Team 8: The Political Market

Technical Report

GitLab URL: https://gitlab.com/kevinchenftw/thepoliticalmarket/

Kevin Liang, Kevin Chen, Diyuan Dai, Anisha Kollareddy, Vaishnav Bipin

1. Motivation

Money in government has always been a significant political discussion point, and sometimes seems to have gained traction over the last few years. As a team, we were particularly curious about the data surrounding this topic, and felt we could fulfill a civic duty by educating the interested public about the relevant data. Elected members of Congress are expected to represent their constituents in their home state. But at the same time, it would also be in their interest to consider the major companies in the state as well. Industry success creates jobs and stable economies, so there is already a major connection between companies and politicians.

Another metric we were interested in was the stock market. We had the original idea to compare market trends with Congressmember portfolios, since they are required by law to disclose them. But there wasn't as much transparency as we hoped in perusing the data, as a lot of investment portfolios were with external companies and sources. But the idea of the stock market as a whole proved to be valuable, as it was a metric that could directly evaluate a company and compare them to other companies and current events. The market is indeed always fluctuating, but the hindsight of aggregated data is absolute.

Finally, the government as an entity can spend significant money on a variety of contracts. Likewise, it is a great benefit to a company to be selected for a government contract. A Congressmember is still part of the government, and as an elected official, they also benefit in their campaigns to receive money. Campaign financing is also a form of money in government, and can come from many sources. We hope to take the data collected from our three models and delve into their connections; the stock market, government contracts, and Congressmember finances all certainly relate closely to one another, and we are very interested to see what sort of interconnectedness we can identify and visualize.

2. User Stories

Phase II:

Our "users" mostly requested data-oriented stories to correspond with the content of our dynamic website for this phase.

1. **Campaign Finance Page: Add States and Positions:** As a user, I would like to see information on a politician's current position and state affiliation on their card on the Campaign Finance model page. For example, underneath "T. J. Ossoff" and above "Total

Received" would be the words "Georgia Senator". This is because many people don't know the names of all politicians in congress but would be able to recognize their titles.
    a. We implemented this feature by including the necessary data in the database, then querying our API for that data in the Member and MemberPage components in order to construct the pages as requested.
2. **Congress Member Stock Positions as a Page:** As a user, I would like to have a "Congress member stock positions" page that I can access through the navbar. This could be in the form of a table, or any form that presents the data clearly.
    a. We were not able to implement this in Phase II, as it added too much complexity to our database models and links. As we refine our data presentation and organization this should be able to be completed in Phase III.
3. **Style/Color of website:** As your client, I want the website to have more vibrant colors rather than a darker tone. This can be anything that you guys choose. However, an idea I had was when stocks are trending downwards indicate those with red and when they're trending upwards indicate those with green.
    a. We implemented this both by correcting our existing style as requested, as well as meeting as a team to coordinate future style choices in the lighter tone. We tried to reuse CSS as much as possible across components, and made new components themed as closely as possible.
4. **Collect Data on Many Instances of Each Model:** As a user, I would like more than 3 instances of each model. These instances should collect data from sources with a RESTful API.
    a. Our backend/scripts directory includes many of the intermediary data and scraping tools we used to collect our data. We then integrated the collected data with a SQLAlchemy session connected to our hosted database in order to store all our instances, to be accessed later.
5. **Government Models Data:** As a user, I would like to see a grid or table of government contracts. The data for this page should be fetched from your REST API. The data in this grid or table should also be paginated.
    a. We successfully refactored our Government Models into a monolithic Contracts page, with each instance an individual Contract granted to an organization. This gave us plenty of attributes in the model to display. As with all our other models, we applied our pagination implementation to display this data.

Phase I:

We were asked by our “users” for an “About” page with dynamic contribution statistics as well as creating example instances for each of our data models.

1. **About Page Statistics:** As your client, I want to be able to see gitlab statistics in order to see each user's contribution to the project. If possible, I would like to see these in cards. This will show me how active the team is working on the project.
    a. We implemented this through two queries through the GitLab API in the frontend, but implemented the data collectively in a grid for easier comparison.

2. **Campaign Finances Model:** As your client, I want to be able to access campaign finances and filter by contributors, amount, and time. This could be displayed as a table or as a grid. This would allow users to understand the impact of donations on political campaigns.
    a. We implemented this model through our OpenSecrets API, collecting campaign finance data on various members of congress. We chose a card layout to better display the media for each politician.

3. **Government Contracts Model:** As your client, I want to be able to access a "government contracts awarded" component page. It would be helpful to have access to instances with data pertaining to contractors, amount received, etc.. I would love if this could be in a grid/table.
    a. We implemented this model at first with Contractor, Contract, and State with a table for each. This was a shaky implementation so it will definitely change in future phases.

4. **Stock Market Instances:** As your client, I want to be able to access stock market data. It would be helpful to have this data in the form of a table. This would enable users to understand stock information for different politicians
    a. We added instances in the stock market as well as some basic links to politicians, displayable from clicking the model pages. The instance page contains additional attributes not displayed on the table.

5. **Links between data models:** As a client, I want to be able to link between model pages. For example, I should be able to go to a government contract from a related stock. This would allow the user to understand the connection between contracts and stocks
    a. We linked our models through States represented between companies and politicians. We feel this is an appropriate link for the data because both politicians and companies have an interest in their main states.

We were able to complete all our user stories without much hassle, though one story had to be adjusted due to our decision to omit a less-productive model from our RFP step. We all definitely gained important insight by thinking about developing or requesting these issues from the user perspective.

3. RESTful API: Postman

We documented our RESTful API based on the API calls we explored with our preliminary scraping of the data in our model sources. Other API calls reflect our plan to aggregate the data in hopes of easily accessing potential patterns areas.

For Phase II, we implemented our API solely from calls using data in our database that we structured to display our website and data. Unfortunately, this limits a lot of documented flexibility that we originally had planned when matching our various API sources, but at the same time the vast variety of data also made it impossible at times to establish internal links. All of our API calls return comprehensive information regarding each endpoint, so our Phase III consideration will be to implement query parameters to slim down the values corresponding to searching and filtering.

Government Contracts:

- Government Contracts Endpoint
    - Returns list of government contracts awarded.
- Government Contract Endpoint
    - Returns a specific government contract specified by ID.
- Government Contract Vendors Endpoint
    - Returns a list of government contract vendors.
- Government Contract Vendor Endpoint
    - Returns a specific government contract vendor as specified.

Stocks:

- Stocks Endpoint
    - Returns a specific result based on a symbol provided.
- Stock Endpoint

Campaign Finance:

- Campaign Finance Model Endpoint

    - Returns a list of campaign finance models by congress members.

Candidates:

- Candidates Endpoint
    - Returns a list of candidates.
- Candidate Endpoint
    - Returns a specific candidate as specified by ID.
- Candidate by State Endpoint
    - Returns a list of candidates by specified state.

4. Models

All our models and specific instances are rich in attributes. We have listed a few that we can potentially capitalize from our API sources, and our direction after Phase I will likely depend on what trends and links we happen to notice as we work further with scraping and storing.

Government Contracts Awarded:

- Contractors - Information about various government contracts including their industry and homestate, and other relevant states
- Amount Received- Amount of money received per contract
- Service Provided - What was provided as a condition of the contract
- Time Awarded - What time was award given out to the contractor
- State- What state the contractor's HQ is located.

Stock Market Data:

- Company - including industry and other states of operation
- Volume - the number of shares or options traded
- Long-term performance - possibly expressed as percentage over time frame
- Quarterly Info - meet, exceed, or fall behind projected expectations
- P/S value - a ratio comparing market capitalization divided by sales
- Dividends - certain stocks that distribute a portion of earnings to investors
- Recommendations - analyst ratings including strong buy, buy, sell, etc.

Campaign Finance:

- Politician - including political affiliation and represented state

- Contributors - Contributors to their financial campaign
- Amount Received - Amount received for campaign
- Time Received - When contribution was received
- Repeated Contributions - Any repeat contributions
- Election Result - Whether or not said candidate won

5. Tools

The tools we used fall into three categories: development, backend, and frontend. Our development tools allow us to collaborate both as team members as well as with other teams. Our backend tools will eventually allow the site to perform programmatically intensive tasks like dynamically requesting data from APIs and processing this aggregated data in many ways. Our frontend tools allow us to construct the ideal user experience, and every visitor to our site should be able to easily navigate our models and conveniently view the data.

Phase I:

Development:

- Postman - creates and hosts ThePoliticalMarket RESTful API Documentation
- Gitlab - issue tracking, project management, version control, continuous integration

Backend:

- Docker - gives our specific environment for the backend for any user or provider
- wsgi - allows our web server to forward requests to a Python web framework
- Flask - the lightweight Python web framework communicating with through wsgi

Frontend:

- yarn - a package manager for our modules, allows us to locally compile and develop
- React - a library to develop and render our User Interface and visual components
- React-Bootstrap - a library to further construct and stylize our frontend interface

Phase II:

Development:

- Gitlab CI - a continuous integration pipeline specifically integrated into our repository

Backend:

- Pytest - a unit testing library for all our backend Python code

- Flask-SQLAlchemy - provides a powerful coding framework for accessing a SQL database
- Flask-Restless - simplifies generation of backend RESTful API from database calls
- MySQL - a relational database management system to store all website data

Frontend:

- Jest - a unit testing library for our frontend Javascript code with React integration
- Splinter - a unit testing library for our frontend GUI and user experience

6. Database

We chose MySQL over PostgreSQL for our database mainly for simplicity reasons. Since we were all new to SQL/databases, we all found the GUI-based MySQL Workbench very useful for visualizing, organizing, and in some cases inserting the data when needed. Furthermore, since we are accessing our database in our Python backend with Flask-SQLAlchemy, a lot of the actual SQL syntax and queries are abstracted under the hood of the provided library functions and classes. Essentially all our API can be implemented with a combination of Flask-Restless and direct SQLAlchemy functionality, so the additional features of PostgreSQL are less valuable.

We also read that MySQL was more popular and arguably more appropriate for simpler web applications that mainly demanded reading and delivering data, and hence provides more of the essential strengths corresponding with our project scope.

From our backend API implementation, we can then access the data stored in our database from our frontend with the corresponding GET calls. Our model and instance components will wait for the asynchronous query to finish, upon which the data will be stored in the frontend. That data can then be used to render all entries to populate the grids and tables, as well as generate a link routing to the specific instance page. The instance page then queries the API again to obtain the data corresponding to the specific instance, including the additional attributes; all this information can then be formatted with the overall page media and structure.

7. Pagination

Currently, our implementation of pagination is one-to-one with the backend paging system of our database. For the table, each page corresponds to a fixed amount of elements directly queried from our RESTful API, with the page fields generated through Flask-Restless. We made this decision because our contract and stocks models have many entries and it is not feasible to query the entire database without some sort of backend paging. A one-to-one relationship with

the frontend table displaying all the results returned from the backend API requires the least logic and additional code to maintain.

However, this comes at the expense of requiring more overall API calls which may be a performance drawback. It may be possible to revise this in the future to query a factor of the elements-per-page and split that into pages programmatically, reducing the amount of API calls.

8. Testing

As mentioned in our toolchain regarding unit testing, we chose unittest for our Python backend, Jest for our Javascript frontend, and Splinter for our GUI interface. All three frameworks provide essential unit coverage, but we chose those specific options because they best integrate with our application and code.

We chose unittest over Pytest due to our familiarity with its syntax. Jest was the clear choice since our frontend uses React, as Jest has features that directly integrate with React components and hooks. Finally, Splinter, which uses Selenium under the hood, greatly simplifies GUI testing by wrapping everything in easy-to-understand test functionality.

Continuous Integration through Gitlab CI allows us to run our unit tests at every commit, and importantly when we are about to merge a feature into our main production branch. Our test suite will expand as more code is implemented and as more bugs are caught in order to ensure that everything we merge will be fully tested and functional. A lot of work was done to configure and optimize our CI pipeline for dependency caching, allowing the tests to run as fast as possible by minimizing installation and download latency.

9. Hosting

Our site is hosted through Amazon Web Services, with Namecheap providing the domain name and relevant redirects. Within AWS we use S3 and Cloudfront. S3 is essentially a secure container that stores all our files, assets, and source code. Cloudfront is the Content Delivery Network that is linked with S3, meaning it can securely send the necessary content from our website to users. Namecheap provides the name of our project as a domain name.

We had to link the two hosting tools together by requesting a certificate from Amazon. Then, to validate the certificate, we then had to put specific CNAME fields into our Namecheap configuration. With the valid certificate, Cloudfront and Namecheap were able to essentially connect. Once a user navigates to our Namecheap domain name, our website would be

smoothly delivered by Cloudfront. Our SSL certificate is valid for all valid domains of our website, meaning that both our frontend and backend are always able to support https.

For Phase II onwards, our deployed backend runs through an Elastic Beanstalk Application and our MySQL database is hosted through Amazon AWS RDS (Relational Database Services). The service simplifies the creation and live access of the database in the cloud, which allows us and the website to work with the same data at any time, whether through development or live production. Our Namecheap domain routes our api subdomain to our backend environment hosted in AWS Elastic Beanstalk, with various listeners to route both http and https traffic to the proper api URLs.

10. Audience and Users

Even in this initial Phase I of the project, we all noticed a difference in having software developers as our audience. Their user stories were comprehensive, and as fellow developers, we had a very clear direction on how to meet the requirements so far. They were very realistic on their expectations, which was likely due to the fact they were also working on the same components of the project. There was also a good communication pipeline between us, and we were both familiar with how to navigate the GitLab issue system as well. Approaching the end of the phase, we all certainly appreciated the clarity and contents of the user stories they provided us and the convenience of the shared GitLab platform.

In the later phases, when we start to work with our backend and implement our API, these same strengths will certainly apply plus more. Since our audience will be querying our API to get data from our project, they should be quick to notice if there is anything missing or inconsistent. User stories might get more technical, but it might also help to step back and consider other aspects of the experience from a non-developer user as well.

For Phase II, these strengths were definitely realized in the form of our user stories. Our audience of software developers had their own understanding of the work behind the project and were able to assign us stories that mentioned data collection, model/instance page generation, and API calls that kept us in technical focus.