# Deliverable 2

Software Architecture
& Design written report

# Table of Contents

# Overview

Hare is an app that allows a user to take a picture (or upload one) of a packaged products' ingredients label and obtain useful insights and information into the contents of the item. It targets users who want or need to be particularly aware of what's inside products around them (for example, those with allergies). As the general market applies to many if not all consumers, we aim for the application to be very user friendly and provide a great user experience. In Deliverable 1, we covered some of the ways that a user might see and use this application, and in this deliverable, we will cover how the application will be made.

## Revision History

| | | |
|---|---|---|
| 27/03/2019 | 0.5 | Draft |
| 29/03/2019 | 1.0 | Submission |

## Group Members

| | |
|---|---|
| z5161502 | Michael Lloyd |
| z5160635 | Negar Bolourchi |
| z5182793 | Ravija Maheshwari |
| Z5158357 | Julie Duan |
| z5207123 | Jack Callander |

# Software Architecture

## 01 Architectural Requirements

Our web application is intended to be a small, yet very user friendly. From the perspective of software architecture, we only require a decoupled Back-End, Database and mobile compatible Front-End. We expect that this will require the view that the user will see to be largely just a rendering component, and for a great deal of logic to happen behind the scenes on a entry-level developer friendly language.

The application must work on all common web browsers (Chrome, Firefox, Internet Explorer), android phones and iOS. We expect that the development for multiple mediums will guarantee that the logic controlling the front-end will be abstracted and decoupled from the rest of the back-end, while a good deal of the back-end will be dedicated for modelling what the user will see. This would be so that each front-end interface would only need to render the model in a tailored way, instead of piecing together multiple suggestions from the back-end of the stack.

Another requirement lies in development, where it is important for each member of the development team to be able to contribute to the project without large barriers of entry to the framework, language or libraries involved. For this reason, we consider the project's development limited to frameworks including Python, JavaScript, and HTML/CSS processors.

## 02 External Data Sources

Our application requires a number of external sources of information for the application to work without large development on our part. Namely, we need to be able to recognize ingredients from an image, and then get information about those ingredients to pass back to the user. These sources are:

**OpenFDA's Database API** [1]
The OpenFDA database API is an easy access public database with information ranging from drug datasets to packaged product codes and label information. The database is maintained by the Food and Drug Administration (FDA) in the United States which has a credible reputation in regulating product packaging and labels. For this reason, we rely on the dataset of ingredients and substances listed to be possible in packaged foods, interacting with it inside of our own database, and occasionally re-referencing the OpenFDA API.

This source is particularly important, because it provides us with a primary reference point for how to interpret the ingredients on food labels, as well as providing us we very accessible lists of the categories and types that may exist.

**DrugBank.ca Database API[2]**
The Canada-based DrugBank API was used to complete our ingredient and substance data-sets, as this particular database offers several query terms that OpenFDA does not. Notably, there is a large number of synonyms and descriptions for substances contained in the OpenFDA database that OpenFDA does not provide. This means that the Canadian DrugBank API is essential for completing substance profiles that are sent to the user.

**MediaWiki Official Wikipedia API[3]**
We have included the official Wikipedia API as a supporting source of information for common ingredients and substances in packaged products. We also intend to reference where we get information directly to the user, so that they can follow a link to see more themselves. This should greatly improve the user experience as it allows us to cover all the most common substances they are likely to find in products with well written descriptions and outgoing links.

**Google Vision Optical Character Recognition API[4]**
Google Vision offers an Optical Character Recognition (OCR) API that can be used to return interpreted text in an image through only a REST API. It has access to several other Google Cloud services (such as Google AutoML) that allow for the categorization and parsing of the information once it is read, and returns a JSON file as a response to the caller with information ranging from different lines read and the x, and y coordinates for which it read them.

This offers some great advantages to our project over directly running an OCR service inside of our back-end or use another service as this particular service is well maintained, optimized and efficient. We have tested latency, character accuracy, and a variety of different lighting conditions with this API and concluded it to be superior to many alternatives available.

**Google Languages Translation API[5]**
The Translation API provides a simple programmatic interface for translating an arbitrary string into any supported language on Google. Language detection is also available in cases where the source language is unknown. We observe extremely low latency when using this API, like with Google Vision's OCR, and consider it necessary to meet one of our primary problem statements – allowing people to see the ingredients in a product when the packaging is in a foreign language.

**Links**
**[1]** https://open.fda.gov/
**[2]** https://www.drugbank.ca/
**[3]** https://en.wikipedia.org/w/api.php
**[4]** https://cloud.google.com/vision/docs/ocr
**[5]** https://cloud.google.com/translate/docs/

# 03 Architecture Components

In this section, we review and discuss the requirements that are present for each ‘section’ of our web application, and how this may influence the desired architecture for it. To meet the needs of development, the group opted for the abbreviated MERN web-stack (MongoDB, Express, React, NodeJS) which offers the advantage of being entirely written in JavaScript, and for being used to develop applications for many devices. This section elaborates
on each component of this stack, and why it has been chosen.

## Front-End Stack

Our front end has a number of requirements we deem desirable for optimal use. Namely: the Front-End needs to support cross-compatibility among devices, from using a computer’s browser to iOS and Android running applications. The Front-End should be modular enough to enable development to happen in an appropriate time-frame, and it should handle some level of programmatic responsiveness.

On the other end, the Front-End does not need to contain any logic involved with OCR, parsing, database handling or making external API requests.

**ReactJS[6]**
React, commonly misinterpreted as a framework, takes many of the roles that a traditional Front-End framework does. It serves as a library for extending upon several of its base class components, and then is often wrapped with a number of bundling and pre-processors for the code it’s written with to provide an actual Frameworks’ features.

A large advantage of React is that it supports programmatic events and responses directly linked to the components placed on the page. This means that routing and event management can be directly inside of the JSX components and does not need to be directed to an events container or dispatcher (used as a personal library for handling the interactions of users). It is also written for JavaScript, which the development team is familiar with – making it attractive as a framework.

As React is completely independent of the source it is served from, and bundles/processes all of the scripts necessary to run the page before hosting, we can completely decouple the Front-End framework from the Back-End framework, and bridge communication between the two with a custom REST API. This results in lower coupling but does not affect the level of cohesion that architecture has during development.

One large disadvantage of React, is the complexity of what React refers to as “State” inside of class-based components. Management of State for functionality like Routing and event responses can become needlessly complex, messy and highly coupled if not managed properly, and hence has the potential to create issues during development. There are many available solutions, namely the use of a global state management package such as “Redux”. We have instead decided to combat this through the use of standardized personal components, and passing states through routing, which is tree-like in structure, allowing for state methods to be passed along it.

**Babel/JSX Pre-processing[7]**
Babel is a pre-processor for JavaScript that allows a variant of code called “JavaScript X” or “JSX” to be written. This comes bundled into the default React standard but is required for writing higher order JavaScript to HTML. It allows components to be directly declared as pseudo-HTML inside of JavaScript, then upon pre-processing compiled to both HTML and JavaScript separately.

**UI element libraries [8]**
Alongside Babel, we used Grommet (version 2) as a library for pre-written JSX components. As Grommet is standardized, offers icon libraries and a variety of maintained and well managed event handling functions, we considered this ideal for building the website. It allows for rapid prototyping, similar to its’ counterparts such as Bootstrap, or Material Design.

# Back-End Stack

**Express Framework for NodeJS**
After decoupling our Front-End from our Back-End, focus can be given on how the logic, data storage, external API access, and view generation should be managed. We have identified that there should be 4 primary sections of this:

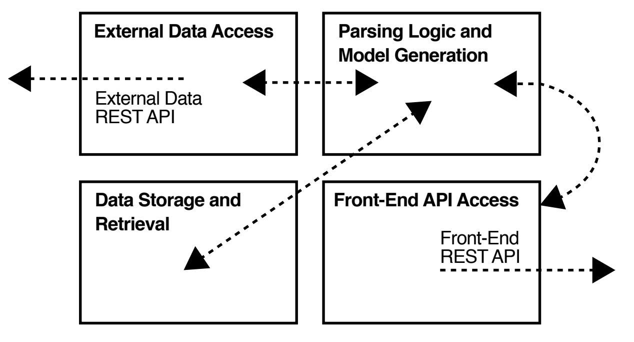


Figure 1
Back-End as four separate components

We expect that the Controller will manage requests from the Front-End and to external API's, and that the Model will collate and generate a page's necessary information container then have the Controller deliver it. The Database should work alongside the Model to collate information that is necessary for the App to run (namely, user profiles and authentication details) as well as assisting information for events on the websites (product pages, ingredient lists and substance details)

**NodeJS[9]**
To assist the development team, and to maintain cohesion on the language-level of the stack, we opted to use JavaScript to create the Back-End's logic, meaning that we inevitably chose NodeJS, a runtime environment for JavaScript outside of the Browser, to host the server on. Node allows for elaborate asynchronous architecture to exist and can become particularly useful in small web-applications.

It can measurably be shown to work well with applications such as Hare, and scalable upon use [10]. While the technical details of running a single-thread environment may seem limiting, they allow for event handling with call-backs and promises to be used – appropriate for an app that is designed to be responsive.

Node also has very large and well supported package libraries, with the Node Package Manager (npm), which allows third party tools and support to be used to assist development as there is a large body of documentation for them on the internet.

**ExpressJS[11]**
Express is a framework used for running and hosting Back-End applications. It runs as a package (module) in NodeJS and can handle several key requirements we have for our applications controller. Namely, Express can handle routing requests, HTML requests and responses, and can provide a REST API, or interact with one.

As Express is a well-maintained framework that runs well with NodeJS, this is an appealing and ideal choice for development. While some of its features require the use of asynchronous programming, it can be used to serve bundled Front-End code, receive requests and respond with objects to the Front-End through an embedded REST API. While compared to Frameworks such as Flask, it may add extra layers of complexity to the architecture of the stack, in our case it opens up compatibility with JSON file database handling.

Express does not have any error handling or testing capabilities alone, so our team has opted to using a combination of Express-JS and "Jest", a JavaScript testing library for running automated unit tests before hosting and compilation.

## Testing

To reduce bugs, streamline development, and allow for iterative group programming, we opted to add automated testing to the running of the application. This assists in logging errors that are found or raised during the runtime of the application, which helps the development team quickly extinguish fires as they arise.

**Jest[12]**
Short for JavaScript Test, Jest is a NodeJS package that can run auto-tests, asyncronous black-box tests and spoof REST API return tests. It's particularly useful for integrating multiple modular components together, logs results.

## MongoDB Database

With our application, we expect to be holding user authentication information, user profile information, saved product information and images, and holding internal records of ingredients and substances users are likely to find in products that scan with the Front-End. The database will only be accessed from the Controller and Model in the Back-End, and thus can be completely decoupled from other components without affecting any functionality in the application.

As there are few or no relational models, we have used to hold this data, it was decided that a "NoSQL" or Non-SQL (Non-Relational) database model could be used, for which MongoDB is the most compatible with the rest of our stack. MongoDB offers an express compatible asynchronous database handling libraries and database hosting framework, as well as an external Database provider (MongoDB Atlas) for cloud storage. It also stores objects as BSON filetypes, a bijective equivalent to the JSON document standard, making it easier for us to parse in JavaScript, and fetch for use in responses.

The simplicity of the document model allows us to only create relational links when we need to wit the use of object ID's, saving us query time and space. For these reasons, and the minimalism of the project, going forward with MongoDB was decided upon.

## Hosting

To host the application's client, servers, database and libraries, we have concluded on the following:

**Front-End Browser Compatibility**
Through our React-based Front-End, we can expect that our application will be compatible with most modern browsers running a JavaScript engine that supports ES6 or above.

**React-Native for iOS and Android[13]**
Given that our application is written with React, we expect that the possibility of a "port" to React-Native, which allows for cross-compatible apps across mobile devices to be written with react, to be possible.

**Back-End Linux Server (Amazon Web Services)[14]**
The web application, and its' respective demo, is hosted on an Elastic-Computer Cloud server with Amazon Web Services, running Ubuntu's LTS distribution, at version 18.04.

**MongoDB Atlas for Database Hosting[15]**
The database is currently hosted for practical purposes with MongoDB's "Atlas", which automatically scales and balances data storage loads -- which while is not predicted to be necessary, allows for very stable database hosting in Australia.

# 04 Choice of Architecture

## Model-View-Controller

After clarifying the structure and components in the full stack, it became clear that our web application was greatly compatible with the Model-View-Controller (MVC)[16] architectural pattern. MVC is commonly used to manage user-interface centric applications (particularly like ours) where the user's view of the app prompts them to respond to a controller, and that in turn generates a model and posts that to the view, which continues the process. In our case, this may generally look like what is detailed in Fig 2.

Since the technical details are not included in this deliverable, we can simplify this diagram to associating the Back-End with both the "Model" and the "Controller", and the Front-End to the "View", which in a technical sense isn't completely possible, there are some parts of the "View" that must control the application – from a design and architecture perspective it is the general structure of the stack.

This has many advantages from a development perspective, as it creates a genuine separation of concerns, decouples primary components (introducing modularity to what would previously be a coupled Back-End) and allows for extension, or migration of its' functionality (for example, one section of the Back-End could be converted into an AWS Lambda[17] function set, and everything would work as normal).

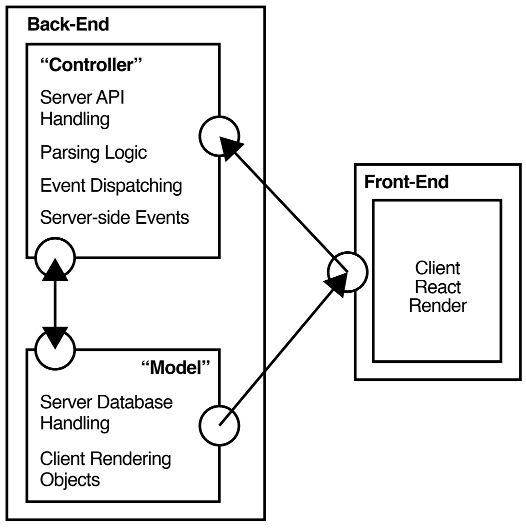


Figure 2
Very simplified model of architecture as MVC

**Links/References (II)**
**[9]** https://nodejs.org/en/
**[10]** https://arxiv.org/pdf/1507.02798.pdf
**[11]** https://expressjs.com/
**[12]** https://jestjs.io/
**[13]** https://facebook.github.io/react-native/
**[14]** https://aws.amazon.com/ec2/
**[15]** https://www.mongodb.com/cloud/atlas
**[16]** https://en.wikipedia.org/wiki/Model–view–controller
**[17]** https://aws.amazon.com/lambda/

# Chosen Architecture

With all of our chosen components for development, we can finally review the components involved in the entire application. Namely, devices, browsers, and external API's. The entire architecture of our application when including them appears as:

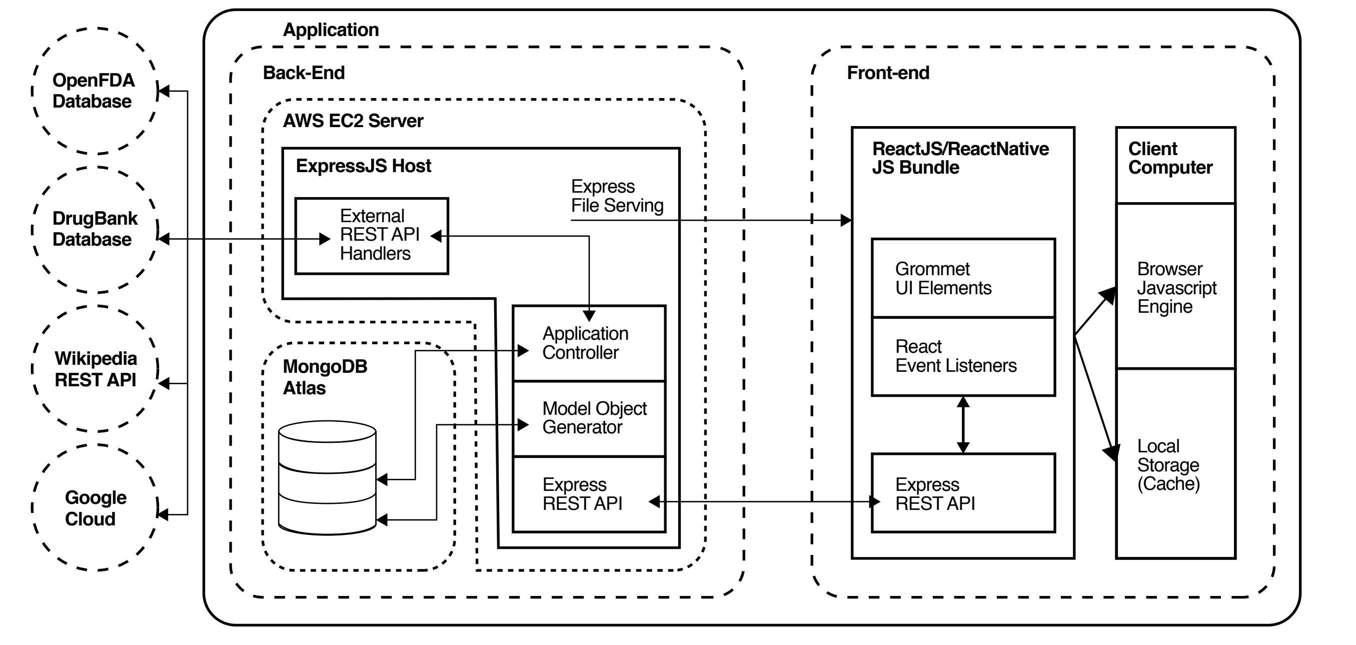


Figure 3
Diagram software architecture

However, when considering the architecture of the stack by itself, we can present it as the following figure. Namely, important sections and groupings have been outlined directly, and some of the technical details are not labelled or included if they don't provide any specific insight into the architecture of the application. This will be used later as a components reference.

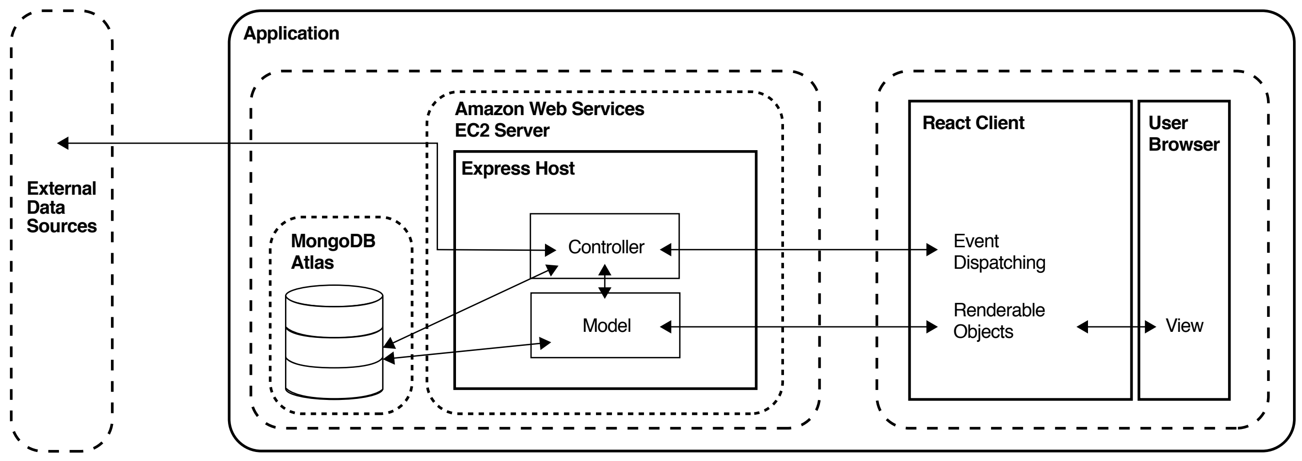


Figure 4
Simplified software architecture diagram

## Key Points Summary

# Software Design

## 05 Use Cases & User Stories

| | | |
|---|---|---|
| **UC01** | **Scan Product** | User scans a product, then uploads it |
| **UC02** | **View Ingredient** | User clicks on an ingredient to find out more information |
| **UC03** | **Log In** | User clicks “Log in” and logs in |
| **UC04** | **Register** | User clicks “Register” and registers for an account |
| **UC05** | **Log Out** | User clicks “Log Out” and logs out |
| **UC06** | **Change Highlighting** | User clicks “Change Highlighting” to change preferences |
| **UC07** | **Compare Products** | Use clicks “Compare” on a product page, and compares |
| **UC08** | **Save Product** | User clicks “Save” on a product page |
| **UC09** | **View Saved Products** | User clicks on profile and “Saved Products” |
| **UC10** | **Load Product** | User clicks on a product, after UC09 |
| **UC11** | **Update Account Details** | User clicks on “Account” and updates account details |

| | | |
|---|---|---|
| **Actor 01** | **Guest User** | Users who intend to use the application, but did not log in. |
| **Actor 02** | **Registered User** | Users who intend to use the application, and have logged in (and registered). |
| **Actor 03** | **OpenFDA** | The US Food and Drug Administrations open source database API. |
| **Actor 04** | **DrugBank (.ca)** | The Canadian DrugBank database API. |
| **Actor 04** | **Google Cloud (Vision/Translate)** | Google’s cloud services, notably their OCR image recognition API service, and their text-to-test language translation service. |
| **Actor 05** | **MongoDB Database** | A hosted instance of a MongoDB Database, with pre-set configurations on document structure. |
| **Actor 06** | **Authentication Agent** | Back-End service to check credentials, and store them. |

# Use Case Diagram

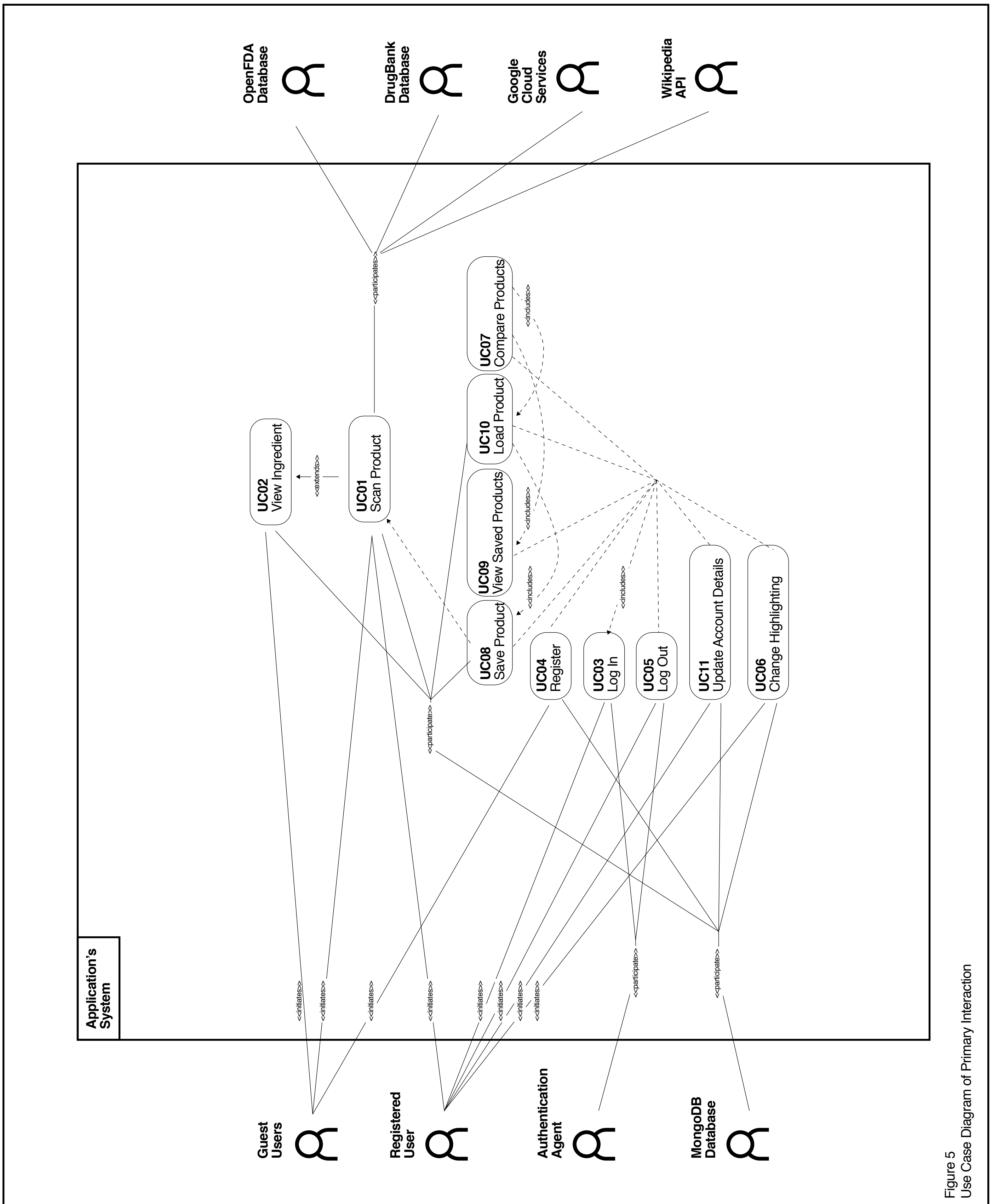
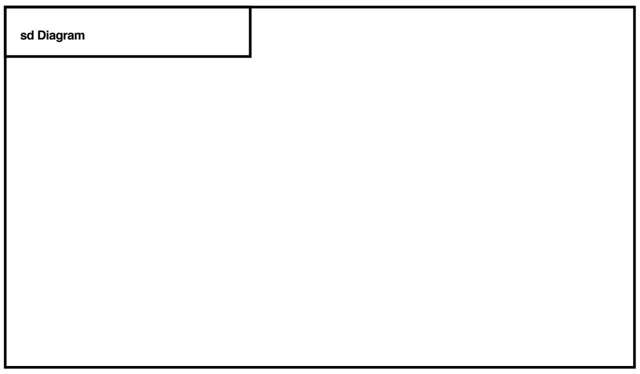


Figure 5
Use Case Diagram of Primary Interaction

# 06 Sequence Diagrams

In this section, the final section of the software design content, we clearly define the notation use in the sequence diagrams, so that their relationship with each use case, and the software architecture previously outlined are clear. Our definition of a sequence diagram is taken from both Wikipedia[18] and IBM[19]. All of the sequence diagrams provided will be referencing the architecture descripted is Figure 4, in Part 1.

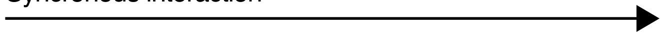
**UML Element Frames**
In the proceeding Sequence diagrams, frames such as below will be used to contain the entire interaction.



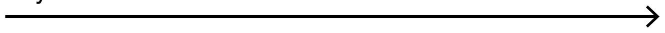
Where "sd" refers to "sequence diagram", and following is the titled interactions name.

**Interactions**
The interaction indications (either as a function or an object participting) are represented with the following arrow style notations.

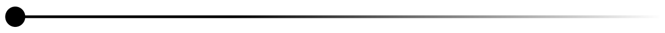
Syncronous interaction



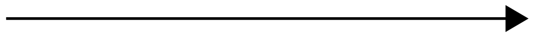
Asynconous interaction

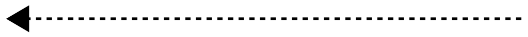


Entrance of interactions for frame

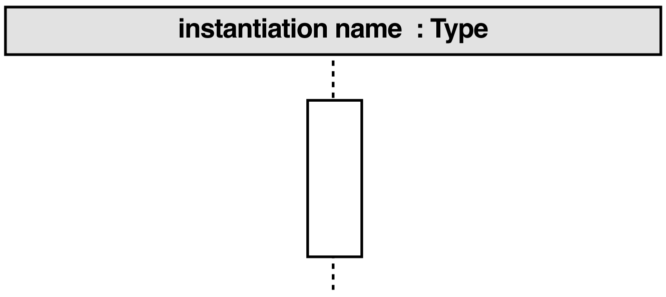


**Interaction types**
Despite the types of interactions, we must also consider whether or not it's "returning" or "sending".

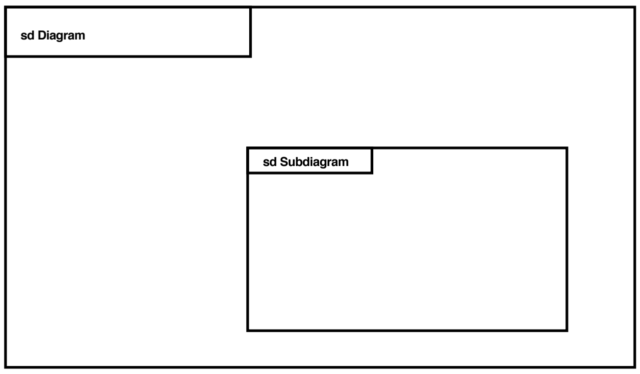
Sending 

Returning 

**Lifelines**
Lifelines are used to denote places where an interaction is "going to" or "returning from", such as container or component. In this particular report, all lifelines will be representative of components, and denoted by:



**Subdiagrams**
If a subset of a diagram is to be outlined as another digram, a subset may be included in the following way:

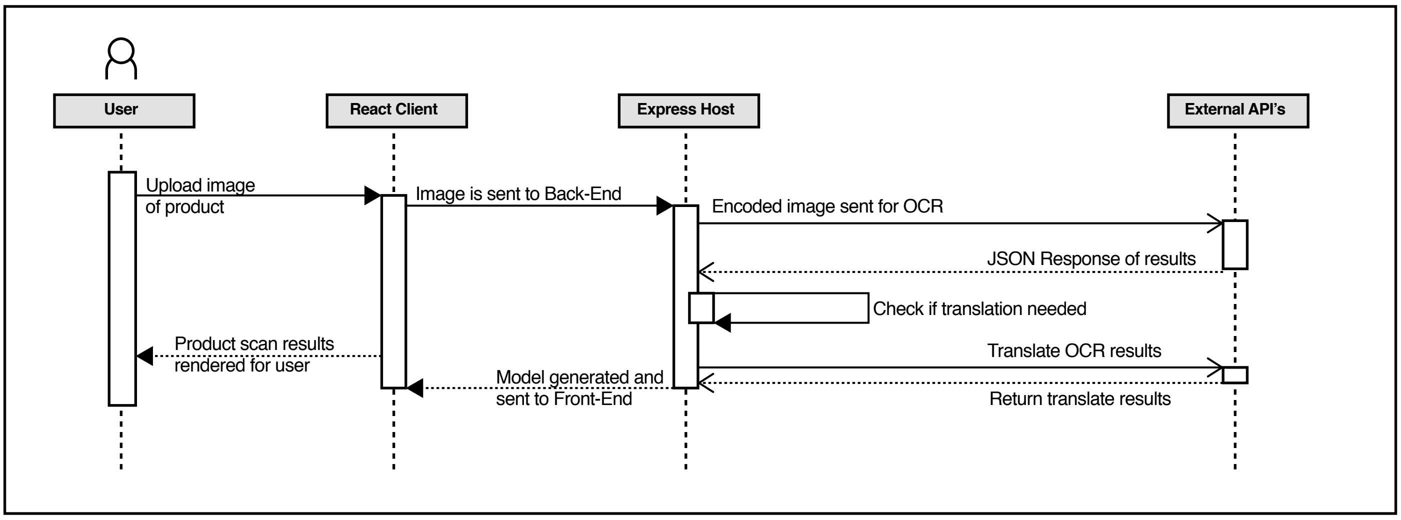


**Reference:**
[18] https://en.wikipedia.org/wiki/Sequence_diagram
[19] https://www.ibm.com/developerworks/rational/library/3101.html
[20] UML 2.0 Specification (http://www.uml.org/)

# Diagrams

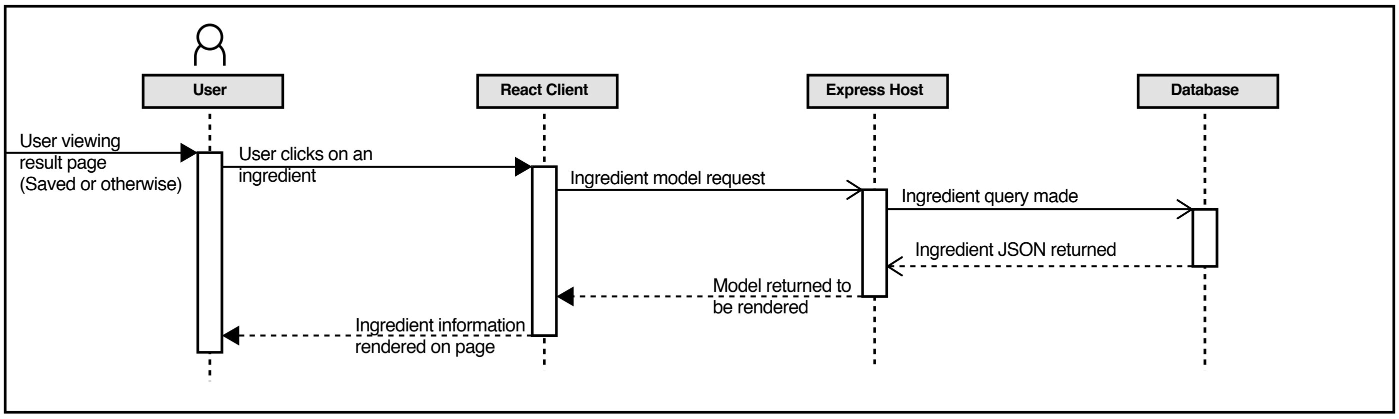
### UC01 Scan Product

User scans a product, then uploads it


User
React Client
Express Host
External API's
Upload image of product
Image is sent to Back-End
Encoded image sent for OCR
JSON Response of results
Check if translation needed
Product scan results rendered for user
Translate OCR results
Model generated and sent to Front-End
Return translate results

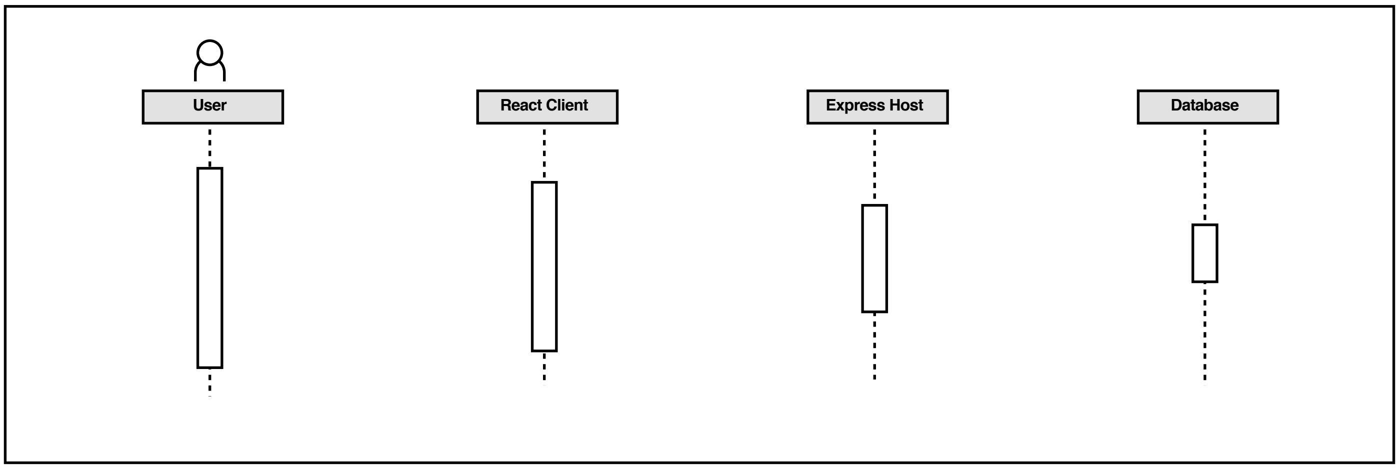
### UC02 View Ingredient

User clicks on an ingredient to find out more information


User
React Client
Express Host
Database
User viewing result page (Saved or otherwise)
User clicks on an ingredient
Ingredient model request
Ingredient query made
Ingredient JSON returned
Model returned to be rendered
Ingredient information rendered on page

### UC03 Log In

User clicks "Log in" and logs in



**UC04 Register**

User clicks “Register” and registers for an account

**UC05 Log Out**

User clicks “Log Out” and logs out

**UC06 Change Highlighting**

User clicks “Change Highlighting” to change preferences