CS373 IDB Technical Report: Group 15,

WorldEats <u>Motivation:</u>

Our goal is to connect food enthusiasts to recipes, restaurants, and cities where they can find delicious food from across the globe.

<u>App Overview:</u>

Our website, worldeats.us, organizes information about recipes and restaurants into three models — recipes, restaurants, and recipes filtered by city. Data is gathered from multiple APIs and linked together through a webpage. Each model has a page of clickable instance cards, and each instance page contains attributes that could be useful to a customer.

<u>User Stories:</u>

Phase I User Stories:

User Story #1: Homepage — *"Would like to see a homepage with some relevant information. Maybe just a short description of what the site will be and how to use it. And also maybe an option to choose a random city/recipe/restaurant based on another input?"*

Response: We will address this problem in phase two of the project. At that point, we plan to have a homepage with information providing an overview of the purpose of our website and how to use it. The suggestion of a randomizer is intriguing. We will consider it as an extra feature to add in a later phase.

User Story #2: Connect the Models — *"Perhaps not in this phase but I would like to see some more interconnectedness between the various models. Maybe allow the user to designate a specific city and the restaurants will change based on the chosen city. Or the user can pick a certain recipe and then they can see restaurants those recipes are available in and what cities those restaurants are located in?"*

Response: Connecting the three models is our top priority moving forwards. We have already connected the restaurants displayed to those mentioned on the cities page and plan to do much more in the future.

User Story #3: More Information on the Primary Recipes Page — *"As a user, I would like some more information earlier when I first click the recipes page. Currently, other than the recipe name and picture, all I have is the number of upvotes. I am a little confused about how exactly* *those upvotes were determined or where they came from so maybe some clarification there. Ideally, I would like to see the cost, time to make, and total calories under the name so I can decide even faster which recipe I am interested in."*

Response: All of the information mentioned is scrapable from the API. Including more detail in the primary recipes page is a justifiable request. We are likely to heed this user suggestion in the next phase.

User Story #4: Cities Page — Add Ability to Sort + Little Bit More Relevant Information — *"As a user, I want to be able to sort the cities based on the ratings so I can ideally choose from the top-rated cities as an example. To add onto this, if I am using the website primarily for food and finding restaurants, I want the city ratings to be based on food and not overall (I'm not sure if you guys are using overall ratings or food ratings but if you guys are using food ratings then ignore this part)."*

Response: The city data is currently the weakest of the three models. We plan to have a recipes API feed into a cities API in later phases, a part of which would be either finding or calculating food ratings on our part.

User Story #5: Restaurants Page — Add Ability to Sort Based on a Variety of Factors — *"As a user, I would like to be able to sort the various restaurants on the page by a couple of different factors. These factors can include but are not limited to price, ratings, type of cuisine, and maybe also its distance from me."*

Response: This feature should naturally follow once complete dependence on the restaurant API is established. It is natural for a user to want to sort cards, so it is certainly on our to-do list.

Phase II User Stories:

User Story #1: Show Tools and APIs Used in the About Page.

Response: Crediting the tools and APIs used is imperative to our team. We have already started doing so in this phase of the project by linking to the APIs and a couple of the most frequently used tools, and we will continue to do so as this project grows in scope and ambition.

As of Phase III, this has been resolved. All major tools and APIs used are linked in the About page in keeping with our desire to credit those who have made this project possible.

User Story #2: "As a customer, I would like to see cities from every state in the U.S., so I can know where some good restaurants are."

Response: We currently have a variety of cities available to browse through, including all cities with populations above 400,000 people in the United States, but we do not have any functionality for states due to our desire to provide a truly international experience. Instead, we aim to implement search bars in the next phase of the project, which will enable users to search for their desired city, taking care of this customer need. Thank you for the idea!

This suggestion was partially rejected, as the website's aim is to create a truly global experience showcasing food and restaurants from across the world. However, we have included all cities with a population of 400,000 people or higher, which we hope will cover every major city not just in the United States, but also the world. Restaurants are linked in each city's page, providing the original desire of the customer.

User Story #3: Add Links Between Recipes/Restaurants/Cities — *"As a customer, I would like to select a recipe and see which restaurants nearby serve that dish/recipe. Or, if I like a certain restaurant, I would like to see where I can find it."*

Response: The links between the three models are beginning to grow in this project phase. In the cities model, one can see the most common cuisines in the area, while in the recipes model, it is possible to get a recipe by cuisine. In later phases, we intend to add more connections, such as viewing the different locations of a restaurant and possibly stating the origins of various recipes.

The links between the three models are progressing as of Phase III. Currently, every item page links to items from the other two models related to it. There is certainly more that can be done on this front, and it will be achieved in the forthcoming Phase IV.

User Story #4: Potentially Add Input for User Location — *"As a customer, I would like to input into the site either my current location or destination if I'm planning a trip. Based on the location I provide, the site can then tailor the recipes/restaurants based on the local cuisine of my location."*

Response: As we establish further links between the models, the above suggestions will be implemented, as discussed in the previous user story. Inputting a location to get restaurants or recipes local to said place will almost certainly be one of our main goals in the next phase when we implement searching, filtering, and sorting in all three models.

Inputting a location to get restaurants or recipes local to said place will almost certainly be one of our main goals in the next phase. Our team sees the usefulness of such a feature, only failing to implement it in Phase III due to time constraints. This will be a leading feature going into Phase IV.

User Story #5: More Detailed City Rating Information — *"As a customer, I would like some more information on how a city is given its rating. I want to be able to see what the city is ranked for its food instead of just simply an overall ranking. Or perhaps, a few different ratings for different attributes like ratings for overall, food, entertainment, etc."*

Response: This feature will certainly come under consideration in a future phase. Currently, we display a food rating for each city, calculated by averaging the Yelp ratings for the restaurants found. While displaying other ratings can provide users with more information, we still wish for the website to be primarily food-oriented. However, this is a fantastic suggestion — we will mull over this in the future.

Although we were unable to make time for considering this feature, it is certainly intriguing. The website's main purpose is to rate a city's food, and the rating displayed reflects that goal. However, it would certainly be interesting to explore further aspects of cities and expand the website's horizons, so we will be discussing this and its feasibility in the coming days.

Overall Assessment:

We were able to resolve almost all of the basic user stories here as our website naturally grew and evolved. However, some of the more advanced features, such as multiple city ratings, had to be scrapped due to a lack of time. These more ambitious user stories would have definitely been implemented given infinite time.

Visualizations:

Recipes: For recipes visualization, we utilized a custom d3 bar chart component to display the frequency of different dish types in our database. Originally, we had hoped to show something more interesting with a scatterplot, but there turned out to be little correlation between variables like recipe cost, likes, number of ingredients, and so on. It turns out that people like all types of recipes pretty evenly.

Restaurants: For the restaurants visualization, we utilized the react-google-maps api to create a map of the world component. Within this map, we placed markers of 100 random different restaurants with their image placed on top of it. This visualization shows the distribution of all the different available restaurants contained within our database, and introduces the functionality of clickable markers where a user who is interested in a restaurant can click on the marker to be directed to its page to learn more information.

Cities: For the cities visualization, we also utilized the react-google-maps api to create a map of the world as a component. On this map, instead of a marker, we created circles of all the available cities where the radius, and therefore the size, of the circle represented the size of the population of said city. This visualization shows the user the distribution of some sample of cities within our database as well as the general population size of any given city. Any user who is interested in knowing more about a city is also free to click on a circle to learn more about the city by being brought to a model page for that city.

Provider:

To create visualizations for our provider, Capitol Capital, we utilized 3 custom made d3 components: a pie chart, bar chart, and scatter plot. Visualization one is a simple pie chart showing the distribution of political parties for politicians in our developers data (a sample of 200) Visualization two shows the top 10 most popular stocks held by politicians in the dataset (also sample of 200). Visualization three simply features a bar plot of politician year first elected vs date of birth. Please note, our developers API was not CORS enabled. This disallowed us from making API calls in code. To get around this to have some visualizations for this phase, we simply saved responses from their API using Postman in a json file in our project.

RESTful API:

We designed a RESTful API that fetches data for recipes, restaurants, and cities from their respective APIs using Postman. The Models section contains further information about the specific APIs. The resultant design is at https://documenter.getpostman.com/view/25838982/2s93CExciz.

Here are our WorldEats API Endpoints:

GET Get Recipes

http://worldeatsapi.link/recipes?getBasicData=True&limit=1

Get all recipes. Can optionally elect to only get some data, or a subset of the recipes.

Parameters:

- limit: an int, defines how many recipes to get
- getBasicData: True/False, parameter presence indicates to get at least basic recipe data for every recipe
- getEquipment: True/False, parameter presence indicates to get recipe equipment list
- getInstructions: True/False, parameter presence indicates to get recipe instruction list
- getIngredients: True/False, parameter presence indicates to get recipe ingredients
- getCuisines: True/False, parameter presence indicates to get recipe cuisines list
- getDishTypes: True/False, parameter presence indicates to get recipe dish type list
- getDiets: True/False, parameter presence indicates to ger recipe diets list

GET Get Specific Recipe

http://worldeatsapi.link/recipes/1

This API retrieves recipes containing the requested ingredient, where 1 is a placeholder for the desired ID.

GET **Get Recipe by Cuisine**

http://worldeatsapi.link/api/models/by-cuisine?cuisines=Mexican&number_related_models=2
Returns the number of cities, recipes, and restaurants tagged with the type of food specified.

GET **Get Specific Restaurant**

http://worldeatsapi.link/restaurants/1
This API retrieves data about a specific restaurant, where 1 is a placeholder for the desired ID.

GET **Get Specific City**

http://worldeatsapi.link/city/1
This API retrieves data about a specific city, where 1 is a placeholder for the desired ID.

GET **Get Specific Restaurant**

https://worldeatsapi.link/city/get-ids?name&country&score&price
This API gets the IDs of all cities that match the search parameter values of name, country, score, and/or price. At least one of the parameters' values is required for the call to work.

GET **Get City IDs by Price**

http://worldeatsapi.link/city/sort-by-price
This API sorts all cities by their average Yelp restaurant price. The call will always be appended with either "?sort_value=asc" or "?sort_value=desc" to specify whether to sort in ascending or descending order.

GET **Get City IDs by Rating**

http://worldeatsapi.link/city/sort-by-score
This API sorts all cities by their average Yelp restaurant rating. The call will always be appended with either "?sort_value=asc" or "?sort_value=desc" to specify whether to sort in ascending or descending order.

GET **Get Recipe IDs**

https://worldeatsapi.link/recipes/get-ids
This API returns the IDs of all recipes that match the search specifications, including at least one of min_calories, max_cook_time, max_calories, max_instructions, max_cost_per_serving, max_ingredients, and name.

GET **Get Recipe IDs Sorted by X**

https://worldeatsapi.link/recipes/sort-by-X?sort_order=

This API returns the IDs of all recipes sorted by attribute X. The parameter sort_order must be provided with either "asc" or "desc" to clarify the order in which to sort the recipes.

GET Get Restaurant IDs
https://worldeatsapi.link/restaurants/get-ids
This API returns the IDs of all restaurants that match the search specifications, including at least one of name, price, min_rating, and min_review_count.

GET Get Restaurant IDs Sorted by X
https://worldeatsapi.link/restaurants/sort-by-X?sort_order=
This API returns the IDs of all restaurants sorted by attribute X. The parameter sort_order must be provided with either "asc" or "desc" to clarify the order in which to sort the restaurants.

Models:

- **General Information:**
  - Each model has three layers - the overall page, the item card, and the item page. There are three pages, one for each model, that display cards correlating to their topics of interest. Using pagination, the cards are split into multiple pages, with twenty cards occupying each page. Each card then holds basic information specific to its item - for example, a restaurant card would be titled with the name of the restaurant, provide a picture, and list the rating and cost of said restaurant. Upon clicking a card, the individual item's page is displayed, containing further information about the specific item as well as links to other models.
  - Across all models, pagination is employed. First, the total number of pages needed is calculated using the total database entries and the fixed value of twenty cards per page. Within useEffect() in each model page file, each item is given a page number to occupy. Then, a "pagination container" is displayed at the top of the page, with two arrows set to go to the next and previous pages respectively when clicked. Finally, the current and total number of pages are printed in the format of "Page {currentPage} of {totalPage}".
  - Searching, filtering, and sorting are all features provided in each model page.
    - Both global and local search bars are implemented through a self-made SearchBar.jsx file, which wraps an input field around a search icon image and an input text field. This is then included in the HTML display of each model page. Upon submitting the query, a call is made to a callback handler, which is defined as handleSearch in each model page file. The function handleSearch then proceeds to fetch data from the database using a specially made search endpoint with "/get-ids". Using "await

fetch(endpoint)" and "await response.json()", the data is fetched, and the IDs are set with the resultant JSON data before reloading the page with only searched entries. Search terms are further highlighted using the titleHighlightSubstring attribute of each card. If the search input is empty, all items of the model are displayed. The global search bar searches all models for results, and redirects the results to a separate search results page, defined in SearchResultsPage.js.

        - Filtering is accomplished by creating separate filter files for each model. Specific attributes, generally integer or float values (such as the number of ingredients for the recipe model) are selected to be the attributes to filter by. The filters are displayed as a mix of sliding bars and dropdown menus, coded using a ReactSlider and HTML "<select>" and "<option>" respectively, which the user can adjust before submitting their choice of filters. Upon submission, an endpoint is created with all of the given attribute values before being fetched using a route defined to handle such endpoints by querying the appropriate places. Similar to search, the fetch identifies all of the IDs of entries matching all of the required values for the attributes, and returns a subset of the data matching the criteria. This is then displayed. A "Clear Filters" button is also provided. When the user clicks on it, a simple fetch of all of the model entries is performed, restoring the page to its original state.
        - Sorting is represented as a dropdown menu on each model page. The user can select which attribute they would like to sort items by, and in which direction. Upon clicking, a handleSort function is called. Each model page file has a handleSort function. handleSort then checks which option was selected, and invokes the appropriate route endpoint to take, which is configured to accept a "sort_order" parameter depending on whether to sort in ascending or descending order (link format: worldeats.api.link/model/sort-by-attribute?sort_order=asc/desc). handleSort appends "asc" or "desc" to the endpoint before allowing the route to take over. Each route queries for all entries and calls order_by on the attribute to sort all entries appropriately. The ordered set is then fetched using the standard "await fetch(endpoint)" and "await response.json()" before being outputted as JSON data.

- **Restaurants:**
    - Scraping APIs and Loading Them Into the Database
        - In "back-end/populating_database/restaurants/restaurant_requester.py," we set up a cities endpoint and access 20 cities stored in our database for the cities model, looping through them. For every city, we call Yelp's API using the city name to retrieve the top restaurants of that city, store the

        response, and convert it to a JSON object. Then, we clean the data for different attributes. With the restaurant's Yelp ID retrieved from the first API call, we call the Yelp API twice more to retrieve various attributes, such as reviews and specific hours of operation. For every city, we are creating a map with two keys: city and restaurants containing the city name and an array of information maps for each restaurant. This map then gets stored in a list called "objects."

        - At the end of the "back-end/populating_database/restaurants/restaurant_requester.py" file, a function named get_processed_restaurants accesses every element in the objects list and creates a Restaurant object, storing all of the attributes located in the map in the objects list. It returns an array of all of the Restaurant objects. The Restaurant object class is stored in "back-end/populating_database/restaurants/restaurant_class.py".
        - "back-end/populating_database/restaurants/restaurant_populate_database.py" calls get_processed_restautants() and passes the returned value into the "insert_into_database()" function in the current file.
        - insert_into_database() accesses each Restaurant object and stores it in the database.
            - All non-object attributes (such as an array or map) get stored in the restaurant table.
            - Each key of the "open_hours" attribute (a map) of the Restaurant object is stored in the "restaurant_hours" table.
            - Each item of the "avail_for" attribute (an array) of the Restaurant object is stored in the "restaurant_avail_for" table.
            - Each item of the "images" attribute (an array) of the Restaurant object is stored in the "restaurant_images" table.
            - Each item of the "reviews" attribute (an array) of the Restaurant object is stored in the "restaurant_reviews" table.
        - "back-end/populating_database/restaurants/restaurant_sql.txt" stores the format of the tables for restaurants for the database.
    - Accessing the database and displaying the information
        - In "src/App.js," route paths are made to Restaurants for "/restaurants" and RestaurantPage for "/restaurants/:id"
        - In "src/pages/Restaurants.js," we create an endpoint for our database to access the different restaurants and their attributes. We create a RestaurantCard for a slice of the restaurants (where the slice depends on the current page) and pass in the restaurant's database ID as an argument.
            - This is also where pagination is implemented for Restaurants.
            - This file is styled in "src/pages/Restaurants.css."

        - In “src/components/RestaurantCard.js," we create an endpoint to access the restaurant using its database ID parameter and attributes. We populate the display card with the attributes.
            - This file is styled in “src/pages/Restaurants.css.”
            - When you click on a RestaurantCard, it takes you to the RestaurantPage.
        - In “src/pages/RestaurantPage.js," we create an endpoint to the database and access the restaurant information using the restaurant ID of the RestaurantCard clicked on. We use these attributes to populate the page.
            - This file is styled by “src/pages/RestaurantPage.css.”

- ## Recipes:
    - Data was gathered for recipes from the Spoonacular API. Three Python scripts were then used to populate the database in a preferable format for the project:
        - requester.py - requests Spoonacular API for a random recipe, appended second API request for nutrition information to the JSON response.
        - request_parser.py - goes through the contents of the API response to get relevant data and put it into a custom data structure.
        - populate_database.py - uses data parsed from request_parser to populate our database using psycopg2.
    - Back End:
        - **Currently, many SQLAlchemy models are defined in models.py regarding recipes, as data that came in lists were put into tables (instructions, ingredients, etc.).**
        - /recipes/id is a route defined in the main flask app.py that returns relevant recipe information for a recipe with a given ID.
    - Front End:
        - Our backend receives a request for the recipe information for a given ID. This is used for various components, such as RecipePage.js and RecipeCard.js. Appropriate elements are then populated with the response data from our backend API.

- ## Cities:
    - Data was gathered using the CountriesNow, Yelp, and Bing Images APIs. A Python script is used to combine this data. It filters our dataset to include cities with 400,000 people or more.
        - citydata.py - requests a list of cities and their population from the CountriesNow API, then queries Yelp data for the 50 most relevant restaurants in that city, averages their data, and pulls an image from the bing API.

- Back End:
    - The SQLAlchemy models are defined in models.py, similar to the recipes, but under the route /city/id.
- Front End:
    - Requests the backend for the needed city/cities and displays information based on the response. City.js, CityPage.js, and CityCard.js are the relevant files for displaying the city page and the list of cities page.

Tools:

We used React and React Router to implement the front end and Bootstrap with CSS to style it. Using Namecheap, we claimed the worldeats.us domain. Our backend server relies on AWS Amplify, which provides the RESTful API that the front end consumes, and Flask, which hosts our API calls. The data is stored in a PostgreSQL database with tables for recipes, restaurants, and cities within an AWS Relational Database Service, with SQLAlchemy serving as the framework for the database. Using Postman, the RESTful API was designed. For testing, Jest and Selenium were used for the front end, while Postman and unittest were employed for the back end. Finally, we used Microsoft Teams and Zoom for communication and VSCode as our development environment.

Hosting:

- The database used is a PostgreSQL database with tables for restaurants, recipes, and restaurants.
    - This database is stored within AWS Relational Database Service for there to be constant uptime in being able to retrieve
    - We have inserted the correct information in a multitude of different ways:
        - First, we scraped the information from our APIs and placed them into several CSV files.
        - From these CSV files, we have created a table within the database corresponding to the same schema of the CSV file, and we have run a PostgreSQL command that copies the values from that CSV file to the correct table.
- To host our backend API calls, we utilized Flask and created a dedicated server on an EC2 instance that is constantly running a reverse proxy of our Nginx server configuration that lets the public listen to our locally hosted gunicorn Flask server.
    - Steps to reconfigure the API:
        - Log into the EC2 instance to which the current API is hosted.

            - This will require having the publick_key.pem file when SSHing into the server.
        - SCP any updated flask files pertinent to your change to the EC2 server and move them over to the ~/back-end folder.
        - Restart the necessary daemon applications.
            - Run the following commands to restart the Flask server:
                - sudo systemctl restart app
                - sudo systemctl restart nginx
        - Confirm your changes are live by pinging the IP address of the EC2 service.
            - https://worldeatsapi.link will bring you to the health page, and checking here will tell if the changes made are live.
    - Since **worldeats.us** is SSL certified with HTTPS, the API must also be HTTPS-certified. The following details how to set it up:
        - Create a Hosted Zone in AWS Route 53, upon which a domain name is given.
        - Configure the hosted zone to route to the EC2 endpoint.
        - Within the EC2 instance:
            - cd into ~/etc/nginx/sites-enabled
            - vim into the file ‘flaskapp’
            - There, the configurations to nginx and the server_proxy will be available.
                - Specify the reconfiguration of the file to listen in on port 443 to allow for SSL HTTPS calls to said server.
            - Utilize **certbot** to configure this.

Additional Notes:

- API data for provider visualizations does not come from API calls in code.This is because our provider did not enable CORS for their API. For this reason, we had to save the response data from Postman in a JSON file for use.
- Selenium GUI tests were implemented in JavaScript. This was a big mistake. We tried for many hours to get it working with the GitLab CI pipeline to no avail. Groups that got this working used Selenium in Python. All this said, the tests run fine locally, and you can take a look in our repo if you’d like to see them.
- Jest front-end unit tests work, but not when d3 is part of the project packages. The fix for getting this working in the CI pipeline was more than non-trivial.