PropXdoesWHAT

Chris Renard Dustin Huang Eder Garza Jae Lee
Kevin Kuney

1 Motivation

We want to help users become aware of laws that affect them personally, particularly those in traditionally underrepresented groups. Since many laws can have complex side effects, due to both complexity in their primary purpose and the unfortunately common practice of including unrelated changes as riders, it can be difficult to keep up with legislation being worked on or voted on that could affect your life. Marginalized groups especially can find changes that affect programs they may rely on buried in otherwise innocuous laws, or be unsure of what groups may be able to advocate for them or help them navigate changes. Our goal is to allow groups to see what recent and upcoming laws may affect them and from there find groups which can better inform them and provide routes to take action against those which would harm them.

2 User Stories

The following are the user stories gathered for Phase I:

As a user, I want to be able to identify which groups are affected by a law.
We solved this issue by mapping the laws to the affected group and displaying them within each law's page. In our API model, we set up a relationship between the laws and the affected groups. We were able to call our API and display the mappings with our client code.

As a user, I want to see the implications of a law.
We solved this issue by having a description for each law queried from our law API displayed on that law's page.

As a user, I want to see what congressmen, senators, city-council, etc. are supporting a law.

We solved this issue by mapping the laws to the sponsoring politician and displaying them within each law's page. In our API model, we set up a relationship between the laws and the policitians involved. We were then able to call our API and display the mappings with our client code.

As a user, I want to see where the politicians lie on the political spectrum.

We solved this issue by pulling the politician's party affiliation from our API and displaying it in our client code.

As a user, I want to know how I can contact my senator/representative.

We solved this issue by displaying each politician's bio, which includes their website and number, by querying the relevant contact information from our API and displaying it on the politician's page.

The following are the user stories gathered for Phase II:

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

We solved this issue by displaying the purpose statement of the website in the homepage.

As a user, I would like to see more laws, politicians associated with these laws, action groups, and affected groups.

We resolved this user story by having the front-end gather the data from the API and displaying it. Our models are linked with foreign keys creating relationships with each other. We have fully loaded the laws and politicians models with data from the 115th congress, and the action groups model with 515 groups.

As a user, I would like each image on the front page carousel to lead me to a certain page.

We solved this issue by making each image a link to various pages.

As a user, I would like an easy way to contact a certain action group.

We solved this issue by displaying the website link for each action group on the model page.

As a user, I would like the affected group's website pages to have a little more info (do a little more than just list laws affecting group and action groups associated)
We solved this issue by displaying more than just laws and action groups related to this affected group.

The following are the user stories gathered for Phase III:

As a user, for the laws model, I want to be able to filter by subject.
We implemented this user story by creating a subject filter option with selections derived from laws in our dataset.

As a user, for the politicians model, I want to be able to filter by Legislator's state.
We implemented this user story by creating a filter by state option for the Politician model page.

As a user, I would like to search for laws, politicians, and action groups from the home page.
We implemented a search bar that triggers full-text searching. We also have full-text search within our models that takes into account any filters applied.

As a user, for the politicians model, I want to be able to filter by republican or democrat.
We implemented this user story by creating a filter by party option for the Politician model page.

As a user, for the laws model, I want to be able to sort by date introduced
We implemented this user story by creating a sort by date introduced option for the Law model page.

3 Models

Each politician model shows each law and action group associated with them. Each law shows each politician and action group associated with them. Each affected group shows each politician and law associated with them. These associations lead to their respective model page. Each action group shows each law that affects the group.

Laws
- Name
- Title
- Description
- Sponsor
- Affected Groups (incoming table relation) (not built)
- Primary Subject
- Congress.gov Link
- GovTrack Link
- Date Introduced
- Date of last vote (if occurred)
- Date bill passed Senate (if occurred)
- Date bill passed House (if occurred)
- Date bill was vetoed (if occurred)
- Date bill was enacted (if occurred)

Politicians
- First Name
- Last Name
- State
- Party Affiliation
- Chamber (House or Senate)
- Phone
- Link to official site
- Link to contact page (if present)

Affected Groups
- Name
- Description

Action Groups
- Name
- Description
- Type/Category
- Link to official site
- Assisted groups (incoming table relation) (not built)
- Statements on laws (incoming table relation, text) (not built)

4 RESTful API

These are the API links we pulled from to fill our database:

- docs.openstates.org/en/latest/api/
- projects.propublica.org/api-docs/congress-api/
- opensecrets.org/open-data/api
- http://www.startguide.org/orgs/orgs00.html

API Endpoints:

- api.propxdoeswhat.me/api/politicians
- api.propxdoeswhat.me/api/affected_groups
- api.propxdoeswhat.me/api/action_groups
- api.propxdoeswhat.me/api/laws

Full API Documentation at:
documenter.getpostman.com/view/4704075/RWMCtUm6

5 Tools

GitLab Git repository hosting, issue tracking, continuous integration
Postman RESTful API testing, development, documentation
Grammarly Spelling/grammar feedback for this report
Piazza Collecting User Stories from end users
Slack Team communication, coordination
Flask Back-End framework, along with SQL-Alchemy and Flask Restless
React Front-End server-side framework, along with React Router
AWS Server hosting for Flask, Node, MySQL, ElasticSearch instances
Selenium GUI testing framework
Mocha Test framework for JavaScript
Chai Assertion library paired with JavaScript testing framework
Docker Virtual system images for front-end and back-end configuration
Enzyme React testing framework
Bootstrap Front-end client-side framework
ElasticSearch Standalone full-text index server based on Lucene

6 Hosting

6.1 Back-End

The back-end is hosted on an AWS EC2 instance. The main file is `Back-end/app.py` running in a docker container on the server. The back-end uses flask, sqlalchemy and flask-restless to create an API hosted at `api.propxdoeswhat.me/api/`.

6.2 Front-End

Also hosted on an AWS EC2 instance, the main file is `frontend/src/app.jsx` running in a docker container on the server. The front-end uses React to build a client page which uses Bootstap to get data from our back-end and serve it to the client. It is hosted at `propxdoeswhat.me`.

7 AWS

First, we launched an EC2 instance with Amazon Linux AMI. Since by default all incoming ports are blocked we added security group rules that allow incoming SSH and HTTP requests from anywhere. Next, we SSH'd into the EC2 instance with the private key file we were given when we launched the instance, the username `EC2-user`, and the public IP address of the instance.

Frontend access:
`ssh -i front-end-private-key.pem ec2-user@propxdoeswhat.me`
Backend access:
`ssh -i back-end-private-key.pem ec2-user@api.propxdoeswhat.me`

Next, we updated the working Linux server running in the AWS cloud and installed docker on it. Then, we transferred our local files onto the server using SFTP. Finally, we built our Docker image and ran our web application on our Docker container.

8 Sort

We allow the users to sort the displayed results by ascending or descending order using the API parameter `order_by`. The Laws model allows sorting by the date introduced. The Politicians model allows sorting by last name. The Action Groups model allows sorting by type.

9 Full-text Search

We use an ElasticSearch instance running on AWS as an external server to host the full-text indexes and perform searches. Indexes are built after database tables are loaded by scripts which dump the tables and load the indexes. Currently, fields are indexed as-is with no stemming or other prossessing. The Flask-Restless API manager was modified to first recognize a new parameter `search` (using a request pre-processor handler), call the ElasticSearch instance with the given query, and then adjust the query filter for the rest of that request to only include results with the returned ids (using a custom `query` model field). If there is no `order_by` parameter, the query also imposes the relevency ordering returned by ES.

10 Filter

The website implements filtering options for each of our models. The laws models filters based on the API parameter `subject`, the politicians models filters based on `party`, `state`, and `chamber`, and the action group model filters by `type`. Whenever a user clicks on a filter on our page, we `GET` a JSON response and generate the respective model with the applied filters. If another filter gets applied from the same filter category, we OR the two filters and display the results. If another filter gets applied from a different filter category, we AND the two filters.

11 Pagination

Our database contains many (hundreds to thousands) instances of laws, politicians and action groups. To give the user a better experience navigating through our data, our API returns paginated subsets of requested data to the front-end client which then allows for navigation via links to previous page, next page, and nearby pages.

12 Database

We use a MySQL 5.6 instance running on Amazon RDS. The Amazon default settings use `latin1` for text (this is bad), we had to manually adjust them to use `utf8mb4` and rebuild tables. Note: `utf8` in MySQL does not meet the utf8 spec, as it will not work with astral glyphs (four-byte characters). While this is unlikely to occur in our dataset, for future-proofing we use `utf8mb4` which correctly implements the standard.

Table: `'laws'`

- `'id' INT UNSIGNED NOT NULL AUTO_INCREMENT` – internal id, auto-generated
- `'bill_id' VARCHAR(32) NOT NULL` – unique bill name across legislatures, ie: hr5515-115
- `'name' VARCHAR(32) NOT NULL` – name of bill, ie: H.R.5515
- `'title' VARCHAR(256) NOT NULL` – short title of bill
- `'subject' VARCHAR(256) NOT NULL` – primary subject of bill
- `'sponsor_id' INT UNSIGNED NOT NULL` – foreign key to 'politicians'
- `'sponsor_bio_id' VARCHAR(16) NOT NULL` – Biography of Congress unique id for sponsor, ie: T000238
- `'cdotgov_url' VARCHAR(256),` – url of bill on congress.gov
- `'govtrack_url' VARCHAR(256),` – url of bill on govtrack
- `'introduced' DATE`
- `'last_vote' DATE`
- `'house_pass' DATE`
- `'senate_pass' DATE`
- `'enacted' DATE`
- `'vetoed' DATE`
- `'desc' TEXT` – propublica summary
- `'raw' TEXT` – raw json from source for this entry

Table: `'politicians'`

- `'id' INT UNSIGNED NOT NULL AUTO_INCREMENT` – internal id, auto-generated
- `'first_name' VARCHAR(64) NOT NULL`
- `'last_name' VARCHAR(64) NOT NULL`
- `'dob' DATE NOT NULL` – date of birth
- `'bio_id' VARCHAR(16) NOT NULL` – Biography of Congress unique id, ie: T000238
- `'chamber' ENUM('house', 'senate') NOT NULL`
- `'state' CHAR(2) NOT NULL`
- `'party' ENUM('R', 'D', 'I') NOT NULL`
- `'site' VARCHAR(256)` – url for official site
- `'contact_form' VARCHAR(256)` – url for official contact form
- `'phone' VARCHAR(32)`
- `'raw' TEXT` – raw json from source for this entry

Table: `'action_groups'`

- `'id' INT UNSIGNED NOT NULL AUTO_INCREMENT` – internal id, auto-generated
- `'name' VARCHAR(256) NOT NULL`
- `'url' VARCHAR(256) NOT NULL` – url of official site
- `'type' VARCHAR(256) NOT NULL` – startguide.org category of group
- `'desc' TEXT`

13 Testing

We are using Selenium Python to test the GUI of our website. We installed Selenium with pip and downloaded the required interface driver for the Firefox browser (`geckodriver`), and then added `geckodriver.exe` in the system PATH. This let us use instances of Firefox WebDriver in our Python unit test to navigate around our website and test our GUI.

Since we are using Flask-Restless and SQLAlchemy to create our RESTFul API, we did not test our back-end because those tools handle it.

We are using Postman for API testing. The Postman test requests GET methods to our API and we make sure that all the fields we need in the database are present and have correct number of pages.

We are using Mocha, Chai, and Enzyme to test our javascript. We installed Mocha, Chai, Enzyme, `babel-preset-env`, `babel-core`, `babel-preset-react`, and `react` in `npm` and ran `mocha` with arguments `-require babel-core/register`.

14 Obstacles

In Phase III of our IDB project we ran into a significant problem when we were trying to implement filter, sort and search. We had an issue with our code because of the way we set up our front-end in Phase I and Phase II and had to scrap much of the front-end code and refactor for Phase III. Although Phase III was supposed to be a lighter workload, this was not the case for us. We turned in our project a day late to ensure we had a working and well developed website.