Policyand.me - *where policy and people meet*

Group 20:

- Abhishek Dayal
- Alec Sweet
- Atharva Pendse
- Joel Uong
- Malcolm Hess

Introduction

Policyand.me is a website designed to engage and inform the community about national legislators, bills, and interest groups. We used Javascript React for the front-end, AWS for our back-end, and Postman to handle our API.

Motivation

The idea for policy and me was conceived when we realized how hard it was for people to understand American policy. Modern politics is weighed down with obscure language and questionable politicians. We decided we wanted to create a community for people with an interest in politics to have civil discussions and debates. In our website, you can find what is currently being done in congress, the status of a bill in congress and how much support it is receiving, and how you can support legislators and interest groups that align with your views.

Tools

- React
  - Javascript based library for building user interfaces
- Sass
  - CSS extension that allows for more functionality
- AWS Elastic Beanstalk
  - Service for hosting our Flask Backend

- AWS RDS service
    - Service for hosting our MySQL database
- AWS Amplify
    - Web service that hosts our React frontend
- Postman
    - API client to perform GET request and create our own API
- Namecheap
    - Domain name service provider
- Jest
    - JS based testing suite used to do unit testing on the frontend
- Selenium IDE
    - Used to test the GUI of the front end
- Flask
    - Python framework used for the Backend
- SQLAlchemy
    - Python tool used to interact with databases
- MySQL
    - Relational Database Engine with some basic features
- D3
    - Used to create data visualizations

Website

Our website uses React material UI for the front-end and a CSS preprocessor called Sass which expands the functionality of CSS. On arrival to the website, the user is greeted with a splash page. From here the user can navigate the website using the bar at the top of the page to reach one of the three model pages or the about page. Each model page contains a few entries that are also clickable buttons that leads to an instance page. The entries are built as objects with the attributes as instance variables.

Pagination

We implemented pagination by making one API call to the back end and getting an array of one of our 3 model instances. The array can be of any size and is currently set to 600. We then display however many page buttons we want, currently 10, and then show a slice of the array based on the page index. At the moment, the page displays 60 entries at a time so if the user selects the 3rd page, the website will multiply the index by the entries, (2x60)=120, and then display a slice of the array with the multiplied index plus the entries (array[120:180]). The values 600 and 60 are accurate for legislators (we use different values for other models, depending on the total number of instances of those models.

Hosting

We used AWS to host our website and obtained our domain name from NameCheap. In order to get AWS and Namecheap to work together, we added multiple nameservers to Namecheap that redirected to AWS route 53 for CName records. We also added the SSL certificate to Route53 to allow our website to use HTTPS (we had to generate separate SSL certificates for the backend and frontend servers). Our frontend server is hosted on AWS Amplify at www.policyand.me, while our backend server is hosted on AWS Elastic Beanstalk at api.policyand.me. Both are secure and can communicate with each other. Our database is hosted with Amazon RDS using MySql.

User Stories

We received feedback on our website and modified our design accordingly.

Phase 1:

Policyand.me user stories

- Make the “Policy and Me” banner a homepage button.

    - This was fixed by making the text banner respond to being clicked and sending the user to the homepage.
- Add pictures of the legislators.
    - This was addressed by adding a photo to the MoreInfo class and displaying it on the page.
- Better text formatting
    - This was addressed by placing each data entry into a box for a cleaner design.
- Better button styles
    - This was addressed by editing the color and the style of our buttons.
- Description of legislators
    - Unfortunately, this is one request that we could not sustainably fulfill as none of the databases we used has this specific information.

<u>Doctorsearch.me user stories</u>

- Rating Scale
    - We asked our developers to make the rating system more clear by adding what the rating was out of. Such as 4.5/5 instead of 5.
- Title
    - We asked them to change the title of the page and they delivered.
- Cities Zipcodes
    - We noticed that their website had 6 numbers in the cities instance but we wanted it to be more clear that those numbers were the zip code.
- Whitespace
    - There was too much whitespace when we clicked on additional info and we wanted this to be cleaned up and suggested that they moved the location and information to be side by side.
- Center titles on model pages

    - We asked our developers to center the titles on the model pages because we thought that would look cleaner.

Phase 2:

<u>Policyand.me user stories</u>

- About page bios
    - Our customer asked us to add a short bio to each person in our about page. We created these bios and had them uploaded in about 30 minutes. Our response was that we would have them done in about 30 minutes and uploaded by midnight.
- Pagination on the bills, legislators, and interest groups
    - Our customer asked us to implement pagination on our model pages so that the user could focus on what was on the screen. Our response to this was to implement pagination on all of our model pages. Our time estimate was about 1 hour.
- Description of legislators
    - Our customers asked us to add small biographies for each of the legislators. Our response to this request was that our APIs did not include a biography and to implement this, we would have to scrape another website like Wikipedia. We felt like that would fall out of the scope of this class.
- Add images relating to the topics for the interest groups section
    - Our customer asked us to add images to our interest group section. Our response to this request was that our APIs did not include any kind of logo or picture element that related to the interest groups. This request was not sustainable as we would have to manually find pictures or logos for each interest group and add them to our database.
- Buttons on the navigation bar disappear when resizing the window

    - Our customers notified us of an issue where if you resize the window, the buttons on the navigation bar disappear. The issue is that the buttons fall off the navbar and are below it, but the text is white and mixes in with the background. Our response was to do a quick fix and make the navbar expand in height but we want to add a hamburger button and make the navbar more robust in the future. This took about 15 minutes.

<u>Doctorsearch.me user stories</u>

- Ratings on Doctor instances
    - We noticed that the rating for doctors without ratings was showing "/5". This was confusing and we asked our developers to make this more clear. We offered them examples to fix this issue with replacement text like "no rating" or "not yet rated". They fixed this issue and now the site displays "No Rating".
- Rework of Doctors Instances
    - We noticed that some of the doctor instances were slightly corrupted and were not displaying the proper image or map location. They responded and asked us to take a look at the website to see if they had fixed the issue. We checked their website and the issue had been resolved.
- Some pages of Doctor does not load
    - While clicking on the page numbers at the bottom of the model page, we noticed that some of the pages did not load. We used our developer tools and saw that the site was getting a null error. We notified our developer of this issue and they promptly resolved it.
- Have a better description in About page
    - We noticed that the about page for doctor search could be improved by adding more information about the goal of their website and some of the features that they provided. We created this issue and they responded by updating the about page.

- Information in cities instance
    - We noticed that the information provided on the cities instance page was centered plain text and could be improved with some formatting. We gave them examples of what they could do such as bolding the categories and right aligning the text to make it look cleaner. They responded by implementing our suggestions.

Phase 3:

Policyand.me user stories

- As a user, I would like even card sizing for the bills page
    - Our customers asked us to make the card size of each bill on the bills page more even so we could achieve a more aesthetic look.We replied telling them that this would take about 15 minutes. We closed this issue by aligning the text in each card in the center
- Bolded categories and data throughout
    - Our customers asked us to make the categories for each of the instance bold. We replied by telling them that this change would take about 15 minutes. We deviated a little from the request and made the attributes bold and the categories normal for a more readable page.
- Commit totals on the about page
    - Our customers asked us to make the total commits on the about page more readable by taking them out of the corner. We responded by telling them that the change would take about 15 minutes. We fixed this by placing the “stats” in the center of the page.
- Pagination bar stuck to the bottom of the page
    - Our customer alerted us to a small bug where the pagination bar get stuck at the bottom of the page. Our response was that this issue would take about 30 minutes to fix. We fixed this issue by adding some padding in between the bottom and the page buttons.

- Whitespace on the details pages
    - Our customers informed us about how on our legislator instance page, we have a lot of white space on the right side. They suggested that we center the data on the screen or make the picture larger. We responded by telling them that we would have to discuss how we wanted to move forward with this issue. For now we decided to push back this issue till next release, where we will try to improve the look of the UI.

Doctorsearch.me user stories

- Create an Icon
    - We asked our developers to create an icon to improve the look of the website. They had a picture of a random guy on the splash page and we thought it would be better if they had an icon instead. This issue took them about 15 minutes to implement.
- Make each row in table of specialties page clickable
    - We noticed that only clickable part was the specialties in the table that they had. We thought that it would be better and more consistent with the website if they made the whole row clickable. They fixed this issue in about 40 minutes.
- Add filtering capabilities to get more relevant searches
    - We informed our developers that it was hard to navigate through all of the pages and that a list or table might make it easier to see. They responded by implementing filtering and allowing the user to search for the doctors they are looking for. This issue took them about 4 hours.
- make developer blocks on the about page same size
    - We noticed a small bug with our developers website where one of the blocks on the about page was slightly bigger than the rest. We thought that it might be a cropping issue and informed them accordingly. This took them about 10 minutes to fix.

- Create a search bar for easy searching
    - We thought that there was a lot of data on their website and a lot of pages to sift through. We suggested the idea of a search bar to help the user find what they are looking for quicker. They implemented this idea in about 5 hours.

RESTful API

Our database is constructed using Postman with information from:

- https://www.opensecrets.org/:
    - Money contribution data for significant interest groups and legislators
- https://projects.propublica.org/api-docs/congress-api/:
    - Primary bill Information
    - Primary legislator Information
    - Connection data between bill and legislator by sponsorship using sponsor_id and bill_id
    - Bill subjects for connection with significant interest group topics
- https://votesmart.org/share/api:
    - Primary significant interest group information
    - Connection data between legislators and significant interest groups by candidate rating between candidateId and sigId for each candidate and each significant interest group
    - Legislator pictures
    - Significant interest group topic data for connection with bill subjects

We created our own API using the information gathered from the websites above. We created a simple schema that allows a user to send a get request for bills, legislators, or interest groups based upon query parameters.
Documentation: https://documenter.getpostman.com/view/8970979/SVzua1tY

Models

Our website contains three different models, each with their own attributes and instance pages.

Bills:

- Legislative bills proposed to become public policy for the United States of America if approved by Congress and the president
- Expected to contain about 8000 entries
- Attributes: congress number, bill id, primary subject, sponsor, status, summary, shortened summary, sponsor state, date Introduced, committees, latest major action, and latest major action date
- Instance page contains: Introduced Date, Primary subject, Sponsor, Committees, Congress number, Number of Cosponsors, Latest major action, Latest major action date, Bill number, Sponsor ID, Sponsor state, and Summary
- Connections:
    - Connects to legislators through the primary sponsoring legislator for a bill
    - Connects to significant interest group through bill's primary subject and significant interest group's topics

Legislators:

- Representatives of the legislative branch of the United States of America who propose and vote on bills to become public policy
- Expected to contain about 500 entries
- Attributes: date of birth, facebook handle, gender, unique id, last name, middle name, first name, percent of missed votes, seniority, title, party, state, twitter handle, percent votes against party, percent votes with party, youtube handle, connection id for Votesmart, personal website URL, photo

- Instance page contains: Name, Picture, Title, Party, State, Gender, Date of birth, Missed vote percentage, Votes with party percentage, Seniority, Facebook, Youtube, Twitter, and website
- Connections:
    - Connects to bills through all bills primarily sponsored by each legislator
    - Connects to significant interest groups through ratings by a significant interest group for a legislator

Significant Interest Groups:

- Organized groups that use members and resources to influence the legislative process of the United States of America to further a particular area of interest in a way that coincides with their views
- Expected to contain about 500 entries
- Attributes: significant interest group id, group name, topics covered, address, description, group's website URL, phone1, phone2, fax, zip code, legislators endorsed, total contributions to legislators, lobbying money spent, democrat contributions, and republican contributions
- Instance page contains: Address, City, Phone, Email, State, Zip, Fax, Website, and Description
- Connections:
    - Connects to bills through significant interest group topics and bill primary subjects
    - Connects to legislators through ratings for each legislator by a significant interest group

DataBase

UML Diagram: https://i.imgur.com/ZYZIZlO.png

The Database is hosted using Amazon RDS with MySQL. The design of the DataBase, as seen in the UML diagram, revolves around three major table models(bills, legislators, and significant interest groups) and three supplementary table models(sigsToBills, ratings, and billsToSigs) which store connection data between the three major models.

- Between the bill and legislators tables is an implicit connection of bill and bill sponsor in which legislator id value corresponds to a bills sponsor_id value.
- Between the bill and significant interest group tables resides:
    - billsToSigs which takes one bill id and provides a list of significant interest group ids(by delimited string) with similar topics to the bills primary subject
    - sigsToBills which takes one significant interest group id and provides a list of bill ids(by delimited string) with a similar primary subject to the significant interest groups topics (this table can also be used to get topic data for a given significant interest group id)
- Between the legislators and significant interest group tables resides the ratings table which provides a unique rating id and rating data for each combination of legislator by id and significant interest group by sigId (ratings data is a percentage or letter grade corresponding to the degree in which a certain legislator represented the interests of a certain interest group)

Testing

<u>Front end unit testing:</u>

For unit testing the frontend we used jest instead of mocha. This allows us to take “snapshots” of our react components which takes the rendered HTML from each component and saves it to a JSON format. Every time the tests are run a new JSON file is created and compared to the old snapshots. We are then alerted to any differences that were rendered. These tests ensure that whenever React renders components, they always look the same.

<u>Front end acceptance testing:</u>

For testing the frontend GUI we used selenium IDE. We ran the IDE from the browser and created a .side file with the testing suite. The tests test the user interface of our website by acting like a user and interacting with the clickable buttons and checking if those buttons loaded the expected elements. The first couple of tests test if the pages load with pagination. The next couple tests check if the user can click on one of the instances, interact with any links, and then click back. Then the robustness tests check if the user can navigate to model, click an instance, click back, click another page, click and instance, click back, and then return to the splash page.

Backend unit testing:

To test the backend we used unit test written in python using the framework that we learned from project one. The main point for these tests was to test if the back end was returning what we expected when we gave it specific parameters. We also wanted to test if the parameters filtered the records correctly. Examples of our tests include getting legislators from a specific state or getting bills written by legislators from a specific state.

Postman testing:

Testing uses Postman internal testing structures to make sure that documented API call return proper responses in a reasonable time with an expected format. Tests between get calls for legislators, bills, and significant interest groups are defined as follows:

1. Check for a successful response, check that the response has a body, check that the body is in JSON as expected.
2. Check the response code to equal the expected success response code.
3. Check that the calls are being made using the proper production environment variable.
4. Make sure that the Content-type of the application response is set to JSON which is the only output format we will support
5. Make sure time taken to receive a response is within reason
6. Lastly, ascertain that on a successful element response the element contains all the expected data attributes.

Filtering

Filtering is currently mostly done on the front-end. We tried to think of a way to move it to the backend using url parameters, but couldn't find a good way to do it. In order to filter, the user has to select an attribute to filter by, and enter a value for that attribute. For example, to filter all legislators from the Democratic party, the user would select 'Party' as the filter by attribute, and enter 'D' as the value. We currently support two filter_by attributes each for legislators and bills, and one for interest groups.

In react, filtering is done by having the search component setting the filter key and filter value in the model page's state. The model page then checks if those values are empty or not when rendering the page. If they are not empty, the native javascript filter function is used to check each if each instance has the specified filter key equal to the filter value. For example instance[filter_key] === filter_value.

Searching

Searching is done on the backend using python string matching library functions. Currently, searching for multiple terms is supported. When a user enters a search query, it is first parsed on the frontend and the spaces are converted to '+' characters, so that it can be sent as a url parameter. In fact, this is also how Google search supports searching for multiple terms. For example, a search query of "the quick brown fox" would result in an api call of the form:
api.policyand.me?search=the+quick+brown+fox

On the backend, the various terms are obtained by splitting along the '+' character. Each term is then matched to all the data in various attributes of each instance, and only the instances that contain all query terms are returned.

The front-end handles searching by adding a component to each of the instances that adds a search bar, sort by selection, filter by selection, and a filter value text field. When the user types into the search bar and clicks search, the component parses the input and sends the string to the API. The instance page then displays the results from that API call. In order to keep pagination working we had to check the length of the new set of instances, divide that by the number of instance we wanted to display and set that as our number of pages. Then we spliced the list based on the number of pages.

Global search is handled as a separate page that is rendered when the user clicks search from the splash page. The global search component renders a page that has a button to switch between each of our models. The button sets the state in the component that calls a function that renders all of the instances of that type with the given search term.

Sorting

Sorting is also handled on the backend by specifying url parameters. For example, if we wanted to sort legislators in alphabetical order of their first names, then the api request would be of the form: api.policyand.me/legislators?sort=first_name.

Sorting is done by first checking if the sort_by attribute is a valid attribute for the instance type. (For example, bills can't be sorted by the attribute first_name since they don't have that attribute). If the attribute is valid, the result list is sorted by that attribute, (alphabetically in case of strings, and in increasing order for integers/doubles). If the attribute is invalid, sorting is not done and the url parameter is ignored. Sorting in the front end is handled in the same way that searching is handled. The search component parses the input from the user and creates a string that is used in the API call. The returned instances are then rendered on the page

Visualizations

We created three visualizations using the data we had. The first visualization is a bar chart that compares the number of sponsored bills for each legislator. The chart makes it easy to see who is sponsoring the highest number of bills. The second visualization is a map that shows the number of interest groups per state. The map allows the user to mouseover and see the specific number of interest groups in that state. It is also colored according to the number of interest groups, with white being the least and dark red being the most. And the last visualization is a bubble chart displaying the frequencies of keywords in bills. This chart is also colored based on the amount of keywords with a range of light to dark blue.

The visualizations were created using D3 examples. Since most D3 visualizations were created using only HTML, we had to recreate them inside of React. The general structure was to load the data into the state of the component, render div classes with specific IDs, and then, in componentDidMount, let D3 render the chart at the div class.

Refactoring

The refactoring we did for the front-end was very minor, removing unnecessary comments and imports, modifying code to use ES6 features (template literals, map data structure, etc), restructuring the code to be more readable, and modifying the stylesheets to use more variables instead of hard-coded colors. One of our goals was to keep the code relatively clean and manageable from the start. Most of the code was written with good coding hygiene throughout the projects. We separated individual components into their own files, we separated the pages from the instance pages, and we used SASS to separate the CSS into its own scss files in another folder.

Similar to the front-end we did not need to make any major changes on the backend. The code was readable and easily understandable, but the biggest problem was that it was all in one file, which we separated into 4 different files, with separate files for Legislators, Bills, and Special Interest Groups and a main file used to connect them. Using a built-in Flask framework called Blueprint’s, we were able to easily compartmentalize the three different models we had.