Beer Buddy

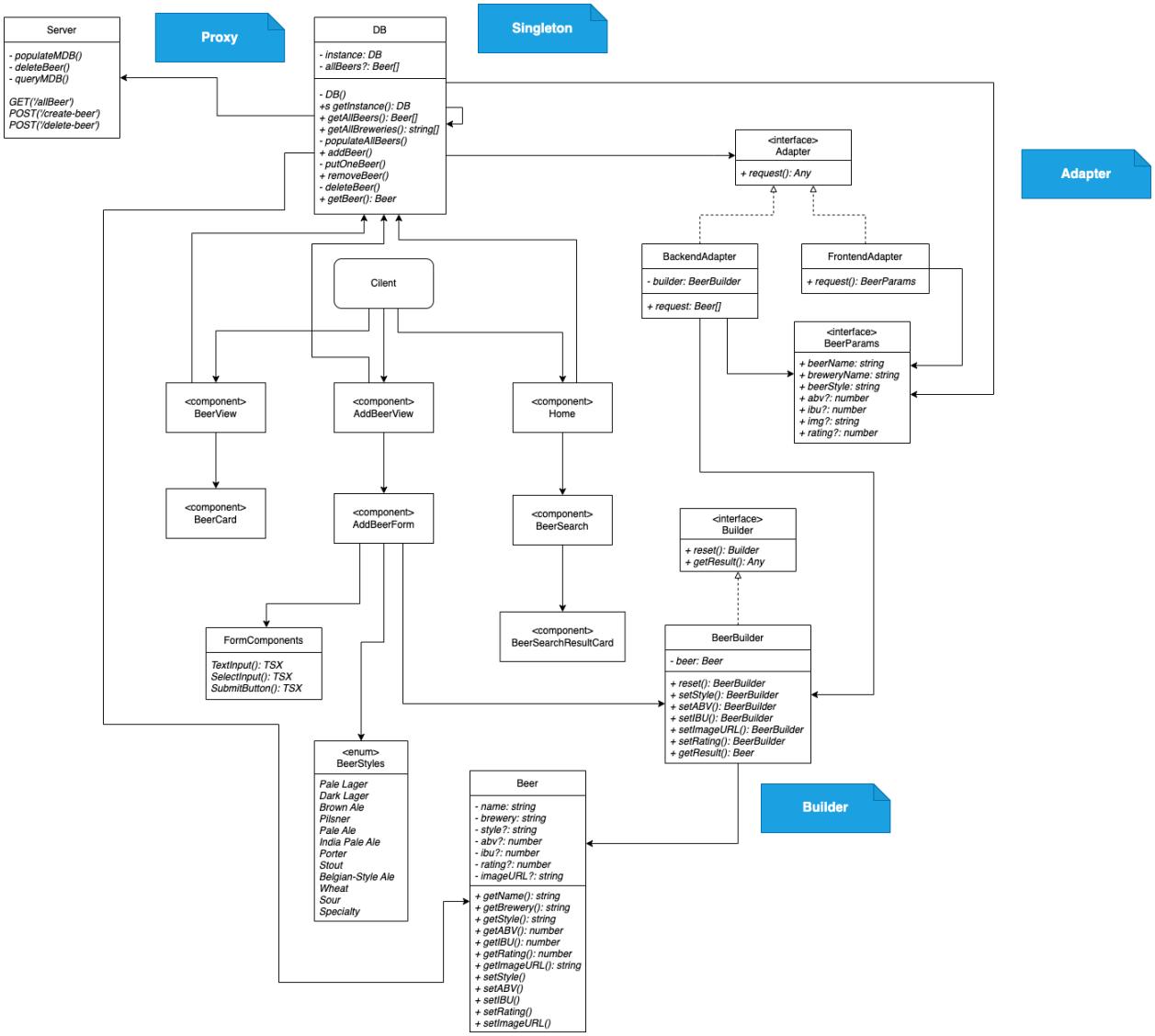
Team Members: Philip Knott, Alex Moss

Final State of System Statement

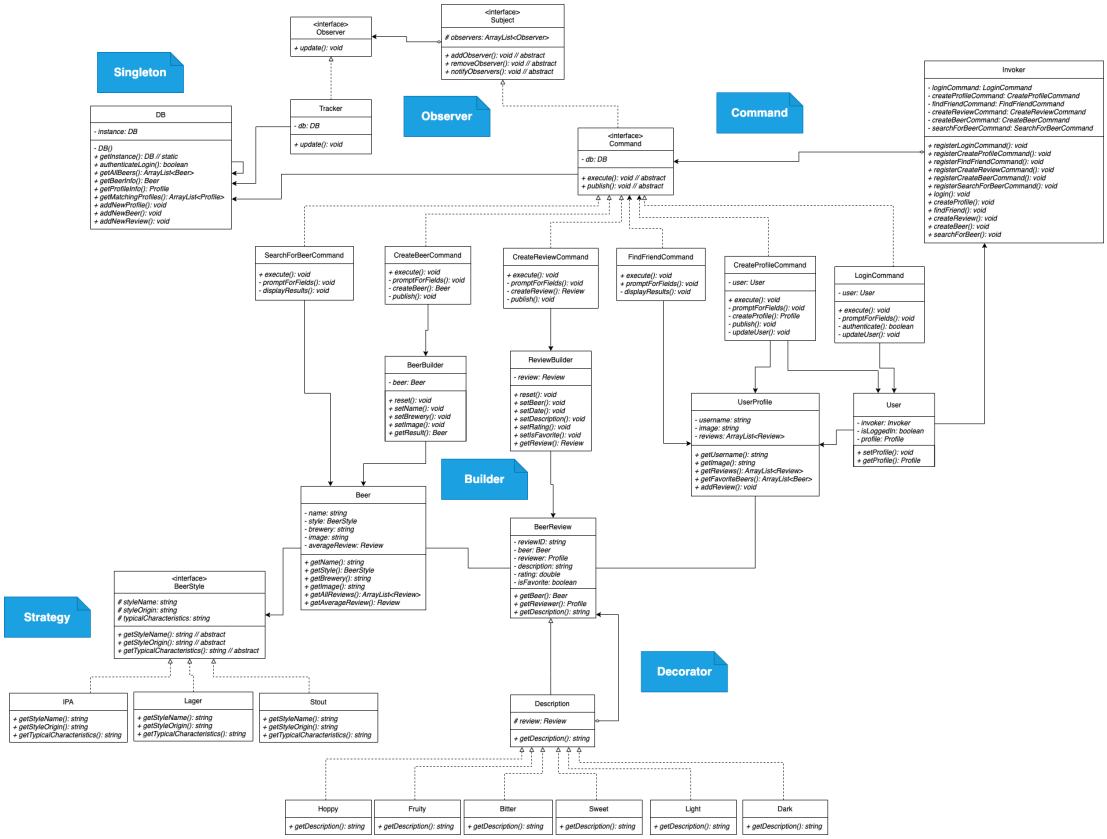
The final state of the system was less than we had initially planned to do, but still contains the main critical feature of our original design and remains a usable platform. We realized that we were too ambitious when we planned to add user accounts and individual reviews tied to each account. We decided that instead there will be a rating system for each individual beer, and that the rating will be anonymous with the idea that one user will be using the site for themself. The scope was scaled back primarily because adding users seemed a bit unnecessary for a rating system. This website serves more like a simple application one can use to keep track of their favorite beers instead, which we believe still accomplished our initial goal.

Final Class Diagram and Comparison Statement

Final UML Diagram:



Project 5 UML Diagram:



As seen above the current system is much smaller than we had initially planned for. This is mostly due to the removal of the users in general and breweries as objects, which corresponded to many patterns. We were planning to use the decorator pattern to append tags to reviews, but since there were not multiple users, there was no need to append multiple reviews, which also mitigates appending more tags. We were planning on using the command pattern for all the users interacting with searching, adding beers, and modifying beers, but we found that it actually wasn't as practical as we initially thought. React mostly did all of it for us, so we thought it wouldn't make sense to do it in a separate pattern. Observer was actually used by using states in React, but we did actually end up using state changes and logging features from React, and we did not implement our own classes for this, so we decided to remove it from the UML diagram. We decided to add a proxy to help keep the frontend and backend communicating indirectly through a proxy because we wanted to keep them on separate ports but we wanted them to communicate with each other, and proxy did an excellent job at this. Finally we added an adapter to again help with the backend and frontend communication. Since we need the data in the front end to be of the object "Beer" we needed to transform the data each time it was sent to or from the backend. The Adapter pattern allowed us to have a simple class that did this for us, so if we needed to change something about how we pass data, it was in this single place.

Third-Party Code vs. Original Code Statement

Other than referencing the documentation for each of the third party frameworks listed below we created this web application from scratch. We referenced these simple to understand the inner workings of each, and how they work with each other, as they are frequently used together and have lots of resources in their documentation that references one another.

Typescript: https://www.typescriptlang.org/
React.ts/React.js: https://reactjs.org/
Node.js: https://nodejs.org/en/docs/
Vite.ts/Vite.js: https://vitejs.dev/
MongoDB: https://www.mongodb.com/
Express.js: https://expressjs.com/
Axios.js: https://www.npmjs.com/package/axios
Bulma: https://bulma.io/

State on OOAD Process

Singleton: This was very helpful in maintaining a single source of truth for data. By using only one instance of our DB class, we were able to cache our data on the frontend, while also providing ready access to make updates to the backend from anywhere across the frontend of our codebase.

Frontend UX/Form inputs: Preventing bugs with react hooks and event handling. By designing our frontend using reactive listeners, we were able to prevent potentially bad user input from entering into our backend by constantly updating the user interface based on the state of the user's current input.

Adapter: This was extremely helpful to reformat data coming from the backend or going to the backend because this was a key element of the program. Each time data had to be sent, it had to be modified to fit the parameters of the backend. Having a single class that simple did it automatically was very useful when we wanted to add more data to the backend calls, or reduce the amount of data being passed.

Proxy: Proxy was very fun to learn and it was extremely unique in the fact that it allowed the backend to extend the frontend so easily. It made connecting the backend and frontend so seamless.