08 React.js

Multimedia Engineering (PS siehe 36609b)

Nils Hellwig, M.Sc.
Lehrstuhl für Medieninformatik
FAKULTÄT FÜR INFORMATIK UND DATA SCIENCE

Universität Regensburg

Überblick Themen

- Einführung in React.js
- JSX Grundlagen
- Functional Components
- State & React Hooks
- Formulare in React

Blogspace

Logout Nils

Add Post

Title

Description

Datei auswählen Keine ausgewählt

Add Post

Latest Posts

Max Mustermann
@max

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 (kommerziell) genutzt werden und ist Open-Source

Historie und Entwicklung von React

- Veröffentlichung im Juli 2013 (V0.3.0)
- Aktuelle Version: 19.1 (März 2025)
- Es wird von Meta und einer Gemeinschaft von einzelnen Entwicklern und Unternehmen gepflegt
- React wird von Tech-Giganten wie Instagram, Netflix, Dropbox, Discord verwendet

Hacker News is a social news website focused on computer science, entrepreneurship, and technology.

Historie und Entwicklung von React

Hacker News
new | past | comments | ask | show | jobs | submit

React, a JavaScript library for building user interfaces (facebook.github.io)
287 points by friggeri on May 29, 2013 | hide | past | favorite | 112 comments

dpham on May 29, 2013 | prev | next [–]
Do templates have to live inside the application logic? If so, this is a really poor decision.

mavdi on May 30, 2013 | prev | next [–]
This is terrible, so did we really not learn anything from PHP days? Are we seriously going to mix markup and logic again?

darkchasma on May 29, 2013 | prev | next [–]
I'm confused, every example is 100 times the code I would write using straight html... Is there a benefit I'm missing?

React
Suchbegriff

Vue
Suchbegriff

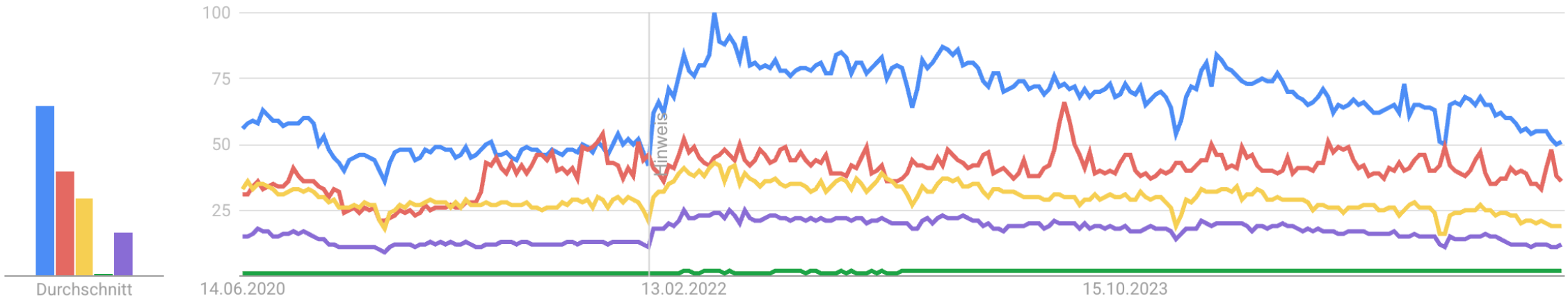
Angular
Suchbegriff

Svelte
Suchbegriff

Laravel
Suchbegriff

Weltweit | Letzte 5 Jahre | Alle Kategorien | Websuche

Interesse im zeitlichen Verlauf



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 unnötige Änderungen an der virtuellen DOM relativ kostengünstig vornehmen
- 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
- Der *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 einer HTML-Datei 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, einschließlich React und React DOM

Universität Regensburg

Möglichkeiten zur Erstellung eines React-Projekts

- **create-react-app (gilt mittlerweile als „veraltet“):**
  Automatisiertes Setup mit Webpack. Kommt mit allem Nötigen für ein React-Projekt (JSX, CSS, Tests).
  Nachteile: Langsamere Build-Zeiten, veralteter Workflow.
- **Vite:**
  Modernes, schnelles Build-Tool für React. Minimaler Konfigurationsaufwand. Flexible Integration mit TypeScript, CSS-Frameworks usw.
- **Next.js:**
  Framework für React mit Server-Side Rendering (SSR) und Static Site Generation (SSG).
  Integriertes Routing ohne zusätzliche Bibliotheken. Ideal für komplexere Projekte.

Quelle: https://vite.dev/guide/

React Environment

- Wenn Node.js installiert ist, kann mithilfe von Vite eine moderne React-Anwendung erstellt werden.

  npm create ermöglicht die Erstellung von Projekt-Templates, u.a. um eine React-Anwendung namens my-react-app zu erstellen:

  ```
  $ npm create vite@latest <Name der neuen Anwendung> -- --template react
  ```

- Danach ins Projektverzeichnis wechseln und Abhängigkeiten installieren:

  ```
  $ cd <Name der neuen Anwendung>  $ npm install
  ```

- Die Anwendung kann dann über einen lokalen Development-Server (standardmäßig unter localhost:5173) gestartet werden mit: `$ npm run dev`

Das erste -- trennt **npm**-Argumente von den Argumenten für **Vite**
--template react ist ein **Vite-internes Argument**, das das React-Template auswählt.
Ohne das doppelte -- würde npm den Parameter falsch interpretieren.

React Environment

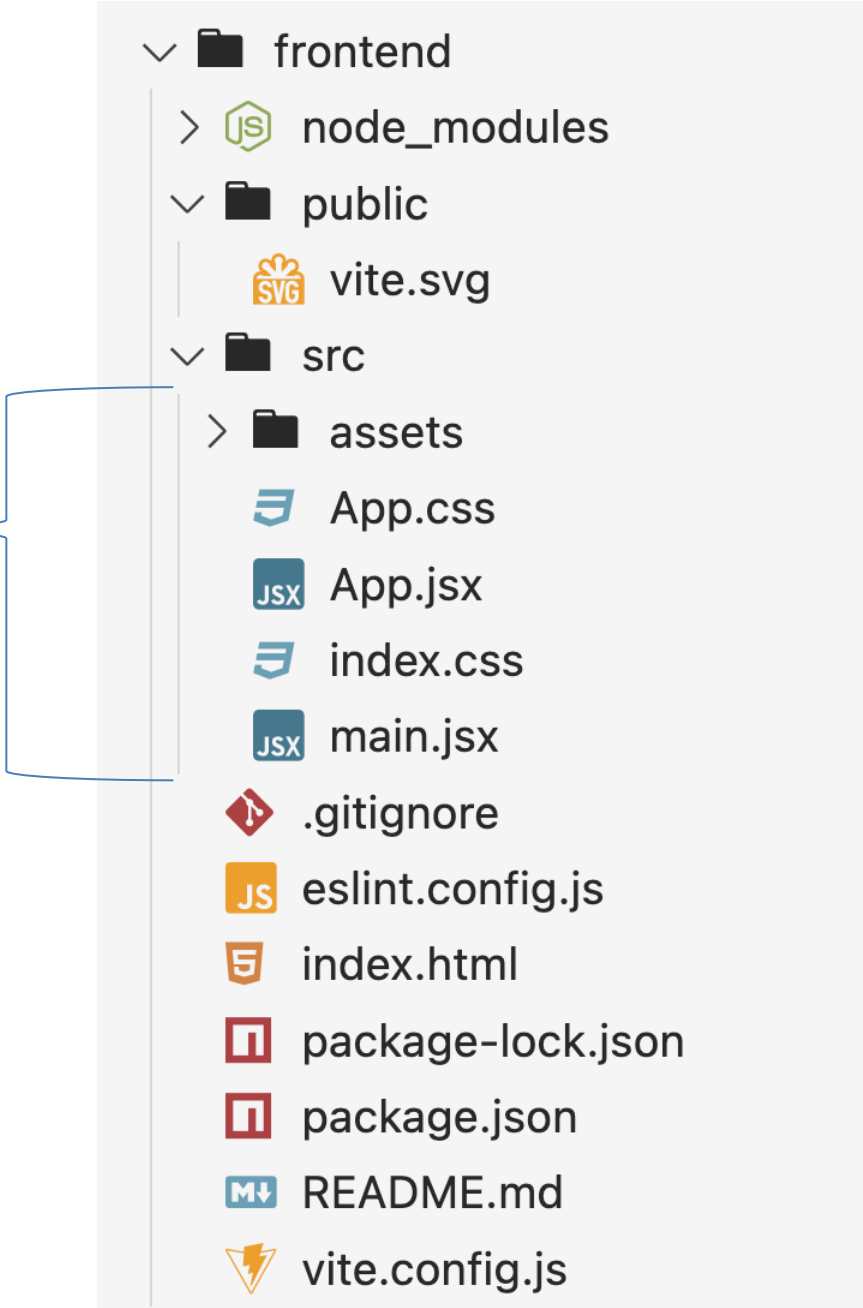
- Entwickelt man mit npm run dev, läuft lokal ein Web-Server, man hat Zugriff auf ein paar Entwicklungstools – das ist gut zum Testen.
- Aber für das Veröffentlichen im Internet brauchst du:

  (1) schnellen, kompakten Code

  (2) keine Entwickler-Tools (z.B. Tests etc.)
- Ein Build-Prozess (`npm run build`) erstellt fertige HTML-, JS- und CSS-Dateien im Ordner `dist/`

React Environment

- **node_modules/** – Enthält alle installierten npm-Pakete
- **public/** – Statische Dateien, direkt im Browser aufrufbar
- **vite.svg** – Beispielbild im public-Ordner
- **src/** – Quellcode der Anwendung (React-Komponenten, CSS etc.)
- **assets/** – Bilder, Medien und andere statische Assets
- **App.jsx** – Haupt-React-Komponente der Anwendung
- **App.css** – Styles speziell für App.jsx
- **index.css** – Globale CSS-Definitionen
- **main.jsx** – Einstiegspunkt, verbindet React mit HTML
- **.gitignore** – Legt fest, welche Dateien Git ignorieren soll
- **eslint.config.js** – Regeln zur Codequalität (Linting)
- **index.html** – HTML-Vorlage mit
- **package.json** – Projektinfo, Abhängigkeiten und npm-Skripte
- **README.md** – Projektbeschreibung und Hinweise
- **vite.config.js** – Vite-spezifische Einstellungen und Plugins



React Environment

frontend > index.html > ...

```html
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <link rel="icon" type="image/svg+xml" href="/vite.svg" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Vite + React</title>
  </head>
  <body>
    <div id="root"></div>// #root
    <script type="module" src="/src/main.jsx"></script>
  </body>
</html>
```

- Innerhalb des Ordners befindet sich die Datei `index.html`
- `index.html` enthält das grundlegende HTML-Gerüst der Anwendung und definiert das Ziel-Element mit der ID `"root"`, in das die React Anwendung gerendert wird

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 `createElement()` und/oder `appendChild()`-Methoden zu platzieren
- Beispiel:

```
const myElement = <h1>I Love JSX!</h1>;

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

root.render(myElement);
```

React JSX: Expressions

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

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

- Sollen wie hier zwei Absätze geschrieben werden, muss man sie in ein übergeordnetes Element (hier – 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

if-Statements außerhalb von JSX-Code	Ternary Expressions

`const x = 5;`
`let text = "Goodbye";`
`if (x < 10) {`
`  text = "Hello";`
`}`
`const myElement = <h1>{text}</h1>;`
	`const x = 5;`
	`const myElement = <h1>{(x) < 10 ? "Hello" : "Goodbye"}</h1>;`

React Components

- React-Komponenten sind wie Funktionen, die HTML-Elemente zurückgeben
- Komponenten sind unabhängige und wiederverwendbare Codestücke
- Es gibt zwei Arten von Komponenten: Class Components und Functional Components

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 erstellt oder entfernt wird

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 HTML zurück und verhält sich ähnlich wie eine Class Component.

React Components: Functional Components

- Beispiel:

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

React Components aus einer anderen Datei laden

MyComponent.jsx

```
const MyComponent = () => {
    return <div>Hello World!</div>;
};
export default MyComponent;
```

App.jsx

```
import React from 'react';
import MyComponent from './MyComponent';

function App() {
  return (
    <div>
      <MyComponent />
    </div>
  );
}

export default App;
```

React Props (Properties)

- Props sind Argumente bzw. Eigenschaften, die an React-Komponenten übergeben werden können.
- Props werden als **Objekt** an die Komponente übergeben, mit Schlüssel-Wert-Paaren.
- Zugriff über Methoden-Variable oder Destructing {…}
- Beispiel:

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

Destructuring *ist eine Kurzschreibweise in JavaScript, mit der man Werte aus Objekten oder Arrays direkt in einzelne Variablen extrahiert.*

Destructing

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

React Props (Properties)

- Mithilfe von Functional Components können wir dann unsere Anwendung bauen.

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

function Garage() {
  return (
    <>
      <h1>Who lives in my garage?</h1>
      <Car brand="Ford" />
    </>
  );
}
```

React Props (Properties)

- Wird eine Variable übergeben, sollten geschweifte Klammern { } verwendet werden

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

function Garage() {
  const carName = "Ford";
  return (
   <div>
     <h1>Who lives in my garage?</h1>
     <Car brand={carName} />
   </div>
  );
}
```

React Conditional Rendering: if

```
function Weather({isSunny}) {
      if (isSunny) {
            return <SunnyDay/>;
      }
      return <RainyDay/>;
}
```

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>}
      </>
    );
}

Aufruf:
const cars = ['Ford', 'BMW', 'Audi'];
<Garage cars={cars} />
```

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

React Conditional Rendering: Ternary Operator

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

<MissedGoal/>

<MadeGoal/>

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

Aufruf:

```
root.render(<GoalStausItem isGoal={false} />);
```

React Lists

- In React werden Listen mit einer Art von Schleife gerendert
- 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>
        </>
    );
}
```

React Lists: Keys

- Wenn man den Code ausführt, funktioniert er, man erhält jedoch eine Warnung, dass für die Listenelemente kein „key“ vorhanden ist
- Schlüssel ermöglichen es React, Elemente zu verfolgen
- Auf diese Weise wird, wenn ein Element aktualisiert oder entfernt wird, nur dieses Element neu gerendert und nicht die gesamte Liste
- Keys für jedes Element innerhalb einer Liste **müssen** eindeutig sein
- Es ist jedoch erlaubt, dass Keys in verschiedenen Listen bzw. innerhalb einer Anwendung mehrfach vorkommen

React Lists: Keys

Im Allgemeinen sollte der Schlüssel eine eindeutige ID sein, die jedem Element zugewiesen wird

```
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 kann 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 Events

- Genau wie HTML-DOM-Ereignisse kann React Aktionen basierend auf Benutzerereignissen durchführen
- React hat die gleichen Ereignisse wie HTML: click, change, 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 AddProductToList() {
    const addProductToList = () => {
        alert("Product added!");
    }

    return (
      <button onClick={addProductToList}>Add Product</button>
    );
}
```

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>
    );
}
```

React Events: Argument an einen Event-Handler übergeben

```
<button onClick={sayHello("Hello")}>Add Product</button>
```

So wird **wird `sayHello("Hello")` sofort beim Rendern der Komponente ausgeführt** – und **nicht erst beim Klick** auf den Button. Der Rückgabewert des Ausdrucks wäre keine Funktion!

React Hooks

- Hooks wurden zu React in Version 16.8 hinzugefügt
- Hooks ermöglichen es Functional Components, auf einen State und andere React-Funktionen zuzugreifen
- Mit Hooks können wir auf React-Funktionen wie einem State und Lebenszyklusmethoden zugreifen

React Hooks: useState

- Die React `useState` - Hook ermöglicht es uns, den Zustand einer Functional Component zu verfolgen
- Der State bezieht sich im Allgemeinen auf Daten oder Eigenschaften, die in einer Anwendung verfolgt werden müssen
- Um die `useState` - Hook zu verwenden, müssen wir diese zunächst in unsere Komponente importieren
- Die `useState` - Hook kann verwendet werden, um Strings, Zahlen, Booleans, Arrays, Objekte und jede Kombination davon speichern

React Hooks: useState

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

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

function FavoriteColor() {
    const [color, setColor] = useState("red");
    return <h1>My favorite color is {color}!</h1>
}
```

Wir initialisieren unseren Zustand durch den Aufruf von useState in unserer Funktionskomponente

```
Andere (unübliche) Schreibweise:
const stateArray = useState("red");
const color = stateArray[0];
const setColor = stateArray[1];
```

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

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

function Counter() {
    const [counter, setCounter] = useState(0);
    return (
        <>
            <h1>Counter: {counter}!</h1>
            <button type="button" onClick={() => setCounter(counter+1)}>Add +1</button>
        </>
    )
}
```

React Hooks: useState – wieso nicht eine “normale“ JS-Variable?

```
function Counter() {
  let counter = 0;

  return (
    <>
      <h1>Counter: {counter}!</h1>
      <button type="button" onClick={() => counter++}>Add +1</button>
    </>
 );
}
```

- Bei jedem Klick auf den Button wird counter++ zwar ausgeführt, **aber React rendert die Komponente nicht neu**.
- Das heißt: Der Wert auf dem Bildschirm **ändert sich nicht**, obwohl sich die Variable im Hintergrund verändert.
- Bei jedem Render wird let counter = 0; wieder neu gesetzt → der Zähler springt nie wirklich hoch.

React Hooks: useState - Objekte

- Wenn der State aktualisiert wird, wird der gesamte Status 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(previousState => {
        return { ...previousState, color: "blue" }
    });
}

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

React Hooks: useEffect

- **Side Effects ausführen:** `useEffect` wird verwendet, um „Side Effects“ (z. B. Daten laden, Timer starten) nach dem ersten Rendern auszuführen.
- **Abhängigkeiten steuern die Ausführung:** Durch ein Abhängigkeitsarray ([ ]) kannst du festlegen, wann der Effekt ausgeführt wird – z. B. nur einmal, bei bestimmten Zustandsänderungen oder bei jedem Render.

React Hooks: useEffect

1. **Keine Dependencies übergeben**

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

2. **Ein leeres Array**

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

3. **State Werte**

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

React Hooks: useRef

1. Einsatzzweck: useRef kann verwendet werden, um direkt auf ein DOM-Element zuzugreifen – z. B. ein Input-Feld.

```
import { useRef } from "react";

function FocusInput() {
  const inputRef = useRef(null);

  const focusInput = () => {
     inputRef.current.focus();
  };


  return (
    <>
      <input ref={inputRef} type="text" />
      <button onClick={focusInput}>Fokus setzen</button>
    </>
  );
}
```

React Hooks: useRef

2. Einsatzzweck: Persistenter Wert ohne Re-Render – anders bei einer einer useState-Variable, die einen Re-Render bewirkt

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

function RenderCounter() {
   const [count, setCount] = useState(0);

   // useRef speichert einen Zähler, ohne einen Re-Render auszulösen
   const renderCount = useRef(1);

   useEffect(() => {
      // Erhöhe den Render-Zähler bei jedem Render
      renderCount.current += 1;
   });

   return (
    <>
      <h1>Count: {count}</h1>
      <button onClick={() => setCount(count + 1)}>+1</button>
      <p>Komponente wurde {renderCount.current} Mal gerendert.</p>
    </>
   );
}
```

React Hooks: useContext

- `useContext` ermöglicht es, einen State global zu nutzen
- Er kann zusammen mit dem `useState` Hook verwendet werden, um einen State zwischen tief verschachtelten Komponenten einfacher zu teilen als mit `useState` allein
- 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

- Zur Veranschaulichung auf der nächsten 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

```
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>
         </>
     );
}
```

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>
        </>
    );
}
```

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>;
                })}
        </>
    );
};
```

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 eigene 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:
           <input
              type="text"
              value={name}
              onChange={(e) => setName(e.target.value)}
           />
        </label>
     </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:
             <input
                 type="text"
                 value={name}
                 onChange={(e) => setName(e.target.value)}
             />
          </label>
          <input type="submit" />
       </form>
    )
}
```

React Router

- `vite/react` umfasst standardmäßig keine Schnittstelle für Page-Routing
- Der „React Router“ ist dafür die beliebteste Lösung
- Installation des Routers:

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

https://reactrouter.com/start/declarative/routing

React Router: Demo

```
import { useState } from "react";
import reactLogo from "./assets/react.svg";
import viteLogo from "/vite.svg";
import "./App.css";
import Home from "./components/Home";
import About from "./components/About";
import Contact from "./components/Contact";
import { Route, Routes } from "react-router-dom";

function App() {
  return (
   <div>
     <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
        <Route path="/contact" element={<Contact />} />
     </Routes>
   </div>
  );
}

export default App;
```

React Router: <Link/>

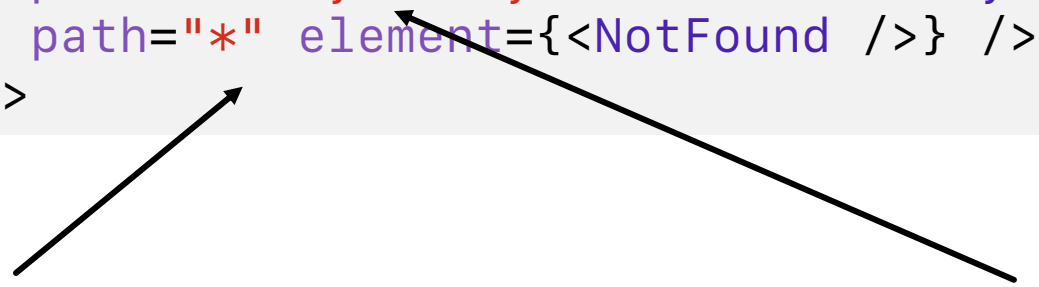
- Ersetzt <a>-Tags für clientseitige Navigation falls der Router genutzt wird.
- Link **verhindert das Neuladen** der Seite, was bei klassischen `<a href="...">` passiert.
- States bleiben erhalten wenn Seite gewechselt wird (siehe Beispiel auf GRIPS/GitHub)

```jsx
import { Link } from "react-router-dom";

function Navigation() {
  return (
   <nav>
     <Link to="/">Startseite</Link>
     <Link to="/kontakt">Kontakt</Link>
   </nav>
  );
}
```

React Router: 404 Page / Individueller Pfad

```
<Routes>
  <Route path="/" element={<Home />} />
  <Route path="/about" element={<About />} />
  <Route path="/contact" element={<Contact />} />
  <Route path="/city/:city" element={<City />} />
  <Route path="*" element={<NotFound />} />
</Routes>
```



Die path="*"-Route fängt **alle Pfade ab**, die nicht vorher definiert wurden.

„:“ definiert einen **dynamischen Pfad** in deiner React Router-Konfiguration. Sie wird aktiv, wenn ein Nutzer eine URL wie /city/berlin oder /city/muenchen aufruft.

React Komponenten mit CSS stylen

- Inline Styling

```
<h1 style={{color: "red"}}>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 './App.css';
const Header = () => {
    return (
       <>
        <h1>Hello Style!</h1>
        <p>Add a little style!.</p>
       </>
    );
}
```

React Conventions

- Pascal Case für Komponentennamen und Dateinamen von Komponenten
- camelCase für Funktionen (und deren Dateinamen)

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
- ⚠ **Wichtig zu wissen:** `useEffect` wird **falls der StrictMode verwendet wird** doppelt aufgerufen, um Fehler zu erkennen
- Liste der Funktionalitäten des Strict Modes: https://reactjs.org/docs/strict-mode.html

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
- 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