

# React Basic

# Tobias Kaiser



# Tobias Kaiser



Senior Consultant

> 8 Jahre Entwickler

~ 3 Jahre React

Berater, Entwickler, **Trainer**



# Agenda

## **1. TypeScript**

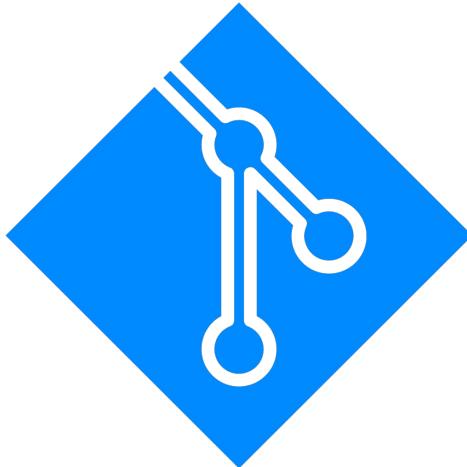
- TypeScript Basics
- Typen, Variablen und Funktionen
- Arbeiten mit Arrays und Interfaces
- Module und Asynchronität

## **2. Grundlagen React**

- Einstieg in React
- Grundlagen von React
- React anwenden - Übungsprojekt
- Testen von React-Komponenten
- Anbindung einer Web-API

## **3. Fragen und Weiteres**

# **Repository zum Kurs**



Repository klonen von:  
<https://github.com/TobiasKNterra/react-basic>

# TypeScript

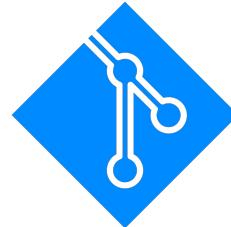
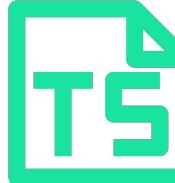
A screenshot of a Java code editor showing a large watermark 'TypeScript' in white, semi-transparent font across the center of the screen. The code in the editor is related to Azure Functions and includes imports for Microsoft Azure Functions and Java utilities, as well as annotations like @FunctionName and @AuthorizationLevel. The code handles requests from an HTTP trigger and returns responses based on query parameters.

# TypeScript Basics



# Was ist TypeScript?

- ECMAScript mit statischer Typisierung
- TypeScript wird nicht direkt ausgeführt, sondern zu JavaScript kompiliert
- Entwicklung & Support von Microsoft
- Open Source unter der Apache-2-Lizenz



# Warum TypeScript?

- JavaScript ist dynamisch typisiert
- Zuweisung von Werten an Variablen haben keine typbasierten Einschränkungen
- Schlechte Wartbarkeit:
  - Refactorings
  - Auto-Completion
  - Code-Navigation

# TypeScript und ECMAScript

- **2009: ECMAScript 5**
  - Zielplattform sind ältere Browser
- **2015: ECMAScript 6**
  - Module, Klassen, Arrow-Funktionen etc.
  - Generator Functions, Async/Await etc.
- **TypeScript**
  - Erweitert ECMAScript um statische Typisierung
  - Beinhaltet Sprachfeatures neuerer Versionen



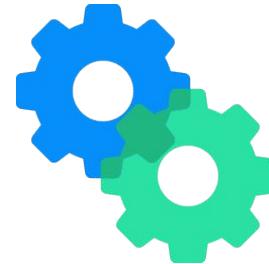
# TypeScript Projekt starten

1. Öffne den Ordner `ts-project` in deiner IDE.
2. Navigiere dein Terminal in den gleichen Ordner und führe `npm install` und `npx tsc` aus.
3. Es sollte eine Datei `js/index.js` erstellt worden sein.
4. Führe diese mit `node js/index.js` aus.

# Typen, Variablen und Funktionen



# Typisierung



## Variablen:

- **JavaScript:** let name = "Text";
- **TypeScript:** let name: string = "Text";

## Funktionen:

- **JavaScript:** function sum(a, b) { ... }
- **TypeScript:** function sum(a: number, b: number): number { ... }

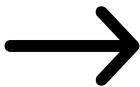
# Basis Typen

```
let aNumber: number;  
aNumber = 10;  
  
let aString: string;  
aString = "Text";  
  
let aBoolean: boolean;  
aBoolean = true;  
  
let anArray: any[]  
anArray = ["eins", "zwei", 3];  
  
let anotherArray: Array<any>  
anotherArray = ["eins", "zwei", 3];
```

- **number** - Fließkomma- oder Ganzzahlen
- **string** - Textdaten
- **boolean** - True/False
- **array** - Array

# Basis Typen

```
let aNumber: number;  
aNumber = 10;  
  
let aString: string;  
aString = "Text";  
  
let aBoolean: boolean;  
aBoolean = true;  
  
let anArray: any[]  
anArray = ["eins", "zwei", 3];  
  
let anotherArray: Array<any>  
anotherArray = ["eins", "zwei", 3];
```



```
let aNumber;  
aNumber = 10;  
  
let aString;  
aString = "Text";  
  
let aBoolean;  
aBoolean = true;  
  
let anArray;  
anArray = ["eins", "zwei", 3];  
  
let anotherArray;  
anotherArray = ["eins", "zwei", 3];
```





# Funktionen und Variablen

1. Erstelle eine Variable vom Typ string und weise ihr einen Text zu.
2. Erstelle eine Funktion, die ein Argument vom Typ string entgegennimmt.
3. Die Funktion soll den übergebenen Wert auf der Konsole mit console.log(...) ausgeben.
4. Rufe die Funktion mit der erstellten Variable auf.
5. Kompiliere den Code und führe ihn aus.

# Basis Typen

```
let anObject: object;
anObject = {a:"Ein Text", b:5};

let aTuple: [string, number];
aTuple = ["Ein Text", 5];

// Compile Error
aTuple = [5,"Ein Text"];

enum Color {
  Red,
  Green,
  Blue,
}

let c: Color = Color.Green;
```

- object - Nicht primitive Typen
- tuple - Array mit einer festen Größe und bekannten Typen
- enum - Namen für Zahlen

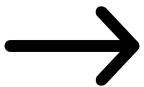
# Basis Typen

```
let anObject: object;
anObject = {a:"Ein Text", b:5};

let aTuple: [string, number];
aTuple = ["Ein Text", 5];

// Compile Error
aTuple = [5,"Ein Text"];

enum Color {
  Red,
  Green,
  Blue,
}
let c: Color = Color.Green;
```



```
let anObject;
anObject = { a: "Ein Text", b: 5 };

let aTuple;
aTuple = ["Ein Text", 5];
// Compile Error
aTuple = [5, "Ein Text"];

var Color;
(function (Color) {
  Color[Color["Red"] = 0] = "Red";
  Color[Color["Green"] = 1] = "Green";
  Color[Color["Blue"] = 2] = "Blue";
})(Color || (Color = {}));

let c = Color.Green;
```



# **undefined & null**

- **null** und **undefined** sind primitive Typen
- **null** - Etwas ist gerade nicht verfügbar
- **undefined** - Etwas wurde nicht initialisiert

```
let value: number | undefined;

const stringValue: string | undefined = value?.toString()

if(value)
{
    const stringValue: string = value.toString();
}
```

# Union Type

```
let aUnion: number | string;  
  
aUnion = "Ein Text";  
aUnion = 5;  
  
function aFunction (a: number | string)  
{  
    console.log(a);  
}  
  
aFunction(5);  
aFunction("Ein Text");
```

- Union Types sind Vereinigung von zwei Typen
  - In der Signatur über “|” verbunden
  - Beide Typen sind zuweisbar
  - Nutzbar sind die Gemeinsamkeiten beider Typen



# Funktionen und Variablen

1. Erstelle eine Funktion, die als Argument entweder eine Variable vom Typ `string` oder vom Typ `number` entgegennimmt.
2. Rufe die Funktion mit beiden Möglichkeiten auf.
3. Gib den Wert der Variable auf der Konsole aus mit `console.log`
4. Führe die Funktion mit `node js/index.js` aus.

# Union Type

```
let aUnion: "value1" | "value2";  
  
aUnion="value1";  
aUnion="value2";  
  
// Compile Error  
aUnion="Ein anderer text";  
  
const aFunction = (a: "value1" | "value2") =>  
  console.log(a);  
  
aFunction("value1");  
  
//Compile Error  
aFunction("Ein anderer Text");
```

- Unions können zur Definition von Schnittstellen verwendet werden
- Mögliche Werte werden mit “|” getrennt
- Compile Error, wenn ein anderer Wert zugewiesen wird



# Funktionen und Variablen

1. Erstelle eine Funktion, die als Argument nur “hello” oder “world” entgegennimmt.
2. Rufe die Funktion mit beiden Möglichkeiten auf.
3. Gib den Wert der Variable auf der Konsole aus mit `console.log`
4. Führe die Funktion mit `node js/index.js` aus.

# Type Inferencing & any

Typen können explizit angegeben werden

```
let einText: string = "Text"  
einText = 5; //Compile Error
```

Inferred Typangabe

```
let einName = "Name";  
einName = 5; //Compile Error
```

any

```
let einWort: any = "Wort";  
einWort = 5; // ok
```

Typen bei Funktionen

```
// 'a' und 'b' sind beide Type any  
// Ist Ok, aber mit Warnung  
  
function anyArguments(a, b:any) {};
```

# **var & let**

- **var und let** definieren jeweils eine Variable in JavaScript

# var & let

- var und let definieren jeweils eine Variable in JavaScript

```
let a = 1;  
let b = 2;  
  
if (true) {  
    let a = 98;  
    b = 99;  
}  
  
console.log(a);  
console.log(b);
```

```
var a = 1;  
var b = 2;  
  
if (true) {  
    var a = 98;  
    b = 99;  
}  
  
console.log(a);  
console.log(b);
```



# var & let

1. Führe das vorangegangene Beispiel aus.
2. Was wird auf der Konsole ausgegeben?



# var & let

- **var und let** definieren jeweils eine Variable in JavaScript
- var ist funktionsbezogen
- let ist blockbezogen

```
let a = 1;
let b = 2;

if (true) {
  let a = 98;
  b = 99;
}

console.log(a); //1
console.log(b); //99
```

```
var a = 1;
var b = 2;

if (true) {
  var a = 98;
  b = 99;
}

console.log(a); //98
console.log(b); //99
```

# Kontanten - const

- Konstanten können mit **const** deklariert werden
- Sind als unveränderlich markiert
- Blockbezogen wie bei **let**

```
const a = 5;  
  
// Compile Error  
a = 10;
```

```
const anObject = { a: 5};  
  
// Ok  
anObject.a = 10;  
anObject["a"] = 12;
```

# Template-Strings

- Werden durch ” ` “ (Backtick) geöffnet und geschlossen
- Multiline-Strings
- Variablen in Strings ersetzen (String-Interpolation)

```
let text: string = `Hallo ich bin ein Template-String!`;
```

```
let text: string = `Hallo ich bin ein  
Template-String in zwei Zeilen!`;
```

```
let aNumber:number = 5;
```

```
let text: string = `Nr ${aNumber} lebt!`;
```



# Template Strings

1. Erstelle eine Funktion, die ein Argument vom Typ **string** entgegennimmt.
2. Der übergebene Wert soll in folgenden **string** eingebettet werden:  
“Hallo Welt, [*Hier Parameter*] !”
3. Der erstellte **string** soll mit **console.log** auf der Konsole ausgegeben werden.
4. Rufe die Funktion mit einem Text auf.

# Arrow-Funktionen

- Kürzere Syntax zur Definition von Funktionen
  - `(a: number,b: number) => {...}`
  - ‘{}’-Klammern und return Statement sind optional, wenn Funktionsbody nur aus einer Anweisung besteht:
    - `(a: number) => a+1`

```
function sum(a: number) {  
    return a+1;  
}
```

```
const sum = (a: number) => {  
    return a+1;  
}
```

```
const sum = (a: number) => a+1;
```

# Generics

- TypeScript beinhaltet Generics

```
interface Array<T> {
    reverse(): T[];
    sort(compareFn?: (a: T, b: T) => number): T[];
    map<U>(func: (value: T, index: number, array: T[]) => U): U[];
}
```

# Arbeiten mit Arrays und Interfaces



# Arbeiten mit Arrays

- Über die Elemente eines Arrays kann iteriert werden
- Ein Array kann sortiert und gefiltert werden
- Zugriff über den Index oder **at**-Methode (ES2022)
- Weitere Funktionen: **push**, **map**, **forEach**

```
let anArray: number[] = [1, 2, 3];  
  
anArray.push(4);      // [1, 2, 3, 4]
```

```
let anArray: Array<number> = [1, 2, 3];  
  
anArray[1]; // 2  
anArray.at(1); // 2
```

# Arbeiten mit Arrays - Map

JS

- Elemente eines Arrays lassen sich mit der Map-Methode umwandeln
- **Map** erwartet als Argument eine Funktion

```
let anArray: number[] = [1, 2, 3];

let listOfStrings: string[] = anArray.map<string>((v: number) => v.toString()); // ["1","2","3"]
```

# Arbeiten mit Arrays - Filter

JS

- Elemente eines Arrays lassen sich mit der **Filter-Methode** eingrenzen
- **Filter** erwartet als Argument eine Funktion

```
const anArray: number[] = [1, 2, 3, 4]

const filtered: number[] = anArray.filter((v: number) => v > 2); // [3, 4]
```

# Arbeiten mit Arrays - forEach

- Über die Elemente eines Arrays kann mit der forEach-Methode iteriert werden
- **forEach** erwartet als Argument eine Funktion

```
let list: number[] = [1, 2, 3];

list.forEach((v: number) => console.log(v.toString()));

/*
"1"
"2"
"3"
*/
```

# Interfaces

- Mit interfaces können Strukturen mit einem Namen beschrieben werden
- Interface existieren nur in TypeScript und dienen nur zur Typprüfung
- Es wird kein JavaScript Code kompiliert

```
interface Person {  
  name: string,  
  nr: number,  
  pets: string[],  
  isDeveloper(): boolean  
}
```

```
const person: Person = {  
  name:"Tobias",  
  nr:12,  
  pets:["cat", "dog"],  
  isDeveloper: () => true  
}
```



# Interfaces und Zuweisung

1. Definiere das Interface **Person** und erstelle ein **Objekt**, das dieses Interface erfüllt.
2. Erstelle eine Funktion, die ein Argument vom Typ **Person** entgegennimmt.
3. Gib alle Werte der Charakteristik **Person.pets** mit **console.log** auf der Konsole aus.
4. Falls die Person ein Entwickler ist, soll als Prefix vor jede Ausgabe ein “**Entwickler -** “ gestellt werden.
5. Führe die Funktion aus.

# Optionale Parameter bei Funktionen

- Keine Überladung von Funktionen möglich
- Parameter können daher oft optional sein
- Parameter werden mit ‘?‘ als optional gekennzeichnet
- Alternativ kann mit “=“ auch ein Standardwert zugewiesen werden

```
function withOptionalParameters(a:number, b?:number, c: number = 3)
{
    ...
}

withOptionalParameters(2);

withOptionalParameters(2,5);

withOptionalParameters(2,5,10);
```

# Optionale Parameter bei Interfaces

- Eigenschaften von Interfaces können als optional gekennzeichnet werden
- Optionale Eigenschaften werden mit einem ‘?’ gekennzeichnet

```
interface Person {  
    name: string,  
    nr: number,  
    pets?: string[],  
    isDeveloper: () => boolean  
}  
  
const person: Person = {  
    name:"Tobias",  
    nr:12,  
    isDeveloper:() => true  
}
```



# Optionale Parameter

1. Passe das Interface **Person** an, sodass pets optional ist.
2. Erstelle ein **Objekt**, das dieses Interface erfüllt.
3. Erstelle eine Funktion, die ein Argument vom Typ **Person** entgegennimmt.
4. Gib alle Werte der Charakteristik Person.pets mit **console.log** auf der Konsole aus, **falls diese vorhanden ist**.
5. Wenn keine pets definiert sind, soll der Text “No Pets!” ausgegeben werden.
6. Führe die Funktion aus.

# Strukturelle Typisierung

- Typ-System von TypeScript ist strukturell
  - Typen sind kompatibel, wenn deren Struktur gleich ist
  - Name bzw. Bezeichnung des Typs spielt keine Rolle
- Java's Typensystem ist nominal
  - Der Name und die Deklaration des Typs müssen übereinstimmen

```
interface Foo {  
    name: string,  
    nr: number  
}  
  
interface Bar {  
    name: string,  
    nr: number,  
}  
  
let fooVar: Foo = {name:"Name", nr:5};  
let barVar: Bar = {name:"Text", nr:3};  
  
// Ok, weil die Typen strukturell gleich  
// sind  
fooVar = barVar;
```

# Strukturelle Typisierung

- Typ-System von TypeScript ist strukturell
  - Typen sind kompatibel, wenn deren Struktur gleich ist
  - Name bzw. Bezeichnung des Typs spielt keine Rolle
- Java's Typensystem ist nominal
  - Der Name und die Deklaration des Typs müssen übereinstimmen

```
interface Foo {  
    name: string,  
    nr: number,  
    isFoo: boolean  
}  
  
interface Bar {  
    name: string,  
    nr: number,  
}  
  
let fooVar: Foo = {name:"Name", nr:5, isFoo:true};  
let barVar: Bar = {name:"Text", nr:3};  
  
// Compile Error  
fooVar = barVar;  
  
// Ok  
barVar = fooVar
```

# Spread Operator ‘...’

- Der Spread-Operator wird mit ‘...’ genutzt
- Das Zielobjekt wird “ausgebreitet”
- Kann auf Objekte und Arrays angewendet werden

```
const numbers = [1, 2, 3, 4];
const newArray = [...numbers, 5, 6, 7];

console.log(newArray);
// Output: [1, 2, 3, 4, 5, 6, 7]
```

```
interface Foo {
  name: string,
  nr: number,
  isFoo: boolean
}

interface Bar {
  name: string,
  nr: number,
}

let barVar: Bar = {name:"Text", nr:3};
let fooVar: Foo = {...barVar, isFoo:true};
```

# Module und Asynchronität



# Module

- Sammlung von Code, Variablen etc. unter einem Namen
- **Kapselung** (höher als Klassen und Funktionen)
- Module können andere Module referenzieren
- Singletons
- Jede Datei ist ein Modul
  - Voraussetzung ist ein Import- oder Export-Statement
- Import-Pfad kann relativ oder global sein
  - Letzteres muss konfiguriert werden

# Named Export

- Top Level-Variablen und/oder Funktionen können exportiert werden
- Werden unter ihrem Namen exportiert

```
//module.ts

export const variable1: number = 42;
export const variable2: string = 'Ein Text';

export function AFunction() {...}
```

```
import {variable1, variable2, aFunction} from
"./module.ts";
```

# Default Export

- Module können **einen default Export** angeben
- Name ist beim Import frei wählbar
- Es sind keine '{ }' Klammern beim Import nötig

```
//module.ts  
default export const variable1: number = 42;
```

```
import meineVariable from "./module.ts";
```

# Default & Named Exports

- Ein Modul kann **named exports** und **default exports** haben
- Es sind mehrere **named exports** möglich, aber nur ein **default export**

```
//module.ts
export const variable1: number = 42;
export const variable2: string = 'Ein Text';

export default function aFunction() { ... }
```

```
import eineFunktion, {variable1, variable2}
from "./module";
```



# Module

1. Erstelle ein Modul, welches eine Funktion als Default-Export und das Interface Person als Named-Export bereitstellt.
2. Die Default-Funktion soll als Argument ein Objekt vom Typ Person entgegennehmen.
3. Importiere die Funktion und das Interface in der index.ts
4. Führe die Funktion aus.
5. Es soll die gleiche Ausgabe auf der Konsole ausgegeben werden, wie in den vorangegangen Übungen.

# Asynchronität mit Promise

- Ein *Promise* repräsentiert das Ergebnis einer asynchronen Operation
- Der Wert einer solchen Operation wird zu einem späteren Zeitpunkt nachgeliefert
- Ein *Promise* kann sich in folgenden Zuständen befinden:
  - *Wartend (pending)* - Initialer Zustand, weder *erfüllt* noch *abgewiesen*
  - *Erfüllt (fulfilled)* - Die Operation wurde erfolgreich beendet
  - *Abgewiesen (rejected)* - Die Operation ist fehlgeschlagen
- Kann mit **then** oder **await** aufgelöst werden

# Asynchronität mit Promise

JS

- Eine Funktion kann ein Objekt vom Typ **Promise** zurückgeben.
- oder die Funktion ruft eine **asynchrone Operation** auf und **muss** dann mit **async** gekennzeichnet werden und gibt automatisch ein Objekt vom Typ **Promise** zurück.

```
const returnsAPromise = (): Promise<number> => {
    return new Promise<number>(...);
}
```

```
const returnsAPromise = async function(): Promise<number>
{
    ... // Hier asynchrone Operation
    return 3;
}
```

# Asynchronität mit Promise

JS

```
const promisedValue: Promise<number> = returnsAPromise();
```

```
promisedValue.then((value:number) => {
    //Do something with value
}).catch((error: Error) => {
    //Error handling
})
```

```
try {
    const value: number = await returnsAPromise();
    //Do something with value
}
catch (error)
{
    //Error handling
}
```



# Asynchronität mit Promise

1. Nutze die Funktionen unter `src/promise/promise.ts`, um jeweils ein **Promise** zu erzeugen.
2. “Löse die Promises auf” mit **then** und **catch**.
3. Gib die Ergebnisse der Promises auf der Konsole aus mit **console.log**.

# Grundlagen React

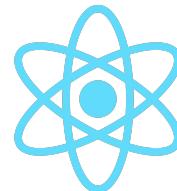
# Grundlage React

# Einstieg in React



# Was ist React?

- Eine JavaScript Bibliothek zur Erstellung von webbasierten Benutzeroberflächen
- Wurde 2011 von Facebook (heute Meta) entwickelt
- Wird seit 2013 als Open-Source-Projekt weitergeführt



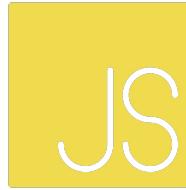
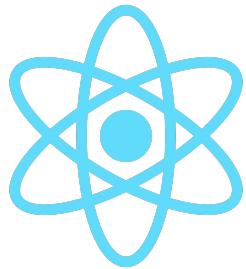
# Was ist React?

- Single Page Application
- Komponentenbasierte Entwicklung von Oberflächen
- Kann verwendet werden für
  - Webanwendungen
  - Mobile Apps
  - Fullstack-Anwendungen
- Großes Ökosystem an Modulen
- Komponenten werden in einer Baumstruktur angeordnet
- Unidirektonaler Informationsfluss

# Warum nutzen wir React?

- React ist nicht unbedingt “nötig”
- Alles kann mit JavaScript realisiert werden
- Nachteile von JavaScript gegenüber React:
  - Komplexer Code
  - Fehleranfällig
  - Wartungsintensiv
  - React ist “einfacher”

# Warum nutzen wir React?



# Was werden wir machen?

- Entwicklung von Komponenten
- Styling
- Statemanagement
- Routing
- Testen von Komponenten
- Anbindung einer Web-API

# Werkzeuge

- Vite als build Werkzeug
  - Webserver zur Entwicklung
  - mit *Rollup* einen Bundler für den späteren JavaScript Code
- TypeScript
- JSX (JavaScript Syntax Extension)
  - Syntaktischer Zucker in der Entwicklung
  - HTML-ähnliche Tags



# Anwendung starten

- Terminal im Projektverzeichnis öffnen (react-project)
- `yarn install`
  - Installiert alle notwendigen Abhängigkeiten
- `yarn run dev`
  - Startet den Entwicklungsserver und die Anwendung



Getting started

# Anwendung starten

Welcome to  
**React Basic**



# Projektstruktur

- node\_modules
  - Installierte Pakete bzw. Abhängigkeiten
- src
  - Arbeitsplatz
  - Komponenten, hooks, Modelle und Logik der Anwendung
- public
  - Dateien die an den Browser ausgeliefert werden bspw. Bilder
- tests
  - Konfiguration der Testumgebung
- .eslintrc.cjs
  - Konfiguration von ESLint
  - Statische Codeanalyse
- index.html
  - Einstiegspunkt unserer Anwendung
- Readme.md
  - Readme
- package.json
  - Essentielle Informationen für das Projekt
  - Definiert Abhängigkeiten
  - Projekt Scripts
- vite.config.js
  - Beschreibt das Verhalten der Entwicklungsumgebung

# Grundlagen React



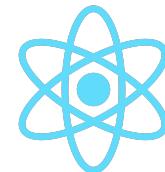


# Einen neuen Button hinzufügen

1. Füge der Anwendung einen neuen Button hinzu.
2. Der Text des Buttons soll “Klick mich!” lauten.
3. Der Button soll unter der Überschrift und über dem React-Logo zu sehen sein.

# React DOM Komponenten

- Komponenten für Formulare:
  - <input>
  - <select>
  - <textarea>
- Ressourcen und Metadaten Komponenten
  - <link>
  - <meta>
  - <script>
  - <style>
  - <title>
- Alle HTML Komponenten



# JSX - JavaScript Syntax Extension

```
<div className="halloWelt">  
  Hallo Welt  
</div>
```



```
react.createElement(  
  "div",  
  { className: "halloWelt" },  
  "Hallo Welt");
```

# React Komponente

- Eine Funktion, die ein JSX Element als Rückgabewert definiert
- ...

# React Komponente

- Eine Funktion, die ein JSX Element als Rückgabewert definiert
- ...

```
export default function EigeneKomponente() {  
  return <div> Hallo </div>;  
}
```



# Eine eigene Komponente hinzufügen

1. Erstelle einen Unterordner **components** in **src**.
2. Erstelle einen Unterordner **CustomButton** im Verzeichnis **src/components**.
3. Lege eine neue Datei **CustomButton.tsx** an.
4. Die neue Komponente soll den gleichen Button erzeugen, den wir in **App.tsx** genutzt haben und diesen ersetzen.
5. Wichtig: CustomButton muss in App.tsx importiert werden.

# Styling

- Das Styling einer Komponente erfolgt mit CSS
- Styles können mit einer .css Datei genutzt werden
  - Styles werden global angewendet
  - `import "./EigeneKomponente.css";`
  - `<h1 className="css-class"> Text </h1>;`
- Styles können mit einer .module.css Datei genutzt werden
  - Styles werden nur lokal in der Komponente genutzt
  - `import styles from "./EigeneKomponente.module.css"`
  - **styles** ist ein Objekt, auf das zugegriffen werden kann:
    - `<h1 className={styles.h1}> Text </h1>;`
- Über className kann der Komponente CSS-Klassen zugeordnet werden



# Unsere Komponente stylen

1. Kopiere die **.css** und die **.module.css** Dateien aus **styles/CustomButton** nach **scr/components/CustomButton**.
2. Weise der Eigenschaft `className` des Buttons die CSS-Klassen “button” zu.
3. Nutze die Styles aus der **.css** Datei und dann aus der **.module.css** Datei, um den Button anzupassen.
4. Was fällt dir im DOM auf?

# Daten in einer Komponente anzeigen

- JSX ermöglicht es, dass wir HTML im JavaScript/TypeScript verwenden
- Über '{ }' lässt sich Programmcode *escapen*

```
return <h1>{eineTextVariable}</h1>;
```



# Button-Text variabel setzen

1. Lege in der **CustomButton**-Komponente eine lokale Variable **buttonText** vom Typ **string** an.
2. Weise dieser Variable den Text ‘**Click me!**’ zu.
3. Nutze den Text zur Beschriftung des Buttons.

# Konditionelles Rendern

- Es gibt keine spezielle Syntax für konditionelles Rendern
- Wird mit JavaScript bzw. TypeScript umgesetzt



```
<div>  
  
{isLoggedIn ?  
  <AdminPanel /> : <LoginForm/>}  
  
</div>;
```

```
let content;  
  
if (isLoggedIn) {  
  content = <AdminPanel />;  
} else {  
  content = <LoginForm />;  
}  
  
return <div>{content}</div>;
```

```
<div>{isLoggedIn &&  
  <AdminPanel />}</div>
```



# Konditionelles Rendern

1. Lege in der App-Komponente eine boolsche Variable **withButton** an.
2. Unser CustomButton soll nur angezeigt werden, wenn die Variable auf **true** gesetzt ist.
3. Probiere mehrere Möglichkeiten aus.

# Events

- Ein Event ist ein **Ereignis und/oder eine Aktion**
- Ein Event wird zur **Laufzeit** im Browser “abgefeuert”
- Events sind eine JavaScript-Mechanik
- Beispiele für Events:
  - Der Nutzer klickt auf eine Schaltfläche
  - Ein Tastendruck auf der Tastatur
  - Die Website hat das Laden beendet
- Events werden durch **Eventhandler oder auch Eventlistener** verarbeitet
- Eventhandler werden meistens mit führendem **on** gekennzeichnet, wie:
  - **onKeypress**
  - **onClick**
  - **onResize**

# Auf Events reagieren

- Ein Button löst beim Klicken das Event **click** aus
- Im JSX-Element **<button>** kann zur Verarbeitung dieses Events eine **onClick-Methode** registriert werden

```
return (
  <button onClick={() =>
    console.log("Ich wurde geklickt!")}>
    Click me!
  </button>
);
```

```
const onClick = () => {
  console.log("Ich wurde geklickt!");
}

return <button onClick={onClick}>Click
me!</button>;
```



# Events

1. Definiere in **CustomButton.tsx** eine Funktion **onClick**.
2. Diese Funktion soll den Text “Ich wurde geklickt!” auf der Konsole ausgeben.
3. Rufe diese Methode beim Klicken des Buttons auf.

# Parameter einer Komponente - *Props*

- Informationsaustausch von der **Eltern-Komponente** zu ihren **Kindern**
- Ähnlichkeit zu HTML-Attributen, aber es können jede JavaScript/TypeScript Werte übergeben werden
- Wir haben schon **className** kennengelernt

```
<div className="css-Klasse"> Hallo Welt </div>
```

# Parameter einer Komponente - *Props*

```
interface KomponenteProps {  
  text: string;  
}  
  
export default function Komponente(props: KomponenteProps) {  
  return <h1>{props.text}</h1>;  
}
```

```
<Komponente text={"Hallo Welt"} />
```

# Parameter einer Komponente - *Props*

```
interface KomponenteProps {  
  text: string;  
  onClick: () => void;  
}  
  
export default function Komponente(props: KomponenteProps) {  
  return <button onClick={props.onClick}>{props.text}</button>;  
}
```

```
<Komponente  
  text="Hallo Welt"  
  onClick={() => console.log("Hallo Welt!")}>  
</Komponente>
```

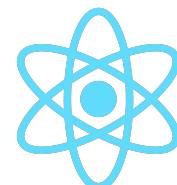


# Parameter einer Komponente - *Props*

1. Definiere ein **Interface** für unseren **CustomButton** in **CustomButton.tsx**.
2. Es soll die **Beschriftung** des Buttons enthalten sein.
3. Rufe den **CustomButton** in **App.tsx** mit der Beschriftung “Click me!” auf.

# Hooks

1. Über *Hooks* lassen sich verschiedene React-Funktionalitäten in einer Komponente nutzen
2. Bereits “eingebaute” React-*Hooks* sind z.B. useState und useEffect
3. “Eingebaute” Hooks lassen sich zu eigenen Hooks kombinieren
4. Vergleichbar mit der Komposition von Klassen



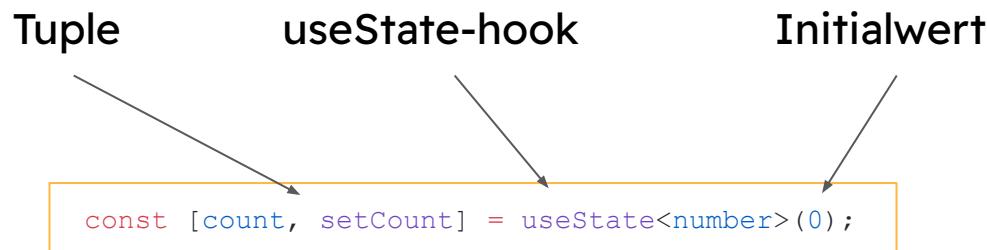
# Zustand einer Komponente - **States**

- Die Variablen, die in einer Komponente initialisiert werden, sind nach der *return*-Anweisung verloren
- Eine Komponente kann sich beim nächsten Rendern nur an Werte “erinnern”, wenn diese als *States* definiert wurden
- Ein *State* wird mit dem *useState-hook* erstellt:

```
const [count, setCount] = useState<number>(0);
```

# Zustand einer Komponente - States

- *States* können nicht konditionell erstellt werden
- *States* sollten immer am Anfang einer Komponente initialisiert werden und **müssen auf dem Top-Level der Funktion definiert werden**



# Zustand einer Komponente - *States*

- *States* werden beim **mounten** mit dem angegebenen Initialwert initialisiert
- Bleiben über die Dauer mehrerer Render-Cyclen gültig



**ACHTUNG:** Der Wert wird am Ende eines Render-Cyclus aktualisiert.

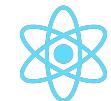
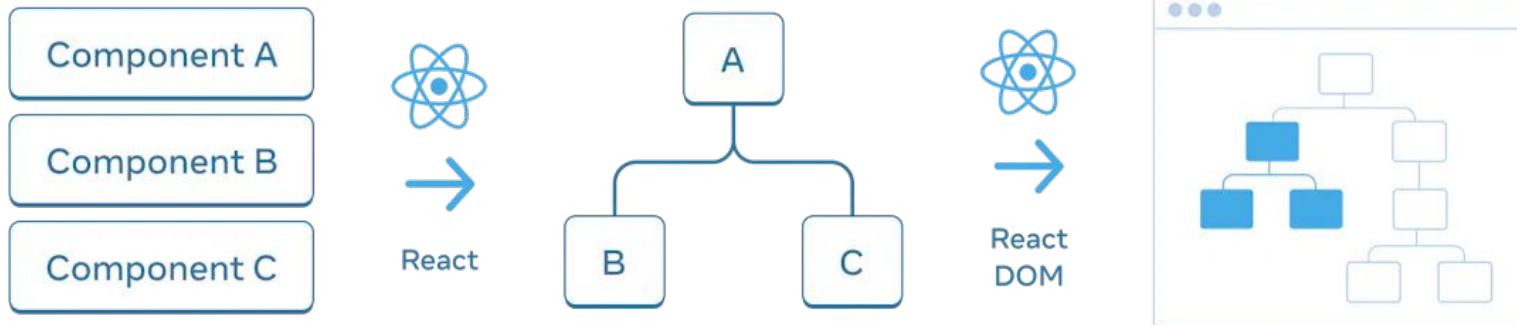
Nicht währenddessen!



# Button mit Zustand

1. Unser **CustomButton** soll um einen **State** vom Typ **number** erweitert werden.
2. Der **State** soll mit 0 initialisiert werden.
3. Dieser **State** soll bei jedem Klicken des Buttons hochgezählt werden.
4. Zusätzlich soll die Beschriftung des Buttons um den aktuellen Zählerstand erweitert werden: **{Beschriftung} {Zähler hier}**.

# Unsere UI ist eine Baumstruktur

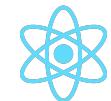
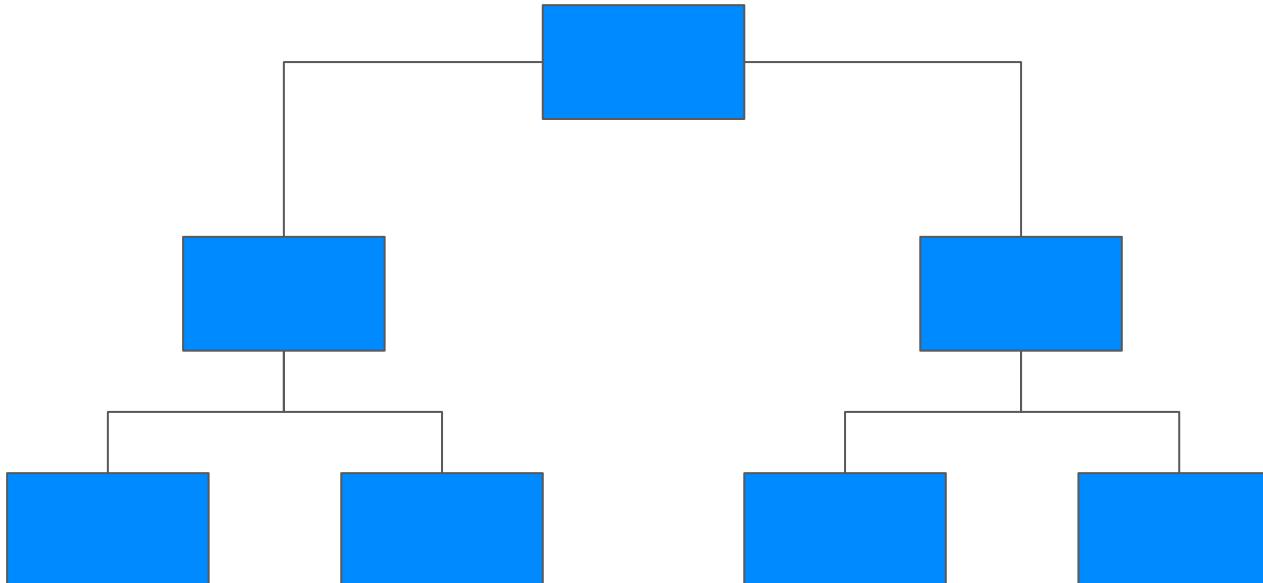


# Unsere UI ist eine Baumstruktur

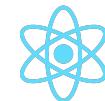
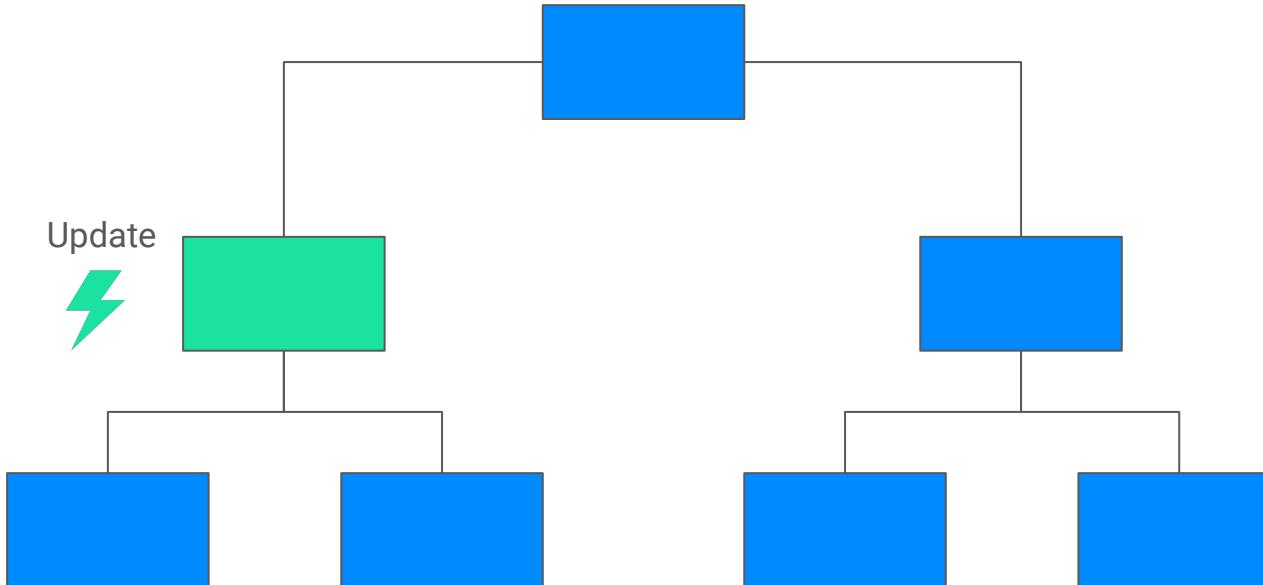
- Änderungen eines *States* hat ein “neu”-Rendern zur Folge.
- Die Komponente, die den *State* erstellt hat, und alle Kinder werden neu gerendert.
- **Rendern in React => Komponenten-Funktion aufrufen.**
- React baut einen internen *virtuellen DOM* auf, der mit dem aktuellen DOM verglichen wird.
- Nur Änderungen werden auf den aktuellen DOM übertragen.



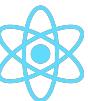
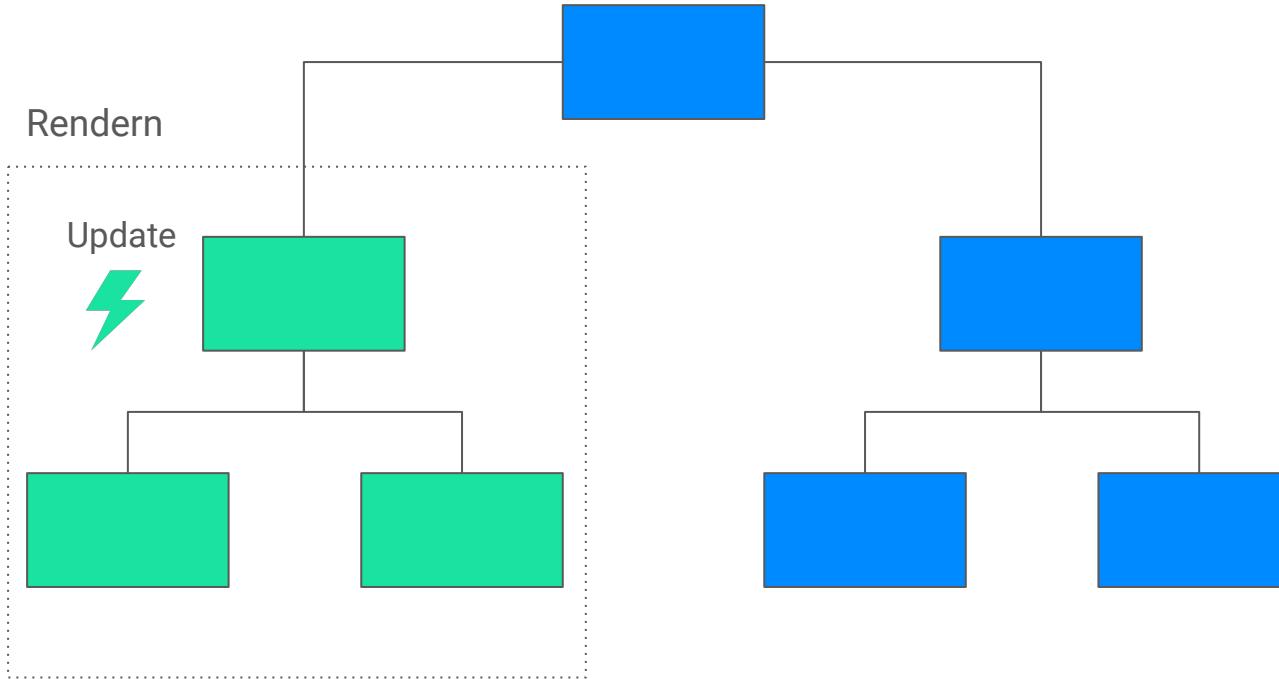
# Unsere UI ist eine Baumstruktur



# Unsere UI ist eine Baumstruktur

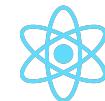
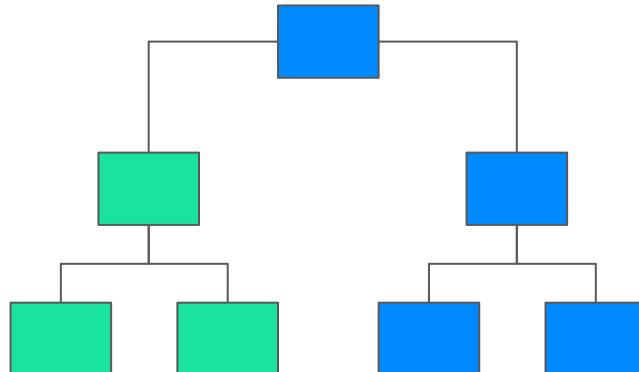


# Unsere UI ist eine Baumstruktur



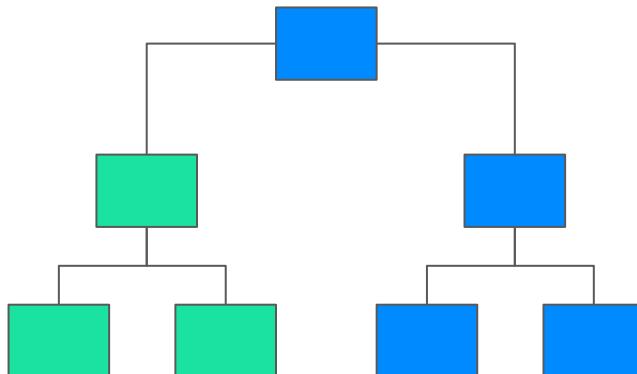
# Unsere UI ist eine Baumstruktur

Virtueller DOM



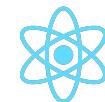
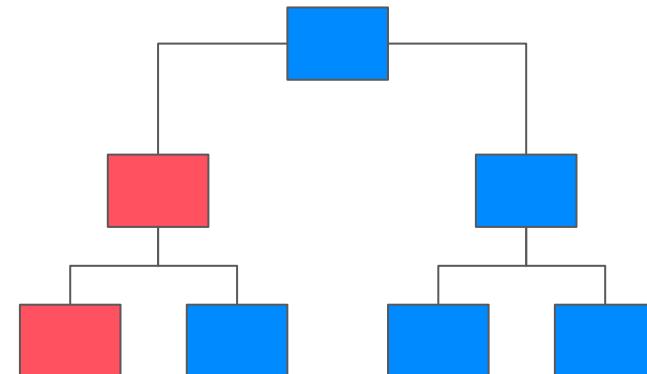
# Unsere UI ist eine Baumstruktur

Virtueller DOM



Update

Änderungen  
der UI



# Lifecycle von React Komponenten

- **Mount**

- Eine Komponente wird mounted, wenn sie zum Bildschirm hinzugefügt wird
- States werden initialisiert

- **Update**

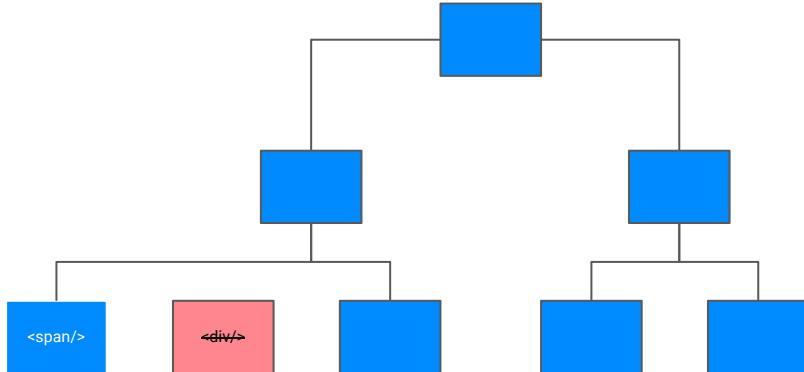
- Eine Komponente wird geupdated, wenn ein Zustandswechsel stattgefunden hat oder sich die *Props* geändert haben
- Meistens in Zusammenhang mit einer Interaktion

- **Unmount**

- Eine Komponente wird unmounted, wenn sie vom Bildschirm entfernt wird

# Mount und Update

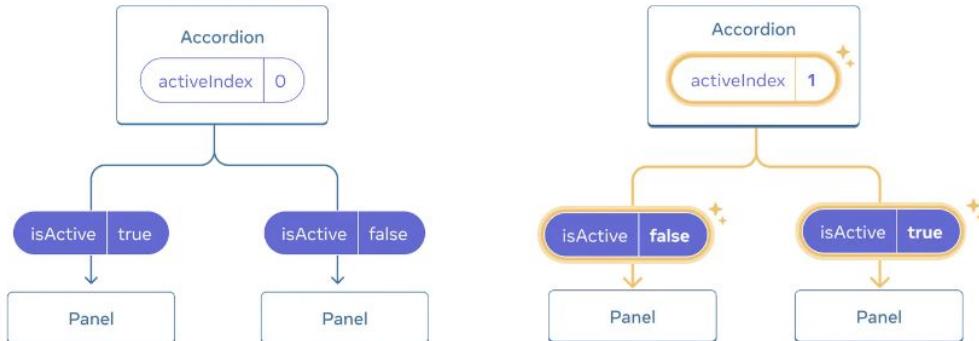
- React entscheidet, ob eine Komponente *updated* oder *unmounted/mounted* werden soll nach dem Typ:



- Optional kann einer Komponente auch ein Key mitgegeben werden. Dann wird die Komponente auch bei einem wechselnden Key neu hinzugefügt.

# Austausch zwischen Komponenten

- React unterstützt nur eine **unidirektionale** Kommunikation von einer Elternkomponente zu den Kindern.
- Geteilte Daten müssen von der Elternkomponente verwaltet werden.
- Über **Callback-Funktionen** können die Kinder diese Daten anpassen.





# Austausch zwischen Komponenten

1. Erweitere das **Interface** von unserem **CustomButton** um eine **onClick-Methode**.
2. Die **onClick-Methode** soll beim Betätigen des Buttons ausgelöst werden.
3. Entferne den **internen State** aus unserem **CustomButton**.
4. Füge einen zweiten **CustomButton** in **App.tsx** hinzu.
5. Beide Buttons sollen in ihrer Beschreibung den aktuellen **gesamten Zählerstand anzeigen**.
6. Beim Klicken auf einen Button soll der gesamte Zählerstand um 1 inkrementiert werden.



# Unsere zweite Komponente

1. Erstellt einen neuen Ordner: **components/CustomThemeButton**.
2. Erstellt in diesem Ordner eine neue Komponente **CustomThemeButton** (**CustomThemeButton.tsx**).
3. Unser neuer Button soll als Props folgende Werte entgegennehmen können:
  - a. type: “primary” | “secondary” | “tertiary” (union)
  - b. onClick: () => void (optional)
  - c. text: string
4. Kopiert die Datei **styles/CustomThemeButton.module.css** nach **components/CustomThemeButton**.
5. Je nach übergebenen Typ soll der Button den entsprechenden Style aus der CSS-Datei nutzen.
6. Nutze den Button in der **App.tsx**.

# Übungsprojekt: Ein News-Blog



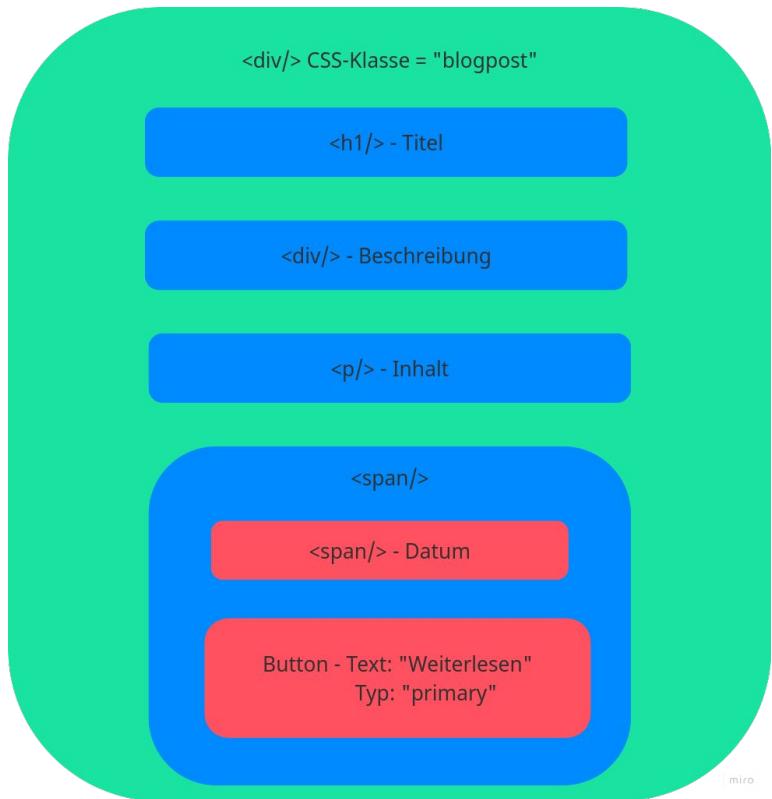


# Darstellung eines Blogpost's (1)

1. Erstelle eine neue Komponente **BlogPost.tsx** unter **src/components/BlogPost**.
2. Die Komponente soll ein **BlogPostModel** aus **models/BlogPostModel** entgegennehmen können und eine **onClick-Methode**.
3. Kopiere die Datei **BlogPost.module.css** aus **styles/** nach **src/components/blogPost**.



# Darstellung eines Blogpost's (1)





# Darstellung eines Blogpost's (2)

1. Unter  ***testData/testData.ts*** findest du Testdaten von Blogposts.
2. Importiere für dieses Beispiel die Konstante ***singleBlogPost*** aus  ***testData/testData.ts***.
3. **Teste** die erstellte Komponente **BlogPost** mit diesen Daten in der **App.tsx**.
4. Die **onClick**-Methode soll die **ID des Blogposts** auf der Konsole ausgeben.

# Darstellung mehrerer Blogposts

1. Um eine **Collection** von Datensätzen zu **rendern**, bietet sich die **map-Funktion** an:

```
const list: string[] = ["one", "two", "three"];  
  
return (  
  
  <div>  
  
    {list.map((item: string) => (<div key="...">{item}</div>))}  
  
  </div>  
  
) ;
```



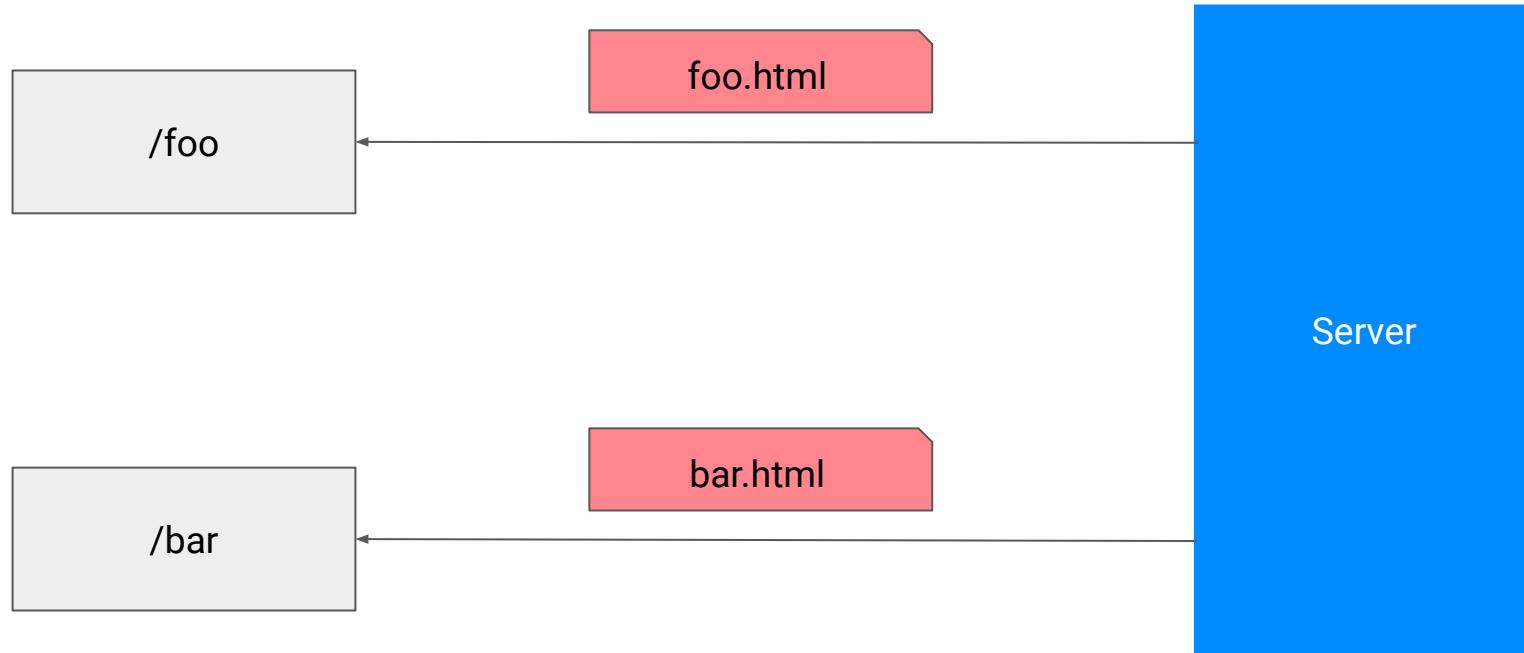
# Darstellung mehrerer Blogposts

1. Unter  ***testData/testData.ts*** findest du Testdaten von Blogposts.
2. Importiere für dieses Beispiel das Konstante Array ***multipleBlogPost*** aus  ***testData/testData.ts***.
3. Rendere die Blogposts in der App.tsx mittels der map-Funktion.
4. Ändere die CSS-Klasse des Containers in App.tsx von “**container**” auf “**container-grid**” zur besseren Darstellung.
5. Entferne die anderen Element **innerhalb** des Containers.

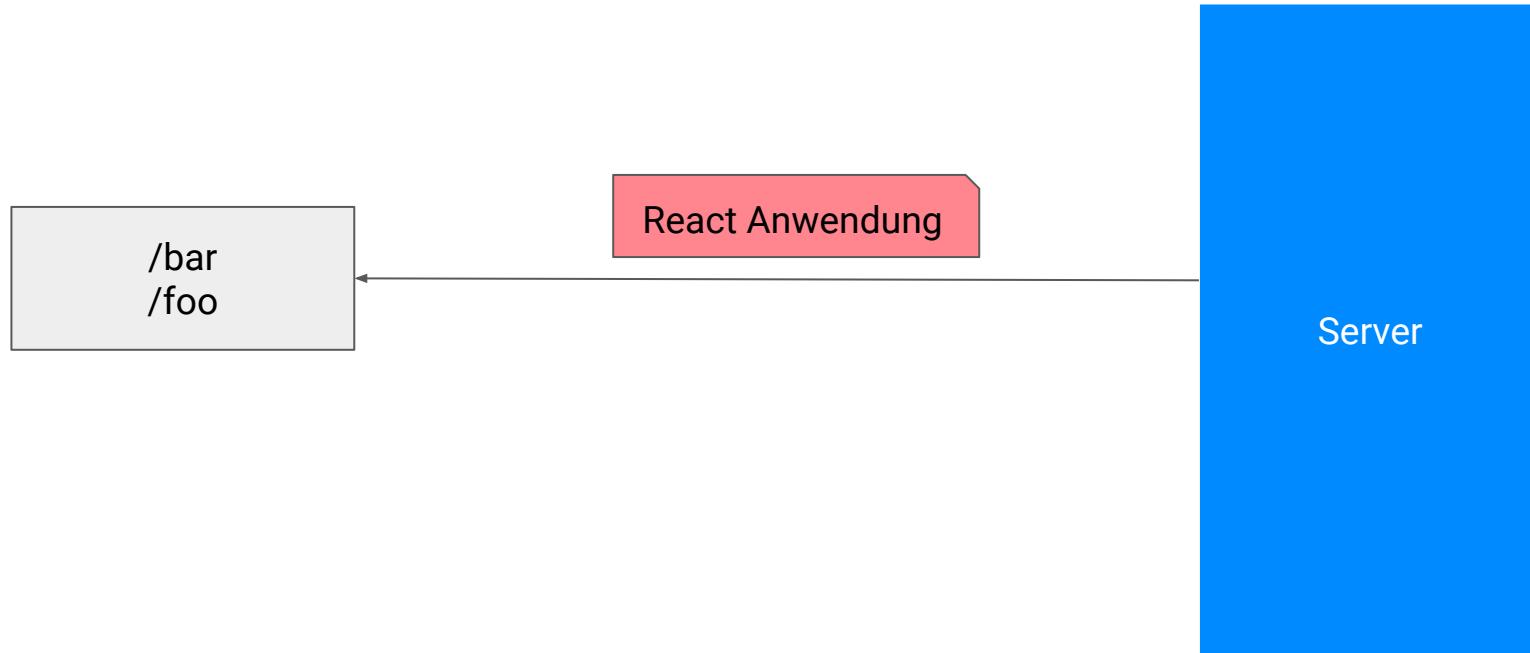
# Routing

- Der Wechsel zwischen verschiedenen URLs sollte auch den visuellen Inhalt ändern
- Verschiedene Pfade laden verschiedene Ressourcen
- **Früher:** Verschiedene Pfade bedeuten unterschiedliche Requests mit anderen Antworten und unterschiedlichen HTML-Dateien
- **Heute (SPA):** Ein initialer Request lädt die ganze Anwendung und eine Pfadänderung wird von dieser Anwendung verwaltet

# Routing



# Routing



# Routing

- Routing ist kein Teil des React-Packages
- Wird über eine *Third-Party-Package* realisiert:
  - react-router-dom
  - Sehr umfangreich und enthält viele Features zur Navigation und zum Routing
- Neues Package hinzufügen:

```
yarn add react-router-dom
```

# Routing

```
<BrowserRouter>
  <Routes>
    <Route path="/" element={<h1>Home</h1>} />
    <Route path="/route1" element={<Page1 />} />
    <Route path="/path/route2" element={<Page2 />} />
    <Route path="/path1">
      <Route path="/subpage1" element={<SubPage1 />} />
      <Route path="/subpage2" element={<SubPage2 />} />
    </Route>
    <Route path="/path/:id" element={<PageWithId />} />
    <Route path="*" element={<Page404 />} />
  </Routes>
</BrowserRouter>
```



# Routing

1. Erstelle drei neue Komponenten unter src/pages:
  - a. Overview/OverviewPage.tsx
  - b. Details/DetailsPage.tsx
  - c. NotFoundPage.tsx
2. **OverviewPage.tsx** soll unsere Blogposts darstellen.
3. Implementiere einen **BrowserRouter** in **App.tsx** mit folgenden drei Routen:
  - a. “/overview” -> **OverviewPage.tsx**
  - b. “/overview/:id” -> **DetailsPage.tsx**
  - c. “\*” -> **NotFoundPage.tsx**

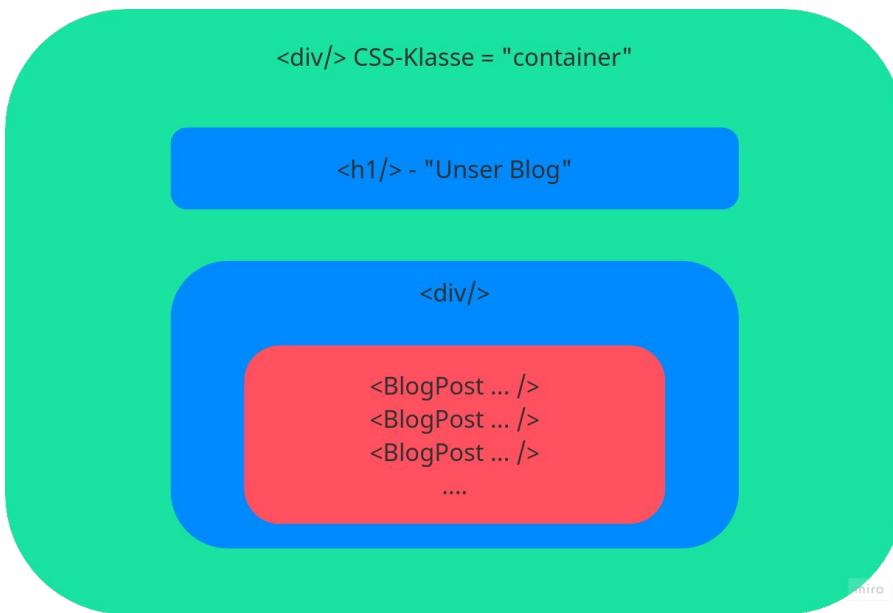


# Routing - Overview

1. Kopiere die Styles von **styles/pages/Overview/OverviewPage.module.css** nach **src/pages/Overview**.
2. Importiere die Styles von **OverviewPage.module.css** in **OverviewPage.tsx**.
3. Zeige die Blogposts der Testdaten aus der vorherigen Übung an.
4. Passe die Struktur der Seite und die CSS-Klassen wie folgt an:



# Routing - Overview



miro

# Blogpost Details darstellen

Folgende Aufgaben müssen von der **DetailsPage** übernommen werden:

1. ID des Blogposts aus der URL ermitteln
2. Den richtigen Blogpost aus den Testdaten ermitteln
3. Die Werte des Blogposts darstellen

# Blogpost ID aus der URL ermitteln

- **react-router-dom** enthält einen **hook** dafür -> useParams().
- import: `import { useParams } from "react-router-dom"`
- Liefert den Wert unter dem Namen, der in der **Route** definiert wurde.

```
<Route path="/path/:id" element={<PageWithId />} />
```



```
const { id } = useParams();
```

# Blogpost aus den Testdaten ermitteln

- Arrays in JavaScript verfügen über eine **find**-Methode
- So lässt sich der entsprechende Blogpost aus den Testdaten ermitteln
- **find** liefert entweder den **gesuchten Blogpost** zu einer ID oder **undefined** zurück





# Blogposts Details Seite erstellen

1. Zeige den richtigen Blogpost auf der **DetailsPage** an.
2. Das Styling der **DetailsPage** liegt unter  
`src/styles/DetailsPage.module.css`
3. Falls kein Blogpost zu der **ID** gefunden werden konnte, dann soll  
ein “**Not found**” angezeigt werden.



# Blogposts Details Seite erstellen

<div/> CSS-Klasse = "container"

<h1/> - Titel des Blogpost's

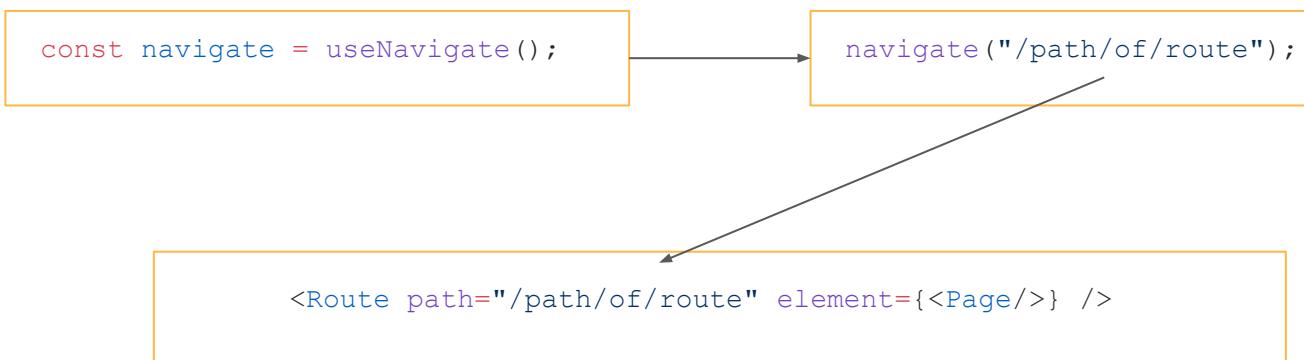
<h2/> - Beschreibung des Blogpost's

<p/> - Inhalt

React

# Navigation

- **react-router-dom** enthält einen **hook** dafür -> `useNavigation()`.
- `import { useNavigation } from "react-router-dom"`
- Liefert eine **navigate-Methode** zur Navigation





# Navigation

1. Der **CustomThemedButton** in der Komponente **BlogPost** soll auf die **DetailsPage** navigieren.
2. Als **:id** soll die ID des entsprechenden BlogPosts eingesetzt werden.
3. Füge einen **CustomThemedButton** vom Typ “**secondary**” der **DetailsPage** hinzu, der den Nutzer auf die **OverviewPage** navigiert
4. Der Button auf der **DetailsPage** soll unter dem Inhalt des Artikels sein.

# Testen von Komponenten



# Wie werden Komponenten getestet?

- Das Testen von Komponenten beinhaltet immer ein Rendern
- Je nach Testverfahren bzw. Art des Tests kann das Rendern in einem **richtigen Browser** oder in einem **browser-ähnlichen System** stattfinden
- Es werden (Nutzer-)Aktionen ausgeführt und der Output wird mit einem Erwartungswert verglichen

# Arten von Tests

- **Unit-Tests**
  - Eine einzelne Komponente wird gerendert
  - Verhalten wird anhand der Ausgabe verifiziert
- **Integrationstests**
  - Ein Zusammenschluss von Komponenten wird gerendert
  - Verifikation und Prüfung ist analog zu Unit-Tests
- **End-To-End-Tests**
  - Die ganze Anwendung wird gestartet
  - Große Teile der Anwendungen sollen hierbei getestet werden

# Unser Testsetup

- Unit- und Integrationstests
- Die Tests und das Rendern werden in einem ***browser-ähnlichen System*** simuliert
- **Vitest:**
  - JavaScript testing framework
  - Bietet Testrunner, Mocking und Prüfungen
- **React-Testing-Library:**
  - Bibliothek mit Hilfsfunktionen zum Testen von React-Komponenten



# Testen

1. Gemeinsames Testen von unseren Komponenten
2. **Tests zu CustomButton:**
  - a. Button wird korrekt angezeigt
3. **Tests zu CustomThemeButton:**
  - a. Button wird korrekt angezeigt
  - b. OnClick-Funktion wird genutzt, sobald auf den Button gedrückt wird
4. **Tests zu BlogPost**
  - a. Blogpost wird in der Komponente korrekt angezeigt
  - b. OnClick-Funktion wird genutzt, sobald auf “Weiterlesen” gedrückt wird

# Anbindung einer Web-API



# Backend-Server starten

- Ein Terminal unter  `${Projektverzeichnis}/docker` öffnen und `docker compose up --build -d` ausführen
- Im Browser die Adresse zum Abrufen der Daten prüfen:



# GET-Request mit der **fetch**-Methode

- JavaScript stellt eine **fetch-Funktion** zur Verfügung, die eine **GET-Anfrage** an einen **Endpunkt** stellen kann.
- ```
const responsePromise: Promise<Response> = fetch("http://localhost:3333/posts");
```
- Das Ergebnis ist ein *Promise*, welches eine *Response* liefert, sobald diese verfügbar ist.
- Die Daten einer *Response* lassen sich mit der asynchronen **json-Methode** in ein TypeScript/JavaScript Objekt umwandeln:
  - ```
const tsObject: Promise<BlogPostModel[]> = response.json();
```



# Blogposts über eine Web-API lesen

1. Erweitere die Komponente **Overview.tsx**.
2. Erfrage die Blogposts über die URL des gestarteten Servers:  
<http://localhost:3333/posts>
3. Gib das übertragene Array auf der Konsole aus mit  
**console.log(...)**.
4. Speichere die erhaltenen Daten in einem **State**. Was fällt dir auf?

# Anbindung eines externen Systems

- React stellt bereits einen passenden Hook zur Verfügung: **useEffect**
- **useEffect** erwartet als Parameter eine Funktion
- Sobald die Komponente in den DOM eingefügt und gerendert wurde, wird die übergebene Funktion ausgeführt
- **useEffect** muss immer auf dem **Top-Level** deiner Komponente aufgerufen werden

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

export default function Komponente() {
  useEffect(() => {
    // Code here!
  }, []);

  return (<div>Hello World!</div>);
}
```



# Blogposts über eine Web-API lesen

1. Nutze **useEffect**, um die gespeicherten Daten in einem **State** zu speichern.
2. Nutze den Inhalt des States zum Rendern der Blogposts.



# Blogpost über eine Web-API lesen

1. Erweitere die Komponente **DetailsPage.tsx**.
2. Erfrage den Blogpost über die URL des gestarteten Servers:  
**http://localhost:3333/posts/\${id}**
3. Speichere den erhaltenen Post in einem **State**.
4. Nutze den State zur Anzeige der Informationen des Blogposts.

# Fragen und Weiteres

A screenshot of a Java code editor with a large orange watermark in the center containing the word 'Fragen'. The background shows code related to a REST API endpoint for a 'User' resource, including annotations like @Path, @GET, @Produces, and @ApiOperation. The code handles various HTTP methods and authentication levels.





**Tobias Kaiser**  
[tobias.kaiser@nterra.com](mailto:tobias.kaiser@nterra.com)



# Vielen Dank für die Aufmerksamkeit

# Eure Meinung ist uns wichtig!

A screenshot of a Java code editor with a large, semi-transparent watermark in the center reading "Eure Meinung ist wichtig!". The background shows Java code for an Azure Function, specifically a class named "HttpTriggerJava" with methods like "main" and "get". The code includes annotations such as @FunctionName("HttpTriggerJava") and @Input("req"). The Java code is written in white and grey on a dark blue background.

nterra



**Die Expert\*innen für Integration**

 [www.nterra.com](http://www.nterra.com)