React
Best Practices

2 – Komponenten strukturieren

Timo Mämecke
TH Köln // MI Master // Weaving the Web
25. Juni 2019

Inhalt

Grundlegendes zu Komponenten.

Typische Fehler vermeiden.

Patterns zum Strukturieren.

Grundlegendes zu Komponenten

Neue Komponenten schreiben ist euer täglich Brot.

Nicht blocken lassen

Neue Komponenten anlegen soll kein Aufwand sein.

Komponenten anlegen, refactoren, löschen, ist ganz normal.

Werdet schnell darin, neue Komponenten anzulegen.

Beispiel: Code Snippet anlegen

Jede Komponente entsteht bei mir so:

```
import React from 'react'
import PropTypes from 'prop-types'

function SomeComponent(props) {
  return <div />
}

SomeComponent.propTypes = {}

export default SomeComponent
```

```
import React from 'react'
import PropTypes from 'prop-types'

function SomeComponent(props) {
  return <div />
}

SomeComponent.propTypes = {}

export default SomeComponent
```

Video: vom Snippet zur Component

Functional Components vs Class Components

```jsx
function Greeting({ name }) {
  return <div>Hello, {name}!</div>
}
```

```jsx
class Greeting extends React.Component {
  render() {
    return <div>Hello, {this.props.name}!</div>
  }
}
```

Vor React 16.8:
Keinen State, keinen Lifecycle.

Haben State & Lifecycle.

Functional Components vs Class Components

```
function Greeting() {
  const [name, setName] = useState('...')
  useEffect(() => {
    fetch('name-api.tld')
      .then(res => res.json())
      .then(json => setName(json.name)
  })

  return <div>Hello, {name}!</div>
}
```

```
class Greeting extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      name: '...'
    }
  }

  componentDidMount() {
    fetch('name-api.tld')
      .then(res => res.json())
      .then(json => this.setState({ name: json.name })
  }

  render() {
    return <div>Hello, {this.state.name}!</div>
  }
}
```

Seit React 16.8:
Hooks erlauben State &
Lifecycle.

Functional Components vs Class Components

Seit Hooks haben Class Components keine Vorzüge mehr.

Class Components sind nicht tot und müssen nicht umgebaut werden.

Functional Components mit Hooks sind schlanker und verständlicher.

Typische Fehler

Unübersichtlicher Code

```
render() {
  let sets = []
  for (let step = 0; step < this.state.setCount; step++) {
    sets.push(<Score key={step} index={step} score={this.setScore.bind(this)}/>)
  }
  return (
    <div>
      {this.props.league && <HeaderWrapper><PlayerSelectAndShow preSelect={this.state.team1.concat(this.state.team2)} break={
      </HeaderWrapper> }
      {this.state.completed && <div>
        <h4 className='aHeadline asSmall'>Result</h4>
        { sets }
        <AddSet className='headlineFont' onClick={()=>this.setState({setCount: this.state.setCount + 1})}>+</AddSet>
      </div>}
      { this.state.completed && this.state.sets.size > 0 &&  <Button onClick={()=> this.saveMatch()}>
        Save Match
      </Button> }
    </div>
  )
 }
}
```

Inkonsistente Einrückung und Whitespaces, kein Zeichen-Limit pro Zeile, ...

Zu große Components

Props falsch angewendet

Keine PropTypes. 🙁

Große Objects in einer Prop. 🙁

Typische Fehler vermeiden

Konsistenter Code-Style:
- Single Quotes vs Double Quotes
- Einrückung
- *mehr dazu in "5 – Code Quality & Testing"*

Übersichtliche Komponenten:
- Eine Component hat eine bestenfalls nur eine Aufgabe
- Große Komponenten in mehrere kleine aufbrechen
- Stateful Components vs. UI Components
- PropTypes auch als Dokumentation nutzen

Wie ich Features baue

1. Großes Feature in kleine Teile aufbrechen.
2. Feature-Teil in einer großen Komponente runterschreiben.
3. Alle Concerns erkennen, um die sich diese Komponente kümmert.
   (State, Lifecycle, asynchrone Daten laden, UI darstellen, ...)
4. Komponente in ihre Concerns refactoren.

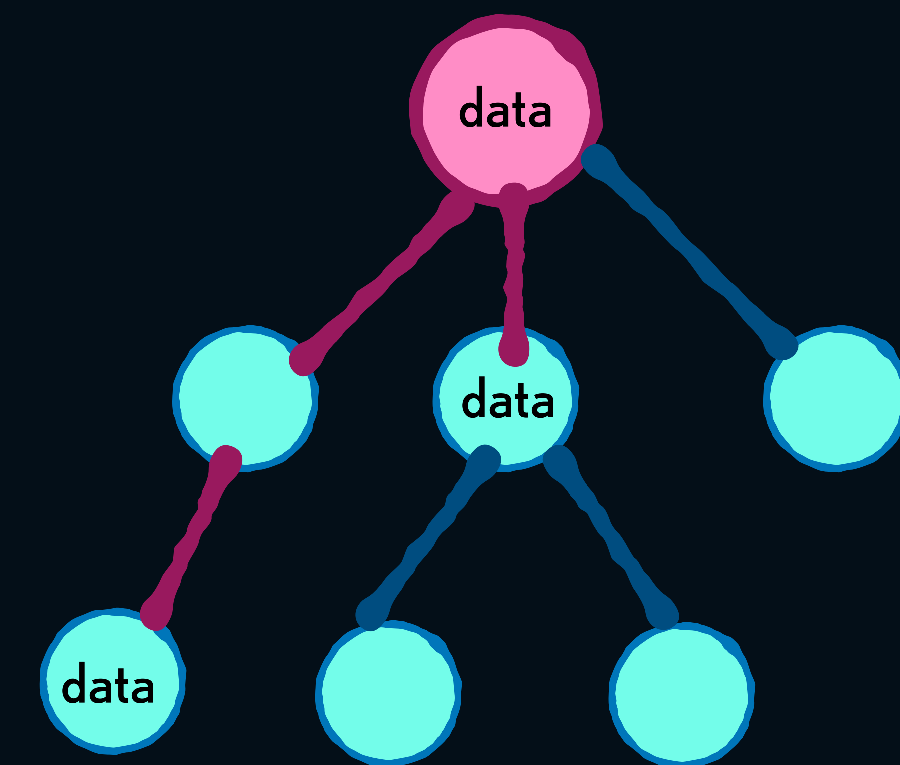
Patterns

Lifting the State

Problem:
mehrere Components basieren auf gleichem State.

Falsch:
State in mehreren Components synchron halten.

Richtig:
State in gemeinsame Eltern-Komponente verschieben.
State via Props von oben nach unten durchreichen.



* Siehe: https://reactjs.org/docs/lifting-state-up.html

Render Props

Problem:

- Components werden immer tiefer verschachtelt.
- Props werden immer weiter durchgereicht. "Prop Drilling"

Lösung: Render Props[1]

Komponenten nicht direkt voneinander abhängig machen, sondern dynamisch rendern.

```
<Dashboard renderTitle={user => (
  <Headline>Hello {user.name}!</Headline>
)} />
```

[1] https://reactjs.org/docs/render-props.html

Context

Problem:

- Props werden immer weiter durchgereicht. "Prop Drilling"
- Gleicher State wird an sehr vielen unterschiedlichen Stellen benötigt

Lösung: Context[1]

Gemeinsame Daten in Context schieben

```
function Dashboard() {
  const user = useContext(UserContext)
  return <Headline>Hello {user.name}!</Headline>
}
```

[1] https://reactjs.org/docs/context.html#when-to-use-context

Regeln für Props

Props "flach halten".
- Große Objects in einer Prop vermeiden.
- Lieber mehrere Props mit kleinen oder gar keinen Objects.

Immer PropTypes nutzen.

PropTypes mit Objects[1]:
- Alle Werte von Objekten mit `shape()` definieren, die verwendet werden.
- Nur Werte von Objekten definieren, die auch verwendet werden.

[1] Gleiches gilt für Arrays.

Thinking in React

Vor dem Coden analysieren, wie die Komponenten aufgebaut sind. Dabei direkt Namen definieren.

* Siehe: https://reactjs.org/docs/thinking-in-react.html

tl;dr

Neue Komponenten erstellen sollte schnell von der Hand gehen.

Functional Components und Hooks nutzen.

PropTypes nutzen.

Komponenten nicht zu groß werden lassen. Ständig refactoren und Patterns für Composition anwenden.