

EINFÜHRUNG IN REACT

Grundlagen der modernen Frontend-Entwicklung

PROGRAMM

1

30. April

1. Was ist React?
2. JSX und Komponenten

2

30. April

1. Komponentenlogik
2. CSS mit Tailwind

3

07. Mai

1. Conditional & List Rendering
2. Client-Side Rendering
3. Routing

4

07. Mai

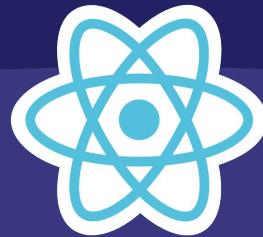
1. UI Testing mit Vitest & React Testing Library

MATERIALIEN

GitHub-Repository für:

- Beispiele
- Aufgaben
- Lösungen
- Folien

<https://github.com/MadiiW/ReactTutorial>

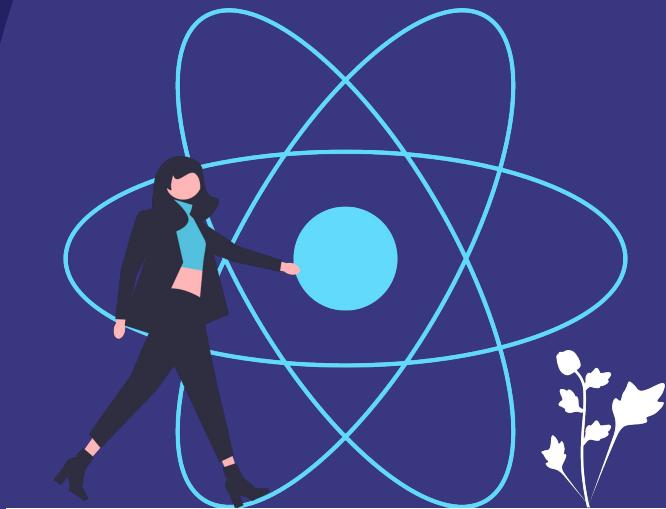


WAS IST REACT?

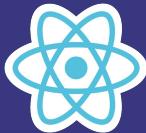


WAS IST REACT?

- JavaScript Bibliothek zur Erstellung interaktiver Benutzeroberflächen
- Entwickelt von Meta (Facebook) 2013
- Basiert auf wiederverwendbaren, zustandsbasierten Komponenten
- Besonders geeignet für Single Page Applications (SPAs)
- Hohe Performance durch Einsatz eines virtuellen DOM
- Entwicklung von Web-Apps und mobilen Anwendungen möglich



Quelle: <https://undraw.co/search/react>



WER BENUTZT REACT?

NETFLIX

airbnb

Uber

Meta

Quellen:

Netflix: https://upload.wikimedia.org/wikipedia/commons/thumb/0/08/Netflix_2015_logo.svg/2560px-Netflix_2015_logo.svg.png

Airbnb: https://upload.wikimedia.org/wikipedia/commons/thumb/6/69/Airbnb_Logo_B%C3%A9n%C3%A9s.svg/2560px-Airbnb_Logo_B%C3%A9n%C3%A9s.svg.png

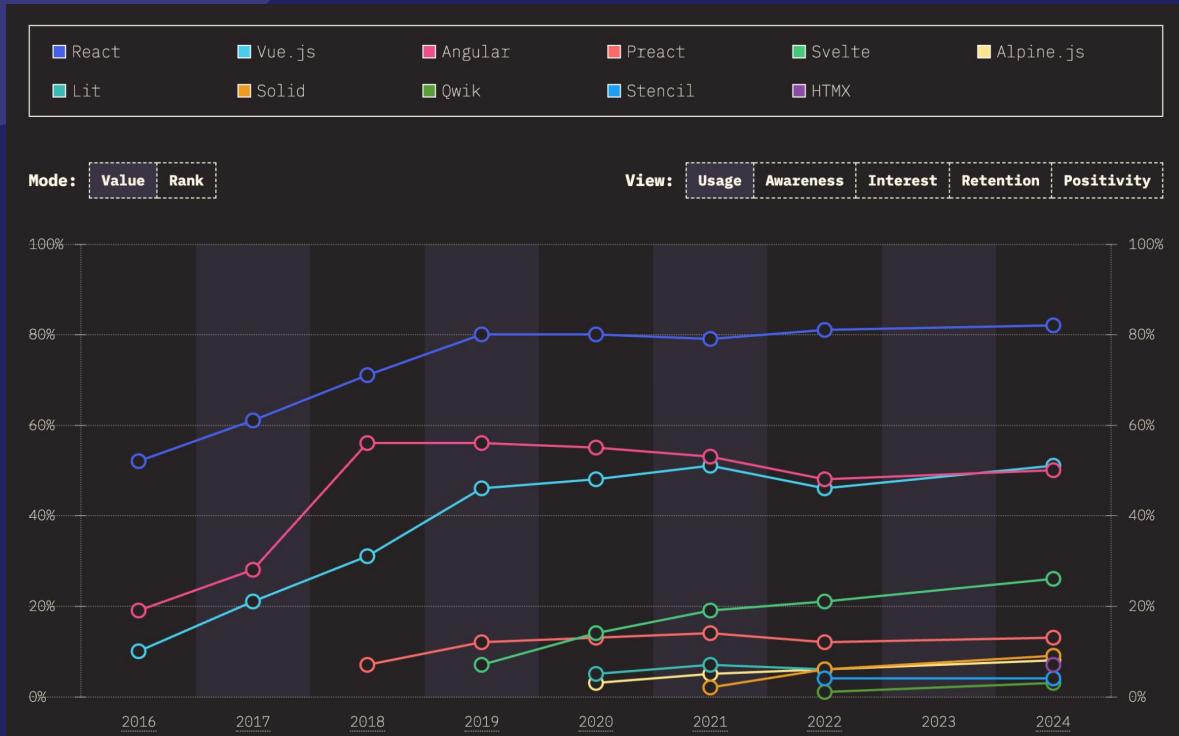
Uber: https://upload.wikimedia.org/wikipedia/commons/c/cc/Uber_logo_2018.png

Meta: <https://upload.wikimedia.org/wikipedia/commons/a/ab/Meta-Logo.png>

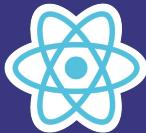
Welches ist das am
häufigsten genutzte
Frontend-Framework?



VERGLEICH BELIEBTER FRONTEND-FRAMEWORKS



Quelle: <https://2024.stateofjs.com/en-US/libraries/front-end-frameworks/>



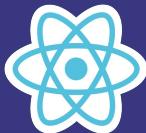
VERGLEICH BELIEBTER FRONTEND-FRAMEWORKS

REACT

- Bibliothek zur Erstellung von Benutzeroberflächen
- Entwickler wählen und kombinieren selbst zusätzliche Tools z.B. für Routing, HTTP Requests, etc.

ANGULAR, VUE

- Framework
- liefert ein komplettes Set an Tools & Richtlinien inklusive z.B. Routing, HTTP Requests etc.



VERGLEICH BELIEBTER FRONTEND-FRAMEWORKS

REACT

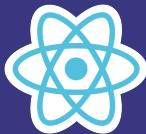


Quelle:
<https://images.selbst.de/schraube-ausbohren-istock-takayib-1341174464.jpg.id=ff1feba7.b=selbst.w=1200.rm=sk.webp>

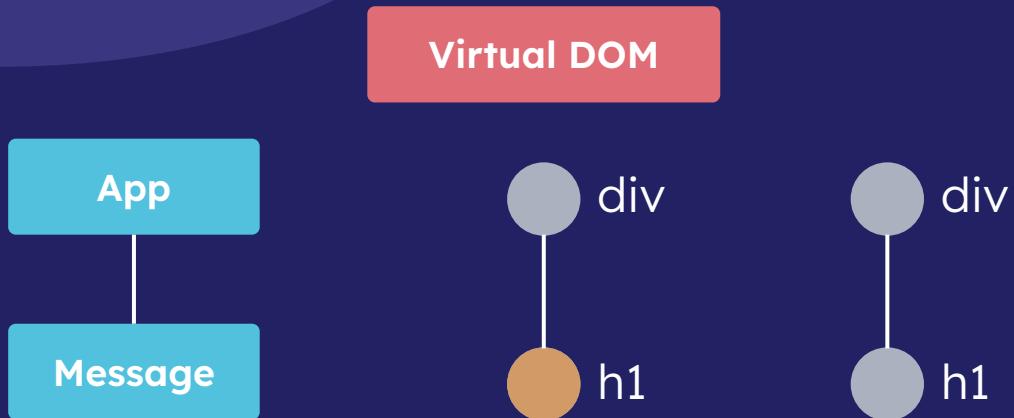
ANGULAR, VUE



Quelle: https://victorien.de/wp-content/uploads/2021/09/AdobeStock_228601171.png

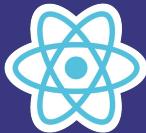


KERNKONZEPT - VIRTUAL DOM



→ schnell und effizient
→ kein manuelles
DOM handling nötig

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



TYPESCRIPT VS JAVASCRIPT

TypeScript:

- Superset von JavaScript mit zusätzlichen Features
- wird in JavaScript Code transpiliert
- statisch typisiert (JavaScript ist dynamisch typisiert)

```
let x = 5
x = "Hallo" // In JavaScript erlaubt – kein Fehler, da der Typ nicht festgelegt ist

let x: number = 5
x = "Hallo"// Fehler in TypeScript: "Hallo" ist kein number
```



VORTEILE TYPESCRIPT



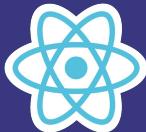
Fehlererkennung bereits zur Entwicklungszeit



Mehr Code-Sicherheit



Eindeutigere Code-Struktur



REACT PROJEKT ERSTELLEN

Optionen zur Erstellung eines React-Projekts:

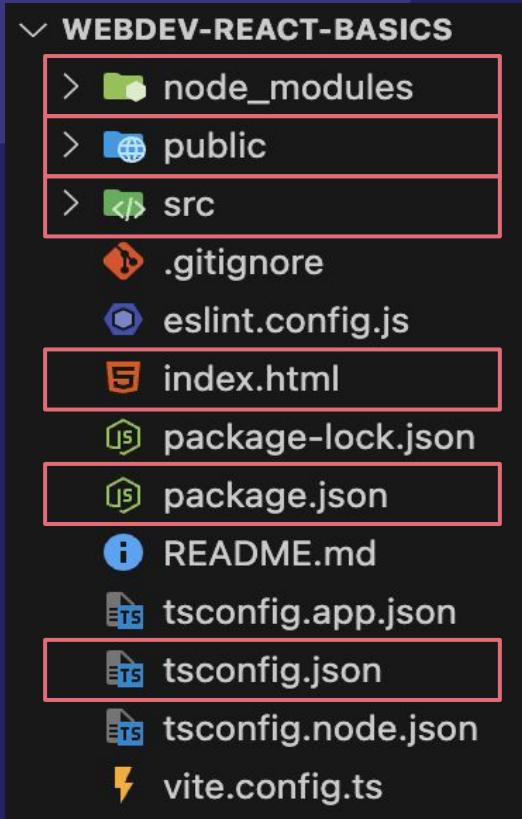
1. Create React App (CRA): offizielles Tool des React Teams, aber langsam (deprecated seit 14.02.2025)
2. Vite: sehr schnell, kleine Bundle Größe, kann jedes JS Projekt erstellen (svelte, vue, vanilla JS, react, etc.)

Erstellung eines React-Projektes mit Vite:

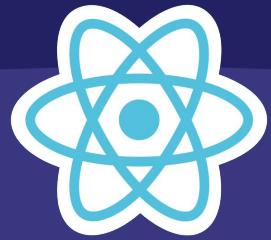
```
npm create vite@latest my-react-app
→ select a framework: select react
→ select a variant: select TypeScript
cd my-react-app
npm install
npm run dev
```



REACT PROJEKTSTRUKTUR



- **node_modules/**: Enthält alle Drittanbieter-Bibliotheken, die über npm installiert wurden
- **public/**: Beinhaltet öffentlich zugängliche Dateien wie Bilder
- **src/**: gesamter Quellcode der Anwendung
- **index.html**: HTML Template, in das React die Anwendung “einbettet”
- **package.json**: Informationen über Projekt z.B. Name, Version, Liste der Abhängigkeiten
- **tsconfig.json**: wie TypeScript zu JavaScript kompiliert wird



JSX



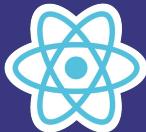
JSX

- JavaScript Syntax Extension (JavaScript + XML)
- weit verbreitet, aber keine Voraussetzung für React
- HTML-ähnlicher Markup-Code in einem JavaScript-Kontext
 ⇒ vereint Logik und UI Content in einer Datei
- JSX-Elemente kompilieren zu JavaScript-Code:
`React.createElement(type, options, children)`

```
<h1 className="heading">JSX ist super!</h1>
```

kompiliert zu

```
React.createElement("h1", {className: "heading"}, "JSX ist super!")
```



JSX-REGELN

1

Alle Tags müssen geschlossen werden → auch Tags wie **img** oder **br**

```
<h1 className="heading">JSX ist super!</h1>

```

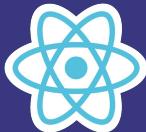


2

JavaScript-Code und Kommentare in JSX zwischen geschweiften Klammern

```
const name = 'Max Mustermann';
const greeting = <h1>Hallo, {name}!</h1>

const commentExample = <h1>{/* */}</h1>;
```



JSX-REGELN

3

Attribute wie in HTML, aber in camelCase:

- `onClick` statt `onclick`
- ACHTUNG: `className` statt `class`

4

JSX muss **immer** genau ein Element zurückgeben

```
return (
  <h1>Hallo</h1>
  <p>Willkommen!</p>
);
```



```
return (
  <div>
    <h1>Hallo</h1>
    <p>Willkommen!</p>
  </div>
);
```

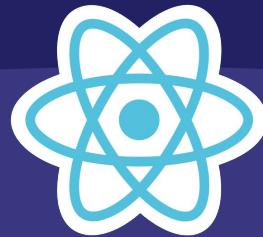


```
return (
  <React.Fragment>
    <h1>Hallo</h1>
    <p>Willkommen!</p>
  </React.Fragment>
);
```



```
return (
  <>
    <h1>Hallo</h1>
    <p>Willkommen!</p>
  </>
);
```





KOMPONENTEN



KOMPONENTEN

- wiederverwendbare UI-Bausteine
- funktionieren wie HTML-Tags
→ Verschachtelung
- geben JSX zurück
- Konvention:
 - Komponenten großschreiben
 - HTML-Tags kleinschreiben

Hallo, WebDev-Kurs!

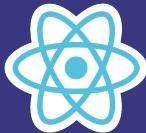
gerenderte Komponente

```
export default function Greeting() {  
  const name = "WebDev-Kurs";  
  
  return (  
    <h2 style={{ backgroundColor: "white",  
borderRadius: 8, padding: 8 }}>  
      Hallo, {name}!  
    </h2>  
  );  
}
```

Export in Greeting.tsx

```
import Greeting from "./components/Greeting";  
  
function App() {  
  return (  
    <>  
      <h1>JSX ist super!</h1>  
      <Greeting></Greeting>  
    </>  
  );  
}
```

Import in App.tsx



FUNCTIONAL VS. CLASS COMPONENTS

FUNCTIONAL

```
function Welcome() {  
  return <h1>Hallo!</h1>;  
}
```

- simpler
- heutzutage Standard

CLASS

```
import React from "react";  
  
class Welcome extends React.Component {  
  render() {  
    return <h1>Hallo!</h1>;  
  }  
}
```

- mehr Boilerplate
- früher Standard



EXPORT UND IMPORT VON KOMPONENTEN

Export

Default

```
export default function Welcome() {  
  return <h1>Hallo!</h1>;  
}
```

Import

```
import Welcome from  
"./components/Greetings";  
  
import Hallo from  
"./components/Greetings";
```

Wichtig

- nur ein default-Export pro Datei
- Name beim Import egal

Named

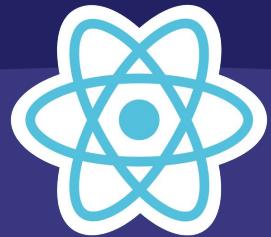
```
function Welcome() {  
  return <h1>Hallo!</h1>;  
}
```

```
function WelcomeTwo() {  
  return <h1>Hallöchen!</h1>;  
}
```

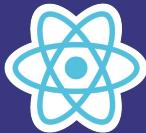
```
export { Welcome, WelcomeTwo };
```

```
import { Welcome, WelcomeTwo } from  
"./components/Greetings";
```

- mehrere Exporte pro Datei
- Importname muss gleich sein



KAPITEL 1 RECAP & ÜBUNG



RECAP KAPITEL 1

- React = weit verbreitete JavaScript-Bibliothek zur Erstellung moderner Frontends
- Virtual DOM minimiert echte DOM-Updates → schnellere UI
- TypeScript bietet einige Vorteile ggü. JavaScript
- Einfache Projektinitialisierung mit Vite
- JSX macht das Schreiben von UI einfach & lesbar – wird zu JavaScript umgewandelt
 - Syntaxregeln müssen beachtet werden!
- Komponenten sind wiederverwendbare UI-Bausteine die exportiert bzw. importiert werden müssen
- Unterschiede zwischen Named und Default Exports/Imports!
- Funktionskomponenten sind heute der Standard



ÜBUNG 1 - JSX UND KOMPONENTEN

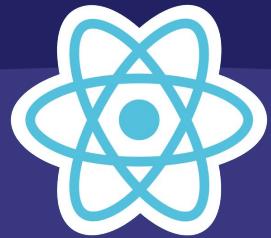
1. Initialisiere ein React-Projekt mit Vite
2. Erstelle im `src`-Folder einen Ordner `components`
3. Erstelle 2 Components und style sie mit Inline-CSS (siehe rechts)
 - a. `Greeting` (2 Überschriften)
 - b. `AboutMe` (1 Überschrift + 1 Liste)
4. Binde die beiden Components in deiner `App.tsx` ein
 - a. `Greeting` mit `React.createElement()`
 - b. `AboutMe` mit JSX-Syntax

Hallo Welt!

Das ist mein erstes React Projekt! 🚀

Über mich:

- 23 Jahre alt
- CSM-Studentin
- im 4. Semester



KOMPONENTENLOGIK PROPS



PROPS

- Props (“Properties”) = Eingabeparameter für Komponenten
- in JSX wie HTML-Attribute
- werden vom Parent Component übergeben
- read-only in Child Component
- auch Funktionen oder Komponenten
- TypeScript Besonderheit: Interface als Props-Typ

```
interface GreetingProps {  
  name: string;  
}  
  
function Greeting(props: GreetingProps) {  
  return <h1>Hallo, {props.name}!</h1>;  
}
```

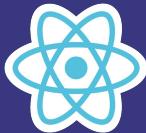
```
<Greeting name="Max" />
```



DESTRUCTURING

```
interface GreetingProps {  
  name: string;  
  isLoggedIn: boolean;  
}  
  
function Greeting({ name, isLoggedIn }: GreetingProps) {  
  return (  
    <div>  
      <h1>Hallo, {name}!</h1>  
      {isLoggedIn ? <p>Willkommen zurück!</p> : <p>Bitte einloggen.</p>}  
    </div>  
  );  
}
```

⇒ Zugriff nicht immer über `props.xyz` → Code wird lesbarer und klarer



OPTIONALE PROPS UND DEFAULT-WERTE

```
interface GreetingProps {  
  name?: string;  
}  
  
function Greeting({ name = 'du' }: GreetingProps) {  
  return <h1>Hallo, {name}!</h1>;  
}
```

```
<Greeting /> // Hallo, du!  
<Greeting name="Max" /> // Hallo, Max!
```

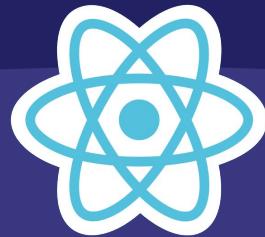


CHILDREN-PROP

```
interface LayoutProps {  
  children: React.ReactNode;  
}  
  
function Layout({ children }: LayoutProps) {  
  return (  
    <section>  
      <h1>Überschrift</h1>  
      <main>{children}</main>  
    </section>  
  );  
}
```

```
<Layout>  
  <p>Das ist der Inhalt meines Layouts.</p>  
</Layout>
```

- Layout als Wrapper für andere Inhalte
- Inhalte können Text, HTML, JSX oder eigene Komponenten sein



KOMPONENTENLOGIK

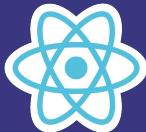
EVENT HANDLING



EVENT HANDLING

- Interaktionen wie z.B. Klicks und Hovern wird über Event Handler geregelt
- Event Handler sind eigene Funktionen, die auf Benutzerinteraktionen reagieren
- Sie werden als Props an JSX-Elemente übergeben

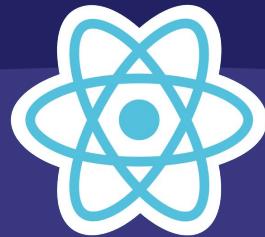
```
export default function ClickButton() {  
  function handleClick() {  
    alert('You clicked me!');  
  }  
  
  return (  
    <button onClick={handleClick}>  
      Click me  
    </button>  
  );  
}
```



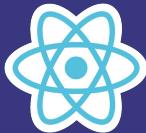
EVENT HANDLING HINWEISE

- Event Handler Funktionen werden innerhalb der Komponente definiert
- Benennungskonventionen: handle + Eventname
z.B. handleClick, handleMouseEnter
- **Nicht aufrufen!** Funktion wird übergeben, nicht direkt ausgeführt

```
onClick={handleClick} // richtig  
onClick={handleClick()} // falsch
```



KOMPONENTENLOGIK HOOKS



HOOKS

- Spezielle Funktionen, mit denen Funktionskomponenten auf React Features zugreifen können
- Jede Hook beginnt mit “use” z.B. useState, useEffect, useContext
- Man kann eigene Hooks erstellen

Regeln für die Verwendung von Hooks:

- Nur in React Funktionskomponenten verwenden
- Nur im “top level” (also ganz oben) der Komponente verwenden
⇒ nicht in Bedingungen, Schleifen oder inneren Funktionen



HOOKS: USESTATE

- State = Gedächtnis einer Komponente
- useState(): Verwaltung von State in Funktionskomponenten
- Wird genutzt, wenn sich Inhalte über die Zeit ändern sollen z.B. bei Benutzerinteraktion
- Beispiel: aktuelles Bild in einer Galerie

State Variable

```
import { useState } from 'react';
export default function ClickCounter() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Du hast {count} mal geklickt.</p>
      <button onClick={() =>
        setCount(count + 1)}>
        Klick mich!
      </button>
    </div>
  );
}
```

Setter Funktion



WARUM REICHT EINE NORMALE VARIABLE NICHT AUS?

- Lokale Variablen gehen beim erneuten Rendern verloren
- Änderungen an lokalen Variablen lösen kein neues Rendern aus

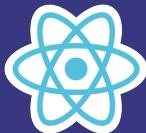
Deswegen useState() verwenden!



```
export default function ClickCounter() {
  let count = 0;

  function handleClick() {
    count = count + 1;
  }

  return (
    <div>
      <p>Du hast {count} mal geklickt.</p>
      <button onClick={handleClick}>
        Klick mich!
      </button>
    </div>
  );
}
```

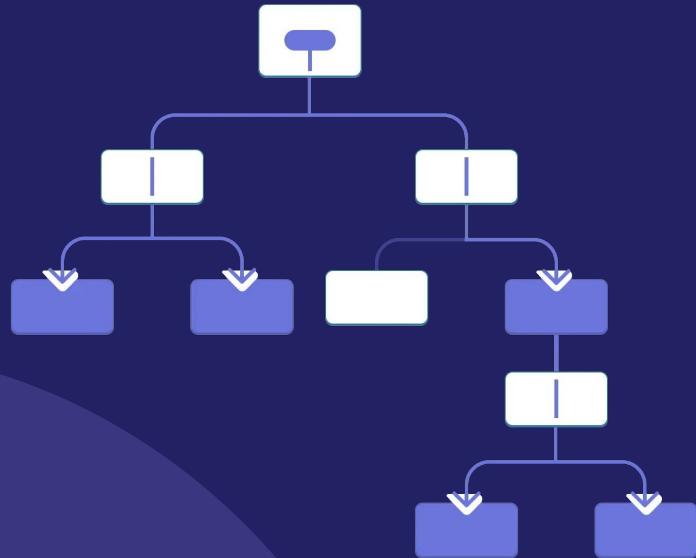


HOOKS: CONTEXT

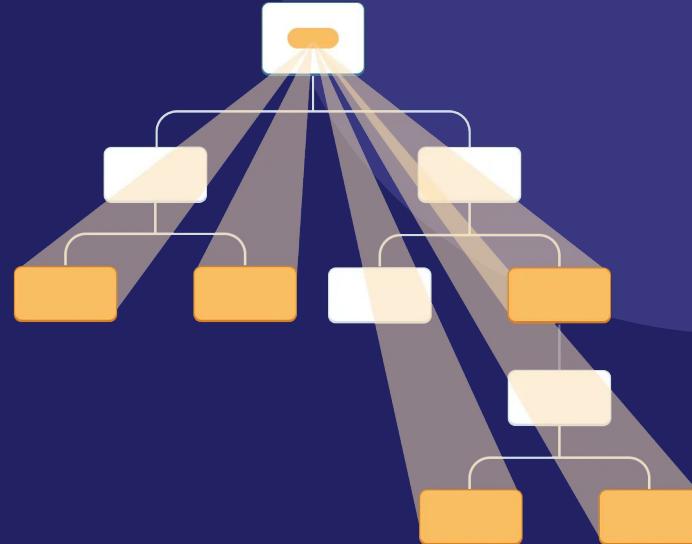
- Props: direkte Übergabe zwischen Komponenten
- Context: Speicher für globale Daten
 - Parent Component stellt Daten für alle folgenden Components im Tree bereit
- Code wird sauberer und verständlicher
- Anwendungsbeispiele:
 - Theme
 - Sprache
 - Environment
 - Login Status
 - ...



HOOKS: CONTEXT



Props Drilling



Mit Context



HOOKS: USECONTEXT

1

Context erstellen

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

type AuthContextType = {
  isSignedIn: boolean;
  setIsSignedIn: (value: boolean) => void;
};

export const AuthContext = createContext<AuthContextType>({
  isSignedIn: false,
  setIsSignedIn: () => {},
});
```

AuthContext.tsx



HOOKS: USECONTEXT

2

Context im Parent Component mit Hilfe eines Providers bereitstellen

```
import { useState } from "react";
import AuthDisplay from "./components/AuthDisplay";
import { AuthContext } from "./contexts/AuthContext";

function App() {
  const [isSignedIn, setIsSignedIn] = useState(false);

  return (
    <AuthContext.Provider value={{ isSignedIn, setIsSignedIn }}>
      <AuthDisplay></AuthDisplay>
    </AuthContext.Provider>
  );
}

export default App;
```

App.tsx



HOOKS: USECONTEXT

3

Context in Child Component benutzen

```
import { useContext } from "react";
import { AuthContext } from "../contexts/AuthContext";

export default function AuthDisplay() {
  const context = useContext(AuthContext);

  function handleClick() {
    context.setIsSignedIn(!context.isSignedIn);
  }

  return (
    <>
      <h1 className="heading">Eingeloggt? {context.isSignedIn ? "Ja" : "Nein"}</h1>
      <button onClick={handleClick}>Login-Status ändern</button>
    </>
  );
}
```

AuthDisplay.tsx



HOOKS: USEEFFECT

useEffect(*function, dependencies*)

- Hook für Nebeneffekte
- Standardmäßig Ausführung bei jedem Rendern
- 2 Parameter:
 - function: definiert Verhalten (“Was soll passieren?”)
 - dependencies (optional): definiert Abhängigkeitsliste (“Wann soll es passieren?”)
- Anwendungsbeispiele:
 - API Calls
 - Timer
 - Event Listener

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

export default function Counter() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    console.log("Count wurde geändert:", count);
  });

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>
        +1
      </button>
    </div>
  );
}
```



HOOKS: USEEFFECT - MÖGLICHE WERTE FÜR DEPENDENCY ARRAY

Wird ausgeführt...

```
useEffect(() => {  
  console.log("Count wurde geändert:", count);  
});
```

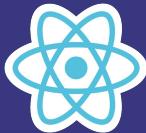
... bei jedem Rendern

```
useEffect(() => {  
  console.log("Count wurde geändert:", count);  
, []);
```

... beim ersten Rendern

```
useEffect(() => {  
  console.log("Count wurde geändert:", count);  
, [count]);
```

... wenn **count** sich ändert



USEEFFECT BEISPIEL: API CALL

```
export default function PokemonPage() {
  const [pokemon, setPokemon] = useState<PokemonData | null>(null);
  const [loading, setLoading] = useState(true);

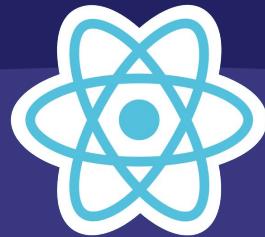
  useEffect(() => {
    fetch("https://pokeapi.co/api/v2/pokemon/1")
      .then((res) => res.json())
      .then((data: PokemonData) => {
        setPokemon(data);
        setLoading(false);
      })
      .catch((error) => {
        console.error("Fehler beim Laden des Pokémon:", error);
        setLoading(false);
      });
  }, []);

  if (loading) {
    return <p className="text-center mt-8 text-gray-500">Lade Pokémon...</p>;
  }

  if (!pokemon) {
    return <p className="text-center mt-8 text-red-500">Fehler beim Laden</p>;
  }

  return (
    <div className="max-w-md mx-auto p-6 bg-white rounded-xl shadow-md text-center">
      <h2 className="text-2xl font-bold uppercase mb-4">{pokemon.name}</h2>
      <p className="mt-2 text-gray-700">
        Höhe: {pokemon.height / 10} m<br />
        Gewicht: {pokemon.weight / 10} kg
      </p>
    </div>
  );
}
```

```
interface PokemonData {
  name: string;
  height: number;
  weight: number;
}
```



CSS MIT TAILWIND



TAILWIND

tailwindcss

- Utility-First CSS Framework
→ viele vordefinierte CSS-Hilfsklassen, statt große, wiederverwendbare CSS-Klassen
- Hilfsklassen direkt in HTML/JSX-Komponenten nutzbar
- erlaubt zentrale Definition von Farben, Schriftarten, Abständen etc.



schnell & flexibel



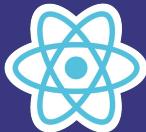
vorhersehbar
→ einfaches Debugging



benötigt kaum externe Styling-Dateien



einfaches Umsetzen von Dark Mode und responsivem Design



INTEGRATION IN VITE-REACT-PROJEKT

- 1 tailwindcss und @tailwindcss/vite mit npm installieren

```
npm install tailwindcss @tailwindcss/vite
```

- 2 Tailwind zu Vite Config hinzufügen

```
import { defineConfig } from "vite";
import react from "@vitejs/plugin-react";
import tailwindcss from "@tailwindcss/vite";
```



```
export default defineConfig({
  plugins: [react(), tailwindcss()],
});
```

vite.config.ts

- 3 @import "tailwindcss"; in index.css hinzufügen

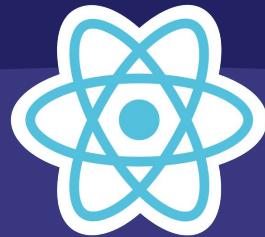
- 4 npm run dev



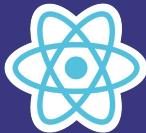
TAILWIND BEISPIEL

```
<div className="justify-items-center">
  <div className="max-w-[200px] md:max-w-[400px] lg:max-w-[800px] bg-cyan-500 shadow-md rounded-2xl p-8 duration-300 hover:scale-105">
    <h3 className="text-3xl font-semibold text-white">Tailwind Beispiel</h3>
    <p className="text-cyan-200 mt-2">
      Das ist ein Beispieltext
    </p>
  </div>
  <div className="max-w-[200px] md:max-w-[300px] lg:max-w-[600px] mt-8 shadow-md border-2 border-cyan-700 rounded-2xl p-8 duration-300 hover:-translate-y-2">
    <p className="text-cyan-800 mt-2">Hier steht noch mehr Text.</p>
  </div>
</div>
```

- vordefinierte Klassen wie z.B. `text-3xl` können direkt in `className` benutzt werden
- responsives Design (“Mobile First”)
- Animationen
- Events (Focus, Hover, ...)



KAPITEL 2 RECAP & ÜBUNG



RECAP KAPITEL 2

- Kommunikation zwischen Komponenten über Props
 - Props brauchen immer einen Typen in TypeScript!
 - Destructuring, Optionals und Default-Werte helfen beim Arbeiten mit Props
 - Nodes können als children-Props übergeben werden
- EventHandler-Funktionen werden als Props übergeben
- Hooks sind Funktionen mit denen Funktionskomponenten auf React-Funktionen zugreifen können
 - dürfen nur im top level von Funktionskomponenten aufgerufen werden
 - useState: verwaltet Zustände von Variablen
 - useContext: definiert globale Variablen und verhindert damit Props Drilling
 - useEffect: definiert Nebeneffekte (beim ersten Rendern, bei jedem Rendern oder bei Veränderung bestimmter Variablen)
- Tailwind bietet vordefinierte CSS-Klassen für schnelles und einfaches Styling



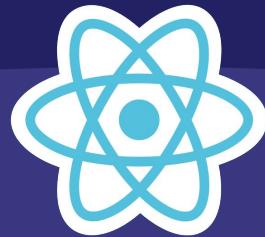
ÜBUNG 2 - KOMPONENTENLOGIK UND CSS

Bevor du anfängst: Schaue dir den bestehenden Code gut an! Hilfestellungen zu den Aufgaben stehen im Code und in der README-Datei. Tailwind ist bereits installiert.

- 1) Nutze den `AudioProvider` in `App.tsx`
- 2) Lasse den `Song des Tages` auf der `ChartsPage` anzeigen
- 3) Schreibe in der `ChartsPage` einen API Call, der die aktuellen Top 10 Songs von Deezer abruft
- 4) Implementiere eine `SongCard` Component (mit CSS Styling)

The screenshot shows a mobile application interface. At the top, there is a 'Song des Tages' card featuring a portrait of Gracie Abrams and the song 'That's So True'. Below this is a 'Top 10 Charts' section displaying ten songs with their titles, artists, and small preview images. The song 'Skyfall' by Adele is highlighted with a blue background. The chart entries are:

Rank	Song	Artist
1	Get Lucky (Radio Edit - feat. Pharrell Williams and Nile Rodgers)	Daft Punk
2	Rose	APT. Rosé
3	tau mich auf	Zartmann
4	Skyfall	Adele
5	Anxiety	Doechi
6	Azizam	Ed Sheeran
7	Messy	Lola Young
8	Something Just Like This	Coldplay
9	Die With A Smile	Lady Gaga
10	Ordinary	Alex Warren



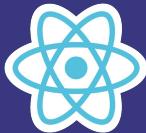
CONDITIONAL RENDERING



CONDITIONAL RENDERING

- Wenn Komponenten unterschiedliche Inhalte je nach Zustand oder Eingabe anzeigen müssen
- JSX kann mit ganz normaler JavaScript Syntax konditional gerendert werden

```
function Greeting({ isLoggedIn }) {  
  if (isLoggedIn) {  
    return <h1>Willkommen zurück!</h1>;  
  }  
  return <h1>Bitte logge dich ein.</h1>;  
}
```



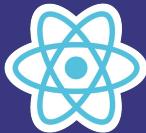
CONDITIONAL RENDERING SHORTCUTS

```
function Greeting({ isLoggedIn }) {  
  if (isLoggedIn) {  
    return <p>Eingeloggt</p>;  
  }  
  return <p>Nicht eingeloggt</p>;  
}
```

Ternary Operator

(condition ? **exprIfTrue** : **exprIfFalse**)

```
function Greeting({ isLoggedIn }) {  
  return (  
    <p>  
      {isLoggedIn ? 'Eingeloggt' : 'Nicht  
        eingeloggt'}  
    </p>  
  );  
}
```

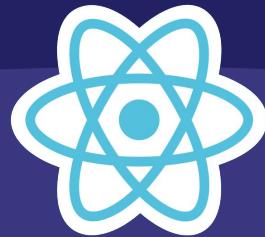


CONDITIONAL RENDERING SHORTCUTS

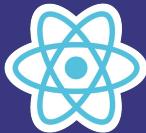
Logical AND operator (&&)

```
function Notification({ hasMessage }) {  
  return (  
    <div>  
      <h2>Startseite</h2>  
      {hasMessage && <p>Du hast eine neue Nachricht!</p>}  
    </div>  
  );  
}
```

“Wenn hasMessage true ist,
dann (&&) Text rendern,
andernfalls nichts rendern.”



RENDERN VON LISTEN



LISTEN IN PLAIN HTML

```
export default function UserList() {  
  return (  
    <ul>  
      <li>Anna</li>  
      <li>Tom</li>  
      <li>Dieter</li>  
    </ul>  
  );  
}
```

Was wenn man 1000 Einträge anzeigen möchte?



NACHTEILE

- Manuelles Schreiben jedes Elements
- Keine Wiederverwendbarkeit
- Nicht dynamisch und schwer aktualisierbar

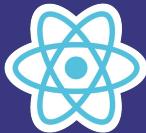


RENDERING LISTS

- **Dynamisches Rendering durch `map()`**
- **Wiederverwendbare Komponente**
- **Daten kommen aus Arrays oder Objekten**

```
const users = ['Anna', 'Tom', 'Dieter'];

export default function UserList() {
  return (
    <ul>
      {users.map(user => <li>{user}</li>)}
    </ul>
  );
}
```



DAS “KEY” PROBLEM

⚠ Warning: Each child in a list should have a unique “key” prop.

- key = eindeutiger Bezeichner für jedes Listenelement
- Wichtig bei dynamischen Änderungen (sortieren, löschen, hinzufügen)
- Hilft React, DOM richtig zu aktualisieren

```
{users.map(user => <li>{user}</li>)} // Kein Key!
```

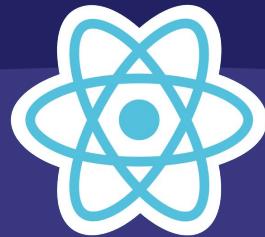
```
{users.map(user => <li key={user}>{user}</li>)} // Mit Key
```



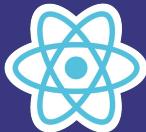
LISTEN FILTERN

- `filter()` gibt ein neues Array mit den passenden Elementen zurück
- Kombination mit `map()` ermöglicht sofortiges Rendern der gefilterten Liste
- Kann mit jeder Bedingung kombiniert werden
(`user.age > 18`, `item.done === false`, etc.)

```
const users = [  
  { id: 1, name: 'Anna', active: true },  
  { id: 2, name: 'Tom', active: false },  
  { id: 3, name: 'Dieter', active: true }  
];  
  
export default function ActiveUserList() {  
  const activeUsers = users.filter(user =>  
    user.active);  
  return (  
    <ul>  
      {activeUsers.map(user => (  
        <li key={user.id}>{user.name}</li>  
      ))}  
    </ul>  
  );  
}
```



CLIENT-SIDE RENDERING



CLIENT SIDE RENDERING

- Rendering erfolgt im Browser (Client)
- Server liefert nur minimales HTML-Gerüst
- JavaScript-Bundle wird vom Client geladen
- React rendert die UI nach dem Laden des JavaScripts



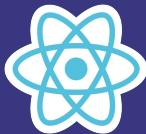
VORTEILE

- Schnelle Navigation nach dem ersten Laden
- Weniger Serverlast
- Hohe Interaktivität durch dynamisches Rendering



NACHTEILE

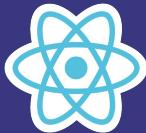
- Langsame Erstladezeit
- Schlechtere SEO, ohne Zusatzmaßnahme



SERVER SIDE RENDERING UND STATIC SITE GENERATION

Server Side Rendering (SSR): HTML wird bei jeder Anfrage neu auf dem Server erzeugt

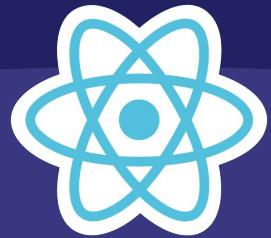
Static Site Generation (SSG): HTML wird zur Build-Zeit (also beim Erstellen der Anwendung) einmalig erzeugt und bei jeder Anfrage wiederverwendet



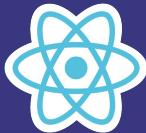
SERVER SIDE RENDERING UND STATIC SITE GENERATION IN REACT???

- React allein bietet keine SSR oder SSG „out of the box“
- **Möglich mit viel Aufwand** durch manuelles Rendern auf dem Server z. B. mit ReactDOMServer und Express
- Komplex und fehleranfällig, z. B. bei Routing, Datenvorbereitung, Rehydration

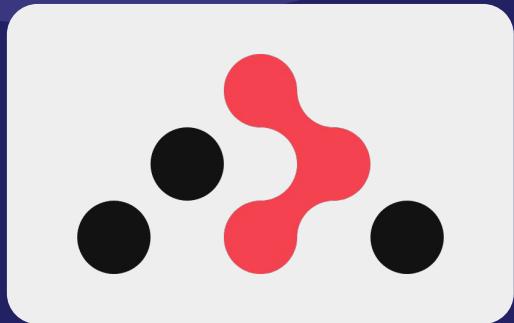
Besser: Framework wie Next.js verwenden



ROUTING



REACT ROUTER



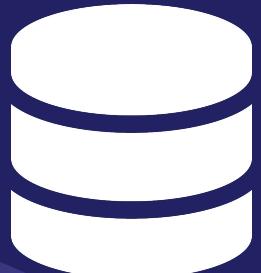
- Standardbibliothek für Routing in React
- ermöglicht Navigieren zwischen Seiten in einer Single Page App
- URL-basiertes Rendering

Installation mit npm:

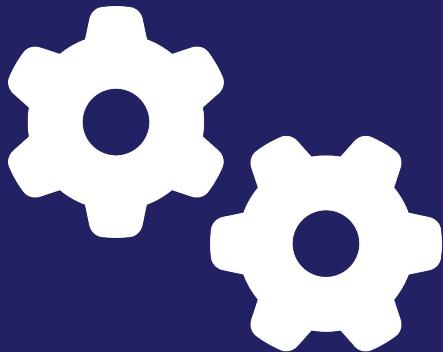
```
npm install react-router-dom
```



REACT ROUTER MODI



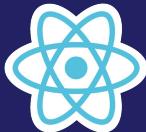
DATA



FRAMEWORK



DECLARATIVE

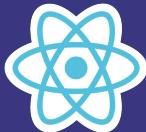


REACT ROUTER

```
import { BrowserRouter, Routes, Route } from "react-router-dom";
import HomePage from "./pages/HomePage";
import CounterPage from "./pages/CounterPage";

export default function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route index element={<HomePage />} />
        <Route path="/counter" element={<CounterPage />} />
      </Routes>
    </BrowserRouter>
  );
}
```

- **BrowserRouter**-Wrapper als Basis
- **Routes**-Container sorgt dafür, dass nur eine Route gerendert wird
- pro Route: **Route mit path und element (erwartet JSX)**
- **index** definiert Startseite für einen Pfad



DYNAMISCHES ROUTING

```
import { BrowserRouter, Routes, Route } from
"react-router-dom";
import HomePage from "./pages/HomePage";
import UserPage from "./pages/UserPage";

export default function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<HomePage />} />
        <Route path="users/:id" element={<UserPage />} />
      </Routes>
    </BrowserRouter>
  );
}
```

App.tsx

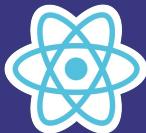
```
import { useParams } from
"react-router-dom";

export default function UserPage() {
  const { id } = useParams();

  return (
    <div>
      <h1>User Seite</h1>
      <p>ID: {id}</p>
    </div>
  );
}
```

UserPage.tsx

- Schreibweise mit Doppelpunkt erlaubt Einsetzen von Parametern (:id)
- Zugriff auf Routenparameter mit useParams()-Hook



VERSCHACHTELTES ROUTING

```
import { BrowserRouter, Routes, Route } from "react-router-dom";
import DashboardPage from "./pages/DashboardPage";
import ProfilePage from "./pages/ProfilePage";
import SettingsPage from "./pages/SettingsPage";

export default function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/dashboard" element={<DashboardPage />}>
          <Route path="profile" element={<ProfilePage />} />
          <Route path="settings" element={<SettingsPage />} />
        </Route>
      </Routes>
    </BrowserRouter>
  );
}
```

App.tsx

```
import { Outlet } from "react-router-dom";

export default function DashboardPage() {
  return (<>
    <h1>Dashboard</h1>
    <Outlet/>
  </>);
}
```

DashboardPage.tsx

```
export default function SettingsPage() {
  return <h2>Einstellungen</h2>;
}
```

SettingsPage.tsx

- **Route** kann andere **Routes** enthalten
- Inhalt der inneren Route wird über **Outlet** in Parent Component eingebunden
- im Beispiel: auf der Seite gibt es jetzt die Routen **/dashboard/profile** und **/dashboard/settings**



LAYOUT ROUTING

```
import { BrowserRouter, Routes, Route } from "react-router-dom";
import DashboardLayout from "./pages/DashboardLayout";
import ProfilePage from "./pages/ProfilePage";
import SettingsPage from "./pages/SettingsPage";

export default function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route element={<DashboardLayout />}>
          <Route path="profile" element={<ProfilePage />} />
          <Route path="settings" element={<SettingsPage />} />
        </Route>
      </Routes>
    </BrowserRouter>
  );
}
```

App.tsx

```
import { Outlet } from "react-router-dom";

export default function DashboardLayout() {
  return (<>
    <h1>Dashboard</h1>
    <Outlet/>
  </>);
}
```

DashboardLayout.tsx

```
export default function SettingsPage() {
  return <h2>Einstellungen</h2>;
}
```

SettingsPage.tsx

- Layout-Route braucht keinen Pfad
- Kindrouten sind dann direkt über eigenen Pfad erreichbar, mit Layout der Elternroute

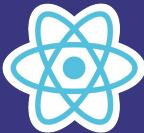


WILDCARD-ROUTE

```
import { BrowserRouter, Routes, Route } from "react-router-dom";
import HomePage from "./pages/HomePage";
import CounterPage from "./pages/CounterPage";
import NotFoundPage from "./pages/NotFoundPage";

export default function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<HomePage />} />
        <Route path="/counter" element={<CounterPage />} />
        <Route path="*" element={<NotFoundPage />} />
      </Routes>
    </BrowserRouter>
  );
}
```

- `path="*"` definiert Wildcard-Route → fängt alle Routen ab die nicht explizit definiert werden
- Beispiel: 404-Seite (Page not found)

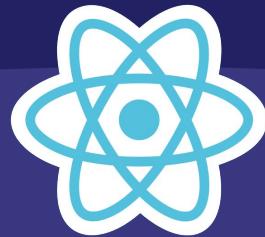


LINKS

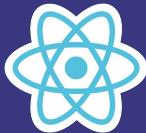
```
<Link to="/">Zurück</Link>
<Link to="/dashboard">Zum Dashboard</Link>
```

```
<NavLink
  to="/dashboard"
  style={({ isActive }) => ({
    fontWeight: isActive ? "bold" : "normal",
    textDecoration: isActive ? "underline" : "none",
  })}
>
  Dashboard
</NavLink>
```

NavLink ermöglicht Zugriff auf isActive-Zustand des Links

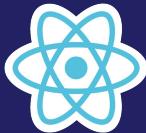


KAPITEL 3 RECAP & ÜBUNG



RECAP KAPITEL 3

- Conditional Rendering von JSX mit Ternary und &&-Operator möglich
- Rendern von Listen durch `.map()` (Key wird benötigt)
- Für SSR und SSG lieber Next.js verwenden
- Standardtool für Routing in React: React Router
 - benötigt `BrowserRouter` und Definition der verschiedenen Routes
 - liefert:
 - Dynamisches Routing
 - Verschachtelung von Routen
 - Layouts
 - Wildcard-Routen
 - einfache Verlinkung von Seiten



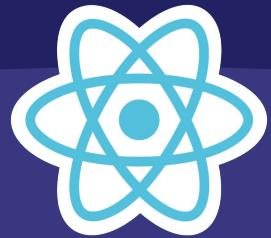
ÜBUNG 3 - ROUTING

Bevor du anfängst: Schaue dir den bestehenden Code gut an! Hilfestellungen zu den Aufgaben stehen im Code und in der README-Datei.

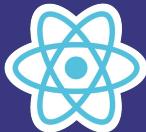
- 1) Installiere den React Router
- 2) Implementiere die Component `SidebarLink`, die anzeigt wenn ein Link gerade aktiv ist
- 3) Nutze `AppLayout.tsx` um die Sidebar auf allen Seiten anzuzeigen
- 4) Implementiere eine `SearchPage`, auf der man in einem Input-Field nach Songs suchen kann

The screenshot shows a search interface for the Deezer Player. At the top, there is a search bar containing the text "Gracie Abrams" and a green "Suchen" button. Below the search bar, the text "Deezer Player" is displayed. On the left, there are navigation links: "Home" and "Search". The main area is titled "Suchergebnisse" (Search results). It lists ten song entries, each with a small thumbnail image, the song title, the artist name "Gracie Abrams", and a play button icon. The songs listed are:

- That's So True by Gracie Abrams
- I Love You, I'm Sorry by Gracie Abrams
- I miss you, I'm sorry by Gracie Abrams
- Close To You by Gracie Abrams
- Call Me When You Break Up by Selena Gomez
- Risk by Gracie Abrams
- Mess It Up by Gracie Abrams
- 21 by Gracie Abrams
- us. by Gracie Abrams
- Death Wish (Live from The O2...) by Gracie Abrams
- I know it won't work by Gracie Abrams
- I Told You Things by Gracie Abrams



FRONTEND TESTING



WAS IST FRONTEND TESTING?

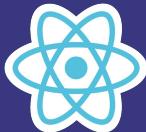
- Testen der Benutzeroberfläche (UI) einer Softwareanwendung
- Elemente, die direkt für Nutzer sichtbar oder interaktiv sind
- Sicherstellen, dass Komponenten wie erwartet funktionieren und auf Benutzeraktionen korrekt reagieren

Automatisierte
Tests

Durch Tools automatisiert ausführbare Tests
z.B. Unit- oder E2E-Tests

Manuelle
Tests

Durch echte Nutzer oder Tester z. B.
Usability-Tests



WARUM VON FRONTEND TESTING



Frühes Erkennen von Fehlern
zur Reduktion von Kosten



Sicherstellung der
Funktionalität



Sicherstellung der
Performance



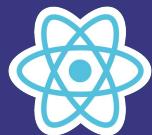
Gewährleistung der
Code-Stabilität



Optimierung der
Benutzerfreundlichkeit und
Barrierefreiheit



Cross-Browser-Kompatibilität



TESTING PYRAMIDE

Mehr
Kosten

E2E

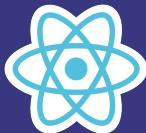
Langsamer

Integration

Unit

Weniger
Kosten

Schneller



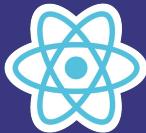
WEITERE FRONTEND TESTS

Visual Regression Tests

Accessibility Tests

Performance Tests

Cross Browser Tests

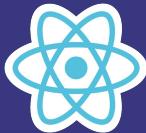


UNIT-TEST

- Isoliertes Testen einzelner Funktionen oder Komponenten
- Schnell und einfach zu schreiben
- Keine Abhangigkeiten zu externen Systemen
- Gute Fehlerlokalisierung

Ask yourself...

- Wird ein Button angezeigt?
- Wird eine Callback-Funktion beim Klicken aufgerufen?
- Wird der richtige Text bei bestimmten Props gerendert?



DOS BEIM UNIT TESTING



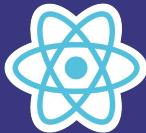
Einzelne Komponenten oder Funktionen testen



Externe Abhängigkeiten mocken (z. B. API-Calls, Router, Kontext)



Edge Cases und Fehlerzustände berücksichtigen



DON'TS BEIM UNIT TESTING



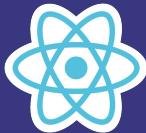
Kombinationen mehrerer Komponenten oder komplexe Logik auf einmal testen



Tests, die vom globalen Zustand oder Netzwerk abhängig sind



Implementierungsdetails testen (→ lieber Verhalten testen)



VITEST

- Test-Runner speziell für Vite-Projekte (nutzt denselben Build Prozess)
- Zuständig für Organisation und Ausführung der Tests
- Entwickelt für schnelle Builds der Tests
- API ähnlich zu Jest



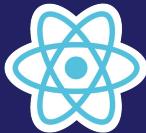
Schneller als Jest (dank Vite + esbuild)



Native ESM-Unterstützung



Einfache Konfiguration – direkt in vite.config.js



VITEST - TESTAUFBAU

- `describe()`: Gruppiert zusammengehörige Tests
- `it() / test()`: Definiert einen einzelnen Testfall
- `expect()`: Stellt eine Erwartung an das Ergebnis
- `toBe()`: Matcher Funktion

```
import { describe, it, expect } from  
'vitest'  
  
describe('Math operations', () => {  
  it('adds two numbers', () => {  
    expect(2 + 2).toBe(4)  
  })  
})
```



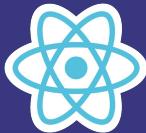
VITEST - HOOKS

- Code ausführen vor oder nach Tests (**Setup und Teardown**)
- **beforeAll**: einmal vor allen Tests in einem **describe**
- **afterAll**: einmal nach allen Tests in einem **describe**
- **beforeEach**: vor jedem einzelnen Test
- **afterEach**: nach jedem einzelnen Test

```
describe('Mit Hooks', () => {
  beforeAll(() => {
    console.log('Starte Test-Suite')
  })

  it('Test 1', () => {
    expect(1).toBe(1)
  })

  afterAll(() => {
    console.log('Test beendet')
  })
})
```



REACT TESTING LIBRARY (RTL)

"The more your tests resemble the way your software is used, the more confidence they can give you." - Kent C. Dodds (Schöpfer der React Testing Library)

- Testen von React-Komponenten aus Sicht des Benutzers
- Fokus auf Benutzerinteraktionen und sichtbares Verhalten
- Vermeidung von Implementierungsdetails bei Tests

RTL und Vitest komplementieren sich:

- RTL stellt Test-Werkzeuge bereit
- Vitest übernimmt Testausführung



BASIC TEST MIT VITEST + RTL

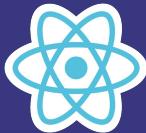
- `render(<Button />)` stellt die Komponente im Testumfeld bereit
- `screen.getByText('Clickme')` findet das Button-Element anhand des sichtbaren Textes
- `expect().toBeInTheDocument()` Überprüft, ob das Button-Element tatsächlich im DOM vorhanden ist

```
export function Button() {  
  return <button>Click me</button>;  
}
```

Button.tsx

```
import { render, screen } from '@testing-library/react';  
import { describe, it, expect } from 'vitest';  
import { Button } from './Button';  
  
describe('Button', () => {  
  it('zeigt den Button-Text an', () => {  
    render(<Button />);  
    expect(screen.getByText('Clickme'))  
      .toBeInTheDocument();  
  });  
});
```

Button.test.tsx



FEATURES IN RTL

render():

- Rendert eine React-Komponente in einer isolierten Testumgebung
- Macht die Komponente bereit für Abfragen und Nutzerinteraktionen
- Keine echte Browserumgebung notwendig (läuft in JSDOM)
- Beispiel: `render(<Button />);`

screen + Abfrage:

- Zugriff auf die gerenderte UI
- Nutzt nutzerzentrierte Abfragen: `getByText`, `getByRole`, `getByLabelText`
- Beispiel: `screen.getByRole('button');`

userEvent:

- Simuliert echte Nutzeraktionen: Klicken, Tippen, etc.
- Beispiel: `await userEvent.click(screen.getByText('Submit'))`



SETUP VITEST & REACT TESTING LIBRARY

Setup React Projekt mit Vite

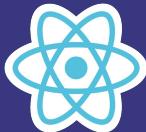
```
npm create vite@latest my-app  
cd my-app  
npm install
```

Benötigte Abhängigkeiten installieren

```
npm install --save-dev vitest  
npm install --save-dev jsdom  
npm install --save-dev @testing-library/react @testing-library/jest-dom  
@testing-library/user-event
```

Test laufen lassen

```
npm run test
```



SETUP VITEST & REACT TESTING LIBRARY

```
"scripts": {  
  "test": "vitest"  
}
```

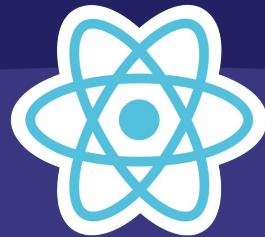
package.json

```
import { afterEach } from 'vitest'  
import { cleanup } from '@testing-library/react'  
import '@testing-library/jest-dom/vitest'  
  
// runs a clean after each test case (e.g.  
// clearing jsdom)  
afterEach(() => {  
  cleanup();  
})
```

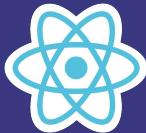
setup.js

```
import { defineConfig, UserConfig }  
from 'vite'  
import react from  
'@vitejs/plugin-react'  
  
export default defineConfig({  
  plugins: [react()],  
  test: {  
    environment: 'jsdom',  
    setupFiles: './tests/setup.js',  
  }  
} as UserConfig)
```

vite.config.js

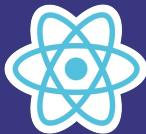


KAPITEL 4 RECAP & ÜBUNG



RECAP KAPITEL 4

- Ziel von Frontend Testing: Sicherstellen, dass die UI aus Nutzersicht korrekt funktioniert
- Warum testen?
 - Fehler frühzeitig erkennen
 - Funktionalität sicherstellen
 - ...
- Vitest: Schneller, moderner Test-Runner für Vite-basierte Projekte
- React Testing Library (RTL): Ermöglicht die UI wie echte Nutzer, nicht wie Entwickler zu testen
- React Testing Library und Vitest komplementieren sich



ÜBUNG 4 - UNIT TESTING

Bevor du anfängst: Schaue dir den bestehenden Code gut an! Hilfestellungen zu den Aufgaben stehen im Code und in der README-Datei. Die React-Komponente `Counter` ist bereits vorbereitet.

- 1) Implementiere die 3 vorbereiteten Testfälle in der Datei `Counter.test.tsx`
- 2) Überprüfe ob alle 3 Tests erfolgreich durchlaufen

Hinweis: Es sind keine Änderungen an der Komponente nötig

```
✓ src/components/Counter.test.tsx (3 tests) 117ms
  ✓ Counter > zeigt den Anfangszähler 16ms
  ✓ Counter > erhöht den Zähler bei Klick 59ms
  ✓ Counter > zeigt eine Nachricht nach 5 Klicks 40ms
```

```
Test Files 1 passed (1)
Tests 3 passed (3)
```

```
Start at 18:59:46
```

```
Duration 849ms (transform 33ms, setup 148ms, collect 45ms, tests 117ms, environment 336ms, prepare 45ms)
```



QUELLEN

- React Dokumentation: <https://react.dev/learn> (27.04.2025)
- W3Schools - useEffect: https://www.w3schools.com/react/react_useeffect.asp (16.04.2025)
- React Router: <https://reactrouter.com/home> (20.04.2025)
- React Routing mit TypeScript: <https://galaxies.dev/quickwin/react-router-typescript> (20.04.2025)
- Tailwind: <https://tailwindcss.com/> (20.04.2025)
- React Testing:
<https://victorbruce82.medium.com/vitest-with-react-testing-library-in-react-created-with-vite-3552f0a9a19a> (21.04.2025)
- React Einführung: <https://www.geeksforgeeks.org/reactjs-introduction/> (26.04.2025)
- Framework Vergleich:
<https://medium.com/@vijendrapro22/angular-vs-react-vs-vue-2024-a-comprehensive-comparison-d3621d1c9234> (26.04.2025)
- Testing Pyramide:
<https://medium.com/comsystoreply/mastering-frontend-testing-essential-best-practices-b0924655e765> (02.05.2025)
- Unit Testing: <https://kentcdodds.com/blog/static-vs-unit-vs-integration-vs-e2e-tests> (02.05.2025)
- React Logo: https://imgbin.com/png/XB8Dgn1d/react-logo-icon-png#google_vignette
- Icons: <https://fontawesome.com/search?o=r&ic=free&s=solid&ip=classic>