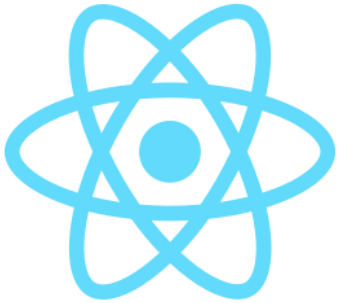
Front-End: React -Teil 1

Web-Engineering II

Prof. Dr. Sebastian von Klinski

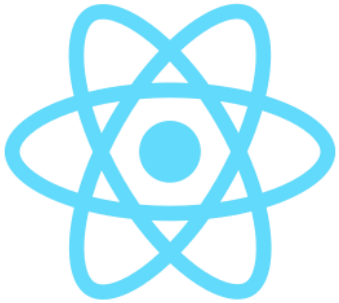
React



- JavaScript-Softwarebibliothek (kein Framework!)
- Entwickelt von Facebook
- Komponenten werden als selbst definierte HTML-Tags repräsentiert
- Häufig für die Umsetzung von Single-Page-Applications (client-seitig)
- Mit Node.js aber auch serverseitiges (Vor-)Rendering möglich
- Open Source Projekt
- Ansatz
  - Aktualisieren des DOM aufwändig
  - Schaffung von Virtual DOM
  - DOM-Diffing: selektive Aktualisieren des DOM auf Basis eines Vergleichs zwischen ursprünglichem und geändertem Virtual DOM

  → im DOM wird nur der wirklich geänderte Teil aktualisiert
- Performance von React im Vergleich zu anderen Frameworks sehr gut
- Aber: Vermischung von Rendering und Logik (Verstoß gegen MVC-Paradima)

React



- Setzt nicht alle Aspekte einer Webanwendung um
  - Definition von Komponenten
  - Verwalten des States innnerhalb der Komponenten
  - Layout wird mit CSS gemacht
- Es gibt mehrere Aspekte, für die weitere Software-Komponenten benötigt werden
  - Verwalten von Anwendungs-State: z.B. Redux
  - Kommunikation mit REST-Backend: Redux-Thunk, Axios, etc.
- Ohne diese weiteren Software-Komponenten wäre professionelle Umsetzung von Web-Seiten aufwändig
- Abweichender Ansatz zu Frameworks wie Angular:
  - Für Umsetzung von Anwendung können viele Komponenten zusammengeführt werden
  - Es sind viele Umsetzungsalternativen möglich

Komponenten

Komponenten

- Komponenten können…
  - Als Funktion oder als Klasse umgesetzt werden
  - Bei Instanziieren werden Komponente Properties übergeben (props)
- Funktionskomponente:

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

- Klassenkomponente:

```
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

Komponenten: Klassen und Funktionen

- Funktionskomponenten
  - Haben keinen State
  - HTML-Content wird mit return() zurückgegeben
- Für Klassenkomponenten gilt:
  - Klassen haben einen State
  - Der HTML-Content wird in der Funktion render() zurückgegeben
  - Haben einen Life-Cycle
    - Der Konstruktor wird beim Instanziieren der Komponente aufgerufen
    - Die Funktion componentDidMount() wird aufgerufen, wenn die Komponenten im DOM eingebaut wurde
    - Die Funktion componentWillUnmount() wird aufgerufen, kurz bevor die Komponente aus dem DOM entfernt wird

Funktionskomponente: Timer

Kom-
ponen-
te

```
function Clock(props) {
  return (
            <div><h1>Hello, world!</h1>
               <h2>It is {props.date.toLocaleTimeString()}.</h2>
            </div>
  );
}
```

Instanz
im
DOM
einbauen

```
function tick() {
  ReactDOM.render(
    <Clock date={new Date()} />, document.getElementById('root')
  );
}
```

aktualisieren

```
setInterval(tick, 1000);
```

Komponenten: Timer als Funktion oder Klasse

- Bei der Umsetzung als Funktion
  - Muss der Trigger zum Aktualisieren von Außen kommen
  - Muss die Komponente immer wieder in den DOM kopiert werden
- Bei Klassenkomponenten ist der Implementierungsansatz anders
  - Klassenkomponente hat sowohl State als auch Life-Cycle
  - Dadurch kann Komponente eigenständiger umgesetzt werden
  - Keine Trigger von außen notwendig
  - Durch Setzen von State

Komponenten: Timer als Klasse

Im Konstruktor wird das Anfangsdatum gesetzt

Sobald die Komponente angezeigt wird, wird die automatische Aktualisierung gestartet

Sobald die Komponente Nicht mehr angezeigt wird, kann der Timer gestoppt werden

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  componentDidMount() {
    this.timerID = setInterval(
      () => this.tick(),
      1000
    );
  }

  componentWillUnmount() {
    clearInterval(this.timerID);
  }
…
```

Komponenten: Timer als Klasse

Funktion ändert State, dadurch wird Komponente neu gerendert

Render-Methode gibt aktualisierten HTML-Code zurück

Komponente wird eingefügt ohne State!

```
…
 tick() {
    this.setState({date: new Date() });
}
render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is
{this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );}}

ReactDOM.render(
  <Clock />,
  document.getElementById('root')
```

Klassenkomponenten

- Der State sollte immer über die Methode setState() verändert werden, nie über direkte Zuweisung
- Nur wenn setState() aufgerufen wird, weiß React, dass die Komponente aktualisiert werden muss

```
// Nicht gut
this.state.comment = 'Hello';
```

- Mit setState()

```
// Gut
this.setState({comment: 'Hello'});
```

Rendering

Render-Funktion

- Komponenten werden mit der render()-Methode in den DOM-Tree eingefügt

```
ReactDOM.render(<p>Hello</p>, document.getElementById('root'));
```

- Parameter
  - HTML-Code
  - Das DOM-Element, in dem der HTML-Code eingefügt werden sollt
- Mit ReactDOM.render() wird Root-Komponente gesetzt
- Es können auch mehrere Root-Komponenten gesetzt werden, es wird jedoch davon abgeraten
  - Root-Komponenten haben eigenen Kontext
  - Direkter Austausch zwischen Root-Komponenten ist nicht möglich
  - Performance wird bei mehreren Root-Komponenten beeinträchtigt

Komponenten

- React-Komponente kann als selbstdefiniertes Tag verwendet werden
- Schritte zum Einfügen von Komponenten
  1. Komponente definieren
  2. Komponente instanziieren
  3. Komponente im DOM einfügen

```
// Komponente definieren
function Welcome(props) {
        return <h1>Hello, {props.name}</h1>;
}
// Komponente anlegen
const element = <Welcome name="Sara" />;
// Komponente im DOM einfügen
ReactDOM.render(
  element,
  document.getElementById('root')
);
```

Komponenten

- Komponenten können auch mehrfach instanziiert werden

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}

function App() {
  return (
    <div>
      <Welcome name="Sara" />
      <Welcome name="Cahal" />
      <Welcome name="Edite" />    </div>
  );
}

ReactDOM.render(
  <App />,
  document.getElementById('root')
);
```

Komponenten: Properties

- Bei Funktionskomponenten werden die Properties im Tag übergeben
- Sie können/ sollen in der Komponente nur gelesen, nicht verändert werden

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}

function App() {
  return (
    <div>
      <Welcome name="Sara" />
      <Welcome name="Cahal" />
      <Welcome name="Edite" />
    </div>
  );
}
```

Komponenten: Nesting

- Durch Schachtelung können Komponenten einfacher gestaltet werden

```
function Comment(props) {
  return (
    <div className="Comment">
      <div className="UserInfo">
        <img className="Avatar"
          src={props.author.avatarUrl}
          alt={props.author.name}
        />
        <div className="UserInfo-name">
          {props.author.name}
        </div>
      </div>
      …
    </div>
  );
}
```

Komponenten: Nesting

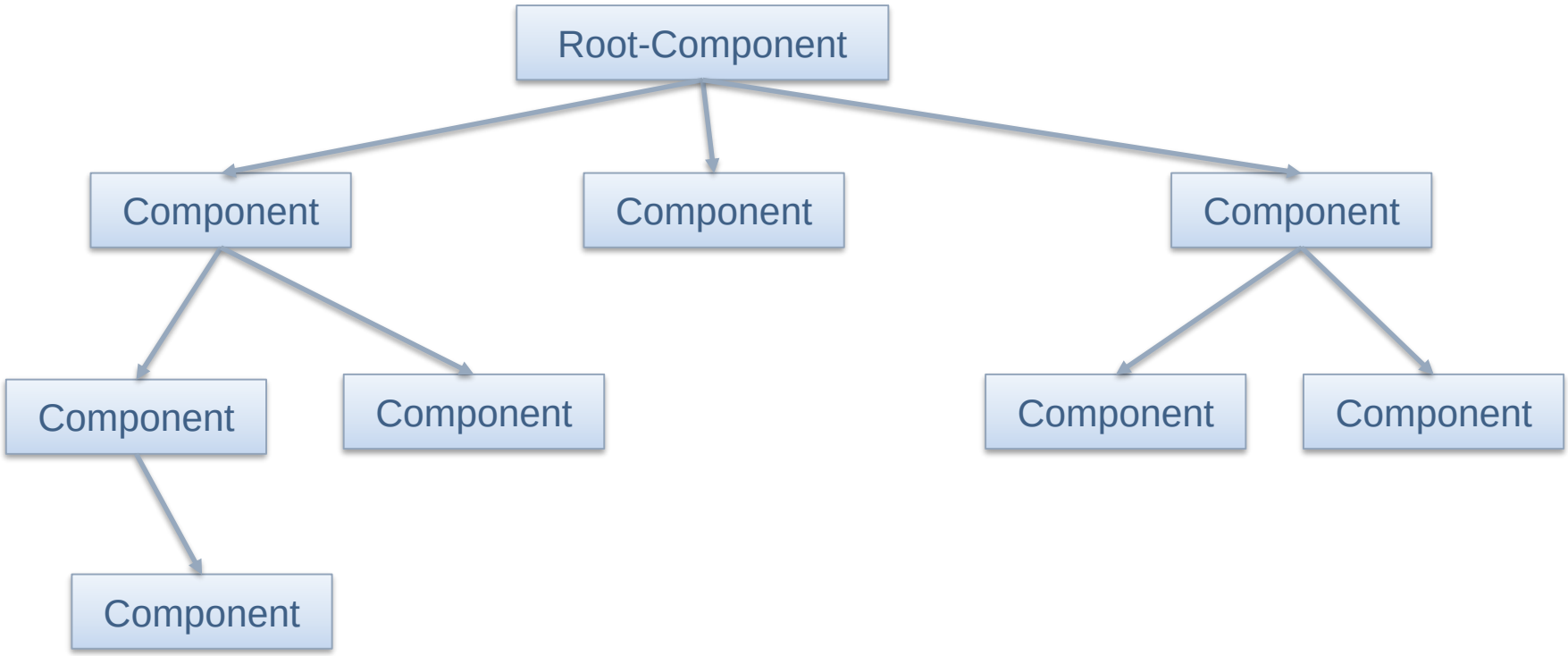
- Komponenten werden über die Custom-Tags ineinander geschachtelt

```
function Avatar(props) {
  return (
    <img className="Avatar"
              src={props.user.avatarUrl}
              alt={props.user.name}    />
);}
```

```
function UserInfo(props) {
  return (
     <div className="UserInfo">
       <Avatar user={props.user} />
       <div className="UserInfo-name">{props.user.name}</div>
    </div>
);}
```

Komponenten: Nesting

- React-Anwendungen setzt sich in der Regel aus hierarchisch strukturierten Komponenten zusammen



JSX

React: JSX

- JavaScript XML: erlaubt Einbettung von HTML im JavaScript-Code
- XHTML-Code ohne aufwändiges createElement(), etc
- Konvertiert HTML-Tags in React-Elemente

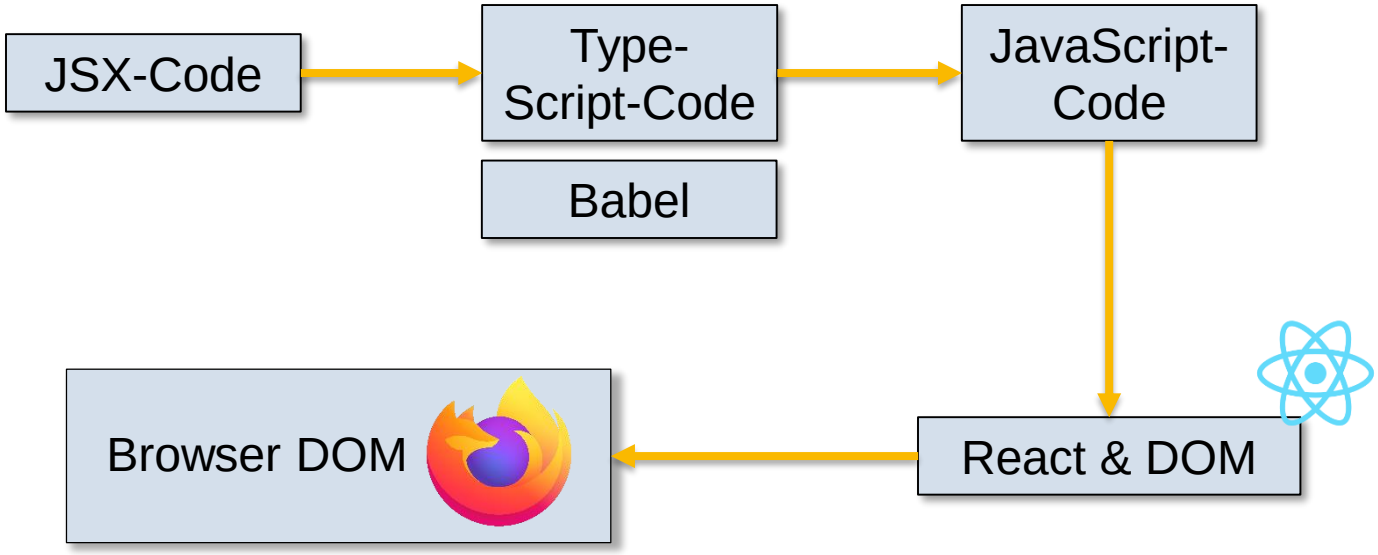
```
// Ohne JSX
const myelement = React.createElement('h1', {}, 'I do not use JSX!');
// Mit JSX
const myelement = <h1>I Love JSX!</h1>;

ReactDOM.render(myelement, document.getElementById('root'));
```

- Auch Einbettung von dynamischen Elementen möglich

```
const myelement = <h1>React is {5 + 5} times better with JSX</h1>;
```

React: JSX-Transkompilation

- JSX wird durch Transkompilation in React-Anwendung integriert (Source-to-Source-Kompilation)
- Übersetzung zunächst nach Babel oder Type-Script → dann nach JavaScript-Code → dann Einbindung in React-Komponente → dann Einbettung des Elements im DOM-Tree



React: JSX

- Ganze HTML-Blöcke können in runden Klammern geschrieben werden
- Wie bei XML müssen Elemente geschlossen werden

```
const myelement = (
  <ul>
    <li>Apples</li>
    <li>Bananas</li>
    <li>Cherries</li>
  </ul>
);

const myelement2 = <input type="text" />;
```

React: Fragments

- Bei JSX muss auf der obersten Ebene immer ein einzelnes Root-Element sein.
- Das bedeutet, dass die folgenden Zeilen nicht möglich sind:

```
const myelement = (
    <li>Apples</li>
    <li>Bananas</li>
    <li>Cherries</li>
);
```

Fehler

React: Fragments

- In manchen Fällen würde es keine Sinn machen, ein weiteres Root-Element einzufügen

```
class Table extends React.Component {
  render() {
    return (
      <table>
        <tr>
          <Columns />
        </tr>
      </table>
    );
  }
}
```

React: Fragments

- Wenn ein unnötiges Root-Element vermieden werden soll, können die Elemente in ein React-Fragment eingebettet werden
- Das Fragment-Element wird beim Rendering nicht übernommen

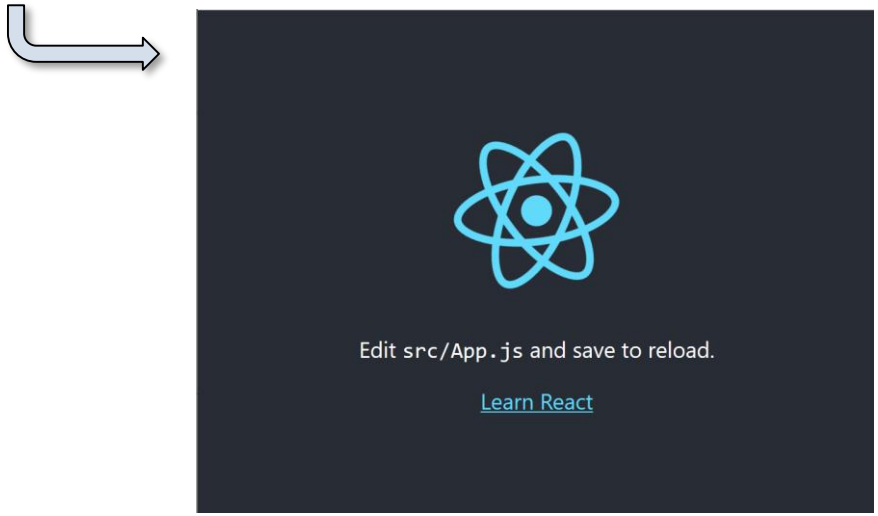
```
class Columns extends React.Component {
  render() {
    return (
      <React.Fragment>
        <td>Hello</td>
        <td>World</td>
      </React.Fragment>
    );
  }
}
```

Anwendung anlegen

React: Setup

- Aufsetzen von Anwendung am besten über das Modul „create-react-app“

```
npm install -g create-react-app
create-react-app my-app
cd my-app
npm start
```




```
your-app
├── README.md
├── node_modules
├── package.json
├── .gitignore
├── public
│   ├── favicon.ico
│   ├── index.html
│   └── manifest.json
└── src
    ├── App.css
    ├── App.js
    ├── App.test.js
    ├── index.css
    ├── index.js
    ├── logo.svg
    └── serviceWorker.js
```

React: Startprozess

- Aufruf von „npm start“ startet das React-Start-Skript
- React-Start-Skript öffnet schließlich index.html
- Mit dem Laden von index.html wird index.js ausgeführt
  - In index.html gibt es keine Referenz zu index.js
  - Verknüpfung wird automatisch im React-Script durch Webpack umgesetzt
- Webpack
  - Open-Source-JavaScript-Modul-Packer
  - Wird verwendet unter anderem für das Zusammenführen von JavaScript und HTML-Seiten
  - Wenn „index.html“ aufgerufen wird, ergänzt Webpack automatisch index.js aus dem „src“-Verzeichnis

React: Startprozess

- Im index.html gibt es in der Regel ein Div mit der ID „root“
- Der Knoten könnte jedoch auch beliebigen anderen Namen haben
- Bei Standardanwendung wird dieser Knoten im DOM durch index.js mit der React-Anwendung ersetzt

```
<html>
  <body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
    <div id="root"></div>
  </body>
</html>
```

React: Startprozess index.js

- Index.js integriert React-Anwendung in den DOM
  1. Laden von React und ReactDOM
  2. Laden der „App“: React-Funktionskomponente
  3. Über DOM wird DIV-Element mit ID „root durch React-Komponente ersetzt

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';
…
ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);
```

React: Startprozess App.js

- In der Datei App.js wird häufig die Root-Komponente definiert
- Sie muss jedoch nicht App heißen, kann beliebigen anderen Namen haben
- Mit return wird JSX-Code mit React-Komponenten zurückgegeben
- Mit „export default“ wird angegeben, welche Komponente standardmäßig mit „require“ importiert wird

```
import React from 'react';
import logo from './logo.svg';
import './App.css';

function App() {
  return (
    <div className="App">
      … [HTML-Content]
    </div>
  );
}
export default App;
```

React: Startprozess

- Hinweis: Startprozess dauert sehr lange…
- Aber Aktualisierungen werden automatisch übernommen!
- Nach Änderung einer Datei wird automatisch in Konsole Aktualisierung bestätigt

```
Compiled successfully!

You can now view de.coservices.daif.core.react in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://192.168.0.45:3000

Note that the development build is not optimized.
To create a prod
```

CSS-Einbettung

CSS-Einbettung

- Layout wird bei React genauso wie bei HTML mit CSS umgesetzt
- Viele Ansätze zum Einbetten von CSS möglich
- Beispielsweise
  1. Laden von CSS-Datei in Komponente
  2. Laden von CSS-Modul in Komponente
  3. Einbetten von CSS-Code im Modul
- Es gibt weitere Ansätze, doch das sind die gängigen Ansätze

CSS-Einbettung

- In Komponenten können CSS-Dateien direkt geladen werden
- Wichtig: class-Attribut muss umbenannt werden in „className“

```
import React from 'react';
import './DottedBox.css';

const DottedBox = () => (
  <div className="DottedBox">
    <p className="DottedBox_content">Get started with CSS styling</p>
  </div>
);

export default DottedBox;
```

```
.DottedBox {
  margin: 40px;
  border: 5px dotted pink;
}
…
```

CSS-Einbettung: Anwendung der Styles

- Zum Anwenden der CSS-Styles werden Attribute in den Tags gesetzt
- Hinweise: nicht alle Aspekte von Bibliotheken wie Bootstrap können übernommen werden, weil sie JavaScript beinhalten.

```
…
 <input
        type="number"
        name="id"
        id="id"
        value={id}
        onChange={this.handleChange} />

       <input className=“loginButton" type="button" value="Submit"
              onClick={this.handleSubmit} />
     </form>
…
```

CSS-Einbettung

- Alternativ können CSS-Styles auch als **CSS Modules** geladen werden
- Das Modul sollte *.module.css heißen
- Zugriff auf Styles über das importierte CSS-Modul

```
import React from 'react';
import ReactDOM from 'react-dom';
import styles from './mystyle.module.css';

class Car extends React.Component {
  render() {
    return <h1 className={styles.bigblue}>Hello Car!</h1>;
  }
}

export default Car;
```

CSS-Einbettung

- CSS-Styles können auch in Komponente selber definiert werden
- Dieser Ansatz ist jedoch nicht empfehlenswert, da Wiederverwendung nicht möglich ist

```
import React from 'react';

const divStyle = {
  margin: '40px',
  border: '5px solid pink'
};
const Box = () => (
  <div style={divStyle}>
    <p>Get started with inline style</p>
  </div>
);

export default Box;
```

CSS-Einbettung: Anwendung der Styles

- In render()-Methode können Properties für bedingte Formatierungen verwendet werden
- **Hinweis**:
  - „class“ ist ein Schlüsselwort in JavaScript
  - JSX wird in JavaScript transkompiliert
  - Zur Vermeidung von Problemen wird daher häufig anstelle von „class“ das Attribute „className“ verwendet!
  - Es kann aber auch class verwendet werden

```
…
render() {
  let className = 'menu';
  if (this.props.isActive) {
    className += 'menu-active';
  }
  return <span className={className}>Menu</span>
}
…
```

Properties und State

Properties und State

- Properties sind Objekte, die von „oben“ nach „unten“ gereicht werden (von Parent-Komponenten zu Child-Komponenten)
- State sind die Objekte in einer Klasse
- Properties und State stellen den „lokalen“ State in den Funktionen und Klassen dar
- Grundlage für das Rendering der Komponenten
- Properties sollten grundsätzlich nicht geändert werden, sondern immer von Parent-Komponente vorgegeben werden.
- In Klassen ergänzen Properties den lokalen State.
- Der State in Klassen kann verändert werden

→ Funktionskomponenten sollten die eigenen Daten nie verändern, nur darstellen!

→ Klassen können hingegen den eigenen Zustand verwalten und ändern

Properties: Beispiel

- Characters[] werden in Klassenkomponente definiert und an Child-Komponente weitergegeben
- Bei Übergabe wird der Name der Property festgelegt

```
class App extends Component {
    render() {
        const characters = [
            { name: 'Charlie', job: 'Janitor', },
            { name: 'Mac', job: 'Bouncer'   },
        ]
        return (
            <div className="container">
                <Table characterData={characters} />
            </div>)
    }
}
```

Properties: Beispiel

- Table-Klasse holt sich den Array aus den Properties und leitet den Array an TableBody-Element weiter

```
class Table extends Component {
    render() {

        const { characterData } = this.props

        return (
            <table>
                <TableHeader />
                <TableBody characterData={characterData} />
            </table>
        )
    }
}
```

Properties: Beispiel

- TableHeader ist einfache Funktion, mit der der Header-Code delegiert wird

```
const TableHeader = () => {
    return (
        <thead>
            <tr>
                <th>Name</th>
                <th>Job</th>
            </tr>
        </thead>
    )
}
```

Properties: Beispiel

- TableBody ist Funktion, die Properties erhält
- Aus den Properties wird der characterData-Array geholt und über „map“ durchlaufen
- Für jedes Element wird ein HTML-Table-Row-Element in „rows“ geschrieben.

```
const TableBody = props => {
    const rows = props.characterData.map((row, index) => {
        return (
            <tr key={index}>
                <td>{row.name}</td>
                <td>{row.job}</td>
            </tr>
        )
    })

    return <tbody>{rows}</tbody>
}
```

State in Klassenkomponenten

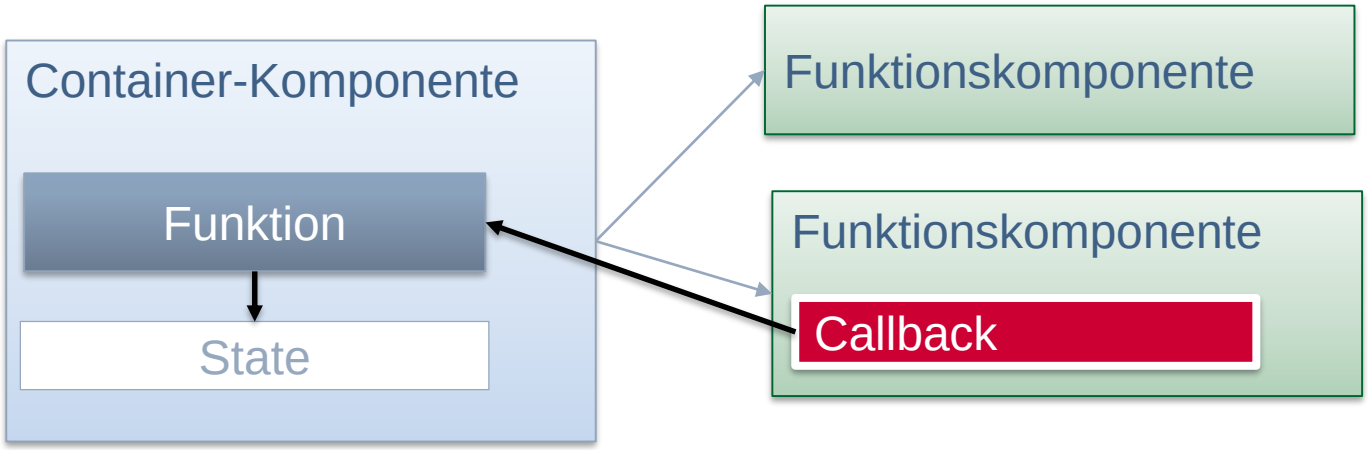
- Der State in der Klasse umfasst Daten aus dem Backend und zum aktuellen Stand der Client-Anwendung, die noch nicht persistiert sind
- Beispiel: Waren im Shopping-Cart, die noch nicht bestellt wurden

```
class App extends Component {
  state = {
    characters: [],
  }
}
```

- Im Gegensatz zu Funktionskomponenten können Klassen Methoden zum Ändern des States definieren
- Die Methoden können an Child-Komponenten per Funktions-Pointer weitergereicht werden

State in Klassenkomponenten

- Gängiger Ansatz
  - Klassenkomponente ist „Container“-Komponente, die den State verwaltet
  - Die Klassenkomponenten beinhaltet mehrere Funktionskomponenten
  - Die Funktionskomponenten übernehmen die Darstellung der Weboberfläche
  - Die Klassenkomponente gibt an die Funktionskomponenten Funktions-Pointer, damit Funktionskomponenten State verändern können



State in Klassenkomponenten

- Beispiel für Methode zum Ändern des States
- Schritte zum Löschen von Eintrag im Array
  1. Array aus dem State holen
  2. Den State neu setzen mit dem gefilterten Array

```
…
removeCharacter = index => {
        const { characters } = this.state

        this.setState({
            characters: characters.filter((character, i) => {
                return i !== index
            }),
        })
    }
…
```

State in Klassenkomponenten

- Weiterleiten der Methode an Funktionskomponenten
  - Funktion wird mit dem Namen „removeCharacter“ an Table-Komponente weitergeleitet

```
…
render() {
  const { characters } = this.state

  return (
    <div className="container">
      <Table characterData={characters} removeCharacter={this.removeCharacter} />
    </div>
  )
}
…
```

State in Klassenkomponenten

- Funktionskomponente kann Methode an weitere Funktionskomponente weiterleiten
- Die Methode muss jedoch zunächst aus den Properties übernommen werden

```
const Table = props => {
  const { characterData, removeCharacter } = props

  return (
    <table>
      <TableHeader />
      <TableBody characterData={characterData}
                 removeCharacter={removeCharacter} />
    </table>
  )
}
```

State in Klassenkomponenten

- In der Funktionskomponente kann Methode an Event von HTML-Komponenten gehangen werden (z.B. an onClick())

```
const TableBody = props => {
   const rows = props.characterData.map((row, index) => {
      return (
         <tr key={index}>
            <td>{row.name}</td>
            <td>{row.job}</td>
            <td>
               <button onClick={() => props.removeCharacter(index)}>Delete</button>
            </td>
         </tr>
      )
   })

   return <tbody>{rows}</tbody>
}
```

Listen

Darstellen von Listen

- Listen werden häufig in -Elementen dargestellt

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <li>{number}</li>
  );

  return (
    <ul>{listItems}</ul>
   );
}

const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
  <NumberList numbers={numbers} />,
  document.getElementById('root')
);
```

Darstellen von Listen

- Zum Bearbeiten von Listen sollten den Listenelementen IDs zugewiesen werden

```
const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((number) =>
  <li key={number.toString()}>{number}</li>
);
```

- Besser ist es, wenn die Objekte richtige IDs haben

```
const todoItems = todos.map((todo) =>
  <li key={todo.id}>{todo.text}</li>
);
```