TECHNICAL REPORT: PHASE 4

foodforthoughtt.me

Contributors:
Murray Lee, Alexander Ng, William Suh, Steven Sun, Kelvin Yu

Motivation

The goal of this website is to provide a place for users to build a more healthy diet by drawing correlations between certain foods and diseases and also providing some restaurants to find these foods. By allowing users to see and understand the risks behind consuming these certain foods we can notify more people about food-related diseases.

User stories

These are the new user stories we received from our customers.

1. Reformat each page to put information in the center of the page instead of pushed against the left side of the screen.
2. Get many instances of each model, and create pages that break the number of instances shown on the page up. Provide a way to navigate these pages.
3. On the model pages, provide 5 pieces of data as a preview for each instance. Organize each preview into tables or cards.
4. Create 3 model pages with links to each of the instances of that model
5. Create a home page that links to each of the 3 model pages

These are not the user stories we were supposed to receive so our team has developed a couple for the meantime

1. Customers will probably want to filter foods by their nutrition stats such as carbs.
2. Customers will probably want to be able to filter foods by disease relevance so they know which foods would help the condition and which ones to avoid.
3. Customers will probably want to filter restaurants by distance, price, ratings.
4. Customers will probably want an easy way to find restaurants that sell the foods that will benefit their selected disease.
5. Customer will probably want to apply multiple filters on 2 models to produce a list of relevant targets e.g. picking a disease and selecting foods that would benefit it and then showing restaurants that sell it by proximity, price, rating, etc.

These user stories will probably take us the semester to build up a smart filtering system that would allow clients to find the most relevant information to them. This is an important function of our website though because a lot of information already exists on these topics and to set us apart from the competition we need to deliver this data in a relevant way.

RESTful API/Database

We implemented our RESTful API using flask and SQLAlchemy in python, around our three models, foods, diseases, and restaurants. The API contains GET, POST, and Delete requests for our three models. The POST request puts information into the database, GET requests gives information from the database, and DELETE as the name implies deletes an instance from the database. The delete request will mostly be used for debugging and error correcting purposes.

For our database we used MySQL.

We implemented search and sort api requests. For sort adding a ?order_by=attribute_name will return a sorted list in ascending order based on the attribute. For search we added a new endpoint to our api. Making a post request to the /search endpoint with a list of string will return a list of list of instances that contains the keyword in the string.

Models

Our three models consist of Foods, Diseases, and Restaurants. Each of these models has various attributes that will allow us to create many similarly-structured instances.

Instances of models were added to our database using Postman through POST requests. Attributes were decided based on what we felt were most relevant to the connections between our different models.

Foods

Our foods model will contain nutrition label information because this information is standardized and abundant. This convenience will allow us to catalog many of them for comparison. The Food model consists of the following attributes:

1. Name – The name of the food.
2. Serving Size – The recommended serving size in grams.
3. Calories – Number of calories per serving.
4. Total Fat – Total fat in grams. It includes saturated and trans fats.
5. Carbohydrates – Total carbohydrates per serving in grams.
6. Protein – Protein per serving in grams.
7. Sodium – Sodium per serving in milligrams.
8. Category – Cuisines it belongs to.
9. Image – Picture to visualize instance.

Diseases

Our disease model currently contains multiple attributes but diseases don't follow such a convenient standard like the food's nutrition label example. This can make databasing the diseases harder but another solution could be to rely less on database storage and more on front-end data entry. Currently, without implementation this model represents some of the information we wish to convey to each client but the delivery and structure are subject to change. The Disease model consists of the following attributes:

1. Name – The name of the disease.
2. Specialty– The medical specialty this disease is listed under e.g. infectious diseases
3. Symptoms – The symptoms associated with this disease.
4. Causes – How patients typically contract the disease.
5. Category – What cuisines best describe the foods recommended.
6. Image – Visualization if disease
7. Frequency – How many people are afflicted in the U.S. each year.
8. Deaths – How many people die from the disease each year.

Restaurants

Our restaurant model consists of typical information our clients would expect. These attributes will allow us to provide our clients with a restaurant. The Restaurant model consists of the following attributes:

1. Name – Name of the restaurant.
2. Location – Address of the restaurant.
3. Rating – Overall customer review rating on the scale of 1-5 stars.
4. Cuisine – The type of food sold by this restaurant.
5. Cost – The relative cost on a scale of $ – $$$ as seen on sites such as Yelp.
6. Image – Visualization of restaurant food

Tools

1. Gitlab for version control and keeping track of our progress and todos through issues. It is also used as a source to pull statistics to the about page on the website we are hosting.

2. AWS was the host we decided to use for our website. We used this amazon service to remotely handle many of the backend requirements that our website

has. AWS also provides many applications and tools to assist in this endeavor, namely S3, Cloudfront, Certificate Manager and IAM.

3. Slack is our main tool for team communication. This application helps us keep our conversations organized and declutter our GitLab. We are considering integrating Slack with our GitLab so it keeps us more in check with issues and statuses of this group project.

4. Postman was what we designed our API on. This tool allows us to design an API that embodied the functionality our website would provide. This is extremely useful because it provides us with a clear vision of what we need to create without fully committing to the overheads of development.

5. React is our main front end tool using bootstrap as a framework. Using React we will be able to aesthetically organize our information and allow clients to search and filter through various pages.

API

1. Zomato API has a database of over 1.5 million restaurants in over 10000 cities. It also contains detailed information about their respective menus. This API best represents our model because using the detailed information on the menu, we can track which foods we can recommend, and how suitable a restaurant's menu is for someone suffering from certain conditions.

2. Healthfinder.gov API is hosted by the U.S. Department of Health and Human Services. This source is probably the least controversial and provides a wealth of information related to diseases. Not only does it provide information on health conditions and diseases, but it also has information for nutrition and advice on healthy living. This provides us with options to extract the data we need for our models, but also to tie them together.

3. Edamam API contains the nutritional analysis of foods, meal recommendations, and recipe search. This will assist us in pulling stats of foods and cross-referencing them with restaurants and health conditions, which is another useful way to create ties between our 3 models.

Hosting

The website was hosted on AWS. This not only saved us the trouble of having to set up our own server, but it also gave us scalability should the amount of data we store needs to increase or if web traffic increases.
We use one central IAM user with permissions to all of our AWS services. All services are hosted in the ohio region.
We use an S3 bucket to host our website. It holds all the source files and other assets used in our frontend. It has public read access. This allows us to handle the distribution of the front end with CloudFront. CloudFront was used to verify the domain and configure https. Certificate Manager is used to create a certificate for our domain, foodforthoughtt.me, to use in CloudFront. Our domain is hosted through Namecheap. We added CNAMES, for our CloudFront and Elastic Beanstalk.
Elastic Beanstalk hosts our backend python flask server. It has its own separate Certificate for api.foodforthoughtt.me subdomain and added CNAME in Namecheap as well. AWS RDS is used to host our mySQL database. Nothing special with the hosting was used here.

Pagination
There are 9 instances per page and regardless of whichever model you go to, it starts off on page 1. As a user clicks onto another page, the instances associated with that page are grabbed and loaded onto the page. The pages are shown at the bottom and the instances are shown on the page are updated each time a user clicks onto a different page.
The tools used to implement pagination were React and CSS. We used React to build the UI component of handling going from one page to another and loading the instances corresponding to each page. Once we grabbed the instances for each page, we used CSS to style/format the data in an aesthetic way.

Testing
Our GUI tests are written using the python selenium package and make use of the Chrome web browser (via the chromedriver file included in the testing directory). The main target of the tests is user experience. We wanted to make sure the pages loading properly when clicked on, and we plan to expand it to testing more various other parts of the website.

The server testing makes use of the built-in Python unittest framework, including making heavy use of its object mocking capabilities. We test the set up, making SQL calls, and making attribute calls.

The Javascript testing was implemented using Mocha, a testing framework designed for React. Our mocha tests test the React code we have written using describe() and it(). The tests we wrote tested some functions in food the retrieved an attribute such as getCalorie or getProtein.

On testing our api we used postman. We made post, get, and delete requests and got successful results. The delete however can only be run once without changing the id number. We exported a json file that goes through our tests.

DB

We set up the database with RDS on Amazon Web Services. We decided on PostgreSQL because it is supposed to be easier to set up. After the database was set up on AWS, we used the tool pgAdmin4 to easily interact with the database. This was used to create the different tables to hold data and set up their schema. We also frequently used it to add (and later remove) test rows from the database, to conduct test queries before adding them to the server code, and for general maintenance of the database.
After having written the queries in pgAdmin4 that selected the information we needed to return for each of our server endpoints, it was simple enough to execute these with SQLAlchemy.

Filtering

Filtering was implemented completely on the front-end. The element array used to render cards dynamically was simply adjusted using an array.filter method. This way, a new set of cards was displayed upon narrowing down filter settings. These filter settings were saved as state variables and used as parameters to use within the .filter function.

Specifically, each card is displayed using props, where a dictionary of attributes is passed through. These cards are displayed with a grid component which stores a list of dictionaries, with each dictionary corresponding to a card. These attributes (such as name, 5 filterable attributes, image) are displayed on the card inside the render function. Upon selection of an attribute, if it's a numerical attribute (such as calories), two input boxes for numbers would pop up. Using these, a minimum and maximum value for the current attribute to be filtered would be stored. Otherwise, a drop-down menu of every single possible attribute stored within the card grid's instance array would appear, and multiple could be selected. These drop-downs were also unique to each attribute, so multi-attribute filtering with multiple selections per attribute is possible. Filtering occurs based off the current selected attribute on the top drop-down. Based on attribute, the grid's global list was modified using a list.filter function. This call to .filter would return true and consequently add an element to the new list only if the filter input or selections matched with the card instance's attribute value.

Searching

Searching was implemented on the front-end. The same practice as Filtering was implemented, except the .filter method simply went through all relevant keys and returned cards for instances which contained key values containing whatever was passed through the search bar.

To elaborate on how the .filter for search was implemented, the function would return true if, after looping through every key (aka attribute) in each instance's dictionary, the search term was found in the value of any attribute's key. Upon handling multiple terms, a logical or was conducted. As in, if any of the search terms split by whitespace was found in its entirety inside a value, that card instance was added to the list of search results. We created a custom search card for search results that was simply a card with card-text for each key/value pair. We only displayed the instance name and its five attributes. Upon hitting the "search" button, the native element array for all instances was cleared, and thus the card grid displaying off that list disappeared. Immediately after, another previously empty element array for search results was filled with the results of the .filter function, and this time the new list was displayed using the unique search cards. To go back to the old grid, the user must hit the reset button or refresh the page. Unfortunately, we couldn't figure out how to search upon hitting enter in the search form. Highlighting was implemented by slicing the value string from the dictionary inside the unique search card component after passing in the search parameters through props. Then, using string manipulation and formatting, we re-displayed the string as <Text> items, and bolded every other item using styles. This worked because when slicing out parameters to be bolded, there will only be "cuts" around what is to be bolded, so after checking if the first "slice" was to be bolded, the bolding pattern would always result in every other "slice" being bold.

Sorting

Sorting was implemented on the front-end through a custom sort on the values within each attribute. This was done completely on frontend because it was the quickest and cleanest way. When no attribute is chosen, elements are sorted by name ascending. The builtin Javascript sort is used on a custom comparison based on dictionary values.

Again, we wrote a custom compare method that simply returned -1 if key1<key2, 1 if key1>key2, 0 if equal. There was no modification to the comparison operator required since all we sorted by were numbers and strings. A .map function was applied to the native instance element array based on this simple comparison function, and the result was re-displayed through render.

Visualizations

Visualizations was implemented on the front-end using d3 which is a simple way to display data in a clear and presentable way. This could only be completed on the frontend so all of the members on the front-end team worked on it. We were planning on creating a bar chart to effectively display the number of calories given in a particular food, a pie chart to portray the number of restaurants given a specific rating out of five, and a timeline for the average age affected for a given disease. Due to complications and poor understanding of d3 we were only able to create bar charts for each model.

Refactoring

As each of our model pages is a separate js file, and they're essentially clones of each other with some different hard-coded instances such as header and background image, we implemented a deep refactor in which all the shared functions for items such as drop-down menus, search forms, pagination components, and cards were put into globalized js files and consequently exported from. This way, instead of having the majority of each model page's code copy pasted from each other, they all fetched from one single file that contained all shared functions.