React Hooks

While the `useState` hook is one of the most common ones you will end up using, it is not the only one that will show up a lot. Hooks were added with React 16.8 which was released in February of 2019. One important thing to note about this is that most code written before then will probably not be using Hooks. As React progresses, however, more and more companies and code bases will be embracing hooks. We'll take a look at different hooks today and throughout the week.

Current Hooks

Currently, there are several Hooks that can be used that come pre-built but the API also gives you the ability to create your own custom hooks. Let's take a quick look at the pre-existing hooks and a quick overview of what they do. Below, you'll also see the ones we will be covering in this course. There are a few more than the ones below but they are either not used very often, or not something you're likely to run into as a junior developer.

- `useState`: Used to keep track of component state. This can also be used at a top level component and then have data passed down as `props`
- `useEffect`: This can take the place of lifecycle functions, and can also be used to clean up any functionality when the component un-mounts. You can also use this hook to create functionality that fires each time a component starts up or in conjunction with potential API calls of some sorts.
- `useContext`: We'll be covering Context providers tomorrow as another option for state management. Context basically allows you to have access to multiple different pieces of data across multiple child components through the use of a context provider. This will make more sense as the week progresses.
- `useReducer`: Use reducer lets you simplify custom state logic; if you're keeping track of multiple pieces of state that use complex logic, this will be helpful. Reducers are used with what is called flux based state management. You'll hear `redux` mentioned quite a bit in the React world as it was at one point the main go to for state management. Redux however is a third party library and, as such, can sometimes add needless complexity to an application. If you have older code, it will still be important to use however. With Hooks, the useReducer hook allows you to utilize the functionality of redux while using JUST React by itself.
- `useCallback`: Use callback allows you to pass in a function that can then create a `memoized` version of itself. We'll get into this in more detail, but one of the key takeaways for memoization is that it can prevent needless re-renders of a component.
- `useMemo`: Similar to `useCallback`, you can create a memoized value of ANY type. This will then ONLY change the value if the dependency of the hook changes, not just if ANYTHING in the component changes.
- `useRef`: Use ref allows you to store values that don't cause a re-render when it changes. This allows you to utilized the DOM and do things like focus on elements, change values, check for things or honestly anything that requires some sort of reference to a DOM element.
- `useId` - Used for generating unique IDs that are stable across the server and client, while avoiding hydration mismatches. Not used really unless implementing SSR (server-side rendering) for your application.
- `useSyncExternalStore` - Recommended for reading and subscribing from external data sources in a way that’s compatible with concurrent rendering features like selective hydration and time slicing.

Additional Notes

One thing not mentioned above is how you would actually implement the hooks themselves. We will be getting into that in more depth as the weeks continue but a couple things to note right off the bat are the following:

- Hooks should ONLY be at the top level of a component. This doesn't mean in the top most parent component, this means in the functional component directly and not in a loop, regular function, etc.
- If you're making a custom hook and it could be used in multiple components, put it in its own file. This should ideally be done regardless, but for something incredibly simple used in only one component, it might be worth it to leave in the requisite component.
- Avoid complex logic that NEEDS to be re-run at multiple points during a components use. We'll talk about this more as the weeks progress.