

CS 4850 - Fall 2024

INDY-7 — Red Nutrition App

**Michael Ehme, Chris Sarzen, Mai Mai, Silas
Hammond**

Sharon Perry

4/14/24

<https://ksuseniorprojectnutritionapp.github.io/Website/>

<https://github.com/KSUSeniorProjectNutritionApp/backend>

No. of Components: 3 (Frontend, Backend, DB)

**No. Lines of Code: 615 source code lines, 16 lines of .yaml,
1,628 TypeScript**

No. Of Database Entries: 350,000

Table of Contents

1 -

Introduction.....	3
2 - Design Approach.....	3
4 – Non-functional Requirements.....	7
5 – External Interface Requirements.....	7
6 – System Architecture and Design.....	<u>8</u>
7 – Application Testing.....	9
8 – Version Control.....	10
9 - Development.....	11
10 - Conclusion.....	12

1.0 Introduction

1.1 Overview

With so many people having issues that are related to their diet routine, weight problems being the most common, our team decided to develop an app that can not only help manage food restrictions but save people time and money. This application will help alleviate health management issues by letting users make informed decisions for their daily diet routine by providing nutritional information. Everyone should be able to know quickly and easily what is in the food they eat, without having to scan through rows and columns of ingredients they can't even pronounce.

1.2 Project Goals

This project's objective was to develop a fully functioning and polished mobile app for keeping track of caloric intake and monitoring other aspects of one's health. To accomplish this, we used React Native to develop a front end which allows users to select their goals and get active assistance in moderating what they eat. This is done through the scanning of barcodes of any food purchased. These barcodes are used to query the Food Data Central API, which we have downloaded to an S3 bucket for retrieval, to provide the user with as much information as needed. This include possible allergies, calorie totals, protein totals, and more relevant information.

The mobile app will include standard user registration and authentication flows. At a minimum, username/password authentication will be implemented, but time permitting we will also be implementing 2FA and OAuth2. Care will be taken to handle user data securely and safely.

1.3 Definitions and Acronyms

UI - User Interface

ORM - Object-Relational Mapping - lets you manipulate and query a data using an object-oriented paradigm

AWS - Amazon Web Services API - Application Programming Interface

OAuth2 - Authentication framework used to authorize users when logging into the application

JPA –

2.0 Design Approach

2.1 High Level Design Overview

The front end was built using React Native and its libraries for camera usage. Alongside this, we used Typescript, VSCode, XCode, and Android Studio to aid in development. A straightforward interface was built to allow reading barcodes, item searching, onboarding, login, and user preferences. The backend was built using Java, and is hosted using AWS. Each user instance of the application will connect to an AWS Edge Location and make contact with an Elastic Container Service, which is able to query the database

and S3 Bucket to return necessary information. Any searches made were done using Hibernate search libraries. All information has been downloaded using the FoodData Central API. Security design at the time of writing is incomplete, but is set to include AWS VPC and security groups, an elastic load balancer to interface with the outside world, and an AWS secrets manager for sensitive data.

2.2 Goals and Guidelines

The primary goals for the design of the Nutrition App included a user centric design, quick access to large amounts of data, and cross platform compatibility. We prioritized a design that is both intuitive and engaging for users. We also wanted to adhere to FEA mobile development guidelines for data security to give our users a safe and private experience. Cross platform compatibility with Android and iOS was made straightforward due to the use of react-native, which allows us to recycle the same code for both platforms.

2.3 Development Methods

The key technologies for this project's development are React-Native, Java Spring Boot, and Hibernate.

2.4 User Characteristics

Primary users would consist of health-conscious people, fitness enthusiasts, people with dietary restrictions. We believe that most people no matter their current goals would benefit from using this app, as ignorance of what one is eating is a leading factor in diminished health.

2.5 System Frontend

The application's UI was built using React Native, a framework for developing cross-platform mobile applications in JavaScript or TypeScript. The reason we chose this was so that we would have a single codebase to support both iOS and Android platforms. The frontend will allow users to scan barcodes, display the nutritional information of an item they scan, and provide interfaces for tracking daily nutritional intake.

2.5.1 User Interface Design

The approach taken for the UI was focused on accessibility and a simple style. The app includes 8 different screens: a welcome screen, a login screen, an account creation screen, a barcode scanning screen, a profile screen, a history screen, a settings screen, and a food detail screen.

2.5.1.2

Section 508 Compliance The app is designed with accessibility in mind, adhering to Section 508 standards to ensure it will be usable by individuals with disabilities. This includes the ability to interface with a screen reader, adding different color contrast settings, and easy access to interactive elements in the app.

2.6 System Backend

Server-side logic will be implemented in Java using the Spring Boot framework to create the microservices that will interface with the frontend, and Hibernate for ORM. The backend will expose

APIs for the frontend to interface with, enabling functionalities such as getting nutritional information from a products barcode, managing user profiles and logging nutritional intake.

2.6.1 Database Design

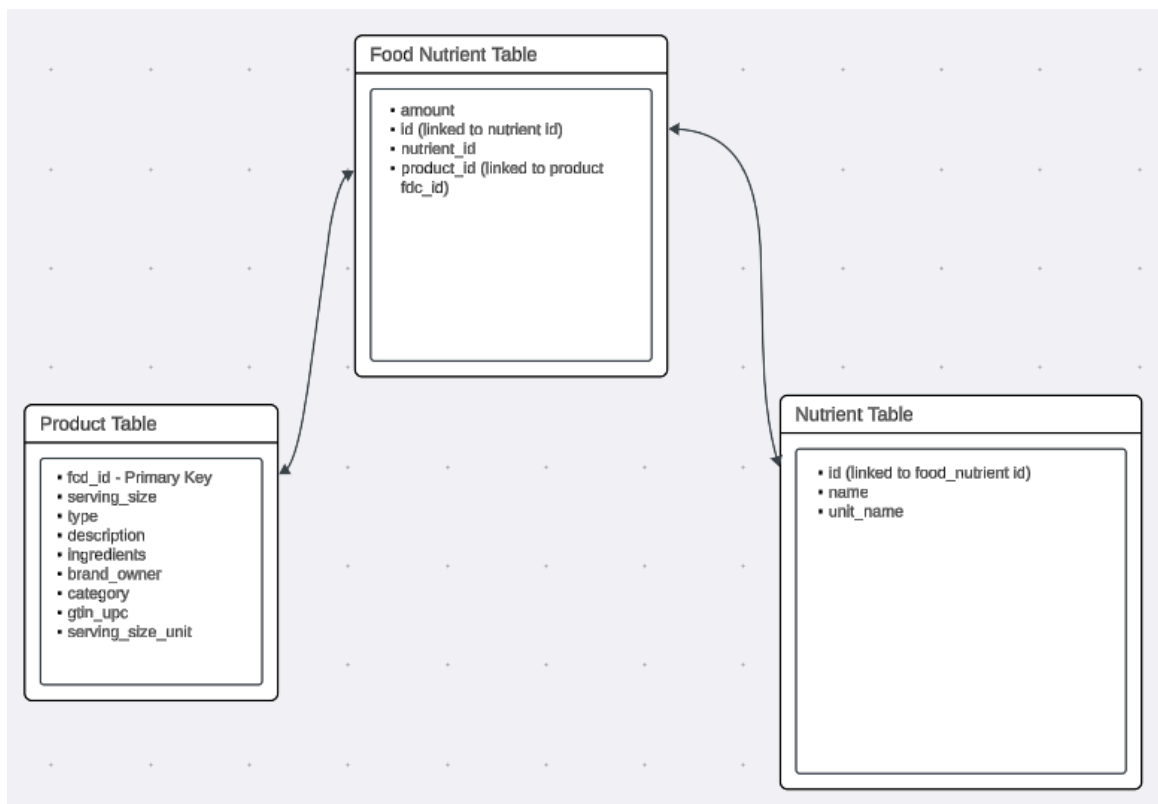
The application's database was populated using the Food Data Central API. This information was then divided into three database tables. The first table, labeled product

2.7 Database Design

2.7.1 Data Objects and Resultant Data Structures

Data objects will include product, nutrient, and food nutrient. These correspond to the structure of the database tables. Product details general information of a product, such as its brand and description. The product table also contains the barcode id, which acts as a foreign key for the food nutrient table. The food nutrient table allows us to list multiple ingredients that may be present in one product and creates a one-to-many relationship from the product table to the food nutrient table. Finally, the food nutrient table has unique ids for each ingredient, which allows us to access the nutrient table and get that ingredients name and unit type.

2.7.2 Database Structure Diagram



2.8 Data Storage and Retrieval

Data is converted by downloading the Food Data API database locally, and individually reading in each entry as an object using JPA/Hibernate. These objects are then placed into a local table using PostgreSQL, and finally hosted using AWS to allow any users to access the database. Additionally, Hibernate has built in features which allowed us to implement a search method that does not require exact matches.

2.9 User Machine-Readable Interface

2.9.1 Inputs

Users input food items by either scanning a barcode, or looking up an item by name.

2.9.2 Outputs

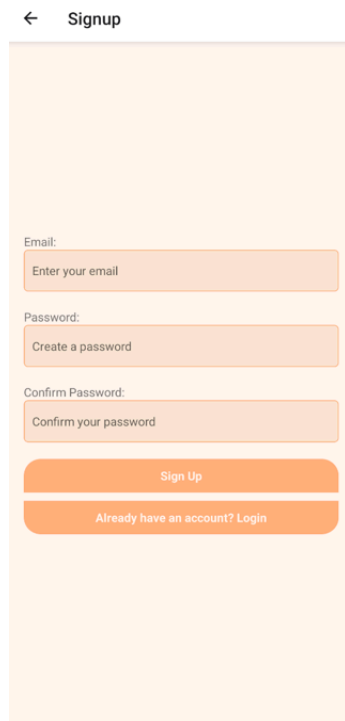
The app outputs comprehensive nutritional information.

3.0 Functional Requirements

3.1 User Login and Sign Up

Login button prompts for username and password, Sign up button prompts for the user's full name, a username, and a password with confirmation.

3.1.1 User Login and Sign Up Sample



3.2 User Authentication

User Authentication is performed with OAuth2.

3.3 Home Page

Home Page displays on login. Home page includes a navigation bar to other screens.

3.3.1 Home Page Sample



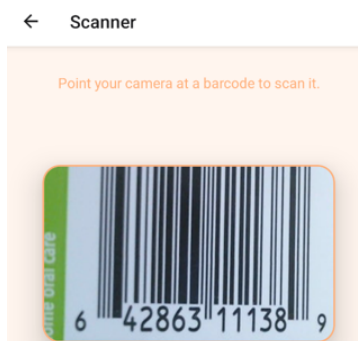
3.4 Navigate To User Profile

Profile button on navigation bar redirects to user profile screen.

3.5 Barcode Scanner

Camera button redirects to the scanner screen to provide a simple interface for the user. Scanner automatically detects barcodes once on the scanner screen.

3.5.1 Barcode Scanner Sample



3.6 Nutrition Results

Product information displays after the app scans a barcode.

3.6.1 Nutrition Results Sample

← Nutrition

Nutrition Facts	
Product Name	Placeholder Brand
Serving Size	0g
Calories	0 kcal
Total Fat	0g
Saturated Fat	0g
Trans Fat	0g
Cholesterol	0mg
Sodium	0mg
Total Carbohydrate	0g
Dietary Fiber	0g
Total Sugars	0g
Added Sugars	0g
Protein	0g

3.7 User and App Settings

Allow for notifications and app edits to be made by the user.

3.7 User and App Settings Sample

← Settings

Settings

Profile

Edit Profile
Change Password

Notifications

Push Notifications ☒

Email Notifications ☒

Display

Dark Mode ☐

Help & Support

FAQs
Contact Us

About

Terms of Service
Privacy Policy
Version: 1.0.0

4.0 Non-Functional Requirements

4.1 Security

The team chose OAuth 2.0 because the application uses the SpringBoot framework which has Spring Security built into it, and for its flexibility and scalability.

4.2 Usability

The application uses a standardized login/register screen. After the user has login/register the application will only display a button which allows the software to access the mobile device camera in order to scan item barcodes. The application displays a scrollable nutritional information table with nutrients on the column and their value on the rows.

5.0 External Interface Requirements

5.1 User Interface Requirements

The application offers an intuitive UI accessible on both iOS and Android. Key requirements include a Home Screen, Barcode Scanning Screen, User Profile Screen, and a screen that displays the nutritional information of a scanned product.

5.2 Hardware Interface Requirements

The app requires the following hardware components from the device running it: Camera, Touchscreen, and an Internet Connection. The camera is used in identifying a barcode and obtaining the barcode ID in order to access the corresponding database entry. An internet connection is necessary to pull from the database and display the necessary information to the user.

5.3 Software Interface Requirements

iOS and Android versions must be up to date with the app. A food data API is required so that we can store that information in a database and fetch the nutritional information. A local database is required for storing user data and the nutritional information.

5.4 Communication Interface Requirements

API calls are required for the frontend and backend and communication with AWS. This is done using the barcode as a key for the data that is being stored in the AWS.

6. System Architecture and Design

6.1 Logical View

A traditional client server model is used with mobile devices connecting to a centralized server which in turn connects to a database. REST APIs are used for all server endpoints in keeping with industry standards.

6.2 Hardware Architecture

There are two main components to the hardware architecture: mobile devices and cloud hosting. Both of these pose minimal hardware security challenges.

6.2.1 Security Hardware Architecture

6.2.1.1 Mobile Devices

Mobile device hardware security is rarely the responsibility of app developers. As such, the team shall focus on other, more pressing areas of security.

6.2.1.2 Cloud hosting

Cloud hosting removes all responsibility of the hardware security from the developer.

6.2.2 Performance Hardware Architecture

6.2.2.1 Cloud hosting

DevOps engineers can choose the kind of machine to host their projects. However, our team has chosen to utilize serverless architecture obfuscating all hardware.

6.3 Software Architecture

6.3.1 Security Software Architecture

OAuth2.0 is used to authenticate the user on login. All API's are secured with the Spring Security module. A VPC (Virtual Private Cloud) is used to protect the database and server from unwanted outside traffic. Traffic is let into the VPC by a load balancer on specific ports only.

6.3.2 Performance Software Architecture

The server is containerized and placed in AWS (Amazon Web Services) ECS (Elastic Container Service). ECS allows services to be assigned a given amount of CPU and memory without worrying about the underlying architecture. As such, the container can be scaled to meet any performance needs. Mobile devices will implement promises and async API calls to increase performance and responsiveness.

6.4 Information Architecture

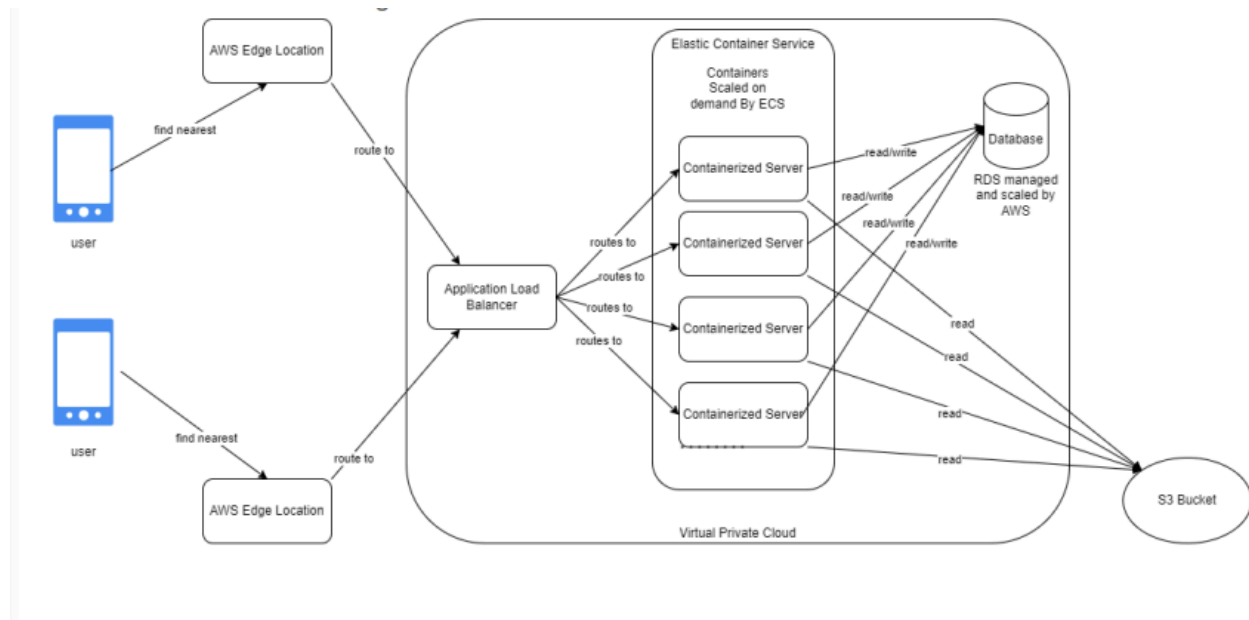
The data that will be stored will be the LJJSON files that contain the nutrition facts of every item on the Food Data Central API. The files will be hosted in an AWS S3 bucket and streamed into the database on a regularly scheduled cronjob.

6.5 Security Architecture

4.5.1 OAuth2: OAuth 2.0 — OAuth

4.5.2 Spring Security: Spring Security

6.6 Architecture Diagram



7. Application Testing

7.1 Application Requirements Testing Plan

For testing our application, we wanted to ensure that any requirements we had laid out earlier on in this project had been met. As such, we designed a document containing each of them and ran a demo of the app. From there, we simply navigated through the UI testing each functionality individually to ensure that they were all in working order.

7.2 Application Requirements Testing

	A	B	C
1	Requirement	PASS	FAIL
2	User Log in and Sign Up	1	
3	User Authentication		1
4	Home Page	1	
5	User Profile	1	
6	Barcode Scanner	1	
7	Nutrition Results	1	
8	Database Access	1	
9	App Settings	1	
10	AWS Cloud Hosting		

8. Version Control

8.1 Version Control Methods

For version control we used a joint Github repository which we would branch as needed to keep up to date with the work our group mates were doing.

8.2 Version Control Pushes

In order to create a DevOps pipeline with Continuous Deployment enabled, the following technologies and strategies were used. On push or PR to main (formerly master), a GitHub action is automatically run. The action performs the following steps at a high level.

- Builds the project (and tests for any errors)
- Retrieves AWS credentials
- Connects to the AWS Elastic Container Registry
- Builds the container image using Spring Boot build packs
- Pushes the image to ECR
- Downloads the AWS task definition
- Updates the AWS task definition
- Deploys the updated task

9. Development

9.1 Development Process Backend

We had primary developers for both front end and backend. The process for backend development was to first find a way to load in all the data we needed to be able to query. Once this data was loaded in, we were able to test our queries to ensure their functionality. After this was done, the focus shifted to hosting using AWS. When hosted, final features such as allergy detection and daily totals were added in as well.

9.2 Development Process Frontend

On the frontend, everything was done using React Native and Typescript. First, the plans for the UI were drawn out to show what we wanted the app to look like. Following this, the bare UI itself was designed from the ground up and the barcode scanner functionality was built in. The last step for front end was to connect with the backend so that the code for queries could be used in conjunction with the UI.

9.3 Frontend Roadblocks

One of the major roadblocks we faced was the decision to utilize React Native CLI instead of Expo. This choice significantly affected our development velocity due to the necessity for manual updates and maintenance of Gradle files and podfiles. Each update required a detailed and careful approach to ensure compatibility and functionality across different versions and platforms, which often led to time-consuming troubleshooting sessions.

9.4 Development Roadblocks

Although creating a full-text search API from scratch sounds challenging, selecting appropriate tools made the process significantly easier. Using Spring Boot, JPA/Hibernate, and Hibernate Search allowed for easy and rapid development of the API. However, creating such an API inherently requires all data to be stored by the team and not accessed independently through a 3rd party API. Acquiring the data is relatively easy, but ingesting and indexing the data was non-trivial. Most notably, the data was acquired as a LJSON file approximately 3GB large. Using Jackson to ingest the data and Hibernate Search to index the data takes a significant amount of time. In practice, the data cannot be ingested on a local machine because of these time constraints. Because Spring Boot includes build packs, it is simple to containerize the application. However, the containerized application requires volume mounts for the index files. Once containerized

the image can be easily pushed to the desired repository for hosting. Hosting took a non-trivial amount of time because of a desire to minimize costs. In the end, the team opted to open additional VPC endpoints to simplify the process. However given more time, we would like to host the project without opening these endpoints and occurring the associated costs.

9.5 Decisions on Technologies

We decided to use Docker for containerizing our service so that it can be easily run from any device without any extra preparation being necessary. When we settled on hosting from AWS, we ran into several issues along the way, but found it to be the most effective in our goal to host everything from a cloud service.

10. Summary

10.1 Conclusions and Takeaways

Throughout the course of this project, we have found a great deal of success in implementing our original goals. Having fulfilled all the prioritized requirements of the project, we have gained a great deal of experience in React Native, AWS, and backend Java development. We hope to move forward and apply these concepts and expand even further on our current app to create something that can make genuine positive change in the health and wellness of its users.