CAFEBABE: Alex Karle, Shahar Dahan, Elaina Bliss, Andrew McCluskey, Alex Cobian, Alfonso Merkl

Routing:
In app.js we route to the following pages:

- VotePage: localhost:8080/#/vote
  - This renders the Vote component
- SettingsPage: localhost:8080/#/usersettings
  - This renders the Settings component
- HomePage: localhost:8080/
  - This renders the Home component
- TrendPage: localhost:8080/#/trends
  - This renders the Trends component
- ArchivePage: localhost:8080/#/archive/1
  - This renders the Archive component for page 1
- ArchivePage2: localhost:8080/#/archive/2
  - This renders the Archive component for page 2
- CalendarPage: localhost:8080/#/calendar
  - This renders the Calendar component

React Components

- archive.js/archive2.js (two archive components to emulate pagination)
  - Renders 3 archiveEvents per page, in reverse chronological order
- archiveEvent.js
  - Renders an event for the archive, with its corresponding name, date, and summary
- Calendar.js
  - Renders a calendar view for events (buggy)
- home.js
  - Maps the home thumbnails for each candidate in alphabetical order on the home page.
- homeThumbnail.js
  - Pulls formatting information from the database and have a link to display a page with further information and a Twitter feed about each individual candidate. These are used in home.js to be displayed on the home page.
- settings.js
  - Displays the settings page and allows for users to make and save changes to their user settings.
- Sidebar.js
  - Allows us to dynamically render the sidebar. When a new page is opened through react router, the sidebar re-renders to show which page is active.

- Trends.js
    - Combines user and candidate info to fill in graphs to display statistical voting patterns of users over the weeks. Also uses react-chartjs [https://github.com/jhudson8/react-chartjs] as a place to store that information.
- Vote.js
    - Vote’s main purpose is to have a header and tabs which determine which candidates you will see and be able to vote on. Depending on which tab you are in, vote will either load the component voteDem, voteInd, or voteRep.
    - Vote also loads your user data from the db (getUserData) and will present who you currently vote for.
    - Vote also is the spot we set the function voteChange which will update the user data (setUserData) so that your “vote” property will be updated when you click on a vote button. This function is passed down as a parameter to voteDem, voteInd, and voteRep.
    - Lastly, vote has alerts which will pop up when you click a vote button (changing the state of “justVoted”). When you click a tab, the alert will disappear.
- voteDem.js
    - VoteDem will use getAllCandidatesOfParty to get all democratic candidates and then create voteThumbnails for each of them by mapping over the list of them in render.
    - VoteDem also loads the party logo and title.
    - Alerts appear here if the justVoted property passed down from vote turns true.
- voteInd.js
    - VoteInd will use getIndCandidates to get all independent candidates and then create voteThumbnails for each of them by mapping over the list of them in render.
    - VoteInd also loads the party logo and title.
    - Alerts appear here if the justVoted property passed down from vote turns true.
- voteRep.js
    - VoteRep will use getAllCandidatesOfParty to get all republican candidates and then create voteThumbnails for each of them by mapping over the list of them in render.
    - VoteRep also loads the party logo and title.
    - Alerts appear here if the justVoted property passed down from vote turns true.
- voteThumbnail.js
    - VoteThumbnail is the actual thumbnail that has a candidates image, name, and a vote button that has had the changeVote passed down to it through the vote and voteDem/voteDem/voteInd components.
    - Clicking the vote button will call the onVote method in voteDem/voteDem/voteInd which then calls the changeVote method in vote which updates the user.
    - VoteThumbnails load their color and info dynamically from the candidate object in the db.

Server Methods

- getUserData(userId, cb)
    - Returns the user’s data in the db
- setUserData(userId, newUserObject, cb)
    - Updates a user’s data in the db
    - Returns the updated data
- getUserPoliticalAffiliation(userId, cb)
    - Returns the political affiliation of a user
- getUserName(userId, cb)
    - Returns the fullName of a user.
- getCandidate
    - Gets a specific candidate with a given candidate id
- getSomeEvents
    - Used for pagination and only gets 3 relevant events from database
- getAllEvents
    - Gets all events from database
- getAllCandidates
    - Returns an array of all of the candidates
- getAllCandidatesOfParty
    - Returns an array of all candidates of a particular party (determined by parameter partyId). Used in the voting pages to only display certain candidates based on which tab you are in.
- getIndCandidates
    - Returns an array of all candidates of parties with id’s 3 or 4 (green or libertarian). Used to return all independent candidates in one method for the voteInd component.
- getParty
    - Returns the party object that has an ID that matches the parameter partyId. Used in the homeThumbnail component to get info on parties.
- getAllWeeks
    - Returns all weekly states for use in the trends section.
- getAllUserRaceGender
    - Used in trends to get only the necessary information (race and gender) to fill in the graphs

Pages

- Home (with candidate pages): home.js, homeThumbnail.js, sidebar.js (used in all)
  - Displays candidate thumbnails with links to a candidate page that shows up in front of the home page. The candidate page includes a photo, description, slogan, logo, age, party affiliation, and an embedded Twitter feed for each candidate. Andrew and Elaina collaborated on these aspects. These included updating the database to hold all necessary information for the candidates to be used in the candidate pages and embedding the Twitter feeds. Elaina handled the CSS and formatting as well as merged all of the CSS files for the individual pages and reformatted them into one file to be pulled by index.html and, therefore, all of the pages. Andrew made the sidebar used in all pages.
- Trends: trends.js
  - Displays weekly snapshots of users votes to display. Goes through ballotBoxes in weeklyStates to get which users vote for which candidates for that specific week. And then fills in the graphs based on known user information and whichever week is selected. This was done by Alex C.
- Archive: archive.js, archive2.js, archiveEvent.js
  - Displays events in reverse chronological order. Each event renders its name, date, summary, and corresponding party color. The archive page also uses pagination, allowing users to cycle through newer and older events. This component was done by Neal.
- Calendar:
  - Displays a page with a calendar view. Ideally this would have events be able to show up so that users can plan accordingly, but this is proving difficult currently. Currently, double clicking on a day that has an event tells you all about the events that day. This is being done by Shahar.
- Vote: vote.js, voteDem.js, voteInd.js, voteRep.js
  - Vote will display who you currently vote for and then give you options to vote for any of the candidates in our db. It has 3 tabs that allow you to narrow your search to either democratic candidates, republican candidates, or independent candidates. This is the only page where you can change your vote in the website. Alex K is responsible for the entirety of this page, its components, and the server methods related to it.
- Settings: settings.js
  - This page allows the user to store their user information including name, email, password, age, and other personal information. This information can be updated and saved. Alex K is responsible for creating the HTML and original css for this page (the static stuff). Andrew and Elaina

collaborated on making the page dynamic and updating the CSS and formatting of the page.

Bugs

- **Archive pagination: not written to support multiple pages/general pagination - currently written to only emulate pagination between two pages, which each have their own component, but server method exist to allow pagination to be general**
- **Calendar: Has an issue getting the events to display in the calendar view. The calendar we use has minimal, if any, support for adding events that can be viewed. I cannot figure out a way to make it so that a day with an event is displayed differently.**

Updated ER Diagram:

We changed the ER diagram minorly so that instead of events containing Ballot Boxes which contain vote objects that have user and candidate ids, we created a new entity called Weekly State which would store a date and a single Ballot Box which would store the many votes.

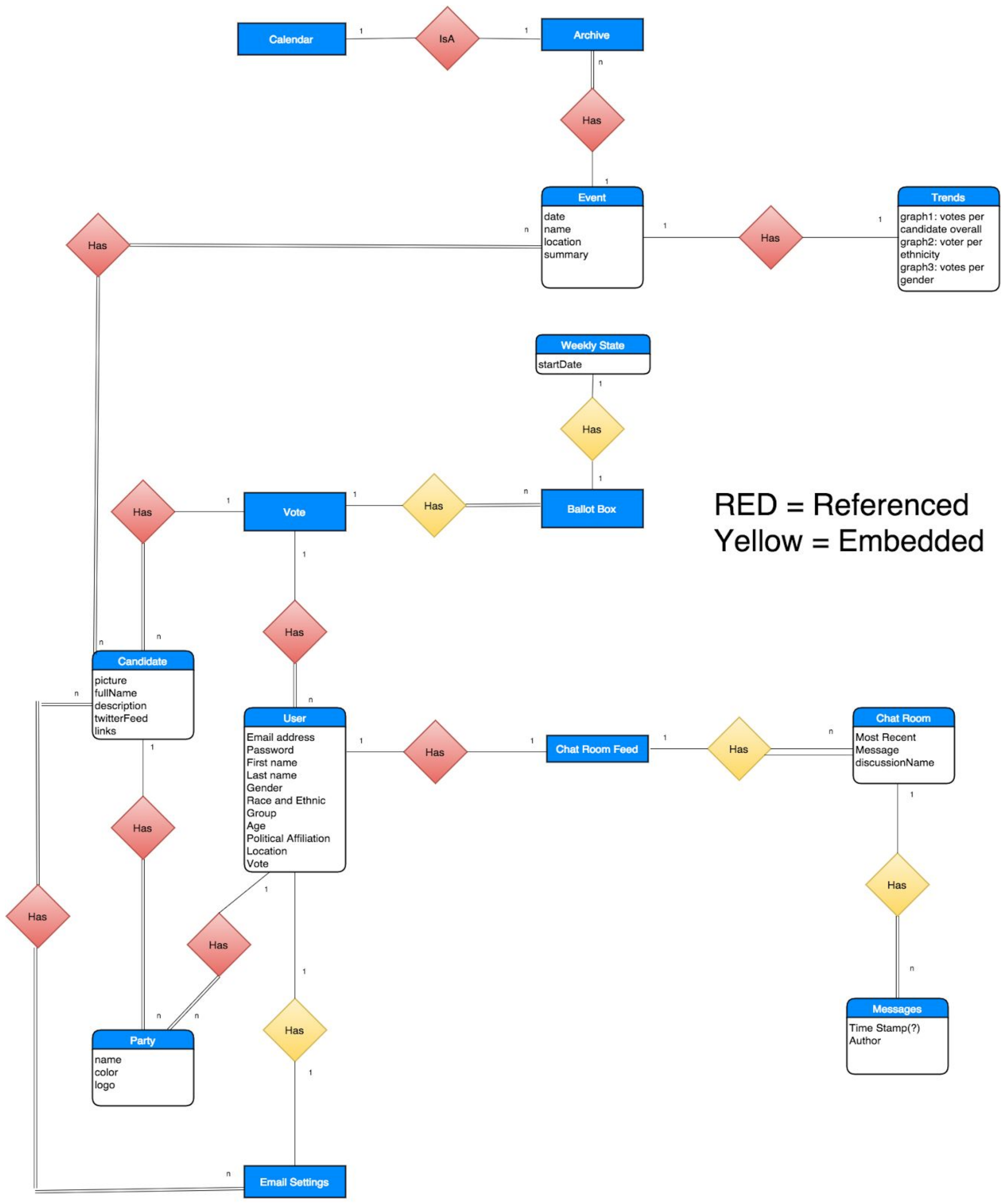
We decided to make this change because we realized that having users vote on specific events was a flawed system for the following reasons:

1. We wanted users to be able to vote independent, but the vast majority of upcoming events are primaries and independents don't have primaries to vote in
2. Due to the unequal distribution of events affiliated with parties, we run into a weird problem where some users may be voting more than others
3. Our events are affiliated with a party, so do we let users only vote for events with the party they are affiliated with or do we let them vote for any event? Either way is problematic--if we only let them vote within their affiliated party, we run into UI design errors where it's awkward and clunky for a user to vote outside their party (they would have to change their affiliation in settings) and if we allow them to vote for all events it would mean users could effectively vote for all 3 types of candidates at the same time.

We came up with an elegant solution, which we hope will be implementable on the database level (once we actually implement the database):

1. Instead of users voting on events, we have users vote on just candidates. We do this by adding a "vote" property to the user in the db which will contain a 0 if they have not voted and the candidate id of who they last voted for if they have voted.
2. To ensure that we have enough data to make trends interesting, we will save the state of every user's vote every week. We save this in a Weekly State entity which will contain a ballot box (the collection of votes from every user that week) and date. We store a collection of weekly state entities in the db to show change over time.
3. This allows us to show change in data over time in trends, one of our main features in the product.
4. We have updated our ER diagram to have this entity, and have updated our initial data to use it.

As Alex K was responsible for voting, he spearheaded this change process (with feedback help from Andrew). Alex K updated the ER diagram.


Calendar
IsA
Archive
Has
Event
date
name
location
summary
Has
Trends
graph1: votes per candidate overall
graph2: voter per ethnicity
graph3: votes per gender
Has
Weekly State
startDate
Has
Vote
Has
Ballot Box
Has
Has
Candidate
picture
fullName
description
twitterFeed
links
User
Email address
Password
First name
Last name
Gender
Race and Ethnic Group
Age
Political Affiliation
Location
Vote
Has
Chat Room Feed
Has
Chat Room
Most Recent Message
discussionName
Has
Messages
Time Stamp(?)
Author
Has
Has
Has
Has
Party
name
color
logo
Email Settings
RED = Referenced
Yellow = Embedded

Email Settings: Alex Karle's Honors Feature

Outstanding Bugs / Unfinished Parts:

There are no bugs with the UI or with the way the server changes the mock db and stores state; however, the actual emailing functionality has not been implemented yet. Subscribing to a candidate will actually update our db and will save this change; but the code does not actually email you yet. I will be implementing this in the future. I plan to do this using the npm package nodemailer https://www.npmjs.com/package/nodemailer.

React Components:

In app.js there is a component called EmailSettingsPage which simply renders the first main component, EmailSettings. I set the react-router to route to this page when the url was localhost:8080/#/emailsettings. EmailSettings is the component that is thus loaded into the main page part of our webapp when the url is like this. The only place that links to this url is in settings where there is a "notify me!" link that brings the user to this page. I decided to make it a separate page from regular settings so that it was isolated for the honors feature. The relationships and descriptions of the components rendered by the EmailSettingsPage are best described below:

- EmailSettings:
    - EmailSettings is rendered in app.js when the path is localhost:8080/#/emailsettings (because this routes to EmailSettingsPage which only renders EmailSettings).
    - EmailSettings' main job is to provide a basic html layout for the page (with a header and appropriate columns) and then to load candidateSub components.
    - It gets all the candidates from the database using the getAllCandidates server method, and sets the state of the component to be this list of candidates. It does this so that it can create a candidateSub component for each candidate.
    - In the render method, it will then create a candidateSub component (this is the actual subscription box) for each of the candidates, passing the candidate info so that the candidateSub component can load the info / color dynamically.
- CandidateSub:
    - The candidateSub component is the actual subscription box (a colorful rectangle) where the name, image, and party of the candidate are displayed alongside a button through which the user can update their email settings in the db.
    - Clicking the subscribe/unsubscribe button will call handleSubClick() which will call my own personal server methods subscribe() or unsubscribe() which will update a user's emailSettings in the db. emailSettings is a list of candidate id numbers which a user is subscribed to. It is contained in the user object in the db as described in the ER diagram (above). Subscribe will add the candidate id to that list and unsubscribe will splice the candidate out of the list (similar to a likeCounter but instead of adding your user id you add the candidate id that you want to subscribe to). When clicking the button, it checks if you are subscribed already and either subscribes or unsubscribes appropriately.

    - Another server method I created was getEmailSettings so as to read the current state from the db and update the state of candidateSub so that I could properly render the button as “subscribe” if the user was not subscribed to the candidate the candidateSub was about or “unsubscribe” if the user was already subscribed.
    - The state of candidateSub also includes party so that I could get the candidate’s party and resolve it so as to display the full name of the party (candidate objects have references to parties, not the actual name of the party).

<u>Server Methods:</u>

- subscribe:
    - Takes in a candidate id and a user id and a callback. It will find the user matching the userId and then add the candidate id to that user’s emailSettings property in the database. It then emulates a server return with the emailSettings list of candidate id’s.
- unsubscribe:
    - Takes in a candidate id and a user id and a callback. It will find the user matching the userId and then remove the candidate id from that user’s emailSettings property in the database. It then emulates a server return with the emailSettings list of candidate id’s.
- getEmailSettings:
    - Takes in a user id and returns that user’s emailSettings property which is a list of candidate id’s that the user is subscribed to. Used for rendering the current state of the user’s email settings.
    -

Chat: Shahar Dahan's Honors Feature

Outstanding Bugs / Unfinished Parts:

Currently, the chatList does not render the last active time correctly, but I plan on finishing that using the 'moment' npm package. Additionally, the CSS is slightly messed up right now which causes the chat page to look a little weird. This just involves tinkering around with it in the future. Additionally, it would be nice if the Chat could have a way to add other debates as users wanted to, but in the meantime it works well, taking the user's name and party affiliation to color their username appropriately, and messages are displayed and saved in the database like they should be.

React Components:

In App.js, the website routes to chat with localhost:8080/#/chat. This renders a 'chatList' component, which was necessary in order for the Chat's themselves to be dynamic and change on request from the user. The ChatList will render the Chat component, depending on which is clicked, and then will render the messages component in the Chat box. Messages did not need to be a separate component, until I realized that they would change depending on who was writing the message. This whole process is linked to at every state of the game through the navbar we have located on the side of the page. The chat feature is still an individual aspect of the project, so it should still be isolated enough. Here is a more step by step process for how the components are rendered.

1. ChatList
    - This is rendered straight through the App.js page. This will automatically load a dummy chatBox (which happens to be Hillary Clinton, but might be the last one the user used in the future). Then it proceeds to get all the ChatBoxes in an array, and display their name, and in the future the time which they were last active for. This is all done using the getAllChat server method. Upon clicking on any of the names, the page will re-render the ChatBox, which will update the title of the page slightly, and give you a new chatBox. ChatList is responsible for keeping track of the active chatBox that it renders. Upon rendering a chatBox, it passes it the state.active value so that Chat knows which to render.
2. Chat
    - This uses a server method called getChat, which takes in the passed down active value (stored as toRender) and the callback function which simply updates the state. It loads all the messages. It does not currently do anything about the most recent message it receives, but it does handle sending the messages to the database using postMessage, which was modelled after the facebook method. This ChatBox will render the MessageEntry and the Messages.
3. MessageEntry
    - This handles creating the interactive form at the bottom of the Chat. It enables the user to press the pencil button to write, or to simply press enter. It prevents the user from entering any blank messages. It uses OnPost which is passed back to the Chat box.
4. Messages

- This simply displays the message. It checks the user’s party affiliation and colors it appropriately. It also gets the user’s name and uses that instead of the ID to display the message. It has no server methods.

<u>Server Methods Created:</u>
getChat(id, cb)
Gets the particular chat with all of it’s properties.

getAllChats(cb)
Gets every chat in order to populate the chatList.

postMessage(chatBoxId, author, contents, cb)
Posts a message to the appropriate chatBox. Will also save the author which is . used by messages to get party affiliation and username.

getUserPoliticalAffiliation(userId, cb)
This was created for the chat, but can have applications elsewhere. Using a userID, will get the political affiliation.