MyAllergyAssistant - Technical Guide

‘MyAllergyAssistant - Android app to scan food ingredients for allergens’

Students:

- Daragh Prizeman - 19459734
- George Eskander - 19451972

CA400 Final Year Project

Supervisor: Paul Clarke

Date of Completion: 05/05/2023

Abstract

MyAllergyAssistant is an Android app that allows users to scan or search for a product and be informed if the product is safe for them to eat or not based on their account allergens. Users will also receive alerts if a product they have previously scanned is reported by another user for containing unlisted allergens. The app was developed using React Native with TypeScript for the frontend and a combination of AWS services for the backend, including Lambda, DynamoDB, Cognito, SNS.

Declaration

We declare that this material, which we now submit for assessment, is entirely our own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of our work. In our README.md, we outline which source files were created and modified by us. We understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should we engage in plagiarism, collusion or copying. We have read and understood the Assignment Regulations. We have identified and included the source of all facts, ideas, opinions, and viewpoints of others in the assignment references. Direct quotations from books, journal articles, internet sources, module text, or any other source whatsoever are acknowledged and the source cited in the assignment references. This assignment, or any part of it, has not been previously submitted by us or any other person for assessment on this or any other course of study. We have read and understood the referencing guidelines found at http://www.dcu.ie/info/regulations/plagiarism.shtml, https://www4.dcu.ie/students/az/plagiarism and/or recommended in the assignment guidelines.

Table of Contents

1. Introduction

1.1. Overview

Our project is an Android app called ‘MyAllergyAssistant’ which aims to help users with food allergies to be informed about whether or not a product is safe to eat.
Users can create an account, and select their food allergens, that will be linked to their account. Then when a user scans a product’s barcode, or searches for a product, our app will fetch the product data from Open Food Facts API, translate the output to English if not already, and using our allergen identification algorithm, the app will detect any of the user’s allergens within the output. The user will be informed if the product is either ‘Safe to eat’, ‘May not be safe to eat’, or ‘Not safe to eat’. If a product is not safe to eat, the app will display to the user which of their allergens was found in the product ingredients. Users can also view if the product has been reported by other users of our app for unlisted suspected allergens. When a user scans a product’s barcode, they are automatically opted-in to receive alerts if that product is reported by another user in the future, although you can disable alerts for each product if you wish.

Similarly, users can take a picture of the ingredients listed on the product packaging, to find out if it contains any of their user allergens. Our app uses optical character recognition (OCR) to extract text from the image, which we then run through our allergen identification algorithm to detect any of the user’s allergens and display the results to the user.

1.2. Motivation

We both live with people who have food allergies, so we are very conscious of what products we bring into the house while shopping for food. We feel that this app would help make the shopping experience easier, faster, and safer for both people with food allergies and for those who live with people who have allergies and are conscious about the products they bring home.

One potential use case, where MyAllergyAssistant could be especially useful, is if you are doing some food shopping while abroad, in Spain for example, and you want to know if a particular product contains any of your allergens, but you can’t read the label because you don’t speak Spanish. Using our app, you could simply scan the barcode of the product, and you will be shown whether or not the product is safe to eat, as well as being able to view all the product information such as the name of the product, ingredients, allergens, traces of allergens, and if any other users of our app have reported that product before.

We believe there is a need for this app because it will help keep users more informed about the products that they purchase and consume, as users are able to scan a product’s ingredients or barcode and see if it contains any of their allergens, or if the product has been reported by other users for containing any additional allergens not listed on the packaging. We hope that by keeping users informed about the products in this manner, that this app will help prevent consumers from having unexpected allergic reactions.

1.3. Glossary

AWS - Amazon Web Services, cloud services provided by Amazon
Firebase - Backend cloud services and application development platform provided by Google.
Cognito - AWS service, provides authentication, authorization, and user management for your apps
Google Cloud - Cloud services provided by Google, used for federated sign-in with Google for Cognito.
FCM - Firebase Cloud Messaging, cross-platform solution for sending messages without cost, used for mobile push notifications
SNS - Simple Notification Service, an AWS service that is a fully managed Pub/Sub service, allowing for A2P (applicaton-to-person) notifications for mobile push notifications for our use.
API - Application Programming Interface, it's a way for applications to communicate with each other.
REST - Representational State Transfer
API Wrapper - simplifies interaction with an API.
API Gateway - AWS service, allows you to create, maintain, and secure your APIs with scalability.
CRUD - Create, Read, Update, Delete operations used with data models.
Amplify - AWS service, used for simplifying the development of full-stack web and mobile apps, used with Authentication and API Gateway (see section 4.1.).
Lambda - AWS service, a serverless compute service which allows for functions or microservices in the cloud which are only run on-demand with scalability.
ECR - AWS service, Elastic Container Registry, Docker container registry making it easy to store and deploy Docker images. It is used for containerized Lambda functions.
DynamoDB - AWS service, gives you NoSQL databases with speed, flexibility, and scalability.
Open Food Facts - application which provides information about food products from around the world which can be retrieved by barcode, accessible via API call.
OFF - Open Food Facts abbreviation, see above
JavaScript - Programming language, most commonly used for the web.
TypeScript - JavaScript with types
React Native - Allows you to create cross-platform mobile apps using the React framework.
Redux - Manageable global store for centralising your application state.
Jest - Allows for testing of JavaScript/TypeScript-based apps.
OCR - Optical Character Recognition, recognises text from an image.
UI - User Interface
Noisy data - inaccurate or messy data, in this case referring to inaccurate text output produced by OCR

2. Research

Before beginning this project, we had no experience with the following technologies:

- AWS
- Message-oriented middleware (Publish/Subscribe), Push Notifications, and FCM
- OCR
- Barcode Scanning
- Image manipulation, image uploading, and OpenCV
- Persistent storage on mobile devices
- Working with / or setting up a camera, using JS worklets, and Vision Camera
- Federated Identity/Login
- Fuzzy algorithms (string similarity / matching)
- Open Food Facts
- Creating a microservice from scratch

As we had no experience beforehand with the majority of features and technologies involved in this app, we conducted a lot of research before beginning by reading through the documentation in section 12.2. When problems would arise during development, we had to learn on the go by debugging and searching for solutions online.

3. Development workflow & use of GitLab

The majority of the development of this app was conducted using a pair-programming approach using Visual Studio Code's 'LiveShare' feature, and JetBrains 'Code With Me' feature. This allowed us both to work in the same environment simultaneously. We found this beneficial as we could debug problems together and ensure that we both understood everything along the way.

There were some situations in which pair programming was not possible, such as when one person was unavailable during certain hours, etc. In this case, the other person would work locally, and then when we were both next available, we would review the changes together to ensure we both understood what the code is doing, and is implemented correctly.
Also, as AWS Lambda does not have a feature like 'LiveShare' or 'Code With Me' where we can work together simultaneously on the same code, one person would create the Lambda function and deploy it, then we can both review it, test it using print/console.log() statements and checking the CloudWatch logs to ensure we are getting the correct input and output etc.

We used a Trello board to break our project down into 36 subtasks (cards) of varying priority.

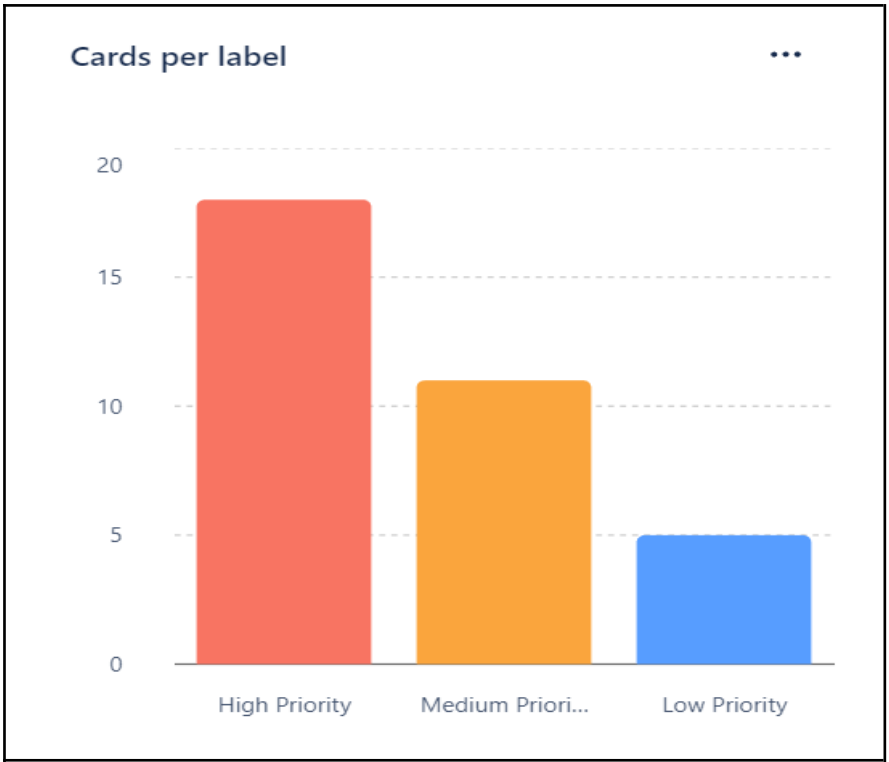

Figure 1. Breakdown of sub-task priority levels

This allowed us to work from a task backlog, and work our way through the project in a structured manner. It also helped us to visualise our progress along the way by seeing how many tasks were in the different categories: ‘To Do’, ‘In Progress’, ‘QA” and ‘Done’.

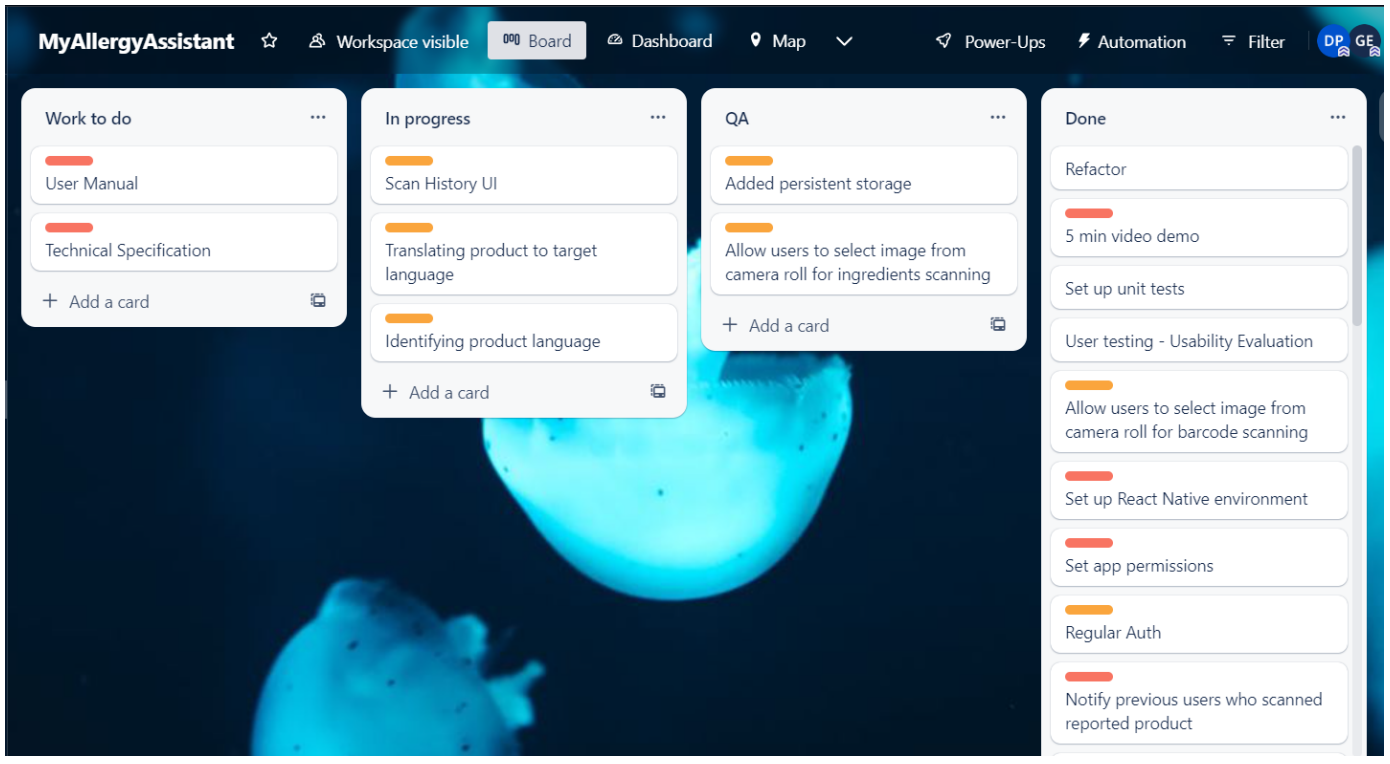

Figure 2. Our Trello board

We first began working on source code for this project on the 19th of November 2022. As as can be seen from our Git commit history, we took a break over Christmas and then returned to regular commits from the 29th of December 2022.

4. Design

4.1. System Architecture

Below is a high-level diagram illustrating the architecture of our system, as well as an explanation of what each component is used for.

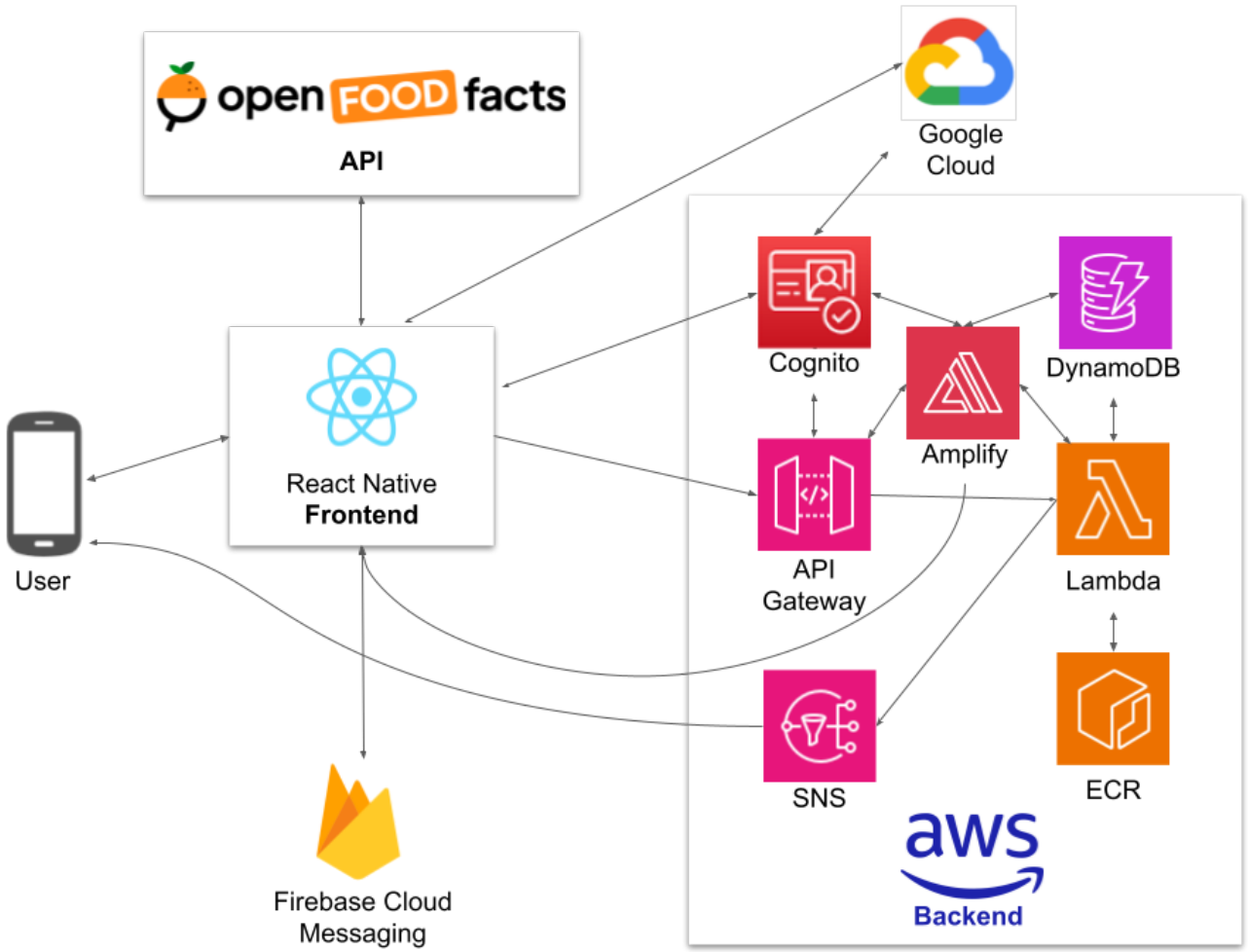

Figure 3. System Architecture Diagram

Frontend

First, here's a high-level explanation of the connections with the frontend in Figure 3.

Connections with Frontend	Explanation
User	App runs on the user's device, with app data being stored. Frontend displays the app to the user.
Open Food Facts API	Frontend communicates with OpenFoodFacts API to retrieve product information by barcode or search.
Firebase Cloud Messaging	Used for mobile push notifications, sends a token to the frontend to be registered by **SNS**
Google Cloud	Used for federated sign in, and for translation of product data to English.
Cognito	Used for authentication/authorisation. Managing our user pools.
API Gateway	Frontend makes calls to myAPI in API Gateway, which then triggers a **Lambda** function.
Amplify	Provides a simpler interface to our API and Auth (via an Authenticator component and an event hub listener to handle user authentication)

We developed the frontend of our application using React Native with React Native CLI, TypeScript, and Redux. The frontend makes use of our API wrapper which can make calls to AWS to communicate with our backend and Open Food Facts. Authentication is configured with Amplify for the frontend to simplify the Auth logic and provide a pre-built Authentication UI using our Cognito setup. Amplify is also used in our API wrapper to simplify our calls to our API.

Our application makes use of Firebase and SNS for mobile push notifications, where a FCM token (idempotent until expired) is created upon opening the app and sent to SNS in our backend, see section 6.3 and 6.4. This registers the app for background/foreground notifications, which is used for alerting the user of reports of products.

The frontend makes use of on-device OCR and barcode scanning through Google's ML Kit. These are coupled with our scanner, which uses a camera supporting real-time detection of barcodes and ingredients through a JS worklet which processes every frame. The camera or gallery can be used to input images for OCR or Barcode scans as well using Google's ML Kit. Our OCR post-processing step is also done on-device, which makes use of our allergen

identification algorithm. This algorithm is also used on the product ingredients retrieved from Open Food Facts through barcodes. For more information, see section 5.3. and 6.1 .

Our OCR preprocessing step is done in our backend as it is too much of a computationally intensive process for the frontend, for more information, see section 8.2. Google Cloud is used for translating product data prior to using the allergen identification algorithm for barcode scans.

For our testing framework, we used Jest for our automated tests.

Backend

We developed the backend with the use of AWS, Firebase, and Google services to create a serverless distributed microservice.

Cognito is used for our authentication and authorisation. We have two Cognito user pools where users can have accounts registered through Cognito either normally, or with federated sign-in for Google accounts which is done through configuration with Google Cloud and Cognito.

Amplify connects to the frontend for user authentication and provides configuration settings to create a pre-built UI for authentication using Cognito. Amplify is also used for simplifying our calls to our API from the frontend.

Simple Notification Service (SNS) is used for mobile push notifications. SNS Endpoints are generated by users through passing their FCM token to our REST API to SNS via Lambda. SNS is triggered from the reportsLambda, and is used to send users a notification when a product is reported. Endpoints are stored through notificationsLambda. Containerized lambda functions are managed or built using Docker images stored in ECR.

The main way users communicate with our backend is through our API hosted and managed on our API Gateway, myAPI. All of our API resources use both Cognito user pools as authorizers for authorization. All resources are based on single Lambda functions with CRUD. Each table has an associated Lambda function. There are other lambda functions such as for registering a FCM token with SNS or preprocessing an image.

For our testing framework, we use Jest for our automated tests. To test AWS services without affecting production, jest-dynalite is used for testing with a local DynamoDB instance, and lambda-local is used for creating local versions of our lambda functions which we can use in tests without cost and less latency. Mocks are created for other services such as SNS.

Open Food Facts API

Open Food Facts provides information about food products from around the world.

Our frontend communicates with Open Food Facts API directly through our API wrapper in the frontend, features used include:

- Retrieving products by barcode

- Searching for products through a query

Results are processed in the frontend once received from OFF's API for barcode scan results and search results.

4.2. UI / UX Design

When considering the usability of the app for users of the app, we focused on three key criteria.

- **User Control and Freedom**
- **Consistency**
- **Visibility of System Status**

Here are some examples where we tried to follow these criteria

User Control and Freedom: We give the user the freedom to scan in the way they want by selecting scan modes. This can be done from either Home or the Scan screen. They also have the freedom to toggle notifications on or off, and view their barcode scans again after taking them using Scan History in Profile.

Consistency: The safety indicators in scan results all have the same style.

Visibility of System Status: When entering the Scan screen or changing the scan mode, the user is informed of what mode they are now using. When changing your allergy profile and saving changes, a confirmation message will appear when changes have been saved.

We conducted user testing of the app via a usability evaluation and questionnaire. The feedback we received was addressed and overall feedback of the usability of the app was positive. For more information, refer to section 7.5. user testing.

5. Implementation

5.1. Data Flow Diagram

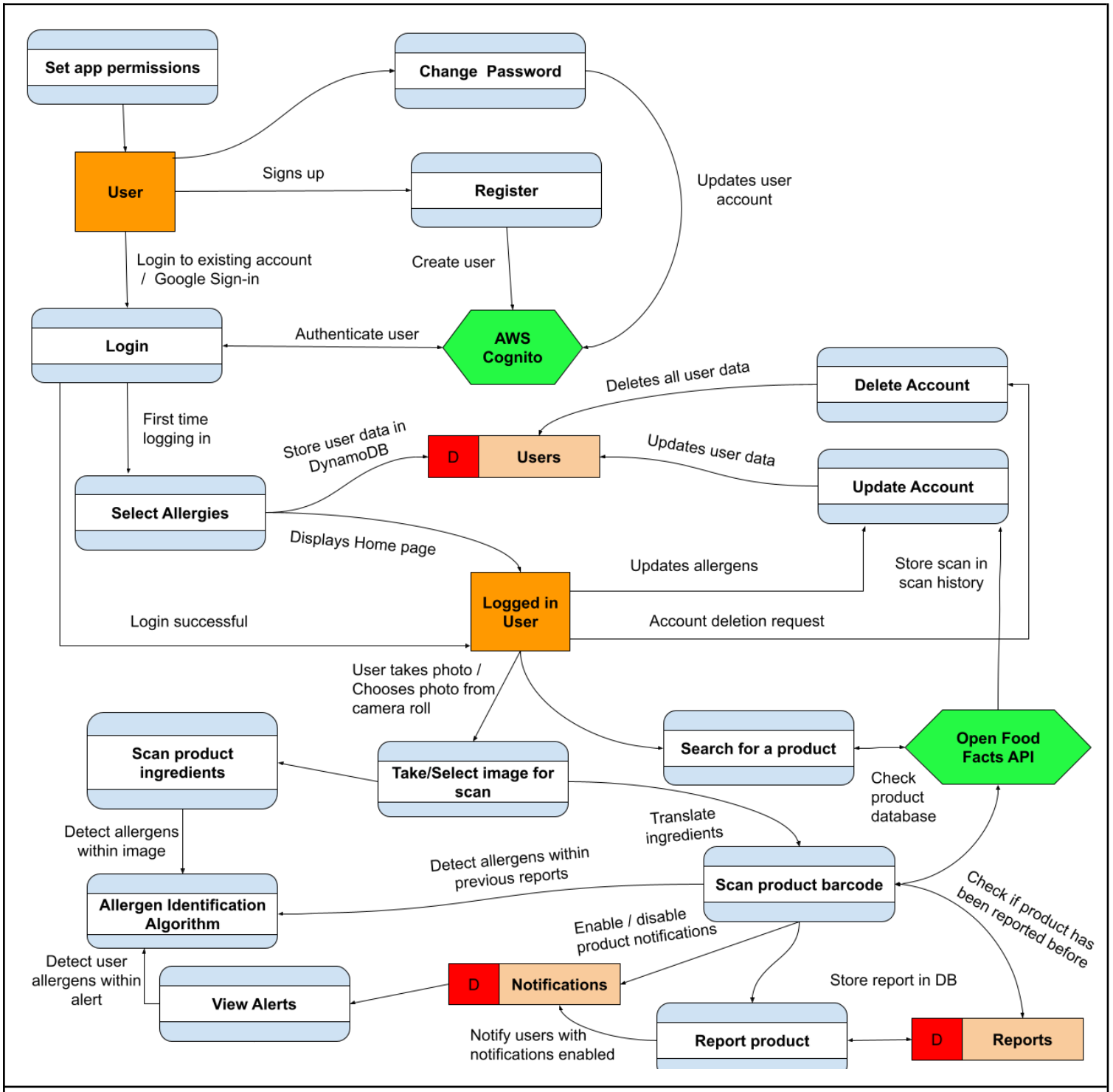

Figure 4. Data Flow Diagram for MyAllergyAssistant

5.2. Sequence Diagram of Login

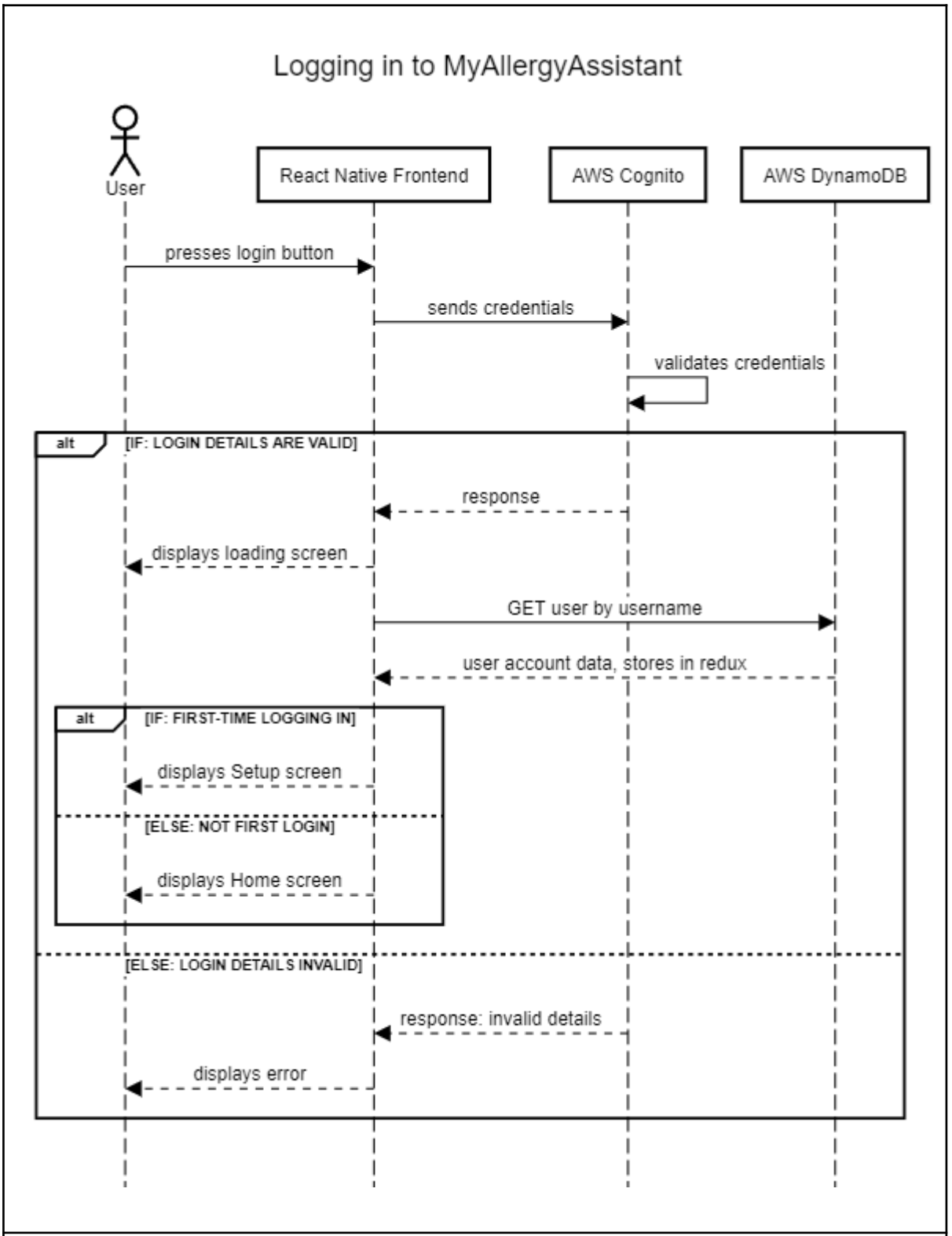

Figure 5. Login Sequence Diagram

In figure 5 above, you can see how the different components of our system interact with each other when a user logs into their account in our app. Firstly, the user presses the login button (either after entering their email and password or by signing in with Google).

The frontend then sends these login credentials to AWS Cognito to authenticate the user. If the credentials are invalid, the user is shown an error message. If the credentials are valid, the user is logged into their account, and shown a loading screen, while the frontend sends a GET request to the User table in AWS Dynamo, to get all the user's data (allergens etc.). Once Dynamo responds with the user's data, the user is shown either the Home or Setup screen and the data is stored locally using redux (persistent) which allows the user to automatically be logged back in when they close and reopen the app.

5.3. Sequence Diagram of Scanning a Barcode & Reporting a Product

On the next page, Figure 6 is a sequence diagram which shows how the different components of our system interact with each other when a user scans a product barcode. First the user scans a product's barcode by pointing the camera at the barcode while in either 'Scan Barcode' or 'Scan Both' mode. Once the barcode is detected by (barcode-scanner), the frontend makes a GET request to OpenFoodFacts API. If the product doesn't exist in the product database, an error message is displayed to the user. Otherwise, if the product does exist in OpenFoodFacts database, then the frontend gathers the necessary data from the response, and displays the product page to the user. The frontend checks the Report table in DynamoDB, to see if the product has been reported, so it can display that information to the user, and the product is added to the user's scan history in DynamoDB's User table. Then the frontend translates the response data from OpenFoodFacts if required, and applies the allergen identification algorithm on the products ingredients, to detect any user allergens. Once the algorithm has produced a result, the verdict is displayed to the user (Safe to eat, May not be safe to eat, or Not safe to eat) along with a table showing where certain allergens were found in the output text, if any.

The diagram below also shows the sequence of events that happen when a user enables notifications for a product. Note: Product notifications are enabled by default, but users have the option to toggle them on or off for each product they scan. Firstly the user presses the notifications switch, which triggers the frontend to send a request to both the User table and Notifications table in DynamoDB to update the notifications state for that product on both the product page and scan history page. When Dynamo responds that it was successfully added to the tables, then the notifications switch is toggled to display the change to the user.

As shown in the diagram, when a user submits a report for a product, the report gets added to the Report table in DynamoDB, and when Dynamo responds that it was successfully added, the report is displayed as submitted for the user, and they have the option to delete the report. Within the ReportsLambda, when a report is added via PUT or POST to the Reports table in Dynamo, a GET request is made to the Notifications table in order to get the list of users that need to be alerted of this report. Then using AWS SNS, each of the user endpoints is sent a notification that the product was reported along with the suspected allergens. Each user will then be shown the alert in the Alert tab the next time they log in. The code for this can be seen in section 6.3 and 6.4.

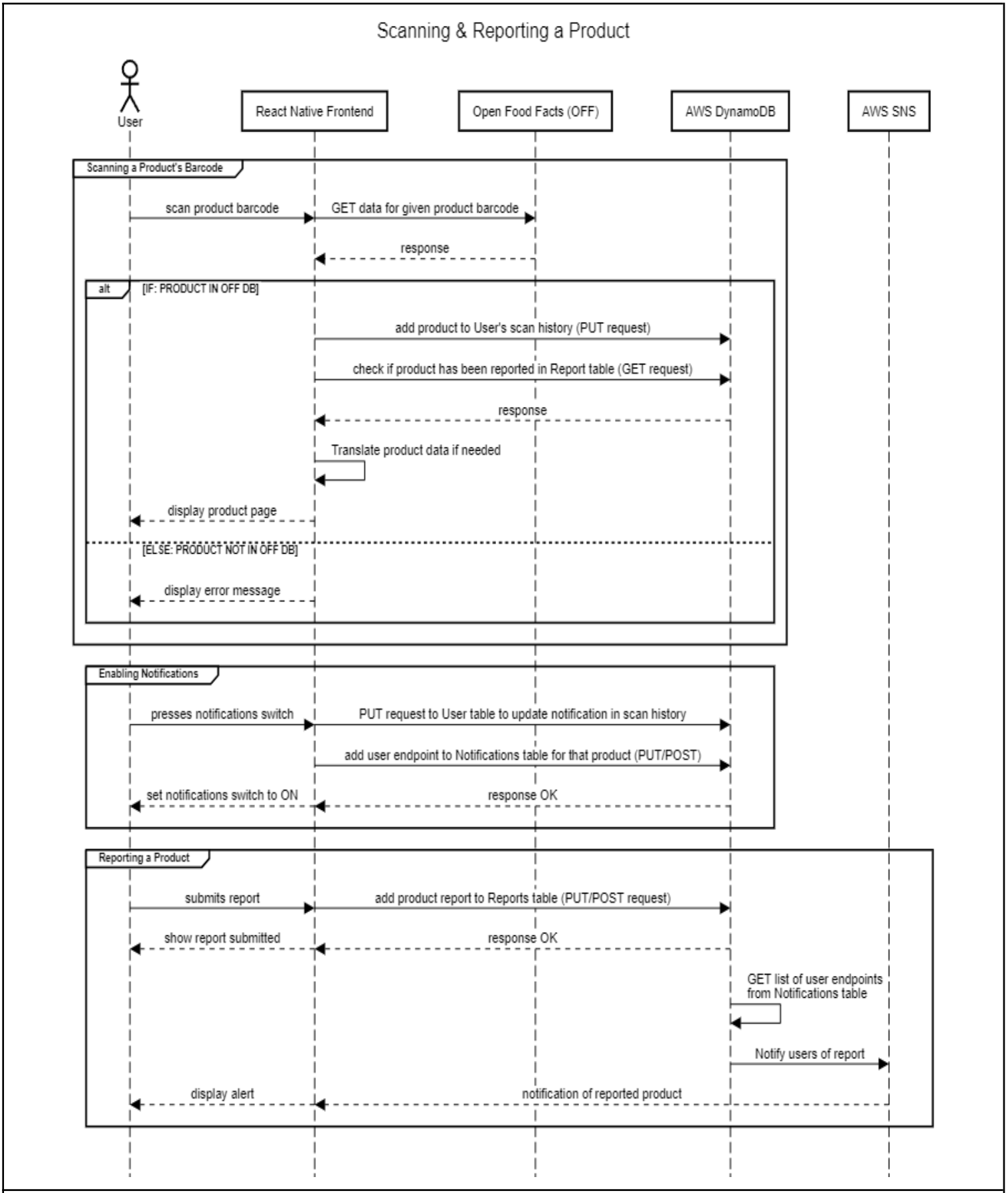

Figure 6. Scanning a Barcode and Reporting Product - Sequence Diagram

5.4. DynamoDB Tables and Lambda Functions

As shown in figure 7 below, we have three tables in our Dynamo database: **User, Report,** and **Notifications**.

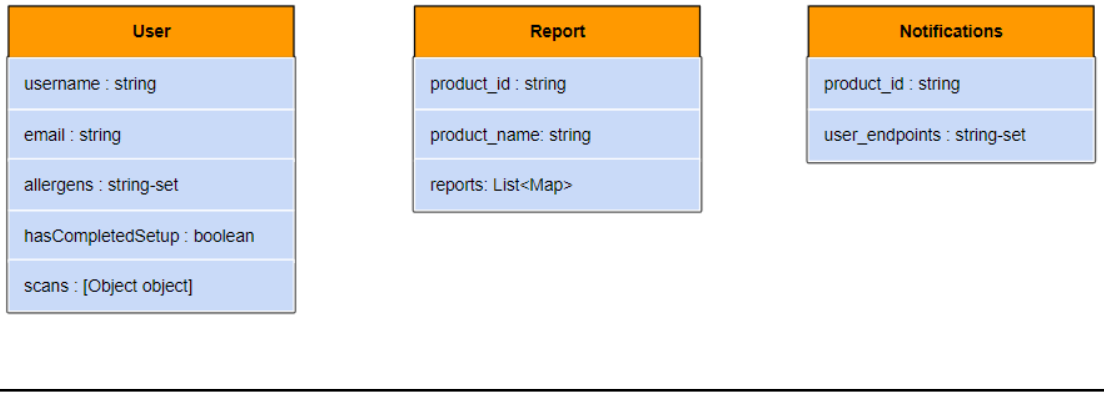

Figure 7. Our DynamoDB tables and their attributes

Each table's CRUD operations are handled by its own Lambda function**.** These are **usersLambda**, **reportsLambda**, and **notificationsLambda** respectively. Each lambda function is triggered by making a call from the frontend to our AWS API gateway, myAPI. We make calls to this API in the file '**src/frontend/api.ts**'. As an example, below is the code snippet used to make a call to the /users myAPI endpoint, which triggers the userLambda, which then deletes the user from the User table in Dynamo.

```
export async function deleteUser({username, email} : User) {
  API.del('myAPI', '/users', {
    body: {
      Key: {username: username, email: email},
    },
    headers: {
      'Content-Type': 'application/json',
      Authorization: `${(await Auth.currentSession())
        .getIdToken()
        .getJwtToken()}`,
    },
  })
    .then(res => {
      return res;
    })
    .catch(err => {
      return err;
    });
}
```

Below you can see examples of items from each Dynamo table and their attributes.

Example of an item in User table:

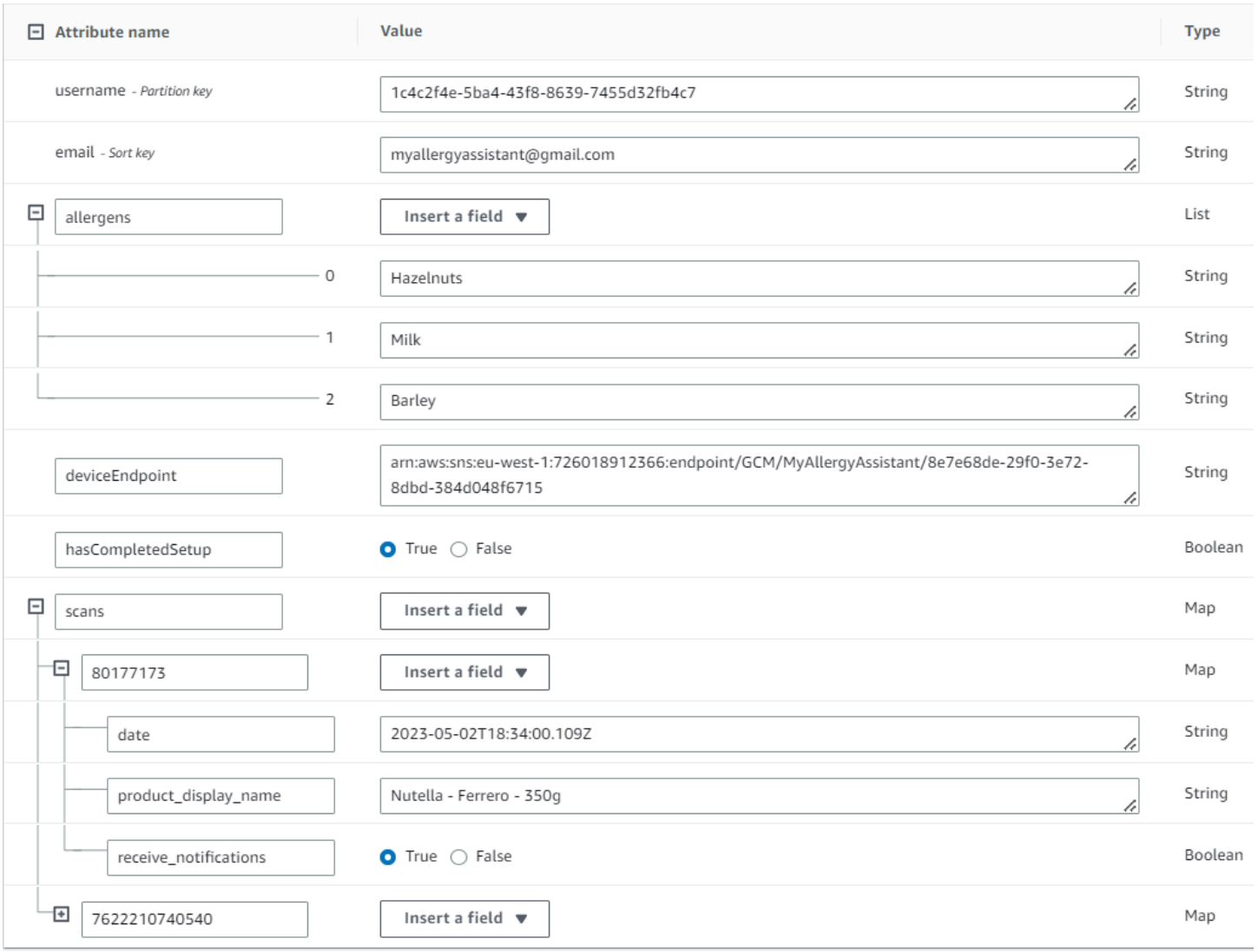

Figure 8. Example of an item in the User table in DynamoDB

Example of an item in the Reports table:

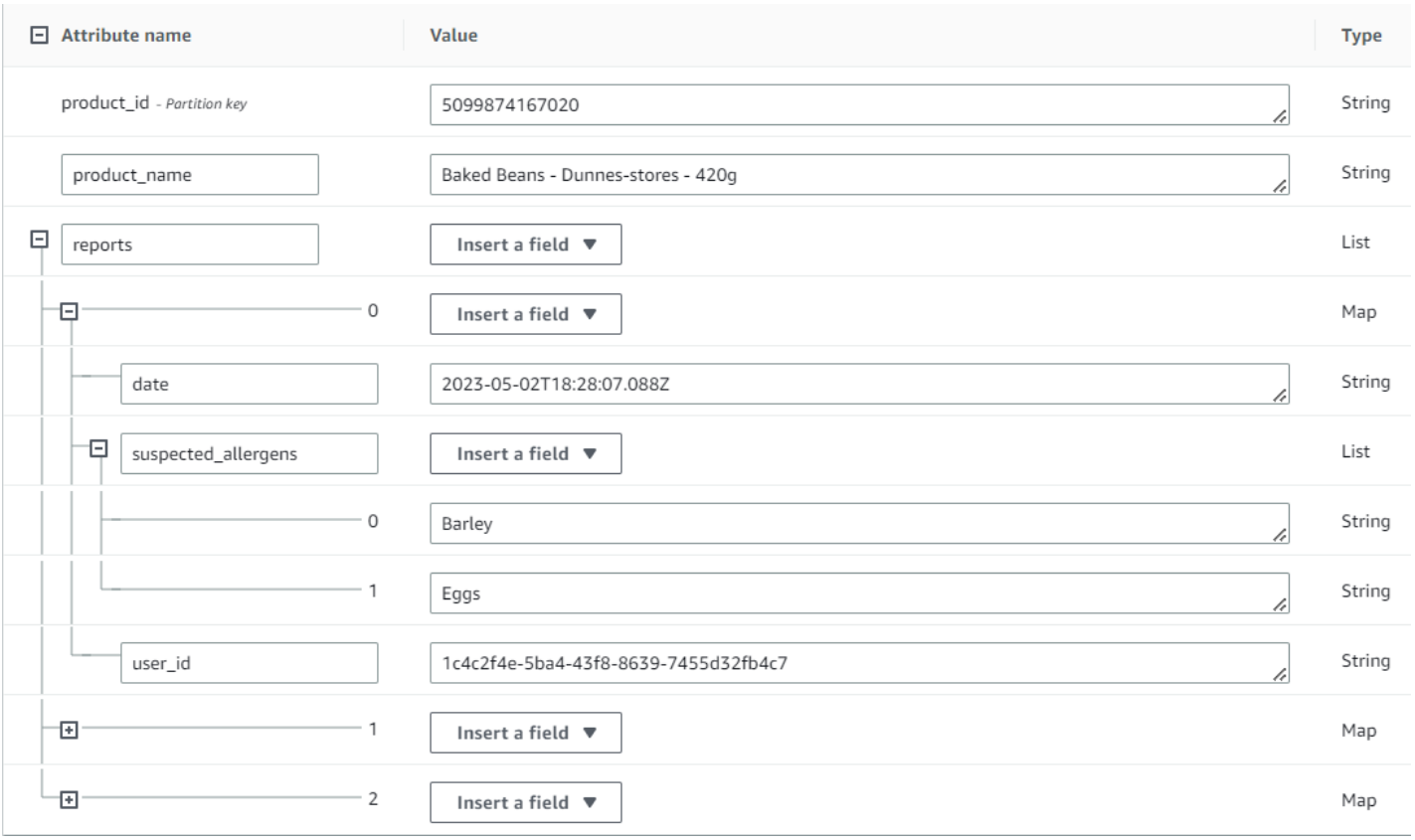

Figure 9. Example of an item in the Reports table in DynamoDB

Example of an item in the Notifications table:

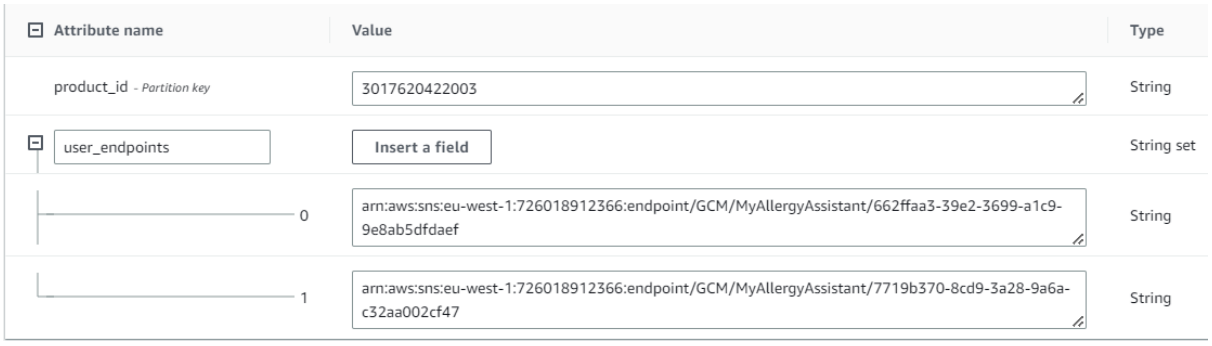

Figure 10. Example of an item in the Notifications table in DynamoDB

As seen in Figure 8, the **scans** attribute in the User table is an object of the form:

```
{
     product_barcode: {
         date: date,
         product_display_name: string
         receive_notifications: bool,
     },
     ...
}
```

Similarly, as seen in Figure 9, the **reports** attribute in the Reports table is a list of objects:

```
[{

        date: date,
        suspected_allergens: string-set
        user_id: string,

 },
 ...
]
```

There are two other lambda functions which are responsible for preprocessing an image given to OCR, and registering a Firebase FCM token for use by AWS SNS for notifications. These are **ocr-preprocessing,** and **registerDeviceToken** respectively.

ocr-preprocessing is currently the only lambda which is built with a Docker image package type, and other lambda functions have been built conventionally using zip files. Our preprocessing OCR step makes use of OpenCV, fast-deskew, and scipy python packages, manipulating a base64 image using various preprocessing techniques such as grayscaling, binarisation, denoising, deskewing, and more to improve OCR accuracy. Containerized Lambda functions are built with the help of our shell-script 'push_dockerised_function.sh'.

All of our lambda code and shell-script can be viewed from the **'src/backend/lambda_functions'** directory.

6. Sample Code

6.1. Allergen Identification Algorithm

The allergen identification algorithm is a fuzzy algorithm which is able to find allergens in noisy output. The key methods of the algorithm is to create links between allergens and tokenize output, use exact matching for output tokens clear of noise, and fuzzy matching for output tokens containing noise. An allergen text could be recognised by OCR differently,

such as ‘mi)k’ as opposed to ‘milk’. In this scenario ‘milk’ will be detected by our algorithm and users who have a milk allergy will be warned, even though the OCR picked up ‘mi)k’.

We compiled our allergens dataset (which can be found in ‘**src/frontend/allergens.json**’), by using the allergens list from the Food Safety Authority of Ireland’s website. Each allergen in our dataset has a list of variations. For example, ‘milk’ will be detected if any of the words ‘milk’, ’dairy’ or ‘butter’ are found in the text. The linking of allergens (including those containing variations in how they’re written) help us do this.

For the token to pass as an unlikely but possible allergen it must reach a minimum similarity percentage threshold of 0.4. It is deemed a likely allergen if its similarity percentage is above 0.67. These values have been fine-tuned through automated and ad-hoc testing.

For more information, see the **ocr-postprocessing** test suite and the code to better understand the capability of the algorithm in **‘frontend/__tests__’ , and ‘frontend/ocr-postprocessing.ts’** files respectively.

```
// function returns the allergenKey(s) that match to a given word
// e.g. if word=="wheat", allergenKeys will = ["wheat", "gluten"]
function getAllergenKeyswithValue(word:String,
allergensData:Array<Object>) {
 let allergenKeys = [];
 // for each allergen object in array
 allergensData.forEach((allergen) => {
   // get the list of values (possible aliases of that allergen)
   let valueArray = Object.values(allergen);
   // if that list contains the word, then add to result array
   if (valueArray[0].includes(word)) {
     allergenKeys.push(Object.keys(allergen)[0].toLowerCase());
   }
 });
 return allergenKeys;
}
...
function addProbableAllergenIfRated(ingredient, match) {
 const allergenSimilarityMaxThreshold = 0.67; // fine-tuned through
testing for normal cases
 const allergenSimilarityMinThreshold = 0.4; // for severely misspelt
words

 // if word is a close match, add allergen to results
 if (match.rating > allergenSimilarityMaxThreshold) { // for example,
'penuts' instead of 'peanuts' would pass here
   addAllergensToResultData(probableMatchedAllergens,
getAllergenKeyswithValue(match.target, possibleAllergens), match.target,
ingredient, true);
 }
```

```
 // for severely misspelt words, they must pass a rating of 0.4
 else if (match.rating >= allergenSimilarityMinThreshold) {
   // in this case, it should be added to may contains as its not
certain enough to be correct
   addAllergensToResultData(mayContain,
getAllergenKeyswithValue(match.target, possibleAllergens), match.target,
ingredient, false);
 }
}

...
function getAllergensFromText(ingredients, user) {
 ingredients = ingredients?.toLowerCase().split(",");

 if (ingredients) {
   let userAllergensSet = new Set([...user.allergens.map((x: string) =>
x.toLowerCase())])

   // go through OCR ingredients, find allergens
   probableAllergens(ingredients);

   // compare allergens list processed from ocr output with user list
   let userAllergensFound = intersect(userAllergensSet,
probableMatchedAllergens);
   let mayContainUserAllergens = intersect(userAllergensSet,
mayContain);

   // compile results
   let result = getResult(userAllergensFound, mayContainUserAllergens,
probableMatchedAllergens, mayContain, Object.fromEntries([...listedAs]),
Object.fromEntries([...likelyAllergensListedAs]));

   ...

   return result;
 }
}
```

6.2. Sample of Allergen Identification Test Cases

Below is a code snippet taken from **'frontend/__tests__/ocr-postprocessing-test.ts'**, showing some examples of the test cases involved in testing our allergen identification algorithm.

```
it("should not get allergens from text if ingredients is undefined", ()
=> {
   expect(getAllergensFromText(undefined, user)).toEqual(undefined);
})

it("should identify barley, wheat, and rye in allergens found if gluten
in text", () => {
   const res = getAllergensFromText("gluten", user);
   expect(res.allergens).toEqual(["gluten", "wheat", "rye", "barley"])
})

it("should not list gluten-free as gluten as a likely allergen", () => {
   const res = getAllergensFromText("gluten-free", user);
   expect(res.allergens).toEqual([]);
   expect(res.mayContain).toContain("gluten");
})

it("should list peanuts as an allergen if spelt as 'penuts'", () => {
   const res = getAllergensFromText("penuts", user);
   expect(res.allergens).toContain("peanuts");
})
```

6.3. Sending Alerts to Users using AWS SNS

Below is the function used to send notifications to user endpoints, which can be found in **reportsLambda** in **src/backend/lambda_functions/reportsLambda/index.js**. This function is invoked whenever a report is added to the Report table in DynamoDB, in order to notify all users who have alerts enabled for that product that it has been reported. Firstly, a GET request is made to Dynamo's Notifications table in order to get the list of user endpoints for the given product. Then it iterates over each endpoint and publishes the custom notification about the product alert to each user.

```
async function notifyUsers(productID, productName, report) {
    let notificationTable = "Notifications";
    if (process.env.ENV && process.env.ENV !== "NONE") {
        notificationTable = notificationTable + '-' + process.env.ENV;
    }

    // call notifications GET
    let customQueryParams = {
        TableName: notificationTable,
        Key: {product_id: productID}
    }
```

```
    let getNotis = await dynamo.get(customQueryParams).promise();
    if (getNotis.Item?.user_endpoints) {
        for (let endpoint of getNotis.Item.user_endpoints.values) {
            // send SNS notification to each user_endpoint to notify
about new report.
            try {
                const GCMdata = {
                    "notification": {
                        "title": "MyAllergyAssistant",
                        "body": "A product you scanned was reported",
                        "data": {
                            "product_id": productID,
                            "product_name": productName,
                            "report": report
                        }
                    }
                }
                const data = {
                    "GCM": JSON.stringify(GCMdata)
                }
                const messageJson = JSON.stringify(data);
                const message = {
                    MessageStructure: "json",
                    Message: messageJson,
                    TargetArn: endpoint
                };
                // publish message
                await sns.publish(message).promise();
            } catch(err) {
                console.log(err);
            }
        }
    }
}
```

6.4. Receiving alerts in the frontend and displaying them to the user

From the frontend, there is both a foreground and a background listener in '**src/frontend/components/AuthenticatedApp.tsx**' that is always listening for incoming notifications, as can be seen below.

```
Notifications.events().registerNotificationReceivedForeground((notificat
ion: Notification, completion) => {
        dispatch(addNotification({username: username, notificationData:
```

```
notification.payload["data"], date: new Date()}));
        completion({alert: true, sound: true, badge: true});
      })


Notifications.events().registerNotificationReceivedBackground((notificat
ion: Notification, completion: (response: NotificationCompletion) =>
void) => {
        dispatch(addNotification({username: username, notificationData:
notification.payload["data"], date: new Date()}));
        completion({alert: true, sound: true, badge: true});
      })
```

When a notification is received, addNotification() is invoked in order to add the notification to the user's app storage (redux), so that it will remain even after logout or closing the app. The below code snippet is also from src/frontend/components/AuthenticatedApp.tsx, and is used to display the number of unopened alerts on the Alerts tab badge.

```
const notificationsBadge = notifications?.filter((alert) =>
(!alert?.isOpened)).length > 0 ? notifications.filter((alert) =>
(!alert?.isOpened)).length : undefined;
```

```
<Tab.Screen
   name="Alerts"
   default={true}
   component={AlertScreen}
   options={{
      headerTitleAlign: "center",
      tabBarBadge: notificationsBadge,
      tabBarBadgeStyle: {borderRadius: 10},
      tabBarIcon: () =>
        <FontAwesome5 name={"bell"} solid size={styles.tabIcon.height}/>
   }}
/>
```

For example, in the below image (Figure 11.), you can see that there are 2 unopened alerts, indicated by the red dot on the right side. So the badge on the Alert tab shows the number 2.

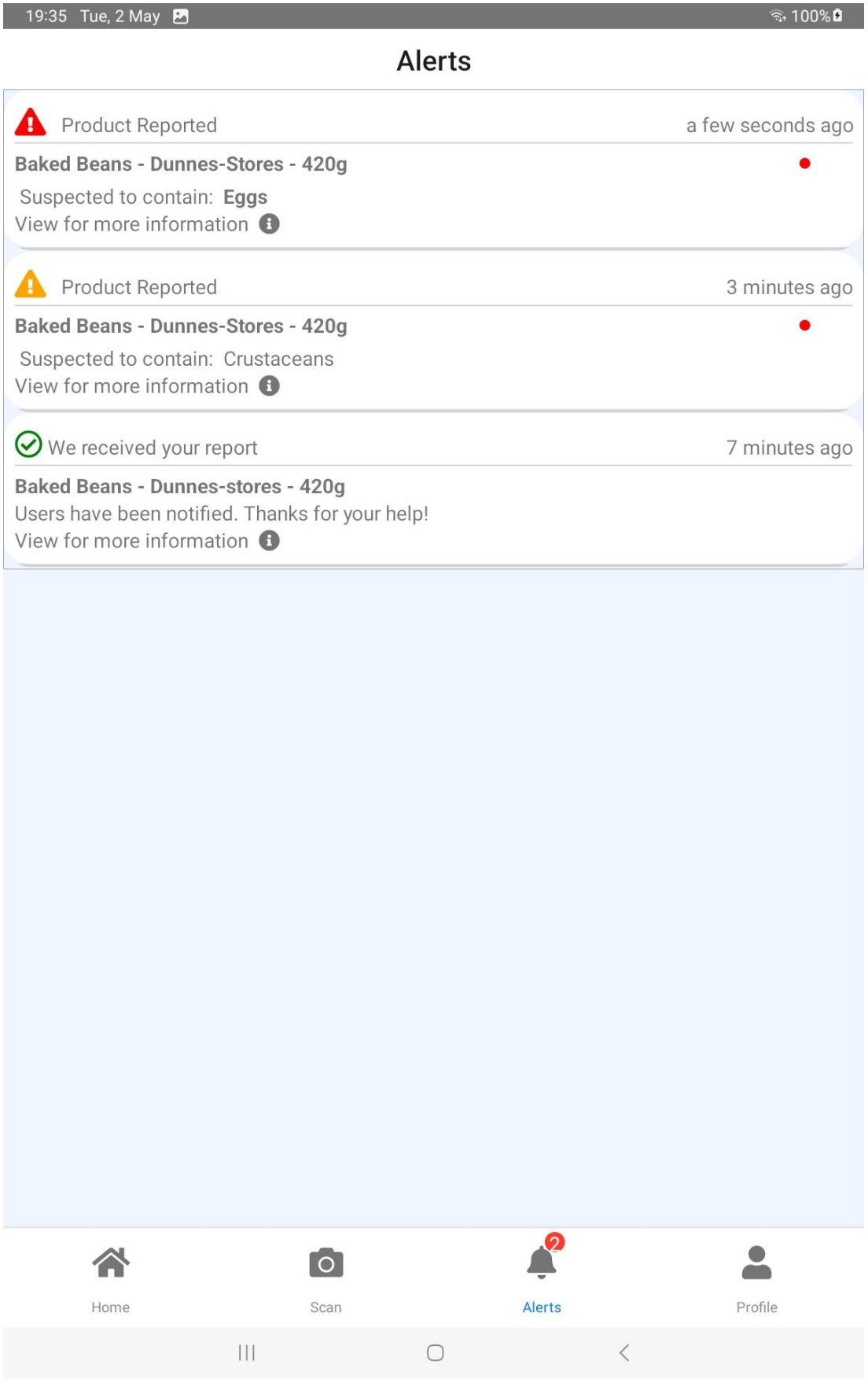

Figure 11. Example screenshot of Alerts screen of MyAllergyAssistant

Alerts are displayed to the user when they go to the Alert screen via our AlertReportRetrieved component which can be found in **src/frontend/components/AlertReportRetrieved.tsx**. The code is shown below.

```
function AlertReportRetrieved({alert}) {
    const userAllergens = useAppSelector(state =>
state.appData.accounts[state.user.username].allergens);

      // check if alert contains any of user's allergens
    const containsMatch = (listA, listB) => {
        if (listB) {
            let matchFound = false;
            listB.forEach((item) => {
                if (new Set(listA).has(item)) {
                    matchFound = true;
                }
            })
            return matchFound;
        }
    }
    return (
        <>
            <View style={styles.container}>
                <Text style={{marginTop: 5}}>
                    <FontAwesome5 style={styles.alertIcon}
                        color={containsMatch(userAllergens,
alert.item.suspectedAllergens) ? "red" : "orange"}
                        name="exclamation-triangle"
                        size={20}
                    />
                    {"   "}
                    Product Reported
                </Text>

                <Text
style={styles.date}>{moment(alert.item.date).fromNow()}</Text>
            </View>

            <Text numberOfLines={1}
style={styles.productName}>{alert?.item.productName}</Text>
            <Text style={styles.suspectedAllergens}> Suspected to
contain:{"  "}
                {alert?.item.suspectedAllergens.map((allergen, index) =>
{
                    if (new Set(userAllergens).has(allergen)){
                        return <Text key={index.toString()}
```

```
style={{fontWeight: "bold"}}>{allergen}  </Text>
                    }
                    else {
                        return <Text key={index.toString()}>{allergen}
</Text>
                    }
                })}
            </Text>

            <View style={styles.view}>
                <Text>View for more information{"  "}</Text>
                <FontAwesome5 style={{marginRight: 5}}
name="info-circle" size={20}/>
            </View>
        </>
    )
}
```

As you can see, we check each alert to see if it contains any of the user's allergens. If a report does contain user allergens, it is highlighted in the colour red with the matching allergen displayed in bold. Otherwise, the report will be orange, as seen in the above figure 11.

7. Testing Strategy

7.1. Automated Test Results

Frontend

Below is the output when running our frontend test suites, invoked by running 'jest' from the src/frontend directory. As you can see, all 37 test cases pass, across 5 different test suites. You can view the individual test suites from the 'src/frontend/__tests__/' directory.

```
> frontend@0.0.1 test
> jest

 PASS  __tests__/translation-test.ts
 PASS  __tests__/ocr-test.ts
 PASS  __tests__/barcode-test.ts
 PASS  __tests__/ocr-postprocessing-test.ts
 PASS  __tests__/ui/App-test.tsx (46.611 s)

Test Suites: 5 passed, 5 total
Tests:       37 passed, 37 total
Snapshots:   0 total
Time:        47.644 s
Ran all test suites.
```

Backend

Below is the output when running our backend automated tests, invoked by running 'jest' from the src/backend directory. As you can see, all 42 test cases pass, across 4 different test suites. You can view the individual test suites from the 'src/backend/__tests__/' directory.

```
> test
> jest

PASS  ./registerDeviceToken-test.js (5.55 s)
PASS  ./report-test.js (6.228 s)
PASS  ./notifications-test.js (6.73 s)
PASS  ./user-test.js (7.091 s)

Test Suites: 4 passed, 4 total
Tests:       42 passed, 42 total
Snapshots:   0 total
Time:        7.747 s
Ran all test suites.
```

7.2. Continuous Integration and Deployment (CI/CD)

We used GitLab CI/CD pipelines to help check if a new commit introduced any erroneous changes into our application, by creating a file called ".gitlab-ci.yml".

We have four stages that test our backend and frontend. Two stages that set up the backend and frontend for testing, **amplify** and **mobile-build** respectively. Two other stages that run our tests for the backend and frontend, **backend-test** and **ui-test** respectively.

Our pipelines can be viewed from our repo, which began from December 29th 2022. Every time we made a commit to our repo, the pipeline would run, and we would get an email notifying us if the pipeline failed, so that we could fix it quickly.

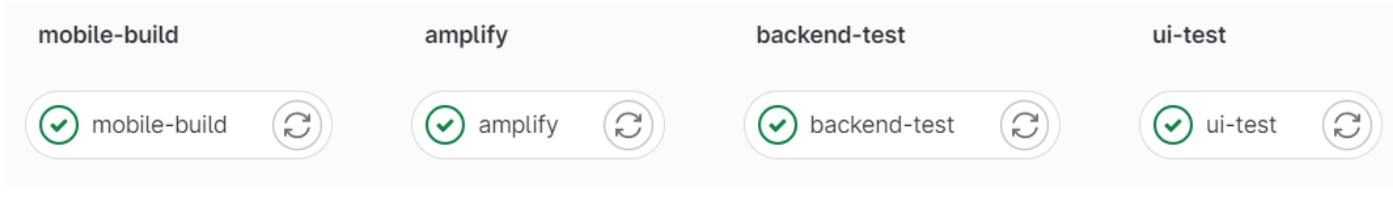

Figure 12. An example of our 4 pipeline stages passing on GitLab

7.3. Unit Testing

We wrote unit tests for both frontend and backend, these can be found in "frontend/__tests__", and "backend/__tests__". We focused on writing unit tests for the functionality we deemed essential. Some planned integration tests were replaced by unit

tests, because for example, Android's File System used by on-device OCR/Barcode scanning was blocking us.

An example of a unit test in our backend is where we mocked our notification service, which uses AWS SNS to create a unit test, to see if messages can be published correctly. This was done using the aws-sdk-mock package.

In our frontend, for example, we automatically create data-driven unit tests for OCR (@react-native-ml-kit/text-recognition) by mocking the on-device OCR functionality where the mock gets the text from our test data files, and compare the results to the expected results to determine whether the test passed or failed.

7.4. Integration Testing

We wrote integration tests for both backend and frontend, these can be found in “frontend/__tests__”, and “backend/__tests__”. We implemented many of the integration tests we deemed essential.

An integration test in our backend for example, we created a test which uses AWS Lambda, DynamoDB, and SNS, where we run a lambda function, which uses an instance of DynamoDB locally, and a mocked SNS client.

An integration test in our frontend for example, we test our allergen identification algorithm where a test uses OCR output and our postprocessing module (ocr-postprocessing.ts exports a function getAllergensFromText) with some single and data-driven tests.

7.5. User Testing

Once we had our ethical approval form approved, and our app ready for user testing, we set out to gather users from DCU and family/friends to conduct a usability evaluation of the app which began March 2023. You can view our plain language statement, tasks, and questionnaire in the appendices of the ‘ethics.pdf’ located in the ‘docs’ directory in our repo. We conducted testing with 11 users, exceeding our minimum number of users we expected to test of 8. The feedback we got was very positive and it allowed us to improve certain aspects of our app’s UI. The full user testing feedback can be found in **docs/user-testing-feedback.pdf** in our repo.

Below is a list of changes that we implemented to our app as a result of the user feedback we received:

- We added the safety indicator to barcode scanning as well as ingredients scanning. (Previously was just for ingredients scanning)
- We added a toast notification that says ‘Changes saved’ when a user updates their account allergens, so they know they have been successfully updated.
- We added a Help button to the scan screen to give users more information on how to scan products.
- We added text to indicate what scan mode you are currently in ‘Scan Barcode’, ‘Scan Ingredients’ or ‘Scan Both’.

- We added text to the modal when scanning ingredients, to recommend cropping your image.
- We added colours to the buttons on all modals. Green for ‘Yes’ buttons and red for ‘No’/’Cancel’ buttons. (Previously these buttons were all white)
- We inverted the order of the alerts on the Alert screen, so that the most recent ones would appear at the top. (Previously the oldest was at the top.)
- We added text to indicate when product data was being translated, and the ability to press the text to view the original text in the source language.
- We adjusted the text at the bottom of the search results screen to make it clearer how many pages of results there are.
- We added a table to show where our algorithm detected the user’s allergens in the product data. Users can now see the word(s) in the product data that caused the product to be not safe to eat.

8. Problems and Resolutions

8.1. Capturing an optimal image of product ingredients for OCR

Initially, we would process each frame in real-time and apply OCR to each frame, and whenever the word ‘ingredient’ was found, it would then automatically take a photo and scan, similar to how barcodes currently are scanned.

This caused issues where the image of the ingredients text was of poor quality, because the photo could be taken automatically before the user was ready. So the user had to keep creating a new result until they were satisfied with the quality of the image.

We solved this problem by giving the user more control over the image taken and when to scan. This was done by identifying if the word ‘ingredient’ was detected in the current frame or not, as before. However when detected, instead of automatically taking the picture, we highlight the scanner button to take a picture on the scan UI as an indication to scan and prevent it from scanning automatically. Below is the code snippet used to highlight the scanner button in a red colour when the keyword ‘ingredient’ is detected, giving the user more control over when to scan the product.

```
// presentation in a child component
return (
      ...
<ScannerButton>
  <TouchableOpacity
    activeOpacity={100}
    onPress={onTakeScan}
  >
    <FontAwesome5Icon
        color={isDetected ? "red" : "rgba(255,255,255,0.5)"}
        name={isDetected ? "dot-circle" : "circle"}
```

```
        size={70}
    />
  </TouchableOpacity>
</ScannerButton>
      ...
)

// logic from a parent component
useEffect(() => {
 const ocrCondition = scanMode === ScanMode.Text || scanMode ===
ScanMode.Detect;

 if (ocrCondition && ocrResult?.result?.blocks?.length > 0 &&
ocrResult.result?.text != "" &&
ocrResult.result?.text.toLowerCase().includes("ingredient")) {
     if (!ingredientsFound) {
       takePhotoHandler().then((photo) => {
         setPhoto("file://" + photo.path)
       }).then(() => {
         setIsDetected(true)
         setIngredientsFound(true);
       });
     }
     else {
       setIngredientsFound(false);
     }
 }
 else {
   setIsDetected(false);
 }
}, [ocrResult])

...
```

When the user begins to scan, they can view and crop / rotate the image to ensure they are happy with the image to be scanned. See our user manual for reference on how to do this.

8.2. Improving and identifying allergens in noisy data

Initially, the way we identified allergens was by taking the OCR output, splitting it into tokens, and matching them exactly to any allergens in our dataset. This wasn't ideal as the OCR output could contain noisy data.

We solved this by inputting the output through the allergen identification algorithm, which is a fuzzy algorithm that can capture allergens from noisy text where the algorithm finds out how likely a token is an allergen, and if there is a variation in the name of an allergen, it can be

linked to the main name of an allergen. See code snippets of the algorithm above in section 6.1.

We realised through our unit testing that this would still produce a lot of inaccurate results, so we wanted to reduce the noisiness of the output. For that, we decided to do preprocessing on the OCR image prior to using the allergen identification algorithm. As mentioned earlier in section 5.4, this was done through a containerized function, **ocr-preprocessing**, which manipulates an image to improve it for OCR. We used various preprocessing techniques such as deskewing (rotation), grayscaling, binarization, and more.

See our preprocessing code in our repo in the file

'backend/lambda_functions/ocr-preprocessing/lambda_handler.py'

For barcode scans, which process an OFF product response, sometimes the product information would appear in a language that was not in English. This would greatly decrease the effectiveness of the allergen identification algorithm as it does not currently account for translation. We solved this by combining the information from OFF separated by spaces, translating the combined information with Google Translate API. Once translated into English, we could put it through our allergen identification algorithm.

For reference of this, see

'frontend/components/scan/BarcodeScanResult/IngredientsAllergensTracesText.tsx'

8.3. Testing AWS services

As we have never worked with AWS before this project and have limited experience with cloud computing, we found testing AWS services to be a challenge. We only had experience in setting up local test environments such as with a Django server running on localhost. Tests for Lambda functions could be written via the console, however we found this limiting and could potentially affect our database in production.

We solved this problem by setting up and running an instance of DynamoDB, and Lambda functions both stored locally. We also mocked other services such as SNS, and wrote system tests to see if Cognito interacts correctly with our app for example. This allowed us to test our backend Lambda functions without manipulating the actual database used in production.

8.4. Overcoming the Lambda deployment size limit

When deploying our **ocr-preprocessing** lambda function, we came across an issue where we could not deploy the function as the .zip deployment file created was larger than 50mb.

With some research, we found that by containerizing your function, you can deploy a Docker image with a size of up to 10gb. We created an ECR repository to hold our Docker images and deployed the Lambda function using an image via the AWS console.

We automated the process of building and adding the images to our repo with the push_dockerised_function.sh script as we mentioned earlier in section 5.4.

8.5. Giving users a way to verify their result easily

Initially, the allergen identification algorithm would simply find what allergens were found, including what were likely, and uncertain allergens in the output.

I think replace this sentence with

While conducting user testing for our project, some of the feedback included adding the ability to see where exactly an allergen was detected in the product data, so that the user could verify if it was accurate.

To solve this problem, we displayed a table on the scan result screen that lists the allergens found as well as the string(s) in which each allergen was found. We achieved this by tracking what allergens in the text were written as in the algorithm and then feeding the result of the allergen identification algorithm into a component called **AllergenListedAsTable** which can be found in the file 'src/frontend/components/scan/AllergenListedAsTable.tsx'.

Furthermore, we ordered the allergens found by the user's allergens and with the user allergens being highlighted in bold at the top of the table, to help the user verify their result quickly.

8.6. Persistent storage - automatic sign-in

Another problem that we encountered was that the app would get stuck on a white screen when you close and re-open the app.

We realised that this was happening because the user data was being cleared from Redux when the app closed, leaving the user unauthenticated. We fixed this problem by using Redux Persist which is a Redux tool that saves the app's state to Async Storage on the Android device, allowing users' sessions to be saved and automatically logging them back in the next time they open the app, without having to re-enter their details.

8.7. Time restrictions

As there was a fixed deadline for this project, we had to prioritise the key functionality for the app. As mentioned in section 3, we used a Trello board to break the project down into subtasks and assign a level of priority to each subtask. If we had more time, we could have further improved the UI and implemented additional features, such as those mentioned in section 10. Future Work.

8.8. Problems installing packages

In some cases, errors would arise when installing a new package, so we would have to make small edits to some of the source code within the package. We did this using the npm package 'patch-package', which allowed us to save the edited package, so the package is automatically patched as part of the 'npm install' command. We patched 5 packages in total, which can be seen in '**src/frontend/patches/**' directory.

9. Reflection

Overall we are both very happy with the result of this 4th year project, as we have delivered on all functional requirements, as well as additional features like:

- The ability to view the products that you have previously scanned. (Scan History)
- The ability to search for a product
- Mobile Push Notifications
- Federated Login - the ability to register and login with Google instead of having to enter an email and password.
- Real-time detection of barcodes and ‘ingredient’ keyword
- The ability for the user to crop or rotate an image for OCR
- Image preprocessing, such as denoising and grayscaling, for improved OCR accuracy
- Persistent storage - saving user data so they can be automatically logged in next time they open the app.

We are proud of our work and think MyAllergyAssistant can be very useful to users with food allergies. If we had more time, we would have liked to conduct more user testing to get even more feedback to continually improve the app.

10. Future Work

Future work for this project could include:

- Launching the app on the Google Play Store.
- Support for more platforms e.g. iOS
- Potential features to add:
    - Give users the ability to enter their own custom allergens, instead of having to choose from the preset list.
    - Add translation feature to ‘Ingredients Scanning’. (Currently OCR only works for English text, but Barcode Scanning does currently translate product data)
    - Allow users to review their ingredients scan history. This would involve storing the images that users take in our backend, which may give rise to cost and security implications. (Currently we do not store any images taken by users)
    - Allow users to add new products / update product information through our app, rather than redirecting them to OpenFoodFacts.

11. Installation Guide

You can download the release APK of our app, by visiting the following the Google Drive link on your Android device:

https://drive.google.com/file/d/1IsLh5KXE5Iw9Zrdraig_EZ_1EL4J3E2k/view?usp=sharing

Alternatively, if you would like to run this app locally, you must follow these steps:

1. Clone our git repository

```
git clone
https://gitlab.computing.dcu.ie/eskandg2/2023-ca400-eskandg2-prizemd2.gi
t
```

2. Navigate to the ‘src/frontend’ folder

```
cd 2023-ca400-eskandg2-prizemd2/src/frontend
```

3. Install the required dependencies.

```
npm install
```

4. Connect your android device via USB
5. Run the local metro server

```
npm start
```

6. Install the app onto your connected Android device

```
npm run android
```

12. References

12.1. Logo Images

Below are links to where we obtained the images of each logo used in the architecture diagram:

- AWS services logos: https://aws.amazon.com/
- Google cloud logo: https://cloud.google.com/
- OpenFoodFacts logo: https://world.openfoodfacts.org/
- React Native logo: https://reactnative.dev/

12.2. 3rd Party Components Used

Below is a list of the 3rd party components used in MyAllergyAssistant, with links to their documentation:

- OpenFoodFacts API - https://openfoodfacts.github.io/api-documentation/
- Google Cloud - https://cloud.google.com/
- Firebase Cloud Messaging - https://firebase.google.com/docs/cloud-messaging

- AWS Cognito - https://docs.aws.amazon.com/cognito/index.html
- AWS DynamoDB - https://docs.aws.amazon.com/dynamodb
- AWS SNS - https://docs.aws.amazon.com/sns/
- AWS Lambda - https://docs.aws.amazon.com/lambda/
- AWS ECR - https://docs.aws.amazon.com/ecr/
- AWS Amplify - https://docs.aws.amazon.com/amplify/
- AWS API Gateway - https://docs.aws.amazon.com/apigateway

12.3. Packages Used

Below is a list of some of the important packages we used during the development of MyAllergyAssistant with links to their documentation. The full list of packages and versions we used in our code can be found in 'src/frontend/package.json'.

"react",
"react-native",
"typescript",
"@aws-amplify/ui-react-native" - provides a template to manage auth UI,
"@react-native-async-storage/async-storage" - persistent storage for mobile apps,
"@react-native-ml-kit/barcode-scanning" - used to detect barcodes in images selected from gallery,
"@react-native-ml-kit/text-recognition" - used for OCR of images from gallery,
"Aws-amplify" - makes Auth and API use simpler,
"Google-translate-api-x" - translate product data into English if not already,
"Moment" - to easily display how long ago a product was scanned/reported,
"Patch-package" - to fix the issue from section X.X,

"React-native-image-crop-picker" - to allow the user to crop an image,
"React-native-image-picker" - to allow the user to select an image from their camera roll,
"React-native-inappbrowser-reborn" - to keep the user in the app when signing in through google,
"React-native-notifications" - to allow the app to receive notifications,
"React-native-permissions" - to accept camera and notification permissions,
"React-native-screens" - for supporting mobile app navigation,

"React-native-vector-icons" - to display icons in our app. e.g. for each tab,
"React-native-vision-camera" - camera for Scan screen, can do real-time OCR and Barcode scanning,
"React-redux" - to store app data in device cache,
"Redux-persist" - to allow automatic login by remember user data,
"String-similarity" - used in allergen identification algorithm to compare product data/ingredients to user allergens, ,
"Vision-camera-code-scanner" - used for real-time barcode scanning,
"Vision-camera-ocr" - used for real-time OCR
"@aws-sdk/client-sns" - used in Lambda to register endpoints / publish messages,
"Aws-sdk-mock" - mocks AWS services, used for backend test suite,
"Jest", - used to run unit tests on our code
"lambda-local" - runs Lambda functions locally, used for backend test suite,

"Jest-dynalite" - runs DynamoDB locally, used for backend test suite,

Python packages used for image preprocessing as mentioned in section 8.2:

Opencv-python-headless, fast-deskew and scipy.