

Technical Report

This report provides sufficient detail for an uninitiated development team to take over work on the project.

It is written for **software developers** (not end-users).

1. Motivation & Purpose

This site helps solve the problem that many low-income and underserved communities face: lack of access to food. This site connects people in need of food-related services to nearby food banks/pantries that can provide them with assistance. This site also serves the purpose of allowing volunteers, organizations, and sponsors to search for programs and food banks that can benefit from their time and contributions. Our final motivation behind this site is to allow food banks to increase their visibility and search for donor and volunteer aid in order to continue providing the vital services they do.

2. User Stories

- **Story 1:**

Our customer group suggested having an option to filter/sort foodbanks by their urgency level. We had actually already implemented this by the time the user story came in since we have filtering for each model page. And, one of the filtering options was over the instance attribute “urgency” for foodbanks, so this ended up answering their request implicitly. Estimated Time: N/A Actual Time: N/A

- **Story 2:**

Our customer group pointed out that our search mechanism produced results with a style was difficult to read and understand at a glance. We fixed this by changing the color scheme - originally it was white-on-white - and by arranging the results in a more neat, grid-like fashion. This made the UI much easier to read and use. Estimated Time: 15 minutes Actual Time: 25 minutes

- **Story 3:**

Our customer group pointed out a hold-over error from Phase 2 that we had yet fix with Phase 3 features: our programs’ service types fields were not properly populated, nor were they working with the filters correctly. This was fixed by our scrapers by finding more accurate and varied results based on the websites for these programs. The filtering, once implemented, then picked up these changes naturally, and it resolved the issue. Estimated Time: 30 minutes Actual Time: 20 minutes

- **Story 4:**

Our customer group noticed that the instance cards in our Foodbanks and Sponsors model pages were not easily distinguishable from one another since there were no borders between the cards. We fixed this by adding in light borders between the instances to make the card-like nature of the instances more readily apparent. Estimated Time: 5 minutes Actual Time: 5 minutes

- **Story 5:**

Our customer group had a bit of a misunderstanding of our sponsors model page. They mistakenly thought that it listed sponsorship opportunities for users, whereas it actually lists former and current supporters of the foodbank instances we have displayed on our site. However, we did implement their side point, which was to have tiers of contribution levels for each sponsors. We added this feature through our scrapers as an attribute of each sponsor instance. Estimated Time: 20 minutes Actual Time: 35 minutes

- **Story 6** Our customer group requested that foodbanks and programs have the ability to be sorted based on geogrpahical location. We chose to implement filtering and not sorting, but we did adapt this request for filtering by having filtering options by city and state for both foodbanks and programs. Estimated Time: 30 minutes Actual Time: 40 minutes

3. RESTful API Documentation

- **URL:** <https://www.postman.com/downing-group-7/dafranc-s-workspace/collection/uhwer5y/foodbank-api-v2>
- For each model, we have GET requests for obtaining either a specific instance via its ID, or you can get all instances for that model. We also have a filtering endpoint and a searching endpoint for organizing instance pages on the frontend.

GET Examples

Foodbanks

- GET `/v1/foodbanks` — returns a specified range of the list of foodbanks.
 - Query params: `size` (int), `start` (int)
 - Filtering params: `city` (String), `state` (String), `zipcode` (String), `urgency` (String), `eligibility` (String), `languages` (String)
 - Sorting params: `city` (String), `state` (String), `zipcode` (String), `urgency` (String), `eligibility` (String), `languages` (String), `name` (String), `-city` (String), `-state` (String), `-zipcode` (String), `-urgency` (String), `-eligibility` (String), `-languages` (String), `-name` (String)

- Example: <https://api.foodbankconnect.me/v1/foodbanks?size=10&start=1&state=TX&urgency=High>
- GET `/v1/foodbanks/<id>` — returns a single foodbank instance.
 - Query params: ID (int)
 - Example: <https://api.foodbankconnect.me/v1/foodbanks/123>

Programs

- GET `/v1/programs` — returns a specified range of the list of programs.
 - Query params: `size` (int), `start` (int)
 - Filtering params: `frequency` (String), `eligibility` (String), `cost` (String), `program_type` (String), `host` (String)
 - Sorting params: `frequency` (String), `eligibility` (String), `cost` (String), `program_type` (String), `host` (String), `name` (String), `-frequency` (String), `-eligibility` (String), `-cost` (String), `-program_type` (String), `-host` (String), `-name` (String)
 - Example: <https://api.foodbankconnect.me/v1/programs?size=10&start=2&frequency=Month>
- GET `/v1/programs/<id>` — returns a single program instance.
 - Query params: ID (int)
 - Example: <https://api.foodbankconnect.me/v1/programs/123>

Sponsors

- GET `/v1/sponsors` — returns a specified range of the list of sponsors.
 - Query params: `size` (int), `start` (int)
 - Filtering params: `name` (String), `affiliation` (String), `contribution` (String), `city` (String), `state` (String)
 - Sorting params: `name` (String), `affiliation` (String), `contribution` (String), `city` (String), `state` (String), `-name` (String), `-affiliation` (String), `-contribution` (String), `-city` (String), `-state` (String)
 - Example: <https://api.foodbankconnect.me/v1/sponsors?size=10&start=2&state=CA&contribution=High>
- GET `/v1/sponsors/<id>` — returns a single sponsor instance.
 - Query params: ID (int)
 - Example: <https://api.foodbankconnect.me/v1/sponsors/123>

Searching

- GET `/v1/search` — returns instance pages relating to the query phrase.
 - Query params: `q` (String)
 - Example: <https://api.foodbankconnect.me/v1/search?q=houston>
-

4. Models

- **Food Banks:**

This model holds all of the food banks and pantries that provide food-related services.

- **Programs:**

This model holds all of the programs that involve assisting food banks such as food drives.

- **Sponsors:**

This model holds all of the sponsoring organizations that support the food banks by making donations or by leading programs.

5. Instances

- **Food Banks:**

The food bank instances are individual food pantries. Their webpages contain links to the food banks' official websites as well images of the banks and/or their logos. Their attributes are as follows: name, city, zip, capacity, open hours, services, about, and urgency.

- **Programs:**

The program instances are individual programs/events. Their webpages contain links to the official websites for the programs as well as photos of flyers and other descriptive images relating to the event. Their attributes are as follows: name, type, eligibility, frequency, cost, about, and hosting foodbank.

- **Sponsors:**

The sponsor instances contain the corporations, businesses, charities, etc. that contribute to food banks either through monetary donations, food donations, or by hosting programs. Their webpages contain images of their logos as well as links to their official websites. Their attributes are as follows: name, contribution, about, contribution amount, contribution unit, affiliation, and past involvement (which itself contains the type of event, name, and date).

6. Frontend

The frontend of our site uses React. We have our landing page, about page, and then our three model pages as static (always existent) pages to which React routes service. The about page is populated with data in regards to our commits and issues from a small JavaScript program that calls GitLab's API to get information about our repository and our team members. The model pages

present grid-based layouts of all of the cards/instances present for that model. This information is retrieved using the range-based GET endpoint of our API. The instance pages themselves, which number in the hundreds, are created and routed through React dynamically and are populated with attributes and media by calling the ID-based GET endpoint of our API.

7. Toolchains & Development Workflow

- **Languages:** HTML, JavaScript, CSS, Python, JSX, Docker, SQL, and Make
 - **Frameworks:** Bootstrap CSS, Flask, and PostgreSQL
 - **Libraries:** We made use of the Bootstrap public library for CSS objects and formatting as well as the BeautifulSoup Python library for web scraping. We also used SQLAlchemy for modifying the database.
 - **Build tools:** We have a Makefile that condenses pushing and pulling, a .yml that runs pipelines to upload our source files to the AWS hosting site and run tests automatically, various Dockerfiles to maintain our Docker images for each part of the website, and a .gitignore that keeps our working space clean and free from clutter.
 - **Testing tools:** Our .yml runs all of our tests automatically. We have backend tests using Pytest and Postman as well as Frontend tests using Selenium and Jest.
 - **Version control:** We made use of GitLab for source control. We have a GitLab project/repository that contains the most up-to-date version of our site as well as all of our source code and media files. We occasionally forked the repository to preserve the website's state for grading purposes.
-

8. Hosting & Deployment

We obtained our domain name from Namecheap.com, where we currently redirect our domain name and its www subdomain to our account on Amazon Web Services' CloudFront hosting service. On Namecheap, we also have records for each of the two domains (www and non-www) in order to validate our SSL certificate, which gives our URL HTTPS access instead of HTTP access. The hosting of our site occurs on Amazon Web Services where we have an S3 bucket set up for static website hosting, which is handled by CloudFront. CloudFront is also the platform we used to obtain our SSL certificate.

We have a .yml file set up to automatically upload the build files that React produces upon deployment as well as run our tests. We have GitLab listed as an IAM on our AWS account, and we have the relevant keys stored in our GitLab repo's environment variables. We also have a Makefile that performs basic operations relating to pushing and pulling from the repository. Our .gitignore

currently just ignores instances of the Git log text file and various temporary debugging files that popped up throughout our development of the site.

9. Challenges & Solutions

Challenge: Google's Custom Search API kept returning error codes when the scrapers were run.

- After doing some research and debugging, we discovered that the issue was that we were making more requests per minute than is allowed by Google's threshold. We were well within Google's overall daily limit of 10,000 requests, which added to our confusion since we were seemingly well within the safe range. But, after discovering that the issue was the per-minute limit, we added in small sleep's between Google API calls in order to prevent the API from being overloaded. We also learned a bit more about how Google API returns its data, which allowed us to reduce redundant calls per instance into just one call that produced all of the necessary results.
-

10. Database and Scraping

For the web scraping, we predominantly used ProPublica's API as a jumping-off point for more in-depth scraping for each instance. We used ProPublica to get the names of all of the foodbanks, programs, and charitable organizations. Then, we made use of Google's Custom Search API to query specifically about an individual instance. We then used the BeautifulSoup library to parse through a specific instance's website, checking for the existence of the /about and /aboutus subdomains in order to obtain descriptive text. Based on certain keywords, we inferred the type of programs, type of organization, urgency of foodbank, etc. and populated the corresponding JSON fields.

Our database can be filled with the results of these scrapers in a few different ways. One, there is a stage in our .yml file that automatically runs the scrapers in a separate step in the gitlab CI/CD pipeline. Originally we created a docker image and ran it in AWS, but that isn't necessary as we already have it containerized in gitlab. We mainly leave this disabled however since clearing and refilling the database on every push is unreasonable. We also can run the scrapers manually to have the database re-populate after a significant change is made and new data is needed.

11. Paging

For paging, we mainly handle various page sizes with our backend API's GET endpoint that uses ranges. This range-based endpoint allows us to get chunks of instances in our chosen page size, which is 20. We then display the number of instances being shown on the model page's grid along with the total number of instances, obtained from using the full range on the GET endpoint. For filtering, we dynamically display the number of cards showing that meet the filtering requirements.

12. Filtering, Sorting, and Searching

Filtering, sorting, and searching are all performed primarily by endpoints in the backend. Our frontend lays out the UI and then calls the backend endpoints with queries and parameters matching the user input to the browser, whether its text input in the search bar or clicks onto our sorting/filtering dropdowns and boxes. For filtering, certain attributes have dropdown menus with pre-set values, but for attributes like cities and ZIP codes, we allow the user to type in the full value themselves. For searching, we have a separate search bar that will navigate to a search results page after clicking "enter" on the page. Highlighted results are then displayed that match the user's search phrase. For sorting, our implementation was to add an additional query parameter called "sort" to our GET range endpoint to determine the order in which the filtered instances are returned. Our frontend then handles this by registering user clicks on each sortable attribute as either an ascending or descending value passed to the sort function.

13. Visualizations

We used D3, a JavaScript library that offers convenient data visualization tools, to create dynamically generated charts of both our own site and our developer group's site using the respective API's of each site. We used D3 to create charts that explored different avenues of each site such as the representation of data across certain attributes like state and city. These charts were then populated by frontend calls to the API's using different filters to get the data relevant to the chart.