Hi! I'm going to review a few points here so you have a better understanding of why I made certain decisions.

1. **Data Structures, time complexity, and clientIDs**
    1. I made the internal data stores both objects. I knew I wanted to try and devise ways to reduce time complexity as much as possible, so I wanted to leverage direct lookup with constant time complexity as much as possible, rather than iterating even, though that is usually more intuitive.
        1. I'm assuming that a lot of clients (millions) could hit this server, so interfacing with the boundsObj which holds the client data needs to be fast.
        2. I'm assuming that the number of buoys is much smaller (less than 3k), so while time complexity is nice, it isn't as crucial here as it is with the other data structure.
        3. I'm also assuming, while possible, that it won't be likely you will see very many requests coming in for buoys being added or for updates, its just not realistic. Swells won't be changing their height and period on the millisecond.
        4. What will change on the milliseconds is people scrolling and zooming that map like children (I do this), so the subscribeToBounds needs to be as streamlined as possible.
        5. That said, all time complexities are either constant or linear, no more.
    2. I wanted to do a direct lookup on the boundObj, so the way I came up with doing that was to store the clientID as the key to each client's object on the boundsObj data structure.
        1. Honestly, all remaining data structures and their manipulations were based off of that goal.
        2. But how would I know which client was sending me new bounds? The key to that is that the socket.id lets you keep track of the connection, but more on that later. At first I was going to try and make a clientID, I started thinking of who (client/server) was going to create the ID, and how it was going to be stored. I then came up with the idea of the server creating the clientID, sending it down to the client, on each connection, and then updating a sessionID property on the client's object with the socket.id each connection/reconnection. However, the rules for the project showed the request for the subscribeToBuoys without a client ID parameter, so in an assumption that you did not want me to add any other parameters, I had to come up with another idea. I ended up using the socket.id as the clientID because it is available with every connection/request, so I stored each client object with their socket.id as the key.
            1. However, you also have to think about lost connections. Each time the user disconnects and reconnects, the socket.id changes, which means we would lose all of their previous data. That's not ideal because we would have to resend them all new data every time a connection dropped, and we would have a build up of memory leaks in the server, as well, because we would lose control of that user's client object.
                1. To solve this, I sent the socket.id down to the client on each connection. The client would then store that id in the Redux store, but more importantly in the Browser's LocalStorage. Also, on new connections (or reconnects), I have the client emit a request with their clientID (old socketID before connection loss), I then go and replace the old clientID on the client's object with the new socketID. Then I send it back down to them. So there is this constant refreshing of the ID as clients reconnect. This ensures that the server always has the most up-to-date socketID and the client does as well. I could have done this with a separately generated ID, and that would have worked fine with us just updating a sessionID property each reconnect to keep track of the current socket.id (We need the socket.id for future buoyNotifications, so

we know who to send those to). The issue would have come when a new set of bounds came in (socket.id is the only indicator of who made the request), and we would have to look a client up on the boundsObj. We would have had to iterate through each client object and check their sessionID against the request's current socket.id. We no longer can do a direct lookup since the sessionID would be nested in the object. That ruins my whole goal of keeping the time complexity suppressed. So using the socket.id as the clientID allows us to do a constant time complexity lookup when a new bound is sent in, as well as still keeping track of the socket.id when we need to emit a buoyNotification to specific clients.

3. Sorry, I know that was wordy, and a bit weird to think about, but hopefully that makes sense.

3. Updating buoys and checking if a buoy already exists on the addBuoy method are also constant because the name of the Buoy is the key of their object as well, so incoming updates will happen quickly.

2. Promises

1. I wrapped each method in a promise, so that as concurrency is increased those methods won't block others. I didn't get a chance to test concurrency because the test harness wouldn't connect with my server. I will discuss that further in the test section.
2. One item I would be concerned with is race conditions between methods, but I didn't really have a way to test that in the short term.

3. subscribeToBuoys Method

1. I fire off a request on each map movement or zoom to update the server with the new bounds. I chose to do this over waiting until the movement stops because I wanted to give the user a more dynamic feel vs waiting until they stop moving the map to actually see buoys update.
    1. The result of this is A LOT of requests getting sent to the server. While this can be an issue if many users are sending many requests, I didn't notice any slowdown on one client. If need be, we could debounce the requests, or find ways to reduce blocking and time complexity on the server.

4. Updating the client when buoys are added or updated

1. When an addBuoy method is invoked, the server takes that buoy, iterates through all of the client objects checking their bounds while doing so. If it finds a client whose bounds wrap that buoy it will add the buoy to their buoy object and then push a buoyNotification to them with the new buoy.
2. When an updateBuoy method is invoked the server will iterate through each client checking their buoy objects for the updated buoy. When it finds one, it will update the height and period, as well as send a buoyNotification
3. We needed to store the socket.id on the client's obj for this reason, so we could individually reach out and update them when there were buoy updates.

5. Extra Credit

1. When bounds change, only send down buoys the client doesn't already have.

1. I do this buy copying the old buoy object before the update, then diff'ing the old object against all the buoys in that bound, if a new buoy is found that wasn't on the old object, I send a buoyNotification. If one was already in the old object, I don't send the notification.
    1. However, I do blow out the old client buoy object and set brand new properties on it for each bound subscription that comes in.

2. Pre-caching

1. I do this by artificially adding onto the bounds once the request hits the server.
    1. I calculate the distance between bound corners using the distance formula, then I increase each bound by that distance. The assumption is that if the user scrolls

in any direction by one viewport length, they will have buoys waiting for them. (Close enough)
    2. I chose to use the calc'd distance over an arbitrary distance to take into account the fact that the cache will need to be greater or smaller depending on the zoom level, so as to not over or under cache. All boundaries are within the the normal lat/lon limits.
2. Also as an added feature, I never remove the buoys that have been rendered on the client. So once a user views an area, they will be able to leave and come back to the area, and see buoy icons. While that buoy data may be stale, until the response returns, it at least gives the illusion of real-time snappy responses. When the response hits, the stale buoys will update.
3. Based off of the wording in the project rules "For updated subscriptions, you might choose to optimize by only initially returning buoys that exist within the new bounding box that didn't exist in the previous bounding box" I was going to make a separate client bound and a cache bound with only the client bound receiving updates from buoys that are added or updated after subscription (so the extended. cache boundaries wouldn't have received updates after subscription). However, after thinking about it, it wouldn't make sense to only update the viewport bounds, but not update the cache as well. We would want to keep the cache area up-to-date as well. So I decided to make one bounds object, the cache bounds which includes the inner client viewport as well as the extended portion.

6. Testing

1. I wasn't able to hookup with the testing harness. I believe because I used socket.io which builds upon the native websocket implementation. From what I have read, you need the socket.io client to interface with a server using socket.io, or you need to create your own similar implementation. I didn't realize this until after building the application. My client has access to the socket.io-client which allows it to interface.
2. So I used some of your static data, as well as my own exploratory testing to test it as best as I could. There were no errors when running the static data.
3. I was not able to test how the application handled concurrent requests due to not being able to use the harness, so please take that into account when reviewing this.
4. You can find my static test file in the repo, I looped over arrays of data sending a request for each one with no time-interval set.