Software Requirements Specification

Receipt Reader Mobile App

Brain Johnson, Ibrahim Sydock, Jacob Fisher, Ichinnorov Tuguldur, Jatinder Singh, Julio Espinola Rodas, Mason Ringbom, Nathnael Seleba, William Ent

06/03/2023

Table of Contents

1.0 Introduction

1.1 Purpose

The purpose of this document is to explain the Receipt Reader mobile app, how it functions, how the software supporting it works, and what its use to the user is. The rest of this document will go over a general description of use, limitations, and design. Covering interfaces, functional and non-functional requirements, and usability. As well as how separate components of the system interact and a high-level design of how they are made.

1.2 Definitions, Acronyms, Abbreviations

UI - Refers to the mobile app user interface. What the user interacts with.
OCR - Ocular character recognition. How the receipt data is extracted.
Tesseract.js - An OCR library to be used by Ts or Js.
Ts/Js (Typescript/Javascript) - Typescript a coding language that is Javascript with explicit types. Javascript coding language. Both languages are used for our mobile app, server side for manipulating and storing data, as well as our UI.
React Native - Library used to build out our mobile UI supports Android and IOS.
Docker - A public platform able to package a part of an application independently from the project as a container.
Expo - is an open-source platform for making universal native app for Android, iOS, the web with JavaScript and React.

1.3 Intended Audience and Reading Suggestions

The intended audience is anyone interested in learning more about how the Receipt Reader runs behind the scenes and what is necessary to drive it, as well as developers looking to understand the software system's different parts and expected functionality. It is suggested to refer to the table of contents above if you are looking for an individual section of the system. General users will get the most out of the Overall Description and System features whereas developers should cover all parts.

1.4 Product Scope

The product is a Receipt Reader mobile app. This app works by the user scanning a receipt from a store and reading the purchased items. This is then logged on to the user account, allowing them to see purchase spending habits, create budgets, and compare stores to purchase goods. The user only interacts with the mobile app UI. The receipt reading and processing as well as the database is all behind the scenes.

1.5 References

"IEEE Guide for Software Requirements Specifications," in IEEE Std 830-1984 , vol., no., pp.1-26, 10 Feb. 1984, doi: 10.1109/IEEESTD.1984.1192

Facebook. “Facebook/React-Native: A Framework for Building Native Applications Using React.” *GitHub*, https://github.com/facebook/react-native#-requirements.

Tesseract. “Tesseract OCR.” *GitHub*, https://github.com/tesseract-ocr/tesseract

2.0 Overall Description

2.1 Product Perspective

The Receipt Reader software is a new software acting as a replacement or alternative to other receipt-reading mobile applications. Brought on from the need to aggregate and organize users’ individual spending habits and help promote financially positive decisions.

2.2 Product Functions

Receipt Reader software has a few significant functions. To be able to quickly and smoothly scan and process user receipt data along with storing this data and to be able to track spending and keep track of a custom user budget.

2.3 User Classes and Characteristics

Types of users that use this product and will benefit from it are any user that belongs to a class that purchases groceries and goods as well as having access to a mobile device. The most common type of user will be one that is financially literate and strives to stay on budget or measure personal spending. Any general education level can interact with this software. The most important user class is the financially lower-middle-class users that are using this app for budgeting. The software needs to ensure accuracy for the user relying on this data.

2.4 Operating Environment

The Receipt Reader software can operate in any environment that allows the user to take a clear photo or upload a saved image. The app must be able to coexist with a camera on a user's device and or the user's saved images to upload them to the app. The software will operate on any IOS or Android device that can support taking photos or storing images. More on specific hardware specifications and support in *3.2.1 Supported Devices.*

2.5 Low Level Design

2.5.1 Tesseract OCR

The Tesseract OCR that we implemented utilizes a link to an image and provides character data based

on the image that was provided within the link. Within our codebase, the raw data that is output includes specific line data, based on horizontal lines and a corresponding confidence rating based on the individual confidence of the words within each line. As stated on Tesseract's about page: "Tesseract 4 adds a new neural net (LSTM) based OCR engine which is focused on line recognition but also still supports the legacy Tesseract OCR engine of Tesseract 3 which works by recognizing character patterns." This quote further explains in technical detail what is produced by the used version of Tesseract and what data is being analyzed based on the provided image.
The images that are provided by us utilized an image preprocessing step, an attempt to binarize the image to improve edge quality and attempt to improve the overall result of Tesseract's output. "Tesseract supports various image formats including PNG, JPEG, and TIFF. Tesseract supports various output formats: plain text, hOCR (HTML), PDF, invisible-text-only PDF, TSV, and ALTO (the last one - since version 4.1.0)." The currently uploaded and taken photographs provided to Tesseract are in the form of "jpeg" and the plain text output format.
After the application receives the plain text output, the data is formatted into a two-dimensional array, where the first dimension is the individual raw data of the line, and the second dimension is the overall confidence of the corresponding line. This array of data is then sent to an item extraction module to manipulate that data into a receipt format.
To transform the raw data into a receipt format, we follow a three-step process. First, we use the Levenshtein distance algorithm to extract the store name accurately. Once we have the store name, we use custom regular expressions tailored to the specific store to identify and extract relevant information such as the item name and its price. Finally, we format the extracted data into a receipt format for the database.

The Levenshtein distance algorithm is used to calculate the minimum number of character edits that are needed to transform one string into another. The way we use the algorithm in our software is we compare the words that were extracted using the OCR with a dictionary of store names and if the word from the extracted text is a 50% or more match then it is corrected to the name of the store in the dictionary that it was close to matching. This process reduces the risk of errors due to typos or spelling mistakes and enables us to identify the correct store name with a high degree of confidence.

Regular expressions are a sequence of characters that define a search pattern. Once we have the store name we can use regular expressions that are tailored to the store we extracted in the first step. For example, if we determined that the name of the store is "Fred Meyers" we know to search for a pattern that extracts text following a 9-11 digit number because all Fred Meyers receipts have their item name and price on the same line as the 9-11 digit number. By defining and applying custom regular expressions, we can efficiently extract the desired information from the unstructured data and finally change it into a receipt format.

2.5.2 Database

The softwares database is a postgreSQL database hosted on Supabase this consists of three tables. This consists of the user table, the receipt table, and the item table. The database will be interacted with by the multiple layers that contain database functions.

2.5.2.1 User Table

The user table has the columns user_id INT, username TEXT, password TEXT, budget NUMERIC. This table auto generates the user_id the unique identifier used within the app to identify user data. The username is stored as a unique field to not allow multiple users to have the same username. The password is a string that has been hashed by bcrypt to be safely stored within the database. This is used to with the username to login in and access the users_id and data. The budget is a number stored in the user table to be identified by the user_id. This is where we create, read, update, or delete the user's current budget.

2.5.2.2 Receipt Table

The receipt table contains receipt_id INT, user_id INT, store_name TEXT, total NUMERIC, date TIME, url TEXT This table contains an autogenrater for the receipt_id, this id allows us to perform CRUD operations on individual receipts. user_id is used in this table to separate what users data we can access. The store_name field contains the data passed by a users uploaded receipt on insert. The total is the scanned or user entered amount for the receipt that is stored on upload. The column date is a DateTime object stored that is the date on the receipt, the user entered date, or the date of entry. Lastly the table contains url which is the url to the hosted image of the receipt.

2.5.2.3 Item Table

The item table is the last table our user contains which holds the lowest level user data. The item table consists of the fields item_id INT, receipt_id INT, user_id INT, item_name TEXT, price NUMERIC. item_id is the unique identifier for each item. The three id's are used to determine whose data the item belongs to, which receipt, and what item id in the case of duplicates. This allows us to safely perform all CRUD operations on items. item_name is the user inputted or scanned item name from the receipt. price is the user inputted or scanned price for the individual item. Each item row will belong to a receipt row and is identified with the receipt_id.

2.5.2.4 Edge Functions

The Receipt Reader software depends on edge functions to support our infrastructure and handle multidatabase calls. The software uses Supabase hosted edge functions which are always running. The software relies on the receipt edge function. This takes in a POST of data from the mobile app containing a Receipt object. Which consists of the fields user_id, url, store_name, total, and an array of Item objects that contain item_name and price. The edge function on retrieval of POST data from the mobile app will insert the receipt data of the POST data into the receipt table for the passed user_id. This means user_id, store_name, total, and url will be inserted. After this insert the receipt_id will be returned as a response from inserting the new receipt. The data's Item object array will be looped and user_id and receipt_id will be added to each Item object in the array. Then a bulk insert of the item data will be used to insert all the items for that receipt into the item table. Following this a response from the edge function will be returned alerting a success or failure to the mobile app.

2.5.2.5 Storage Bucket

The software to run needs a way to store and access images of receipts. The OCR needs this to scan a user's receipt and extract data as well as the mobile app to display past receipts. Database tables cannot efficiently store image files so the image itself is hosted. We used Supabase hosted buckets, a storage solution for large files, to host these images. The storage bucket interacts with the mobile app upon a picture being uploaded or taken by a user. The image base 64 data is then passed to the bucket along with the URI's extension to know the file type. This is stored with a name of the format (user_id + "Image" + user's receipt count + "." + file type). The bucket then returns a url for the OCR to access and the database to store for future use upon success. Or the mobile app is notified of a failure upon an error occuring.

2.5.3 Expenses

The Mobile Application needs a way to analyze the data that gets stored in the database. We decided to use the Expenses page, which uses a public library known as React-Native-Chart-Kit to generate a dynamic bar graph to display the user's spending data in an easy-to-read format.

The Expenses page has four main buttons on it that populate the bar graph with different data over variable time spans.
The first of these is the ‘Weekly’ button. This populates the graph with the total spent of the current week, located in the ‘W4’ bar, and the previous three weeks, located in the ‘W3’, ‘W2’, and ‘W1’ bars. This allows the user to compare their current spending for the week against their spending for the previous weeks to help keep their spending in check.
The second button is the ‘Monthly’ button. Much like the ‘Weekly’ button, this button populates the bar graph with the total spent during the current month and the previous three months, to allow the user to compare each month’s data. The ‘Yearly’ button is also similar, however it populates the graph with the data from the current year’s spending and the previous three years’ instead.
Lastly, the fourth button is the ‘Quarterly’ button. This populates the graph with the total spent of the current quarter and the previous three quarters, letting users compare their spending over roughly the last year.
To accomplish these functions, the graph takes advantage of the ‘Date()’ object in Javascript to find the current date, and uses it in combination with different algorithms to find the start and end of each variable time span. These mathematically determined dates, along with database calls using the functions located in ‘aggregationFun.js’, allows the Expenses page to quickly find the specific data specified by the user’s button press.

2.5.4 Docker

In order to run the Tesseract OCR separately from the mobile app, it needed to be packaged independently so it could run anywhere on any computing device. Docker provided the most straightforward and convenient way to contain the OCR and provide a public, consistent, and deployable solution. To create the OCR as a container, a read-only template had to be created so that Docker knows how to build the container. This is done using a DockerFile which holds the necessary instructions.
The first instruction dictates the parent image to build from, which in this case would be node.js version 14, as the Tesseract is a javascript library. All javascript files and the package.json files are included using the COPY instruction so that the container knows that those are the files that build the OCR software. A RUN instruction is also used so that the dependencies will automatically be installed, allowing the app to run correctly and without user input. Finally, the EXPOSE instruction is used so that the container knows that wherever it is hosted, it will be done on port 300, and the CMD instruction will start the software immediately after the dependencies are installed. This allows for immediate use once the container is built from the image.

2.5.5 AWS EC2

After hosting our OCR in Docker, AWS EC2 allows our application to access it from anywhere without the need for us to manage the server on our own. Upon clicking the upload/save button on the mobile application, the image will be sent to the database, which will return the image URL. This URL will then be transmitted to our server or EC2 instance, which in turn will send it to the OCR engine for data extraction. The extracted data will be formatted, and the EC2 instance will then relay it back to our application.

To elaborate on this from a more technical perspective: In order to communicate with the AWS EC2 instance, our React Native mobile application will use RESTful APIs over HTTPS. Based on a RESTful architecture, APIs will respond to requests using JSON. APIs are developed using Node.js and Express, and they have access to the necessary resources. A Virtual Private Cloud (VPC) hosts the EC2 instance behind the scenes for security and isolation. You can monitor the EC2 instance with Amazon CloudWatch or a

local EC2 instance, which provides real-time metrics on resource utilization, network traffic, and more. The system will be audited and transparent using AWS CloudTrail to record API calls and changes. The design we use ensures secure, scalable, and reliable communication between our mobile app and an AWS EC2 instance.

2.5.6 Mobile App

To build the mobile app UI, React Native is required since it has to work on both Android and iOS. Then on both devices expo should be installed. Expo is going to be used to develop the application.

SignIn page

The first thing the user is going to see when they run the app through Expo is the sign-in screen. On the sign-in screen, there is a logo, username textbox, password textbox, a forgot your password button, a sign-in button, and create account button respectively. Users are going to have three options: to sign in with their credentials, click the forgot password button which the app is going to navigate to the forgot password screen, or if the user wants to create an account click “Create Account” which navigates to create an account page.

Forgot Password

Once a user clicks on the forgot password button, the user is redirected to the forgot password screen which is going to have the logo of the app in the middle, and three textboxes in which the user is going to insert their username, create a password, and confirm password respectively. And lastly, a reset button is going to redirect the user to sign in with the credentials that they just inserted.

Create Account

Once the user clicks the create account button on the sign-in page they are going to be directed to the create account screen. The create account page is going to have the username, create a password, and confirm password text boxes followed by the signup button. Once a user enters their credentials and clicks on the sign-up page they are going to be directed to the dashboard screen.

Dashboard tab

Once the user signs in the dashboard screen are going to have a text on the top telling the user their weekly spending. Then there is going to be a set budget button where the user is going to set a budget and that budget is going to be displayed after “This Week’s Budget:” on the dashboard. Then in between the set budget and the “This week’s budget,” there should be a chart where the closer the user spends money and get closer to the budget the chart color is going to go from green to red. Then below it is going to show the most 5 recent receipts taken with an option to show all receipts uploaded. The example below illustrates how the dashboard screen is going to look like.

Set budget

When a user clicks on the set budget button on the dashboard they are going to be directed to the budget screen which has a textbox for the user to input a budget and an update budget button that redirects them to the dashboard screen.

Expenses tab

When the user clicks on the expense tab, it navigates them to the expense screen. The expense screen is going to have a bar graph that is going to show user spending based on buttons below the graph: Weekly, Monthly, Quarterly, and Yearly. Then whichever option the user clicks is going to show them their expense graph.

Camera tab

When a user clicks on the camera tab, they are directed to the camera screen, where they can take a photo or

upload a photo. In the top right corner, they have the option to turn the flashlight on and off. On the bottom left corner, there is an upload button where users are directed to their phone gallery to select a photo of a receipt. In the bottom middle, they can take a picture by clicking the camera icon button. Once the user clicks the camera capture button they are going to see three options: Save/Upload, Re-take, and crop.
The crop button navigates the user to the photo they took and lets them crop and lets then either go back to the previous screen or save the image they crop which redirects them to the previous screen.
The Retake button lets the user retake a photo.
The Save/Upload directs users to the edit page where we can see the intelligently extracted receipt with the store name, items with their prices, and the total. We can edit the item name and price, and also have the option to delete the item and price. We can edit the extracted and also add items and prices if the app didn't fully add every receipt item. Then lastly we have a save button which saves the whole data.

Profile tab

When the user clicks on the profile tab, they are going to see "Welcome" text with their user name and how many receipts they have uploaded. Then they have two buttons: About and Sign Out. When the user clicks the about button they are redirected to the About Screen where they see the people who worked on the project and which organization they represent. When the user clicks on the sign-out button on the profile tab they are signed out and redirected to the sign-in screen.

2.6 User Documentation

The user will have access to the help and FAQ sections of the mobile app and the ability to send an email to the developers for user support. Email is the main and only form of contact for the developers for help not listed in the FAQ section. There will be no standard support time for responses.

2.7 Assumptions And Dependencies

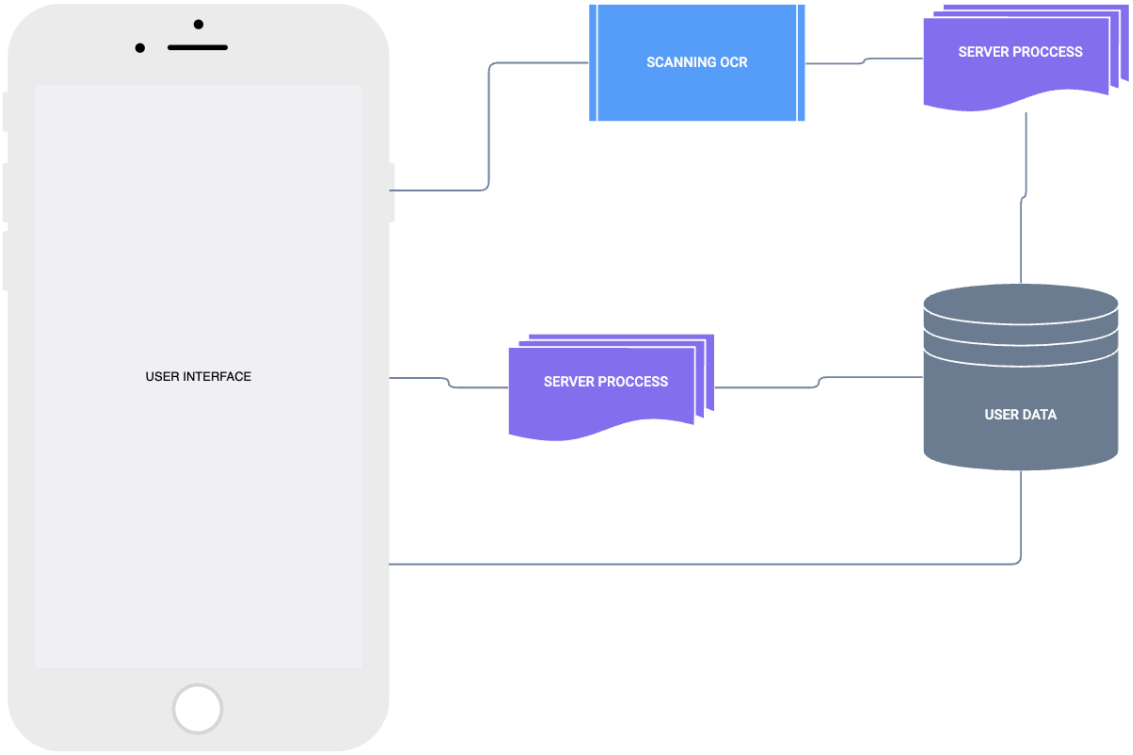
The Receipt Reading System depends on Tesseract.js remaining supported and functioning. It is the driver of our whole system and what extracts user data from receipts. As well as AWS and our Database both continue to support us and provide our backend architecture. Lastly, the IOS / Android OS that the user runs can function with our mobile app.

3.0 External Interface Requirements

3.1 System Design Overview

The system design overview shows the design of the below interfaces and how they work as well as interact with each other.

3.1.1 Interface architecture design



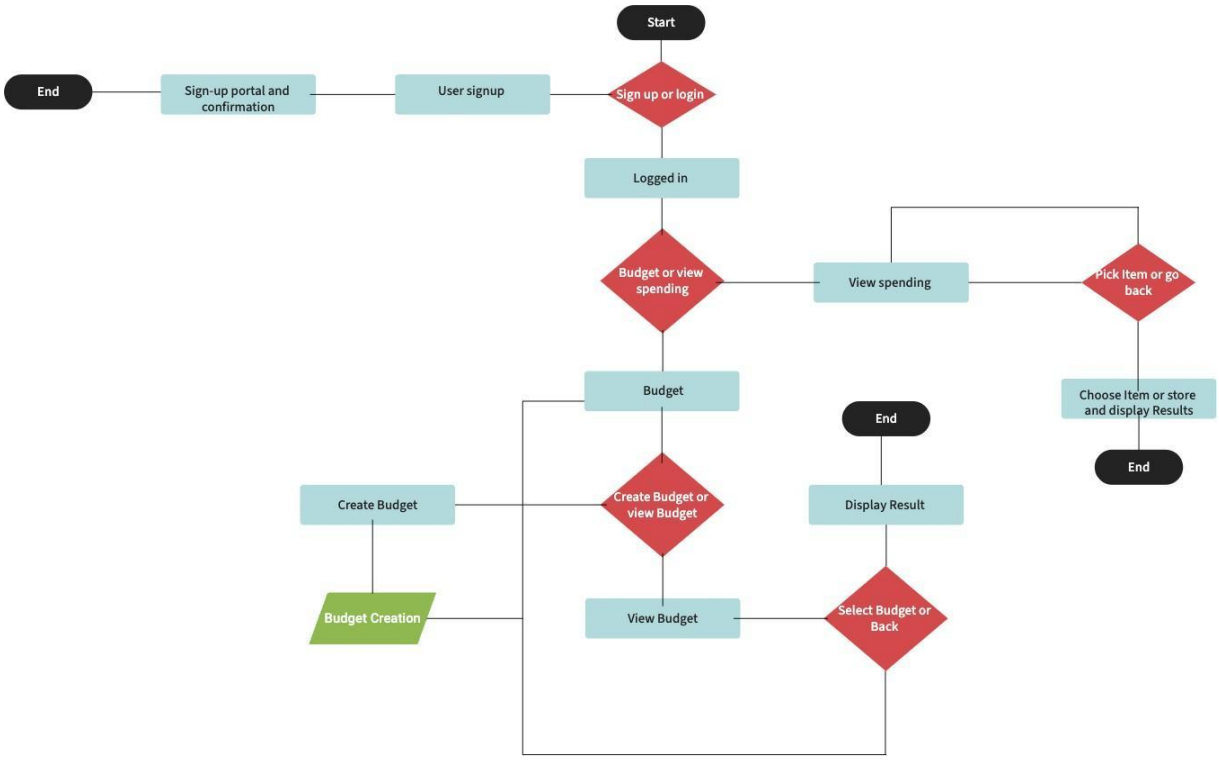
3.2 User interfaces

3.2.1 System User Interface Mobile App

The user interface is the mobile app for the receipt reader software. The user app is where the user can interact with the receipt reading technology by scanning a new receipt or uploading an image of a receipt. The mobile app UI maintains a priority on simplicity and restricting any erroneous calls and commands.

The user interface needs a home button to always allow the user to get back to home, A logout button and sign-in button to allow changing of displayed data, A simple uploading system to add receipt data, A clear directory to walk through creating a budget, and the system needs a general info section for comparing spending history and cheapest options.

3.2.2 Mobile App Decision Mock



3.3 Hardware Interfaces

The Receipt Reading software only supports mobile hardware. The hardware running the Receipt Reader system does not need a camera as the user can upload photos as files. The user's without a camera on their mobile device will be limited and unable to scan receipts with a camera. The mobile app on the mobile hardware will connect the user to the user's data stored in the database. Due to processing data, displaying data, and supporting react components there are limited devices supported. According to the React-Native GitHub, we may target devices that support IOS 12.4 and Android 5.0 (API 21) or above.

3.3.1 Supported Devices

3.3.1.1 IOS Devices - IOS 12.4+

At minimum the user must use a device listed below or newer.

- Iphone 5s
- Iphone 6
- Ipad Air

- Ipad Mini 2

3.3.1.2 Android Devices - Android 5.0+

There are numerous Android / Google phones not listed,
At minimum the user must use a device listed below or newer.

- Samsung Galaxy Note 3
- Samsung Galaxy s5
- HTC One

3.4 Software Interfaces

3.4.1 Database

Database connections are from the system's mobile app to the database to upload new user data. As well as from the database back to the system to populate saved user data as well as user authentication.

3.4.2 Tesseract Js

Interacts with the system as the main OCR library. Takes the system's mobile app scanning of a receipt photo or a file from the user's device and scans the text to identify and extract receipt information and items.

3.4.3 Server Host

The OCR library is held in a container on a server to accept and evaluate image URLs provided by the mobile app. It can be hosted on any computing device anywhere. A URL is obtained through the mobile app sending a post request to the container with the following response holding the receipt information.

3.4.4 React Mobile App

Mobile app language for IOS and Android operating systems. The Mobile app interacts with the user displaying data and information. The Mobile app UI is the software interface that is at the start of initiating uploading data and the endpoint for receiving stored data.

3.5 Communications Interfaces

Communications for this product are for user verification. On user signup or changing credentials a medium of communication will be needed to interact with the server / database for signing up or changing info. This can be done through an email.

4.0 System Features

4.1 User Budgeting

4.1.1 Description and Priority

A high priority feature for the receipt reader software. Other than the actual scanning of receipts this is the main feature of the software. The user can use this functionality to create budgets to monitor spending versus current spending habit data.

4.1.2 Stimulus / Response

The user will click on make a budget to begin the budget creation process. The user will then be walked through a budget creation tutorial by the app allowing them to use receipt data to craft a budget.

4.1.3 Functional Requirements

Users do not need data to create a budget. The budget will last until the user deletes it. The budget needs the database and UI to be created and maintain information.

4.2 User Account Creation

4.2.1 Description and Priority

Critical priority feature. The feature is needed to protect the user’s data and allow the user to access the receipts they have logged. Without this portal, the user’s data wouldn’t be protected on the device they use. This also is how the system decides the data being accessed.

4.2.2 Stimulus / Response

On the mobile app launch, the user signup portal will populate. The user will have the option to sign in or create an account. On selection of sign in the user will be able to sign in with a combination of the username and password. On sign-up, the user will be able to create an account with our system.

4.2.3 Functional Requirements

The system needs a database and communication through email to sign up the user. The system needs to respond to blank inputs and inputs that don't follow the rule set properly.

5.0 Nonfunctional Requirements

5.1 Performance Requirements

The user's data needs to load in an appropriate amount of time as they open the mobile app and log in, Within 3 seconds as the app loads. The receipt scanning needs to load within 1-3 seconds per receipt since this can happen asynchronously in the background; it can be increased if we allow the user to move on while the data processes. There cannot be any user lag when accessing budgets or other item-based data.

5.2 Safety Requirements

User data must be protected and secured with encryption or some other means. No leaks should be possible, possibly exploiting the user, what they buy, or where they shop. No physical risk in using the app.

5.3 Security Requirements

The user must use a form of secure log in to access data and identify that it is the user. The data stored must be accessible to anyone but the user.

5.4 Software Quality Attributes

The software shall be modular as the number one quality. The receipt scanning shall be portable to scan from any input as long as it is passing in an image. The database will be hosted and able to be called for inputting or extracting data through any application that can access a database. The mobile app should be very robust given the simplicity of the app and the control it keeps over the user errors should be non-existent.

5.5 Business Rules

No personal user data may be accessed by anyone but that user. General area data such as the lowest price of an item in the area may be accessed by anyone this is general info. Any admin function shall not be able to access specific user info.

7.0 Installation

7.1 Node JS

The majority of the mobile app runs using javascript libraries, thus it is important to download all the dependencies necessary for the software to function. Make sure to follow the steps in order to ensure the best installation experience.

1. Download the repository as a zip file (https://github.com/Reciept-reader/reciept-reader).
2. Unzip the contents and place it in whichever directory you like.
3. Navigate to the ./MobileApp directory.
4. Open a new terminal or one already in use.
5. Enter the command "npm i" (without quotes). This will automatically download the dependencies required to run the application. This includes libraries such as Expo, React, and Supabase which all provide the structure to the mobile app and its database connections.
6. Then enter the "npm start" command to start running the project.
7. After the project is running, a QR code will be printed to the terminal.
8. Either download Expo Go on your mobile device and scan the code,
9. OR run it through an emulator (such as Android Studio for Windows or Xcode for Macs) installed on your device.
10. The mobile device and computer must be connected to the **same exact** network.
11. Follow 7.2 or 7.3 to get the OCR container running.
12. Enjoy the data returned from scanning any receipt you take a photo of!

7.2 AWS EC2

1. Stop running the app if it is started.
2. Generate a key pair for the EC2 instance following the instructions here: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/create-key-pairs.html
3. Download the generated key pair file.
4. Open a terminal or command prompt.
5. Change directory to the location of the downloaded .pem file (holds the key pairs)
6. Run the following command:ssh -i “downloadedFileName.pem" ec2-user@ec2-35-89-83-59.us-west-2.compute.amazonaws.com (where downloadedFileName.pem

is instead the value of the downloaded file name without the quotes).

7. Pull the repo from github on your local EC2 (all you need is the docker-container folder)
8. Change directory to the docker container folder
9. Run the “node index.js” command to start the container.
10. Run “host ip_address_here” to get the link to the EC2 container at the desired ip address (the value will look something like “ec2-___.compute-1.amazonaws.com.” Make sure to append https:// before it so that it can be accessed via HTTPS).
11. Change the constant “linkToContainer” in MobileApp/src/function/connectToOCRFun.js, line 4, to the new url where the EC2 is hosted.
12. Run the app again with the new link value.

7.3 Docker

If the user desires to have the container run somewhere other than AWS EC2, they can do so by downloading and building the ocr image here:
https://hub.docker.com/repository/docker/ibrahimsydock/tesseract-receipt-reader/general

Instructions on how to pull and run it can be found in its “README” description. Make sure to stop running the app and change the constant “linkToContainer” in MobileApp/src/function/connectToOCRFun.js, line 4, to the new url where the container is hosted. Then run the app again by entering “npm start” in the terminal that is pointing to the MobileApp directory.