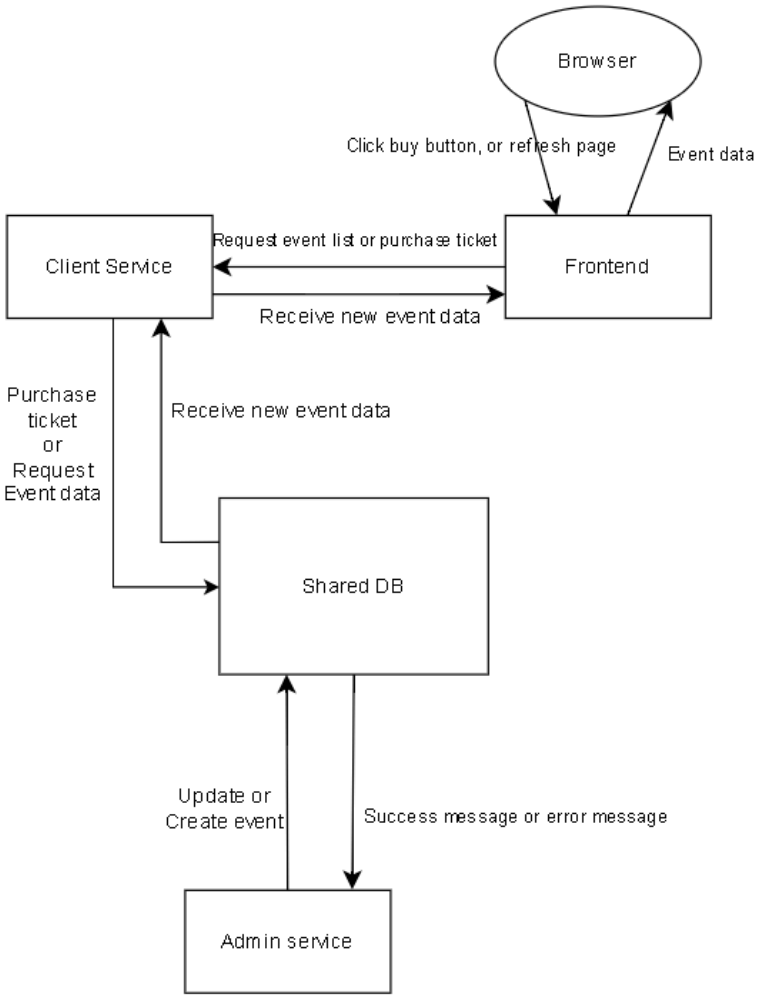

The frontend talks to the client service via GET and POST API calls for requesting events and purchasing tickets for events. When the page's Buy Ticket for (event name) is pressed a POST API call is made, which causes the client service to accept(if tickets_total for event is not equal to the tickets_sold value for event in the database) or fail(if they are equal). When the client service responds then the frontend updates every event "card" in the event list that has been returned by the API call, or if it fails it will return an error message.

Both the client service and admin service use the same database, both can update the database whenever a POST request is made successfully(and PUT request for updating an event as an admin).

Database Concurrency:

When updating the database, a concurrency problem like a race condition likely would not be possible, due to various factors like the database architecture as well as choices made with the database constraints.

Because SQLite is used and we are not using WAL, and a check constraints for the tickets_sold require it being between or equal to 0 and tickets_total, it would be impossible for a raise condition to occur with the tickets_sold value, as you would not be able to set the tickets_total value lower than tickets_sold or vice versa. In any case where this is attempted an error would be thrown.

Notably with Atomic updates:
Since we only have one table and are doing one update at a time, we are using atomic updates, even though we do not use “START TRANSACTION” within our code.

Accessibility:

There all accessibility requirements are implemented:

- Use semantic HTML elements: Ensure headings, lists, buttons, and labels use proper
- HTML tags.
- Add ARIA attributes where appropriate to describe UI components for screen readers.
    - Notably aria-live is set to assertive for notifications, but otherwise polite
- Ensure keyboard navigability: All interactive elements (buttons, links) should be accessible via keyboard (Tab key).
- Provide visible focus indicators: When tabbing through elements, it should be clear which element is focused.
    - There is a black outline when doing so
- Use meaningful button labels: For example, instead of just “Buy Ticket,” use labels that screen readers can read clearly.
    - “Buy ticket for ” event name
- Add alt text for any images or icons (if applicable)
    - No images or icons

Code quality:
1. Function Documentation

- Function headers
    - These follow the code quality standards, though it should be noted that these look a bit unique for the frontend as prop-types is used since it was insinuated to not use typescript(in the diagram the main file for the javascript code is titled App.js rather than something with a .tsx extension). Most documentation standards for react are oriented around typescript, so there are a few odd looking labels, though most appear when highlighting values in an IDE.

- Inline comments
    - There aren't many inline code comments as there isn't much complicated code as explained in code quality. It should be noted that when there is the most frequent use is to explain why when used

2. Variable naming
    - There are few instances of single or vague variable names unless temporarily used in a loop. Here is an example of 2 vague names within a line:
        - setEvents((prev) => prev.map((e) => (e.id === event.id ? event : e)));

    Since this is in effect a loop, and e is the values being the incremented values, it is clear that they are the "old" event values. Prev is simply just the previous state.

3. Modularization
    - All of our code is modular and broken down into smaller functions.
    - All of our code uses the concept of "Seperate concerns"
4. Error Handling
    - All inputs are validated before being inserted into the database
    - There are errors thrown respectively
    - There are countless numbers of try-catch blocks
    - The error messages are clear for any realistic problems that may occur
5. Consistent formatting
    - Spacing is consistent
    - There are frequently breaks in code for readability
    - A linter was used(this is how prop validation was discovered)
6. I/O Handling
    - All API requests that can be done are written with examples within the README file, as well as an example