React Hooks

Make stateless function components stateful

Jan Baer

25. Februar 2019

React Hooks - What is it

- React Hooks make it possible to take a functional component and add a state to it, or hook into lifecycle methods like **componentDidMount** and **componentDidUpdate**.
- React Hooks can be used since v16.8.
- With React Hooks you can use functional components for more use cases and prevent to write bloated classes.

React Hooks - Which hooks comes out of the box

- Basic Hooks
  - useState
  - useEffect
  - useContext
- Additional Hooks
  - useReducer
  - useRef
  - useCallback
  - useMemo
  - ...

useState

- Returns a stateful value, and a function to update it. During the initial render, the returned state (state) is the same as the value passed as the first argument (initialState). The setState function is used to update the state. It accepts a new state value and enqueues a re-render of the component.
- Is perfect for managing a state on a single component. But it isn't for shared states.
- You can use a state object but also only a single value as a state
- You can use **useState** multiple times in the same component
- **useState** takes an inital state value or a function as parameter to lazy initialize the value

useState

```
const [{state}, set{State}] = useState(initialState);
...
setCounter(counter + 1);
```

```
const [state, setState] = useState(() => {
  const initialState = someExpensiveComputation(props);
  return initialState;
});
```

useEffect

- The Effect Hook lets you perform side effects in function components. If you're familiar with React class lifecycle methods, you can think of useEffect Hook as componentDidMount, componentDidUpdate, and componentWillUnmount combined.

useEffect

- The Effect Hook lets you perform side effects in function components. If you're familiar with React class lifecycle methods, you can think of useEffect Hook as componentDidMount, componentDidUpdate, and componentWillUnmount combined.
- useEffect let you control how often it should be called

useEffect

- The Effect Hook lets you perform side effects in function components. If you're familiar with React class lifecycle methods, you can think of useEffect Hook as componentDidMount, componentDidUpdate, and componentWillUnmount combined.
- useEffect let you control how often it should be called
- When you return a cleanup function then it will be called when the component will be unmounted

useEffect

```
useEffect(() => {
  document.title = 'The will happen after the first' +
    'render cycle was finished';

  return function cleanup() {
    // Here you can unsubscribe any eventhandlers
  };
}, []);
```

- Example

useContext

- Accepts a context object (the value returned from React.createContext) and returns the current context value, as given by the nearest context provider for the given context.
- When the provider updates, this Hook will trigger a rerender with the latest context value.
- Example
- Example2

Rules of Hooks

- Only call hooks at the top level of your function. Don't put them in loops, conditionals, or nested functions. In order for React to keep track of your hooks, the same ones need to be called in the same order every single time.
- Only call hooks from React function components, or from custom hooks. Don't call them from outside a component (what would that even do?). Keeping all the calls inside components and custom hooks makes your code easier to follow too, because all the related logic is grouped together.
- The names of hooks must start with "use". Like useState or useEffect (well, not those two, those are taken).

Write your own hooks

- Custom hooks are using mostly the original React hooks like **useState** and **useEffect** together to prevent duplicated code.

External hooks

- mobx-react-lite