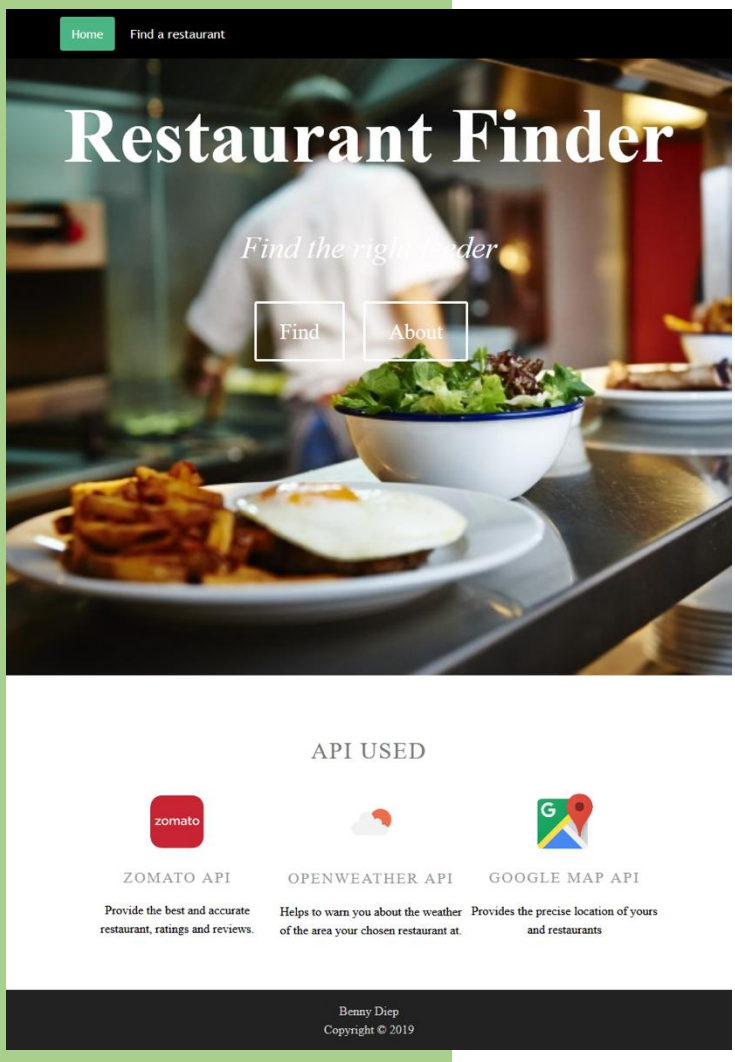
2019

RestaurantFinder



CAB432

Assignment 2

Thanh Tuan Diep - n9607234

Maria Berge - N9770992

10/21/2019

Contents

Introduction

Purpose & description

Restaurant Finder helps users decide on the best restaurant that fits their cravings by providing all the necessary information about the restaurant as well as images of the food in an interactive environment. In addition, the application provides the current weather forecast at the location of the restaurant so that users can be prepared. Using Zomato API, OpenWeather API and Google Map API, users can search for and display restaurant images and details as well as reviews of a specific restaurant.

Upon launching the application, users have the option to search for a specific restaurant name, a type of cuisine, a specific suburb or use their current location to search for a nearby restaurant. Once results come in, the map will move and set its centre to the most relevant result, but user can zoom out and choose the restaurant of their choice. Notice that all results are shown as markers on the map and when clicked on, an information window will open with all the useful details inside as well as a slider of images of the food taken by the customers who have eaten there. A number of reviews are also included at the bottom of the window.

Please note that most of the materials regarding APIs and the application structures are similar to the one used in Assignment 1.

Moreover, for this assessment, the Restaurant Finder application will be deployed to a cloud server with the capabilities of scaling and persistence to reduce the stress when the application under heavy load. The health probe and load balancing rules along with other settings will be discussed in the technical description.

Services used

Zomato API

https://developers.zomato.com/Zomato

Zomato API provides various API endpoints that can be used to retrieve details about cuisines, restaurants, locations, daily menus, images of the restaurant or food as well as reviews from customers, and their profiles. Restaurant Finder only uses one API endpoint from Zomato> /search, which is used to search for restaurants by the user's query keywords which could be cuisines, locations, restaurant names, etc. This endpoint gives a 'dynamic' response which include details of restaurants, photos, reviews and reviewer profile details.

OpenWeather API

https://openweathermap.org/api

OpenWeather has numerous API endpoints which give current forecast weather along with statistical weather data and even air pollution. There are options to search these by city name, city ID, geographic coordinates or zip code. For this application, retrieving the current weather using latitudes and longitudes should satisfy the requirements.

Google Map API

https://developers.google.com/maps/documentation/javascript/tutorial

Google Maps JavaScript API is a popular, well-instructed JS library for creating maps. The library also provides customizable markers and pop-up information windows. For this application, the API is mostly used for visualising data and locations of the restaurant retrieved from Zomato API.

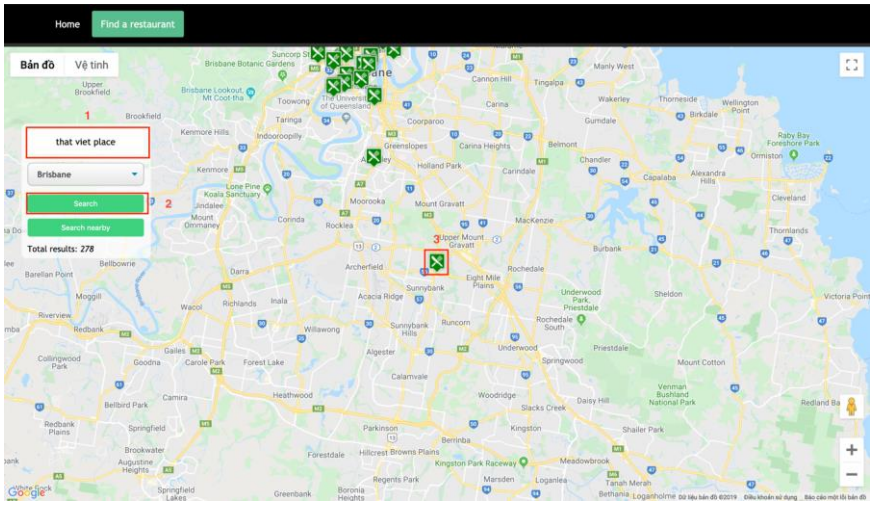
In addition, Geolocation is used to find the users current location for one of the use cases. The Google Maps JavaScript library is included in the front-end html <script> tabs and can be obtained by CDN, the map is rendered and displayed once the browser is fully loaded.

Use cases

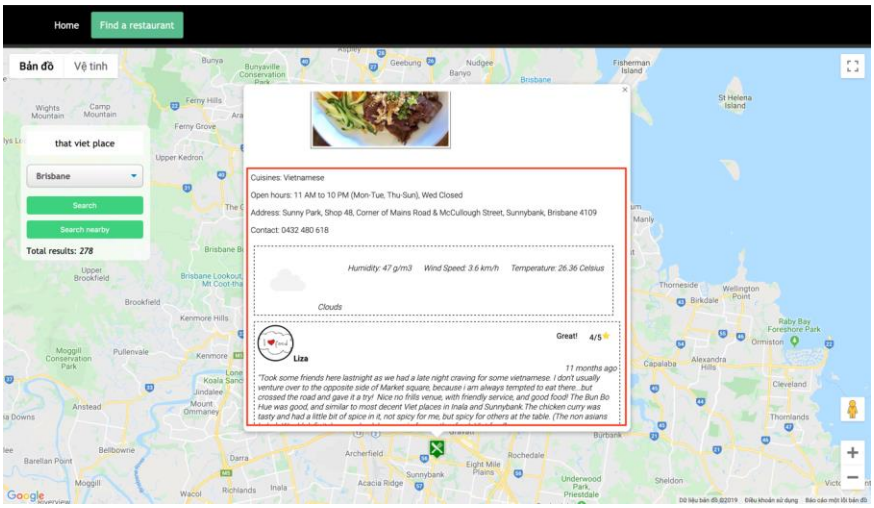
Use case 1

As a user, I want to search for a specific restaurant by name, so that I can see what their opening hours are and the current weather at the location.

This user launches the search page of Restaurant Finder, wanting to search for a restaurant named “That Viet Place” as they typed the name into the search and click Search. The results are then presented as below.



The map will zoom and centre at the first result that it finds. The user then clicks on it and reads the weather forecast as well as the opening time of the restaurant.

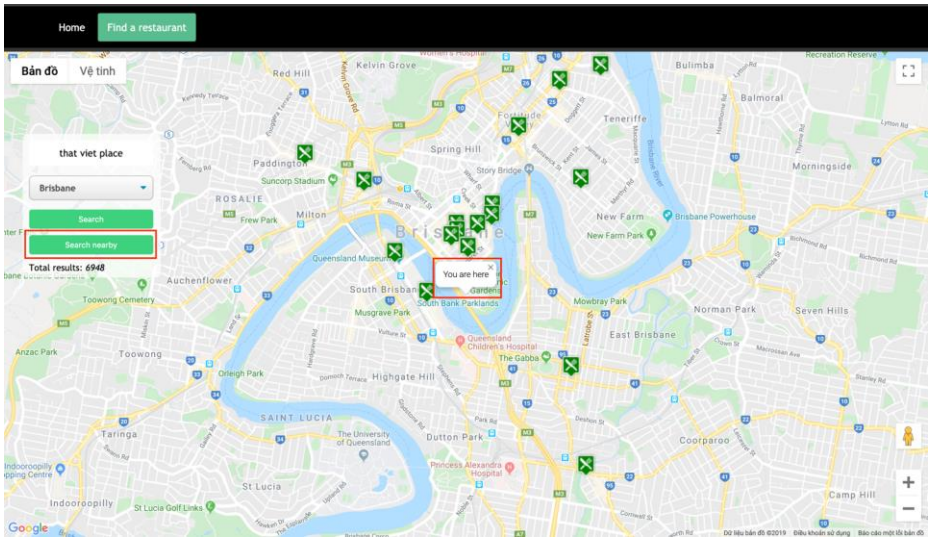


After that, the user can zoom out and check out the other similar restaurants

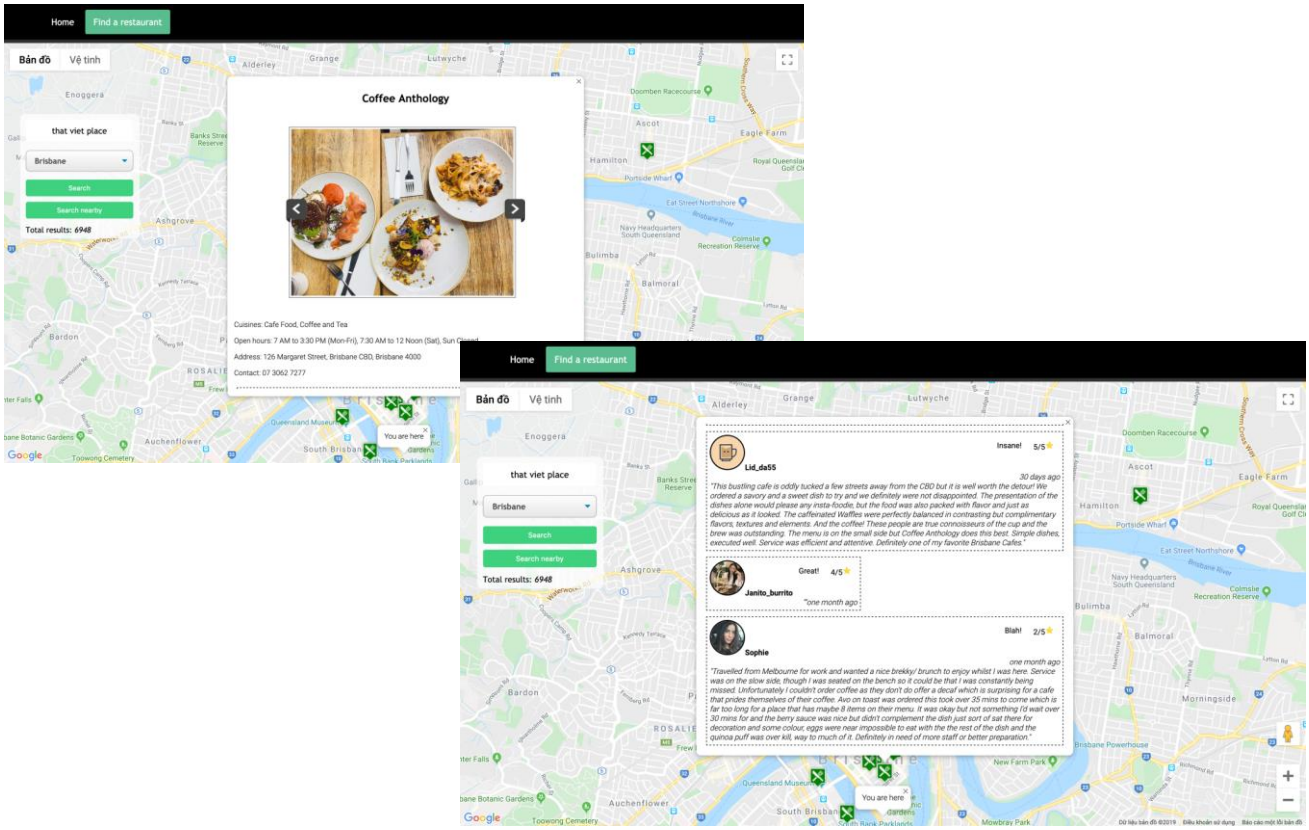
Use case 2

As a user, I want to search for a restaurant that is near my location, so that I can find a decent restaurant that is open and has good ratings.

This user navigates to the search page of Restaurant Finder and clicks the Nearby button in the search box. A popup box will then appear asking for permission to share your location for the browser, the user clicks Allow and is then presented with the results below.



The user can then click around and choose whichever restaurant is the best to have dinner at by looking at the reviews.



Technical breakdown

Architecture

Context diagram

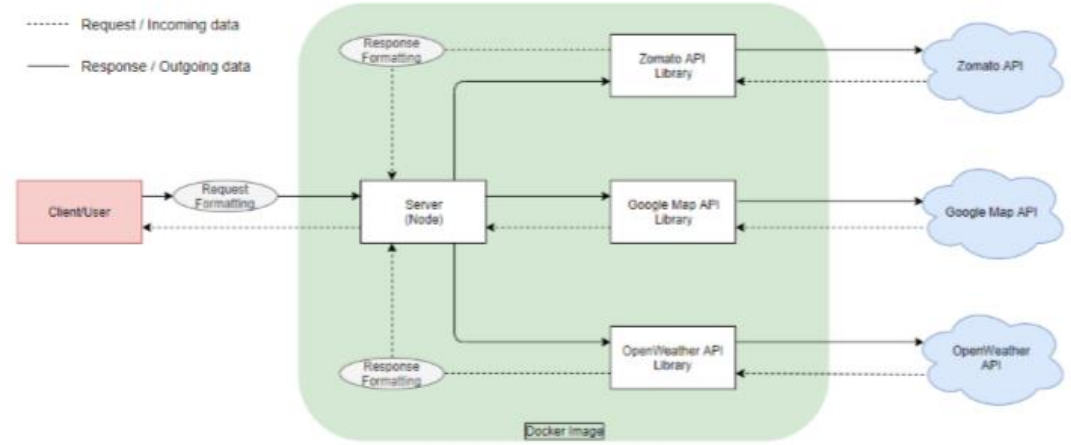


Figure 1 Flow chart of how data is handled

As the objective of the application is to handle the data and most of the processing in the back-end then serve it to the front, **Figure 1** shows that the request made by the user is initially handled by the server, which then formats a corresponding request to send to the correct APIs, GET the response and finally filter the important details in the back-end which are then sent back as formatted results to the browser. The browser will display the results on the map with decorations.

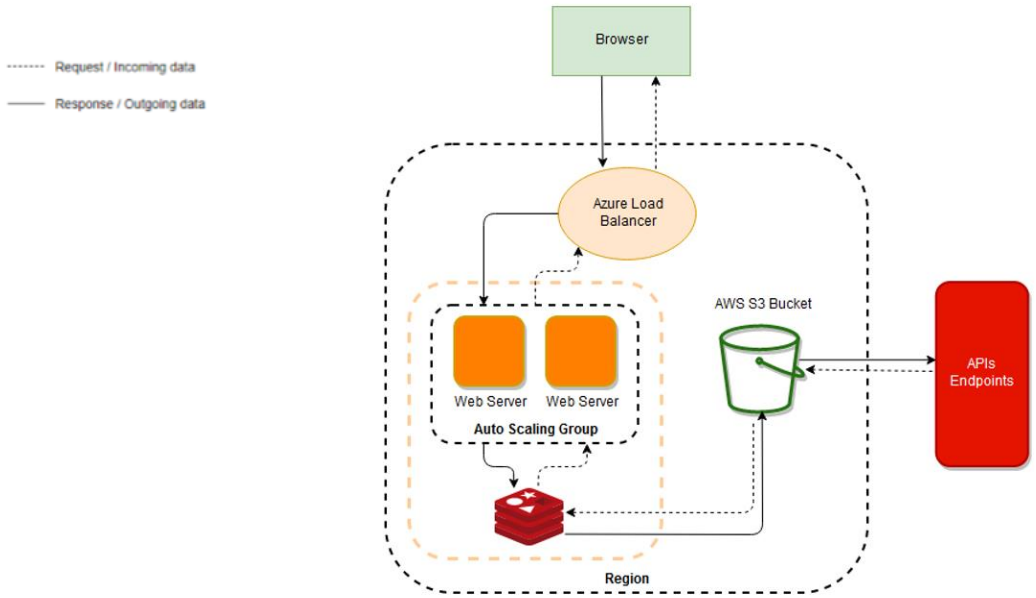


Figure 2 Cloud Architecture

Figure 2 demonstrates the flow of the cloud application architecture, where the browser resembles the user making requests to the cloud instance. These requests will be dealt with by the load balancer which will serve instances for the browser. Initially, the response of this request will be checked in the Redis cache. If the data is not found, the process moves on and checks the long-term storage, AWS S3. At this point, if the query is still not found, the server will send a GET request to the

relevant API. The response will then be filtered, saved to the Redis and S3 storage, in case the same request might be initialed again. This will speed up the response time and provide a better user experience. Once the formatted results are saved to the storage, they are sent to the browser.

In situations where the instances are loaded with requests in a short amount of time, the load balancer will initiate the scaling up process for the web server as the scaling mechanism is based on the total average network out. This will be discussed in detail in a later section. The health probe of the load balancer has been set to 60 seconds per interval, as well as having an unhealthy threshold of 5 consecutive failures. In addition, port 80 is exposed to the internet, and port 3000 exposed to the web server.

Client

Upon the initial launch of the application, the client will make a request to the server, where the page is then served. After that, any interaction by the user to the client such as searching for restaurant, clicking a marker and viewing images will be sent to the back-end using JQuery AJAX GETs. After receiving the process payload from the server, the client-side then display it on Google Map.

Server

Upon receiving GET requests from the front-end, the server process matches the relevant request and makes GET requests to external APIs using middleware Axios. Once it has received the responses, the server filters and processes the results then sends them back to the client-side, where the results will be displayed. Routes are created to handle appropriate and sufficient code to manage and process incoming AJAX requests and outgoing GET request to APIs. Some middleware are used for the server-side including Axios, Helmet and Morgan for logging.

```
[Mon, 16 Sep 2019 07:35:54 GMT] "GET /" 200 0.267
[Mon, 16 Sep 2019 07:35:55 GMT] "GET /search" 200 0.354
[Mon, 16 Sep 2019 07:36:00 GMT] "GET /search/full?lat=-32.2602&lon=148.017" 200 1065.000
```

Figure 3 Logging format for the backend

Response filtering

The Zomato and OpenWeather API responses are not ready-to-use material and quite lengthy in terms of depth. Besides removing the irrelevant information from the responses, the server also restructures the JSON responses, making it simpler and shorter in depth. This will make reading the information on the client-side easier. The figure below shows the comparison between the raw responses from the APIs to the server versus the processed response from the server to the client-side.

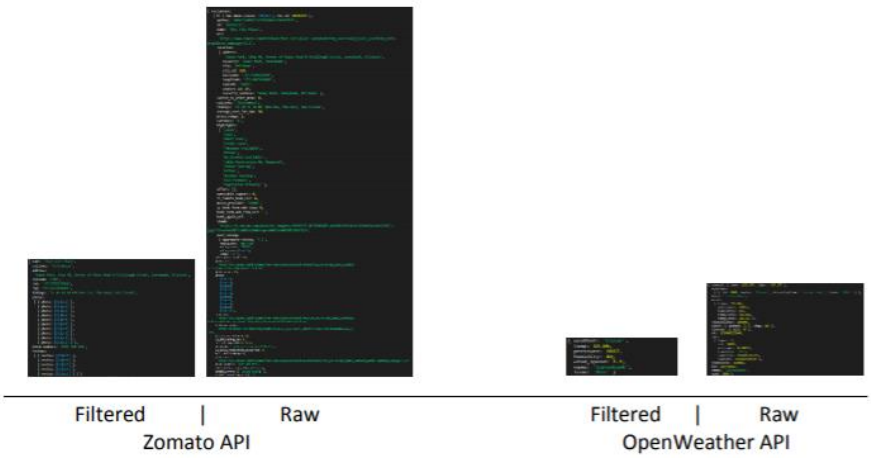


Figure 4 Comparison between fileterd and raw data

Persistence

For the persistence part of the assignment we are using Redis for cache storage and AWS S3 for long term storage.

If a user decides to search for a specific location the Redis key will be saved as:

```
redisKey = `location:${req.query['lat']}-${req.query['lon']}`;
```

Otherwise if the user performs a general search the key will be stored as:

```
redisKey = `restaurant:${req.query['q']}-${q3}`;
```

The application then searches for existing results in Redis and if these exist, they are served from Redis. If they don't, it searches S3 instead. If found, they are served from S3 and saved to the Redis Cache. If the result isn't found in either Redis or S3, it is served from the Zomato API and stored in S3 and Redis.

The persistence can be seen when using the app as when the user initially selects one of the markers on the map the information will load slightly slow as it is being served from the Zomato API, but when they click on it again it will be loaded a lot quicker.

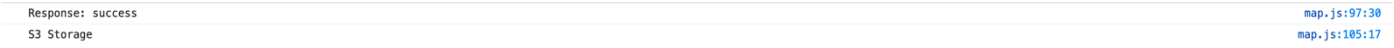
This can also be seen in the network tab when we click on the web development option in the browser. First, we search for "that viet place" in the application. The image below shows the response time and the size of the response which are 2932ms and 282.57 KB, respectively.

Figure Response time of a new query

We try to search for the same query the second time as seen in **Figure 6**. Note that the response has dropped down to 19ms and the size is cached.

Figure 6 Response time of an old query

In order to check whether it is stored in S3 as well, we use 'flushall' command in the redis terminal to clear all the cached data.



Scaling

Scaling is done using Microsoft Azure. A load balancer is set up to scale based on the average amount of network out as it represents the amount of data consumed by the users and is a good indicator of how many people are using the application. The reason it is scaled according to the average is that we want to track and scale it based on stable and consistent data, to avoid having it scale due to random spikes and only when necessary. The application is to scale down at 2MB (average) and up at 4MB (average). This scaling criteria might seem low. However, if this number is consistent over a longer period of time, the sum of the total network out would be much greater.

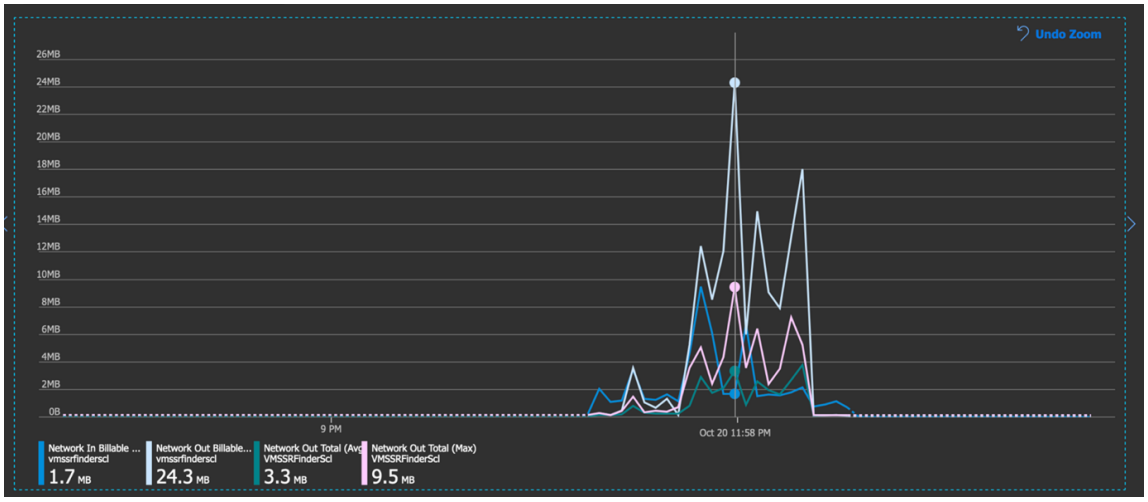


Figure 8 Line plot of the network usage during the Postman test

Note in **Figure 8** that when the average network out total peaks at 3.3 MB, the value of the sum of network out billable is peaking at 24.3MB. This means for short session usage, the application will scale down, and during a long session with consistent requests being made, it will trigger the scaling up process.

The scaling performance was tested using Postman, which sent 64 different requests iterating 50 times, with each request averaging about 200 kB.

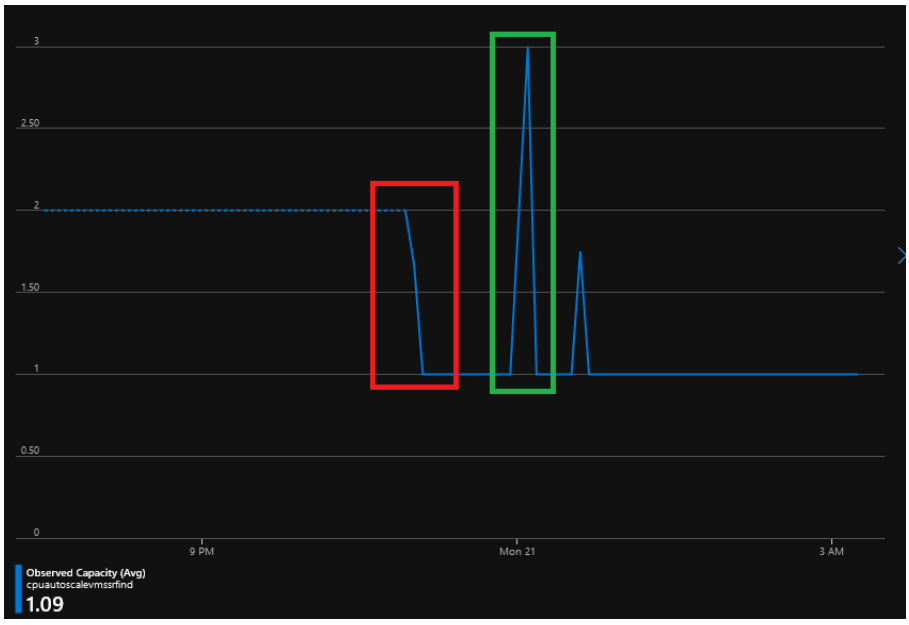


Figure 9 Second Scaling tests

Figure 5 shows the process of scaling up and down during the second scaling tests. The red rectangle marks the scaling down when the system is freshly initiated, it scales from 2 down to 1. For the green rectangle, it pinpoints the scaling up process when the Postman tests were running, calling approximate 2 unique requests per second and each request is about 200 kilobytes in size.

Test plan

Task	Expected Outcome	Result	Screenshot/s (Appendix A)
Search for restaurants by name	Displays results on map	PASS	01
Empty search keys query	Alert box popup asks to fill out keywords	PASS	02
Search for nearby restaurants (allow location)	Displays results on map	PASS	03
Search for nearby restaurants (not allow location)	Alert box popup asks to allow to proceed	PASS	04
Click on markers	InfoWindow pops up with relevant details	PASS	05
Click on next button of the photo slider in InfoWindow	Next photo appears, current photo disappears	PASS	06
Click on previous button of the photo slider in InfoWindow	Previous photo appears, current photo disappears	PASS	07
Click on the profile picture of the reviewer	Redirects to the reviewer's Zomato profile	PASS	08
Search for a new query	Response in approximately 2000ms – must be from Zomato API	PASS	09
Search for the same query as Case 09	Drastic drop in response time – must be from Redis Cache	PASS	10
Use Postman to generate requests	Application scales up	PASS	11
Not send any requests to application	Application scales down	PASS	12
Click on new marker	Response in seconds – must be from OpenWeather API	PASS	13
Click on the same marker second time	Drastic drop in response time – must be from Redis Cache	PASS	14
Search for cuisines in other cities using the dropdown list	Results should be corresponding to each city	PASS	15
No results	Alert pop-up appear	PASS	16

Difficulties

S3

One difficulty regarding the use of AWS S3 is that the credentials for Amazon Web Services Educate accounts reset every 3 hours, so if the application was to be continuously running for longer than this it would be needed to use another long-term storage service.

Zomato API

One of the drawbacks of using Zomato API is that it only provides 20 results per request and the maximum results it can shows for a unique query is 100 results. Also, for each restaurant, Zomato only provides 5 to 7 reviews. This makes the option of generating load by processing the results very limited.

Extensions

mongoDB

As stated in the earlier section, S3 has a limitation for AWS Educate users as the credentials key refreshes every session and each session lasts about 3 hours.

```
AWS Access
    Session started at: 2019-10-22T09:29:07-0700
    Session to end  at: 2019-10-22T12:29:07-0700
    Remaining session time: 2h59m54s
```

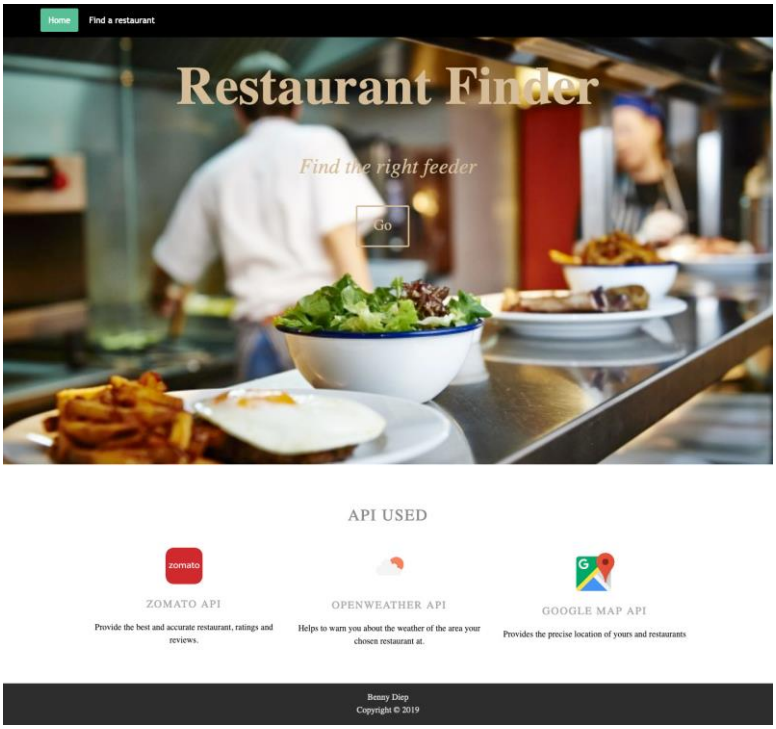
The only way to fix this limitation is either to upgrade the AWS account or use another long-term storage service such as mongoDB. One of the advantages of using this is its ability to store the items as JSON format instead of having to convert from JSON to text in order to save them in S3.

Addition of processing usage

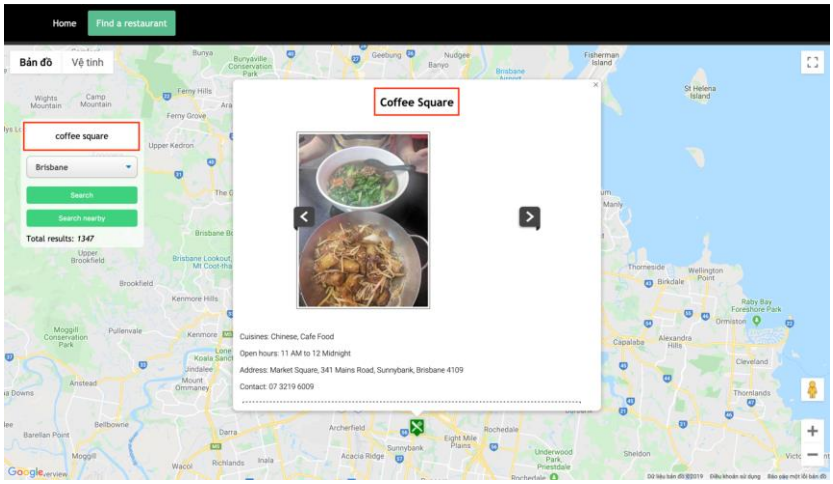
It is clear that the web application is not a CPU power hungry application. During the scaling tests, the max averaged percentage CPU usage is around 3.38%. This means that we could maximize the value of the scaled instances' CPU power by implementing additional features such as calculating the top 10 best rated restaurants by cuisine and area using the same data from the APIs. And plot some graphs using d3js.

User guide

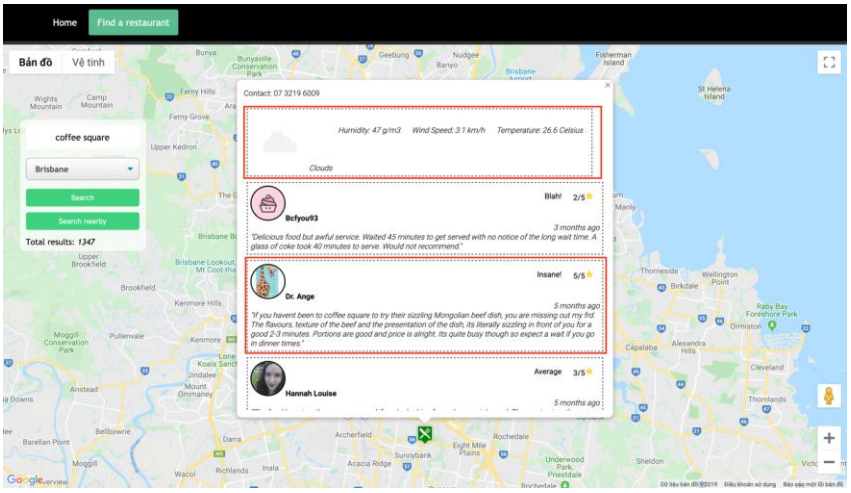
When launching the index page, the user can see a short description of the application as well as which APIs are used. The user can click the button Go or Find a restaurant in the navigation bar.



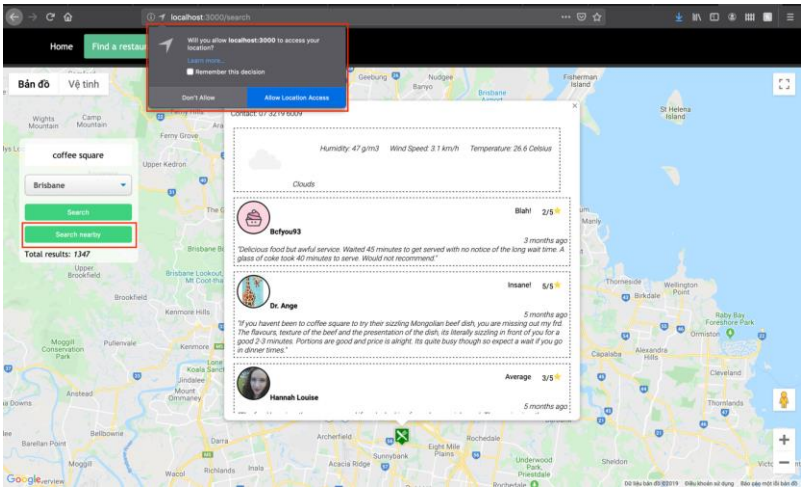
User is then redirected to a map with a search bar and two buttons. The user can enter the name of the restaurant that they want to search for. In this case, its “Coffee Square”. The results will be shown as below.



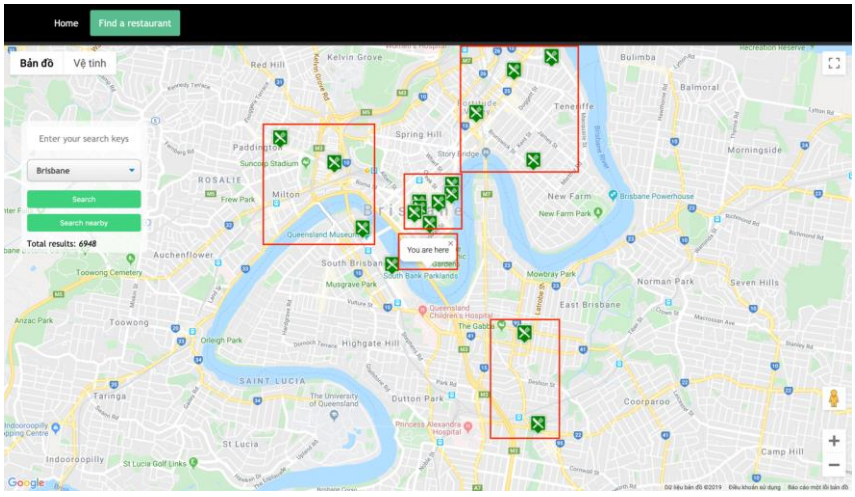
They can now scroll down the InfoWindow to check the current weather at the location as well as reading all most recent reviews of that restaurant.



Now if the user feels a bit lazy, they can click Nearby button to find out the location near the location they are at. A prompt will pop up and ask for their permission to use their location for the search. Note that the user has to click *Allow* in order to proceed.

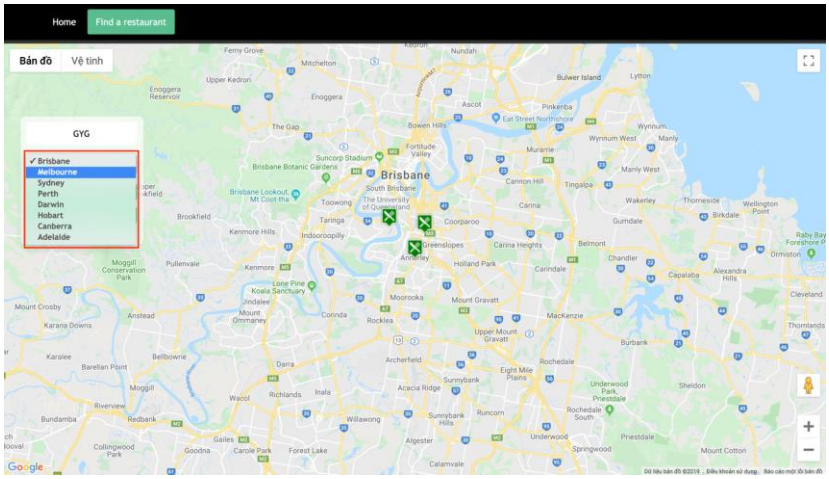


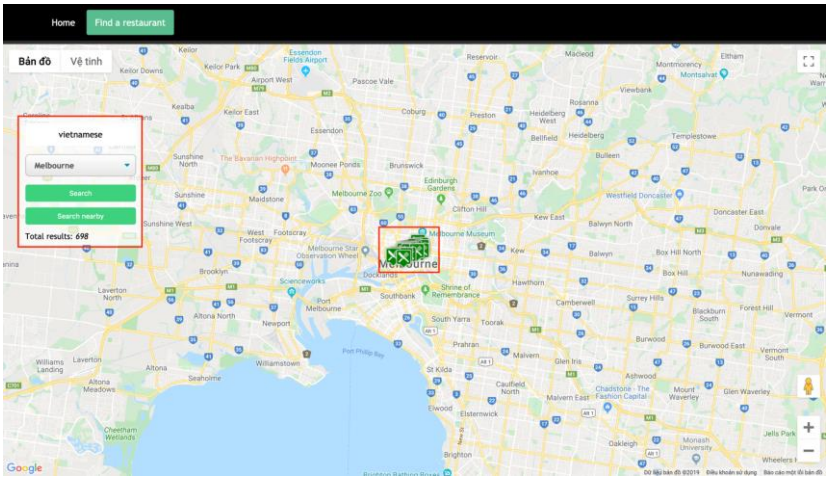
Once they have clicked Allow, the request is sent and might take a few seconds for it to come with results as shown below.



Once all the results shown, the user can continue check new places without moving an inch and might be able to find a hidden gem.

Note that now it is possible to search for other cities as well such as Melbourne, Sydney, etc. This would increase the accuracy for each request.



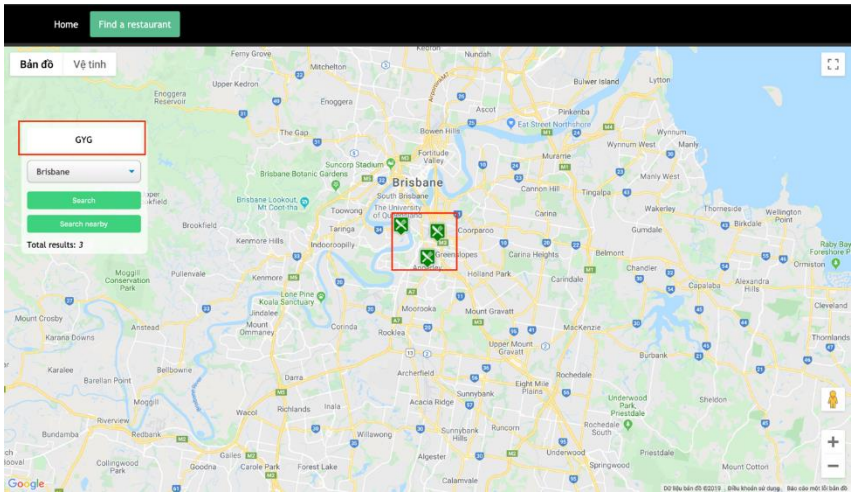


References

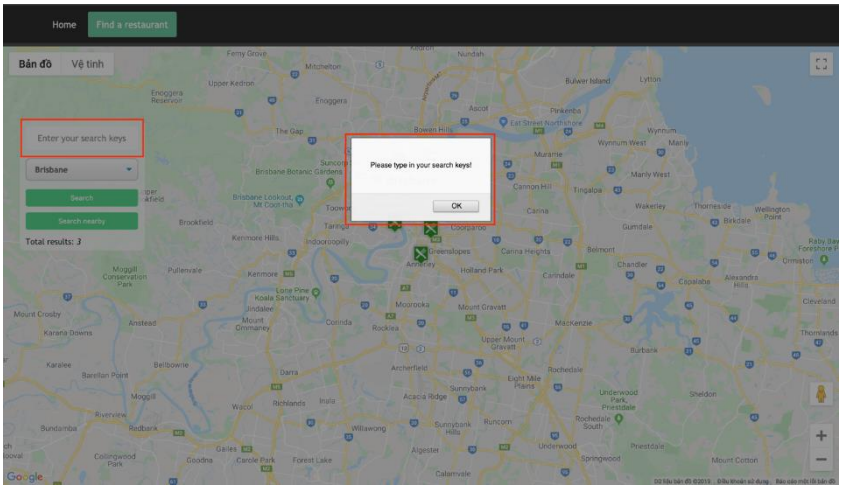
Diep, T., T. (2019). *CAB432 Assignment 1 Mashup API Web Application.* Unpublished manuscript, Queensland University of Technology.

Appendix A: Test Cases Screenshots

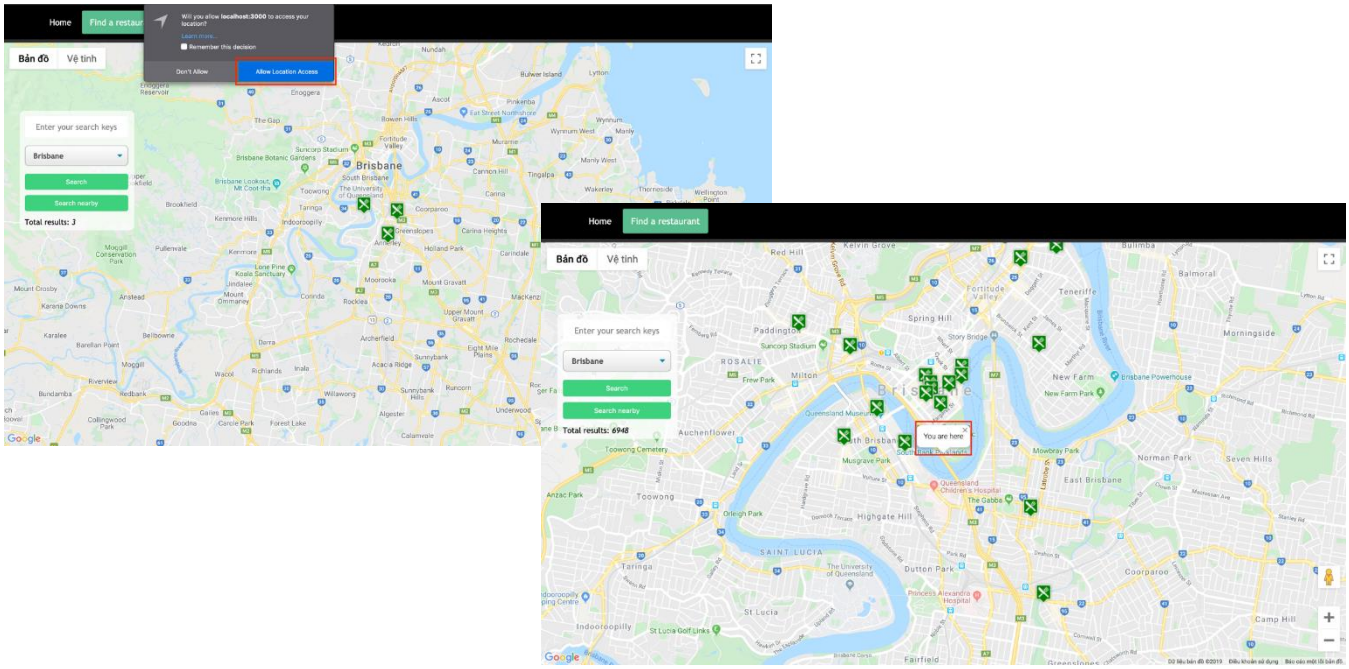
Case 01:



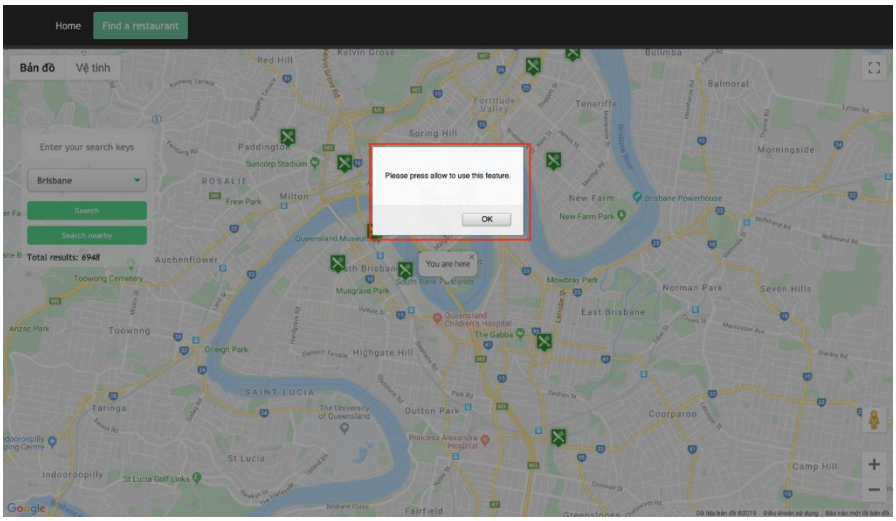
Case 02:



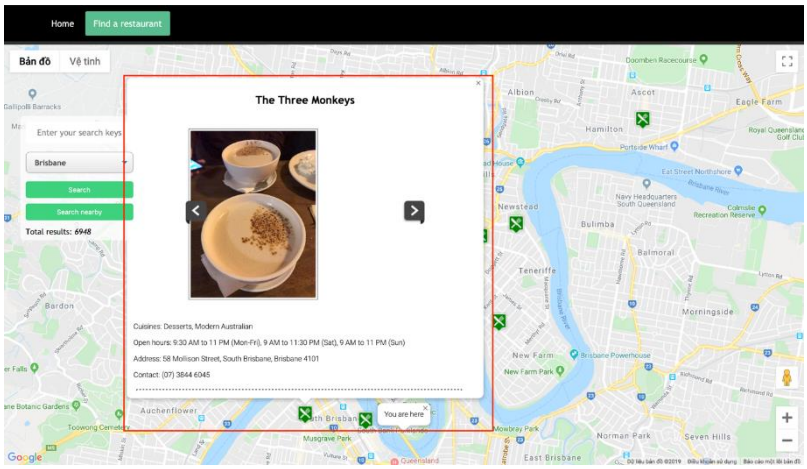
Case 03:



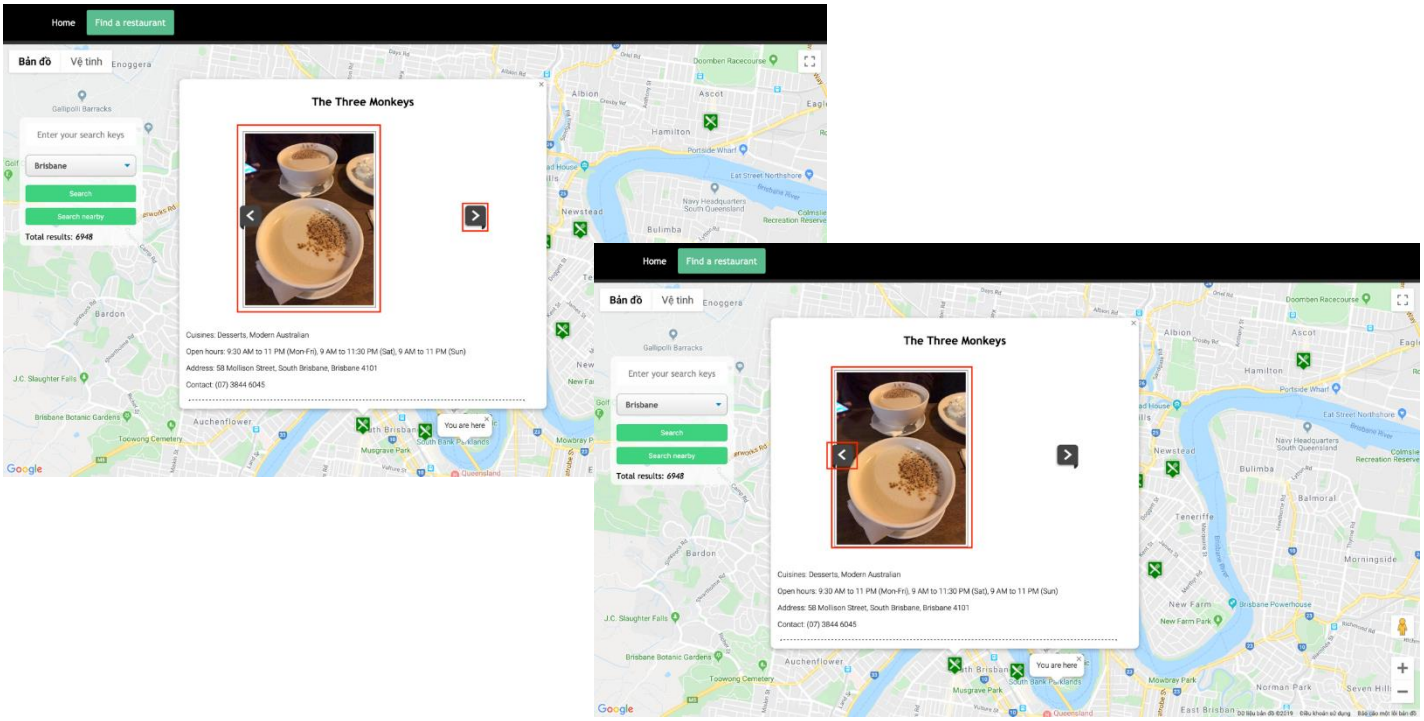
Case 04:



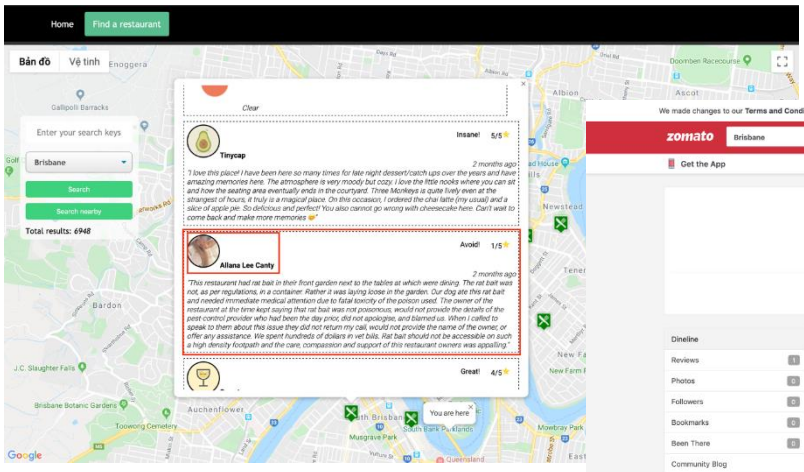
Case 05:

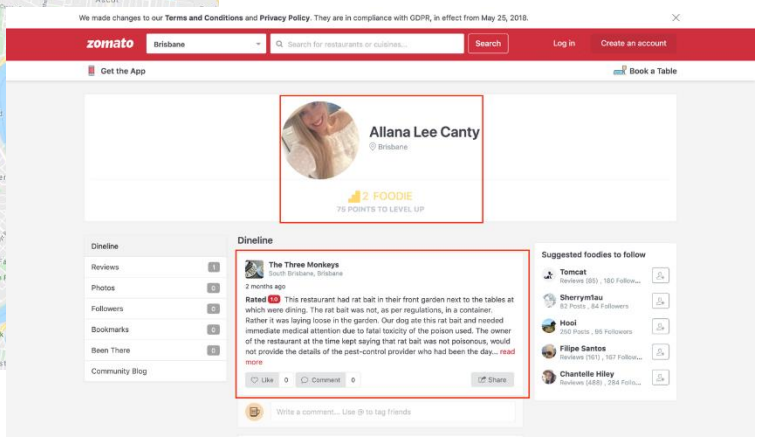


Cases 06 & 07:

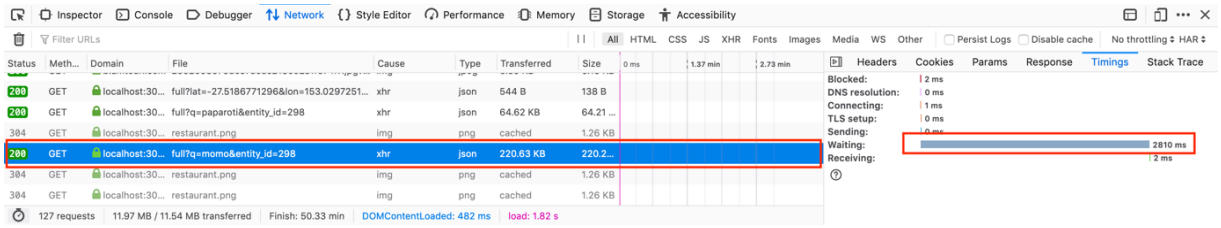


Case 08:

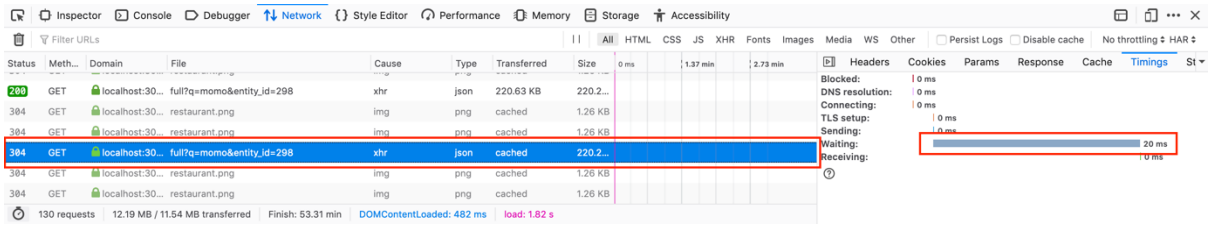




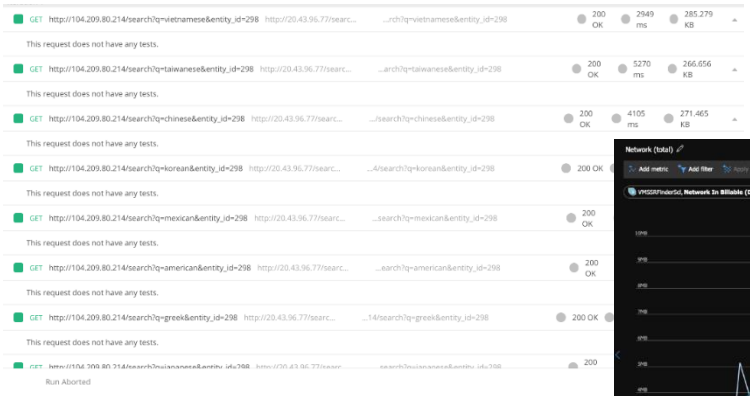
Case 09

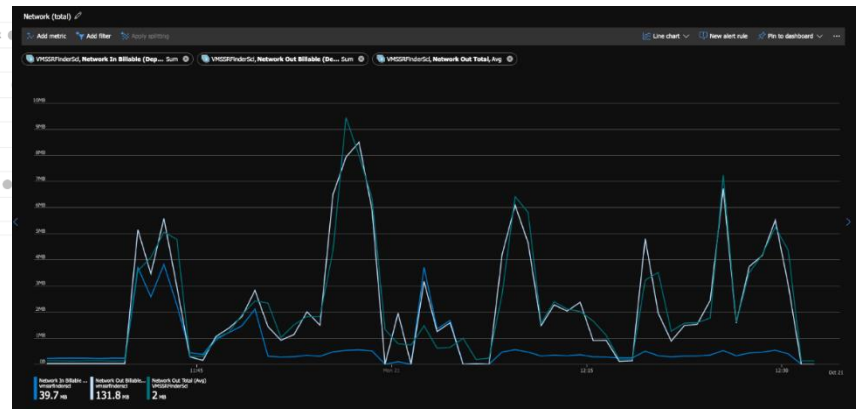


Case 10



Cases 11 & 12

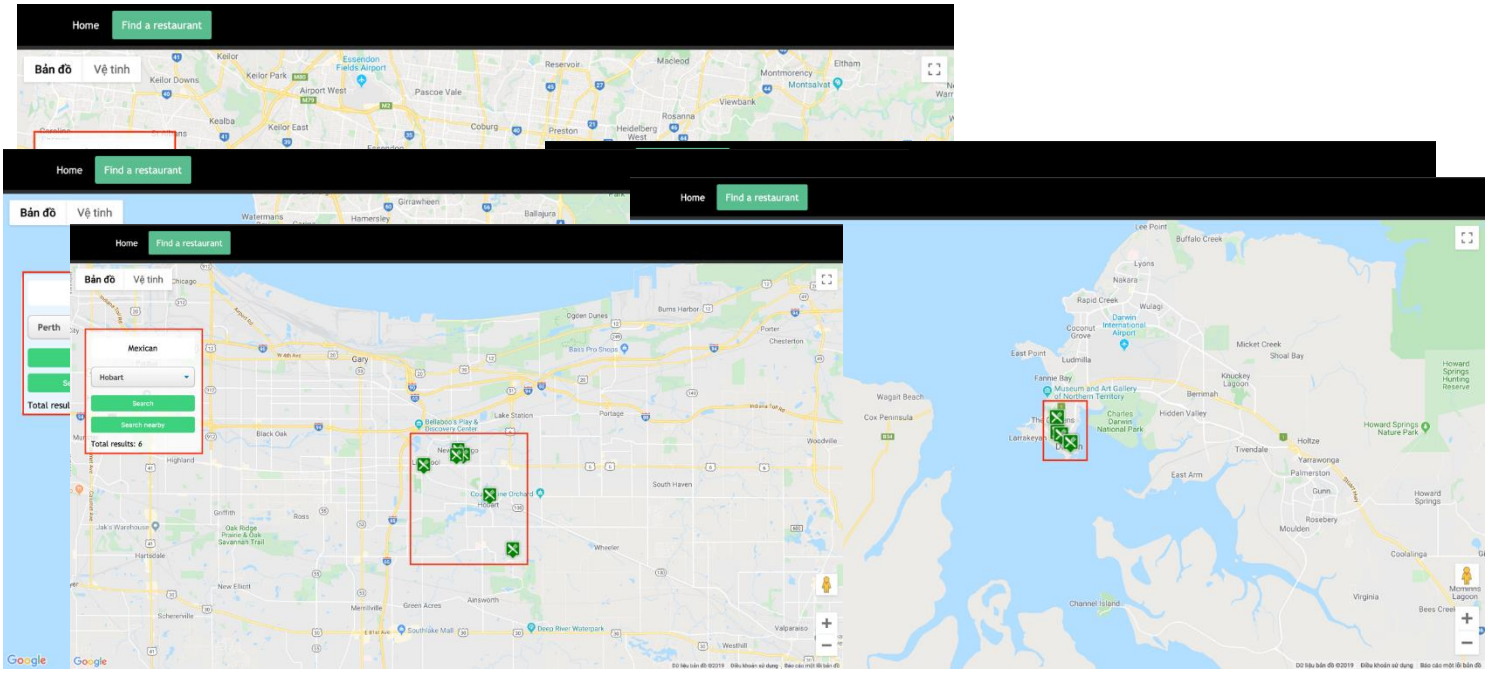




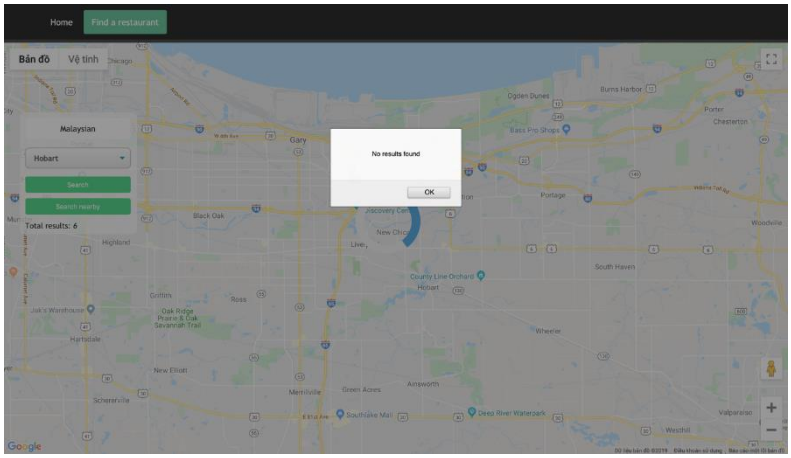
Cases 13

Case 14

Case 15



Case 16


Home
Find a restaurant
Bản đồ
Vệ tinh
Malaysian
Hobart
Search
Search nearby
Total results: 6
No results found
OK
Google

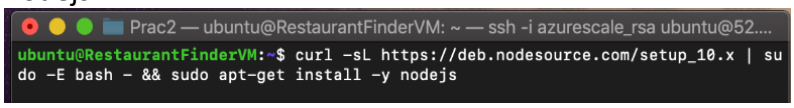
Appendix B: Deployment Instructions

The first step of the deployment is to push the web application onto a virtual machine service such as Azure, AWS, etc. For this assignment, Azure services are used. Most of the settings for this are the default, in this case 1vcpu and 2GB of storage was used while running Ubuntu 18.04 LTS.

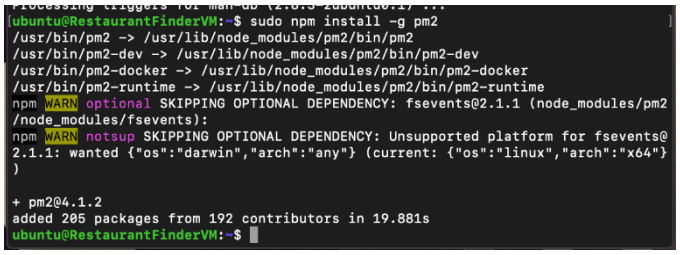
The method in **Appendix C** is used to obtain the public key to run the VM in the terminal.

The next step is to install nodejs and npm on the VM. Using the following commands:

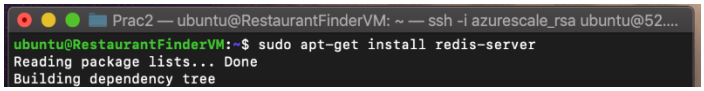
curl -sL https://deb.nodesource.com/setup_10.x | sudo -E bash - && sudo apt-get install -y nodejs



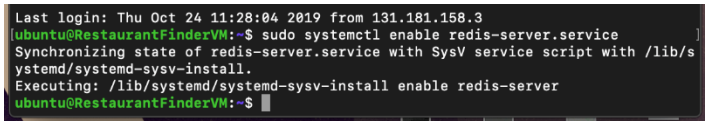
Next is installing pm2 using the following commands:
sudo npm install -g pm2



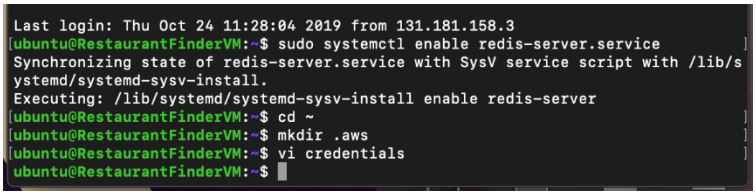
After that, we will need to install Redis using the following command:
sudo apt-get install redis-server



Next thing is to enable Redis-server running on system boot using the following command:
sudo systemctl enable redis-server.service



Next is creating .aws credentials files for AWS S3 services on the VM. Follow the step in the screenshot below and paste your AWS keys in. Note that if this is an AWS Educate account, this key will refresh within the next 3 hours.

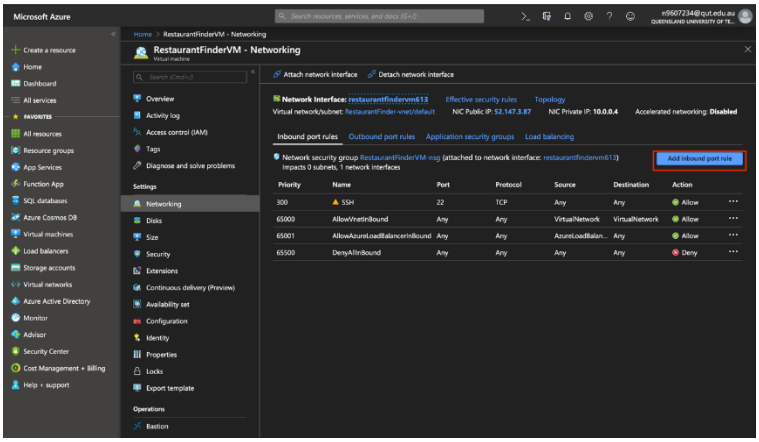


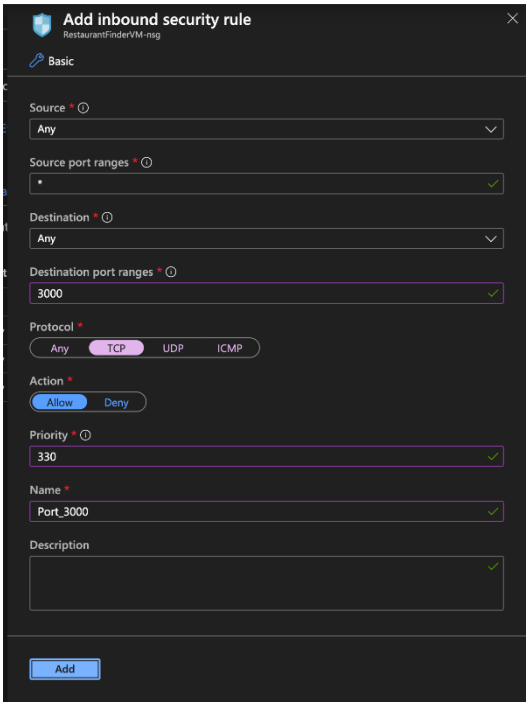
Next step is pulling the web application using git.

```
fatal: not a git repository (or any of the parent directories): .git
[ubuntu@RestaurantFinderVM:~$ git init
Initialized empty Git repository in /home/ubuntu/.git/
[ubuntu@RestaurantFinderVM:~$ git pull https://github.com/potatomato98/CAB432Assi
gnment2
[Username for 'https://github.com': potatomato98
[Password for 'https://potatomato98@github.com':
remote: Enumerating objects: 3296, done.
remote: Counting objects: 100% (3296/3296), done.
remote: Compressing objects: 100% (2412/2412), done.
remote: Total 3296 (delta 716), reused 3277 (delta 702), pack-reused 0
Receiving objects: 100% (3296/3296), 7.49 MiB | 4.14 MiB/s, done.
Resolving deltas: 100% (716/716), done.
From https://github.com/potatomato98/CAB432Assignment2
 * branch            HEAD       -> FETCH_HEAD
```

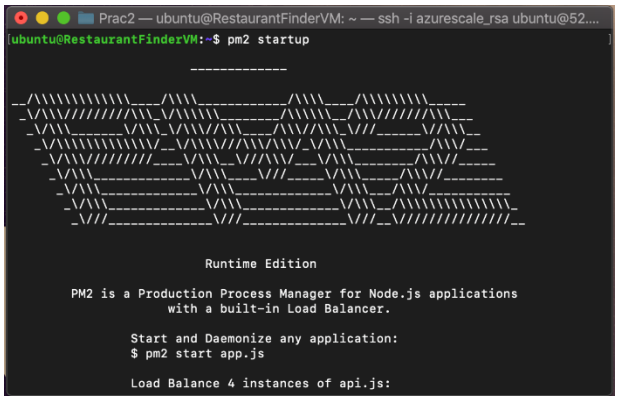
Before setting the application to run upon boot, we will have to install all the dependencies using npm install.

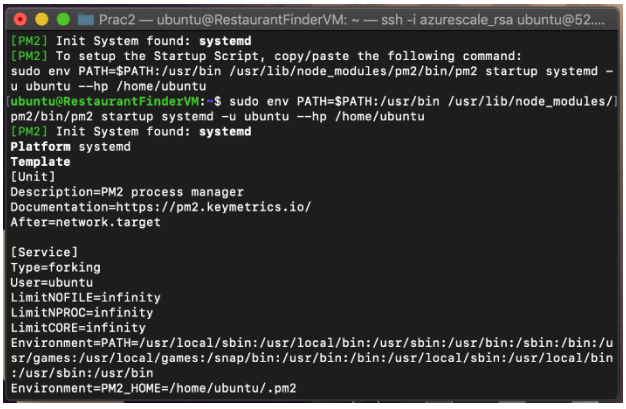
Before typing pm2 startup or running it using npm start, we will have to expose port 3000 on Azure for the VM. Under Networking tab, click Add inbound port rule. Modify as the screenshots then click Add.



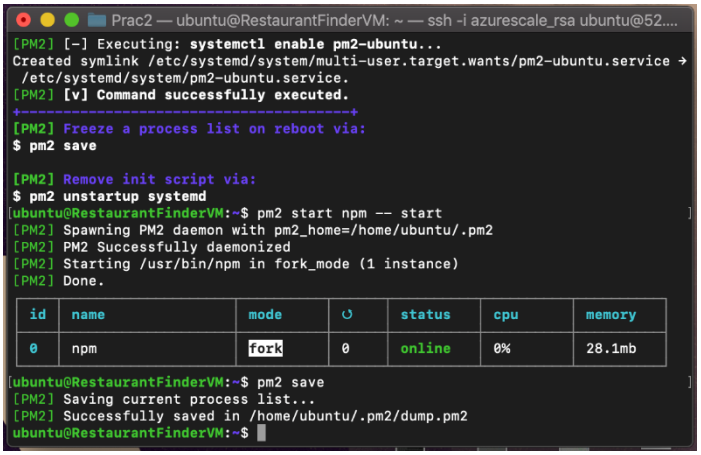


Once the port is exposed, we go back to VM terminal and type pm2 startup, then follow the steps.

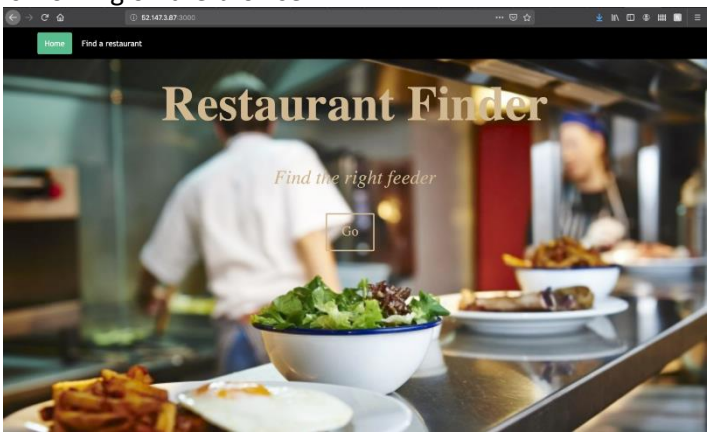




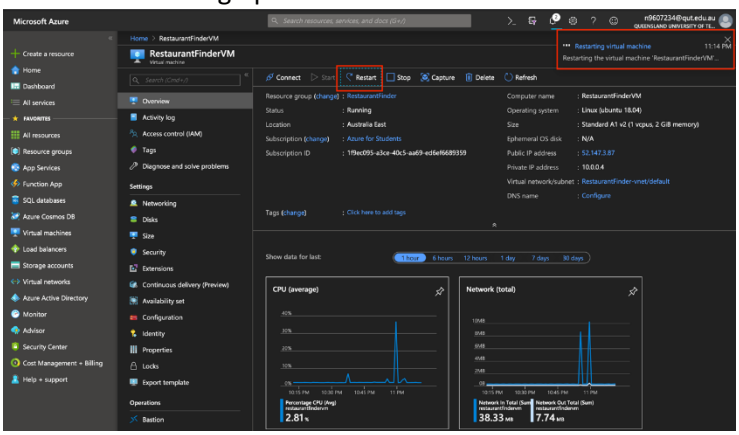
Following the screenshot below, we type pm2 start npm – start. Then type pm2 save to save the startup commands.

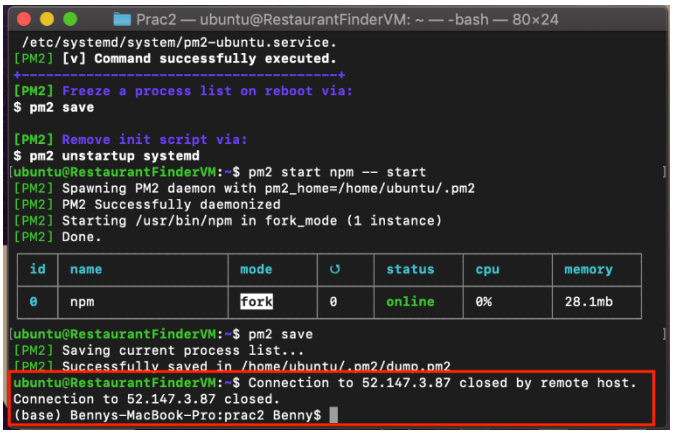


After that, check if it is working on the browser.



It works! Now we restart the VM, “after a few failures”, it should start running the app once it done booting up.



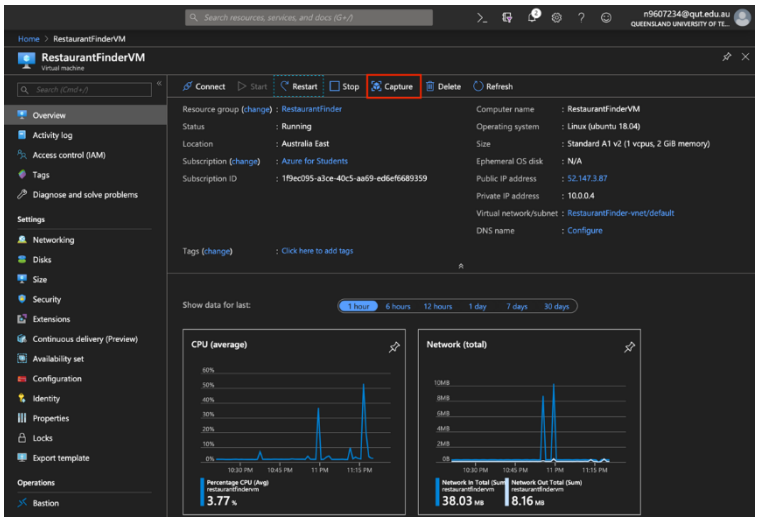


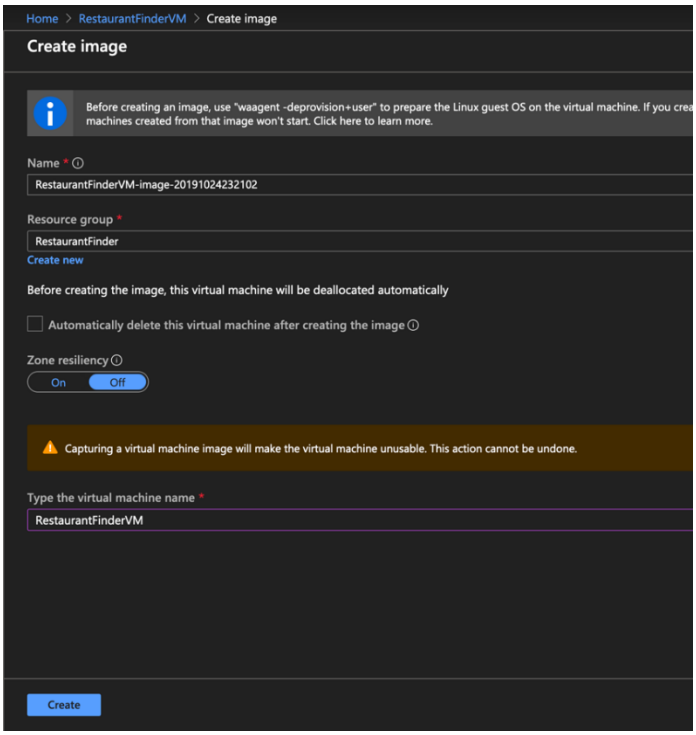
Now that it is running at boot, we abstract it as an image to later create the actual instances to scale. First, we log back to the VM terminal and type the following:

sudo waagent –deprovision

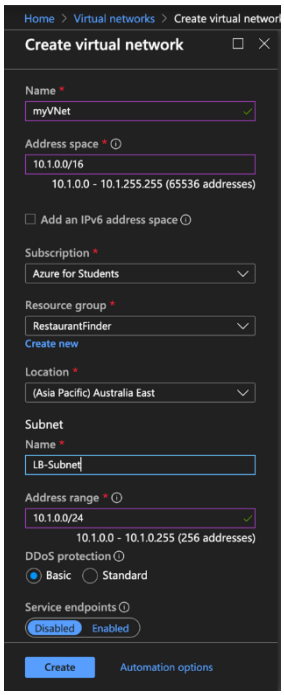
```
Last login: Thu Oct 24 12:35:41 2019 from 131.181.158.3
ubuntu@RestaurantFinderVM:~$ sudo waagent -deprovision
WARNING! The waagent service will be stopped.
WARNING! Cached DHCP leases will be deleted.
WARNING! root password will be disabled. You will not be able to login as root.
WARNING! /etc/resolv.conf will NOT be removed, this is a behavior change to earl
ier versions of Ubuntu.
Do you want to proceed (y/n)y
ubuntu@RestaurantFinderVM:~$
```

Now, we go back to Azure, click Capture on the VM, then type the name of the VM for confirmation.

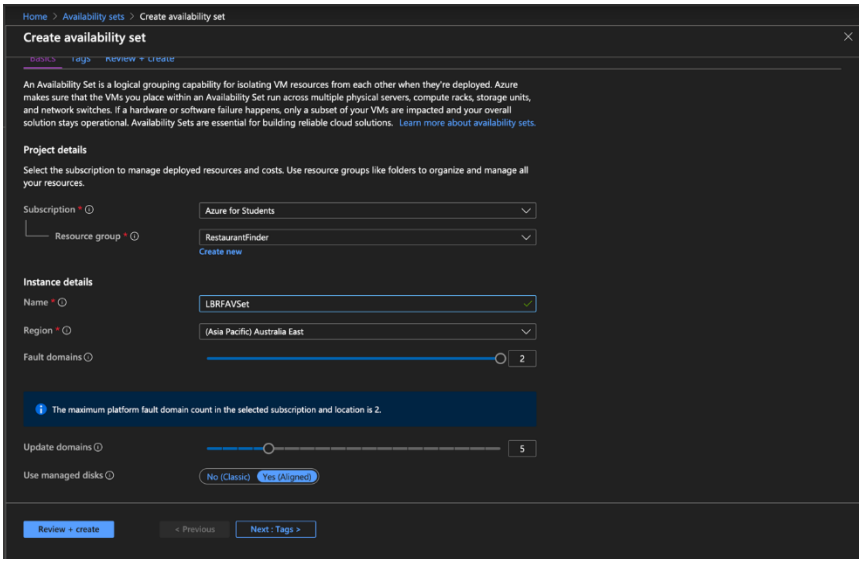




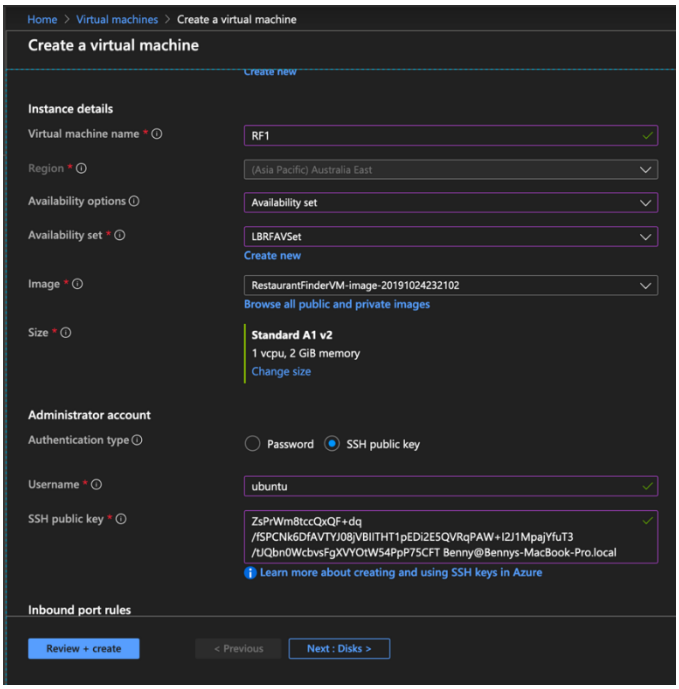
Once this is done, we can delete the VM for more spaces. Before creating the instance pool, we will have to create a virtual network as shown below:

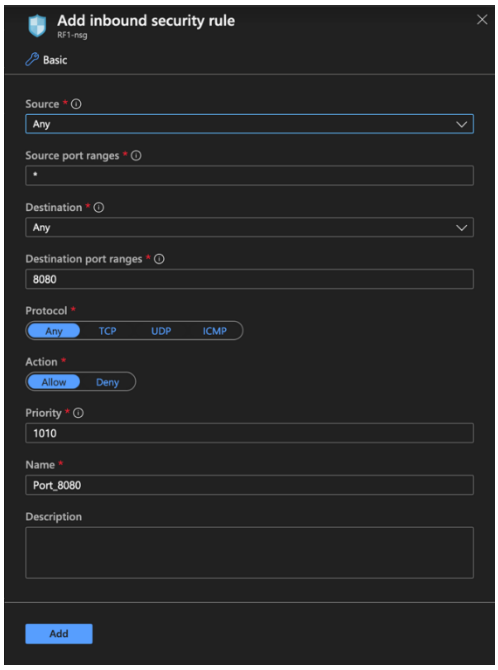


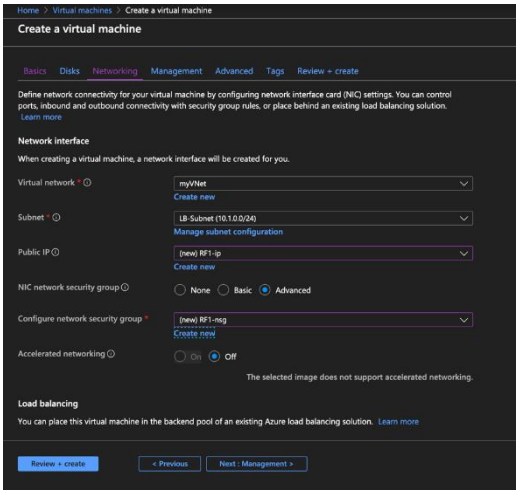
Now, we create an Availability Set and Add VMs:

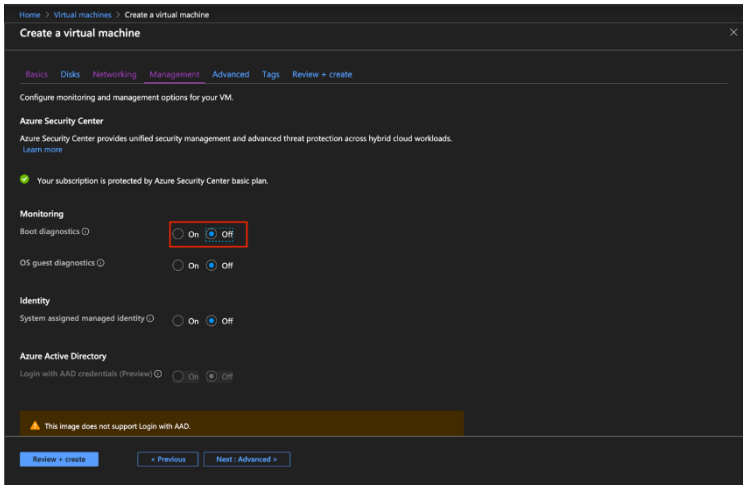


Now we create 2 VMs using the image created above.

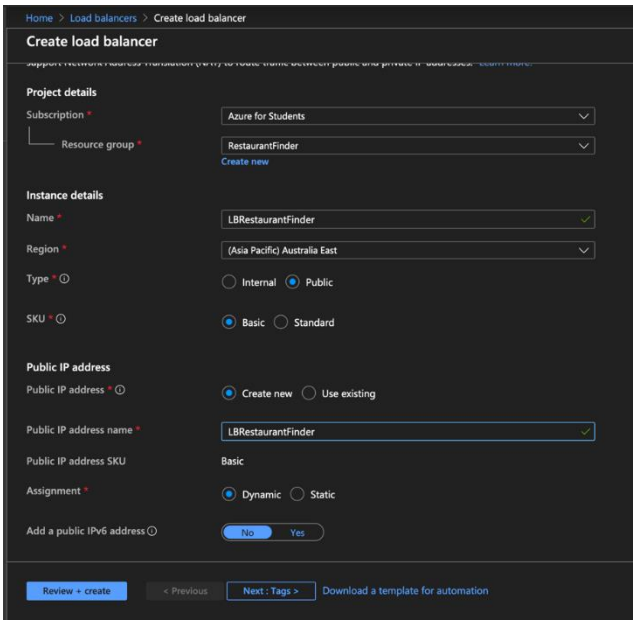




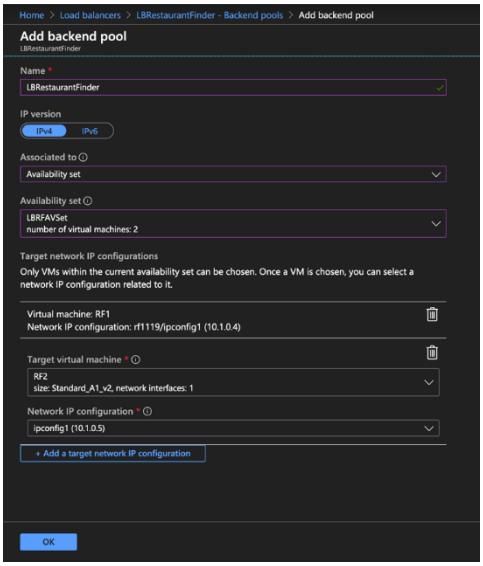


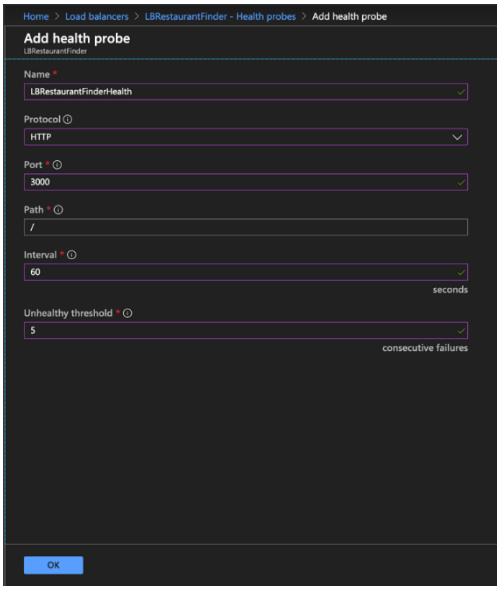


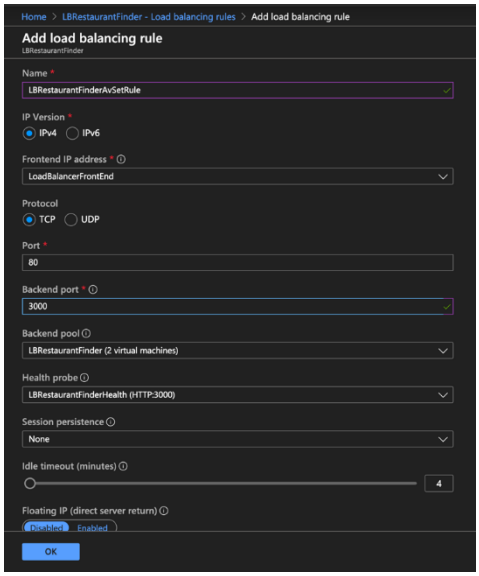
For the second VMs, we repeat the same steps with the same settings as above. The next step would be to create a load balancer.



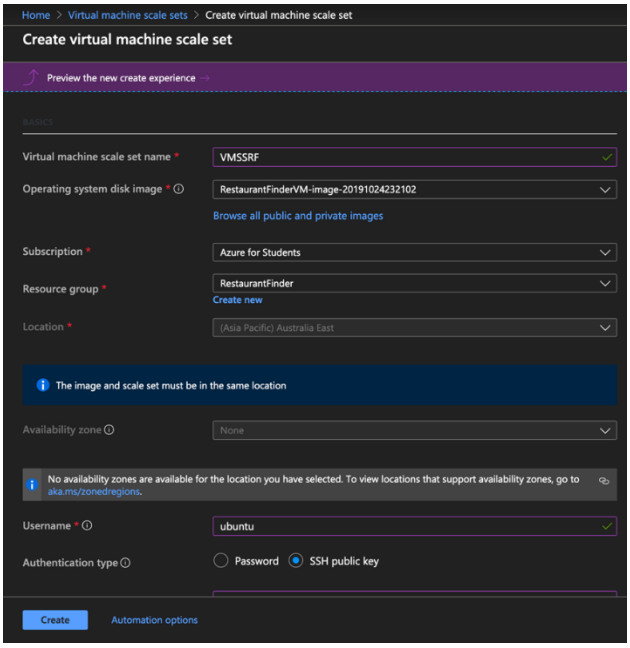
After created the load balancer, there are a few things we need to setup that is the backend pool, health probe and load balancing rules.

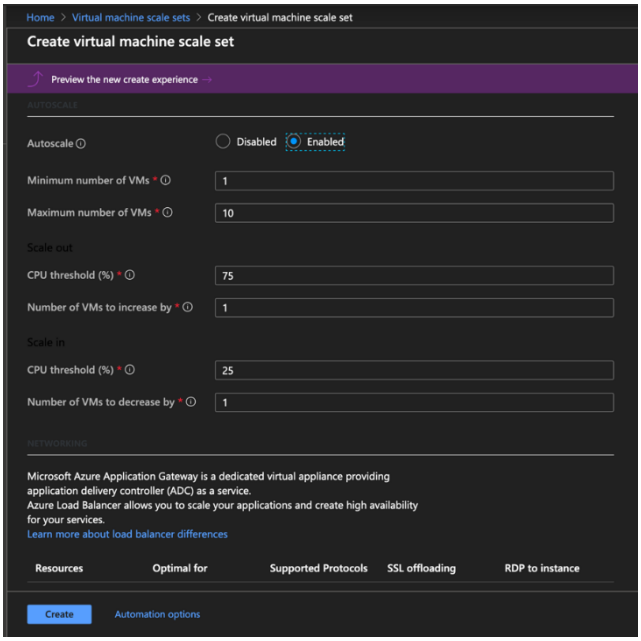


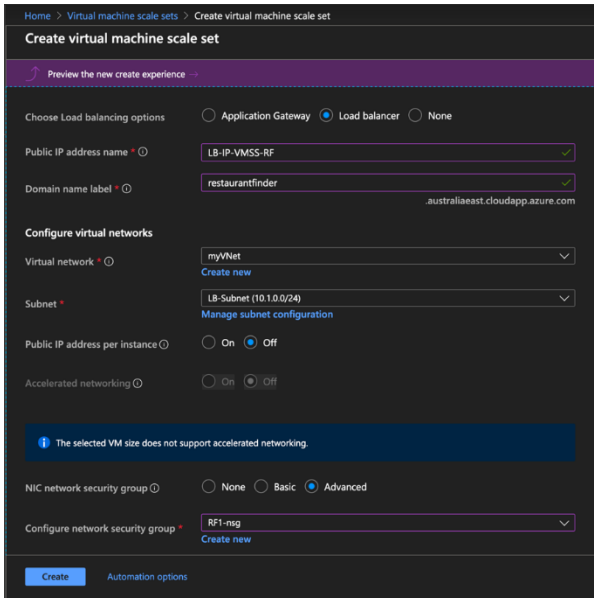




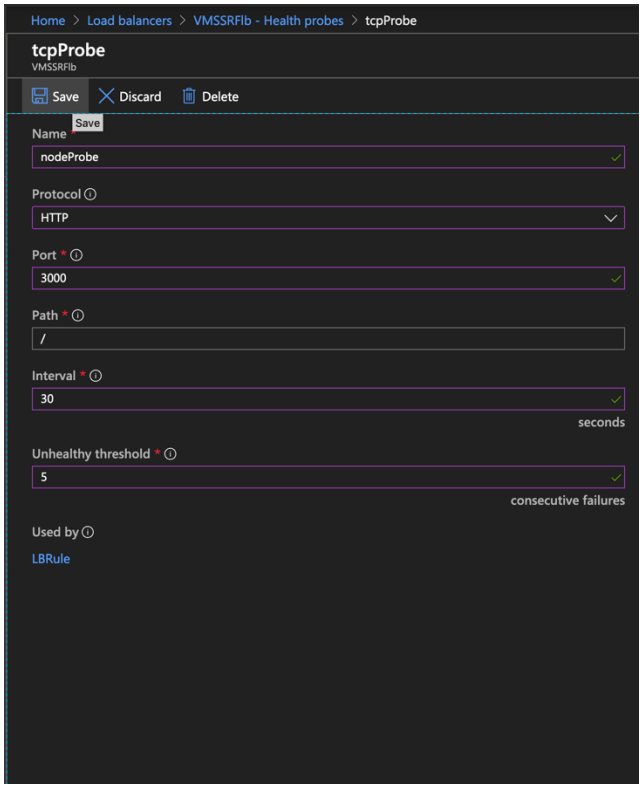
The next thing would be creating a Virtual Machine Scale Set.

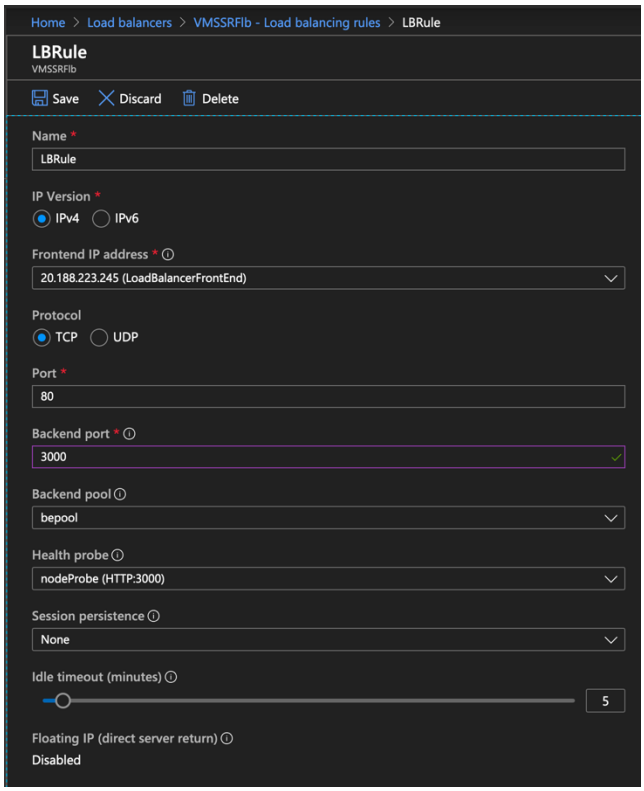




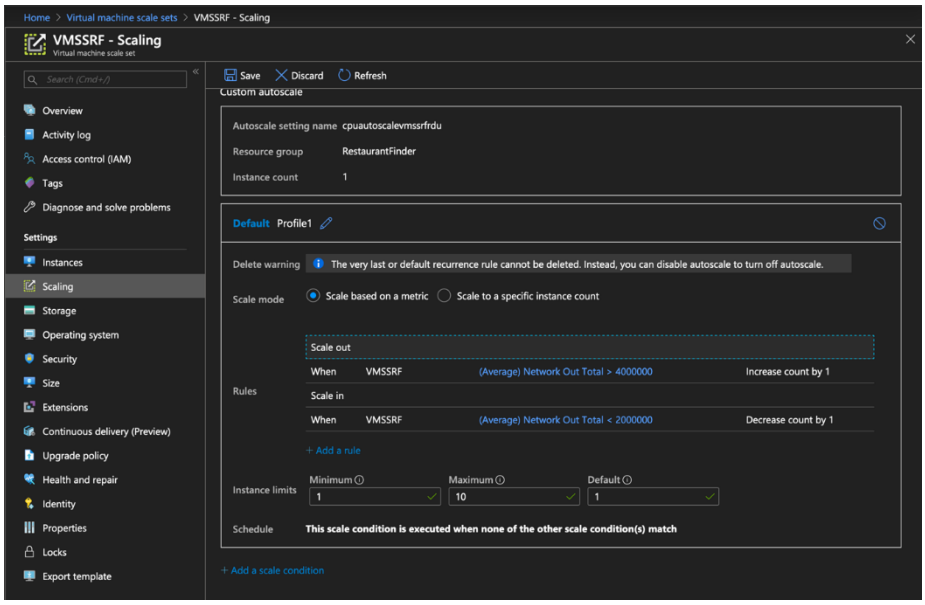


The next steps will be setting up the health probe and load balancing rules for the new load balancer for the VMSS.





The final step would be changing the scaling condition for the VMSS. For this web application, we want it to scale up when the average of the total network out is more than 4MB and scale down when its lower than 2MB.



That was the last step. Now the app should scale accordingly. And we can delete the 2 VMs can we created previously.