1. Project Overview

What's the story of your grocery cart on a trip to a store? You came across a meal prep a friend suggested or you've been fancying a cheesy dish you saw over the internet or let's just bake that chocolate fudge cake already! Consider you decided on the chocolate cake. As most cases,you looked up the recipe online in the store and bought the ingredients for it. You come home and realize you already had a long forgotten jar of flour and a bag of carrots in the fridge that are treading close on the expiry line! You could have saved a lot more if the thought of a nice chunky carrot cake somehow slipped across your mind, isn't it?

In this day and age of independent people, there's always a dilemma we face frequently in our day-to-day life; whether to cook food at home or eat outside? Now, we may not have a good idea of what ingredients we have at our home and what we can cook using them. We may not even know about the restaurants we have nearby our location. This uncertainty due to lack of information can lead us to taking inefficient decisions. But, worry no more because we have got just the solution you need!

aLaCart is a web application which can help you take informed decisions based on what's there in your fridge and where you are located. Our application not only keeps track of what ingredients you have at home but also suggests recipes you can make using them along with the cooking time. We can also provide you with a text message on your phone to let you know what's there in your fridge to make it more convenient. If you prefer to eat outside instead, our application can provide you with the restaurants nearby your location. If you want to know more about the restaurants, you can check all their details with a click of a button. Apart from this, you can have a look at what's cooking at your friends' homes through the live feed network! You too can share your interesting findings and recipes with your friends to help them make better dishes.

2. Solution

We propose a solution of webapp, where we can store ingredients we have in the fridge, on the cloud. We modify it to retrieve recipes that can be cooked, restaurants nearby by locating user entered pincode

Smart Refrigerator:

- aLacart offers an option to add groceries to the app in text as well as uploading image
- We have included image recognition algorithm to identify object/ingredients user trying to add to the fridge
- User is enabled to delete ingredients from the cloud with texts
- User is provided with the facility of viewing all the items in fridge.
- Each item is displayed with a time stamp of its creation.
- aLacart provides SMS notification service to provide user the list of ingredients in the fridge.

Restaurant Locator

- aLacart is designed to provide a map consisting of the nearby restaurants when users enter their location using a pincode
- Each Restaurant is displayed with its url, rating and name
- App offers proximity of restaurants within certain range

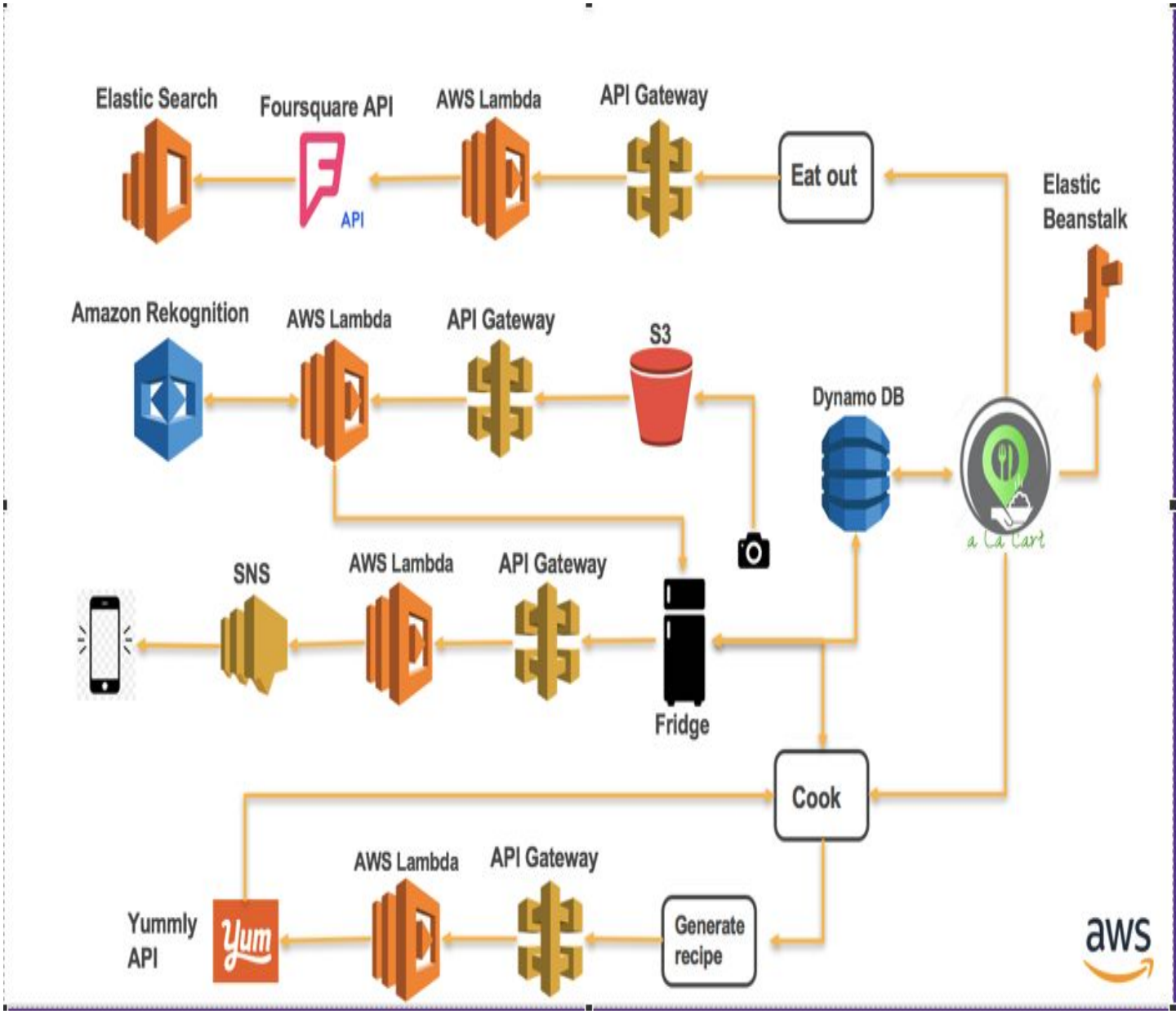
Food Network

- We have designed a Food Network which is similar to facebook feed, where a user can see all the posts other users have made
- We give an option for the user to upload the image of the dish they cooked by referring to the recipes suggested by aLacart
- We offer intelligent ways of maintaining the session to improve user experience by showing previously browsed items
- aLacart displays top rated dishes in the feeds as suggestions

Recipe generator:

- aLacart is intelligent enough to suggest dishes you can cook based on the ingredients you have in the fridge.
- Each Recipe is generated with Cooking time, so that user can calculate total amount of time he/she needs from current location to tell finishing cooking recipe
- Each Recipe is displayed along with a rating to give the user a better idea

3. Architecture Diagram



The above architecture diagram can be broken down into four major components:

Restaurants request handling:
In this, we are using an API gateway which triggers the lambda function which is responsible to obtain the data required to call the FourSquare API with the appropriate input parameters. All the data obtained from the API is then put into the ElasticSearch Service

Image handling:
When each user uploads an image of the dishes he/she cooked, it gets stored into the S3 and a link to that image is stored in the DynamoDB for that particular user. A lambda function is then triggered which is responsible to use the Amazon Rekognition API to obtain the ingredients present in the image.

Fridge content handling:
Each user's fridge contents are stored in the Dynamo DB. Whenever a user wishes to view/add/delete the contents in the fridge, the Dynamo DB is queried. Furthermore, a lambda function is triggered to use the SNS client which in turn is responsible to send a text message to the user describing the contents in his/her fridge

Recipe generation:
Here, if the user wishes to cook at home after looking at the ingredients present in the fridge, an API gateway is used to trigger the lambda function which is responsible for obtaining the contents in the user's refrigerator and calling the Yummly API with the appropriate input parameters. After obtaining the recipe results from the API, the contents are reflected on the front end

The entire web application is deployed on Elastic Beanstalk to ensure scalability and efficient throughput from the system.

4. Design, implementation choices and Algorithms

A. *Server*:

- Web app uses php server for core logic. App is designed to accommodate different services for the future.
- Web application php server of version 5.6.
- Web application has couple of services which are completely serveless, they are built using aws resources. Ex: Restaurants locator and SMS
- Web application has serverful architecture for recipe generation and live feeds
- Server is deployed in aws elastic beanstalk which is faster compared to other servers like django
- Choosing PHP as server for the web application was to ease deployment process and for the flexibility it provides to maintain sessions when user login.

B. *Database*

- App uses aws NoSQL DynamoDB for database. Using cloud resources provides us with freedom of expanding quickly when user base grows.
- aLacart uses 4 tables
    - User Info - Maintaining login sessions, credentials
      Primary Key: email id
    - MultiMedia - Used for storing fields of live feeds
      Primary key: email id
    - Session: Used for storing browsing history of user to display suggestion next time when he logs in
    - Refrigerator : Used for storing items in the fridge
      Primary key: email id
- DynamoDB allows developers to purchase a service based on throughput, rather than storage which essentially cut the cost of maintaining storage.

C. *Restaurant data*:

- App uses aws elasticsearch to store restaurents data for faster access
- Foursquare api is used to get data of restaurants
- Vicinity is used filter out only nearby restaurants

D. *aws Rekognition*

- App uses aws Rekognition for image recognition
- For the sake of accuracy, user is displayed with suggestion of photo is trying to upload. It is up to the user to select the item suggested by the recognition system

E. *Recipe Generation*:

- App uses Yummly api for recipe generation
- Application intelligently uses part of refrigerator contents to suggest possible recipes that can be cooked
- We classify ingredient to get near accurate recipe. Ingredients can be classified into
    - Essentials
    - Vegetables
    - Spices
- Combination of these classification helps to filter out odd recipe generations.

Ex: Egg, cucumber, yogurt, sugar, chocolate does not generate any recipe, however if we use combination among, we get near accurate results.

F. *Notification*

- App uses aws sns for SMS notification

G. *API designs*

- All the API's are designed with aws lambda coupled with API Gateway

H. *Storage*

- Every Image uploaded by user is stored in aws s3.
- Each image links are stored in DynamoDB so that user gets feed of entire aLacart network

Algorithmic depiction of system

Amazon Rekognition API:

It's not for nothing that the Amazon Rekognition API is used widely for image and video analysis. Since the image recognition model is trained on a vast dataset, we could obtain accurate results giving us the required ingredients present in a user uploaded image (taken from S3). Also, the API usage is fairly simple and straightforward. The API returns the labels associated with an image along with a confidence value. We used the 'detectLabels' function of the API and set the confidence level to 75% to ensure that only the appropriate labels are returned.
A sample response can be seen below:

```
{
   "Confidence":87.81458282470703,
   "Name":"Car"
},
{
   "Confidence":87.81458282470703,
   "Name":"Vehicle"
},
{
   "Confidence":82.21033477783203,
   "Name":"Sedan"
},
{
   "Confidence":78.62909698486328,
   "Name":"Boardwalk"
},
{
   "Confidence":78.62909698486328,
   "Name":"Path"
},
{
   "Confidence":78.62909698486328,
   "Name":"Pavement"
},
{
   "Confidence":78.62909698486328,
   "Name":"Sidewalk"
},
```


Yummly API:

The Yummly Recipe API is a well-known API which lets us integrate recipe search into web applications. It provides a lot of control over the search by including various input parameters and providing various filters to the responses.Alternatively, we can also use the ‘RecipePuppy’ API to obtain more number of results. But, unfortunately, it doesn’t provide us with the cooking time which was a deal breaker in our case.

A snippet of a sample response from the Yummly API can be seen below:

Get Recipe response sample


```
{
    "attribution": {
        "html": "<a href='http://www.yummly.com/recipe/Hot-Turkey-Salad-Sandwiches-Allrecipes'>Hot Turkey Salad
Sandwiches recipe</a> information powered by <img src='http://static.yummly.com/api-logo.png'/>",
        "url": "http://www.yummly.com/recipe/Hot-Turkey-Salad-Sandwiches-Allrecipes",
        "text": "Hot Turkey Salad Sandwiches recipes: information powered by Yummly",
        "logo": "http://static.yummly.com/api-logo.png"
    },
    "ingredientLines": [
        "2 cups diced cooked turkey",
        "2 celery ribs, diced",
        "1 small onion, diced",
        "2 hard-cooked eggs, chopped",
        "3/4 cup mayonnaise",
        "1/2 teaspoon salt",
        "1/4 teaspoon pepper",
        "6 hamburger buns, split"
    ],
    "flavors": {
        "Salty": 0.004261637106537819,
        "Meaty": 0.0019220244139432907,
        "Piquant": 0,
        "Bitter": 0.006931612268090248,
        "Sour": 0.009972159750759602,
        "Sweet": 0.0032512755133211613
    },
    "nutritionEstimates": [
        {
            "attribute": "ENERC_KCAL",
            "description": "Energy",
            "value": 317.4,
            "unit": {
                "name": "calorie",
                "abbreviation": "kcal",
                "plural": "calories",
                "pluralAbbreviation": "kcal"
            }
        },
        {
            "attribute": "FAT",
            "description": "Total lipid (fat)",
            "value": 13.97,
            "unit": {
                "name": "gram",
                "abbreviation": "g",
                "plural": "grams",
                "pluralAbbreviation": "grams"
            }
        },
        {
            "attribute": "FASAT",
            "description": "Fatty acids, total saturated",
            "value": 2.7,
            "unit": {
                "name": "gram",
                "abbreviation": "g",
                "plural": "grams",
                "pluralAbbreviation": "grams"
            }
        },
        {
            "attribute": "CHOLE",
            "description": "Cholesterol",
            "value": 0.11,
            "unit": {
                "name": "gram",
                "abbreviation": "g",
                "plural": "grams",
                "pluralAbbreviation": "grams"
            }
        },
```


For our web application, we took advantage of the ‘allowedIngredient[]’ search parameter where we provide the contents of the user’s fridge by querying the Dynamo DB and creating a comma

separated list of ingredients. The response fields which were relevant to our application were : ‘name’, ‘ingredients’, ‘rating’ and ‘cooking time’

FourSquare API:

FourSquare API is a popular API to obtain information about various venues. We took advantage of this API to search for the restaurants nearby a location and obtained their details. At first, we had resorted to use the Yelp API to get the details. But, unfortunately, the API documentation changed and was not reflected in time on their website. We decided to use the FourSquare API instead since it was easy and convenient to use. Also, it provides us with plethora of details about a particular venue. Attached below is a snippet of a sample response from the FourSquare API as well as a snippet of the filters and responses given by the API


```
"venue": {
  "id": "412d2800f964a520df0c1fe3",
  "name": "Central Park",
  "contact": {
    "phone": "2123106600",
    "formattedPhone": "(212) 310-6600",
    "twitter": "centralparknyc",
    "instagram": "centralparknyc",
    "facebook": "37965424481",
    "facebookUsername": "centralparknyc",
    "facebookName": "Central Park"
  },
  "location": {
    "address": "59th St to 110th St",
    "crossStreet": "5th Ave to Central Park West",
    "lat": 40.78408342593807,
    "lng": -73.96485328674316,
    "labeledLatLngs": [
      {
        "label": "display",
        "lat": 40.78408342593807,
        "lng": -73.96485328674316
      }
    ],
    "postalCode": "10028",
    "cc": "US",
    "city": "New York",
    "state": "NY",
    "country": "United States",
    "formattedAddress": [
      "59th St to 110th St (5th Ave to Central Park West)",
      "New York, NY 10028",
      "United States"
    ]
  },
  "categories": [
    {
      "id": "4bf58dd8d48988d163941735",
      "name": "Park",
      "pluralName": "Parks",
```

Parameters

Name	Example	Description
ll	44.3,37.2	**required** unless `near` is provided. Latitude and longitude of the user's location. Optional if using `intent=global`
near	Chicago, IL	**required** unless `ll` is provided. A string naming a place in the world. If the `near` string is not geocodable, returns a `failed_geocode` error. Otherwise, searches within the bounds of the geocode and adds a geocode object to the response.
intent	checkin	One of the values below, indicating your intent in performing the search. If no value is specified, defaults to `checkin`.
radius	250	Limit results to venues within this many meters of the specified location. Defaults to a city-wide area. Only valid for requests with `intent=browse`, or requests with `intent=checkin` and `categoryId` or `query`. Does not apply to `intent=match` requests. The maximum supported radius is currently 100,000 meters.
sw	44.3,37.2	With `ne`, limits results to the bounding box defined by the latitude and longitude given by sw as its south-west corner, and ne as its north-east corner. The bounding box is only supported for `intent=browse` searches. Not valid with `ll` or `radius`. Bounding boxes with an area up to approximately 10,000 square kilometers are supported.
ne	44.1,37.4	See `sw`.
query	tacos	A search term to be applied against venue names.
limit	10	Number of results to return, up to 50.
categoryId	4bf58dd8d48d11094, 4bf58dd8d1bd941735	A comma separated list of **categories** to limit results to. If you specify categoryId, specifying a `radius` may improve results. If specifying a top-level category, all sub-categories will also match the query. Does not apply to `intent=match` requests.
llAcc	10000.0	Accuracy of latitude and longitude, in meters.
alt	0	Altitude of the user's location, in meters.
altAcc	10000.0	Accuracy of the user's altitude, in meters.

Value	Description
checkin	Finds venues that the current user (or, for userless requests, a typical user) is likely to checkin to at the provided `ll`, at the current moment in time. This is the `intent` we recommend most apps use.
global	Finds the most globally relevant venues for the search, independent of location. Ignores all parameters other than `query` and `limit`.
browse	Find venues within a given area. Unlike the `checkin intent`, browse searches an entire region instead of only finding venues closest to a point. A region to search can be defined by including either the `ll` and `radius` parameters, or the `sw` and `ne`. The region will be circular if you include the ll and radius parameters, or a bounding box if you include the sw and ne parameters.
match	Finds venues that are near-exact matches for the given parameters. This intent is primarily used when trying to harmonize an existing place database with Foursquare's and is **highly sensitive** to the provided location. The results will be sorted by best match first, taking distance and spelling variations into account. `name` and `ll` are the only required parameters for this intent, but we also suggest sending `phone`, `address`, `city`, `state`, `zip`, and `twitter` for better results. There's no specified format for these parameters—we do our best to normalize them and drop them from the search if unsuccessful.

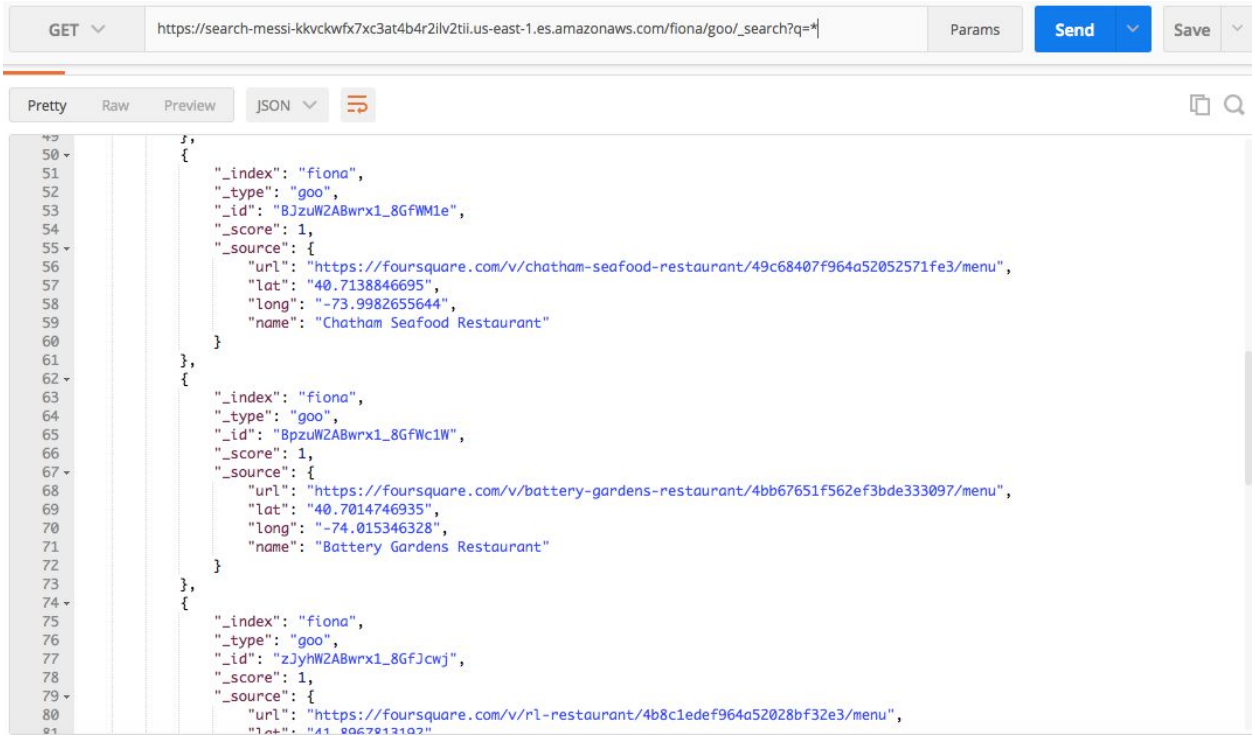
Response Fields

Field	Description
id	A unique string identifier for this venue.
name	The best known name for this venue.
contact	An object containing none, some, or all of `twitter`, `phone`, and `formattedPhone`. All are strings.
location	An object containing none, some, or all of `address` (street address), `crossStreet`, `city`, `state`, `postalCode`, `country`, `lat`, `lng`, and `distance`. All fields are strings, except for `lat`, `lng`, and `distance`. Distance is measured in meters. Some venues have their locations intentionally hidden for privacy reasons (such as private residences). If this is the case, the parameter `isFuzzed` will be set to true, and the `lat`/`lng` parameters will have reduced precision.
categories	An array, possibly empty, of **categories** that have been applied to this venue. One of the categories will have a `primary` field indicating that it is the primary category for the venue. For the complete category tree, see **categories**.
verified	Boolean indicating whether the owner of this business has claimed it and verified the information.
stats	Contains `checkinsCount` (total checkins ever here), `usersCount` (total users who have ever checked in here), and `tipCount` (number of tips here).
url	URL of the venue's website, typically provided by the venue manager.
menu	An object containing `url` and `mobileUrl` that display the **menu** information for this venue.
price	An object containing the price `tier` from 1 (least pricey) - 4 (most pricey) and a `message` describing the price tier.
hereNow	Information about who is here now. If present, there is always a `count`, the number of people here. If viewing details and there is a logged-in user, there is also a `groups` field with `friends` and `others` as `type`s.
createdAt	Seconds since epoch when the venue was created.
photos	A count and groups of photos for this venue. Group types are `checkin` and `venue`. Not all items will be present.
tips	Contains the total `count` of tips and `groups` with `friends` and `others` as `groupType`s. Groups may change over time.
beenHere	Contains `count` of the number of times the acting user has been here. Absent if there is no acting user.
shortUrl	A short URL for this venue, e.g. http://4sq.com/Ab123D
canonicalUrl	The canonical URL for this venue, e.g. https://foursquare.com/v/foursquare-hq/4ab7e57cf964a5205f7b20e3
like	Indicates if the current user has liked this venue.
dislike	Indicates if the current user has disliked this venue.
roles	Present if and only if the current user has at least one assigned role for this venue. The value is a list of all of the current user's assigned roles for this venue. Possible values for each element of the list are `manager` and `employee`. Subject to change as additional roles may be defined.

From the parameters available in the API, we have used the ‘ll’ parameter which is used to input the latitude and longitude of an area to be searched for and the ‘query’ parameter to limit the results to restaurants only. From the response fields, we found the responses ‘name’, ‘lat’, ‘lng’, ‘address’, ‘contact’, ‘url’ and ‘menu’ to be relevant and useful for our web app.

Elastic Search:

We populated aws elasticsearch with restaurants data, we apply vicinity to filter out restaurants nearby.



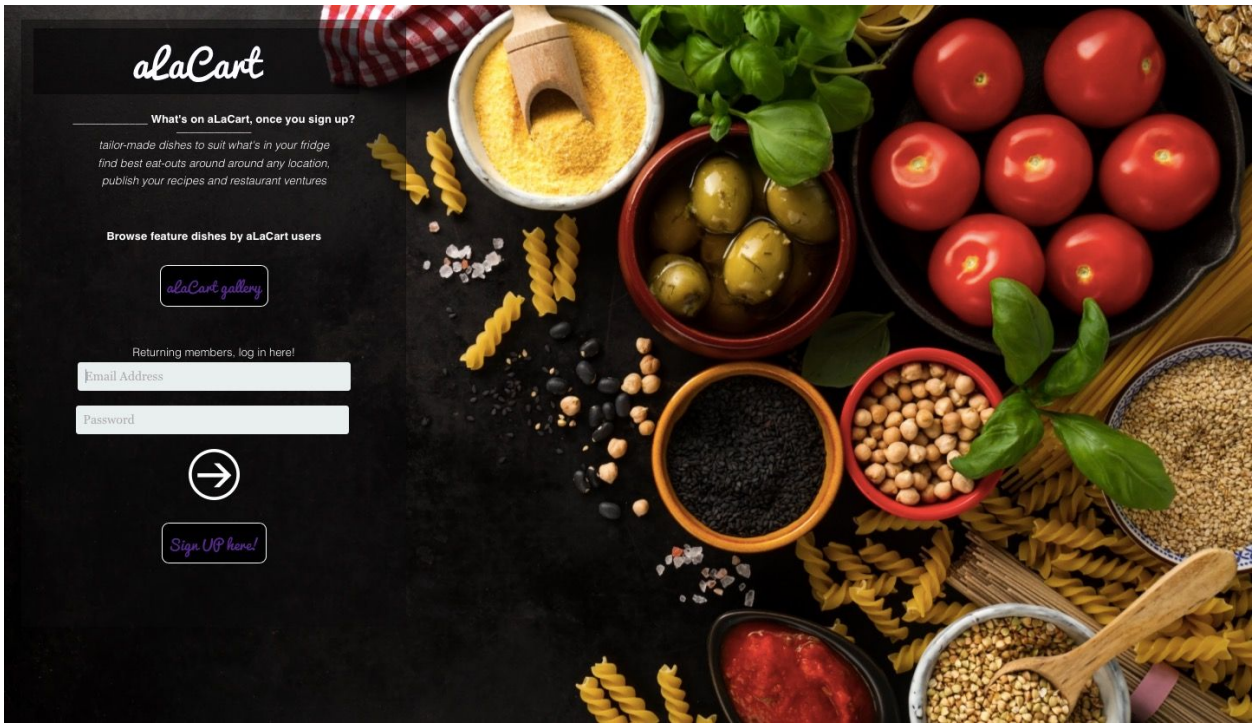
We parse above elasticsearch response to display restaurants on on the map. We utilise each fields to plot, to show details of restaurants etc.

5. Results and UI:

Attached below are some of the screenshots of our web application:

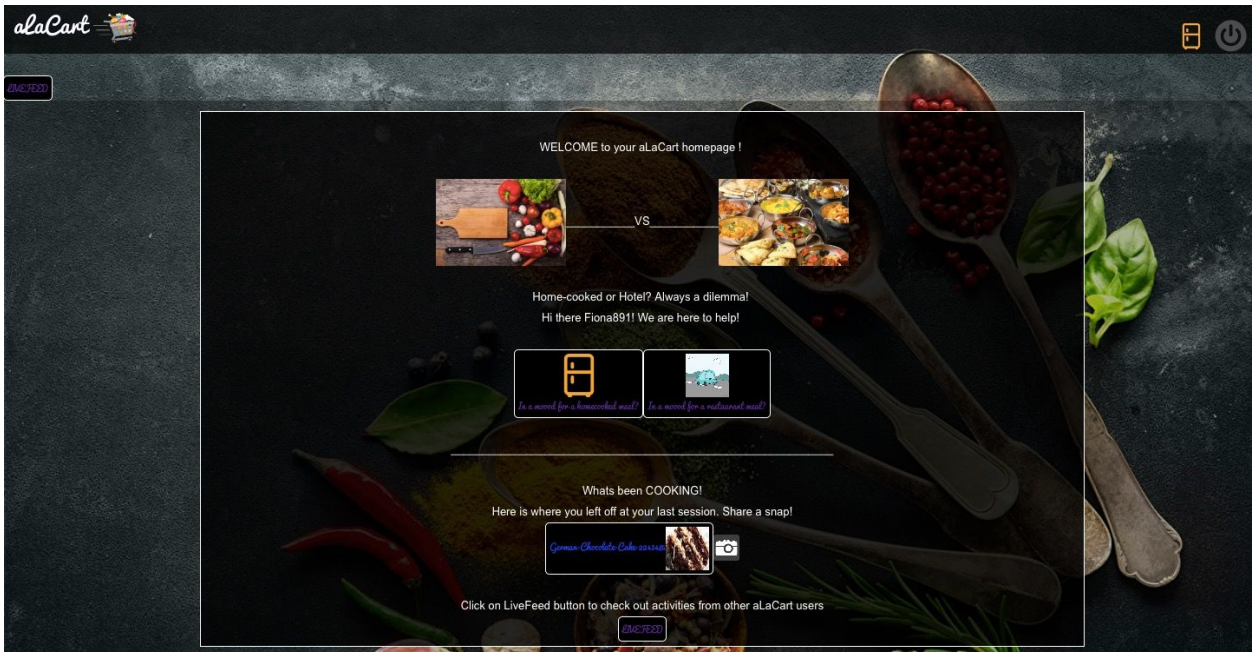
Home page:

The home page enables the user to register/ log into the aLacart web application



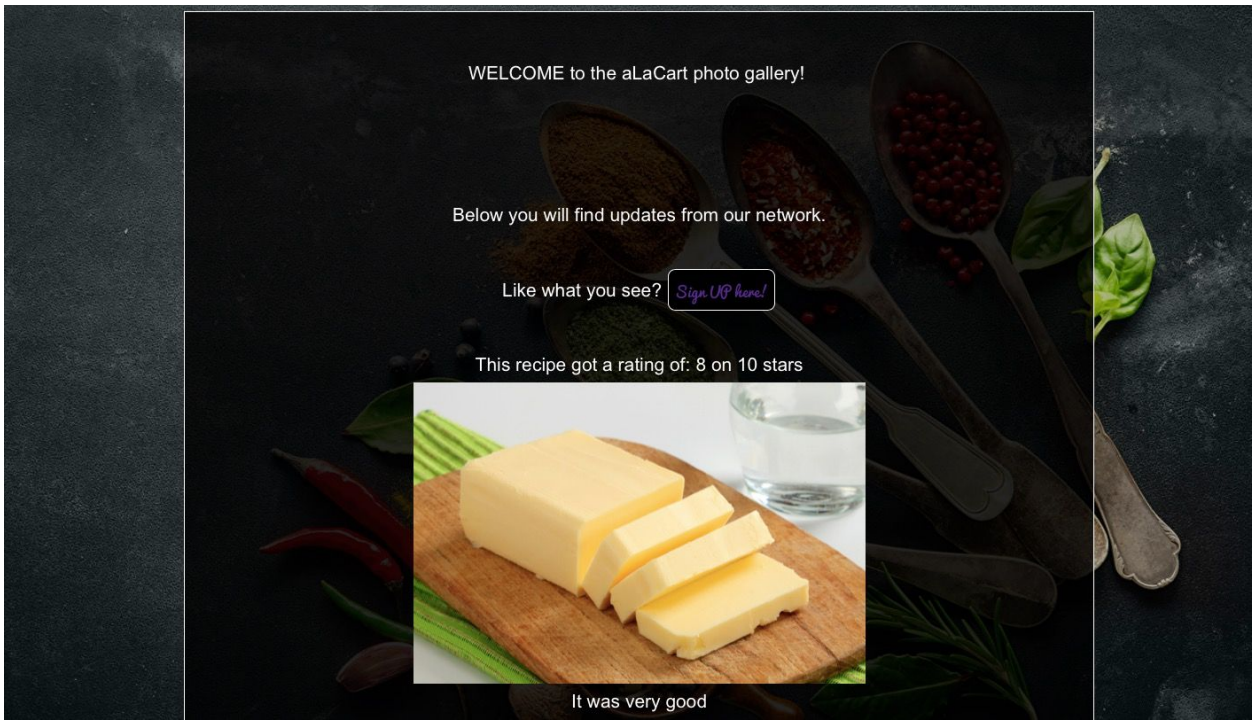
Cooking vs Eating outside:

This page provides the user with an option to select whether he wishes to cook at home or eat outside. Also, due to session maintenance, it's possible for the user to pick up where he left from in his previous session. The user can also opt to view the LiveFeed to check out his friends' establishments.



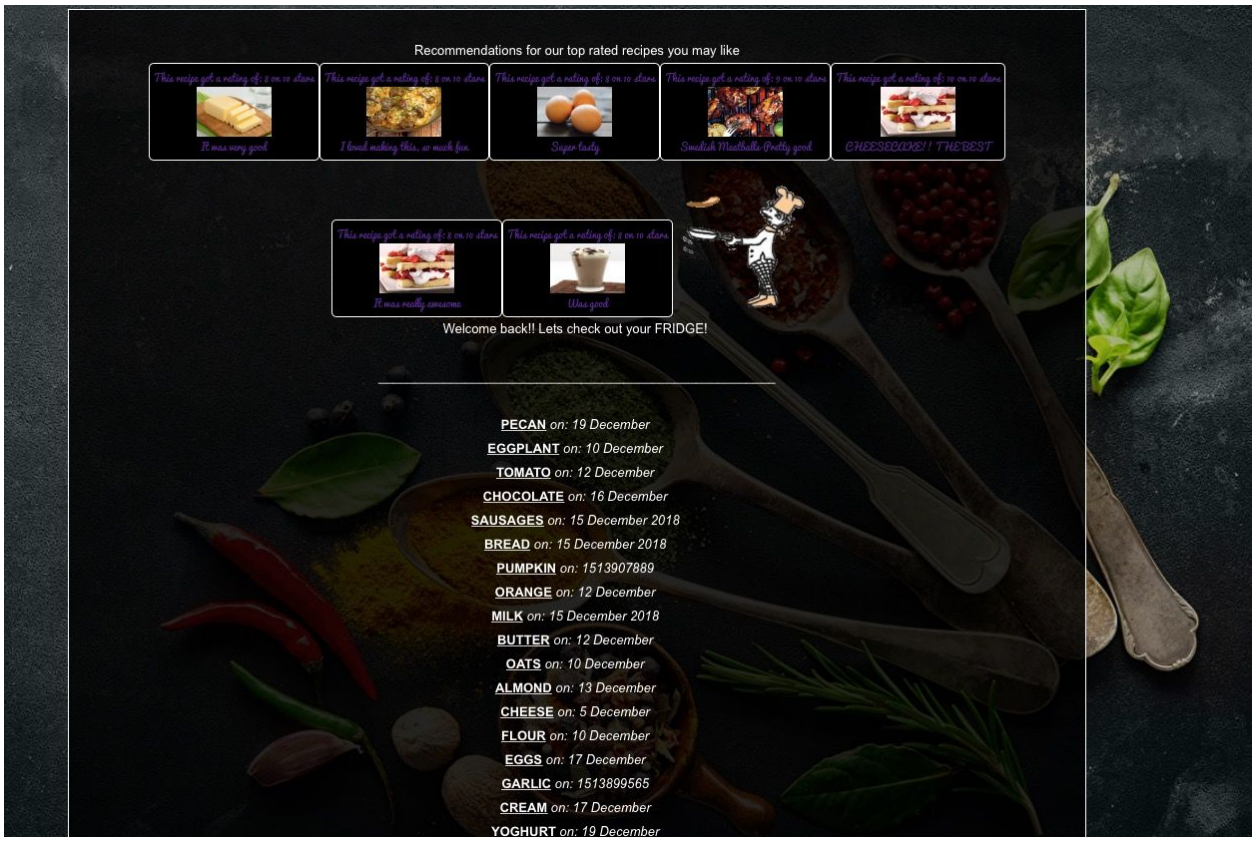
Live feed snippet:

As you can see, this is a snippet of a recipe a user has uploaded. For testing purposes, the image just contains butter but it could be anything the user has cooked and wishes to share with the network. The recipe has a rating of 8/10 stars which makes this dish to be near the top spot in case of recommendations. The user is also capable of writing a personal review for the dish.



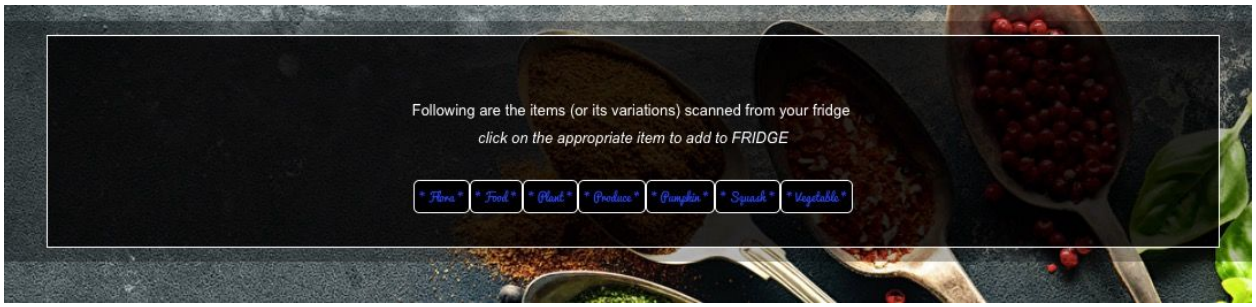
Checking ingredients in the fridge:

This page displays all the interesting recipes the user can currently make from the contents in his fridge. These recipes have been displayed on the basis of the confidence level from the yummly api



Adding the ingredients to your fridge by uploading an image:

In this page, the user can upload an image of his fridge or a new item he recently bought. The Amazon Rekognition API returns a list of labels associated with the image, i.e., the food items present in the image. Although AI has advanced a lot in these years, it's still possible that the API may return absurd labels. Hence, the user is allowed to select the ingredients which he desires to be added to his fridge

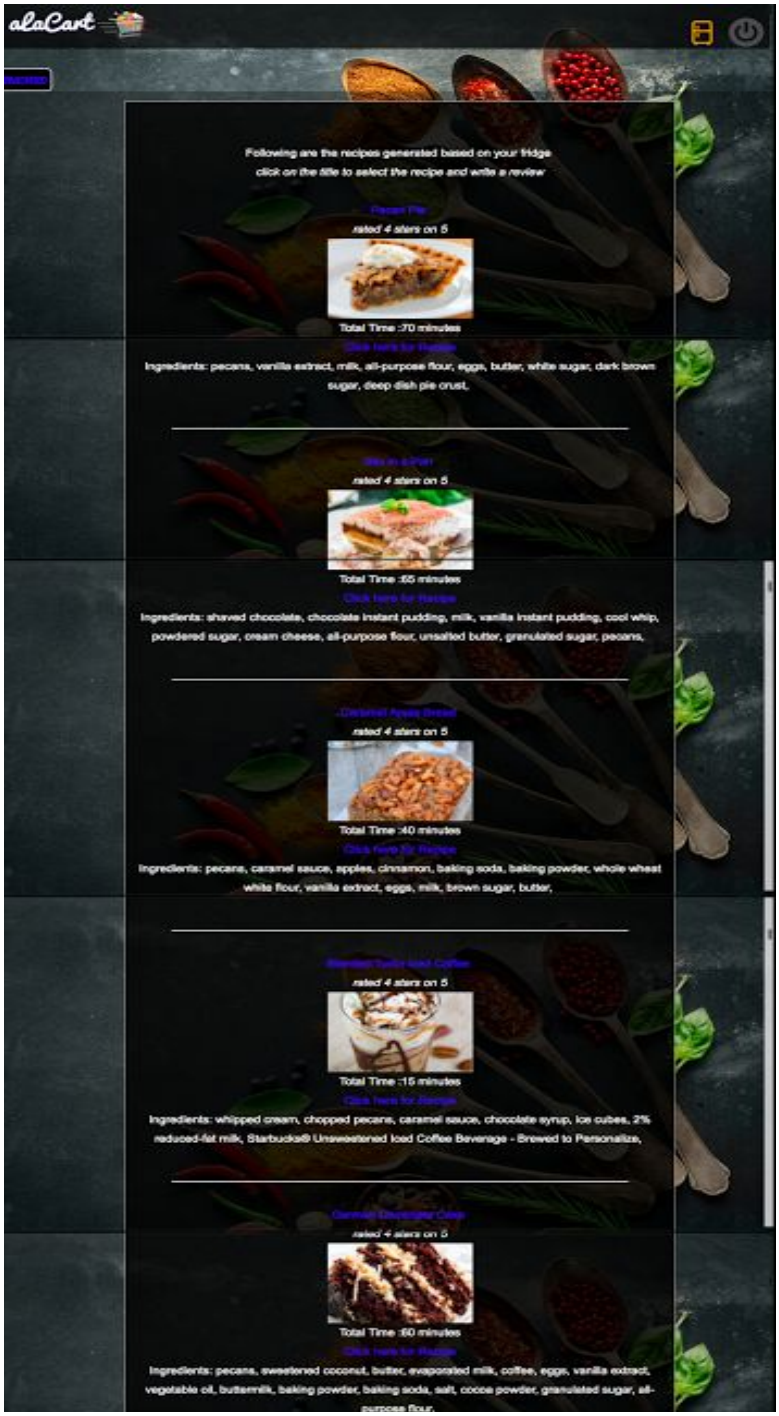


Sending the ingredients to your mobile phone:

For the user's convenience, the ingredients in the fridge can be transferred to his/her mobile phone by entering the mobile phone number (along with the extension) in this page

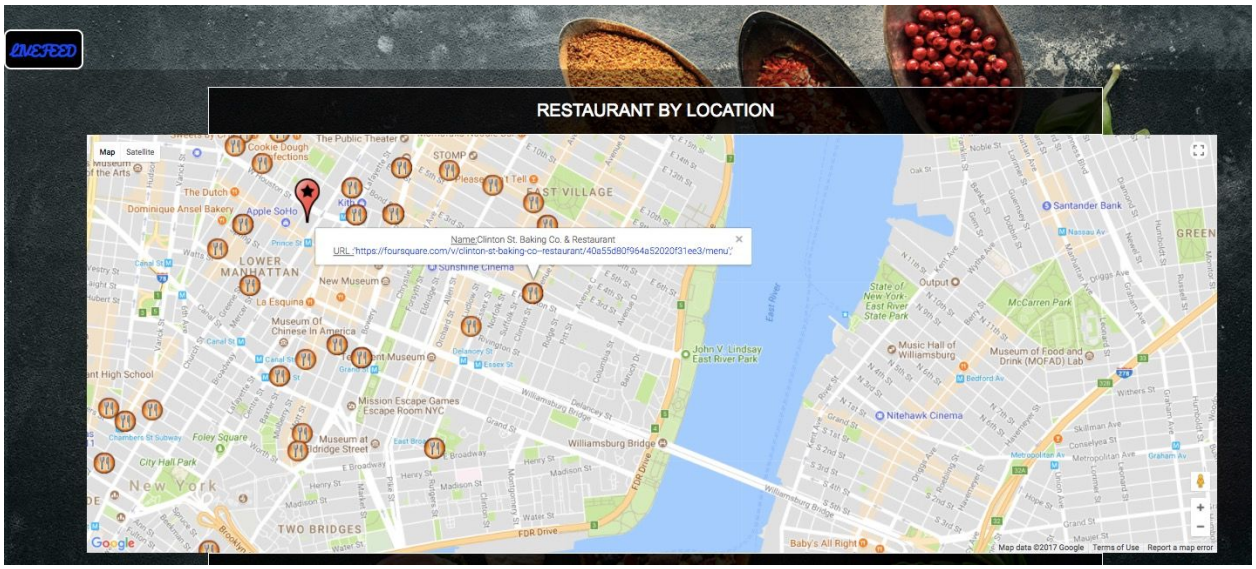
Recipe recommendations snippet:

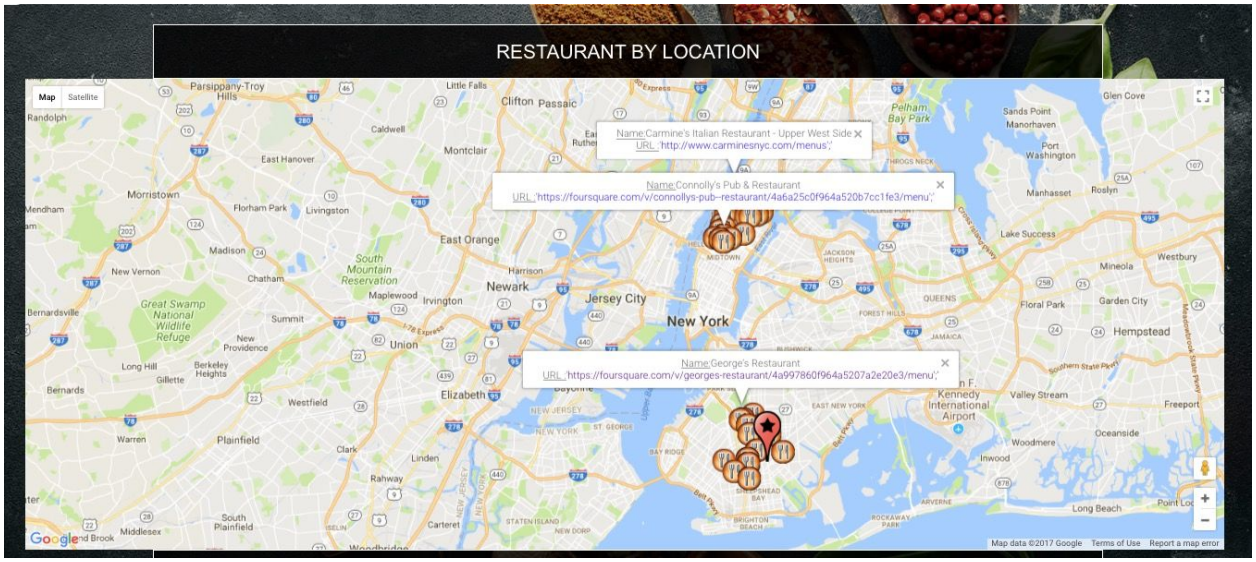
The screenshot contains a snippet of all the recipes the user can cook along with the details such as ingredients, cooking time and average rating. As an example, you can see all the recommendations provided when the fridge contains ingredients like pecan, eggplant, tomatoes, chocolate, sausages, bread, etc.



Finding nearby restaurants:

After entering the pincode of the location, the user is redirected to a page which contains a map displaying all the nearby restaurants based on proximity filtering. Also, the user can drag the location marker to a different place and within less than half a second, the restaurants near that location are populated





6. Summary:

We have successfully implemented all the features mentioned in the project proposal. We managed to use and integrate different AWS resources (ElasticSearch, Lambda, DynamoDB, S3, SNS, Elastic Beanstalk) and APIs (Amazon Rekognition, Yummly, FourSquare) into our website to ensure we can deliver a complete, scalable and well-designed web application. Because of modularizing the application and having isolated components integrated together, the feedback from the application is instant. The web app is ready to be deployed and can be used by the masses.

7. Git Link

https://github.com/fionasequeira/aLaCart_v2