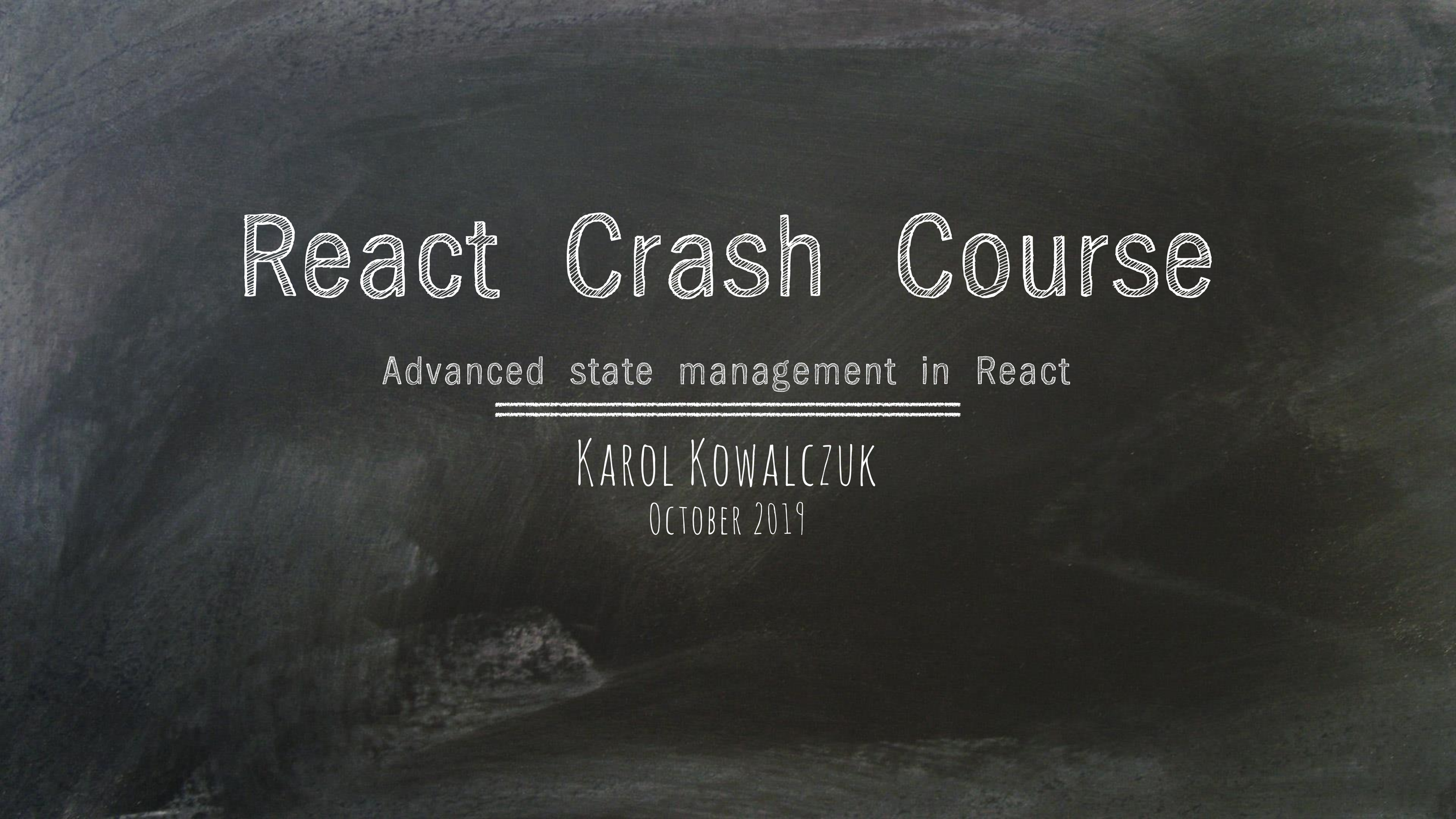

React Crash Course
Advanced state management in React
Karol Kowalczuk
October 2019

Agenda

1. Logic sharing concept evolution
2. React hooks and Context API
3. You do not need Redux
4. Unit testing
5. More fancy features of modern React

Logic sharing concept evolution 1

* In React world everything is a component
* Due to React's nature, it's pretty straightforward to reuse components in the component hierarchy tree
* What about sharing the logic (state management, lifecycle hooks, etc.) between multiple components of different type?

Logic sharing concept evolution 2

Long time ago, when React emerged and shocked the frontend community around the globe, the main technique used for sharing logic between components was the implementation of Higher Order Components (HOC)

YEAH, „H" STANDS FOR „H" ...

Logic sharing concept evolution 3

HOC

* Just a function that takes a component as an input and returns wrapped component
* Multiple „wrappers” can be easily composed
* Wrapped component can still be tested in the isolation from the HOC

Logic sharing concept evolution 4

```
class Orders extends React.Component {
  constructor(props) {
    super(props)
    this.state = { metadata: undefined }

    this.clearCart = this.clearCart.bind(this)
  }

  componentDidMount() {
    getMetadata("orders")
      .then(m => this.setState({ metadata: m }))
  }

  componentWillUnmount() {
    sendMetadata("orders", this.state.metadata)
  }
```

```
  clearCart() {
    this.setState({ metadata: { ...this.state.metadata, items: [] } })
  }

  render() {
    const { metadata } = this.state

    return metadata
      ? (
        <div className="orders-wrapper">
          <p>Your cart contains {metadata.items.length} items.</p>
          <button onClick={this.clearCart}>Clear cart</button>
        </div>
      )
      : <Spinner />
  }
}
```

Logic sharing concept evolution 5

```
class Favorites extends React.Component {
  constructor(props) {
    super(props)
    this.state = { metadata: undefined }

    this.deleteFavorite = this.deleteFavorite.bind(this)
  }

  componentDidMount() {
    getFavorites()
      .then(m => this.setState({ metadata: m }))
  }

  componentWillUnmount() {
    sendMetadata("favorites", this.state.metadata)
  }

  deleteFavorite(id) {
    this.setState({
      metadata: {
        favorites: this.state.favorites.filter(f => f.id !== id)
      }
    })
  }
```

```
  render() {
    const { metadata } = this.state

    return metadata
      ? (
        <ul className="favorites-list">
          {metadata.favorites.map(f => (
            <Favorite
              key={f.id}
              name={f.name}
              onDelete={this.deleteFavorite(f.id)}
            />
          ))}
        </ul>
      )
      : <Spinner />
  }
}
```

Logic sharing concept evolution 6

* Both components use calls to get/set metadata
* These calls are made in componentDidMount() and componentWillUnmount() lifecycle hooks

Logic sharing concept evolution 7

```
// Orders component
componentDidMount() {
  getMetadata("orders")
    .then(m => this.setState({ metadata: m }))
}

componentWillUnmount() {
  sendMetadata("orders", this.state.metadata)
}
```

```
// Favorites component
componentDidMount() {
  getFavorites()
    .then(m => this.setState({ metadata: m }))
}

componentWillUnmount() {
  sendMetadata("favorites", this.state.metadata)
}
```

Clearly, that sending/receiving metadata logic can be shared

Logic sharing concept evolution 8

Steps to create HOC

1. Decide what parameters may differ for different use cases of sharing the same logic (callbacks, action handlers, names, etc.) – those will be included in the parameters list of wrapping function
2. Put the common logic within `return class extends React.Component` statement
3. Internal state maintained by the components that are going to share the same logic should be now passed to them via props
4. Remember to always pass-through props that are not altered by wrapper component

Logic sharing concept evolution 9

```
function withMetadata(WrappedComponent, getData, setData) {
  return class extends React.Component {
    constructor(props) {
      super(props)
      this.state = { metadata: undefined }

      this.changeMetadata = this.changeMetadata.bind(this)
    }

    componentDidMount() {
      getData()
        .then(d => this.setState({ metadata: d }))
    }

    componentWillUnmount() {
      setData(this.state.metadata)
    }
```

```
    changeMetadata(newData) {
      this.setState({ metadata: newData })
    }

    render() {
      return this.state.metadata
        ? (
          <WrappedComponent {...this.props}
            metadata={this.state.metadata}
            onMetadataChange={this.changeMetadata}
          />
        )
        : <Spinner />
    }
  }
}
```

Full example: https://codesandbox.io/s/hoc-bslqb

Logic sharing concept evolution 10

HOC caveats

* What props are passed to the component via HOC and what props should still be supplied to the wrapped component?
* Static functions?
* Display name?
* Is component-granularity of composition enough?
* HOC hell?

Logic sharing concept evolution 11

Obviously, despite serving their purpose as the component sharing mechanism just fine, HOCs are not perfect. That's why React community came up with slightly better idea – render props pattern (also called function as a child)

Logic sharing concept evolution 12

Render props

* Passing function as a prop is nothing unusual
* In RPP* that function is used in a client component to render children
* Represents the shared piece of logic as a declarative model (in other words – very react-ic), the composer (we) decides how to consume it

*Render Props Pattern

Logic sharing concept evolution 13

```
class Stopwatch extends React.PureComponent {
  constructor(props) {
    super(props)
    this.state = { elapsedIntervals: 0 }
    this.intervalId = null

    this.stopCounting = this.stopCounting.bind(this)
    this.logElapsed = this.logElapsed.bind(this)
  }

  componentDidMount() {
    this.intervalId = setInterval(
      () => {
        const currentElapsed = this.state.elapsedIntervals
        this.setState({ elapsedIntervals: currentElapsed + 1 })
      }, 1000
    )
  }

  stopCounting() {
    if (this.intervalId) {
      clearInterval(this.intervalId)
      this.intervalId = null
    }
  }
```

```
  componentWillUnmount() {
    this.stopCounting()
  }

  logElapsed() {
    console.log(`Elapsed seconds: ${this.state.elapsedIntervals}`)
  }

  render() {
    return (
      <div className="stopwatch-container">
        <p>Seconds elapsed: {this.state.elapsedIntervals}</p>
        <button onClick={this.stopCounting}>Stop</button>
        <button onClick={this.logElapsed}>Log interval</button>
      </div>
    )
  }
}
```

Logic sharing concept evolution 14

```
class FancyTimer extends React.PureComponent {
  constructor(props) {
    super(props)
    this.state = { elapsedIntervals: 0 }
    this.intervalId = null

    this.stopCounting = this.stopCounting.bind(this)
    this.restartCounting = this.restartCounting.bind(this)
  }

  setTimer() {
    this.intervalId = setInterval(
      () => {
        const currentElapsed = this.state.elapsedIntervals
        this.setState({ elapsedIntervals: currentElapsed + 1 })
      }, 200
    )
  }

  componentDidMount() {
    this.setTimer()
  }
```

```
  stopCounting() {
    if (this.intervalId) {
      clearInterval(this.intervalId)
      this.intervalId = null
    }
  }

  componentWillUnmount() {
    this.stopCounting()
  }

  restartCounting() {
    this.stopCounting()
    this.setState({ elapsedIntervals: 0 })
    this.setTimer()
  }

  render() {
    return (
      <div className="stopwatch-container">
        <h3
          style={{ cursor: "pointer" }}
          onClick={this.restartCounting}
        >
          {this.state.elapsedIntervals}
        </h3>
        <p>(Click on time to restart)</p>
        <button onClick={this.stopCounting}>Stop</button>
      </div>
    )
  }
}
```

Logic sharing concept evolution 15

* Both components use intervals API to manage time ticks (with different parameter though)
* They allow clients to trigger „timer stop” action

Logic sharing concept evolution 16

```
// Stopwatch
componentDidMount() {
  this.intervalId = setInterval(
    () => {
      const currentElapsed = this.state.elapsedIntervals
      this.setState({ elapsedIntervals: currentElapsed + 1 })
    }, 1000
  )
}

stopCounting() {
  if (this.intervalId) {
    clearInterval(this.intervalId)
    this.intervalId = null
  }
}

componentWillUnmount() {
  this.stopCounting()
}
```

```
// FancyTimer
componentDidMount() {
  this.setTimer()
}

stopCounting() {
  if (this.intervalId) {
    clearInterval(this.intervalId)
    this.intervalId = null
  }
}

componentWillUnmount() {
  this.stopCounting()
}
```

Both components manage time in a similar manner

Logic sharing concept evolution 17

Following RPP

1. Decide what parameters may differ for different use cases of sharing the same logic (callbacks, action handlers, names, etc.) – those will be passed as props to component using render function as a prop
2. Put the common logic within the component accepting render prop
3. Internal state maintained by the components that are going to share the same logic will be now passed to the render prop function
4. Component rendering function as a child must expose the shared logic model as render prop function params

Logic sharing concept evolution 18

```
class TimeTicker extends React.Component {
  constructor(props) {
    super(props)
    this.state = { elapsedIntervals: 0 }
    this.intervalId = null

    this.stopCounting = this.stopCounting.bind(this)
    this.restartCounting = this.restartCounting.bind(this)
  }

  setTimer() {
    this.intervalId = setInterval(
      () => {
        const currentElapsed = this.state.elapsedIntervals
        this.setState({ elapsedIntervals: currentElapsed + 1 })
      }, this.props.intervalMs
    )
  }

  componentDidMount() {
    this.setTimer()
  }

  stopCounting() {
    if (this.intervalId) {
      clearInterval(this.intervalId)
      this.intervalId = null
    }
  }
```

```
  componentWillUnmount() {
    this.stopCounting()
  }

  restartCounting() {
    this.stopCounting()
    this.setState({ elapsedIntervals: 0 })
    this.setTimer()
  }

  render() {
    return this.props.render({
      elapsedIntervals: this.state.elapsedIntervals,
      stopCounting: this.stopCounting,
      restartCounting: this.restartCounting,
    })
  }
}
```

Full example: https://codesandbox.io/s/rpp-935zy

(or play with downshift library to fully unleash the RPP power)

Logic sharing concept evolution 19

Note: function prop does not have to be named „render" (it's just a convention).

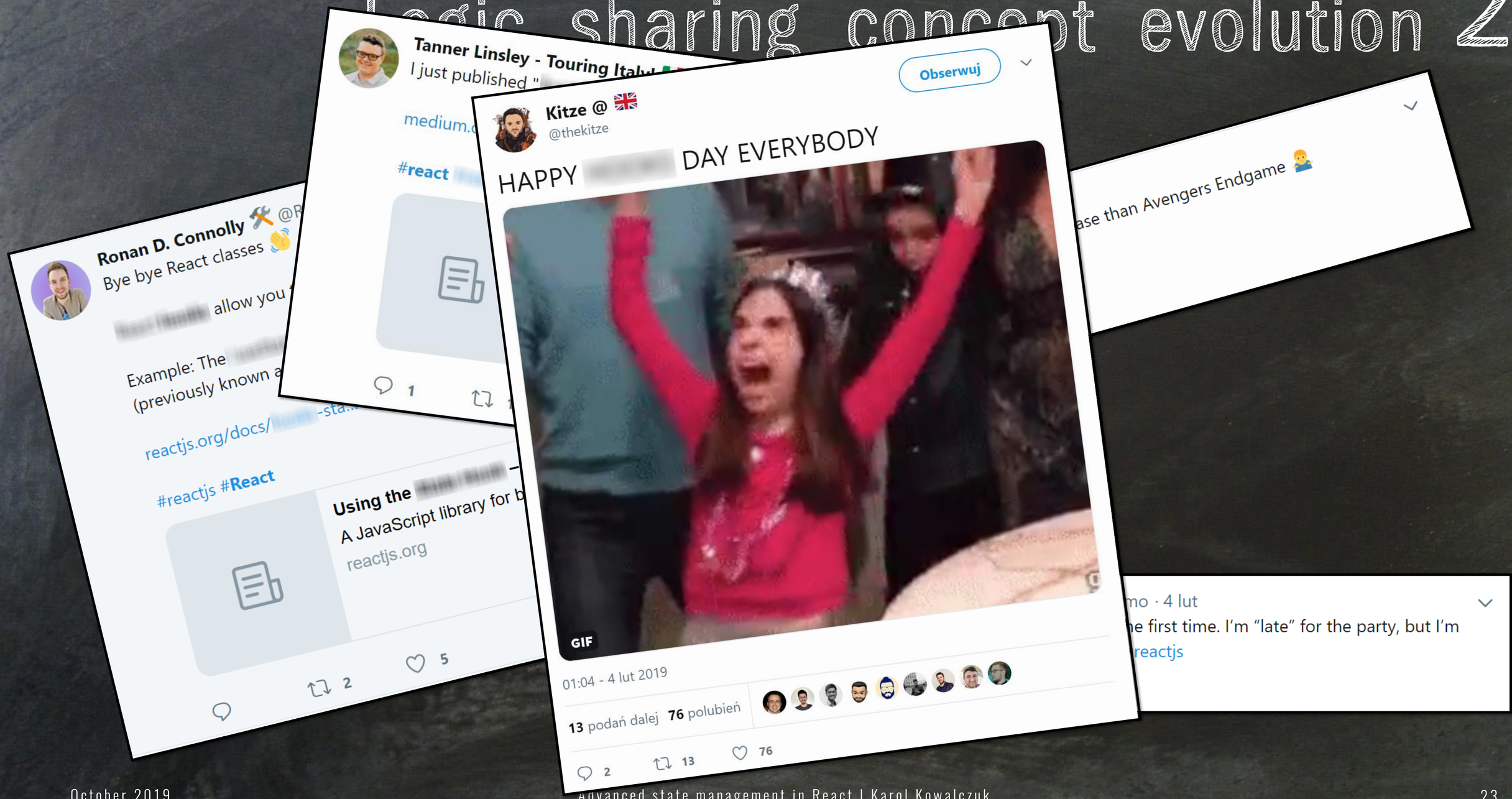
You are allowed to use name that correctly reflects its purpose.

Logic sharing concept evolution 20

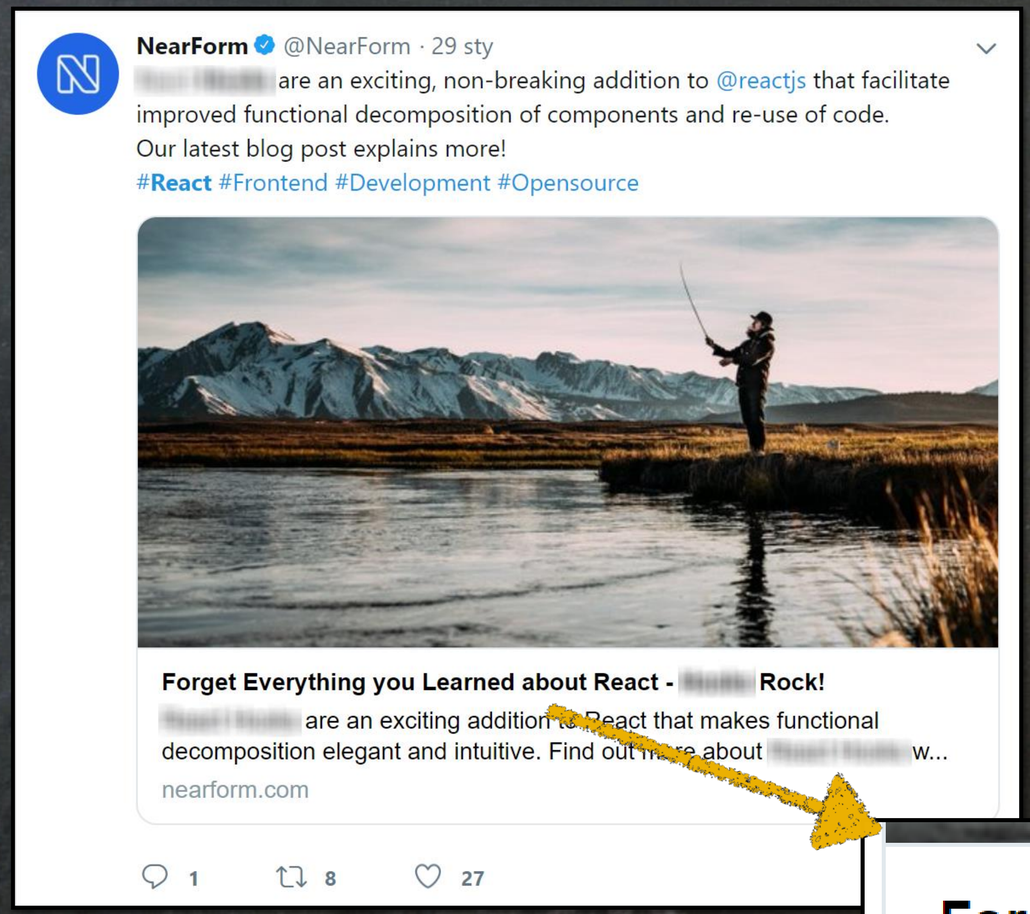
Issues with RPP

* Performance?
* ShouldComponentUpdate?
* Testing the higher level component?

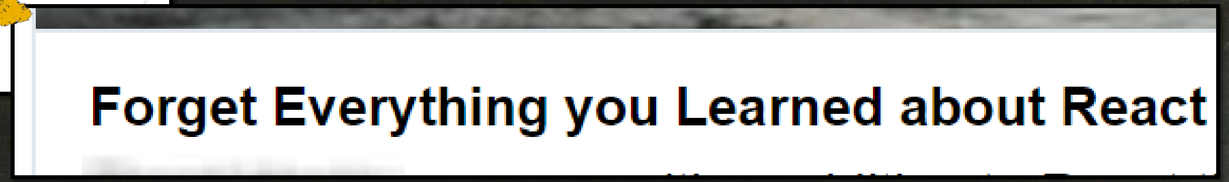
logic sharing concept evolution 21



Logic sharing concept evolution 21



???



React Hooks and Context API 1

* Starting from React 16.8 (when React hooks were released) function components are capable of maintaining their internal state...
* ...and lifecycle hooks as well
* React hooks release redefined the way of react-ic thinking

React Hooks and Context API 2

* Class components are not and will never become deprecated (~~backwards~~ old-fashioned compatibility)
* Certain rules need to be followed while dealing with hooks, but – they are worth it
* Behind the name „React Hook" stands just a pure function that cooperates with React internal APIs

React Hooks and Context API 3

* React provides a few built-in hooks (they belong to React namespace) and their names are prefixed with „use”
* We can combine the hooks together (by creating custom hooks) to create reusable logic handlers
* Custom hooks names ~~may~~ should be preceeded with „use” as well

React Hooks and Context API 4

Do not ever call hooks inside loops, conditions, nested functions and async callbacks!

The call order must be preserved.

React Hooks and Context API 5

useState

* Built-in hook dedicated for keeping & modifying parts of the isolated components state
* Returns an array containing the pair of state setter and getter
* Accepts initial value of the sub-state as an input parameter

React Hooks and Context API 6

useState

```jsx
const Counter = (props) => {
  const [counter, setCounter] = React.useState(0)

  return (
    <div className="counter-wrapper">
      <p>Counter value: {counter}</p>
      <button onClick={() => setCounter(counter + 1)}>+</button>
      <button onClick={() => setCounter(counter - 1)}>-</button>
    </div>
  )
}
```

=

```jsx
class CounterClass extends React.Component {
  constructor(props) {
    super(props)
    this.state = { counter: 0 }

    this.setCounter = this.setCounter.bind(this)
  }

  setCounter(value) {
    this.setState({ counter: value })
  }

  render() {
    const { counter } = this.state

    return (
      <div className="counter-wrapper" >
        <p>Counter value: {counter}</p>
        <button onClick={() => this.setCounter(counter + 1)}>+</button>
        <button onClick={() => this.setCounter(counter - 1)}>-</button>
      </div>
    )
  }
}
```

React Hooks and Context API 7

useState

Mount phase

```
const LoginForm = () => {
  const [username, setUsername] = React.useState("")
  const [password, setPassword] = React.useState("")
  const [showPassword, setShowPassword] = React.useState(false)

  return (
    // ...
  )
}
```

Internal component instance

displayName: „LoginForm”

state: []

// other internal component
// data

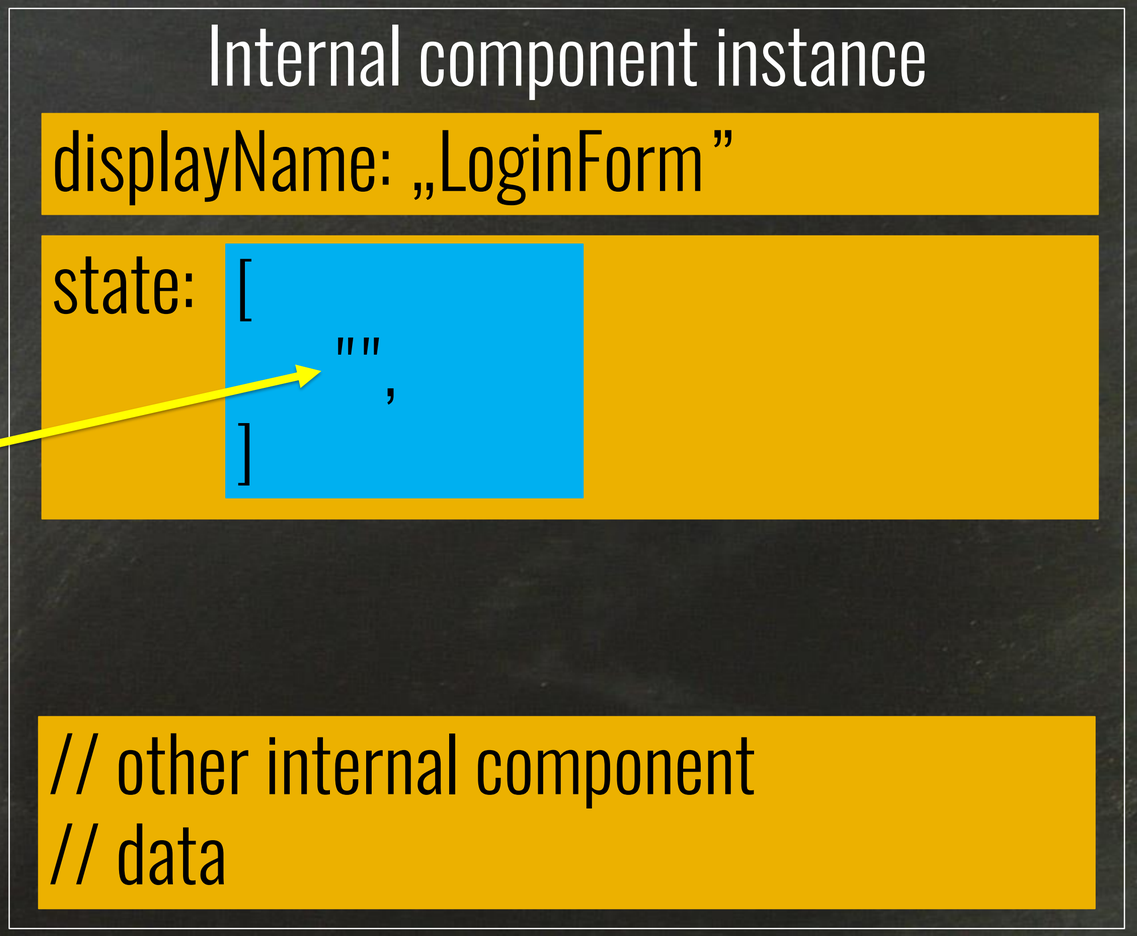
React Hooks and Context API 8

useState

Mount phase

```
const LoginForm = () => {
  const [username, setUsername] = React.useState("")
  const [password, setPassword] = React.useState("")
  const [showPassword, setShowPassword] = React.useState(false)

  return (
    // ...
  )
}
```

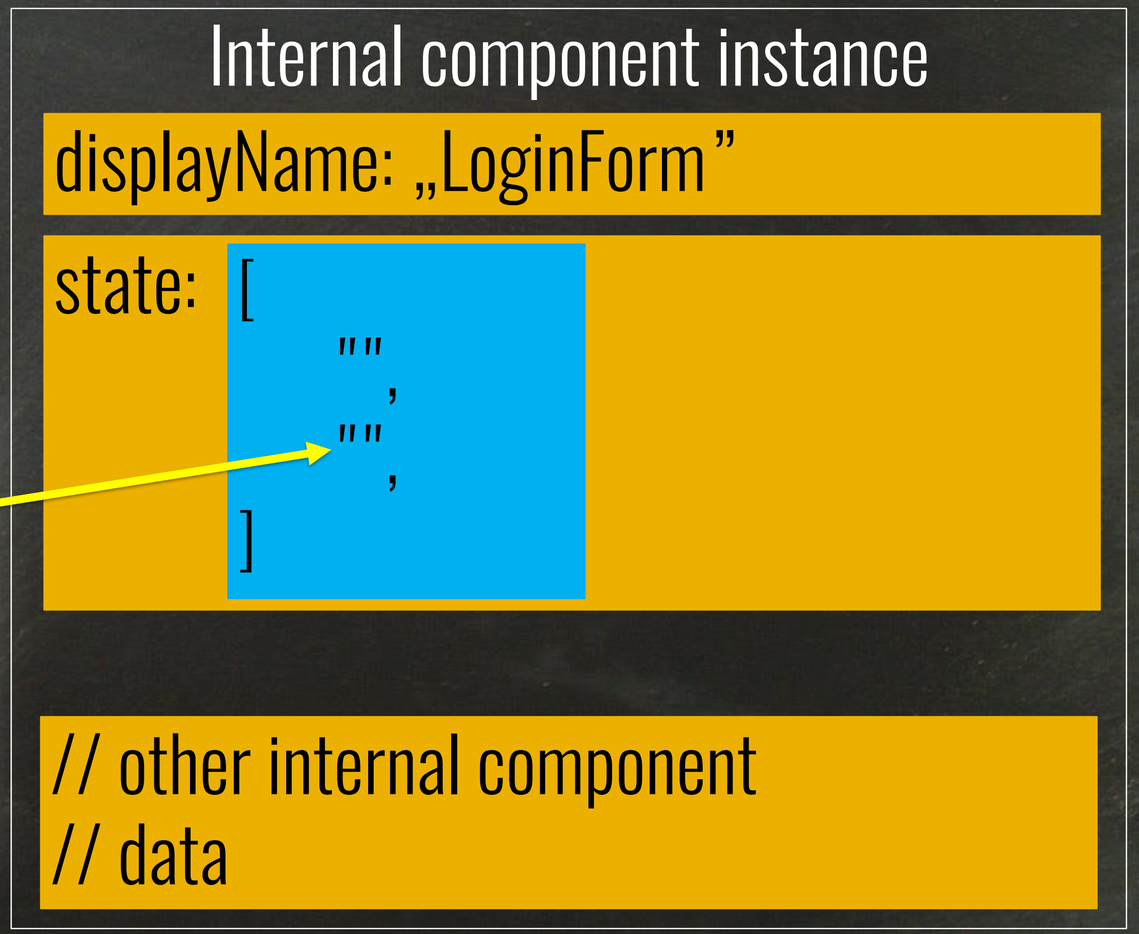


useState

Mount phase

```
const LoginForm = () => {
  const [username, setUsername] = React.useState("")
  const [password, setPassword] = React.useState("")
  const [showPassword, setShowPassword] = React.useState(false)

  return (
    // ...
  )
}
```

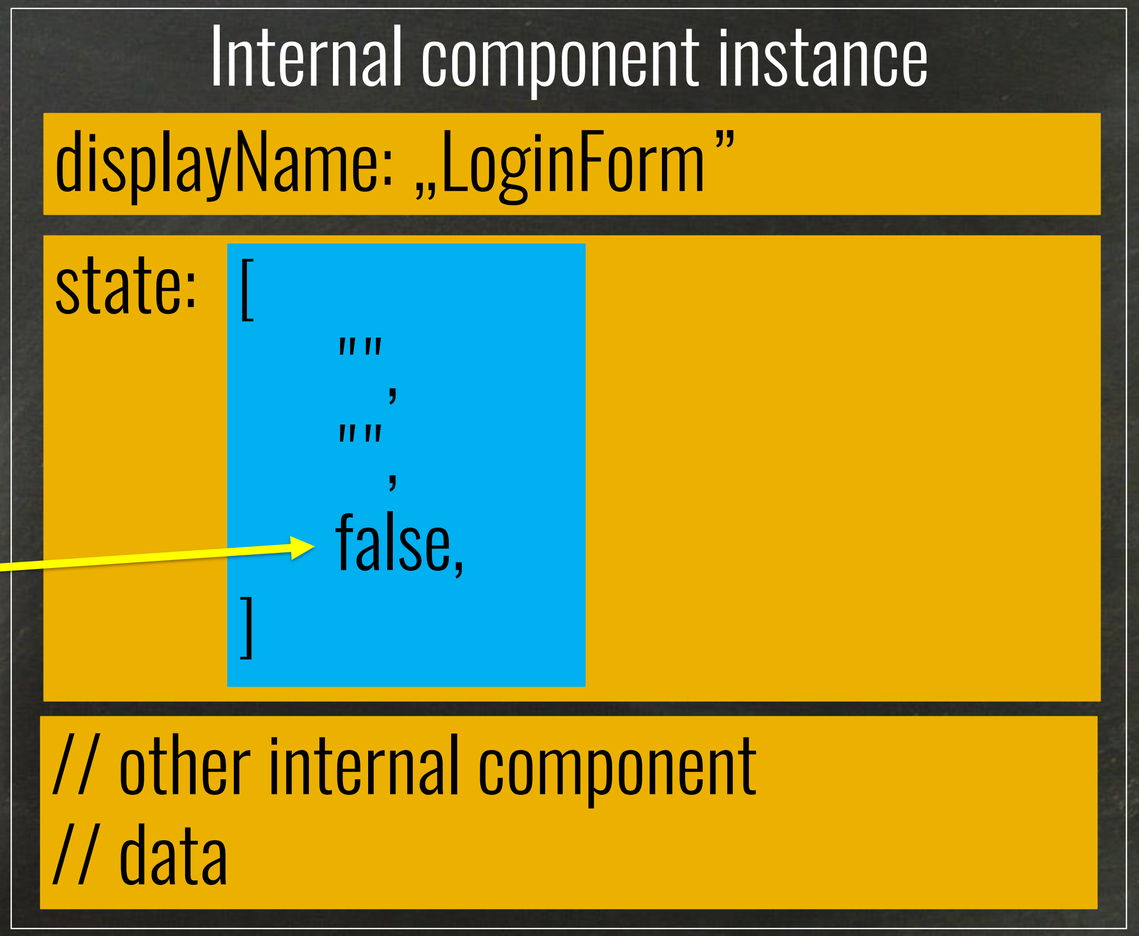


useState

Mount phase

```
const LoginForm = () => {
  const [username, setUsername] = React.useState("")
  const [password, setPassword] = React.useState("")
  const [showPassword, setShowPassword] = React.useState(false)

  return (
    // ...
  )
}
```



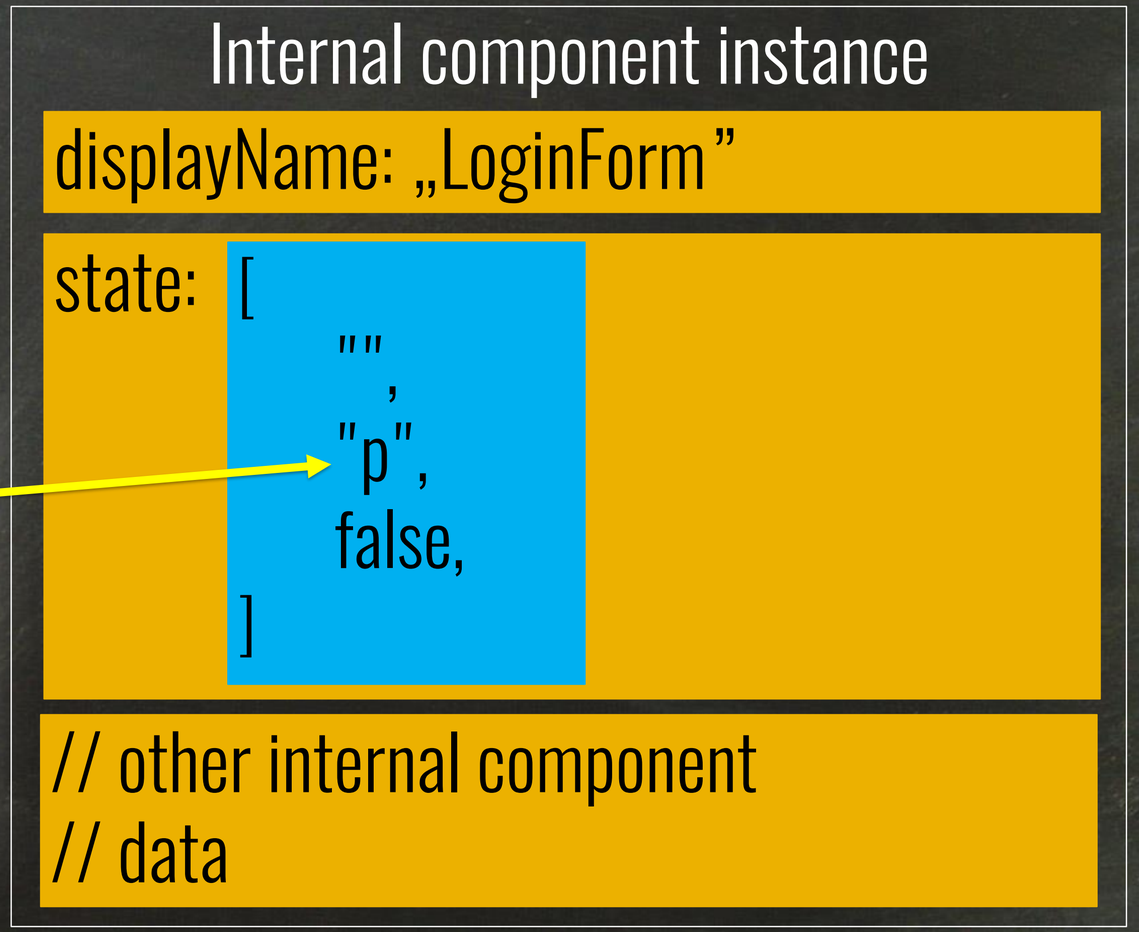
React Hooks and Context API 11

useState

Password change action

```
const LoginForm = () => {
  const [username, setUsername] = React.useState("")
  const [password, setPassword] = React.useState("")
  const [showPassword, setShowPassword] = React.useState(false)

  return (
    1 USER TYPES „P” IN PASSWORD INPUT
  )
}
```



useState

Update phase

```
const LoginForm = () => {
  const [username, setUsername] = React.useState("")
  const [password, setPassword] = React.useState("")
  const [showPassword, setShowPassword] = React.useState(false)

  return (
    // ...
  )
}
```

Internal component instance

displayName: „LoginForm”

state: [
"",
"p",
false,
]

// other internal component
// data

React Hooks and Context API 13

useState

Update phase

```
const LoginForm = () => {
  const [username, setUsername] = React.useState("")
  const [password, setPassword] = React.useState("")
  const [showPassword, setShowPassword] = React.useState(false)

  return (
    // ...
  )
}
```

Internal component instance

displayName: „LoginForm”

state: ["", "p", false,]

// other internal component
// data

useState

Update phase

```
const LoginForm = () => {
  const [username, setUsername] = React.useState("")
  const [password, setPassword] = React.useState("")
  const [showPassword, setShowPassword] = React.useState(false)

  return (
    // ...
  )
}
```

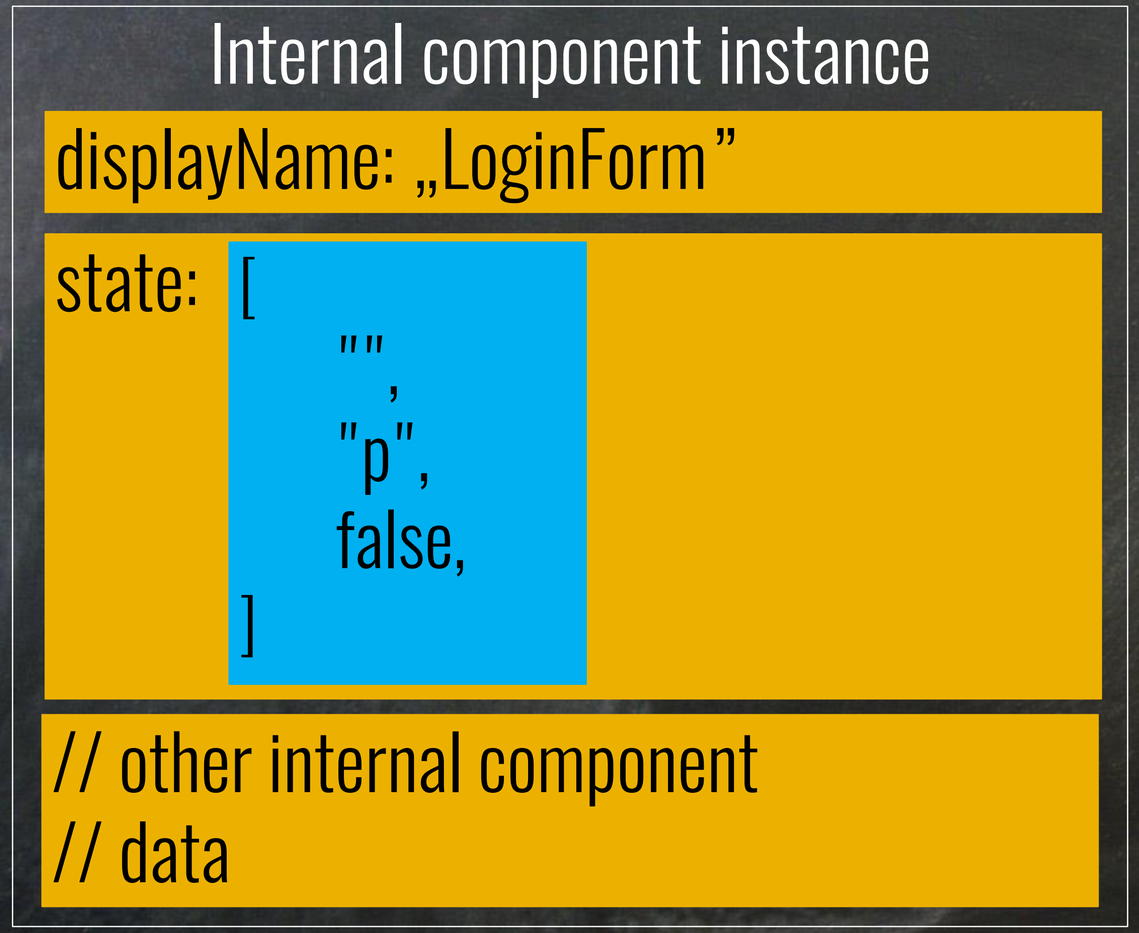
Internal component instance

displayName: „LoginForm”

state: ["", "p", false,]

// other internal component
// data

React Hooks and Context API 15



State for nodes created via function components is represented in React's internal structure as a list.

It does not know anything about the names.

Order matters.

React Hooks and Context API 16

useState

* Sub-state setter may accept a function as its call parameter
* In certain scenarios we want the state to be calculated based on the most recent counterpart
* Usage: setElapsed(elapsed => elapsed + 5)

React Hooks and Context API 17

useEffect

* Another built-in hook which interacts with React internal API in terms of handling side effects
* Function component's equivalent of lifecycle hooks (componentDidMount, componentDidUpdate, componentWillUnmount)

React Hooks and Context API 18

useEffect

* Accepts a callback function as an input parameter
* Dependencies list determining whether the effect should be run must be passed as a second argument
* cleanup code that should be run when component re-renders/unmounts may be defined as return value from the callback

React Hooks and Context API 19

useEffect

```
const TimeCounter = () => {
  const [elapsedSeconds, setElapsedSeconds] = React.useState(0)

  React.useEffect(() => {
    const timerId = setInterval(() => {
      setElapsedSeconds(elapsedSeconds => elapsedSeconds + 1)
    }, 1000)

    return () => {
      clearInterval(timerId)
    }
  }, [])

  React.useEffect(() => {
    if (elapsedSeconds && (elapsedSeconds
      console.log(`User spent ${elapsedSeco
    }
  }, [elapsedSeconds])

  return (
    <div className="time-spent-container">
      <p>You spent {elapsedSeconds} seconds on this page.</p>
    </div>
  )
}
```

See it in action:

https://codesandbox.io/s/useeffect-bv30w

```
class TimeCounterClass extends React.Component {
  constructor(props) {
    super(props)
    this.intervalId = null
    this.state = { elapsedSeconds: 0 }
  }

  componentDidMount() {
    this.intervalId = setInterval(() => {
      this.setElapsedSeco (this.state.elapsedSeconds + 1)
    }, 1000)
  }

                                    his.state

                                  Seconds % 5 === 0)) {
                                  psedSeconds}s on this page`)

                                  Unmount() {
                                  arInterval(this.intervalId)
  }

  setElapsedSeconds(value) {
    this.setState({ elapsedSeconds: value })
  }

  render() {
    const { elapsedSeconds } = this.state

    return (
      <div className="time-spent-container">
        <p>You spent {elapsedSeconds} seconds on this page.</p>
      </div>
    )
  }
}
```

React Hooks and Context API 20

useRef

* As name indicates, useRef() can be used for holding references to DOM elements in the same manner as component class property initialized with React.createRef()
* But not only! It can also persist references to objects „owned” by instances of function-based components through multiple updates

React Hooks and Context API 21

useRef

* The referenced object can be then accessed via .current property of variable initialized with useRef() hook
* Changes to mutable objects referenced that way will NOT cause component rerenders

React Hooks and Context API 22

useRef

```
const EnhancedForm = () => {
  const inputRef = React.useRef()

  return (
    <div className="enhanced-form">
      <input placeholder="Username" type="text" ref
      <button onClick={() => inputRef.current
        Focus input
      </button>
    </div>
  )
}
```

```
class EnhancedFormClass extends React.PureComponent {
  constructor(props) {
    super(props)

    this.inputRef = React
  }

  type="text" ref={this.inputRef} />
  => this.inputRef.current.focus()}>
  </div>
  )
  }
}
```

Slightly more complex example:
https://codesandbox.io/s/useref-sm47i

useMemo

* When React rerenders function-based component, the whole function body is re-executed
* In some cases (when we're doing an expensive computation), we may accidentally introduce performance penalty

useMemo

* With useMemo() we can hint React whether the value has to be re-computed during the update
* It accepts the function that must return a computed value
* In second argument we should declare what values that may have been changed over time could cause the need of recomputation

React Hooks and Context API 25

useMemo

```
const ItemsForm = (props) => {
  const [selectedId, setSelectedId] = React.useState(undefined)
  const [query, onQueryChange] = React.useState("")

  const filteredItems = React.useMemo(() => {
    return props.itemsList
      .filter(i => i.name.toLowerCase().includes(query.toLowerCase()))
  }, [props.itemsList, query])

  return (
    <form className="selectable-group">
      <span>Filter items</span>
      <input value={query} onChange={(e) =>
      {filteredItems.map(fi => {
        return (
          // ...
        )
      })}
    </form>
  )
}
```

See it in action:

https://codesandbox.io/s/useMemo-0jm1j

```
class ItemsFormClass extends React.PureComponent {
  constructor(props) {
    super(props)
    this.state = { selectedId: undefined, query: "", filteredItems: [] }

    this.setSelectedId = this.setSelectedId.bind(this)
    this.onQueryChange = this.onQueryChange.bind(this)
  }

  static getDerivedStateFromProps(props, state) {
    return {
      ...state,
      filte                                    t
                                               ase().includes(this.state.query.toLowerCase()))
                                               ery })

  render() {
    return (
      <form className="selectable-group">
        <span>Filter items</span>
        <input
          value={this.state.query}
          onChange={(e) => this.onQueryChange(e.target.value)}
        />
        {this.state.filteredItems.map(fi => {
          return (
            // ...
          )
        })}
      </form>
    )
  }
}
```

useCallback

* Behaves exactly the same as useMemo(), but returns a memoized function instead of primitive/object
* Useful for storing functions within function-based components (especially for custom change handlers)
* Simulates behavior of class-based components' methods

React Hooks and Context API 27

useCallback

```
const Uppercaser = () => {
  const [uppercasedValue, setUppercasedValue] = React.useState("")

  const onValueChange = React.useCallback((newValue) => {
    setUppercasedValue(newValue.toUpperCase())
  }, [])

  return (
    <input
      value={uppercasedValue}
      onChange={(e) => onValueChang
    />
  )
}
```

See it in action:
https://codesandbox.io/s/useCallback-7x5kf

```
class UppercaserClass extends React.Component {
  constructor(props) {
    super(props)
    this.state = { uppercasedValue: "" }

    this.onVal                    ValueChange.bind(this)
  }

                                : newValue.toUpperCase() })

        <input
          value={this.state.uppercasedValue}
          onChange={(e) => this.onValueChange(e.target.value)}
        />
      )
    }
}
```

useLayoutEffect

* Yet another built-in hook dedicated for very specific scenarios
* Signature is the same as for useEffect()
* In contrary to useEffect() applies effect before the DOM is painted on screen (application may be unresponsive for a while)
* Most likely you will not need it

useImperativeHandle

* Remember refs?
* useImperativeHandle() can enrich the object retrieved by ref prop with custom property/handler
* Most likely you will not need it (if you do, there is 99.9%* chance that you're doing something wrong)

*No research on that yet, I'm working on it

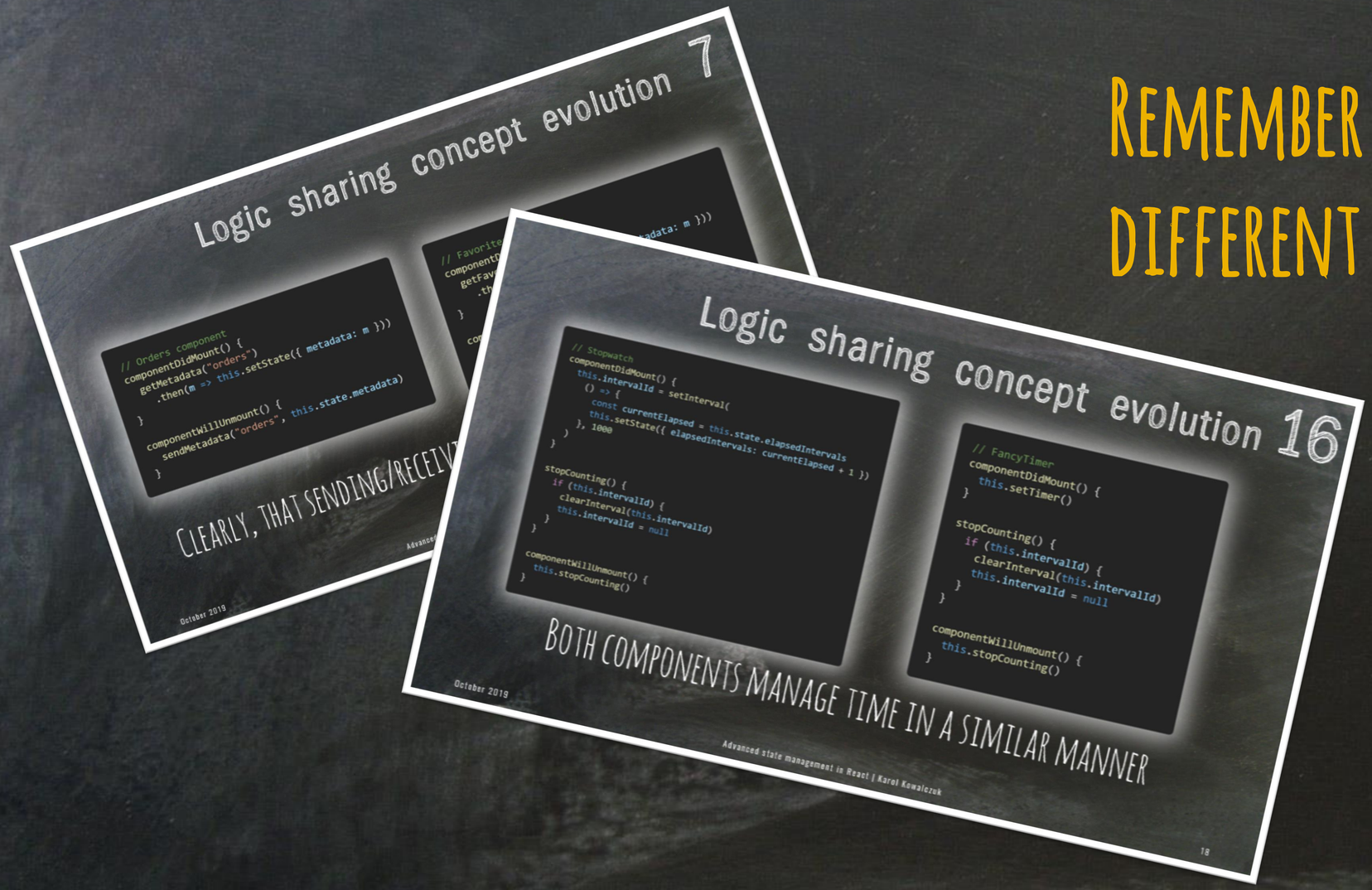
useDebugValue

* From React docs: „show a label in DevTools next to this Hook”...
* ...rate yourself how useful is that
* Most likely tou will not need it

React Hooks and Context API 31

Static getDerivedStateFromProps() and error handling mechanism are not covered by hooks (yet).

React Hooks and Context API 32



Remember those two? Let's try different approach now.

Hooks approach.

React Hooks and Context API 33

```
export function useMetadata(getData, setData) {
  const [metadata, setMetadata] = React.useState(undefined)
  const dataToSend = React.useRef()

  React.useEffect(() => {
    getData().then(d => setMetadata(d))

    return () => {
      if (dataToSend.current) {
        setData(dataToSend.current)
      }
    }
  }, [])

  React.useEffect(() => {
    dataToSend.current = metadata
  }, [metadata])

  return {
    metadata,
    changeMetadata: setMetadata
  }
}
```

Full source:

https://codesandbox.io/s/hoc2hooks-h6sys

React Hooks and Context API 34

```
export function useCounter(intervalMs) {
  const intervalId = React.useRef(null)
  const [elapsedIntervals, setElapsedIntervals] = React.useState(0)

  const setTimer = React.useCallback(() => {
    intervalId.current = setInterval(() => {
      setElapsedIntervals(elapsed => elapsed + 1)
    }, intervalMs)
  }, [intervalMs])

  const stopCounting = React.useCallback(() => {
    if (intervalId.current) {
      clearInterval(intervalId.current)
      intervalId.current = null
    }
  }, [])
```

```
  const restartCounting = React.useCallback(() => {
    stopCounting()
    setElapsedIntervals(0)
    setTimer()
  }, [stopCounting, setTimer])

  React.useEffect(() => {
    setTimer()

    return stopCounting
  }, [])

  return {
    elapsedIntervals,
    stopCounting,
    restartCounting
  };
}
```

Full source: https://codesandbox.io/s/rpp2hooks-bf0m6

Sharing logic with hooks

* Very easy to compose
* Less code = same results
* Testing in isolation can be fun (really!)
* Simple and more readable than alternatives

Hands-on

Hooks, hooks and more hooks

source available: B-01-react-hooks

Context

* Special object that can be retrieved at any level below its provider in the component tree without passing props down explicitely
* Can be accessed using either Context Consumer component or useContext() hook

Context

* Whenever changes, ALL CONSUMERS (no matter what part of context they're consuming) will be RE-RENDERED
* Might serve the purposes of common state management libraries, so eliminates them from dependencies list
* Any component down the tree may utilize multiple contexts

Creating context

Define what is going to be provided

```
const defaultContext = {
  isLoggedIn: false,
  favorites: [],
  cart: [],
}

const MovieDatabaseContext = React.createContext(
  defaultContext
)
```

React Hooks and Context API 40

```
const MovieDatabaseProvider = (props) => {
  const [isLoggedIn, setLoggedIn] = React.useState(defaultContext.isLoggedIn)
  const [favorites, setFavorites] = React.useState(defaultContext.favorites)
  const [cart, setCart] = React.useState(defaultContext.cart)

  const addToFavorites = React.useCallback((movie) => {
    setFavorites(favorites => [...favorites, movie])
  }, [])

  return (
    <MovieDatabaseContext.Provider
      value={{
        isLoggedIn, // shortcut for "isLoggedIn: isLoggedIn"
        favorites,
        cart,
        addToFavorites,
      }}
    >
      {props.children}
    </MovieDatabaseContext.Provider >
  )
}
```

Providing context

Embedding component that supplies data

Alternatively use class syntax

Consuming context

Notify that some data comes from context above

function-based component version 1 (hooks)

```
const FavoritesList = (props) => {
  const { favorites } = React.useContext(MovieDatabaseContext)
  // ...

  return (
    <ul className="favs">
      {favorites.map(f => {
        <li key={f.id}>{f.name}</li>
      })}
    </ul>
  )
}
```

Consuming context

Notify that some data comes from context above

function-based component version 2 (.consumer API)

```
const FavoritesList = (props) => {
  return (
    <ul className="favs">
      <MovieDatabaseContext.Consumer>
        {(value) => {
          const { favorites } = value

          return (
            favorites.map(f => {
              <li key={f.id}>{f.name}</li>
            })
          )
        }}
      </MovieDatabaseContext.Consumer>
    </ul>
  )
}
```

Consuming context

Notify that some data comes from context above

class-based component version

```
class FavoritesList extends React.Component {
  static contextType = MovieDatabaseContext

  render() {
    const { favorites } = this.context // React-specific property

    return (
      <ul className="favs">
        {favorites.map(f => {
          <li key={f.id}>{f.name}</li>
        })}
      </ul>
    )
  }
}
```

React Hooks and Context API 44

Hands-on

Use context to store application state

sources available: B-02-context-api, B-03-whole-app-state-in-context

Context API

When

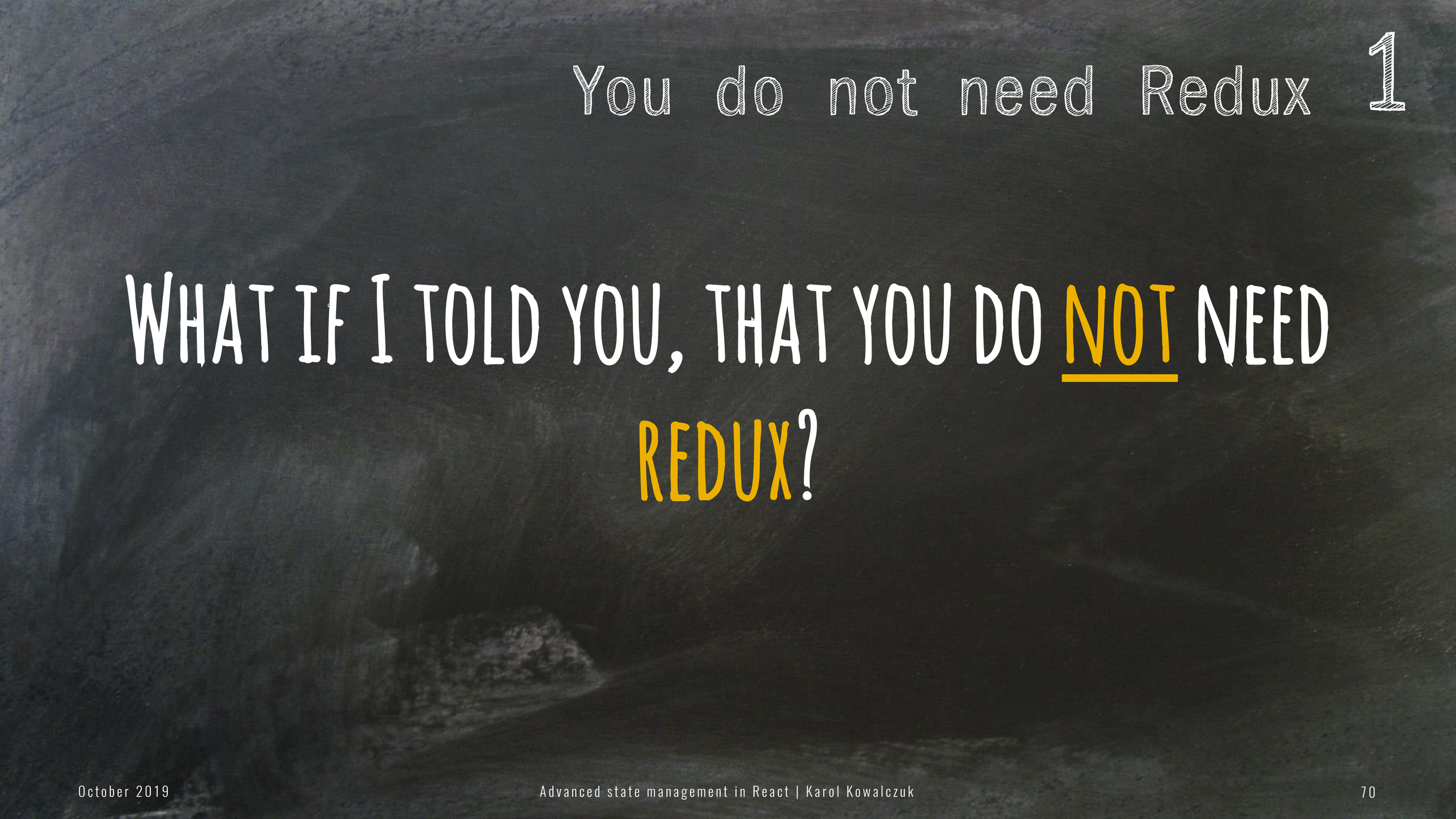
* Global state (themes, internationalization, user info)
* Per-feature logic connected to external APIs
* Need of serialization

When not

* Providing really tiny piece of data
* Component-specific state (single input, two checkboxes, etc.)
* Being fed up with passing the same props down the tree*

*It's really not enough to justify Context use

You do not need Redux 1



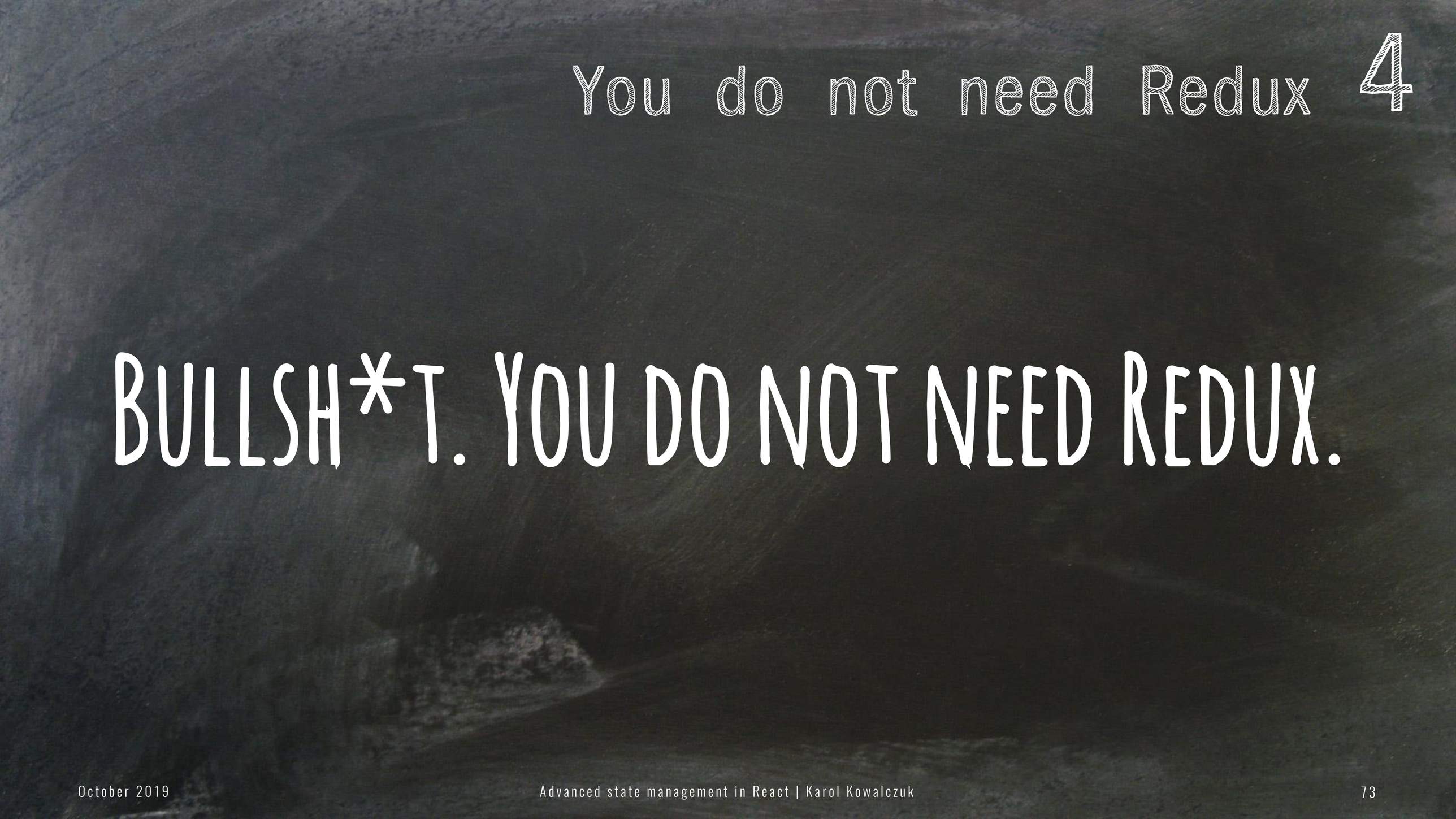
You do not need Redux 2

Context API is there for you.

In fact, Redux is built on top of Context API.


You do not need Redux 3
But rererererereducers!!!
Community can't be wrong!!!
I always used it!!!
Flux 4 life!!!

You do not need Redux 4



You do not need Redux 5

Redux action creator

A state update intention

```
export const addToFavorites = (videoId) => {
  return {
    type: "VIDEO_ADD_TO_FAVORITES",
    videoId,
  }
}
```

You do not need Redux 6

```
export default videoReducer = (state = initialState, action) => {
  switch (action.type) {
    case "VIDEO_ADD_TO_FAVORITES":
      return {
        ...state,
        favorites: [...state.favorites, action.videoId],
      }
  }
}
```

Redux reducer

Converts previous state and executed action into new state

You do not need Redux 7

Redux store

Always keeps the most recent state

```
import { combineReducers, createStore } from "redux"

import videoReducer from "services/video/videoReducer"

const reducers = combineReducers(
  videoReducer,
)

const store = createStore(reducers)
export default store
```

You do not need Redux 8

```
import React from "react"
import ReactDOM from "react-dom"
import { Provider } from "react-redux"

import store from "services/store"

ReactDOM.render(
  // Looks familiar?
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById("app")
)
```

Attaching the store

Making store available in the entire component tree

Consuming the store

Three magic steps needed to attach redux to any component

```
const mapStateToProps = (store) => {
  return {
    favorites: store.videosState.favorites,
  }
}

const mapDispatchToProps = (dispatch) => {
  return bindActionCreators({
    addToFavorites: VideoActions.addToFavorites,
  }, dispatch)
}

export default connect(
  mapStateToProps,
  mapDispatchToProps,
)(VideosContainer)
```

You do not need Redux 10

Choose wisely

Context API + hooks

* React-way of doing things
* Easy to test
* No additional dependencies
* Async actions handled with no hassle

Redux

* Enforces following FLUX principles
* Middleware
* A lot of boilerplate and magic HOCs
* Magic strings for action names

You do not need Redux 11

Choose wisely

You do not need Redux 12

Still hungry for reducer?

You do not need Redux 13

useReducer

* Built-in!
* When next state strongly depends on previous state
* Returns [reference to the most recent state, dispatch function]
* input params: reducer, initial state (, init function)

You do not need Redux 14

useReducer

```
const initialState = {
  country: "PL",
  input: "0",
  cost: 0,
}

const calculatorReducer = (state, action) => {
  switch (action.type) {
    case "INPUT": {
      return {
        ...state,
        input: action.stringValue,
        cost: state.country === "PL" ? (+acti
      }
    }
    case "COUNTRY_CHANGE": {
      return {
        ...state,
        country: action.country,
        cost: action.country === "PL" ? (+state.in
      }
    }
    default:
      throw new Error(`Action handler does not exist for ${action.type}`)
  }
}
```

See it in action:

https://codesandbox.io/s/usereducer-76gus

```
const Calculator = () => {
  const [calculatorState, dispatch] = React.useReducer(
    calculatorReducer, initialState)

  return (
    <div className="calculator-form">
      <span>Months</span>
      <input
        type="number"
        value={calcu
        onCha
                                              stringValue: e.target.value })}
                                              === "PL"}
                                              type: "COUNTRY_CHANGE", country: e.target.id })}
                                              >Poland</label>
        id="EE"
        type="radio"
        checked={calculatorState.country === "EE"}
        onChange={e => dispatch({ type: "COUNTRY_CHANGE", country: e.target.id })}
      />
      <label for="EE">Estonia</label>
      <h2><strong>{calculatorState.cost} PLN</strong></h2>
    </div>
  )
}
```

Library choice

@testing-library/react

- Enforces good practices
- Limited API contains only everything that's needed
- Lightweight

enzyme

- Does not prevent from testing implementation details
- Required additional setup for test framework

Hands-on

Add some unit tests

sources available: B-04-tests

More fancy features of modern React 1

React.lazy

Lazy load components only when needed

```
const Content = () => {
  const { activeTabId } = React.useContext(AppContext)

  const RecipesContainer = React.lazy(
    () => import("../recipes/RecipesContainer"))
  const ShoppingListContainer = React.lazy(
    () => import("../shoppingList/ShoppingListContainer"))

  return (
    <div className="container content">
      <React.Suspense
        fallback={<h5>Loading data...</h5>}
      >
        {activeTabId === "Recipes"
          ? (
            <RecipesContextProvider>
              <RecipesContainer />
            </RecipesContextProvider>
          )
          : <ShoppingListContainer />
        }
      </React.Suspense>
    </div>
  )
}
```

More fancy features of modern React 2

React.Suspense will allow the components „wait" for some code to load and declaratively specify a loading state (fallback UI)

Not yet available in stable release.

More: https://reactjs.org/docs/concurrent-mode-suspense.html

More fancy features of modern React 3

Concurrent mode

Not yet available in stable release.

More: https://reactjs.org/docs/concurrent-mode-intro.html

Summary 1

* HOCs and render props ~~are~~ were cool
* After hooks release, nothing is the same anymore
* useEffect() is not 1:1 equivalent of lifecycle hooks, but it can act as their implementation for function-based components
* Memoization should be a habit
* You really do not need redux in 2019+
* Good things are coming, so... stay tuned!