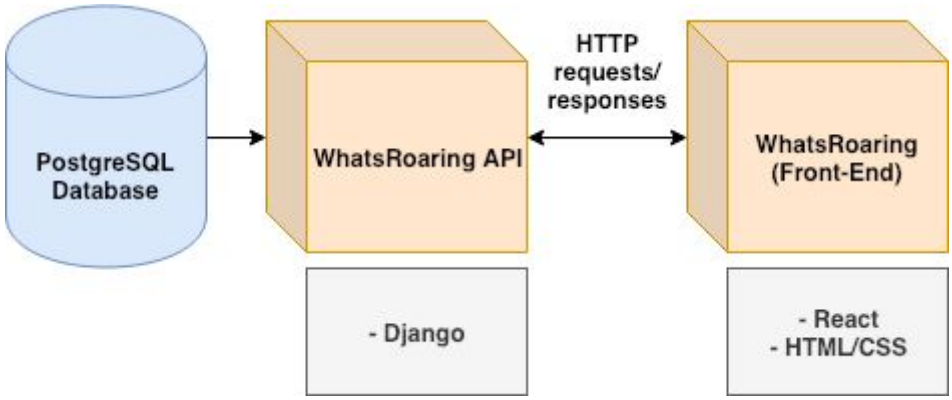
whatsRoaring

Programmer's Guide

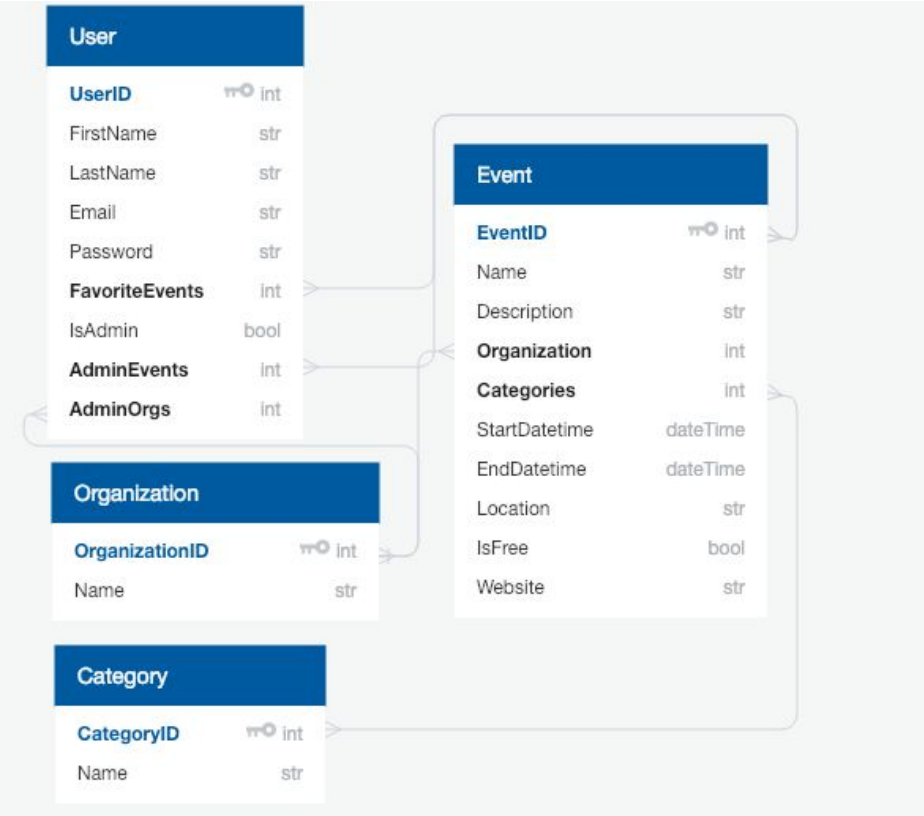
System Overview

Our application, WhatsRoaring, is a three-tier application. The three tiers are a PostgreSQL database, a Django-based backend, and a React-based frontend.



The PostgreSQL database stores information about events, users, categories, and organizations. The database communicates with the Django backend through SQL queries. The Django backend is built to receive RESTful POST/GET requests from the frontend, retrieve the desired information from the database, process and format that information as necessary, and return the requested information to the frontend.

Database



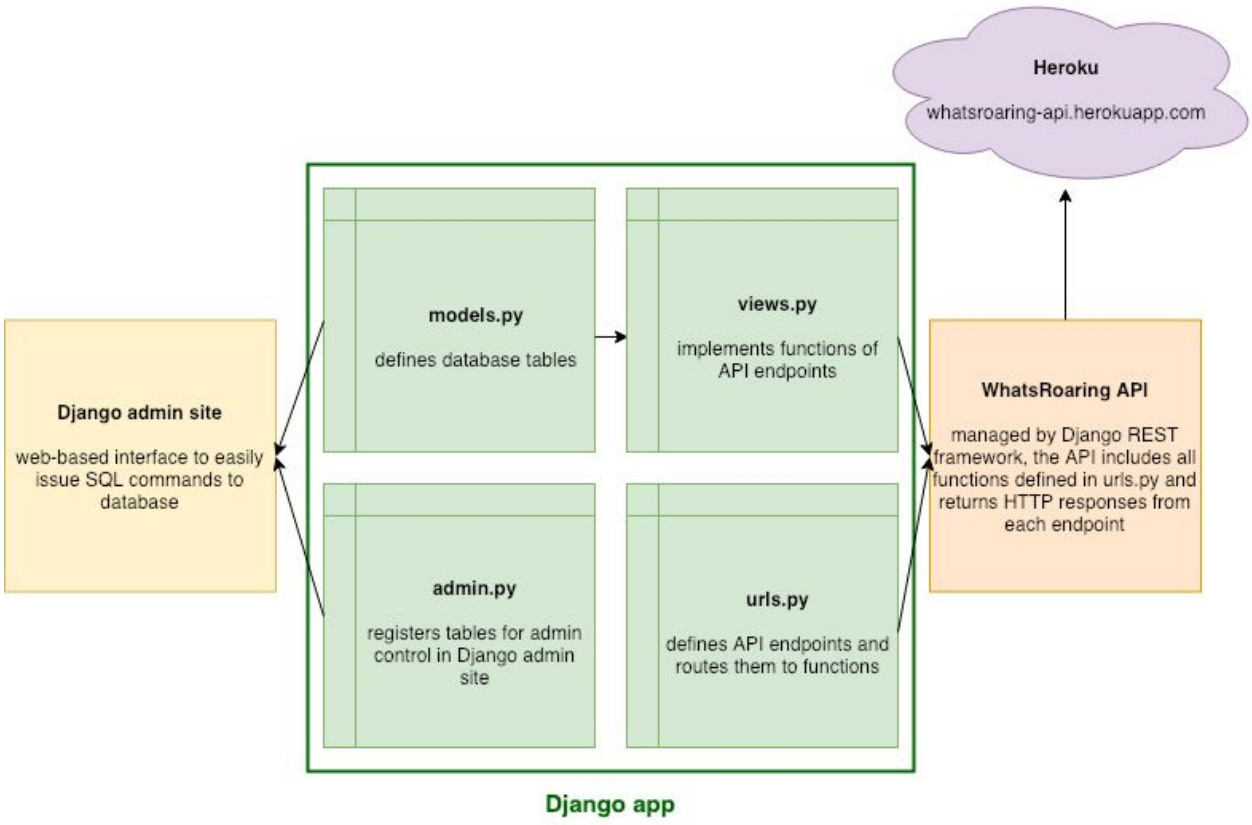
The database contains four tables: Event, User, Organization, and Category. The Organization and Category tables simply store the names of organizations and categories. Every event is associated with at most one organization and any number of categories, so we decided to create separate abstractions for these two models.

The Event table stores information about events. Each event has a name, description, location, start time, end time, website, boolean indicating whether it is free, organization, and list of categories. The event name, start time, end time, and boolean indicating whether it is free are required fields. The organization field uses OrganizationID as a foreign key into the Organization table, as each event is organized by exactly one organization. The categories field is a many-to-many field with CategoryIDs as values.

The User table stores information about user accounts. Each user has a first name, last name, email, encrypted password, boolean indicating whether he or she is an admin, favorite events, admin events, and admin organizations. Favorite events and admin

events are many-to-many fields containing EventIDs as values. Admin organizations is a many-to-many field with OrganizationIDs.

Backend



The backend is a Django app with a Python codebase. The most important files are under “calendarapp/” and define the database tables, functions, and URLs.

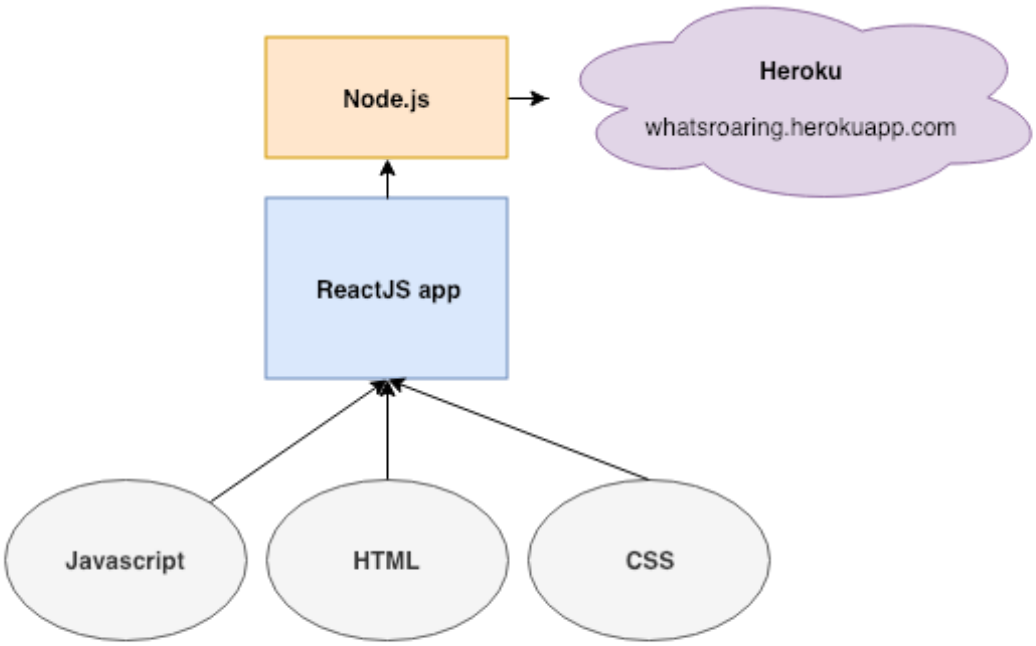
Database Calls

In settings.py, we specify our PostgreSQL account URL so that Django can read and write to our database. We can easily define new tables in models.py, where each class is a table with many types of fields available, including ForeignKey, ManyToMany, and OneToMany. Django provides an admin site for every app, which we can access at whatsroaring-api.herokuapp.com/admin. The site offers an interface for admins to view and edit tables in the database. The file admin.py sets up the tables to be accessed by the admin site.

Django REST Framework

The backend is set up as a RESTful API. With the Django REST framework, all we have to do is define functions in `views.py` and URLs in `urls.py`. That way, we can create a new API endpoint by writing a function, such as `getEvents()`, in `views.py` and adding "/getEvents" and getEvents to `urls.py`.

Frontend



Overview

Our frontend is a ReactJS app, which is built on JavaScript, HTML, and CSS. The React app is compiled using Node.js and deployed on Heroku as whatsroaring.herokuapp.com.

It consists of React components that render either full pages or small components (such as a button or dropdown menu). These components update their states responsively. Every component renders itself through HTML code and CSS styling. Components may make HTTP GET/POST requests to whatsroaring-api.herokuapp.com using the Axios JavaScript library.

Components

React apps consist of components, which include their own states, props, and HTML rendering instructions. We categorize components as Pages or Components (not to be confused with React components), where Pages contain the HTML code for an entire page, and Components render a small, reusable element, such as a button or dropdown menu. Every component may have an associated CSS file for styling.

The main App component consists of all the pages that are visible to the user. React Router allows the app to route each page to a specific URL, such as the Calendar component to "/calendar". Each Page, in turn, calls the necessary smaller components for that page. For example, the Landing page has *about*, *team*, *timeline*, and *documentation* sections, so it imports the About, Landing, Timeline and Documentation components to populate the page, in addition to the rest of its HTML rendering code.

Components responsively update their states, but when setting the state, it may be necessary to write a callback function. A callback function is called only once the state has been set. Since state setting is an asynchronous process, the programmer should make sure that any functions depending on the updated state are called only after the state has truly been updated.

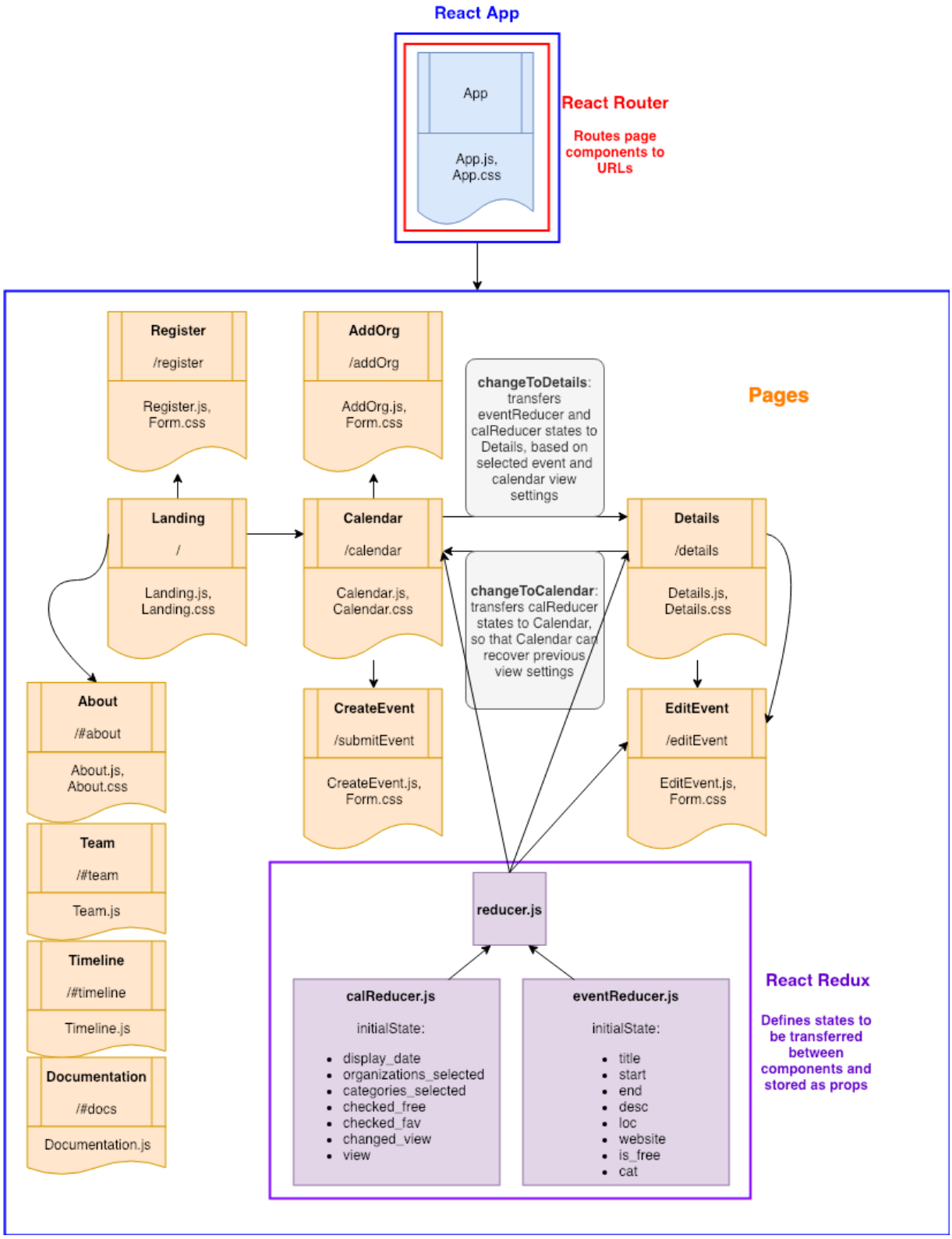
HTTP Requests

We use the Axios JavaScript library to make HTTP GET/POST requests to the API (whatsroaring-api.herokuapp.com). Axios makes it easy to input the parameters to be sent to a URL, as well as define how to handle HTTP responses.

Redux

Components may pass information between themselves using the React Redux library. We have defined two reducers: `eventReducer` and `calReducer`, which contain states that can be passed between components. To use Redux, a component must have `mapStateToProps` and `mapDispatchToProps` functions. The `mapStateToProps` function maps reducer states to the props of the component. The `mapDispatchToProps` function transfers reducer states from the current component to the next component. An example usage of Redux is in Calendar and Details. When the user selects an event, the Calendar page must transfer information about that event to the Details page. In turn, the Details page must transfer to the Calendar information about the previous calendar view

when the user decides to return to the Calendar page. This design decision is detailed in the [Interesting Design Problems] section.



Design

In addition to our own HTML/CSS techniques, we utilize React Bootstrap and Material UI design elements. Every component that uses such elements includes the relevant imports at the top of its JavaScript file.

Interesting Design Problems

In this section, we explain interesting design decisions of our app.

Overall Design Choices

Heroku: We chose to use Heroku for ease of deployment early in our planning process. Because of this decision, one unsurprising design decision was the choice to use a PostgreSQL database, since Heroku requires PostgreSQL as the database management system.

Django: We chose to use a Django backend due to its compatibility with Python and its popularity. It makes it easy to interact with our database by simply defining models and views. The Django REST framework also provides an easy way to create API endpoints.

React: We chose ReactJS because of a neat open-source calendar component, so that we could utilize that and build on top of it. Our app is structured as a standard React app, with a central App calling Pages and Pages calling Components.

Decoupling Frontend and Backend into Two Apps

One interesting design problem was the issue of deploying a single Heroku app with Django backend and React frontend. The app refused to build and would not deploy, even after several attempts to rework build packages with TA help. With the input of TAs, we chose to split our app into two apps: WhatsRoaring (frontend) and WhatsRoaring API (backend). Django turned out to be a good backend choice for this because of its REST framework. Each app is deployed as a separate Heroku app.

The frontend app makes calls to the back-end for information (e.g., events to populate the calendar, or categories to populate the filtering dropdown). The backend app makes calls to the database and delivers that information to the frontend upon request.

This posed a problem when attempting to integrate CAS authentication into our app. The challenge originated from the difficulty of transferring username information between the frontend and backend of our app. After confirming the problem with the CS staff, we concluded that authentication and the ability to link events with users was necessary to the app. This user-specific data, used for letting admins edit and delete their own events, and for letting all users favorite events, was crucial to our app's functionality. We decided to implement our own authentication system. The system was implemented using Django because of its built-in password encryption.

Minimizing API Calls

One of the most important design decisions was choosing to minimize calls from the frontend to the backend, as those calls are resource-heavy (note the time it takes for the calendar to initially populate). We wanted our app to work as quickly as possible, so when data could be stored in the frontend and passed between frontend pages, we took that opportunity.

Most notably, the Calendar page has already called the backend for relevant information on each event, including the events' times, locations, etc. when the calendar is populated. This information is needed by the Details page, so instead of making a call to the backend, the Calendar page packages the event's information as a prop (see mapDispatchToProps) and sends it to the Details page. The Details page reads the prop and is then able to populate the page with event information. Similarly, the Details page can package information to send to other pages — the `changeToCalendar` method packages information that allows the calendar page to retain its state (in time and the choices for filtering), and the `changeToEditEvent` method packages the information necessary to pre-populate the EditEvent page.