

Demo

<https://www.loom.com/share/253dd2c2a2b4401c8f910286d4c2ced2>

Github

https://github.com/yhtoon/senior_project

Part 2. Spark Data Processing and store in MySQL

Requirements

Prior to beginning our Spark processing we looked into all inconsistencies that existed in the data. This helped us determine what we needed to preprocess, process, and post-process.

For preprocessing, we had to handle null values for all fields for each table downloaded. We also had to fix zip-codes to be able to geocode using the Google API. Most zip codes were nine digits long, which needed to be cut to the first five in order to successfully geocode. We also created a hashmap for state names and their respective abbreviations since Google API can geocode “Arkansas” but not “AK”, for example. This preprocessing helped us speed up our overall processing since inconsistencies were thrown out early on.

We then decided to process by table type and map (for heatmap) with the necessary keys to parse data more quickly. These tables were stored in DynamoDB. Our postprocessing was strictly used for setting weights for the coordinates we generated for each donation using the Normalization Formula. Then, we processed geolocations, and tables that are nested and created necessary keys and tables and exported them to DynamoDB where we stored our necessary tables.

Design and Implementation

Nested tables

In order to store all the required information for each candidate and committee, we found it necessary to nest tables. We needed to display a table for each key (candidate, committee, and custom keys). To do this, we grouped all the essential information by ID(either candidate or committee) and outer joined the information with distinct IDs by ID.

Custom Key

One problem we faced was that it is not optimal to use filters on an API call to “search” for a candidate in a table with each candidate in their own row. This is because DynamoDB runs a “scan” operation, which is $O(n)$ and takes ~35 seconds to go through 8K+ candidates. This is costly and slow. As a solution, we concatenated the main three filters (office, senate, and house) into a partition key in which each key describes an office. This way, a “search” is a query that takes constant time (99.9981617647% less costly) and returns a list of all candidates needed for the table.

The other application is an optimization for our competitor analysis feature. Instead of having a table with all competitors for every candidate, we had a table of seats with all runners. This way, we don't have n very similar rows for an office of n competitors.

Data Storage

We use a Key-Value database which is basically big hashmaps. We do not scan the full table or use relational query. We are storing information about candidates, committees, states, analysis of contributions from individual to candidate, and so on. We have one DDB table for each table feature and also the master files (candidate and committee info).

Heatmap

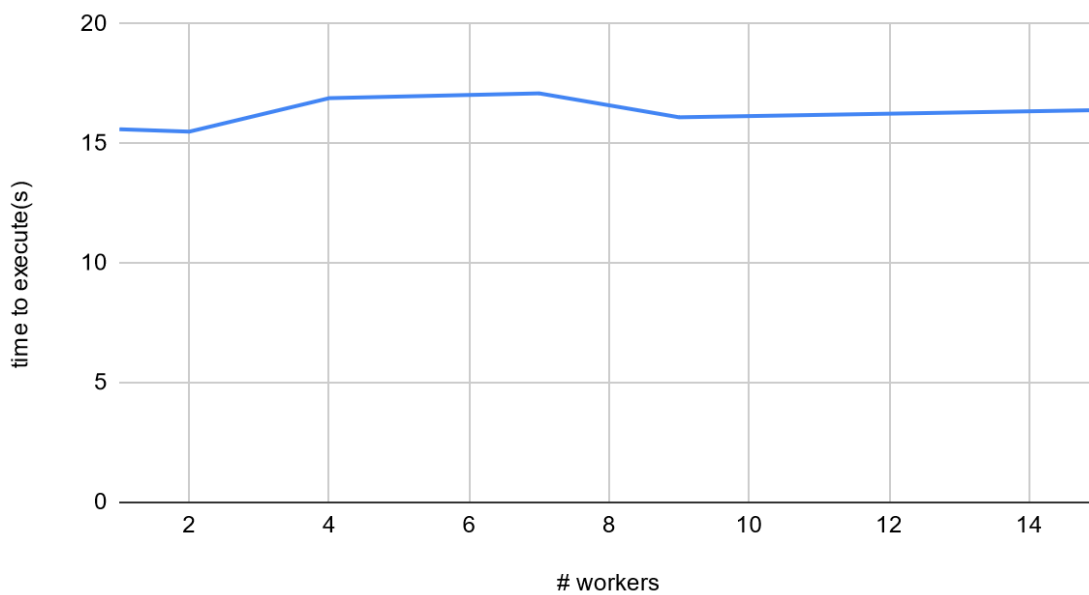
We were able to geocode zip codes and state names using the Google Geolocation API. This API works by returning the geolocation using imports such as GoogleV3 from Geopy, a Python client for geocoding web services. To minimize the number of zip codes to geolocate, we selected distinct zip codes from the tables. For the state names, we created a hashmap with the abbreviations as keys and written state names as values. This way, we were able to geocode state's using the written name instead of the abbreviations that were in the original dataset.

Evaluation

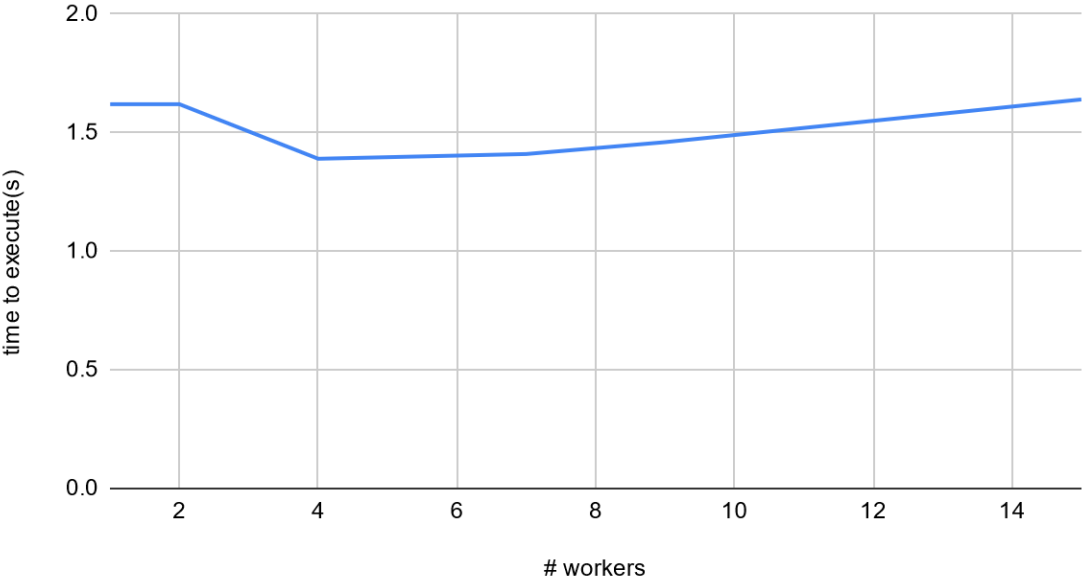
To evaluate the execution time of our spark data processing we chose 3 blocks of SQL queries necessary for our project. We tested the execution time with 15, 9, 7, 4, 2, and 1 workers. We collected multiple data points for each number of workers. For the Committee Info Page graphs we found the average of our data for each number of workers and plotted the results as a line graph. For the Explore Table Execution Time graph, however, we decided to display all the data points as a scatter plot because the results were interesting.

The first execution was always the slowest in our case and by a significant amount. This can be seen by the outliers in the Explore Table Execution time graph, having points far above the “main” cluster. Apart from that, we found that the minimum time tends to occur when there are a smaller amount of workers, but the average times were usually pretty close among all the worker amounts. On average, the execution times recorded for 2 and 4 workers took the shortest amount of time, with 7 and 1 workers being close behind.

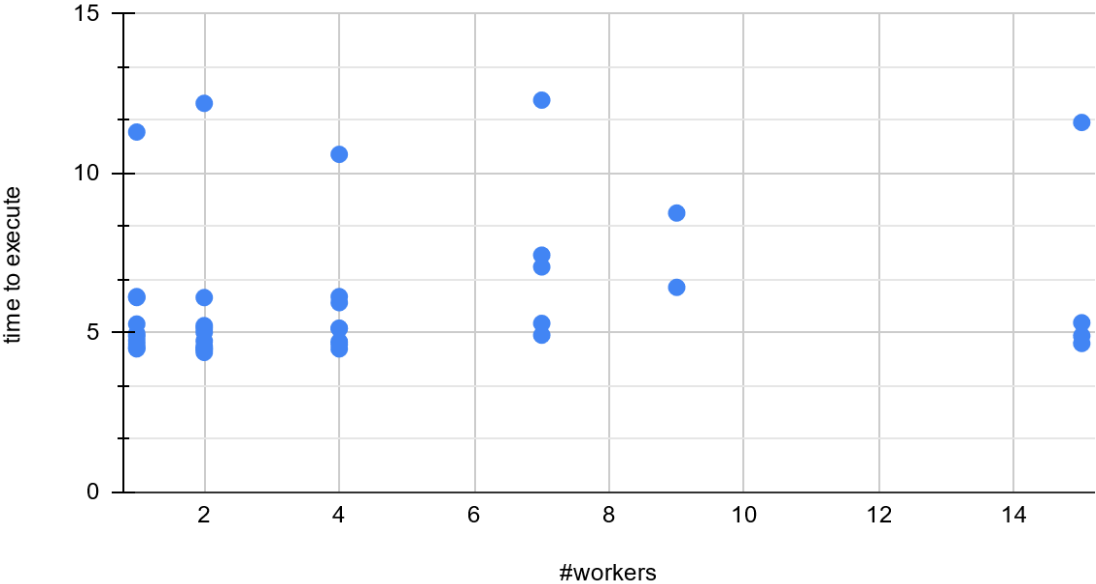
Committee Info Page Execution Time Graph 1



Committee Info Page Execution Time Graph 2



Explore Table Execution time



Screenshots

Nested tables

```
#Nested-Loops
#Reference from david_analytic_state.ipynb

unique = spark.sql("""
SELECT DISTINCT ID
FROM donorTablePre_5
""")

noName = spark.sql("""
SELECT ID AS ID, INDIVIDUAL, CITY, STATE, ZIP_CODE, OCCUPATION, TOTAL_MONEY_DONATED
FROM donorTablePre_5
""")

finalTable = unique.join(
    noName
    .groupBy("ID")
    .agg(collect_list(struct(noName.INDIVIDUAL, noName.CITY, noName.STATE, noName.ZIP_CODE, noName.OCCUPATION, noName.TOTAL_MONEY_DONATED)).alias("donateTable"))
    , "ID"
    , "outer"
)

finalTable.printSchema()
```

```
root
|-- ID: string (nullable = true)
|-- donateTable: array (nullable = true)
|   |-- element: struct (containsNull = false)
|   |   |-- INDIVIDUAL: string (nullable = true)
|   |   |-- CITY: string (nullable = true)
|   |   |-- STATE: string (nullable = true)
|   |   |-- ZIP_CODE: string (nullable = true)
|   |   |-- OCCUPATION: string (nullable = true)
|   |   |-- TOTAL_MONEY_DONATED: long (nullable = true)
```

```
{
  "CAND_ID": "H2AZ09191",
  "stateTable": [
    {
      "COMMITTEE": 2112230,
      "INDIVIDUAL": 10600,
      "STATE": "IL",
      "TOTAL": 2122830
    },
    {
      "COMMITTEE": 85950,
      "INDIVIDUAL": 770890,
      "STATE": "AZ",
      "TOTAL": 856840
    },
    {
      "COMMITTEE": 698660,
      "INDIVIDUAL": 12500,
      "STATE": "VA",
      "TOTAL": 711160
    },
    {
      "COMMITTEE": 251724,
      "INDIVIDUAL": 375000
    }
  ]
}
```

Custom Key

```
"CID": "HCA1",
"candidate": [
  {
    "CAND_ELECTION_YR": "2020",
    "CAND_ICI": "C",
    "CAND_ID": "H8CA01257",
    "CAND_NAME": "DENNEY, AUDREY L",
    "CAND_OFFICE": "H",
    "CAND_OFFICE_DISTRICT": "1",
    "CAND_OFFICE_ST": "CA",
    "CAND_PCC": "C000664400"
  },
  {
    "CAND_ELECTION_YR": "2022",
    "CAND_ICI": "C",
    "CAND_ID": "H2CA01250",
    "CAND_NAME": "GEIST, TIMOTHY DAVID",
    "CAND_OFFICE": "H",
    "CAND_OFFICE_DISTRICT": "1",
    "CAND_OFFICE_ST": "CA",
    "CAND_PCC": "C000664400"
  }
]
```

Filters

Attribute name	Type	Condition	Value	
CAND_OFFICE	String	Equal to	H	Remove
CAND_OFFICE_ST	String	Equal to	CA	Remove
CAND_OFFICE_DISTRICT	String	Equal to	1	Remove

Add Filter

RunReset

Completed Read capacity units consumed: 272

Data Storage

Tables (8) Info

↺

Actions ▾

Delete

Create table

Find tables by table name

Any table tag ▾

< 1 >

⚙

<input type="checkbox"/>	Name ▲	Status	Partition key	Size	Table class
<input type="checkbox"/>	AnalyticsCommieTable	✔ Active	CAND_ID (S)	2.3 megabytes	DynamoDB Standard
<input type="checkbox"/>	AnalyticsIndividualTable	✔ Active	CAND_ID (S)	1.5 megabytes	DynamoDB Standard
<input type="checkbox"/>	AnalyticsStateTable	✔ Active	CAND_ID (S)	333.7 kilobytes	DynamoDB Standard
<input type="checkbox"/>	CommieInfoTable	✔ Active	COMM_ID (S)	8 megabytes	DynamoDB Standard
<input type="checkbox"/>	CommunistParty	✔ Active	COMM_ID (S)	3.4 megabytes	DynamoDB Standard
<input type="checkbox"/>	CompetitorTable	✔ Active	COMP_ID (S)	165.7 kilobytes	DynamoDB Standard
<input type="checkbox"/>	DoraTheExplorerTable	✔ Active	CAND_ID (S)	1.3 megabytes	DynamoDB Standard
<input type="checkbox"/>	exploreFiltered	✔ Active	CID (S)	1.3 megabytes	DynamoDB Standard

Above figure is Amazon DynamoDB where we store our relevant tables.

Part 3. Build Web Interface

Requirements

It was necessary for us to have tables prepared for each page of the web interface. This included a table for Explore, Analytics (separate tables for individual, state, and committees), Committee Information, and Competitor Analysis. For the Analytics page, we had to include an array of geolocations and weights for the heatmap. This includes the longitude and latitude of the points of interest for each donor, as well as their weight respective to their donation for the heatmap intensity. We also needed to have a working modal to display the Competitor Analysis and Committee Information pop-up pages.

Design and Implementation

UX

Before we implemented our web interface using Typescript React, we designed and prototyped our pages and features using Figma. Screenshots of our prototypes are pasted below under Evaluation.

Frontend (Client)

We used Typescript React and Shopify Polaris for its component library & Design Principles.

Backend (Server + Database)

We use Node.js, Express, and Amazon DynamoDB to be able to search contributions, candidates, committees, etc by specific keywords.

Explore Page

The Explore page is split into left and right halves. On the left side, there exists a two-sectioned card that describes the key features of the product and also a concise tutorial of how to get started on the explore search feature. On the right, we have a top-bottom split. At the top, we have three selection components for the three main filters (office; state; district) that describe a specific seat that candidates run for. At the bottom, we have the Explore table that displays candidates according to the filters up top. This table component is populated by one of our DynamoDB APIs and has two additional optional filters (year of election; incumbent challenger status) to help our users find their desired candidate(s) more easily. Users can then click on the name of any candidate on the table and will be taken to the Analytics page.

Analytics

The Analytics page lives in the Analytics component, where it is split into Left and Right pages. On the right side lives the three separate Analytics tables where users can sort by money raised by Individuals, Committees, and State. These tables display information from their respective tables in DynamoDB by sending the Candidate ID selected in the Explore component to the Analytics component, both children of App, and calling the API that lives in app.ts and dynamo.ts in the server folder.

The left side of Analytics includes the Title and general information of the candidate along with the Heatmap that is a representation of the Individual and Committee tables. Design and implementation of the map is discussed below.

Heatmap

We implemented our heatmap using React Leaflet, a React component for Leaflet maps. React Leaflet integrates Open Street Map, which is a free wiki map of the world. Our heatmap takes an array of coordinates and weights from each candidate that are a result of the geocoded zip codes we generated using Google's API using Spark. Weights are determined by the ranking of the overall donation for each candidate.

The heatmap works as a visual representation of the distribution and concentration of donations for each candidate by Individuals and Committees. We multiplied our weights by 2000 to increase the intensity of the heat for visual purposes. The heatmap is implemented in our "Map" component, which is passed the Candidate ID and a flag from the Analytics Table component, where users switch between our three different tables.

Modals - Competitor Analysis and Committee Information

In the Analytics page, users can select the "View Competitor Analysis" button on the bottom left to view all Candidates that are running for the same office in the same state and district, if applicable. This Modal component is implemented using Shopify Polaris' "Modal" component which acts as a pop-up that displays information for the selected user from the Explore page.

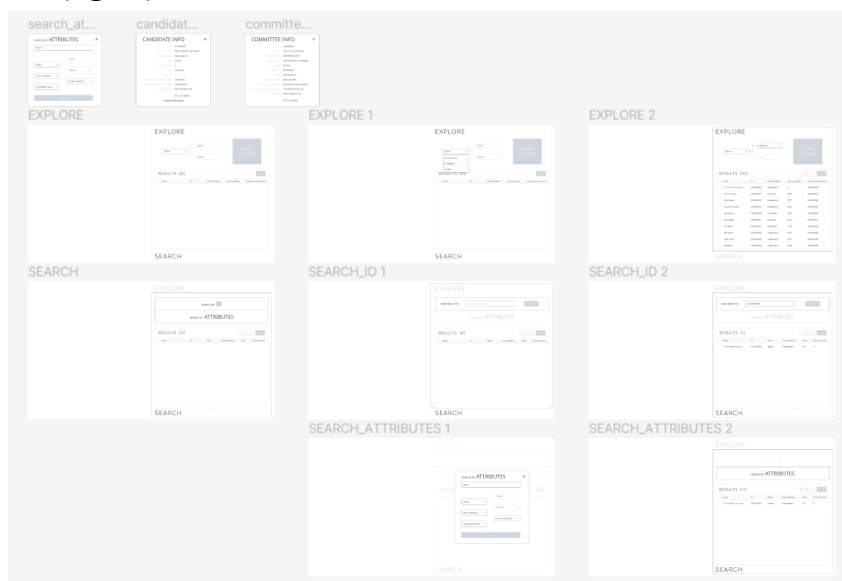
Our Modal component is split into Left and Right pages, where the left side includes a table from either Competitor Table or Commie Table component. If the user selects a Committee name from the Committee table, the latter is shown.

Both Competitor and Committee tables utilize Shopify Polaris' Table component, with the data being pulled from their respective tables stored in DynamoDB. We can pull this information using the API calls we have in app.ts and Dynamo.ts, stored in the server folder.

The tables are ranked by top recipients for the Committee modal and top raised for the Competitor analysis.

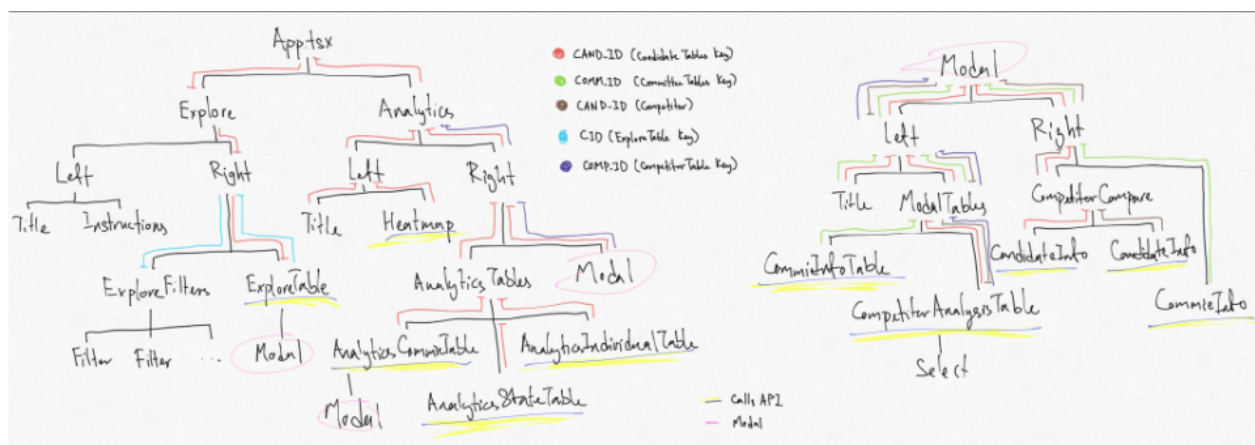
Evaluation

UX (figma)



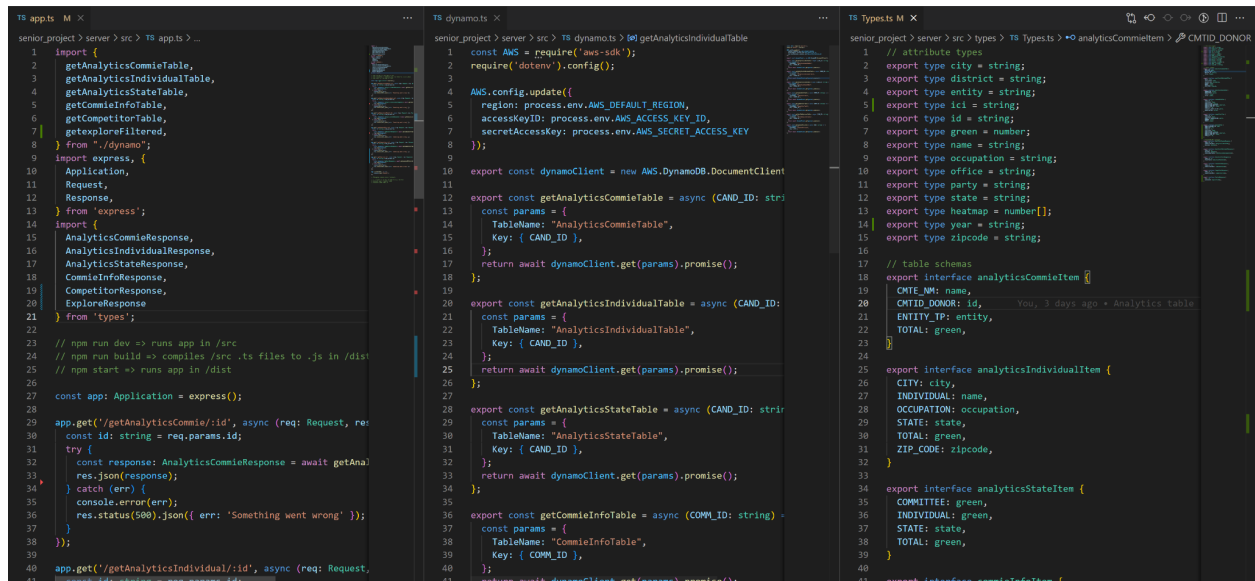
Above figure shows the initial designs of search and explore interfaces in figma.

FrontEnd



Above figure is the component tree and primary key flow of the whole application.

BackEnd



The screenshot displays three side-by-side code editors in a dark theme. The left editor, titled 'TB app.ts', shows an Express.js application setup with imports for various response types and a route for '/getAnalyticsCommieTable'. The middle editor, titled 'TB dynamo.ts', shows the AWS SDK configuration and the implementation of the 'getAnalyticsCommieTable' and 'getAnalyticsIndividualTable' functions using the AWS DynamoDB client. The right editor, titled 'TB Types.ts', defines the TypeScript interfaces for the API responses, including 'AnalyticsCommieItem', 'AnalyticsIndividualItem', 'AnalyticsStateItem', and 'CommieInfoItem'.

The backend consists of three main files. We have the Types.ts file that defines schemas for every API response, along with all the attribute types. Next, we have dynamo.ts that houses all DynamoDB APIs that are used to pull data from our 8 DDB tables. Lastly, we have the app.ts file that contains all API calls that connect the frontend to the DynamoDB APIs.

```
1 const AWS = require('aws-sdk');
2
3 require('dotenv').config();
4
5 AWS.config.update({
6   region: process.env.AWS_DEFAULT_REGION,
7   accessKeyId: process.env.AWS_ACCESS_KEY_ID,
8   secretAccessKey: process.env.AWS_SECRET_ACCESS_KEY
9 });
10
11 export const dynamoClient = new AWS.DynamoDB.DocumentClient();
12
13 export const getAnalyticsCommieTable = async (CAND_ID: string) => {
14   const params = {
15     TableName: "AnalyticsCommieTable",
16     Key: { CAND_ID },
17   };
18   return await dynamoClient.get(params).promise();
19 };
20
21 export const getAnalyticsIndividualTable = async (CAND_ID: string) => {
22   const params = {
23     TableName: "AnalyticsIndividualTable",
24     Key: { CAND_ID },
25   };
26   return await dynamoClient.get(params).promise();
27 };
```

The figure shown above is dynamo.ts file where we call dynamo client and export our tables to the dynamo server.

```

2  export type city = string;
3  export type district = string;
4  export type entity = string;
5  export type ici = string;
6  export type id = string;
7  export type green = number;
8  export type name = string;
9  export type occupation = string;
10 export type office = string;
11 export type party = string;
12 export type state = string;
13 export type heatmap = number[];
14 export type year = string;
15 export type zipcode = string;
16 export type design = string;
17 export type org = string;
18 export type cmttp = string;

```

```

19
20 // table schemas
21 export interface analyticsCommieItem {
22   CMTE_NM: name,
23   CMTID_DONOR: id,
24   ENTITY_TP: entity,
25   TOTAL: green,
26 }
27
28 export interface analyticsIndividualItem {
29   CITY: city,
30   INDIVIDUAL: name,
31   OCCUPATION: occupation,

```

Above figure shows the Types.ts file where we export interfaces such as analytics, explores, and information about candidates and committees so that we can display them on the React app.

```

1  export const OfficeKeys = new Map<string, string>([[ 'President', 'P'], [ 'Senate', 'S'], [ 'House', 'H']]);
2
3  export const StateKeys = new Map<string, string>([
4    [ '-', '-' ], [ 'Alabama', 'AL'], [ 'Alaska', 'AK'], [ 'Arizona', 'AZ'], [ 'Arkansas', 'AR'],
5    [ 'California', 'CA'], [ 'Colorado', 'CO'], [ 'Connecticut', 'CT'], [ 'Delaware', 'DE'],
6    [ 'Florida', 'FL'], [ 'Georgia', 'GA'], [ 'Hawaii', 'HI'], [ 'Idaho', 'ID'],
7    [ 'Illinois', 'IL'], [ 'Indiana', 'IN'], [ 'Iowa', 'IA'], [ 'Kansas', 'KS'], [ 'Kentucky', 'KY'],
8    [ 'Louisiana', 'LA'], [ 'Maine', 'ME'], [ 'Maryland', 'MD'], [ 'Massachusetts', 'MA'], [ 'Michigan', 'MI'],
9    [ 'Minnesota', 'MN'], [ 'Mississippi', 'MS'], [ 'Missouri', 'MO'], [ 'Montana', 'MT'], [ 'Nebraska', 'NE'],
10   [ 'Nevada', 'NV'], [ 'New Hampshire', 'NH'], [ 'New Jersey', 'NJ'], [ 'New Mexico', 'NM'], [ 'New York', 'NY'],
11   [ 'North Carolina', 'NC'], [ 'North Dakota', 'ND'], [ 'Ohio', 'OH'], [ 'Oklahoma', 'OK'],
12   [ 'Oregon', 'OR'], [ 'Pennsylvania', 'PA'], [ 'Rhode Island', 'RI'], [ 'South Carolina', 'SC'],
13   [ 'South Dakota', 'SD'], [ 'Tennessee', 'TN'], [ 'Texas', 'TX'], [ 'Utah', 'UT'], [ 'Vermont', 'VT'],
14   [ 'Virginia', 'VA'], [ 'Washington', 'WA'], [ 'West Virginia', 'WV'], [ 'Wisconsin', 'WI'],
15   [ 'Wyoming', 'WY']
16 ]);
17
18 export const DistrictKeys = new Map<string, number>([
19   [ 'AL', 7 ], [ 'AK', 1 ], [ 'AZ', 9 ], [ 'AR', 4 ],
20   [ 'CA', 52 ], [ 'CO', 8 ], [ 'CT', 5 ], [ 'DE', 1 ],
21   [ 'FL', 28 ], [ 'GA', 14 ], [ 'HI', 2 ], [ 'ID', 2 ],
22   [ 'IL', 17 ], [ 'IN', 9 ], [ 'IA', 4 ], [ 'KS', 4 ], [ 'KY', 6 ],
23   [ 'LA', 6 ], [ 'ME', 2 ], [ 'MD', 8 ], [ 'MA', 9 ], [ 'MI', 13 ],
24   [ 'MN', 8 ], [ 'MS', 4 ], [ 'MO', 8 ], [ 'MT', 2 ], [ 'NE', 3 ],
25   [ 'NV', 4 ], [ 'NH', 2 ], [ 'NJ', 12 ], [ 'NM', 3 ], [ 'NY', 26 ],
26   [ 'NC', 14 ], [ 'ND', 1 ], [ 'OH', 15 ], [ 'OK', 5 ],
27   [ 'OR', 6 ], [ 'PA', 17 ], [ 'RI', 2 ], [ 'SC', 7 ],
28   [ 'SD', 1 ], [ 'TN', 9 ], [ 'TX', 38 ], [ 'UT', 4 ], [ 'VT', 1 ],
29   [ 'VA', 11 ], [ 'WA', 10 ], [ 'WV', 2 ], [ 'WI', 8 ],
30   [ 'WY', 1 ]

```

Above figure is constants.ts file where we store a hashmap for individual states as the state abbreviation and the values as the coordinates to display on the heatmap.

Contributions

Part 2:

Analytics Tables

- Individual: Aung Thu Hein
- Committee: Amneh Alsuqi
- State: David

Committee Info Page Table: George Orduno Galicia

Competitor Analysis Table: Amneh Alsuqi

Explore Table: David

Candidate Master Table: Aung Thu Hein

Committee Master Table: George Orduno Galicia

Python post-process: David

DynamoDB implementation: David

Part 3:

Frontend

- Explore Page
 - Filters: David
 - Table: David
- Analytics Heatmap: George Orduno Galicia, Amneh Alsuqi, David
- Analytics Tables
 - Individual: Aung Thu Hein
 - Committee: Amneh Alsuqi
 - State: David
- Modal
 - Committee Information: George Orduno Galicia
 - Competitor Analysis Modal: Amneh Alsuqi
- Constants
 - AbbreviationKeys hashmap: George Orduno Galicia
 - Reverse hashmaps: Aung Thu Hein

Backend

- Types files: Amneh Alsuqi
- DynamoDB APIs: David
- Get APIs: All Members