanington and B. Martin, “Universal Methods of Design,” pp. 194, 180, 2012.
- [16] “Heuristic Evaluations and Expert Reviews.” [Online]. Available: <https://www.usability.gov/how-to-and-tools/methods/heuristic-evaluation.html>
- [17] J. M. Spool, “Great Designs Should Be Experienced and Not Seen.” [Online]. Available: <https://articles.uie.com/experiencedesign/>
- [18] A. Cooper, R. Reimann, D. Cronin, and C. Noessel, “About Face: The Essentials of Interaction Design, 4th Edition,” pp. 508, 509, 549, 2014.

- [19] C. Banga and J. Weinhold, *Essential Mobile Interaction Design: Perfecting Interface Design in Mobile Apps*. Addison Wesley, 2014.
- [20] Pete Hunt, “Why did we build react?” [Online]. Available: <https://facebook.github.io/react/blog/2013/06/05/why-react.html>
- [21] Facebook Open Source, “React Introduction.” [Online]. Available: <https://facebook.github.io/react/>
- [22] F. Lardinois, “Facebook Open-Sources React-Native,” 2015. [Online]. Available: https://techcrunch.com/2015/03/26/facebook-open-sources-react-native/?ncid=rss&utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+Techcrunch+%28TechCrunch%29
- [23] B. Eisenman, “React Native Introduction,” 2016. [Online]. Available: <https://www.infoq.com/articles/react-native-introduction>
- [24] Redux, “Three Principles Redux.” [Online]. Available: <http://redux.js.org/docs/introduction/ThreePrinciples.html>
- [25] —, “Redux Actions.” [Online]. Available: <http://redux.js.org/docs/introduction/ThreePrinciples.html>
- [26] —, “Redux Reducers.” [Online]. Available: <http://redux.js.org/docs/basics/Reducers.html>
- [27] React Navigation, “React Navigation Documentation.” [Online]. Available: <https://reactnavigation.org/docs/intro/>
- [28] “React Navigators.” [Online]. Available: <https://reactnavigation.org/docs/navigators/>
- [29] Geeky Ants, “Native Base.” [Online]. Available: <https://nativebase.io/>
- [30] F. Bento, “Open source ERP’s i18n,” in *ACM SIGDOC European Chapter/ Eurosigdoc Workshop on Open Source and Design of Communication - OSDOC ’10*, 2010, p. 49. [Online]. Available: <http://www.scopus.com/inward/record.url?eid=2-s2.0-79952512818&partnerID=40&md5=b82a46fdd1e07905be1833a5f59772fe%5Cnhttp://portal.acm.org/citation.cfm?doid=1936755.1936771>
- [31] B. Ediger, *Advanced Rails*. O’Reilly Media, 2008. [Online]. Available: <http://proquestcombo.safaribooksonline.com/book/web-development/ruby/9780596510329/firstchapter>
- [32] A. Zaytsev, “i18n for React Native.” [Online]. Available: <https://github.com/AlexanderZaytsev/react-native-i18n>
- [33] “Moment.js.” [Online]. Available: <http://momentjs.com/>
- [34] Facebook Open Source, “React-Native Networking.” [Online]. Available: <https://facebook.github.io/react-native/docs/network.html>
- [35] Mozilla Developer Network, “Javascript Promise.” [Online]. Available: https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Promise
- [36] —, “Using Fetch.” [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch
- [37] Just Arrived, “Just Match API Github.” [Online]. Available: https://github.com/justarrived/just_match_api
- [38] S. Klabnik, Y. Katz, D. Gebhardt, T. Kellen, and E. Resnick, “JSON API 1.0.” [Online]. Available: <http://jsonapi.org/>

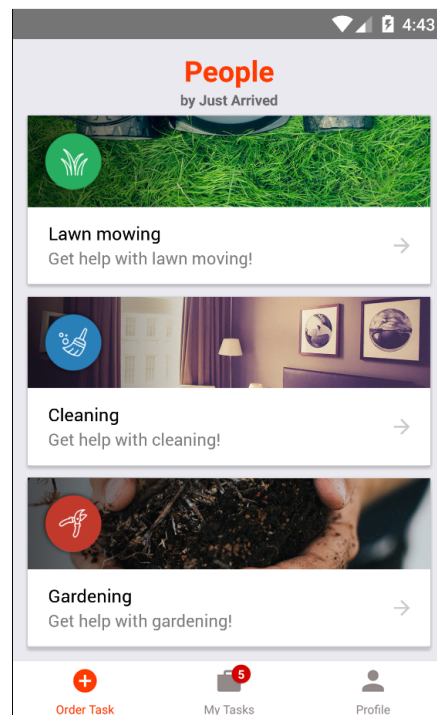
- [39] Just Arrived, “Just Arrived API documentation.” [Online]. Available: https://api.justarrived.se/api_docs
- [40] Taskrabbit, “TaskRabbit.” [Online]. Available: <https://www.taskrabbit.com>
- [41] Freelancer, “Freelancer.” [Online]. Available: <https://www.freelancer.com>
- [42] Welcome Movement, “Welcome Movement.” [Online]. Available: <http://welcomemovement.se>
- [43] Spotify, “Spotify.” [Online]. Available: <http://spotify.com>
- [44] D. M. Brown, *Communicating Design: Developing Web Site Documentation for Design and Planning*. New Riders, 2010, vol. 39, no. 1.
- [45] “Sus Lyckvi.” [Online]. Available: <http://cse.gu.se/om/interaktionsdesign/forskare/sus-lyckvi>
- [46] N. C. Zakas, “ESLint.” [Online]. Available: <http://eslint.org/docs/about/>
- [47] Airbnb, “Airbnb Eslint.” [Online]. Available: <https://github.com/airbnb/javascript>
- [48] Git, “Git.” [Online]. Available: <https://git-scm.com/>
- [49] Github, “Github.” [Online]. Available: <https://github.com/justarrived/p2p-client>
- [50] V. Driessen, “Git Flow.” [Online]. Available: <http://nvie.com/posts/a-successful-git-branching-model/>
- [51] Git, “Pull Request.” [Online]. Available: <https://git-scm.com/docs/git-request-pull>
- [52] Ebert, “Ebert.” [Online]. Available: <https://ebertapp.io/>
- [53] Google, “Material Design Components.” [Online]. Available: <https://material.io/guidelines/patterns/navigation.html#>
- [54] —, “Material Design Cards.” [Online]. Available: <https://material.io/guidelines/components/cards.html#>
- [55] —, “Material Design Color and Contrast.” [Online]. Available: <https://material.io/guidelines/usability/accessibility.html#accessibility-color-contrast>
- [56] —, “Material Design Onboarding.” [Online]. Available: <https://material.io/guidelines/growth-communications/onboarding.html#onboarding-top-user-benefits>
- [57] J. Saito, “Making a case for letter case,” 2016. [Online]. Available: <https://medium.com/@jsaito/making-a-case-for-letter-case-19d09f653c98>
- [58] Google, “Material Bottom Navigation.” [Online]. Available: <https://material.io/guidelines/components/bottom-navigation.html>
- [59] Apple, “ios Tab Bars.” [Online]. Available: <https://developer.apple.com/ios/human-interface-guidelines/ui-bars/tab-bars/>
- [60] icons8, “Icons8.” [Online]. Available: <https://icons8.com/>
- [61] Google, “Material Design Navigation Drawer.” [Online]. Available: <https://material.io/guidelines/patterns/navigation-drawer.html>
- [62] Firebase, “Firebase.” [Online]. Available: <https://firebase.google.com/>
- [63] Just Arrived, “Just Match API.” [Online]. Available: https://github.com/justarrived/just_match_api
- [64] “React Navigation Introduction.” [Online]. Available: <https://reactnavigation.org/blog/2017/1/Introducing-React-Navigation>

A

All Views of the Application



(a) Login screen



(b) Choose task type

Figure A.1: Login screen and choose task type screen.

A. All Views of the Application

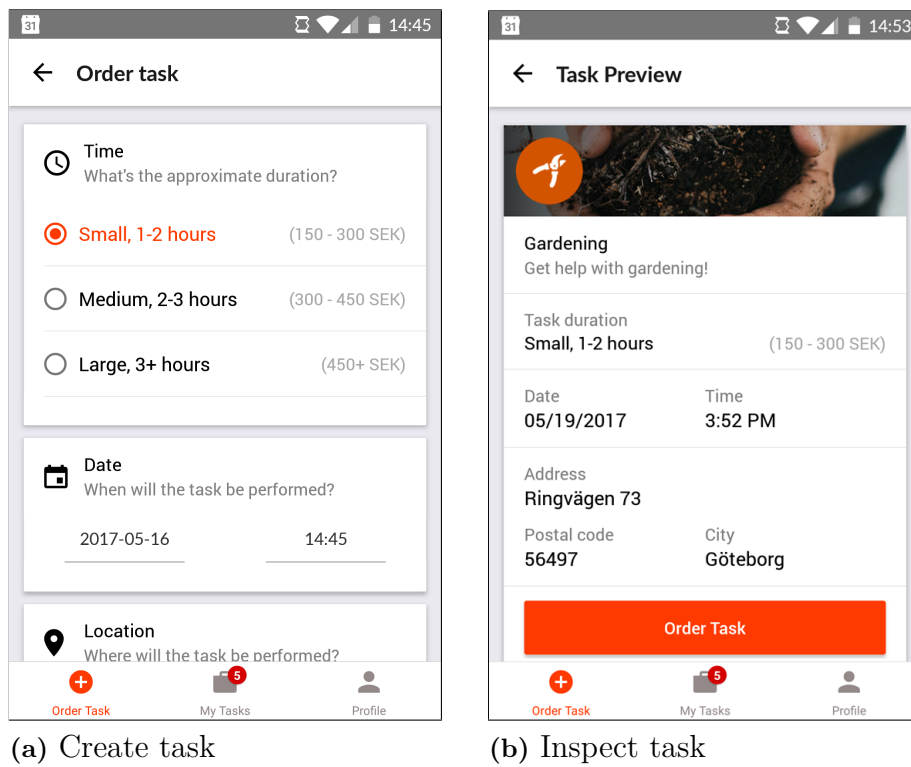


Figure A.2: Create and inspect task information.

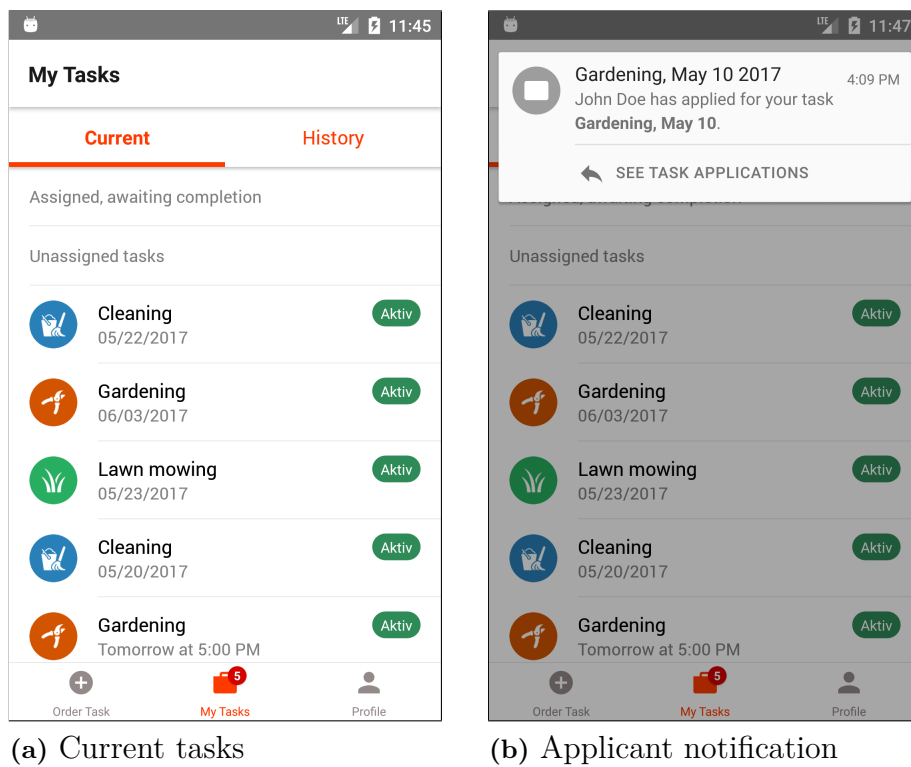


Figure A.3: View owned tasks and receive a push notification.

A. All Views of the Application

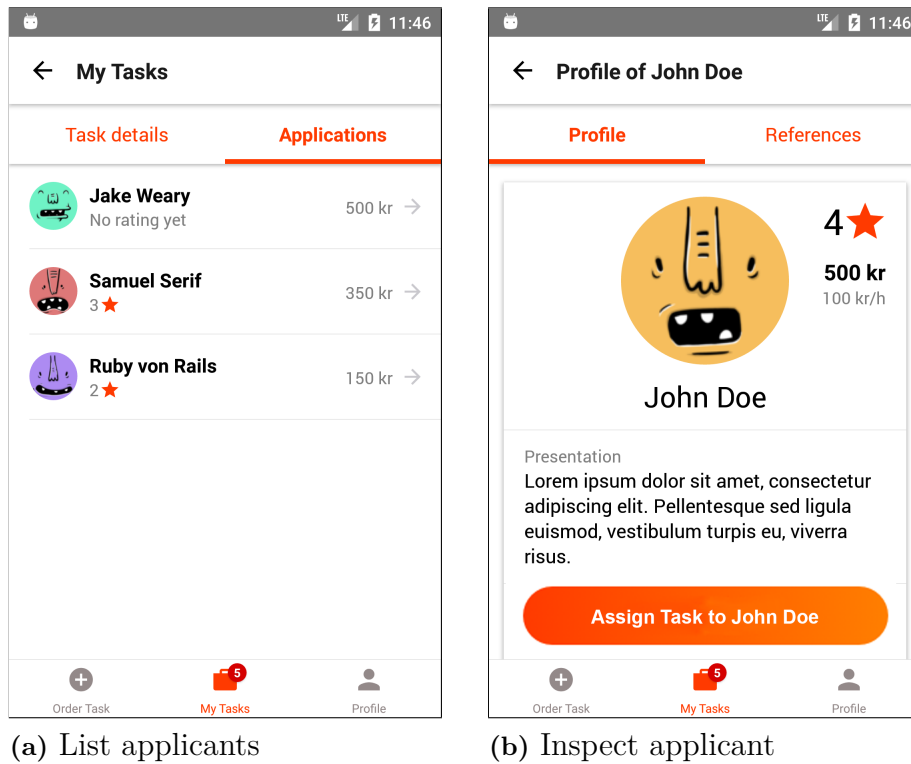


Figure A.4: Views listing and displaying applicant information.

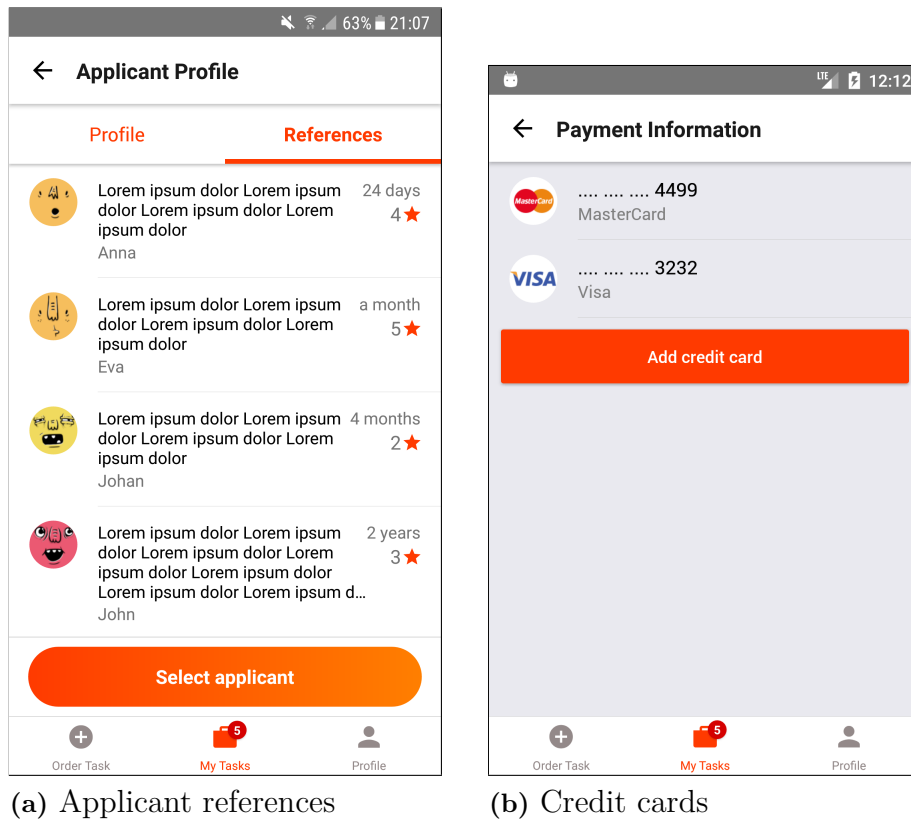


Figure A.5: Views displaying applicant references and listing credit cards.

A. All Views of the Application

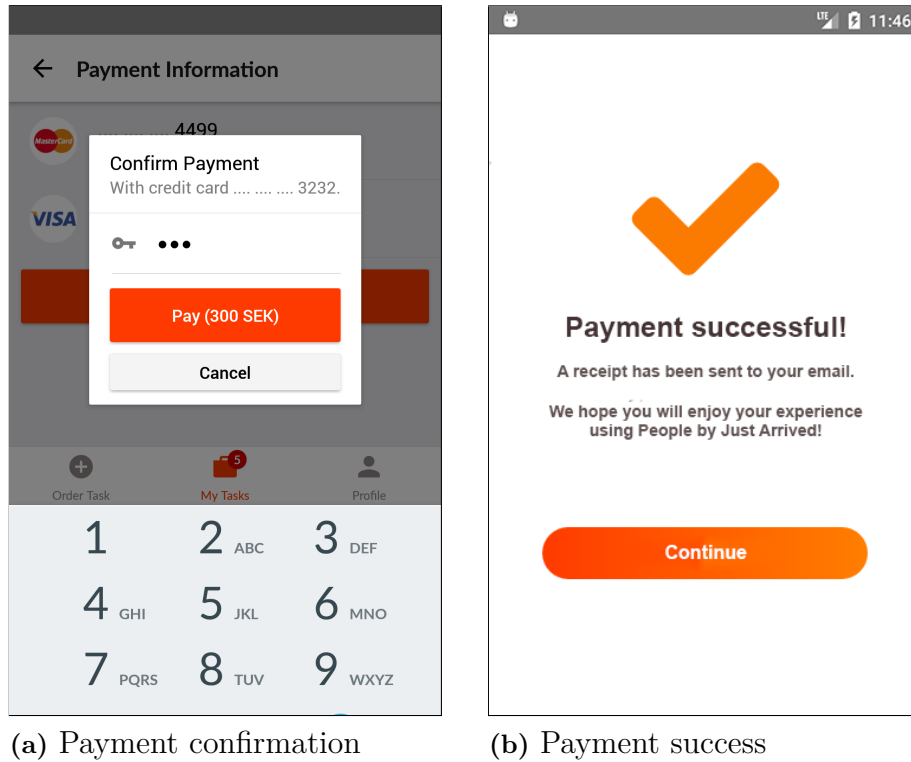


Figure A.6: Payment confirmation and payment success screens.

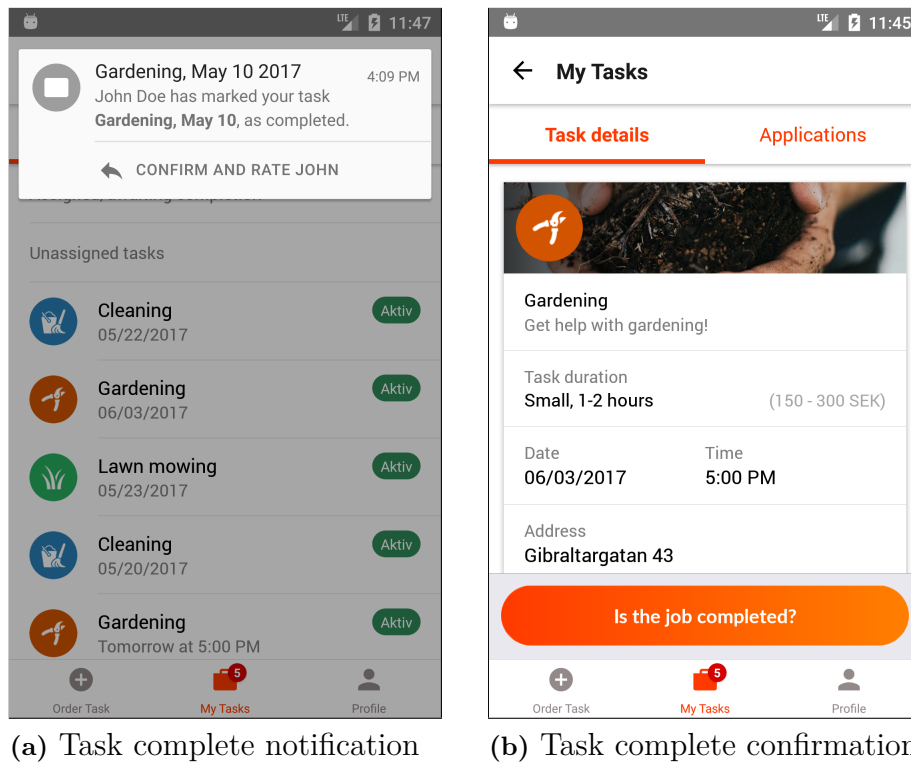
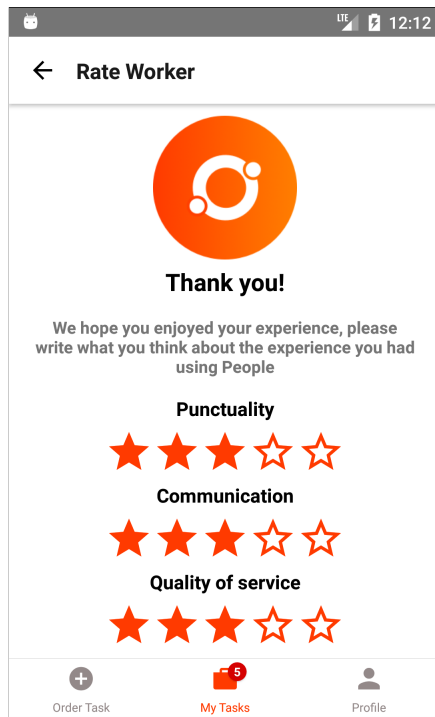
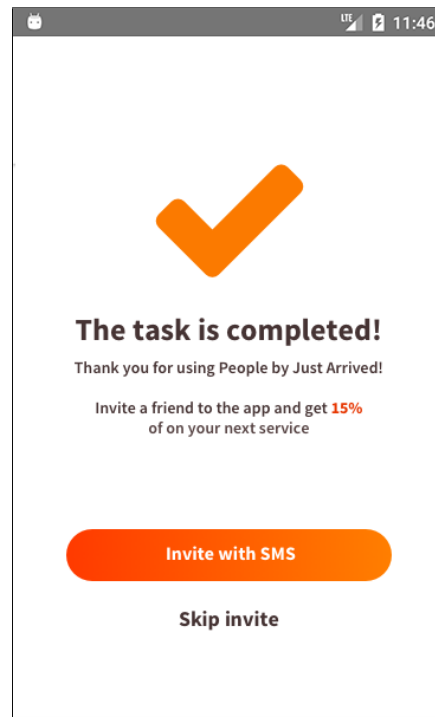


Figure A.7: Task completion notification and confirmation screen.

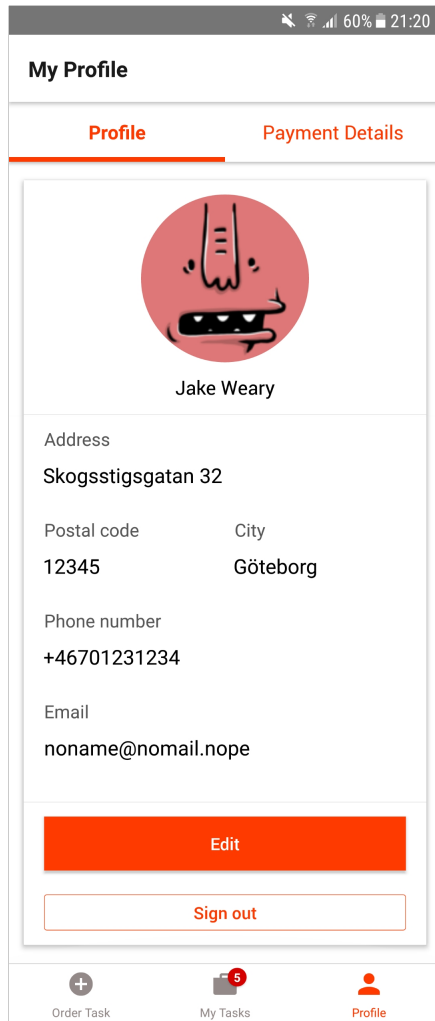


(a) Rate worker

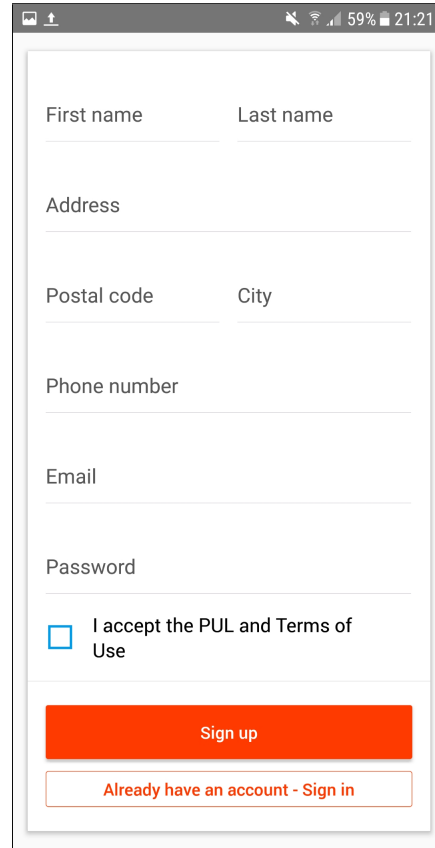


(b) Task complete screen

Figure A.8: Rate worker and task completed screens.



(a) User profile



(b) Account creation

Figure A.9: User profile and account creation screens.