PolitiPulse
By: Adam Botens and Austin Edwards

Introduction

In an era where political engagement and informed decision-making are more crucial than ever, deciphering complex political information remains a significant barrier for many voters. This difficulty underscores the need for accessible and comprehensible resources to bridge the knowledge gap between citizens and their government. Enter PolitiPulse, an innovative online platform designed to enhance political awareness and engagement among voters. PolitiPulse is a comprehensive hub for political information, primarily focusing on current and prospective members of Congress, offering users a gateway to a wealth of information, including details about elected representatives, pertinent news, legislative bills, and public statements.

A standout feature of this platform is its pioneering 'Bill Transformer.' This tool is designed to demystify congressional bills, stripping away complex jargon to render them more accessible and digestible for the average voter. By simplifying the language and condensing the content, the Bill Transformer fosters a greater understanding of legislative processes, encouraging deeper involvement in local and national politics.

PolitiPulse, with its user-friendly interface and rich, informative content, not only narrows the knowledge chasm but also empowers voters to make more informed decisions. This document dives into PolitiPulse's architecture, detailing its technical stack and the intricacies of its frontend and backend design. It further explores the platform's key features and projected role in fostering political engagement and enhancing voter empowerment. Through this exploration, we aim to underscore the significance of such platforms in today's politically charged environment.

Technology

PolitiPulse, an innovative platform designed to enhance political awareness and engagement, leverages a sophisticated blend of modern technologies to create a seamless and interactive user experience. At the heart of its technical architecture are carefully selected tools and frameworks that cater to the platform's functionality and scalability. A visual representation of how these technologies are integrated can be found in the block diagram in Figure 1 below.

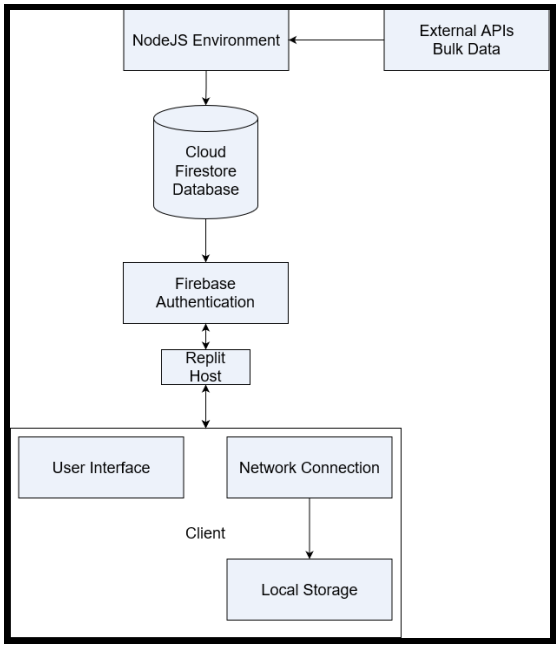

Figure 1. A block diagram depicting the components of PolitiPulse.

Frontend Development

For the front end, PolitiPulse harnesses the power of React, a widely acclaimed JavaScript library known for its efficiency in building user interfaces, particularly single-page applications. React's component-based architecture makes it a perfect fit for developing complex interfaces with dynamic content, such as PolitiPulse, which demands real-time updates and interactivity. Complementing React, Replit is an integrated development environment (IDE). Replit offers real-time code updates, which is instrumental for a platform like PolitiPulse that requires constant iterations and improvements. Additionally, Replit's user-friendly interface and affordability make it an attractive choice for development and hosting, ensuring that the team can focus on creating an impactful user experience without worrying about server management overhead.

Backend Development

PolitiPulse's backend, powered by Node.js, establishes a robust and scalable environment perfectly suited for handling high traffic and complex data requests. The platform utilizes Firebase's Firestore, a NoSQL database chosen for its flexibility and scalability. This decision is particularly advantageous given the variable size of data field entries and minimal relational interdependencies between different data collections. Firestore's schema-less nature perfectly fits PolitiPulse's diverse data set, which includes user profiles and intricate political information. Axios, a promise-based HTTP client, is integral to this setup, which facilitates efficient communication with the ProPublica, Congress.gov, ChatGPT, and Top Congress News APIs. The integration with these APIs enhances this capability, allowing PolitiPulse to fetch and store comprehensive legislative data, such as bill information and congressional activities. This combination of technologies ensures that PolitiPulse can efficiently process and store a wide

array of political data, making it an invaluable tool for users seeking up-to-date political information.

Authentication

Authentication is a critical component of PolitiPulse, and Firebase Authentication is chosen for this purpose. This choice is anchored on its seamless integration with the Firebase ecosystem and its robust set of features, including, but not limited to, secure user authentication, easy management, and compatibility with various authentication providers. Firebase Authentication simplifies managing user sessions and data, ensuring users can securely access PolitiPulse without hassle.

API Integration

To enrich the platform with comprehensive political content, PolitiPulse integrates several APIs. The ProPublica API offers detailed information about Congress members and legislative activities. Congress.gov is tapped for official legislative data, ensuring users can access authentic and up-to-date information. Finally, OpenAI's API is utilized for its groundbreaking chat completions model, enabling easy experimentation of the Bills Transformer feature. The U.S. Congress Top News Rapid API is also integrated to provide users with the latest news and updates. These APIs are critical in making PolitiPulse a comprehensive hub for political information. The integration of these APIs is achieved through Axios, a promise-based HTTP client that simplifies the process of sending asynchronous HTTP requests to REST endpoints. Axios enhances the platform's ability to fetch data efficiently and reliably.

In summary, the technology stack behind PolitiPulse is a thoughtful amalgamation of React, Replit, Node.js, Firestore, Firebase Authentication, and various APIs, each chosen for its specific strengths and compatibility with the overall system. This combination of technologies facilitates a robust and user-friendly platform and ensures that PolitiPulse remains scalable and responsive to the evolving needs of its users.

Insights and Challenges

Team One

Team One began their project by focusing on the application's structure and navigation. The initial phase involved creating the application's skeleton, which is essential for establishing the framework and flow. This task required the team to quickly learn and adapt to new technologies, including React and Bootstrap, moving away from traditional CSS. They also faced the challenge of navigating file directories through code, mainly when dealing with hard links.

The next significant focus for the team was optimizing website navigation. They dedicated several weeks to ensuring efficient navigation, minimizing unnecessary database calls, and reducing lag for users with poor internet connections. After experimenting with various

solutions, the team decided to use React-Router-Dom. Completing the user interface for the navigation bar was a key accomplishment in this stage of development.

Finally, Team One tackled the aspect of authentication. With Firebase chosen as their cloud storage solution, they had to navigate the complexities of FireAuth. Setting up the basic Register and Login functions was relatively straightforward. However, integrating these functions into a React Context and creating a mirrored database in Firestore for additional user information posed significant challenges. One notable issue at the project's conclusion was the inability to delete user accounts through the interface due to Firestore's security features, requiring admin intervention for such actions.

Team Two

Navigating the complexities of the technologies employed in PolitiPulse presented a notable challenge, primarily due to the rapid evolution and updates these technologies frequently undergo. The disparity in information found across various tutorials, official documentation, and other resources added to the complexity. Being new to web design, we encountered a steep learning curve, especially in understanding concepts like asynchronous functions, promises, and HTTP requests. Nonetheless, the advanced capabilities and efficiencies of these technologies made learning worthwhile. Their utilization undeniably facilitated faster and more streamlined development despite the initial hurdles.

Fortunately, the technologies we chose for the back end remained consistent throughout the project, aligning with our initial plans. One notable deviation was our decision to forgo Cloud Functions for Firebase, which we initially considered incorporating. Upon further deliberation, the team concluded that Cloud Functions was optional for our project scope, as we did not intend to implement scheduled automatic updates. Since this was the primary rationale for considering Cloud Functions, we decided against their inclusion in the first version of PolitiPulse. In retrospect, this decision aligned well with our project needs, ensuring a more focused and streamlined development process.

Design

To design PolitiPulse, the team separated the application into two separate fields. Both team members tackled the front-end and back-end components of the website, with one member focusing on content storage, display, and database architecture while the other focused on the website architecture, navigation, and user authentication. The UML diagram for PolitiPulse can be found in Figures 2 and 3 of the appendix.

Frontend

In the following detailed sections, we will explore various components, contexts, and pages that constitute the design of PolitiPulse. This comprehensive examination covers the application's setup, key components responsible for user interface and data presentation, and the crucial context for user authentication. Each component and page serves a specific purpose, contributing to the application's functionality and user experience. We will explain their roles,

functionalities, and implementations, providing a comprehensive understanding of how they work together. Figure 4, seen below, serves as a visual aid for the project file and folder structure.

```
src
|   App.jsx
|   firebase.jsx
|   index.jsx
|
+---backend
|       fetchBills.js
|       fetchHouse.js
|       fetchHouseImages.js
|       fetchNews.js
|       fetchSenate.js
|       fetchSenateImages.js
|       fetchStatements.js
|       fetchVotes.js
|       transformBills.js
|
\---frontend
    +---components
    |       BillPages.jsx
    |       HouseMembers.jsx
    |       NavBar.jsx
    |       SenateMembers.jsx
    |
    +---contexts
    |       contexts.jsx
    |       states.jsx
    |
    +---img
    |       avatar.png
    |
    \---pages
        +---house
        |       Home.jsx
        |       HouseCurrentElected.jsx
        |       HouseElection.jsx
        |
        +---senate
        |       SenateCurrentElection.jsx
        |       SenateElection.jsx
        |
        \---user
                About.jsx
                Login.jsx
                Profile.jsx
                Register.jsx
                Signout.jsx
```

Figure 4. Tree of PolitiPulse folder and file structure.

Components

Index.jsx: This component is the fundamental entry point for a React-based web application. It imports the necessary libraries, including React, ReactDOM, and the main application component, "App." It also imports "BrowserRouter" from the "react-router-dom" library for handling client-side routing. The script uses "ReactDOM.createRoot" to render the entire application within the HTML element with the I.D. "root." Using "React.StrictMode" ensures that potential issues in the application are highlighted during development. Overall, this script serves as the starting point for the web application, setting up the essential dependencies and rendering the main application component, thus initiating the user interface and routing functionality.

App.jsx: A key component of the React-based web application, defining the overall application structure and routing. It begins by importing essential components, pages, and libraries, including the application's navigation bar component, various pages related to the House, Senate, and user profiles, and the necessary routing components from "react-router-dom." The script also imports the "AuthProvider" from the application's custom contexts for managing user authentication.

The "App" function wraps the entire application within the "AuthProvider" to manage the authentication state. The application's primary structure consists of a navigation bar, imported as "NavBar," and a container div for rendering different pages based on the routing configuration. The "Routes" component defines the routing paths and maps them to specific page components, ensuring users can navigate seamlessly between different views. This script is the central hub for structuring the web application, enabling navigation, and maintaining user authentication.

NavBar.jsx: The "NavBar" React component renders the top navigation bar on web pages. It uses "react-router-dom" components and "react-bootstrap" for styling. The navigation links are organized into dropdown menus for the "House," "Senate," and "User" sections. Depending on the user's authentication status, it conditionally displays links such as "Login" and "Register" for logged-out users and "User Profile" and "Sign Out" for logged-in users. The custom "Clink" component handles link styling based on whether a link is active.

firebase.jsx: Exports the initialized authentication and Firestore instances as auth and db, respectively, making them accessible throughout the application. This code is crucial for enabling user authentication and real-time database interactions in the web app. It is recommended that this file be added to your .gitignore.

BillDetails.jsx: This component in the PolitiPulse web application displays detailed information about legislative bills. It utilizes React and various libraries for state management, routing, and data fetching. Upon receiving the bill I.D. from the URL parameters, it fetches the corresponding bill's data from Firebase Firestore. The component then renders this information in a user-friendly manner, presenting details such as the bill's title, date, description, number, status, and actions.

Additionally, it provides a link to the full bill text. The component also processes and displays a simplified version of the bill text, replacing newlines with HTML line breaks for readability. It employs Bootstrap for styling, ensuring a clean and responsive user interface.

HouseMembers.jsx: The HouseMembers component is responsible for displaying detailed information about members of the House of Representatives. It utilizes React and various libraries for state management, routing, and data fetching. When provided with a specific House member's I.D. through URL parameters, it fetches the corresponding member's information from Firebase Firestore and recent bills they have voted on.

The component's layout is divided into two main sections: Member Info and Recent Bills Voted On. The Member Info section presents essential details about the House member, including their name, congressional district, date of birth, social media profiles, gender, party affiliation, and more. The component also includes an image of the member, if available.

The Recent Bills Voted On section displays a list of bills the House member recently voted on. It provides information such as the bill title, I.D., roll call number, voting question, and the member's position on each bill. Users can click individual bill links to access more detailed information about each bill.

The component includes loading indicators when fetching data and handles scenarios where no recent bills are available. It employs Bootstrap for styling, ensuring a responsive and visually appealing user interface.

SenateMembers.jsx: This component is nearly identical to HouseMembers. The critical difference is that it retrieves Senate member information from the ‘senate’ collection and expects a ‘senateMemberId’ parameter passed to the URL.

Contexts

contexts.jsx: This code defines an authentication context in a React application using Firebase Authentication. It creates a context called AuthContext, provides a custom hook useAuth to access authentication-related functionality, and sets up an AuthProvider component to manage the authentication state.

Inside the AuthProvider, it initializes state variables currentUser (to store the current authenticated user) and loading (to track the loading state while checking authentication). It uses the useEffect hook to listen for changes in the user's authentication status using Firebase's onAuthStateChanged function. When the authentication state changes, it updates the currentUser and sets loading to false.

Finally, it provides the currentUser value through the context so that other components in the application can access the authenticated user's information. The children prop represents the child components that this AuthProvider will wrap.

states.jsx: This code simply defines an array of objects for the states. Each object has two properties: value and name. The value of each object is the state's abbreviation, and the name is the full name of the state.

The array includes the 50 states of the United States and territories like Guam, Puerto Rico, American Samoa, the Northern Mariana Islands, the U.S. Virgin Islands, and the District of Columbia.

Pages

Home.jsx: The homepage offers a comprehensive overview of various data elements. Firstly, if applicable, it retrieves information about Congress members, focusing on those representing the user's district and state. If the user is not logged in, it randomly selects a state and district to be queried. Using Firebase Firestore queries, it assembles Senate and House members' details and displays them within a Card component. Each member's name is linked to their profile page.

Additionally, the component fetches and showcases recent statements made by Congress members. Utilizing Firestore queries once again, it aggregates these statements and presents them in a Card. Each statement includes the member's name, statement type, and a convenient link to access the complete statement.

Furthermore, the "Home" page gathers recent news articles for the user's convenience. It conducts Firestore queries to access and exhibit these articles within a Card. Each news item contains essential information, including its title, a brief description, publication date, source, and a hyperlink to access the full article.

This responsive layout elegantly organizes the content, placing Congress members on the left side of the screen along with recent statements and news articles on the right. In summary, the "Home" component offers users an informative and engaging homepage experience, delivering essential information about their Congress members, recent statements, and relevant news articles in a user-friendly manner.

HouseCurrentElected.jsx: This page displays information about the current elected House representatives. It utilizes Firebase Firestore queries to fetch relevant data from the database. Specifically, it retrieves House members of the 118th Congress, effectively simulating pagination.

Upon fetching the data, the component organizes it into a responsive layout. Each House member is a Card element containing essential details such as their first and last name, party affiliation, state, district, and Congress number. The Card also includes an avatar image, with the background color corresponding to the member's party (e.g., dark blue for Democrats, dark red for Republicans).

This component calculates the current page, the starting and ending indices of the displayed members, the total number of results, and the total number of pages. Users can navigate between pages using the "Previous" and "Next" buttons, which are appropriately disabled when necessary.

In addition, the component provides users with information about the range of displayed results, enhancing the user experience by indicating the subset of House members currently shown.

HouseElection.jsx: serves as a user interface for displaying a list of House candidates and includes pagination capabilities. This component interacts with a Firestore database to fetch data and presents it in an organized manner. The main features of this component include state management, data fetching, pagination, and candidate information display.

State variables are used to manage the house data array, representing the list of House candidates, the page variable indicating the current page number, and perPage, which defines the number of items displayed per page.

The fetchPost function is responsible for fetching data from Firestore. It constructs a Firestore query to retrieve House candidates with a "next_election" value of '2024'. When executed, the query returns a snapshot of the data, mapped to an array of objects and stored in the house state variable.

Data fetching occurs automatically when the component mounts, thanks to the useEffect hook.

Pagination is implemented by calculating the start and end indexes based on the current page, allowing the display of a subset of the candidate data. Pagination controls, including "Previous" and "Next" buttons, enable users to navigate the candidate list easily.

Candidate information is presented using React Bootstrap's "Card" component. This information includes last name, first name, party affiliation, state, district, congress information, and the next election date.

Users can seamlessly move between pages of candidate data by clicking the "Previous" and "Next" buttons, providing an enhanced user experience. In summary, the "HouseElection" component offers an efficient and user-friendly interface for browsing and exploring information about House candidates running in the 2024 election.

SenateCurrentElection.jsx: Designed to display a list of senators and provide pagination functionality. It interacts with Firestore to fetch data for senators serving in the 118th Congress. Key features of this component include state variables, data fetching, pagination, and senator data display.

State variables are used to manage the senate data array, the page representing the current page number, and perPage indicating the number of items displayed per page. The fetchSenates function retrieves senator data from Firestore by executing a query. Data fetching is triggered when the component mounts using the useEffect hook.

Pagination functionality is implemented by calculating the start and end indexes for displaying a subset of senator data based on the current page. Pagination controls like "Previous" and "Next" buttons are provided for user convenience.

Senator data is presented using React Bootstrap's "Card" component, displaying details such as last name, first name, party, state, and congress information.

Users can easily navigate between pages of senator data by clicking the "Previous" and "Next" buttons, enhancing the user experience. This component offers an organized and user-friendly interface for browsing and exploring information about senators serving in the 118th Congress.

SenateElection.jsx: Nearly identical to HouseElection, essentially just replacing ‘house’ with ‘senate’.

Registration.jsx: Facilitates user registration with a clean and user-friendly interface. It is encapsulated within a Bootstrap-styled container, ensuring a responsive design. Users must provide their email address, choose a password, confirm the password, select their state from a dropdown menu, and specify their voting district.

Upon submission of the registration form, the onSubmit function comes into action. It first validates that the entered passwords match, providing a basic level of security. Then, it utilizes Firebase Authentication to create a new user account, associating it with the provided email and password. This ensures that users can securely log in to their accounts in the future.

Additionally, it stores user data in the Firestore database. This data includes the user's unique ID, email, chosen state, and voting district. Storing this information is essential for personalized user experiences and access control.

Furthermore, upon successful registration, the component leverages React Router's useNavigate hook to redirect users to their user profile page, enhancing the overall user experience. This redirection ensures that users can immediately access and interact with their profiles after completing the registration process.

Profile.jsx: Designed to display and update users' profile information, including their email, state, and voting district. It offers a seamless user experience with a clean and straightforward interface. When a user logs in, the component uses the Firebase Authentication's onAuthStateChanged function to check if a user is authenticated. If so, it retrieves the user's unique ID and fetches their profile data from Firestore, including email, state, and voting district. This ensures that the user's existing information is readily available.

Users can update their state and voting district information through the form provided. The "onSubmit" function listens for form submissions. If the user chooses to update their information, the updateFireStore function is called, which updates the Firestore document associated with the user's ID. It ensures data integrity and keeps the user's information up to date.

The form includes fields to display the user's current email, state, and voting district. The state is displayed as a dropdown menu with options for all U.S. states, allowing users to select and update this information quickly. The voting district is represented as a number input field.

Finally, the component handles security. If a user attempts to update their email address without re-authenticating, it handles the situation by notifying the user to re-authenticate for added security.

Login.jsx: Provides a user-friendly interface for user authentication. It is designed to allow users to log in to their accounts with minimal effort. The component utilizes Firebase Authentication for secure login functionality.

The component is contained within a responsive container, ensuring it is properly aligned and centered on the page. This responsive design enhances the user experience, regardless of the device used.

Within the component, the "LoginComp" function is responsible for handling the login process. It retrieves the user's email and password from the input fields, preventing the default form submission action from handling the login process programmatically.

When users submit their login credentials, the function calls Firebase Authentication's signInWithEmailAndPassword method. If the provided credentials are valid, the user is successfully authenticated, and the application navigates to the home page ("/"). In case of an error, the component handles it gracefully by displaying relevant error messages to the user, ensuring transparency and user-friendliness.

The form consists of two essential fields: email and password. Users are required to input their email address and password to log in securely. The "Login" button triggers the authentication process, providing a straightforward and intuitive login experience.

Signout.jsx: Integrates with Firebase Authentication to allow users to securely sign out of their accounts. This function is designed to provide a straightforward and user-friendly sign-out experience.

Within the function, it first retrieves the authentication instance using Firebase Authentication's getAuth method. This instance ensures that the sign-out process is performed securely and with the user's current authentication status.

The navigate hook from React Router is used to handle navigation after the sign-out process is completed. It directs users back to the home page ("/") once they have successfully signed out, ensuring a smooth transition and clear feedback to the user.

The core functionality of signing out is accomplished by calling signOut(auth), where auth is the previously obtained authentication instance. If the sign-out is successful, a message is logged to the console, indicating that the sign-out process was completed successfully. Subsequently, the user is navigated back to the home page.

In case of an error during the sign-out process, the function catches the error and logs an error message to the console. Regardless of whether the sign-out is successful, the user is always navigated back to the home page, maintaining a consistent user experience.

Backend

The PolitiPulse backend is the critical infrastructure for sourcing data from diverse API endpoints and seamlessly integrating it into the Firestore database. Each fetch from an API and

push to Firestore is run inside of its own individual script. This modular approach ensures efficient bug management and continuous refinement of our backend architecture.

Collections

The PolitiPulse database comprises seven collections: bills, house, news, senate, statements, users, and votes. The bills, house, news, senate, statements, and votes collections are all built using data from external APIs, while the user collection is built following the registration of a user. A diagram of the PolitiPulse NoSQL database can be seen below in Figure 5:

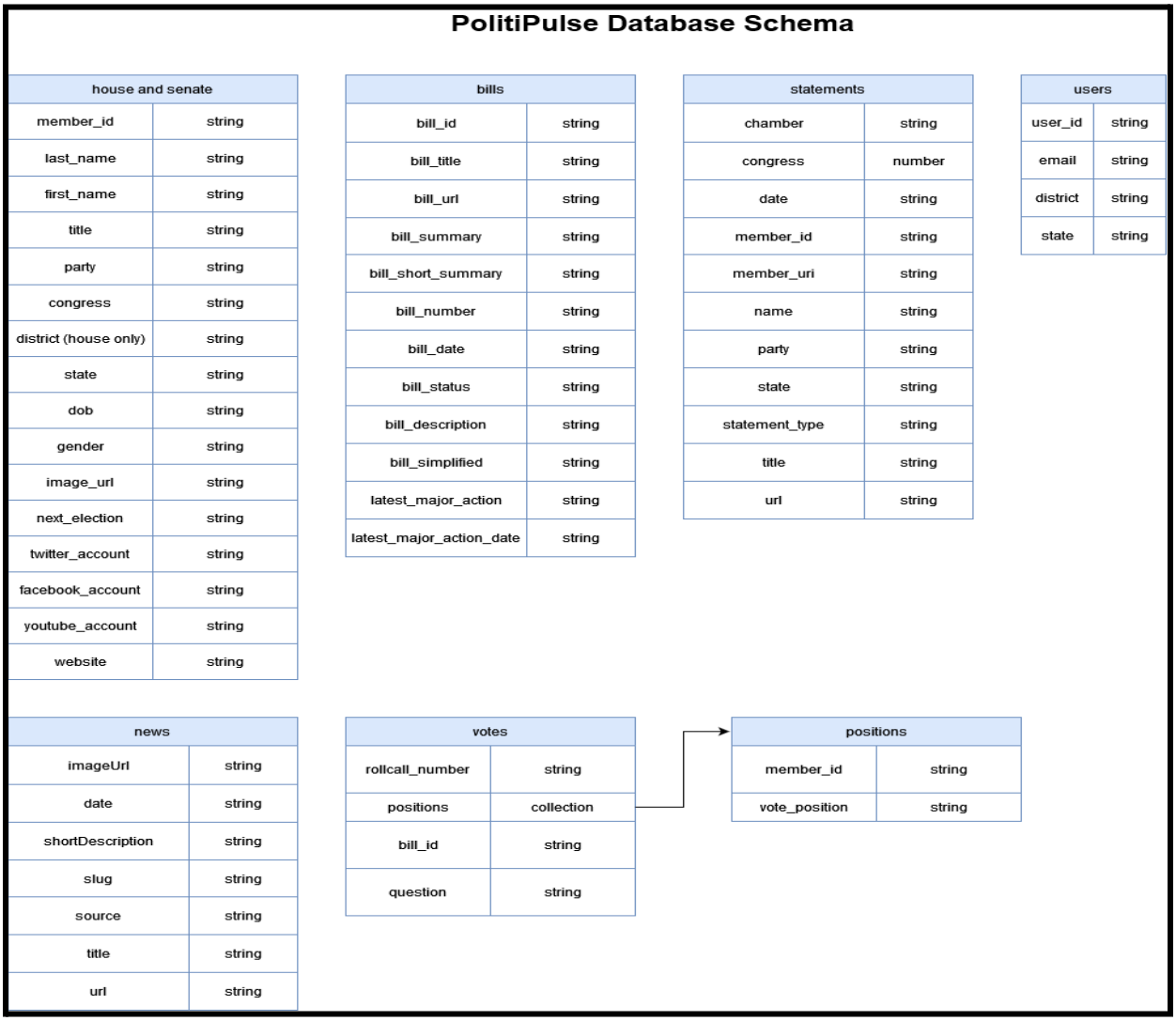

Figure 5: A visual representation of the PolitiPulse database.

Scripts

PolitiPulse currently consists of nine separate scripts for fetching and pushing data between external APIs and the database. Each script is structured similarly, with imports and configurations declared before the primary driving asynchronous functions, ‘FetchAndPushData’

or ‘FetchAndPushImage.’ A main function was added to each script for orchestration. Some of the scripts contain offset and pagination algorithms inside their main function. The following scripts are employed to construct and maintain the PolitiPulse backend:

fetchBills.js: This script retrieves congressional bill information from the ProPublica Congress API and integrates it into Firestore. Key functionalities include initializing the Firebase Admin SDK, crafting HTTP requests with Axios, processing data, and implementing error handling. Regular execution updates the 'bills' collection, providing users with the latest legislative developments.

Each bill's number (bill_number) and short title (bill_title) offer quick identification, while the full title (bill_description) provides a more comprehensive understanding. The introduction date (bill_date) and current status (bill_status) keep users informed about the bill's timeline and activity. Furthermore, the script captures the latest major action and its date, offering insight into recent legislative developments. For in-depth analysis, it stores both a detailed summary (bill_summary) and a brief version (bill_short_summary) alongside a URL (bill_url) for users to access full bill information directly. This comprehensive approach ensures that users can access essential legislative details, enhancing the platform's utility in promoting political engagement.

fetchHouse.js: Dedicated to House of Representatives member information retrieval, this script initializes the Firebase Admin SDK, utilizes Axios for API requests, and stores data systematically in the 'house' collection. Like the fetchBills script, it retrieves congressional member information from the ProPublica Congress API. Error handling ensures reliability and focuses on providing detailed political information about House members.

The script retrieves member data, including names, titles, states, districts, parties, dates of birth, gender, and social media accounts, and stores each member's information in Firestore, maintaining a database of congressional members. This process is crucial for keeping the platform's data up-to-date and accurate, allowing users to access current information about their representatives.

fetchHouseImages.js: Interfacing with the Congress.gov API, this script retrieves and updates images of House members in the 'house' collection. It initializes Firebase Admin SDK, employs Axios for API integration, and iteratively updates member images. Robust error handling enhances reliability.

It retrieves member IDs from the 'house' collection and uses these IDs to fetch image URLs from the Congress.gov API. Once the image URLs are obtained, the script updates each Firestore document with the corresponding image URL. This process ensures that the platform displays the latest visual representations of congressional members, enriching the user experience by providing a more personal and engaging interaction with political data.

fetchNews.js: Fetching U.S. congressional news from the 'US Congress Top News' API, this script initializes the Firebase Admin SDK, utilizes Axios for API integration, and stores news articles in the 'news' collection. Error handling and data verification enhance reliability, ensuring users access timely political news.

It retrieves news data from an external API designed explicitly for top congressional news and updates the Firestore database with this information. By storing news items in the 'news' collection of Firestore, the script ensures that users of PolitiPulse have access to current and relevant congressional news, thereby enhancing the platform's value as a resource for political engagement and information.

fetchSenate.js: Once again retrieving information from the ProPublica Congress API, this script focuses on Senate member information retrieval. It initializes Firebase Admin SDK, uses Axios for API requests targeting the Senate, and stores data in the 'senate' collection. Error handling ensures reliability, providing comprehensive data about Senate members. It is similar to fetchHouse.js.

Similar to the fetchHouse.js script, it retrieves member data, including names, titles, states, parties, dates of birth, gender, and social media accounts, and stores each member's information in Firestore, maintaining a database of congressional members.

fetchSenateImages.js: Retrieve and update Senate member images; this script interacts with the Congress.gov API. It initializes Firebase Admin SDK, uses Axios for API integration, and employs iterative updates. Robust error handling ensures visual accuracy in the 'senate' collection. Like many other scripts, it is nearly a mirror of its House counterpart.

fetchStatements.js: Fetching from the ProPublica Congress API and storing public statements from U.S. congressional members, this script initializes Firebase Admin SDK, utilizes Axios for API requests, and stores data in the 'statements' collection. Error handling and pagination support enhance reliability, providing timely official statements.

It retrieves statements, including URLs, dates, titles, statement types, and member details. Each statement is stored in the 'statements' collection with its associated information, ensuring that PolitiPulse users can access the latest and most relevant congressional statements for informed political engagement. The member ID links statements to a user’s congressional members.

fetchVotes.js: Aggregating and storing congressional vote data found in the ProPublica Congress API, this script initializes Firebase Admin SDK, uses Axios for API integration and employs advanced data retrieval. It handles errors gracefully, ensuring users have access to detailed voting records.

This script is designed to fetch and store congressional vote data, including roll call numbers and associated bill IDs, from the ProPublica API into Firestore. It then proceeds to retrieve detailed member voting information for each roll call. The script ensures that each voting record in Firestore includes the bill ID, the specific question posed in the vote, and the positions of individual members. This comprehensive approach gives PolitiPulse users a deeper understanding of congressional voting patterns, enhancing the platform's role as a political engagement and analysis tool.

transformBills.js: This script enhances PolitiPulse by integrating OpenAI's language model to simplify the text of congressional bills. It fetches the original bill text as XML processes it and then uses OpenAI to generate a simplified version. The script updates Firestore's 'bills' collection with this simplified text, making complex legislative language more accessible to users. This

functionality is particularly valuable for users seeking to understand legislative documents without specialized legal or political knowledge, promoting broader political engagement and understanding.

Collectively, these scripts enable PolitiPulse to provide users with up-to-date and comprehensive political information, fostering informed civic engagement.

How to Build and Deploy

Downloading the Code from GitHub

1. Visit the GitHub repository URL: https://github.com/aedwards17/PolitiPulse.
2. Click on the 'Code' button and choose 'Download ZIP,' or use Git to clone the repository using the command: git clone https://github.com/aedwards17/PolitiPulse.git.

Setting up the Project on Replit

3. Log in to your Replit account. If you do not have one, create a new account at Replit.
4. Click on the '+' button to create a new repl.
5. Select 'Import from GitHub.' Paste the URL of your GitHub repository and import the project.

Setting up the Server

6. Replit provides a simple web server environment. Configure your app's entry point (like index.js or app.py) correctly.
7. The front end will require a key for Firebase. Environment variables are not available on the front end, so it is necessary to add 'firebase.jsx' to your .gitignore file for security purposes.

Creating and Integrating Firebase Database

8. Go to the Firebase Console.
9. Create a new project or select an existing one.
10. Navigate to the 'Firestore Database' section and create a new database.
11. Create an individual Replit for each backend script, selecting Node.js as the environment for each repl.
12. Store API keys for ProPublica, Firebase, Firestore, Congress.gov, and Top Congress News in Replit's 'Secrets (Environment Variables).'
13. Run each backend script at least once to ensure proper setup.

Running the Application

14. Back in the main repl, hosting our cloned files, use the 'Run' button on Replit to start your application.
15. Test the application to ensure it connects to Firebase and operates as expected.

Deploying the Application

16. Setting Up Your Repl:
    a. First, ensure your Repl works correctly using the "Run" button.
    b. If your project requires building into static files (like for a web project), execute the necessary build command (e.g., npm run build) and ensure the output directory (like dist) is correct.
17. Creating a Deployment:
    a. Open the Deployments tab by clicking the "Deploy" button at the top right of the workspace or by opening a new pane and typing "Deployments."
18. Configuring Your Deployment:
    a. Configure the build command and specify the public directory. This is where your static files are built. For example, if your static files are in the dist directory, specify that.
    b. Set up 'Index' and 'Not Found' pages. Your index.html in the public directory will serve as the home page. You can also provide a custom 'Not Found' page with a 404.html file in the same directory.
19. Starting Your Deployment:
    a. After configuration, click "Deploy" to start the deployment process. Once complete, you will get details like the URL and build logs.
20. Billing Considerations:
    a. For Replit Core (previously Hacker or Pro) subscribers, static deployments are free, with up to 100 static deployments included.
    b. Users on the free plan need to add a credit card for creating a Static Deployment. There is a 10 GiB limit on outbound storage transfer, with charges applicable for additional usage.

Additional Notes

21. Always ensure your Firebase rules are set correctly to prevent unauthorized access.
22. Contact the authors of this repository for any assistance.

Known bugs

In the documentation for the PolitiPulse project, several known bugs that impact the functionality and user experience have been identified. One such issue is a limitation in the bill transformation functionality, where bills larger than 4,096 tokens, equivalent to approximately 10,000 to 24,000 characters, cannot be processed. This limitation could hinder the analysis of longer legislative documents, potentially impacting the comprehensiveness of the information provided by the application.

Another significant issue is related to the data management within Firebase Authentication (FireAuth). As it stands, once data is created, it cannot be altered except through manual deletion. This constraint is due to the application not meeting Google's stringent security parameters for user-initiated data modifications in Auth. Specifically, the application lacks the implementation of phone authentication security parameters, and it is not set up to utilize

fingerprint verification. This limitation on data management could affect the user's ability to update or modify their information securely and conveniently, thus impacting the overall user experience.

These bugs highlight areas for potential improvement in future updates of the PolitiPulse application, emphasizing the need for enhanced functionality in statement retrieval, handling of large legislative documents, and secure, flexible data management within the authentication system.

Future work

In terms of future enhancements for the PolitiPulse project, several key areas have been identified for development and improvement. First, there is a plan to rename collections to streamline and optimize database organization. This initiative is aimed at enhancing data management and accessibility.

Furthermore, the development team intends to create a master script responsible for executing all backend scripts automatically every 24 hours. This automation would significantly improve the efficiency of data updates and maintenance processes. Another ambitious aspect of future work involves training a Large Language Model (LLM) to tackle the challenge of transforming large bills. This approach could address the current limitation in handling extensive legislative documents.

Additionally, improvements in pagination are on the agenda. This enhancement will likely provide a more seamless and user-friendly navigation experience within the application. Security is also a top priority, with plans to upgrade security protocols to meet Google's standards. Achieving this would enable the functionality for users to make changes to their information within the Firebase Authentication system, thereby enhancing user autonomy and data management capabilities.

Efforts to decrease the number of database calls during page load are also underway. This optimization aims to enhance the application's performance and user experience by reducing load times. Lastly, the team plans to incorporate a feature for quick lookup of congressional districts using a user’s zip code. This addition would greatly facilitate users’ access to relevant political information based on their geographical location.

Conclusion

These planned improvements and additions collectively underscore a commitment to evolving and refining the PolitiPulse application, ensuring it remains a valuable and efficient tool for users interested in political information and legislative processes. Through integrating advanced technologies, including API connections and machine learning models, PolitiPulse offers an innovative approach to understanding political processes. Its ability to simplify legislative language and provide real-time updates on political developments represents a significant step towards increasing political awareness and engagement among users. As the platform continues to evolve, its potential to empower voters and influence political discourse

becomes increasingly evident. PolitiPulse, therefore, stands as a testament to the transformative power of technology in democratizing information and fostering a more informed and engaged electorate.

Appendix

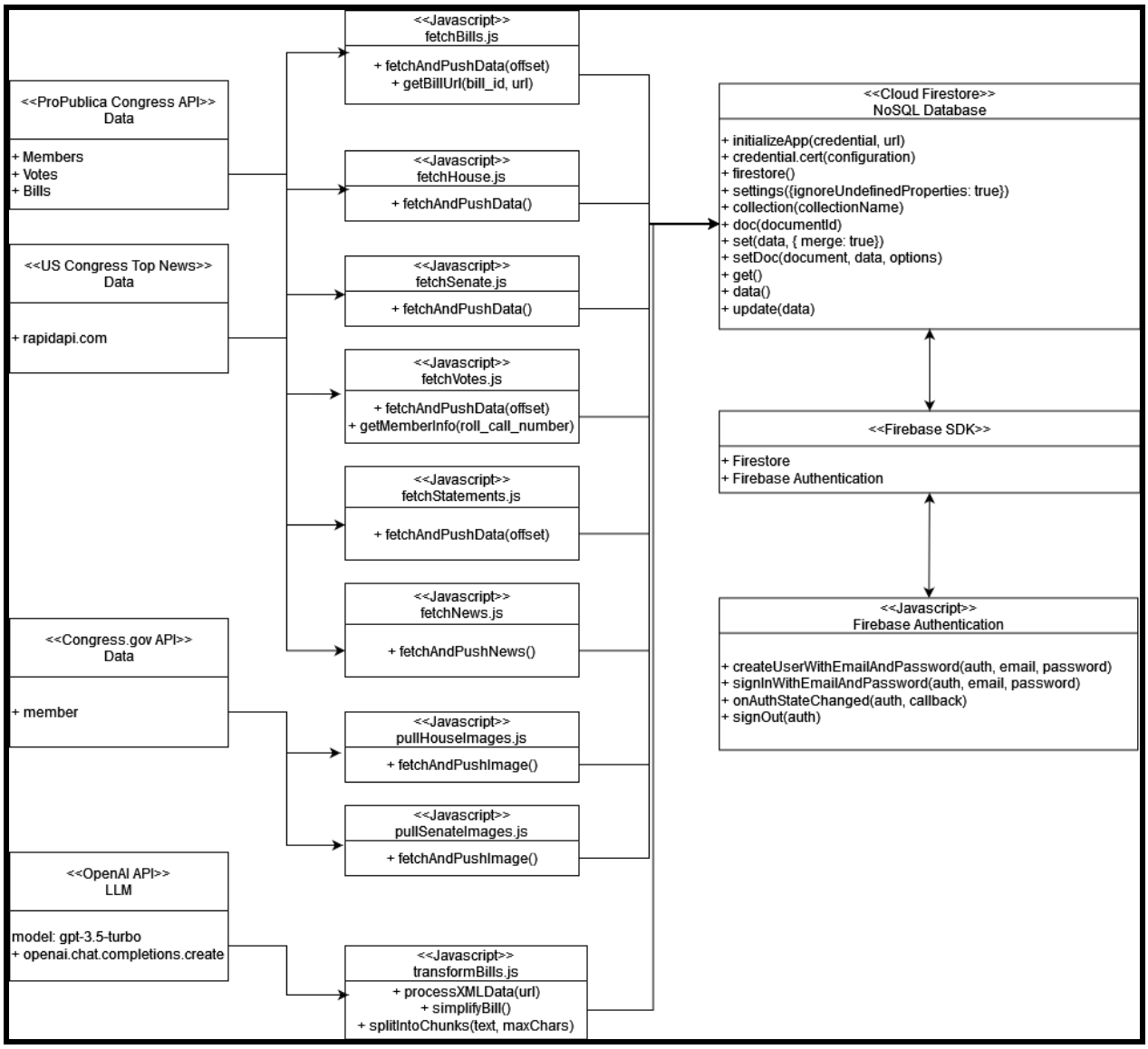

Figure 2: Backend UML for PolitiPulse.

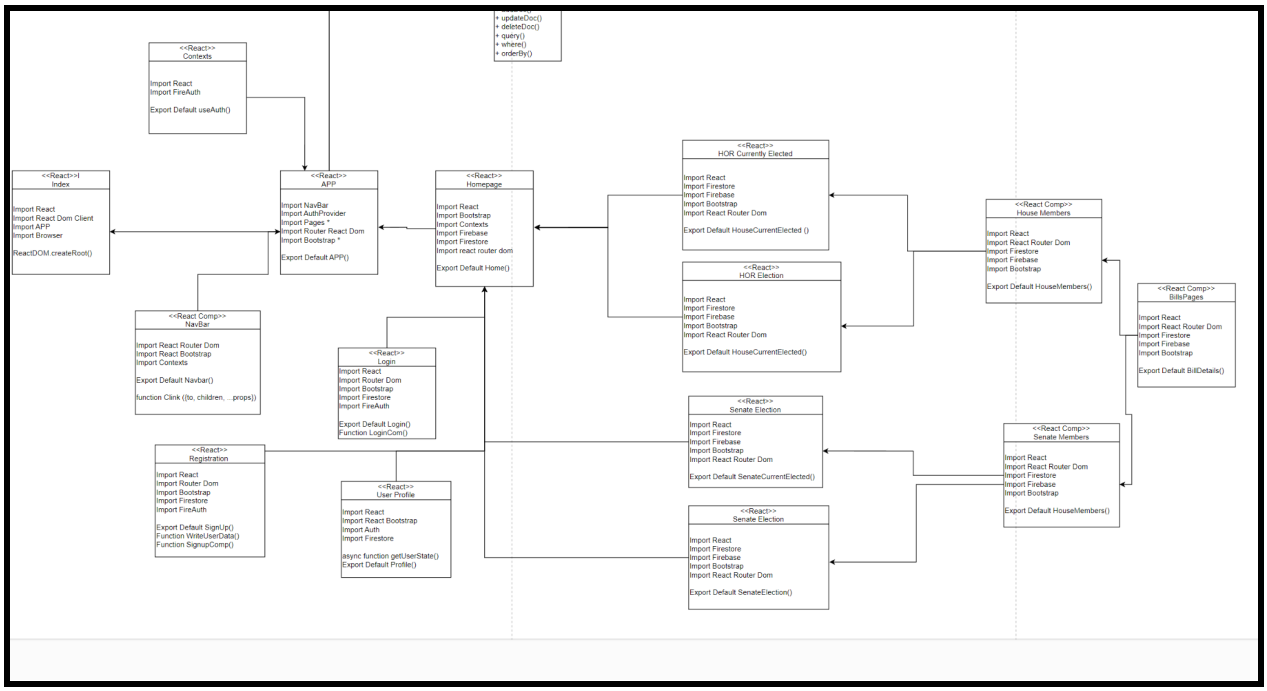

Figure 3: Frontend UML for PolitiPulse.