Masaryk University
Faculty of Informatics

Receipt database with OCR scan

Bachelor's Thesis

Petr Janík

Brno, Spring 2021

Masaryk University
Faculty of Informatics

Receipt database with OCR scan

Bachelor's Thesis

Petr Janík

Brno, Spring 2021

This is where a copy of the official signed thesis assignment and a copy of the Statement of an Author is located in the printed version of the document.

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Petr Janík

Advisor: Ing. Lukáš Grolig

Acknowledgements

I would like to thank my supervisor Ing. Lukáš Grolig for bringing up an incredibly interesting assignment for this thesis.

Secondly, I would like to acknowledge all my friends, family and colleagues who sent me images of receipts and took part in the survey.

I would especially like to thank my sister for helping me proofread the thesis.

I wish to express my deepest gratitude to my girlfriend for her endless support, encouragement and advice.

Abstract

This thesis aims to develop a mobile and desktop application to store sales receipts in a digitalised form, removing the need to keep the original. The application uses optical character recognition to extract data from a receipt image and pre-fill the database record. It utilises word embeddings to assign emojis to receipt items and image processing techniques to transform the receipt photo into a more readable form.

The theoretical part introduces React Native in the context of other cross-platform development frameworks.

An emphasis is put on testing and continuous integration as an integral part of creating reliable software.

Keywords

React Native, OCR, mobile development, Android, Windows, testing, continuous integration, natural language processing, word embedding, machine learning, sales receipts, cross-platform development, image processing, OpenCV

Contents

Introduction **1**

1 Existing solutions **3**

2 Image data extraction **5**
2.1 Optical character recognition 5
2.2 Data extraction from text 6

3 Choice of platforms **7**

4 Choice of technology **8**
4.1 Xamarin 8
4.2 .NET Multi-platform App UI 8
4.3 Flutter 9
4.4 Capacitor 9
4.4.1 Electron 9
4.5 React + PWA + Electron 9
4.6 React Native 10

5 React Native in detail **12**
5.1 Creating new React Native application 12
5.1.1 Ignite 12
5.1.2 Expo 12
5.2 Native Modules 13
5.3 Platform-specific code 15

6 Development **17**
6.1 Version control system commit messages 17
6.2 Component library 17
6.3 Logging 21
6.4 Testing 22
6.4.1 Unit tests 22
6.4.2 Component tests 22
6.4.3 Snapshot testing 23
6.4.4 Visual testing 24
6.4.5 End-to-end testing 27

6.5 Continuous integration 28

7 Implementation **30**
7.1 Storage 30
7.1.1 Textual data 30
7.1.2 Image data 32
7.2 Authentication 33
7.3 Adding a receipt 35
7.3.1 Uploading image from Windows 36
7.4 Form Recognizer 38
7.5 Python API 39
7.5.1 Item categorization 39
7.5.2 Image processing 41
7.5.3 Deployment 48
7.6 Styles 49
7.6.1 Dark mode 49
7.7 Real-time user interface 51
7.8 Offline support 52
7.9 Features 52

8 Financial sustainability **55**

9 Evaluation **58**
9.1 Emoji assignment 59
9.2 Image processing 60

10 Conclusion **62**

A Appendix **63**
A.1 Attachments 63
A.2 Authentication activity diagrams 64
A.3 Version control system commit messages 67

Bibliography **68**

List of Tables

7.1 Comparison of pre-trained word embeddings models. 42

List of Figures

6.1 Storybook for React Native running on a device. 19
6.2 Storybook for Web running in a browser. 21
6.3 Chromatic user interface for reviewing visual change of a component. The scan icon moved up. 25
6.4 Image generated by Loki indicating a visual change to the component. The scan icon moved up. 26

7.1 Data model of Receipts Scanner. 31
7.2 Login screen of Receipts Scanner on Android. 34
7.3 Settings screen of Receipts Scanner on Android. 35
7.4 Activity diagram of adding a new receipt. 37
7.5 Comparison of an image before and after Gaussian blur. 43
7.6 Comparison of a fixed and Otsu's thresholding. 44
7.7 Comparison of an image before and after dilation. 45
7.8 Photo of a receipt with detected corners. 46
7.9 Result of adaptive thresholding applied on an image without and with Gaussian blur. 47
7.10 Comparison of the original and the processed image. 48
7.11 User interface of Receipts Scanner on Android in the dark mode. 50
7.12 User interface of Receipts Scanner on Windows in the dark mode. 51

8.1 Number of stored receipts by the number of users. 56
8.2 Operational costs of using Form Recognizer compared to other services. 58

9.1 Results of a survey evaluating how accurate the emojis assigned to the receipt items were. 60
9.2 Results of a survey comparing which image of a receipt was more readable. 61

A.1 User sign-in activity diagram. 64
A.2 User sign-up activity diagram. 65
A.3 Google authentication activity diagram. 66
A.4 Meaning of emojis in commit messages. 67

Introduction

During everyday life, people take part in transactions. Be it grocery shopping or refuelling of a car. Despite more and more goods being offered online, offline transactions still form an important part of the overall purchases. Proof of such transactions is, in most cases, a receipt. This proof is necessary when complaining or returning goods. However, it can get tough to keep track of all the receipts, and people can be rejected when asking for a refund simply because they either did not keep their receipt or put it among all the other receipts and cannot find it. The receipt can also get damaged in such a way that it is not readable anymore.

The goal of my thesis is to develop an application that would serve as a user's digital receipt database. It would allow the user to take a photo of a receipt, and by using optical character recognition, it would extract the textual information from the photo. This extracted information would be used to pre-fill the receipt record. The receipt records should be editable and searchable, e.g. by the merchant name or items bought. The application should be available on Android and Windows.

In the first chapter, I describe currently available applications trying to solve this task, their main features and drawbacks.

In the second chapter, I provide an overview of techniques used for data extraction from images.

The next chapter focuses on choosing target platforms for which the application should be developed.

The fourth chapter introduces available technologies for developing cross-platform applications. It also provides an overview of the chosen framework — React Native — which is further elaborated in the subsequent chapter.

An essential part of the thesis is testing and continuous integration. The types of testing used to ensure the high quality and reliability of the application are described in the sixth chapter.

The seventh chapter describes the implementation and functionality of the application. It shows how Microsoft Form Recognizer [1] was used to extract the information from a receipt photo. This chapter also includes a description of a receipt item categorization using word

embeddings. The image processing using the OpenCV[1] library is featured as well.

The subsequent chapter is concerned with the business model of Receipts Scanner. It elaborates the costs of third-party services used in Receipts Scanner and proposes solutions for financing the application.

The ninth chapter provides an evaluation of the application.

The outcome of the thesis is an application named Receipts Scanner. It can be used to conveniently organize receipts and access them both from a mobile device and desktop.

The patterns used in this application can be used for developing other React Native applications.

1. https://opencv.org/

1 Existing solutions

Applications addressing the need for unified digital storage of receipts are usually referred to as expense managers. The most popular expense manager for iOS[1] is called Smart Receipts[2]. It is also available on Android[3], but it is superseded by an application called Receipt Scanner[4] in terms of downloads. They allow users to take photos of receipts and fill in additional information such as total price, date of purchase or expense category. These receipts can then be searched by name, price or category. Both applications provide an option to organize expenses into reports. Users can export these reports in the CSV[5] format; Smart Receipts also supports PDF export. Another feature is a visualization of expenses throughout time and across categories.

The main feature I will focus on is the automatic information extraction from the receipts. It is available for $0.1 per scan in Smart Receipts, as of April 2021, and it is free in Receipt Scanner. Other applications that offer this feature are Expensify[6], Receipt Lens[7] and Dext[8]. All are available for Android and iOS. The amount of information extracted from the picture is, however, limited. Usually, it includes only merchant name, total paid amount, tax amount, date, currency and category. Any information about individual items is missing.

---

1. Mobile operating system developed by Apple.
2. `https://www.smartreceipts.co/`
3. Mobile operating system developed by Open Handset Alliance and sponsored by Google.
4. Receipt Scanner on Google Play:
`https://play.google.com/store/apps/details?id=com.easyexpense/`.
5. Comma-separated values.
6. Expensify on App Store:
`https://apps.apple.com/au/app/expensify-receipts-expenses/id471713959`.
7. Receipt Lens on App Store:
`https://apps.apple.com/us/app/receipt-lens-receipt-scanner/id1424280764`.
8. Dext on App Store:
`https://apps.apple.com/us/app/dext-finance-and-accounting/id418327708`.

An open-source[9] alternative to the mentioned applications is Receipt Manager[10]. It uses Tesseract[11] for text recognition from the image and a fixed set of rules to extract the information. The main idea of this project is that it does not rely on any third-party services. It is available on iOS, Android and the Web. All these clients connect to a server on which the data extraction runs. It is deployed separately as a Docker container[12].

9. Open-source software is software whose source code is released with permission to use, copy, distribute and modify [2]. It is usually free to use. It is often developed collaboratively on platforms such as GitHub.
10. Receipt Manager on GitHub: `https://github.com/ReceiptManager`.
11. Tesseract is a library and a command line program for extracting textual data from images. Tesseract on GitHub: `https://github.com/tesseract-ocr/tesseract`.
12. Docker containers are Docker container images run by Docker Engine. Docker container image is a standalone, executable package of software that includes everything needed to run an application. It will always run the same, regardless of the computing environment. [3] See `https://www.docker.com/`.

2 Image data extraction

On a high level, the data extraction from the photo of a receipt involves two steps: optical character recognition and data extraction from text.

2.1 Optical character recognition

First, the raw textual data need to be extracted. This is done using Optical Character Recognition (OCR). OCR is the process of translating images of handwritten, typewritten, or printed text into a format understood by machines. It works by pre-processing the image[1] and segmenting the text into lines, then words and eventually into individual characters to which a code, such as ASCII, is assigned. [4]

The code can be assigned based on template matching. An image of an unknown character is compared to a set of images of known characters. The code of the most similar character is chosen. The similarity can be computed as a difference between the pixel's grayscale value in the unknown character and a pixel's grayscale value in the known character [5]. This approach works well[2] for receipts because the characters are printed and are thus almost the same, except for minor deficiencies in printing. Receipts also usually use the same or a very similar font.

Another approach is to extract the most relevant information from the character image. This is called feature extraction. An example of such a feature could be the density of dark pixels in a given part of the image. These features are used as inputs for machine learning models such as K-Nearest Neighbors and Support Vector Machines[3]. [6]

The third approach is to use a Convolutional Neural Network. It has the highest computational cost among the three mentioned approaches and requires a large dataset of images to learn on, but it has the best performance. [4]

---

1. Image pre-processing refers to a set of techniques (e.g. noise reduction), that are applied to the image to make it more suitable for the subsequent steps.
2. The template matching technique reached a recognition rate of 97.36% in [5].
3. Supervised machine learning models for classification and regression problems.

2.2 Data extraction from text

The second step is to assign a meaning to the parts of the extracted text. This can be done by pattern matching, dictionary matching or using more complex methods such as neural networks.

Pattern matching uses the fact that some data have a specific format compared to other data available on the receipt. For example, dates and phone numbers can be extracted using regular expressions.

Dictionary matching tries to find a value from a list of possible values. An example would be finding a merchant name in a predefined list of merchants.

All methods can take advantage of the actual text position. The merchant name is most likely to appear on the top, the individual items in the middle and the total price at the bottom. However, receipts vary significantly in the locations of different data. Therefore this method, referred to as Zone OCR, is more suitable for forms and invoices where the format is known.

3 Choice of platforms

According to Pew Research Center [7], as of 2021, 85% of adults in the United States own a smartphone. There is no dispute about smartphone being a ubiquitous device in people's lives. Most smartphones have a decent camera that allows taking photos of receipts in sufficient quality. Hence, most of the focus has been on developing a mobile application. The next question that needs to be answered is whether to target Android, iOS or both. According to IDC's market research [8], in 2020, 84.1% of smartphones were equipped with Android and 15.9% with iOS. The development of iOS applications requires a Mac[1] and Xcode[2]. Some workarounds exist around this limitation, but most of them rely on using a Mac in the cloud. At the beginning of the writing of the thesis, I did not have access to Mac. In order to be able to target more devices and given the mentioned limitation, I chose to develop an Android application.

Although smartphones are convenient for taking the photo of the receipt right after the purchase, some users might prefer a desktop application to view and search the receipts. According to Statcounter [9], Windows[3] operating system makes for 75.55% of desktop operating systems, macOS[4] comes second with 16.5%[5]. For the same reasons (ability to cover more devices and having Windows PC), I chose Windows as the supported desktop operating system.

1. Mac, also known as Macintosh, is a family of personal computers sold by Apple. Current models include MacBook Pro and iMac.
2. Xcode is software by Apple for developing applications for Apple products. See `https://developer.apple.com/xcode/`.
3. Group of operating systems developed by Microsoft.
4. Desktop operating system developed by Apple.
5. Data have been collected based on website views.

4 Choice of technology

As established in the previous Chapter 3, the targeted platforms are Android and Windows. The traditional approach would be to implement a separate application for each platform, for Android using Java or Kotlin, and for Windows using Universal Windows Platform (UWP) or Windows Presentation Foundation (WPF). This approach suffers from the apparent disadvantage of writing the code twice. The functionality of the application differs only slightly on Android and Windows, so the preferred solution is to have a cross-platform application. A cross-platform application is an application that can run on multiple platforms without a need for having a separate code base for each.

There are many available frameworks that aid cross-platform development.

4.1 Xamarin

"Xamarin is an open-source platform for building modern and performant applications for iOS, Android, and Windows with .NET." [10] With Xamarin, the business logic and backend code are written in C# and fully shared. Xamarin.Forms extends this code reuse to the user interface (UI). The user interface is defined using XAML[1]. It targets iOS, Android, and Windows.

4.2 .NET Multi-platform App UI

.NET MAUI is an evolution of Xamarin.Forms. It targets Android, iOS, Windows and macOS. It is still in a preview version[2].

1. eXtensible Application Markup Language is a markup language based on XML used to define user interfaces [11].
2. Roadmap of.NET MAUI: `https://github.com/dotnet/maui/wiki/Roadmap`.

4.3 Flutter

"Flutter is Google's UI toolkit for building beautiful, natively compiled applications for mobile, web, and desktop from a single codebase." [12] It uses Dart programming language. The desktop support is available as a beta release [13].

4.4 Capacitor

Capacitor is an open-source native runtime for building Web Native[3] apps. It enables development of cross-platform iOS, Android, and Progressive Web Apps [15]. Capacitor is developed by Ionic. The application is built using web technologies (JavaScript[4], HTML[5], and CSS[6]) and rendered using Web View. Web View resembles a full-screen browser, but it is not limited to browser APIs[7]. It can call native APIs such as sensors and storage.

It is possible to create a desktop application by wrapping the web application inside Electron. Capacitor has built-in support for Electron.

4.4.1 Electron

Electron is a framework that enables creating desktop applications with JavaScript, HTML, and CSS. These applications can then be run on macOS, Windows, or Linux [19]. Electron is used, for example, by Visual Studio Code and Slack.

4.5 React + PWA + Electron

React is a JavaScript library for building user interfaces created by Facebook. It is one of the most used libraries for building web applications.

3. A Web application that accesses native APIs [14].
4. JavaScript is a scripting language targeted specifically to the Internet [16].
5. Hypertext Markup Language is a document-formatting language used to create Web pages [17].
6. Cascading Style Sheets add style (e.g., fonts, colors, spacing) to Web documents [18].
7. Application programming interface.

It uses JSX, which is a combination of JavaScript and XML[8], for example, `const element = <h1>Hello {'react'.toUpperCase()}</h1>;`. React is declarative and component-based, which means that complex UIs are composed of multiple simpler, independent and reusable components. The components are parametrized with properties (very often called *props*).

React itself is not a cross-platform framework. However, if we release the constraint of having a native mobile application, it is possible to create a progressive web application (PWA). Most React starter templates provide this functionality. The PWA looks very similar to the native application. It is installed directly from a browser by visiting the website that provides the PWA. PWA can be installed both on mobile and desktop. In a similar manner to Capacitor, desktop support is possible by wrapping the React application inside Electron.

4.6 React Native

React Native is a JavaScript framework created by Facebook. It is based on React, but instead of targeting the browser, it targets mobile platforms — Android and iOS [21]. There are community-supported projects[9] that bring React Native to other platforms, such as React Native Windows, React Native macOS and React Native Web. It is used by companies such as Facebook, Instagram and Shopify.

Unlike Capacitor, which uses Web Views, React Native is compiled into native code, allowing developers to create applications that look and feel like native applications written in Java/Kotlin (for Android) and Objective-C/Swift (for iOS). It also comes with performance benefits.

As opposed to React, which uses elements that correspond to HTML elements, e.g. `<div>` and `<p>`, React Native uses platform-agnostic elements such as `<View>` and `<Text>`. Those are then translated to native UI elements. It also uses its own abstraction of CSS.

8. Extensible Markup Language is a W3C open standard for describing data using embedded tags [20].
9. `https://reactnative.dev/docs/out-of-tree-platforms`

React Native exposes JavaScript interfaces for platform APIs so that the applications can access platform features like the camera or the user's location.

By default, React Native can run on iOS and Android. To run on Windows, React Native Windows is installed into an existing React Native application. After compiling the React Native code, the result will be a UWP application, which can run on Windows PC and other UWP supported devices such as Xbox.

React Native's motto *Learn once, write anywhere.* means that when a developer learns the concepts and syntax of React Native, they should be able to write programs for any platform (that React Native supports). This motto is not to be confused with *Write once, run anywhere.*, which would promise that the same code would be able to run on any platform. However, during the time, and with lots of community efforts, React Native has developed into a framework whose users usually report around 90% code reuse, which means that only 10% of code needs to be platform-specific.

React Native provides easy ways to write platform-specific code that does not affect other platforms. This concept is described in Section 5.3.

I chose to develop Receipts Scanner in React Native for its previously mentioned benefits, i.e. native controls, performance, a wide range of supported platforms and also due to my familiarity with React.

5 React Native in detail

5.1 Creating new React Native application

The official way of creating a new React Native application is to use React Native CLI[1]. The application produced by the CLI contains only one component. It does not use any additional libraries apart from a linter[2] and a testing framework.

Developing applications with React Native is often repetitive, and the same setup needs to be made for every application. To reduce the time it would take to install all required libraries and to enforce the best practices, application boilerplates have been created.

5.1.1 Ignite

The most used boilerplate for React Native is Ignite[3]. Ignite is a CLI and an application boilerplate at the same time. It is opinionated, meaning that it enforces specific project structure and libraries. Navigation, application state management, Storybook component library[4], Detox end-to-end tests[5] and Reactotron[6] (for inspecting the application) are already set up. The CLI can be used for generating new components and screens.

5.1.2 Expo

Expo[7] is a framework built on top of React Native. An application created with Expo CLI can run on iOS, Android and the Web. It provides many external tools that help with the development process. Many libraries for React Native have been created by Expo. Those

1. Command Line Interface
2. A static analysis tool that highlights errors and enforces good programming practices.
3. Ignite on GitHub: `https://github.com/infinitered/ignite`.
4. Described in Section 6.2.
5. Described in Section 6.4.5.
6. Described in Section 6.2.
7. `https://docs.expo.io/`

libraries can be often used also in applications that do not run on Expo.

It is not possible to add a React Native for Windows to the Expo project. For this reason, Receipts Scanner was set up with the React Native CLI instead.

Expo will likely support React Native for Windows in the future[8].

In the meantime, the Expo built for the Web could be run on Windows using Electron[9].

5.2 Native Modules

Native modules provide access to the native platform API that is not available by default in JavaScript [22], for example, an API to select a picture on the device. Other reasons for using native modules are taking advantage of existing native libraries or running multi-threaded code.

Native modules for Android are written in Java or Kotlin. For Windows, they can be written in C++ or C#, but C++ is the preferred language, primarily due to its performance.

Native modules can be written directly in the application or added as an external package. React Native Directory[10] provides a list of available native modules. They can be filtered by the supported platform because the native module is often unavailable on all platforms that React Native supports.

In my opinion, not having a native module's functionality available for a given platform is the most considerable drawback of using React Native. Most native modules are made for Android and iOS only and cannot be therefore used on the Web or Windows. The missing Windows support is an obstacle encountered many times during Receipts Scanner development.

8. Feature request to support React Native for Windows in Expo: `https://expo.canny.io/feature-requests/p/support-for-react-native-windows-apps`.
9. Expo has been introduced in Section 4.4.1.
10. `https://reactnative.directory/`

In case a native module is not available on a given platform, there are four available options:

1. **Omitting the module.**

   If the functionality provided by the native module is not critical, it might be omitted on a given platform. This is possible by having a different code for each platform. This concept is described in Section 5.3.

   In Receipts Scanner, this approach has been taken with Google Sign In[11], which is not available on Windows.

2. **Mocking the module.**

   In the context of this thesis, mocking means substituting an object or a whole module with a simpler implementation. The simpler object is called a mock.

   The approach of mocking a module has been applied in Receipts Scanner on NetInfo module, which provides information about the internet connection. Currently, it is not possible to use NetInfo on Windows[12]. Consequently, it has been temporarily mocked on Windows always to return information about the device being online.

3. **Replacing the module.**

   In case a different module that provides similar functionality is available, it can be used instead. If such a module does not exist, it can be implemented.

   Receipts Scanner implements its own *File Picker* module to provide the functionality of picking an image from the file system on Windows.

4. **Adding the missing platform to the module.**

   Most of the modules are open-source projects to which anyone can contribute. React Native is a relatively new ecosystem and

11. Authentication using Google, further described in Section 2.
12. GitHub issue:
`https://github.com/react-native-netinfo/react-native-netinfo/issues/306`.

relies on community contributions. In general, anyone can add a new platform to an existing library.

5.3 Platform-specific code

Sometimes it is necessary to have a different code for a different platform. The main reasons are:

- some platform does not support certain functionality, or the same functionality needs to be implemented differently,
- the application on a particular platform should behave differently,
- the styles need to be different.

React Native provides two ways of separating code by platform:

1. **`Platform` module**

   By writing `Platform.OS === 'windows'`, it is possible to determine if the code is running on Windows.

2. **Platform-specific file extensions**

   If there are two files, `Button.android.ts` and `Button.windows.ts` and the `Button` is imported as `import Button from './Button';`, the file with the extension of the platform on which the application is running will be picked.

In Receipts Scanner, the `Platform` module's functionality is used, for example, in the following places:

- to allow scanning of a receipt only on Android,
- to use different styles for each platform,
- to use Google Sign In only on Android,
- in Firebase adapter[13] to use a different Firebase library for each platform,

13. Described in Section 7.1.2.

- to use a different `Modal`[14] component for each platform.

Most of these were driven by libraries not having support for React Native Windows.

14. A component that appears on top overlaying other components. The background is usually grayed out.

6 Development

6.1 Version control system commit messages

When working with a version control system[1], it is good to indicate the type of source code change in a commit message. This can be done either verbally or using an emoji. Using emojis in commit messages provides an easy way of identifying the purpose or intention of a commit. In the Receipts Scanner repository, the notation in Figure A.4 has been used. It is loosely based on Gitmoji[2]. The Figure A.4 shows a Unicode[3] representation of an emoji and its meaning.

6.2 Component library

When developing a React application, it is a good practice to have a set of basic components from which more complex ones are built up. The set of those foundational components is called a component library or a design system. An efficient way of building the component library is by using a tool that allows the development of each component in isolation.

Storybook[4] is a widely used tool for developing UI components in isolation. It is based on plugins. Thanks to its open-source nature, there are many plugins created by the community, which extend the core functionality of Storybook.

Storybook is based on the concept of stories. A story is a permutation of the component with given properties. Each component can have multiple stories. A `Button` component could have one story displaying the Button in its *enabled* state and a second one for the *disabled* state.

Storybook stories serve as a gallery of available components and as a documentation of their properties. They can be used for snap-

1. Software for tracking changes in files, for example, Git.
2. Gitmoji is an initiative to standardize and explain the use of emojis on GitHub commit messages. See `https://gitmoji.dev/`.
3. *"The standard for digital representation of the characters used in writing all of the world's languages."* [23]
4. `https://storybook.js.org/`

shot and visual tests, which are described in Sections 6.4.3 and 6.4.4 respectively.

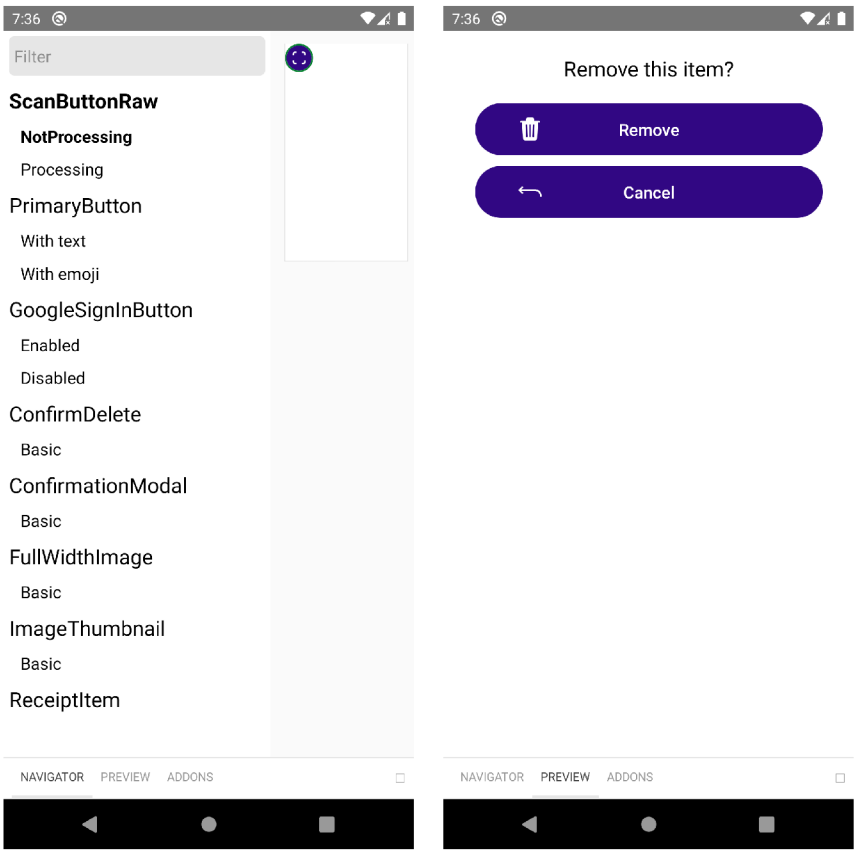
Storybook supports multiple platforms, including React and React Native. To better distinguish between the two, I will refer to Storybook for React as *Storybook for Web* because React is a web framework.

Whereas Storybook for Web runs in a web browser, Storybook for React Native needs to run on a mobile device because React Native uses native components that cannot be displayed in the browser.

Running Storybook on a mobile device is achieved by rendering the Storybook component instead of the root component that would normally be rendered. To not have to replace the component manually in code every time, Receipts Scanner takes advantage of Reactotron[5]. Reactotron is a desktop application that provides a suite of tools for easier development in React Native. It is integrated inside the project, so it is possible to switch between displaying Storybook and the actual application from the Reactotron UI.

Figure 6.1 shows the component library and the interface of Storybook for React Native.

5. Reactotron on GitHub: `https://github.com/infinitered/reactotron`.



(a) Components menu.

(b) Detail of *Confirmation Dialog* component.

Figure 6.1: Storybook for React Native running on a device.

It is possible to install a Storybook server that enables navigation between stories from a browser. However, the stories are still displayed only on mobile device. In order to show the stories in the browser, a react-native-web[6] library must be used. This library provides the same elementary components that are used to create React Native applications. The difference is that instead of translating those components into Android and iOS native components, it translates them into HTML elements. There are two options for running the Storybook in a browser:

6. `https://necolas.github.io/react-native-web/`

1. **Adding Web support to the whole application.**

   All react-native components are replaced by react-native-web components by aliasing react-native imports to react-native-web in the bundling[7] process of the whole application. This makes the whole application run in a browser. The Storybook functionality is, therefore, a side-effect.

   The Expo[8] applications can run in a browser by default. They are set up with react-native-web from the beginning. No extra configuration is necessary to have Storybook for React Native running in the browser.

2. **Installing Storybook for Web.**

   It is possible to install Storybook for Web and alias react-native imports to react-native-web in the bundling process of Storybook. This approach is used in Receipts Scanner. Figure 6.2 shows the component library and the user interface of Storybook for Web.

   An advantage of this approach is that even though Storybook for React Native is in version 5, the Storybook for Web can use the latest version 6 (as of May 2021). In version 6, all stories in the projects are found automatically without having to list each story explicitly. It also uses a more concise and more intuitive syntax for writing stories.

   Another advantage is that the Storybook for Web can be used with other libraries for web development, such as Chromatic for visual testing, which is described in Section 6.4.4. Storybook for Web can be built into static files and served as an application independent of the React Native application.

   The main disadvantage is that there are two separate Storybooks in the project. A small amount of extra code needs to be written for each story to make it visible in both Storybooks.

7. Bundling is the process of transpiling all source files and assets of the application, usually into one large JavaScript file that can be run across different browsers.
8. Expo has been introduced in Section 5.1.2.

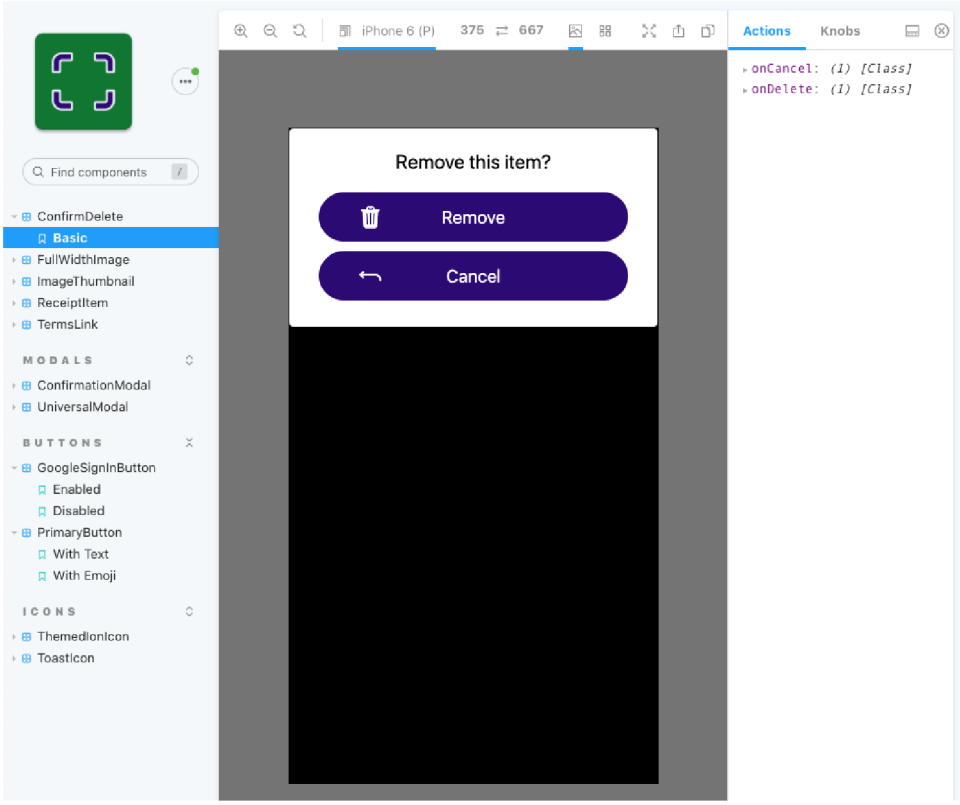


Figure 6.2: Storybook for Web running in a browser.

6.3 Logging

The logging in Receipts Scanner happens via methods of a `LOG` object. This abstraction ensures that the application does not depend on a particular logging system.

All logs are sent to the console in development mode. `WARN` and `ERROR` level messages are sent into Sentry[9], both in development and release mode. Sentry is a third party service that collects logs, errors, and events sent through the network. This enables monitoring of the application behaviour on users' devices.

9. `https://sentry.io/`

6.4 Testing

Testing is an integral part of developing a reliable software. The Receipts Scanner uses five types of tests: unit tests, component tests, snapshot tests, visual regression tests and end-to-end tests. Apart from visual tests, all tests use a JavaScript testing framework Jest[10] that is included by default in new React Native applications. The tests run in a Node.js[11] environment that mimics the environment of a React Native application.

6.4.1 Unit tests

Unit tests cover a single piece of functionality, usually individual functions. In Receipts Scanner, their main goal is to assure that the utility functions such as RGB to HEX color conversion or receipt filtering work.

Mocking

When the tested function or object has any dependencies, they are often mocked out.

6.4.2 Component tests

Component tests test individual React components. An example could be a test for a *Receipts List* component. Let us assume that the component calls an API and displays the returned receipts. The test would mock the API call by returning a fixed list of receipts and then assert that those receipts are visible.

In React Native, there is an option to choose between two libraries for component tests: react-test-renderer and React Native Testing Library.

react-test-renderer renders components to pure JavaScript objects without depending on the DOM[12] or a native mobile environment [24]. It provides methods for locating elements, for example, by component

10. `https://jestjs.io/`
11. Node.js is a JavaScript runtime. See `https://nodejs.org/`.
12. Document Object Model.

type (e.g. `Text`) or properties (e.g. finding a button with `title` value equal to *Submit*).

It is possible to get the rendered component as a JSON[13] and use it for snapshot testing. Snapshot testing is described in Section 6.4.3.

The issue of testing with react-test-renderer is that the tests rely on implementation details of the components. React Native Testing Library solves this problem by providing utility functions on top of react-test-renderer [26] for locating elements, e.g. by the visible text or accessibility label so that tests resemble how users interact with the application. It is inspired by React Testing Library [26], which is a widely used testing library for React.

React Native Testing Library also adds a `fireEvent` method that can simulate *press*, *scroll* and *text changed* events. Thus, both the rendering of the element and interaction with it can be tested.

Mocking

Because the components are rendered with react-test-renderer into pure JavaScript objects without depending on a native mobile environment, any native modules need to be mocked.

6.4.3 Snapshot testing

Snapshot tests verify that the rendered markup of the components stays the same. When the snapshot tests are run for the first time, the rendered markup is saved in a file. During subsequent runs, the newly rendered markup is compared to the saved one. The tests fail when there are differences between the two. Those differences are usually highlighted for the developer to assess whether or not this change was intentional. If it was, the saved snapshots are updated. Otherwise, the unintentional changes are fixed.

It is possible to create snapshot tests with Jest and react-test-renderer by rendering the component to JSON and comparing it to the previously rendered version[14]. This process can be automated with Storybook. Using a Storybook addon StoryShots[15], all stories in the project

13. JavaScript Object Notation is a lightweight data-interchange format [25].
14. See snapshot testing with Jest: `https://jestjs.io/docs/snapshot-testing`.
15. `https://www.npmjs.com/package/@storybook/addon-storyshots`

are turned into snapshot tests. This is a very efficient and inexpensive way of testing because the stories most of the time already exist.

Snapshot testing catches only changes in markup. If, for example, an image on a given URL changes, the snapshot tests will still pass.

On the other hand, changing markup does not necessarily mean that the result is different. It is often the case that even though the markup has changed, the page looks the same as before, and the tests report false positives. It can be hard to tell whether the change is intended, especially when dealing with large snapshots.

Mocking

Because the StoryShots addon uses react-test-renderer internally to render the components and save their snapshot, any native modules need to be mocked.

6.4.4 Visual testing

Visual testing addresses the issues of snapshot testing mentioned earlier. It is not crucial if the markup changed. The important thing is whether the component looks as desired. Visual tests, also called visual regression tests, assure that once the expected look of a component was achieved, it will stay unchanged.

Visual testing compares screenshots of newly rendered components with the old screenshots, also called reference images. When visual tests run for the first time, a set of screenshots is made. During subsequent runs, those screenshots are compared to the stored ones. Possible differences are highlighted. There are various tools that can highlight the differences between images. Those tools can be independent of tools used to take the screenshots.

A widely used tool for visual testing in React community is Chromatic[16]. It creates screenshots of the Storybook stories. It is not possible to use Chromatic with Storybook for React Native. Nonetheless, Receipts Scanner repository uses Storybook for Web too, by utilizing react-native-web. The Chromatic is therefore set up for testing components with Storybook for Web.

16. `https://www.chromatic.com/`

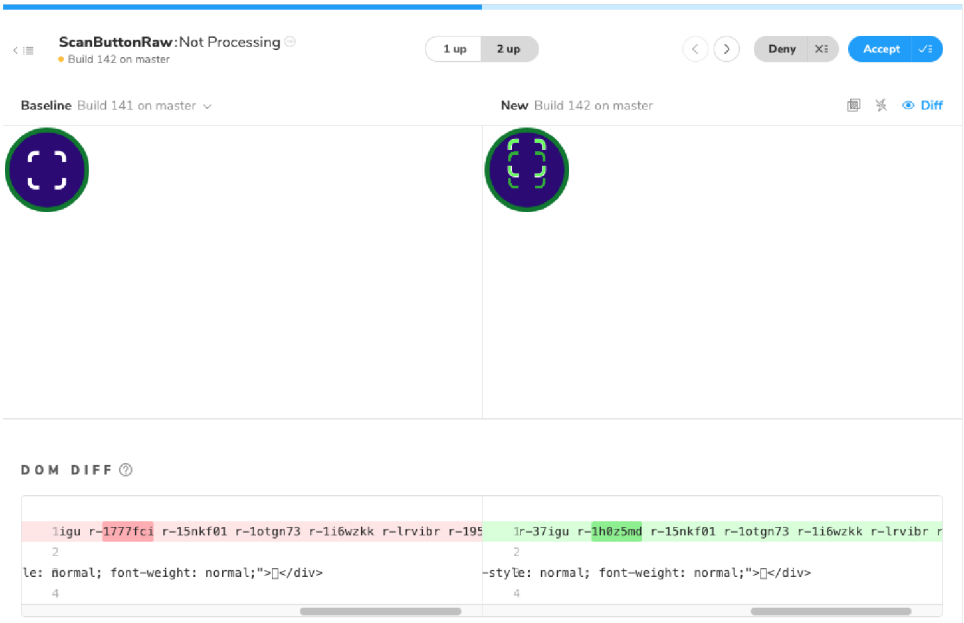


Figure 6.3: Chromatic user interface for reviewing visual change of a component. The scan icon moved up.

Chromatic runs in the continuous integration (CI) pipeline[17]. The changes need to be reviewed on the Chromatic website created for the given CI run. Chromatic also publishes the Storybook as a website.

Figure 6.3 shows how the review process of a visual change in Chromatic UI looks. Both the visual change and the change in the markup are visible.

An alternative to Chromatic is a tool called Loki[18], which also integrates with Storybook. It can take screenshots in a Chrome browser, on an iOS simulator or on an Android emulator, which means that it can be used both with Storybook for Web and Storybook for React Native.

Loki creates images that highlight the visual change. A visual change of a *Scan Button* component is shown in Figure 6.4.

17. A series of steps executed when changes are pushed to the remote project repository. Described in Section 6.5.
18. `https://loki.js.org/`

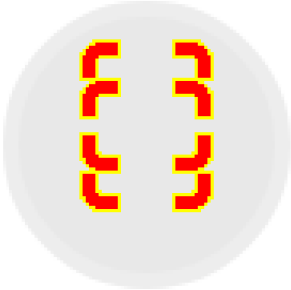


Figure 6.4: Image generated by Loki indicating a visual change to the component. The scan icon moved up.

I would not recommend using Loki for Storybook for React Native. The tests are unstable, and the screenshots differ even though the component itself has not changed. This is caused by the component not being fully loaded when the screenshot is made. Loki uses a running Storybook server for controlling which story is shown, which sometimes does not work, resulting in flashing between different stories. Loki performs better with Storybook for Web, but it cannot take screenshots of modal components.

The third approach of visual testing I tried was a combination of Storycap[19] and Reg Suit[20]. Similar to Chromatic, Storycap and Reg Suit work only with Storybook for Web.

First, Storycap takes screenshots of the stories. Reg Suit then downloads the previous screenshots from Google Cloud Storage, compares them with the new ones and stores the differences.

Storycap and Reg Suit should run in the CI pipeline. From my experience, Reg Suit had issues determining the hash of the previous commit. This hash is used to pick correct images from the Google Cloud Storage bucket for comparison. For this reason, all stories were being reported as new, and no comparison took place.

Naturally, it is not good to use all three solutions for visual testing together. I would suggest using only Chromatic because it has a fast setup, provides a clean user interface for reviewing the changes, and seems the most reliable solution among those I have tried.

19. Storycap on GitHub: `https://github.com/reg-viz/storycap`.
20. Reg Suit on GitHub: `https://github.com/reg-viz/reg-suit`.

Similarly to snapshot tests, the visual tests are created automatically from stories. The visual testing is therefore very convenient.

Mocking

Visual tests require Storybook to be running. If the Storybook is running on an Android device, no mocking is necessary. If Storybook for Web is used, native modules that do not have web support need to be mocked.

The need for mocking can be alleviated by separating the view part of the application from the business logic. The components can be divided into two categories: pure and connected. Pure components are components that display data based on their properties. Connected components connect to the database, make API calls and depend on native modules. They pass the data to pure components. In general, stories are written for pure components. This way, no mocking of native modules is needed.

6.4.5 End-to-end testing

End-to-end tests are used to test user interaction with the application from the beginning to the end.

An example of an end-to-end test would be testing the sign-in functionality, i.e. after the required fields of the sign-in form are filled in, and the *Sign in* button is pressed, a screen for the authenticated users is shown.

Multiple end-to-end testing frameworks for React Native exist. Receipts Scanner uses Detox[21] end-to-end tests. Detox is a popular choice for React Native testing, mostly for its speed and reliability. Apart from test logs, it can capture screenshots and videos of the running tests, providing greater insight into why tests are failing.

A more recent project, Cavy[22], takes a different approach from Detox. It does not use any native code. All functionality is achieved with pure JavaScript. This makes the setup more straightforward, but the functionality is limited.

21. Detox on GitHub: `https://github.com/wix/Detox`.
22. `https://cavy.app/`

Whereas Detox can locate elements by text or label, Cavy needs each component that is being tested to be wrapped with a special function. This creates noise in the code

The oldest testing framework of the ones mentioned is Appium[23]. Similar to Selenium WebDriver[24], it uses WebDriver protocol[25]. It could be used for testing both the Android and the Windows version of Receipts Scanner.

Appium tests are referred to as *black-box* tests. Black-box testing ignores the internal mechanism of a system and focuses solely on the outputs generated in response to selected inputs and execution conditions [27]. In practise this means, that when a certain element is expected to be visible, Appium polls until the element becomes visible or until a timeout occurs. This usually leads to flaky and non-deterministic tests.

Detox has been created to address this issue. Detox tests are so-called *gray-box* tests, which means that the tests are aware of the internal state of the application. Detox automatically waits until all asynchronous operations (such as querying a server or animations) end, and only then is the assertion triggered.

Mocking

Because the Detox tests run on the device (Android or iOS), the native modules need not be mocked. It is still possible to mock external dependencies to make the tests faster and more stable.

6.5 Continuous integration

Continuous integration is a practise of frequently merging code changes into a central repository. The changes are validated by creating a build and running automated tests. [28]

Receipts scanner takes advantage of GitHub Actions[26]. Every time new code is pushed to the main branch of the GitHub repository, three

23. `http://appium.io/`
24. Selenium is software for automated testing of web applications.
25. WebDriver protocol is used to control the browser.
26. `https://github.com/features/actions`

workflows are run. A workflow consists of multiple jobs that run in parallel. Each job consists of multiple sequential steps.

During the first workflow,

- the code is checked for any linting or type errors,
- a suite of unit and snapshot tests runs,
- Storybook is deployed and visual tests run,
- a release version of Android application is built,
- a release version of Windows application is built.

The second workflow runs end-to-end tests on an emulated Android device.

The third workflow runs tests of the Python API[27]. The workflows do not contain the Python API deployment logic. Instead, a repository hook is set up in the Google Cloud Platform Console[28] to deploy the latest version of the Python API.

27. Python API is described in Section 7.5.
28. A user interface to manage services provided by Google.

7 Implementation

7.1 Storage

The Receipt Scanner needs to store two kinds of data: textual data, i.e. data about the receipts and the users' credentials, and the receipt images.

7.1.1 Textual data

For textual receipt information, document-oriented databases are a perfect fit because they make it easy to store data in the same format and shape as they are used in the application.

The document-oriented database is a type of NoSQL[1] database that stores data in the form of documents. All data of a given object (for example, receipt) can be stored in one document. The documents are stored mostly in JSON, BSON[2], XML or YAML[3] format. Unlike SQL databases, document-orient databases do not have a schema. This makes them more flexible.

The documents are stored in collections. Documents can also contain subcollections of documents.

Two very popular document-oriented databases are MongoDB[4] and Google Firestore[5]. MongoDB uses BSON to store documents, Firestore uses JSON. Both databases support data types such as strings, numbers, arrays and timestamps[6].

Although document-oriented databases do not have a schema, it is beneficial to have a fixed data model so that the shape of the stored data is predictable. The data model for Receipts Scanner is shown in

---

1. NoSQL database is a database that does not employ the relational model of data [29].
2. Binary JSON.
3. YAML is a human-readable data serialization language.
4. `https://www.mongodb.com/`
5. `https://firebase.google.com/products/firestore`
6. Supported data types in MongoDB: `https://docs.mongodb.com/manual/reference/bson-types/`.
Supported data types in Google Firestore: `https://firebase.google.com/docs/firestore/manage-data/data-types`.

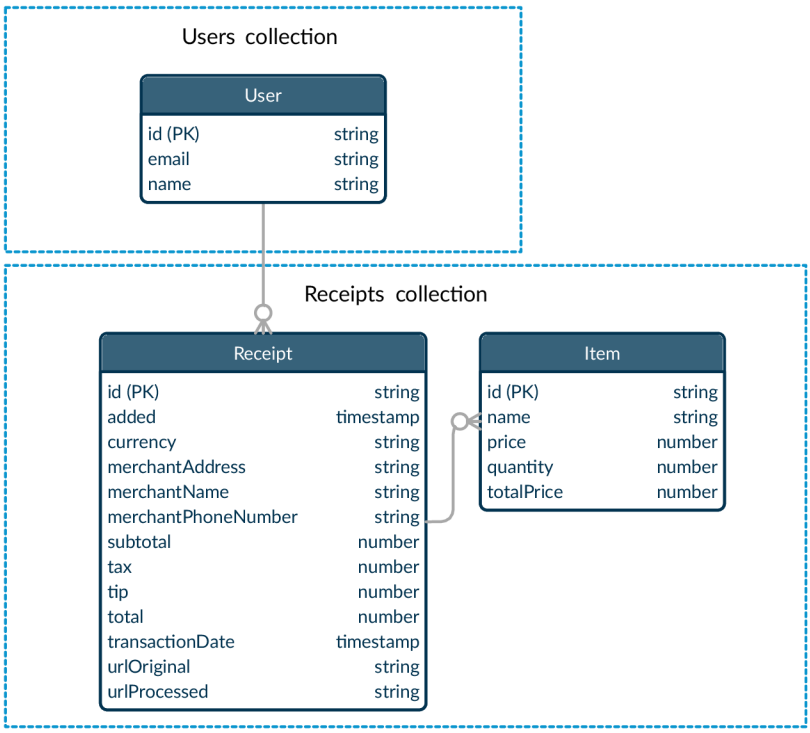


Figure 7.1: Data model of Receipts Scanner.

Figure 7.1. It shows how the data are structured into two collections, `Users` and `Receipts`. Because all user data are child nodes of a given user document, it is easy to restrict access to other documents based on the path to that document. The authentication and access rules are described in Section 7.2.

Although the item forms a separate entity, it is not a document but only an object in the array of items of the receipt.

Other notable fields are `urlOriginal` and `urlProcessed`, which specify the URL address of the original receipt photo and a processed photo, respectively.

7.1.2 Image data

Although storing images directly in the document-oriented database would be possible[7], this solution is not optimal.

For receipt images, the most cost-effective storage is object storage. Object storage is storage optimized for storing BLOBs (Binary Large Objects). BLOB is a collection of binary data stored as a single entity. It can be, for example, an image, audio or video.

The images are stored inside `Users/<user_id>/Receipts/` directory. This folder structure enables restricting access to images based on the value of `<user_id>`.

I have chosen Google Firestore to store the textual data and Google Storage to store the photos of receipts. These two cloud-based services are available under a platform called Firebase. The Firebase platform also provides authentication and analytics services to Receipts Scanner.

Firebase is supported on iOS, Android and the Web. An unofficial library named React Native Firebase[8] provides the same services on React Native for Android and iOS. However, it does not support React Native Windows. Consequently, Receipts Scanner uses Firebase for Web on Windows and React Native Firebase on Android. To be able to write the same code that could be run both on Android and Windows, an adapter `firebase.ts` was created. It imports both libraries, and, based on the platform the application is currently running on, it exports the proper objects. The rest of the code that needs to work with Firebase imports the objects from `firebase.ts` instead of importing them directly from Firebase for Web or React Native Firebase.

In theory, the Receipts Scanner could use Firebase for Web both on Android and Windows, but I would expect React Native Firebase to be more suitable and faster since it uses native APIs. React Native Firebase could also get support for React Native Windows in the future.

7. In MongoDB, it is possible to store BSON documents up to 16 MB. Larger files can be split into parts of 255 kB using GridFS [30].
8. `https://rnfirebase.io/`

7.2 Authentication

"Authentication is the process of verifying an entity's identity, given its credentials." [31] In Receipts Scanner, users are being authenticated.

The authentication in the application serves two purposes. The first one is the ability to connect saved receipts with the user who created them. The second reason for authentication is to ensure proper database rules. The rules are set so that only authenticated user can create a new *User* document[9]. The user can read and modify the *User* document if they are authenticated and if the document belongs to them. The last rule ensures that each user can create, read and modify only their receipts inside the *Receipts* collection and not receipts of other users.

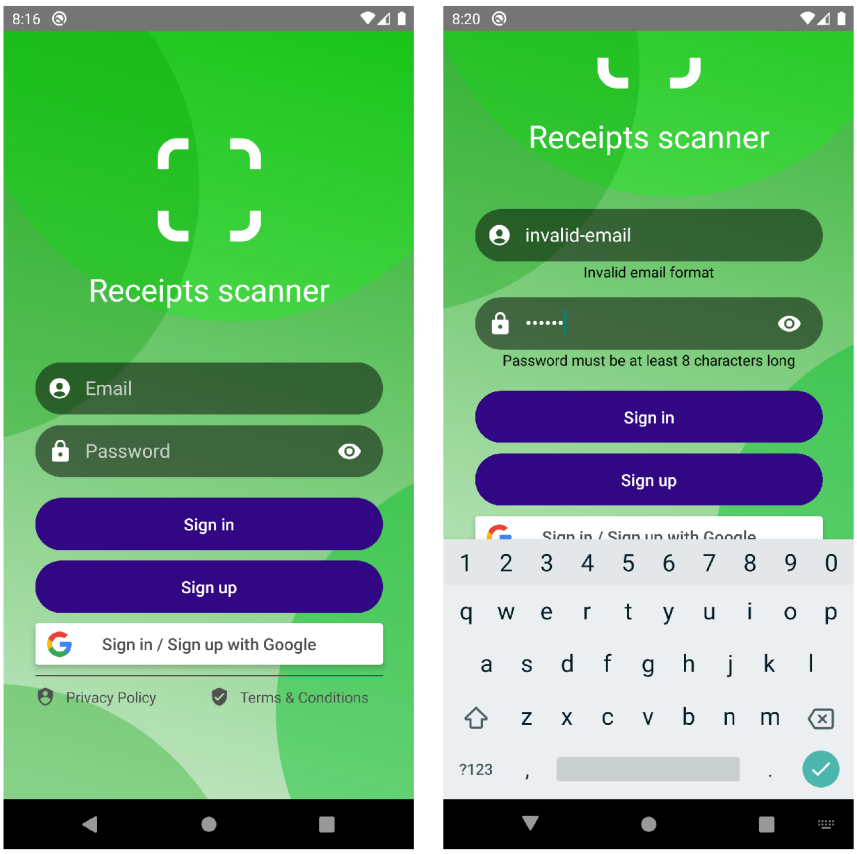
When the user first launches the application, a login screen is displayed. The login screen is depicted in Figure 7.2. It consists of two text input fields and three buttons. The first input field is for email, and the second is for password. It is possible to display the entered password in plain text form by pressing the eye icon next to the password input. Based on the user's preference, they can choose one of the two supported authentication mechanisms:

1. **Authentication with email and password**

   Figure A.1 shows how the process of a sign-in with email and password looks. The form offers a real-time validation, so the user is instantly notified if they have entered an invalid email or a password. The validation is shown in Figure 7.2b.

   Figure A.2 shows the process of a registration with email and password.

9. Documents and data model of Receipts Scanner have been introduced in Section 7.1.



(a) Empty login form.

(b) Form with validation errors.

Figure 7.2: Login screen of Receipts Scanner on Android.

2. **Authentication with a Google account**

   Figure A.3 shows authentication flow using Google Sign In[10] native module. Authentication with Google is not available in the Windows version of Receipts Scanner because the Google Sign In library does not support React Native Windows yet[11].

10. Google Sign In on GitHub: `https://github.com/react-native-google-signin/google-signin`.
11. GitHub issue that tracks adding of support of Google Sign In for Windows: `https://github.com/microsoft/react-native-windows/issues/6531`.

What all three workflows have in common is that the user's session is persisted even if the application is restarted. If a user wanted to use a different account, they could sign out in the settings screen, depicted in Figure 7.3. Clicking the *Sign out* button ends the session and redirects the user back to the login screen.

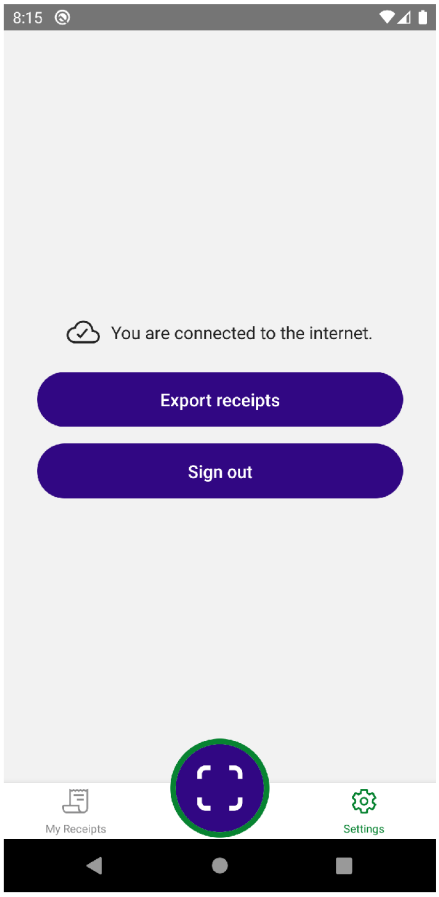


Figure 7.3: Settings screen of Receipts Scanner on Android.

7.3 Adding a receipt

The main functionality offered by Receipts Scanner is adding a receipt and storing it in the user's collection of receipts. This process is visu-

alized in Figure 7.4. It starts by the user selecting an already existing photo of a receipt from a gallery or by taking a new photo with the phone camera. The photo can be cropped to remove the background. It can also be rotated. The rotation does not have any effect on the result of OCR. It affects only how the photo will be shown to the user in their receipts collection.

Next, three sequences of processes happen asynchronously:

- The image of the receipts is sent to Form Recognizer API. The returned data (structured text information extracted from the image) is sent to the Python API `/category` endpoint, where emojis are added to all items that have been recognized on the receipt.

  Form Recognizer and Python API are described in Sections 7.4 and 7.5, respectively.

- The image is uploaded to Firebase Storage.

- The image is sent to the Python API `/process-image` endpoint. The endpoint returns a processed grayscale image that is then uploaded to Firebase Storage.

At the end, the receipt document is added to the user's receipt collection in Google Firestore, and the edit form is shown.

The benefit of the asynchronicity of those processes is that the whole process is much faster than if it happened synchronously. Adding a receipt takes under 8 seconds on average[12].

7.3.1 Uploading image from Windows

On Android, the image capture and selection from the gallery is provided by library react-native-image-crop-picker[13]. However, this library does not support Windows. A library react-native-document-picker[14] used to provide a file selection feature for Windows but is incompatible with the current version of React Native for Windows.

12. Based on measurements of 10 different receipts.
13. `https://github.com/ivpusic/react-native-image-crop-picker`
14. `https://github.com/rnmods/react-native-document-picker`

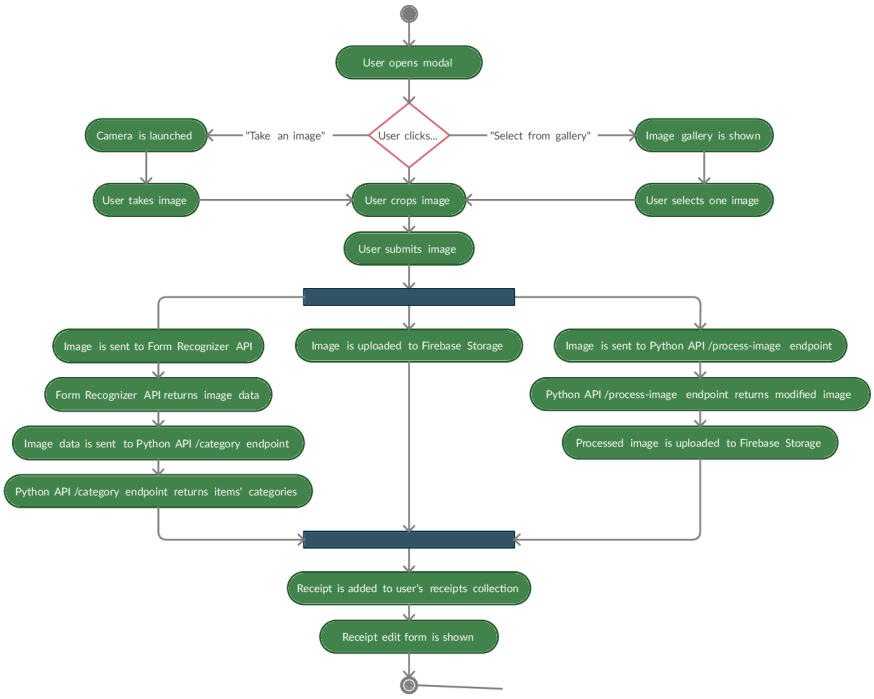


Figure 7.4: Activity diagram of adding a new receipt.

For this reason, a native C++ module *File Picker* was implemented. It allows to pick an image from the Windows Explorer and returns an object containing the path to the image, MIME type[15] and the image data encoded as Base64[16]. This module is then used in the JavaScript code. However, the image upload functionality is not complete on Windows because Firebase Storage for Web uses Web APIs that are not available in React Native running on Windows. An option to overcome this issue would be finding cloud storage that supports the upload of Base64 strings, preferably without any library, using only REST API.

The *File Picker* module could be eventually released as a separate package, so that it could be used by other React Native developers.

7.4 Form Recognizer

Form Recognizer is a cloud service by Microsoft for extracting structured information from documents [1]. It provides a pre-built model that is suitable for recognizing English receipts [32]. It can be accessed through a JavaScript client library[17]. However, this library is built for Node.js runtime, which is different from the React Native runtime. In most cases, React Native uses JavaScriptCore, which is the JavaScript engine that powers Safari [33].

Receipts Scanner implements a service `FormRecognizerClient` with the same interface as the one from the official library, but it relies solely on API calls to the Form Recognizer API. The method to extract the information from the receipt is modelled as a long-running operation, which means that the initial request returns a URL to which the subsequent requests for the result should be made.

After `FormRecognizerClient` is created, its `beginRecognizeContent` method is called with the image that should be processed with the OCR service. The method returns an instance of a `Poller`. The `Poller` provides a method `pollUntilDone` that repeatedly sends requests to the Form Recognizer API until the result is available.

15. MIME (Multipurpose Internet Mail Extensions) type identifies the format of data transmitted on the internet.
16. Base64 is an encoding that encodes binary data into ASCII text.
17. `https://docs.microsoft.com/en-us/javascript/api/overview/azure/ai-form-recognizer-readme?view=azure-node-latest#recognize-receipts`

The final response from the Form Recognizer contains multiple fields[18], such as merchant name, merchant address, transaction date and time, total price, items in the receipt with their name, quantity, price per item and total price for all pieces of the given item. Every field has information about the location in which it was found, i.e. *bounding box* and a confidence value from zero to one.

7.5 Python API

The Python API is a service that runs separately from Receipts Scanner. It serves two purposes: to process an image of a receipt and to categorize receipt items.

It provides three endpoints[19]:

- `GET /` — Returns a health check string. It is used to make sure that the service is running.
- `POST /category` — Expects a JSON containing an item name, possibly consisting of multiple words. It returns a category of that item and an associated emoji.
- `POST /process-image` — Expects a form-data containing an image. It returns a JSON containing the processed version of the image as a Base64 string and the image MIME type.

7.5.1 Item categorization

In order to make the application more visually attractive and to improve the user experience, each item on the receipt is assigned an emoji. The emoji is chosen based on the words that appear in the item name. If the user is not content with the automatically assigned emoji, they can remove it in the receipt edit form.

18. Form Recognizer API documentation: `https://westcentralus.dev.cognitive.microsoft.com/docs/services/form-recognizer-api-v2-1-preview-2/operations/GetAnalyzeReceiptResult`.
19. Detailed documentation of the endpoints is available at `https://documenter.getpostman.com/view/9355808/TzJsfdUK`.

When a user adds a new receipt, the items on the receipt are sent to the Python API `/category` endpoint. The service contains a dictionary of category names as keys and emojis as values. Each item name is compared to all dictionary keys by using method `most_similar_to_given` and the key that is most similar to the item name is returned, together with the emoji.

The method `most_similar_to_given` is provided by the Magnitude library [34]. Magnitude is a tool for utilizing and processing word embeddings. Word embeddings are typically real-valued vectors that encode the meaning of a word, such that the vectors that are close in the vector space are similar in their meaning [35]. Magnitude is a fast and memory-efficient alternative to tools like Gensim[20].

It uses its own vector storage file format `.magnitude`. Popular embedding models such as Google's word2vec, Stanford's GloVe and Facebook's fastText have been pre-converted to the `.magnitude` format for download and usage[21]. Furthermore, each word embeddings model is available in three variants: light, medium and heavy. The difference is that medium and heavy models have advanced support for out-of-vocabulary keys[22]. The heavy model also features faster approximate K-Nearest Neighbors[23].

Magnitude implements functions for looking up vector representations for misspeled or out-of-vocabulary words. This means that it can handle abbreviations, which often appear on receipts, and whole sentences reasonably well.

The interface of the library is agnostic to the underlying model. In Receipt Scanner, the fastText model is used — more specifically, its medium variant, as only the exact similarity search is performed, so there is no need for faster approximate K-Nearest Neighbors. The light variant is not used because it handles item names that contain multiple words poorly.

Another constraint for choosing the model was its size. Google Cloud Run, a cloud service where the model is running, does not

20. `https://radimrehurek.com/gensim/`
21. Pre-converted Magnitude models for download:
`https://github.com/plasticityai/magnitude#pre-converted-magnitude-formats-of-popular-embe`
22. Words that do not appear in the model's vocabulary.
23. A supervised machine learning algorithm for classification and regression problems.

allow containers larger than 8 GB. Google's medium word2vec model is 4.9 GB large compared to fastText, which is only 1.6 GB large. The container with word2vec did not fit into the 8 GB limit.

The used fastText model was pre-trained on over 16 billion words from Wikipedia 2017, Statmt.org news and UMBC news [36]. It contains only English word vectors. However, fastText provides models for other languages[24], which could be converted using the Magnitude converter[25] and used in a localized Receipts Scanner, e.g. with support for the Czech language.

Table 7.1 shows results from various word embeddings models. The first two columns are produced by Google's word2vec, which was trained on 100 billion words from Google News [37]. The third column is fastText. The fourth column is fastText with subwords information. In the subwords model, each word is represented as a bag of character n-grams. Taking the word *where* and $n = 3$ as an example, it will be represented by the following character n-grams:

$$< \text{wh, whe, her, ere, re} >$$

and the special sequence $<$ where $>$. The less-than sign $<$ and the greater-than sign $>$ are special boundary symbols. [38]

The results show that the case matters, e.i. *CHUPA LOLLIPOPS* and *Chupa lollipops* are categorized differently, and how more concrete terms are mapped to more general terms, e.i. *salmon* and *cola* are categorized as *fish* and *cold beverage* respectively.

7.5.2 Image processing

The image processing happens independently of the OCR. The image sent to the Form Recognizer API is the original image because it uses its own image processing techniques and provides better results on original images.

The purpose of image processing in the Receipts Scanner is to provide a cleaner preview of the receipt for the user. The result is a

24. Word vectors for 157 languages:
`https://fasttext.cc/docs/en/crawl-vectors.html`.
25. `https://github.com/plasticityai/magnitude#file-format-and-converter`

Table 7.1: Comparison of pre-trained word embeddings models.

item name	word2vec light	word2vec medium	fastText medium	fastText subwords medium
salty snack	🎅 santa claus	🥨 snack	🥨 snack	🥨 snack
CHUPA LOLLIPOPS	🏑 field hockey	🍬 candy	🍬 candy	🍭 lollipop
Chupa lollipops	💾 floppy disk	🍬 candy	🍭 lollipop	🍭 lollipop
salmon	🐟 fish	🐟 fish	🐟 fish	🐟 fish
shampoo	🧴 lotion	🧴 lotion	🧴 lotion	🧴 lotion
ipad air	💨 air condition	📱 iphone	📱 iphone	📱 iphone
manual weight	👒 woman's hat	🏋 weight lifting	🏋 weight lifting	🏋 weight lifting
frozen pizza	🌶 hot pepper	🍕 pizza	🍕 pizza	🍕 pizza
Cola 1 l	👡 woman's sandal	🥤 cold beverage	🥤 cold beverage	🍫 chocolate bar
eggs	🥚 egg	🥚 egg	🥚 egg	🥚 egg
T MAT CHEDDAR	💾 floppy disk	🧀 cheese	🧀 cheese	🧀 cheese
pork loin chop	🛵 motor scooter	🌭 hot dog	🐖 pork	🐖 pork

(a) Before Gaussian blur. (b) After Gaussian blur.

Figure 7.5: Comparison of an image before and after Gaussian blur.

cropped picture with black text on a white background. The whole process consists of the following steps:

1. **Image loading**

   The image is loaded in a grayscale.

2. **Noise removal**

   Gaussian blur is applied to reduce the Gaussian noise in the image. It is the most common type of noise in an image [4]. The blur is achieved by convolving the image with a Gaussian function.

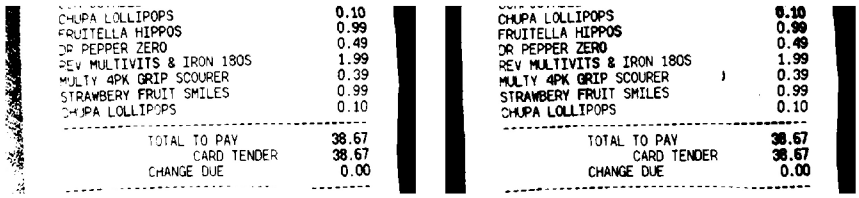
$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}, \tag{7.1}$$

   where x and y are the distances from the origin in the horizontal and vertical axis, respectively. σ is the standard deviation of the Gaussian distribution. [4]

   The result of applying Gaussian blur is shown in Figure 7.5.

3. **Binary thresholding**

   It is beneficial to convert the images to binary images to detect the edges and contours better. Binary thresholding sets pixel value to the maximum value — usually 255 (white) — if



(a) Binarized image with fixed threshold of 127.

(b) Binarized image with threshold of 151 found by Otsu's method.

Figure 7.6: Comparison of a fixed and Otsu's thresholding.

its value is greater than the threshold. Otherwise, the pixel value is set to 0 (black).

$$dst(x,y) = \begin{cases} \texttt{maxval} & \text{if } src(x,y) > \texttt{thresh} \\ 0 & \text{otherwise} \end{cases} \tag{7.2}$$

In Equation 7.2, $src(x,y)$ is a value of a source pixel on coordinates x and y and $dst(x,y)$ is a value of destination, i.e. resulting pixel, on coordinates x and y.

The threshold value can be either fixed, e.g. 127, or computed from the image. Finding the right value for the fixed thresholding is challenging, and each image might have a different optimal threshold value.

Otsu's thresholding calculates the threshold value dynamically from the histogram of each image, such that the found threshold maximizes the between-class variance [39]. Classes are black and white pixels.

Figure 7.6a shows the result of applying fixed binarization with a threshold value of 127. The edges of the receipt in this image would not be correctly recognized because the threshold value is too small.

Figure 7.6b shows the result of applying Otsu's thresholding, which automatically found a threshold value of 151. The edges are clearly visible.

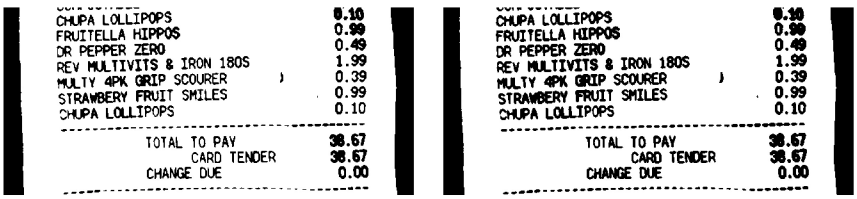
4. **Dilation**

   In order to determine the edges of the receipt more easily, the image is dilated. Dilation increases the size of the grid lines.

   Before performing dilation, the image has to be inverted. Otherwise, an opposite effect, erosion, would be achieved. It is necessary to invert the image back before contour detection, which is described in the next step.

   A kernel slides through the image the same way as in 2D convolution. A pixel is considered white if at least one pixel in the original image under the kernel is white.

   The result of applying dilation is shown in Figure 7.7.



(a) Before dilation. (b) After dilation.

Figure 7.7: Comparison of an image before and after dilation.

5. **Finding the corners of a receipt**

   As the next step, all external contours in the image are found. Contour is a curve joining points of the same color and intensity. The contour with the largest area is most probably the contour of a receipt. Four points that represent the corners of the receipt are then found in the contour and returned. The idea behind finding the corners is that the bottom-left corner has the smallest x coordinate and the largest y coordinate, the bottom-right corner has the largest x a y coordinates etc. The result of this step is visualized in Figure 7.8.

   However, if the contour area is smaller than 30% of the overall image, the next step, *Cropping and warping*, is skipped.

Figure 7.8: Photo of a receipt with detected corners.

6. **Preparing the underlying picture**

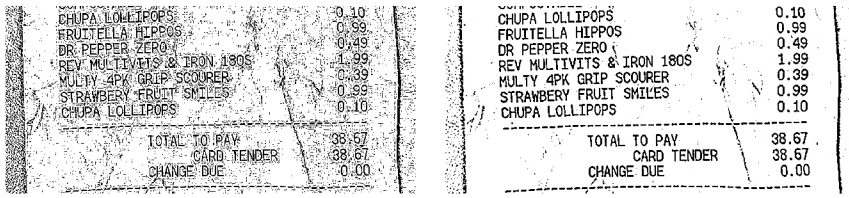
   The previous steps were used to find the corners of a receipt. Although Otsu's thresholding made the edges of the white receipt clearly distinguishable from the dark background, the text of the receipt became unreadable.

   With receipts scans, the situation is different. Since scans usually do not have any background, Otsu's thresholding maximizes the contrast between the paper and the text instead of the contrast between the paper and the background.

   For receipt photos, adaptive thresholding is a better choice. This is also caused by shades in the photos, which are not present in scans.

   With adaptive thresholding, the threshold value for each pixel is based on a small region around it. This method gives better results for images with varying illumination. [40]

   OpenCV library provides two methods of computing the adaptive threshold. It can be computed as the mean of the neighbor-



(a) Without Gaussian blur. (b) With Gaussian blur.

Figure 7.9: Result of adaptive thresholding applied on an image without and with Gaussian blur.

hood area minus a constant C or as a Gaussian-weighted sum of the neighborhood values minus the constant C. Empirically, the Gaussian method with $C = 2$ produced the best results.

Before the adaptive thresholding is applied, the original image is blurred using Gaussian blur. This improves the result significantly, as can be seen in Figure 7.9.

7. **Cropping and warping**

Lastly, a perspective transformation (warping) is applied to the underlying photo produced by the previous step. The found quadrilateral, represented by four corners, is warped into a rectangle of a similar size. This step also crops the receipt from the background.

A comparison of the original image and the final processed image is shown in Figure 7.10.

(a) Original image.

(b) Processed image after adaptive binarization and perspective transformation.

Figure 7.10: Comparison of the original and the processed image.

7.5.3 Deployment

The Python API is a Flask[26] server. It is automatically deployed after each push to the master branch. First, Google Cloud Build[27] builds a Docker image that contains the source files, external dependencies and the Magnitude word embeddings model. The built image is saved into Google Container Registry[28]. Google Cloud Run[29] then runs the image in a container. If the image is not built successfully or the Python API service fails to start, the previously deployed service stays available.

The newly deployed Python API service is tested with tests that run in the repository CI pipeline.

26. `https://flask.palletsprojects.com/en/1.1.x/`
27. `https://cloud.google.com/build`
28. `https://cloud.google.com/container-registry`
29. `https://cloud.google.com/run`

7.6 Styles

Apart from black and white, the Receipts Scanner user interface uses two main colors. The primary color is malachite, `#078331` and the secondary is `#310783`. The primary color has been chosen to resemble the color of banknotes. The secondary color is one of the three colors of malachite's triadic colors[30]. The colors are chosen with respect to the contrast ratio between each other as well as the contrast ratio between those two colors and black and white. This is especially important in the dark mode.

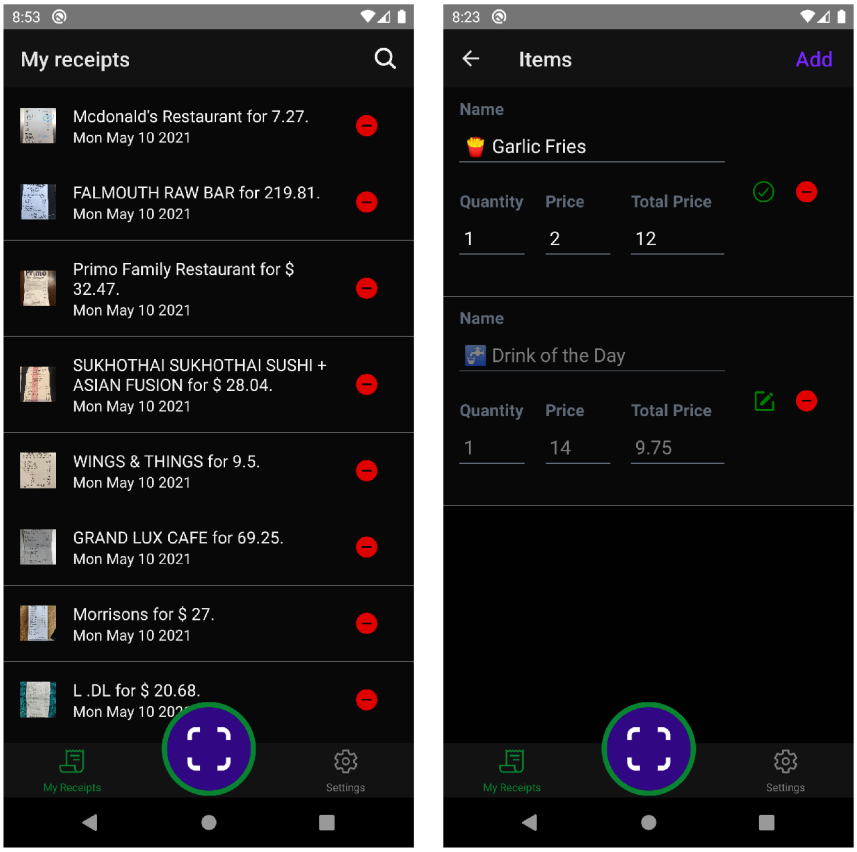
Most of the elements are styled with these two colors to ensure a clean and consistent user interface look.

7.6.1 Dark mode

Receipts scanner switches automatically into the dark mode based on the user's device settings. The dark mode is a popular feature among users. It helps reduce digital eye strain and saves battery on devices with OLED[31] screen.

Figure 7.11 shows the user interface with the dark mode enabled on an Android device. Figure 7.12 shows the application in the dark mode on Windows.

30. The color triad was generated using color-hex tool available at `https://www.color-hex.com/color/078331`.
31. Unlike LCD screens that illuminate using a back panel that always lights up completely, in OLED displays, each pixel is individually lit.



(a) List of receipts. (b) Receipt items.

Figure 7.11: User interface of Receipts Scanner on Android in the dark mode.

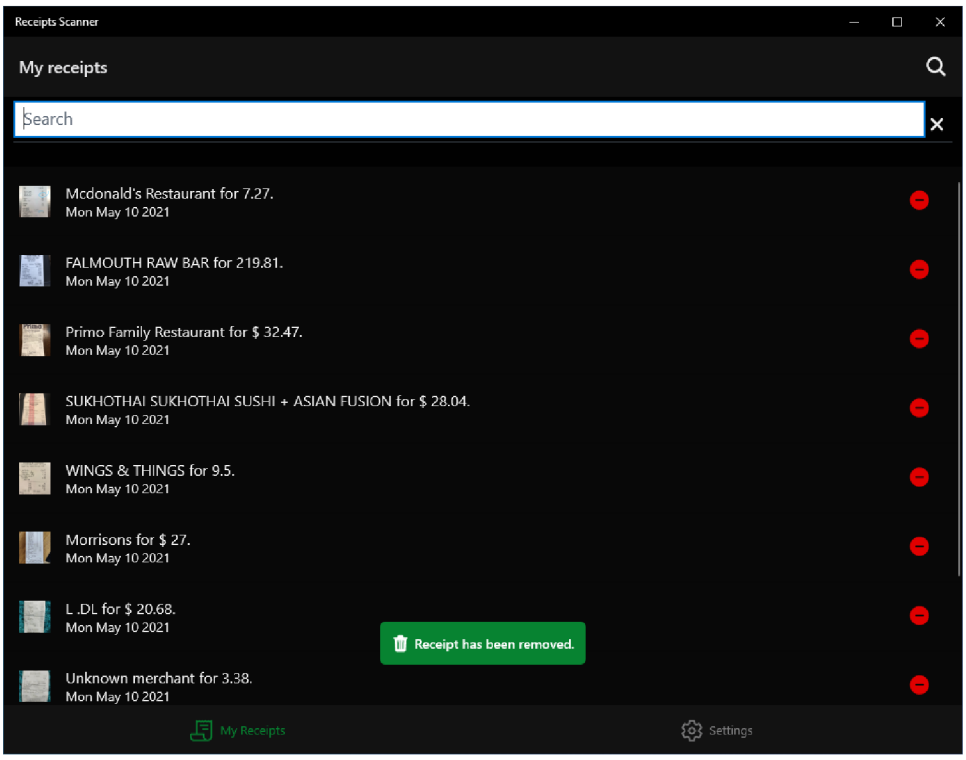


Figure 7.12: User interface of Receipts Scanner on Windows in the dark mode.

7.7 Real-time user interface

The receipts are always up to date. Any change in the receipt database for a given user is almost instantly reflected on all other devices where that user is logged in. This means that a user can scan a receipt on the mobile and fill in the receipt information from the PC. If they decide to delete the receipt, the receipt disappears from the other device as well. This functionality is possible thanks to Google Firebase.

7.8 Offline support

The application works partially offline. Suppose a user is offline and makes any changes to an already existing receipt (edits a receipt, adds a receipt item, deletes a receipt). In that case, the change is reflected in the application UI on the device where the changes were made. Once the user connects to the internet, the changes propagate to the database and other user's devices. The offline changes are persisted even if the application is turned off. This functionality is provided by the Google Firebase library.

The receipt scanning functionality is not possible in an offline mode. If a user tries to scan a receipt without an internet connection, an alert modal window is shown to them.

Ideally, the image would be stored locally, and its thumbnail would be shown among other receipts with a status of *being processed*. After the device connected to the internet again, the image would be sent to Form Recognizer, which would return the extracted data, and the user could complete the pre-filled form.

Another approach would be to use a local model instead of Form Recognizer that would extract the receipt data. This way, the user could complete the complete journey of adding a receipt at once, and the receipt data and the image would be sent to the APIs later on.

Microsoft Form Recognizer does not provide an option to export the model, even if it was a custom one. Therefore a different OCR and data extraction approach would be needed. Furthermore, if the offline model was too large, it should be an opt-in feature of Receipts Scanner.

7.9 Features

This section provides an overview of the functionality available in the Receipts Scanner. Unless stated differently, the feature is available on both platforms, i.e. Android and Windows.

- Authentication
  - Sign-in/registration with email
  - Sign-in/registration with Google — only on Android

  - Email format validation
  - Password length validation
  - Preview of password in plain text on toggle
- Adding of receipt — only on Android
  - Picking the receipt image from gallery
  - Taking a photo of the receipt
  - Receipt image cropping and rotation
  - Automatic data extraction from the photo that pre-fills the form
  - Automatic insertion of emoji to each receipt item
- Receipt preview
  - Preview of the original image
  - Preview of a processed image for better readability
  - Purchase date, merchant name, merchant address, merchant phone number, total price, subtotal, tax, tip and currency.
  - List of receipt items with emoji, name, quantity, price and total price.
- Receipt editing
  - Editing of purchase date, merchant name, merchant address, merchant phone number, total price, subtotal, tax, tip and currency.
  - Form fields validation
  - Fluent navigation between form fields and submitting of form using keyboard only
  - Ability to add, edit and delete a receipt item
- Searching through receipts by merchant name, merchant address and individual names of receipt items
- Receipt deletion

- Dark mode that reflects system settings
- Real-time changes
- Partial offline support
- Internet connectivity notification — mocked on Windows
- Export of receipts data and images
- Notification toast messages

8 Financial sustainability

The Receipts Scanner uses paid third-party services, namely:

- Microsoft Form Recognizer for OCR and data extraction,
- Google Cloud Build, Google Container Registry and Google Cloud Run for the deployment of Python API container,
- Google Firebase for document database and image storage.

The Firebase Authentication is free.

It would be necessary to run the application for some time to figure out the usage statistics, such as the average number of receipts a user uploads. Since those statistics are not available yet, I will provide an *estimate* of the operational costs.

Let us assume that the application has 1000 active users, where each user makes one scan per day on average. The price for Microsoft Form Recognizer is \$0.01 per scan [41]. This means the total cost of this service would be $10{,}000 \cdot 30 \cdot \$0.01 = \$300.00$.

The size of an uploaded image is around 500 KB, the processed image is smaller, around 100 KB large. The price of storing the textual data is negligible compared to other costs and would be lost in the approximations. Cloud Firestore is a very cheap storage [42] and its free tier provides storage for 1 GB, which is about 400,000 average-sized receipts[1].

Whereas the number of scans is linearly proportional to the number of users, the numbers of stored receipts grows quadratically. It can be expressed recurrently as:

$$s(t) = \begin{cases} s \cdot (t-1) + r \cdot u \cdot t & \text{if } t \geq 1 \\ 0 & \text{otherwise} \end{cases} \tag{8.1}$$

or as a function of t:

$$s(t) = \frac{r \cdot u}{2} \cdot t \cdot (t+1), \tag{8.2}$$

1. The size of a stored receipt, 2500 KB, was computed using the firestore-size library available at `https://github.com/alekslario/firestore-size`.

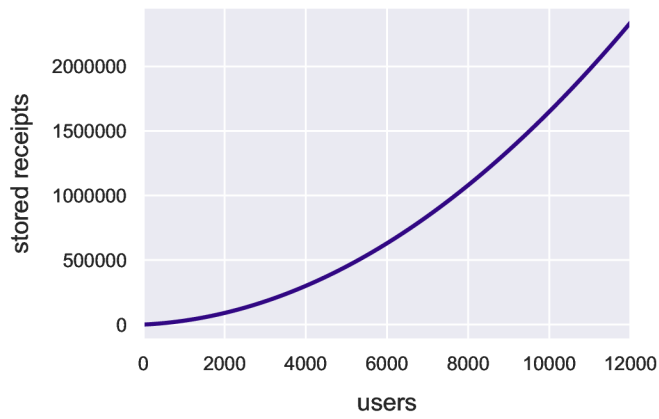


Figure 8.1: Number of stored receipts by the number of users.

where t is a point in time (e.g. month), r is the number of receipts per user per chosen time period (in this case month) and u is the number of new users at the end of the chosen time period (i.e. at the end of a month). Since we assume one receipt per user per day, $r = 30$. Assuming 1000 new users per month, $u = 1000$. Substitution of the values into Equation 8.2 gives

$$s(t) = 15{,}000 \cdot t \cdot (t + 1). \tag{8.3}$$

A graph of this function is shown in Figure 8.1.

The images of receipts are stored in Google Cloud Storage. The price is \$0.026 per GB [43]. For the first month, the cost would be $1000 \cdot 30 \cdot (600\ \text{KB}/1\ \text{GB}) \cdot \$0.026 = \$0.468$. For the second, assuming 1000 new users, it would be \$1.404.

The Google Container Registry is charged the same way as Google Cloud Storage [44], depending on the size of all stored containers. Even though storing only the newest container would suffice, it is good to keep a few previous containers as well. It is then possible to go back to these containers, should the newest one have any errors. Older containers can be deleted[2]. This means that the price for the stored containers will be fixed. The size of one Python API container

2. Google provides an automated way of deleting old containers available at `https://github.com/sethvargo/gcr-cleaner`.

is 2.7 GB. The price of storing the five latest containers would be $5 \cdot 2.7 \cdot \$0.02 = \$0.27$ per month.

Google Cloud Run provides autoscaling, which means that the number of provisioned instances depends on the number of incoming requests. If a minimum number of instances is set to zero, the service goes idle each time the requests are not coming. The service is set up to use 2 CPUs and 6 GB of memory. For each receipt, one request is made to process the image and one to assign categories to items, resulting in $1000 \cdot 30 \cdot 2 = 60{,}000$ requests per month. Assuming that each request comes after the previous request ended and that each request takes 1000 ms, the service will be used only for 120,000 vCPU-seconds[3]. This is included in the free plan. Memory consumption would be 480,000 GiB-seconds[4], which slightly exceeds the free plan [45], incurring \$0.3 of additional costs. If a minimum number of instances was set to one in order to prevent a cold start, the cost would be \$66.00 per month.

Building a container using Google Cloud Build takes approximately 9 minutes. The price is \$0.016 per build-minute [46]. Assuming three builds per day due to code changes pushed to the master branch, the cost per month is $3 \cdot 9 \cdot \$0.016 \cdot 30 = \$12.96$.

The total operational cost per month with 1000 users would be \$314.00, i.e. \$0.314 per user. From Figure 4, it is apparent that most of these costs are due to the OCR service. For this reason, other applications offer the feature of automatic data extraction only after paying a fee. Smart Receipts charges \$0.1 per one OCR scan (as of April 2021).

Therefore, the Receipts Scanner would need to have its own OCR service, which would be free, or become monetized. In my opinion, in-app advertising, such as AdMob[5], would not generate enough revenue. A subscription-based model seems like a more viable option. The third option would be combining the two previous approaches: a limited number of scans and visible advertisement for non-subscribed users, and an unlimited number of scans and removed advertisement for the subscribed ones.

3. vCPU-seconds are calculated as *number of CPUs* × *running time of service* (*sec*).
4. GiB-seconds are calculated as *size* (*GiB*) × *running time of service* (*sec*).
5. Mobile app monetization from Google.

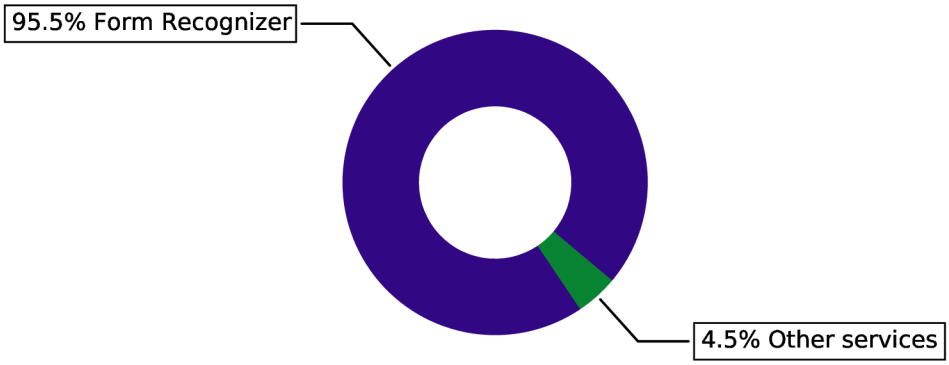


Figure 8.2: Operational costs of using Form Recognizer compared to other services.

9 Evaluation

The application has not been given to real users yet. Hence, there is no available feedback concerning usability, performance and the overall user experience.

The Form Recognizer API used for OCR and data extraction from an image is a third-party service, and as such, measuring its performance would provide little value compared to the rest of the thesis. It would become more important if the current data extraction was insufficient, which would be best found by users in practice.

Quantifying how well Form Recognizer extracts the data from the photo would require a labelled set of receipts. One such set is available as part of the ICDAR2019 Competition on Scanned Receipt OCR and Information Extraction [47]. It provides 1000 labelled receipt scans[1] as well as an evaluation technique. However, the labels from this dataset contain only *company*, *date*, *address* and *total*. Form Recognizer can recognize other information, such as phone number and individual items, too. Furthermore, the main use case in Receipts Scanner is

1. A corrected set of the original images is available on GitHub: `https://github.com/zzzDavid/ICDAR-2019-SROIE`.

recognition of photographed receipts, not receipt scans. Therefore a different dataset would be needed.

A questionnaire was given to 28 people. It consisted of two parts: evaluation of automatically assigned emojis and comparison of the processed image to the original image regarding readability.

9.1 Emoji assignment

This section aims to evaluate how accurately the emoji chosen by the categorization service inside Python API fits the text. The text is the name of a receipt item recognized by Form Recognizer, consisting of one or more words.

Receipts contain very often abbreviations and names of brands. The categorization service cannot handle those items in a meaningful manner. The assigned emojis in this case are rather arbitrary. An example to illustrate this is *P/DY VEGAN STRAW PROT TUB 1KG* classified as 🧸 *teddy bear* or *DR PEPPER ZERO* classified as 🔑 *key*. Another challenge is assigning emojis to restaurant receipts, which often contain peculiar names of dishes.

Seventy items[2] have been selected from the data extracted by Form Recognizer on various receipts, excluding items incorrectly classified as a receipt item and items with which the respondents would be unfamiliar. Duplicates and similar items have not been included in this set either.

For each item and its assigned emoji, the task was to choose whether the emoji is *Not accurate*, *Mildly accurate* or *Accurate*.

As can be seen in Figure 9.1, in the majority of cases, i.e. 53.8%, the chosen emoji was not accurate. Only in 20.8% of cases was the emoji considered accurate.

This feature of Receipts Scanner is meant to bring an element of fun into the application, and as such, its functionality is not critical.

2. All items from the survey are inserted in the archive in the Information System of Masaryk University A.1.

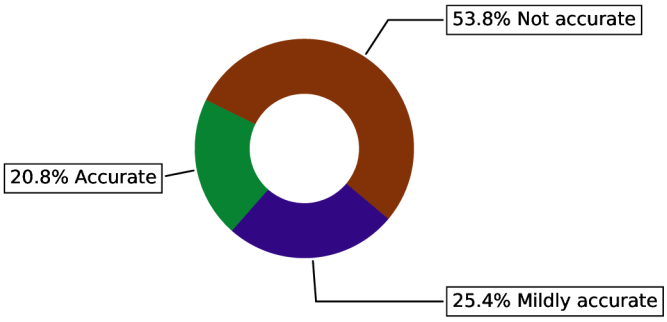


Figure 9.1: Results of a survey evaluating how accurate the emojis assigned to the receipt items were.

9.2 Image processing

The purpose of this section is to evaluate whether the processed images provide any benefits over the original images.

Thirty receipt photos have been processed using Receipts Scanner. The original and processed images have been placed side by side[3]. The task was to choose which image was more readable. As presented in Figure 9.2, respondents found 54.5% of the processed images more readable than the original image.

Sixteen receipts were from the ExpressExpense [48] dataset, and fourteen were taken with modern phone cameras for the purpose of this survey. All new images and one image from the ExpressExpense dataset were considered more readable in the processed form by most respondents. Two ExpressExpense images had an equal number of votes for each option. The common feature of ExpressExpense receipts is that they have lower quality than the newly taken photos. These results show that the image processing provides a more readable form of the original image if the original image is of sufficiently good quality.

Based on the survey results, the image processing feature of Receipts Scanner will provide value to users, as most modern smartphones can take images of high quality. For the individual cases when

3. All images from the survey are inserted into the archive in the Information System of Masaryk University A.1.

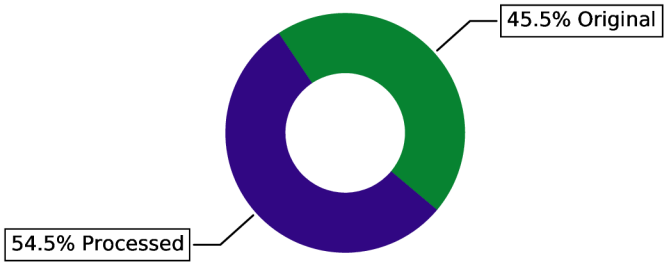


Figure 9.2: Results of a survey comparing which image of a receipt was more readable.

the image processing produces worse or even unreadable images, the original image is still available to the users.

10 Conclusion

The aim of my thesis was to develop an application that would allow users to take a photo of a receipt and pre-fill the database record with the information extracted from the image. The receipts are stored in a central repository and can be accessed from mobile and desktop application written in React Native. The users of the application can search through the receipts, view and edit them.

Compared to similar available applications, the newly developed application can extract information about individual receipt items, automatically assign an emoji based on the item category and provide a receipt preview in a more readable form.

The thesis focuses on development and testing of cross-platform React Native applications.

The desktop application can be further enhanced with features currently available only on mobile as React Native for Windows develops. Future development of the application could also include support for multiple languages and offline data extraction.

The source code of the application is publicly available on GitHub[1] and can be used as a reference for developing other React Native applications.

1. `https://github.com/petr7555/bachelors_thesis_accounting_ocr`

A Appendix

A.1 Attachments

Files inserted into the archive in the Information System of Masaryk University:

- `receipts_scanner.zip` — Source code of the application.
- `thesis.pdf` — Text of the thesis in PDF.
- `questionnaire_materials.zip` — Receipt items and images that have been used in the evaluation survey.
- `screenshots.zip` — Screenshots of Receipts Scanner on Android/Windows in light/dark mode.
- `receipts_scanner.apk` — Android application package for installation of Receipts Scanner.

The latest application source code can be found on GitHub:

- `https://github.com/petr7555/bachelors_thesis_accounting_ocr`

A.2 Authentication activity diagrams

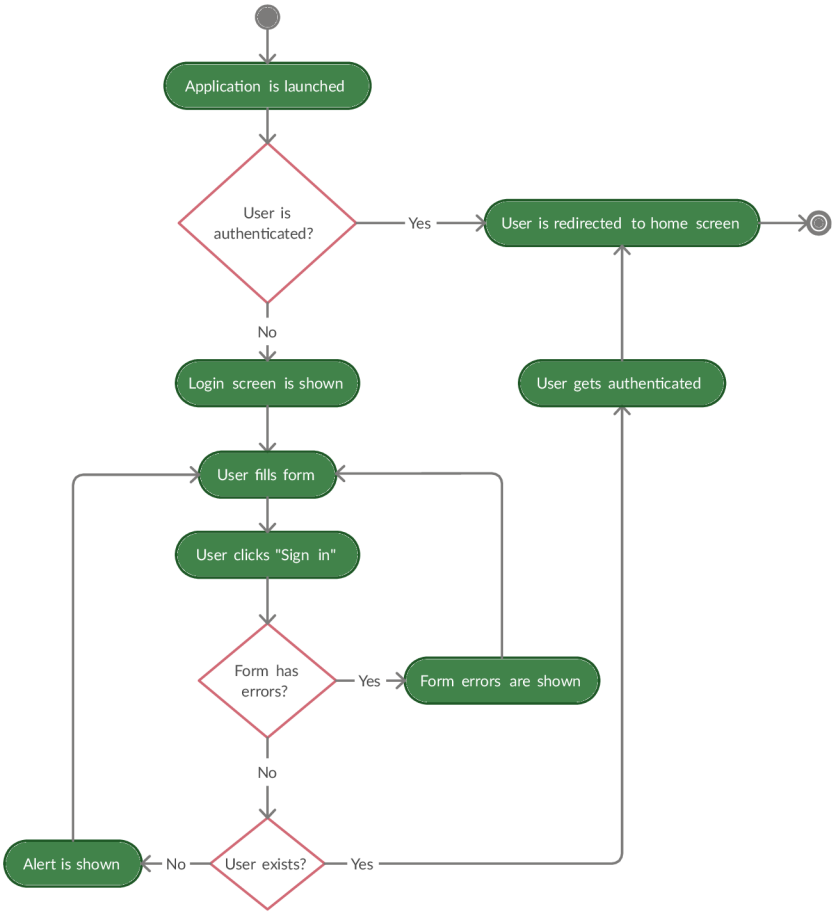


Figure A.1: User sign-in activity diagram.

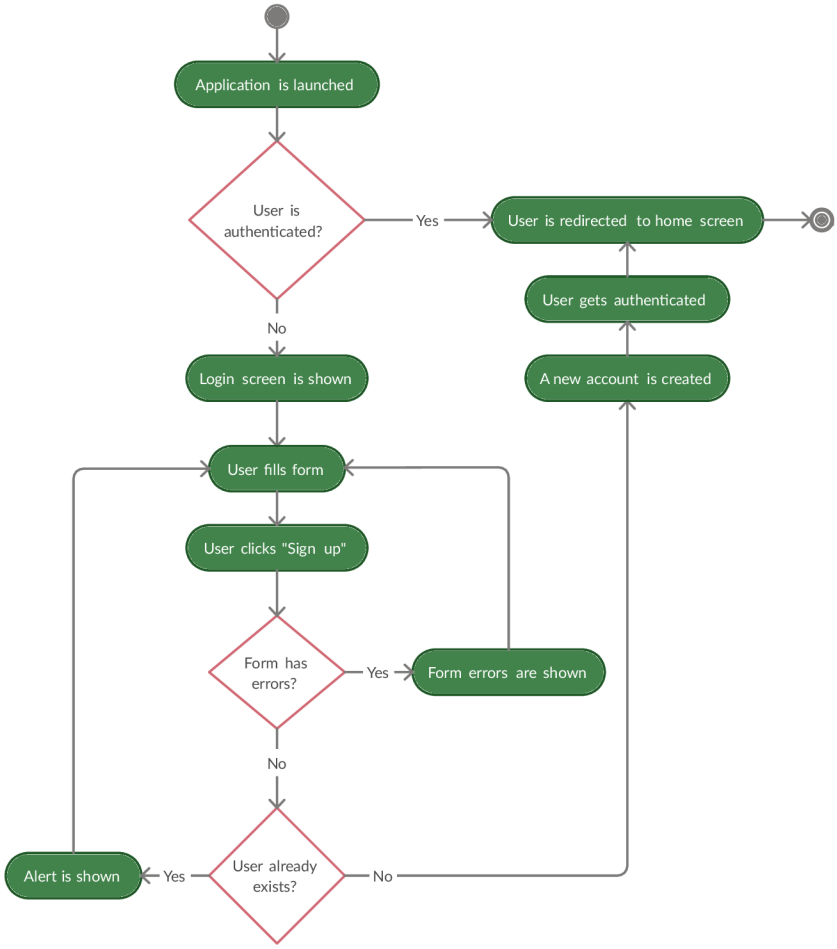


Figure A.2: User sign-up activity diagram.

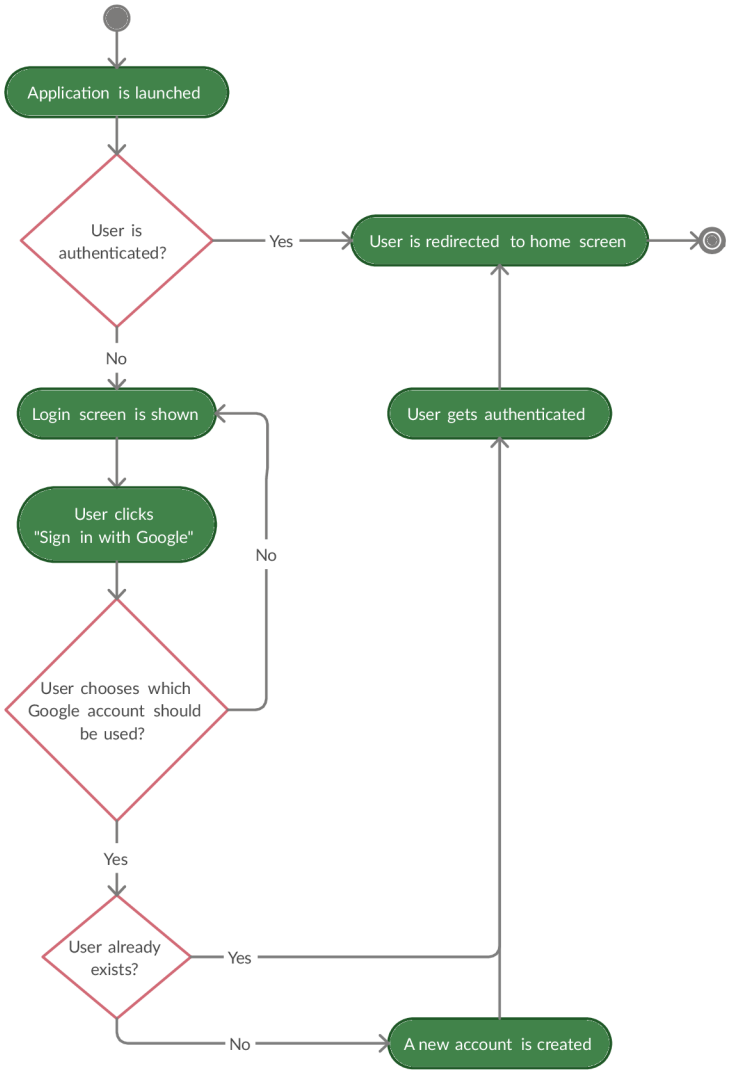


Figure A.3: Google authentication activity diagram.

A.3 Version control system commit messages

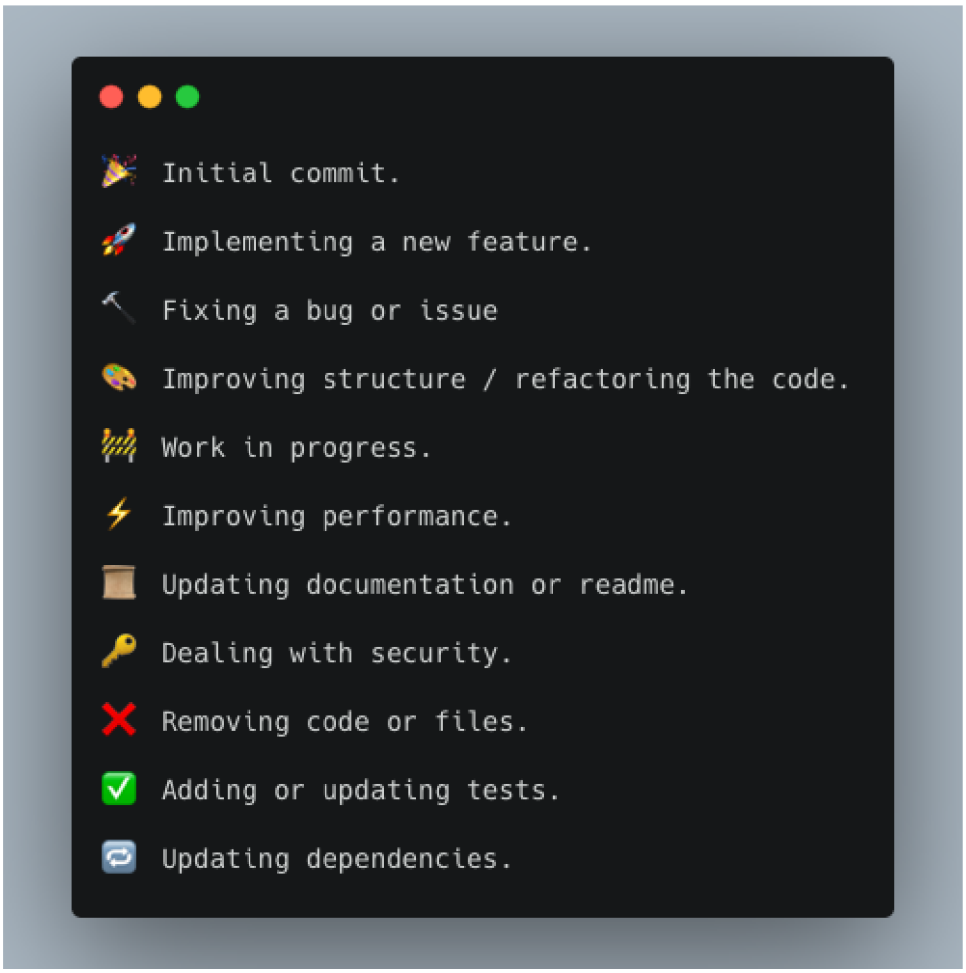


Figure A.4: Meaning of emojis in commit messages.

Bibliography

1. *What is Form Recognizer?* [Online]. Redmond, Washington, USA: Microsoft, 2021-03-15 [visited on 2021-04-01]. Available from: https://docs.microsoft.com/en-us/azure/cognitive-services/form-recognizer/overview.
2. *Definition of Open Source* [online]. Stamford, Connecticut, USA: Gartner [visited on 2021-05-18]. Available from: https://www.gartner.com/en/information-technology/glossary/open-source.
3. *What is a Container?* [Online]. Redmond, Washington, USA: Microsoft, 2021-03-15 [visited on 2021-05-21]. Available from: https://www.docker.com/resources/what-container.
4. SHREYA, Sakshi; UPADHYAY, Yash; MANCHANDA, Mohit; VOHRA, Rubeena; SINGH, Gagan Deep. Optical Character Recognition using Convolutional Neural Network. In: *2019 6th International Conference on Computing for Sustainable Global Development (INDIACom)* [online]. 2019, pp. 55–59 [visited on 2021-04-25]. ISBN 978-9-3805-4434-2. Available from: https://ieeexplore.ieee.org/document/8991268.
5. ZIEGAUS, Marco. *Optical Character Recognition on supermarket receipts* [online]. Passau, Germany, 2016-06-19, chap. 5.4.3, Template matching [visited on 2021-04-01]. Available from: https://www.fim.uni-passau.de/fileadmin/dokumente/fakultaeten/fim/lehrstuhl/sauer/geyer/BA_MA_Arbeiten/BA-ZiegausMarco-201607.pdf. Bachelor's thesis. University of Passau, Faculty of Computer Science and Mathematics.
6. SINGH, Pritpal; BUDHIRAJA, Sumit. Feature extraction and classification techniques in OCR systems for handwritten Gurmukhi Script–a survey. *International Journal of Engineering Research and Applications (IJERA)* [online]. 2011, vol. 1, no. 4, pp. 1736–1739 [visited on 2021-04-25]. Available from: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.299.7852&rep=rep1&type=pdf.

7. *Mobile Fact Sheet* [online]. Washington (D.C.), USA: Pew Research Center, 2021-04-07 [visited on 2021-04-17]. Available from: https://www.pewresearch.org/internet/fact-sheet/mobile/.

8. *Smartphone Market Share* [online]. Needham, Massachusetts, USA: IDC Corporate USA, 2021-04-08 [visited on 2021-04-17]. Available from: https://www.idc.com/promo/smartphone-market-share/os.

9. *Desktop Operating System Market Share Worldwide* [online]. Dublin, Ireland: Statcounter, 2021-03 [visited on 2021-04-17]. Available from: https://gs.statcounter.com/os-market-share/desktop/worldwide. License: CC BY-SA 3.0.

10. *What is Xamarin?* [Online]. Redmond, Washington, USA: Microsoft, 2020-05-28 [visited on 2021-04-17]. Available from: https://docs.microsoft.com/cs-cz/xamarin/get-started/what-is-xamarin.

11. HERMES, Dan; MAZLOUMI, Nina. Building Xamarin.Forms Mobile Apps Using XAML: Mobile Cross-Platform XAML and Xamarin.Forms Fundamentals. In: New York, New York, USA: Apress, c2019, chap. 2, Building Xamarin.Forms Apps Using XAML, pp. 43–44. ISBN 978-1-4842-4030-4. Available from DOI: 10.1007/978-1-4842-4030-4.

12. *Flutter - Beautiful native apps in record time* [online]. Mountain View, California, USA: Google [visited on 2021-04-17]. Available from: https://flutter.dev/.

13. *Desktop support for Flutter* [online]. Mountain View, California, USA: Google, 2021-05-17 [visited on 2021-05-19]. Available from: https://flutter.dev/desktop.

14. *Web Native* [online]. Madison, Wisconsin, USA: Ionic [visited on 2021-05-18]. Available from: https://webnative.tech/.

15. *Capacitor: Cross-platform native runtime for web apps* [online]. Madison, Wisconsin, USA: Ionic [visited on 2021-04-17]. Available from: https://capacitorjs.com/.

16. *Definition of Javascript* [online]. Stamford, Connecticut, USA: Gartner [visited on 2021-05-18]. Available from: https://www.gartner.com/en/information-technology/glossary/javascript.

17. *Definition of HTML* [online]. Stamford, Connecticut, USA: Gartner [visited on 2021-05-18]. Available from: https://www.gartner.com/en/information-technology/glossary/html-hypertext-markup-language.

18. *Cascading Style Sheets* [online]. W3C, 2021-05-13 [visited on 2021-05-18]. Available from: https://www.w3.org/Style/CSS/.

19. *Quick Start Guide* [online]. San Francisco, California, USA: OpenJS Foundation, 2021-05-13 [visited on 2021-05-19]. Available from: https://www.electronjs.org/docs/tutorial/quick-start.

20. *Definition of Extensible Markup Language (XML)* [online]. Stamford, Connecticut, USA: Gartner [visited on 2021-05-20]. Available from: https://www.gartner.com/en/information-technology/glossary/xml-extensible-markup-language.

21. EISENMAN, Bonnie. Learning React Native : building mobile applications with JavaScript. In: Sebastopol, California, USA: O'Reilly Media, 2015, chap. 1, What Is React Native? ISBN 978-1-4919-2900-1.

22. *Native Modules Intro* [online]. Menlo Park, California, USA: Facebook, 2021-03-12 [visited on 2021-04-15]. Available from: https://reactnative.dev/docs/native-modules-intro.

23. *Unicode* [online]. Unicode, 2021-02-10 [visited on 2021-05-19]. Available from: https://unicode.org/glossary/#unicode.

24. *Test Renderer* [online]. Menlo Park, California, USA: Facebook, 2021 [visited on 2021-05-13]. Available from: https://reactjs.org/docs/test-renderer.html.

25. *ECMA-404* [online]. 2nd ed. Geneva, Switzerland: Ecma International, 2017-12 [visited on 2021-05-19]. Available from: https://www.ecma-international.org/wp-content/uploads/ECMA-404_2nd_edition_december_2017.pdf.

26. *Getting Started* [online]. Wrocław, Poland: Callstack, 2021 [visited on 2021-05-13]. Available from: https://callstack.github.io/react-native-testing-library/docs/getting-started/.

27. GAO, Jerry Zeyu; TSAO, H.-S. Jacob; WU, Ye. Testing and Quality Assurance for Component-Based Software. In: Norwood, Massachusetts, USA: Artech House, c2003, chap. 6, Black-box testing methods for software components. ISBN 978-1-5805-3480-2.

28. *Continuous Integration* [online]. Chicago, Illinois, USA: Martin Fowler, 2006-05-01 [visited on 2021-05-10]. Available from: https://martinfowler.com/articles/continuousIntegration.html.

29. *A Comparison of NoSQL Database Management Systems and Models* [online]. New York, New York, USA: DigitalOcean, 2014-02-21/2019-08-09 [visited on 2021-04-20]. Available from: https://www.digitalocean.com/community/tutorials/a-comparison-of-nosql-database-management-systems-and-models. License: CC BY-NC-SA 4.0.

30. *GridFS* [online]. New York, New York, USA: MongoDB, 2021-04-08 [visited on 2021-04-20]. Available from: https://docs.mongodb.com/manual/core/gridfs/.

31. CANKAYA, EbruCelikel. Authentication. In: ed. by TILBORG, Henk C. A. van; JAJODIA, Sushil. Boston, Massachusetts, USA: Springer US, 2011, pp. 61–62. ISBN 978-1-4419-5906-5. Available from DOI: 10.1007/978-1-4419-5906-5_772.

32. *Form Recognizer prebuilt receipt model* [online]. Palo Alto, California, USA: Docker, 2021-04-30 [visited on 2021-05-20]. Available from: https://docs.microsoft.com/en-us/azure/cognitive-services/form-recognizer/overview.

33. *JavaScript Environment* [online]. Menlo Park, California, USA: Facebook, 2021-03-12 [visited on 2021-05-02]. Available from: https://reactnative.dev/docs/javascript-environment.

34. PATEL, Ajay; SANDS, Alexander; CALLISON-BURCH, Chris; APIDIANAKI, Marianna. Magnitude: A Fast, Efficient Universal Vector Embedding Utility Package. In: *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing:*

System Demonstrations [online]. Brussels, Belgium: Association for Computational Linguistics, 2018, pp. 120–126 [visited on 2021-04-01]. Available from DOI: 10.18653/v1/D18-2021.

35. JURAFSKY2020, Daniel; H. JAMES, Martin. Speech and Language Processing: An introduction to natural language processing, computational linguistics, and speech recognition. In: [online]. 3rd ed. draft. c2020, chap. 6, Vector Semantics and Embeddings [visited on 2021-04-29]. Available from: https://web.stanford.edu/~jurafsky/slp3/6.pdf.

36. MIKOLOV, Tomas; GRAVE, Edouard; BOJANOWSKI, Piotr; PUHRSCH, Christian; JOULIN, Armand. Advances in Pre-Training Distributed Word Representations. In: *Proceedings of the International Conference on Language Resources and Evaluation (LREC 2018)* [online]. 2018, chap. 3, Training data, pp. 2–3 [visited on 2021-04-29]. Available from: https://arxiv.org/pdf/1712.09405.pdf.

37. *word2vec* [online]. Mountain View, California, USA: Google, 2013-07-30 [visited on 2021-05-20]. Available from: https://code.google.com/archive/p/word2vec/.

38. BOJANOWSKI, Piotr; GRAVE, Edouard; JOULIN, Armand; MIKOLOV, Tomas. Enriching Word Vectors with Subword Information. *Transactions of the Association for Computational Linguistics* [online]. 2017, vol. 5, p. 137 [visited on 2021-04-29]. ISSN 2307-387X. Available from DOI: 10.1162/tacl_a_00051.

39. GONZALEZ, Rafael C.; WOODS, Richard E. *Digital Image Processing: Third Edition*. 3rd ed. Upper Saddle River, New Jersey, USA: Pearson Prentice Hall, 2008. ISBN 978-0-1316-8728-8.

40. *Image Thresholding* [online]. OpenCV, 2021-04-02 [visited on 2021-05-08]. Available from: https://docs.opencv.org/master/d7/d4d/tutorial_py_thresholding.html.

41. *Cognitive Services Pricing - Form Recognizer API* [online]. Redmond, Washington, USA: Microsoft [visited on 2021-04-28]. Available from: https://azure.microsoft.com/en-us/pricing/details/cognitive-services/form-recognizer/.

42. *Cloud Firestore Pricing* [online]. Mountain View, California, USA: Google, 2021-05-17 [visited on 2021-05-18]. Available from: https://cloud.google.com/firestore/pricing.

43. *Cloud Storage Pricing* [online]. Mountain View, California, USA: Google, 2021-05-17 [visited on 2021-05-18]. Available from: https://cloud.google.com/storage/pricing.

44. *Container Registry Pricing* [online]. Mountain View, California, USA: Google, 2021-05-17 [visited on 2021-05-18]. Available from: https://cloud.google.com/container-registry/pricing.

45. *Cloud Run Pricing* [online]. Mountain View, California, USA: Google, 2021-05-17 [visited on 2021-05-18]. Available from: https://cloud.google.com/run/pricing.

46. *Cloud Build Pricing* [online]. Mountain View, California, USA: Google, 2021-05-17 [visited on 2021-05-18]. Available from: https://cloud.google.com/build/pricing.

47. HUANG, Zheng; CHEN, Kai; HE, Jianhua; BAI, Xiang; KARATZAS, Dimosthenis; LU, Shijian; JAWAHAR, C. V. ICDAR2019 Competition on Scanned Receipt OCR and Information Extraction. In: *2019 International Conference on Document Analysis and Recognition (ICDAR)* [online]. 2019, pp. 1516–1520 [visited on 2021-05-11]. ISBN 978-1-7281-3014-9. Available from DOI: 10.1109/ICDAR.2019.00244.

48. *FREE Receipt Images – OCR / Machine Learning Dataset* [online]. Walnut, California, USA: ExpressExpense, 2019 [visited on 2021-05-06]. Available from: https://expressexpense.com/blog/free-receipt-images-ocr-machine-learning-dataset/.