React-Entwicklung mit TypeScript – Lehrskript

Technisches Skriptum für die HTL Informatik
Version 1.0 | April 2025

von Mathias Schober

Inhaltsverzeichnis

1 Einführung in React mit Vite

React ist eine JavaScript-Bibliothek zur Entwicklung interaktiver Benutzeroberflächen. Anders als bei klassischen Websites, bei denen jeder Seitenwechsel eine neue HTML-Seite vom Server lädt, ermöglicht React den Aufbau von Single-Page-Applications (SPAs). Bei einer SPA wird nur eine einzelne HTML-Seite geladen; React übernimmt dann die dynamische Aktualisierung des DOMs im Browser, sodass Inhalte sich ändern können, ohne die Seite neu zu laden. Dies geschieht effizient durch das Konzept des virtuellen DOM: React hält eine virtuelle Kopie des DOMs im Speicher und synchronisiert nur die Unterschiede mit dem echten DOM, was zu hoher Performance führt.

- **Vite** ist ein modernes Build-Tool und Entwicklungsserver, mit dem sich React-Projekten sehr schnell erstellen und ausführen lassen. Früher wurde häufig *Create React App* verwendet; Vite bietet jedoch deutlich schnellere Start- und Build-Zeiten. Vorteile von Vite sind unter anderem:
- **Blitzschneller Dev-Server:** Vite startet einen lokalen Entwicklungsserver mit *Hot Module Replacement* (HMR), wodurch Änderungen im Code sofort im Browser aktualisiert werden, ohne die Anwendung neu zu laden.
- **Zero-Config TypeScript-Unterstützung:** Vite unterstützt TypeScript out-of-the-box, ohne komplizierte Konfiguration.
- **Schnelle Builds:** Dank moderner Bundling-Strategien und nativer ES-Modul-Verarbeitung sind auch Produktions-Builds effizient und zügig.

1.1 React Projekt mit Typescript und Vite aufsetzen.

1• **Node.js installieren:** Stelle sicher, dass Node.js (empfohlen Version ≥ 18) installiert ist.

• **Vite-Projekt erstellen:** Öffne ein Terminal und führe den Befehl `npm create vite@latest` aus. Gib einen Projektnamen ein, z.B. `meine-react-app`.

• **Template wählen:** Wähle als Framework **React** und als Variante **TypeScript**, wenn Vite danach fragt. Dadurch wird ein Grundgerüst für eine React-App mit TypeScript erstellt.

• **Abhängigkeiten installieren:** Wechsle ins Projektverzeichnis (`cd meine-react-app`) und installiere die Abhängigkeiten mit `npm install`.

• **Entwicklungsserver starten:** Führe `npm run dev` aus, um den Vite-Entwicklungsserver zu starten. Im Terminal siehst du nun die lokale Adresse (z.B. `http://localhost:5173`), unter der die App erreichbar ist. Öffne diese Adresse im Browser.

1.2 Main.tsx

Im Code der Datei **main.tsx** sehen wir, wie React mit der HTML-Seite verbunden wird. Dort wird die Haupt-React-Komponente <App /> in ein DOM-Element (meist mit der ID "root" in index.html) eingebettet:

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App';
```

```
// React an das DOM anhängen
ReactDOM.createRoot(document.getElementById('root') as
HTMLElement).render(
  <React.StrictMode>
    <App /> {/* Einbinden der Hauptkomponente */}
  </React.StrictMode>
);
```

Jetzt wissen wir was React ist und wie man ein neues Projekt mit Vite aufsetzt. Als Nächstes werden wir uns mit JSX und der Komponentenstruktur befassen, um eigene Benutzeroberflächen zu erstellen.

2. JSX und Komponentenstruktur

Ein zentrales Konzept in React ist **JSX** (JavaScript XML). JSX ist eine Erweiterung der JavaScript/TypeScript-Syntax, die es ermöglicht, HTML-ähnliche Strukturen direkt im Code zu schreiben. Zum Beispiel:

```
const element = <h1>Hallo Welt!</h1>;
```

Hierbei ist <h1>Hallo Welt!</h1> weder ein String noch reines HTML, sondern JSX. Dieser Code wird vom Compiler (Babel/TSX) in reguläres JavaScript umgewandelt, das React-Elemente erzeugt. JSX vereinfacht das Schreiben von Komponenten, da Markup und Logik nahe beieinander bleiben können. Wichtig: In JSX muss ein Component-Return immer **ein einzelnes** Elternelement haben (oder einen Fragment-Wrapper <>...</> nutzen), da eine Komponente nur *ein* Element zurückgeben kann.

Komponenten sind die Bausteine jeder React-Anwendung. Eine Komponente ist typischerweise entweder:

- eine **Funktion** (*Function Component*), die JSX zurückgibt, oder
- eine **Klasse** (*Class Component*) mit einer render()-Methode (wird in modernem React kaum noch verwendet, da Hooks Funktionskomponenten bevorzugt).

Im Regelfall erstellen wir Funktionskomponenten in TypeScript. Jede Komponente kann wie ein eigener HTML-Tag verwendet werden. Komponenten werden großgeschrieben benannt (z.B. WelcomeMessage), um sie von normalen HTML-Tags zu unterscheiden. Dateien mit React-Komponenten erhalten die Endung .tsx (da sie TS + JSX enthalten).

Man kann sich eine React-Komponente als eine Funktion vorstellen, die UI produziert. Beispiel einer einfachen Komponente und deren Nutzung:

```
// Eine einfache Komponente definieren
function Welcome() {
```

```
  return <h1>Willkommen bei React!</h1>;
}

// Eine weitere Komponente, die Welcome nutzt
function App() {
  return (
    <div>
      <Welcome /> {/* Nutzung der Komponente wie ein HTML-
Tag */}
      <p>Dies ist eine React-Komponente mit JSX.</p>
    </div>
  );
}

export default App;
```

In diesem Beispiel haben wir zwei Komponenten: Welcome gibt eine Überschrift zurück, und App verwendet Welcome innerhalb eines -Containers. Die <Welcome />-Syntax zeigt, wie Komponenten verschachtelt werden. In JSX können auch JavaScript-Ausdrücke in geschweiften Klammern eingefügt werden, z.B. {2 + 2} oder {user.name} – React rendert dann den ausgewerteten Ausdruck an dieser Stelle.

Zusammenfassend: JSX erlaubt es, die Struktur der Benutzeroberfläche deklarativ im Code zu schreiben, und Komponenten strukturieren die Anwendung in wiederverwendbare, isolierte Einheiten. Als Nächstes betrachten wir, wie Komponenten mit internen Daten umgehen, nämlich den State mit Hooks wie **useState** und **useReducer**.

3. Zustandsverwaltung mit useState

Interaktive Anwendungen brauchen variierbare Daten – in React nennt man solche internen, veränderlichen Daten den **State** (Zustand). Jede Komponenteninstanz kann eigenen State besitzen. Wenn sich der State ändert, rendert React die Komponente (und ihre Kinder) neu, damit die UI den neuen Stand widerspiegelt.
Für Funktionskomponenten stellt der React **Hook** useState eine einfache Möglichkeiten bereit, um State zu verwalten:
useState: Dieser Hook wird für einfachen, isolierten State verwendet. Man ruft const [state, setState] = useState<Typ>(Initialwert) in der Komponente auf. useState liefert ein Array mit dem aktuellen Wert und einer Funktion, um diesen zu aktualisieren. Einen neuen Wert setzt man, indem man setState(neuerWert) oder – falls der neue Wert vom alten abhängt – mit einer Funktion setState(prev => neuerWert) aufruft. React sorgt

dafür, dass bei einem setState die Komponente neu gerendert wird. Der State bleibt zwischen Render-Durchläufen erhalten.

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

function Counter() {
  const [count, setCount] = useState<number>(0); //
initialer State = 0

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

In diesem Beispiel definiert useState(0) einen Zahlen-State count mit Startwert 0. Jedes Mal, wenn der Button geklickt wird, ruft setCount(count + 1) ein State-Update auf – React rendert die Komponente neu und zeigt den aktualisierten Wert an. (Hinweis: Das State-Update passiert asynchron und gebatcht, daher sollte man nie direkt nach setCount auf den alten count-Wert vertrauen, sondern sich auf das nächste Rendering verlassen.)

4. Props und Komponentenkommunikation

Props (Properties) sind die *Eingabedaten* für React-Komponenten. Ähnlich wie Funktionen Parameter erhalten, bekommen Komponenten Props. Datenfluss in React ist im Allgemeinen unidirektional: **von Eltern- zu Kind-Komponenten** (top-down). Das heißt, eine Elternkomponente kann einer Kindkomponente Daten über Props übergeben. Die Kindkomponente rendert diese, darf sie aber nicht verändern (Props sind read-only).

In TypeScript definieren wir die Struktur der Props meist mit einem Interface oder Typ. Beispiel: Wir haben eine Child-Komponente, die einen Namen anzeigen und einen Button hat. Wenn der Button geklickt wird, soll die Child-Komponente der Elternkomponente Bescheid geben (via Callback). Wir definieren die Props so, dass ein Name (name) und eine Callback-Funktion (onButtonClick) übergeben werden können:

```
interface ChildProps {
  name: string;
  onButtonClick: () => void;
}
```

```
function Child({ name, onButtonClick }: ChildProps) {
  return (
    <div>
      <p>Hallo, {name}!</p>
      <button onClick={onButtonClick}>Klick</button>
    </div>
  );
}

function Parent() {
  const handleChildClick = () => {
    alert('Button in Child geklickt!');
  };

  return <Child name="Alice"
onButtonClick={handleChildClick} />;
}
```

Hier übergibt Parent der Child-Komponente zwei Props: den Namen "Alice" und eine Funktion handleChildClick. Die Child-Komponente verwendet name, um einen Gruß anzuzeigen, und ruft onButtonClick auf, wenn der Button gedrückt wird. So kommuniziert das Kind eine Aktion zurück an den Elternteil. Dieses Muster – Daten fließen nach unten, Ereignisse (über Callback-Props) nach oben – ist in React sehr verbreitet.

Man kann beliebige Datentypen als Props übergeben: Primitive Werte (Zahlen, Strings, Booleans), Objekte, Arrays, Funktionen oder auch JSX-Elemente (z.B. mittels des speziellen Props children). children wird verwendet, um verschachtelten JSX-Inhalt an Komponenten zu übergeben (ähnlich einem Slot).

Zusammengefasst: **Props ermöglichen die Komponentenkommunikation in React.** Eltern liefern Daten und Verhalten an Kinder; die Kinder nutzen diese. Falls viele Komponenten tief verschachtelt Daten benötigen (was zu "Prop Drilling" führen würde), kann der Ansatz umständlich werden – hierfür bietet React den Context oder externe State-Management-Lösungen (wie wir sie in Kapitel 11 kennenlernen). Zunächst reicht uns aber der einfache Props-Mechanismus für die üblichen Parent-Child-Datenflüsse.

5. Styling mit klassischem CSS

React selbst legt nicht fest, wie die Styles der Komponenten definiert werden – man kann auf verschiedene Weisen stylen (Inline-Styles, CSS-Module, CSS-in-JS, etc.). In vielen Fällen ist jedoch der einfachste Ansatz, **klassische CSS-Dateien** zu verwenden.

Typischerweise erstellt man eine oder mehrere .css-Dateien und importiert sie in seine Komponenten oder in die App. Mit Vite ist z.B. meist eine index.css oder App.css vorhanden, die in main.tsx oder in der Hauptkomponente importiert wird:

```
import './App.css';

function App() {
  return (
    <div className="app-container">
      <h1 className="title">Meine React App</h1>
      <p className="intro">Willkommen zu meiner
Anwendung!</p>
    </div>
  );
}
```

Angenommen, in der **App.css** sind folgende Styles definiert:

```
/* App.css */
.app-container {
  text-align: center;
  padding: 20px;
}

.title {
  color: darkblue;
}

.intro {
  font-size: 1.2rem;
}
```

Beim Import einer CSS-Datei in einer React-Komponente (wie oben import './App.css') fügt Vite diese Styles global zur Seite hinzu. Die Elemente in der JSX nutzen dann das Attribut className (statt class, da class in JSX ein reserviertes Schlüsselwort ist) um CSS-Klassen zuzuweisen.

Im obigen Beispiel erhält das umgebende die Klasse app-container und damit die definierten Styles (Zentrierung und Padding). Die Überschrift und der Absatz bekommen ihre jeweiligen Klassen und dadurch die Farbe bzw. Schriftgröße gemäß der CSS-Regeln.

Wichtige Hinweise beim Styling in React:

- Es können alle normalen CSS-Features genutzt werden (Selektoren, Media Queries, etc.). Die Styles sind standardmäßig **global** wirksam. Um Konflikte zu vermeiden, empfiehlt es sich, Klassennamen eindeutig zu wählen oder z.B. CSS-Modules einzusetzen (Dateien mit .module.css, die nur lokal in der Komponente gelten). Im reinen "klassischen" Ansatz teilen sich aber alle Komponenten denselben globalen CSS-Namespace.
- Inline-Styles sind möglich über das style-Attribut in JSX, das ein Objekt mit CSS-Eigenschaften (in CamelCase-Schreibweise) erhält, z.B. <div style={{ backgroundColor: 'red' }}>.... Für umfangreiche Styles sind CSS-Klassen jedoch übersichtlicher und performanter.
- Man kann auch CSS-Präprozessoren (Sass, Less) oder Utility-CSS-Frameworks (z.B. Tailwind CSS) verwenden; das Prinzip bleibt gleich: Die React-Komponenten rendern normale HTML-Elemente mit className, und die Styles werden in externen Dateien definiert.

6. React Icons

Bei der Gestaltung einer Benutzeroberfläche möchte man oft Icons (Symbole) einbinden, etwa Pfeile, Sozialmedia-Logos, Menü-Icons etc. **React Icons** ist eine beliebte Bibliothek, die eine Vielzahl von Icon-Packs (Font Awesome, Material Design, Bootstrap Icons, Feather Icons u.v.m.) unter einer einheitlichen Schnittstelle vereint.

Installation: Füge die Bibliothek zum Projekt hinzu:

```
npm install react-icons
```

React Icons Auswählen: https://react-icons.github.io/react-icons/

Verwendung: Jedes Icon wird als React-Komponente exportiert. Man importiert das gewünschte Icon aus dem entsprechenden Paket innerhalb von react-icons.
Beispielsweise steht FaBeer für das Bier-Glas-Icon aus Font Awesome (fa steht hier für das FontAwesome-Paket).

```
import { FaBeer, FaSmile } from 'react-icons/fa';
import { MdCheckCircle } from 'react-icons/md';

function IconDemo() {
```

```
  return (
    <div>
      <h3>Icons Beispiel:</h3>
      <FaBeer style={{ color: 'orange', marginRight: '8px'
}} />
      <FaSmile color="green" size="2em" />
      <MdCheckCircle color="teal" size={24} />
    </div>
  );
}
```

Hier haben wir drei Icons importiert: zwei aus dem Font Awesome-Pack (react-icons/fa) und eines aus Material Design Icons (react-icons/md). Wir können sie wie JSX-Komponenten verwenden (<FaBeer />), und mittels Props oder CSS stylen:

- Das FaBeer Icon erhält einen Inline-Style mit oranger Farbe und etwas rechten Abstand.
- FaSmile bekommt direkt die Props color="green" und size="2em" (die meisten Icon-Komponenten akzeptieren zumindest size und color als Props).
- MdCheckCircle (Häkchen-Kreis Icon) bekommt color="teal" und size={24} (Pixel).

Man kann Icons auch über CSS-Klassen stylen, indem man ihnen z.B. eine className gibt und im CSS entsprechend anspricht. Da react-icons die Symbole als SVG-Grafiken einbindet, sind sie vektorbasiert und frei skalierbar ohne Qualitätsverlust.

Die Bibliothek erlaubt es, nur die benötigten Icons zu importieren (dank individueller ES6-Module). Das obige Beispiel lädt also nur die drei genannten Symbole ins Bundle, nicht die ganzen Icon-Bibliotheken. Auf der React Icons Webseite kann man durch alle verfügbaren Icons browsen und die Importnamen herausfinden.

7. API-Kommunikation mit Fetch und Axios

Webanwendungen müssen oft Daten von externen APIs laden oder an diese senden. In React kann man hierzu die Browser-API fetch nutzen oder die populäre Bibliothek **Axios**, die einige Komfortfunktionen bietet (z.B. einfacheres Auslesen von JSON-Antworten, zentrale Konfiguration von Headern, Abfangen von Fehlern, etc.).

Klassischer Ansatz mit fetch: Die folgende React-Komponente nutzt die nativen Browser-API fetch. Wir können in einem useEffect einen API-Aufruf mit fetch machen und die Daten in den Component State laden. Dabei verwalten wir typischerweise auch Lade- und Fehlerzustände manuell.

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

interface Post {
  id: number;
  title: string;
}

function PostList() {
  const [posts, setPosts] = useState<Post[]>([]);
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState<string | null>(null);

  useEffect(() => {
    setLoading(true);
    fetch('https://jsonplaceholder.typicode.com/posts')
      .then(response => {
        if (!response.ok) {
          throw new Error(`HTTP error! Status:
${response.status}`);
        }
        return response.json();
      })
      .then((data: Post[]) => {
        setPosts(data); // Daten im State speichern
      })
      .catch(err => {
        setError(err.message || 'Fehler beim Laden');
      })
      .finally(() => {
        setLoading(false);
      });
  }, []);

  if (loading) return <p>Lade Posts...</p>;
  if (error) return <p>Error: {error}</p>;

  return (
```

```
    <ul>
      {posts.map(post => (
        <li key={post.id}>{post.title}</li>
      ))}
    </ul>
  );
}

export default PostList;
```

Hier verwenden wir fetch innerhalb von useEffect (mit leerem Dependency-Array, d.h. nur einmal beim Mount). Solange loading true ist, zeigen wir einen Ladehinweis, bei Fehler eine Fehlermeldung, sonst die geladene Liste. . Dieses Muster (State für Daten, Loading und Error, plus useEffect für den Fetch) wiederholt sich bei vielen Komponenten

Klassischer Ansatz mit Axios: Wir können in einem useEffect einen API-Aufruf mit Axios machen und die Daten in den Component State laden. Dabei verwalten wir typischerweise auch Lade- und Fehlerzustände manuell. Zum Beispiel, um eine Liste von Posts zu laden:

```
import axios from 'axios';
import { useState, useEffect } from 'react';

interface Post {
  id: number;
  title: string;
}

function PostList() {
  const [posts, setPosts] = useState<Post[]>([]);
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState<string | null>(null);

  useEffect(() => {
    setLoading(true);
    axios.get('https://jsonplaceholder.typicode.com/posts')
      .then(response => {
        setPosts(response.data);     // Daten im State
speichern
      })
```

```
      .catch(err => {
        setError(err.message || 'Fehler beim Laden');
      })
      .finally(() => {
        setLoading(false);
      });
  }, []);

  if (loading) return <p>Lade Posts...</p>;
  if (error) return <p>Error: {error}</p>;
  return (
    <ul>
      {posts.map(post => <li
key={post.id}>{post.title}</li>)}
    </ul>
  );
}
```

Hier verwenden wir axios.get innerhalb von useEffect (mit leerem Dependency-Array, d.h. nur einmal beim Mount). Solange loading true ist, zeigen wir einen Ladehinweis, bei Fehler eine Fehlermeldung, sonst die geladene Liste. Dieses Muster (State für Daten, Loading und Error, plus useEffect für den Fetch) wiederholt sich bei vielen Komponenten.

8. Side Effects mit useEffect

In einer React-Komponente beschreibt der Render-Code (JSX) primär, *was* angezeigt wird. Doch oft müssen wir auch auf Ereignisse außerhalb des reinen Renderings reagieren – etwa **Daten laden**, einen **Timer starten** oder **manuell in den DOM eingreifen**. Solche Effekte außerhalb des direkten UI-Renders nennt man **Side Effects** (Seiteneffekte).

Für Funktionskomponenten bietet React den Hook useEffect, um Side Effects zu behandeln. Mit useEffect(effectFn, deps) können wir festlegen, dass effectFn ausgeführt wird, *nachdem* die Komponente gerendert wurde (bzw. wenn sich gewisse Abhängigkeiten geändert haben). Ein paar wichtige Punkte zu useEffect:

- Ohne zweites Argument wird der Effekt nach **jedem** Render ausgeführt (das ist selten gewollt).
- Mit einem leeren Abhängigkeitsarray [] wird der Effekt nur einmal nach dem initialen Mount ausgeführt (ähnlich wie componentDidMount bei Klassen).

- Wenn das Array bestimmte State-Variablen enthält, z.B. [count], wird der Effekt nach dem initialen Render **und** jedes Mal, wenn sich count ändert, ausgeführt.

Ein Anwendungsbeispiel: Wir möchten den Dokumenttitel der Webseite aktualisieren, sobald ein Zähler sich ändert:

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

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

  useEffect(() => {
    document.title = `Zähler: ${count}`;
    // Cleanup-Funktion optional: hier nicht notwendig
  }, [count]); // Effekt hängt von 'count' ab

  return (
    <div>
      <button onClick={() => setCount(count + 1)}>Klick
mich</button>
    </div>
  );
}
```

In diesem Beispiel bewirkt useEffect, dass nach jedem Render, bei dem sich count verändert hat, die angegebene Funktion ausgeführt wird. Hier setzt sie document.title auf z.B. "Zähler: 5". Würden wir [] als Abhängigkeitsarray nutzen, würde der Titel nur einmal beim ersten Laden gesetzt werden und danach nicht mehr aktualisiert.

Ein weiterer wichtiger Aspekt: die optionale **Clean-up-Funktion**. Wenn die in useEffect übergebene Funktion selbst wieder eine Funktion zurückgibt, wird diese Rückgabefunktion vor dem nächsten Effektaufruf oder beim Entladen der Komponente ausgeführt. Das wird z.B. für das Aufräumen von Ressourcen genutzt (Timer löschen, Event Listener entfernen). Beispiel:

```
useEffect(() => {
  const id = setInterval(() => console.log('Tick'), 1000);
  return () => clearInterval(id); // Aufräumaktion bei
Unmount oder Effekt-Neustart
}, []);
```

Hier startet der Effekt einen Interval-Timer beim ersten Render (leeres Dependenz-Array sorgt dafür, dass es nur einmal ausgeführt wird), und die zurückgegebene Funktion löscht

den Timer, wenn die Komponente unmounted wird (oder der Effekt neu ausgeführt würde, was bei [] nicht vorkommt).

Mit useEffect können wir also Nebenwirkungen kontrolliert einsetzen. Wichtig ist, die Dependenz-Liste korrekt anzugeben, damit Effekte in den richtigen Momenten ausgelöst werden und keine Endlosschleifen entstehen.

9. React Router für Routing

Eine Single-Page-Application soll dem Benutzer oft mehrere Seiten oder Ansichten bieten (z.B. Startseite, Über-Seite, Dashboard etc.), ohne dass die Seite komplett neu geladen wird. Hierfür verwenden wir **client-seitiges Routing**. Die de-facto Standardbibliothek für Routing in React ist **React Router** (aktuell Version 6). Sie ermöglicht es, Komponenten basierend auf der aktuellen URL zu rendern und die URL zu ändern, ohne die Seite neu zu laden.

Zunächst muss React Router installiert werden:

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

Danach kann man in der Anwendung das Routing einrichten. Typischerweise umschließt man die gesamte App mit einer Router-Komponente (meist BrowserRouter für die HTML5-History API). Innerhalb definiert man die Routen mittels Routes- und Route-Elementen. Hier ein einfaches Beispiel mit zwei Seiten:

```
import { BrowserRouter, Routes, Route, Link } from 'react-router-dom';

function Home() {
  return <h2>Willkommen auf der Startseite</h2>;
}

function About() {
  return <h2>Über uns</h2>;
}

function App() {
  return (
    <BrowserRouter>
      <nav>
        <Link to="/">Home</Link>
        {' | '}
        <Link to="/about">About</Link>
```

```
      </nav>

      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
      </Routes>
    </BrowserRouter>
  );
}
```

In diesem Code:

- <BrowserRouter> aktiviert die Router-Funktionalität (nutzt die Browser-History-API im Hintergrund).
- Die nav enthält <Link>-Komponenten statt normaler <a>-Tags. Ein <Link to="/about">About</Link> generiert ein klickbares Element, das die URL auf /about setzt, *ohne* die Seite neu zu laden (React Router übernimmt den Seitenwechsel intern mittels History API).
- Innerhalb von <Routes> definieren wir mit <Route> die Routen. Das Attribut path gibt den URL-Pfad an, element die zu rendernde Komponente. Im obigen Beispiel führt / zur Home-Komponente und /about zur About-Komponente.
- Wechselt man über die Links die URL, rendert React Router automatisch die entsprechende Komponente, ohne dass ein echter Seitenneuabruf stattfindet.

Man kann auch **dynamische Routen** definieren, z.B. path="/users/:userId". In der entsprechenden Komponente kann man mit dem Hook useParams() den Wert von :userId aus der URL auslesen. Für konditionale Weiterleitungen gibt es <Navigate> (z.B. um programmatic Redirects zu machen) und für komplexere Muster wie verschachtelte Routen das <Outlet>-Konzept. Diese erweiterten Themen kann man bei Bedarf in der React Router Dokumentation vertiefen.

Für uns wichtig ist: React Router ermöglicht es, eine React-App in logische Seiten aufzuteilen und Navigation zwischen ihnen umzusetzen, **ohne** den Komfort einer SPA (schnelle, flüssige Updates) zu verlieren.

10. Refs und useRef

Manchmal benötigen wir **direkten Zugriff auf DOM-Elemente** oder möchten einen Wert außerhalb des normalen React-Datenflusses speichern. Hier kommen **Refs** ins Spiel.

Eine Ref (Referenz) hält einen Wert, der über Render-Zyklen hinweg erhalten bleibt, ohne eine erneute Darstellung auszulösen. Refs werden häufig für zwei Zwecke genutzt:

1. **DOM-Referenzen:** Zugriff auf ein konkretes DOM-Element, um z.B. Fokus zu setzen, Text auszuwählen oder Maße zu nehmen.
2. **Persistente Werte:** Speichern von veränderlichen Werten, die beim Ändern **kein** Re-Render auslösen sollen (z.B. eine Timer-ID, der vorherige Zustand, ein Render-Zähler).

In Funktionskomponenten erzeugen wir Refs mit dem Hook `useRef`. Dieser liefert ein Objekt mit der Eigenschaft `.current`. Anfangs setzen wir `useRef` mit einem Initialwert.

Beispiel 1: eine Eingabefeld-Fokussierung:

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

function TextInputWithFocus() {
  const inputRef = useRef<HTMLInputElement>(null);

  const handleFocus = () => {
    inputRef.current?.focus(); // Falls current nicht null
ist, Fokus setzen
  };

  return (
    <div>
      <input type="text" ref={inputRef} placeholder="Name
eingeben" />
      <button onClick={handleFocus}>Focus Input</button>
    </div>
  );
}
```

Hier wird inputRef als Ref erstellt und dem <input>-Element über das JSX-Attribut ref={inputRef} zugewiesen. Dadurch schreibt React beim Rendern das DOM-Element (ein HTMLInputElement) in inputRef.current. Der Button-Handler kann dann über inputRef.current auf das Input-Feld zugreifen und die Methode .focus() darauf ausführen – das Eingabefeld erhält den Fokus, wenn der Button geklickt wird.

Beispiel 2: Refs zum Speichern eines Wertes über Render-Zyklen. Angenommen, wir wollen in einer Komponente zählen, wie oft sie gerendert wurde:

```
function RenderCounter() {
  const renderCount = useRef(0);
```

```
  renderCount.current += 1;

  return <p>Ich wurde {renderCount.current} mal
gerendert.</p>;
}
```

Bei jedem Render erhöht die Komponente renderCount.current. Weil das Ändern von renderCount.current **kein** State-Update ist, löst es kein erneutes Rendering aus – der Zähler erhöht sich nur, wenn ohnehin ein Render stattfindet (z.B. ausgelöst durch einen Elternstate). So kann man einen Wert persistieren, ohne den Render-Zyklus zu beeinflussen.
Wichtig: Wenn sich eine Ref ändert (d.h. .current einen neuen Wert bekommt), führt dies **nicht** automatisch zu einem Re-Render der Komponente. Refs sollten deshalb nicht benutzt werden, um *Anzeigen* zu steuern – dafür ist State da. Refs sind aber ideal, um imperativ mit dem DOM zu arbeiten oder nicht-darstellungsrelevante Infos zu speichern.

11. State Management mit useReducer

Wenn der **State** einer Komponente komplexer wird – etwa wenn mehrere verwandte Werte gemeinsam aktualisiert werden müssen oder viele verschiedene Aktionen auf denselben State wirken –, stößt useState schnell an seine Grenzen.
Der React-Hook **useReducer** stellt hier eine elegantere, „redux-ähnliche“ Alternative bereit. Anstatt mehrere setState-Aufrufe zu verteilen, definieren wir *eine* zentrale **Reducer-Funktion**, die bestimmt, **wie** sich der State als Reaktion auf **Aktionen** (Actions) verändert. Dadurch bleiben Logik und Datenfluss übersichtlich und vorhersagbar.

11.1 Grundidee und Signatur

```
const [state, dispatch] = useReducer(reducer, initialState,
init?);
```

Parameter	Bedeutung
• reducer	• Rein funktionale Reducer-Funktion (state, action) → newState
• initialState	• Anfangszustand (beliebiger Typ)
• init *(optional)*	• *Lazy*-Initialisierungsfunktion init(rawInitial) → actualInitial – wird nur beim ersten Render ausgeführt

11.2 Einsteigerbeispiel – Zähler mit mehreren Aktionen

```
import { useReducer } from 'react';

/* ---------- State- und Action-Typen ---------- */
type CounterState = { count: number };
```

```
type CounterAction =
  | { type: 'increment' }
  | { type: 'decrement' }
  | { type: 'reset'; payload: number };

/* ---------- Reducer ---------- */
function counterReducer(state: CounterState, action:
CounterAction): CounterState {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    case 'reset':
      return { count: action.payload };
    default:
      return state;            // sollte bei exhaustiver
Typprüfung nie erreicht werden
  }
}

/* ---------- Komponente ---------- */
export default function Counter() {
  const [state, dispatch] = useReducer(counterReducer, {
count: 0 });

  return (
    <div>
      <p>Zähler: {state.count}</p>

      <button onClick={() => dispatch({ type: 'decrement'
})}>-1</button>
      <button onClick={() => dispatch({ type: 'increment'
})}>+1</button>
      <button onClick={() => dispatch({ type: 'reset',
payload: 0 })}>Reset</button>
    </div>
  );
```

```
}
```

Warum useReducer hier sinnvoll ist

- Mehrere Aktionen wirken auf denselben Wert.
- Die State-Update-Logik ist zentral (statt in drei separaten setCount-Aufrufen).
- Durch die *Union-Type* CounterAction kann TypeScript exhaustive checking leisten – vergessene Aktionen fallen beim Kompilieren auf.

11.3 Praxisbeispiel– Todo-Liste mit komplexerem State

```
import { useReducer, ChangeEvent, FormEvent, useState }
from 'react';

/* ---------- Typen ---------- */
interface Todo {
  id: number;
  text: string;
  done: boolean;
}

type TodoState = Todo[];

type TodoAction =
  | { type: 'add'; text: string }
  | { type: 'toggle'; id: number }
  | { type: 'remove'; id: number };

/* ---------- Reducer ---------- */
function todoReducer(state: TodoState, action: TodoAction):
TodoState {
  switch (action.type) {
    case 'add':
      const newTodo: Todo = {
        id: Date.now(),
        text: action.text,
        done: false,
      };
      return [...state, newTodo];

    case 'toggle':
```

```
      return state.map(todo =>
        todo.id === action.id ? { ...todo, done: !todo.done
} : todo,
      );

    case 'remove':
      return state.filter(todo => todo.id !== action.id);

    default:
      return state;
  }
}

/* ---------- Komponente ---------- */
export default function TodoApp() {
  const [todos, dispatch] = useReducer(todoReducer, []);
  const [input, setInput] = useState('');

  const handleSubmit = (e: FormEvent) => {
    e.preventDefault();
    if (input.trim()) {
      dispatch({ type: 'add', text: input.trim() });
      setInput('');
    }
  };

  return (
    <section>
      <h2>Meine Todos</h2>

      <form onSubmit={handleSubmit}>
        <input
          value={input}
          onChange={(e: ChangeEvent<HTMLInputElement>) =>
setInput(e.target.value)}
          placeholder="Neues Todo"
        />
```

```
        <button type="submit">Hinzufügen</button>
      </form>

      <ul>
        {todos.map(todo => (
          <li key={todo.id}>
            <label style={{ textDecoration: todo.done ?
'line-through' : 'none' }}>
              <input
                type="checkbox"
                checked={todo.done}
                onChange={() => dispatch({ type: 'toggle',
id: todo.id })}
              />
              {todo.text}
            </label>
            <button onClick={() => dispatch({ type:
'remove', id: todo.id })}>X</button>
          </li>
        ))}
      </ul>
    </section>
  );
}
```

Vorteile des Reducer-Ansatzes in diesem Beispiel

- **Einziges** State-Objekt für eine Liste, obwohl drei Aktionen (add/toggle/remove) existieren.
- Klar getrennte Verantwortlichkeiten:
  - UI-Elemente lösen **nur** dispatch aus.
  - Der Reducer beschreibt rein *funktional*, wie sich der State ändert.
- Durch switch-Cases plus Union-Typ sind Updates **typsicher** und leicht nachvollziehbar.

11.5 `useReducer` vs. `useState`

Szenario	Hook-Empfehlung
• Einzelne, unabhängige Werte	• `useState`
• Mehrere abhängige Werte, die ***immer gemeinsam*** geändert werden	• `useReducer` (ein Objekt)
• Viele unterschiedliche Aktionen auf denselben State	• `useReducer`

12. State-Management mit useContext

In den vorangegangenen Kapiteln haben wir gesehen, dass **Props** der Standardweg sind, Daten von einer Eltern- zur Kindkomponente zu übergeben. Wird die Komponenten-hierarchie jedoch tief – etwa App → Layout → Page → Card → Button –, entsteht schnell so-genanntes **Prop Drilling**: Zwischenkomponenten leiten Werte weiter, die sie selbst gar nicht benötigen.
Hier setzt die **React Context API** an. Ein **Context** stellt Daten *global* (innerhalb eines Teilbaums) bereit, ohne dass jede Zwischenebene die Props manuell weiterreichen muss. Der Hook **useContext** ermöglicht dabei den bequemen Zugriff auf diese geteilten Zustände innerhalb von Funktionskomponenten.

12.1 Grundbegriffe

Begriff	Aufgabe
Context-Objekt	Wird mit React.createContext<T>() erzeugt und enthält einen *default value*
Provider	<MyContext.Provider value={…}> umschließt einen Teilbaum und liefert den aktuellen Wert
Consumer	Hook useContext(MyContext) gibt in jeder darunter liegenden Komponente den vom nächsten Provider gesetzten Wert zurück

Wichtig: Ein Context ist **kein** State-Management-System perse; er verteilt lediglich Daten. Wie diese Daten aktualisiert werden (z.B. mittels useState, useReducer oder externer Bibliotheken) liegt bei uns.

12.2 Ein einfaches Beispiel – Theme Context

Angenommen, wir möchten das Farbschema („light“/ „dark“) unserer App zentral verwalten.

```
/* ---------- theme-context.tsx ---------- */
import { createContext, useContext, useState, ReactNode }
from 'react';

/* 1) Typen definieren */
type Theme = 'light' | 'dark';

interface ThemeContextValue {
  theme: Theme;
```

```
  toggle: () => void;
}

/* 2) Context anlegen – Platzhalterwert nur für TypeScript */
const ThemeContext = createContext<ThemeContextValue | undefined>(undefined);

/* 3) Provider-Komponente */
export function ThemeProvider({ children }: { children: ReactNode }) {
  const [theme, setTheme] = useState<Theme>('light');

  const toggle = () => setTheme(prev => (prev === 'light' ? 'dark' : 'light'));

  return (
    <ThemeContext.Provider value={{ theme, toggle }}>
      {children}
    </ThemeContext.Provider>
  );
}

/* 4) Custom Hook für bequemeren Zugriff */
export function useTheme() {
  const ctx = useContext(ThemeContext);
  if (!ctx) throw new Error('useTheme must be used inside <ThemeProvider>');
  return ctx;
}
```

Verwendung in der App:

```
/* ---------- main.tsx ---------- */
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App';
import { ThemeProvider } from './theme-context';
```

```
ReactDOM.createRoot(document.getElementById('root') as
HTMLElement).render(
  <React.StrictMode>
    <ThemeProvider>
      <App />
    </ThemeProvider>
  </React.StrictMode>,
);
```

Und in einer beliebigen Kindkomponente:

```
/* ---------- ThemeSwitcher.tsx ---------- */
import { useTheme } from './theme-context';

export function ThemeSwitcher() {
  const { theme, toggle } = useTheme();
  return (
    <button onClick={toggle}>
      Aktuelles Theme: {theme} (klicken zum Umschalten)
    </button>
  );
}
```

Alle Komponenten unterhalb von <ThemeProvider> erhalten dieselbe theme-Instanz, ohne Props weiterzureichen.

12.6 Zusammenfassung

- **useContext** ermöglicht komponentenübergreifendes Teilen von Daten ohne Prop-Drilling.
- Ein **Provider** liefert den Wert, useContext liest ihn in beliebigen Tiefen.

Damit verfügen wir nun über die wichtigsten Werkzeuge, um sowohl **lokalen** (useState), **komplexen** (useReducer) als auch **globalen** State (useContext) elegant zu verwalten.

13. State-Management-Bibliotheken: Jotai und Zustand

Für globale Zustände, die von vielen Komponenten geteilt werden müssen, stößt man mit einfachem Prop-Drilling oder React Context manchmal an Grenzen. Klassisch wurde hier oft Redux eingesetzt, doch es gibt modernere, einfachere Alternativen. Zwei solcher **State-Management-Bibliotheken** sind **Jotai** und **Zustand**. Beide verfolgen unterschiedliche Ansätze, sind aber leichtergewichtig als Redux und funktionieren gut mit Hooks.

13.1 Jotai

Jotai: (Das Wort bedeutet "Atom" auf Japanisch) – entsprechend basiert die Bibliothek auf **atomaren Zuständen**. Ein *Atom* repräsentiert ein Stück Zustand (z.B. ein boolescher Wert, eine Zahl, ein Objekt). Mit dem Hook useAtom(atom) kann jede Komponente diesen Zustand lesen **und** schreiben. Jedes Atom kann als globaler State betrachtet werden (sofern es nicht durch einen speziellen Provider auf einen Teilbaum beschränkt wird). Beispiel: Wir erstellen einen Zähler-State als Atom und verwenden ihn in zwei Komponenten:
Installation von Jotai:

```
npm install jotai
```

Beispiel für die Verwendung:

```
import { atom, useAtom } from 'jotai';

// Atom definieren
const countAtom = atom(0);

function CounterDisplay() {
  const [count] = useAtom(countAtom);
  return <h3>Count: {count}</h3>;
}

function CounterControls() {
  const [, setCount] = useAtom(countAtom);
  return <button onClick={() => setCount(c => c +
1)}>+1</button>;
}
```

Hier definiert atom(0) ein Zahlen-Atom mit Initialwert 0. In CounterDisplay verwenden wir useAtom(countAtom), um den aktuellen Wert zu erhalten (und ignorieren den Setter). In CounterControls erhalten wir ebenfalls Zugriff auf das Atom, nutzen aber nur den Setter, um bei Button-Klick den Wert zu erhöhen. Beide Komponenten teilen sich also den Zustand **countAtom**. Ändert eine Komponente den Wert, sehen es alle anderen, die dasselbe Atom nutzen. Jotai optimiert Renderings dahingehend, dass nur Komponenten neu rendern, die tatsächlich das geänderte Atom verwenden.

Jotai besticht durch seine Einfachheit: Kein Boilerplate, jeder Hook arbeitet direkt mit dem gewünschten Teil des States. Man kann Atome voneinander ableiten (z.B. ein Atom, das doppelt so hoch ist wie ein anderes) und asynchrone Atome definieren (für das Laden von Daten). In der Grundnutzung reicht oft das Beispiel oben: Atome definieren und in beliebigen Komponenten via useAtom darauf zugreifen.

13.2 Zustand

Zustand: ("Zustand" ist das deutsche Wort für State) – hier implementiert als *zentraler Store*. Zustand verzichtet auf den Einsatz des React Contexts oder vordefinierte Actions. Stattdessen erstellt man mit der Funktion create() einen zustandsbehafteten Store (außerhalb von React-Komponenten), und greift innerhalb der Komponenten über einen Hook darauf zu. Beispiel: ein simpler Counter-Store mit Zustand:
Installation von Zustand:

```
npm install zustand
```

Beispiel für die Verwendung:

```
import create from 'zustand';

interface CounterState {
  count: number;
  increment: () => void;
  reset: () => void;
}

const useCounterStore = create<CounterState>((set) => ({
  count: 0,
  increment: () => set(state => ({ count: state.count + 1
})),
  reset: () => set({ count: 0 })
}));

function Counter() {
  const { count, increment, reset } = useCounterStore();
  return (
    <div>
      <p>Zähler: {count}</p>
      <button onClick={increment}>+1</button>
      <button onClick={reset}>Reset</button>
    </div>
  );
}
```

In diesem Beispiel definieren wir mit create einen Store mit Zustand-Variablen (count) und Methoden, die den Zustand verändern (increment, reset). Der Custom Hook useCounterStore verhält sich ähnlich wie ein useState, kann aber von beliebigen Komponenten verwendet werden – alle teilen sich den einen Store. Jeder Aufruf von

increment aktualisiert den Store und löst Re-Renders in allen Komponenten aus, die den relevanten State nutzen. (Standardmäßig abonnieren Komponenten, die useCounterStore() ohne Argument aufrufen, alle State-Änderungen. Man kann optional auch mit Selektor-Funktionen useCounterStore(state => state.count) nur auf Teilzustände abonnieren, um unnötige Re-Renders zu vermeiden.)

12.3 Wann sollte man solche Libraries einsetzen?

Wenn ein Zustand von vielen Komponenten genutzt werden muss oder über viele Ebenen weitergereicht werden müsste (Prop Drilling), lohnt sich üblicherweise ein globales State-Management. Jotai und Zustand sind zwei von vielen möglichen Lösungen (weitere wären z.B. Redux Toolkit, MobX, XState, Recoil etc.). Für kleine Apps reicht oft der React Context oder das Weiterreichen via Props. Aber bei komplexeren Apps helfen diese Libraries, den Code übersichtlicher zu halten und Zustände zentral zu managen.

14. Erweiterte API Kommunikation mit ReactQuery

React Query (TanStack Query): Diese Bibliothek abstrahiert das Laden von Daten und das Caching. Anstatt für jede Datenladung den obigen Boilerplate-Code zu schreiben, können wir React Query verwenden, das uns Hooks wie useQuery bietet. Vorteile:

- Automatisches **Loading- und Error-Handling**: useQuery liefert u.a. Zustände wie isLoading, error und die Ergebnis-data.
- **Caching** und Refetching: Geladene Daten werden zwischengespeichert. Bei erneutem Aufruf (z.B. Wechsel auf die Seite und zurück) muss nicht unbedingt neu geladen werden – man kann definieren, wann Daten als "stale" gelten. Man kann auch unkompliziert Refresh-Intervalle oder manuelles Neuladen auslösen.
- **Synchronisation und Mutationen**: Wenn Daten verändert werden (z.B. via useMutation für POST/PUT/DELETE), kann React Query zugehörige Queries automatisch invalidieren und neu laden.

Um React Query zu nutzen, installiert man es und umschließt die App mit einem QueryClientProvider:

```
import { QueryClient, QueryClientProvider } from
'@tanstack/react-query';


const queryClient = new QueryClient();


ReactDOM.createRoot(document.getElementById('root')!).rende
r(
  <React.StrictMode>
    <QueryClientProvider client={queryClient}>
      <App />
    </QueryClientProvider>
  </React.StrictMode>
);
```

Dann können wir in Komponenten den useQuery Hook einsetzen. Das obige Post-Beispiel ließe sich so vereinfachen:

```
import { useQuery } from '@tanstack/react-query';
import axios from 'axios';

function PostList() {
  const { data: posts, isLoading, error } =
useQuery(['posts'], async () => {
    const res = await
axios.get('https://jsonplaceholder.typicode.com/posts');
    return res.data;
  });

  if (isLoading) return <p>Lade Posts...</p>;
  if (error) return <p>Fehler beim Laden.</p>;
  return (
    <ul>
      {posts?.map(post => <li
key={post.id}>{post.title}</li>)}
    </ul>
  );
}
```

Beachte: useQuery benötigt einen Query-Key (hier ['posts']) und eine asynchrone Funktion, die die Daten liefert (hier via Axios GET). Es gibt uns posts, isLoading und error. Wir müssen keine eigenen useState-Hooks für diese Zustände definieren – React Query handhabt das. Zusätzlich würde React Query die posts cachen; würde man die Komponente verlassen und später wieder öffnen, zeigt es entweder sofort die gecachten Posts oder lädt im Hintergrund neu (je nach Konfiguration).
Mit React Query lassen sich auch Mutationen (Daten senden) über useMutation abwickeln, und man kann Feintuning betreiben (Refetch-Intervalle, Stale-While-Revalidate-Strategien, etc.). Insgesamt reduziert es den Code für API-Kommunikation und sorgt für eine konsistente Behandlung von Lade- und Fehlerzuständen.
In vielen Fällen reicht für simple Dinge schon Axios oder fetch in einem useEffect, aber bei komplexeren Anwendungen mit viel Server-State ist React Query sehr hilfreich, um den Überblick zu behalten und unnötigen Code zu vermeiden.

15. Custom Hooks

Zusätzlich zu den vordefinierten Hooks können wir in React auch **eigene Hooks (Custom Hooks)** erstellen. Ein Custom Hook ist im Grunde nichts weiter als eine JavaScript/TypeScript-Funktion, die selbst Hooks aufruft. Die einzige formale Voraussetzung: Der Funktionsname muss mit "use" beginnen, damit React erkennt, dass

es sich um einen Hook handelt (und die Regel "Hooks nur auf Top-Level aufrufen" entsprechend geprüft wird). Custom Hooks dienen vor allem der **Wiederverwendbarkeit von Logik**. Wenn mehrere Komponenten dieselbe Logik benötigen (z.B. einen bestimmten Zustand und Effekte), kann man diese Logik in einen Custom Hook auslagern, statt sie zu duplizieren.
Ein einfaches Beispiel ist ein Hook zum Umschalten eines booleschen Wertes – nennen wir ihn useToggle. Dieser Hook kapselt einen useState-Aufruf und liefert einen bequemen Toggle-Handler zurück:

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


function useToggle(initial: boolean = false): [boolean, ()
=> void] {
  const [value, setValue] = useState(initial);
  const toggleValue = () => setValue(v => !v);
  return [value, toggleValue];
}
```

Hier haben wir einen Hook useToggle definiert, der einen booleschen State verwaltet. Er nimmt optional einen Initialwert (Standard false) und gibt ein Tupel zurück: den aktuellen Wert und eine Funktion, um den Wert umzuschalten. Intern verwendet useToggle den normalen useState Hook.
Die Verwendung eines Custom Hooks ist genauso wie bei einem Built-in Hook:

```
function ToggleComponent() {
  const [isVisible, toggleIsVisible] = useToggle(false);


  return (
    <div>
      <button onClick={toggleIsVisible}>
        {isVisible ? 'Verstecken' : 'Anzeigen'}
      </button>
      {isVisible && <p>Dieser Text wird ein- und
ausgeblendet.</p>}
    </div>
  );
}
```

In ToggleComponent benutzen wir useToggle. Jedes Mal, wenn der Button geklickt wird, rufen wir toggleIsVisible() auf, was den isVisible-Wert umkehrt. Der Absatz mit dem Text wird abhängig von isVisible gerendert oder entfernt. Custom Hooks können beliebig komplex sein: Man kann mehrere Hooks darin kombinieren (z.B. einen State und einen Effect), Parameter akzeptieren und Werte oder Funktionen zurückgeben. Wichtig ist, dass auch in Custom Hooks die Hooks-Regeln gelten (nur auf oberster Ebene innerhalb des Hooks aufrufen, nicht in Schleifen oder Bedingungen). Durch Custom Hooks erreicht

man sauberen, wiederverwendbaren Code und kann die Komplexität in einzelne Logikbausteine aufteilen.

16 Projektstruktur und Best Practices

16.1 Projektstruktur

Eine gut organisierte Projektstruktur erleichtert Wartung und Zusammenarbeit. Es gibt nicht "die eine" richtige Struktur für alle Projekte, aber hier sind einige bewährte Ansätze für React mit TypeScript:

```
my-app/
├── public/          # Statische Assets und index.html
└── src/
    ├── components/ # Wiederverwendbare (UI-)Komponenten
    ├── pages/      # Seiten-Komponenten (für Routing)
    ├── hooks/      # Custom Hooks
    ├── context/    # React Contexts für globale Zustände
    ├── services/   # Externe Dienste (API-Aufrufe, Utils)
    ├── types/      # Zentrale TypeScript-Typdefinitionen
    ├── App.tsx     # Hauptkomponente
    ├── main.tsx    # Einstiegspunkt (ReactDOM.createRoot)
    └── index.css   # Globale CSS-Styles (ggf. weitere CSS-
Dateien)
```

Kleinere Projekte kommen mit weniger Ordnern aus. Wichtig ist, dass verwandte Dinge gruppiert sind. Z.B. können Komponenten auch nach Features gruppiert werden (ein Ordner pro Feature mit Unterordnern für Komponenten, Styles, Tests).

16.2 Best Practices:

- **Benennung:** Komponenten-Dateien benennt man nach der Komponente (z.B. LoginForm.tsx enthält die LoginForm-Komponente). Komponentennamen im PascalCase, Hooks mit Prefix *use* (useAuth.ts mit useAuth() Hook).
- **Kleine, fokussierte Komponenten:** Es ist besser, mehrere kleine Komponenten zu haben, als eine riesige Komponente mit zu vielen Verantwortlichkeiten. Kleine Komponenten sind wiederverwendbar und einfacher zu verstehen und zu testen.
- **Eine Komponente pro Datei:** Das ist üblich (außer bei sehr kleinen Hilfs-Subkomponenten), so entspricht der Dateiname der Komponente. Das erleichtert das Auffinden und Importieren.
- **TypeScript konsequent nutzen:** Definiere für Props eigene Interfaces oder Type Aliases (oder verwende React.FC<PropType>). Vermeide any – nutze stattdessen Union-Typen, Generics oder Utility-Typen von TypeScript, um Typen präzise auszudrücken. Aktiviere strict Mode in der tsconfig (bei Vite-TS-Projekten meist standardmäßig aktiv).

- **Linting & Formatting:** Nutze Tools wie **ESLint** (mit passenden Plugins für React/TS) und **Prettier**, um konsistente Code-Qualität sicherzustellen. Linter finden viele Fehler (z.B. vergessene Abhängigkeiten in useEffect oder ungenutzte Variablen) bereits während des Entwickelns. Formatierer sorgen für einheitlichen Stil (Einrückungen, Anführungszeichen, etc.).
- **React-spezifisch:**
  - Verwende immer sinnvolle **Key-Props** bei Listen (.map-Renderings), damit React im Diffing die Elemente eindeutig identifizieren kann. Ohne keys kann es zu ineffizienten Re-Renders oder Bugs kommen.
  - **Vermeide direkte DOM-Manipulation** in Komponenten (wie document.getElementById oder manuelles Klassen-Hinzufügen). Solche Dinge sollten entweder über React (State/Props) gesteuert oder über Refs gehandhabt werden. React hält die Kontrolle über den DOM; externe Manipulation kann zu Inkonsistenzen führen.
  - **Events und Funktionen:** Arrow-Function-Eventhandler in JSX (onClick={() => ...}) sind bequem, erzeugen aber bei jedem Render eine neue Funktion. Das ist meist unproblematisch, kann aber in Einzelfällen Performance kosten. Wenn nötig, definiere Funktionen außerhalb/oberhalb der Komponente oder nutze useCallback, um die Referenz stabil zu halten.
  - Achte auf **zustandslose Komponenten**: Wenn eine Komponente keinen eigenen State oder keine Hooks braucht, kann sie ruhig eine reine Ausgabe-Komponente sein (d.h. einfach eine Funktion, die von Props abhängt). Solche Komponenten können ggf. mit React.memo umhüllt werden, um unnötige Re-Renders zu vermeiden.
- **Trennung von Logik und Darstellung:** Halte Komponenten möglichst *präsentationsorientiert*. Geschäftslogik oder aufwändige Berechnungen sollten in Dienste/Utilities (z.B. in services/ oder Hilfsfunktionen) oder in Custom Hooks ausgelagert werden. Die Komponente sollte sich primär um die Darstellung kümmern. Das macht sie einfacher verständlich und testbar.
- **Kommentare und Dokumentation:** Bei komplexerer Logik in Komponenten oder Hooks sind Kommentare nützlich. Beschreibe in JSDoc-Kommentaren die wichtigen Funktionen, Hook-Rückgabewerte oder Prop-Typen, damit andere Entwickler (oder du selbst in ein paar Wochen) die Absicht schneller nachvollziehen können.

17. useMemo und useCallback zur Performanceoptimierung

Beim Rendern einer React-Komponente wird die Funktionskomponente vollständig ausgeführt – d.h. auch alle Berechnungen in der Render-Phase und alle Funktionsdefinitionen werden bei jedem Render neu erstellt. In vielen Fällen ist das kein Problem, aber manchmal führt es zu unnötigem Aufwand:

- Eine aufwändige Berechnung (z.B. große Schleifen oder komplexe Datenverarbeitungen) wird bei jedem Render neu ausgeführt, obwohl sich die Eingabedaten gar nicht geändert haben.
- Eine Funktion wird bei jedem Render neu definiert, was für React so aussieht, als wäre es eine andere Funktion (neues Objekt im Speicher). Wenn diese Funktion als Prop an Kind-Komponenten weitergereicht wird, könnten dadurch diese Kinder unnötig neu gerendert werden, selbst wenn sich ihre sonstigen Props nicht geändert haben.

Für solche Fälle bietet React zwei Hooks: **useMemo** und **useCallback**. Beide *memoisieren* (also cachen) etwas zwischen Render-Vorgängen, solange sich ihre Abhängigkeiten nicht geändert haben:

- useMemo(fn, deps) berechnet den Rückgabewert von fn nur dann neu, wenn eine der Abhängigkeiten in deps sich geändert hat. Es liefert den zwischengespeicherten Wert zurück.
- useCallback(fn, deps) ist ähnlich, liefert aber die **Funktion** selbst zurück (bzw. bei unveränderten deps dieselbe Funktionsinstanz), anstatt deren Rückgabewert.

Ein Beispiel verdeutlicht den Unterschied. Angenommen wir haben ein Array, das wir sortiert benötigen:

```
import { useMemo, useCallback } from 'react';

const values = [3, 9, 6, 4, 2, 1];

const sortedValues = useMemo(() => {
  console.log('Sorting array...');
  return [...values].sort();
}, [values]);

const getSortedValues = useCallback(() => {
  console.log('Sorting array...');
  return [...values].sort();
}, [values]);

console.log(sortedValues);     // Ausgabe: [1, 2, 3, 4, 6, 9]
console.log(getSortedValues()); // Ausgabe: [1, 2, 3, 4, 6, 9]
```

Hier würde sortedValues per useMemo nur einmal berechnet und für folgende Renders gespeichert, solange values gleich bleibt. getSortedValues ist eine memoisierte Funktionsreferenz, die bei unverändertem values jedes Mal dieselbe Funktion zurückliefert. Beide würden erst neu berechnet/erstellt, wenn sich an values etwas ändert. (Zur Verdeutlichung haben wir Konsolenausgaben eingebaut; man würde sehen, dass "Sorting array..." seltener erscheint als Renders erfolgen.)

In der Praxis nutzt man useMemo vor allem, um *teure Berechnungen* zu vermeiden, die bei unveränderten Eingaben nicht jedes Mal neu durchgeführt werden sollen. useCallback wird hauptsächlich eingesetzt, um Funktionen zu stabilisieren, die an optimierte Kind-Komponenten weitergegeben werden. Oft in Kombination mit React.memo – einem Helfer, der Funktionskomponenten nur neu rendert, wenn sich Props geändert haben. Ist eine Callback-Prop dank useCallback referenziell stabil, erkennt React.memo, dass sich diese Prop nicht geändert hat, und kann ein Re-Render des Childs überspringen.
Ein einfaches Anwendungsbeispiel für useCallback ist das Verhindern unnötiger Neu-Renderings einer Kind-Komponente:

```
const handleClick = useCallback(() => {
  console.log('Button clicked');
}, []);
```

Hier bleibt handleClick über alle Renders hinweg gleich (leeres Dependency-Array, also keine Abhängigkeiten, die sich ändern könnten), sodass diese Funktion z.B. an ein Child übergeben werden kann, ohne dass das Child bei jedem Render des Elternteils eine neue Funktionsreferenz sieht.
Wichtig: Sowohl useMemo als auch useCallback sind **Optimierungswerkzeuge**. Sie können in manchen Fällen die Performance verbessern, aber sie fügen selbst auch etwas Komplexität und minimalen Overhead hinzu. Man sollte sie gezielt einsetzen, wenn tatsächlich wiederholte teure Berechnungen oder unnötige Re-Renders auftreten. In vielen Fällen ist React auch ohne diese Hooks schnell genug. Aber wenn deine Anwendung bei bestimmten Interaktionen langsam wirkt, können useMemo/useCallback (neben anderen Techniken) helfen, Engpässe zu beheben.

18. Performanceoptimierung in React

React ist von Haus aus relativ performant durch Virtual DOM und effizientes Rendering. Oft muss man keine speziellen Optimierungen vornehmen. Dennoch gibt es Situationen, in denen man nachjustieren kann. Hier eine Zusammenfassung wichtiger Performance-Tipps:

- **Produktions-Build verwenden:** Im Entwicklungsmodus ist React absichtlich etwas langsamer (wegen vieler Warnungen und Hilfen). Stelle sicher, dass für Live-Betrieb und Performance-Tests immer der Produktionsbuild genutzt wird (bei Vite: npm run build, dann die gebaute App serven). Der Produktionsbuild ist minifiziert und enthält keine Debug-Warnungen.

- **Unnötige Re-Renders vermeiden:** Achte auf die Komponentenstruktur. Wenn eine Elternkomponente neu rendert, rendert standardmäßig auch ihr gesamter Kindbaum neu. Das ist meistens okay und dauert nur Millisekunden. Falls es zum Problem wird, kann React.memo helfen: Damit kann man eine Komponente *memoisieren*, sodass sie bei gleichen Props den vorherigen Output wiederverwendet und nicht neu rendert. Beispiel:

```
const MyComponent = React.memo(function MyComponent({ value
}) {
  console.log('render');
  return <p>{value}</p>;
});
```

MyComponent rendert jetzt nur noch, wenn sich value ändert (das console.log würde also seltener auftreten). **Hinweis:** React.memo führt eine flache Prop-Vergleich durch. Für komplexe Prop-Objekte kann man eine eigene Vergleichsfunktion übergeben.

- **Gezielt Hooks einsetzen:** Wie in Kapitel 7 erläutert, helfen useMemo und useCallback dabei, teure Berechnungen bzw. Funktionsinstanzen zwischenzuspeichern. Setze diese Hooks gezielt ein, wenn eine Berechnung wirklich merklich Performance frisst oder ein Callback bei jedem Render unnötig neu erstellt wird und viele Child-Komponenten betrifft. Andernfalls sind sie nicht nötig und bringen keine spürbaren Vorteile.
- **Code-Splitting / Lazy Loading:** Lade große Komponenten oder Module erst, wenn sie gebraucht werden. React bietet React.lazy und <Suspense> für dynamisches Nachladen. Dadurch wird die initiale Bundle-Größe verringert und die App startet schneller. Beispiel:

```
import React, { Suspense } from 'react';
const HeavyComponent = React.lazy(() =>
import('./HeavyComponent'));

function App() {
  return (
    <Suspense fallback={<div>Lädt...</div>}>
      <HeavyComponent />
    </Suspense>
  );
}
```

Hier wird HeavyComponent (z.B. eine große, selten gebrauchte Komponente) erst geladen, wenn App diese tatsächlich rendern möchte. Bis dahin zeigt <Suspense> einen Lade-Indikator. So wird unnötiger JavaScript-Code nicht von Anfang an geladen.

- **Lange Listen optimieren:** Wenn du sehr lange Listen (Hunderte/ Tausende Elemente) darstellst, kann das Rendern und Updaten der Liste langsamer werden. Techniken wie **Virtualisierung** (z.B. mit Libraries wie react-window oder react-virtualized) rendern nur die gerade sichtbaren Listenelemente im DOM, anstatt alle auf einmal. Das reduziert die Anzahl der DOM-Elemente drastisch und verbessert Performance und Speicherverbrauch.
- **Teure Operationen aus dem Render auslagern:** Vermeide es, im JSX/Render große Rechenlogik auszuführen. Falls du z.B. eine aufwändige Filterung oder Sortierung machen musst, führe sie außerhalb des return (oder in useMemo) aus, anstatt direkt im JSX (wo sie bei jedem Render ausgeführt würde). So bleibt die Render-Methode leichtgewichtig.

- **Profiling & Messen:** Nutze die React Developer Tools (insbesondere den **Profiler**) um herauszufinden, welche Komponenten oft und lange rendern. Performance-Optimierung sollte möglichst *datengetrieben* sein – optimiere primär dort, wo Engpässe messbar sind. Oft stellt sich heraus, dass gefühlte Probleme gar nicht von Re-Renders kommen, sondern z.B. von langsamen API-Antworten oder Bildern – dann setzt man dort an.

Quellenverzeichnis

Kategorie	Literaturangabe
React	React Documentation. *React – Main Concepts*. Verfügbar unter https://react.dev/learn
Vite	vitejs.dev. *Vite Guide –React+TypeScript*. Verfügbar unter https://vite.dev/guide/
TypeScript	Microsoft, *TypeScript 5.4 Handbook*. https://www.typescriptlang.org/docs/
React Router	Remix Software. *React Router v6 Documentation*. https://reactrouter.com
Axios	Axios Project. *Axios 1.6 API Reference*. https://axios-http.com
React Icons	React Icons. *Icon-Library Docs*. https://react-icons.github.io/react-icons
Fachbuch	*Schwarzmüller,M. (2025). React Key Concepts: An in-depth guide to React's core features (2.Aufl.) Packt Publishing. Packt*
Fachbuch	*Barklund,M.&Mardan, A.(2023). React Quickly (2.Aufl.) Manning Publications*