**NILS HARTMANN**
https://nilshartmann.net

Slides: https://react.schule/wdc2023

# React auf dem Server

# Remix

oder

# Next.JS

# NILS HARTMANN

nils@nilshartmann.net

## Freiberuflicher Entwickler, Architekt, Trainer aus Hamburg

## Java, Spring, GraphQL, React, TypeScript

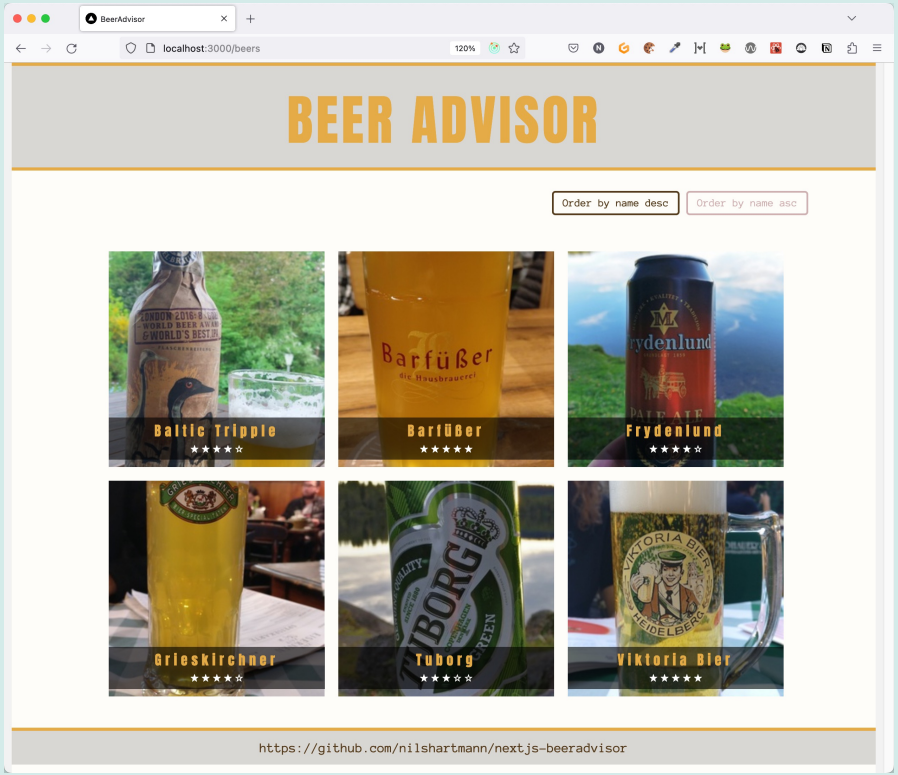

https://graphql.schule/video-kurs


https://reactbuch.de

# Go full-stack with a framework

React is a library. It lets you put components together, but it doesn't prescribe how to do routing and data fetching. To build an entire app with React, we recommend a full-stack React framework like Next.js or Remix.

https://react.dev/



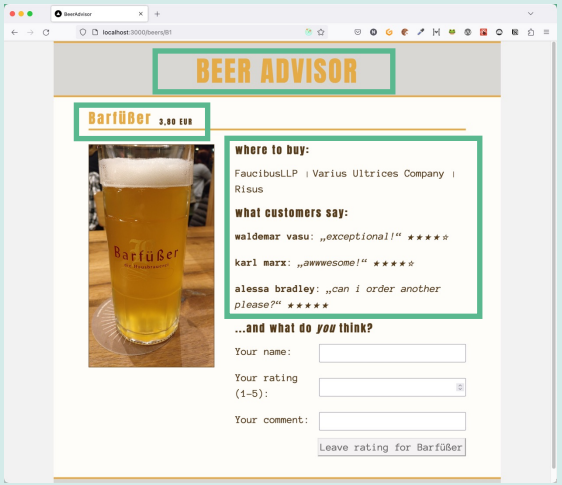

Beispiel-Code: https://github.com/nilshartmann/nextjs-beeradvisor

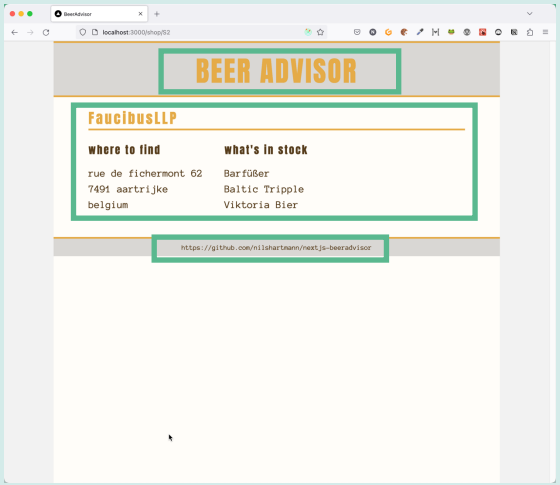
## EIN BEISPIEL...

# EIN BEISPIEL

## Was macht die Beispiel-Anwendung aus?

- Viel statischer Content ☺️

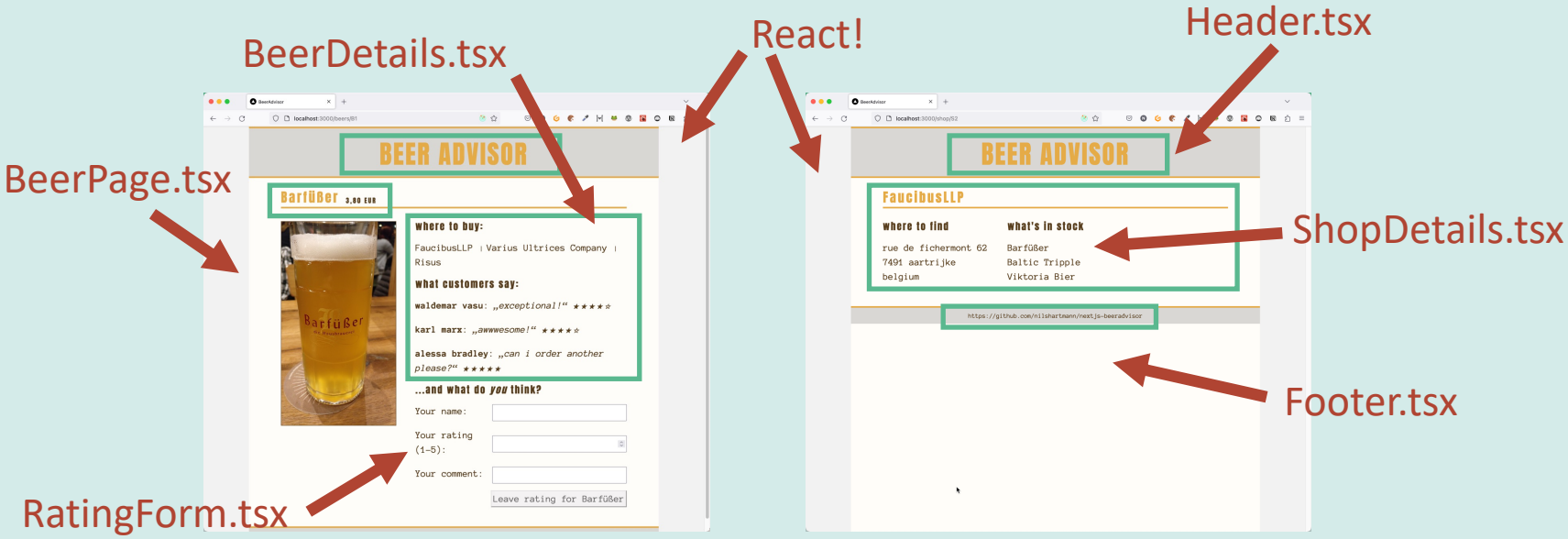


# EIN BEISPIEL

## Was macht die Beispiel-Anwendung aus?

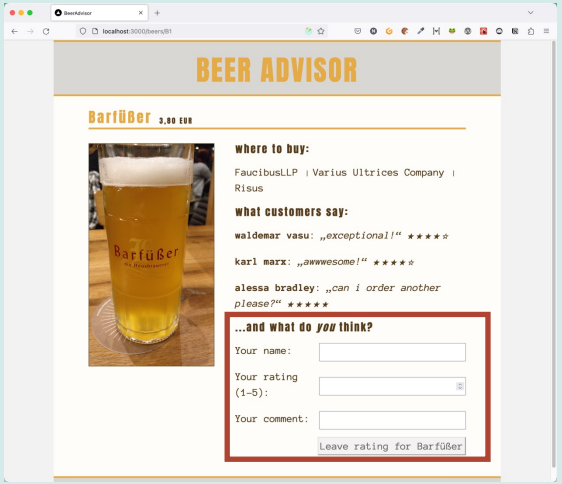
- Viel statischer Content 😌
- Viel JavaScript 😱

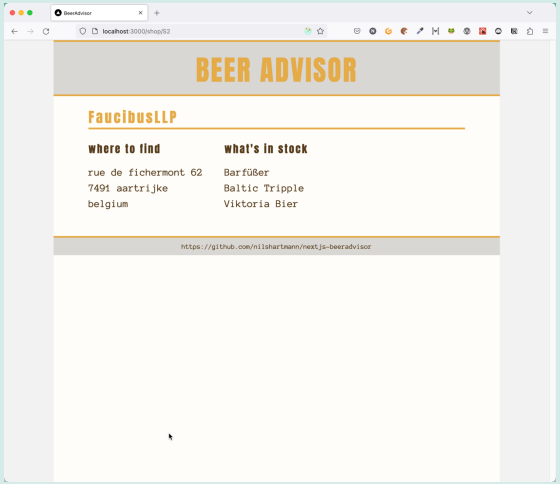


# EIN BEISPIEL

## Was macht die Beispiel-Anwendung aus?

- Viel statischer Content ☺️
- Viel JavaScript 😱
- ...gleichzeitig wenig Interaktion 😔



## Anforderung

👉 Die Seiten sollen möglichst schnell für den Benutzer **sichtbar** und **bedienbar** sein

## Mögliche Probleme

- (Viel) JavaScript-Code, der...

## Mögliche Probleme

- (Viel) JavaScript-Code, der...
  - ... vom Browser geladen werden muss

## Mögliche Probleme

- (Viel) JavaScript-Code, der...
  - ... vom Browser geladen werden muss
  - ... interpretiert und ausgeführt werden muss

## Mögliche Probleme

- (Viel) JavaScript-Code, der...
  - ... vom Browser geladen werden muss
  - ... interpretiert und ausgeführt werden muss

  - ...und mit jeder neuen Komponente mehr wird

# "Fullstack Architektur-Vision"

https://react.dev/learn/start-a-new-react-project#which-features-make-up-the-react-teams-full-stack-architecture-vision

# "Fullstack Architektur-Vision"

https://react.dev/learn/start-a-new-react-project#which-features-make-up-the-react-teams-full-stack-architecture-vision

- **React Server Components** (RSC):
  - Komponenten, die auf dem Server, Client und im Build gerendert werden können
  - Data Fetching "integriert"

# "Fullstack Architektur-Vision"

https://react.dev/learn/start-a-new-react-project#which-features-make-up-the-react-teams-full-stack-architecture-vision

- **React Server Components** (RSC):
  - Komponenten, die auf dem Server, Client und im Build gerendert werden können
  - Data Fetching "integriert"

- **Suspense**:
  - Platzhalter für "langsame" Teile einer Seite
  - Mit Streaming können diese Teile einer Seite "nachgeliefert" werden, sobald sie gerendert sind

## React empfiehlt "Fullstack-Framework"

- **Server Components** erfordern Rendern auf dem Server oder im Build

- Dazu braucht man ein "**Fullstack-Framework**"

## React empfiehlt "Fullstack-Framework"

- **Server Components** erfordern Rendern auf dem Server oder im Build
- Dazu braucht man ein "**Fullstack-Framework**"
- "**Framework**" ist verharmlosend, weil es sich in der Regel um einen kompletten Stack samt Build-Tools und Laufzeitumgebung handelt

## React empfiehlt "Fullstack-Framework"

- **Server Components** erfordern Rendern auf dem Server oder im Build
- Dazu braucht man ein "**Fullstack-Framework**"
- "**Framework**" ist verharmlosend, weil es sich in der Regel um einen kompletten Stack samt Build-Tools und Laufzeitumgebung handelt
- Deswegen werden solche Frameworks auch als "**Meta-Frameworks**" bezeichnet (=> Sammlung von Frameworks)

## React empfiehlt "Fullstack-Framework"

- **Next.js** entspricht den Vorstellungen des React-Team
  - Mit dem **App-Router**, stabil ab Next.js 13.4
  - Einige React-Entwicker sind zu Vercel in das Next.js-Team gewechselt

## React empfiehlt "Fullstack-Framework"

- **Next.js** entspricht den Vorstellungen des React-Team
  - Mit dem **App-Router**, stabil ab Next.js 13.4
  - Einige React-Entwicker sind zu Vercel in das Next.js-Team gewechselt
- **Remix** ist aus dem React Router Projekt entstanden
  - Viele ähnliche Konzepte ("React Router mit Server")

# Routing

## Dateisystem-basierte Routen

- Die Routen werden auf dem Server definiert
- Die Frameworks arbeiten mit Ordner- bzw. Datei-Konventionen
- Unterstützung für typische Anforderungen wie dynamische Segmente, Route-Gruppen vorhanden
- Layout-Komponenten können hierarchisch organisiert werden

## Routen in Next.JS

- In Next.js ist ein Ordner eine Route, wenn darin eine `page.tsx`-Datei liegt
  - `src/app/beer/[beerId]/page.tsx`

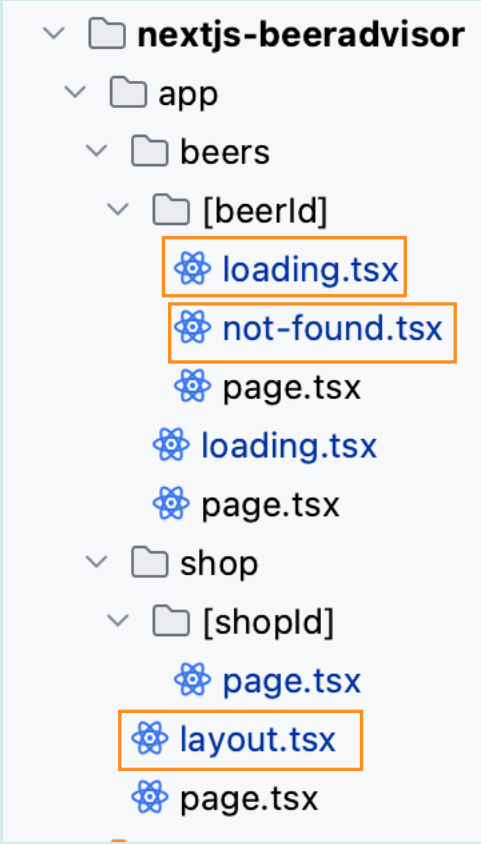
## Routen in Next.JS

- In Next.js ist ein Ordner eine Route, wenn darin eine `page.tsx`-Datei liegt
  - `src/app/beer/[beerId]/page.tsx`
- Diese Datei exportiert eine React-Komponente, die für die Route dargestellt werden soll

```tsx
export default function BeerPage({ params }: BeerPageProps) {
  return <h1>Hello!</h1>
}
```

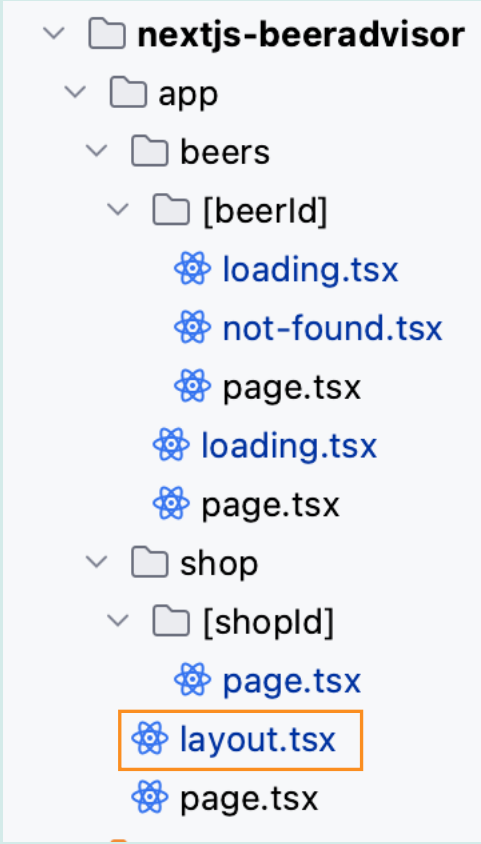
## Routen in Next.JS

- In einem Route-**Verzeichnis** kann es weitere Dateien (mit festgelegten Namen geben), die Komponenten für verschiedene Zwecke exportieren:
  - layout.tsx: Layout-Komponente
  - error.tsx: eine Komponente, die die im Fehlerfall angezeigt wird (ähnlich Error Boundaries im Client)
  - loading.tsx: wird gerendert, während noch Daten für die aktuelle Route geladen werden (bzw. Promises ausstehen)
  - not-found.tsx: wird aufgerufen, wenn eine Komponente in der Route einen 404-Fehler erzeugt



## Routen in Next.JS

- Jede Route kann eine Layout-Komponente haben
- Dieser Komponente wird die darzustellende Seite als children-Property übergeben
- Wenn eine Route keine Layout-Komponente hat, wird im Baum oberhalb nach der nächstgelegenen Layout-Komponente gesucht
- Layout-Komponenten können verschachtelt sein



## Routen in Remix

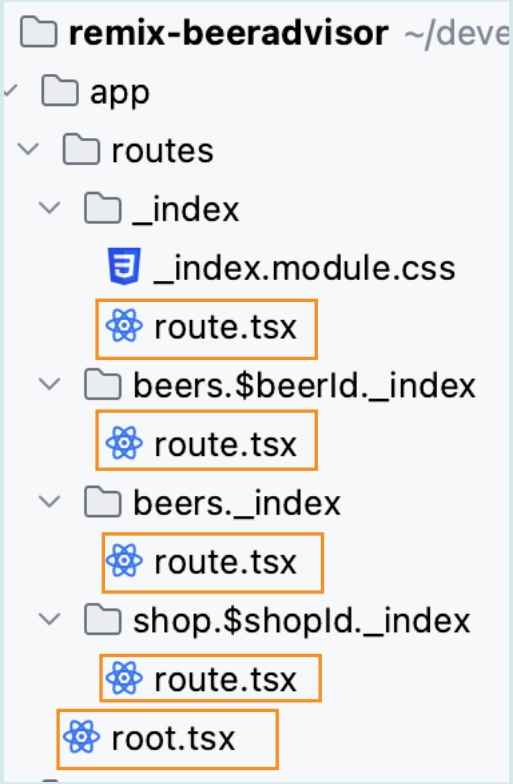
- Alle **Dateien** im `routes`-Ordner sind Routen.

- Durch Punkt getrennt werden die Pfade abgebildet ("flat filenaming convention")

  - `src/app/routes/root.tsx # Root-Layout`
  - `src/app/routes/beers.tsx # Layout für /beers`
  - `src/app/routes/beers._index.tsx # Seite für /beers`
  - `src/app/routes/beers.$beerId.tsx # Layout für /beer/B1`
  - `src/app/routes/beers.$beerId._index.tsx # Seite für /beer/B1`

## Routen in Remix

- Routen können auch in Verzeichnissen organisiert werden
- Der Verzeichnisname entspricht
  dann dem Routen-Name (ebenfalls "flat")
- Im Verzeichnis muss es eine `route.tsx`-Datei geben
- Der Verzeichnis-Ansatz eignet sich, wenn man weitere
  Komponenten/Dateien zu einer Route ablegen möchte
  ("Co-Location")



## Routen in Remix

- Eine Route-Komponente ist entweder eine "Seite" oder ein "Layout"
- Layouts können geschachtelt sein
- Die "Seite" muss jeweils mit `_index.tsx` (bzw. Verzeichnisname `._index`) benannt sein
- Die Layout-Komponenten können die Seite mit der <Outlet />-Komponente einbinden

```
export default function App() {
  return (
    <html lang="en">
      <body>
          <header>
              <h1>
                Beer Advisor
              </h1>
          </header>
          <main>
            <Outlet />
          </main>
      </body>
    </html>
  );
}
```

## Routen in Remix

- Routen-**Dateien** in Remix können neben der Komponente (Default Export) weitere **Funktionen** exportieren:
  - `headers`, `links`, `meta` zum Hinzufügen von Daten in das `head`-Element der aktuellen Seite
  - `loader`- und `action`-Funktion für Data Fetching und zum Verändern von Daten

## Routen: Unterschiede

- Next.JS
  - organisiert die Routen in hierarchischer Verzeichnisstruktur
  - klare Trennung zwischen Layout und Seite. Beide Dateien liegen im Ordner der Route.
  - Nur page.tsx-Dateien erzeugen eine aufrufbare Route

## Routen: Unterschiede

- Next.JS
  - organisiert die Routen in hierarchischer Verzeichnisstruktur
  - klare Trennung zwischen Layout und Seite. Beide Dateien liegen im Ordner der Route.
  - Nur page.tsx-Dateien erzeugen eine aufrufbare Route
- Remix
  - "Flache" Pfade
  - Layout und Komponente unterscheiden sich durch Namenskonvention und liegen parallel
  - Layout-Komponente ist "normale" Komponente mit `<Outlet />`-Element
  - Eine Layout-Komponente erzeugt eine aufrufbare Route.

# Data Fetching

## Anforderungen

- Übersichtsseite
  - Während die Seite geladen wird, soll ein Loading-Indikator angezeigt werden
- Bier-Detail-Seite
  - Soll angezeigt werden, sobald die Bier-Informationen und Bewertungen im Backend geladen wurden (lokale DB)
  - Für die Namen der Shops soll ein Platzhalter angezeigt werden, bis die Namen ermittelt wurden (remote Service)

# Data Fetching

## Next.JS

# Server Components

# Introducing Zero-Bundle-Size React Server Components

December 21, 2020 by Dan Abramov, Lauren Tan, Joseph Savona, and Sebastian Markbåge

2020 has been a long year. As it comes to an end we wanted to share a special Holiday Update on our research into zero-bundle-size React Server Components.

https://legacy.reactjs.org/blog/2020/12/21/data-fetching-with-react-server-components.html

# SERVER COMPONENTS

**Idee:** Komponenten werden nicht im Client ausgeführt

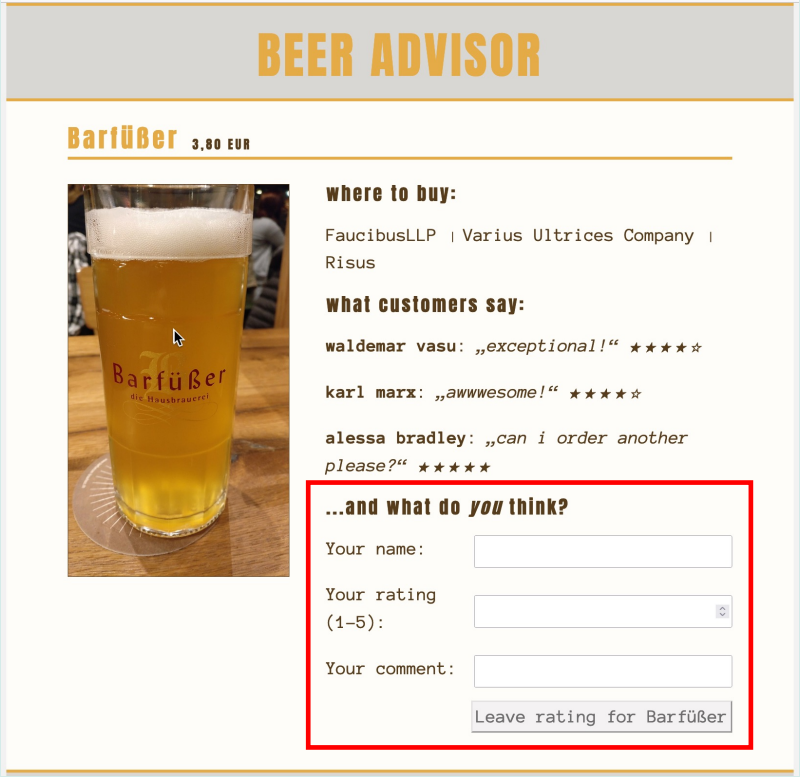
- Sie stehen auf dem Client nur fertig gerendert zur Verfügung
- Der Server schickt lediglich eine *Repräsentation der UI,* aber *keinen Code*

👉 "Zero-Bundle-Size"

# Arten von Komponenten

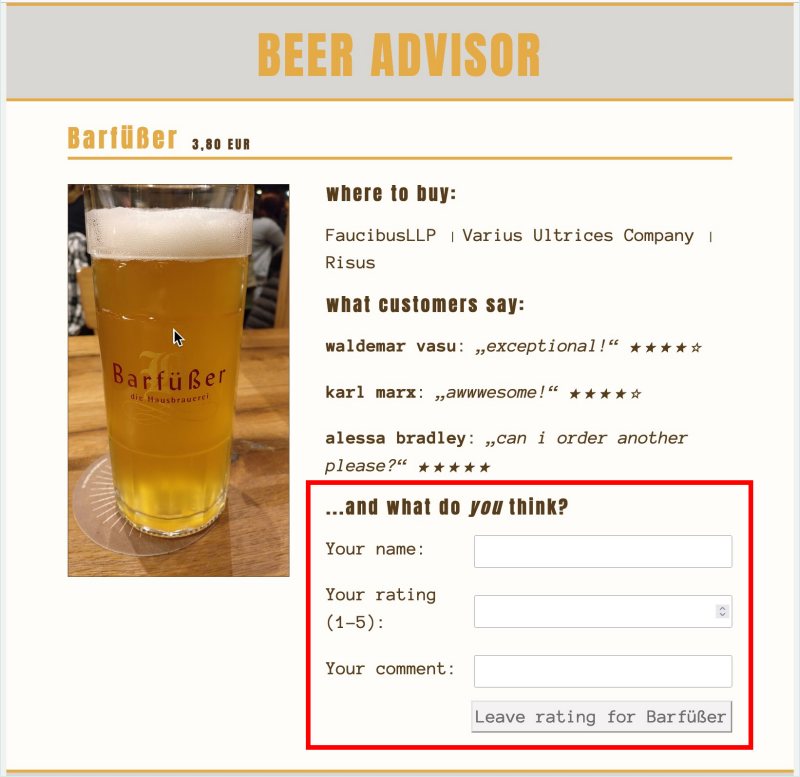
## Client-Komponenten (wie bisher)

- Werden auf dem Client gerendert



## Client-Komponenten (wie bisher)

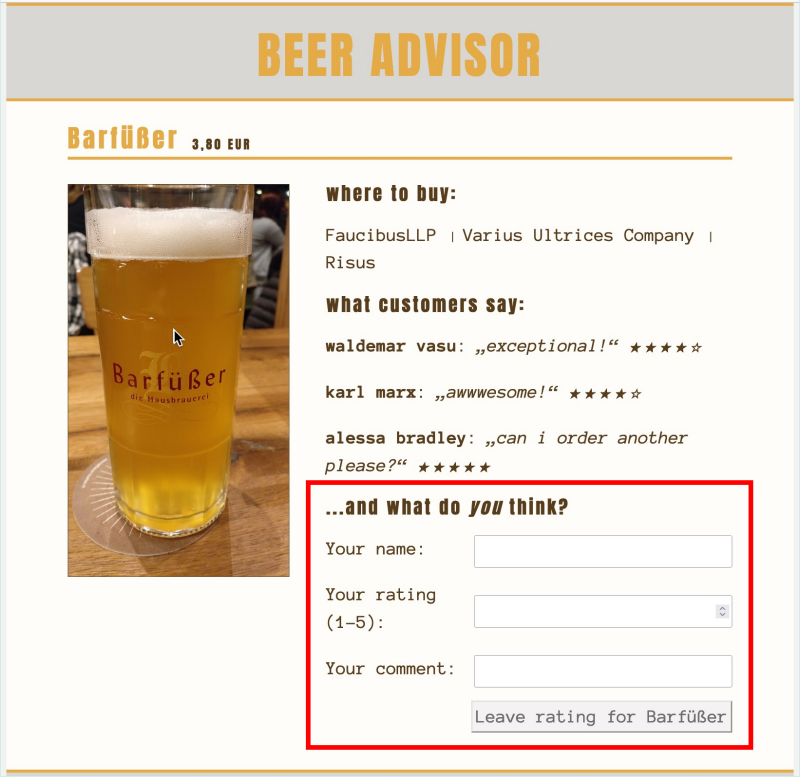
- Werden auf dem Client gerendert
- oder auf dem Server 🙄



## Client-Komponenten (wie bisher)

- Werden auf dem Client gerendert
- oder auf dem Server 🙄

- JavaScript-Code immer zum Client gesendet
- Können deshalb interaktiv sein



## Neu: Server-Komponenten

- werden auf dem Server gerendert

## Neu: Server-Komponenten

- werden auf dem Server gerendert
- oder im Build 🙄

## Neu: Server-Komponenten

- werden auf dem Server gerendert
- oder im Build 🙄

- liefern UI-Beschreibung zum React-Client zurück (kein JavaScript-Code)

## Neu: Server-Komponenten

- werden auf dem Server gerendert
- oder im Build 🙄

- liefern UI-Beschreibung zum React-Client zurück (kein JavaScript-Code)

```
export default function LandingPage() {
  return (
    <div>
      <h1>Welcome!</h1>
      <p>Before you enter, please confirm that you are old enough to drink beer?</p>

      <Link href={"/beers"}>Yes, I am</Link>
    </div>
  );
}
```

## Neu: Server-Komponenten

- werden auf dem Server gerendert
- oder im Build 🙄

- liefern UI-Beschreibung zum React-Client zurück (kein JavaScript-Code)

- können asynchrone Funktionen sein!

## Neu: Server-Komponenten

- werden auf dem Server gerendert
- oder im Build 🙄

- liefern UI-Beschreibung zum React-Client zurück (kein JavaScript-Code)

- können asynchrone Funktionen sein!

- können Server Umgebung und Ressourcen nutzen (!)
  - Datenbanken
  - Filesystem

## Beispiel: Übersichtsseite

- React Server Component: JS-Code kommt nicht in den Browser
- Wird nur auf dem Server (oder im Build) ausgeführt
- Data Loading direkt in der Komponente

```tsx
export default async function BeerListPage() {

  const beers = await prisma.beer.findMany({
    select: {
      id: true, name: true,
      ratings: { select: { stars: true } } },
  });

  return (
    <div className={styles.BeerOverview}>
      {beers.map((beer) => (
        <BeerImage
          key={beer.id}
          name={beer.name}
          stars={calcAverageStars(beer.ratings)}
        />
      ))}
    </div>
  );
}
```

/app/beers/page.tsx

# Data Fetching

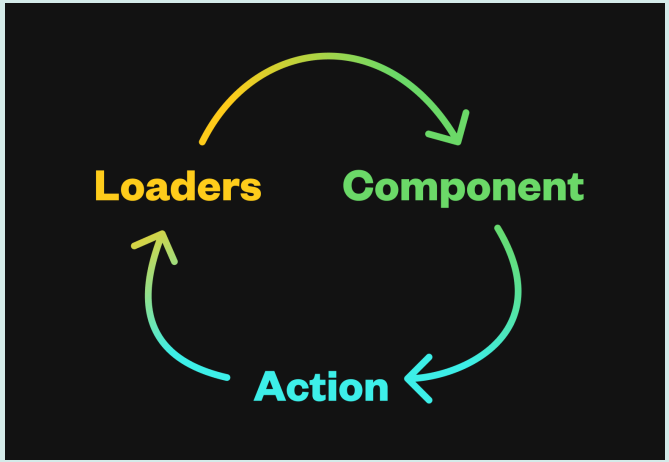
## Remix

- **Keine RSC in Remix**

> The tl;dr is that we are optimistic about adding support for RSC in Remix v3 and we are anxious to do our part in the effort to prove the technology in multiple frameworks. The capabilities of RSC are interesting, but Remix v2 relies on current stable React features, which at the time of this writing doesn't include RSC. When RSC is stable, you can expect that Remix will support it.

https://remix.run/blog/remix-v2

- **Data Fetching in Remix**




https://remix.run/docs/en/2.0.0/discussion/data-flow

## Data Fetching in Remix

- Jede Routen-Datei kann eine Loader-Funktion exportieren

- Die Loader-Funktion bekommt die Request-Parameter übergeben und kann damit die Daten für die Route ermitteln

```
export async function loader({ request }: LoaderFunctionArgs) {

  const beers = await prisma.beer.findMany({
      select: { id: true, name: true,
        ratings: { select: { stars: true, } } }
    });

  return json({ beers });
}
```

app/routes/beers._index/route.tsx

## Data Fetching in Remix

- Jede Route-Datei kann eine Loader-Funktion exportieren
- Die Loader-Funktion wird nur auf dem Server ausgeführt:
  - beim ersten Aufruf der Seite
  - beim Navigieren per fetch API
- Der Code der Loader-Funktion kommt nicht auf den Client! (vergleichbar mit RSC)
- Erst wenn die Daten geladen wurden, wird die Route gerendert

## Data Fetching in Remix

- In der Routen-Komponente kann mit `useLoaderData` auf die geladenen Daten zugegriffen werden

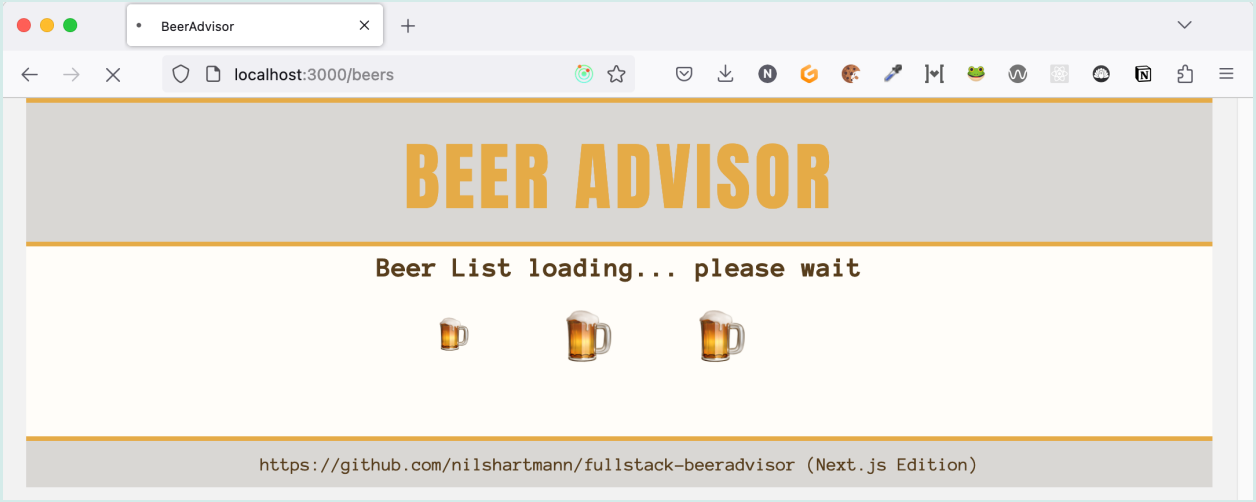
```tsx
export async function loader({ request }: LoaderFunctionArgs) {...}

export default function BeerListPage() {

  const beersData = useLoaderData<typeof loader>();

  return (
    <div className={styles.BeerOverview}>
      {beersData.beers.map((beer) => (
        <BeerImage
          key={beer.id}
          name={beer.name}
          stars={calcAverageStars(beer.ratings)}
        />
      ))}
    </div>
  );
}
```

app/routes/beers._index/route.tsx

# Data Fetching

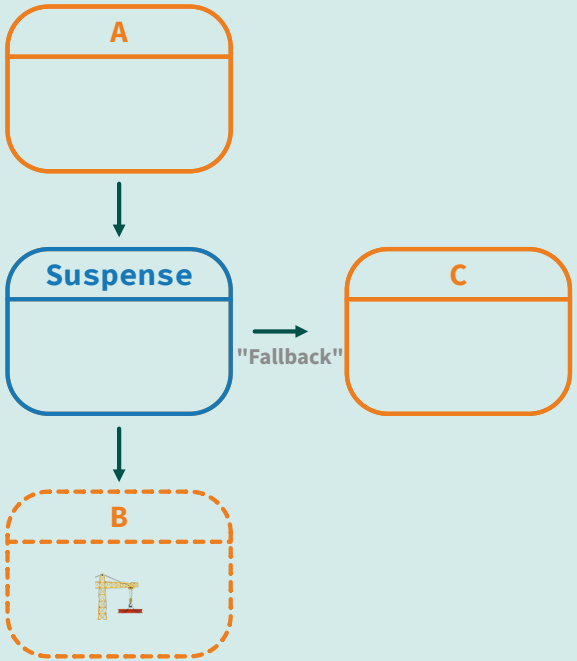
## Warte-Hinweis

# Anforderung: Überssichtsseite soll Warte-Hinweis ausgeben



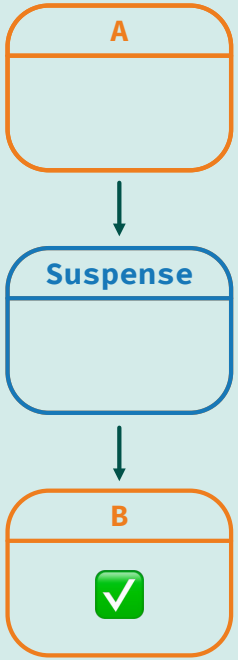

- Bis alle Promises einer Route aufgelöst werden, kann Zeit vergehen
- **React** kann mit Suspense in der Zeit eine Fallback-Komponente rendern

**Suspense:** Unterbricht das Rendern, solange "etwas" fehlt



# SUSPENSE

**Suspense:** Unterbricht das Rendern, solange "etwas" fehlt



## Anforderung: Überssichtsseite soll Warte-Hinweis ausgeben

- Next.JS legt eine Suspense-Komponente um die page-Komponente

## Anforderung: Überssichtsseite soll Warte-Hinweis ausgeben

- Next.JS legt eine Suspense-Komponente um die page-Komponente
- Die Fallback-Komponente wird in der `loading.tsx`-Datei implementiert

```tsx
export default async function BeerListPage() {
  const beers = await prisma.beer.findMany({
    select: {
      id: true, name: true,
      ratings: { select: { stars: true } } },
  });

  return (
    <div className={styles.BeerOverview}...>
  );
}
```

/app/beers/page.tsx

```tsx
export default function BeerPageLoading() {
  return (
    <LoadingIndicator placeholder={"🍺 "}>
      Beer List loading... please wait
    </LoadingIndicator>
  );
}
```

/app/beers/loading.tsx

## Anforderung: Überssichtsseite soll Warte-Hinweis ausgeben

- In Remix muss der Loader explizit ausdrücken, dass ein Teil der Daten auf später "verschoben" wird (defer)

```
export async function loader() {

  const beers = await prisma.beer.findMany({
      select: {
        id: true, name: true,
        ratings: { select: { stars: true, }, },
      }});

  return defer({ beers });
}
```

app/routes/beers._index/route.tsx

## Anforderung: Überssichtsseite soll Warte-Hinweis ausgeben

- In Remix muss der Loader explizit ausdrücken, dass ein Teil der Daten auf später "verschoben" wird (defer)
- Auf die Promises muss dann mit der Async-Komponente von Remix gewartet werden
- Die React Suspense-Komponente wird von Hand eingebunden

```tsx
export async function loader() {

  const beers = await prisma.beer.findMany({
      select: {
        id: true, name: true,
        ratings: { select: { stars: true, }, },
      }});

  return defer({ beers });
}
```

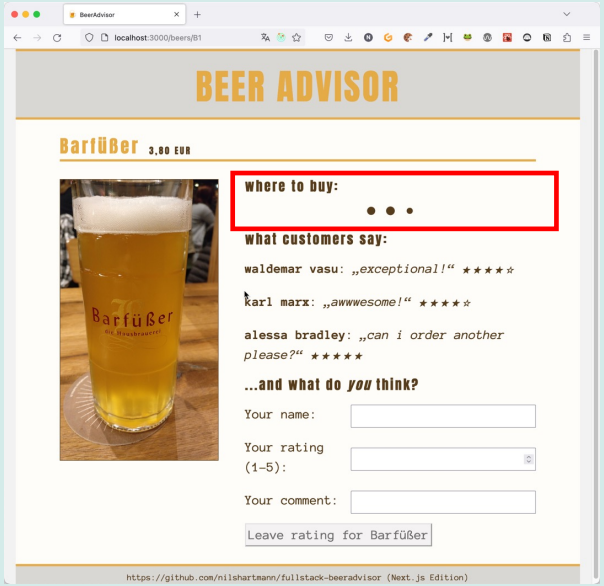
app/routes/beers._index/route.tsx

```tsx
export default function BeerListPage() {
  const beersPromise = useLoaderData<typeof loader>();

  return (
    <Suspense fallback={<LoadingIndicator placeholder={"🐠"} />}>
      <Await resolve={beersPromise.beers}>
        {(beers) => (
            <div className={styles.BeerOverview}>
              {beers.map((beer) => ...)}
            </div>
        )}
      </Await>
    </Suspense>
  );
}
```

# Anforderung: Bier-Detail-Seite soll nicht auf Namen der Shops warten

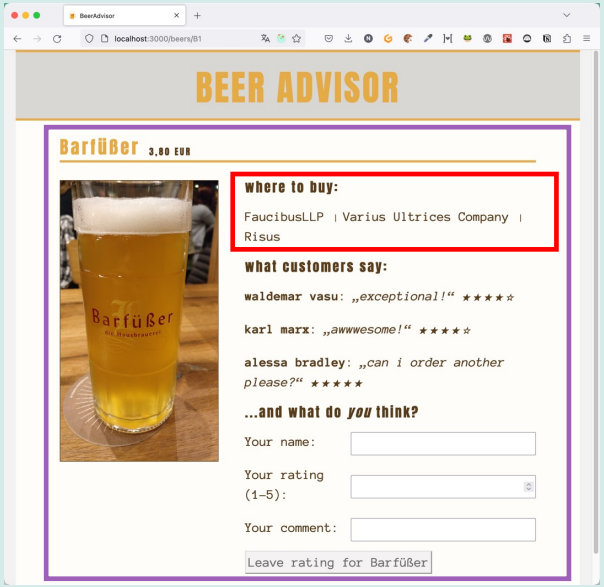
- Hier gibt es zwei Datenquellen: Bier-Details und Shops

nach hinten



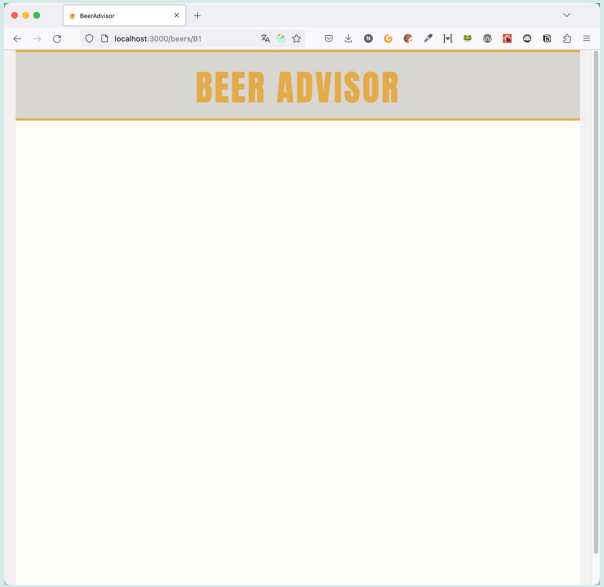
## Paralleles Data Fetching

- Hier gibt es zwei Datenquellen: Bier-Details und Shops



## Mögliches Problem: "Wasserfall" beim Daten laden

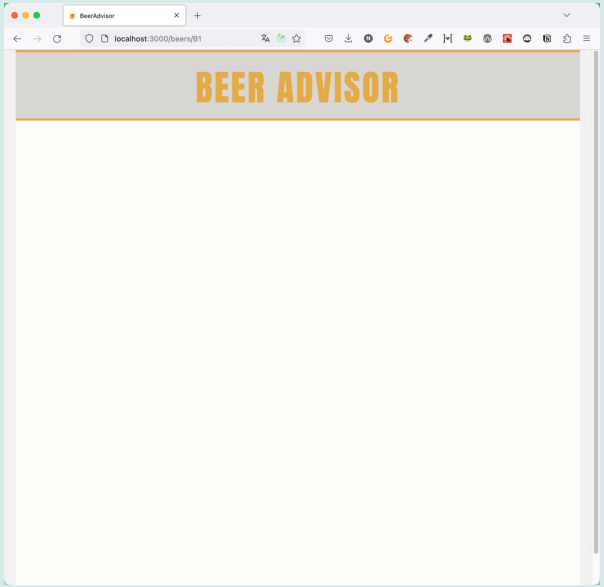
- Klassische React-Anwendung: Daten werden nacheinander geladen...




Zeit

## Mögliches Problem: "Wasserfall" beim Daten laden

- Klassische React-Anwendung: Daten werden nacheinander geladen...

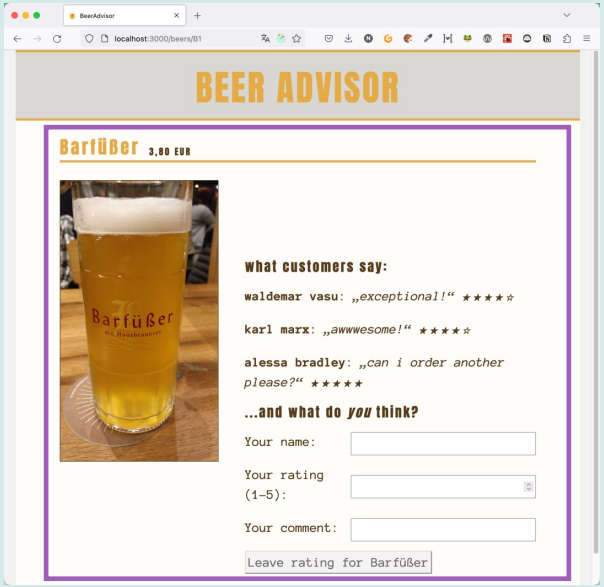



```
useEffect(loadBeerFromDatabase)
```

Zeit

# Mögliches Problem: "Wasserfall" beim Daten laden

- Klassische React-Anwendung: Daten werden nacheinander geladen...



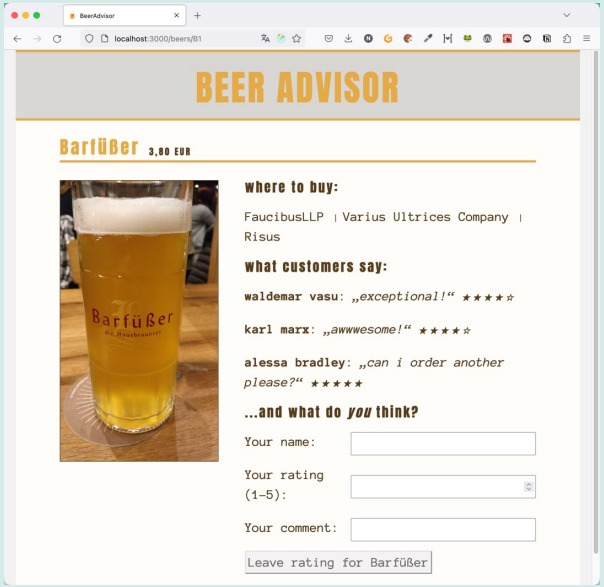

useEffect(loadBeerFromDatabase)

useEffect(loadShopsFromRemoteApi)

Zeit

# Mögliches Problem: "Wasserfall" beim Daten laden

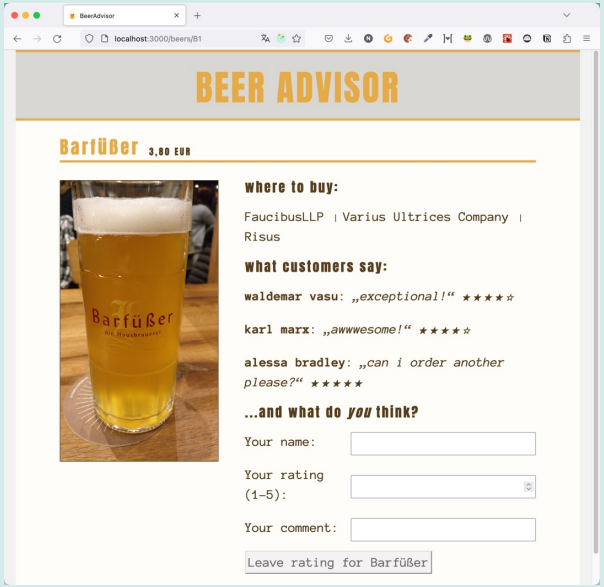
- Klassische React-Anwendung: Daten werden nacheinander geladen...




```
useEffect(loadBeerFromDatabase)

                                  useEffect(loadShopsFromRemoteApi)
```

# Mögliches Problem: "Wasserfall" beim Daten laden

- Klassische React-Anwendung: Daten werden nacheinander geladen...




```
useEffect(loadBeerFromDatabase)

                    useEffect(loadShopsFromRemoteApi)
```

Zeit

😐 Wasserfall...

## Mögliches Problem: "Wasserfall" beim Daten laden, Server-Edition

- Mit RSC bzw. in Next.JS können alle Server Components ihre eigenen Daten laden
- In Remix können bzw. müssen die loader alle Daten für die komplette Route laden

```
export default async function BeerPage({ params }: BeerPageProps) {

  const beer = await loadBeer(params.beerId);

  return (
    <div>
      <h1>{beer.name}</h1>
      <Shops beerId={params.beerId} />

      <div...>
    </div>
  );
}
```

```
async function Shops({ beerId }: {beerId: string}) {

  const shops = await loadShops(beerId);

  return (
    <div className={styles.Shops}...>
  );
}
```


😒 Wasserfall...

# Mögliches Problem: "Wasserfall" beim Daten laden, Server-Edition

- In Remix können bzw. müssen die Loader alle Daten für die komplette Route laden

Zeit

```
export async function loader({ params }: LoaderFunctionArgs) {
  const { beerId } = params;

  const shops = await loadShops(beerId);

  const beer = await loadBeer(beerId);

  return json({ shops, beer });
}
```

😒 Wasserfall...

```
export default function BeerPage() {
  const {shops, beer} = useLoaderData<typeof loader>();

  return (
    <div>
      <h1>{beer.name}</h1>

      <Shops shops={shops} />

      <div...>
    </div>
  );
}
```

# Paralleles Laden von Daten

- In Next.js können in einer Komponente mehrere Requests parallel gestartet werden

```
export default async function BeerPage({ params }: BeerPageProps) {
  const shopsPromise = loadShops(params.beerId);

  const beer = await loadBeer(params.beerId);

  return (
    <div className={styles.Beer}>
      <h1>{beer.name}</h1>
      <Shops shopsResponse={shopsPromise} />

      <div...>
    </div>
  );
}
```

## Paralleles Laden von Daten

- In Next.js können in einer Komponente mehrere Requests parallel gestartet werden
- Die (nicht aufgelösten) Promises werden dann an die Unterkomponenten gegeben

```
export default async function BeerPage({ params }: BeerPageProps) {
  const shopsPromise = loadShops(params.beerId);

  const beer = await loadBeer(params.beerId);

  return (
    <div className={styles.Beer}>
      <h1>{beer.name}</h1>
      <Shops shopsResponse={shopsPromise} />

      <div...>
    </div>
  );
}
```

## Paralleles Laden von Daten

- In Next.js können in einer Komponente mehrere Requests parallel gestartet werden
- Die (nicht aufgelösten) Promises werden dann an die Unterkomponenten gegeben
- BeerPage kann sich darstellen, ohne auf Shops warten zu müssen

```tsx
export default async function BeerPage({ params }: BeerPageProps) {
  const shopsPromise = loadShops(params.beerId);

  const beer = await loadBeer(params.beerId);

  return (
    <div className={styles.Beer}>
      <h1>{beer.name}</h1>
      <Shops shopsResponse={shopsPromise} />

      <div...>
    </div>
  );
}
```

```tsx
type ShopsProps = {
  shopsResponse: Promise<ShopsReponse>;
};

async function Shops({ shopsResponse }: ShopsProps) {
  const shops = await shopsResponse;

  return (
    <div className={styles.Shops}>
      {shops.data.map((shop, ix) => ...)}
    </div>
  );
}
```

# Paralleles Laden von Daten

- In Remix kann das Laden einzelner oder aller Daten "deferred" werden (haben wir schon gesehen)

```
export async function loader({ params }: Lo
  const { beerId } = params;

  const shopsPromise = loadShops(beerId);

  const beer = await loadBeer(beerId);

  return defer({ shopsPromise, beer });
}
```

# Paralleles Laden von Daten

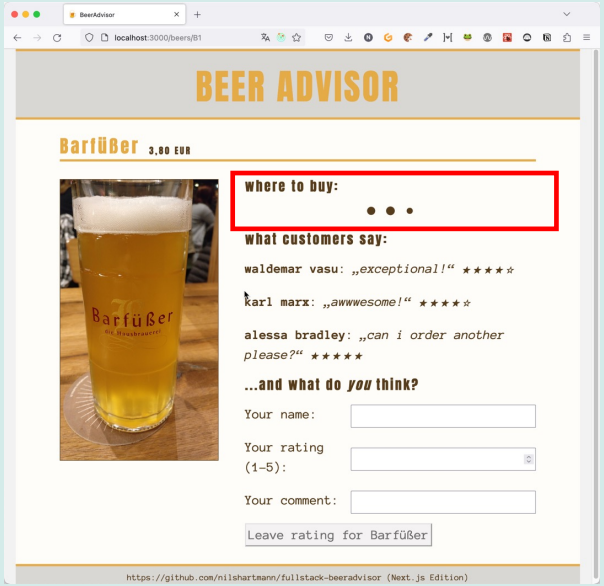
- In Remix kann das Laden einzelner oder aller Daten "deferred" werden (haben wir schon gesehen)
- In den Komponenten muss dann mit `Async` darauf gewartet werden
- Mit Render-Prop oder `useAsyncValue`

```
export async function loader({ params }: Lo
  const { beerId } = params;

  const shopsPromise = loadShops(beerId);

  const beer = await loadBeer(beerId);

  return defer({ shopsPromise, beer });
}
```

```
export default function BeerPage() {
  const { shopsPromise, beer } = useLoaderData<typeof loader>();

  return (
    <div>
      <h1>{beer.name}</h1>
      <Await resolve={shopsPromise}>
        <Shops />
      </Await>

      <div...>
    </div>
  );
}

function Shops() {
  const shops = useAsyncValue() as ShopsReponse;
  return (
    <div className={styles.Shops}>
      {shops.data.map((shop, ix) => ...)}
    </div>
  );
}
```

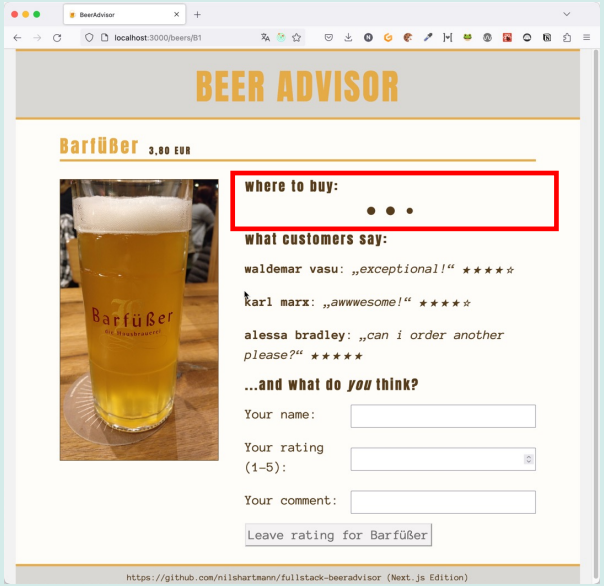
## Anforderung: Bier-Detail-Seite soll nicht auf Namen der Shops warten

- ...stattdessen während des Ladens einen Hinweis ausgeben



## Anforderung: Bier-Detail-Seite soll nicht auf Namen der Shops warten

- ...stattdessen während des Ladens einen Hinweis ausgeben
- Dazu laden wir die Daten parallel (wie gesehen) und setzen Suspense-Boundary



## Suspense

- Die React Suspense-Komponente kann um alle (asynchronen) Komponenten gelegt werden
- React bricht dann das Rendern an der Stelle, rendert den Platzhalter und schickt das Ergebnis zum Client
- Sobald die ausstehenden Promises (auf dem Server) aufgelöst werden, rendert React die fehlenden Komponenten (auf dem Server), sendet sie an den Client und baut sie dort in die Darstellung ein
- Wenn eine asynchrone Komponente (bzw. Async in Remix) nicht mit Suspense umschlossen ist, geht React in der Komponenten Hierarchie soweit nach oben, bis eine Suspense-Komponente gefunden wird
- In Next.js wäre das auf oberste Ebene die loading.tsx-Komponente

# Beispiel: Suspense in Next.js #1

```tsx
export default async function BeerPage({ params }: BeerPageProps) {

  const shopsPromise = loadShops(params.beerId);

  const beer = await loadBeer(params.beerId);

  return (
    <div>
      <h1>{beer.name}</h1>

      <Suspense fallback={<LoadingIndicator secondary />}>
        <Shops shopsResponse={shopsPromise} />
      </Suspense>

      <div...>
    </div>
  );
}
```

## Beispiel: Suspense in Next.js #2

Auf die Namen der Biere warten, die ein Geschäft verkauft

```tsx
export default async function ShopPage({ params }: ShopPageProps) {
  const { data: shop } = await fetchShop(params.shopId);

  return (
    <div>
      <h1>what&apos;s in stock</h1>

      <div>
        {shop.beers.map((beerId) => (
          <Suspense key={beerId} fallback={<LoadingIndicator />}>
            <BeerInStock beerId={beerId} />
          </Suspense>
        ))}
      </div>
    </div>
  );
}
```

```tsx
async function BeerInStock({ beerId }: BeerInStockProps) {
  const beer = await loadBeer(beerId);

  if (!beer) {
    return null;
  }

  return (
    <div className={styles.Beer}>
      <Link href={`/beers/${beerId}`}>{beer.name}</Link>
    </div>
  );
}
```

# Beispiel: Suspense in Remix

```tsx
export default function BeerPage() {
  const { shopsPromise, beer } = useLoaderData<typeof loader>();

  return (
    <div>
      <h1>{beer.name}</h1>

      <Suspense fallback={<LoadingIndicator secondary />}>
        <Await resolve={shopsPromise}>
          <Shops />
        </Await>
      </Suspense>

      <div...>
    </div>
  );
}
```

# Data Fetching

## Zusammenfassung

## Data Fetching in Remix

- In Remix werden Daten in Loader-Funktionen geladen
  - Es gibt einen Loader pro Route, der muss alle Daten für alle Komponenten der Route laden
  - Parallel werden auch die Loader der Parent-Routen ausgeführt
  - Untereinander sehen die Loader ihre Daten nicht, es kann also zu doppelten Requests kommen (wenn z.B. zwei Loader die Daten eines Users laden)
- Die geladenen Daten werden dann von der Route-Komponente in die Unterkomponenten gegeben werden (klassisches React-Pattern)
- Der JavaScript-Code für die Komponenten wird weiterhin an den Client übertragen

## Data Fetching in Next.js

- In Next.js werden Daten direkt in den Komponenten geladen (RSC).
  - Das muss nicht die Top-Level "Seiten"-Komponente sein, sondern irgendwo in der Hierarchie
  - Next.js unterdrückt doppelte fetch-Aufrufe innerhalb eines Requests
  - Asynchroner Code / arbeiten mit Promises direkt in den Komponenten mit Suspense möglich
  - JavaScript-Code von RSC wird nicht auf den Client übertragen

## Data Fetching in Next.js

- React Server Components lassen sich teilweise schon zur Buildzeit rendern
- Next.js unterscheidet zwischen zwei Render-Modi:
  - Es gibt statische Routen, die nur zur Buildzeit gerendert und zur Laufzeit fertig ausgeliefert werden (ähnlich wie statischer Content)
  - Dynamische Routen werden bei jedem Request neugerendert, können aber gecached werden (z.B. eine bestimmte Zeit lang)
- Das ist sehr flexibel, kann aber auch verwirrend sein, zumal das opt-out und nicht opt-in und an vielen Stellen transparent ist

## Data Fetching in Next.js und Remix

- Beide Frameworks rendern auf dem Server vor (SSR)
- Eventuell Platzhalter für fehlende Daten
- Laden dann das JavaScript. Dadurch ist die Seite schnell sichtbar, aber u.U. erst später interaktiv bedienbar

# Data Mutations

# Verändern von Daten: Hinzufügen einer Bewertung

# Verändern von Daten: Hinzufügen einer Bewertung

## Anforderungen:

- Submit-Button ist disabled, bis alle Felder ausgefüllt sind
- Während Speicher-Request läuft, soll Submit-Button ebenfalls disabled sein und Meldung angezeigt werden
- Nach dem Submit wird ggf. eine Fehlermeldung angezeigt
- Nach dem erfolgreichen Speichern wird neue Bewertung in der Liste angezeigt
- Nach dem Speichern wird das Formular geleert

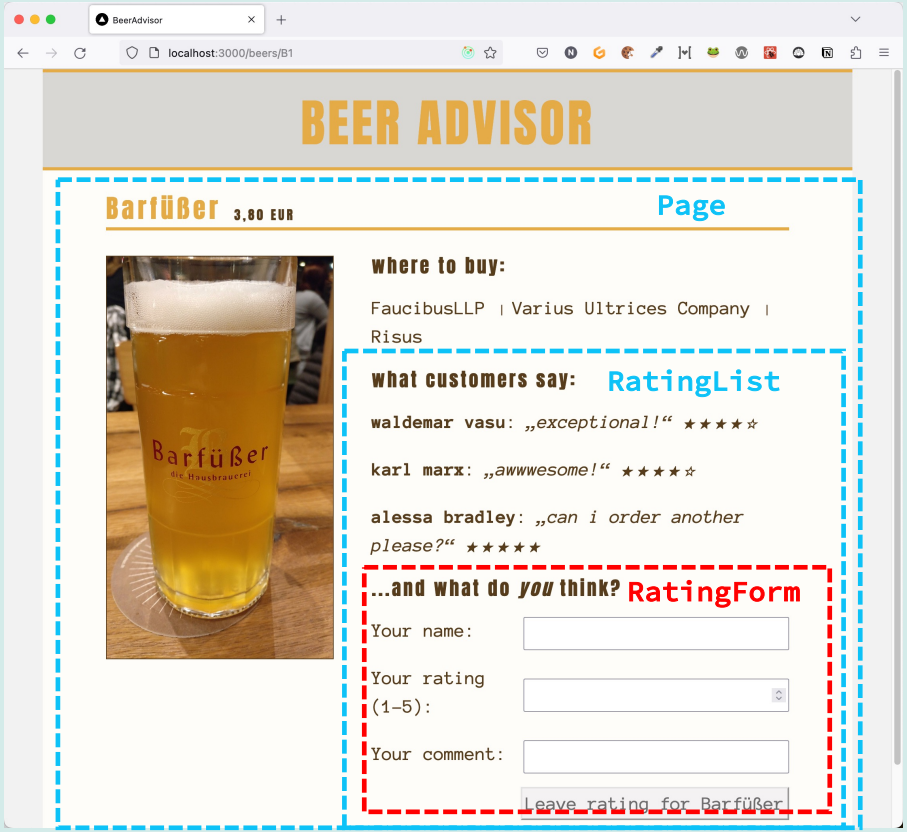
# Data Mutations

## Next.js

# MUTATIONS

## Verändern von Daten:

### Hinzufügen einer Bewertung

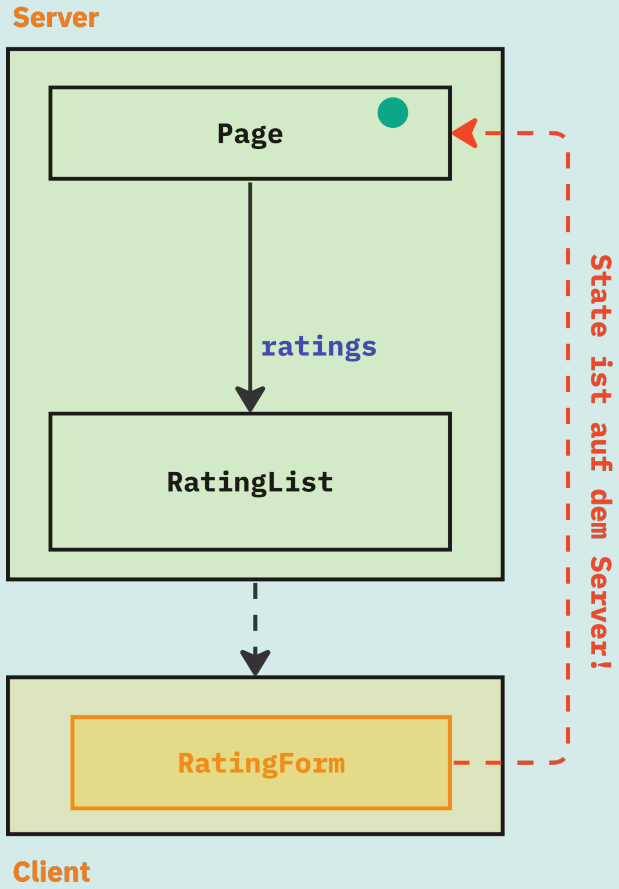



Server-Komponente

Client-Komponente

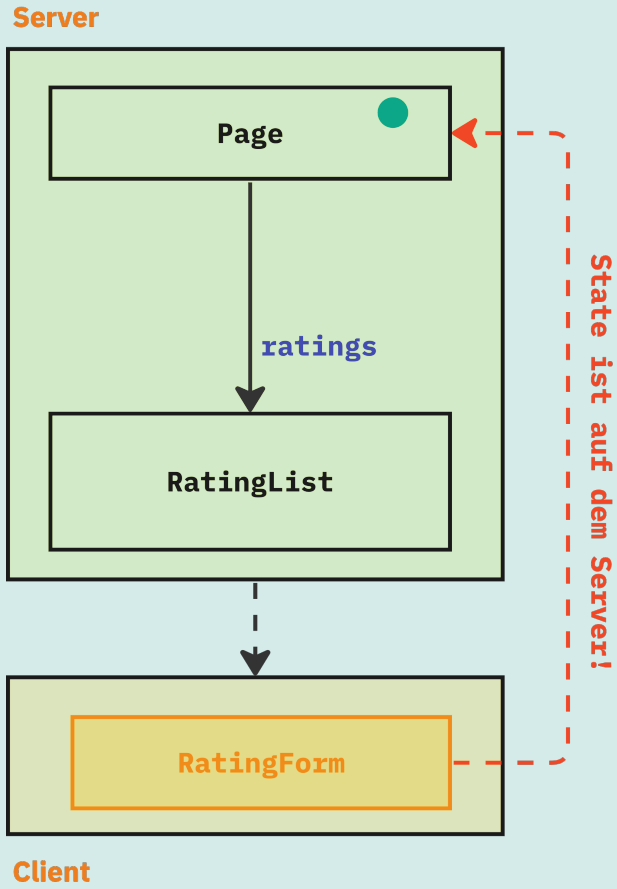
## Verändern von Daten

- Nach dem Verändern von Daten muss die UI aktualisiert werden
- Mangels State auf dem Client geht das aber nicht wie bislang



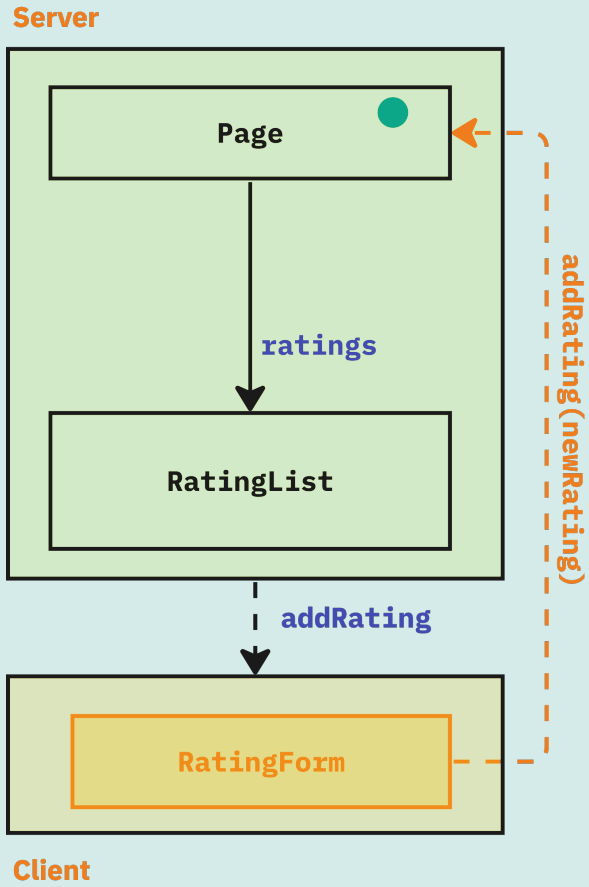
## Verändern von Daten

- Nach dem Verändern von Daten muss die UI aktualisiert werden
- Mangels State auf dem Client geht das aber nicht wie bislang
- Der Server muss nach Datenänderungen aktualisierte UI liefern



## Server Actions

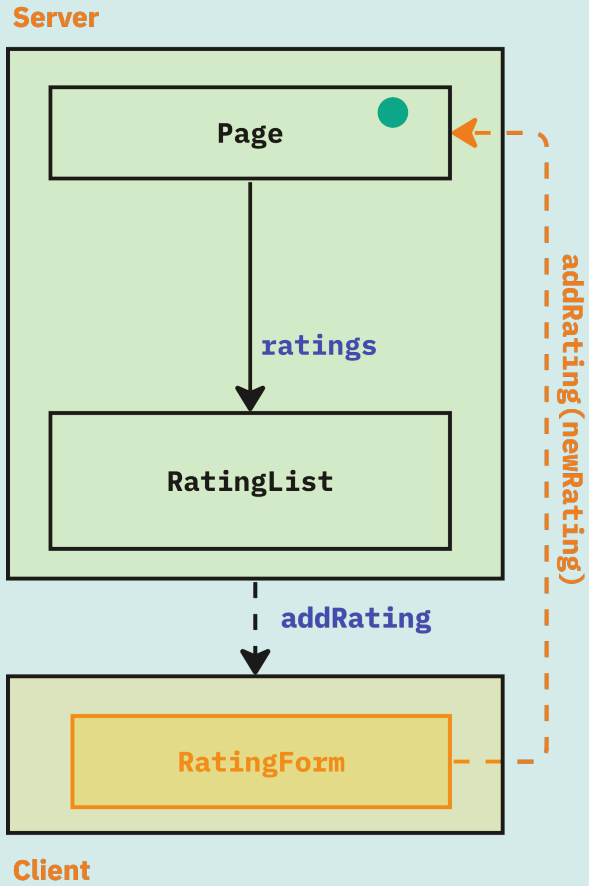
- **Experimentelles** React-Feature
- Das ist eine Art Remote Funktion, die aus einer Server- oder Client-Komponente aufgerufen werden kann
- Ausgeführt wird die Funktion auf dem Server
- In Next.js kann man darin angeben, welche Routen sich durch die Änderung der Daten verändert haben und neu gerendert werden müssen



# MUTATIONS

## Server Actions

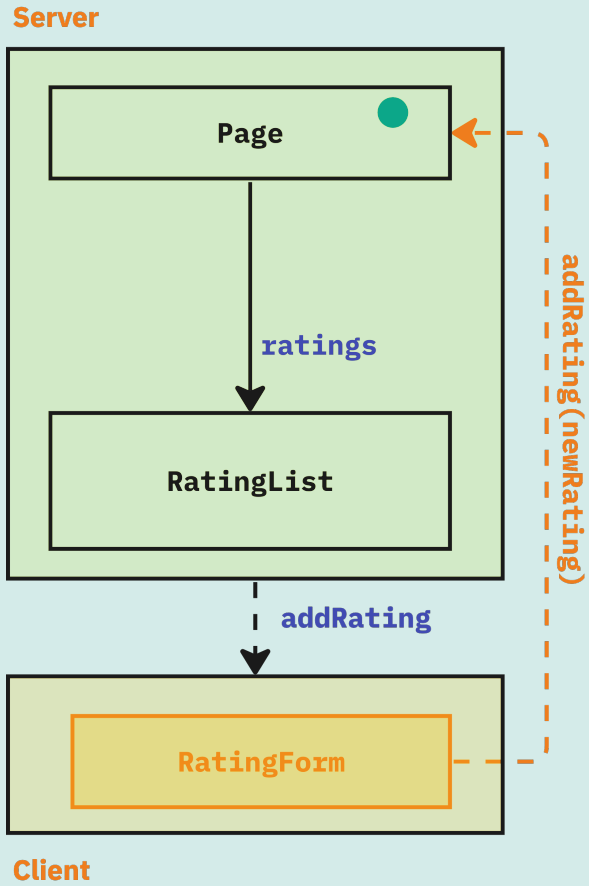
```
"use server";

export async function addRating({
  beerId, username, stars, comment
}: AddRatingRequestBody) {

  const errors = validateInput({ beerId, username, stars, comment });

  if (errors) {
    return {
      result: "error", errors, data: { username, stars, comment },
    };
  }

  await prisma.rating.create({
    data: {beerId: beerId...},
  });

  revalidatePath(`/beers/${beerId}`);

  return { result: "success" };
}
```



# MUTATIONS

## Server Actions

```
"use server";

export async function addRating({
  beerId, username, stars, comment
}: AddRatingRequestBody) {

  const errors = validateInput({ beerId, username, stars, comment });

  if (errors) {
    return {
      result: "error", errors, data: { username, stars, comment },
    };
  }

  await prisma.rating.create({
    data: {beerId: beerId...},
  });

  revalidatePath(`/beers/${beerId}`);

  return { result: "success" };
}
```



# MUTATIONS

## Formular

- Die Server Action-Funktion wird aus dem Formular heraus aufgerufen
- Next.js stellt dafür automatisch einen Endpunkt zur Verfügung
- Das zurückgelieferte Ergebnis kann zur Aktualisierung des Formulars benutzt werden

```
export default function RatingForm({ beerName, beerId }: RatingFormProps) {
  const [username, setUsername] = useState("");
  const [comment, setComment] = useState("");
  const [stars, setStars] = useState("");

  const [errors, setErrors] = useState<Record<string, string>>({});

  const onSubmit = async () => {
    const response = await addRating({username: username...});

    if (response.result === "success") {
      setUsername(""); setComment(""); setStars(""); setErrors({});
      return;
    }

    setErrors(response.errors);
  };

  const buttonEnabled = !!username && !!stars && !!comment;

  return (
    <div className={styles.Form}>
      <form action={onSubmit}>
        <fieldset...>
      </form>
    </div>
  );
}
```

## Anforderung: Button disablen beim Speichern

- Der (experimentelle) Hook `useFormStatus` von React gibt Informationen über den laufenden Submit-Request zurück

```
export default function RatingForm({ beerName, beerId }: RatingFormProps) {
  // ...

  const onSubmit = async () => {...};
  const buttonEnabled = !!username && !!stars && !!comment;

  return (
    <div className={styles.Form}>
      <form action={onSubmit}>
        <fieldset...>
        <SubmitButton disabled={!buttonEnabled}>
          <>Leave rating for {beerName}</>
        </SubmitButton>
      </form>
    </div>
  );
}


function SubmitButton({ children, disabled }: SubmitButtonProps) {
  const formStatus = useFormStatus();

  return (
    <button type={"submit"} disabled={disabled || formStatus.pending}>
      {formStatus.pending ? <>Saving </> : children}
    </button>
  );
}
```

# Data Mutations

## Remix

## Data Mutations in Remix

- Analog zu zur Loader-Funktion gibt es in Remix eine action-Funktion, die zu einer Route gehören
- Auch diese Funktion muss in der Route-Datei enthalten sein
- Diese Funktion erhält das Request-Objekt, das zum Beispiel die Formular-Daten enthält
- Nach Ausführung der Funktion sieht Remix die Route als "verändert" an und aktualisiert die Darstellung

```
export async function action({ request, params }: ActionFunctionArgs) {
  const { beerId } = params;
  invariant(beerId, "beerId param missing");

  const formData = await request.formData();
  const username = getFormString(formData, "username");
  const stars = parseInt(getFormString(formData, "stars"));
  const comment = getFormString(formData, "comment");

  const errors = validateFormData(formData);
  if (errors) {
    return json({
      result: "error",
      errors,
    });
  }

  const id = uuid();
  await prisma.rating.create({
    data: { id,  beerId,  username,  stars, comment }
  });

  return json({
    result: "success",
  });
}
```

## Data Mutations in Remix

- Es gibt eine Remix-spezifische Form-Komponente
- Die verhält sich fast wie die HTML form-Komponente, bietet aber die Möglichkeit, sorgt aber beim Submit dafür, dass die Action automatisch aufgerufen wird
- Mit `useActionData` kann man auf das Ergebnis der Action zugreifen

```tsx
export default function RatingForm({ beerName }: RatingFormProps) {
  const data = useActionData<typeof action>();
  const errors = data?.result === "error" ? data.errors : undefined;

  return (
    <div className={styles.Form}>
      <Form method={"POST"}>
        <fieldset...>
      </Form>
    </div>
  );
}
```

## Disablen des Buttons Remix

- Mit dem `useNavigation-Hook` von Remix kann der Request Status überprüft werden

```
export default function RatingForm({ beerName }: RatingFormProps) {
  const navigation = useNavigation();
  const data = useActionData<typeof action>();
  const errors = data?.result === "error" ? data.errors : undefined;

  return (
    <div>
      <Form method={"POST"}>
        <fieldset...>

        <button disabled={navigation.state === "submitting"} type={"submit"}>
          Leave rating for {beerName}
        </button>

        {navigation.state === "submitting" && <p>Saving...</p>}
      </Form>
    </div>
  );
}
```

## Validierung in Remix

- Remix propagiert nahezu durchgängig die serverseitige Validierung
- Client-seitige Validierung nur mit HTML Constraint API
- Begründet wird das unter anderem damit, dass man dann für die Formulare nicht unbedingt JavaScript braucht und man ohnehin auf dem Server validieren muss

- Anforderung "Submit-Button disablen solange nicht alle Felder ausgefüllt sind" ist zwar mit Remix möglich, verstößt aber offensichtlich gegen dessen Philosophie

Next.js

# Sonstiges

Remix

# Server Funktionalität

## In jedem (realistischen) Fall wird ein JavaScript-basierter Server benötigt

- Beide Frameworks unterstützen mehrere Platformen
- Man kann eigene Adapter schreiben
- Das ist aber ein großer Unterschied zu klassischen Single-Page-Anwendungen
  - Betrieb!
  - Sicherheit!
  - Monitoring!

# SERVER REQUESTS

## Ein Teil der Anwendung läuft auf dem Server...

- Informationen, die vom Client kommen, können bzw. müssen verwendet werden
  - Cookies, Headers, URL, Search Parameter etc.
- In Next.JS erfolgt der Zugriff über Next-spezifische Funktionen, die man in einer Komponente verwenden kann
  - `cookies(), headers()`
  - Verändern teilweise das Rendering-Verhalten
- In Remix werden wird ein Request-Objekt an die Loader per Argument übergeben
  - Entspricht dem Request-Objekt aus der Browser fetch API

## Middleware

- Eine Middleware erlaubt es, "pauschal" auf eingehende Requests zu reagieren und auch die Responses zu verarbeiten
    - Beispiel: Security oder Logging
- In Next.JS gibt es eine `middleware`-Funktionen, die für jeden Request aufgerufen wird, und die zum Beispiel Redirects machen kann oder Cookies und Response Header setzen
- In Remix gibt es keine Middlewares

# API ROUTES

## Mit API Routes kann man nicht-UI Routen bauen (z.B. für API)

- In Next.JS verwendet man dazu eine route.ts-Datei, die im Verzeichnis der entsprechenden Route liegt
    - Diese kann GET, POST, PATCH etc. Funktionen exportieren
- In Remix gibt es "Resource Routes"
    - Das sind Dateien, die zwar eine Loader- oder Action-Funktion exportieren aber keine Komponente

# Sonstiges

## Static Site Generation (SSG)

- Next.js unterscheidet zwischen statischem und dynamischem Rendering
  - Bei statischem Rendering werden die Komponenten bereits im Build gerendert und nicht mehr bei einem Request. Das geht nur für Komponenten, die keine Request-Informationen benötigen
  - Dynamisches Rendering rendert eine Komponente bei jedem Request
- Remix macht diese Unterscheidung nicht, hier wird immer dynamisch gerendert

## Styling / CSS

- CSS-Bibliotheken funktionieren in der Regel mit beiden Frameworks
  - Insbesondere CSS Modules und TailwindCSS
- Bei JS-in-CSS muss man aufpassen, es kann Probleme geben
  - Support wird aber besser
- Next.js bringt Komponenten mit, die Font- und Bild-Loading optimieren

## Bundling

- Bundler ist integriert, im Prinzip "Implementierungsdetail"
  - Wir müssen uns drauf verlassen, dass der gewählte Bundler und Toolstack funktioniert
  - Das gilt natürlich auch für nicht JS/TS-Artefakte wie CSS, Bilder etc.
- Next.JS entwickelt TurboPack ("Webpack Nachfolger"), noch Beta
- Remix verwendet intern esbuild
- Beide haben einen dev-Server
  - Remix wirkt unstabil (nach Fehler häufig Neustart notwendig, Port ändert sich)

## TypeScript

- TS wird von beiden unterstützt
    - Sowohl was die APIs als auch was den Toolchain angeht
- Insgesamt wirkt der TS-Support in Remix etwas lieblos, z.B. Params etc. nicht als Generic implementiert

# Fazit

## Data Fetching

- **Remix: sehr einfaches Modell**
  - Lifecycle einfach
  - Zugriff auf Daten (useLoaderData) sehr einfach
  - Klare Trennung Server (loader) und Client (Komponenten), aber Client-Kompnenten werden immer auch komplett auf den Client übertragen (kein RSC)
  - Aber: Macht Kompromisse, z.B. können Loader nicht auf Daten von anderen Loadern zugreifen

# FAZIT

## Data Fetching

- **Next.js**: mehr React Standard APIs
- von React explizit empfohlen
- höhere Verbreitung, größeres Team (?)
- Verwendung von asynchronem Code sehr einfach (Standard JS)

## Routing

- **Beide Frameworks sorgen für schnelle erste Darstellung**
  - Serverseitiges Rendern
  - Einsparen von JavaScript, das in den Browser geladen werden muss

# FAZIT

## Routing

- **Beide Frameworks sorgen für schnelle erste Darstellung**
  - Serverseitiges Rendern
  - Einsparen von JavaScript, das in den Browser geladen werden muss
- **Remix Konventionen finde ich verwirrend**
  - Unterschied zwischen Pages und Layout nicht trivial
  - Namenskonventionen (flache Hierachie) finde ich unpassend

# FAZIT

## Routing

- **Beide Frameworks sorgen für schnelle erste Darstellung**
  - Serverseitiges Rendern
  - Einsparen von JavaScript, das in den Browser geladen werden muss
- **Remix Konventionen finde ich verwirrend**
  - Unterschied zwischen Pages und Layout nicht trivial
  - Namenskonventionen (flache Hierachie) finde ich unpassend
- **Next.js trennt klarer zwischen Seite und Layout**
  - Hierarchische Verzeichnisstruktur, so wie man das gewohnt ist
  - Nachteil: viele page.tsx und layout.tsx-Dateien

**NILS HARTMANN**
https://nilshartmann.net

https://reactbuch.de


# Vielen Dank!

Slides: https://react.schule/wdc2023

Fragen & Kontakt: nils@nilshartmann.net

Twitter: @nilshartmann