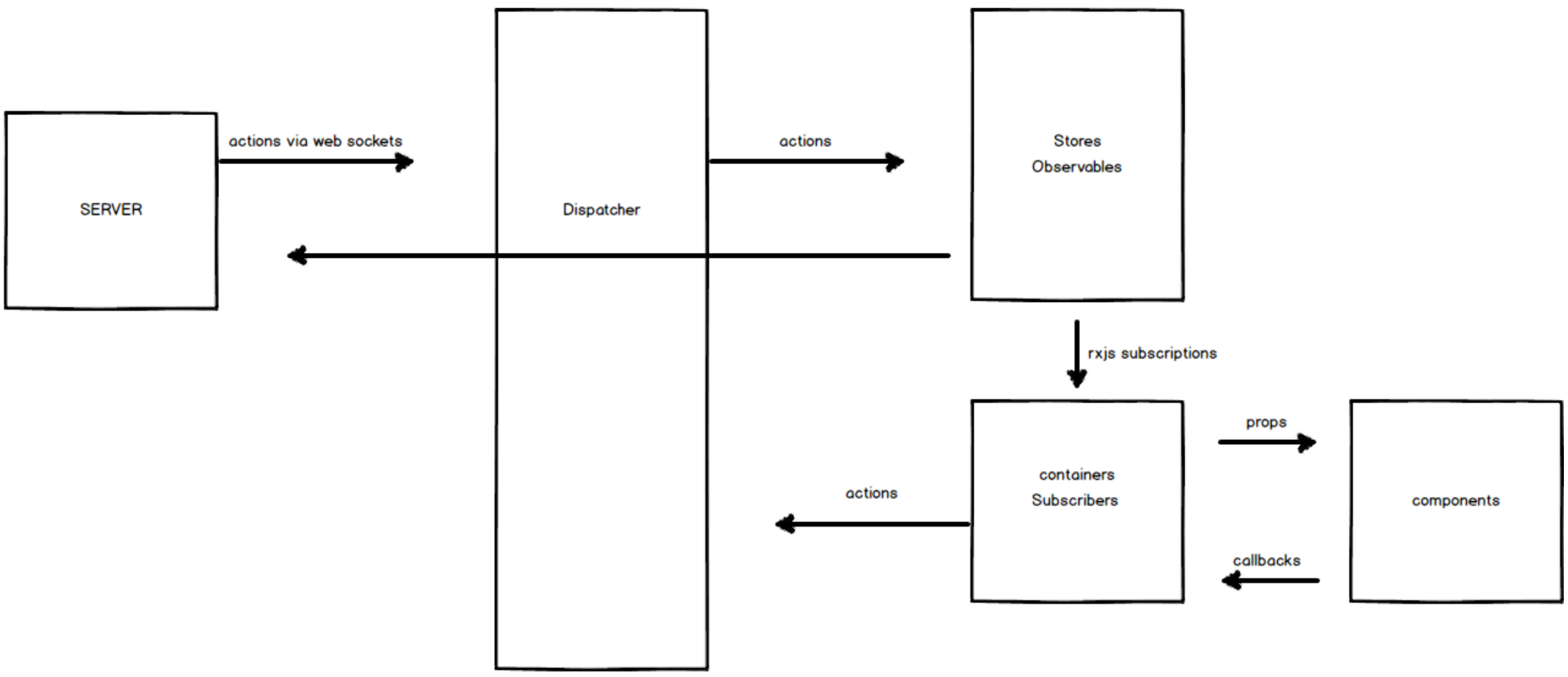
Front End Architecture

Uni-directional data flow

component => callback => action => store => action => dispatcher => store => RxJs subscription => container => c



We used a flux/redux architecture combined with the RxJs observables library.

- Our dispatcher is basically an object that you can subscribe to events from and then publish events to. It contains a list of subscriptions

and convenience methods. It is used both on the client and on the server as sort of an event bus. The purpose of the dispatcher is to dispatch actions.

- Our actions are just javascript objects with a property called "type", and potentially a property called "status". The action simply describes what you want to do, or that something was done.
- Our dispatcher guy gets actions from our containers. Once he gets those actions, he dispatches the actions to whoever is listening to them. The only things that are going to be listening for actions, are the stores.
- The stores (lobby store, user store, game store...etc) are simply classes that subscribe to our big dispatcher guy, receive actions, and then they publish rxjs observables that can be listened to from our containers.
- Our containers are simply react components that have the ability to interact directly with both the dispatcher and the stores. The containers do two things: They subscribe to the stores in order to listen to changes from the app state. They also have callbacks that create actions, send them to the dispatcher.
- Finally, the components are just standard react components.

EXAMPLE: SENDING A CHAT MESSAGE

- Our chatbox is a normal react component. It doesn't interact or know anything about stores or actions or web sockets - it's completely isolated.
- When the user hits enter on the chatbox, it will invoke a callback saying "yo! the user wants to send a message!", that'll then be sent to its container - which in this case is either going to be the game container or the lobby container.
- Then, either one of these containers is going to receive that callback, and dispatch an action to the dispatcher saying "hey dispatcher dude! the user wants to do something, go do it!". In this case, it'll be like a "send lobby chat message" action or a "send game chat message" action.
- The dispatcher is then going to receive that action, and notify anybody who's subscribed - any store that's subscribed to that action, that that action was performed. So in this case, perhaps the lobby store receives the action that says "user wants to send a message to the lobby", and so now that action if finally in the store.
- Now the stores are where all the business logic happen. They can do a variety of different things, such as validation and reject actions. In this case, the store will send that message directly to the server via web sockets. The server will then perform whatever logic you want to perform in the server. The server might perform an action such as sending the chat message to the lobby, and might also generate a new action saying "hey! there's a new message in the lobby".
- The server, through web sockets, will be hooked up directly to our dispatcher dude - and sends an action saying "a chat message was added". Then the our dispatcher will let everybody who's subscribed to that dispatcher know that the message was accepted and added. The dispatcher will then tell the lobby store - "hey, this chat message was added, go ahead and update your subscriptions!".
- The stores will then publish that data via rxjs Observables, and because the containers are subscribed to the observables - the container components will then be updated.
- Basically the container components will be subscribed to those observables, and anytime new data comes in, it will then invoke react setState on the component. And that will result in a re-render. The new data will then be passed as props to its sub components. In this case, the messages property on the chat component.