DIGITAL ACCESS TO SCHOLARSHIP AT HARVARD
DASH.HARVARD.EDU

HARVARD LIBRARY
Office for Scholarly Communication

Eatable: An Application That Helps People With Food Allergies Check and Locate Allergen-Free Food Products

Citation
Prajapati, Sabina. 2017. Eatable: An Application That Helps People With Food Allergies Check and Locate Allergen-Free Food Products. Master's thesis, Harvard Extension School.

Permanent link
http://nrs.harvard.edu/urn-3:HUL.InstRepos:33826056

Terms of Use
This article was downloaded from Harvard University's DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA

Share Your Story

The Harvard community has made this article openly available.
Please share how this access benefits you. Submit a story .

Accessibility

Eatable: An Application That Helps People with Food Allergies Check and Locate Allergen-Free Food Products

Sabina Prajapati

A Thesis in the Field of Information Technology

for the Degree of Master of Liberal Arts in Extension Studies

Harvard University

May 2017

Abstract

Due to the limited number of allergen-free foods found in the market and the stores' inconsistencies in carrying such foods, people living with food allergies have a hard time finding foods they can safely eat. Food allergic people spend a lot of time reading ingredient lists of products at the store to verify their safety without knowing whether the store even carries any allergen-free products, and when the product is not found in the store, they go to a different store with the same uncertainty. In this project, I have created an application that solves this problem by enabling users to locate allergen-free foods through a mobile app so that they don't have to waste their time in search of safe foods. Eatable is a cloud based hybrid mobile application built using crowd-sourced data from users. It integrates with various third-party web services to build product-to-store mapping database that serves as the foundation of the system. The application is built with technologies that are performant, flexible and scalable.

Dedication

Dedicated to my son, Navayush Gurung, whose struggle with food allergies made me aware of the challenges of finding allergen-free foods, and inspired me to create this application.

Acknowledgement

I would like to thank my husband for his unwavering encouragement and support throughout the project. To my son, for understanding that his mom couldn't play with him because she had to work. To my family, for always advising, loving and supporting me. I would like to thank my thesis director, Eric Geiseke, for providing invaluable technical guidance throughout the project. Finally, to my thesis advisor, Jeff Parker, for helping me refine my thesis document.

Table of Contents

Chapter 1 Introduction 1
Chapter 2 Prior Work 3
Chapter 3 Requirements 5
    Mobile Access 5
    User Management 5
    Product Verification 5
    Mapping Data Collection 6
    Product Locate 7
    Store Search 7
    Non-Functional Requirement 7
    Use Case 8
Chapter 4 System Overview 9
Chapter 5 Design and Technology Choices 11
    Application Design 11
    Mobile Client Design 13
    Backend System Design 17
Chapter 6 Product Store Mapping 23
Chapter 7 Implementation 26
    RESTful API Module 26
        UserController Interface 27
        ProductController Interface 27
    Business Service Module 28
        IProductService Interface 31
        IStoreService Interface 32
        IUserService Interface 32

Data Access Service Module ... 33
IProductDAO Interface ... 38
IStoreDAO Interface ... 38
IUserDAO Interface ... 39
Mobile Client ... 39
User Interface ... 42
Register Page ... 42
Home Page ... 45
Locate Page ... 47
Eatable Foods Page ... 49
Login Page ... 51
Settings Page ... 52
Chapter 8 Development and Test Environment Setup ... 54
Mobile Client ... 54
Backend System ... 57
Chapter 9 Deployment and Evaluation ... 59
Chapter 10 Summary and Conclusion ... 61
References ... 64

List of Figures

Figure 1. User actions supported by the Eatable application ... 8

Figure 2. The Eatable System Architecture ... 9

Figure 3. Multiple clients using the same Backend System ... 13

Figure 4. The original sketches of UI design ... 15

Figure 5. Eatable User Interface ... 16

Figure 6. The Eatable data model ... 18

Figure 7. MongoDB Collections and their properties ... 19

Figure 8. The three modules of the Backend System ... 20

Figure 9. The UML class diagram of the Backend System ... 22

Figure 10. The activities that occur when user scans a product ... 25

Figure 11. The UML class diagram of the Backend Service Module. ... 29

Figure 12. The UML class diagram of the Data Access Module ... 34

Figure 13. The properties of the Product object ... 36

Figure 14. The properties of the Store object ... 37

Figure 15. The properties of the User object ... 37

Figure 16. The <ion-nav-view> directive in index.html ... 40

Figure 17. The AngularUI Router state definitions ... 41

Figure 18. The Register Page ... 43

Figure 19. The Register Page with validation errors ... 44

Figure 20. The Sequence Diagram of the registration process 45

Figure 21. The Home Page 46

Figure 22. The Sequence Diagram of the product verification process 47

Figure 23. The Locate Eatable Page 48

Figure 24. The Eatable Foods Page 49

Figure 25. The Store Locations Page 50

Figure 26. The Eatable Products Page 51

Figure 27. The Login Page 52

Figure 28. The Settings Page 53

Figure 29. The configuration of Nexus 5 Emulator 55

Figure 30. The Eatable application on the emulator 56

Chapter 1 Introduction

Finding allergen-free food is difficult primarily because the number and variety of such foods in the market is very limited. On top of that, stores usually only carry a limited selection of allergen-free foods, and different stores in the same chain won't necessarily stock the same products. Furthermore, it isn't always clear where an item will be shelved. Because of these inconsistencies, it is hard for an allergic person to know about all the available food options he has in the market. Another factor that adds complexity in finding allergen-free foods is the process of determining whether a product is safe to buy. People with food allergies spend substantial amount of their grocery shopping time reading ingredient lists of products to verify their safety, which makes their grocery shopping experience time-consuming and unpleasant.

Food allergy is a growing problem. One of the studies conducted by CDC in 2013 showed that between 1997 and 2011, there was approximately a 50% increase in food allergies among children (*Facts and Statistics*). This is an astounding growth. Similarly, it is estimated that 15 million people in the United States today have food allergies (*Facts and Statistics*), and 1 in every 13 children under the age of 18 suffers from them. Until a cure is found, the only option people suffering from food allergies have is to completely avoid foods that they are allergic to. This is not, however, as easy to do as it may seem because most common allergens that have caused reactions are basic ingredients like milk, egg, wheat, peanut, tree-nuts, soy, fish and shellfish; that are found in almost all of

the regular foods sold in the market. Although, alternate food market has significantly grown in the last few years and there are more companies making new allergen-free products now than ever before, the number of such items is still very limited and finding them is hard because of the issues described above.

Eatable is an application that aims to solve challenges of finding allergen-free foods by providing a platform that lets users search and locate safe foods through their mobile phones. It also helps users save time by letting them check for product's safety automatically. The application is built using data provided by users. When users use the application to check for a product's safety, it fetches users' location and creates records in its database mapping the products and the stores. It integrates with Google Web Services API and Label API third party services to gather necessary location and product information. The version of the product delivered as part of this project is a Minimal Viable Product (MVP) of Eatable that contains four basic functionalities. The functionalities that it currently supports include a user-management feature that enables the users to register and sign in with the app, a check-product feature that enables the user to check for a product's safety, a store-search feature that enables the user to search for all the available safe products and trace where they are found, and a locate-feature that enables the user to find out where a specific product is sold. The application is a hybrid app built on Ionic framework and is currently available in Android version.

Chapter 2 Prior Work

There are many apps in the market that address various challenges of living with food allergies, but none of them help people locate food products as Eatable does. It addresses a problem that hasn't been addressed before; however the ideas on which it is built upon are not really ground breaking. It has indeed drawn inspiration from many existing applications like eBird (http://ebird.org/content/ebird/), AllergyEats (https://www.allergyeats.com/), ShopWell (http://www.shopwell.com/) and many other food allergy apps.

The mobile barcode scanning functionality Eatable uses to check a product's safety is actually a very common functionality in food allergy app domain. Many apps like ipiit (https://www.ipiit.com/) and ContentChecked (http://www.contentchecked.com/) provide it as their core feature, and almost all the apps use this functionality one way or the other. Conceptually, Eatable is closest to an app called AllergyEats in the sense that both help users find allergen free foods using input from other users. There are however two notable differences. AllergyEats is focused in helping users find allergy-friendly restaurants, while Eatable focuses on grocery products. The second difference is that AllergyEats depends on user's manual feedback, while Eatable creates data automatically without user's explicit input. Since AllergyEats depends on user's voluntary involvement, the data it records can be incorrect and

sporadic, but in Eatable the data is consistently generated and the chances of error are minimal.

The idea to use crowd sourced data to create product-to-store mappings was inspired by the eBird app, which uses similar idea to create bird observation database that contains bird sighting records to help scientists uncover patterns of bird distribution and movements. The app was launched in 2002 by the Cornell Lab of Ornithology and National Audubon Society, and is considered one of the largest and fastest growing biodiversity data resources available today. Without this idea Eatable would not have been possible.

Chapter 3 Requirements

The Eatable application implements the following requirements:

Mobile Access

Users can access Eatable application through mobile devices. The mobile access is integral to the system as it provides product barcode and geo-location data which are required to create product-to-store mappings.

User Management

Eatable requires users to register with the system before using it because it collects user data such as user allergies as part of the registration process. Users configure the application with a list of allergens they wish to avoid. The system currently supports 6 allergens: Milk, Egg, Peanut, Soy, Almond and Shellfish. Users are required to select allergens from this list when registering.

Product Verification

Eatable lets users check product's safety based on their allergies. User initiates the product verification process by scanning the product's barcode. The barcode provides the app with the product's UPC code, which is used to fetch the product's ingredient list from

the Label API (*Label API*) service. The app determines the product's safety by checking the ingredient list of the product against the allergens defined in the user's profile.

The product information obtained from the Label API is also stored in Eatable's database hosted in the cloud so that the subsequent checks for the same barcode do not require calls to Label API.

Mapping Data Collection

The application needs to learn about where the products are sold so that it can inform users where they are found. Each time a user scans a product at a store; the application fetches user's location and creates a record mapping the product to the store. In this way, it builds the mapping database that provides it with the intelligence about which products are found in which stores.

The data collected from the users contain geo-location of the store and the barcode of the product. Google Services API (*Google Maps APIs Web Services*) provides the application with store information and Label API (*Label API*) provides product related data. Store information consists of store name, address, geo-location and google place id. Each store is uniquely identified by its name and address. Product information consists of product name, brand name, description, UPC code and list of allergens the product does not contain. Each product is uniquely identified by its UPC code.

The mapping process does not slow the performance of the verification functionality by executing the mapping task in the background thread. The mapping records, product and store data must all be unique in the Eatable database.

Product Locate

Eatable lets users locate products using any term in the product's name, brand or description. The result is a list of store locations that carry the products that match the search string. The store locations are sorted by distance from user's current location in ascending order. Users are able to see the distance value from their current location to each store as well as bring up the store on a Google Map.

Store Search

Store Search lets users see all the stores in Eatable database that carry products safe for them. The result is grouped by store brands such as Stop and Shop, Market Basket, Whole Foods etc. Each result for a brand shows the total number of unique products and the total number of locations that carry these products. Users can drill down on the result to see the addresses of these locations and how many unique products are sold at each location. Users can also see the list of products sold at each location. The product page displays the name, brand and description of the product.

Non-Functional Requirement

The overall goal of the Eatable system from a non-functional point of view is to create a performant, scalable and flexible application that can adapt to changes with minimal disruption and handle increasing volume of data without affecting the system's performance. It is expected that new allergens will be added to the system, and the nature and volume of its data will change with app's increasing use. The feature set will grow

per user and market needs. There is also a possibility for the system to support new user interfaces or change parts of the Backend System. These changes may fit within the design realm of the system or may require changes in one or more integral components of the system. Systems that are not built with good design are usually not able to withstand such changes. Therefore, Eatable must be built as a modular, robust, extensible, flexible and scalable system where components are loosely coupled and each module is implemented on open/close design principle.

Use Case

Eatable currently supports only one type of user who can perform actions shown in the Use case diagram Figure 1.

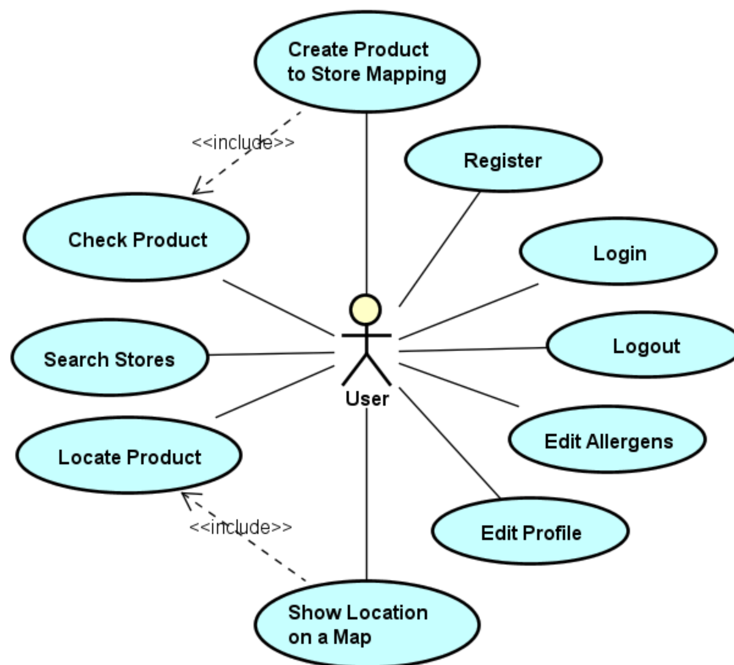

Figure 1. User actions supported by the Eatable application

Chapter 4 System Overview

The Eatable system consists of independent client and backend modules that interact with each other through RESTful API. The Backend System is hosted in Amazon EC2 (*Elastic Compute Cloud (EC2)*) while the Client System is a hybrid mobile app that provides User Interface to the application and depends on the Backend System for all its data service needs. MongoDB (*MongoDB 3.4: Your Database Evolved*), a document oriented NoSql database, is used for data persistence. The system depends on the third party Service APIs to find product and store information. The Google Services API (*Google Maps APIs Web Services*) provides store and location information to the system while the Label API (*LabelAPI*) provides product related information. Figure 2 shows the overall System Architecture of the application.

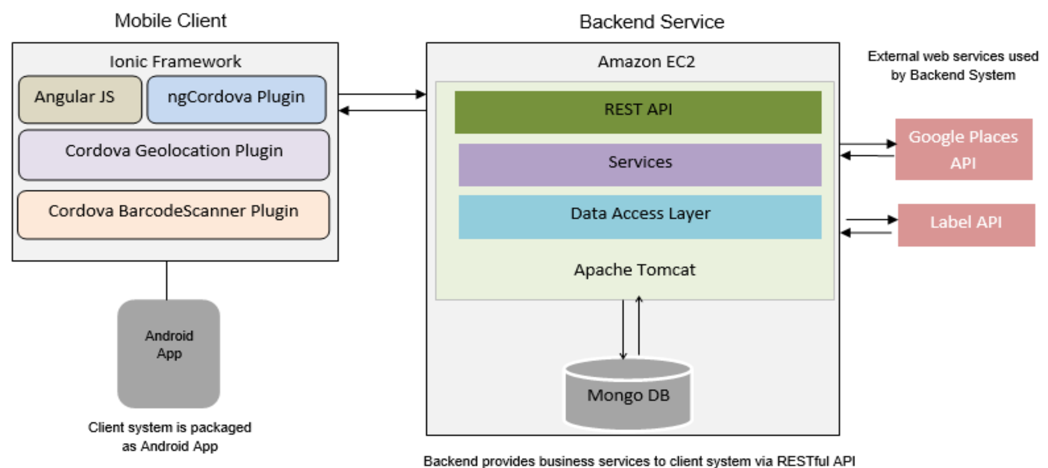

Figure 2. The Eatable System Architecture

The Mobile Client module is built on the Ionic framework (*Getting Started with Ionic*) and deployed as an Android app for this project. The ngCordova (*ngCordova - Simple extensions for common Cordova Plugins*), Cordova BarcodeScanner (*$cordovaBarcodeScanner*) and Cordova GeoLocation (*$cordovaGeolocation*) plugins are used to provide the required hardware API access to the mobile app. The REST calls to the Backend System are facilitated by the $http Angular service that provides mechanism to make HTTP requests to the remote servers.

The Backend System module is a Java application that is deployed in the Apache Tomcat 8 web server (*Apache Tomcat*). It provides the business and the data logic to the application. It consists of a RESTful API module, a Business Service module and a Data Access Service module. The RESTful API module provides service interfaces to the mobile client. The Business Service module provides all the business logic including the data verifications and creating product-to-store mapping records. The Data Access Service module is responsible for all the persistence logic. These three modules are designed to be loosely coupled and do not share data objects. Only Data Access layer is aware of the underlying Mongo DB while only the RESTful API layer manipulates the client objects.

Chapter 5 Design and Technology Choices

Many technologies exist to implement an application like Eatable, so we evaluated a number of technologies to choose the best ones that satisfy the project's functional and non-functional requirements. This chapter describes these choices.

Application Design

The app was originally proposed as a native Android app mainly because of the performance reason since hybrid applications (Bristowe John, March 2015) are slower than the native apps. The native apps are built for a specific operating system with platform specific technologies and run on the device's operating system with direct access to the platform APIs. On the other hand, the Hybrid apps run within a component called 'native' wrapper that packages the web app into a mobile application. The UI is also rendered in custom components such as WebView and UIWebView unlike the native apps which are rendered in native UI framework. All this adds an extra layer of processing to the hybrid apps making them slower than their native counterparts.

However, further research showed that this latency is insignificant on new frameworks and systems, especially if the application doesn't use many native component resources. This is true with Eatable as it only uses GPS and Camera components, and additionally, the access to these components is limited to only one

feature. Since the performance hindrance is not an issue, it was decided that Eatable would be implemented as a hybrid app.

One of the biggest advantages of creating a hybrid application is that it can run on multiple platforms, which means it can reach more devices without the need to create platform specific versions. This is important for a crowd sourced application like Eatable whose success depends on the number of users it can reach. Making the application hybrid also avoided the need for platform specific knowledge.

The design to separate the client and the backend systems was inspired by Open/Closed software design principle (Meyer, 1988), which states that a software should be open for extension but closed for modification. This design also meets the non-functional requirement of the project defined in Chapter 3. Separating the Backend System and the Mobile Client System makes it possible in a higher level to extend or change one of the components without impacting the other. For example, Amazon Echo can be added as a new client to Eatable without requiring any changes to the Backend System. Figure 3 shows potential clients that can be added to the Backend System. Similarly, Backend System can be replaced or updated with minimal impact on the clients.

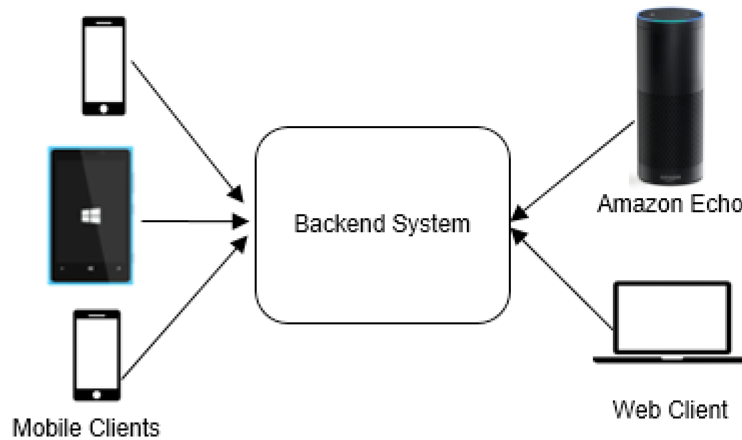

Figure 3. Multiple clients using the same Backend System

Mobile Client Design

The Ionic framework is an HTML5 UI development framework for hybrid mobile applications built on top of Apache Cordova (*Android Platform Guide*) and AngularJS (*AngularJS*). The Apache Cordova packages HTML5 web applications into native apps so that these applications can be deployed to mobile devices, and AngularJS provides front-end web application framework to Ionic. Although comparable to frameworks such as Bootstrap, it is much more than an UI framework. Ionic simplifies the entire process of building interactive hybrid mobile apps in addition to providing rich set of interactive components. It has powerful command line interface (CLI) tools that make it easy to create, update, package and deploy apps on emulators and real devices. Ionic also supports live-reload feature that automatically applies new changes to the running app, which is very handy during the development process. Apart from building a responsive UI, it also customizes the UI components based on the targeted platform to give them

native look and feel. Despite being one of the newest technologies in the hybrid app development world, Ionic is considered one of the best with great documentation support, which is why it was selected.

Choosing Ionic framework defaulted UI implementation framework to AngularJS. The AngularJS is a very powerful front-end JavaScript framework that provides model-view-controller (MVC) architecture to the UI development. It has components and services that simplify the web development while creating an interactive and responsive UI. The HTML directives bind HTML elements to the data models and the built-in Services provide various functionalities to the application. It has become very popular in building single-page web applications because of its data binding and dependency injection features.

The Cordova plugins used in the mobile app are provided by ngCordova library that wraps core Cordova plugins into AngularJS versions. These plugins provide platform specific functionalities to the app. The BarcodeScanner plugin provides access to the camera to scan the product, and the GeoLocation plugin provides access to the GPS to fetch user's geo-location data.

Figure 4 shows initially sketched rough wireframes of the Eatable UI. The sketches capture the structure of the prospective pages and navigation between them.

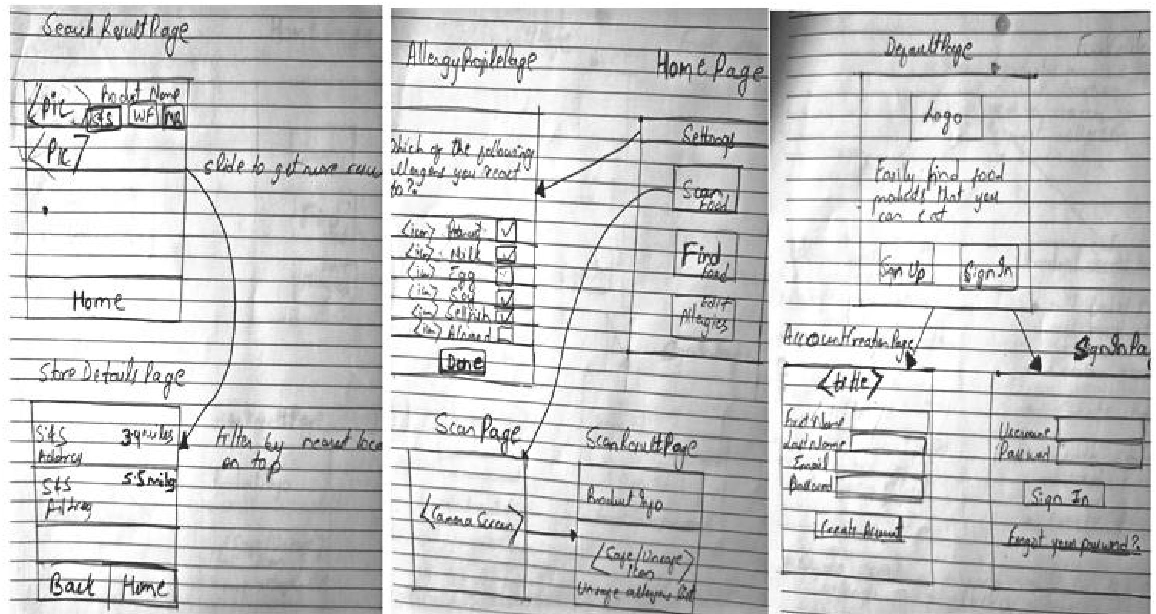

Figure 4. The original sketches of UI design

Figure 5 shows the implemented pages of Eatable UI. The figure also captures the navigation model of the pages. The main content panel of Home page serves as a gateway to all of the Eatable's functionalities while the Setting page provides user management functions. The selected color theme gives the UI a clean look and feel.

Figure 5. Eatable User Interface

Backend System Design

Java and Tomcat 8 Web server were chosen to implement the Backend System because these technologies provide cross-platform framework to build enterprise level object-oriented web applications. Java is a very robust language and Apache Tomcat is one of the most popular open-sourced Web servers today. MongoDB (*MongoDB 3.4: Your Database Evolved*) was selected as a Database Management System instead of MySql as was originally proposed because MongoDB is more flexible and scalable in nature than MySql. Morphia (*Morphia*), a mapping framework that maps Java objects to MongoDB documents is used as an Object Relational Mapping (ORM) system to persist data models to the MongoDB.

Figure 6 shows structured data model diagram of Eatable which is quite simple since the app currently offers only a handful of features, but this structured model can easily become very complex as new features are added to the system.

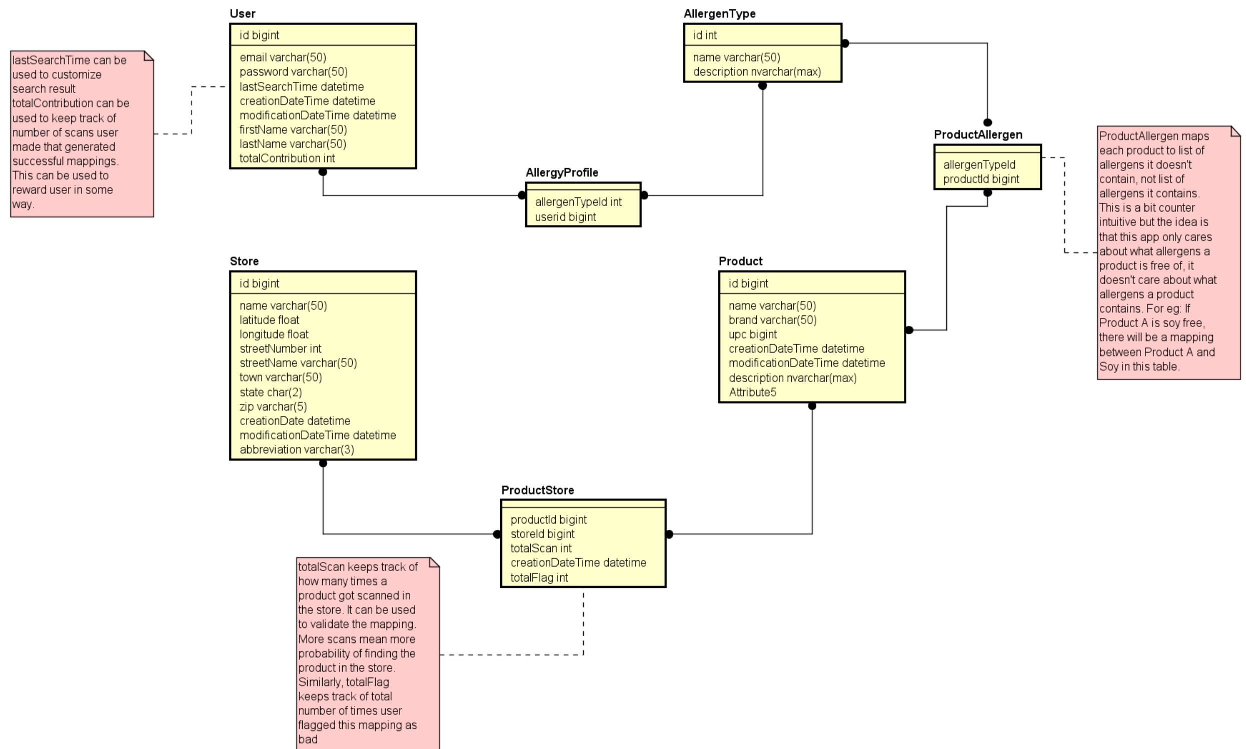

Figure 6. The Eatable data model

On the other hand, MongoDB being a non-structured and document based database, its data models are usually simpler, and can adapt to schema and data changes with less disruption. Figure 7 shows the same data model represented by three MongoDB Collections. A Collection is analogous to a table in traditional databases, but unlike tables where the schema is rigid, the MongoDB Collection doesn't enforce a schema. Since MongoDB allows two documents in the same Collection to have different properties and for the documents to be embedded into one another, a new type of data can be inserted in the Collection without any change to the model. It is also designed to scale so it can support large volume of data without impacting the system performance.

```
> db.getCollectionNames()
[ "Product", "Store", "User" ]
> db.Product.find({"upc": "661799264327"}).pretty()
{
        "_id" : ObjectId("584c8e3be882401c9a094dc9"),
        "className" : "com.app.mongo.data.model.ProductModel",
        "upc" : "661799264327",
        "name" : "GOBEYOND BEYOND ICECREAM",
        "brandName" : "GOBEYOND",
        "description" : "BEYOND ICECREAM",
        "noAllergen" : [
                "shellfish",
                "egg",
                "fish",
                "peanut",
                "tree nuts",
                "sesame seeds",
                "gluten",
                "wheat",
                "coconut"
        ],
        "stores" : [
                ObjectId("584c89d6e882401c9a094dbc"),
                ObjectId("584c8ab5e882401c9a094dc2"),
                ObjectId("584c8a1ae882401c9a094dbe"),
                ObjectId("584c8a2be882401c9a094dbf"),
                ObjectId("584c8bace882401c9a094dc6")
        ],
        "creationDate" : ISODate("2016-12-10T23:22:35.603Z"),
        "modificationDate" : ISODate("2016-12-10T23:22:35.603Z")
}
> db.Store.findOne()
{
        "_id" : ObjectId("584c6dcbe8824017aa50d8c2"),
        "className" : "com.app.mongo.data.model.StoreModel",
        "streetNumber" : 471,
        "streetName" : "Salem Street",
        "city" : "Medford",
        "state" : "Massachusetts",
        "stateAbbr" : "MA",
        "storeName" : "Stop & Shop",
        "country" : "United States",
        "zipcode" : "02155",
        "creationDate" : ISODate("2016-12-10T21:04:11.100Z"),
        "modificationDate" : ISODate("2016-12-10T21:04:11.100Z"),
        "geoLocation" : "42.423044, -71.089060",
        "googlePlaceId" : "ChIJcdRHI05x44kR2MqPTo_iKkE"
}
> db.User.findOne()
{
        "_id" : ObjectId("584d5a27e882401c9a094dca"),
        "className" : "com.app.mongo.data.model.UserModel",
        "userName" : "ngurung@gmail.com",
        "password" : "e10adc3949ba59abbe56e057f20f883e",
        "firstName" : "Navayush",
        "lastName" : "Gurung",
        "totalContribution" : 0,
        "allergy" : [
                "peanut",
                "milk"
        ]
}
```

Figure 7. MongoDB Collections and their properties

Figure 8 shows how the three independent backend modules are related to each other. Each module provides its service through interface so that the implementation details are not exposed to the calling module. The RESTful API and the Business Service modules have data object mapper components that map data objects between the modules. Dozer (http://dozer.sourceforge.net/) library was selected as the data mapper.

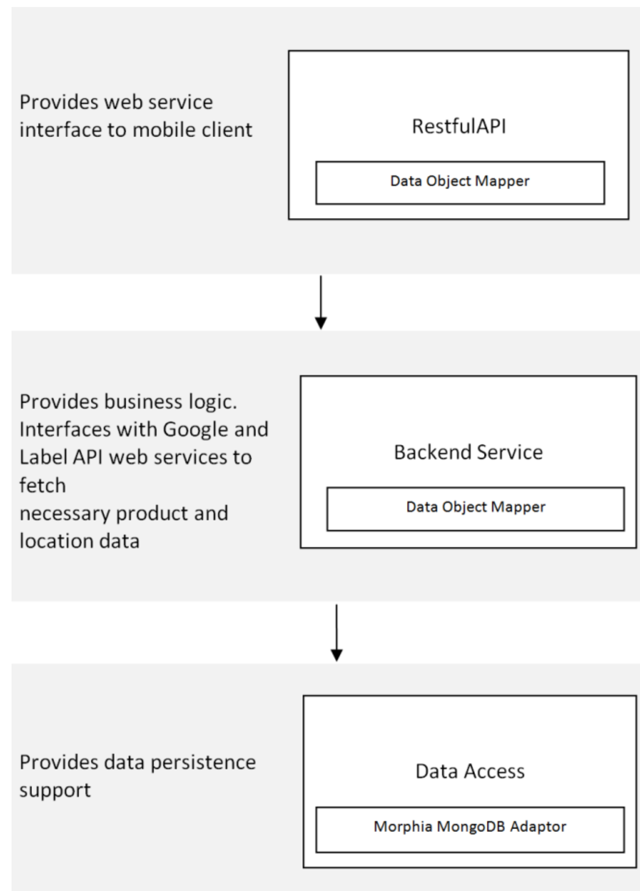

Figure 8. The three modules of the Backend System

The RESTful API module is implemented in the Jersey library (*Jersey*), which is a stable and a popular framework to implement Java based RESTful web services. Jersey

provides a framework to implement web services both as hosts and consumers. It supports JSON format, which is the data exchange format of the application. It provides annotations to easily convert Java classes into web services, and it has a robust client API to implement web service clients. In all, Jersey is a good fit for the Backend System to implement RESTful interfaces.

The Google Services API used by the Backend System includes the Google Places API, the Google Maps Distance Matrix API and the Google Maps API. The Google Places API provides the store information to the Backend System. The Google Maps Distance Matrix API calculates the distance between the user's current location and the stores, and the Google Maps API provides the logic to show the store location on a Google map.

The Label API provides product information such as ingredient list, name, brand, and description to the Backend System. Label API is a database of grocery products found in the United States, and provides access to its data through REST interface. This service was chosen because no other product databases provided Web service access to their data.

The UML class diagram in Figure 9 shows how different components in the Backend System relate to one another.

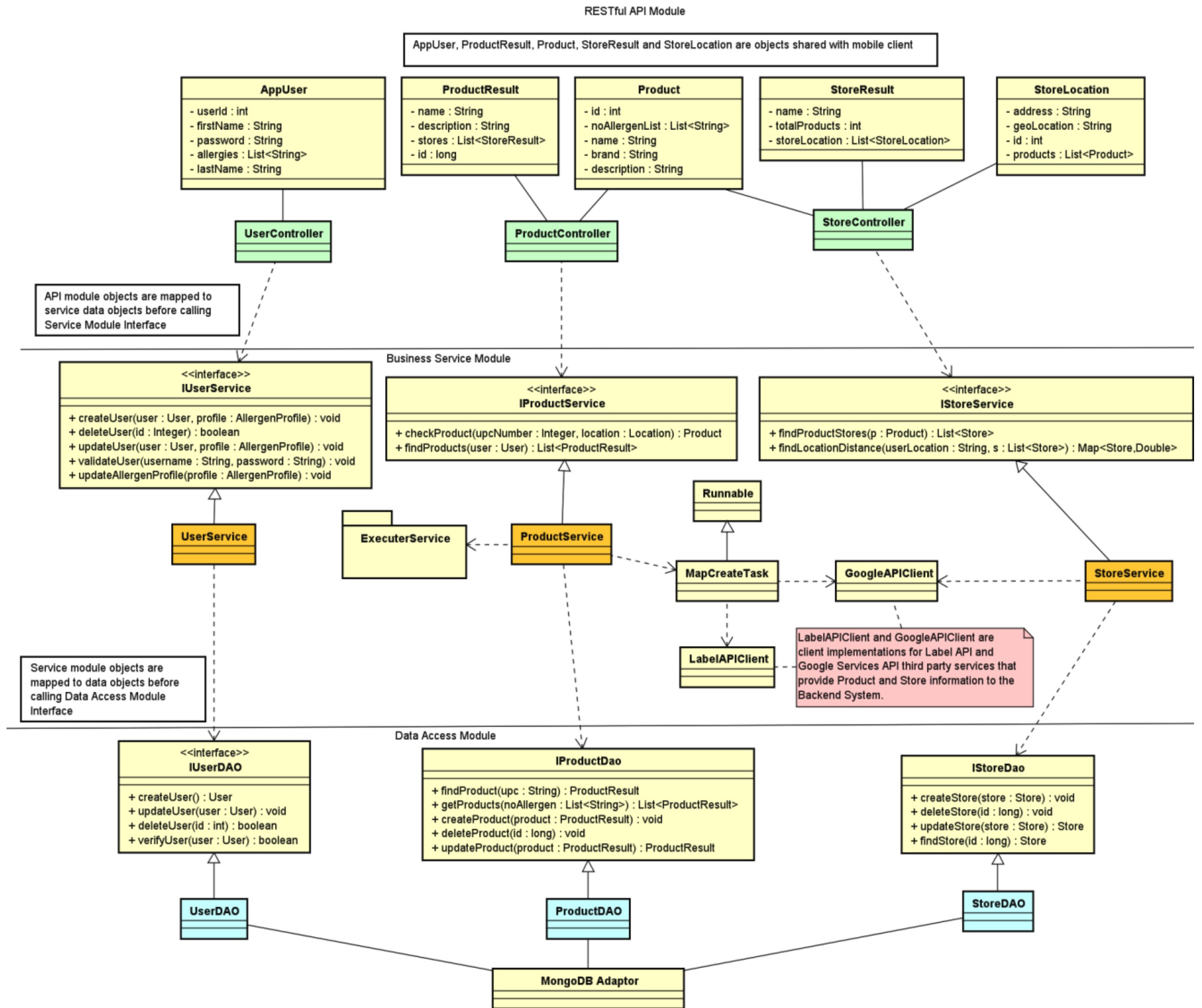

Figure 9. The UML class diagram of the Backend System

Chapter 6 Product Store Mapping

The main functionality that differentiates Eatable from other food allergy apps is its ability to locate products to inform users where to shop. Eatable provides this functionality by creating product-to-store mapping records that maps the products to the stores that sell them using crowd-sourced user input and third party integrations.

Crowdsourcing means using input from the application users to generate potentially large amount of meaningful data. In crowd sourced applications, users become the active contributors of the data to the system rather than simply being its consumers. There are many applications that are built on this idea. As mentioned in the Prior Work section, eBird is one such application that has inspired this project. Some other crowdsourcing applications include Waze (https://www.waze.com/) and OpenStreetMap (https://www.openstreetmap.org/).

Eatable crowd sources the scan data from its users when they use the app to scan the food products. Every scan initiates the process of creating mapping records because that is when the system can fetch the information such as the product's UPC code and the user's current geo-location, that are needed to create the mappings. The scan data is fed to the Backend System, which uses various third party services mentioned in the Backend Design Chapter 5 to fetch product and store information. It uses the UPC code to find information about the scanned product and uses the geo-location data to find the store

from where the product was scanned. Once the system finds this information, it creates record mapping the product to the store.

Activity diagram in Figure 10 shows how different components of the system interact to create product-to-store mapping record.

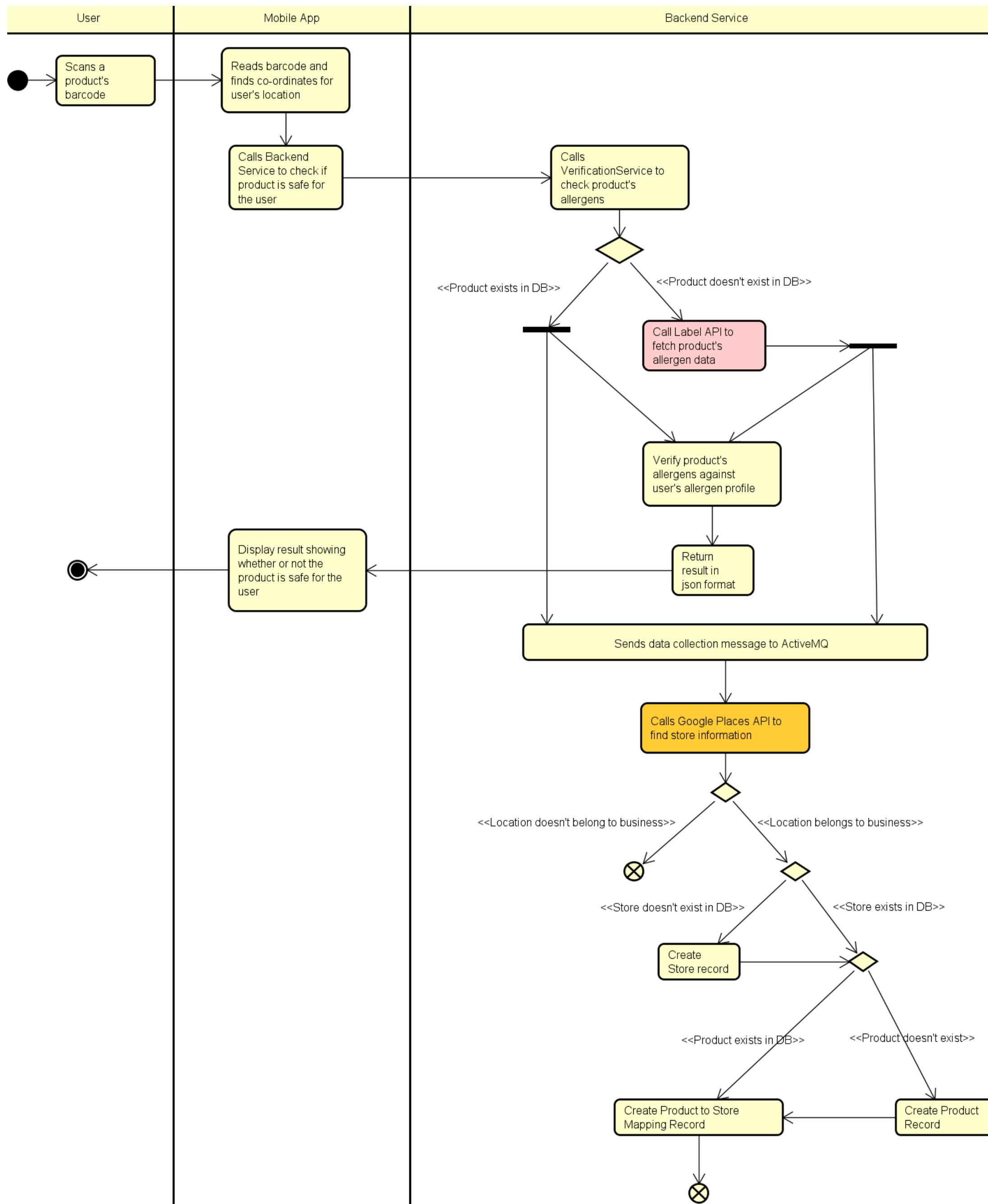

Figure 10. The activities that occur when user scans a product

Chapter 7 Implementation

This chapter provides implementation details of the Backend System modules and the Mobile Client. It describes technologies and service interfaces implemented by each module to meet the product requirements. It also includes the definition of the User Interface components and describes how they provide access to various features supported by the system.

RESTful API Module

The RESTful API Module provides RESTful service interface to the mobile client. It depends on the Business Service Module for business logic needs and implements its own set of data objects that are shared with the client. It divides the interface into ProductController and UserController services.

The interface classes are defined as web services through Jersey annotations. The @Path annotations on the classes identify the URI paths the classes serve. Using this annotation, the ProductController is mapped to the '/product' path and the UserController is mapped to the '/user' path. The public methods in the classes are also annotated with the @Path annotations to define the URI paths of the resources. The methods also include HTTP verb annotations to define the type of HTTP requests (GET, POST, PUT, etc) they process. The Jersey servlet that runs in the Tomcat server container receives all the web service requests, and funnels these requests to the appropriate classes and methods based

on the defined URI paths. The @Consumes and @Produces annotations are used to define the format of the data exchange.

Following sections describe the service interfaces provided by the RESTful API module:

UserController Interface

The UserController interface provides the following methods to support the user management functionalities described in User Management requirement, Chapter 3.

- createUser: The mobile client uses this method to register a new user with the system. The method takes an AppUser object as a parameter that contains all the information required to register a new user. The Response contains the newly created User object if the call succeeds.
- validateUser: The mobile client uses this method to validate users when they login to the system. The method takes two strings, username and password. The Response contains reason for failure if the validation fails.
- updateUser: The mobile client uses this method to update data of the User object

ProductController Interface

The ProductController interface provides the following methods to check and locate products.

- checkProduct: The mobile client uses this method to check the product's safety. The caller provides the UPC code of the scanned product and the user's allergen list. The controller calls the Business Service module to fetch the product by UPC

code and compares its allergen list with the user's allergens to verify product's safety. It returns AllergenCheckResult object that contains the result, and if the result is negative, it also contains the list of allergens for which the verification failed.

- findProducts: This method takes noAllergen as a parameter that holds a list of allergens and returns all the products in the Eatable database that do not contain the allergens listed in the parameter. The method also fetches the store mapping records for each product in the result using the StoreService interface and aggregates the result by the store brand. The mobile client uses this method to show all the products the user can safely eat along with the store locations that carry these products.
- locateProduct: The mobile client uses this method to locate a particular product by a search term. The method takes the user's current geo-location data as a parameter. It returns all the store locations that carry products whose name, description or brand name match the search string. The result also contains the distance from the user's current location to the each store location in the result. The controller uses StoreService and ProductService interfaces to fetch the required data.

Business Service Module

The Backend Service module implements business service interfaces for the RESTful API module. It provides different interfaces for the Product, Store and User

objects to keep the logic separate. As mentioned in the design section of Chapter 5, it depends on the Data Access Service module to provide data persistence and implements its own set of data models. Figure 11 UML diagram shows the interfaces and the classes implemented in the Business Service module.

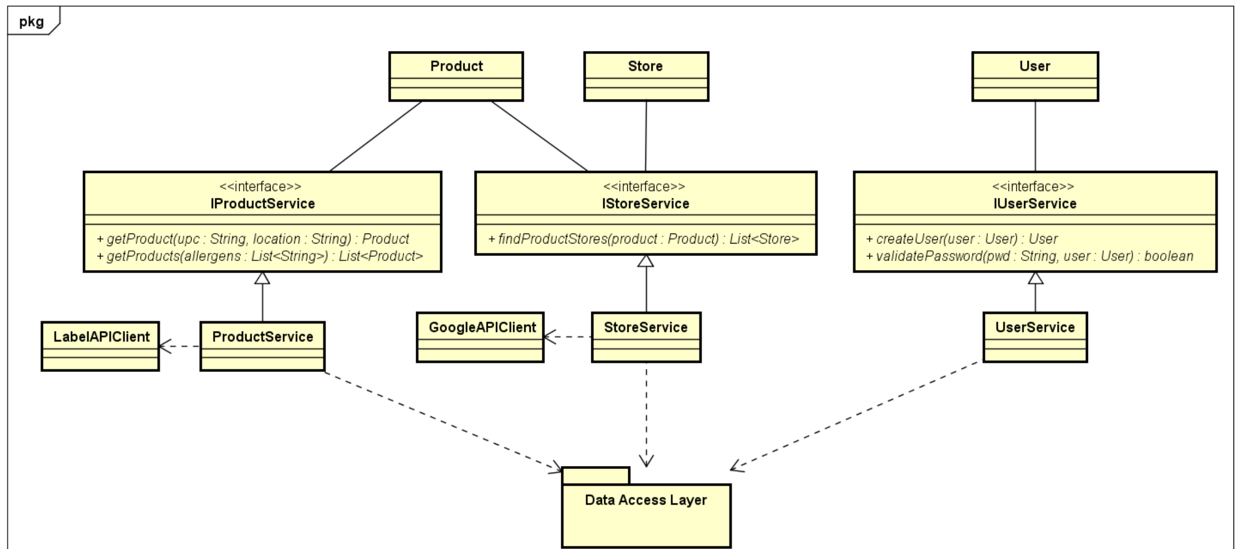

Figure 11. The UML class diagram of the Backend Service Module.

The Backend Service module implements the LabelAPIClient and the GoogleAPIClient classes as web service clients using the Jersey client library. The LabelAPIClient manages service calls to the Label API service, and the GoogleAPIClient calls Google API services. The API keys are required to use these services so the application first needs to register itself with these services and get API keys before it can make service calls to them.

The LabelAPIClient uses the ‘label’ and ‘searchprods’ resources to fetch product information from the Label API service. Both the methods take UPC code as a parameter and returns different type of product related information. The ‘label’ resource returns a list of allergens along with their allergen values for the matched product. The allergen value is a number that represents how much weight a particular allergen has in a product. For example, if an allergen has allergen_value=0, it means that the product doesn’t contain the allergen. The LabelAPIClient uses this result to build ‘noAllergen’ list for the product. The ‘searchprods’ resource provides product detail information including name, brand and description to the system.

The GoogleAPIClient uses the ‘place/nearbysearch’ resource to find location information. It takes geo-location data as an input and returns place information such as the place id and the name. The method also accepts options to limit the search by certain radius and filter the search by keywords. The search radius is limited to 50 meters and the filter keyword used in the call is ‘grocery.’ These options make sure the correct grocery store is returned. The ‘nearbysearch’ method however doesn’t return the address of the place so the GoogleAPIClient needs to make a second call to get the complete set of store information. It uses ‘place id’ value returned by the previous call to make a call to the ‘place/detail’ resource to get the store address. The distance between the store and user’s location is calculated using Google Distance Matrix API. The ‘distancematrix’ resource takes geo-location data of the start and end points as parameters, and returns the calculated distance for each end point.

Following sections describe the interface methods provided by the Backend Service module interfaces:

IProductService Interface

IProductService interface provides business logic to the Product object. It also implements logic to create product-to-store mapping records.

- getProduct: This method takes UPC and geo-location as parameters which are provided by the mobile client and returns a Product object that matches the given UPC code. The method calls IProductDAO findProduct method to fetch the product record from the database. Null is returned if the product is not found. The method also creates a new product-to-store mapping task and submits it to the ExecuterService to asynchronously process the task in the ThreadPool. The task includes calling LabelAPIClient and GoogleAPIClient to fetch product and store information.
- findProducts: This method takes a list of allergens as a parameter and returns a list of Product objects that do not contain any of the allergens in the parameter list. The method calls IProductDAO getProducts method to search these products.
- findProductByName: This method takes productName as a parameter and returns a list of Product objects whose name, brand or description contains the given name. The method calls IProductDAO findProductByName method to find the products.

IStoreService Interface

IStoreService interface provides the business logic to the Store object. It also implements logic to calculate store distances.

- findProductStores: This method takes a Product object as a parameter and returns a list of Store objects that carry the given Product. It calls IStoreDAO findProductStores method to do the search.
- findLocationDistance: This method takes user's current geo-location data and a list of Stores as parameters to calculate the distance from user's location to each of the given store. The method returns a Map containing the Stores and their calculated distances sorted by the ascending order. It calls GoogleAPIClient to do the calculation.

IUserService Interface

IUserService interface provides the business logic to the User object. It implements logic to create, fetch, update and validate users.

- createUser: This method takes a User object as a parameter and creates a new User document in the database using IUserDAO createUser method. The passwords are hashed with MD5 algorithm before storing it to the database.
- validatePassword: This method takes a plain text password and a User object as parameters to validate the user. The method calls IUserDAO getUser method to fetch the user that matches the User object and returns true if the given password matches the user's password.

- findUser: This method takes a username as a parameter and returns the User object that matches the username. It calls IUserDAO getUser method to fetch the user record.
- updateUser: This method takes a User object as a parameter that contains the updated user information and updates it in the database. It calls IUserDAO updateUser method to do the update and returns true if the update succeeds.

Data Access Service Module

Data Access Service Module implements data service interfaces for the Business Service Module. Continuing with the pattern of separating interfaces by object type, it implements separate interfaces for the Product, Store and User objects. The UML class diagram in Figure 15 gives an overview of the Data Access Service Module implementation.

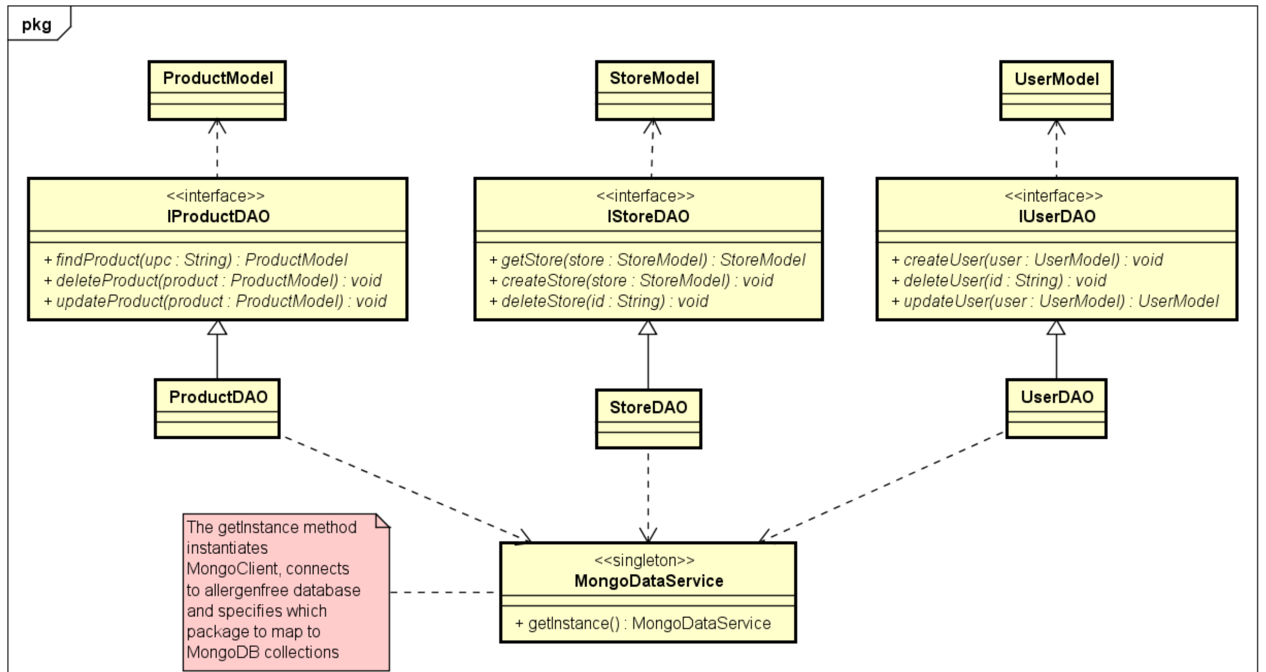

Figure 12. The UML class diagram of the Data Access Module

The MongoDataService.java Singleton (*Singleton Pattern*) class implements a facade to Morphia. Implementing MongoDataService as a Singleton class ensures that there is only one instance of the MongoClient in the system. Morphia is configured to connect to 'allergenfree' database that is created in the setup phase as described in Chapter 8. All the data model classes are placed in 'com.app.mongo.data.model' package whose classes are mapped to the MongoDB documents. The data models that currently exist in the package are the ProductModel, the StoreModel and the UserModel.

The MongoDB Morphia documentation ("Morphia Documentation") provides a comprehensive guide to all the available mapping properties. The existing data models use only a handful of these properties which are defined as annotations in the model

classes. The @Entity annotation maps the model to the MongoDB Collection. The @Id defines the unique identifier for the model, and the @Indexed annotation defines which properties are indexed, and the @IndexOptions annotation defines which properties have unique constraints. Morphia requires annotations only on the properties that have special constraint needs; the rest of the properties are automatically mapped to the document attributes in the Collection that correspond to their names.

The ProductModel object is mapped to the Product Collection. It has properties as shown in Figure 12. Each Product is uniquely identified by its UPC code. The unique constraint defined in model ensures its uniqueness in the database. The 'noAllergen' property lists all the allergens the product is free of. The 'store' property contains reference Ids of the Store Collections that carry the product.

```
> db.Product.find({"upc": "661799264327"}).pretty()
{
        "_id" : ObjectId("584c8e3be882401c9a094dc9"),
        "className" : "com.app.mongo.data.model.ProductModel",
        "upc" : "661799264327",
        "name" : "GOBEYOND BEYOND ICECREAM",
        "brandName" : "GOBEYOND",
        "description" : "BEYOND ICECREAM",
        "noAllergen" : [
                "shellfish",
                "egg",
                "fish",
                "peanut",
                "tree nuts",
                "sesame seeds",
                "gluten",
                "wheat",
                "coconut"
        ],
        "stores" : [
                ObjectId("584c89d6e882401c9a094dbc"),
                ObjectId("584c8ab5e882401c9a094dc2"),
                ObjectId("584c8a1ae882401c9a094dbe"),
                ObjectId("584c8a2be882401c9a094dbf"),
                ObjectId("584c8bace882401c9a094dc6")
        ],
        "creationDate" : ISODate("2016-12-10T23:22:35.603Z"),
        "modificationDate" : ISODate("2016-12-10T23:22:35.603Z")
}
```

Figure 13. The properties of the Product object

The StoreModel object is mapped to the Store Collection and has properties as shown in Figure 13. Stores are uniquely identified not just by the name but also by the address. Two stores with the same name but located at different addresses are considered unique. Along with the name and address, the Store Collection also has geoLocation and googlePlaceId properties. The 'geoLocation' property identifies store's geographical location data. It is provided by mobile client when user scans a product and is used to show the store in the Google Map. The 'googlePlaceId' is provided by the Google Services API and uniquely identifies an address.

```
> db.Store.findOne()
{
        "_id" : ObjectId("584c6dcbe8824017aa50d8c2"),
        "className" : "com.app.mongo.data.model.StoreModel",
        "streetNumber" : 471,
        "streetName" : "Salem Street",
        "city" : "Medford",
        "state" : "Massachusetts",
        "stateAbbr" : "MA",
        "storeName" : "Stop & Shop",
        "country" : "United States",
        "zipcode" : "02155",
        "creationDate" : ISODate("2016-12-10T21:04:11.100Z"),
        "modificationDate" : ISODate("2016-12-10T21:04:11.100Z"),
        "geoLocation" : "42.423044, -71.089060",
        "googlePlaceId" : "ChIJcdRHI05x44kR2MqPTo_iKkE"
}
```

Figure 14. The properties of the Store object

The UserModel object is mapped to the User Collection. It has properties as shown in Figure 14. Each user is uniquely identified by the username property, which corresponds to a person's email. The 'allergy' property lists all the allergens that the user wants to avoid.

```
> db.User.findOne()
{
        "_id" : ObjectId("584d5a27e882401c9a094dca"),
        "className" : "com.app.mongo.data.model.UserModel",
        "userName" : "ngurung@gmail.com",
        "password" : "e10adc3949ba59abbe56e057f20f883e",
        "firstName" : "Navayush",
        "lastName" : "Gurung",
        "totalContribution" : 0,
        "allergy" : [
                "peanut",
                "milk"
        ]
}
```

Figure 15. The properties of the User object

The following sections describe the interface methods implemented by the Data Access Service module.

IProductDAO Interface

The IProductDAO interface provides CRUD operations to the Product Collection.

- findProduct: This method takes a UPC code as a parameter and returns a ProductModel object that matches the given UPC code.
- findProductByName: This method takes a product name as a parameter and returns a list of ProductModel objects whose names match the given name.
- getProducts: This method takes a list of allergens as a parameter and returns a list of ProductModel objects that do not contain allergens listed in the parameter.
- createProduct: This method takes a ProductModel object and a StoreModel object as parameters and creates a new Product Document that maps to the given Store Collection.
- addStoreToProduct: This method takes a ProductModel object and a StoreModel object as parameters and creates a mapping between them.
- deleteProduct: This method takes a Product Id as a parameter and deletes the Product Document from the database that matches the given id.
- updateProduct: This method takes a ProductModel object that contains the updated information as a parameter and updates the existing Product Document with the new information.

IStoreDAO Interface

The IStoreDAO interface provides CRUD operations to the Store Collection.

- getStore: This method takes a StoreModel object that contains store address as a parameter and returns a StoreModel object that matches the address.
- createStore: This method takes a StoreModel object as a parameter and creates a new Store Document in the Store Collection.
- findProductStores: This method takes a ProductModel object as a parameter and returns a list of StoreModel objects that carry the given Products.

IUserDAO Interface

The IUserDAO interface provides CRUD operations to the User Collection.

- createUser: This method takes a UserModel object as a parameter and creates a new User Document in the User Collection.
- getUser: This method takes a username as a parameter and returns a UserModel object that matches the given username.
- updateUser: This method takes a UserModel object as a parameter and updates the existing User Document with the updated data provided in the UserModel parameter.

Mobile Client

The Mobile Client implements the User Interface using the Ionic directives (*Ionic Documentation*) and the AngularJS components. The main layout of the UI in the index.html is defined by the <ion-nav-view> directive, which works with the AngularUI Router (*AngularUI Router*) to render pages based on the state of the application. The

individual pages are defined as templates that get injected in the <ion-nav-view> section of the index.html when the application changes its state. For example, when the user accesses the app for the first time, the application transitions to the default state, which is defined as the 'login' state, causing the AngularUI Router to inject the Login Page template in the <ion-nav-view> section to show the Login Page on the UI.

```
<body ng-app="app">
<div style="">
  <div style="">
    <ion-nav-view class="calm">
     <ion-nav-bar class="bar-stable">

      </ion-nav-bar>
      <!-- Main app view -->
    </ion-nav-view>
    </div>
</div>
</body>
```

Figure 16. The <ion-nav-view> directive in index.html

The application states are defined in the AngularUI Router JavaScript file. Each application state is defined by a State name, a URL, a Template name and a Controller name. The State name uniquely identifies a state of the application. The URL defines the path of the state. The Template name identifies the HTML template to inject when the application transitions to this state. And, the Controller name identifies the AngularJS Controller to initialize when the transition occurs. Figure 17 shows some examples of the states defined in the application.

```
$stateProvider

  .state('allergies', {
  url: '/allergies',
  templateUrl: 'templates/allergies.html',
  controller: 'allergiesController'
})

.state('store', {
  url: '/store',
  templateUrl: 'templates/store.html',
  controller: 'storeController'
})

.state('result', {
  url: '/result/',
  templateUrl: 'templates/result.html',
  controller: 'resultController'
})
```

Figure 17. The AngularUI Router state definitions

The AngularJS Controllers are the components that manage the data and provide functions to the Angular applications. Each template page is associated with its own Controller. These controllers use various AngularJS services to provide functionalities to the page they control. The $scope, the $rootScope and the $http are some of the core AngularJS Services used by the implemented Controllers. The $scope is a service that binds the view to the controller. Each view has its own instance of the $scope service. The $rootScope on the other hand is a global object with one instance that is available to the entire application. The $rootScope is used in the app to set the system level property such as the Backend System URL. The $http service provides framework to make web service calls to the Backend System. These services are injected to the Controllers when they get instantiated.

The Mobile Client also uses custom AngularJS services (*AngularJS Services*). These services are used by the Controllers in the same way as the core services and can be used to provide a centralized location for the data and the functions that need to be shared across the app. The Eatable app implements one such service named ‘dataService’ to centralize the access to the common product and store data that are shared by different views.

User Interface

This section describes the content, navigation and functionalities of the UI pages.

Register Page

The Register Page provides user registration functionality to the Mobile Client as specified in the User Management Requirement, Chapter 3. The user navigates to the Register Page by clicking the ‘create an account’ link on the Login Page. The Register Page has two panels. The top panel provides textfields for users to enter their First Name, Last Name, Username and Password. All the textfields are required fields. The username must be a valid email address. The bottom panel provides radio button sliders for 6 supported allergens from where users can select the allergens they want to avoid. The ‘Sign Up’ button submits the data. Form validations ensure all the values are correctly filled in, and appropriate error messages are displayed on top of the view if any validation fails. If user successfully signs up, the user is directed to the Home Page.

Figure 18. The Register Page

The register page is bound to the ‘registerController.’ ‘Sign Up’ button, on submit, invokes ‘register’ method in the controller which makes HTTP POST request to the UserController interface in the Backend System to post the registration data. If the registration succeeds, the newly registered user object is returned, which is set in the loggedInUser variable of the $rootScope service so that the user object is accessible to the other parts of the app, and the user is redirected to the Home Page. If the registration fails, the user remains on the Register Page with an error message displayed on the top of the page as shown in Figure 19.

EatAble

All fields are required
Username must be a valid email address

First Name
Last Name
Userame(email)
Password

Select Allergens

Peanut
Milk
Egg
Soy
Shellfish
Almond

Sign up

Figure 19. The Register Page with validation errors

Figure 17 Sequence Diagram shows the flow of calls that occur in the application when a new user is registered in the system.

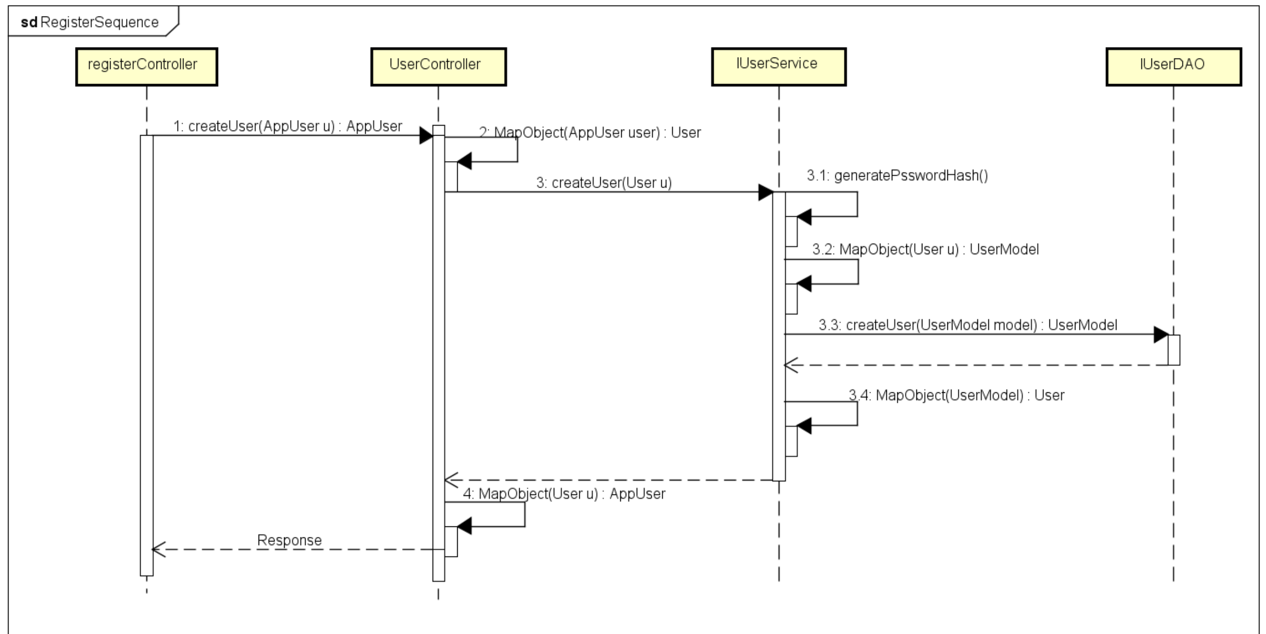

Figure 20. The Sequence Diagram of the registration process

Home Page

The Home Page serves as a gateway to all the features specified in Chapter 3. It has three buttons in the main panel and one button in the footer. The three buttons on the main panel are: ‘Is Eatable?’, ‘Locate Eatable’ and ‘Eatable Foods’. The Settings button is placed in the footer. ‘Is Eatable?’ button invokes product verification functionality. ‘Locate Eatable’ button takes users to the Locate Page from where they can search for stores that carry products they are interested in. ‘Eatable Foods’ button takes users to the Stores Page from where they can search and filter stores that carry products safe for them based on their allergies. ‘Settings’ button takes users to the Settings Page from where they can edit their profile information.

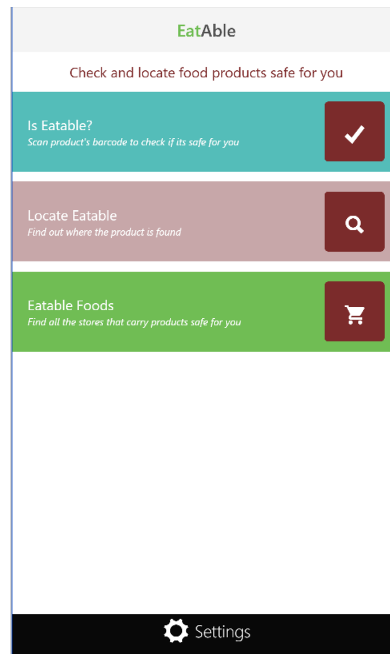

Figure 21. The Home Page

The Home Page is bound to the 'homeController'. Clicking on 'Is Eatable?' button initiates BarcodeScanner plugin which starts the phone's camera allowing users to take product's barcode picture. The BarcodeScanner provides the UPC code of the scanned product to the application. The controller also fetches geo-location data of the current user location using GeoLocation plugin and calls the ProductController interface in the Backend System to validate the product for allergen safety. This is when the Backend System also creates product to store mapping records as described in Product Store Mapping, Chapter 6.

Figure 19 Sequence Diagram shows the flow of events that occur when user clicks 'Is Eatable?' button to check product's allergen safety.

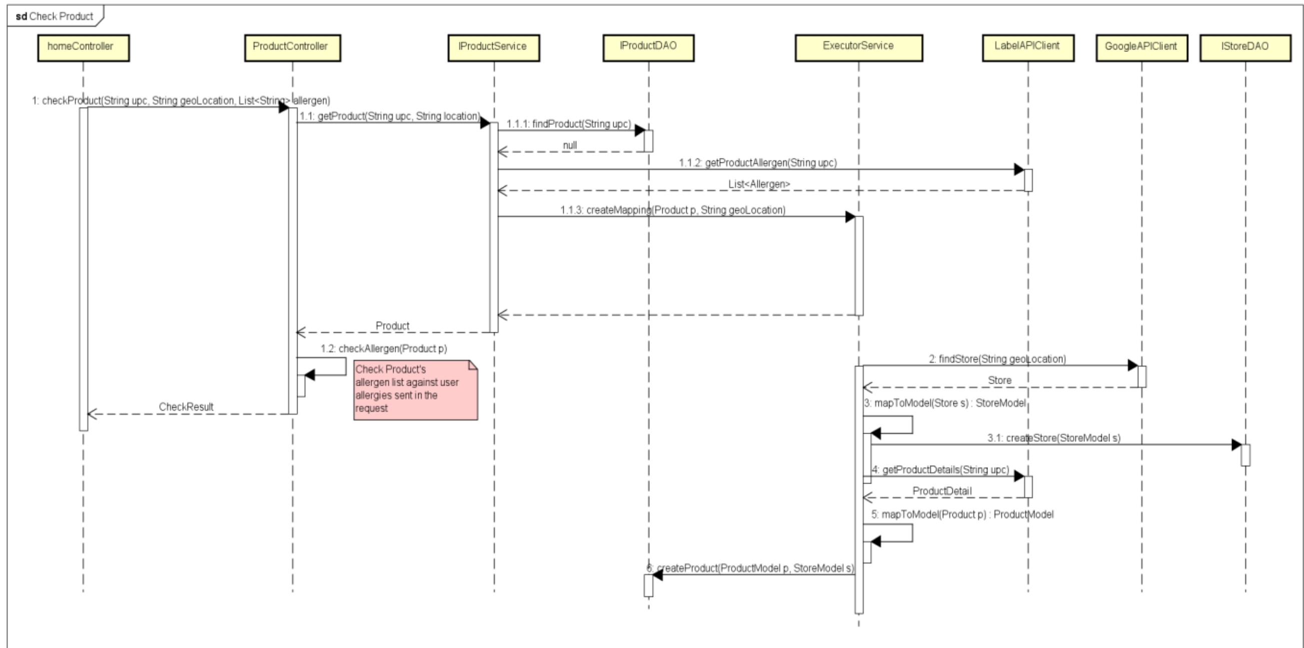

Figure 22. The Sequence Diagram of the product verification process

Locate Page

The Locate Page provides Locate Product functionality specified in Chapter 3. The page is accessed from the ‘Locate Eatable’ button on the Home Page and allows users to search for stores that carry products they are interested in. The page provides one textfield that takes a search string. The search string can be any term in the product’s name, brand or description. The search starts once the user hits the search button or the enter key. The search is case insensitive. The result of the search is displayed on the bottom panel.

The result shows a list of store locations in a tabular format ordered by the distance from the user’s current location in ascending order. The distance from the user’s current location to each of the location is displayed in a circular button on the right hand

side of the result row. The address of the store along with the product information is displayed on the left. The distance button is clickable and takes the user to a page that shows the location of the store on the Google Map.

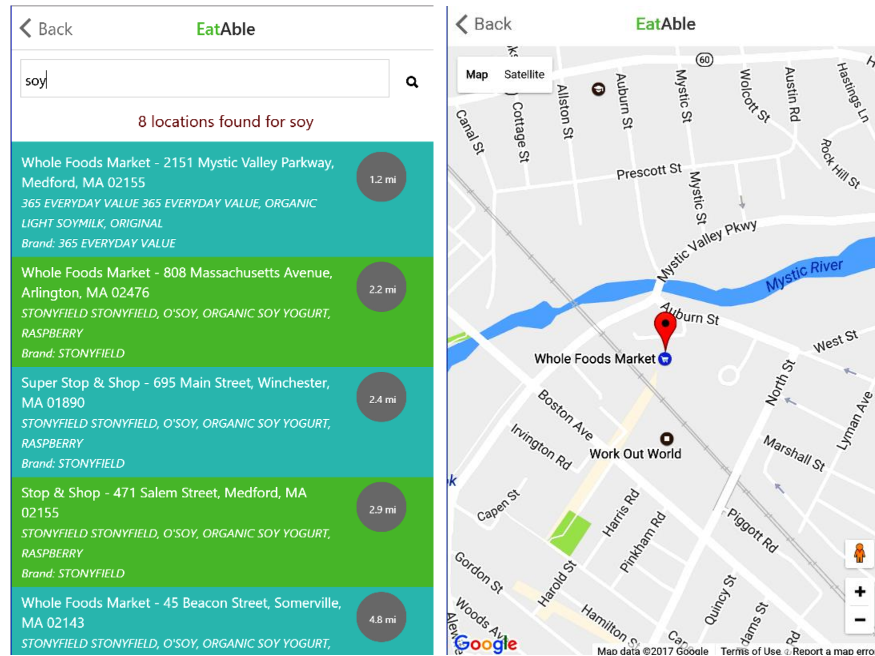

Figure 23. The Locate Eatable Page

The page is bound to the 'locateController'. On clicking the search button or the enter key, the Controller fetches user's current location using GeoLocation plugin and makes a HTTP GET request to the ProductController interface in the Backend System to initiate the search. The location on the Google Map is displayed using Google Map API JavaScript.

Eatable Foods Page

The Eatable Foods page is accessed from the 'Eatable Foods' button on the Home Page. The page shows all the store locations in the Eatable database that carry products safe for the user. This page implements the search and filter requirement specified in the Store Search Requirement, Chapter 3. The page shows store locations grouped by the store brand as shown in Figure 21, along with the total number of locations and unique products. This provides users with an aggregated view of all the store brands and how many unique products each brand carries along with the total number of the brand's available locations.

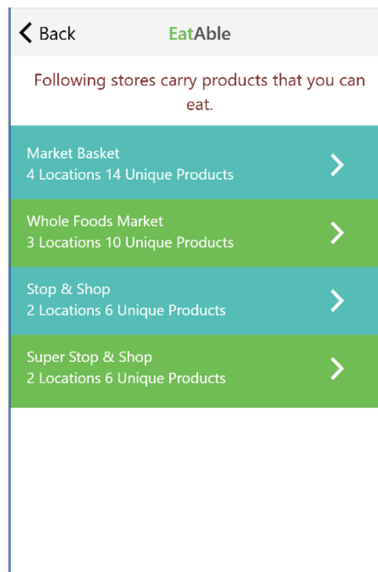

Figure 24. The Eatable Foods Page

The arrow button on the right of the result row takes users to a sub-page that shows store locations of the selected store brand. This sub-page shows addresses of the

store locations in a tabular format along with the total number of unique products found at that location in a rounded button as shown in Figure 22.

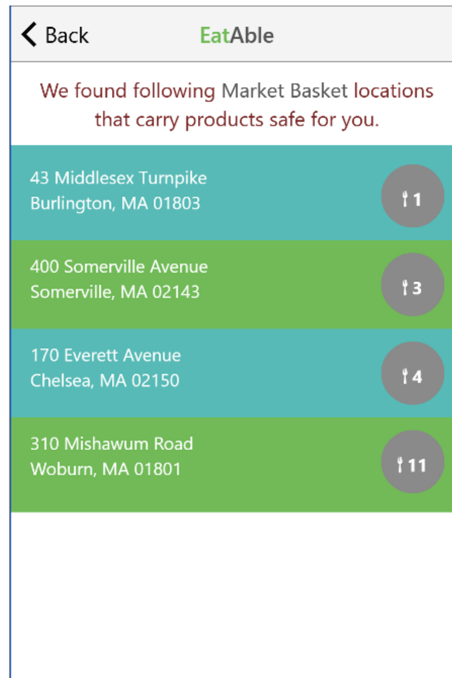

Figure 25. The Store Locations Page

The product number button on the Store Location sub-page takes users to another sub-page that shows a list of all the unique products found at that particular location. The product list is displayed in a tabular format, and it shows a product's name and description.

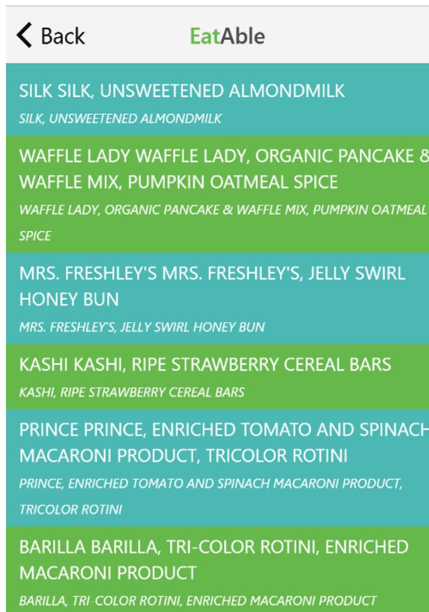

Figure 26. The Eatable Products Page

The Eatable Foods Page uses the ‘dataService’ to provide store and product mapping data shared by the different views of the page. Each sub-page is bound to different controllers that manage data for a particular view but they all depend on the same data service. The ProductController in the Backend System provides ‘findProducts’ service that returns all the necessary mapping data in one method call as an aggregated dataset so the ‘dataService’ makes only one HTTP GET call to the Backend System to show all the views.

Login Page

The Login Page provides functionality for users to login to the system. The page contains Username and Password textfields and a registration link. The Login button initiates the authentication process. Successful authentication redirects users to the Home

Page runs the app in the authenticated session. If the authentication fails, the users remain on the Login Page with an error message displayed on the top of the page.

Figure 27. The Login Page

The page is bound to the ‘loginController.’ Clicking on the Login button invokes ‘login’ function in the controller that calls the UserController interface in the Backend System to validate the user.

Settings Page

The Settings Page allows users to edit their profile information. The page is accessed through the Settings button on the footer of the Home Page. The page allows users to edit their First Name, Last Name and allergen selection. The panel on the top provides pre-filled editable First Name and Last Name textfields, and the panel on the bottom provides radio button controls to modify the allergen selection. The Update

button submits the changes to the server. If the update succeeds, the authenticated user in $rootScope is updated with the new user information and the user is directed back to the Home Page. If the update fails, an appropriate error message is displayed on the top of the page. The page also provides a ‘Sign Out’ button on the top left corner that lets users log out of the system. When user logs out, the user object in the $rootScope is set to null and the authenticated session of the app ends.

Figure 28. The Settings Page

The Settings Page is bound to the ‘settingsController’. The Update button, on submit, invokes the ‘update’ function in the controller that calls the UserController interface in the Backend System to update the user.

Chapter 8 Development and Test Environment Setup

The first step towards implementing a software project is to setup development and test environments where one can build and test the application. This chapter describes the software components and processes involved in setting up these environments for implementing the Eatable project.

Mobile Client

The mobile development environment consists of the Ionic framework, the WebStorm IDE (*WebStorm*), the Apache Cordova, the Android SDK and other dependent software libraries. Ionic Framework's guide for Windows machine ("Installing Ionic and its Dependencies") provides detailed step by step instruction on how to install and setup Ionic to start building an app. The Node.js library is required for the installation. It is also used internally by the Ionic framework. The latest version of the Apache Cordova was installed to provide cross-platform functionality to Ionic, and the Android platform tools were installed to build the mobile client as an android app. The Cordova guide ("Android Platform Guide") provides guidance on all these required Android tools and libraries needed to build an android app, which includes the Cordova-Android, the Android Stand-alone SDK and the Android SDK packages. Java 8 was installed, and JAVA_HOME and ANDROID_HOME path variables were setup pointing to the Java JDK and the Android SDK home directories respectively.

The Ionic provides 'start' tool that creates a ready-to-run boiler-plate ionic project to help people get started with it. This startup project is pre-configured with all the dependencies in place along with a project structure that is extensible, so it was very easy to get started with the mobile app project.

Local testing of the app is done in Emulator which provides test runtime environment of a mobile device. Android SDK installation comes with AVD Manager to create emulators for various android devices. Figure 26 shows the settings of the Nexus 5 Emulator configured to test the mobile app.

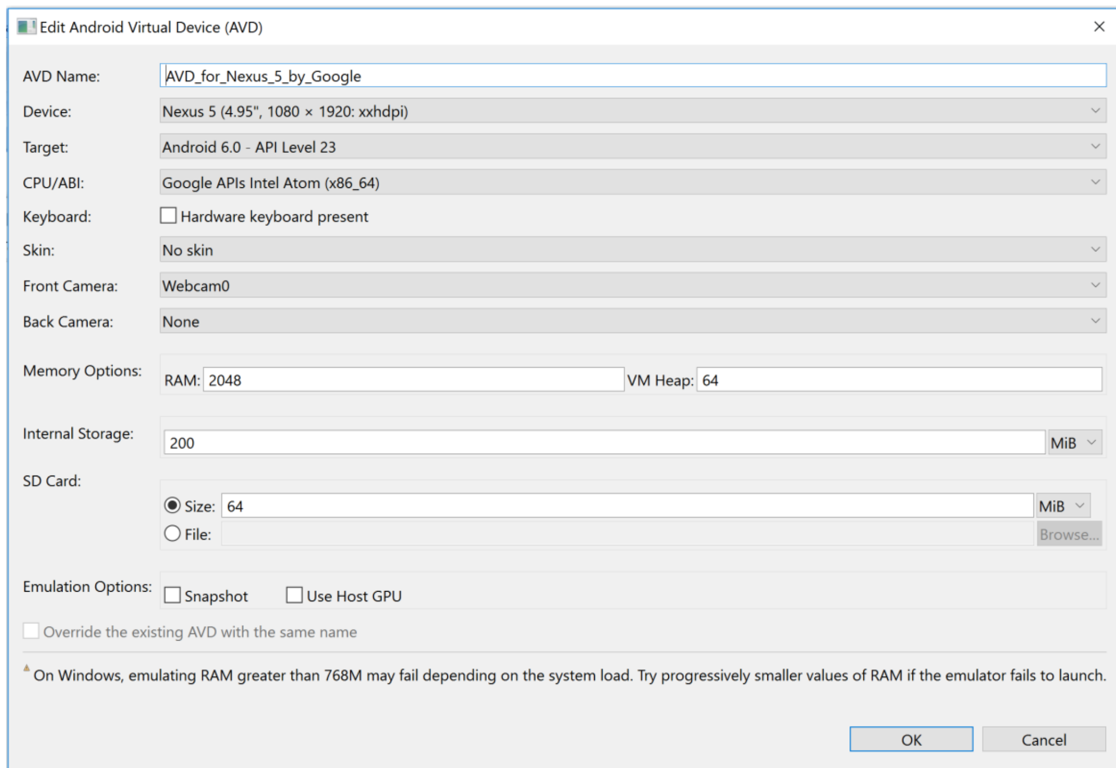

Figure 29. The configuration of Nexus 5 Emulator

Loading the app on the emulator for testing is also fairly simple since Ionic provides command line tools to easily build and deploy the apps on testing devices like emulators. The ‘ionic run’ command is used to build and deploy the app. The command takes app type and target device as parameters that identify the platform of the app and the name of the testing device. Figure 27 shows the final version of the app running on the Nexus 5 Emulator.

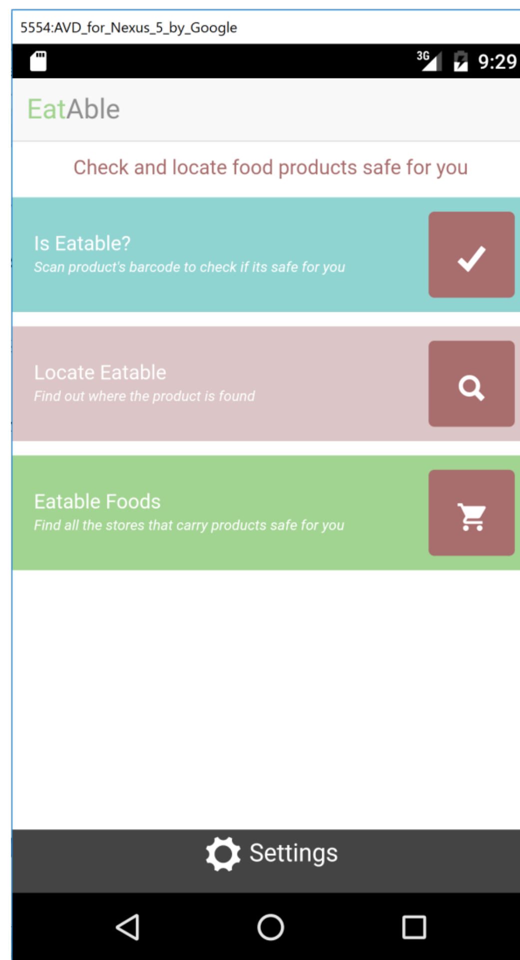

Figure 30. The Eatable application on the emulator

Backend System

The Backend System development environment consists of the Tomcat 8 Wb server (*Apache Tomcat*), the ANT library (*The Apache Ant Project*), Java 8, MongoDB and Eclipse IDE (http://www.eclipse.org/neon/). The CATALINA_HOME, JAVA_HOME and ANT_HOME path variables are required to make the libraries available to the build system. These variables point to Tomcat 8, Ant and Java installation directories respectively. Eclipse Neon is used as an Integrated Development Environment (IDE). The Tomcat installation provides startup script that can be used to test the setup. The MongoDB community site (*Install MongoDB Community Edition on Windows*) provides the step by step instruction on installing MongoDB. The installation comes with MongoDB shell to execute the queries. The database for the Eatable application is named as ‘allergenfree’.

The Backend System is made up of three maven (*Maven – Eclipse IDE*) java projects to represent each of the backend modules. The AppBackend project is a web application project that serves as a REST interface to the Backend System. The BackendData and the BackendService are regular java projects that implements persistent layer and business layer respectively. The dependency on the Dozer library provides functionality to map java objects and the log4j library provides logging framework to the projects. The Jersey and its dependent libraries as specified in Jersey documentation (*Jersey*) provide web service APIs to the AppBackend project. The Jersey servlet and the mappings are configured in web.xml to handle the web requests. The BackendData project depends on the Morphia library that is used to map the java objects to the

MongoDB documents. Jersey client implementation libraries provide APIs to the BackendService project to implement client interface to the external web services the project depends on. Standalone test cases were written to test the individual modules and Advanced REST client library (https://advancedrestclient.com/) provided support for end-to-end testing.

Chapter 9 Deployment and Evaluation

The Backend System is deployed in Amazon AWS in EC2 instance, and the mobile app is downloadable as an Android Application Package (APK) file from https://www.dropbox.com/s/lbt6qszll6g4v70/android-debug.apk?dl=0/. The system runs in Basic 64-bit Amazon Linux AMI operating system. The AMI is configured with all the necessary setup described in Amazon AWS Guide (*Setting Up with Amazon EC2*). Java 8, Tomcat8 and MongoDB are installed on the AMI. MongoDB documentation (*Manually Deploy MongoDB on EC2*) provides a comprehensive guide on installing MongoDB in EC2 instance. There are configurations that are not required in local installation of the database but are required in EC2. Database named 'allergenfree' is created in MongoDB to store data for Eatable application. The 'war' deployment archive file is generated from Eclipse using its Export functionality and deployed to Tomcat 8 deployment folder.

The mobile app was downloaded on the Google Pixel phone and tested at various stores in the different cities. It was tested across multiple store brands. There are records from the Whole Foods Market, Market Basket and Stop and Shop in the database. The test cities include Medford, Cambridge, Somerville, Chelsea, Woburn and Winchester. Some bugs encountered in the initial testing are already addressed. For most part, the app performed quite well. The results were quick and mapping records got correctly generated. Among the products scanned, the system was unable to find about 30% of

them; these products didn’t exist in the Label API database. This is one of the risks the system has as it depends on the external service to provide the product information, so if the products are missing in the Label API database, the app will not be able to verify them. The Label API Company claims to contain 80% of the products found in US market so this is probably acceptable, and is the best option that exists today.

Based on the initial user feedback, there are some feature recommendations that the app could benefit from. Navigating to store location directly from the app is one of them; currently users can only see the locations on the map but cannot start navigation from there. Another recommendation was to integrate map and distance feature in store search functionality as well; currently it is provided only in the locate feature. There were questions about the accuracy of the store result. How does the app know when a store stops carrying a product? Currently, there is no way for the system to know this, but it seems this issue needs to be addressed to make the app successful because the ability to correctly locate products is the foundation of the app, and the user needs to feel confident that when the app says a product is found in a particular store that it will be there.

Chapter 10 Summary and Conclusion

In this thesis project I have developed a scalable and a flexible crowd sourced application named 'Eatable' to help people with food allergies easily locate foods that are safe for them. I have built a unique system using scan and location data from the users to solve a problem that no other currently available app in the market has solved. The system is built with extensibility and scalability in mind both from the design aspects and the technological choices. The decision to implement the app as hybrid was the first step towards achieving this goal. This allows the app to be deployed to different mobile platforms without code rewrite. The separation of the Mobile Client from the Backend System through RESTful interface, and limiting the client as an UI façade while making the Backend implement all the business logic makes it possible for any front end to be integrated with the Eatable system. Being a crowd sourced application, expected to evolve, it is anticipated that the nature and the volume of its data will change with its growth; it was imperative to choose a database that could adapt to such changes. MongoDB, a highly performant, flexible and scalable database fit the bill. The components within the system are also designed to be loosely coupled. The Backend System is divided into independent modules through interfaces, and their data objects are contained within them.

While the app is complete in its current form and provides the basic mapping and locating functionalities, the possibility to extend the application is endless. Although the allergens currently supported are the most common culprits, the list is very limited. This list can be extended not only to support more allergens but also to support other food sensitivities (Gluten) and choices (Paleo diet). Because of the technical choices employed in the application, such extensions can be easily implemented.

Based on the initial feedback, there are features I would like to implement before officially deploying it. Supporting more allergens and providing iOS version of the app are the top ones in my list. Another improvement that can be made on the application is to further integrate the capabilities of the Google Map. In addition to showing the store location on the map, the app could also enable users to navigate to the location directly from the app. One limitation of the application in its present form, that needs to be addressed, is its inability to determine when stores stop carrying products that they once carried or when they run out of stock. Because this is an application whose main functionality is to locate products, I think it's imperative that when the app says a product is found in a store that it is there. However, this is not an easy problem to solve unless the stores provide access to their inventories through API, which is not feasible at this point. Tapping into suppliers' inventory management systems is another alternative, but more research needs to be done in this area to learn if this is feasible. One way to mitigate this problem is to acquire this information from its users. The app can provide easy-to-use functionality from where the users can flag missing products. However, this solution depends on user's feedback, which is not guaranteed. The plan is to continue with the

development and research of this app to turn it from a Minimal Viable Product to a Real Valuable Product that can be used to help improve quality of life for its users by making it easier to identify and locate foods that are safe to eat.

References

AngularJS. (n.d). Retrieved from https://angularjs.org/

AngularUI Router. (n.d). Retrieved from https://github.com/angular-ui/ui-router/

AngularJS Services. (n.d). Retrieved from https://docs.angularjs.org/guide/services/

The Apache Ant Project. (n.d). Retrieved from http://ant.apache.org/bindownload.cgi

Apache Tomcat. (n.d). Retrieved from http://tomcat.apache.org/download-80.cgi

Android Platform Guide. (n.d). Retrieved from http://cordova.apache.org/docs/en/latest/guide/platforms/android/index.html

Appel, Rachel. (2014, November). Modern Apps : Mobile Web Sites vs. Native Apps vs. Hybrid Apps. Retrieved from https://msdn.microsoft.com/en-us/magazine/dn818502.aspx/

Bristowe John. (2015, March). What is a Hybrid Mobile App?. Retrieved from http://developer.telerik.com/featured/what-is-a-hybrid-mobile-app/

$cordovaBarcodeScanner. (n.d). Retrieved from http://ngcordova.com/docs/plugins/barcodeScanner/

$cordovaGeolocation. (n.d). Retrieved from http://ngcordova.com/docs/plugins/geolocation/

Elastic Compute Cloud (EC2) – Cloud Server and Hosting – AWS. (n.d). Retrieved from https://aws.amazon.com/ec2/

Facts and Statistics. (n.d.). Retrieved from https://www.foodallergy.org/facts-and-stats

Getting Started with Ionic (n.d). Retrieved from http://ionicframework.com/getting-started/

Google Maps APIs Web Services. (n.d). Retrieved from https://developers.google.com/maps/web-services/overview/

Installing Ionic and its Dependencies. (n.d). Retrieved from

https://ionicframework.com/docs/guide/installation.html

Install MongoDB Community Edition on Windows. (n.d). Retrieved from https://docs.mongodb.com/manual/tutorial/install-mongodb-on-windows/

Ionic Documentation. (n.d). Retrieved from http://ionicframework.com/docs/v1/

Jersey. (n.d). Retrieved from https://jersey.java.net/

LabelAPI. (n.d). Retrieved from http://developer.foodessentials.com/api/

Manually Deploy MongoDB on EC2. (n.d). Retrieved from https://docs.mongodb.com/ecosystem/platforms/amazon-ec2/#deploy-mongodb-ec2

Maven – Eclipse IDE. (n.d). Retrieved from https://www.tutorialspoint.com/maven/maven_eclispe_ide.htm

MongoDB 3.4: Your Database Evolved | MongoDB. (n.d). Retrieved from https://www.mongodb.com/mongodb-3.4/

Morphia. (n.d). Retrieved from http://mongodb.github.io/morphia/

Morphia Documentation. (n.d). Retrieved from http://mongodb.github.io/morphia/1.3/getting-started/quick-tour/

ngCordova - Simple extensions for common Cordova Plugins - by the Ionic Team – by the Ionic Team. (n.d). Retrieved from http://ngcordova.com/

Setting Up with Amazon EC2. (n.d). Retrieved from http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/get-set-up-for-amazon-ec2.html

Singleton Pattern. (n.d). Retrieved from http://www.oodesign.com/singleton-pattern.html/

WebStorm. (n.d). Retrieved from https://www.jetbrains.com/webstorm