The new React

Maurice de Beijer - @mauricedb

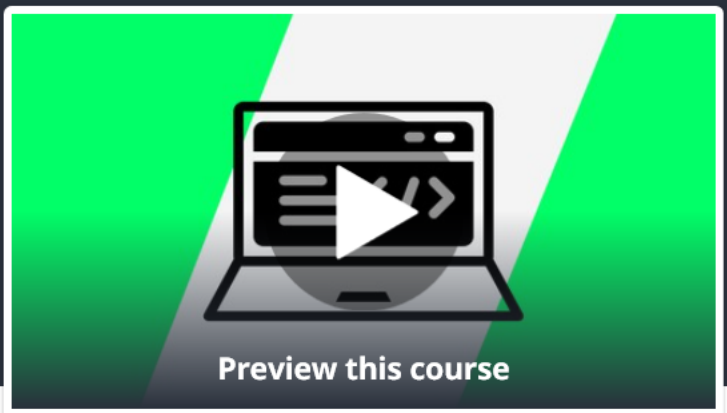
- Maurice de Beijer
- The Problem Solver
- Microsoft MVP
- Freelance developer/instructor
- Twitter: @mauricedb
- Web: http://www.TheProblemSolver.nl
- E-mail: maurice.de.beijer@gmail.com

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]



€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

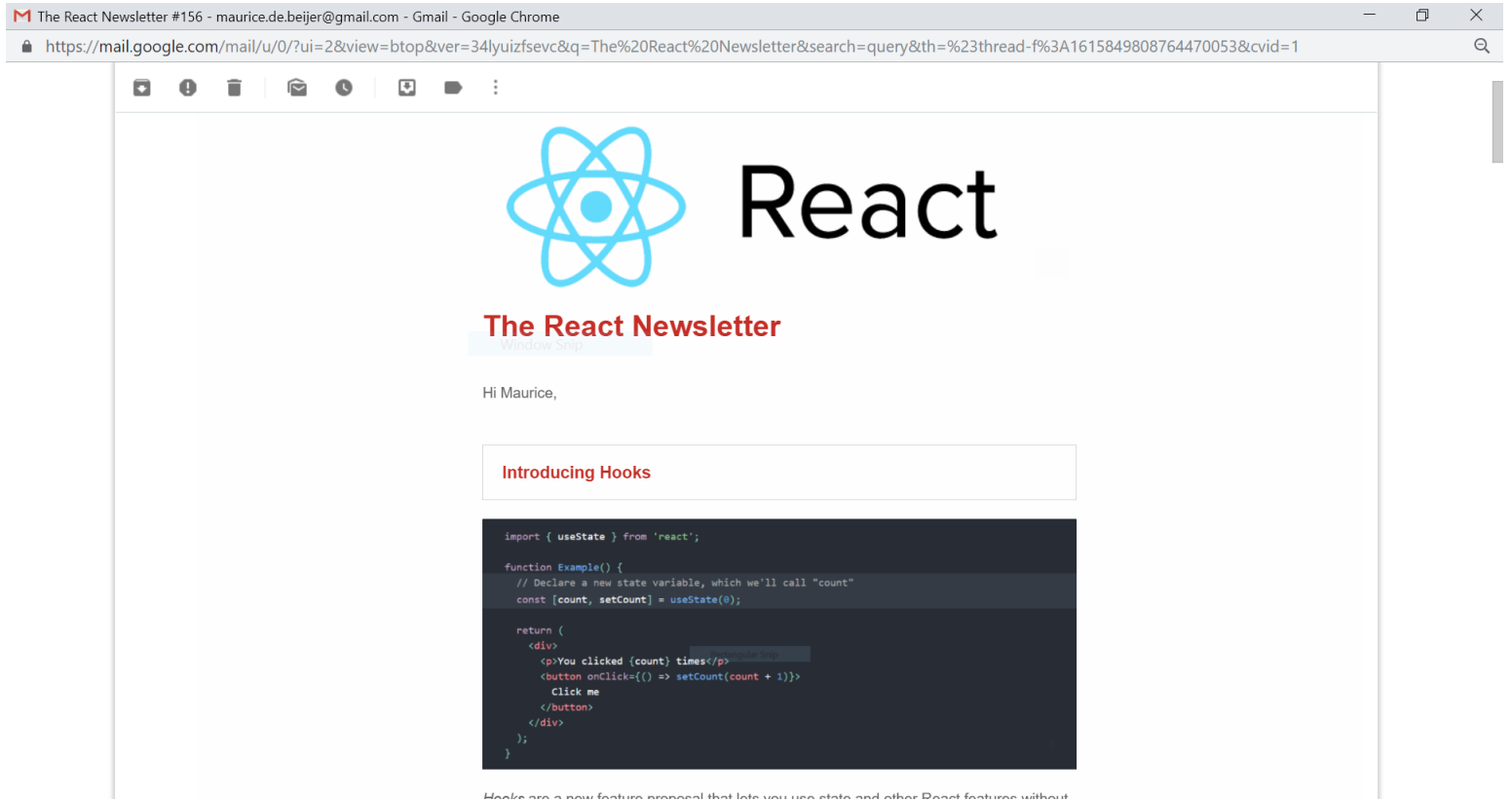
What you'll learn

- 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.

The React Newsletter

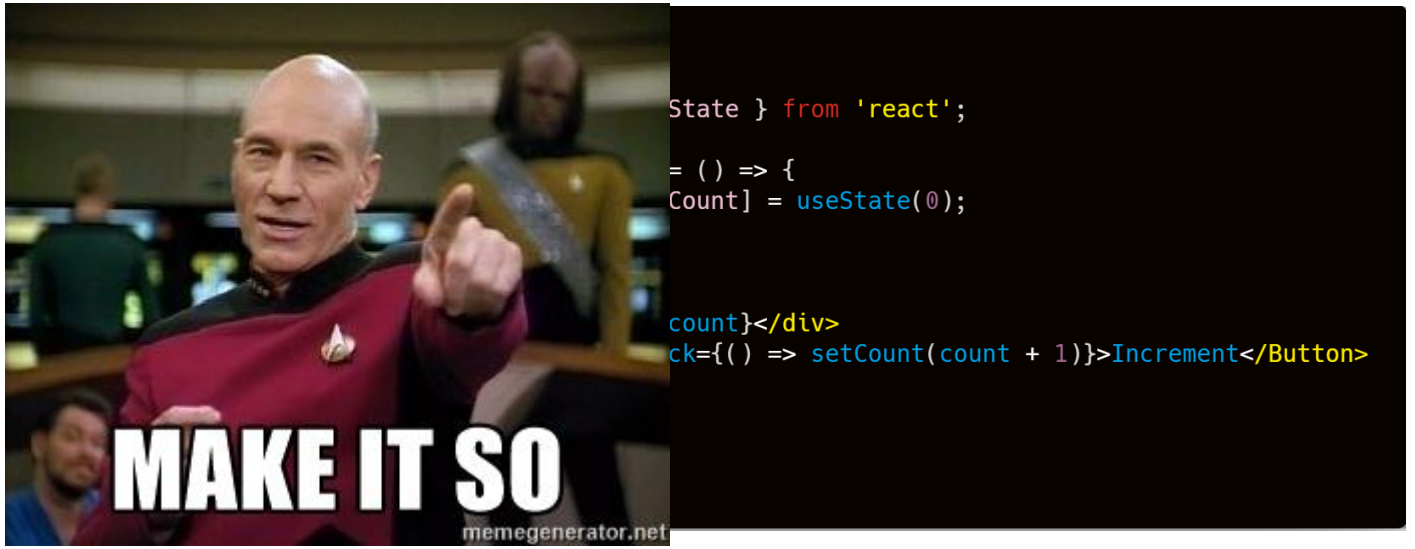


http://bit.ly/ReactNewsletter

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



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

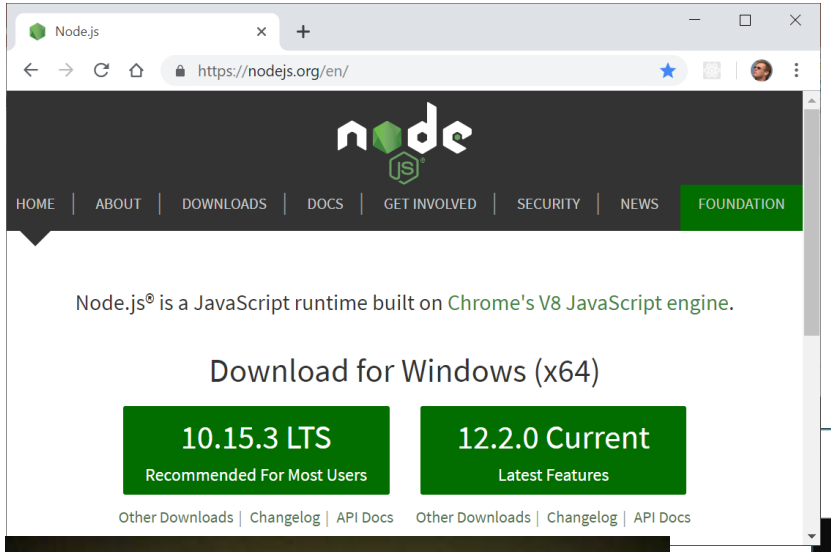
Type it out by hand?

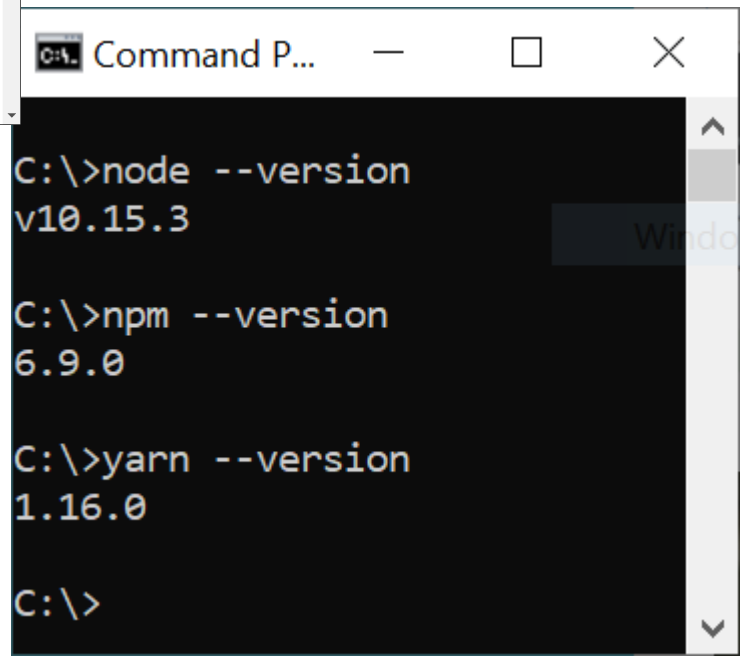
"Typing it drills it into your brain much better than simply copying and pasting it. You're forming new neuron pathways. Those pathways are going to help you in the future. Help them out now!"

Prerequisites

Install Node & NPM

Install Node.js & NPM





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

```
class CounterState extends Component {
  state = { count: 0 };

  render() {
    const { count } = this.state;

    return (
      <div>
        <div>Count: {count}</div>
        <Button onClick={() => this.setState({ count: count + 1 })}>
          Increment
        </Button>
      </div>
    );
  }
}

```

With Hooks

```
1 const CounterState = () => {
2   const [count, setCount] = useState(0);
3
4   return (
5     <div>
```

```
div>
> setCount(count + 1)}>Increment</Button>
```

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

```
class ClockEffect extends Component {
  state = { now: new Date() };
  handle = 0;

  componentDidMount() {
    this.handle = setInterval(() => this.setState({ now: new Date() }), 1000);
  }

  componentWillUnmount() {
    clearInterval(this.handle);
  }

  render() {
    const { now } = this.state;

    return <AnalogClock time={now} />;
  }
}
```

With Hooks

```
1 const ClockEffect = () => {
2   const [now, setNow] = useState(new Date());
3
4   useEffect(() => {
5     const handle = setInterval(() => setNow(new Date()), 1000);

erval(handle);

e={now} />;
```

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

```
class FetchJokes extends Component {
  state = { jokes: null, error: null, loading: true };

  async componentDidMount() {
    try {
      const rsp = await fetch(
        'http://api.icndb.com/jokes/random/10/?limitTo=[nerdy]&escape=javascript'
      );
      if (rsp.ok) {
        const data = await rsp.json();
        this.setState({ jokes: data.value, loading: false });
      } else {
        throw new Error(rsp.statusText);
      }
    } catch (e) {
      this.setState({ error: e, loading: false });
    }
  }

  render() {
    const { loading, error, jokes } = this.state;

    if (loading) {
      return <Loading />;
    }

    if (error) {
      return <div>{error && error.message}</div>;
    }
    return (
      <ListGroup>
        {jokes.map(item => (
          <ListGroup.Item key={item.id}>{item.joke}</ListGroup.Item>
        ))}
      </ListGroup>
    );
  }
}

```

With Hooks

```
const FetchJokes = () => {
  const [state, setState] = useState({
    jokes: null,
    error: null,
    loading: true
  });

  useEffect(() => {
    async function fetchData() {
      try {
        const rsp = await fetch(
          'http://api.icndb.com/jokes/random/10/?limitTo=[nerdy]&escape=javascript'
        );
        if (rsp.ok) {
          const data = await rsp.json();
          setState({ jokes: data.value, error: null, loading: false });
        } else {
          throw new Error(rsp.statusText);
        }
      } catch (e) {
        setState({ jokes: null, error: e, loading: false });
```

useContext()

- Accepts a **context object** and returns the current context value
  - Much easier then 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

```
1 export const Greeter = () => {
2   const greeting = useContext(LanguageContext);
                                  eting}</div>;
```

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

```
const FetchWithCustomHooks = () => {
  const [state, setState] = useState({
    jokes: null,
    error: null,
    loading: true
  });

  useEffect(() => {
    async function fetchData() {
      try {
        const rsp = await fetch(
          'http://api.icndb.com/jokes/random/10/?limitTo=[nerdy]&escape=javascript'
        );
        if (rsp.ok) {
          const data = await rsp.json();
          setState({ jokes: data.value, error: null, loading: false });
        } else {
          throw new Error(rsp.statusText);
        }
      } catch (e) {
        setState({ jokes: null, error: e, loading: false });
      }
    }

    fetchData();
  }, []);

  const { loading, error, jokes } = state;

  if (loading) {
    return <Loading />;
  }

  if (error) {
    return <div>{error && error.message}</div>;
  }
  return (
    <ListGroup>
      {jokes.map(item => (
        <ListGroup.Item key={item.id}>{item.joke}</ListGroup.Item>
      ))}
    </ListGroup>
  );
};

```

The component with custom hook

```jsx
const FetchWithCustomHooks = () => {
  const { loading, error, data: jokes } = useFetch(
    'http://api.icndb.com/jokes/random/10/?limitTo=[nerdy]&escape=javascript'
  );

  if (loading) {
    return <Loading />;
  }

  if (error) {
    return <div>{error && error.message}</div>;
  }

  return (
    <ListGroup>
      {jokes.map(item => (
        <ListGroup.Item key={item.id}>{item.joke}</ListGroup.Item>
      ))}
    </ListGroup>
  );
};

```

The custom hook

```
export function useFetch(url) {
  const [state, setState] = useState({
    data: null,
    error: null,
    loading: true
  });

  useEffect(() => {
    async function fetchData() {
      try {
        const rsp = await fetch(url);
        if (rsp.ok) {
          const data = await rsp.json();
                                    data.value, error: null, loading: false });

                                    sp.statusText);


                                    ll, error: e, loading: false });
```

useDebugValue()

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

With Hooks

```
export function useFetch(url) {
  const [state, setState] = useState({
    data: null,
    error: null,
    loading: true
  });

  useDebugValue(state, state =>
    state.loading ? 'loading' : state.error ? 'error' : 'loaded'
  );
```

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()

```
1 const FetchWithCustomHooks = () => {
2   const { loading, error, data: jokes } = useFetch(
3     'http://api.icndb.com/jokes/random/10/?limitTo=[nerdy]&escape=javascript',
4     useMemo(() => ({ headers: { accept: 'application/json' } }), [])
5   );
6
7   if (loading) {
8     return <Loading />;
9   }
10
11  if (error) {
```

```
message}</div>;
```

```
m.id}>{item.joke}</ListGroup.Item>
```

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

```jsx
const People = () => {
  const [people, setPeople] = useState(initialPeople);
  const [selected, setSelected] = useState(initialPeople[0]);

  return (
    <div className={classes.container}>
      <Card className={classes.card}>
        <PeopleList
          people={people}
          selected={selected}
          setSelected={setSelected}
        />
      </Card>

      <Card className={classNames(classes.card, classes.editor)}>
        <PersonEditor
          people={people}
          setPeople={setPeople}
          selected={selected}
          setSelected={setSelected}
        />
      </Card>
    </div>
  );
};

```

Update selection

```jsx
const PeopleList = ({ people, selected, setSelected }) => (
  <ListGroup variant="flush">
    {people.map(item => (
      <ListGroup.Item
        key={item.id}
        className={classNames({ active: item === selected })}
        onClick={() => setSelected(item)}
      >
        {item.fullName}
      </ListGroup.Item>
    ))}
  </ListGroup>
);

```

Update state

```jsx
const PersonEditor = ({ selected, people, setSelected, setPeople }) => (
  <Form.Group controlId="fullName">
    <Form.Label>Full name</Form.Label>
    <Form.Control
      type="text"
      value={selected.fullName}
      onChange={e => {
        const newPerson = { ...selected, fullName: e.target.value };
        setSelected(newPerson);
        setPeople(
          people.map(item => {
            if (item.id === newPerson.id) {
              return newPerson;
            }

            return item;
          })
        );
      }}
    />
  </Form.Group>
);

```

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

```
function reducer(state, action) {
  switch (action.type) {
    case 'select':
      return { ...state, selected: action.payload };
    case 'change':
      return {
        ...state,
        selected: action.payload,
        people: state.people.map(item => {
          if (item.id === action.payload.id) {
            return action.payload;
          }

          return item;
        })
      };

    default:
      return state;
  }
}

```

Dispatch selection

```jsx
const PeopleList = ({ people, selected, dispatch }) => (
  <ListGroup variant="flush">
    {people.map(item => (
      <ListGroup.Item
        key={item.id}
        className={classNames({ active: item === selected })}
        onClick={() => dispatch({ type: 'select', payload: item })}
      >
        {item.fullName}
      </ListGroup.Item>
    ))}
  </ListGroup>
);

```

Dispatch state changes

```
1 const PersonEditor = ({ selected, people, dispatch }) => (
2   <Form.Group controlId="fullName">
3     <Form.Label>Full name</Form.Label>
4     <Form.Control
5       type="text"
6       value={selected.fullName}
```

```
...selected, fullName: e.target.value };
nange', payload: newPerson });
```

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>
                                    rol
                                    stNameRef}
                                    xt"
                                    elected.firstName}
                                    ={e =>
                                    >
```

Remembering other values using useRef()

Wrong solution

```
let mounted = Date.now();
let lastRender = Date.now();

const CounterState = () => {
  const [count, setCount] = useState(0);
  const now = Date.now();
  const mountedElapsed = (now - mounted) / 1000;
  const lastRenderElapsed = (now - lastRender) / 1000;
  lastRender = now;

  return (
    <div>
      <div>Count: {count}</div>
      <Button onClick={() => setCount(count + 1)}>Increment</Button>
      <div>Mounted time: {mountedElapsed.toFixed(1)} seconds</div>
      <div>Time since last render: {lastRenderElapsed.toFixed(1)} seconds</div>
    </div>
  );
};

```

With useRef()

```
1 const CounterState = () => {
2   const [count, setCount] = useState(0);
3   const now = Date.now();
4   const times = useRef({
5     mounted: now,
6     lastRender: now
7   });
8   const mountedElapsed = (now - times.current.mounted) / 1000;
9   const lastRenderElapsed = (now - times.current.lastRender) / 1000;
10  times.current.lastRender = now;
```


```
Count(count + 1)}>Increment</Button>
dElapsed.toFixed(1)} seconds</div>
r: {lastRenderElapsed.toFixed(1)} seconds</div>
```

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

```jsx
import People from './people';

const LazyPeople = () => {
  const [selected, setSelected] = useState(false);

  return (
    <div>
      <Form.Check
        type="checkbox"
        checked={selected}
        onChange={() => setSelected(!selected)}
        label="Display characters"
      />

      {selected && <People />}
    </div>
  );
};

```

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}
                  () => setSelected(!selected)}
                  olay characters"


                  llback={<Loading />}>
                  && <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/>

```
const StrictAndLazyPeople = () => {
  const [selected, setSelected] = useState(false);

  return (
    <StrictMode>
      <Form.Check
        type="checkbox"
        checked={selected}
        onChange={() => setSelected(!selected)}
        label="Display characters"
      />

      <Suspense fallback={<Loading />}>
        {selected && <People />}
      </Suspense>
    </StrictMode>
  );
};

```

Fixing the strict error

Before

```
const PeopleList = ({ people, selected, dispatch }) => (
  <ListGroup variant="flush">
    {people.map(item => (
      <ListGroup.Item
        key={item.id}
        className={classNames({ active: item === selected })}
        onClick={() => dispatch({ type: 'select', payload: item })}
      >
        {item.fullName}
      </ListGroup.Item>
    ))}
  </ListGroup>
);

```

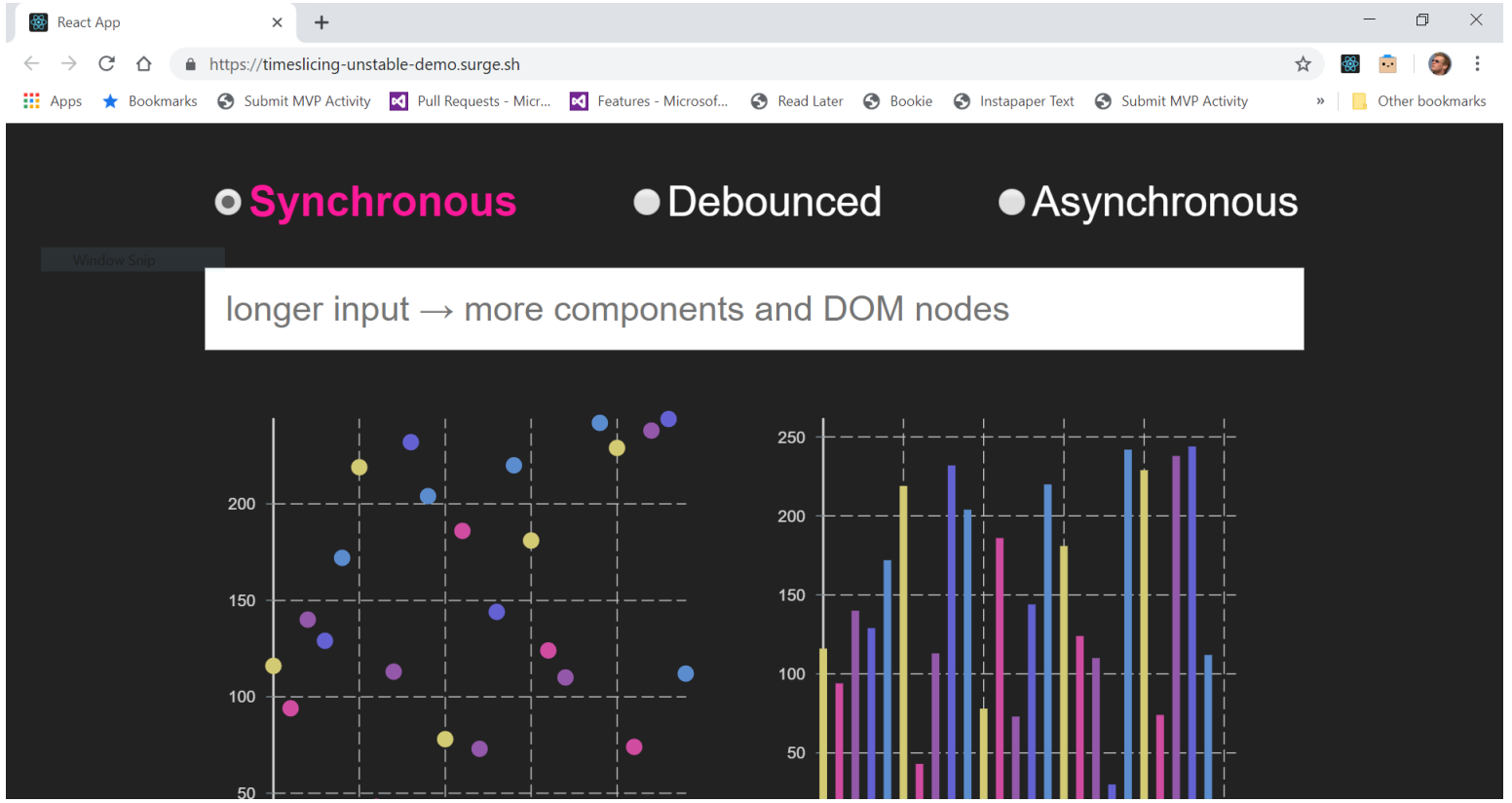
No more <ListGroup/>

```
1 const PeopleList = ({ people, selected, dispatch }) => (
2   <div className="list-group list-group-flush">
3     {people.map(item => (
4       <ListGroup.Item
5         key={item.id}
6         className={classNames({ active: item === selected })}
                                  atch({ type: 'select', payload: item })}
```

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

```
1 const root = document.getElementById('root')
2 ReactDOM
                                    eRoot(root)
                                    );
```

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