MetroPulse-NYC: Phase III

IDB Group 6

- Alex Cabrera (gac2827)
- Kamil Kalowski (ktk582)
- Kyston Brown (kmb6273)
- Thomas Moody (tjm4482)

Motivation

The city of New York has recently published concerning statistics about the rise of hypertension problems in the city. In particular, the disease is disproportionately affecting black communities, with the prevalence of hypertension in the community being almost double (44%) that of the white community (23%). More concerning is the fact that of those in the community suffering from the disease, only 30% report having controlled blood pressure. With a number so low, we aim to help this community to more easily keep their blood pressure in check by finding local blood pressure testing centers, and in case of emergency find a facility able to help them stabilize their condition. The development of our website focused on answering the following questions (TODO?):

1. How can black NYC residents suffering from hypertension control their blood pressure consistently?
2. Where can black NYC residents find urgent medical care when experiencing hypertension-related blood pressure issues?
3. What facilities geared at aiding their high blood pressure can be found in their district?

User Stories

Update missing information in some instance pages

- This happened because some instances on our data source have incomplete information, so there is nothing we can do to fix it other than delete the entries (which we'd rather not do to avoid pruning important centers/hospitals). We decided to try to add placeholder text (e.g. missing title or description) for Phase 4 of the project, as a separate issue.

Integrate some rich, interactive media

- As with a previous request to a dynamic map, we wanted to avoid relying on third party APIs and embedded components on our website. Self-hosting some of these components is possible (like with OpenStreetMaps), but the memory requirements are too much for our server. We closed this issue as not planned.

it would be nice to have the image media be unique to each instance

- Not exactly a bug on the website. Some instances have missing information, so retrieving an image for them is impossible. The repeated images are placeholders so that there is something to show at least. In the case of neighborhoods, the incomplete instances are close together so there appears to be a lot of repetition. We want to try to avoid this by having rotating placeholder images for Phase 4.

background image on the home page

- Having a white background was a style decision to have the information pop-out more for visitors. The instance pages used to have background images, but those two were also removed to simplify the design. Thus, we closed this issue as not planned.

Display instance links as cards

- We decided to target this for Phase 4. We're currently avoiding it due to some instances linking to a huge amount of other instances. Partially alleviated by limiting the number of nearby facilities being linked to each other, but might find a different fix later on.

RESTful API

https://documenter.getpostman.com/view/29785582/2s9YJZ3jGy

Models

	Filterable/Sortable Features	Media

Test Centers	1. Name 2. Neighborhood 3. Borough 4. Zip Code	1. Map Image 2. Center Photo
Medical Facilities	1. Name 2. Neighborhood 3. Borough 4. Zip Code	1. Map Image 2. Facility Photo
Neighborhoods	1. Name 2. Borough 3. Population 4. Number of Test Centers 5. Number of Medical Facilities	1. Map Image 2. Area Pictures

Tools

- **React:** A javascript library for designing websites.
- **React-bootstrap:** A framework providing components from the bootstrap project reworked for react development.
- **React-router:** A react package providing a navigation system in-between different pages of the project.
- **Zoom:** Video conferencing platform used for group meetings.
- **Ed Discussion:** Chat/Forum hybrid for posting questions and communicating with other team members.
- **Postman:** A platform/app for building, testing, and deploying APIs.

Frontend Design

The frontend of our website is laid out in the standard manner for a React project, all inside the frontend folder of the repository. The entire code for each component we developed is inside the src folder, each file separated into their own independent folders inside based on their functionality.

We have five main categories of functionality:

- **Components:** We turned the cards and details (for individual instance pages) into individual components for testing purposes

    - **Instances (CenterCard, HoodDetails, etc):** Separate js files each representing an instance for a particular model, all using bootstrap components for better-looking functionality. Both files are laid out very similarly to one another. First there is an *Info object which is in charge of displaying all the information of each instance (Phone number, address, neighborhood, nearby facilities, etc.) which is passed in as an argument. This *Info object is then used as a component of a *Details object, which is created in *Instance.js which passes in the center fields from the api to create the details, passes a portion to *Info, and renders images next to the information. All instances also use some CSS to style themselves as cards, imported from the Instance.css file. For Phase II, they were slightly modified to match the newer API design. For example, all images needed to be adapted to use URLs rather than hard-coded images in the repository. Default images for instances missing urls were also added. URLs for connection between instance pages were also modified to use the new API's data. Additionally, new *Instance handlers (see pages folder) were created to fetch and orchestrate the models in the new frontend design (as described in the pages section).

    - **Pagination Indicator:** A tricky component; it takes in the total number of pages available for a given instance, the current page, and a current page handler. The handler is used to launch new pages when either selecting a page number button or using arrows to navigate forward or backwards. Naturally, this function takes in a page index as an argument. In terms of the component itself, it uses a map function to chain together a maximum number of buttons together, each tied to launching a particular page number with the aforementioned handler. These buttons are made to handle user clicks through the onClick attribute, thus resulting in them being interactive for the user.

    - **Navbar:** The navbar is comprised as its own component which is built from the Bootstrap framework. It has two modes which are search-mode

(which is of course yet to be implemented) and non-search-mode which allows the user to redirect themselves to any of the model pages (medical facilities, test centers, etc.). If in search mode, a search input field is rendered and a close button appears to get out of the search mode.

    - **Search Bar:**

- **Pages:** The individual model pages that display the cards are separated into two js files for each model (e.g. Pages/Centers/CenterInstance and Pages/Centers/CenterModel)

    - **Instances:** The instance files fetch the data from the api using the getCenter() function in utils/api. This gets all the data for an individual instance (a test center) from the api. The *Instance objects are used in App.js to route the card links to an individual instance page.

    - **Models:** The model files fetch the total amount of instances using the getCenters() function in utils/pagination.js. This gets the total_size field from the api which is the total amount of center elements. This is used so the number of total pages can be set for pagination purposes. It then maps through the centers array and displays a CenterCard for each individual instance of a center. It is used in App.js to route the navbar links (/test, /medical, /hoods) to the model pages which display all the cards.

- **Homepage:** The homepage gives an overview of hypertension and how it affects the African-American community in New York City. We give a quick statistic and provide a link to a well cited clinic (also a clinic that we use for one of the instances of our medical facilities) that gives a more in depth overview on Hypertensions causes, symptoms, and dangers. Using inline css and defining a styles object for a container, the header, and title, we return a div with a header with name of our application, another header explaining exactly what hypertension is, and a paragraph giving a simple statistic about the disproportionate number of people in the African American community suffering from hypertension, and finally a footer asking for support. We also instantiate a constant the stores the URL link that we use to redirect users to the clinical information page when a user clicks learn more on our webpage We export this HomePage component at the bottom which is later used as a route. This component is focused on engaging users and providing contextual information.

- **About:** This category is responsible for the About page, It corresponds to a JSON file with data about project members, a CSS formatting file for the page as well, and the main About.js file that forms the page itself. The JSON file has as

many members as decided upon and contains the data to fill out the Bio, username, email, etc. If the data on the Bio cards needs to change only the JSON file needs to be updated, if more data is added to the cards, the JSON must be expanded. The About page itself contains two functions dedicated to pulling from our Gitlab API to get commits and issues and the main app function to manifest the page. Because of the design of the Gitlab API, we needed to query 2 separate pages of the api url using a page=page_num parameter. The page itself is mostly a couple of text block sections and two card grids, of resource cards (resCard) and team member cards (team card). There are also href links to the documentation and installation pages of each resource which we used.

- **App(.js):** This is the file that ties all our work together. Here, we set up a navbar that is shown on every page of the website. More importantly, we set up routing for each of our components to be able to link and navigate to/from them from every other page (specially the homepage). For this, we use the react-router package, and create “Route” objects for every page in the website.

- **Sorting and Filter:** For each model page (e.g. pages/Centers/CenterModel.js) there are state variables for **sort** that keeps track of what attribute to sort by (name, id, population, borough), **order** (the sort order- ascending or descending) and separate state variables for what attribute to filter by (borough, neighborhood, zip code, FIPS county code) that store which value we want to filter (e.g. Brooklyn, Bronx, Manhattan, or Queens for **boroughFilter**). Each state variable also has a handler that sets the variable based on what the field is set to in the drop down menu from components/CustomDropDown.js. These state variables are passed into the get_[model]_length() (to get how many instances of each model are after being filtered for displaying the amount of results) and fetch_[model]() functions which are defined in utils/pagination.js. These functions add the query parameters based on the filtering and sorting state variables to the backend URL.

- **Search page:** We have a general search page with 3 columns, one for each model that gets triggered when a user inputs a proper piece of text in the nav bar search. This search bar is a component within our navbar.js. If the user doesn’t type any text, we prompt them to re-enter a text. Once they enter a valid search text, we query our backend search api endpoint with the string the user just entered and return a json response. We are then rendering the same card objects through these json responses. We also use regex to highlight the search terms that the user looked for, return a span and mark.

Backend Design

The backend was designed around the requests, Flask, SQLAlchemy, and Flask-SQLAlchemy libraries, with the final database being hosted on Amazon RDS and the final API being hosted on an AWS EC2 instance. The design is as follows (Starting on the backend folder of the project):

- **Scrapers (wiki.py, medical.py, etc):** These files handle scraping all third party APIs and are used to populate the database. In particular, the medical.py, neighborhoods.py, and centers.py files fetch the primary data, fill out dictionaries to use, and call additional helper scrapers to grab external information. Our additional scrapers handle querying google for map images (google_static_maps.py), google for business images (google_images.py), and wikipedia for neighborhood information (wiki.py). All APIs used use the REST protocol, and are queried through the requests python library. Additional formatting is done with the json library.

- **Database(.py):** This file handles all database setup and population, using Flask-SQLAlchemy models. It first initializes a flask app normally, and then sets sqlalchemy parameters to point to our database, hosted on an RDS instance (using PostgreSQL). Here, each “instance” has its own separate model, which gets populated by calling each scraper and putting in all the information. Population occurs in the “populate” functions, which are orchestrated to run by the “populate_database” function, which is in turn run in main. Additionally, the “final_relations” function sets up all SQL relationships between centers and hospitals, which due to their many-to-many nature require the hc/ch association tables. Neighborhoods also establish relationships to hospitals and test centers, but do not require a separate “relation” function due to backpopulation being handled by SQLAlchemy.

- **Endpoints(.py):** This file handles actually setting up the API using flask for each endpoint. This imports the db/app created in “database.py” and queries it to retrieve information to show in each endpoint request. All information in each request is deserialized with the “marshmallow-sqlalchemy” library, except for the relationships which are turned into JSON dictionary objects with only the important parameters. Each endpoint (with multiple results available) accepts pagination parameters (page, per_page) and multiple query parameters defined in postman (borough, nta, etc). These are internally processed by sqlalchemy and responses are built and sent back as JSONs. There are also “specific” instance endpoints available, which allow you to get a single result.

- **Search API:** Part of endpoints.py, but with a significantly different design. Search is implemented through postgres specific features rather than standard sqlalchemy built in features. Specifically, we use ts_vector and ts_query functions to perform a full-text search of our desired parameters, and filter our query by the result. A problem with this approach is that it will only perform text matching, but no substring matching. This is alleviated by the inclusion of prefix matching in postgres with the (:*) wildcard, but that is as good as it gets.

Hosting

Frontend

We set up an account for the AWS Console. We navigated the Amplify service, clicked connect app and connected our GitLab repository. We then edited the amplify.yml and build settings to ensure our app was building properly. We then set up a domain (https://www.metropulse.link) with Route 53, navigated to hosted zones, selected the public hosted zone, and clicked create. We navigated back to Amplify and went to domain management, and added the domain we just created in Route 53. The record sets were luckily set up for us, including the CNAME record ANAME record. We then just had to wait for the domain changes to propagate across the internet, which took approximately 20 minutes which was great as it can sometimes take up to 48 hours. SSL was enabled naturally by Amplify. It's also worth mentioning that Amplify uses a AWS Cloudfront distribution behind the scenes to ensure the application works with low latency for anyone who wants to use it across the world. A little bit on how CloudFront achieves this: CloudFront caches content in multiple data centers around the world, persists a pool of persistent connections to origin servers, and continuously monitors the health of the network and broader internet.

Backend

The API, a flask app, is hosted on an Amazon EC2 instance server. Elastic Compute Cloud (EC2) is essentially a virtual server that provides resizable compute capacity. We used an Ubuntu server with 32GB of storage, ssh'd into it, set up docker, and used gunicorn as our Python Web Server Gateway Interface HTTP server. Finally, we just detached ourselves from the docker container and let the magic run without any of our manual interference. The database is a PostgreSQL instance hosted on Amazon RDS. The security permissions were modified to accept requests from any IP address, but kept the username/password permissions for writing to it. No additional modifications were made to it.

Challenges

We ran into these challenges:

1. **Project RFP:** It took our group a long time to understand the project prompt and find an appropriate topic to build upon. Once our project idea was approved, we had less than a week left to build the website, so we had to more strictly partition development time into our schedules.

2. **AWS Deployment:** We struggled with properly deploying our website through AWS amplify. It did not give any errors during the build process, but it would not appear in either our domain name or AWS's own address. We ended up trying to transfer our domain to AWS Route53, but that was taking too long, so we ended up setting up a new domain with AWS Route53 itself which proved to be seamless. All the correct host names were automatically set up and we didn't have to worry about any DNS probing.

3. **JavaScript/React:** None of us were very familiar with JS, React, APIs, or Postman before this project. So, it was challenging to figure out how our tools worked while we were actively using them. However, we overcame the difficulties before the project was due.

4. **Google API Scraping:** Google is very strict with its querying limits on their Places API, so we had to be very careful with testing so that our free credits did not run out. Unfortunately, one of our models had 1000 instances which exhausted our api key after a few database deployment attempts. Thus, we needed to write helper functions to modify the database tables with a new api key.

5. **Frontend Refactor:** We had to completely restructure our frontend in order to make it compatible with the Selenium tests, since the tests have to be run on an individual component. This took us quite a bit of time to work out.