

# The new React

Maurice de Beijer - @mauricedb



- Maurice de Beijer
- The Problem Solver
- Microsoft MVP
- Freelance developer/instructor
- Twitter: @mauricedb
- Web: <http://www.TheProblemSolver.nl>
- E-mail: [maurice.de.beijer@gmail.com](mailto:maurice.de.beijer@gmail.com)



Gift This Course



Wishlist

# Master RxJS 6 Without Breaking A Sweat

Learn how to solve common programming problems using RxJS

HIGHEST RATED



4.7 (11 ratings)

730 students enrolled

Created by Maurice de Beijer   Last updated 12/2018   English   English [Auto-generated]



Preview this course

## What you'll learn

- ✓ After this course you will be able to see where using RxJS makes sense.
- ✓ You will be able to solve common programming problems using RxJS.

## Requirements

- Basic understanding of JavaScript is required.
- A PC with Node, NPM, a modern browser like Chrome or FireFox and a code editor you like is required.
- Any previous knowledge of RxJS is not required.

€10.99 ~~€99.99~~ 89% off

5 hours left at this price!

Add to cart

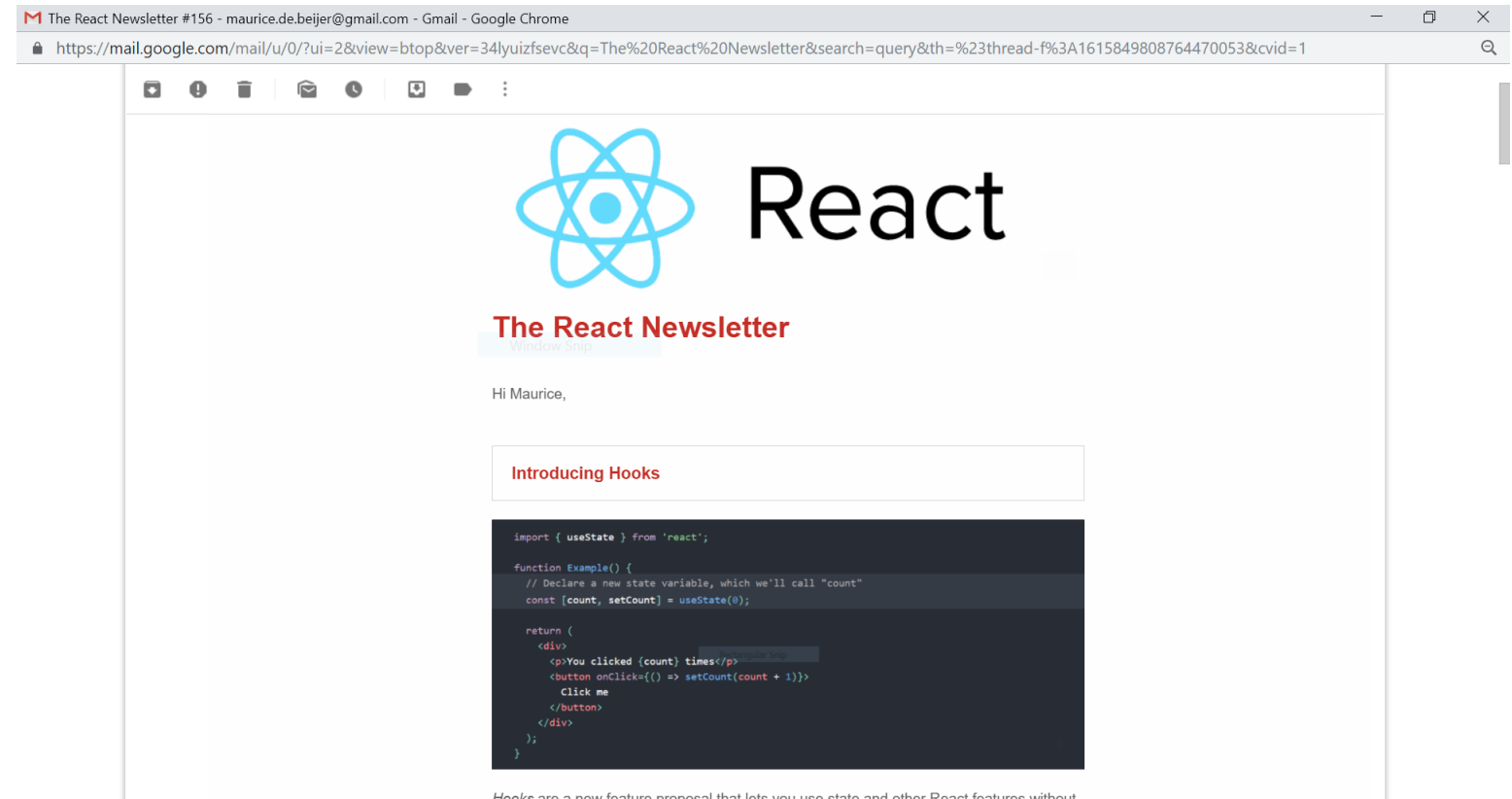
Buy now

30-Day Money-Back Guarantee

This course includes

- 2.5 hours on-demand video
- Full lifetime access

# The React Newsletter



<http://bit.ly/ReactNewsletter>

# Topics

- Intro into React hooks
- Basic hooks
  - useState()
  - useEffect()
  - useEffect() with asynchronous actions
  - useContext()
  - Custom hooks
- Advanced hooks
  - useDebugValue()
  - useMemo()
  - useReducer()
  - useRef()
- ~~Suspense~~
  - ~~lazy()~~
  - ~~StrictMode~~
  - ~~ConcurrentMode~~
  - ~~React Cache~~

# Follow along



```
State } from 'react';  
  
= ( ) => {  
  Count] = useState(0);  
  
count}</div>  
ck={() => setCount(count + 1)}>Increment</Button>
```

- Git repository with slides and code
  - <https://github.com/mauricedb/jsnation-react-hooks>

Type it out  
by hand?

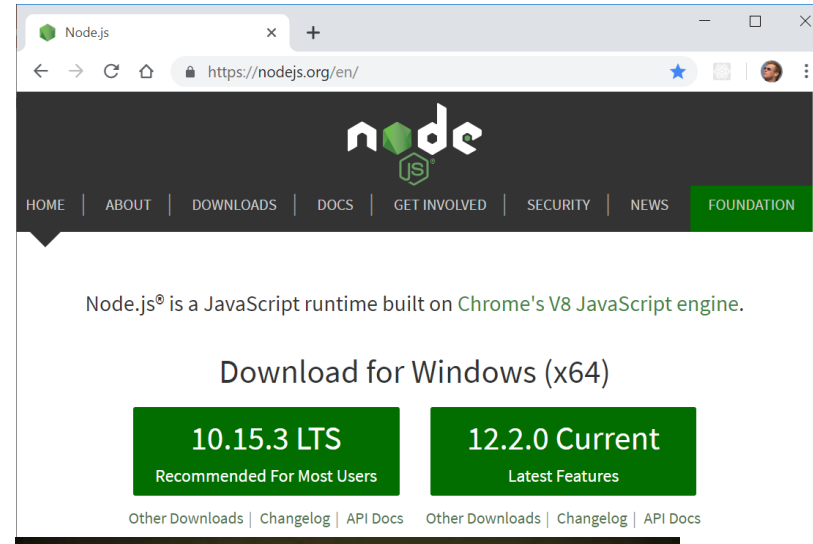
*"Typing it drills it into your brain much better than simply copying and pasting it. You're forming new neuron pathways. Those pathways are going to help you in the future. Help them out now!"*

# Prerequisites

Install Node & NPM



# Install Node.js & NPM



```
Command P...  
C:\>node --version  
v10.15.3  
  
C:\>npm --version  
6.9.0  
  
C:\>yarn --version  
1.16.0  
  
C:\>
```



# Basic hooks

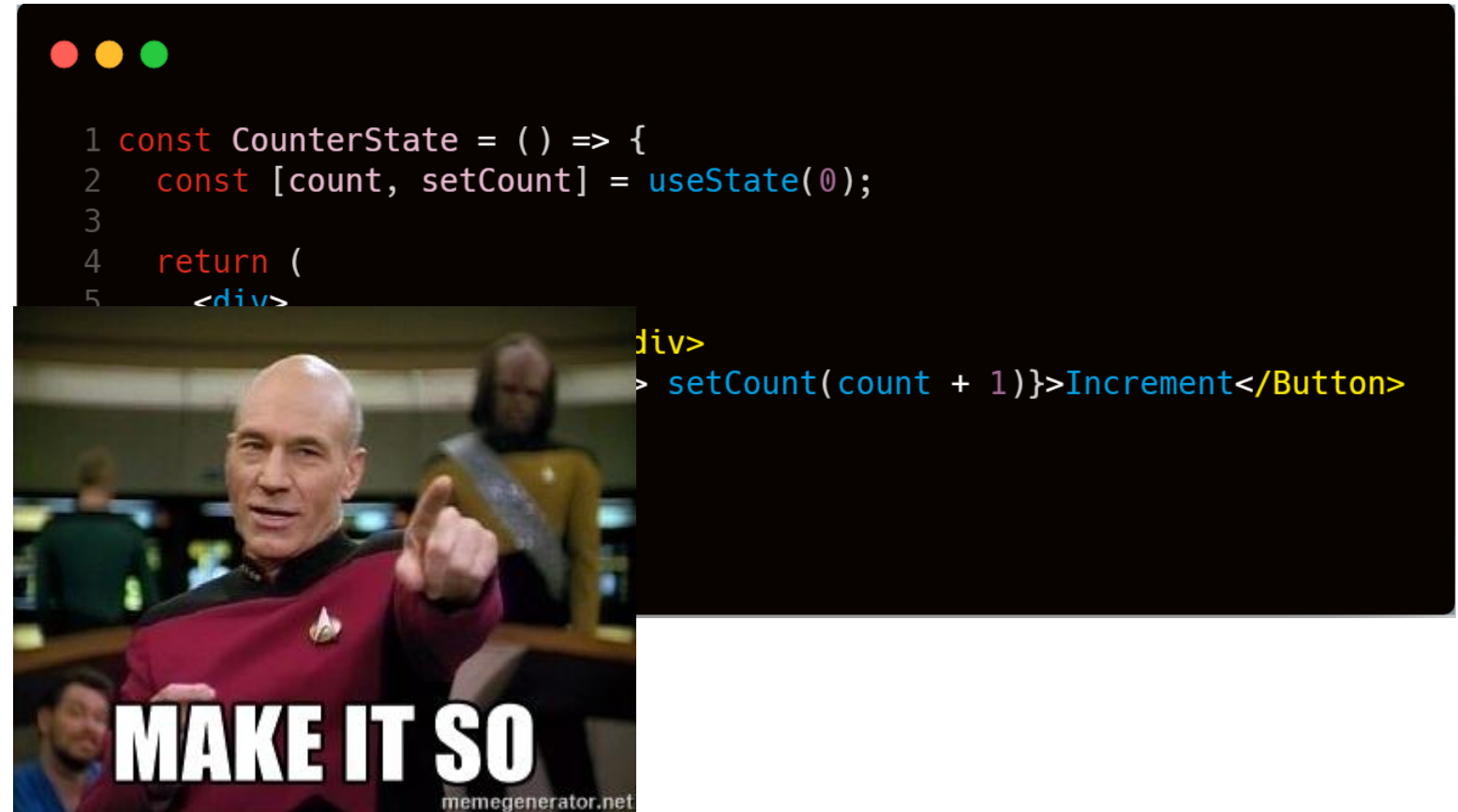
# useState()

- Returns a **persisted stateful value** and a **function to update** it
  - Values can be object or scalar values
- Starts with an **initial value**
  - Can be a lazy initialization function
- The updater function replaces the original state
  - Doesn't merge state like the class based setState()
  - The update value can also be a function

# Class based

```
1 class CounterState extends Component {  
2   state = { count: 0 };  
3  
4   render() {  
5     const { count } = this.state;  
6  
7     return (  
8       <div>  
9         <div>Count: {count}</div>  
10        <Button onClick={() => this.setState({ count: count + 1 })}>  
11          Increment  
12        </Button>  
13      </div>  
14    );  
15  }  
16 }  
17
```

# With Hooks



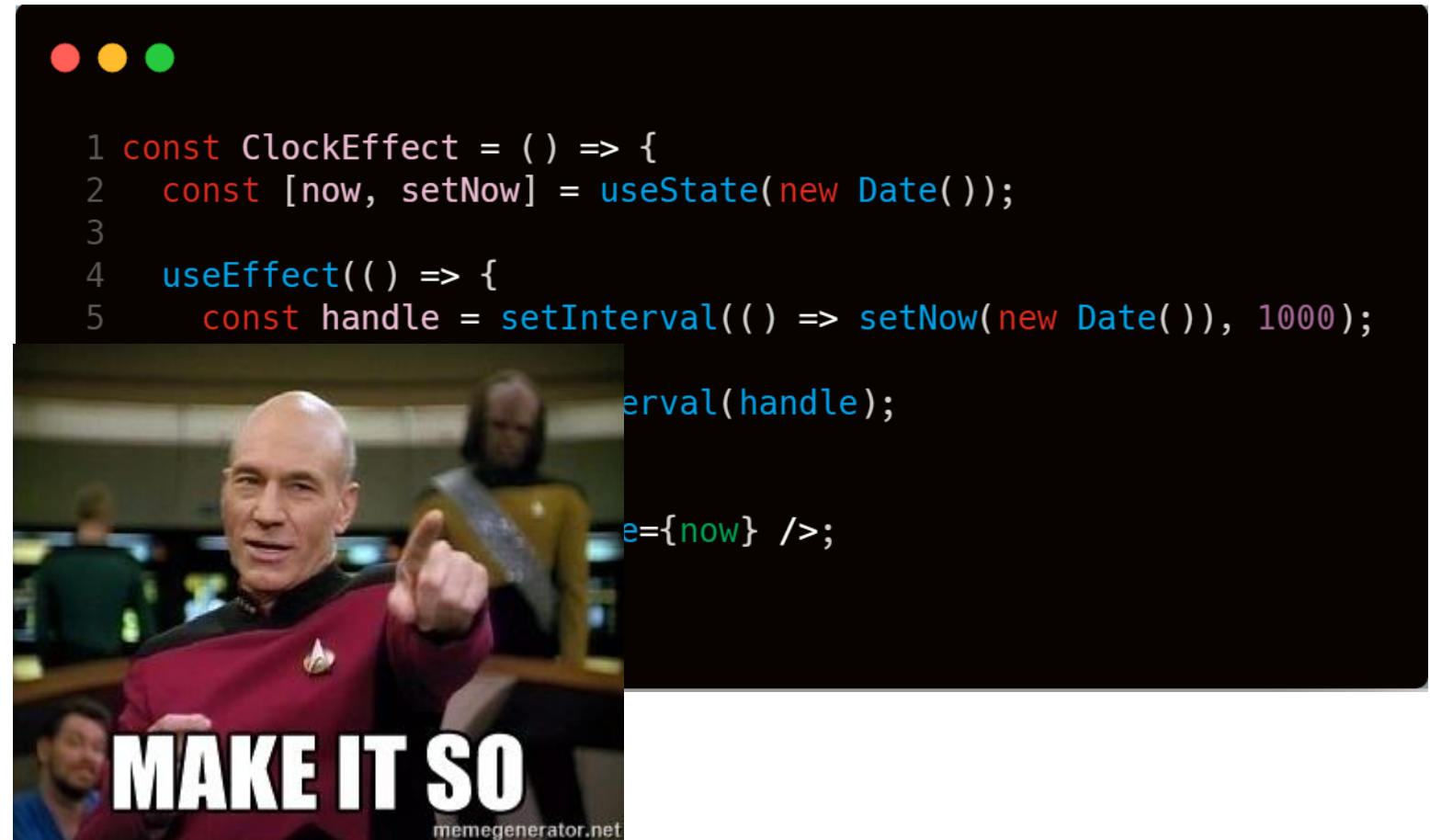
# useEffect()

- Accepts a function that contains **imperative code**
  - The code is used to execute (asynchronous) side effects
- useEffect() fires **after layout and paint**, during a deferred event
  - Use useLayoutEffect() for code that should not be deferred
- Runs both on component **mount** as well as **updates**
  - Control when using the dependencies array
- Optionally return a **cleanup function**

# Class based

```
1 class ClockEffect extends Component {  
2   state = { now: new Date() };  
3   handle = 0;  
4  
5   componentDidMount() {  
6     this.handle = setInterval(() => this.setState({ now: new Date() }), 1000);  
7   }  
8  
9   componentWillUnmount() {  
10    clearInterval(this.handle);  
11  }  
12  
13  render() {  
14    const { now } = this.state;  
15  
16    return <AnalogClock time={now} />;  
17  }  
18 }
```

# With Hooks





# useEffect() with asynchronous actions

- useEffect() is great for **async actions** like AJAX requests
  - 🖐️ Make sure to never return a promise 🖐️
- Use the effect cleanup function to cancel a pending request

# Class based

```
1 class FetchJokes extends Component {
2   state = { jokes: null, error: null, loading: true };
3
4   async componentDidMount() {
5     try {
6       const rsp = await fetch(
7         'http://api.icndb.com/jokes/random/10/?limitTo=[nerdy]&escape=javascript'
8       );
9       if (rsp.ok) {
10        const data = await rsp.json();
11        this.setState({ jokes: data.value, loading: false });
12      } else {
13        throw new Error(rsp.statusText);
14      }
15    } catch (e) {
16      this.setState({ error: e, loading: false });
17    }
18  }
19
20  render() {
21    const { loading, error, jokes } = this.state;
22
23    if (loading) {
24      return <Loading />;
25    }
26
27    if (error) {
28      return <div>{error && error.message}</div>;
29    }
30    return (
31      <ListGroup>
32        {jokes.map(item => (
33          <ListGroup.Item key={item.id}>{item.joke}</ListGroup.Item>
34        ))}
35      </ListGroup>
36    );
37  }
38 }
39
```

# With Hooks



```
1 const FetchJokes = () => {
2   const [state, setState] = useState({
3     jokes: null,
4     error: null,
5     loading: true
6   });
7
8   useEffect(() => {
9     async function fetchData() {
10      try {
11        const rsp = await fetch(
12          'http://api.icndb.com/jokes/random/10/?limitTo=[nerdy]&escape=javascript'
13        );
14        if (rsp.ok) {
15          const data = await rsp.json();
16          setState({ jokes: data.value, error: null, loading: false });
17        } else {
18          throw new Error(rsp.statusText);
19        }
20      } catch (e) {
21        setState({ jokes: null, error: e, loading: false });
22      }
23    }
24  });
25}
```

```
    s } = state;
    r.message}</div>;
    tem.id}>{item.joke}</ListGroup.Item>
```

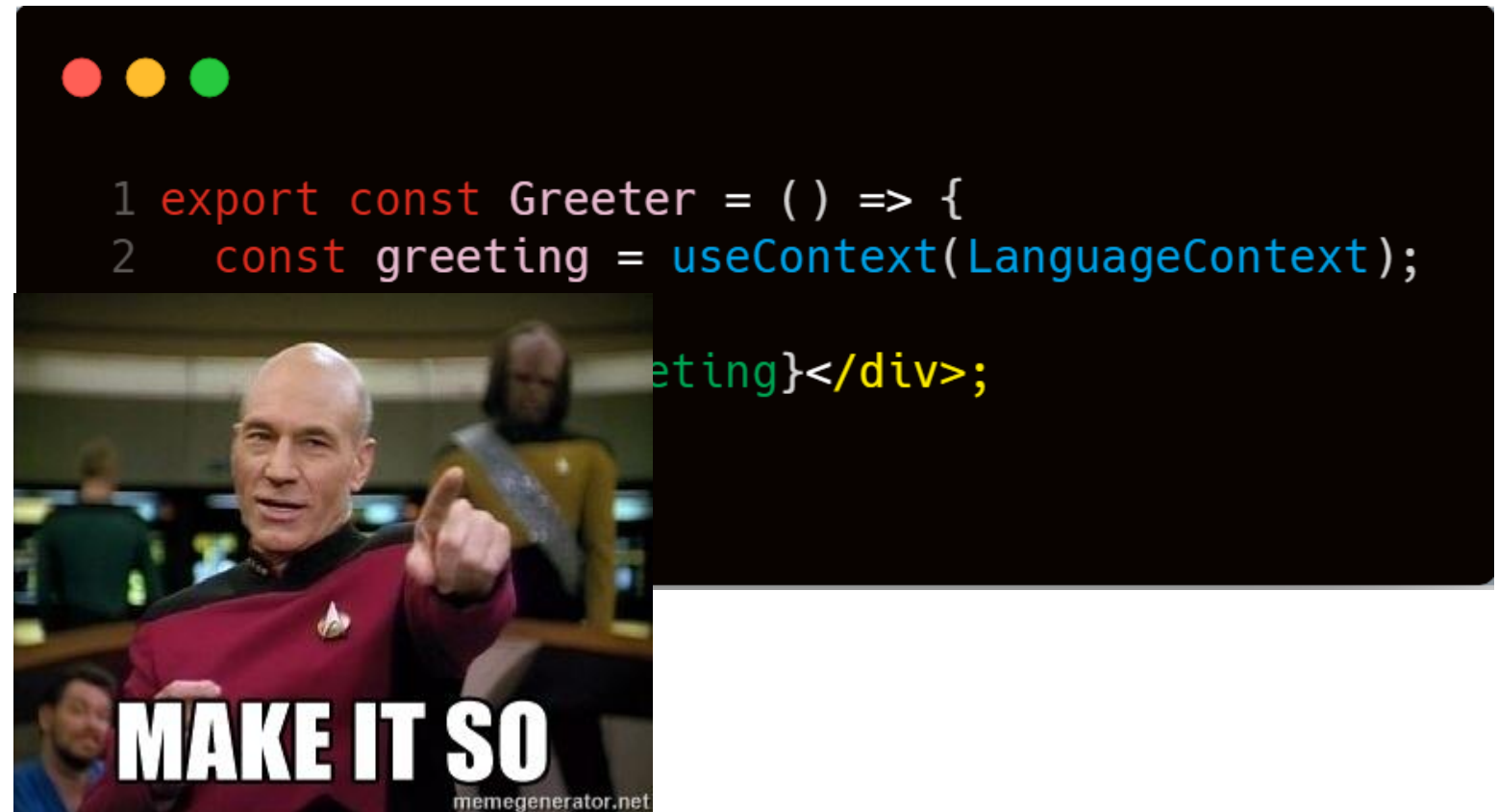
# useContext()

- Accepts a **context object** and returns the current context value
  - Much easier than render props 🐱
- Needs to be part of **<Context.Provider/>** component subtree

# Render props

```
1 export const Greeter = () => {  
2   return (  
3     <LanguageContext.Consumer>  
4       {greeting => <div>{greeting}</div>}  
5     </LanguageContext.Consumer>  
6   );  
7 };  
8
```

# With Hooks



# Custom hooks

- Extract code from a component into a **reusable** custom hook
  - Makes the code more reusable
- **Can use other hooks** as needed
  - Just like a functional component
- Must be named **use**Something()
  - For React to recognize it as a hook
- Publish to NPM for others to use if you like
  - See: <https://nikgraf.github.io/react-hooks/>

# Before

```
1 const FetchWithCustomHooks = () => {
2   const [state, setState] = useState({
3     jokes: null,
4     error: null,
5     loading: true
6   });
7
8   useEffect(() => {
9     async function fetchData() {
10      try {
11        const rsp = await fetch(
12          'http://api.icndb.com/jokes/random/10/?limitTo=[nerdy]&escape=javascript'
13        );
14        if (rsp.ok) {
15          const data = await rsp.json();
16          setState({ jokes: data.value, error: null, loading: false });
17        } else {
18          throw new Error(rsp.statusText);
19        }
20      } catch (e) {
21        setState({ jokes: null, error: e, loading: false });
22      }
23    }
24
25    fetchData();
26  }, []);
27
28  const { loading, error, jokes } = state;
29
30  if (loading) {
31    return <Loading />;
32  }
33
34  if (error) {
35    return <div>{error} && error.message</div>;
36  }
37  return (
38    <ListGroup>
39      {jokes.map(item => (
40        <ListGroup.Item key={item.id}>{item.joke}</ListGroup.Item>
41      ))}
42    </ListGroup>
43  );
44 };
45
```



# The component with custom hook



```
1 const FetchWithCustomHooks = () => {
2   const { loading, error, data: jokes } = useFetch(
3     'http://api.icndb.com/jokes/random/10/?limitTo=[nerdy]&escape=javascript'
4   );
5
6   if (loading) {
7     return <Loading />;
8   }
9
10  if (error) {
11    return <div>{error} && error.message</div>;
12  }
13
14  return (
15    <ListGroup>
16      {jokes.map(item => (
17        <ListGroup.Item key={item.id}>{item.joke}</ListGroup.Item>
18      ))}
19    </ListGroup>
20  );
21 };
22
```

# The custom hook

```
1 export function useFetch(url) {
2   const [state, setState] = useState({
3     data: null,
4     error: null,
5     loading: true
6   });
7
8   useEffect(() => {
9     async function fetchData() {
10      try {
11        const rsp = await fetch(url);
12        if (rsp.ok) {
13          const data = await rsp.json();
14          setState({ data: data.value, error: null, loading: false });
15          sp.statusText);
16        } else {
17          setState({ data: null, error: e, loading: false });
18        }
19      } catch (e) {
20        setState({ data: null, error: e, loading: false });
21      }
22    }
23    fetchData();
24  });
25 }
```



## useDebugValue()

- Displays a label for a custom hook in **React DevTools**
-  Useful for custom Hooks that are part of shared libraries 

# With Hooks

```
1 export function useFetch(url) {  
2   const [state, setState] = useState({  
3     data: null,  
4     error: null,  
5     loading: true  
6   });  
7  
8   useDebugValue(state, state =>  
9     state.loading ? 'loading' : state.error ? 'error' : 'loaded'  
10  );
```



# Additional Hooks

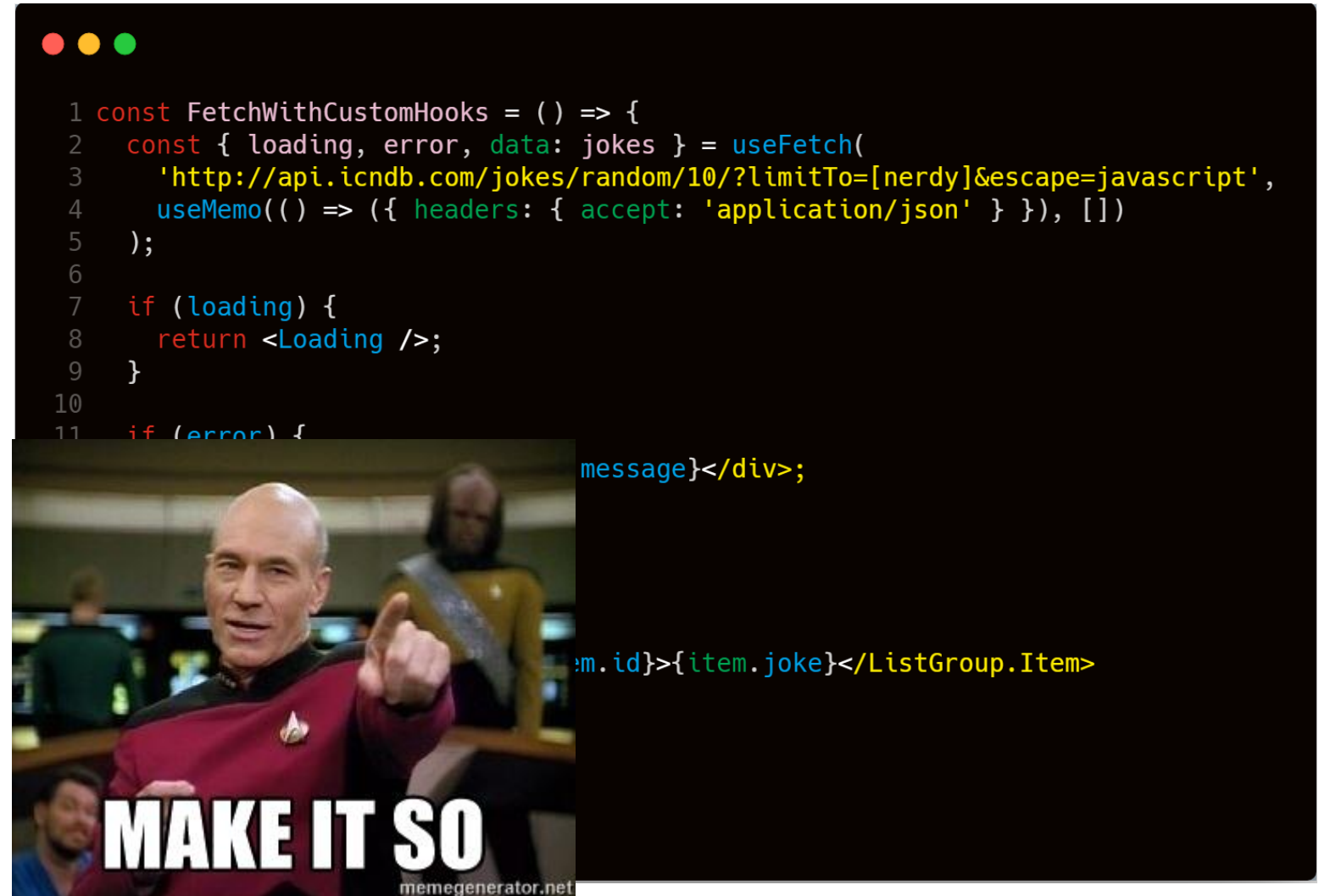
# useMemo()

- Returns a **memoized value**
  - Same inputs return the same result
- It's a performance optimization
  - Not as a guarantee

# Before

```
1 const FetchWithCustomHooks = () => {
2   const { loading, error, data: jokes } = useFetch(
3     'http://api.icndb.com/jokes/random/10/?limitTo=[nerdy]&escape=javascript'
4   );
5
6   if (loading) {
7     return <Loading />;
8   }
9
10  if (error) {
11    return <div>{error} && error.message</div>;
12  }
13
14  return (
15    <ListGroup>
16      {jokes.map(item => (
17        <ListGroup.Item key={item.id}>{item.joke}</ListGroup.Item>
18      ))}
19    </ListGroup>
20  );
21 };
22
```

# With useMemo()





# useReducer()

- A **more powerful** version of useState()
  - Uses a reducer function to manage state
  - Just like Redux
- The **reducer function** always has the same signature
  - (oldState, action) => newState
- Preferred when state logic is more **complex**
  - Much easier to write in a TDD style
- 👉 The dispatch function won't change with re-renders 👉



# With useState()

# Manage state

```
1 const People = () => {
2   const [people, setPeople] = useState(initialPeople);
3   const [selected, setSelected] = useState(initialPeople[0]);
4
5   return (
6     <div className={classes.container}>
7       <Card className={classes.card}>
8         <PeopleList
9           people={people}
10          selected={selected}
11          setSelected={setSelected}
12        />
13      </Card>
14
15      <Card className={classNames(classes.card, classes.editor)}>
16        <PersonEditor
17          people={people}
18          setPeople={setPeople}
19          selected={selected}
20          setSelected={setSelected}
21        />
22      </Card>
23    </div>
24  );
25 };
26
```

# Update selection

```
1 const PeopleList = ({ people, selected, setSelected }) => (  
2   <ListGroup variant="flush">  
3     {people.map(item => (  
4       <ListGroup.Item  
5         key={item.id}  
6         className={classNames({ active: item === selected })}  
7         onClick={() => setSelected(item)}  
8       >  
9         {item.fullName}  
10      </ListGroup.Item>  
11    )})  
12  </ListGroup>  
13 );  
14
```

# Update state

```
1  const PersonEditor = ({ selected, people, setSelected, setPeople }) => (  
2    <Form.Group controlId="fullName">  
3      <Form.Label>Full name</Form.Label>  
4      <Form.Control  
5        type="text"  
6        value={selected.fullName}  
7        onChange={e => {  
8          const newPerson = { ...selected, fullName: e.target.value };  
9          setSelected(newPerson);  
10         setPeople(  
11           people.map(item => {  
12             if (item.id === newPerson.id) {  
13               return newPerson;  
14             }  
15             return item;  
16           })  
17         );  
18       }  
19     }  
20   />  
21 </Form.Group>  
22 );  
23
```



With useReducer()

# useReducer()

```
1 const People = () => {
2   const [state, dispatch] = useReducer(reducer, {
3     people: initialPeople,
4     selected: initialPeople[0]
5   });
6
7   return (
8     <div className={classes.container}>
9       <Card className={classes.card}>
10        <PeopleList
11          people={state.people}
12          selected={state.selected}
13          dispatch={dispatch}
14        />
15      </Card>
16
17      <Card className={classNames(classes.card, classes.editor)}>
18        <PersonEditor selected={state.selected} dispatch={dispatch} />
19      </Card>
20    </div>
21  );
22 };
23
```

# The reducer function

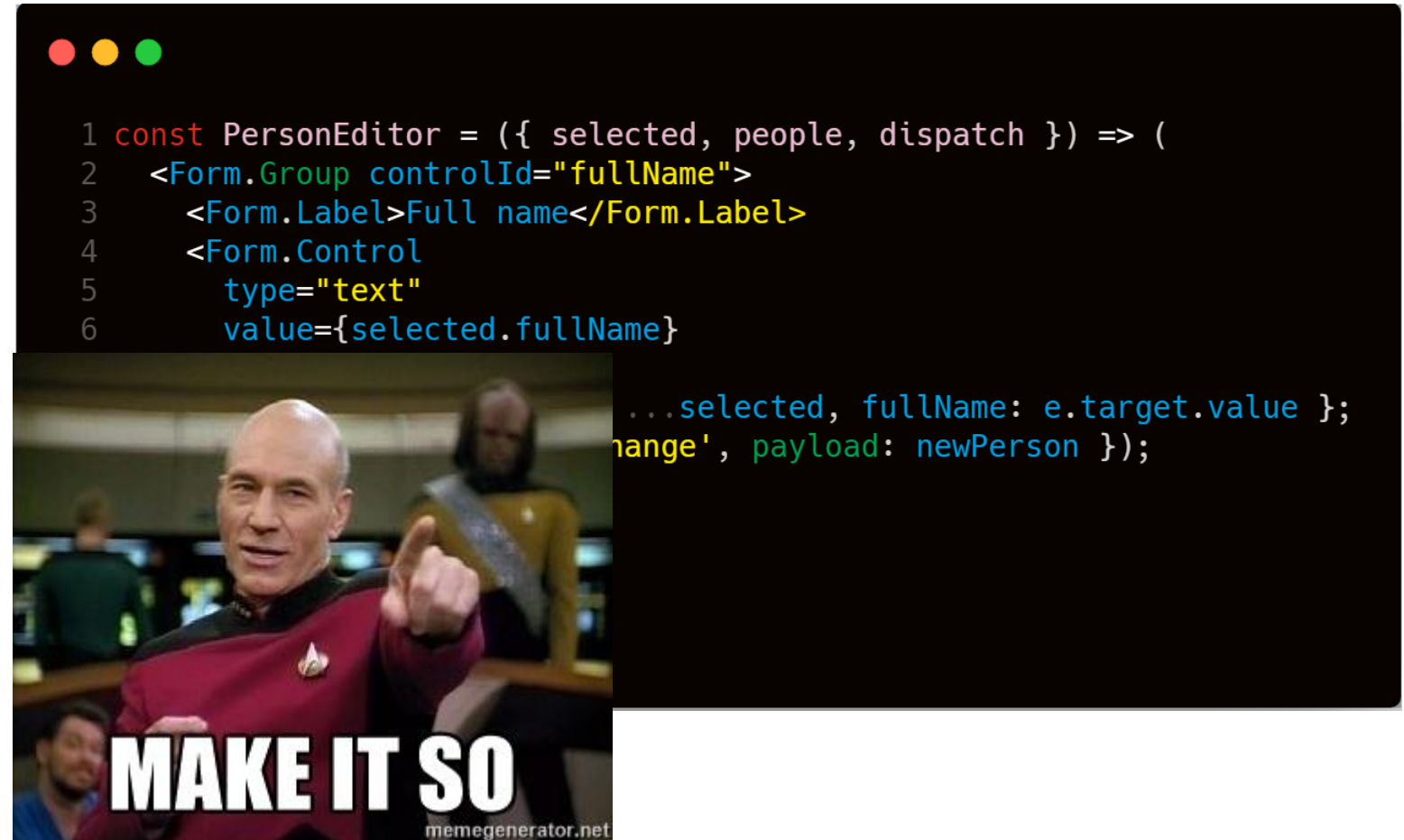
```
1 function reducer(state, action) {
2   switch (action.type) {
3     case 'select':
4       return { ...state, selected: action.payload };
5     case 'change':
6       return {
7         ...state,
8         selected: action.payload,
9         people: state.people.map(item => {
10           if (item.id === action.payload.id) {
11             return action.payload;
12           }
13
14           return item;
15         })
16       };
17
18     default:
19       return state;
20   }
21 }
22
```



# Dispatch selection

```
1 const PeopleList = ({ people, selected, dispatch }) => (  
2   <ListGroup variant="flush">  
3     {people.map(item => (  
4       <ListGroup.Item  
5         key={item.id}  
6         className={classNames({ active: item === selected })}  
7         onClick={() => dispatch({ type: 'select', payload: item })}  
8       >  
9         {item.fullName}  
10      </ListGroup.Item>  
11    )}}  
12  </ListGroup>  
13 );  
14
```

# Dispatch state changes



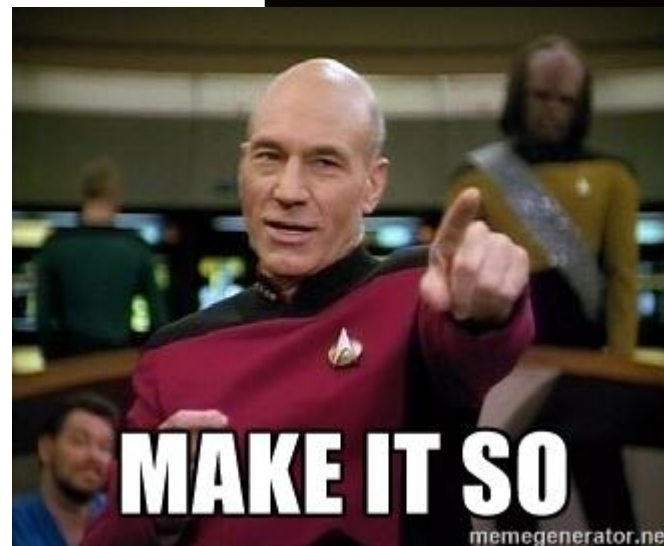
# useRef()

- **Maintain state** between each re-render
  - Without triggering a render on updates
- Typically used for **DOM object** and `useEffect()`
  - But can be used for any kind of state

# Manipulating DOM objects with useRef()

# Using a DOM object

```
1 const PersonEditor = ({ selected, dispatch }) => {  
2   const firstNameRef = useRef();  
3  
4   useEffect(() => {  
5     firstNameRef.current.focus();  
6   }, [selected.id]);  
7  
8   return (  
9     <Form>  
10       <Form.Group controlId="firstName">  
11         <Form.Label>Firstname</Form.Label>
```



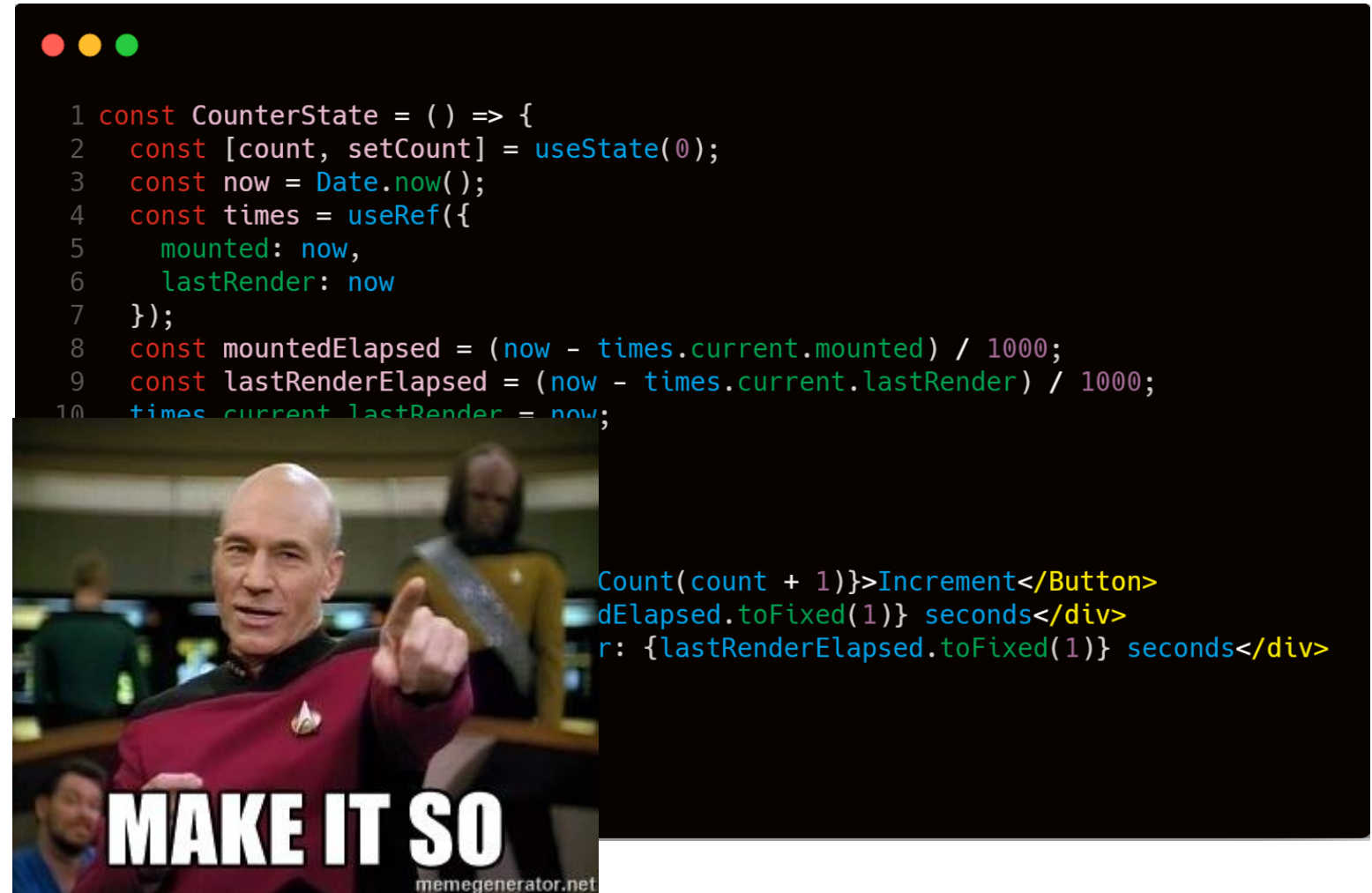
```
    <Form.Control  
      ref={firstNameRef}  
      type="text"  
      value={selected.firstName}  
      onChange={e =>  
        dispatch(e.target.value)}  
    />  
  )>  
</Form>  
</PersonEditor>
```

# Remembering other values using useRef()

# Wrong solution

```
1 let mounted = Date.now();
2 let lastRender = Date.now();
3
4 const CounterState = () => {
5   const [count, setCount] = useState(0);
6   const now = Date.now();
7   const mountedElapsed = (now - mounted) / 1000;
8   const lastRenderElapsed = (now - lastRender) / 1000;
9   lastRender = now;
10
11   return (
12     <div>
13       <div>Count: {count}</div>
14       <Button onClick={() => setCount(count + 1)}>Increment</Button>
15       <div>Mounted time: {mountedElapsed.toFixed(1)} seconds</div>
16       <div>Time since last render: {lastRenderElapsed.toFixed(1)} seconds</div>
17     </div>
18   );
19 };
20
```

# With useRef()





Maurice de Beijer

@mauricedb



# Concurrent React

# lazy()

- Create a component that is **loaded dynamically**
  - With automatic bundle splitting by Webpack
- Requires a **<Suspense />** component as a parent
  - The fallback UI is required
  - Typically a spinner or something similar

# Eager Loading

```
1 import People from './people';
2
3 const LazyPeople = () => {
4   const [selected, setSelected] = useState(false);
5
6   return (
7     <div>
8       <Form.Check
9         type="checkbox"
10        checked={selected}
11        onChange={() => setSelected(!selected)}
12        label="Display characters"
13      />
14
15      {selected && <People />}
16    </div>
17  );
18 };
19
```

# Lazy Loading



```

1 const People = lazy(() => import('./people'));
2
3 const LazyPeople = () => {
4   const [selected, setSelected] = useState(false);
5
6   return (
7     <div>
8       <Form.Check
9         type="checkbox"
10        checked={selected}
11        () => setSelected(!selected)}
12       play characters"
13
14       llback={<Loading />}>
15       && <People />}

```



# StrictMode

- The `<StrictMode />` component helps **prepare for async rendering**
  - Warns about unsafe lifecycle functions like `componentWillMount()`
  - As well as other deprecated React features
- Will try to detect **illegal side effects** by rendering twice
  - The same applies to some state management functions
-  Does **nothing** in a **production build** 
  - Will make a development build run slower

# Not strict

```
1 const StrictAndLazyPeople = () => {
2   const [selected, setSelected] = useState(false);
3
4   return (
5     <>
6       <Form.Check
7         type="checkbox"
8         checked={selected}
9         onChange={() => setSelected(!selected)}
10        label="Display characters"
11      />
12
13      <Suspense fallback={<Loading />}>
14        {selected && <People />}
15      </Suspense>
16    </>
17  );
18 };
19
```

# Using <StrictMode/>

```
1 const StrictAndLazyPeople = () => {
2   const [selected, setSelected] = useState(false);
3
4   return (
5     <StrictMode>
6       <Form.Check
7         type="checkbox"
8         checked={selected}
9         onChange={() => setSelected(!selected)}
10        label="Display characters"
11      />
12
13      <Suspense fallback={<Loading />}>
14        {selected && <People />}
15      </Suspense>
16    </StrictMode>
17  );
18 };
19
```

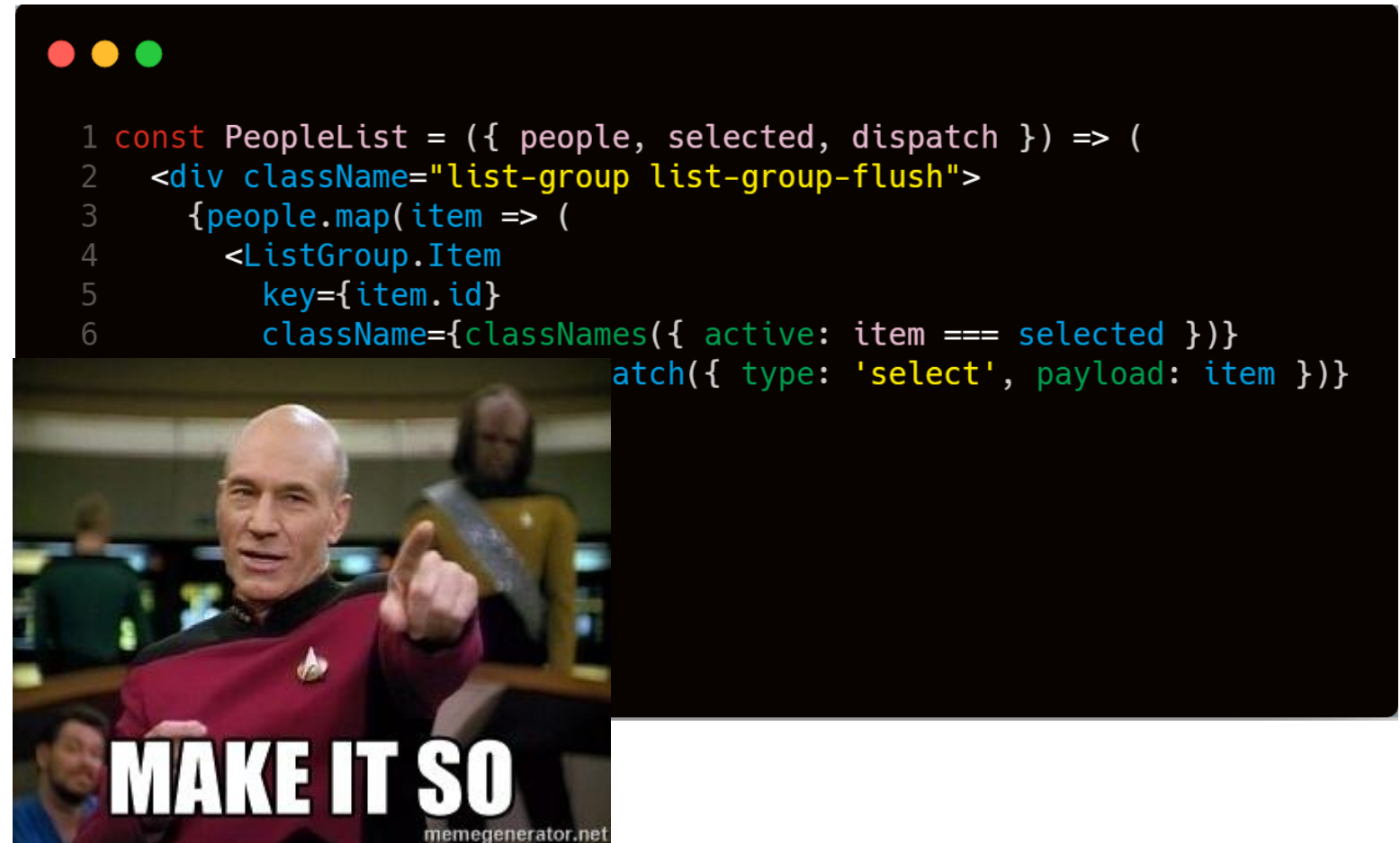


# Fixing the strict error

# Before

```
1 const PeopleList = ({ people, selected, dispatch }) => (  
2   <ListGroup variant="flush">  
3     {people.map(item => (  
4       <ListGroup.Item  
5         key={item.id}  
6         className={classNames({ active: item === selected })}  
7         onClick={() => dispatch({ type: 'select', payload: item })}  
8       >  
9         {item.fullName}  
10      </ListGroup.Item>  
11    )}}  
12  </ListGroup>  
13 );  
14
```

No more  
<ListGroup/>

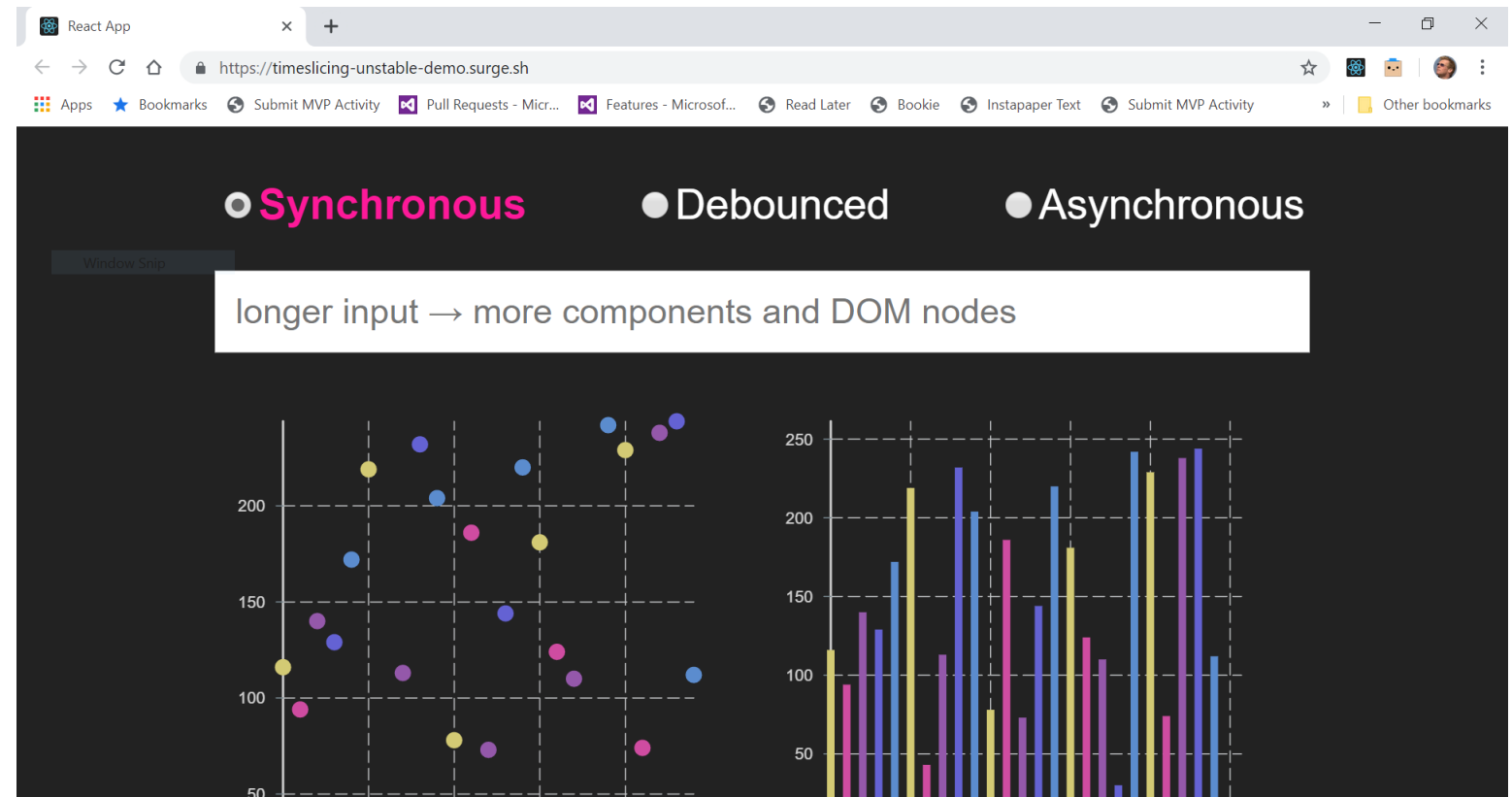


# Concurrent mode



- React can render in **concurrent mode**
  - Rendering happens in time slices
- React can **prioritize user events** over other work
  - Results in more responsive applications
- Either for the whole application using `ReactDOM.createRoot()`
  - Or a component subtree using `<ConcurrentMode/>`

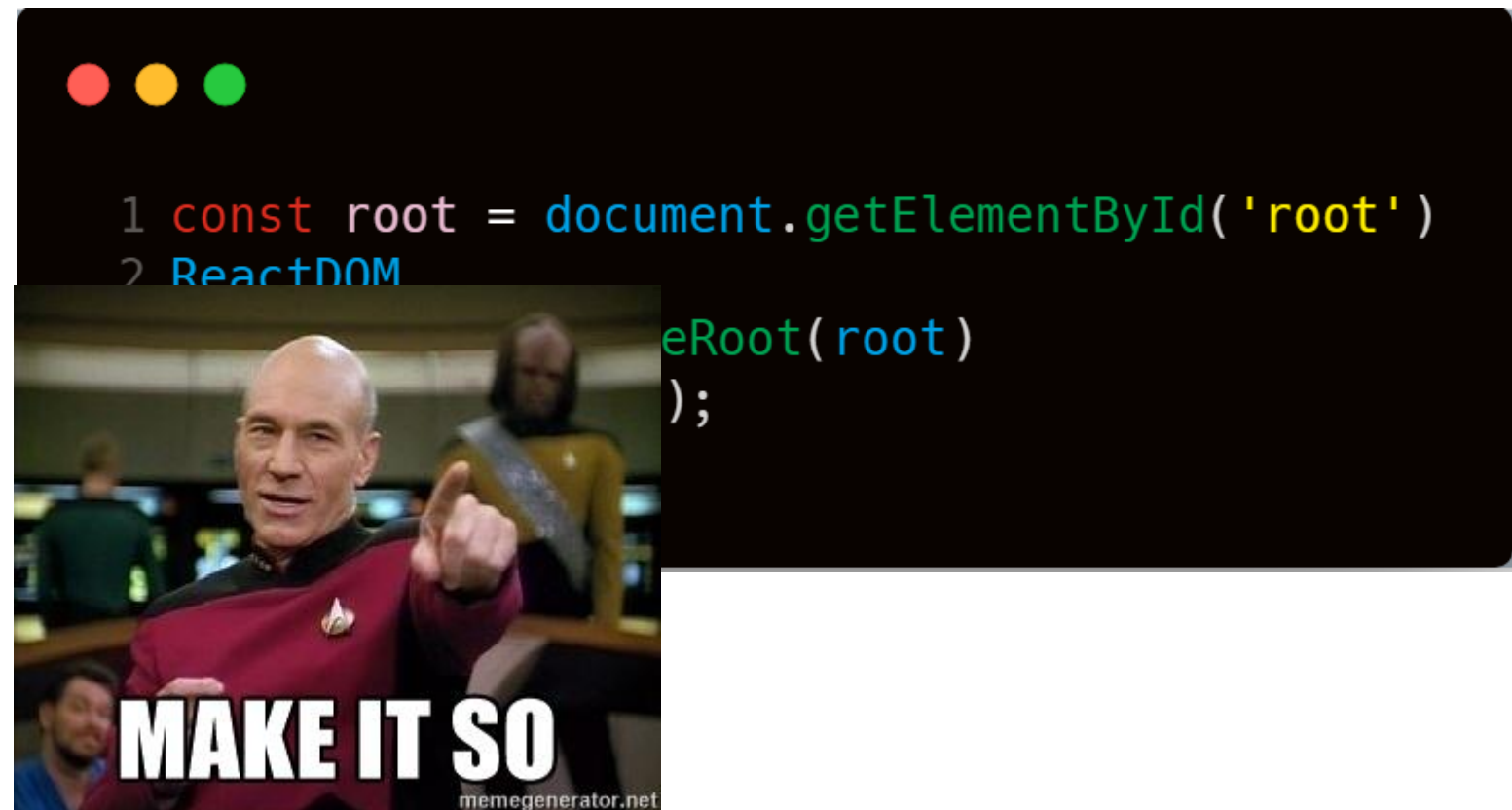
# Time Slicing Demo



# Current rendering

```
1 const root = document.getElementById('root')  
2 ReactDOM.render(<App />, root);  
3
```

# Concurrent rendering



# React Cache



- Creates a resource manager to **load data asynchronous**
  - Can work with anything that return a promise
- Can be called from a components **render()** function
  - Will pause rendering until the promise resolves
- 💔 The NPM package is broken for the current React version 💔



# Rendering asynchronous data

```
1 import jokesResource from './jokes-resource';
2
3 const FetchJokes = () => {
4   const { error, jokes } = jokesResource.read();
5
6   if (error) {
7     return <div>{error} && {error.message}</div>;
8   }
9   return (
10     <ListGroup>
11       {jokes.map(item => (
12         <ListGroup.Item key={item.id}>
13           {item.joke}
14         </ListGroup.Item>
15       ))}
16     </ListGroup>
17   );
18 };
19
```

# The asynchronous resource

```
1 import { unstable_createResource } from 'react-cache';
2
3 const jokesResource = unstable_createResource(async () => {
4   try {
5     const rsp = await fetch(
6       'http://api.icndb.com/jokes/random/10/?limitTo=[nerdy]&escape=javascript'
7     );
8     if (rsp.ok) {
9       const data = await rsp.json();
10      return { jokes: data.value, error: null };
11    } else {
12      throw new Error(rsp.statusText);
13    }
14  } catch (e) {
15    return { jokes: null, error: e };
16  }
17 });
18
```

# Conclusion

- The future of React is bright
- Functional components and hooks is the future of components
  - But class based components will continue to work
- Concurrent React makes large complex application responsive
  - Easy to implement in most cases
- React Cache will make data loading easier
  - But not quite yet

Maurice de Beijer

@mauricedb

