

09 React

Einführung in die moderne (Full-Stack) Webentwicklung mit
JavaScript, Node.js und React.js

Nils Constantin Hellwig
Wissenschaftliche Hilfskraft
RECHENZENTRUM



Universität Regensburg

Überblick Themen

- Was ist React?
- Components
- JSX
- Events
- Hooks
- Router
- Styling

Was ist React?

- JavaScript-Bibliothek zur Erstellung von Benutzeroberflächen
- Ermöglicht die Erstellung von Single-Page Anwendungen (SPA) (z.B. unsere To-Do App)
mit wiederverwendbaren UI-Komponenten
- React kann kostenlos und kommerziell genutzt werden und ist Open-Source

Historie und Entwicklung von React

- Veröffentlichung im Juli 2013 (V0.3.0)
- Aktuelle Version: 18.2.0 (Juni 2023)
- Es wird von Meta (ehemals Facebook) und einer Gemeinschaft von einzelnen Entwickler*innen und Unternehmen gepflegt
- React wird von Tech-Giganten wie Meta, Netflix, Dropbox, Discord verwendet

Wie funktioniert React? Virtual DOM!

- Anstatt direkt das DOM des Browsers zu verändern, erstellt React im Speicher ein Virtual DOM
- Das Virtual DOM ist eine leichtgewichtige JavaScript-Darstellung des Document Object Model (DOM)
- Die Aktualisierung des virtuellen DOM ist vergleichsweise schneller als die Aktualisierung des tatsächlichen DOM (über JavaScript)

Wie funktioniert React? Virtual DOM!

- Das Framework kann Änderungen an dem virtuellen DOM relativ kostengünstig vornehmen (weniger rechenintensiv)
- Das Framework findet dann die Unterschiede zwischen dem vorherigen virtuellen DOM und der aktuellen DOM und nimmt nur die notwendigen Änderungen an der tatsächlichen DOM vor
- Das *Virtual DOM* wird auch von anderen Web-Frameworks wie Vue.js verwendet

React im Browser

- Der einfachste Weg, React zu verwenden, ist, React direkt in einem HTML-Dokument zu laden
- Man beginnt damit, drei Skripte einzubinden:

```
<script src="https://unpkg.com/react@18/umd/react.development.js" crossorigin></script >
```

```
<script src="https://unpkg.com/react-dom@18/umd/react-dom.development.js" crossorigin></script >
```

```
<script src="https://unpkg.com/@babel/standalone/babel.min.js"></script >
```

- Die ersten beiden Skripte ermöglichen es, React-Code in JavaScript zu schreiben
- Babel ermöglicht JSX-Syntax und ES6 in älteren Browsern zu schreiben

React im Browser

```
<!DOCTYPE html>
<html>
  <head>
    <script src="https://unpkg.com/react@18/umd/react.development.js" crossorigin></script>
    <script src="https://unpkg.com/react-dom@18/umd/react-dom.development.js" crossorigin></script>
    <script src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
  </head>
  <body>
    <div id="mydiv"></div> // #mydiv

    <script type="text/babel">
      function Hello() {
        return <h1>Hello World!</h1>;
      }

      ReactDOM.render(<Hello />, document.getElementById("mydiv"));
    </script>
  </body>
</html>
```


React Environment

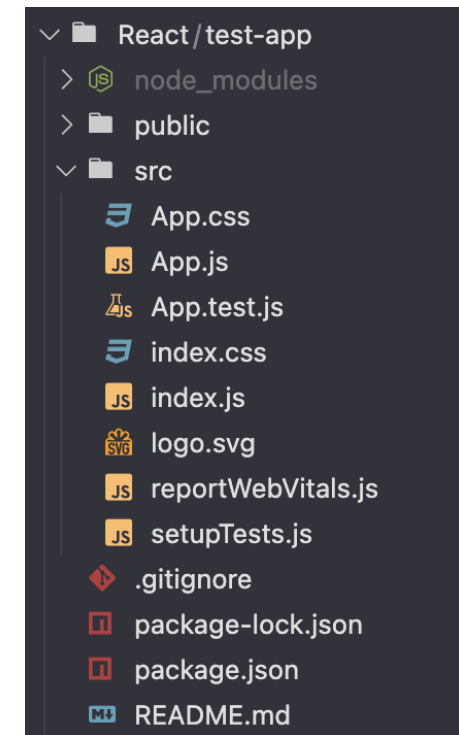
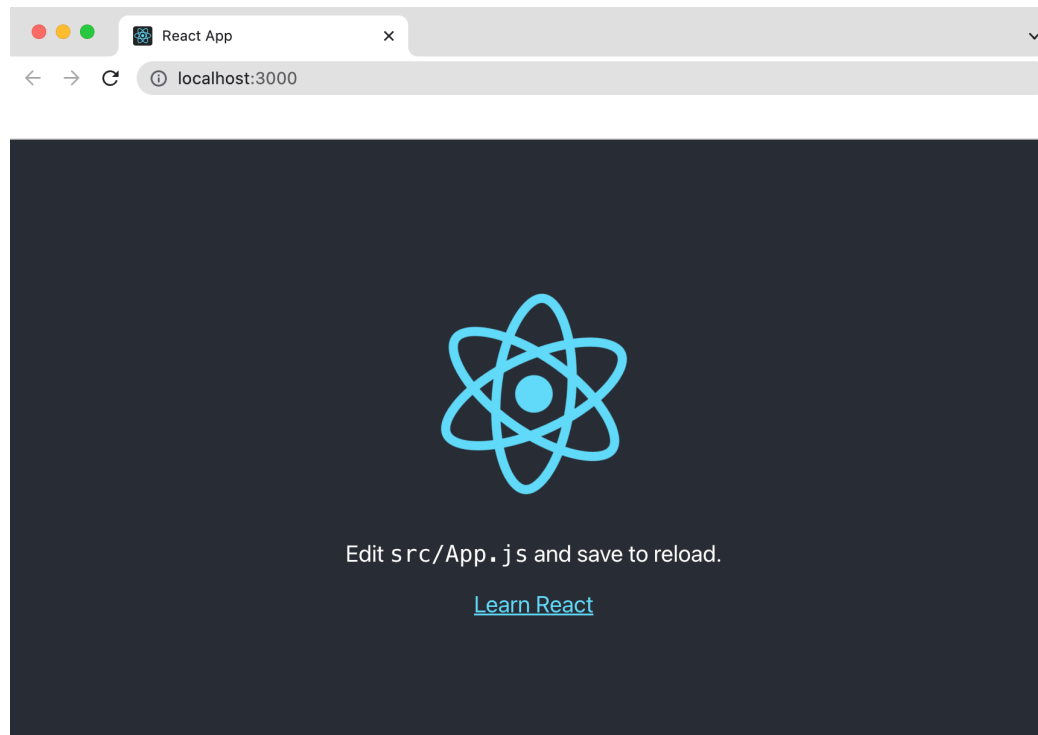
- Die Verwendung von React mithilfe von Skript-Importen kann für Testzwecke in Ordnung sein aber für die Produktion sollte ein React-Environment eingerichtet werden
- Ein React-Environment bietet eine Entwicklungsumgebung, die für die React-Entwicklung optimiert ist
- Dazu gehören Tools zum Builden, Testen und Debuggen des React-Codes
- Ein React Environment mit npm bietet die Möglichkeit, Abhängigkeiten des Projekts zu verwalten

React Environment

- Wenn man Node.js und somit npx installiert hat, kann mithilfe von `create-react-app` eine React-Anwendung erstellt werden
- Befehl, um eine React-Anwendung namens my-react-app zu erstellen:

```
$ npx create-react-app <Name der neuen Anwendung>
```
- Die Anwendung kann dann in einem Development-Server (`localhost:3000`) gestartet werden durch Ausführen von `npm start`

React Environment



React – /public

- Der `/public` - Ordner im Hauptverzeichnis eines React-Projekts dient als Ort für statische Dateien (u.a. Favicon, Bilder, `manifest.json`), die direkt von einem Webbrowser geladen werden können
- Innerhalb dieses Ordners befindet sich auch die Datei `index.html`, diese enthält das grundlegende HTML-Layout der Anwendung und definiert das Ziel-Element mit der ID `"root"`, in die die React Anwendung gerendert wird

React: Render HTML

```
index.html M x
React > test-app > public > index.html
1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="utf-8" />
5     <link rel="icon" href="%PUBLIC_URL%/favicon.ico" />
6     <meta name="viewport" content="width=device-width, initial-scale=1" />
7     <meta name="theme-color" content="#000000" />
8     <meta name="description" content="Web site created using create-react-app" />
9     <link rel="apple-touch-icon" href="%PUBLIC_URL%/logo192.png" />
10    <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
11    <title>React App</title>
12  </head>
13  <body>
14    <noscript>You need to enable JavaScript to run this app.</noscript>
15    <div id="root"></div> // #root
16  </body>
17 </html>
18
19
20
```

React: Render HTML

- React rendert HTML auf einer Webseite mit Hilfe von `render()`
- Der Zweck der Funktion besteht darin, den angegebenen HTML-Code innerhalb des angegebenen HTML-Elements anzuzeigen:

```
const root = ReactDOM.createRoot(document.getElementById('root'));  
root.render(  
  <App />  
);
```

React JSX: Grundlagen

- JSX steht für JavaScript XML und ist eine Syntax-Erweiterung für JavaScript, die es ermöglicht, HTML-ähnliche Strukturen in JavaScript zu schreiben
- JSX ermöglicht es uns, HTML-Elemente in JavaScript zu schreiben und sie im DOM ohne Methoden wie `createElement()` oder `appendChild()`-Methoden zu platzieren / manipulieren
- Beispiel:

```
const myElement = <h1>Hello World!</h1>;
```

```
const root = ReactDOM.createRoot(document.getElementById('root'));
```

```
root.render(myElement);
```

React JSX: Expressions

- Mit JSX können Expressions innerhalb geschweifter Klammern { } geschrieben werden
- Eine Expression kann eine React-Variable, eine Eigenschaft oder ein anderer gültiger JavaScript-Ausdruck sein
- Beispiel:

```
const myElement1 = <h1>React is {5 + 5} years old</h1>;
```

```
const myElement2 = <h1>React is {"10" + " years"} old</h1>;
```


React JSX: Expressions

- Das Einfügen eines großen JSX-Blocks ist ebenfalls möglich:

```
const myElement = (  
  <div>  
    <p>Apples</p>  
    <p>Bananas</p>  
    <p>Cherries</p>  
  </div>  
) ;
```

```
const myElement = (  
  <p>Apples</p>  
  <p>Bananas</p>  
  <p>Cherries</p>  
) ;
```

Funktioniert nicht!

- Sollen wie hier zwei Elemente zusammengefasst werden, muss man sie in ein übergeordnetes Element (hier `</div>` – Element) einfügen.

React JSX: className

- Das `class` - Attribut ist ein Attribut für HTML-Elemente, da JSX jedoch als JavaScript gerendert wird und das `class` - Attribut ein reserviertes Wort in JavaScript ist, dürfen Sie es in JSX nicht verwenden
- Stattdessen wird in JSX für Klassen das `className` – Attribut verwendet
- Wenn JSX gerendert wird, werden `className` - Attribute in `class` - Attribute übersetzt
- Beispiel:

```
const myElement = <h1 className="myclass">Hello World</h1>;
```

React JSX: if-Statements

- React unterstützt if-Anweisungen, jedoch nicht innerhalb von JSX
- Um if-Anweisungen in JSX verwenden zu können, sollten diese außerhalb von JSX eingefügt werden oder man verwendet stattdessen den Ternären Operator

React JSX: if-Statements

if-Statements außerhalb von JSX-Code	Ternärer Operator
<pre>const x = 5; let text = "Goodbye"; if (x < 10) { text = "Hello"; } const myElement = <h1>{text}</h1>;</pre>	<pre>const x = 5; const myElement = <h1>{x < 10 ? "Hello" : "Goodbye"}</h1>;</pre>

React Components

- React-Komponenten sind unabhängige und wiederverwendbare Codestücke
- Es gibt zwei Arten von Komponenten: Class Components und Functional Components

The screenshot shows a web application titled "To-Do App". It features a text input field with the placeholder "Was hast du geplant?" and a button labeled "Task hinzufügen". Below this is a section header "Tasks" followed by a list of two tasks: "Essen gehen" and "Schuhe kaufen". Each task has two buttons: "BEARBEITEN" and "LÖSCHEN". Red bounding boxes highlight the app title, the input field, the "Task hinzufügen" button, the "Tasks" header, and the individual task entries with their respective action buttons.

React Components: Class Components

- Class Components sind JavaScript-Klassen, die mit dem Schlüsselwort `class` definiert werden
- Sie besitzen eine eigene "state"-Eigenschaft, die es ermöglicht, den Zustand der Komponente zu speichern und zu ändern
- Vor React 16.8 waren Klassenkomponenten die einzige Möglichkeit, Zustand und Lebenszyklus einer React-Komponente zu kontrollieren
- Class Components verwenden Lebenszyklusmethoden, die aufgerufen werden, wenn die Komponente im DOM initialisiert oder entfernt wird

React Components: Class Components

- Beispiel:

```
class MyComponent extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {  
      message: "Hello World",  
    };  
  }  
  render() {  
    return <h1>{this.state.message}</h1>;  
  }  
}
```



```
const myElement = <MyComponent />;
```

React Components: Functional Components

- Functional Components sind mittlerweile verbreiteter, da sie im Vergleich zu Class Components kompakter (weniger Code, einfacher zu verstehen)
- Eine Functional Component gibt ebenfalls JSX-Code zurück

React Components: Functional Components

- Beispiel:

```
function Car() {  
  
    return <h2>Hi, I am a Car!</h2>;  
  
}
```

React Components aus einer anderen Datei laden

MyComponent.js

```
import React from 'react';  
const MyComponent = () => {  
  return <div>Hello World!</div>;  
};  
export default MyComponent;
```

App.js

```
import React from 'react';  
import HelloWorldComponent from './MyComponent';  
  
function App() {  
  return (  
    <div>  
      <MyComponent />  
    </div>  
  );  
}  
  
export default App;
```

React Props (Properties)

- Props sind Argumente, die an React-Komponenten übergeben werden
- Props werden über HTML-Attribute an Komponenten weitergegeben
- Beispiel:

```
function Car(props) {  
    return <h2>I am a { props.brand }!</h2>;  
}  
  
const myComponent = <Car brand="Ford" />;
```

React Props (Properties)

- Weiteres Beispiel:

```
function Car(props) {  
  return <h2>I am a { props.brand }!</h2>;  
}  
function App() {  
  return (  
    <>  
      <h1>Who lives in my garage?</h1>  
      <Car brand="Ford" />  
    </>  
  );  
}  
const root = ReactDOM.createRoot(document.getElementById('root'));  
root.render(<App />);
```

React Props (Properties)

- Auch Variablen können übergeben werden:

```
function Car(props) {  
  return <h2>I am a { props.brand }!</h2>;  
}  
function App() {  
  const carName = "Ford";  
  return (  
    <div>  
      <h1>Who lives in my garage?</h1>  
      <Car brand={carName} />  
    </div>  
  );  
}  
const root = ReactDOM.createRoot(document.getElementById('root'));  
root.render(<App />);
```

React Props (Properties)

- Auch Objekte können übergeben werden:

```
function Car(props) {  
  return <h2>I am a { props.brand.model }!</h2>;  
}  
  
function App() {  
  const carInfo = { name: "Ford", model: "Mustang" };  
  return (  
    <div>  
      <h1>Who lives in my garage?</h1>  
      <Car brand={ carInfo } />  
    </div>  
  );  
}  
  
const root = ReactDOM.createRoot(document.getElementById("root"));  
root.render(<App />);
```

React Events

- Genau wie HTML-DOM-Ereignisse kann React Aktionen basierend auf Benutzerereignissen durchführen
- React hat die gleichen Ereignisse wie HTML: click, mouseover usw.
- React-Ereignisse werden in der *camelCase*-Syntax geschrieben:
`onClick` anstelle von `onclick`

React Events: Unterschied zu HTML

- React:

```
<button onClick={addProductToList}>Add Product</button>
```

- HTML:

```
<button onclick={addProductToList()}>Add Product</button>
```


React Events: Beispiel

```
function App() {  
  const addProductToList = () => {  
    alert("Product added!");  
  }  
  
  return (  
    <button onClick={addProductToList}>Add Product</button>  
  );  
}  
  
const root = ReactDOM.createRoot(document.getElementById("root"));  
root.render(<App />);
```

React Events: Argument an einen Event-Handler übergeben

Um ein Argument an einen Event-Handler zu übergeben, verwendet man eine Arrow-Function:

```
function SayHelloButton() {  
  const sayHello = (message) => {  
    alert(message);  
  }  
  
  return (  
    <button onClick={() => sayHello("Hello")}>Add Product</button>  
  );  
}  
  
const root = ReactDOM.createRoot(document.getElementById("root"));  
root.render(< SayHelloButton />);
```

React Events: React Event Object

Event-Handler haben Zugriff auf das React-Event, das die Funktion ausgelöst hat:

```
function HelloWorld() {  
  const shoot = (message, event) => {  
    alert(event.type, event.clientX, event.clientY);  
  }  
  return (  
    <button onClick={ (event) => shoot("Click!", event) }>Click!</button>  
  );  
}
```

```
const root = ReactDOM.createRoot(document.getElementById("root"));  
root.render(<HelloWorld />);
```

React Conditional Rendering: if

```
function Weather(props) {  
  if (props.isSunny) {  
    return <SunnyDay/>;  
  }  
  return <RainyDay/>;  
}  
  
const root = ReactDOM.createRoot(document.getElementById('root'));  
root.render(<Weather isSunny={true} />);
```

React Conditional Rendering: Logical && Operator

```
function Garage(props) {  
  const cars = props.cars;  
  return (  
    <>  
      <h1>Garage</h1>  
      {cars.length > 0 &&  
        <h2>{cars.length} cars in your garage. </h2>  
      }  
    </>  
  );  
}  
  
const cars = ['Ford', 'BMW', 'Audi'];  
const root = ReactDOM.createRoot(document.getElementById('root'));  
root.render(<Garage cars={cars} />);
```

Der &&-Operator bedeutet, dass der nachfolgende Ausdruck nur ausgeführt wird, wenn der vorherige Ausdruck wahr (true) ist

React Conditional Rendering: Ternäre Operator

```
function GoalStatusItem(props) {  
  const isGoal = props.isGoal;  
  return (  
    <>  
      { isGoal ? <MadeGoal/> : <MissedGoal/> }  
    </>  
  );  
}
```

Das ?: (Ternärer Operator) in diesem Beispiel ist eine Abkürzung für eine if-else-Anweisung

```
const root = ReactDOM.createRoot(document.getElementById('root'));  
root.render(<GoalStausItem isGoal={false} />);
```

<MissedGoal/>



<MadeGoal/>



React Lists

```
<div id="tasks">  
  <TaskItem task={"Essen gehen"}/>  
  <TaskItem task={"Bananen kaufen"}/>  
</div> // #tasks
```

- In React können Listen mit Schleifen gerendert werden
- Die JavaScript-Methode `.map()` eines Arrays ist im Allgemeinen die bevorzugte Methode dazu

React Lists: Beispiel

```
function Car(props) {  
  return <li>I am a { props.brand }</li>;  
}  
  
function Garage() {  
  const cars = ['Ford', 'BMW', 'Audi'];  
  return (  
    <>  
      <h1>Cars in the garage</h1>  
      <ul>  
        {cars.map((car) => <Car brand={car} />)}  
      </ul>  
    </>  
  );  
}  
  
const root = ReactDOM.createRoot(document.getElementById('root'));  
root.render(<Garage />);
```


React Lists: Keys

- Wenn man den Code ausführt, funktioniert er, man erhält jedoch eine Warnung, dass für die Listenelemente kein Attribut `key` vorhanden ist
- Ein `key` ermöglichen es React, Elemente zu verfolgen
- Keys für jedes Element innerhalb einer Liste **müssen** eindeutig sein
- Es ist jedoch erlaubt, dass Keys in verschiedenen Listen bzw. innerhalb der Anwendung mehrfach vorkommen

React Lists: Keys

Im Allgemeinen sollte der `key` eine eindeutige ID für jedes der Elemente sein

```
function Garage() {  
  const cars = [  
    {id: 1, brand: 'Ford'},  
    {id: 2, brand: 'BMW'},  
    {id: 3, brand: 'Audi'}  
  ];  
  return (  
    <>  
      <h1>Cars in the garage</h1>  
      <ul>  
        {cars.map((car) => <Car key={car.id} brand={car.brand} />)}  
      </ul>  
    </>  
  );  
}
```

React Lists: Keys

Alternativ könnte man auch den Array-Index als Schlüssel verwenden

```
function Garage() {  
  const cars = ['Ford', 'BMW', 'Audi'];  
  return (  
    <>  
    <h1>Cars in the garage</h1>  
    <ul>  
      {cars.map((car, index) => <Car key={index} brand={car.brand} />)}  
    </ul>  
    </>  
  );  
}
```

React Lists: Keys

- Das Verwenden des Array-Index als Key wird nicht empfohlen, weil es Probleme verursachen kann, wenn sich die Reihenfolge der Elemente im Array ändert
- Wenn man beispielsweise ein Element aus der Liste entfernt oder hinzufügt, wird dies die Indizes der verbleibenden Elemente ändern
- Dadurch werden die Keys der Elemente auf der Benutzeroberfläche ebenfalls geändert, was dazu führen kann, dass React die Elemente neu rendert

React Hooks

- Hooks wurden zu React in Version 16.8 hinzugefügt
- Explizit für Functional Components
- Mit Hooks können wir auf React-Funktionen wie einem State und Lebenszyklusmethoden zugreifen

React Hooks: `useState`

- Die React `useState` - Hook ermöglicht es, den State einer Functional Component zu verfolgen
- Der „State“ ermöglicht das Speichern und Ändern von Daten innerhalb einer Komponente.
- Diese Daten können während der Lebensdauer der Komponente verändert werden.
- Die `useState` - Hook kann verwendet werden, um Strings, Zahlen, Booleans, Arrays, Objekte und jede Kombination davon speichern

React Hooks: useState

```
import React, { useState } from 'react';

function Counter() {
  // Verwendung von useState für den Zustand
  const [count, setCount] = useState(0);

  const increment = () => {
    // Ändern des Zustands mit setCount
    setCount(count + 1);
  };

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={increment}>Increment</button>
    </div>
  );
}
```

React Hooks: useState – Wieso verwenden wir keine Variable?

```
import React from "react";

function Counter() {
  // Verwendung einer normalen Variable statt useState
  let count = 0;

  const increment = () => {
    count = count + 1;
  };

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={increment}>Increment</button>
    </div>
  );
}
```


React Hooks: useState – Wieso verwenden wir keine Variable?

In React werden Functional Components immer dann erneut gerendert, wenn sich ihr State ändert!

- **Zustandserhaltung:** Mit useState können wir den Zustand unserer Komponente verwalten und ihn zwischen den Rendervorgängen erhalten.
- **Reaktivität:** Durch die Verwendung von Hooks wie useState teilen wir React mit, dass sich der Zustand der Komponente geändert hat und ein erneutes Rendering erforderlich ist.

React Hooks: useState

Wir können nun auf eine State-Variable überall in unsere Komponente zugreifen:

Funktion zur
Aktualisierung des States

```
const [color, setColor] = useState("");
```

Der aktuelle State

React Hooks: useState - Objekte

- Wenn der State aktualisiert wird, wird der gesamte State

überschrieben

- Was, wenn man nur ein Attribut aktualisieren möchte

(z.B. die Farbe unseres Autos)?

- Würde man nur `setCar({color: "blue"})`

aufrufen, würden `brand`, `model` und `year` entfernt

werden

```
const [car, setCar] = useState({  
  brand: "Ford",  
  model: "Mustang",  
  year: "1964",  
  color: "red"  
});
```

React Hooks: useState - Objekte

- Stattdessen kann ein Objekt so aktualisiert werden:

```
const [car, setCar] = useState({
  brand: "Ford",
  model: "Mustang",
  year: "1964",
  color: "red"
});

const updateColor = () => {
  setCar({
    ...car, color: "blue"
  });
}

return (
  <>
    <button type="button" onClick={updateColor} >Blue</button>
  </>
)
```

React Hooks: useState - Objekte

- Hat man ein Array von Objekten und möchte ein bestimmtes Objekt mit einer bestimmten ID aktualisieren kann man dies mit einer `.map()` - Funktion

```
const [data, setData] = useState([
  {
    id: 1,
    name: "Nils",
  },
  {
    id: 2,
    name: "Max",
  },
]);
const updateData = (id, update) => {
  const updatedData = data.map((item) => {
    if (item.id === id) {
      return { ...item, ...update };
    }
    return item;
  });
  setData(updatedData);
};

updateData(1, { name: "Thomas" });
```

React Hooks: useEffect

- Die useEffect - Hook ermöglicht die Ausführung von „side effects“ in einer React-Komponente, wie z.B. das Laden von Daten von einem Server
- Die Hook wird in der Functional Component verwendet und nimmt zwei Argumente entgegen: (1) eine Funktion, die ausgeführt werden soll, und (2) ein Array von Dependencies

React Hooks: useEffect

1. Keine Dependencies übergeben

```
useEffect(() => {  
    // Runs on every render  
});
```

2. Ein leeres Array

```
useEffect(() => {  
    // Runs only on the first render  
}, []);
```

3. Props / State Werte

```
useEffect(() => {  
    // Runs on the first render and any time any dependency value changes  
}, [dependency]);
```

React Hooks: useEffect – Keine Dependencies

setTimeout führt eine Funktion (1. Argument) aus, wenn die Zeit abgelaufen ist (2. Argument, in Millisekunden)

```
import { useState, useEffect } from "react";

function Timer() {
  const [count, setCount] = useState(0);
  useEffect(() => {
    {
      setTimeout(() => {
        setCount(count + 1);
      }, 1000);
    }
  });
  return <h1>I've rendered {count} times!</h1>;
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Timer />);
```

(1) Es wird weiter gezählt, obwohl es nur einmal zählen sollte

(2) useEffect wird bei jedem Rendering ausgeführt. Das heißt, wenn sich count ändert, wird ein Re-rendering durchgeführt, das dann erneut die useEffect – Funktion ausführt

React Hooks: useEffect – Leeres Array

```
import { useState, useEffect } from "react";
import ReactDOM from "react-dom/client";
function Timer() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    setTimeout(() => {
      setCount((count) => count + 1);
    }, 1000);
  }, []); // <- empty brackets
  return <h1>I've rendered {count} times!</h1>;
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Timer />);
```

} Wird nur beim ersten Rendering ausgeführt

React Hooks: useEffect – State als Dependency

```
import { useState, useEffect } from "react";
import ReactDOM from "react-dom/client";

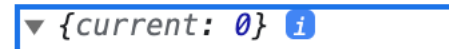
function Counter() {
  const [count, setCount] = useState(0);
  const [calculation, setCalculation] = useState(0);
  useEffect(() => {
    setCalculation(() => count * 2);
  }, [count]); // <- add the count variable here
  return (
    <>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>add</button>
      <p>Calculation: {calculation}</p>
    </>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Counter />);
```

useEffect-Hook, die von dem State `count` abhängig ist. Wenn die State-Variable aktualisiert wird, wird der Effekt erneut ausgeführt

React Hooks: useRef

- `useRef` ist eine Hook, die eine veränderbare Referenz auf ein DOM-Element oder einen Wert speichert
- Es ermöglicht den Zugriff auf das DOM-Element oder einen Wert, ohne dass ein erneutes Rendern ausgelöst wird.
- Die `useRef` - Hook gibt ein Objekt zurück, das eine `.current` Eigenschaft hat, die den aktuellen Wert enthält



```
▼ {current: 0} ⓘ
```

React Hooks: `useRef`

- Wenn wir mit der `useState` - Hook versuchen würden zu zählen, wie oft eine Komponente erneut gerendert wird, würden wir in eine Endlosschleife geraten, da diese Hook selbst ein erneutes rendering verursachen würde
- Um dies zu vermeiden, kann die `useRef` - Hook verwendet werden

React Hooks: useRef

Wenn wir versuchen würden, mit der `useState` - Hook zu zählen:

```
import { useState, useEffect } from "react";
import ReactDOM from "react-dom/client";
function Timer() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    setCount(count + 1);
  });
  return <h1>I've rendered {count} times!</h1>;
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Timer />);
```

React Hooks: useRef

Wenn wir versuchen, mit der `useRef` - Hook zu zählen:

```
function App() {  
  const [value, setValue] = useState(0);  
  const count = useRef(0);  
  useEffect(() => {  
    count.current = count.current + 1;  
  });  
  
  return (  
    <div>  
      <p>{value}</p>  
      <button onClick={() => setValue(value + 1)}>Click me</button>  
      <h1>Render Count: {count.current}</h1>  
    </div>  
  );  
}
```



Die `useRef` – Hook löst
kein erneutes
Rendering aus

React Hooks: `useRef`

- `useRef` ermöglicht außerdem den Zugriff auf DOM-Elemente, indem es ein Referenzobjekt zur Verfügung stellt, das auf das Element verweist
- Durch Hinzufügen des `ref` – Attributs bei einem Element kann direkt auf ein Element zugegriffen werden

React Hooks: useRef

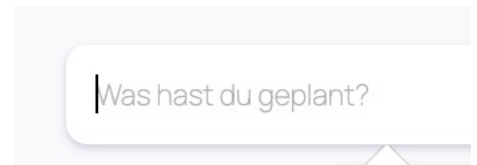
Beispiel:

```
import { useRef } from "react";  
import ReactDOM from "react-dom/client";
```

```
function App() {  
  const inputElement = useRef();  
  const focusInput = () => {  
    inputElement.current.focus();  
  };  
  return (  
    <>  
      <input type="text" ref={inputElement} />  
      <button onClick={focusInput}>Focus Input</button>  
    </>  
  );  
}
```

} `inputElement.current`
entspricht dem DOM-Objekt

```
const root = ReactDOM.createRoot(document.getElementById('root'));  
root.render(<App />);
```



React Hooks: useContext

- `useContext` ermöglicht es, einen State global zu nutzen
- Er kann zusammen mit der `useState` Hook verwendet werden, um einen State zwischen tief verschachtelten Komponenten einfacher zu teilen als nur unter Verwendung von `useState`
- Der zu verwaltende State (bereitgestellt über `useState`) sollte von der höchsten übergeordneten Komponente im Stapel, die Zugriff auf den State benötigt gehalten werden

React Hooks: useContext

```
function Component1() {
  const [user, setUser] = useState("Jesse Hall");
  return (
    <>
      <h1>`Hello ${user}!`</h1>
      <Component2 user={user} />
    </>
  );
}

function Component2({ user }) {
  return (
    <>
      <h1>Component 2</h1>
      <Component3 user={user} />
    </>
  );
}

function Component3({ user }) {
  return (
    <>
      <h1>Component 3</h1>
      <Component4 user={user} />
    </>
  );
}
```

```
function Component4({ user }) {
  return (
    <>
      <h1>Component 4</h1>
      <Component5 user={user} />
    </>
  );
}

function Component5({ user }) {
  return (
    <>
      <h1>Component 5</h1>
      <h2>`Hello ${user} again!`</h2>
    </>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Component1 />);
```

React Hooks: useContext

- Zur vorherigen Folie: Wir haben viele verschachtelte Komponenten
- Die Komponenten am oberen und unteren Ende des Stapels benötigen Zugriff auf den State
- Um dies ohne `useContext` zu tun, müssten wir den Zustand als "props" durch jede verschachtelte Komponente weitergeben
- Dies wird als "prop drilling" bezeichnet und sollte möglichst vermieden werden

React Hooks: useContext

- Um den State nicht durch jede verschachtelte Komponente durchreichen zu müssen mithilfe von Props kann `useContext` verwendet werden
- Um einen Kontext zu erstellen, muss `createContext` importiert und initialisiert werden
- Verwendung des Context-Providers, um den Baum der Komponenten zu umhüllen, die den State-Context benötigen
- Beispiel: Nächste Folie

React Hooks: useContext

```
import { useState, createContext, useContext } from "react";
import ReactDOM from "react-dom/client";

const UserContext = createContext();

function Component1() {
  const [user, setUser] = useState("Jesse Hall");
  return (
    <UserContext.Provider value={user}>
      <h1>`Hello ${user}!`</h1>
      <Component2 />
    </UserContext.Provider>
  );
}

function Component2() {
  return (
    <>
      <h1>Component 2</h1>
      <Component3 />
    </>
  );
}
```

```
function Component3() {
  return (
    <>
      <h1>Component 3</h1>
      <Component4 />
    </>
  );
}

function Component4() {
  return (
    <>
      <h1>Component 4</h1>
      <Component5 />
    </>
  );
}

function Component5() {
  const user = useContext(UserContext);

  return (
    <>
      <h1>Component 5</h1>
      <h2>`Hello ${user} again!`</h2>
    </>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Component1 />);
```

React Hooks: Custom Hooks

- Benutzerdefinierte Hooks, die es ermöglichen, wiederverwendbare Logik in React-Komponenten zu teilen
- Sollten mit "use" beginnen und können andere bereits vorhandene Hooks wie `useState` oder `useEffect` verwenden

React Hooks: Custom Hooks

```
import { useState, useEffect } from "react";
import ReactDOM from "react-dom/client";
const Home = () => {
  const [data, setData] = useState(null);
  useEffect(() => {
    fetch("https://jsonplaceholder.typicode.com/todos")
      .then((res) => res.json())
      .then((data) => setData(data));
  }, []);
  return (
    <>
      {data &&
        data.map((item) => {
          return <p key={item.id}>{item.title}</p>;
        })
      }
    </>
  );
};

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Home />);
```

React Hooks: Custom Hooks

- Wenn man Komponentenlogik hat, die von mehreren Komponenten verwendet werden muss, kann diese Logik in eine benutzerdefinierte Hook überführt werden
- Die benutzerdefinierte Hook sollte dann in einer eigenen Datei sein:

```
import { useState, useEffect } from "react";
const useFetch = (url) => {
  const [data, setData] = useState(null);
  useEffect(() => {
    fetch(url)
      .then((res) => res.json())
      .then((data) => setData(data));
  }, [url]);

  return [data];
};

export default useFetch;
```


React Hooks: Custom Hooks

Schließlich kann die benutzerdefinierte Hook geladen werden:

```
import ReactDOM from "react-dom/client";
import useFetch from "../useFetch";

const Home = () => {
  const [data] = useFetch("https://jsonplaceholder.typicode.com/todos");
  return (
    <>
      {data &&
        data.map((item) => {
          return <p key={item.id}>{item.title}</p>;
        })
      }
    </>
  );
};

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Home />);
```

React Hooks: Custom Hooks

- Wir haben eine neue Datei mit dem Namen `useFetch.js` erstellt, die eine Funktion namens `useFetch` enthält, die die gesamte Logik enthält, die zum Abrufen unserer Daten erforderlich ist
- `url`-Variable, die an den benutzerdefinierten Hook übergeben werden kann
- In `index.js` importieren wir unsere `useFetch` - Hook und verwenden sie wie jede andere Hook
- Nun können wir die benutzerdefinierte Hook in jeder Komponente wiederverwenden, um Daten von jeder URL abzurufen

React Forms

- Änderungen können durch Hinzufügen von event-Handlern bei einem `onChange` - Attribut gesteuert werden
- Wir können die `useState` - Hook verwenden, um den aktuellen Stand zu speichern

React Forms: Input-Feld

```
import { useState } from 'react';
import ReactDOM from 'react-dom/client';

function MyForm() {
  const [name, setName] = useState("");
  return (
    <form>
      <label>Enter your name:</label>
      <input
        type="text"
        onChange={ (event) => setName(event.target.value) }
      />
    </form>
  )
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<MyForm />);
```

React Forms: Input-Feld - value

```
import { useState } from "react";
import ReactDOM from "react-dom";

function MyForm() {
  const [name, setName] = useState("");

  const handleInputChange = (event) => {
    setName(event.target.value);
  };

  const handleClearClick = () => {
    setName("");
  };

  return (
    <form>
      <label>Enter your name:</label>
      <input type="text" value={name} onChange={handleInputChange} />
      <button type="button" onClick={handleClearClick}>Clear</button>
    </form>
  );
}

const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(<MyForm />);
```

React Forms: Submitting Forms

Das Submit-Event kann abgefangen werden, indem man einen Event-Handler in das `onSubmit` - Attribut für das `<form>` - Element hinzufügt

```
function MyForm() {  
  const [name, setName] = useState("");  
  const handleSubmit = (event) => {  
    event.preventDefault();  
    alert(`The name you entered was: ${name}`)  
  }  
  return (  
    <form onSubmit={handleSubmit}>  
      <label>Enter your name:</label>  
      <input  
        type="text"  
        value={name}  
        onChange={ (e) => setName(e.target.value) }  
      />  
      <input type="submit" />  
    </form>  
  )  
}
```

Page-Routing: Navigation zwischen verschiedenen Seiten oder Ansichten in einer Webanwendung.

React Router

- `create-react-app` umfasst standardmäßig keine Schnittstelle für Page-Routing
- Der React Router ist dafür die beliebteste Lösung
- Um den React Router zu einem Projekt hinzuzufügen, führt man diesen Befehl im Terminal im

Hauptverzeichnis der Anwendung aus:

```
npm i react-router-dom
```

React Router: Demo

```
import React from "react";  
import ReactDOM from "react-dom/client";  
import { BrowserRouter, Route, Routes } from "react-router-dom";
```



Komponenten für den
Router importieren

```
const Home = () => <h1>Home</h1>;  
const About = () => <h1>About</h1>;  
const Contact = () => <h1>Contact</h1>;  
  
const root = ReactDOM.createRoot(document.getElementById("root"));  
root.render(  
  <BrowserRouter>  
    <Routes>  
      <Route path="/" element={<Home />} />  
      <Route path="/about" element={<About />} />  
      <Route path="/contact" element={<Contact />} />  
    </Routes>  
  </BrowserRouter>  
) ;
```


React Router: Erklärungen zu dem komplexen Beispiel (Code in GRIPS)

- Wir wrappen den Inhalt zuerst in `<BrowserRouter>`.
- Dann definieren wir unsere `<Routes>`
- `<Route>`s können verschachtelt werden, die erste `<Route>` hat den Pfad `/` und stellt die `<Layout>` - Komponente dar
- Die verschachtelten `<Route>`s erben und ergänzen die übergeordnete Route
- So wird der Pfad `"blogs"` mit dem übergeordneten Pfad kombiniert und zu `"/blogs"`
- Die Komponente der Komponente `<Home>` hat keinen Pfad, sondern ein `index` – Attribut, wodurch diese Route als Standardroute für die übergeordnete Route festgelegt wird, die `"/` ist
- Wenn man den Pfad auf `"*"` setzen, fungiert er als Auffangbecken für alle undefinierten URLs (ideal für eine 404-Error Page)

React Router: Erklärungen zu komplexerem Beispiel (Code in GRIPS)

- Die Layout-Komponente hat `<Outlet>` - und `<Link>` - Elemente
- Das `<Outlet>` zeigt den Inhalt der aktuell ausgewählte Route an
- `<Link>` erlaubt das Navigieren zu einer anderen Route

React Komponenten mit CSS stylen

- Inline Styling

```
<h1 style={{color: "red", backgroundColor: "black"}}>Hello Style!</h1>
```

Da das Inline-CSS in einem JavaScript-Objekt geschrieben wird, müssen Eigenschaften mit Bindestrich-Trennzeichen, wie z.B. background-color, in Camel-Case-Syntax geschrieben werden: backgroundColor

- CSS Stylesheet: CSS-Deklarationen können in eine separate CSS-Datei gespeichert werden und importiert werden

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './App.css';
const Header = () => {
  return (
    <>
      <h1>Hello Style!</h1>
      <p>Add a little style!</p>
    </>
  );
}
```

React Conventions

- PascalCase für Komponentennamen und Dateinamen von Komponenten
- `src/components` für Komponenten

React Production Build

- In React gibt es einen Build-Prozess, der dafür sorgt, dass der Code in eine produktionsfähige Form gebracht wird, dieser kann mithilfe von `npm run build` gestartet werden
- Der Build-Prozess in React erzeugt statische Dateien (im `/build` - Ordner).
- Das bedeutet, dass der Code in eine Form gebracht wird, die von einem Webserver ausgeliefert werden kann, ohne dass weitere Serverseitige Skripte ausgeführt werden müssen

React StrictMode

```
<React.StrictMode>  
  <App />  
</React.StrictMode >
```

- Der Strict Mode sorgt dafür, dass potenziell unsichere Praktiken und Fehlerquellen im Code erkannt werden. So können Sicherheitslücken vermieden und die Gesamtsicherheit der Anwendung erhöht werden
- Prüfungen im Strict Mode werden nur im Entwicklungsmodus durchgeführt, sie haben keine Auswirkungen auf den Produktions-Build
- Beispiele: Identifikation veralteter oder nicht empfohlener React-APIs, Identifikation langsamer Renderings bzw. unnötiger Re-Renderings
- Liste der Funktionalitäten des Strict Modes: <https://reactjs.org/docs/strict-mode.html>

React StrictMode

```
<React.StrictMode>  
  <App />  
</React.StrictMode >
```

- **Seit React 18:** StrictMode rendert Komponenten zweimal (in der Entwicklung, aber nicht in der Produktion) beim Mounting (erstes Erscheinen in der Benutzeroberfläche), um Probleme im Code zu erkennen

React StrictMode: useEffect

1. Keine Dependencies übergeben

```
useEffect(() => {  
    // Runs on every render  
});
```

2. Ein leeres Array

```
useEffect(() => {  
    // Runs only on the first render  
}, []);
```

3. Props / State Werte

```
useEffect(() => {  
    // Runs on the first render and any time any dependency value changes  
}, [dependency]);
```


Externe Bibilotheken

- Phosphor Icons (React Library verfügbar):
<https://github.com/phosphor-icons/homepage#phosphor-icons>
- React Toastify: <https://www.npmjs.com/package/react-toastify>
- React Beautiful DnD (eher komplex): <https://github.com/atlassian/react-beautiful-dnd>
- JS Cookie (nicht nur für React): <https://www.npmjs.com/package/js-cookie>
- Draft.js: <https://draftjs.org/>
- Radix: <https://www.radix-ui.com/docs/primitives/overview/getting-started>

Nachtrag: CORS

- CORS steht für **C**ross-**O**rin **R**esource **S**haring
- CORS ist ein Browser-Mechanismus, der den kontrollierten Zugriff auf Ressourcen außerhalb einer gegebenen Domäne ermöglicht
- CORS ermöglicht es, dass eine Website in einem Browser Ressourcen von einer anderen Domain anfordern kann

Nachtrag: CORS

- Standardmäßig erlauben Webbrowser keine Cross-Origin-Anfragen. Das bedeutet, dass eine Website nur auf Ressourcen zugreifen kann, die sich auf derselben Domain befinden.
- Beim Senden einer Cross-Origin-Anfrage prüft der Browser, ob der Server die entsprechenden CORS-Header in der Response enthält.
- Wert von "Access-Control-Allow-Origin" muss mit dem Wert der aktuellen Domain übereinstimmen oder einen genereller Wert wie "*" (alle Domains) enthalten

Nachtrag: CORS

```
{  
  "Access-Control-Allow-Origin": "http://localhost:3000",  
  "Access-Control-Allow-Methods": "GET, POST, OPTIONS",  
}
```

- Standardmäßig erlauben Webbrowser keine Cross-Origin-Anfragen. Das bedeutet, dass eine Website nur auf Ressourcen zugreifen kann, die sich auf derselben Domain befinden.

Weiterführende Literatur / Quellen

- Offizielle Dokumentation zu React:
<https://reactjs.org/docs/getting-started.html>
- Ausführliches React Tutorial mit anschaulichen Beispielen:
<https://www.w3schools.com/react>
- Thinking in React:
<https://react.dev/learn/thinking-in-react>
- Mehr zu React naming conventions:
<https://www.upbeatcode.com/react/react-naming-conventions/>
- Popularität von React:
<https://trends.google.com/trends/explore?date=all&q=React,Vue%20js,Angular>