NPOLink

CS 373 – Software Engineering
IDB Project – Phase 1
10am Class- Group #6
npolink.me

By Georgina Garza, Gerardo Mares, Edgar Marroquin, Paul Purifoy, and Jacob Zillifro

---

Motivation

NPOLink is a web application which allows users to attain information about various non-profit organizations. Using this website, users are able to find out about non-profits in their area, non-profits by category, and even find ways to volunteer for and help these non-profits. We noticed that often websites either just gave direct information about a non-profit's name and tax-exempt code/non-profit category, or simply listed events to volunteer for. We wanted users to be able to find non-profits that they could actively be passionate about based on either location, category, or both, and find out various information about it.

User Stories

1. As a user, I should be able to get all of the organizations within one category so that I may find one that suits what I'm looking for.
    - Our RESTful API has been designed with a call that will directly allow for this to occur (GET Specific Category). This call will data will be originally scraped from the Nonprofit Explorer API, and then be held in our database. Our GET call will specifically access our database.
    - Estimated Implementation Time: 3hrs
    - Actual Implementation Time:
2. As a user, I should be able to extract the location of an organization so I can know if it is relevant to me or not.
    - Our RESTful API has been designed with a call that will directly allow for this to occur (GET NPO by Name). This will provide various data about the organization, which will include the location.
    - Estimated Implementation Time: 3hrs
    - Actual Implementation Time:
3. As a user, I should be able to extract the codes for any organization and find out what the code means so I can learn about the operations of the organization.
    - Our RESTful API has been designed with a call that will directly allow for this to occur (GET NPO by Name). This will provide various data about the organization, which will include the tax-exempt code.
    - Estimated Implementation Time: 3hrs
    - Actual Implementation Time:
4. As a user, I should be able to get volunteer opportunities connected to an organization so that I can know what is available to do for an organization.
    - Our RESTful API has been designed with a call that will directly allow for this to occur (GET NPO by Name). This will provide various data about the organization, which will include the volunteer opportunities.
    - Estimated Implementation Time: 3hrs
    - Actual Implementation Time:
5. As a user, I should be able to extract the description of every organization so that I can be informed about what each non-profit does.
    - Our RESTful API has been designed with a call that will directly allow for this to occur (GET NPO by Name). This will provide various data about the organization, which will include the description.
    - Estimated Implementation Time: 3hrs
    - Actual Implementation Time:
6. As a user, I would like to see multiple types of multimedia for instance pages so I can get a better insight into the instance.
    - We have added various forms of multimedia for our instance pages since Phase 1. For example, we have added maps onto our Location and Nonprofit instance pages

    - Estimated Implementation Time: 2hrs
    - Actual Implementation Time: 2hrs
7. As a user, I should be able to go to any of the three model pages and see multiple pages of data.
    - Our UI calls our API to gather the paginated results of elements in our database. In real time, when a model page is first loaded up an API call to page 1 is automatically made. From there, when a user clicks on a specific page number, the API is called for that page number.
    - Estimated Implementation Time: 3hrs
    - Actual Implementation Time: 3hrs
8. As a user, I should be able to go to one of the three model pages and see updated information so that I can see the most updated instances at all times.
    - Our database is updated through scraping every few days so that our model pages will be as up to date with instances at all times.
    - Estimated Implementation Time: 1hr
    - Actual Implementation Time: 1hr
9. As a user, I should be able to go to any instance of any model and have a page that shows related instances of the other two models.
    - For each instance in the database, we have stored the related models as keys, which we can then access the information for by the key. On our UI for each instance, we make a clickable link to the related models for that instance, which allows a user to see the relations and connections.
    - Estimated Implementation Time: 2hrs
    - Actual Implementation Time: 1.5hrs
10. As a user, I should be able to see five pieces of information for each instance on the model page.
    - For each model page, on the grid for the instances, we always show 5 attributes. For nonprofits, we show an image, name, description, EIN, and location. For categories, we show category name, category code, image, parent category, and a short description. For locations, we show the name of the location, city, state, a description, and an image.
    - Estimated Implementation Time: 1hr
    - Actual Implementation Time: 1hr
11. As a user, I should be able to filter on multiple attributes like category codes, parent codes, etc.
    - For each model page, we have different attributes that can be used for filtering. For the nonprofits model page, a user can filter by the number of projects/events available and the location. For the locations model page, a user can filter by the state. For the categories page, a user can filter by nonprofits available and the parent category.
    - Estimated Implementation Time: 2hr
    - Actual Implementation Time: 2.5hr

12. As a user, I should be able to do a search across all model pages so that I could see associations between the models given some kind of keyword. I want this search functionality available no matter what page I'm on.
    - On our website, we have a universal search. A user can input multiple words in this search, and it will look across all instances of the database. The results will then be shown and organized by the different model types. We have a search feature on our navigation bar, so a universal search can be accessed from any page on the website.
    - Estimated Implementation Time: 1hr
    - Actual Implementation Time: .5hr
13. As a user, I would like to be able to search within a specific model. I want this so I could search by keywords such as a city or a specific category that I'm interested in.
    - Each model page has a specified search for that model. This search will look through all attributes of each instance of that model, and give a result of the instances that contain that keyword.
    - Estimated Implementation Time: 3hr
    - Actual Implementation Time: 2hr
14. As a user, I should be able to sort the model pages. I want to be able to search by different attributes, and I would also like to reverse the sort so I can, for example, sort A-Z and Z-A.
    - Each model page has its own sort, which allows the instances to be sorted alphabetically. Although each instance can be automatically sorted alphabetically, in some instances we also sort in other ways, such as alphabetically by state for the locations model page. These sorts have a way to be ascending or descending (so A-Z and Z-A).
    - Estimated Implementation Time: 2hr
    - Actual Implementation Time: 2hr
15. As a user, I should be able to do some sort of combination between sorting, searching, and filtering to get a more specific query.
    - Our webpage automatically takes care of this. There is no restriction from being able to filter, sort, and search. Therefore, a combination of these should be no issue at all.
    - Estimated Implementation Time: 0hr
    - Actual Implementation Time: 0hr

RESTful API

The overall documentation for out RESTful API can be found at the following link: https://documenter.getpostman.com/view/5491513/RzZ3K1zY

All of our calls will be formatted in the way the documentation has specified, and return a JSON object. Here is some information about the various GET calls that we have implemented:

Nonprofits Endpoint

- GET NPOs
  - Retrieves a list of all non-profit organizations within our database.
- GET NPOs by Page
  - Retrieves a list of 12 non-profit organizations within our database based on the page number given.
- GET NPOs by ID
  - Returns the non-profit organization with the given ID.
- GET NPO by Location
  - Returns a list of the non-profit organizations that are located in the location ID given.
- GET NPO by State
  - Returns a list of non-profit organizations that are located in the state specified by its 2-letter abbreviation.
- GET NPO by City
  - Returns a list of non-profit organizations that are located in the city and state specified.
- GET NPO by Location and Category
  - Returns a list of non-profit organizations that are located in the location ID given and that have the category ID given.
- GET NPO by State and Category Code
  - Returns a list of non-profit organizations that are located in the 2-letter abbreviation of the state given and that have the category code
- GET NPO by Keyword(s)
  - Retrieves a non-profit organization based on a keyword found in either the description or list of events.

Locations Endpoint

- GET All Locations
  - Retrieves all the locations for non-profit organizations held within our database.
- GET Locations by page
  - Retrieves a list of 12 locations within our database based on the page number given.
- GET Location by Location ID
  - Returns the location with the given location ID.
- GET Locations by Category
  - Returns a list of locations that contain non-profit organizations that have the category ID that is given.
- GET Locations by Category Code
  - Returns a list of locations that contain non-profit organizations that have the category code that is given.
- GET Locations by Nonprofit ID
  - Returns the location of the specified non-profit organization ID.

Categories Endpoint

- GET All Categories
  - Retrieves all categories for non-profit organizations held within our database.
- GET Categories by page
  - Retrieves a list of 12 categories within our database based on the page number given.
- GET Category by Category ID
  - Returns the category with the given category ID.
- GET Categories by Location ID
  - Returns a list of categories which have associated non-profit organizations in the location ID given.
- Get Categories by Nonprofit ID
  - Returns the category of the non-profit organization specified by the non-profit organization ID given.

Models

1. Non-profit Organizations
   - This model shows various non-profit organizations.
   - Attributes so far:
     i. Name (String): The name of the organization.
     ii. Description (String): The description/mission of the organization.
     iii. Image/Logo (URL (String) to image): A logo/image for this organization.
     iv. Address (String): The address of where this nonprofit is located.
     v. Location (Key/Link): In the database a key, on the UI a link, to the Location instance of Locations model for this nonprofit.
     vi. Category (Key/Link): In the database a key, on the UI a link, to the Category instance of Categories model for this nonprofit.
     vii. Volunteer Opportunities (List of Events (Each has Name (string), Description (string), Link)): A list of volunteer opportunities for this nonprofit organization where each holds information about the name, description, and link for a user to find out more.
2. Locations
   - This model shows various locations that have non-profit organizations.
   - Attributes so far:
     i. Name (String): The city and state combination for this location.
     ii. Description (String): A description of this city.
     iii. City (String): The city of this location.
     iv. State (String): The state of this location.
     v. Image (URL (String) to image): An image that represents this city

vi. Organizations (List of Keys/Links): In the database, this is a list of keys of the non-profit organization instances located in this location, and on the UI a list of clickable links to those nonprofits.
vii. Categories (List of Keys/Links): In the database, this is a list of keys of the category instances located in this location, and on the UI a list of clickable links to these categories.

3. Categories
    - This model shows various categories for non-profit organizations.
    - Attributes so far:
        i. Name (String): The associated name of the category for the given category code.
        ii. Description (String): A description of what the non-profits in this category represent.
        iii. Code (String): The code associated with this category. This is considered the id for the category.
        iv. Parent Category (Char): The category code which this category is listed under. For example, given category code 'A01' the parent category would be 'A'.
        v. Image (URL (String) to image): An image that represents this category's parent category.
        vi. Organizations (List of Keys/Links): In the database, this is a list of keys of the non-profit organizations which have this as their category code, and on the UI a list of clickable links to these non-profits.
        vii. Locations (List of Keys/Links): In the database, this is a list of keys of the location instances of Locations where non-profits for this category code exist, and on the UI, a list of clickable links to these locations.

Tools

1. Namecheap
    - We used Namecheap in order to obtain our URL (npolink.me), and as a way to help redirect our AWS Elastic Beanstalk application to the correct URL (see more information under Hosting).
2. Amazon Webservices – S3
    - Currently, we are taking a zip of our repository files and holding them in an S3 bucket so that our Elastic Beanstalk can have access to the code. We hope to further implement this in a more dynamic way so that when the repository updates, the website does as well.
3. Amazon Webservices – Elastic Beanstalk
    - We are using AWS's Elastic Beanstalk service to host our website. By grabbing the code from the S3 bucket, and then redirecting to our URL, this application hosts our web application.
4. Docker
    - We are using Docker as a simple way to run the application.

5. Docker-Compose
    - We are using Docker-Compose in order to create a way to work with multiple containers to run both the backend and the frontend at the same time. This will allow for greater efficiency in our program and will be further implemented as the website becomes dynamic.
6. Gitlab
    - We currently use Gitlab as a way to hold our code and work interactively. We create issues within our repository in order to keep track of what still needs to be completed and make various branches so that different aspects of the project may be worked on concurrently without merge conflicts.
7. Postman
    - To use Postman, one of our team members created a workspace and then downloaded the Postman app (available for download online) in order to begin.
    - We used Postman to scrape the data that is currently being represented statically on our website. To do this, we looked into the details of our API's and discovered the type of ‘GET' calls we would need to make in order for our team to get the information we needed.
    - We also used Postman to design our API, and create the documentation for it. Refer to RESTful API portion for more details. This was created by considering what users may be interested in being able to access using our API, and designing various ‘GET' requests to meet these needs.
    - In Phase 2 we began using Postman for the unit tests of our API. Postman allowed us to make example calls and check the results against what we would expect.
8. React-Bootstrap
    - React-Bootstrap was used across the entirety of our website in order to provide quality layouts and styling for our web application's frontend. By using react in general, we were able to break up our application into components, allowing for faster designing of each page. For example, a component could be made which represented the layout for each of the model pages.
9. ReactStrap
    - In addition to React-Bootstrap, in a few instances, we used ReactStrap for our frontend. This included the home page, navigation bar, and pages for the instances of the various models. ReactStrap furthered our possibilities for our layout and provided ways in which we could easily add interactive tiles.
10. Flask
    - Flask is a python framework we are using for the backend of our application. As our web application is not dynamic yet, we have not gotten in too deep to using Flask.
11. Grammarly
    - Grammarly was used in order to clean up and polish the writing within the technical report.
12. Slack

- Slack was a simple tool the group used to interact and communicate throughout the project.
- Slack was directly integrated with our GitLab repository in order for automatic messages to be sent about recent updates about code and issues.

13. AWS-RDS/PostgreSQL
    - We host our database on the AWS provided tool, RDS. RDS is AWS's way to manage a relational database. We then specifically use PostgreSQL. This allows for easy connection between our elastic beanstalk and database.
14. PlantUML
    - PlantUML is an online service that helps a user create a UML diagram. We used this in our group to create our UML diagram which represents our database, including the various models and how they relate to each other in the database.
15. HTML5 UP
    - We used HTML5 UP as a way to improve the creativity of the front end. By using this, we were able to more clear templates for our pages, especially the homepage, and our navigation bar. We were able to use HTML5 UP in connection with our previous React and Bootstrap Elements.
16. SQLAlchemy
    - SQLAlchemy was used as the interface to access our database. This tool allowed us to create code for scraping data and creating relations in the python language.
17. Python: unittest
    - We used the unittest feature from python as a way to create unit tests for our backend, and also our acceptance tests of the GUI. This allowed for simple and continual testing, and therefore proper unit testing through the gitlab pipeline.
18. Selenium
    - Selenium was a tool we used to create the acceptance tests of the GUI. This tool is an easy way to bring up a mock web driver and test that certain links can be reached, and then that these links contain the appropriate information.
19. Mocha
    - Mocha was the tool we used to create the unit tests of our frontend.

Hosting

We are currently using both Namecheap and Amazon Web Services to host our website. We used Namecheap in order to obtain our URL, npolink.me, through their education account options. Once we had the URL, we worked on the website itself and put that code into a zip, and placed it into an AWS S3 bucket. Following this, we set up the Elastic Beanstalk application's environment, which was dependent on the code in the S3 bucket, and obtained the automatic URL this application was being hosted on. From here, we used AWS's Route 53 Management Console to create a way for the Elastic Beanstalk application to automatically redirect the given URL to redirect to our custom one. To do this, we created a Hosted Zone, with the name of our application, and within that created a CNAME Record Set with the value set to the automatic URL from the Elastic Beanstalk application. This set up the AWS side. We

then had to go into Namecheap and create redirect URL's from our website to that of the automatic URL for the application. This can be done by creating CNAME records and URL redirect records through Namecheap. At the end of these steps, the web application was then properly hosted on our custom URL.

Pagination

Backend

For each model page (Nonprofits, Locations, Categories), we have created a 'GET' request that is specific to pagination. For each of these pagination calls, we have the API result 12 instances per page. The call itself will also have information about the page number being returned, the number of results (always 12 except last page) for that page, the instances themselves, and the total number of pages.

Frontend

We used bootstrap pagination in order to display a page bar at the bottom of the screen for each of these model pages. The range of page numbers shown depend on the page the user is currently on, and the maximum number of pages. When a model page (Nonprofits, Locations, Categories) is first loaded, the first page will already be loaded in by calling the API appropriately for page one of that model. From here, whenever a user clicks to go to a new page, an API call will be made to that model with the specified page number, which will then update the results and display the various instances on the page.

Database

Using python scripts, we first scraped our various API's to get the data and combine it together into the models we wanted to create. From here, we loaded the data into our AWS RDS (PostgreSQL) database. This was then able to be our database for our API. We run the scripts every few days as to make sure we always have the most up-to-date and relevant information available.

Testing

We had a few various forms of testing.

Acceptance Tests

Our acceptance tests themselves tested the GUI of our website. To do this, we used Selenium, which allowed us to call up a mock web browser which could then call specific URL's, make sure specific clickable links worked, and that text was showing up on a page as it should be. By performing these tests, we can ensure that our UI works and loads pages correctly.

Backend Tests

We were able to perform backend tests using Python's built in unittest. This allowed our tests to be automatically run when running the file, for ease in checking if these tests pass not only from our command line, but our pipeline as well. For our backend, we perform a test on every single function that exists. This means that we perform a test for every possible scenario for our

routes. We have these tests across information for nonprofits, locations, categories, and the configuration.

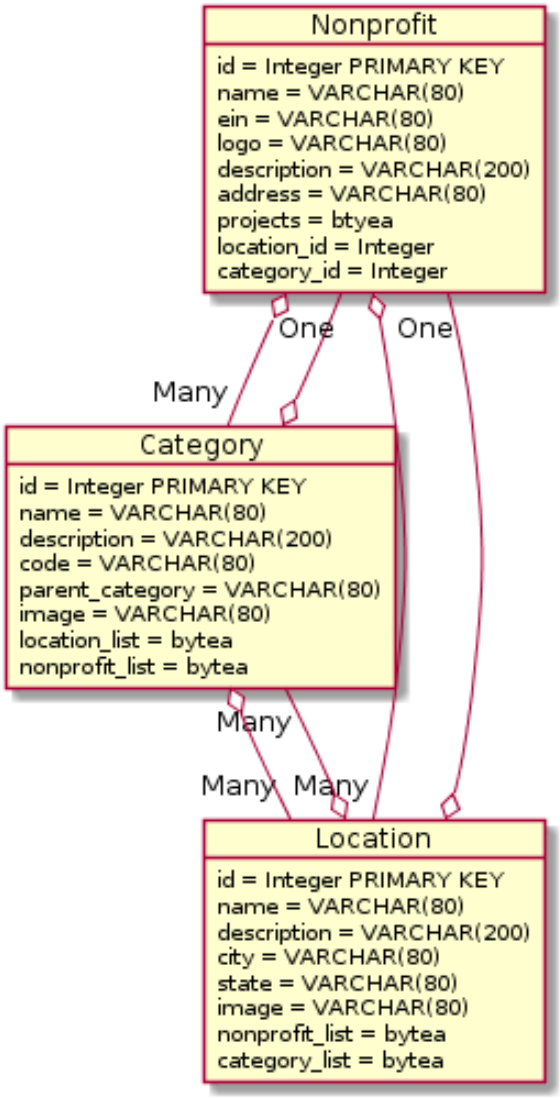
Frontend Tests

We were able to test our frontend, aka the JavaScript, using the tool Mocha. This allowed us to test various functions within the javascript in our code to ensure that it works properly, and will therefore produce the appropriate information onto our frontend.

API Tests

We have tests for every single one of our API calls, therefore ensuring that all the results from calls to our API return correctly. We were able to create and perform these tests by using Postman. Since we wanted each API call to be tested, for ease, we had our scrips loop through all the various pages, for example.

UML Diagram



Nonprofit

These are nonprofit organizations. To get information about non-profit organizations, we scraped from two separate API's: Non-Profit Explorer and Global Giving. We combined the data we got from the two APIs in order to create one model for each non-profit. We were able to do this by looking at the EIN, which is a unique way to identify each non-profit. Each non-profit has the following columns:

- Id: This is an integer that is the primary key for the nonprofit organization.
- Name: This is the name of the non-profit organization, represented as a string.
- EIN: This is the EIN associated with the nonprofit organization, and is specific to each organization, represented as a string.
- Logo: This is a string of the URL to the logo of this organization.
- Description: This is a string with the description for this organization.
- Address: This is a string with the address of where this organization is located.
- Projects: This is a list of projects associated with this non-profit organization. In our database, this is stored as raw bytes.
- Location_id: This is the primary key/id of the location where this non-profit is located, which is stored as an integer.
- Category_id: This is the primary key/id of the category of this nonprofit, which is stored as an integer.

The non-profit table has a many-to-one relationship to the category and location tables. This is because each non-profit can only be associated with one location and one category, but locations and categories may be associated with many non-profit organizations.

Category

These are the various categories associated with nonprofit organizations. We were able to gather a list of categories from the Non-Profit Explorer API. We then used the information that we scraped to figure out each sub-category's parent category, and from here were able to find an image associated with that parent category. Each category has the following columns:

- Id: This is an integer that is the primary key for the category.
- Name: This is the name of the category, represented as a string.
- Description: This is a string with the description for this category.
- Code: This is the code (letter and numbers) of this category, represented as a string.
- Parent_Category: This is the code of the parent category (a letter) for this category, represented as a string.
- Image: This is a string of the URL to the image associated with this category (which will also be the image associated with the parent category).
- Location_list: This is a list of location primary keys/ids that are associated with this category. In our database, this is stored as raw bytes.
- Nonprofit_list: This is a list of nonprofit primary keys/ids that are associated with this category. In our database, this is stored as raw bytes.

The category table has a many-to-one relationship to the nonprofit table, but a many-to-many relationship to the location tables. This is because each nonprofit only has one category, but each category could be associated with many nonprofits. Additionally, there can be many locations associated with many categories.

Location

These are locations that have nonprofit organizations. We were able to gather a list of locations based off of the data that we got from the nonprofit model. Once we got this list, we were then able to call the Wikipedia API in order to scrape data for an image and description for each city. Each location has the following columns:

- Id: This is an integer that is the primary key for the location.
- Name: This is the name of the location, stored as a string that will be in the form ("City, State").
- Description: This is the string with the description for this location.
- City: This is the string that has the specific name of the city.
- State: This is the string that has the specific state for this location.
- Image: This is a string of the URL to the image associated with this location.
- Nonprofit_list: This is a list of nonprofit primary keys/ids that are associated with this location. In our database, this is stored as raw bytes.
- Category_list: This is a list of category primary keys/ids that are associated with this location. In our database, this is stored as raw bytes.

The location table has a many-to-one relationship to the nonprofit table, but a many-to-many relationship to the category tables. This is because each nonprofit only has one location, but each location could be associated with many nonprofits. Additionally, there can be many categories associated with many locations.

Filtering

The filtering for each model page is a little different, however, all are very similar in implementation. There are different attributes that a user can filter by depending on the model page:

- Nonprofits Page
  - Available projects/events
  - Location
- Locations Page
  - State
- Categories Page
  - Parent Category
  - Nonprofits available

For our frontend, we simply have a drop-down menu with the options that are available for that model page. Once a user clicks on an option, we update the instances being shown. In our

backend, we go through the instances of that model page and look for the ones that contain what we are filtering for. From here, we get a proper result and display it on the webpage accordingly.

Searching

We have 2 different forms of searching: a universal search, and then a search that is specific to each model page.

- Model Page Search
    - On our frontend, we have a search bar available at the top of each model page. A user can enter any text or keywords into the search bar, and then once they want to search, they simply click the submit button. This sends the information the user entered to the backend, but on the frontend, the instances which have those keywords will then pop up with those keywords the user searched highlighted.
    - The frontend calls the API call that is specific to the search of that model page. This search then looks for the keywords that the user entered across all the attributes within the tables specific to that model page. The call then returns a list of the instances which contain the keywords.
- Universal Search
    - On our frontend, we have a page that is dedicated to the universal search. On this page, there is a search bar, and similarly to the model pages, a user can enter as much or as little text as they want, then click submit. After clicking submit, three separate sections, one for each model page will show up. Within each section, whichever instances that pertain to the keywords searched will pop up with the keywords that had been searched highlighted.
    - In the backend, when the user clicks submit on the universal search, a call to our backend/API will be made. Underlying though, this will call the three separate API calls for search on each individual model page. This will then return 3 lists, one for each model page, with the list of instances that contain the user's submitted keyword(s).

Sorting

Each model page has its own sort. These sorts always sort alphabetically, either ascending (A-Z) or descending (Z-A). Each model page does sort a little differently. For example, although each page can just be sorted alphabetically by the name of the instances, our location page can also be sorted alphabetically by the state. Our backend is able to create these sorts by looking at the columns in the table using SQL Alchemy.