CS 373 - SOFTWARE ENGINEERING

FIGHTPOVERTY.online
RANKING CHARITIES TO FIGHT POVERTY

By: Justin Berman, Chris Amini, Charlie Matar, Yijie Tang, Kevin Singh.

Introduction

Fight Poverty is a web application that aggregates and ranks charities from all around the United States whose main purpose is to aid in the fight against poverty and provide helpful tools and resources to help those in need.

As of the most recent 2016 Census[1], there are about 40.6 million people in America living in poverty situations, or roughly about 12.7%. We still have a long way to go to improve the living conditions in this country, and this is our way of contributing.

Motivation

This project started as a way for us to provide a useful service to our community, using our software engineering and web applications skills. Upon browsing several websites, we realized how tedious it could be for someone to find a trustworthy and reliable charity, based on their interests and close to their community. That's when we became aware we could simplify the process by parsing and accumulating all these different charities into a single website, implementing a system that ranks them in order of their reliability and orders them by city and county. This would therefore simplify and make it easier for people to find a charity they trust and is close to their community.

User Stories

1. As a user, I want to be able to identify charities by the type of cause it supports.
    a. If you navigate to the Charities tab of our website, you will be able to click on any of the displayed charities and more information will pop up on the specified charity, including its cause.
    b. On the backend, we plan to have a RESTful API that will return charities given a cause. The expected format of the request and response can be seen here under the GET request titled "Charities by cause."
    c. Discussion: we have a very limited set of causes since we are focused on charities specifically aiming to fight poverty. The user might be disappointed there are very few different types of charities in our database.
    d. Estimated completion time: 6 hours after RESTful API set up (which should take 8 hours)
2. As a user, I want to be able to compare two different charities between cities and counties.
    a. Our website provides two different tabs, one for cities, another for counties, where you can access and compare the different charities in each region.
    b. On the backend, we plan to have RESTful API's that return charities in a city or county when searching for either. The expected format of the requests and responses can be seen here in all GET requests in the Cities and Counties folders. Those responses would return charity IDs which can then

be used in the GET request titled “Charity by id” to get the data of the charities the user wants to compare.

   c. Discussion: a better user experience would be to have an input form/checkboxes to compare charities in different regions, rather than forcing the user to jump around the website to find the charities the user wants to compare. Need to look into a way to do this.
   d. Estimated completion time: 20 hours after RESTful API set up
3. As a donator, I want to know more information about how my money is being spent by the charity I choose to support.
   a. Any charity you select through our website provides you with all of the information on the selected charity, including what they do, how your donations help them, what their financial rating is based on how well they spend money, how they provide their service and more.
   b. On the backend, we plan to have a RESTful API that will return detailed information about a charity. All expected data can be seen in the GET requests in the Charities folder at this link.
   c. Discussion: a user might not be satisfied with the data we have, though we can easily provide links to sources that might give more information.
   d. Estimated completion time: 6 hours after RESTful API set up
4. As a user, I want to be able to find charities closest to me.
   a. Our charities are organized by location. Furthermore, clicking on any charity will give you its full address and zip code.
   b. On the backend, we plan to have a RESTful API that will return charities given a city/state, county/state, or zip code. The expected format of the request and response can be seen at this link in the GET request titled “Charities by location.”
   c. Discussion: a better user experience would be to have a search box where the user can provide their location to find charities, rather than forcing the user to jump around the website to find the charities closest to the user.
   d. Estimated completion time: 12 hours after RESTful API set up
5. As a user, I want to be able to click on a charity from the list and get information about its rank, city, county, what they do to fight poverty, and other statistics.
   a. All of this is available by default on each charity published on our website.

b. On the backend, we plan to have a RESTful API that will return detailed information about a charity. All expected data can be seen in the GET requests in the Charities folder at this link.
c. Estimated completion time: 4 hours after RESTful API set up

6. As a user, I would like to see more cities, charities, and counties.

a. With more cities and counties on our website, our users will be able to see even more charities they can contribute to. This, in turn, will help everyone from around the United States fight poverty.
b. Our database now contains hundreds of cities, charities, and counties.
c. Estimated completion time: 5 hours

7. As a user, I would like to see more information about what the website does on the homepage.

a. When users visit the homepage they will want to see what the site offers. We can do a better job of using our home page to easily get our site's message across.
b. The homepage of our website now contains information about what our website has to offer.
c. Estimated completion time: 45 minutes

8. As a user, I would like to see the sources of Accountability Rating and Financial Rating at the bottom of the page.

a. Users want to see what makes up the Accountability and Financial ratings. What sources did we use to calculate these ratings?
b. Our scores are now embedded with a link to CharityNavigator's site where they explain how they rate charities.
c. Estimated completion time: 30 minutes

9. As a user, I would like to see the FightPoverty's rating populated.

a. Our charities pages have, "FightPoverty's Rating: Algorithm in development" as of right now. Users would like an actual statistic put in place of that.

b. The FightPoverty's rating is now populated. We used the same algorithm as CharityNavigator to decide the rating.
c. Estimated completion time: 1 hour

10. As a user, I would like to see the FightPoverty's Rank.

a. Our charities pages have, "FightPoverty's Rank: Algorithm in development" as of right now. Users would like an actual statistic put in place of that.
b. We made some adjustments and now we show users CharityNavigator's Accountability Score and Financial Score as well as FightPoverty's rating.
c. Estimated completion time: 1 hour

RESTful API

Documentation to our RESTful API is published via Postman here. Requests are formatted as detailed in the documentation, and responses come as JSON objects as seen in the examples in the sidebar on the right.

We used Postman to document the API's (see information on Postman in the Tools section). In order for anyone to pick up with continuing work on the RESTful API in Postman (without revealing our username and password), simply find the latest Postman collection from the GitLab repo here, and import that file into your Postman app. All of our RESTful API's would then appear inside your Postman and you can test/add/edit/delete API's as needed. Tests are also included in Postman for each API.

Right now responses with multiple objects from the RESTful API come in the following form:

```
{
      "num_results": int,
      "objects": [all elements requested],
      "page": int,
      "total_pages": int
}
```

The "Pagination" section below gives more details on what each element means and how they should be used.

Charity API

When requesting a charity via any of the charity API’s (nothing specified, charity’s unique id specified, charity’s name specified, charity’s zip code specified, charity’s cause specified), each charity object looks like this:

```
{
  "address": "461 Glenbrook Road",
  "cause": "Food Banks, Food Pantries, and Food Distribution",
  "charity_navigator_accountability_score": 100,
  "charity_navigator_financial_score": 85,
  "charity_navigator_score": 89.39,
  "city": {
    "county_id": 17,
    "id": 26,
    "name": "Stamford",
    "state": "Connecticut"
  },
  "city_id": 26,
  "county": {
    "county_poverty_percentage": 8.6,
    "county_poverty_population": 79966,
    "id": 17,
    "name": "Fairfield County",
    "state": "Connecticut"
  },
  "county_id": 17,
  "fight_poverty_score": 89.39,
  "id": 1,
  "mission_statement": "The Food Bank of Lower Fairfield County is lower Fairfield
County's primary hunger-relief organization. We provide food to approximately 75
non-profit agencies and programs that serve low income people in our six town
service area through bags of groceries and congregate meals. These include soup
kitchens, food pantries, child care programs, homeless shelters, senior centers,
domestic violence safe houses, and rehabilitation programs. Our mission statement
is to raise awareness of, and promote action to combat, hunger in these
communities.",
  "name": "The Food Bank of Lower Fairfield County",
  "zip_code": 6906
}
```

All elements in this response are explained in the “Models” section. Notably, the charity will also come with a “city” and “county” object. This reflects the city and county the charity is in.

City API

When requesting a city via any of the city API's (nothing specified, city's unique id specified, city's name specified), each city object looks like this:

```
{
  "charities": [
    {
      "address": "97 North Hatfield Road",
      "cause": "Food Banks, Food Pantries, and Food Distribution",
      "charity_navigator_accountability_score": 100,
      "charity_navigator_financial_score": 97.5,
      "charity_navigator_score": 98.23,
      "city_id": 1,
      "county_id": 1,
      "fight_poverty_score": 98.23,
      "id": 3,
      "mission_statement": "Mission: To feed our neighbors in need and lead the
community to end hunger.<br><br>Vision: A Western Massachusetts where no one goes
hungry and everyone has access to nutritious food.<br><br>Overarching value: We
believe that everyone has the right to healthy food regardless of their
circumstances. <br><br>The Food Bank is the leader provider of emergency food and
other food assistance to more than 223,457 individuals at risk of hunger in the
four counties of Western Massachusetts (Hampden, Hampshire, Franklin and Berkshire
counties). We also lead the community to end hunger through public education and
advocacy.<br><br>For more information on how you can get involved in your
community, visit www.foodbankwma.org<br><br>",
      "name": "The Food Bank of Western Massachusetts",
      "zip_code": 1038
    }
  ],
  "county": {
    "county_poverty_percentage": 11.9,
    "county_poverty_population": 16535,
    "id": 1,
    "name": "Hampshire County",
    "state": "Massachusetts"
  },
  "county_id": 1,
  "id": 1,
  "name": "Hatfield",
  "state": "Massachusetts"
}
```

Again, all elements are explained in the “Models” section below. Notably, the city will also come with an array of “charities” as well as a “county” object. This reflects all charities that are inside the city and the county the city is in.

County API

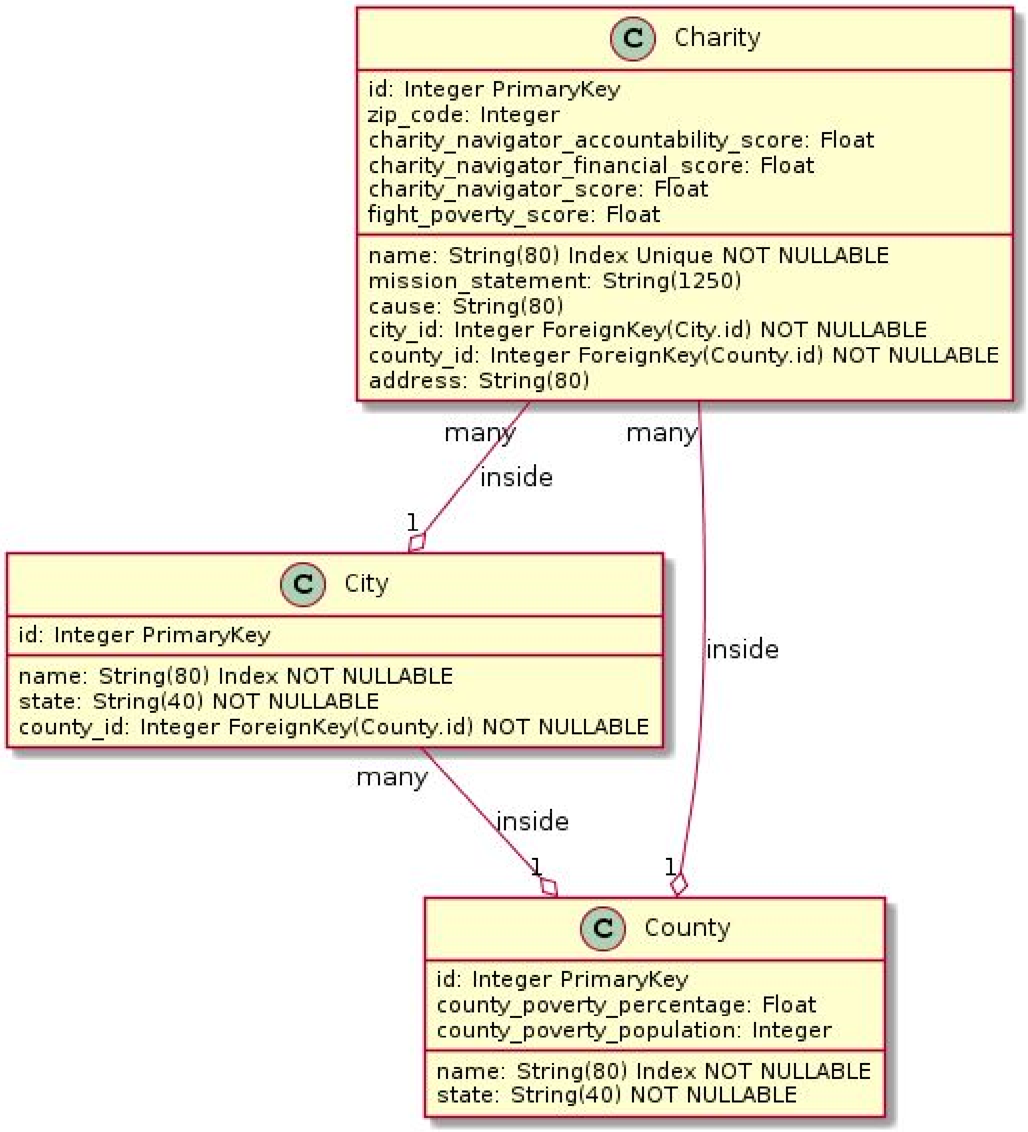
When requesting a county via any of the county API’s (nothing specified, county’s unique id specified, county’s name specified), each county object looks like this:

```
{
  "charities": [
    {
      "address": "97 North Hatfield Road",
      "cause": "Food Banks, Food Pantries, and Food Distribution",
      "charity_navigator_accountability_score": 100,
      "charity_navigator_financial_score": 97.5,
      "charity_navigator_score": 98.23,
      "city_id": 1,
      "county_id": 1,
      "fight_poverty_score": 98.23,
      "id": 3,
      "mission_statement": "Mission: To feed our neighbors in need and lead the
community to end hunger.<br><br>Vision: A Western Massachusetts where no one goes
hungry and everyone has access to nutritious food.<br><br>Overarching value: We
believe that everyone has the right to healthy food regardless of their
circumstances. <br><br>The Food Bank is the leader provider of emergency food and
other food assistance to more than 223,457 individuals at risk of hunger in the
four counties of Western Massachusetts (Hampden, Hampshire, Franklin and Berkshire
counties). We also lead the community to end hunger through public education and
advocacy.<br><br>For more information on how you can get involved in your
community, visit www.foodbankwma.org<br><br>",
      "name": "The Food Bank of Western Massachusetts",
      "zip_code": 1038
    },
    {
      "address": "265 Prospect Street",
      "cause": "Food Banks, Food Pantries, and Food Distribution",
      "charity_navigator_accountability_score": 89,
      "charity_navigator_financial_score": 86.14,
      "charity_navigator_score": 87.48,
      "city_id": 2,
      "county_id": 1,
      "fight_poverty_score": 87.48,
      "id": 4,
      "mission_statement": "The Northampton Survival Center is an emergency food
pantry that provides low-income individuals and families in 18 communities in
Hampshire County with free food, clothing, personal care items, and referrals for
emergency assistance. From its two locations in Northampton and Goshen, the Center
distributes over 650,000 pounds of food each year. From its Hilltown Pantry in
Goshen, the NSC distributes another 40,000 pounds of food each year in the nine
northern hilltowns. All together, from its two locations, the NSC distributes about
2,500 pounds of food in nutritionally-balanced food boxes every weekday.",
      "name": "Northampton Survival Center",
      "zip_code": 1060
    }
  ],
  "cities": [
    {
      "county_id": 1,
      "id": 1,
      "name": "Hatfield",
      "state": "Massachusetts"
    },
    {
      "county_id": 1,
      "id": 2,
      "name": "Northampton",
      "state": "Massachusetts"
    }
  ],
  "county_poverty_percentage": 11.9,
  "county_poverty_population": 16535,
  "id": 1,
  "name": "Hampshire County",
  "state": "Massachusetts"
}
```

Again, each element is explained in the “Models” section next. Notably, each county object also contains a “cities” array as well as a “charities” array. The “cities” array has all cities in that county and the “charities” array has all charities in that county.

Models

UML Diagram


Charity
id: Integer PrimaryKey
zip_code: Integer
charity_navigator_accountability_score: Float
charity_navigator_financial_score: Float
charity_navigator_score: Float
fight_poverty_score: Float
name: String(80) Index Unique NOT NULLABLE
mission_statement: String(1250)
cause: String(80)
city_id: Integer ForeignKey(City.id) NOT NULLABLE
county_id: Integer ForeignKey(County.id) NOT NULLABLE
address: String(80)
many
inside
1
many
inside
City
id: Integer PrimaryKey
name: String(80) Index NOT NULLABLE
state: String(40) NOT NULLABLE
county_id: Integer ForeignKey(County.id) NOT NULLABLE
many
inside
1
1
County
id: Integer PrimaryKey
county_poverty_percentage: Float
county_poverty_population: Integer
name: String(80) Index NOT NULLABLE
state: String(40) NOT NULLABLE

Charity

These are charities from around the US specifically focused on fighting poverty. In order to scrape for charities, we first requested a key from the Charity Navigator website. Then we used this python module to scrape data from the API into a JSON file. See this folder to see which queries were used and the respective JSON files containing the data. Here is the Charity Navigator API we used to actually request data. How we got city and county data will be detailed in the following sections.

Each charity has the following columns:

- *id*: this is an Integer that serves as the primary key to a charity, thus it uniquely identifies each charity.
- *zip_code*: this is the charity's zip code reflected as an integer. This was scraped from Charity Navigator.
- *charity_navigator_accountability_ score*: a float between 1 and 100 reflecting Charity Navigator's score for how well the charity can be held accountable and how transparent it is in its operations. Here is more detailed information on how Charity Navigator goes about coming up with this score.
- *charity_navigator_financial_score*: a float between 1 and 100 reflecting Charity Navigator's score for how well the charity spends money (e.g. on charitable services versus salaries). Here is more detailed information on how Charity Navigator goes about coming up with this score.
- *charity_navigator_score*: a float between 1 and 100 reflecting Charity Navigator's aggregated calculation of both financial and accountability scores. Here is the formula Charity Navigator uses to come up with this score.
- *fight_poverty_score*: a float between 1 and 100 reflecting Fight Poverty's rating of the charity based on a blend of Charity Navigator's scores, as well as the poverty percentage and population of the county each charity operates in. This score has not been calculated yet for charities but we plan on implementing it as follows: there will be a multiplier ranging from 0.75 to 1.25 based on how poor a charity's county is and how large the poverty population is. Thus, the poorer the county and the larger the poverty population, the higher the multiplier. This multiplier will be applied to Charity Navigator's overall score and will be maxed out at 100.
- *name*: the name of the charity. This is a required field for a charity. This data was scraped from Charity Navigator.
- *mission_statement*: the charity's mission statement provided to Charity Navigator.
- *cause:* the charity's cause. Right now we only have two causes ("Food Banks, Food Pantries, and Food Distribution" and "Homeless Services") since these were the causes most directly geared toward fighting poverty we could find charities for. We may make this its own table in the future. This was scraped from Charity Navigator.
- *city_id*: an integer reflecting the unique id of the city the charity is in. This id is an id of another city inside the FightPoverty database, thus it is a foreign key. It is

required to know the id of the city the charity is in when inserting the charity. This allows us to link charities to cities easily. The city name was scraped from Charity Navigator. We used the city name and state name provided by Charity Navigator to then locate the city in our database.
- *county_id*: an integer reflecting the unique id of the county the charity is in. This id is an id of another county inside the FightPoverty database, thus it is a foreign key. It is required to know the id of the county the charity is in when inserting the charity. This allows us to link charities to counties easily. Plus, this will allow us to more easily determine the fight_poverty_score based on the county's poverty stats. We used the county name and state name provided by Charity Navigator t then locate the county in our database.
- *address*: the charity's street address provided by Charity Navigator.

The Charity table has a many-to-one relationship to both City and County tables. The charity's relationship to both city and county is an aggregation because each city and county is composed of an aggregation of charities -- if city and county were deleted, each charity would still exist independently.

City

These are cities around the US which have charities fighting poverty. The data we used has been scraped from both the US census as well as a Zip Code database.

To scrape from the US census, we used the same Python module linked above to scrape the data from the API into JSON files. These folders show exact queries used and JSON files: cities folder, counties folder, states folder, and zip code folder.

To scrape from the Zip Code database, we downloaded the CSV file from that page linked, then used this python module, to convert the downloaded CSV into this JSON file.

Each city has the following columns:

- *id*: this is an Integer that serves as the primary key to a city, thus it uniquely identifies each city.
- *name*: the name of the city. This is a required field.
- *state*: the state the city is in. This is a required field.

- *county_id*: an integer reflecting the unique id of the county the city is in. This id is an id of another county inside the FightPoverty database, thus it is a foreign key. It is required to know the id of the county the city is in when inserting the city. This allows us to link cities to counties easily.

The City table has a many-to-one relationship to the County table. The city's relationship to a county is an aggregation because a county is composed of an aggregation of cities -- if a county were deleted and/or its boundaries relocated, cities would exist independently.

County

To get county information we used the following data from the US Census. We used the same Python module linked above to scrape the data from the API into a JSON file. This counties folder shows the API used and the JSON file.

Each county has the following columns:

- *id*: this is an Integer that serves as the primary key to a county, thus it uniquely identifies each county. It is created by the FightPoverty database when a county is inserted into the database.
- *county_poverty_percentage*: this is a float between 0 and 100 that reflects the percentage of people living in poverty in a county. It is collected from the US census. More information can be found here.
- *county_poverty_population:* this is an integer that reflects the number of people living in poverty in a county. It is also collected from the US census. More information can be found here.
- *name:* the name of this county. This is a required field.
- *state:* the state the county is in. This is a required field.

Tools

Namecheap

We used Namecheap.com to register and manage our domain. Their dashboard makes it simple to manage your domain's properties, redirects, and more. Plus, all of their domains include WhoIsGuard privacy protection free of charge for the duration of your domain.

Amazon Web Services - Elastic Cloud Computing (EC2)

Since the second phase of the project required us to have a dynamic website, this time we needed to have some computing power to run our applications, not just a storage location. That's why we moved all the contents of our website to the Elastic Compute Cloud [EC2] service of AWS. Through proper network and storage management, we managed to fit three Docker containers inside a single t2.micro EC2 instance, one for Flask, another for MySQL, and the last one for React.

Docker

We used Docker to divide and containerize Flask (our backend application), React (our frontend application) and MySQL (our database). By doing this we are able to isolate each application, eliminating app conflicts and simplifying the automation pipeline. Both the Flask and the React Docker containers are set up in such a way that a simple push to the Gitlab 'deployment' branch can update the backend and frontend files, rebuild the containers and restart them. The MySQL Docker container does not need to be restarted, but in the event we wanted to, all data is secure and persistent in the EC2 volume, making sure no data is ever lost accidentally.

Amazon Web Services - Elastic Load Balancing (ELB)

At first, we didn't think we would need a load balancer, but after deploying the React app into a Docker container within EC2 we realized we needed to implement some port forwarding, especially since Namecheap (our DNS provider) would not accept the desired React port as part of the CNAME address. Therefore, we created an application load balancer using AWS Elastic Load Balancing. This load balancer not only allows us to forward the default port 80 to our React's port, but it also helps distribute the incoming traffic of our application from three different AWS regions.

Flask

Flask is a Python framework we are using to expose the FightPoverty RESTful API. Inside the Flask app here, we connect to a MySQL database, set up the tables stored in the database, and expose various endpoints to that database. SQLAlchemy is used for all MySQL-related functions (connecting to the database and constructing the tables) and Flask-Restless combines with SQLAlchemy to create endpoints that query the database. More information on Flask can be found here.

React

React was used to build our dynamic single-page web application. The main benefit of React is that it allowed us to split up the static website into reusable components. These components enable us to create a website that shows displays information about a countless number of charities, counties, and cities, without having to hardcode all the pages of information.

Gitlab

Gitlab is where we hosted our development repository. It's the perfect place for collaboration, issue tracking and most importantly, continuous integration and deployment. It helped us automate our deployment to our EC2 instance and publish our changes for both the frontend and backend with every push

Selenium

Selenium is a suite of tools to automate web browsers across many platforms. Selenium WebDriver was used to create robust, browser-based regression automation suites and tests.

Postman

Postman enables us to have an easy to use tool to test and document our website's RESTful API's. It also provides an easily accessible reference for anyone to consult when attempting to use our RESTful API's via automatically generated documentation linked here.

Bootstrap

Bootstrap is a great front-end framework that allowed us to easily design our responsive web application. It's easy to learn, you can get it up and running in minutes and it does most of the heavy lifting for you. Although you don't need to know CSS to use Bootstrap, it is recommended that you have a strong understanding of CSS and HTML beforehand to be able to make unique and elegant websites.

Google Docs

We used Google Docs to collaborate on our project proposal, requirements, technical report and anything else that required us to share information through writing.

Hosting

We used a combination of Namecheap and AWS EC2 to host our website online. Namecheap's DNS made it simple for us to obtain our domain, create our API's subdomain, manage our redirects, and link our domain to our EC2 instance.

AWS EC2, on the other hand, allowed us to host our files, run our apps on Docker containers, easily automate our changes through Gitlab's pipelines and link our instance to our domain to make it available online.

Pagination

Frontend

We used a react-js-pagination package to incorporate pagination into our website. The package provides us with an easy-to-use pagination component that dynamically updates the model pages with different instances of charities, cities, and counties from the database when the user navigates through different page numbers.

Backend

Flask-restless provides pagination to the API's created via its API manager. Each request to api.fightpoverty.online/api/charity, api.fightpoverty.online/api/city, and api.fightpoverty.online/api/county returns a JSON object response with the following:

```
{
      "num_results": <Integer giving the total number of objects of this
type inside database>,
      "objects": [<An array of length specified when creating the API, in
our case 9>],
      "page": <Integer denoting this response's page number>,
      "total_pages": <Integer denoting the total number of pages of this
object in database>
}
```

Thus the RESTful API provides all information needed to know what page to be on, how many pages there are, and a set max number of elements that will be returned (the number of objects per page specified when the API was created).

DB

The DB is a MySQL (5.7) database running inside a Docker container. It was pulled from Docker hub here. The host machine running the Docker image is an Amazon EC2 instance exposing the database on port 3306. See the “Models” section above for schemas used in the database.

How data was inserted into the database

1. Collected all locations we have charities for using this Python module. Locations were saved into this file.
2. Added all counties and cities we have charities for into the database using this Python module. A county would get added first, and then the id assigned to it by the database was used to then add the city (referencing the county id the city is in).
3. Added all charities into the database using this Python module. For each charity, the module queries the database using the charity’s city/state to find the city and county IDs the charity is in. Those IDs were then used to add the charity into the database.
4. Lastly, this Python module was used in this Python module to set each county’s poverty percentage and population using US census data.

Testing

Unit tests of the JavaScript code using Mocha

9 Mocha unit tests were made to test the Javascript code used in the frontend. We tested the calls to the backend to make sure the frontend was receiving the correct data and information to be displayed. Three sets of tests were conducted on each model component (cities, counties, and charities):

- *get* methods: The *get* methods for each model were tested to see if they were returning the correct number of instances (nine) per page to be displayed in each model page.
- *getSpecific* methods: The *getSpecific* methods of each model were tested to see if they would return the correct instance of whatever instance was asked for. This is to ensure that an instance of one model successfully links to the proper instances of other models.
- *getMore* methods: The *getMore* methods of each model were tested to make sure that the model pages can receive different pages of instances from the database. This is to ensure that the user will get unique instances when clicking on different page numbers.

Acceptance tests of the GUI using Selenium

10 acceptance tests of the GUI using Selenium were created. With Importing from unittest and using the Firefox webdriver we created tests to check for failures in our websites GUI. The assertIn(a,b) method was commonly used to check for failures. We covered testing various parts of the GUI such as page titles, page headings, text, and buttons. Three various strategies to locate elements in a page were used:

- *Find_elements_by_class_name*: Use this when you want to locate an element by class attribute name. With this strategy, the first element with the matching class attribute name will be returned. If no element has a matching class attribute name, a NoSuchElementException will be raised.

- *Find_elements_by_css_selector*: Use this when you want to locate an element by CSS selector syntax. With this strategy, the first element with the matching CSS selector will be returned. If no element has a matching CSS selector, a NoSuchElementException will be raised.
- *Find_elements_by_xpath*: XPath is the language used for locating nodes in an XML document.

Unit tests of the Python code using unittest

10 unit tests were used to test the Python code used in the backend. In addition to testing the Flask server, the unit tests also test code used to populate the FightPoverty database. Many utility modules were used to get data from various RESTful API's around the web and ultimately into the FightPoverty database. Here are more details for each test:

- *test_json_utils*: tests the read/write json utilities used to read and write json files by creating a dict, writing the dict to a file, then reading the dict from the file. These json utilities were used in numerous modules.
- *test_json_utils2*: tests the read/write json utilities used to read and write json files by creating an empty dict, writing the empty dict to a file, then reading the empty dict from the file.
- *test_json_scraper:* tests the scraper used to pull data from various RESTful API's around the web, as well as the FightPoverty API. This test makes sure the json scraper gets a response from the API, places it into a json file, and the json file has the expected response.
- *test_json_scraper2*: tests the scraper used to pull data from various RESTful API's around the web, as well as the FightPoverty API. In addition to testing what the first test tests, this test's request also includes a page number parameter.
- *test_state_name_from_abbrev*: tests the utility that takes a state's abbreviation as input and outputs the state's full name. Both the charities data and zip code data we scraped stored state abbreviations. This utility helped convert when using those data sets. Also, this utility will likely be used to convert user input into a state name.

- *test_state_name_from_abbrev2*: makes sure the state name from abbreviation utility returns an empty response for a state it does not know about. Sometimes the data has odd state names which would break other modules if this does not happen.
- *test_state_name_from_num*: tests the utility that takes a state's number as input and outputs the state's full name. The US census keeps track of state's in numbers, which complicated some modules. This utility helps when using US census data.
- *test_state_name_from_num2*: makes sure the state name from number utility returns an empty response for a state it does not know about. This prevents other scripts from halting or erroring out.
- *test_sql_utils*: tests that a SQL connection can be established providing acceptable credentials to our test database to the utility.
- *test_flask_restless*: tests that the Flask server is up and running and provides the expected response to the API's root endpoint.

Unit tests of the RESTful API using Postman

Postman enables you to test each RESTful API in real-time. After writing test scripts for an API, when hitting "Send" inside the Postman app, Postman will automatically run the test scripts. We have general tests for each API to make sure they are working, as well as API-specific tests for each to make sure the body of the response is as expected.

Our general tests make sure of the following for every API's response: the status code is 200, the status is OK, there is no error, the response has a JSON body, and there is no error with the JSON body. Here are the general tests:

```
pm.test("expect response to have status 200", function () {
    pm.response.to.have.status(200);
});

pm.test("expect response to have OK status", function () {
    pm.response.to.have.status("OK");
});

pm.test("response should not have error", function () {
    pm.response.to.not.be.error;
});
```

```
pm.test("response should have json body", function () {
    pm.response.to.have.jsonBody("");
});

pm.test("response should not have json body error", function () {
    pm.response.to.not.have.jsonBody("error");
});
```

Our specific tests make sure each response has expected values inside them. For example, one test for Charities paginated looks like this to make sure the response says there is the right number of Charities in the database:

```
pm.test("expect 571 total charities in database", function() {
    pm.expect(pm.response.json().num_results).to.eql(571);
});
```

To see the tests, import the Postman collection here into your own Postman app. You will then be able to see all tests under the "Tests" tab for each API.

References

1. https://www.census.gov/library/publications/2017/demo/p60-259.html
2. https://aws.amazon.com/s3/