CS373 IDB Technical Report: Group 15, WorldEats

Motivation:

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

App Overview:

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.

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.

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!

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.

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.

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.

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 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.

Models:

- **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 and
    - 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.