Connected Components and Performance

The changes to `<PilotsList>` bring up a key topic: performance. There's several things that are valuable to understand here.

Basic Performance Considerations

By default, whenever a React component re-renders, React will re-render all of its descendents. That means if the root component were to call `this.setState()`, the entire component tree will re-render. It's very likely that the majority of components in the tree would receive the exact same data as before and render the same output. React still has to diff the virtual DOM tree to determine if anything changed, so any render output that didn't change is effectively "wasted" effort. (React's `shouldComponentUpdate` method can be used to skip rendering for a component and its descendents, usually by doing comparisons to see if its props have really changed.)

Redux helps with this by limiting the sub-trees that actually need to re-render. `connect` generates wrapper components that manage subscriptions to the store, and each individual connected component instance is a separate subscriber. **After each dispatch, every connected component will re-run its `mapState` function, and do shallow equality checks on the result to see if the returned values have changed since the last time. If the values returned by `mapState` are different, then the wrapper component will re-render the "real" component**.

> **Note**: "shallow equality" means doing `===` reference comparisons on each individual field **within** the object returned from `mapState`. The return object itself will always be different - what matters is if `currentResult.someField === lastResult.someField`, and so on for each field in the object.

This has some important implications:

First, `mapState` **functions should run as fast as possible**. This means that a `mapState` function should minimize the amount of work it has to do, and do that work quickly. **Avoid doing very expensive work in** `mapState` **unless absolutely necessary!** This includes complex filtering and transformations. Memoized selector functions, such as the ones created by **reselect**, can ensure that expensive work is only done when something actually changed.

There's one very specific performance anti-pattern that involves use of Immutable.js. According to its author, Lee Byron, **calling** `toJS()` **is extremely expensive, and should therefore NOT be done in** `mapState`**!**. (There's several other performance concerns to take into consideration with Immutable.js - see the list of links at the end of this post for more information.)

Second, **returning the same variable references as part of the** `mapState` **result is necessary to eliminate wasted re-renders**. That means that if you're returning the same types of data at the same keys from `mapState`, but the keys point to different variable references each time, `connect` will think things have changed and re-render your component. **One of the most common examples of this is using** `Array.map()` **inside of** `mapState`. Every time you use `map()`, you create a new array reference. Again, memoized selectors can help with this by ensuring that the same values are returned.

Third, **overall performance is a balance between the overhead of more** `mapState` **calls, and time spent by React re-rendering**. Redux subscriptions are O(n) - every additional subscriber means a bit more work every time an action is dispatched. Fortunately, per the earlier quote from the FAQ, **benchmarks have shown that the cost of more connected components is generally less than the cost of more wasted re-rendering**.

Connected Performance Example

The classic example for examining Redux performance would be a list of 10,000 Todo items. The basic setup would have only the parent component connected, and directly passing a Todo object to each child. In this case, editing the text of one Todo will cause the list to re-render, and thus all 10,000

children to re-render as well.

However, if the list component only passes IDs to each child, and each child is connected and looks up its own Todo item by ID, then most of the time that Todo item will be the same and the list item component won't need to re-render. **This is one of several reasons why normalized data is so useful in Redux**, because normalized data makes it easy to look up a specific item by its type and ID.

There's an excellent slideshow called **High Performance Redux** that discusses this concept in detail, with demos of varying approaches and their performance.

Performance Concerns with *Project Mini-Mek*

Based on all that information, let's do a quick review of our implementation of a connected `<PilotsList>`.

The good news is that we *are* using the "connected list passing IDs to connected chilren" pattern. The bad news is there's several aspects that are not fully optimized yet:

- The `mapState` for `<PilotsList>` is using `Array.map()` to create a list of all Pilot IDs. **That list will be a different array reference every time**, causing `<PilotsList>` to re-render.
- Meanwhile, the `mapState` for `<PilotsListRow>` is creating a new object for the `pilot` prop each time as well, so `<PilotsListRow>` will also keep re-rendering
- Neither `<PilotsList>` nor `<PilotsListRow>` are using any memoized selector functions at all
- We're also creating a new Redux-ORM `Session` instance every time `mapState` is run, for each connected component.

Fortunately, given the size and scope of ***Project Mini-Mek***, performance isn't actually a real concern right now. Because of that, **we'll skip performance optimizations for now, and investigate those at a later time.** For now, we've at least examined some of the major performance concerns to be aware of, and know where to look when it's time to actually implement optimizations.