Funktions- und Klassenkomponenten

Web-Engineering II

Prof. Dr. Sebastian von Klinski

Funktions- und Klassenkomponenten

- Schon mit ersten Release von React gab es Funktions- und Klassenkomponenten
- Funktionskomponente
  - JavaScript-Funktion
  - Ergebnis wird mit „return“ zurückgegeben
  - Vor dem „return“ ist Komponentenlogik

```
import StartPage from "./react/views/StartPage"

function App() {
    return (
      <div>
        <StartPage />
      </div>
    )
}

export default App;
```

Varianten von Funktionskomponenten

- Funktionskomponenten können auf 3 Arten geschrieben werden

```
function Function1(props) {
    return (
        <Function2 />
    )
}

const Function2 = () => {
    const greeting = 'Hello Function Component!';
    return <Headline value={greeting} />;
};

const Headline = ({ value }) =>
    <h1>{value}</h1>;

export default Function1
```

JavaScript-Funktion

Arrow-Funktion

Verkürzte Arrow-Funktion

Klassenkomponente

- JavaScript-Klasse (extends „React.Component“)
- Erbt zahlreiche Aspekte von Component (z.B. setState())
- Hat Instanz-Variablen props und state
- render()-Funktion muss überschrieben werden
- Ergebnis wird in „render“-Funktion mit „return“ zurückgegeben
- Klassenkomponenten kann Konstruktur, Funktionen und in Render-Funktion vor dem „return“ Logik beinhalten

```
import React from "react";

import StartPage from "./react/views/StartPage"

class AppKlasse extends React.Component {
    render() {
        return <StartPage />;
    }
}
export default AppKlasse
```

Unterschiede: Syntax

- Bis 2019 konnten komplexe Anwendungen nur mit Klassenkomponenten umgesetzt werden
- Nur in Klassenkomponenten…
  - State-Management (Verwalten eines Zustands)
    - Funktion setState()
  - Life-Cycle-Management (Ausführen von Funktionen zu einem bestimmten Zeitpunkt)
    - componentDidMount()
    - componentWillUnmount()
    - etc.

Einführung von Hooks

- Möglichkeiten von Funktionskomponenten haben sich in 2019 mit dem React-Release 16.8 verändert
- Einführung von Hooks
- Durch Hooks…
  - Können Funktionskomponenten einen State verwalten.
  - Können in Funktionskomponenten Life-Cycle-Funktionen verwendet werden
- Hooks sind Funktionen von React, mit denen spezielle Aspekte umgesetzt werden können, die in Funktionen sonst nicht möglich wären
- Hooks können nur in Funktionskomponenten verwendet werden
- Mit den Hooks kann in Funktionskomponenten fast alles gemacht werden, was in Klassenkomponenten schon möglich ist

Übersicht über Hooks

- Häufig genutzte Hooks
  - **useState**: State-Variablen anlegen
  - **useEffect**: Ausführen von Funktionen nach dem Zeichnen der Komponente
  - **useContext**: Verwenden von Kontextobjekten
  - **useReducer**: State-Management wie bei Redux
- Selten verwendete Hooks
  - useCallback, useMemo, useRef, useImperativeHandle, useLayoutEffect, useDebugValue

State-Management

Grundlagen

- Daten in Komponenten können unterschieden werden in **props** und **state**
  - Props sind statisch und sollten nicht verändert werden
  - State sind lokale Daten, die verändert werden können
  - Änderung im State sind observable → Neuzeichnen der Komponente
- In Klassenkomponente werden Props im Konstruktor übergeben

```
class StartPage extends Component {
    constructor(props) {
        super(props);
        this.state = {
            user: null
        };
    }

    render() {
        return <h1>Who is living young, wild, and free? - {this.props.name}</h1>;
    }
}
```

Hinweise:
- Konstruktor der Super-Klasse muss aufgerufen werden
- Für Zugriff auf props in Klassenkomponente muss „this“ verwendet werden

Grundlagen

- In Funktionskomponente werden props als Funktionsparameter übergeben

```
function Foo(props) {

    return <h1>Hello, {props.name}</h1>;

}
```

- Die Übergabe von Props wird über Vater-Komponente für Funktions- und Klassenkomponenten gleichermaßen umgesetzt

```
return (
        <>
            …
            <Foo name="Manfred" />
        </>
    )
```

State ändern: Klassenkomponente

```
class ClassComponent extends React.Component {
 constructor(props) {
   super(props);
   this.state = {
     count: 0
   };
 }

 render() {
   return (
     <div>
       <p>count: {this.state.count} times</p>
       <button onClick={() => this.setState({ count: this.state.count + 1 })}>
         Click
       </button>
     </div>
   );
 }
}
```

Counter wird beim Anlegen der Komponenten initialisiert

Mit this.setState() wird State geändert. Dadurch wird die Komponente automatisch neu gezeichnet!

State in Funktionskomponente: useState()-Hook

- Verwenden des Hooks useState()
  - Initialwert wird übergeben: useState(initialwert)
  - Es wird in einem Array die Referenz auf den Wert sowie die Funktion zurückgegeben, mit der der Wert geändert werden kann

```
const FunctionalComponent = () => {
const [count, setCount] = React.useState(0);
return (
   <div>
     <p>count: {count}</p>
     <button onClick={() => setCount(count + 1)}>Click</button>
   </div>
 );
};
```

- Beispiel zeigt: Funktionskomponenten sind tendenziell kleiner als Klassenkomponenten
- Funktional gibt es keinen Unterschied

Life-Cycle-Management

Klassenkomponenten

- Basisklasse von React-Komponente implementiert diverse Funktionen, die zu bestimmten Zeiten aufgerufen werden
- Können überschrieben werden, um bestimmte Aspekte umzusetzen
- Beispiel: componentDidMount() wird aufgerufen, wenn Komponente im DOM-Tree eingehangen wurde

```
class ClassComponent extends React.Component {

 componentDidMount() {
   console.log("Hello");
 }

 render() {
   return <h1>Hello, World</h1>;
 }
}
```

Klassenkomponenten: Life-cycle-Methoden

- **Methoden für das Mounting von Komponenten (ausgew.)**
  - constructor()
  - render()
  - componentDidMount()
- **Methoden für das Aktualisieren (ausgew.)**
  - shouldComponentUpdate()
  - render()
  - componentDidUpdate()
- **Methode für das Un-Mounting**
  - componentWillUnmount()
- **Error-Handling**
  - componentDidCatch()

Funktionskomponente

- Ersatz für componentDidMount()

```
const FunctionalComponent = () => {

 React.useEffect(() => {console.log("Hello"); }, []);

 return <h1>Hello, World</h1>;

};
```

- useEffect() bekommt 2 Parameter
  - Der Effekt: eine Funktion, die ausgeführt werden soll
  - Dependency List: ein Array von beobachtbaren State-Objekten
- Der Effekt wird ausgelöst, wenn sich eines der beobachtbaren Objekte sich ändert

Funktionskomponente

- Ersatz für componentWillUnmount()

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

        return () => {
            console.log("Bye");
        };

 }, []);
```

- Wie beim Ersatz von componentWillMount verwenden von [] als Observable-List
- Jedoch Rückgabe von Arrow-Funktion mit „return“
- Auswerten der Rückgabe wird ausgeführt, wenn Komponenten aus DOM entfernt wird

Funktionskomponente

- Automatisches Ausführen von Funktionen bei Änderung eines State-Items
- Wenn counter sich ändert, wird books-Array erweitert

```
function BooksList () {
   const [books, updateBooks] = React.useState([]);
   const [counter, updateCounter] = React.useState(0);

   React.useEffect(function effectFunction() {
       if (books) {
           updateBooks([...books, { name: 'A new Book', id: '...'}]);
       }
   }, [counter]);

   const incrementCounter = () => {
       updateCounter(counter + 1);
   }
   ...
}
```

ComponentDidMount und componentWillUnmount

- Klassenkomponente

```
componentDidMount() {
  window.addEventListener('mousemove', () => {})
}

componentWillUnmount() {
  window.removeEventListener('mousemove', () => {})
}
```

- Funktionskomponente

```
useEffect(() => {
  window.addEventListener('mousemove', () => {});

  // returned function will be called on component unmount
  return () => {
    window.removeEventListener('mousemove', () => {})
  }
}, [])
```

State-Management mit useReducer

useReducer-Hook

- useReducer() erlaubt ähnliches State-Management über Reducer wie bei Redux, jeder eher bezogen auf eine Komponente
- Sollte verwendet werden, wenn kein Redux verwendet wird und Komponente einen komplexeren State hat (mit mehreren Werte, die in Kombination verändert werden sollen)
- In diesen Fällen besser als useState(), weil dort nur Änderungen von einem State-Attribut vorgenommen werden können

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

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return {count: state.count + 1};
    case 'decrement':
      return {count: state.count - 1};
    default:
      throw new Error();
  }
}
function Counter() {
  const [state, dispatch] = useReducer(reducer, initialState);
  return (
    <>
      Count: {state.count}
      <button onClick={() => dispatch({type: 'decrement'})}>-</button>
      <button onClick={() => dispatch({type: 'increment'})}>+</button>
    </>
  );
}
```

Store mit React-Funktionen umsetzen

- Mit React-Context kann einfacher Redux-Store nachgebildet werden

```
import React, { createContext, useReducer, useContext } from 'react';
…
const StoreContext = createContext(null);

export function StoreProvider({ children }) {
  const [state, dispatch] = useReducer(reducer, defaultState);
  const value = { state, dispatch };

  return <StoreContext.Provider value={value}>{children}</StoreContext.Provider>;
}

export const useStore = () => useContext(StoreContext)
```

Store mit React-Funktionen umsetzen

- Nutzen des erzeugten Stores

```
import React from 'react';
import { useStore } from './store-provider';

export default function Counter() {
  const { state, dispatch } = useStore();

  return (
    <section className="counter">
      <div className="value">{state.counter}</div>
      <button onClick={() => dispatch({ type: 'COUNTER_INC' })}>Add</button>
      <button onClick={() => dispatch({ type: 'COUNTER_DEC' })}>Subtract</button>
    </section>
  );
}
```

useReducer-Hook

- Es fehlt die selektive Update-Funktion über mapStateToProps()
- Bei State-Änderungen werden stets alle Komponenten aktualisiert
- Bei kleinen Anwendungen ok, bei großen wird das zum Problem
- Ersetzt nicht globalen Anwendungs-State-Tree von Redux

Allgemeines

Was ist besser?

- Derzeit ist Nutzung von Funktionskomponenten weiter verbreitet
- Es gibt jedoch keinen funktionalen Vorteil
- Der Hauptunterschied besteht in der Syntax
- Funktionskomponenten sind in der Regel kürzer, das bedeutet jedoch nicht, dass der Code immer besser lesbar oder wartbarer ist
- Es wurde vermutet, dass Funktionskomponenten schneller sind, das konnten Performance-Tests jedoch nicht belegen
- Es gibt noch einige Aspekte in Klassenkomponenten, die nicht mit Funktionskomponenten möglich sind
- Bei sehr großen Anwendung kann mit Klassenkomponenten gerade in Verbindung mit Redux besser gesteuert werden, welche Komponenten neu gezeichnet werden.