React Hooks (ReactJS version >= 16.8)

**So why should I use functional components at all?**

You might ask yourself why you should use functional components at all, if they remove so many nice features. But there are some benefits you get by using functional components in React:

1. Functional component are much **easier to read and test**because they are plain JavaScript functions without state or lifecycle-hooks
2. You end up with **less code**
3. They help you to use **best practices**. It will get easier to separate container and presentational components because you need to think more about your component’s state if you don’t have access to setState() in your component
4. The React team [mentioned](https://reactjs.org/blog/2015/10/07/react-v0.14.html#stateless-functional-components) that there may be a **performance** boost for functional component in future React versions

**What is a Hook?** A Hook is a special function that lets you “hook into” React features. For example, useState is a Hook that lets you add React state to function components. We’ll learn other Hooks later.

**When would I use a Hook?** If you write a function component and realize you need to add some state to it, previously you had to convert it to a class. Now you can use a Hook inside the existing function component. We’re going to do that right now!

**useState**

In a function component, we have no this, so we can’t assign or read this.state. Instead, we call the useState Hook directly inside our component.

The only argument to the useState() Hook is the initial state. Unlike with classes, the state doesn’t have to be an object. We can keep a number or a string if that’s all we need. (If we wanted to store two different values in state, we would call useState() twice.)

It returns a pair of values: the current state and a function that updates it.

**Synthetic event**

If you want to access the event properties in an asynchronous way, you should call event.persist() on the event, which will remove the synthetic event from the pool and allow references to the event to be retained by user code.

**useEffect**

The Effect Hook lets you perform side effects in function components:

Data fetching, setting up a subscription, and manually changing the DOM in React components are all examples of side effects. Whether or not you’re used to calling these operations “side effects” (or just “effects”), you’ve likely performed them in your components before.

If you’re familiar with React class lifecycle methods, you can think of useEffect Hook as componentDidMount, componentDidUpdate, and componentWillUnmount combined.

**For after and every render cycle useEffect is invoked**

**Arguments in useEffect:**

1. A function which executes after each render cycle.
2. An array with a dependency of the function (including external dependencies used) only when the dependency is changed the function will re-run.

when empty array is passed as second argument useEffect acts like componentDidMount it runs only once for after the render cycle.

Clean up return function

**useCallback:**

Returns a [memoized](https://en.wikipedia.org/wiki/Memoization" \t "_blank) callback.

Pass an inline callback and an array of dependencies. useCallback will return a memoized version of the callback that only changes if one of the dependencies has changed. This is useful when passing callbacks to optimized child components that rely on reference equality to prevent unnecessary renders (e.g. shouldComponentUpdate).

**useref:**

Essentially, **useRef** is like a “box” that can hold a mutable value in its .current property. You might be familiar with refs primarily as a way to access the DOM. If you pass a ref object to **React** with <div ref={myRef} /> , **React** will set its .current property to the corresponding DOM node whenever that node changes