Motivation

The motivation behind our project was to educate people about something that affects us all - politics. Once we knew we wanted politicians to be one of our models, we determined the other two should be states and news sources. States correlate well with politicians and the news is how we keep up with them. We want our users to be able to educate themselves about not only their state and representatives, but any of them that peak their curiosity. We are designing our website to be user friendly so that anyone can use it. A better understanding of our nation and the people running it will lead to a better present and an even greater future.

User Stories

Phase I User Stories

- "**More horizontal state instance page**: As a user, I would like to see the state pages be formatted more horizontally. Right now the state instance pages' information are stacked vertically, which is a bit hard for the user to read. It could have, for example, have multiple images on around the same level, have the state flag on the top left hand side, the state senators on the top right hand side, and the state map beneath the two. Any way is fine, right now it is a little difficult to read."
    - This was implemented by using cards in rows to display state information.
- "**Detailed News Model Page**: As a user, I would like to see more info about each news source in the grid, rather than just a logo. This would be helpful in quickly gathering more info about a news source before deciding whether to investigate it further or not."
    - This was implemented by adding additional data, such as "year founded" and "number of employees" to each news source in the grid.
- "**Democrat/Republican color coordination**: As a user, I would like to have some sort of color coordination for parties for the politician model page so that users are more prepared for what they will click on. For example, you could have a party column in your table that has a red square/text for republicans and a blue one for democrats (and something else for others). I think that will really help with clarity for distinguishing politicians, and would be useful to sort by in the future."
    - This was implemented by adding an image for each party in the table.
- "**More Welcoming Splash Page**: Currently when going on the website, the vibe doesn't seem very welcoming or have a political theme besides the huge picture

of Ted Cruz. As a user I'd like to see A nicer theme since this is the first impression the user will get."
    - This was implemented by adding different images, cards, and links to the Splash page.
- "**Twitter feed for politicians**: As a user of your website, I would like to see a Twitter feed for each politician or even a few tweets (such as most liked, most retweeted, etc.). As of now, the politician pages display only the number of tweets, which doesn't seem helpful. I think the politician tweets would be an informative feature for the page."
    - This was implemented by using Reacted Twitter Embed Component, which allowed for a twitter feed for each politician as well as a link to "tweet at" them.

Phase I Customer Stories

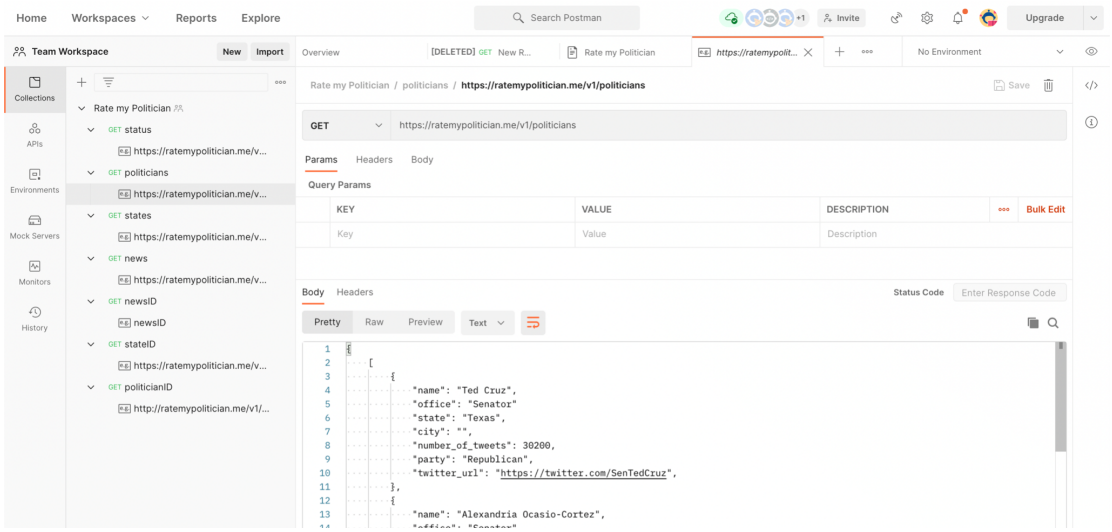
- "**Collect Data on Many Instances of Each Model**: As a user, I would like more than 3 instances of each model. These instances should collect data from sources with a RESTful API."
- "**Style/Color of website**: As your client, I want the website to have more vibrant colors rather than a darker tone. 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."
- "**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."
- "**Campaign Finances 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."
- "**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."

REST API

The design of our RESTful API was done using Postman, a tool that provides testing and documentation capabilities for APIs. For each endpoint that we anticipate implementing in our RESTful API, we defined a number of attributes, including:

- The endpoint name
- The endpoint url
- A description of the purpose of the endpoint
- An example request
- An example response

Editing a example response:



In order to document the API collaboratively, a shared team was created in Postman. In this shared workspace, we then created a collection called “Rate My Politician” to contain the endpoints. We then were able to add each endpoint and its information as well as a sample request. The following endpoints were documented:

- All politicians
    - This endpoint returns a json response with all politicians in the database. It is used for rendering the politicians model page.
    - https://ratemypolitician.me/api/politicians
- All states
    - This endpoint returns a json response with all states in the database. It is used for rendering the states model page.
    - https://ratemypolitician.me/api/states
- All news sources
    - This endpoint returns a json response with all news sources in the database. It is used for rendering the news sources model page.

    - https://ratemypolitician.me/api/sources
- A single politician by id
    - This endpoint returns a json response with a single politician. It is used to render single instances of politicians.
    - https://ratemypolitician.me/api/politicians/<id>
- A single state by id
    - This endpoint returns a json response with a single state. It is used to render single instances of states.
    - https://ratemypolitician.me/api/states/<id>
- A single news source by id
    - This endpoint returns a json response with a single news source. It is used to render single instances of news sources.
    - https://ratemypolitician.me/api/sources/<id>
- A single politician by name
    - This endpoint returns a json response with a single politician. It is used to query the backend for a politician.
    - https://ratemypolitician.me/api/politicians/<name>
- A single state by name
    - This endpoint returns a json response with a single state. It is used to query the backend for a state.
    - https://ratemypolitician.me/api/states/<name>
- A single news source by name
    - This endpoint returns a json response with a single news source. It is used to query the backend for a news source.
    - https://ratemypolitician.me/api/sources/<name>

The api documentation can be found at this link:
https://documenter.getpostman.com/view/12075941/Tz5jeLVC

Models

Our Project #2 agenda is to create a website where users can learn about how various news sources and social media cover politicians in the US. To account for the social media/news aspect, we decided to combine a news and twitter apis to form a News Sources Model. Politicians are a key factor for the website goal, so we decided on a Politicians Model based on a civic information api to create a database of US politicians. Lastly, since every politician is affiliated with a state, we decided on a State Model so users can keep up with politicians in their state and keep up with news stories about political figures prevalent to their everyday lives.

The Structure of the Models are as follows:

News Sources Model:

Instance: News Sources that cover politics

Attributes:

- Name
- Logo
- Number of Employees
- Year Founded
- City
- State
- Alexa company rank
- Twitter bio
- Twitter followers
- Organization type
- Description

APIs: https://newsapi.org
https://bigpicture.io/docs/enrichment/company/

Model Page Layout: The model page is designed so that the news logos are visible in a grid. Clicking on the image will lead to the individual instances pages for each news source. This layout was chosen because most users will recognize news sources by their logos before their names.

Instance Page Layout: Currently, each instance page contains the New Source instance's name, logo, and list of attributes.

Politicians Model:

Instance: US Politicians

Attributes:

- Name
- State
- Political party
- Position
- Number of tweets
- Twitter url
- Number of articles
- Phone number
- Email address

- Website url

API: https://developers.google.com/civic-information

Model Page Layout: The model page is designed so the user sees a large table, where the rows are politicians. The information given for each politician includes their headshot, first name, last name, and link to their respective instance pages. This design was chosen because many people only know either the position title, name, or image of a politician but it is unlikely that they will know all three. So, all information is provided so they can quickly identify who they're looking for based on what they know.

Instance Page Layout: Currently, each instance page contains the politician's name, headshot, link to their social media, and a list of attributes.

States Model:

Instance: US States

Attributes:

- Population
- Name
- 2016 election result
- Median income
- Poverty rate
- Governor name
- How many representatives
- Median age
- Median housing price
- Senator names
- State Flag
- State Map
- Articles about the state
- List of senators and governor

API: https://datausa.io/about/api/

Model Page: The model page is designed so that the user sees cards assigned to each state in a grid. Each card contains the state's name, map, attributes, and link leading the user to the states' instance page. This design was chosen because most people associate an image of a state with the title, so by having both information available to identify the instance, the page becomes more interactive.

Instance Pages: Currently, each instance page contains the state's name, flag, seal, it's attributes, images of the senators and governor from the state, and current news articles.

Connections Between Models:

- News sources have articles on politicians and cover different states
- Politicians each represent and are affiliated with particular states and have social media accounts
- States are represented by both politicians and news sources

Tools

In phase I, the tools used were mainly for the development of the static site and the design of the RESTful API. Postman was used for the API design as described in the REST API section above.

Create React App

For building the static site, a number of tools were used to create a frontend client. Create React App, a toolchain aimed at creating and managing React applications, was used to create the frontend client. In order to create the application, the following script was run:

```
npx create-react-app rmp-frontend
```

React Router

For each page in the static site, a new React component was created, with some pages rendering multiple React components. In order to route between different pages in the site, the React Router library was used which allows the programmer to define routes and create links to those routes across pages. For example, a portion of the routing mechanism is displayed below:

```
<Switch>
     <Route path="/about" component={About} />
     <Route path="/news" component={News} />
     ...
</Switch>
```

React Bootstrap

In order to handle styles, the React Bootstrap package was used. This package allows the programmer to import pre-styled components to easily build a consistently styled site. Examples of components provided by React Bootstrap used in the static site include buttons, padded containers, tables, cards, image carousels, etc. Using React Bootstrap components is demonstrated below in the Header component:

```
import NavBar from 'react-bootstrap/Navbar';
import Nav from 'react-bootstrap/Nav';
import { Link } from 'react-router-dom';

const Header = () => {
    return (
        <NavBar bg="dark" variant="dark">
            <NavBar.Brand as={Link} to="/">Rate My Politician</NavBar.Brand>
            <Nav>
                <Nav.Link as={Link} to="/about">About</Nav.Link>
                <Nav.Link as={Link} to="/news">News Sources</Nav.Link>
                <Nav.Link as={Link} to="/State">States</Nav.Link>
                <Nav.Link as={Link} to="/politicians">Politicians</Nav.Link>
            </Nav>
        </NavBar>
    );
};

export default Header;
```

Jupyter Lab, Pandas

In order to scrape and process the data, Jupyter Lab, a web interface for working with Jupyter notebooks was used. This enables us to run python code flexibly and scrape data from the numerous APIs and process it. Pandas, a python package for dealing with data tables, was also used. This enabled us to make uniform tables to store the data scraped from various APIs and format them to be imported to the database.

Flask

To write the backend server, Flask, a python framework, was used to set up a simple server and define the various endpoints that compose our REST API. In order to

access the database, a connection was defined using the SQL Alchemy connection library for flask. Then, schema were defined for each model using another package called Marshmallow, which allows you to create schema with a defined set of fields. Once this was done, the endpoints were defined and for each one, data from the database was queried, either all the data for a model or a specific instance, filtered using a variable provided in the request url. Below is an example of an endpoint being defined in flask:

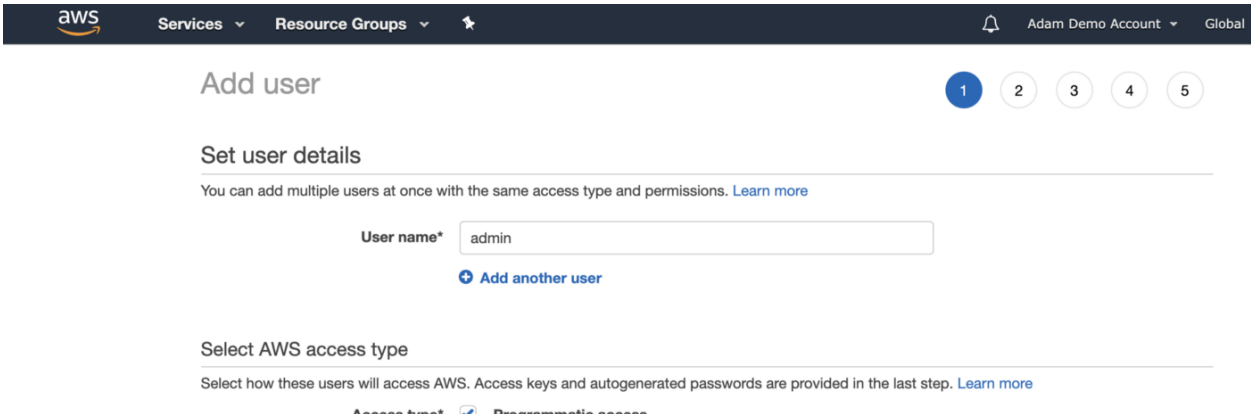
```
#Get all States
@app.route('/api/states', methods=['GET'])
def get_states():
   all_states = States.query.all()
   result = states_schema.dump(all_states)
   return jsonify(result)
```

Hosting

We hosted our website by following the tutorial given at: [How to deploy a website on AWS with Docker, Flask, & React from scratch]
However, we did a couple of things a little different so I'll give a more elaborate view of what we did below.

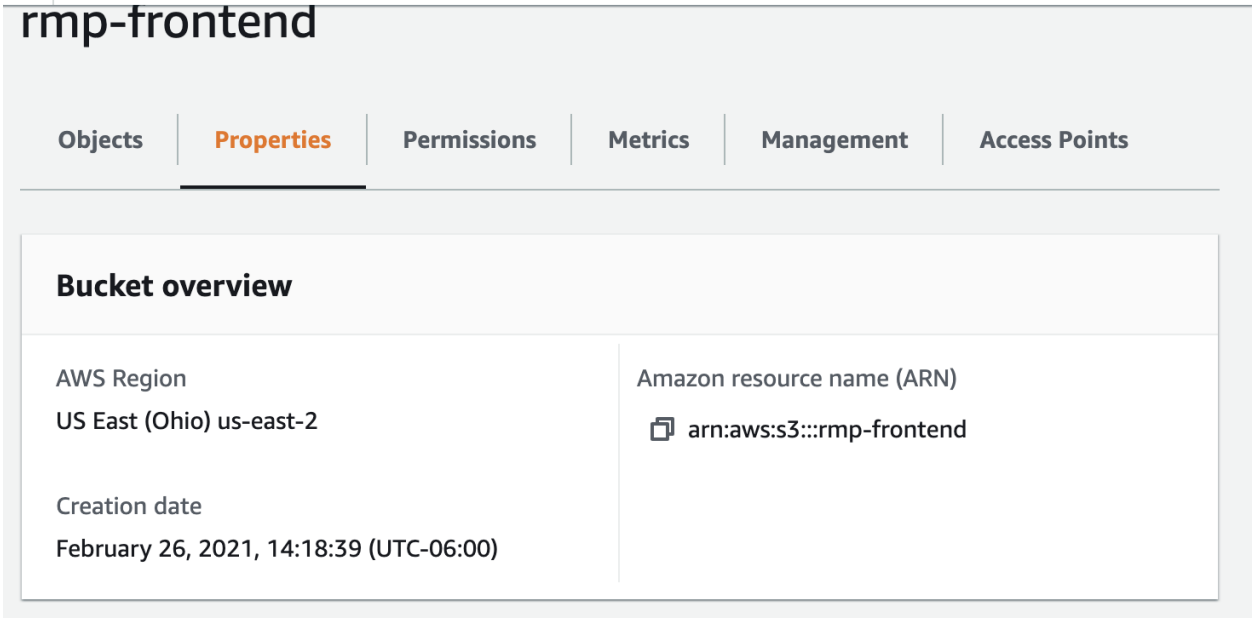
1
We decided to host our website using AWS and we started by creating an account with [Amazon Web Services (AWS) - Cloud Computing Services].
We then downloaded the aws command line interface which you can find a guide for here : [Installing, updating, and uninstalling the AWS CLI version 2 - AWS Command Line Interface]

2
The next step we did was create an AWS user which we did by searching up IAM and going to Users under access management and selecting Add user.



Once we went through and reached the Create Group page we chose the one with the Policy name “AdministratorAccess” with the type as Job Function. We stopped once we successfully reached the Success page.
We then configured AWS by typing **aws configure** in the terminal and following the given instructions.

3
Next was the deployment of our front end and the first thing we did was create an S3 bucket with AWS to store our frontend files.



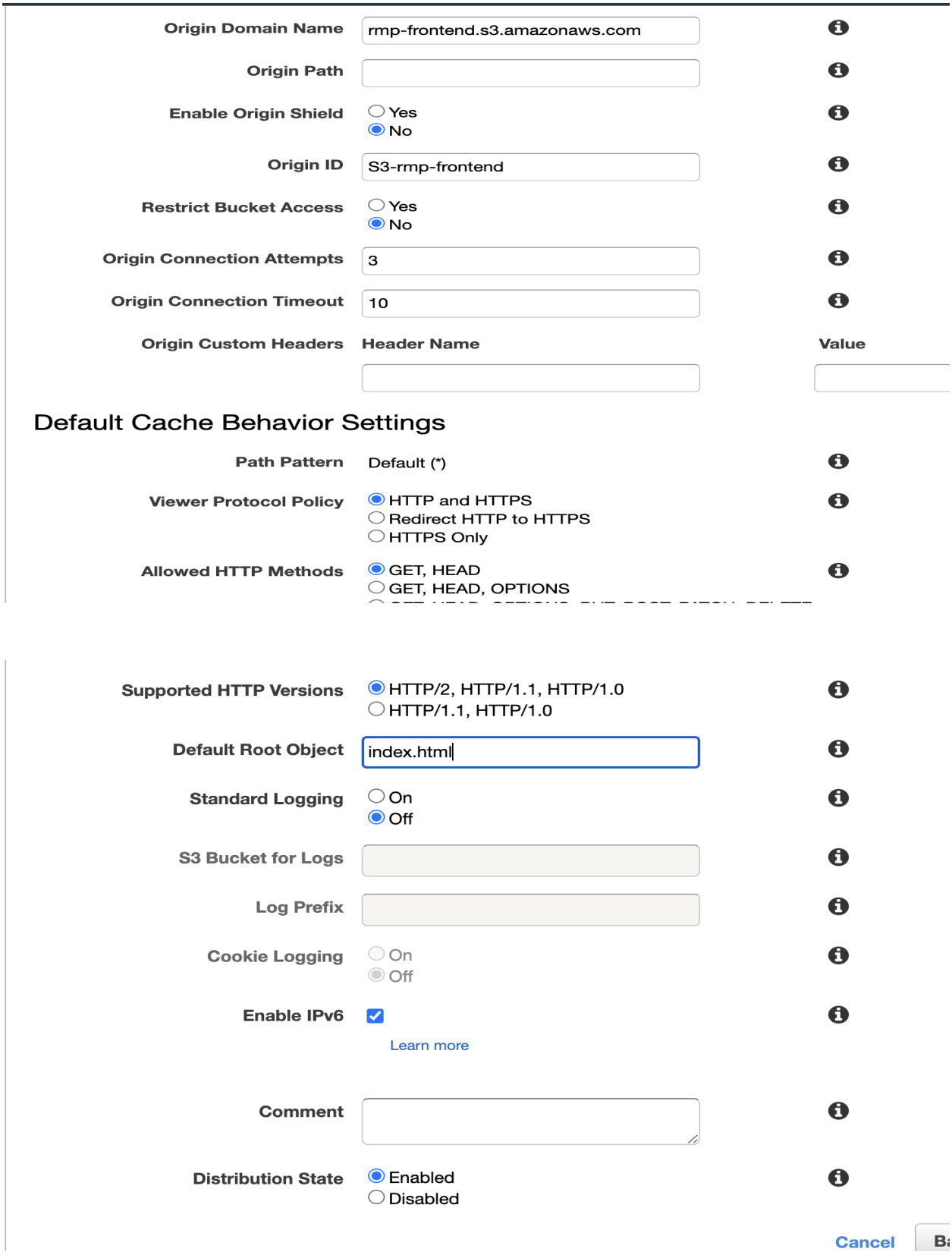
Everything in the bucket had to be public initially. Next we actually built the frontend and pushed it to our bucket using the following commands:

```
cd rmp-frontend
npm run build
aws s3 sync build/ s3://rmp-frontend --acl public-read
```

We also made sure all the files were public so we wouldn’t run into any issues.

4
Then we finally created a cloudfront CDN deployment. We started by creating a distribution in CloudFront and we linked our S3 bucket and set the origin path to /index.html.

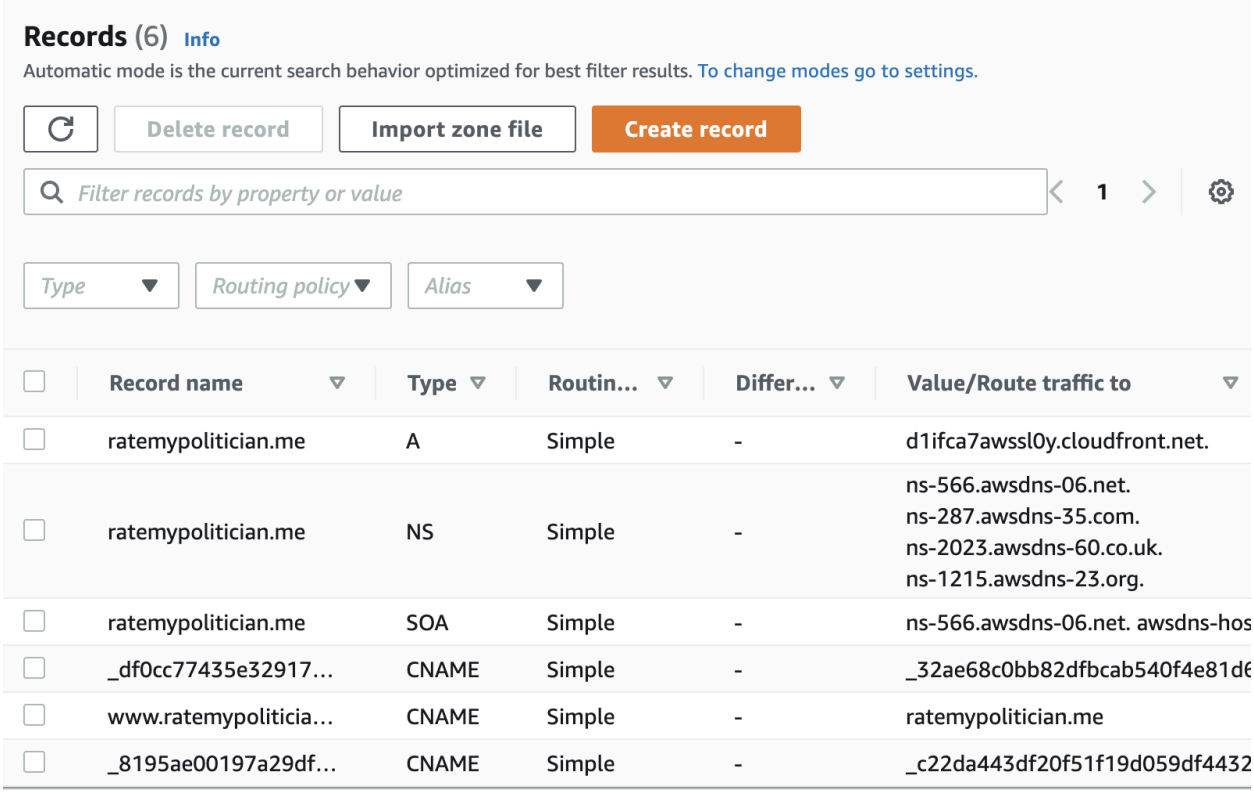
The steps to make the CDN are shown on the next page.



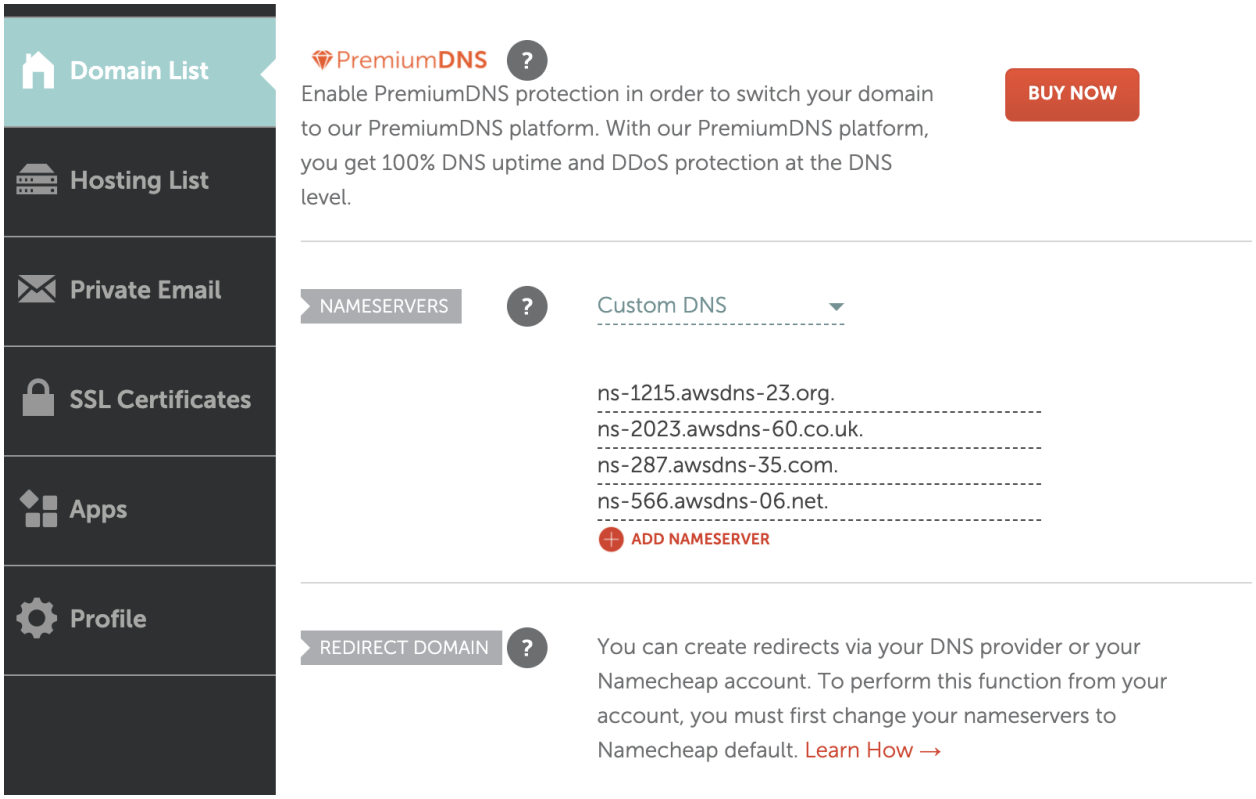
Once the distribution was done processing we confirmed everything was working by clicking on the domain name which you can find under your CDN distribution under the general tab.

5

The final step we had to host our website was to add our custom domain name. The first step was to find the domain which we found at [Namecheap: Buy domain name - Cheap domain names from $1.37]. Once we had the domain name we wanted to use we had to go and create a Hosted zone on Route 53 using the domain name we got from Namecheap.

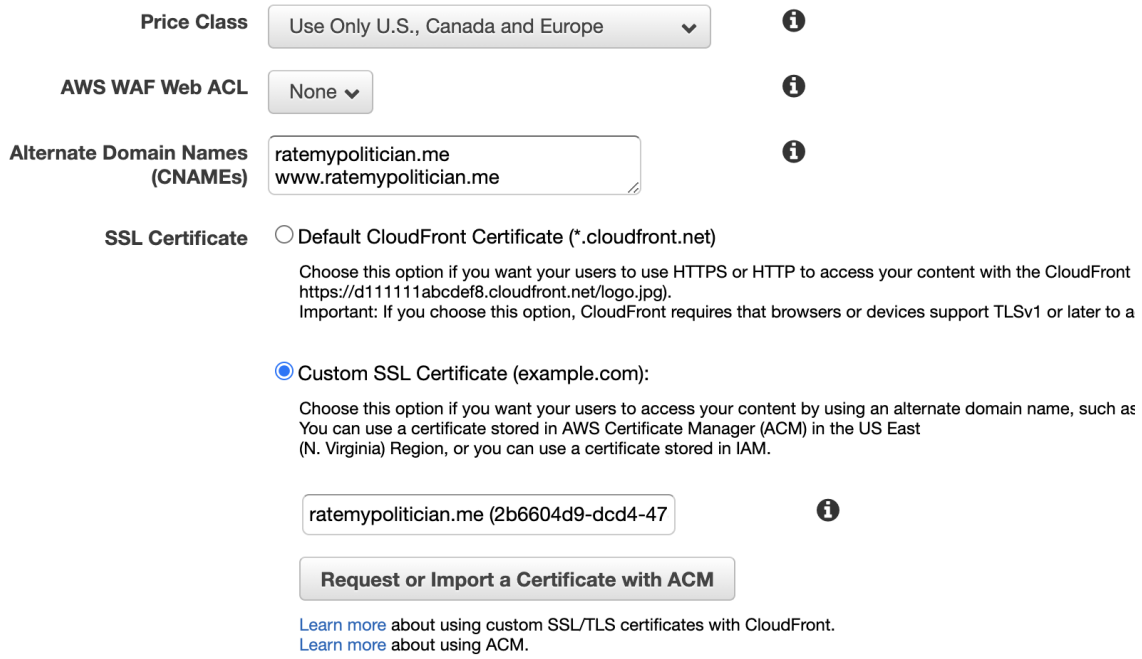


Now in a new tab we went to Namecheap and under the Domain List tab we went to Nameservers and chose Custom DNS and added our NS records from Route 53.

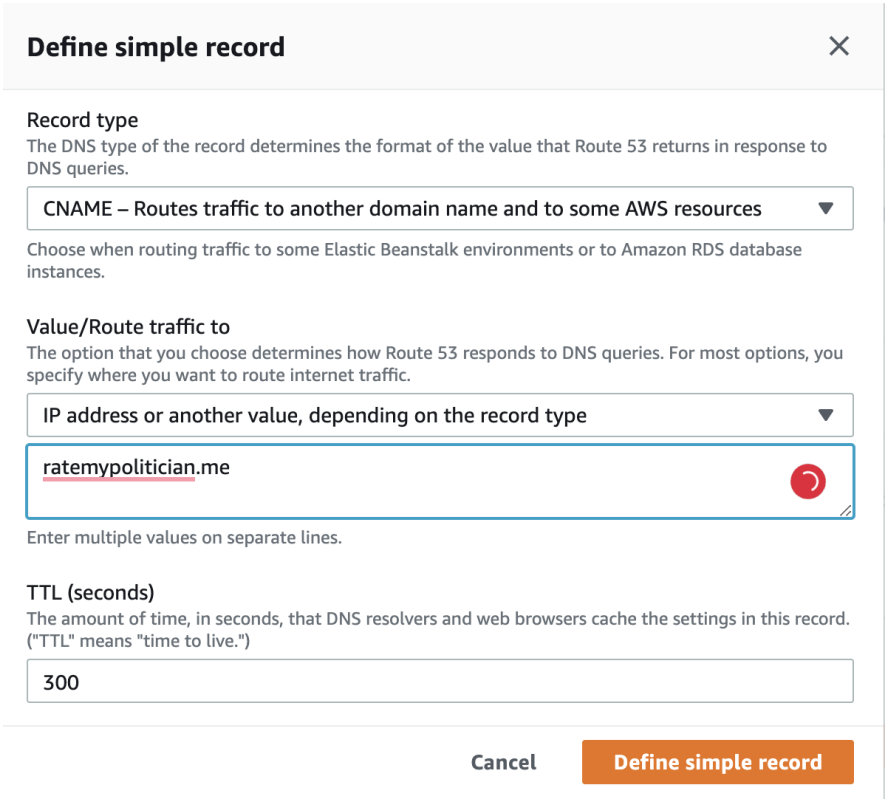


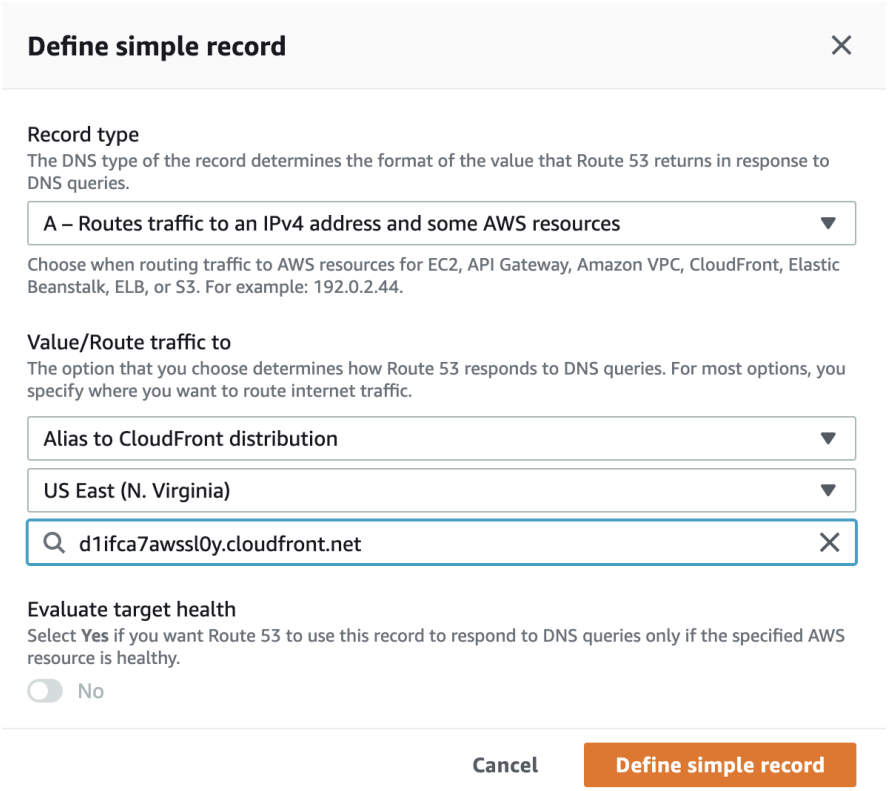
From here, we went back to CloudFront and we clicked the edit distribution option and added our new domain names under alternate domain names(CNAMES). Then we had to request a certificate with ACM as the field will initially be blank and you must wait for it to be validated.

- To do this you select the DNS Validation and choose the option to create a record for the alternate names you added.



Finally on Route 53 we created two new records which were an A record and a CNAME record.

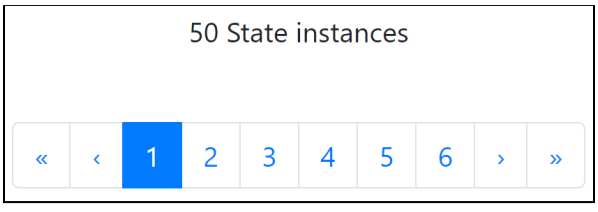




With this both https://ratemypolitician.me and https://www.ratemypolitician.me worked.

Pagination

We were able to implement pagination by using the react-bootstrap pagination feature in each of our model pages. For our States and News Sources pages, their instances are displayed in a grid. While our Politicians page has its sources displayed in a table. Above the bar is a feature informing the user of how many instances there are currently for each model. The resulting navigation bar looks as follows:



The pagination navigation bar was designated to have the following routes:

- **<<** (First) - sending the user to page 1
- **<** (Prev) - sending the user to the page one before the one they are currently on
- **1...n** (List of Page Numbers) - Here we placed all pages available to navigate to in an array so the user can target a specific page

- **>** (Next) - sending the user to the page one after the one they are currently on
- **>>** (Last) - sending the user to the very final page generated

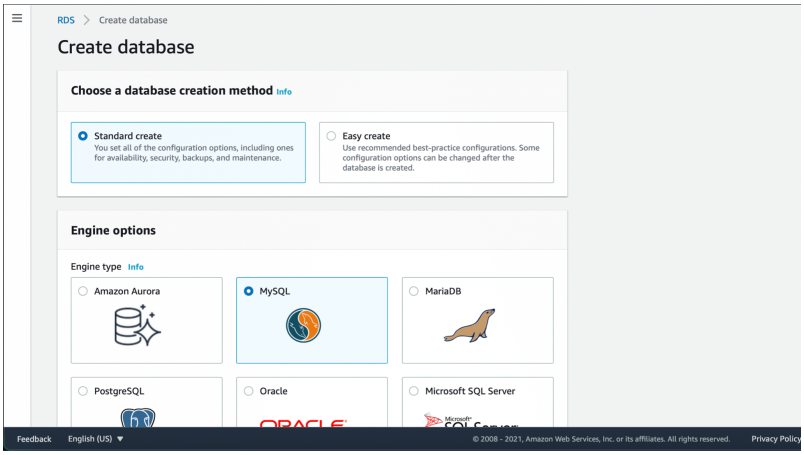
A special case we had to make for the Politicians page was shortening the list of page numbers since the large number of instances generated 86 pages. So, we added in a "**...**" option on the pagination navigation bar after 7 pages.

The Resulting Statistics for Pagination

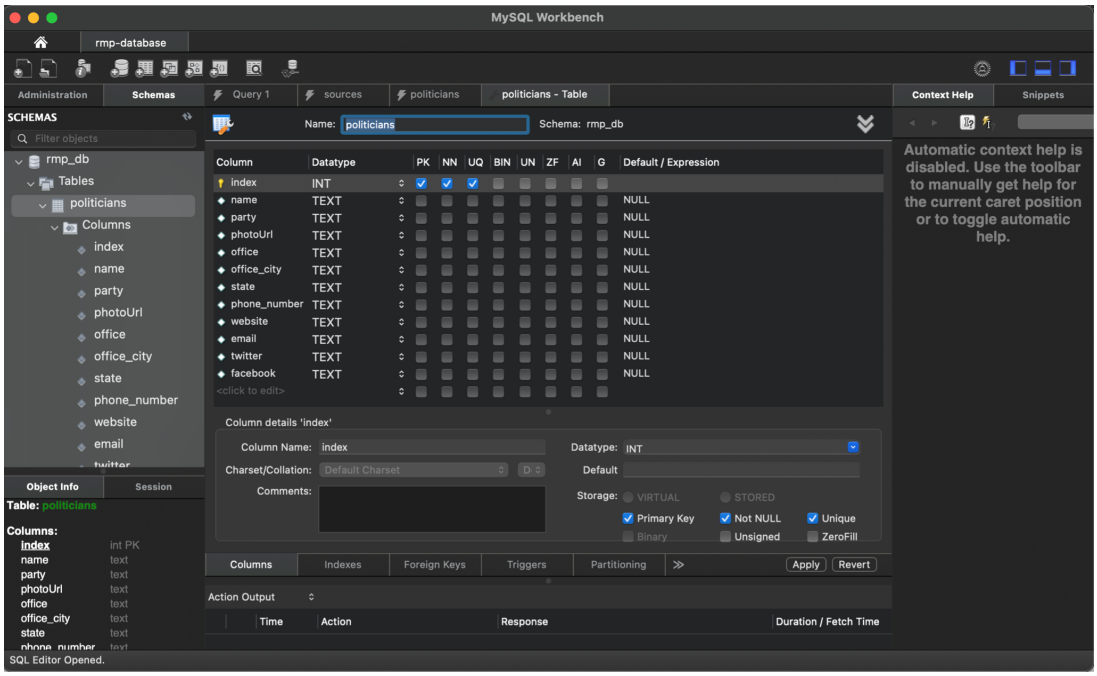
Model Page	Number of Instances per Page	Number of Pages
News Sources	9	12
States	9	6
Politicians	100	86

Database

The database used by the backend to store and retrieve information is a MySQL database hosted on the RDS service on AWS. This service allows you to host a database in the cloud and connect to it from your backend server. The database was set up through this portal in the image on the right. A security profile setting was also modified to allow access from external sources, such as the database client and the backend server.



Once the database was running, we used a database client, MySQL Workbench to load in the data to the database. After having scraped the data and stored it in csv files for each model, the csv data was added to the database as new tables. The column names for each table could then be modified to match the backend. Below is a view of MySQL workbench accessing the metadata for the politicians table:



Testing

Our testing was implemented in four different parts.
We created unit tests for the RESTful API using Postman, unit tests of the JavaScript using Jest, acceptance tests of the GUI using Selenium, and unit tests of the Python code using unittest. Creating the unit tests for JavaScript using Jest was on the easier side to implement and there is an example of it below. We used this to test our frontend functionality and code.

```
import "@testing-library/jest-dom/extend-expect";
import { render } from "@testing-library/react";
import { BrowserRouter } from "react-router-dom";
import Splash from "../components/Splash/Splash";

test("Splash page renders", () => {
    const component = render(
        <BrowserRouter>
            <Splash />
        </BrowserRouter>
    );

    const element = component.getByText("The REAL Truth");
    expect(element).toBeDefined();
});
```

We used unittest for the tests for the backend and the GUI tests. It is a tool that's commonly used for unit testing for Python as the name would suggest. We see an

example of it being shown below for the backend.

```python
import unittest
import json

import app

class APITests(unittest.TestCase):

    def test_get_states1(self):
        all_states = json.loads(app.get_states().data)
        self.assertEqual(len(all_states), 50)

    def test_get_states2(self):
        all_states = json.loads(app.get_states().data)
        fields = ['data_year', 'dem_margin', 'governor_name', 'index', 'landscape_background_url',
        for state in all_states:
            keys = list(state.keys())
            self.assertEqual(keys, fields)

    def test_get_state1(self):
        state = json.loads(app.get_state(0).data)
        self.assertEqual(state['name'], 'Alabama')
```

The unit tests for the RESTful API using Postman was also a straight forward implementation and the documentation for it is found online: Automated API Testing

However, one part of the testing we struggled with was the GUI testing specifically getting Selenium working. We will go over how we did these tests more thoroughly below.

Selenium

To get started with testing for selenium, we had to first download a couple of dependencies.

- pip install selenium
- brew install --cask chromedriver (For MacOS)

Once we got the initial dependencies setup, we needed to download a standalone selenium server for us to be able to run the tests locally. The code used to run the tests locally and on the pipeline are different as you can see in the example below along with a few GUI tests.

```python
import unittest
import time
from selenium import webdriver
#from selenium.webdriver.common.desired_capabilities import DesiredCapabilities

class GUITests(unittest.TestCase):
    def setUp(self):
        # create a new remote Chrome session
        self.driver = webdriver.Remote(command_executor="http://selenium__standalone-chrome:4444/wd/hub",
        #For running locally
        # self.driver = webdriver.Remote(
        #     command_executor='http://127.0.0.1:4444/wd/hub',
        #     desired_capabilities=DesiredCapabilities.CHROME)
        self.driver.implicitly_wait(10)
        self.driver.maximize_window()

        # navigate to the application home page
        self.driver.get("http://www.ratemypolitician.me/")

    def tearDown(self):
        self.driver.quit()
```

We have to create a Remote webdriver so that we can run these tests in the pipeline. We use these tests to see if we can find certain elements on our webpage. For example, we check to see if when we navigate to the states page we can find the More Info button.