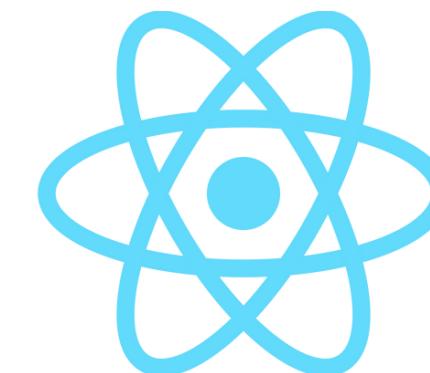


Bastian Jakobi, Marcel Willie, Marvin Pfau

React & Next.js



I Wer wir sind



Marcel Willie

2 Jahre mit React
2 Jahre mit Next.js



Bastian Jakobi

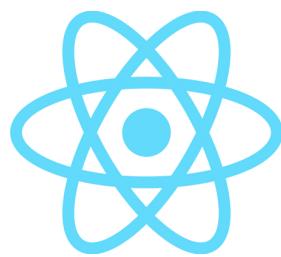
3 Jahre mit React
1 Jahr mit Next.js



Marvin Pfau

2 Jahre mit React
Neuling in Next.js

| Was ist React?



**JavaScript-Bibliothek für
den Aufbau von
Benutzeroberflächen**



**Entwickelt von Facebook,
veröffentlicht 2013**



**Komponenten
basierte Architektur**

I Vorteile von React



Wiederverwendbare Komponenten

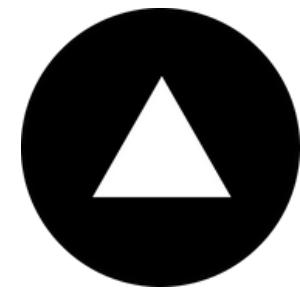


Virtuelles DOM für bessere Leistung und effiziente Updates



Gleichzeitige Entwicklung von Web-Apps und mobilen Anwendungen

| Und was ist Next.js?



Entwickelt von Vercel
(ehemals ZEIT) im Jahr 2016

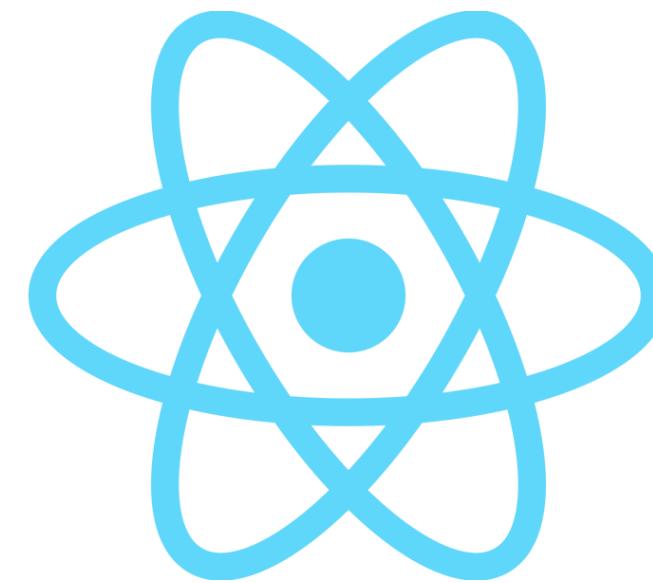


Erleichtert das Erstellen
von React-basierten
Anwendungen im
serverseitigem Rendering

| Wer benutzt Next.js?



| Zeitplan



Vorstellung React
& Typescript

Jetzt



Vorstellung Next.js

Nächste zwei Stunden

| Typescript vs. Javascript

Statisch typisiert
vs.
Dynamisch typisiert



```
let year = 2024
let month = "june"
month = 6 // in javascript erlaubt

let year: number = 2024
let month: string = "june"
month = 6 // nicht erlaubt in typescript
```

| Typescript vs. Javascript

-  **Bessere IDE-Unterstützung**
-  **Erhöhte Code Qualität
und Wartbarkeit**

I Types: Primitive Types

- **string**
- **number**
- **boolean**



```
let myString: string = "Hello, TypeScript!"  
let myNumber: number = 42  
let myBoolean: boolean = true
```

| Types: Arrays

- **number[]**
- **string[]**



```
let myNumberArray: number[] = [1, 2, 3, 4, 5]
let myStringArray: string[] = ["Hello", "World"]
```

| Types: Tuples

- [string, number]



```
let myTuple: [string, number] = ["Hello", 42]
```

| Types: Enums

- enum Color{Red, Green, Blue}



```
enum Color {
    Red,
    Green,
    Blue
}

let myColor: Color = Color.Green
```

| Types: Weitere

- Any
- null und undefined
- object
- Custom Types bei Props

A u f g a b e

00

5 min



I Wie ist ein Projekt strukturiert?

| Aufbau einer Basic UI

```
import Blog from './components/Blog/Blog'

function App() {
  return (
    <>
      <h1 className='header'>My Cool React Blog</h1>
      <Blog />
    </>
  )
}

export default App
```

| Komponenten mit Props

```
● ● ●

import './PostCard.css'

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

interface PostCardProps {
  post: Post
}

function PostCard(props: PostCardProps) {
  const { post } = props
  return (
    <div className="card">
      <img src={'/' + post.imagePath} alt={post.title} className="card-image"/>
      <h2>
        {post.title}
      </h2>
      <p>
        {post.body.substring(0, 100)}...
      </p>
    </div>
  )
}

export default PostCard
```

| Komponenten mit Props



```
import PostCard from  
'./Post/PostCard'  
export default function Post() {  
  
  /* ... load post obj */  
  
  return (  
    <PostCard post={post} />  
  )  
}
```

I Conditional-Rendering

```
● ● ●

import PostCard from '../Post/PostCard'

export default function Post() {

    /* ... */

    if(post == undefined){
        return(<p>Post not found</p>)
    }
    return (
        <>
            <h2>post.title</h2>
            <PostCard post={post} />
        </>
    )
}
```

I Conditional-Rendering



```
import PostCard from "../Post/PostCard";

export default function Post() {
    /* ... */

    return (
        <>
            {post != undefined ? (
                <>
                    <h2>post.title</h2>
                    <PostCard post={post} />
                </>
            ) : (
                <p>Post not found</p>
            )}
        </>
    );
}
```

| Conditional-Rendering

```
● ● ●

import PostCard from "../Post/PostCard";

export default function Post() {
    /* ... */

    return (
        <>
            {post && (
                <>
                    <h2>post.title</h2>
                    <PostCard post={post} />
                </>
            )}
        </>
    );
}
```

A u f g a b e

01

5 min



I Listen rendern in React - map

- **'map'** ist eine Methode, die auf Arrays angewendet wird.
- Sie erstellt ein neues Array, indem sie eine Funktion auf jedes Element des ursprünglichen Arrays anwendet.

| Listen rendern in React - map

```
● ● ●

import React from 'react';

const names = ['Alice', 'Bob', 'Charlie'];

function NameList() {
  return (
    <ul>
      {names.map((name, index) => (
        <li key={index}>{name}</li>
      ))}
    </ul>
  );
}

export default NameList;
```

I React Hooks

- Hooks sind Funktionen, die innerhalb einer funktionalen Komponente aufgerufen werden können
- Erlauben Zugriff auf State, Side Effects und andere Lebenszyklus Events einer React Komponente

“Hooks are functions that let you “hook into” React state and lifecycle features[...]”

| React Hook: useState

- **'useState'** ermöglicht, State in funktionalen Komponenten zu verwalten.
- Gibt nach dem Aufruf **zwei Werte** zurück:
den aktuellen Zustand und eine Funktion,
um diesen Zustand zu aktualisieren.

| React Hook: useState



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

function Counter() {
    // Deklariere eine neue Zustandsvariable namens
    "coonst [count, setCount] = useState<number>(0);

    return (
        <div>
            <p>Du hast {count} Mal geklickt</p>
            <button onClick={() => setCount(count + 1)}>
                Klicke mich
            </button>
        </div>
    );
}

export default Counter;
```

| React Hook: useEffect

- **'useEffect'** erlaubt das ausführen von Seiteneffekten in funktionalen Komponenten.
- Typische Anwendungsfälle: Datenabruf, manuelle DOM-Manipulation, Abonnements.

| React Hook: useEffect

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

function DataFetchingComponent() {
  const [data, setData] = useState<any>([]);
  const [loading, setLoading] = useState<boolean>(true);

  useEffect(() => {
    // Fetch-Daten, wenn die Komponente gemountet wird
    fetch('https://api.example.com/data')
      .then(response => response.json())
      .then(data => {
        setData(data);
        setLoading(false);
      })
      .catch(error => {
        console.error('Fehler beim Abrufen der Daten:', error);
        setLoading(false);
      });
  }, []); // Leeres Array als Abhängigkeit, damit der Effekt nur einmal ausgeführt wird

  if (loading) {
    return <p>Loading...</p>;
  }

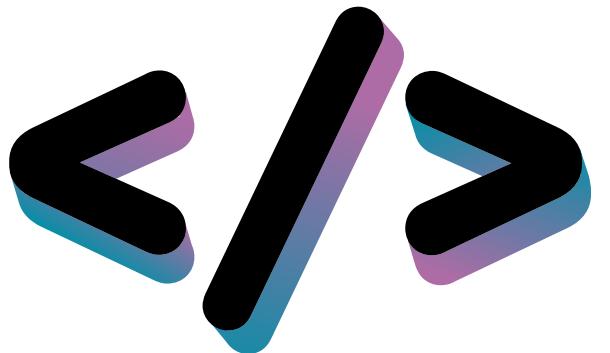
  return (
    <ul>
      {data.map(item => (
        <li key={item.id}>{item.name}</li>
      ))}
    </ul>
  );
}

export default DataFetchingComponent;
```

A u f g a b e

02

30 min



Willkommen zurück

React & Next.js

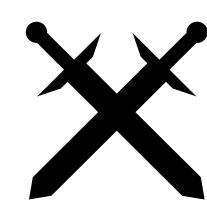
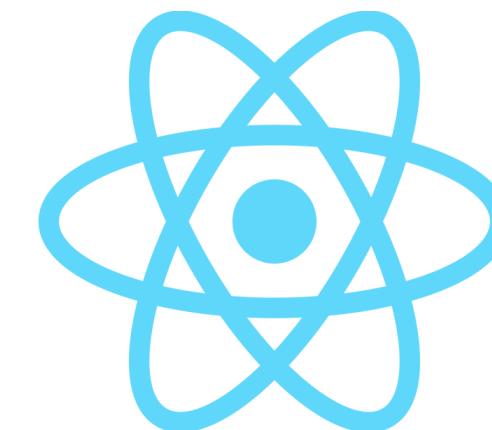


Recap

- TypeScript-Basics
- React-Projekt Aufbau
- React-Komponenten inkl. Props
- Rendern von Listen
- Hooks für Data-Fetching und useState

Unterschiede

zwischen Next.js
und React



| React

- Bibliothek
- Erstellung von Benutzeroberflächen
- Fokus auf Komponentenlogik und UI-Rendering
- Rendering: Client
- Routing: extra React Router

| Next.js

- Framework
- Baut auf React auf
- Serverseitige Funktionen (bspw. API)
- Rendering: Client, Server, Static
- Routing: file-based System
- Code-Splitting

Projektstruktur

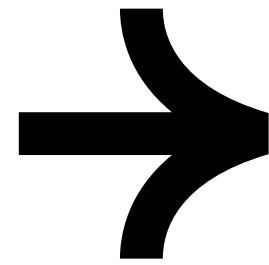
Next.js



V1

```
✓ TEST-PROJECT-2
  > node_modules
  ✓ pages
    ✓ api
      TS hello.ts
      ☀ _app.tsx
      ☀ _document.tsx
      ☀ index.tsx
    > public
    > styles
    ⚒ .eslintrc.json
    ♦ .gitignore
    TS next-env.d.ts
    JS next.config.mjs
    {} package-lock.json
    {} package.json
    ⓘ README.md
    TS tsconfig.json
```

/pages



V2

```
✓ TEST-PROJECT
  > node_modules
  > public
  ✓ src/app
    ★ favicon.ico
    # globals.css
    ☀ layout.tsx
    # page.module.css
    ☀ page.tsx
    ⚒ .eslintrc.json
    ♦ .gitignore
    TS next-env.d.ts
    JS next.config.mjs
    {} package-lock.json
    {} package.json
    ⓘ README.md
    TS tsconfig.json
```

/src/app/

I Grundlegende Verzeichnisse

/public/ → statische Assets

/src/ → Hauptcontainer der Anwendung

/src/app/ → enthält alle Komponenten, separate Seiten der Anwendung

| Optionale Verzeichnisse

/src/app/api/ → Ordner für API-Routen

/src/components/ → Ordner für wiederverwendbare Komponenten

/src/styles/ → Ordner für wiederverwendbare CSS-Styles

/src/utils/ → Ordner für wiederverwendbare Hilfsfunktionen

/src/models/ → Ordner für eigene Typendefinitionen und Interfaces

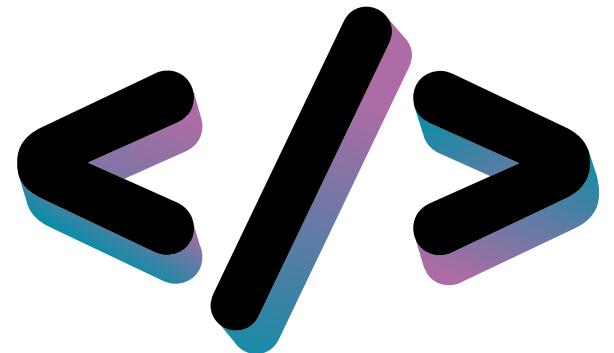
| Konfigurationsdateien

- **next.config.js**: Konfiguration Next.js
- **babel.config.js**: Hinzufügen und Konfiguration von Plugins
- **tsconfig.json**: TypeScript–Konfigurationen
- **.env.local**: Lokale Environment Variablen

A u f g a b e

03

5 min



I File-based Routing

- Routing durch Dateistruktur definiert
- Jedem Ordner innerhalb des /src/app/-Verzeichnisses wird einer URL-Route zugeordnet
- Die **page.tsx** ist die Hauptdatei eines Ordners/Route
- Ordner können beliebig verschachtelt werden
- Kein zusätzliches Routing-Setup erforderlich
- Direkt ersichtlich, welche Datei zu welcher Route gehört

| Beispiele

- `/src/app/page.tsx` → Haupt-URL `/`
- `/src/app/about/page.tsx` → Route `/about`
- `/src/app/products/item/page.tsx` → Route `/products/item`

| <Link/> Element

- <Link> ist eine React-Komponente, die das HTML-Element <a> erweitert
- Nutzt Prefetching für bessere Performance
- Primäre Möglichkeit, zwischen Routen in Next.js zu navigieren

| <Link> Element



```
import Link from 'next/link'

export default function Page() {
  return <Link href="/dashboard">Dashboard</Link>
}
```

I Dynamisches Routing

- Ein Ordner der Struktur [...] ist dynamisch
- Die Datei `/src/app/blog/[slug]/page.tsx` kann über `/blog/1`, `/blog/2`, etc. aufgerufen werden
- `slug` ist im Komponenten über `params` zugänglich

| Dynamisches Routing



```
export default function Page({ params }: { params: { slug: string } })
{ return <div>My Post: {params.slug}</div>
}
```

| Erweiterte Routing-Optionen

- **Fallback-Routen:** error.ts, global-error.ts, not-found.ts (404-Seite)
- **Middleware:** middleware.ts – Logik bevor eine Seite gerendert wird (Authentifizierung, Redirects, etc.)
- **Shared-Layout:** layout.ts – Übergeordnetes Layout
- **Loading-Screen:** loading.ts – Vorgeschaltete Ladeanimation
- **API-Routen:** route.ts – serverseitige Routen, um API-Anfragen zu verarbeiten

error.ts

```
'use client' // Error components must be Client Components

import { useEffect } from 'react'

export default function Error({
  error,
  reset,
}: {
  error: Error & { digest?: string }
  reset: () => void
}) {
  useEffect(() => {
    // Log the error to an error reporting service
    console.error(error)
  }, [error])

  return (
    <div>
      <h2>Something went wrong!</h2>
      <button
        onClick={
          // Attempt to recover by trying to re-render the segment
          () => reset()
        }
      >
        Try again
      </button>
    </div>
  )
}
```

| middleware.ts



```
import { NextResponse, NextRequest } from 'next/server'

// This function can be marked `async` if using `await` inside
export function middleware(request: NextRequest) {
  return NextResponse.redirect(new URL('/home', request.url))
}

export const config = {
  matcher: '/about/:path*',
}
```

| layout.ts

```
● ● ●  
  
export default function Layout({  
  children,  
}: {  
  children: React.ReactNode  
) {  
  return <section>{children}</section>  
}
```

| loading.ts



```
export default function Loading() {  
    // Or a custom loading skeleton  
    const [loading, setLoading] = useState<p>Loading...</p>  
    ...  
}
```

| route.ts

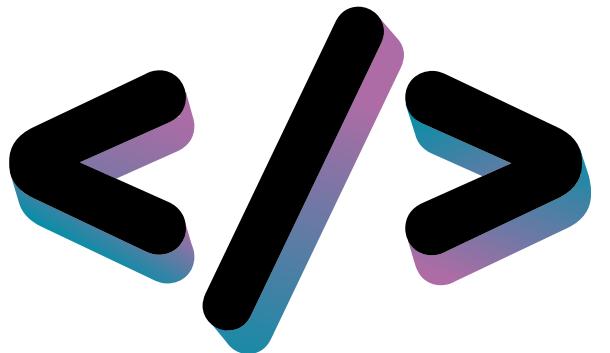


Nach der Übung!

A u f g a b e

04

30 min



I API Route

- Ordner unter **/src/app/api/** werden automatisch als API-Endpunkte behandelt
- Der Pfad bestimmt die URL des Endpunkts
- Die **route.ts** Datei definiert die Logik des Endpunkts
- Beispiel: **/src/app/api/users/route.ts** ist über die URL **/api/user** erreichbar
- Der Name der export Funktion definiert die Funktionalität des Endpunkts (GET, POST, etc.)

route.ts



```
export async function GET(request: Request) {}

export async function HEAD(request: Request) {}

export async function POST(request: Request) {}

export async function PUT(request: Request) {}

export async function DELETE(request: Request) {}

export async function PATCH(request: Request) {}

// If `OPTIONS` is not defined, Next.js will automatically implement `OPTIONS` and set the appropriate
// Response `Allow` header depending on the other methods defined in the route handler.
export async function OPTIONS(request: Request) {}
```

| API Route



```
export async function GET(req: NextApiRequest, res: NextApiResponse)
{  try {
      res.json({status: 200, message:"response message"})
    } catch (e) {
      console.log(e);
      res.json({status: 500, message:"error message"})
    }
}
```

| API Route



```
export async function POST(req: Request, res: NextApiResponse)
{  try {
    const data: any = await req.json()
    const articleBody: string = data.articleBody
    const articleTitle: string = data.articleTitle
    await addArticleToDB(articleTitle,articleBody)
    res.json({status: 200, message:"response message"})
} catch (e) {
    console.log(e);
    res.json({status: 500, message:"error message"})
}
}
```

| Dynamische API Route

- Wie bei den Seiten können API-Routen dynamische Pfade verwenden
- z.B. `/src/app/api/posts/[id]/route.ts`, um Operationen auf einen spezifischen Beitrag durchzuführen
- die id kann über **params** abgerufen werden



```
export async function GET(req: NextApiRequest, { params }: { params: { id: string } }, res: NextApiResponse) {
  try {
    const { id } = params;
    const article = await getArticle(id);
    res.json({status: 200, data: article})
  } catch (e) {
    console.log(e);
    res.json({status: 500, data: e.message})
  }
}
```

Willkommen zurück

React &
Next.js



| Publikums-Frage:

Wie funktionieren relative Verlinkungen?

I Relative Verlinkung

- Wir sind auf **/blog** und möchten nach **/blog/post-1** verweisen
- Next.js akzeptiert nur absolute Links
 - `<Link href=". /post-1">` nicht möglich
 - `<Link href=".. /page-1">` nicht möglich
 - Nur über `<Link href="/blog/post-1">`

Warum?

- Funktioniert bei plain HTML auch nicht
- Wir wissen in jeder Datei unseren aktuellen Pfad (file-based-Routing)

Workaround (!)

- `window.location.pathname + "/post-1"`
- ``${router.asPath}/post-1``

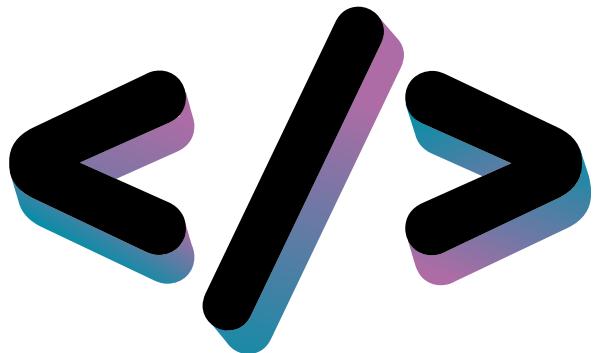
Recap

- **TypeScript- & React-Basics**
- **Next.js-Basics**
- **Statisches & dynamisches Routing**
- **Seiten, Layouts & Middleware**
- **API-Endpunkte**

A u f g a b e

05

Besprechung



| Static-Site-Generation (SSG)

- Ideal, wenn sich die Inhalte nicht häufig ändern
- Inhalte zur Build-Zeit generiert und können als statisches HTML gehostet werden
- Reduziert die Serverlast
- Seiten laden sehr schnell,
- Können über ein Content Delivery Network (CDN) ausgeliefert werden
- Statische Seiten sind einfach zu cachen

I getStaticProps

- getStaticProps erlaubt es, Daten zur Build-Zeit zu holen
- Muss in Komponent-Datei definiert werden
- Next.js führt zur Build-Zeit diese Funktion aus
- Nachdem die Daten geladen wurden, wird die Seite mit den Daten gerendert

I getStaticProps

```
● ● ●  
export async function getStaticProps() {  
  const res = await  
  fetch('https://jsonplaceholder.typicode.com/posts');  
  const posts = await res.json();  
  
  return {  
    props: { posts }, // wird an die Home-Komponente als props  
    übergeben  
  };  
}
```



```
● ● ●  
export default function Home({ posts }) {  
  return (  
    <div>  
      <h1>Mein Blog</h1>  
      {posts.map((post) => (  
        <div key={post.id}>  
          <h2>{post.title}</h2>  
          <p>{post.body}</p>  
        </div>  
      ))}  
    </div>  
  )  
}
```

I | async Server-Component

```
● ● ●

export default async function ServerComponent() {

  const dataList = await (await fetch(`www.example.com/api`)).json() as CustomData[];
  return (
    <div>
      <h2>Blog Posts</h2>
      <div className={styles.postGrid}>
        {dataList.map(data => (
          <CustomElement data={data}/>
        ))}
      </div>
    </div>
  );
}
```

I Server-Side-Rendering (SSR)

- Wird genutzt, wenn die Inhalte deiner Seite häufig aktualisiert werden
- Der gesamte HTML-Inhalt wird auf dem Server generiert.
- Server arbeitet bei jedem Seitenaufruf

I Data-Fetching

- Komponenten können direkt auf Datenquellen zugreifen, ohne die Notwendigkeit, APIs vom Client aus anzusprechen.
- Zugriff auf DB oder Serverseitige Fetch-Request

| getServerSideProps

```
// Beispiel für getServerSideProps in einer Next.js Seite
export async function getServerSideProps(context) {
  const res = await fetch(`https://api.example.com/data`);
  const data = await res.json();

  return {
    props: {
      initData: data,
    },
  };
}
```

```
use server; // Diese Komponente wird auf dem Server ausgeführt
export default function Home({ initData }) {

  return (
    <div>
      <h1>Server-seitig geladene Daten:</h1>
      <pre>{JSON.stringify(data, null, 2)}</pre>
    </div>
  );
}
```

I async Server-Component

```
● ● ●

export const dynamic = "force-dynamic";

export default async function ServerComponent() {

  const dataList = await (await fetch(`www.example.com/api`)).json() as CustomData[];
  return (
    <div>
      <h2>Blog Posts</h2>
      <div className={styles.postGrid}>
        {dataList.map(data => (
          <CustomElement data={data}/>
        ))}
      </div>
    </div>

  );
}
```

I Vorteile SSR

- **Performance:** Server verarbeitet Daten → weniger Arbeit für Client
- **Sicherheit:** Keine Business-Logik im Client → sensitive Daten und Logiken besser geschützt
- **SEO:** Besser für Crawler, da diese oft Probleme mit der Darstellung haben → besseres SEO

I Client-Side-Rendering (CSR)

- Inhalte einer Webseite werden im Browser des Benutzers generiert
- Genutzt bei dynamische Benutzeroberflächen, die auf Benutzerinteraktionen reagieren sollen, ohne dass eine Seite neu geladen werden muss.

I Grundkonzept

- Beim ersten Laden (**Initial Load**) sendet der Server nur das minimale HTML, CSS und JavaScript an den Browser
- Enthält das React-Framework, die Anwendungslogik und die Router-Logik.
- React übernimmt die Kontrolle über das DOM
- Anwendung wird dann im Browser gerendert
- Seiten und Interaktionen werden durch React im Browser gehandhabt
- Keine weiteren Requests an den Server notwendig

I Client Components in Next.js

```
● ● ●

'use client'

import { useState } from 'react'

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

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>Click me</button>
    </div>
  )
}
```

I use client

```
● ● ●  
'use client'  
  
import { useState } from 'react'  
  
export default function Counter() {  
  const [count, setCount] = useState(0)  
  
  return (  
    <div>  
      <p>You clicked {count} times</p>  
      <button onClick={() => setCount(count + 1)}>Click me</button>  
    </div>  
  )  
}
```

- **use client** teilt Next.js mit, dass der gesamte Komponenten im Client-Browser und nicht auf dem Server gerendert werden soll.
- Dies ist besonders nützlich für spezifische Client-seitige APIs, die nicht serverseitig verfügbar sind (Bsp: lokaler Speicher).

| Data-Fetching

- Daten müssen über APIs direkt vom Browser aus geladen werden
- Das kann zu Verzögerungen führen
- Dafür werden Hooks wie useEffect für das Laden Daten und useState für das Updaten der UI benötigt

```
"use client"

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

const DataFetchingComponent = () => {
    const [data, setData] = useState(undefined);

    useEffect(() => {
        const response = await fetch('https://api.example.com/data')
        const data = response.json()
        setData(data);
    }, []);

    if (data == undefined) return <p>Lädt...</p>;
    return (
        <div>
            <h1>Daten von der API</h1>
            <ul>
                {data.map((item, index) => (
                    <li key={index}>{item.name} - {item.detail}</li>
                ))}
            </ul>
        </div>
    );
}

export default DataFetchingComponent;
```

I Warum die Trennung?

- Die Trennung von Client- und Server-Komponenten kann die Ladegeschwindigkeit und die Reaktionsfähigkeit der Anwendung verbessern.
- Server-Komponenten für schnelles initiales Rendering und SEO
- Client-Komponenten für Interaktivität und Client-Funktionalitäten
- Daten und Eingaben werden nicht einfach zum Server gesendet

A u f g a b e

06

20 min



Optimierung



| SEO

Was ist SEO?

- SEO steht für “Search Engine Optimization”
- Ziel ist es die Sichtbarkeit und das Ranking einer Webseite in Suchmaschinenergebnissen zu verbessern

Warum ist das wichtig?

- Höhere Sichtbarkeit in Suchmaschinen führt zu mehr Traffic.
- Verbessert die Benutzererfahrung durch relevante Inhalte und Metadaten.

I Optimierung in Next.js

- Next.js bietet eine Metadata API, wodurch Meta-Daten einfach zu jeder Seite hinzugefügt werden können
- Früher: Hinzufügen der Meta-Daten über die integrierte <Head>-Komponente
- Heute: Hinzufügen der Meta-Daten mit 'export const metadata'
- Dynamische SEO-Meta-Daten passen sich auf Inhalte oder URL-Parameter an

| V1: Statische SEO-Meta-Daten

```
// pages/index.js
import Head from 'next/head'

export default function Home() {
  return (
    <div>
      <Head>
        <title>My Awesome Website</title>
        <meta name="description" content="Welcome to my awesome website." />
        <meta name="keywords" content="awesome, website, interesting content" />
      </Head>
      <main>
        <h1>Welcome to My Awesome Website</h1>
        <p>This is the home page.</p>
      </main>
    </div>
  )
}
```

| V2: Statische SEO-Meta-Daten

```
● ● ●

import type { Metadata } from 'next'

export const metadata: Metadata = {
  title: '...',
  description: '...',
}

export default function Page() {}
```

| Dynamische SEO-Meta-Daten

```
import type { Metadata, ResolvingMetadata } from 'next'

type Props = {
  params: { id: string }
  searchParams: { [key: string]: string | string[] | undefined }
}

export async function generateMetadata(
  { params, searchParams }: Props,
  parent: ResolvingMetadata
): Promise<Metadata> {
  // read route params
  const id = params.id

  // fetch data
  const product = await fetch(`https://.../${id}`).then((res) => res.json())

  // optionally access and extend (rather than replace) parent metadata
  const previousImages = (await parent).openGraph?.images || []

  return {
    title: product.title,
    openGraph: {
      images: ['/some-specific-page-image.jpg', ...previousImages],
    },
  }
}

export default function Page({ params, searchParams }: Props) {}
```

I Caching in Next.js: SSG

Static Generation

- HTML Seiten werden zur Build-Zeit generiert und als statische Dateien gespeichert

Wie funktioniert SSG in Next.js?

- Bei der Bereitstellung der Anwendung werden die Seiten einmalig generiert
- Diese generierten Seiten werden dann von einem Content Delivery Network oder Server zwischengespeichert

I Caching in Next.js: ISR

Incremental Static Regeneration

- Bereits generierte Seiten werden nach einer bestimmten Zeit aktualisiert

Wie funktioniert ISR in Next.js?

- Es ist möglich eine 'revalidate'-Zeit (in Sekunden) festzulegen, nach der eine Seite neu generiert wird, wenn eine Anfrage eintrifft
- Dadurch werden Inhalte aktualisiert ohne ein vollständiges Rebuild der Anwendung

I Caching in Next.js: Client-Side

Client-Side Caching

- Browser speichert bestimmte Ressourcen wie Bilder, Skripte und Stylesheets im Cache

Wie funktioniert Client-Side Caching in Next.js?

- Mittels HTTP-Headern wie 'Cache-Control' kann gesteuert werden, wie lange und unter welchen Bedingungen der Browser die Ressourcen cachen soll
- Dadurch schnellere Ladezeiten und eine reduzierte Anzahl an Server Anfragen

I Cache-Invalidierung

Was ist das?

- Der Prozess, veraltete oder ungültige Daten aus dem Cache zu entfernen und durch aktuelle Daten zu ersetzen

Wie funktioniert Cache-Invalidierung in Next.js?

- Zeitbasiert: Daten werden nach einem bestimmten Zeitraum automatisch ungültig
- Manuell: Entwickler können den Cache gezielt löschen, wenn sich Daten ändern, zum Beispiel durch einen API-Trigger.

I Automatische Optimierung

- Techniken zur Verbesserung der Ladegeschwindigkeit und Performance durch automatische Anpassung und Komprimierung von Ressourcen
- Bessere Benutzererfahrung
- Höheres Ranking in Suchmaschinen

| Bildoptimierung

Next.js verwendet die `<Image>`-Komponente, die Bilder automatisch skaliert, komprimiert und in modernen Formaten bereitstellt.



```
import Image from 'next/image'
import profilePic from './me.png'

export default function Page() {
  return (
    <Image
      src={profilePic}
      alt="Picture of the author"
      // width={500} automatically provided
      // height={500} automatically provided
      // blurDataURL="data:..." automatically provided
      // placeholder="blur" // Optional blur-up while loading
    />
  )
}
```

| Videooptimierung

- Next.js unterstützt die Verwendung von Video-Tag-Optimierungen und Drittanbieter-Tools für die Komprimierung.

```
export function Video() {
  return (
    <video width="320" height="240" controls preload="none">
      <source src="/path/to/video.mp4" type="video/mp4" />
      <track
        src="/path/to/captions.vtt"
        kind="subtitles"
        srcLang="en"
        label="English"
      />
      Your browser does not support the video tag.
    </video>
  )
}
```

I Optimierung von Fonts

- Next.js unterstützt automatische Optimierung durch Ladepriorisierung und Subsetting von Fonts.

```
● ● ●

import { Inter } from 'next/font/google'

// If loading a variable font, you don't need to specify the font weight
const inter = Inter({
  subsets: ['latin'],
  display: 'swap',
})

export default function RootLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return (
    <html lang="en" className={inter.className}>
      <body>{children}</body>
    </html>
  )
}
```

A u f g a b e

07

10 min



Dokumentation, Templates & Deployment

