LA Homeless Helper Technical Report

Purpose

The purpose of this project is to create a database of resources that can help the homeless population in the city of Los Angeles.

User Stories

We received stories from our users and answered them accordingly.

Documentation

https://documenter.getpostman.com/view/28474521/2s9YJZ3jad

Models

Cities in LA County

- Unsheltered population
- Sheltered population
- Total homeless population
- Square miles of city
- Density of total homeless population

Shelters and Services

- Name
- City
- Hours
- Zip code
- Phone Number

Medicare and Medicaid Offices

- Name
- Address
- Hours
- Phone number
- URL for their website

Frontend

How the Architecture Works

- Every page has a folder in the components folder, and they are routed from the main page, App.tsx. Images are hosted in the assets folder.
- We used typescript, bootstrap, and CSS
- The frontend will query the backend and use the pagination provided by the backend to render the resources that the frontend needs

Assets

- The assets folder contains all the images that we need to render our frontend
- These contain images that we scraped from the google images API

Components

- This folder contains all the files that we need to render each individual portion of the website
- Nav contains the files to render the top navigation bar of the webpage
- Home renders our home splash page which provides a description of our website with links to other resources
- About contains the files to render our about page which lists some dependencies that we used to make our website
- Medical contains the files to render our medicare page which lists all of the medicare locations that we have in our database
- Resources contains the files necessary to render our shelters page which lists all of the shelter locations that are in LA county
- We also have a few files to deal with pagination and scraping the images from google images

Dockerfile

- Lastly we have a dockerfile which contains all the dependencies necessary to run our frontend by reading the dependencies from package-lock.json
- There are too many dependencies to list here, but the most important ones are listed below:
    - React
    - Google maps API
    - Node
    - Bootstrap
    - Word-wrap
    - css-tools

Challenges

- It was a learning curve to figure out how to do everything, as neither of us are experienced in Frontend development.
- Before we figured out bootstrap, we had a hard time with the framework for instances

Hosting

- AWS Amplify with Gitlab, domain hosted by AWS
- The Backend is hosted on a private EC2 server on AWS to access this we have to pass through a load balancer which handles requests from the outside world for this

Backend

How the Architecture Works

- We have several parts of the architecture right now:
    - main.py - the main python file that runs the flask API
    - query_database.py - this is the file that queries our mysql database to get the information from it
        - Remove_params.py - this is a helper file that removes unnecessary data from being added to the database
    - populate_database.py - this file is run to populate our mysql database
        - Requires find_coords.py to be run first
    - models.py - this file contains the models used to structure the database
    - query_apis - this queries the APIs that we used to get the information for the database
    - test_backend.py - this file tests the API such that it meets the requirements
- Toolchains: this part was written in Python using Flask, querying the database and setting it up was done using sqlalchemy, we also used docker to run the backend, and mysql for the database itself.

main.py

- This is the main Flask app that contains the functions that need to be called in order to query our API
- Each model route/function currently calls the respective function in queryAPIs which handles the logic to actually request the information. The route returns the json of the requested information as specified in our Postman documentation.
- We also are sure to include information to allow pagination by splitting our jsons that we return into multiple parts

- For more details on what each function does check out our API:
    - https://documenter.getpostman.com/view/28474521/2s9YJZ3jad

query_APIs.py

- This file handles the actual queries to the individual APIs
- query_API takes in a string related to what database is needed(“shelters”, “cities”, or “medicare”) and retrieves the information before returning it in the form of a list of dictionaries. Each dictionary contains all information about that instance.
- query_gitlab queries the gitlab API to get information about commits and issues. This method returns a dictionary mapping string names to a dictionary with keys “commits” and “issues” which map to an integer representing the number of commits made and issues closed respectively. We use two helper functions and then essentially zip those dictionaries together.
    - query_commits is a helper function that returns a dictionary mapping names to their number of commits.
    - query_issues is a helper function that returns a dictionary mapping names to their number of issues closed.

query_database.py

- This file wraps calls to the database for ease of access - requires a .env file set up.
    - The .env file contains information about the database that you are querying
    - Once the information is read from the env file a connection is setup with the database
- query_city takes in a string name such as “Unincorporated - Wiseburn” and returns the dict view of the city object of the requested city if it exists.
- query_cities returns a list of the dict views of all city objects.
- query_shelter takes in a string name such as “Zoe Christian Fellowship - Sfv Rescue Mission” and returns the dict view of the shelter object if it exists.
- query_shelters returns a list of the dict views of all shelter objects.
- query_medicare takes in a string name such as “Whittier Office - Social Security Administration” and returns the dict view of the medicare object if it exists.
- query_medicares returns a list of dict views of all medicare objects.
- Running the file by itself runs some basic assert statements that check if the list methods return the correct number of objects.

.env

- .env files are used for security to keep any passwords outside of the repo.
- For our use, we only need a few fields:
    - db_username - the username you will use to access the database, generally root or admin, ex: `db_username = “root”`
    - db_password - the password for the given username

    - db_url - the url for the database you want to query, localhost if you're using a local database or the link to the database if it is hosted elsewhere
    - db_port - the port your database is opened to, 3306 by default for mysql
    - db_database - the name of the database, if you made your own, it is whatever you named, our AWS instance's is named idbdatabase

populate_database.py

- Running this file should populate the associated database using the database name, url, port, username, and password in the .env file.
- Once a connection has been setup we will drop all existing tables to start from scratch
- Then we create new tables based on our models written in models.py
- We then filter out all unnecessary data using filter_json from remove_params.py before adding all of the data for each table
- We then need to setup relations, so we made a find_closest function to find the closest location for each row in the database
- We then use this function to find the closest location of the other types
    - For shelters we find the closest city and the closest medicare location
    - For medicare we find the closest city and the closest shelter
    - For cities we find the closest medicare location and shelter
    - We got the coordinates for each of the cities from find_coords.py to get approximate coordinates of the city
- We take the closest one of each of the other models and store a copy of the closest one in each row
    - We opt to do this because we wanted to have an easy way to quickly access these values at the cost of some extra space
- This file calls query_API.py's functions to get all the data, filters it, then adds it to the database and commits.

find_coords.py

- This script finds the latitude and longitude of each city.
- Uses a "cities.txt" stored in cs373-idb-01/tmp/cities.txt (../tmp/cities.txt relative to the file's location) which is a list of cities in the form:
  City of Agoura Hills
  City of Alhambra
  City of Arcadia
  ...
  Unincorporated - Whittier Narrows
  Unincorporated - Willowbrook
  Unincorporated - Wiseburn
- This file was necessary because in order to find the closest shelter and medicare office, we need to use some universal coordinates such as longitude and latitude which were not included in the cities database we are using.

- This data is stored in coords.json as a dict of city to latitude and longitude: {"City of Agoura Hills": [34.14791, -118.7657042], "City of Alhambra": [34.0691423, -118.1766937], … , "Unincorporated - Willowbrook": [33.919832799999995, -118.26061713344623], "Unincorporated - Wiseburn": [33.9131734, -118.3750514]}

Remove_params.py

- This file has two methods:
- Filter_json loops through the dictionary provided and removes all the keys that aren't provided in the list of keys that we want to keep
- We then convert all of the keys to lowercase so that way they fit with our models using recursive_convert_keys

Models.py

- This file defines our models using sqlalchemy so we can put them into the database
- We first define a base so we can base our existing files off of a sqlalchemy structure
- We then define our City, Shelter, and Medicare in the following way:
    - First we define the table name
    - Then define all the rows that we want them to have
    - Define a __repr__ method: this is just what gets printed out when we call print on a row
    - Define a to_dict method which allows us to convert a row into a dictionary

Test_backend.py

- This runs unit tests on our backend
- We do this by using Python's unit test framework
- We query our api website and validate the expected requests are the values that we expect them to be
- We ensure that the commits and issues are non-zero values for most of the group members
- We also validate for each model that each of the values that we compare match what we expect to return

Dockerfile

- This is the dockerfile that we used to build our docker image containing all the dependencies that we need for our docker file including but not limited to:
    - Flask - allows us to have our backend website up
    - Flask cors - allows for cross origin resource sharing
    - Requests - allows us to query the APIs we need
    - Geopy - allowed us to find the coordinates for each of our locations
    - Dotenv - allowed us to use an env file so the credentials of our database aren't leaked
    - Sqlalchemy - allows us to query our database using python

    - Tqdm - allows for a loading bar when running our populate_database for style points

Templates

- This contains the temporary pages for our backend so that we can test that our API was working properly as we made changes
- These template files might be deleted or changed since they aren't actually necessary for the API to run, but they do provide a rudimentary UI to our backend

Challenges

- This part of the project was fairly straightforward, so there were few challenges besides the GitLab API. There was a pagination problem which for issues we have monkeypatched, but we solved for commits. Unfortunately, we cannot use the same method for issues as we did for commits, and we will likely have to rewrite query_issues() later. Getting AWS up and running also took significant amounts of time.