HowsMyAir Technical Report
CS 373: Software Engineering
Summer 2019

Group 3
David Bommersbach, Zelin Cui, Kevin Hsiang,
Jason Lihuang, Jamison Miles, Jason Sim

Domain: howsmyair.me

Table of Contents

Introduction

1. **Project Overview**
   Website URL: howsmyair.me
   GitLab Repository URL: https://gitlab.com/jamisonm/cs373-airpollution
2. **Motivation**
   Air pollution has been a serious issue since the start of the Industrial Revolution. Unfortunately, it continues to plague society today in various forms. HowsMyAir will educate the public on where air quality is the best and worst and highlight some of the health consequences of poor air quality. With HowsMyAir, the public will be able to check the air quality of a location on a given day and the related health risks. By integrating this disparate data, citizens can make use of this information before travelling and prepare for adverse air conditions. Users will employ our data to make healthy, conscious decisions based on where to live and visit based on how clean the air is. Air is important; after all, no one can live without it.
3. **Phase I**
   The first phase is focused on deploying a website with some sample information, as well as planning the API and future implementations. We also established several aspects of our workflow. This involved:
   a. Obtaining a domain name through NameCheap and setting up Amazon Web Services
   b. Collecting data for our three models from three RESTful APIs.
   c. Creating a React app and coding our web pages.
   d. Designing our own API using Postman.
4. **Phase II**
   In this phase, we expanded our website to include many instances of each model. This involved:

   a. Collecting the data and implementing our RESTful API.
   b. Creating the front-end framework to display the instances by consuming the API.
   c. Adding pagination to the model pages to organize the model instances.
   d. Making the UML diagram.
   e. Writing unit and acceptance tests to ensure our code functions correctly.
5. **Phase III**
   In this phase we implemented searching, sorting and filtering. This involved:
   a. Adding interactive components that allowed for updating of states.
   b. Creating functionality to dynamically build a URL request with the proper JSON query.
   c. Creating a search page with highlighting of searched elements over fields.
6. **Phase IV**
   This phase was mainly dedicated to visualizations, cleanup, and preparation for the presentation.

User Stories

1. **Phase I**
   a. **Deploy howsmyair.me**
      Actually deploy to the web so that everyone can see your initial static website. This should be the last thing you do for this phase.
      ***Resolved** in Phase I.*
   b. **Create a Flashy Splash Page**
      The page that a user sees when navigating to your URL should have a nice visual and easy access to the content of the site.
      ***Resolved** in Phase I.*

   c. **Create Page for Locations**
      Create a page where the user can see the instances of the Location model. Each location should have a few pieces of information associated with it.
      Resolved *in Phase I.*
   d. **Create an Air Qualities Page**
      Create a page where the user can see different levels of air quality. Possibly use an example location to demonstrate the level of air pollution, or give an example of some health risks associated with each level of air quality.
      Resolved *in Phase I.*
   e. **Create a Page for Health Consequences**
      This page should feature symptoms or risks associated with living in an area with a certain air quality instance. Each consequence should be tied to some air quality level or a range of levels.
      Resolved *in Phase I.*
2. **Phase II**
   a. **Add Some Visuals to Instances**
      For the locations especially, it would be nice to see some visuals for each instance on the model pages.
      Resolved *in Phase II.*
   b. Add More Instances to Each Model
      Once the API is up and running, generate a lot of instances for each model. They should appear on the model pages, and have their own individual pages as well.
      Resolved *in Phase II.*
   c. Finish the Documentation of the API
      Flesh out the documentation of the API, adding an introduction and overview.
      Resolved *in Phase II.*
   d. Rearrange Model Pages

To scale up the model pages, it would be a good idea to place the graphic and title above the table of instances, which could start to get long. For instance, put the map and "Locations" title above the list of cities on the location page.
Since the instances on each model page are now grids of cards, we removed the graphics to avoid making the pages too busy.

   e. **Put More Data on the Model Pages**
   For the lists of locations, health effects, and air qualities, put at least five pieces of information under each instance. This way the user knows a little bit before going to the instance's dedicated page to learn more.
   Resolved *in Phase II.*

3. **Phase III**
   a. **Make Maps Links**
   Maybe the maps on the Location pages could link to https://www.google.com/maps/place/{placename}
   Resolved *in Phase III.*
   b. **Update API Documentation**
   The API Postman docs still have default values for the Introduction, Overview, and more.
   Resolved *in Phase III.*
   c. **Make More of Cards Clickable**
   On the model pages, it would be nice to make the images link to their instance pages as well as the title. As of right now, the user has to click the name of an illness, location, or pollutant to go to their page, but ideally, they could also click on the image as well.
   Resolved *in Phase III.*
   d. **Clean Up Unknown Data**
   Some cards have empty data fields. It would be better if these unknown fields were hidden or at least just said "unknown".
   Resolved *in Phase III.*

e. **Make Cards Uniform**
   The Pollutant and Illness cards don't always line up as well as those on the Locations page. Making those a standard size, or at least matching card heights across rows, would clean up those pages.
   Resolved *in Phase III.*

Models

1. **Locations**
   The locations represent cities. Each instance includes general information people may want to know about each city, including the geographical location of the city (latitude and longitude), its air quality index, and its pictures.

2. **Illnesses**
   These are some of the health consequences and diseases associated with poor air quality. The instances include specific diseases that can be affected or caused by bad air quality, such as bronchitis and asthma.

3. **Pollutants**
   These are hazardous air pollutants that degrade air quality and potentially cause health problems. The instances list the Inhalation Cancer Potency, Inhalation Reference Concentration, and National Ambient Air Quality Standard, along with the 2016 national average production of the pollutants.

Front-end

1. Design

   The splash page features a hero photo of clouds with the navigation bar above. From there, users will be able to click on each of the three instances in addition to the About page.

   Each model page features a grid of instance cards to click on. The instances are paginated so that the model page shows nine instances at a time. After clicking on an instance card, users arrive at the instance page, where they can view more details about the instance alongside the instance photo as well as cards of related instances of two other models.

   The About page is headed by a description of our website and then a table of the group members and their associated information. Finally, there are links at the bottom to our repository, API, sources, and tools used.

2. Implementation

   To cut down on the components we need to implement ourselves, we used Bootstrap in conjunction with React. Components such as the navigation bar and instance cards were implemented by importing NavBar and Cards from reactstrap.

   The splash page is simple a background image with the NavBar component on top. Each model in the Navbar is linked and routed to the appropriate component. Within each model, each instance in the grid of Rows and Columns again links and routes to a separate component.

We obtained the data displayed on the instance cards and instance pages from our API endpoints using the fetch function.

For pagination on the front end, we used react-bootstrap components to achieve a working page bar. Our page bar is “dynamic” in that the neighboring page numbers change around the current page if the current page is not near the start of end of the range of pages. Each of our React components stored the state of the current page, and with a click handler the current page state would change based on the selection made.

On the instance pages, we have a react-bootstrap ListGroup containing the instance’s associated attributes. Each instance has a photo to the right of the ListGroup, and in location instances the photo links to the Google Maps page of the city.

The About page required scraping information from the GitLab API to populate the table of users with number of commits, numbers of issues, etc.

3. Testing

We created unit tests for our python code and any functions defined by us on the front end. We used Mocha for front end Javascript functions and the Python Unittest framework for back-end functions. We also created an acceptance test framework using Selenium that tests the React GUI.

Pagination

For pagination on the front end, we used react-bootstrap components to achieve a working page bar. Our page bar is “dynamic” in that the neighboring page numbers change around the current page if the current page is not near the start of end of the range of pages. Each of our react components stored the state of the current page, and with a click handler the current page state would change based on the selection made.

Domain and Hosting

1. We have two ElasticBeanstalk environments: one to host the front end server docker container and the other to host the backend server docker container. In the future we might want to host both within the same ElasticBeanstalk environment. The steps on how this was accomplished are below.

    - Step 0: Obtain custom domain name from NameCheap
    - Step 1: Create Elastic Beanstalk Environment
    - Step 2: Using CloudFront, create distribution to enable HTTPS, with the origin as the endpoint.
    - Step 3: With Route 53, create public hosted zone for your domain name. You will see 4 name servers.
    - Step 4: With the 4 Amazon names servers, in the service in which the custom domain was obtained, set the custom DNS to those 4 name servers.
    - Step 5: Request SSL Certificates through ACM for your custom domain name.
    - During this step, you will add validation records to your hosted zone. Wait for validation.

- Step 6: Go back to Cloudfront and edit the distribution created with the EB endpoint as the origin — in the general tab, add your custom domain names in the Alternate Domain Names section, and select the custom SSL certificate that you had just created.
- Step 7: In Route 53, create a recordset tying your custom domain name to your CloudFront distribution. Wait, and now you should be able to connect to the Elastic Beanstalk endpoint with HTTPS and a custom domain name.
- Step 8: Go to your Elastic Beanstalk environment dashboard, and in the configuration, make sure load balancing is enabled. Click modify load balancer, and add listener with port 443, and HTTPS. Select the custom SSL Certificate you had just created from before.
- Step 9: Create docker containers for the applications
- Step 10: Create an Dockerrun.aws.json file
- Step 11: zip application and upload using AWS environment GUI for ElasticBeanstalk

Database

1. Setup
   We decided to use PostgreSQL for our database. It is hosted in AWS and fairly simple to set up. We launched a DB instance using RDS, and set up security groups such that our server running in our EC2 instance was able to access the DB server's endpoint. We also whitelisted certain IP addresses for our personal devices so we could access and manipulate our database easily through pgAdmin4, which is a desktop GUI client for Postgresql databases. In creating our database tables, we actually utilized no SQL at all, and was able to do everything through high level function calls with SQLAlchemy (or rather Flask-Restless, which incorporates SQLAlchemy). All we had to do was define our tables in python as classes inheriting from DB.Models

from SQLAlchemy, and simply call create_all() and create_api() methods. Once our database was initialized with the right schema, we were able to easily upload .csv files into our database through pgAdmin4. We built the csv files using JSON data scraped from our various sources using python scripts. After scraping the necessary data, we decided to edit the data using Excel due to our group's higher level of experience with the program. When we finished cleaning up the data with Excel, we saved the files in a csv format and uploaded them to our database through pgAdmin4.

2. Schema
   We have six tables in total: location, illnesslocation, illness, pollutantillnesses, pollutants, and locationpollutants. The illness, location, and pollutants tables are the main tables that store our model instances. The other tables, illnesslocations, pollutantillnesses, and locationpollutants are the bridge tables that manifest the relationships between each model.

Searching, Filtering and Sorting

1. Design Choice
   We decided to implement our filtering using dropdown menus with preset options. This option would allow us to streamline our design and make it easier for the user to know what filter and sorting parameters are available. We saw websites that utilized dialogue boxes to allow a level of customization in the filtering for the user. However, we found these filters to be a bit confusing for us to use properly and this made us decided to avoid this course for our filtering. Our filtering functionality builds on itself: in other words, as you select a filter, the other filters will present options that only make sense (e.g. if you select east coast as a location filter, then the state filter will only then provide east coast states). As for searching, we simply kept a form field in the navigation bar. Our search functionality will

have a default search field for each model, but we allow for the selection of different search fields via a dropdown button (for the models that it makes sense for). The search results shown will also have the search string highlighted for all the fields that make sense.

2. Implementation
   The implementation of searching, sorting and filtering at the frontend involved building the correct URL containing a JSON string. On the backend, Flask-Restless by default takes query parameters as a json string, so everything was taken care of through the create_api() method provided by the Flask-Restless library. Searching, sorting, and filtering within our website are mainly driven by the "filter" and "order_by" json objects - searching is in fact a specific case of filtering where we use the "like" operator on values bounded by the "%" wildcard. On each model page, there are functions buildOrder, buildFilter, buildQuery, and getQuery to help construct the URL with the JSON string. Dropdown items manipulate the current state of each react component, and on clicking a dropdown item, the state of the component is updated, a new URL is constructed and fetched from, and the page is reloaded with new data. For highlighting, we used a react component that takes in a string to highlight as a prop.

RESTful API

1. Overview
   Our back-end API can be found at:
   https://api.howsmyair.me/, and can provide information on the air quality in a given area. It can also make some healthy recommendations upon determining the quality of air in a given area.
2. Endpoints

- GET https://howsmyair.me/location
  Retrieves all locations. Locations are US cities
  Parameters:
  i. Page (e.g. page =1)
  ii. q (json object)
  {"filters":[{"name":"city","op":"like","val":"%25re%25"}],
  “order_by”:[{"filed":"city","direction":"asc"}]}

- GET https://howsmyair.me/locationillnesses
  Retrieves all illnesses associated with a location

- GET https://howsmyair.me/illness
  Retrieves all respiratory illnesses
  Parameters:
  i. Page (e.g. page =1)
  ii. q (json object)
  {"filters":[{"name":"illness_name","op":"like","val":"%25re%25"}],
  “order_by”:[{"filed":"illness_name","direction":"asc"}]}

- GET https://howsmyair.me/pollutantillnesses
  Retrieves all illnesses associated with a pollutant

- GET https://howsmyair.me/pollutants
  Retrieves all pollutants. Pollutants in the database are chemicals or agents either known or suspected to cause respiratory damage
  Parameters:
  i. Page (e.g. page =1)
  ii. q (json object)
  {"filters":[{"name":"chemcial_name","op":"like","val":"%25r%25"}],
  “order_by”:[{"filed":"chemcial_name","direction":"asc"}]}

- GET https://howsmyair.me/pollutantlocations

  Retrieves all pollutants associated with a location

Model Data

Our first model endpoint will be https://howsmyair.me/location and will provide location data about cities across the United States of America. The data for this endpoint will be attained by the following dataset: http://geodb-cities-api.wirefreethought.com/

- Our second model endpoint is https://howsmyair.me/illness and will provide information on the possible illnesses from air pollution. The data for this endpoint will be attained by the following dataset: https://www.wikipedia.org/.
- Our last model endpoint is https://howsmyair.me/pollutants and will provide information on pollutants. The data for this endpoint is provided by https://pubchemdocs.ncbi.nlm.nih.gov/about, the EPA, and Scorecard.goodguide.com

Visualizations

1. Our first visualization model is a bar chart between each state in our database and its average PM2.5 air quality index.
2. Our second visualization model is a heat map of the number of pollutants and illnesses in each U.S. state. We fetch the data from the API and tally the totals in a dictionary, which is used to create the map.
3. The third visualization is a bubble map of the pollutants and their concentrations. The larger the bubble, the higher the concentration and vice versa.

Refactoring

There was a lot to refactor, since once we had working components we used them as templates and copied large swaths of code from one section to another, repeating ourselves often. This was especially apparent in the Search.js, where we had essentially copied all the code from each model page and lumped into search. However, in refactoring we first started with lower hanging fruit before moving to making components more reusable and portable, which required more understanding of how React works. Some of the smaller refactorings included pagination and reducing the number of lines it took: we made the code more compact. We also compacted the function for getting unique values for an object field - within a model page we previously had a function for each attribute of the model. The major refactoring came from making model instances and item components more portable. In order to do that, we used parent wrapper components so that we could pass data between components: "siblings" would call back to the parent to pass state information to one another. This allowed us to greatly reduce the amount of code and increase the clarity in the search page, as well as in the model pages.

Tools

1. **React**
   We used React to code the front-end designs of our website. We used it to link all the webpages together, create a navigation bar, create the model pages and decorate the user interface.
2. **Bootstrap**
   We implemented Bootstrap in our CSS framework so that our website will have a clean layout. We also imported and used components that Bootstrap already provides, such as the NavBar and Cards.
3. **Amazon Web Services**
   We used AWS to host our website. We also used it to transfer files from our local computers to the actual website.
4. **Postman**

We used Postman to design our API and gather data for our instances.

5. **Slack**
   We used Slack as our main platform to communicate with each other. We also set up GitLab so that “Slackbot” notifies us every time someone pushes to our repository.
6. **GitLab**
   We used a GitLab repository to upload our code and store all the files. It’s version control.
7. **Mocha**
   We used Mocha to write unit tests for front-end Javascript code.
8. **Selenium**
   We created a Selenium framework and a couple of acceptance tests to test the front-end GUI interface.