




React Hooks: What, Why, How?



"Hooks are functions that let you “hook into” React state and lifecycle features from function components."

<https://reactjs.org/docs/hooks-overview.html>

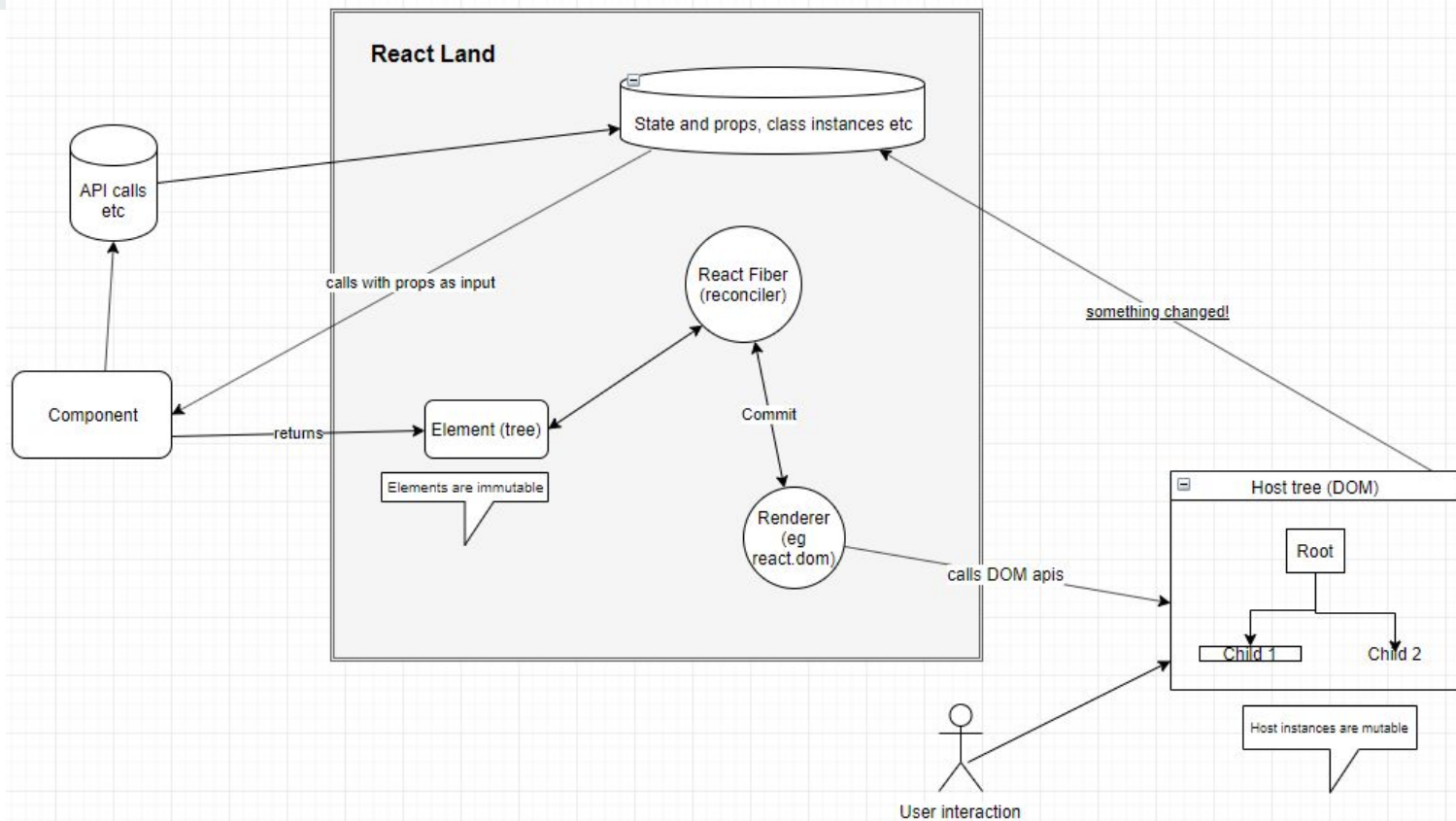
```
1 class App extends React.Component {
2   constructor(props) {
3     super(props);
4     this.state = {
5       value: localStorage.getItem("myValueInLocalStorage") || ""
6     };
7   }
8   componentDidUpdate() {
9     localStorage.setItem("myValueInLocalStorage", this.state.value);
10  }
11  onChange = event => {
12    this.setState({ value: event.target.value });
13  };
14  render() {
15    return (
16      <div>
17        <h1>Hello React ES6 Class Component!</h1>
18        <input value={this.state.value} type="text" onChange={this.onChange} />
19        <p>{this.state.value}</p>
20      </div>
21    );
22  }
23 }
```

```
1 const App = () => {
2   const [value, setValue] = React.useState(
3     localStorage.getItem("myValueInLocalStorage") || ""
4   );
5   React.useEffect(() => {
6     localStorage.setItem("myValueInLocalStorage", value);
7   }, [value]);
8   const onChange = event => setValue(event.target.value);
9   return (
10     <div>
11       <h1>Hello React Function Component!</h1>
12       <input value={value} type="text" onChange={onChange} />
13       <p>{value}</p>
14     </div>
15   );
16 };
17
18 // source: https://www.robinwieruch.de/react-hooks-migration
```

Why?

- Less verbose
- Less repetition
- Keep code together that belongs together
- More composable (no more wrapper hell with Higher Order Components)
- 'this' is hard!







sebmabage commented on Nov 16, 2018

Author

Collaborator



One argument that I keep hearing is that the motivation isn't strong enough because "classes are fine". Supposedly, it's the users trying to learn them that are broken. I think this argument is overfocusing on some clip that maybe someone has highlighted around how classes are hard to learn for newcomers. That's not the point.

The main motivation is that patterns like closures naturally creates copies of values which makes writing concurrent code a lot easier because you can store n number of states at any given point instead of just one in the case of a mutable class. This avoids a number of foot guns where classes seem intuitive but actually yield unpredictable results.


Classes may seem like the ideal thing to hold state since that's what they're designed for. However, React is more written like a declarative function that keeps getting executed over and over to simulate it being reactive. Those two things have an impedance mismatch and that keeps leaking when we think of these as classes.

<https://github.com/reactjs/rfcs/pull/68#issuecomment-439314884>



Demo: Why function and class components are not equivalent

Shamelessly adopted from: <https://overreacted.io/how-are-function-components-different-from-classes/>



“...the conceptual mental model for React is just functions calling other functions recursively. There is a lot of value to express it in those terms to help build the correct mental model.”

<https://github.com/reactjs/rfcs/pull/68#issuecomment-439314884>



“I like to think of React elements as being like frames in a movie. They capture what the UI should look like at a specific point in time. They don’t change.”

<https://overreacted.io/react-as-a-ui-runtime/>

What? Out of the box hooks in single tweet...





How? The rules of hooks

1. Only call hooks from React functions
2. Only call hooks from the top level: not from inside loops, conditions, nested functions

Also...declare all your dependencies, yes, even functions.



Why of the How: Only call hooks at the top level

ReactFiberHooks.js:

```
178 let currentlyRenderingFiber: Fiber | null = null;
179
180 // Hooks are stored as a linked list on the fiber's memoizedState field. The
181 // current hook list is the list that belongs to the current fiber. The
182 // work-in-progress hook list is a new list that will be added to the
183 // work-in-progress fiber.
184 let currentHook: Hook | null = null;
```

This supports things like multiple calls to a hooks per component, removes chance of name clashes and issues where two components both call same base component

<https://overreacted.io/why-do-hooks-rely-on-call-order/>



Why of the How: Declare your dependencies

Effects synchronise the UI to changes in props/state etc (i.e. the data flow)

“The recommendation is to hoist functions that don't need props or state outside of your component, and pull the ones that are used only by an effect inside of that effect. If after that your effect still ends up using functions in the render scope (including function from props), wrap them into useCallback where they're defined, and repeat the process. Why does it matter? Functions can “see” values from props and state — so they participate in the data flow.”

<https://overreacted.io/a-complete-guide-to-useeffect/>



An aside about useCallback()

It does fix a lot of issues BUT it has an overhead:

- Create the function
- Create the dependency array
- Call `react.useCallback()` and all that entails



What? Write your own hooks...

**Demo: A custom hook to deal with window
resize**



What? ...or use other people's

```
const Dogs = ({ onDogSelected }) => (  
  <Query query={GET_DOGS}>  
    ({ { loading, error, data } }) => {  
      if (loading) return "Loading...";  
      if (error) return `Error! ${error.message}`;  
  
      return (  
        <select name="dog" onChange={onDogSelected}>  
          {data.dogs.map(dog => (  
            <option key={dog.id} value={dog.breed}>  
              {dog.breed}  
            </option>  
          ))}  
        </select>  
      );  
    }  
  </Query>  
);
```

```
function Dogs({ onDogSelected }) {  
  const { loading, error, data } = useQuery(GET_DOGS);  
  
  if (loading) return 'Loading...';  
  if (error) return `Error! ${error.message}`;  
  
  return (  
    <select name="dog" onChange={onDogSelected}>  
      {data.dogs.map(dog => (  
        <option key={dog.id} value={dog.breed}>  
          {dog.breed}  
        </option>  
      ))}  
    </select>  
  );  
}
```



```
1 import React, { useState } from 'react';
2 import { useLazyQuery } from '@apollo/react-hooks';
3
4 function DelayedQuery() {
5   const [dog, setDog] = useState(null);
6   const [getDog, { loading, data }] = useLazyQuery(GET_DOG_PHOTO);
7
8   if (loading) return <p>Loading ...</p>;
9
10  if (data && data.dog) {
11    setDog(data.dog);
12  }
13
14  return (
15    <div>
16      {dog && <img src={dog.displayImage} />}
17      <button onClick={() => getDog({ variables: { breed: 'bulldog' } })}>
18        Click me!
19      </button>
20    </div>
21  );
22 }
```

```
export function useQuery<TData = any, TVariables = OperationVariables>(
  query: DocumentNode,
  options?: QueryHookOptions<TData, TVariables>
) {
  return useBaseQuery<TData, TVariables>(query, options, false) as QueryResult<
    TData,
    TVariables
  >;
}
```

```
export function useLazyQuery<TData = any, TVariables = OperationVariables>(
  query: DocumentNode,
  options?: LazyQueryHookOptions<TData, TVariables>
) {
  return useBaseQuery<TData, TVariables>(query, options, true) as QueryTuple<
    TData,
    TVariables
  >;
}
```

```
export function useBaseQuery<TData = any, TVariables = OperationVariables>(
  query: DocumentNode,
  options?: QueryHookOptions<TData, TVariables>,
  lazy = false
) {
  const context = useContext(getApolloContext());
  const [tick, forceUpdate] = useReducer(x => x + 1, 0);
  const updatedOptions = options ? { ...options, query } : { query };

  const queryDataRef = useRef<QueryData<TData, TVariables>>();

  if (!queryDataRef.current) {
    queryDataRef.current = new QueryData<TData, TVariables>({
      options: updatedOptions as QueryOptions<TData, TVariables>,
      context,
      forceUpdate
    });
  }

  const queryData = queryDataRef.current;
  queryData.setOptions(updatedOptions);
  queryData.context = context;

  // `onError` and `onCompleted` callback functions will not always have a
  // stable identity, so we'll exclude them from the memoization key to
  // prevent `afterExecute` from being triggered un-necessarily.
  const memo = {
    options: { ...updatedOptions, onError: undefined, onCompleted: undefined },
    context,
    tick
  };

  const result = useDeepMemo(
    () => (lazy ? queryData.executeLazy() : queryData.execute()),
    memo
  );

  const queryResult = lazy
    ? [result as QueryTuple<TData, TVariables>][1]
    : (result as QueryResult<TData, TVariables>);

  useEffect(() => queryData.afterExecute({ lazy }), [
    queryResult.loading,
    queryResult.networkStatus,
    queryResult.error,
    queryResult.data
  ]);

  useEffect(() => {
    return () => queryData.cleanup();
  }, []);

  return result;
}
```