React Hooks

By Mykhailo Kaduk

August 2020

AGENDA

React Hooks:

Motivation

Basic Approach

Built-in Hooks

Custom Hooks

Hooks vs HOCs

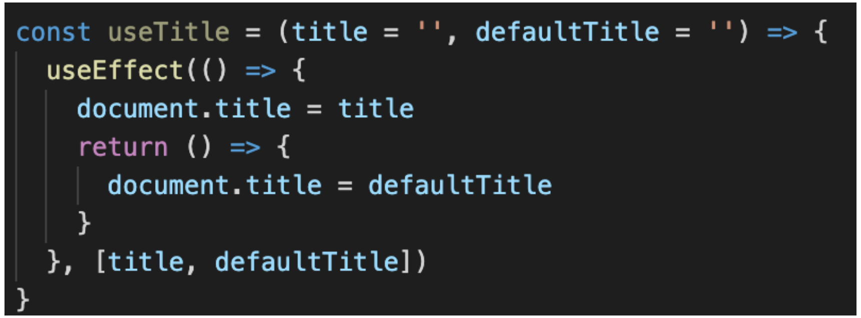
MOTIVATION

Motivation

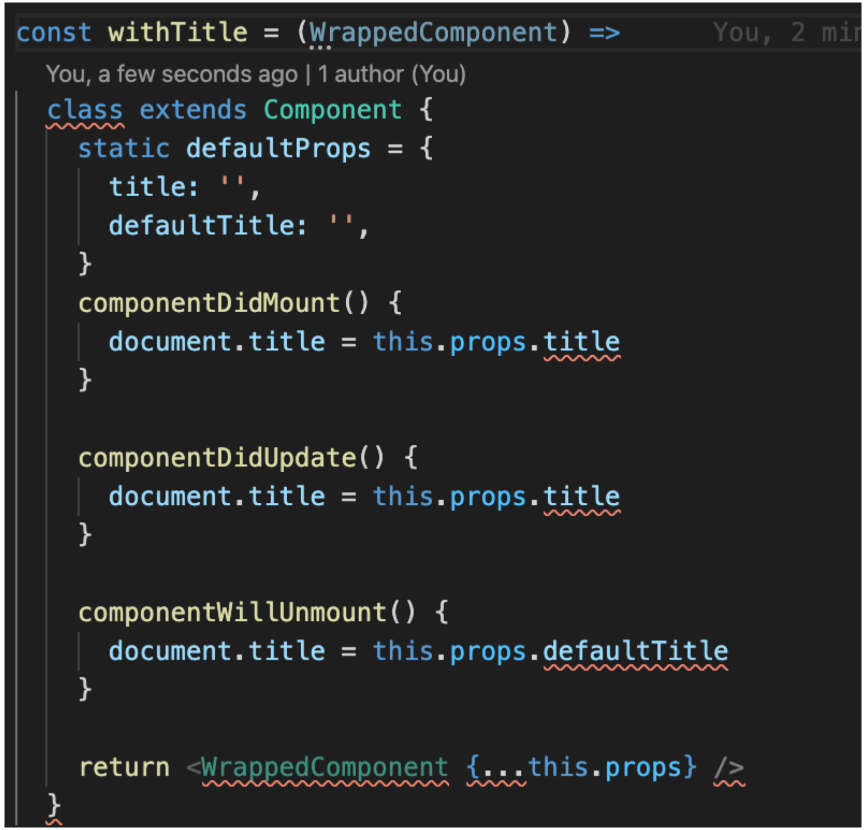
Main reasons of introducing hooks:

1. Reusing logic
2. Giant components
3. Confusing classes (for both, humans and machines)

Motivation



```
const useTitle = (title = '', defaultTitle = '') => {
  useEffect(() => {
    document.title = title
    return () => {
      document.title = defaultTitle
    }
  }, [title, defaultTitle])
}
```



```
const withTitle = (WrappedComponent) =>        You, 2 mi
  You, a few seconds ago | 1 author (You)
  class extends Component {
    static defaultProps = {
      title: '',
      defaultTitle: '',
    }
    componentDidMount() {
      document.title = this.props.title
    }

    componentDidUpdate() {
      document.title = this.props.title
    }

    componentWillUnmount() {
      document.title = this.props.defaultTitle
    }

    return <WrappedComponent {...this.props} />
  }
```

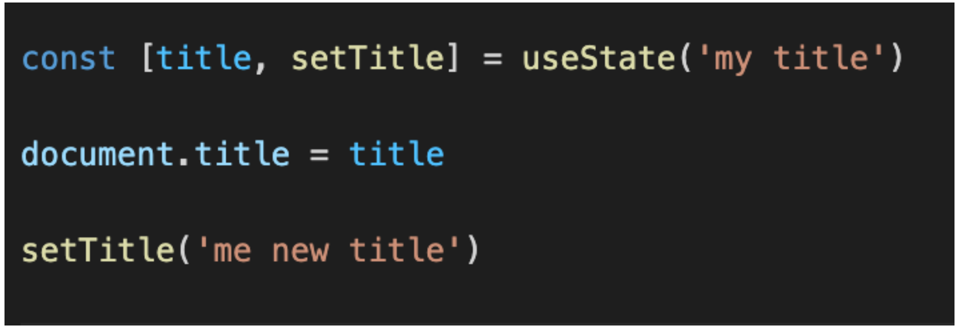
BASIC APPROACH

Basic Approach

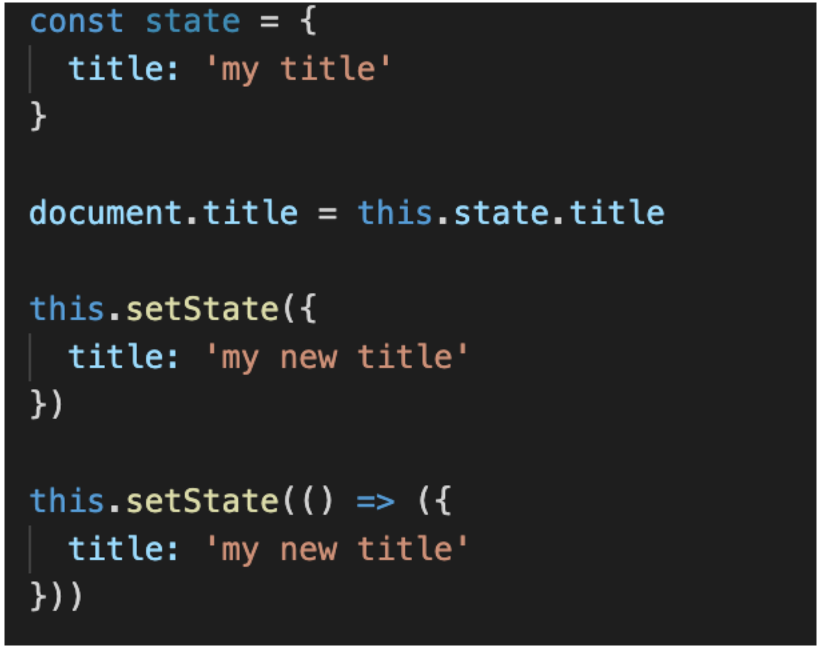
When 3 main problems in current React became obvious to Facebook team, they decided to take next steps:

- Move away from classes
- Write declarative code
- Do not introduce breaking changes

Basic Approach



```
const [title, setTitle] = useState('my title')

document.title = title

setTitle('me new title')
```



```
const state = {
  title: 'my title'
}

document.title = this.state.title

this.setState({
  title: 'my new title'
})

this.setState(() => ({
  title: 'my new title'
}))
```

Basic Approach

Alongside with mental concept, React team introduced three main rules of writing and using hooks.

1. **Only call hooks from a top level,** which means never calling a hook inside conditional statements and loops.
2. **Only call hooks inside React functions,** which means that you cannot use this api anywhere outside components.
3. **ALWAYS call your custom hooks starting from "use" keyword.** It is not so much a rule as a naming convention. It really helps visually separate hook-based logic from everything else and helps to avoid name collisions. Be a good developer, follow this as a rule, don't make it optional.

BUILT-IN HOOKS

Built-in Hooks

Basic Hooks

- useState
- useEffect
- useContext

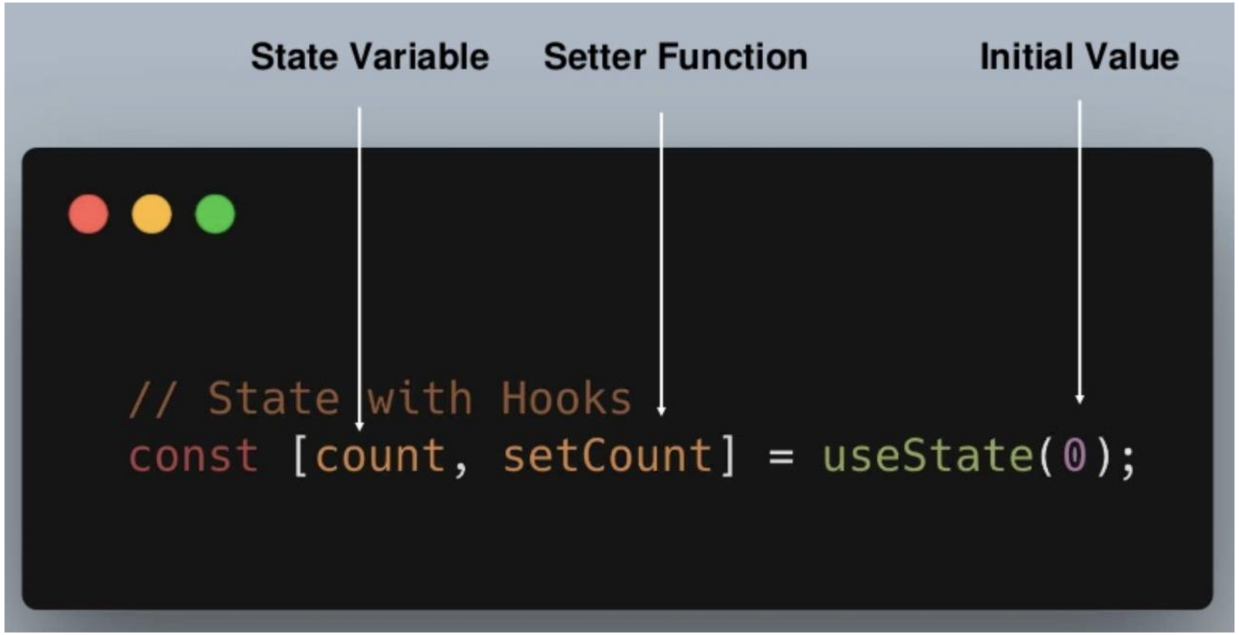
Additional Hooks

- useReducer
- useCallback
- useMemo
- useRef
- useImperativeHandle
- useLayoutEffect
- useDebugValue

Provided by the React Team

useState

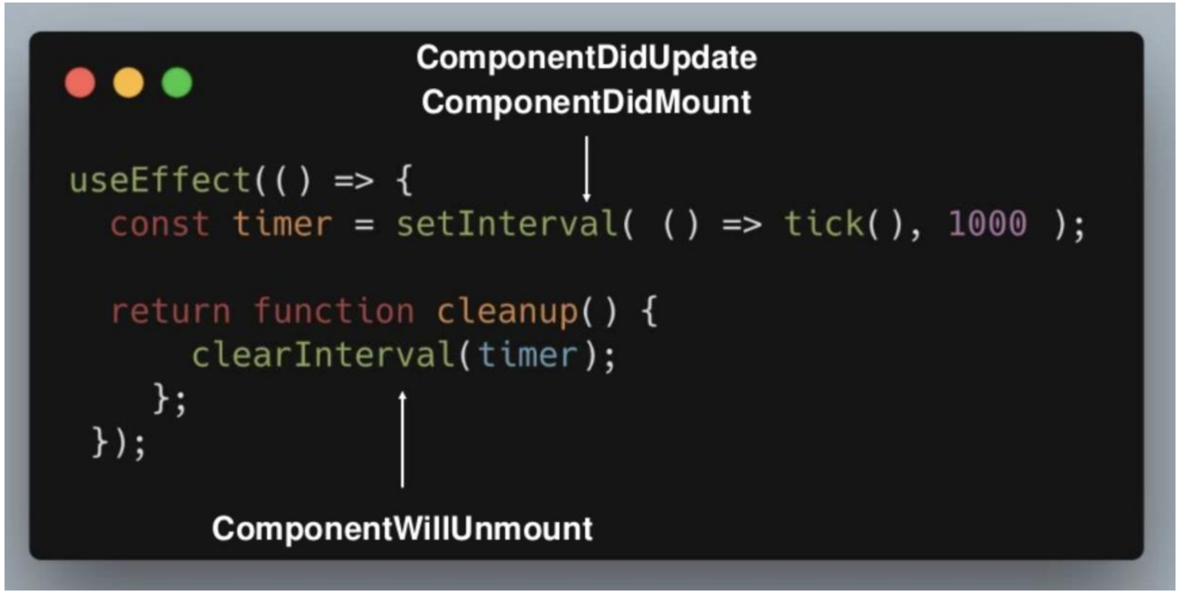
Represents internal state of a component. Takes default value as an argument and provides getter and setter for state usage and manipulations. Setter is async.



```
// State with Hooks
const [count, setCount] = useState(0);
```

useEffect

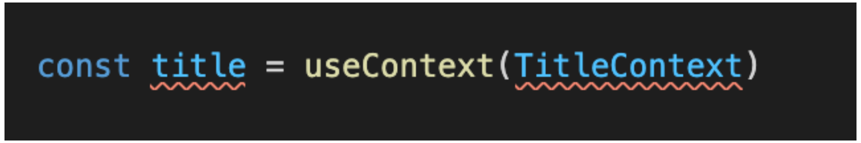
Represents life-cycle-dependent async side-effects, that could be performed from the component. Could react as “componentDidMount”, “componentDidUpdate” and “componentWillUnmount”. Takes callback with side-effects as a first argument. Also accepts optional second argument which is memorizing array, determining variables, that should change in order for hook to fire.



```
useEffect(() => {
  const timer = setInterval( () => tick(), 1000 );

  return function cleanup() {
      clearInterval(timer);
    };
 });
```

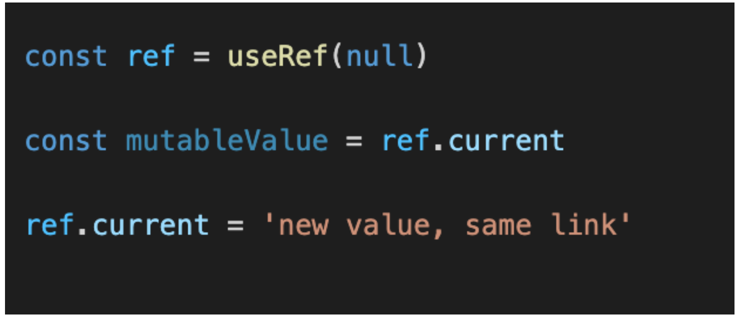
useContext, useRef

useContext - Represents a possibility to work with React Context Api. Takes a context as is only argument. Provides only with a getter.
Changing the value of a context trigger component rerender process, which could not be avoided, even using React.memo.



```
const title = useContext(TitleContext)
```

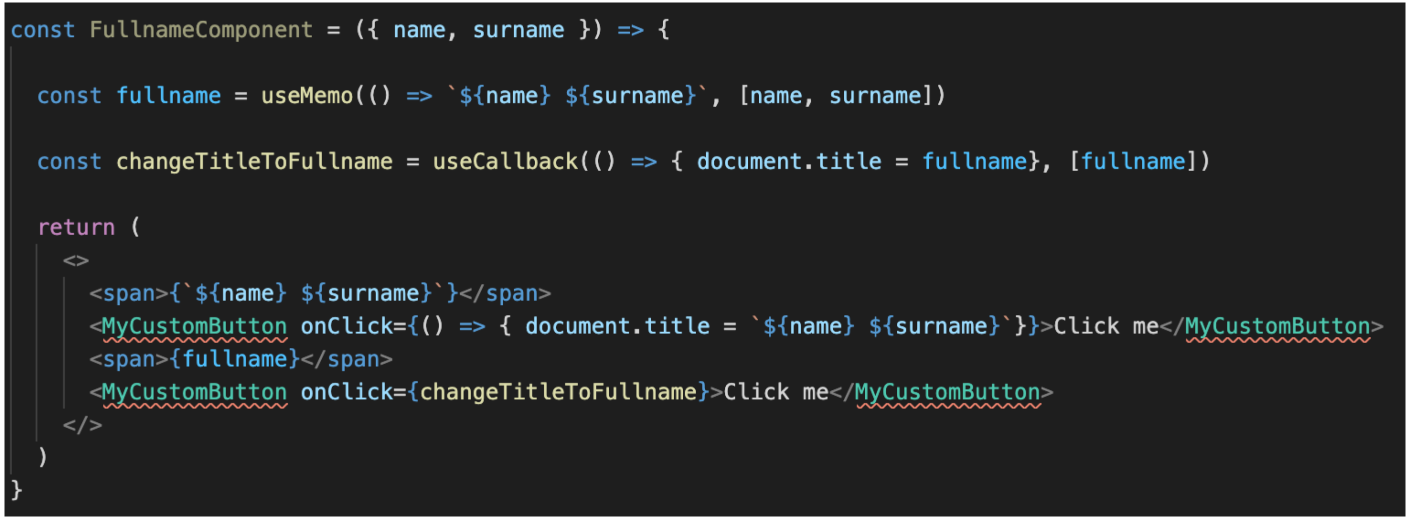
useRef - Represents a possibility to work with mutable values in React.
Changing the value of a ref does not trigger rerender process. Could be used as a link to a React-element.
It is highly recommended to provide default value for a ref.



```
const ref = useRef(null)

const mutableValue = ref.current

ref.current = 'new value, same link'
```

useCallback, useMemo

Both represent a possibility to create memorized of some internally used values (useMemo) and callbacks (useCallback).
useCallback could be implemented via useMemo, so it's basically a shortcut. Mostly used for passing callbacks as props to underlying components in order to prevent them of unnecessary rerendering all the time.
useMemo is typically used to avoid unnecessary calculations on render phase.

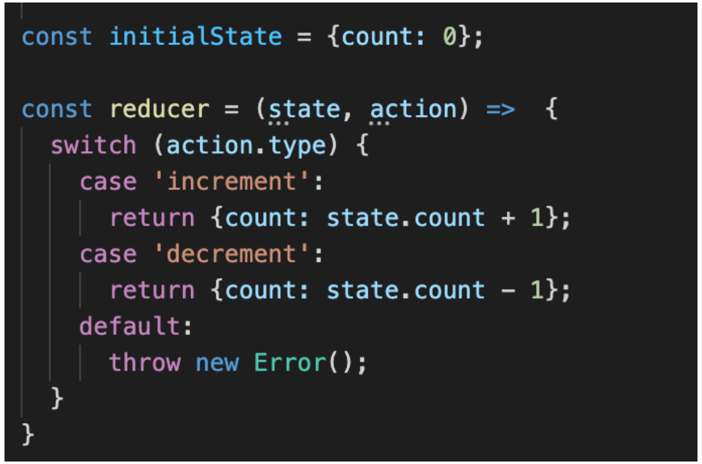


```
const FullnameComponent = ({ name, surname }) => {

  const fullname = useMemo(() => `${name} ${surname}`, [name, surname])

  const changeTitleToFullname = useCallback(() => { document.title = fullname}, [fullname])

  return (
    <>
      <span>{`${name} ${surname}`}</span>
      <MyCustomButton onClick={() => { document.title = `${name} ${surname}`}}>Click me</MyCustomButton>
      <span>{fullname}</span>
      <MyCustomButton onClick={changeTitleToFullname}>Click me</MyCustomButton>
    </>
  )
}
```

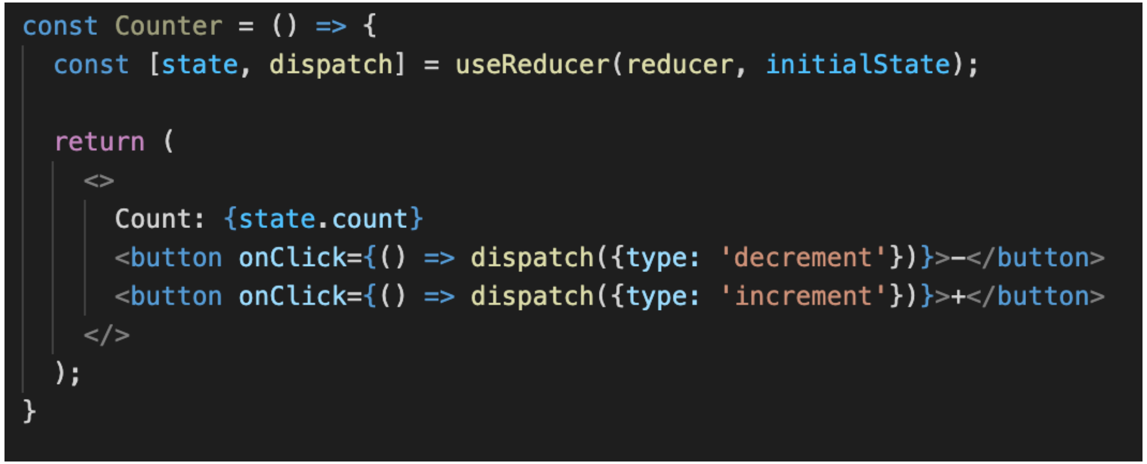
useReducer

Represents possibility to deal with complex state-management inside the component.
Typically should be avoided in favor of **useState.**
One of core benefits is ability to run multiple setters during one rendering cycle.

MUST NOT be confused with Redux and other approaches, created for a state-management. Should be used only for updates on a component level.



```
const initialState = {count: 0};

const reducer = (state, action) =>  {
  switch (action.type) {
    case 'increment':
      return {count: state.count + 1};
    case 'decrement':
      return {count: state.count - 1};
    default:
      throw new Error();
  }
}
```



```
const Counter = () => {
  const [state, dispatch] = useReducer(reducer, initialState);

  return (
    <>
      Count: {state.count}
      <button onClick={() => dispatch({type: 'decrement'})}>-</button>
      <button onClick={() => dispatch({type: 'increment'})}>+</button>
    </>
  );
}
```

CUSTOM HOOKS

Custom Hooks

Most powerful concept of hooks-based development is, of course, potential for code separation and reuse.
That could be done through manually written hooks, which are plain old functions in its core.
The idea is that until developer calls function exclusively inside React components, he/she could put anything, including built-in hooks, inside.

Main benefits of this approach are:

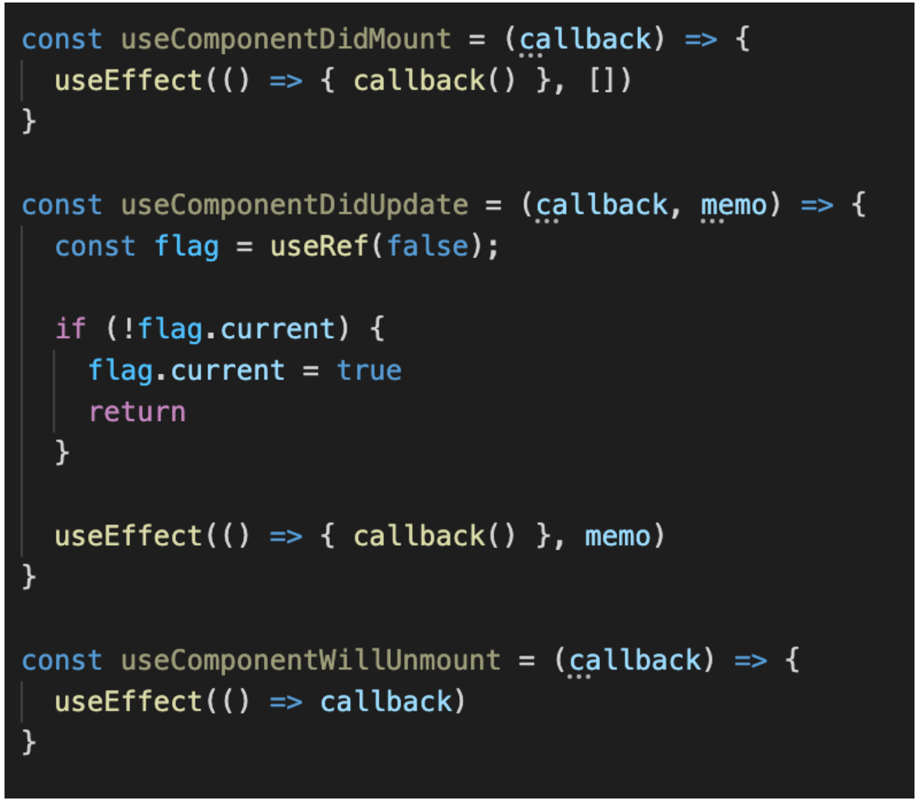
- Write code anywhere aside your component, keeping it neat and clean.
- Hide behind abstraction.
- Share code.
- Unify all (almost) APIs under one codestyle.
- Super fun coding.

There are tons of libraries with production-ready hooks on NPM, which you could use right away.
Since hooks were released, vast majority of main libs, providing API for React development, such as **react-redux, react-router-dom, etc.** released major versions of packages, introducing corresponding methods

But always remember, with great power comes great responsibility.

EXAMPLES

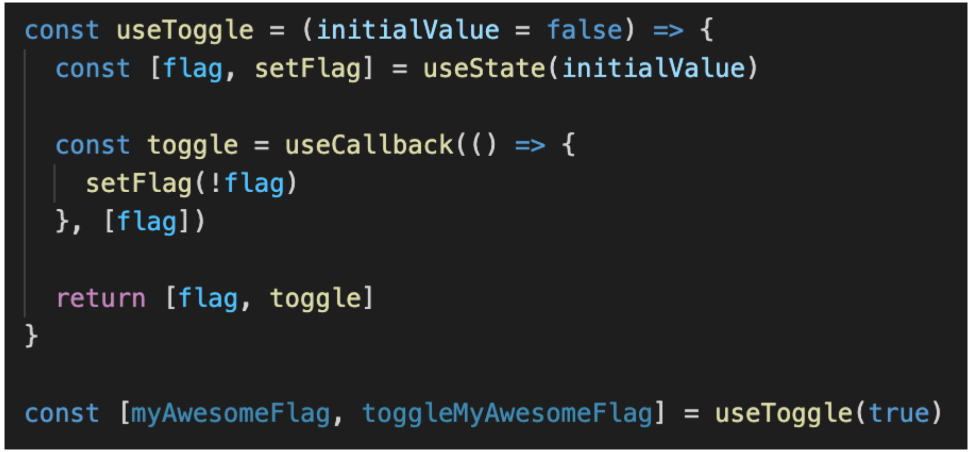
Sometimes you need to use effects only on exact lifecycle step, like mounting, updating or unmounting.



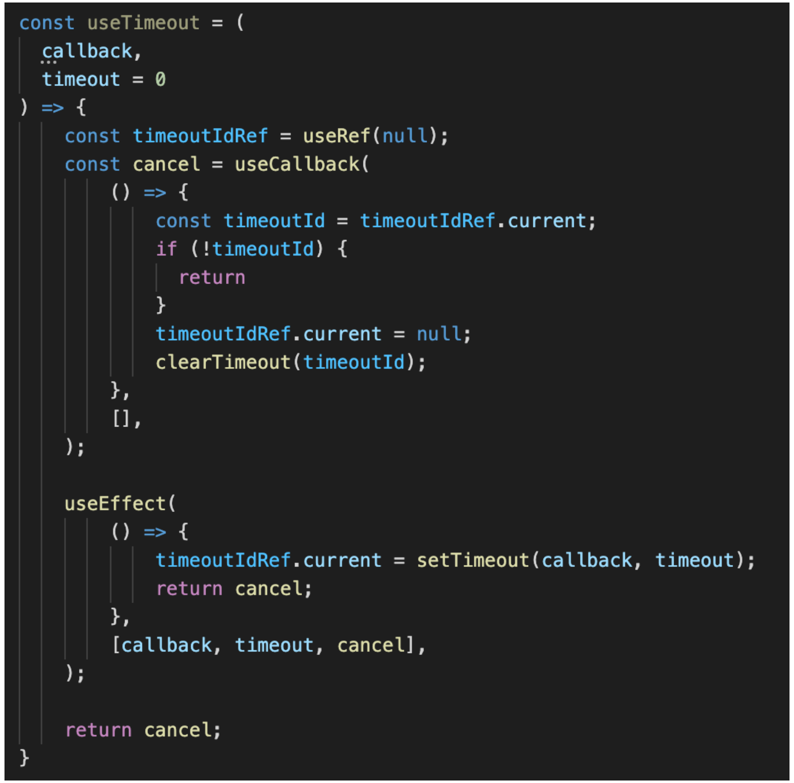
```
const useComponentDidMount = (callback) => {
  useEffect(() => { callback() }, [])
}

const useComponentDidUpdate = (callback, memo) => {
  const flag = useRef(false);

  if (!flag.current) {
    flag.current = true
    return
  }

  useEffect(() => { callback() }, memo)
}

const useComponentWillUnmount = (callback) => {
  useEffect(() => callback)
}
```

EXAMPLES

Here are couple of useful utility hooks for future convenience.



```
const useToggle = (initialValue = false) => {
  const [flag, setFlag] = useState(initialValue)

  const toggle = useCallback(() => {
    setFlag(!flag)
  }, [flag])

  return [flag, toggle]
}

const [myAwesomeFlag, toggleMyAwesomeFlag] = useToggle(true)
```



```
const useTimeout = (
  callback,
  timeout = 0
) => {
    const timeoutIdRef = useRef(null);
    const cancel = useCallback(
        () => {
            const timeoutId = timeoutIdRef.current;
            if (!timeoutId) {
              return
            }
            timeoutIdRef.current = null;
            clearTimeout(timeoutId);
        },
        [],
    );

    useEffect(
        () => {
            timeoutIdRef.current = setTimeout(callback, timeout);
            return cancel;
        },
        [callback, timeout, cancel],
    );

    return cancel;
}
```

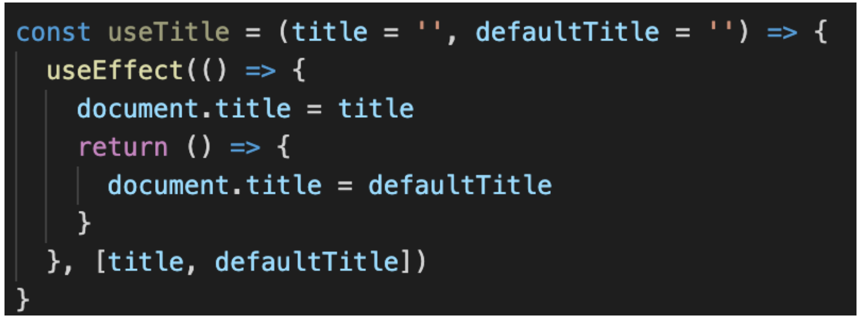
HOCS VS HOOKS

HOCs vs Hooks

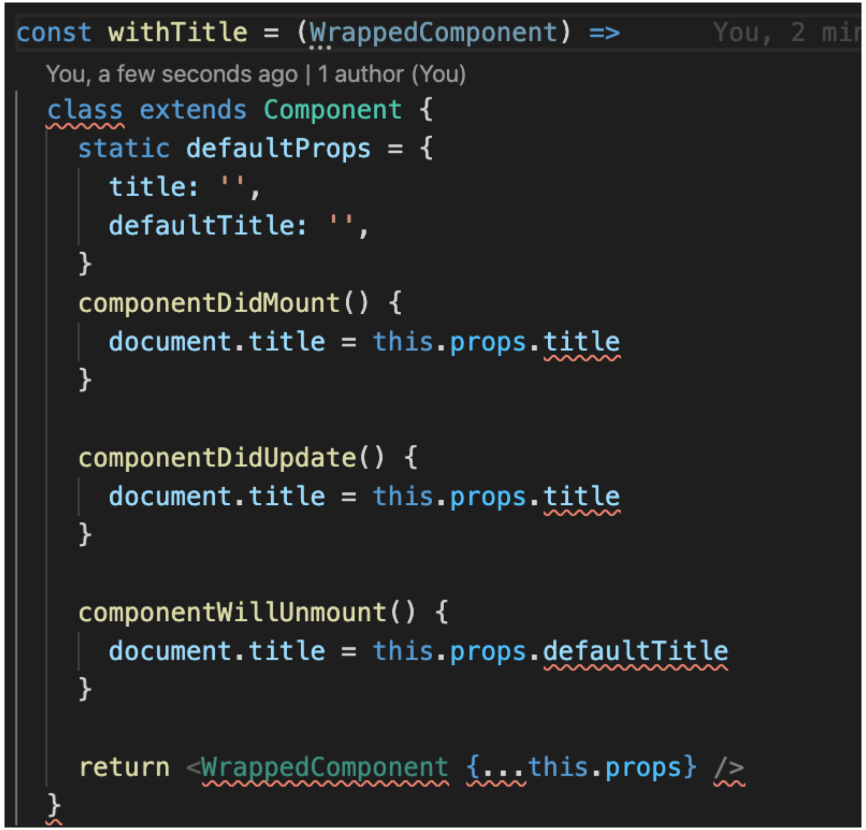
For lots of React developers (especially for beginners) it is a hard question, what to use in a particular case, Higher Order Component or a Hook.
Today both solutions could provide same (almost) developer experience, and there isn’t a defining guide on the topic.
Nevertheless, there are some defining differences, which might help:

- HOCs create wrappers (elements) in Virtual-DOM. Hooks don’t.
- HOCs work from outside the component which gives the opportunity to control components rendering process, Hooks work from inside, which gives way less control over rendering process.
- Working with class-based components allows only working with HOCs, since hooks API is only for functional components.
- Legacy projects came out way before hooks, so they probably heavy-loaded with HOCs.
- HOCs are becoming more and more obsolete in favor of hooks, and thus, libraries stop being maintained and supported.
- HOCs are less obvious, because they could do literally anything with a component (add props, prevent from rendering etc.) so, in some way, they are “modern mixins”.
- Sometimes, wrong order of HOCs is error-prone, which is very hard to find, instead of hooks, which all are triggered inside a component.
- Hooks kill separation between dumb and stateful components, since everything could be stateful now.

HOCs vs Hooks



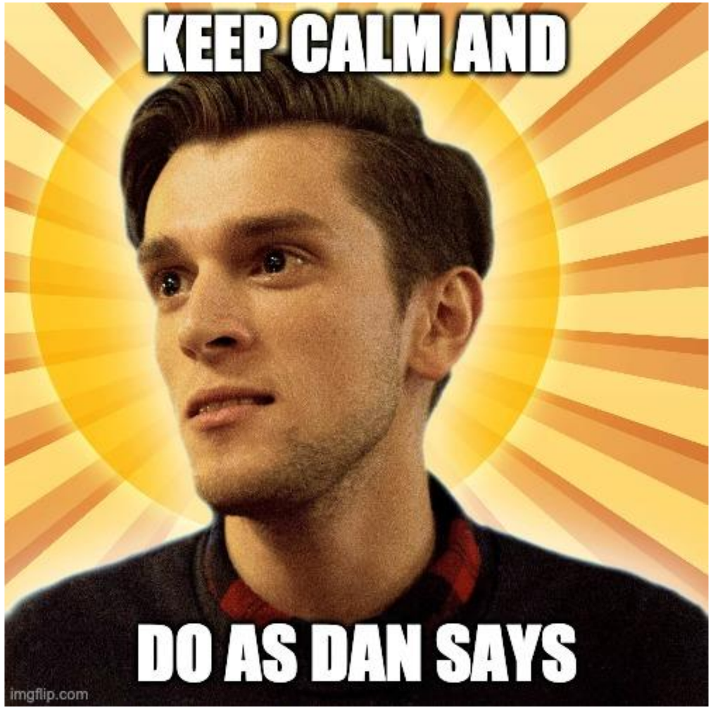
```
const useTitle = (title = '', defaultTitle = '') => {
  useEffect(() => {
    document.title = title
    return () => {
      document.title = defaultTitle
    }
  }, [title, defaultTitle])
}
```



```
const withTitle = (WrappedComponent) =>          You, 2 mi
  You, a few seconds ago | 1 author (You)
  class extends Component {
    static defaultProps = {
      title: '',
      defaultTitle: '',
    }
    componentDidMount() {
      document.title = this.props.title
    }

    componentDidUpdate() {
      document.title = this.props.title
    }

    componentWillUnmount() {
      document.title = this.props.defaultTitle
    }

    return <WrappedComponent {...this.props} />
  }
```

USEFUL LINKS HERE

Last but not least



QUESTIONS?