Nomad Phase III Project Report

Project Name: Nomad
Github: https://github.com/elvinhung/SoftwareLabApp
Canvas Group: Morning-3
Phase III Lead: Alex Kim
URL: software-lab-travel-app.s3-website.us-east-2.amazonaws.com

Name	**Email**	**GitHub**
Alex Kim	alextabinkim@utexas.edu	https://github.com/kimchibean
Manun Chopra	chopra.manun@gmail.com	https://github.com/phenomanun
Elvin Hung	elvinhung@gmail.com	https://github.com/elvinhung
Rishab Chander	rishab.chander@gmail.com	https://github.com/rchand20
Nithanth Ram	nithanth.ram@gmail.com	https://github.com/nithanth

Motivation and Users:

The motivation behind creating this application is to allow users to view and explore different aspects of a location they choose from a perspective of tourism and travel. Our application's interface currently includes restaurants and hotels of specific cities that the user can query and search for. A user will be able to search a city of their choosing and receive the location of that city (latitude and longitude coordinates), the city's population, the city's current weather forecast, and curated lists of both restaurants and hotels in that city. Each model (locations, hotels, and restaurants) will have a picture associated with it for visual aesthetic and the user's convenience. Users of this website can be broadened out and expanded to simply the general public looking to gain more information about specific travel aspects of a city. Certain subsets of this group can be created as well to form specific user groups (i.e. traveling individuals, foodies, students studying abroad, etc). Anyone with either a specific interest in food or travel or just a generic tourist can be a potential user of our website. It serves to simply provide some real-time information about international cities for those seeking it.

Requirements

User Stories

We added 10 more user stories and along with our 10 user stories from Phase I and II. Additionally, we used our Project Board on GitHub to break down components of each story into workable tasks of about 30 min - 1 hr (planned) duration each. Our scrum board contains user stories and tasks that we defined over this first phase. The top 5 issues on our project board are the user stories we completed in Phase 2. We have reproduced the stories below completed in both Phases 1 and 2, as well as those we aim to complete in Phase 3.
Project Board: https://github.com/elvinhung/SoftwareLabApp/projects/1

10 New Stories requirement for Phase 3 highlighted

Phase 3 Completed:

1. As an impatient customer, I want to be able to see many instances in a grid layout so that I can see more information per page. Issue #98
    a. Estimated: 1 hour
    b. Actual: 1 hour
2. As a traveler concerned about the weather, I want to be able to see forecast information to be able to plan my trip safely. Issue #70
    a. Estimated: 2 hours
    b. Actual: 2 hours
3. As a traveler, I want to be able to search by location so that I can see what my options are in each city. Issue #11
    a. Estimated: 2 hours
    b. Actual: 3 hours
4. As someone who doesn't like crowded places, I want to see the population of a city so that I can determine if it's too crowded for me. Issue #73
    a. Estimated: 1 hour
    b. Actual: 1 hour
5. As a foodie, I want to see multiple pictures of the food at a specific restaurant so that I can see if the food looks good. Issue #74
    a. Estimated: 1 hour
    b. Actual: 2 hours
6. As a hotel guest, I want to see what amenities are available at a hotel so that I can enjoy my stay. Issue #75
    a. Estimated: 2 hours
    b. Actual: 2 hours
7. As someone who loves to swim, I want to be able to filter hotels so that I can find one with a pool. Issue #76

    a. Estimated: 3 hours
    b. Actual: 4 hours
8. As someone who enjoys sightseeing, I want to be able to know the top 5 most popular attractions in a given city so that I am able to better enjoy my time there. Issue #38
    a. Estimated: 2 hours
    b. Actual: 4 hours
9. As a high-end customer, I want to sort hotels by highest rating so that I know that I will have a good stay. Issue #102
    a. Estimated: 1 hours
    b. Actual: 2 hours
10. As someone who doesn't know where to start, I want to be able to search through all models at once so that I get lots of options relating to my interests. Issue #101
    a. Estimated: 2 hours
    b. Actual: 2 hours

Phase 2 Completed:

11. As someone who plans ahead, I want to be able to see the exact distance nearby restaurants are from a hotel. Issue #35
    a. Estimated: 2 hour
    b. Actual: 2 hour
12. As a visual person, I want to see a map of each instance's location so that I can have a better sense of where they are in a city. Issue #39
    a. Estimated: 3 hours
    b. Actual: 3.5 hours
13. As someone who is easily overwhelmed, I want to only see 10 instances at a time so that I can view the information at a calm pace. Issue #36
    a. Estimated: 2 hours
    b. Actual: 1 hours
14. As someone on a budget, I want to be able to see the price range of restaurants so that I can determine if I can afford my trip. Issue #37
    a. Estimated: 2.5 hours
    b. Actual: 2 hours
15. As an event coordinator, I want to be able to see contact information for businesses (number, website) so that I can set reservations easily Issue #10
    a. Estimated: 2 hours
    b. Actual: 2 hours

Phase 1 Completed:

16. As a user, I want to be able to see a picture for each model so that I can see the quality of each. Issue #14
    a. Estimated: 1 hour
    b. Actual: 1 hour

17. As a traveler wanting to save costs, I want to be able to see restaurants near my hotel. Issue #15
    a. Estimated: 1 hour
    b. Actual: 1.5 hours
18. As someone who plans ahead, I want to be able to see which restaurants and hotels are there so that I can decide what to do there. Issue #16
    a. Estimated: 1 hour
    b. Actual: 2.5 hours
19. As a traveler, I want to be able to see the addresses for hotels/restaurants so that I can send them to my friends. Issue #17
    a. Estimated: 1 hour
    b. Estimated: 30 minutes
20. As a curious customer, I want to be able to see who developed this application so that I can confirm its reliability. Issue #18
    a. Estimated: 2 hours
    b. Actual: 3 hours

Planned/In Progress:

21. As a person tired of the same food, I want to be able to use my location to search so that I can find new places around me. Issue #13
22. As a foodie, I want to be able to filter search by tags so that I can look for restaurants serving the type of food I want next. Issue #12
23. As a big sports fan, I want to be able to see what teams are in a given city so that I can go out to some games. Issue #77
24. As a vegetarian, I want to see a menu for each restaurant so that I can decide if I can eat there. Issue #100
25. As a traveler, I want to be able to see hotel room rates so that I can make informed decisions. Issue #99

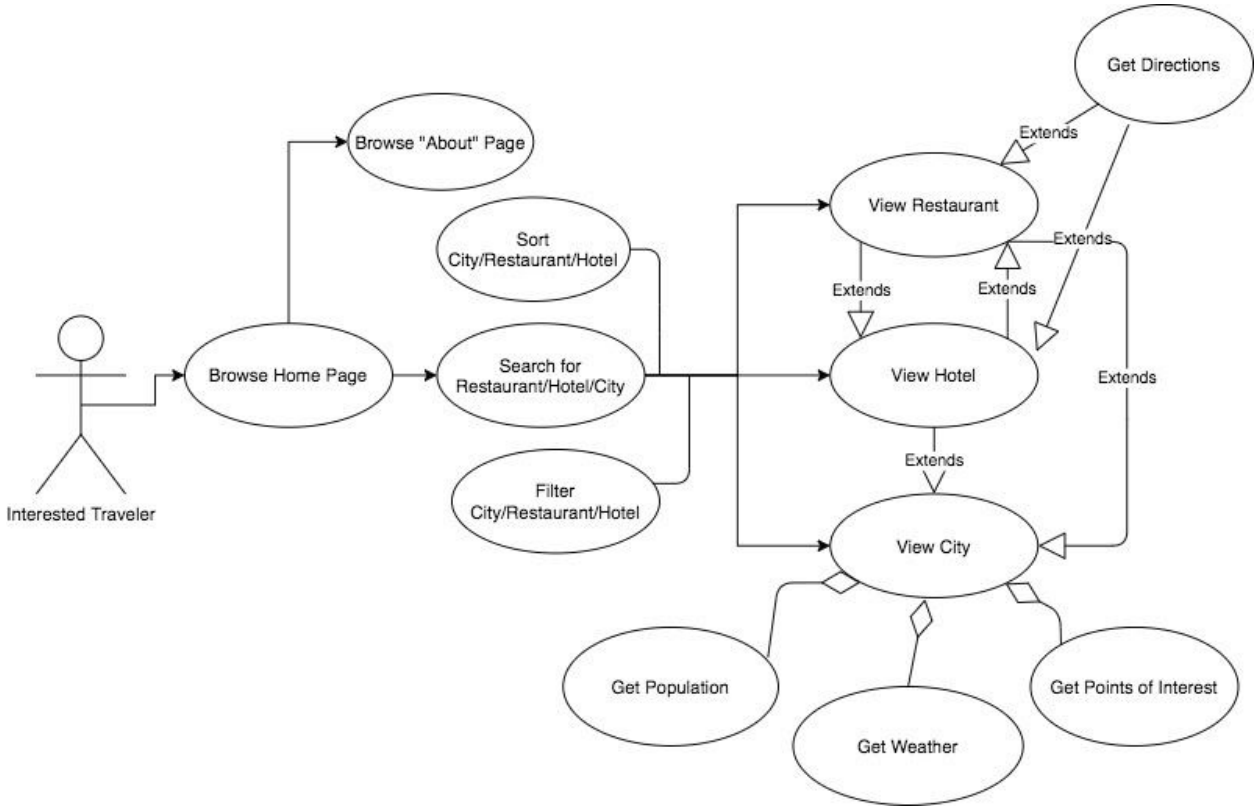

Figure 1. Use Case Diagram

Design

As shown above in Figure 1's use case diagram, the design for our application is thoroughly and logically mapped out. The design for our frontend in this phase consists of dynamic web pages. Relative to this phase, the Home page's functionality consists of navigation to other pages with a search bar for users to query all of our instances of locations. The Home page has a clean aesthetic that lends itself to the design of the rest of the website, with quality images and styling to make it look neat. The Home page, and each subsequent page, also has a navbar which allows for direct access to all of our different pages (each model's page and the about page). Simply clicking the "Nomad" button on the navbar will allow the user to return to the Home page from any part of the website. The website itself utilizes pagination to display our many instances of each model. Since each city has three hotels and three restaurants associated with it, and there are 150 cities in our database, displaying all our instances on a single page would be an impractical design. Instead, our website uses pagination to create multiple pages of our models with 12 instances showing on each page. From the perspective of our backend, we are no longer using static JSON responses as we did in Phase 1. We were able to use our respective APIs in tandem with PyMongo to store all our model instances in a MongoDB database. Our web application now queries the database to retrieve the necessary information or instances it needs to fetch for display. The MongoDB database has respective collections that map to the models our website displays: locations, hotels, and restaurants (distances is another collection

that is used to display how far nearby hotels or restaurants are but isn't a core model of our application itself). These are discussed in more detail in the *Models* section of the report. All requisite information scraped from the JSON responses of our API calls are now respectively bundled together as documents in each model collection in the database to represent our instances.

Our "About" page sends two HTTP requests to the GitHub API in order to retrieve the number of commits each team member has, the number of issues each team member is assigned to, the number of unit tests, and the total repository statistics for our team. The sequence diagram for this operation is shown in Figure 2 below. The "About" page consists of two objects: "contributors" and "stats". These objects are used to represent information for each team member and the total statistics for our team. The class diagram is shown in Figure 3 below.

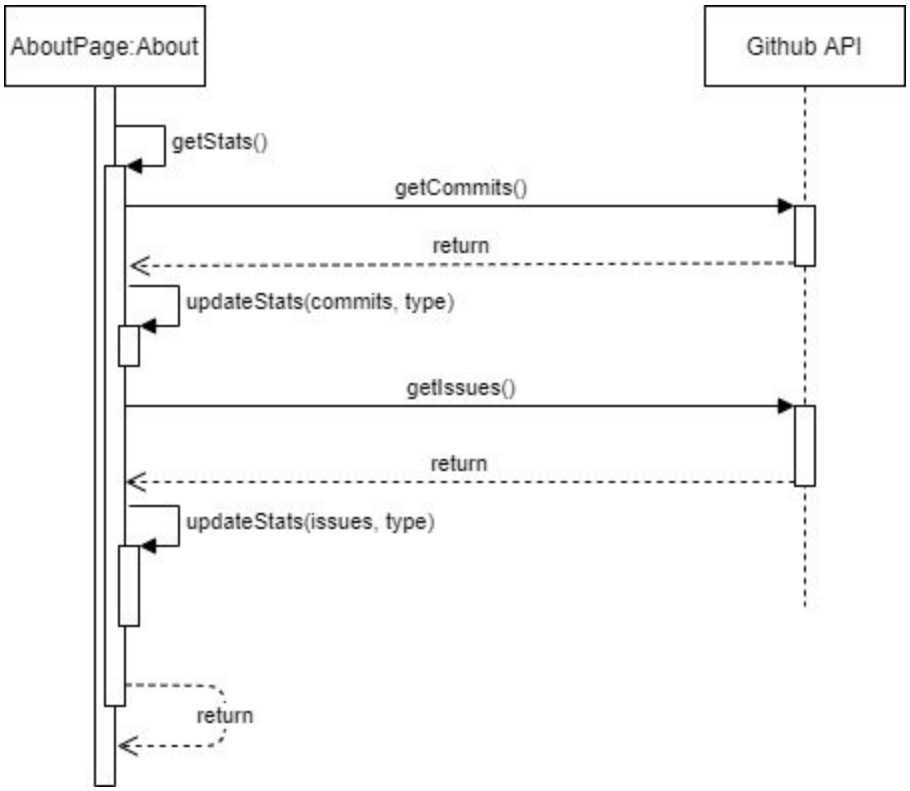

Figure 2. About Page Sequence Diagram

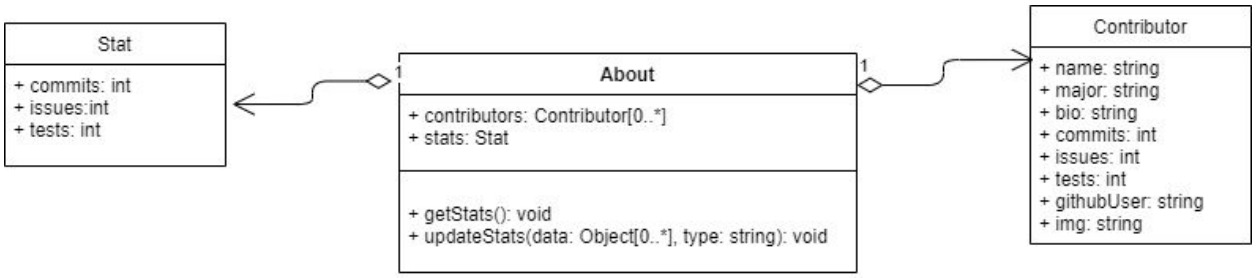

Figure 3. About Page Class Diagram

The Locations, Restaurants, and Hotels page all function in similar ways. Using the Locations page as an example, it uses the query parameters in the URL and makes an HTTP request to our Flask API hosted in AWS to retrieve a list of all locations that match that query. The API allows users of our site to filter models by certain attributes and uses those parameters to query MongoDB.The information returning from the API is used to render Location instances. The sequence diagram for Location pages is shown below in Figure 4. From a Location instance, a user can navigate to a Location Detail page. This displays more information about the instance. The Detail page makes an HTTP request to our Flask API using the “id” in the url. For example in the url example.com/locations/HOU, HOU would be the location id for the city Houston. The API executes a find_one() call, attempting to find a document with an id field matching the one requested, and propagates the result back to the frontend. The sequence diagram for the Location Detail page is shown in Figure 5.

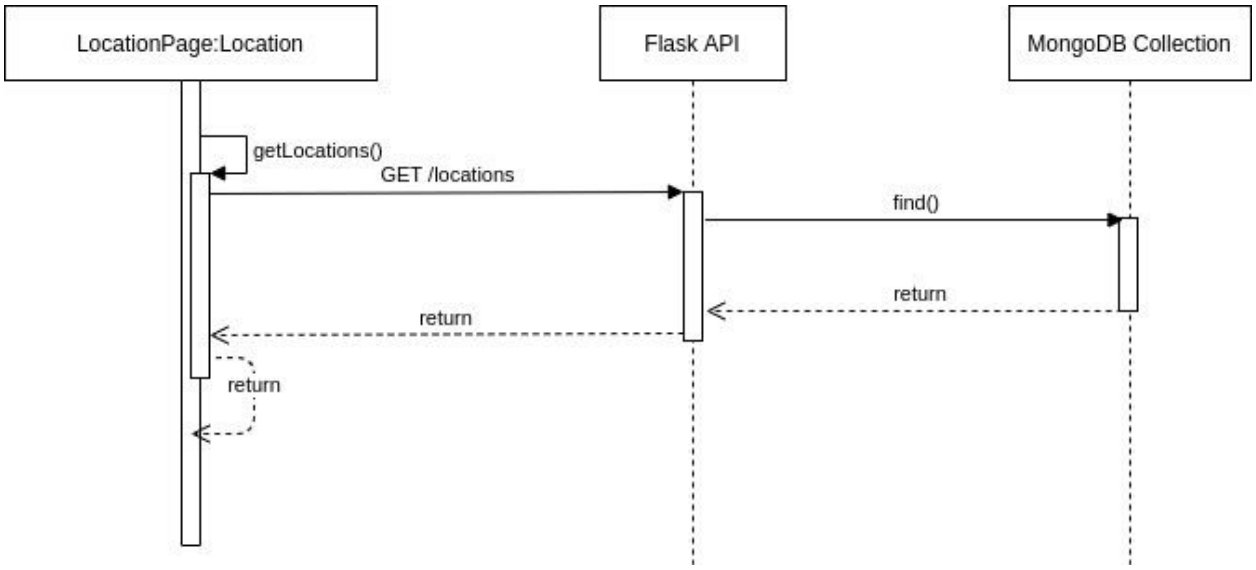

Figure 4. Location Page Sequence Diagram

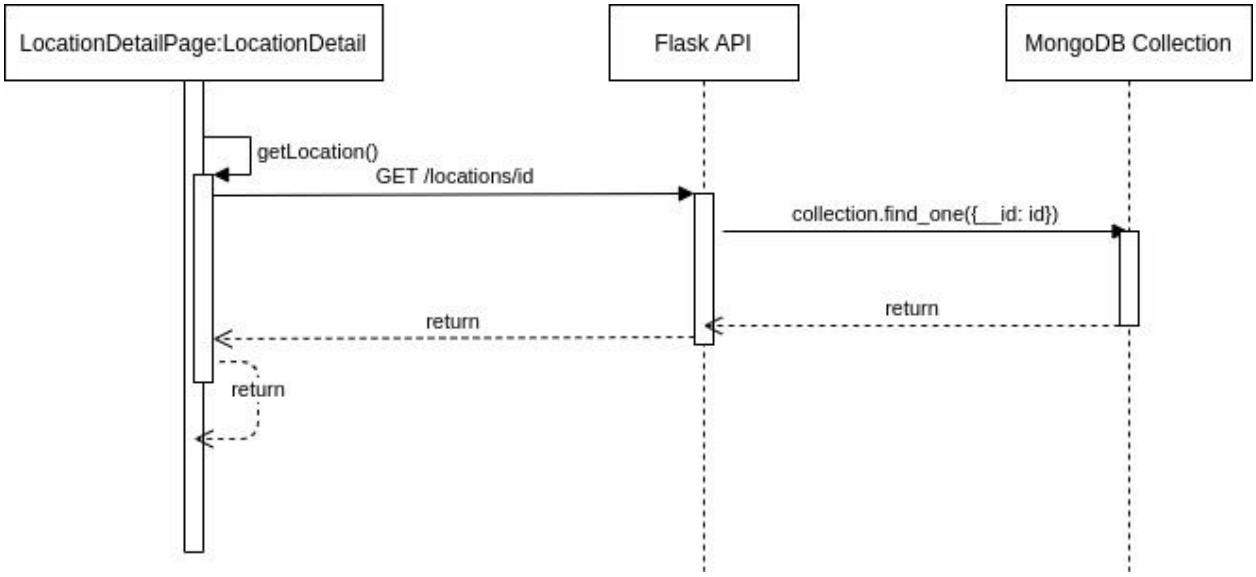

Figure 5. Location Detail Page Sequence Diagram

While the functionality across Locations, Restaurants, and Hotels pages are relatively the same, the objects within the classes are different. This can be shown in the class diagram below.

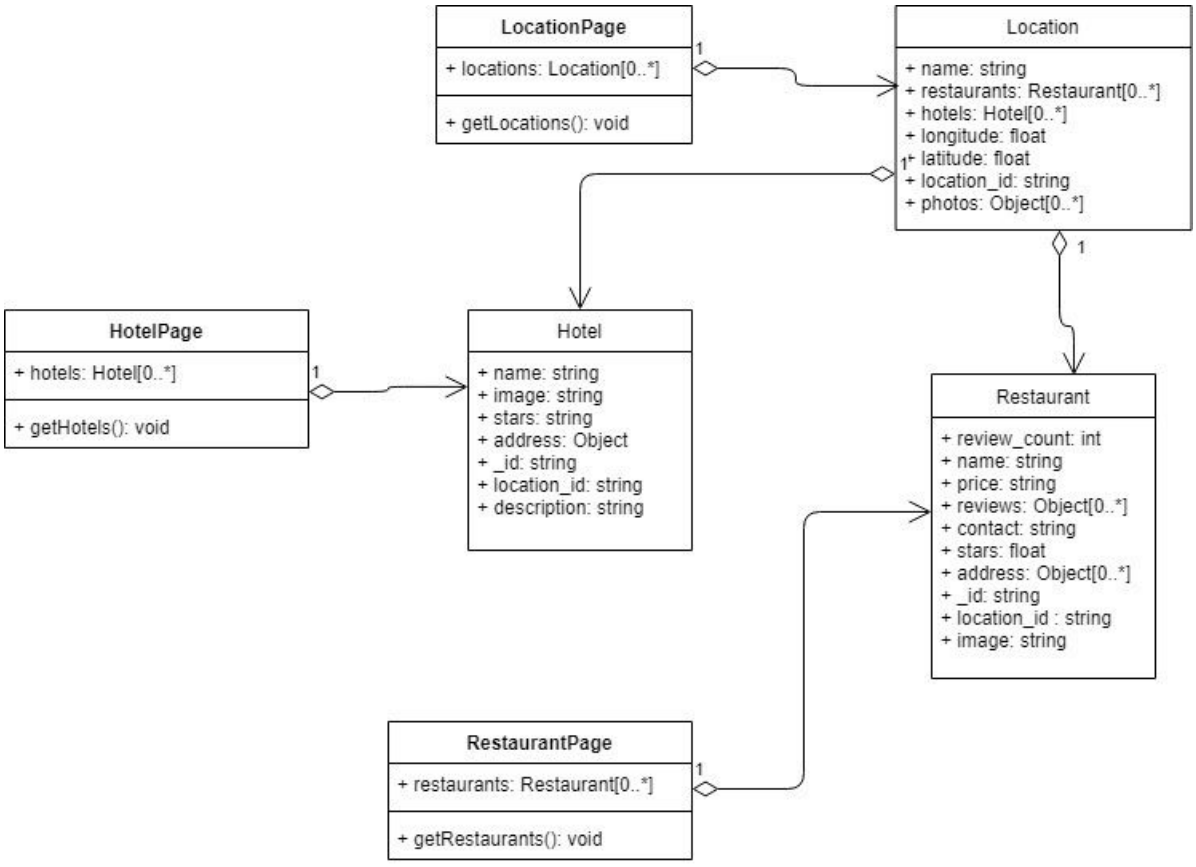

Figure 6. Page Class Diagrams

When an HTTP request is made to the Flask API from either a Restaurant or Hotel Detail Page, the API makes an additional MongoDB Query to request the closest restaurant/hotels to the requested instance. Using a Restaurant Detail Page as an example, a sequence diagram can be observed below.

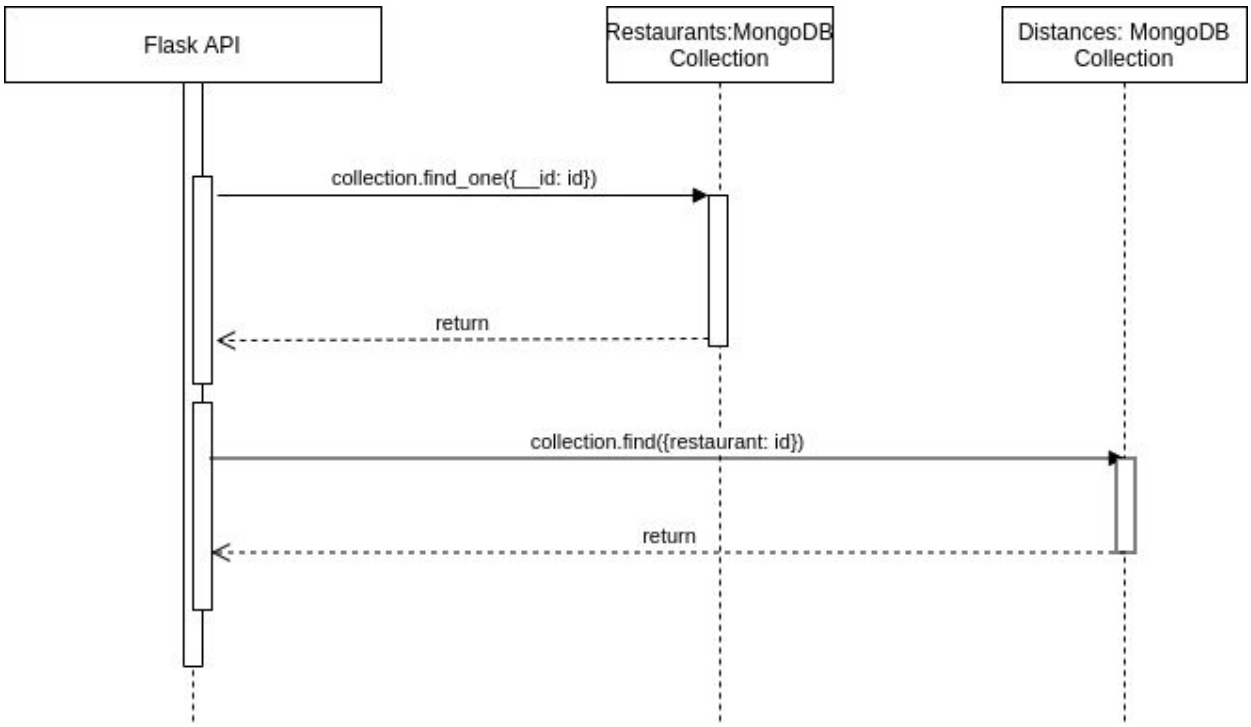

Figure 7. Restaurant/Hotel Detail Page Sequence Diagram

A key functionality implemented in this phase is searching, sorting, and filtering. We implemented this by creating two main things: a Search Page and a Search Component. The Search Page allows us to query all of our instances in our database based on the input from the user. The Search Component consists of the text input and filter button inputs. If a user searches from our Home Page or uses the “All” filter on the Search Component, it redirects them to the Search Page with the query in the URL. Using those query parameters, a GET request is made to the Flask API at the “search” endpoint. This returns all instances that match the query, and the frontend displays that information accordingly. The sequence diagram for the Search Page is shown in Figure 8 below. The class diagram for the Search Page is shown in Figure 9.

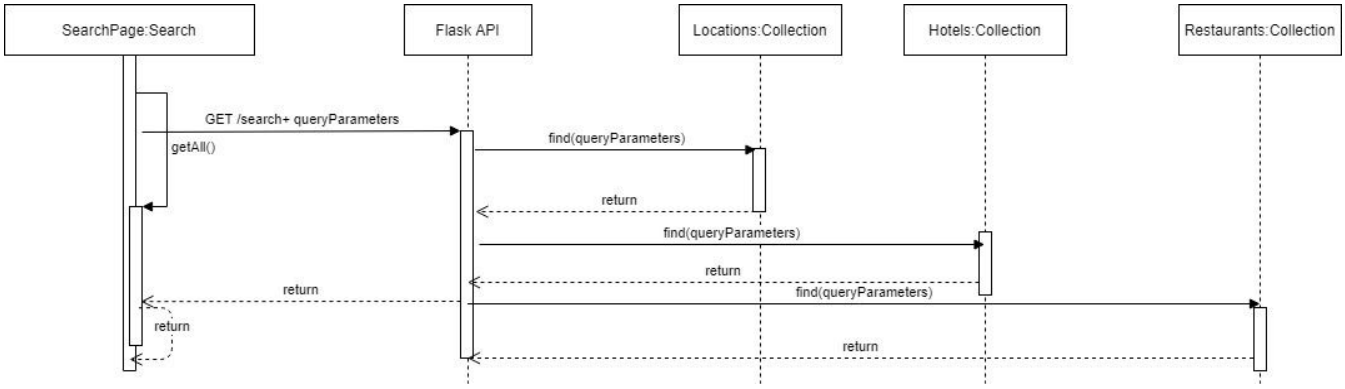

Figure 8. Search Page Sequence Diagram

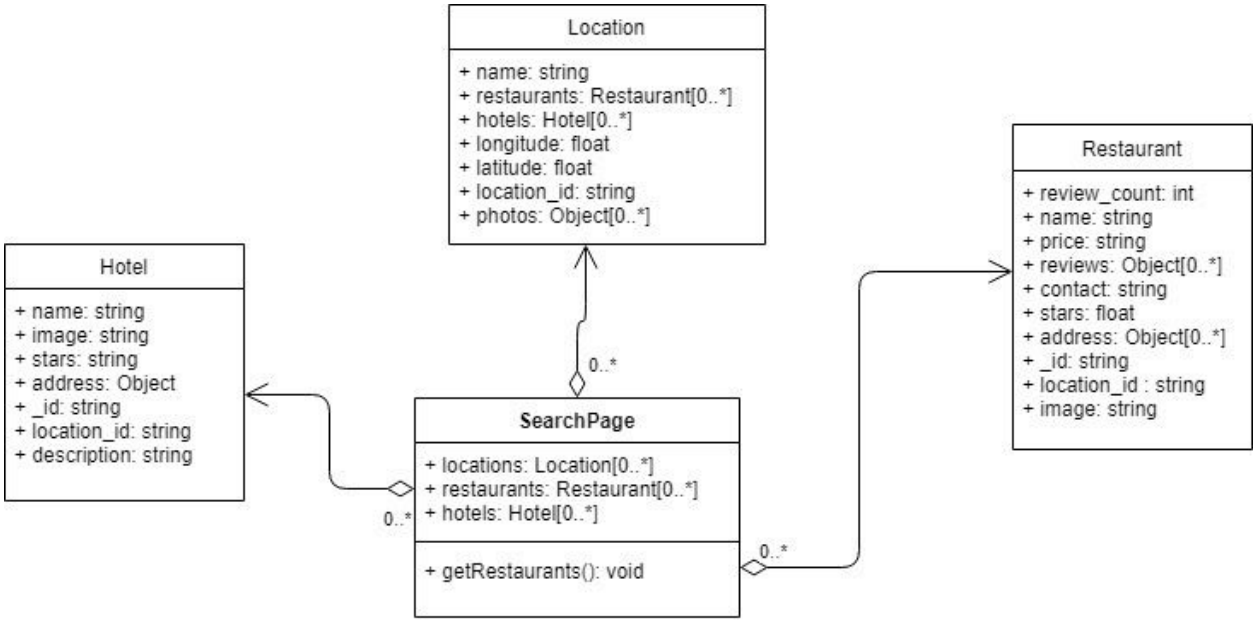

Figure 9. Search Page Class Diagram

The Search Component allows users to select what kind of instance they want to search for and filter/sort accordingly. The user can sort and filter by clicking on a variety of radio inputs. Once the search button or apply button is pressed, the browser will redirect to the correct page with the query parameters being set by the filters selected by the user.

Figure 10 below demonstrates the relationship between the different models as represented in the database, and their respective relationships with one another. Each DB collection's schema is represented by the attributes in the class.

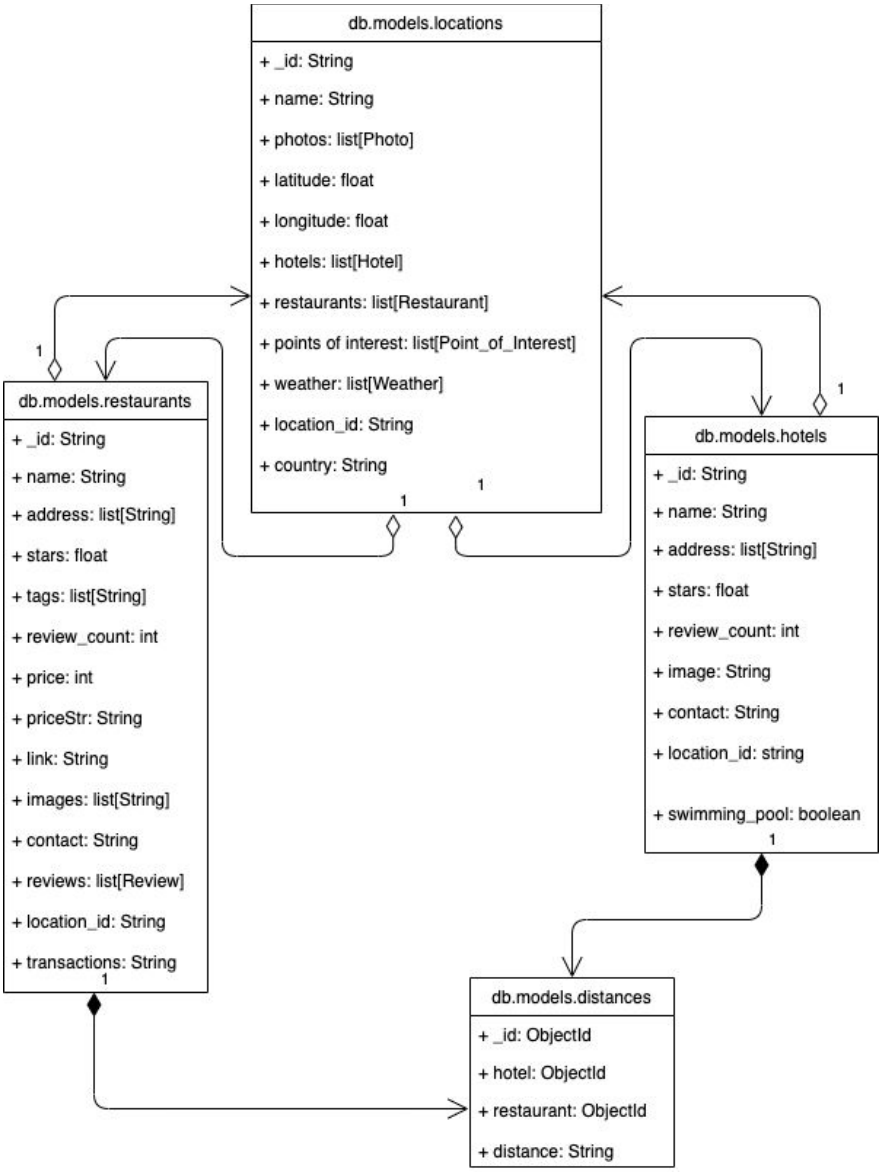

Figure 10. Database Relationship Class Diagram

Testing

Throughout our 3 Phases, we've done testing for both the frontend, GUI, and backend of our project.

For the frontend we used Jest (JavaScript testing framework) and Enzyme (JavaScript testing utility for React). We tested the most important components of our application such as the Locations/Restaurants/Hotels pages, their respective Detail pages, and the Search component. Enzyme allows us to ensure that our components are rendering HTML elements and data as expected. We used Jest to mock our API, so our components don't make a request to our actual API in order to ensure our data we are using is sound.

We tested the GUI using Selenium and JUnit testing. Selenium allows us to emulate and automate user interaction with our website. JUnit gives us the ability to control our Selenium calls and check if they're producing the expected effects. JUnit also gives us the ability to group test cases together in suites, which was very useful for organization. We organized our tests in suites based on these 4 main categories of use cases: navigation, searching, sorting, and filtering.

Backend testing was done using Postman. Test scripts were written for each request to our API to ensure both the success of the HTTP GET and the validity of the data returned.

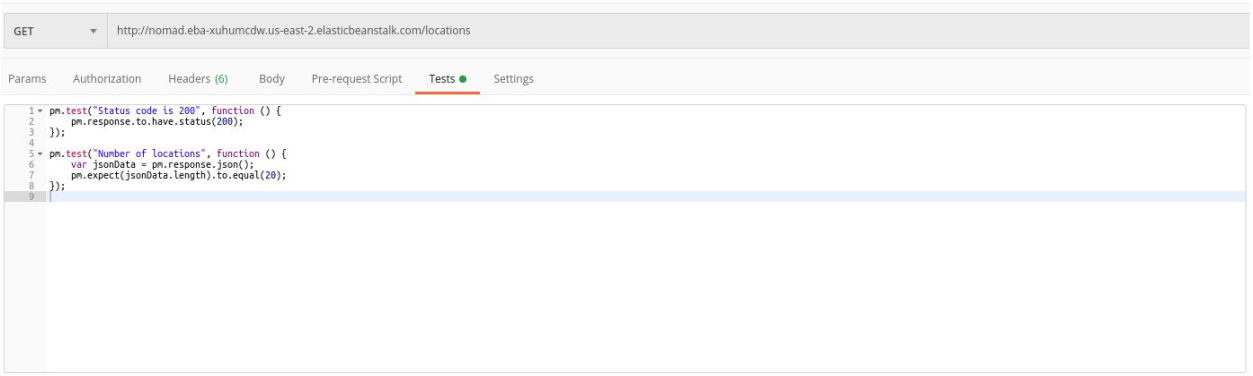

Figure 11. Sample API Call for All Locations

Figure 11 above illustrates a request to all instances of the Location model. The test scripts verify that the status code is 200, meaning the request was successful, and tests the assertion that the correct number of instances was returned. A similar testing pattern was used to test the requests that get all instances of the Restaurant and Hotel models.

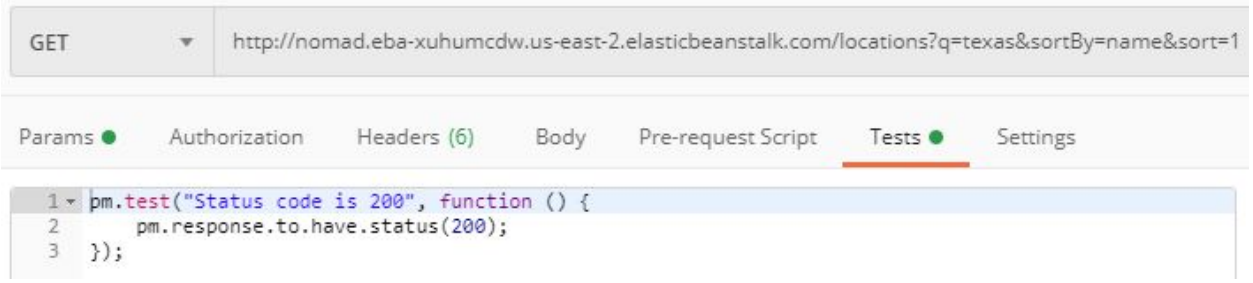

Figure 12. Sample API Call for Locations with Query Parameters

Figure 12 shows a sample request to get all instances of the Location model that match certain query parameters. The test script simply verifies that the request is successful, as there is no way for us to verify the filtered results. This pattern was also used to test the /restaurants and /hotels.

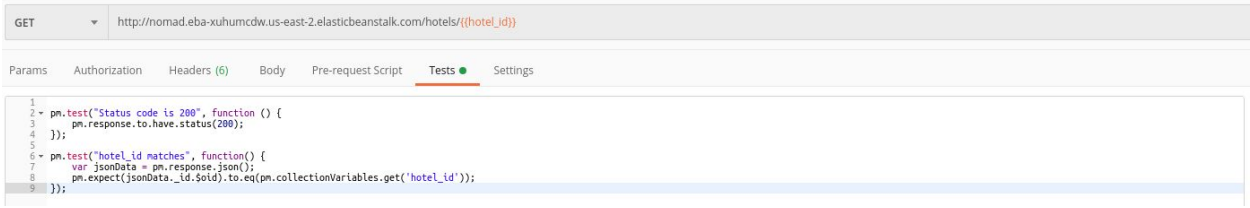

Figure 13. Sample API Call for Single Instance

For API requests that query for a single model instance by __id, the test scripts again verify a 200 status code, and then checks the assertion that the returned document __id matches the id contained in the request, as seen in Figure 13 above. This testing pattern was used for all three models in our design. The full JSON of our Postman Collection can be view at: https://github.com/elvinhung/SoftwareLabApp/blob/master/backend/flask/Phase3-API-tests.postman_collection.json

Models

Our website consists of three data models: Locations, Restaurants, and Hotels. Locations refer to specific cities in the world and contain the Latitude and Longitude of the city, pictures of that specific city, the population, the weather forecast, and 5 points of interests located in the city. The Google Geocoder API was used to obtain the geo coordinates and Google Place API was used to get pictures and points of interest of each location. Along with coordinates and a descriptive picture, each location contains current weather information (temperature, windspeed, humidity, and a three day forecast), population, and five points of interest in that location (with an address, rating, and tags). The points of interest were also obtained using the Google Place API. Each location also references instances of Restaurant and Hotel models that are located in that particular city. The Restaurant model contains the Name, Address, Phone Number, Star Rating, and 3 pictures for each restaurant. They each also have customer reviews, contact information, and pricing information. We utilized the Yelp Fusion API to add all of this information. Lastly, the Hotel model contains the Name, Address, Star Rating, a Description and picture of the hotel, and whether or not the hotel has a swimming pool. The Amadeus Hotel Search API was used to scrape this information.

Both the Restaurant and Hotel models are linked to each other through location. Furthermore, to flush out the connection between the two models we added a metric “nearby” hotels and restaurants for each respective model in order to link them by their actual distance from each other. This helps users who may be planning to stay in certain areas of town plan where they could eat their meals or vice versa. Or for travelers interested in traveling to cities, they can see top hotel accommodations as well as famous restaurants from a variety of price ranges. Circling back to the weather and population attributes of the location model, even though these weren’t fashioned as models, they still required API calls and storage in our MongoDB database. The weather information was obtained using OpenWeatherMap’s OneCall API which provides current weather information (listed at the top of this section) as well as future forecast

information. The population information was a bit trickier to obtain. There was no robust API that supported fetching populations from international cities. So to navigate around this problem, datasets from Opensoftdata were pulled and combined using the pandas library in a Jupyter notebook. From here, all the requisite city populations were compiled into a dataframe and uploaded to the MongoDB collections as a population collection. The location model instances were then able to access the information in these documents to display population numbers on the website.

Tools, Software, Frameworks

Frontend: React.js was used to create the front-end with Bootstrap CSS layered on top to create an appealing UI.
Backend: Python was used to scrape APIs and populate our MongoDB Database. Flask was used to create a RESTful API for our frontend to use to dynamically load data from the DB. It was also used to implement the searching, sorting, and filtering functions on the website. Compass from MongoDB was used as a GUI to view the DB and the data in it.
Hosting: AWS S3 is being used to host the static website in a S3 bucket. Now that our website is dynamic, we deployed our MongoDB cluster on AWS and pull our data from the DB through an API hosted on AWS Elastic Beanstalk.
Testing: Postman was used to test API endpoints to ensure that the correct data was being returned from our requests. We additionally used Postman to test our Flask API. Jest and Enzyme were used to test our React frontend components. For the GUI, we used Selenium and JUnit to emulate user interactions.

Reflection

In Phase 3, we worked more efficiently together even with a time crunch as many of us had midterms and other projects to work on. One thing we did better this phase was work together in smaller developer teams as opposed to more individual work. The shift to online instruction got us used to working together remotely. The frontend team synced up on their work to enhance the visuals and display data via the Flask API. The backend team worked together to add functionality to the Flask API and add more interesting data/attributes across the models. These working in pairs helped us resolve issues faster and communicate effectively as a team. We also really utilized tools like Slack and Zoom to make sure our communication was improving rather than hampering our productivity. Something to improve on our work this phase would be to clarify uncertain requirements from the customers/teaching team ahead of time in order to ensure a uniform development timeline. Aside from that, our team did well this phase and is looking forward to presenting the project soon.