Food Me Once
Phase 3: Technical Report

Development Team:

- Christoper Chasteen, *Full Stack Developer*
- Gyuwon Kim, *Full Stack Developer*
- Shubhendra Trivedi, *Full Stack Developer*
- Brian Dyck, *Full Stack Developer*
- Nithin Pingilli, *Full Stack Developer*

Purpose

Food Me Once is being developed to provide a platform for users to gather information on the food security of communities across the United States. It combines disparate data sources about food security across districts/counties, political representation and legislation to present a well-rounded perspective. It will enable users to understand how political representation affects health outcomes as well as what actions have been undertaken to ensure equitable access to healthy food and eradicating food deserts. The website will generate statistics/visualizations across various dimensions (population, representation, race, etc.).

Motivations

The motivation was very clear. Every city/state has an extreme contrast to health outcomes per the zip code and socioeconomic status. This is visibly apparent driving through different parts of a city/state, and intuition well supported by academic research.
Initial research conducted by the development team provided insight into vast regions of the United States where regular and easy access to what is categorized as healthy food was

not prevalent. “Food deserts” being the term to describe regions where it becomes hard (w.r.t distance and affordability) to gain access to healthy food.
This inspired the genesis of the idea to uncover (if any) trends between food security and political representation as well as what legislative action has been taken. As software engineers, the team decided to leverage their technical skills to provide a spotlight on this issue.

User Stories

Developer Team: PutItInPark

- For each phase, we assign 5 user stories to our development team. They will size and estimate and refine requirements with our input. Following that, the user stories must be completed by the deadline of the phase.

Phase 1

The PutItInPark team was provided 5 user stories for their phase 1 development cycle.

1. It would be useful to see the description of the project along with the aim. Potentially, the details of each model and how they correlate. Additionally, a cool background (not necessary) might be a nice-to-have. Overall, the website can be static for now - it will need to be dynamic in the future phases.

2. A profile picture for each group member. Along with that, a total summary of the entire repository commits + issues would be cool to see. It would not be necessary to make the page look "pretty" for now - just a simple picture + commits & issues.

3. It would be preferred if the model pages had a grid or table with links to the instance pages. Each instance listed on the model page should have 5 attributes. We must be able to click on a link/somewhere to navigate to the instance page. To be clear, if the model does not lend itself to a variety of media (for eg: outdoor activities) - then a table is probably more ideal. However, a model like National Parks might have lots of pictures/videos specific to each park, and hence a good candidate for a grid. Regardless, a table is fine for now too.

4. There should be a link to the API documentation - designed by Postman. It would be ideal to have 'GETs' for models + instances.
    a. return a list of models
    b. return the details of the instances (attributes)
    c. return a detailed list of models with their 5 basic attributes. No PUTs/POSTs.

5. Provide 3 instance pages for each of the 3 models. Each instance page must display the original 5 attributes from the model page + additional data points. Along with that, there must be some sort of media (more than one preferred) on each instance page, that is specific to the instance: links to videos, images, etc. It does not need to be "pretty" or formatted super well.

Phase 2

Per requirements, our development team (PutItInPark) was provided 5 user stories to complete for phase 2.

6. **Implement Pagination** - It would be very helpful to be able to page the instances grid when clicking on a model, allowing users to see only a portion of the full list of instances for each model.
7. **About page improvement** - It would be nice to include:
    - Tool stack
    - motivations for the website along with the intended audience
    - link to your Gitlab repository

8. **Build and consume RESTful API** - Please expand on the existing schema of the Postman RESTful API you designed in the previous phase. As stated earlier, the API must be configured (at minimum) to allow user to request:
    a. List of instances of a model with their 5 basic attributes
    b. List of instances of a model
    c. Detailed attributes of an instance Your API must pull the data from YOUR database hosted on AWS or GCP. Ensure your website is pulling the data from the API dynamically, and no data presented on the site for a model/instance is static.
9. **Highlight table row for model pages** - When the cursor hovers over an instance in the model page highlight the table row. Also, make the entire table row clickable instead of just the name of the instance. Implement this functionality for all instances of every model page.

10. **Edit Model Pages** - It would look nice if the instance pages were presented in a format other than a columned list. Maybe some more color or a change in layout would look good, and could even help with loading the data dynamically to have set containers. A few changes could make these pages really pop, and would make the website that much more appealing.

Phase 3

Per requirements, our development team (PutItInPark) was provided 5 user stories to complete for phase 3.

11. **User Story 1: Search Feature** For this phase, we need searching across the website, as well as per model. Requirements:
    a. Ability to search from the splash page, should search the entire website
    b. Each model page must have the ability to search within it
    c. The returned search results must highlight the terms searched for
    d. Searchability must be google like (handle many terms) (built-in modules should make this easier for you)

12. **Tools on about me page** - We would like to see what tools were used on the about me page. A picture of each tool and a description of how each tool was used to build the website. The tools have to be shown in a grid-like format and special focus on the tools that were not required.

13. **User Story 3: Sort Feature** - You should be able to sort the parks by data they present. You can break this by multiple parameters including price, name, activities they offer, etc. This feature should be on each section of the website and should be done on the same page just reordering the presentation of data.

14. **Rework model to model connections.** - As the website stands right now, the connections between one model to the next lead to a 404 error, this should be fixed and reimplemented to allow for cross-connections between one model page to one or more model pages of different types.

    a. Fix cross implementation
    b. Make sure cross instances photos load correctly

15. **More instances per model** - Currently, all the models have duplicate instances. It would be nice to have more instances without duplicates. Once the backend is fully deployed, we should be able to see various instances for all models, and multiple pages with pagination between them.

Customer Team: [After Work]

As we provided 5 user stories to our dev team, our customers did the same for us. We sized, groomed, estimated and implemented them. The following details each user story, their implementation, and other relevant details.

Phase 1

We were provided 5 user stories for our 1st phase of development. Two of them were implemented and closed, while the remaining 3 were deemed out of the scope of the push and we communicated as such.

1. Have the home page explain the name of the website a bit more, since right now it's a bit confusing and it sounds like its a mismatch between the name and content.
    a. Estimated Weight: 1.
   Implementation: Implemented such that the splash page explains, in brief, the motivation of the site.

2. Switch the layout of legislation to a table view since it doesn't really make sense to have a picture with every new legislation.
    a. Estimated Weight: 2.
   Implementation: Changed from grid to lists

3. Add a petitions page where we can see what current petitions are going on about certain legislation, maybe requires another API.
   Implementation: *This was out of the scope for this phase*

4. I would like to see information about obesity in these districts.
   Implementation: *This was out of the scope for this phase*

5. Can I see the cost of living, income inequality, and homelessness stats on the site?
   Implementation: *This was out of the scope for this phase*

Phase 2

We were again provided 5 user stories for our 2nd sprint. Three of them were implemented and closed, while the remaining 2 were deemed out of the scope of the push and we communicated as such.

6. I would like to see a feature that helps me find a food bank in a selected region. This is relevant for users because it informs them on how they can impact food security in the districts that are shown.
   Implementation: *We deemed this to be out of scope for the purpose of the website, and have communicated this idea to our customers.*

7. I would like to see race demographics by district. This will help us see how different levels of food security affect different people. This might be good to place in the section you see when you click on a certain district, it fits well with the information presented there such as average age and gender ratio.
   a. Estimated Weight: 2
   b. Estimated Time: 2 Hours
   c. Actual Time: ~2 Hours

   Implementation: This task had 3 components. First, we identified the fields in the API dataset being requested. The python script that scraped the API was updated to bring in the fields with race details, and then loaded into the staging schema of our database and subsequently moved into the application schema. Now, when our front-end sent in calls for a district instance, the race demographics data is returned as well.

8. Currently, I can only see 3 districts. I think more districts should be added. I expect this will happen in the future anyway but I would like to see this information as a customer.
   a. Estimated Weight: 3
   b. Estimated Time: 2 Hours
   c. Actual Time: 2 Hours (Additional 6 hours including API work to support this)

   Implementation: This task required a lot more background work to be implemented. Our scraper had to load the data into the database, after which our API had to be implemented such that the appropriate data be returned for each district.

Updated all model pages to include many instances of each model, this was an overall quick task, though designing and implementing the API to support this took more time than tracked for this issue alone

9. On the legislation page, every column in a single row in the table links to the same page. However, I would expect clicking on a representative's name to take me to that representative's page instead of the link to a specific act or bill. For example, in the legislation row with the name Agriculture Improvement Act of 2018, clicking on "K. Michael Conaway" does not take me to https://foodmeonce.me/Representatives/instance/K./Conaway
    a. Estimated Weight: 1
    b. Estimated Time 1 Hour
    c. Actual Time 2 Hours

    Implementation: To address the confusion of clicking on the representative's name, and going to the instance page for a district, we reworked the UI to instead highlight the entire selection of the instance, rather than individual components. We believe that through this change it is no longer implied that a user is able to click on the representative's name, but rather click a link to learn more about the instance for the district they have clicked on. We experimented with having separate links on the page, and found this solution was more User-Friendly.

10. A nice feature to have on the Districts page would be another "Filter By" feature which filters by state. Even better would be a clickable map which will take you to the district page but this might be difficult to implement. Some features like these would help users see specific regions they are looking for.
    Implementation: *This was deemed to be out of scope for this phase of the project and will be pushed to phase 3, which has been communicated with the customer team.*

Phase 3

As expected, we were provided 5 user stories for our 3rd sprint as well. All were implemented, with an overlap on two of the stories.

11. The Districts, Representatives, and Legislation page show sorting and filtering functions but do not have a search bar. A search bar can be added in the blank space to the right of "filter by" to search the specific page. This may not have to be done if a site-wide search will allow searching specific categories.
    a. Estimated Weight: 1

b. Estimated Time: 10m
c. Actual Time: 10m

Implementation: We resolved this issue by implementing separate search bars, one attached to the navigation bar to search site-wide, and one that is only on the individual model pages to search each specific type of model.

12. Add a search bar on the top grey bar which has the logo on the far left, as well as the site name "FoodMeOnce". I think this is a good place for a search bar so it can stay in place as the user moves between pages.
    a. Estimated Weight: 1
    b. Estimated Time: 10m
    c. Actual Time: 10m

    Implementation: We resolved this issue by implementing separate search bars, one attached to the navigation bar to search site-wide, and one that is only on the individual model pages to search each specific type of model.

13. A nice feature to have on the Districts page would be another "Filter By" feature which filters by state. Some features like these would help users see specific regions they are looking for.
    a. Estimated Weight: 3
    b. Estimated Time: 3 Hours
    c. Actual Time: 2 Hours

    Implementation: This feature was implemented as requested through the creation of a dropdown menu, which will allow the user to select the state they want to filter the results by. This allows a quick convenient way to filter results down using the state of the district you are looking for. The drop-down was also added to the representative model page.

14. Some sections on certain legislations are blank. For example on https://foodmeonce.me/Legislations/Instance/59 the Sponsor District is blank. On this page https://foodmeonce.me/Legislations/Instance/77, the summary is blank. Maybe add N/A or unavailable as the text?
    a. Estimated Weight: 1
    b. Estimated Time 1.5 Hour
    c. Actual Time 1.5 Hours

    Implementation: To address this issue we have added senators to the list of representatives rendered on the site. In doing so, there is no longer the potential for

a legislation to not have an associated sponsor, or sponsor district, and the information of the site is more complete.

15. On the district's page, does is gender ratio males to females or females to males? I was not sure and had to look it up so it might be good to specify which one it is.
    a. Estimated Weight: 1
    b. Estimated Time 15 Minutes
    c. Actual Time 15 Minutes

    Implementation: This was a simple problem to address, there is now a description of what the gender ratio represents added as a footnote to the sorting component on the district model page.

RESTful API

Phase 1

For phase 1, we have simply designed what calls to our API might look like in the future, although the website currently operates without a RESTful API. We used Postman to design and document the API calls. The process was fairly intricate, as most of us were attempting to understand implement this for the first time.
In future phases, we will scrape data from at least three distinct databases to populate a cloud instance of Postgres DB of our own (on AWS). The API's we currently plan to scrape from can be viewed at the bottom of this report.

Phase 2

The RESTful API was implemented via a basic Flask application with sqlalchemy utilized to grab the required data from the database. All the expected 6 types of calls that the website would send were handled and tested. We made the decision to use classic SQL to communicate with the database for each call (sqlalchemy engine).
The application source code was deployed on an AWS Elastic Beanstalk instance. Following that, we had the elastic beanstalk instance connected to route 53 to provide the API subdomain as well as to ensure a secure HTTPS connection. With this accomplished, the backend server created with Flask communicates with the cloud PostgreSQL database, deployed on AWS RDS. Any request from the website/elsewhere is routed through this architecture.

The API supports 6 API calls.

- GET /Districts
- GET /Representatives
- GET /Legislations
- GET /Districts/<id>
- GET /Representatives/<id>
- GET /Legislations/<id>
- An example request would be of the form: https://api.foodmeonce.me/Districts

All the API calls without the path parameter <id> returns a JSON file with the corresponding data of the model (list of ALL instances, with their 5 pertinent data items). The remaining 3 API calls with the path parameter <id> returns the corresponding data about the specific instance with that ID of a model.

Phase 3

The existing API was expanded upon to handle sorting, filtering and searching. The team made the decision to implement all the aforementioned 3 features on the back-end as opposed to the front-end as bringing back the requisite results was easier to do via SQL. The new supported calls are:

- GET /Districts/filter?<attribute name>='entered value'
- GET /Representatives/filter?<attribute name>='entered value'
- GET /Legislations/filter?<attribute name>='entered value'
- GET /Districts/sort?attribute=<attribute name>&order=ASC/DESC
- GET /Representatives/sort?attribute=<attribute name>&order=ASC/DESC
- GET /Legislations/sort?attribute=<attribute name>&order=ASC/DESC
- GET/search?attribute='search terms'

Sending requests to the API in the above format will return JSON responses. Where applicable, user-inputted filters/searches are case insensitive.
For sorting, the backend accepts the attribute name as well as the selected order, generates the SQL based on those parameters and then sends a request to the database. The response is parsed and pushed to the front-end using the existing framework.
With regards to filtering, the backend accepts the attribute name(s) as well as the user-selected/inputted values for the attribute(s). This functionality is not limited to one filter at a time, and can handle ALL filters in one request. Where applicable, the minimum

and maximum value for a filter are not required for the values to be filtered, defaults are set when necessary.
Searching (model specific and site-wide) is handled similarly, except in this case the user input is matched against ***each*** column value to pull back all relevant results. The searching can handle multiple terms and each term is checked against all columns. The search results are not ranked or sorted for relevancy yet, as it was not required for this phase - but is a feature we will add in future sprints.

Models

Phase 1

For this phase of the project, we designed three model pages. A model, in this case, is the highest level abstraction of our data sources - District, Representative, Legislation.
Each model page has a table with three instances and 5 pertinent data items. We implemented a dummy filter and search feature for future phases, which does not yet work. Each instance page (linked from their respective model page) has the original 5 attributes, along with further detail and media (social media, Wikipedia, maps, pictures). They also link to the other 2 models (specifically instances of the other 2 models).
A neat feature we implemented on the district instance page is a dynamic map based on the district selected. We decided to make our website dynamic for this phase, which was not required. However, we determined that this would make our development in the future phases much smoother. This meant utilizing the React framework instead of creating static web pages using plain HTML and CSS. Our instance pages dynamically pass the values of the selection from the model page and pull back the relevant data items and media.

Description

1. District: Each district represents a single US congressional district. There are 435 congressional districts, all represented on our dynamic website. The 5 attributes visible on the model page:
    - ➢ average age
    - ➢ median income
    - ➢ population
    - ➢ gender ratio
    - ➢ House Representative

2. Representative - Each representative is a member of the House. Therefore, we will have an equal number of districts and house representatives. The model page represents information around one member of the US House of Representatives. The 5 included attributes on the model page are:
    - ➢ age
    - ➢ political party
    - ➢ years in office
    - ➢ state
    - ➢ District
    - ➢ Type

3. Legislation - Each legislation represents a bill introduced to the House, sponsored by a representative. The bills being displayed here are related to food security only. The 6 included attributes on the model page are:
    - ➢ introduced
    - ➢ status
    - ➢ enacted
    - ➢ party
    - ➢ bill type
    - ➢ sponsor

The relation between each instance is as follows:

- A District is linked to the elected Representative, as well as Legislation that the Representative voted/sponsored on
- A Representative is linked to the District they were elected in, as well as Legislation(s) they sponsored/voted on
- A Legislation is sponsored by a Representative(s), and linked to the impacted District(s).

Note: Not all representatives are attached to legislation. This is critical to our issue of food security by representation and location. Similarly, not all districts have legislations affecting them. This is not an error.

Phase 2

For this phase of the project, we made all of our model pages dynamic. Having a dynamic model page means we have many instances in our table. In order to make it look better, we implemented pagination. Each page has eight instances and the pagination bar is located on the bottom of the page. Users can go to the next page or skip ahead to the last page.

Conversely, users can go to the previous page or the first page. They can also select a specific page they want to visit. The specific page can be the previous two pages or the next two pages. The website sends a GET request for each page, and the API returns the necessary 8 rows of data for the specific page number the user is on.

Phase 3

For Phase 3 of the project, most of the models had predominantly visual changes. We have switched the display of both districts and representatives to cards from plain tables to add more visual appeal when browsing on their respective model pages. Each district now has a map on the model page. This loads fast/not so fast based on the internet connection, and is up for revision during the next phase. The representative cards use the representative image. Secondly, Representative images on the website, such as on the Representative model page, or when displayed as the representative for a district/legislation instance, we now use a fallback image. It defaults to an image of the representative's political party if we are unable to obtain an image from our source. Next, we used each model's visual styling to create a new global search page, where the user is able to view results from searching across the entire site.
The last change was the primary logical change to the codebase - we brought in senators into the website in this phase. Previously, we had limited our website to present just the House Representatives. This involved:

- changing the API calls to allow Senators to be passed through from the database
- Senator instances now link the District attribute to the District model page filtered by their state
- Legislation instances sponsored by a Senator, also link to the District model page filtered by their state for the District attribute

Phase 2 Features:

Pagination

The website now offers pagination for each model. Each page has been limited to 8 instances.

Back-end implementation: The API call for the model pages is specific to the page the user is on. For example, if the user is on page 3, the API will take this information and ensure it returns 8 rows for the specific page. This is implemented by having an 'ORDER BY' clause along with 'LIMIT 8 OFFSET 8*pageNumber' in the SQL command sent to the database. This ensures consistency in the results, along with allowing us to implement pagination easily. After this, it was merely a case of architecting the front-end.

Front-end implementation: We designed a file 'pages.js' that received the metadata about which page the user is on, the previous page number (if > 1). With this, it generated and returned the pagination components (metadata and GUI button data) to the index.js file for the appropriate model page. The model page then used that with the actual data embedded within

Database

We a cloud-based solution for the database. PostgreSQL was determined as the best choice. The database is deployed on an AWS RDS instance. We used two schemas for designing the database: staging schema for the dev environment, and application schema for the production environment. All the data scraped from the RESTful API sources are uploaded to the tables in staging schema. Then we migrated only the data we needed to the application schema.
Application schema has 3 tables: Districts, Members, and Legislations. Then when we call SQL commands, we use join commands to join the columns of three tables to provide the required columns for each API call.
The relations between the 3 tables in the application schema are as follows:

A district has ONE representative

- join on **district.state = representative.state** AND **district.congressional_district = representative.district**

A district MAY have ZERO or MORE legislations passed. A district is linked to legislation sponsored by its House Representative OR legislation passed by the Senator of the corresponding state

- Legislation by House Rep:
    - join on **legislation.sponsor_name = representative.full_name** AND **representative.district = district.congressional_district** AND **representative.state = district.state**
- Legislation by Senator:
    - join on **legislation.sponsor_name = representative.full_name** AND **representative.state = district.state**
- Both results are displayed as legislation for a district (may be null or more)

A representative has ZERO or ONE district based on whether the representative is a Senator or House Rep.

- join on **representative.district = district.congressional_district** AND **representative.state = district.state**

A representative has ZERO or MORE legislations sponsored

- join on **legislation.sponsor_name = representative.full_name**

A legislation has ZERO or ONE district based on the type of representative who sponsored the bill

- Legislation by House Rep:
    - join on **legislation.sponsor_name = representative.full_name** AND **representative.district = district.congressional_district** AND **representative.state = district.state**
- Legislation by Senator:
    - join on **legislation.sponsor_name = representative.full_name** AND **representative.state = district.state**

A legislation has ONE representative

- join on **legislation.sponsor_name = representative.full_name**

Phase 3 Features:

Searching

The website now features searching both locally within a model page, and site-wide from the home page. Each model page supports searching such that when the user presses enter on the search bar with any input, the frontend requests an API call to the backend with the search string given as a query string. Then the backend server parses the query string and splits the search string by spaces so that it can support multiple terms. Then using WHERE <column> like '%<search string> %' for every split search string and for every column in the table of the model, the server returns the relevant data. Thus, the majority of work is done in the backend. Then using these search features in each model, the site-wide search is supported by calling API 3 times to get back any relevant data. The user is prompted to enter a value for the site-wide search if the text-box is empty and the Search button is clicked.

Sorting

Now users can sort the model page results based on that model pages instance variables. When the user selects which attribute and the order to sort by, the frontend calls an API call to the backend with attribute=<name of column>&order=<ASC or DESC> querystring, which the backend server parses and calls the appropriate SQL query to sort the data as requested. Then the returned data is displayed on the model page.

Filtering

Now users can filter the model page results based on that model pages instance variables. In each of the model pages, there are filter bars that the user can put in values to filter for. Then when the user clicks on apply, the frontend collects all the attribute filtering values and sends an API request to the backend server with the collection of attributes as a query string. Then the backend server parses the query string and calls the SQL command using WHERE <name of column> between <min value> and <max value> to gather relevant data and return to the user.

Database

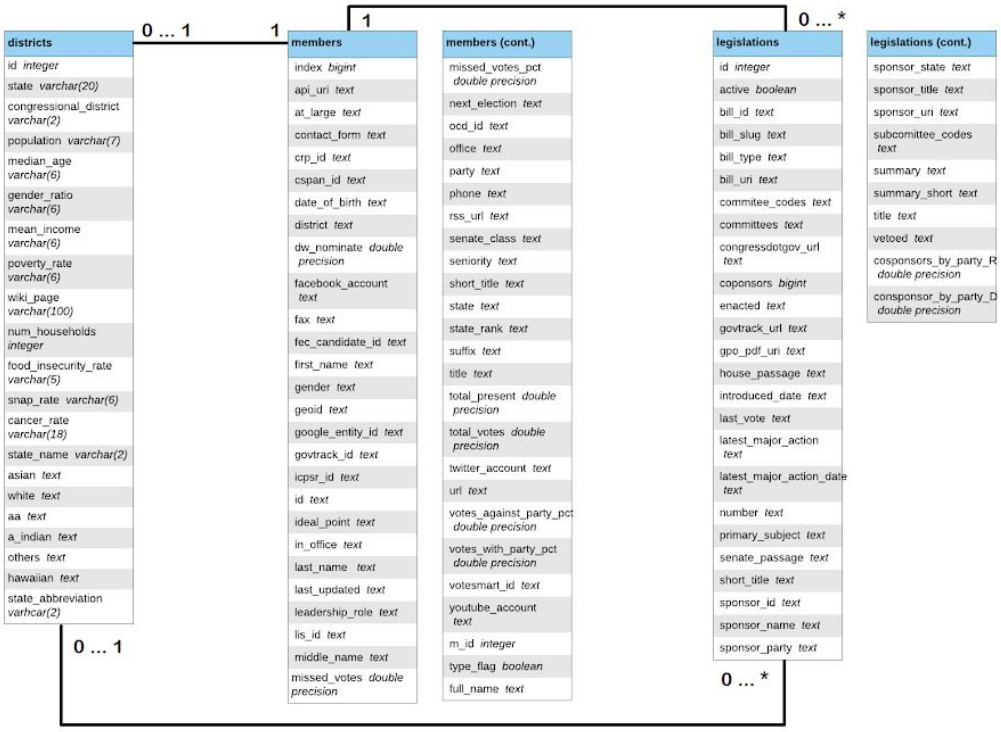
No change to the database this phase. We made the choice of not having any join tables and using our existing framework from phase 2 (defined above). We joined the requisite columns from the 3 tables to bring back all necessary data for a model/instance. It was discussed within the team as well as our TA's that this was sufficient, as long as we provided the columns to join on. We did not find any specific advantage to using a join table with IDs within it. We did implement it and later removed it, as we had what we needed without it. The schema is not complex enough to justify additional tables.

The updated schema can be found below, as well as in the repository separately.

FoodMeOnce Database ERD (UML)

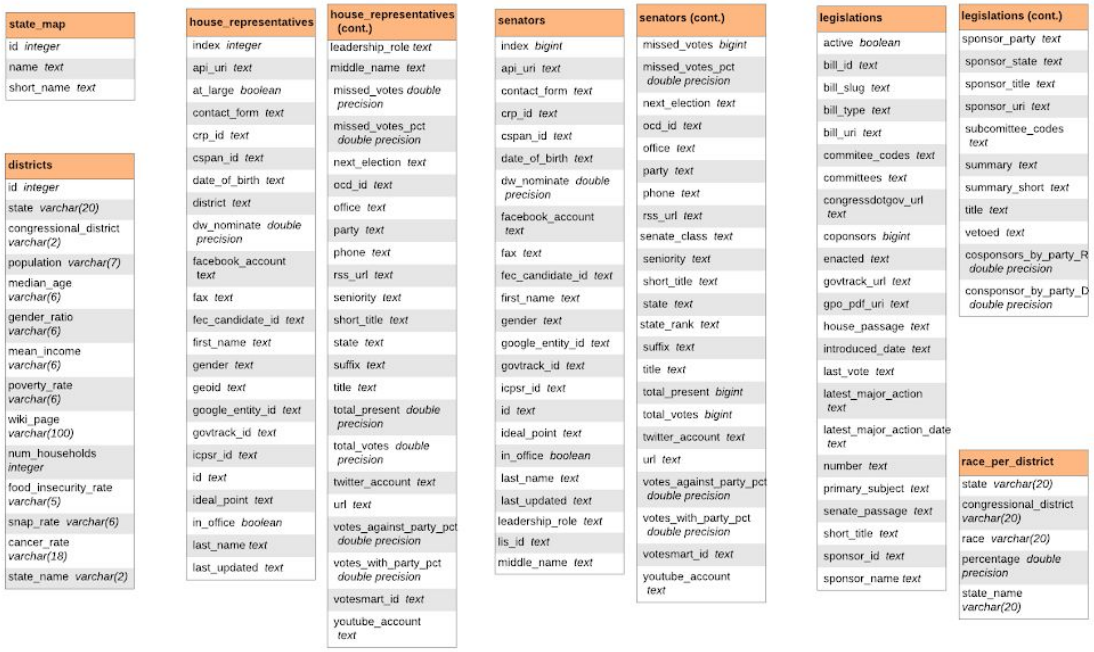
Application Schema:

- Currently each table has a primary key identity column, however there exists no hardcoded relation between the tables.
- The RESTful API requests data from tables within this schema. The table relations depicted here are NOT hardcoded, but referenced for ease of of understanding.
- The API joins on the columns listed on the tech report to pull back the requisite data



Staging Schema:

- The python scripts dumped all the various data sources in different tables into this schema
- From here, the application tables were made, joining data from the tables present in this schema



Testing

Phase 2

We attempted our best to follow test-driven development practices and wrote test cases before or concurrently to our development.
Any and every piece of the application had to be tested. We utilized the GitLab CI/CD features to ensure continuous testing of our changes.
There are 4 main pieces of testing done for this website:

1. Unit testing for backend API server using Postman
2. Unit testing for database and API scrapers using Python (*included in GitLab CI*)
3. Mocha unit tests for javascript code (*included in GitLab CI*)
4. Selenium GUI testing *(included in GitLab CI*)

1) Backend API server - *Postman.json*
   We created a file called postman.json that ran several tests with the API. We ensured each of our API calls (6 calls) was tested thoroughly, checked the returned rows of data for validity, as well as tested our pagination features.

2) Database loads and API scrapers - *backend/tests.py*
   We wrote tests to check that we are successfully connecting to the database and getting the correct API response. Along with that, tests check if we are loading in the correct amount of data from all the APIs we are using. We checked the data numbers for states, districts, representatives, and legislation. We check the methods that we wrote that processes the data and returns a dictionary or dataframe.

3) Selenium Tests - *frontend/guitests.py*
   We tested the GUI of our website using Selenium Webdriver written in a python script titled guitest.py . The GUI components we tested are as follows
   - The Navigation bar, and all of its links
   - Each Model Page, and the links to every instance
   - Pagination of each model page, forwards and backwards
   - The About us Page, and all of the links accessible from it.

   These tests check that each GUI component reacts as expected, checking that the components load correctly, and that they work as intended after having loaded. Through automation, we are able to test each clickable piece of the website across our multi-page model pages and website.

4) Mocha Tests - *frontend/tests.js*
   We tested the react components using a module called Mocha. The react components tested are the following:
   - Each model components: Districts, Representatives, Legislations
   - Each instance components: DistrictInstance, RepresentativeInstance, LegislationInstance.

   In the tests, we tested for each of the components at first render has default, empty state. Then after the pulling from the API, each component has the state variable that's filled with the data from the API. To test for the components correctly pull the data, we used wait until module to wait until the state of the component changes.

Phase 3

1) Backend API server - *Postman.json*
   The existing tests were expanded upon to thoroughly test the new sorting, filtering and searching calls. We were able to ensure we knew how to implement the API based on the predetermined test cases.

2) Database loads and API scrapers - *backend/tests.py*
   No change to the codebase, resulting in no new test cases.

3) Selenium Tests - *frontend/guitests.py*
   To extend testing for phase 3 of the project, we switched to using selenium IDE as it enabled much easier access to complicated components such as dropdowns and toggles for filtering settings. These tests can be found now in a new file labeled guitest.side. The GUI components we extending testing for this phase include
   - The Global Search bar, with several test searches
   - The Local search bar for Districts, Legislation, and Representatives
   - The filter components for all three models
   - The various methods of sorting data for all three models
   - The new cards that we transitioned the data to use rather than a grid

   These tests check that these various methods of searching, filtering, and sorting all work as intended after the call to the API is received. Through this automation, we can ensure that sorting and searching always yields the expected results for a variety of criteria in a short concise manner.

4) Mocha Tests - *frontend/tests.js*
   We extended the react component tests with searching, sorting and filtering. The react components tested are for the following:
   - Each model component with searching: Districts, Representatives, and Legislations
   - Each model component with sorting: Districts, Representatives, and Legislations
   - Each model component with filtering: Districts, Representatives, and Legislations

   In the tests, we tested for each of the components at first render has default, empty state. In the pathname, we specified if it was search, sort or filter and chose an attribute for each of the three cases. Then after the pulling from the API, each component has the state variable that's filled with the data from the API. We again used the wait-until module to wait for the state of the component to change. Overall, we have a total of 30 mocha tests.

GitLab CI

A not insignificant requirement of our testing was the GitLab CI/CD feature. Our configurations for each of the 3 components we were testing can be found in the *.gitlab-ci.yml* file. We have 3 jobs to be run in the pipelines:

1. Front-end_tests (*image: node:12.2.0-alpine*)
2. DB_tests (*image: continuumo/miniconda3:latest*)
3. API_tests (*image: postman/newman_alpine33*)

Each of these 3 phases, runs the test cases mentioned above. We had to ensure we set up the virtual environments correctly and had all the dependencies and version listed our clearly.

Tools

Front-end framework:

- ❖ React Javascript - Used to create the dynamic components of the website. By using React we were able to have the website react and change based on user interaction, rather than relying on static pages.
- ❖ Bootstrap and CSS - Using Bootstrap as a framework for CSS we were able to add detailed styling and UI work to the website, making it friendly for visitors to the site.

- Selenium - Used to test the GUI of the website, through automation we were able to test that changes to the codebase didn't affect the usability of the website on the frontend. These test cases can be seen in the guitest.py file in the repository.
- Mocha - is used to test the frontend javascript code. We test all of our model pages for the correct URL, no data at first render, and data loading correctly from API by checking the table length. For the instance pages, we check the URL for correct id, no data at first render and data loading correctly from API.

Backend:

- Amazon S3 - Website hosted on an AWS S3 bucket.

Domain:

- NameCheap - We acquired the pretty URL from Namecheap, allowing users to access the site through a user-friendly domain.
- Route53 - We acquired an "API" subdomain for API backend

Back-End Tools:

- PostgreSQL - Cloud database hosted on AWS RDS used to store all data scraped from public APIs. Multiple schemas utilized to allow for more streamlined production schema.
- POSTMAN - Used to document and test our backend RESTful service
- SQLAlchemy - Used to connect to and speak with the database. Utilized sqlalchemy engine to load data from APIs as well as request data from database for website/external use
- Flask - Utilized to create backend web application to facilitate API calls to database
- Python - Multiple scripts developed to load data into PostgreSQL DB, and used to implement API calls

IDEs:

- Pycharm IDE
- VSCode

Others

- LucidChart - Design of DB schema
- Docker

Hosting

Frontend

Food Me Once (https://foodmeonce.me) is a website hosted on an AWS S3 bucket. All of the files in the S3 bucket are generated by invoking the "npm run build" command on the command line, which is then pushed via "npm run deploy". We had to configure IAM profiles, to enable automatic deployment to the bucket without having to manually upload the files.
The AWS S3 instance is then connected to Cloudfront to provision the secure https access. A brief issue we encountered was that the S3 instance, continued to serve up a cached (and hence outdated) version of our site, despite newer builds being deployed. We had to edit the Cloudfront settings, to update each time we pushed.
Lastly, we configured Route 53 for our domain name which we got using Namecheap. This allowed for our website to be accessible at foodmeonce.me.

Backend

Food Me Once pulls the data from the API backend, which is created using a Flask python framework. All the source codes for the backend is deployed on an AWS Elastic Beanstalk instance in a zip file. Subsequently, we connect the server to the "API" subdomain of our website. The routing is done by redirecting all the requests to "https://api.foodmeonce.me" to the server running on the AWS Elastic Beanstalk using Route 53.

Data Sources

Our first major data source contains detailed statistics on congressional districts. From details on food deserts to race demographics, are captured here. Secondly, we will scrape data to obtain information about all the political leaders in both the House of Representatives and Senate. Lastly, we will scrape legislation data regarding food, health and food security acts/legislation sponsored in the United States government. We believe that by combining these three distinct sources we can use this information to create a website that allows users to truly make connections between food security and political representation and actions. Links to these data sources can be found below.

Relevant Links

- FoodMeOnce
- FoodMeOnce GitLab Repository
- FoodMeOnce Postman documentation
- FoodMeOnce API

➔ Data Sources (scraped programmatically via RESTful API calls)
  - District Data API
  - Legislation Data API
  - Representative Data API

The FoodMeOnce team can be contacted at FoodMeOnce2019@gmail.com