Technical Report: IDB Phase 2
Group 5 - StormShelters
Revised 10/20/2023

Purpose/motivation

This website is created to provide resources to people in Harris County who have been displaced due to natural disasters. The site highlights both the resources that are available as well as the disasters that have occurred to both provide assistance and bring awareness to the scale at which people are displaced.

Website Architecture

The frontend of the website was made using React and all relevant code is in the frontend folder in the root directory of the project. React has “components” which allow for a single piece of code(for example the navbar) to be used on multiple different pages. It also allows for dynamic rendering of page data. For styling, we are primarily using Bootstrap. Our navigation bar, splash page carousel, and card layout have all been styled using Bootstrap.

The models and instances are comprised of components organized in a hierarchical structure. In the <model>.js files, up to 10 instances of each model are displayed per page. We use React state variables to modify the page number, and our backend will return a desired range based on which page is requested. Data from the JSON API return is sliced into groups of 3, and each card display then creates and displays a Model component given the corresponding chunk.
The ‘cards’ are generated by creating Model components, which are generated in the <model>Model.js files. This parses out the desired fields from our backend API call, organizes them in HTML fields, and provides a simple React link to the instance page. Instance pages are created in <model>Detail.js and generate the component that the Model pages link to.
Page linking is achieved in app.js using Route objects; each link to Component objects that are instantiated by importing the <model>.js files at the top of the file. This allows us to generate a unique URL for each instance while still utilizing the component templates.

The backend of the website is primarily built using Flask and all relevant code is in the backend folder in the root directory of the project. Flask aids in creating routes for different pages and sending the data to the frontend. Behind the Flask layer(app.py) sits SQLAlchemy(models.py and schema.py), which, upon request, queries our database and sends the data to Flask so that it can be packaged and sent to the frontend.

In the model pages, we make asynchronous calls into our backend using the querying syntax: “api.stormshelters.me/<model>/<desired id>”. If desired id is omitted, our API will return all data. We use built-in JSON methods to format the data, and parse it by creating a const variable per attribute of the JSON. In the instance pages, synchronous calls are used to avoid conditions where the component returns before the API has assigned the local object any JSON data (leading to runtime errors). An ‘id’ parameter is parsed in the Model page and stored in each component. This is used to query the database for only one instance of each model.

Our RESTful API data collection process begins in scrape_data.py, which uses the Requests library to filter out the queries we wish to make and store the results in JSON files. Then, in db.py, we read each of the JSON files per model (the Pharmacies model contains data from two APIs, so it reads in two files), loop over the attributes found in the JSON object, and prepare a MySQL query based on the desired fields. We store the raw data in an an extracted_data dictionary that can easily be parsed. Finally, the data is queried from our database in app.py, which supports pagination and individual querying. Schema are created using MySQLAlchemy that contain Model objects which map individual fields to columns in our database. The MySQLAlchemy 'db' object handles querying and takes each Model as an argument. The Schema are then used to return results from the query. This allows for fine-grained filtering of the requested data. Afterwards, each model's function returns the schema_dump and formats the results in a JSON-style POST.

For running the program locally, we rely on a docker container so that all developers can run the code without issue. The docker container is composed of two images, frontend and backend, which are composed by the docker-compose.yml file in the root directory. Generally, to start the docker container, we run `docker compose up`, and to restart the container, we first delete the containers/images using the Docker Desktop GUI, then run the command again.

Frontend

- Model (1) -> API request -> Model Cards (many) -> API request -> object that is parsed to display content
    - Model Cards (many) -> API request -> Instance Card (1 per model) -> Separate page that contains media

Backend

- RESTful API Querying -> JSON data files -> queried into MySQL Database
- Database -> queried into backend API
    - JSON output -> schema dump -> many queries -> Schema -> Models -> fields per DB column

API documentation

Our API: https://documenter.getpostman.com/view/29974721/2s9YJZ3jac

API's that we used:

- "http://api.weatherapi.com/v1/forecast.json" - Weather Data
- "https://api.yelp.com/v3/businesses/search" - Shelter Information
- "https://api.api-ninjas.com/v1/geocoding" - Locations
- "https://api.geoapify.com/v2/places" - Pharmacy data

Models (and attributes)

- Shelters/Food - id, name, url, is_closed, city, rating, address, phone, image
- Pharmacies/medical - id, name, city, address, longitude, latitude
- Cities - id, name, population, temperature, wind speed, condition, precipitation, image

Toolchains

- Development:
    - GitLab repository with CI/CD
    - Visual Studio Code
    - MySQL Workbench
    - Docker
- Website architecture:
    - React
    - Flask
    - SQL Alchemy
- Production hosting:
    - AWS Amplify - Frontend hosting
    - Namecheap - Domain name registration
    - Amazon Route53 - Domain name resolution service
    - AWS EC2 - Backend server hosting
    - AWS RDS - Database hosting
- Testing
    - Jest
    - Selenium
    - Python unit tests

Hosting

The website repository is managed through GitLab. We are using AWS Amplify to host the frontend. When changes are made to the GitLab repository, the amplify yml script is automatically run and updates the frontend. The domain name was acquired through Namecheap, and the AWS Route53 is used to provide the DNS service. We re-route the name servers in Namecheap to AWS.

On the backend, we have not yet figured out continuous integration. When we push code to the Gitlab repository, we have to manually ssh into the backend server, hosted on EC2, and then run the “remote.sh” script in the backend, which goes through the commands to restart the server.

Our MySQL database is managed through an AWS RPS instance.

Challenges

The biggest challenge was collecting our data into a database and successfully fetching that data from a backend API. It was considerably easier to either query the RESTful APIs directly in the frontend or to directly read from our JSON data files, but we found that in the end creating a backend with fine-grained querying helped to keep our design neater and easier to expand.

Figuring out how to piece all the technologies together was also difficult. Using Docker to create environments that satisfied our many dependencies helped a lot, but figuring out what libraries we were using, what versions we should use to avoid conflict, etc. was challenging.

User Stories

1. Rank shelter results by proximity

Not implemented. Given the timeline of when this suggestion was made and the scope of Phase 1, this is an unreasonable ask. For Phase 2, once we have a large number of shelters, the user's proximity is certainly something that we should provide as a filter criteria when sorting the models as this information would be useful for someone seeking refuge.
Update: I did not realizing that filtering will not come until Phase 3, so this user story is still beyond the scope of Phase 2.

2. "For the shelter and emergency relief it would be nice if clicking on the image of the pinpoint on map would take me to google maps so I can map it easily."

Implemented. This is a great idea for interactive media to embed into our website. Instead of creating a custom map with coordinates, we instead provide a simple embed into Google Maps. In a future stage, if we were to store the coordinates for each of the shelters, this may make it easier to implement the priority ranking suggestion.

3. "I would like the app that has a function enable to add shetlers to the favorite list."

Not implemented. This is beyond the scope of what we are planning this app to be. It would require some sort of user data management system as well as the creation of user accounts to save favorites. Although it is a good idea, we are afraid it would complicate the website a bit too much.

4. "I would like a way to remember which places I've volunteered at. That way I can continue to build connections with them and continue working towards the same cause."

Not implemented. Like the other suggestion about having a 'remembered' list, this is a good idea but would require the creation of user accounts. This is too personalized for the type of website that we are creating. Moreover, since we had to get rid of the volunteer model, I don't think this suggestion could be applicable anymore.

5. "As someone who would like opportunities to volunteer for the homeless, I'd like to see the locations of the volunteer opportunities displayed on the model cards so I can choose them easily."

Not implemented. As mentioned earlier, we removed the volunteering model due to API limitations. We do, however, have a Shelters model that can show places that may potentially have volunteer opportunities, especially during a disaster.

Phase 2 User Stories

1. "Make sure the dropdown menu works in phone's screen"

Implemented. Very important UI bug to fix! We replaced the existing Bootstrap Navbar with a React Navbar, which automatically re-sizes based on the screen size. This way, elements will not be hidden when viewed on mobile. Estimated: 20 min; actual: 20 min.

2. “One thing that came up for us that might be important for you guys to consider when working on the backend is to ensure that users cannot do a SQL injection and thus mess up the data in the database.”

'Implemented.' This is not an issue on our website because we have no user input. An SQL injection attack works by querying with malicious inputs, but the only insertion into our database is deep in the backend. Our API does take input, but it is parsed by Python and merely queries the database. Estimated: 0 min; actual: 0 min.

3. “As a user that lives in a city affected by a disaster, I would like to see a list of shelters that I can go to in my city. This should be shown in the city instances.”

Not implemented, yet. This is beyond the scope of Phase II. We will be able to add more granular querying to our backend, however, that will allow for filtering and relating models together. For now, we will just provide suggested models at the bottom of an instance.

4. “I'm not sure if this is just on my screen, but the margins of the text from the edges of the screen are a little small for me and make the text hard to read.”

Implemented. We increased the font size, increased padding between elements, and ensured that we were not displaying too much information at once. Estimated: 1 hr; actual: 45 min.

5.