

NILS HARTMANN

React

Server Components

Slides: <https://react.schule/wjax2021-server-components>

NILS HARTMANN

nils@nilshartmann.net

Freiberuflicher Entwickler, Architekt, Trainer aus Hamburg

Java
JavaScript, TypeScript
React
GraphQL

Trainings & Workshops



<https://reactbuch.de>

[HTTPS://NILSHARTMANN.NET](https://nilshartmann.net)

"-experimental-

```
"dependencies": {  
  "react": "0.0.0-experimental-7ec4c5597",  
  "react-dom": "0.0.0-experimental-7ec4c5597",  
  "react-fetch": "0.0.0-experimental-7ec4c5597",  
  "react-fs": "0.0.0-experimental-7ec4c5597",  
  "react-pg": "0.0.0-experimental-7ec4c5597",  
  "react-server-dom-webpack": "0.0.0-experimental-7ec4c5597",
```



CURRENT STATE

A screenshot of a web browser displaying a blog application titled "React Training Blog". The page shows three blog posts listed vertically. Each post includes a date, a title, a snippet of the content, and a "Read this Blog Post" button.

- Post 1:** Date: 10.01.2021, Title: **Keep calm and learn React!**, Snippet: Pommy ipsum air one's dirty linen fork out plum pu..., Button: Read this Blog Post
- Post 2:** Date: 17.04.2020, Title: **Increasing React developer experience**, Snippet: Tweeting a baseball.Sit on human they not getting ..., Button: Read this Blog Post
- Post 3:** Date: 02.04.2020, Title: **Using Redux with care**, Snippet: Duis autem vel eum iriure dolor in hendrerit in vu..., Button: Read this Blog Post

The sidebar on the right is titled "Tags" and contains a grid of tags:

- React, Tutorial
- Bootstrap, JavaScript
- Best Practice, DX
- Context, Redux, State
- URL, CSS, Routing
- WebDev, Marzipan

<https://github.com/nilshartmann/server-components-blogexample>

EIN BEISPIEL...

EIN BEISPIEL

Was macht die Beispiel-Anwendung aus?

- Minimale Benutzer-Interaktionen (PostEditor)
- ...im Verhältnis sehr viel statischer Content

EIN BEISPIEL

Beispiel: Post-Komponente

- Rendert statischen Content (ein Blog-Post)
- ...benötigt dafür aber 3rd-Party-Libs!

```
import moment from "moment";           ..... Datum formatieren
import marked from "marked";           ..... Markdown nach HTML

export default function Post({ post }) {
  const date = moment(post.date).format("DD.MM.YYYY");
  const body = marked.parse(post.body);

  return <article className="Container">
    <p className="Date">{date}</p>
    <h1>{post.title}</h1>
    <div dangerouslySetInnerHTML={{ __html: body }} />
  </article>
}
```

EIN BEISPIEL

Beispiel: Post-Komponente

- Rendert statischen Content (ein Blog-Post)
- ...benötigt dafür aber 3rd-Party-Libs!

```
import moment from "moment";           290 K (gzipped: 72K)
import marked from "marked";           36 K (gzipped: 11 K)

export default function Post({ post }) { . . . }
```

- ...die bringen einiges an JavaScript-Code mit
- ...den wir zur Laufzeit aber eigentlich nicht brauchen

EIN BEISPIEL

Beispiel: Post-Komponente

- Rendert statischen Content (ein Blog-Post)
- ...benötigt dafür aber 3rd-Party-Libs!

```
import moment from "moment";           290 K (gzipped: 72K)
import marked from "marked";           36 K (gzipped: 11 K)

export default function Post({ post }) { . . . }
```

- ...die bringen einiges an JavaScript-Code mit
- ...den wir zur Laufzeit aber eigentlich nicht brauchen
- ...das kostet unnötig Bandbreite (Code muss übertragen werden)
- ...und Performance (Code muss geparsst und ausgeführt werden)
- ...nur um statisches HTML anzuzeigen 😢

Zero-Bundle-Size

Server

Components

Idee

- Server Components werden nur auf dem Server ausgeführt
- Sie stehen nicht auf dem Client zur Verfügung
- Der Server schickt lediglich eine *Repräsentation der UI, aber keinen Code*

👉 "Zero-Bundle-Size"

SERVER COMPONENTS

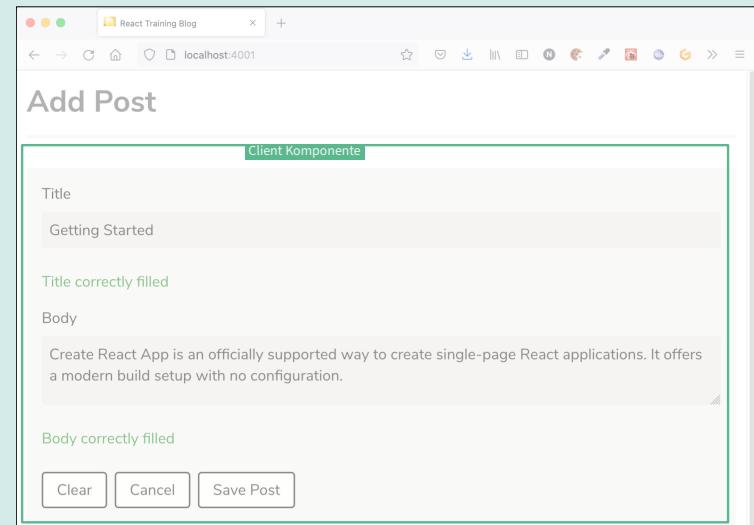
Drei Arten von Komponenten

SERVER COMPONENTS

Drei Arten von Komponenten

- Client-Komponenten

- wie bisherige React-Komponenten, werden *nur* auf dem Client ausgeführt
- können keine Server-Komponenten verwenden



SERVER COMPONENTS

Drei Arten von Komponenten

- **Server-Komponenten (Neu!)**

- werden *nur* auf dem Server ausgeführt
- liefern UI (!) zum React-Client zurück
- API: "normale" React-Komponenten (JS/TS, JSX, ...)

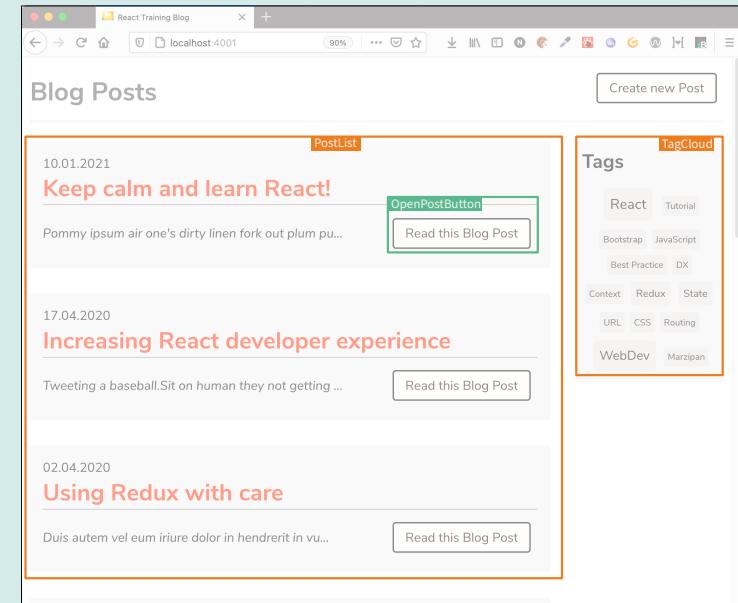
Drei Arten von Komponenten

- **Server-Komponenten (Neu!)**

- werden *nur* auf dem Server ausgeführt
- liefern UI (!) zum React-Client zurück
- API: "normale" React-Komponenten (JS/TS, JSX, ...)
- Restriktionen: kein useState, useEffect, Browser APIs
- aber: können Server Umgebung und Ressourcen nutzen (!)
 - Datenbanken
 - Filesystem

Weiterhin ein Komponenten-Baum

- Ein Teil der Komponenten kommt jetzt jetzt vom Server...
- Der Server rendernt die Komponenten, bis er auf eine Client-Komponente trifft
- **Server Komponenten bzw. deren JS-Code sind nicht auf dem Client vorhanden!**

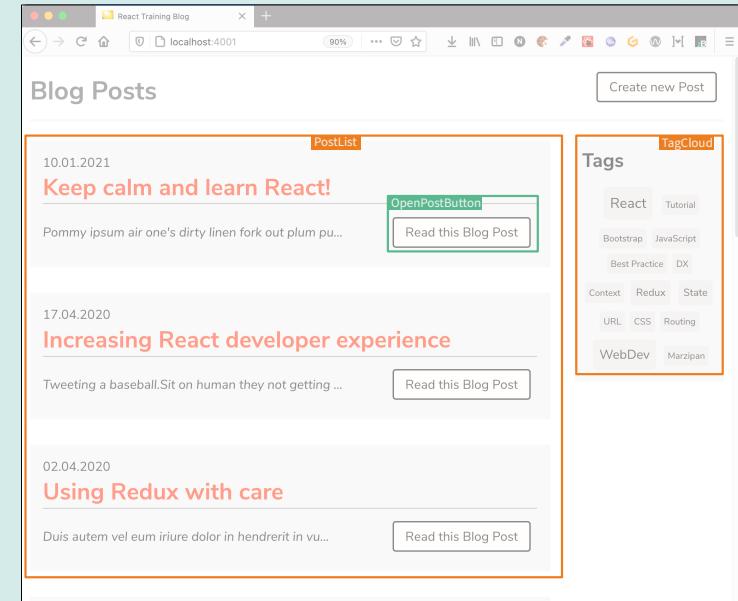


Weiterhin ein Komponenten-Baum

- Ein Teil der Komponenten kommt jetzt vom Server...
- Der Server rendert die Komponenten, bis er auf eine Client-Komponente trifft
- **Server Komponenten bzw. deren JS-Code sind nicht auf dem Client vorhanden!**

Demo

- Server-Komponenten PostListPage, PostList und TagCloud zeigen
- Server-Komponenten "PostList" und "TagCloud" gibt es als Komponenten, aber nicht auf dem Client (-> React Dev Tools)
- Server-Komponenten sind als HTML im DOM (Inspektor CSS-Klassen)
- Netzwerkverkehr (/react): UI Fragmente



Drei Arten von Komponenten

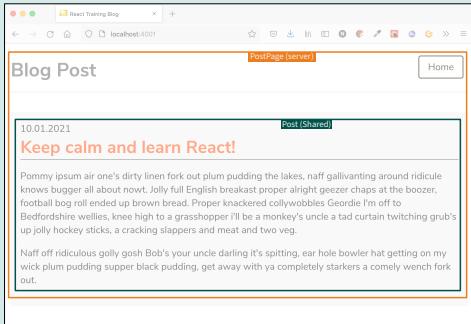
- **Shared Komponenten (Neu!)**

- werden auf dem Server und dem Client ausgeführt
 - es gelten also die Restriktionen von Server und Client-Komponenten
 - können von Server- und Client-Komponenten verwendet werden
-
- der entsprechende JavaScript-Code wird erst auf den Client übertragen, wenn er wirklich benötigt wird

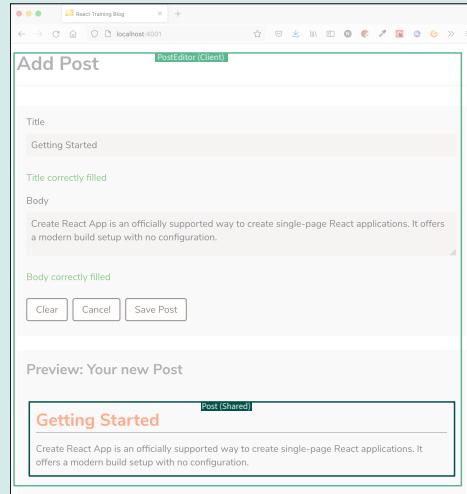
SERVER COMPONENTS

Shared Components

- JS-Code wird erst bei Bedarf auf den Client geladen (ansonsten nur UI)



Verwendung "Post"-Komponente 1:
innerhalb einer Server-Komponente



Verwendung "Post"-Komponente 2:
innerhalb einer Client-Komponente



Demo

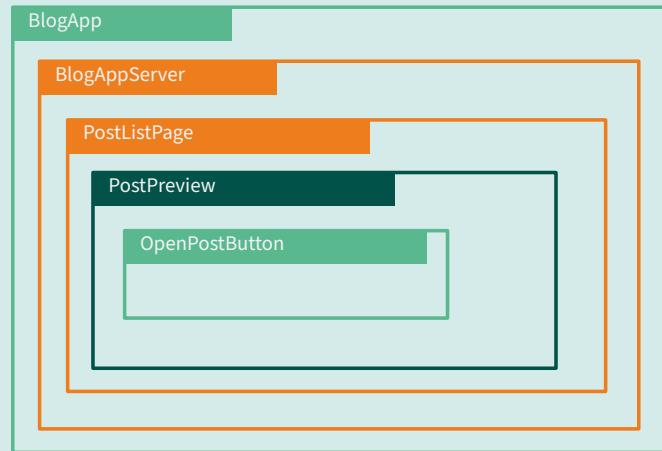
- Post-Seite: keine "Post-Komponente"
- PostEditor: Post-Komponente wird geladen (-> Netzwerk-Tab) und als Komponente gerendert (-> Dev Tools)
- Netzwerk-Tab: unten ist der JS-Code der Komponente

SERVER COMPONENTS

Rendering und Updates

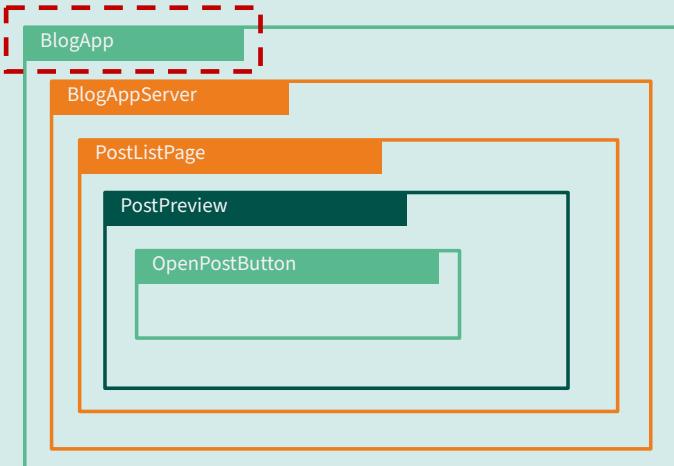
SERVER COMPONENTS

Wenn das eine normale Client-App wäre...



SERVER COMPONENTS

Wenn das eine normale Client-App wäre...



BlogApp würde Liste oder Einzel-Darstellung rendern

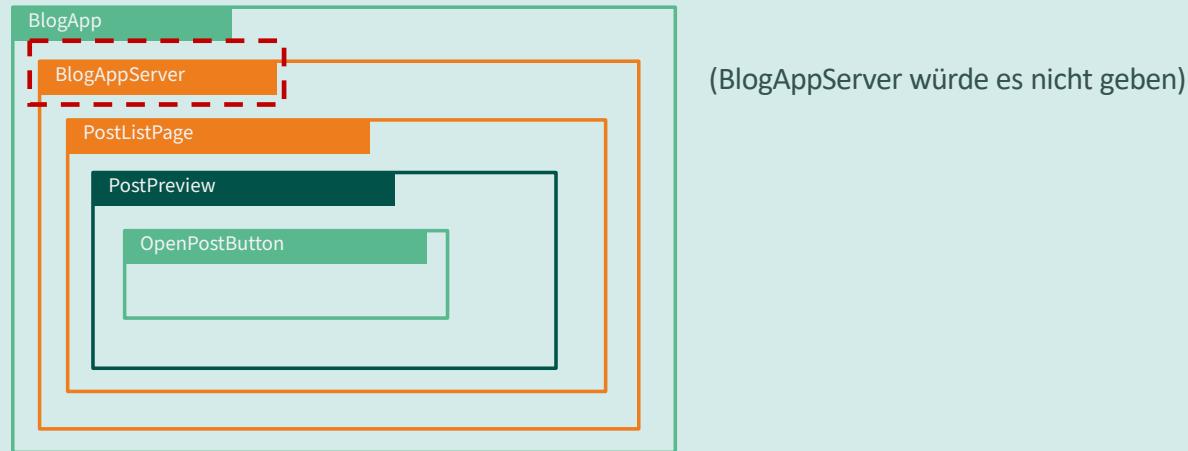
```
// Pseudo Code !!
BlogApp() {
  const [postId] = useState();

  if (postId) {
    return <PostPage id={postId} />
  }

  return <PostListPage />
}
```

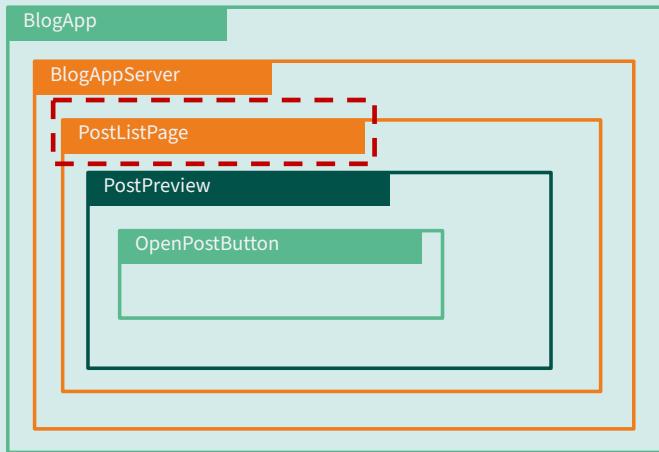
SERVER COMPONENTS

Wenn das eine normale Client-App wäre...



SERVER COMPONENTS

Wenn das eine normale Client-App wäre...



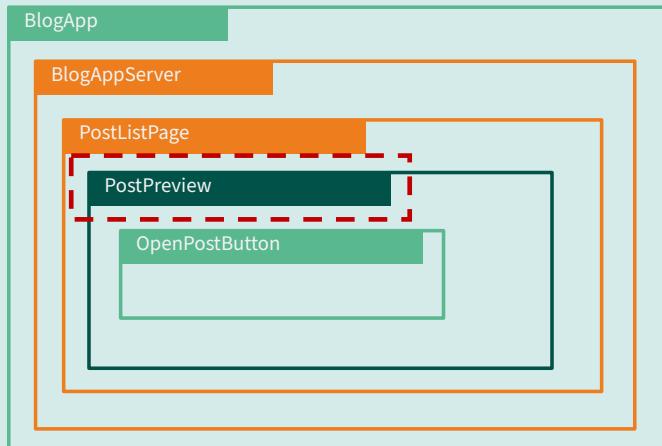
PostListPage würde Daten laden und Post-Vorschauen rendern

```
// Pseudo Code !!
PostListPage() {
  const [posts, setPosts] = useState();
  useEffect(() => {
    const posts = loadPosts();
    setPosts(posts);
  });
}

return
  posts.map(p=><PostPreview post={p}/>
}
```

SERVER COMPONENTS

Wenn das eine normale Client-App wäre...

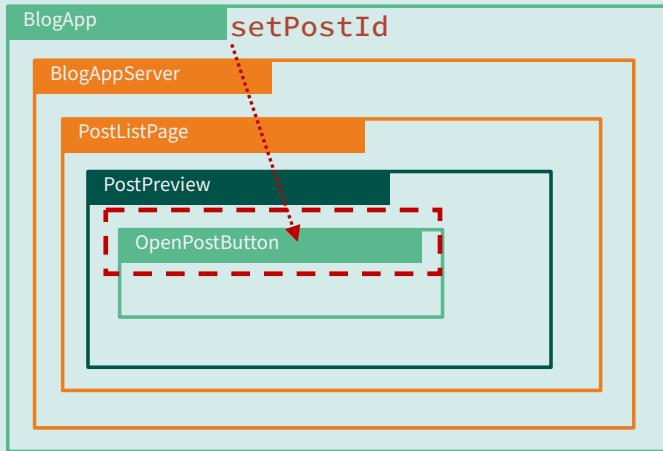


PostPreview würde Post darstellen und Knopf rendern
// Pseudo Code !!

```
PostPreview({post}) {  
  
  return <div>  
    {post.title}  
    <OpenPostButton post={post} />  
  </div>  
}
```

SERVER COMPONENTS

Wenn das eine normale Client-App wäre...



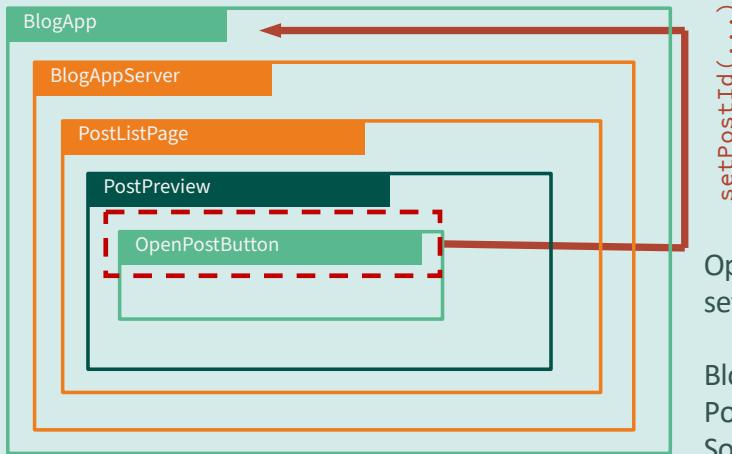
OpenPostButton würde neue Post-Id setzen, z.B.
über durchgereichte setPostId-Funktion

// Pseudo Code !!

```
OpenPostButton({post}) {  
  return <button  
    onClick={  
      () => setPostId(post.id)  
    }Show Post</button>  
}
```

SERVER COMPONENTS

Wenn das eine normale Client-App wäre...



OpenPostButton würde neue Post-Id in BlogApp setzen:

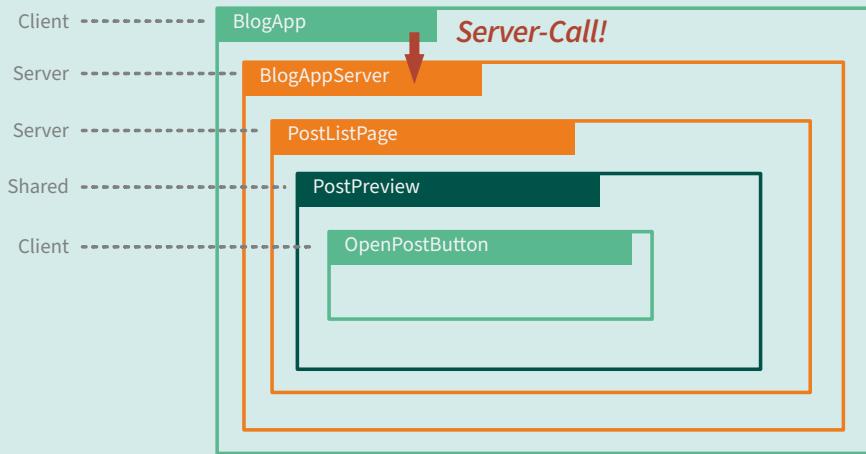
BlogApp würde neu gerendert ✓

PostPage würde gerendert und zeigt Post an ✓

Soweit, so bekannt ✓

SERVER COMPONENTS

Wenn das eine normale Client-App wäre... ...ist es aber nicht!



BlogApp will (muss?) **Server**-Komponenten darstellen

SERVER COMPONENTS

Wenn das eine normale Client-App wäre... ...ist es aber nicht!



BlogApp

- löst Server Request aus

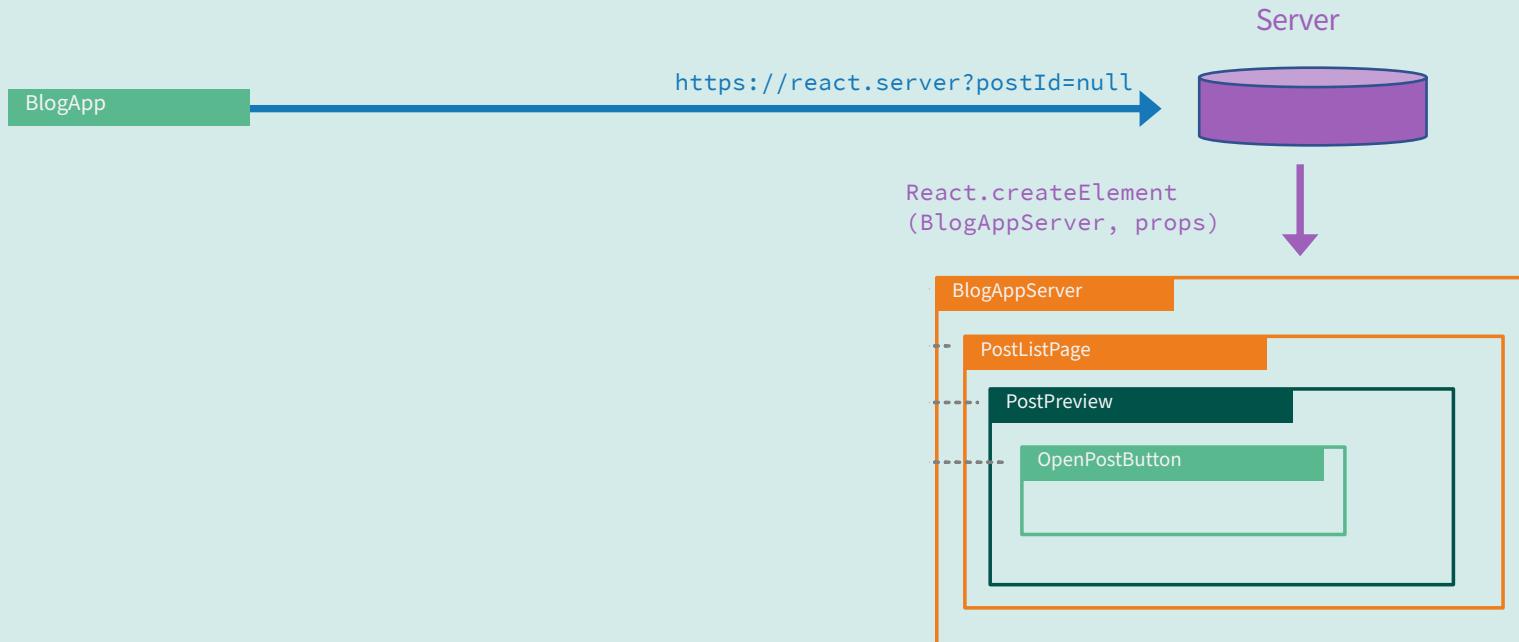
```
function BlogApp() {  
  const [postId, set postId] = // aus Context  
  
  const response = [createFromFetch]("https://react.server?postId=" + postId);  
  
  // ...  
}
```

„Properties“



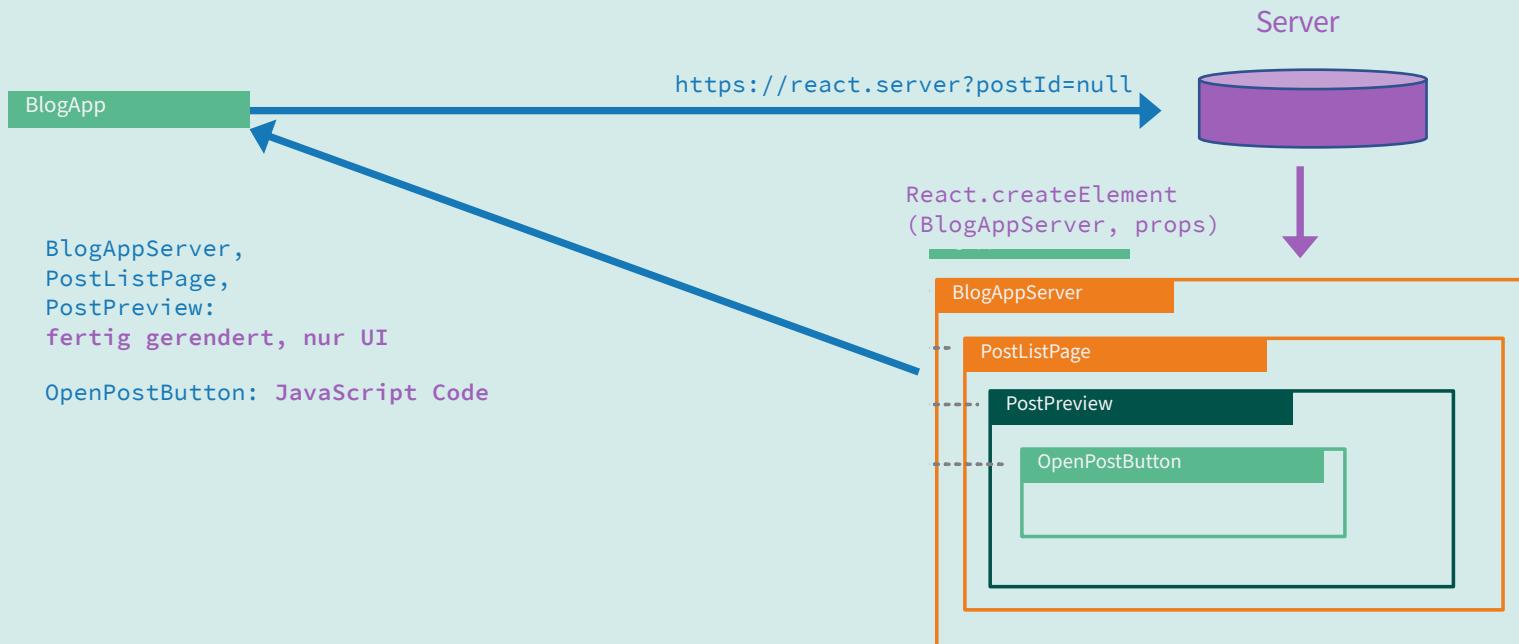
SERVER COMPONENTS

Wenn das eine normale Client-App wäre... ...ist es aber nicht!



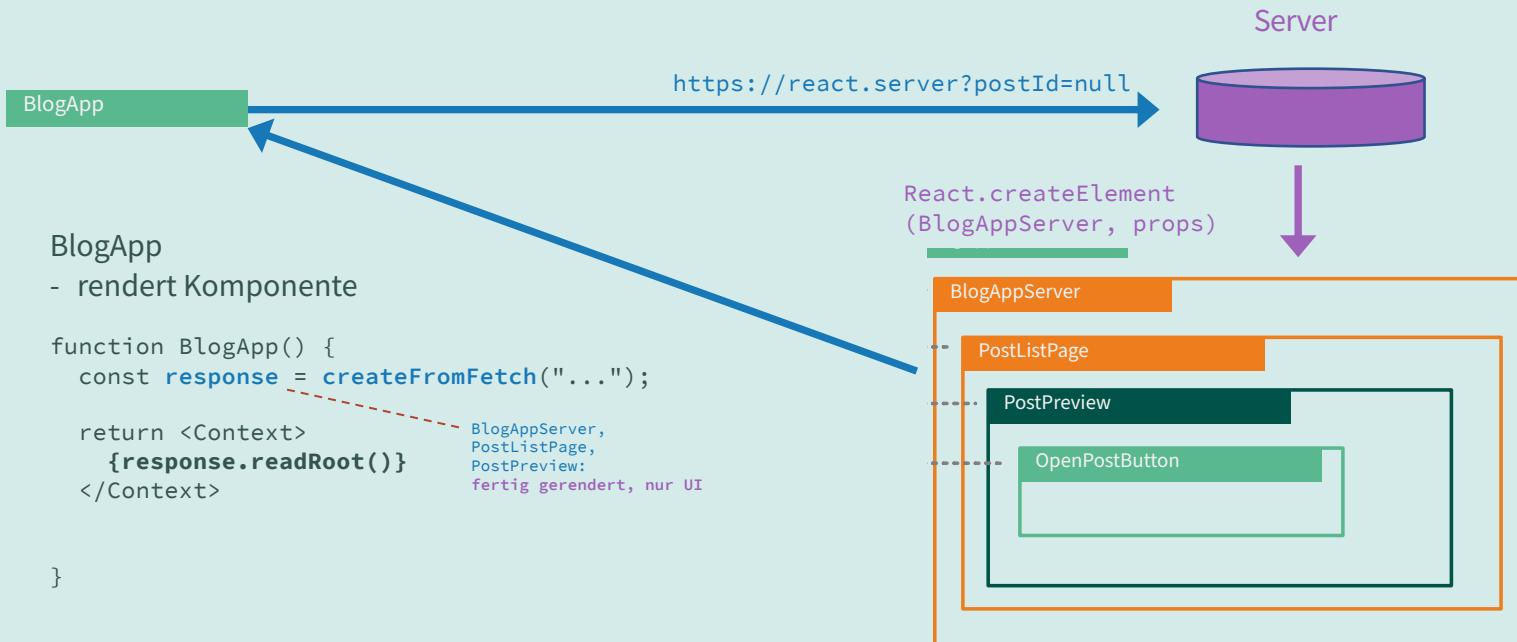
SERVER COMPONENTS

Wenn das eine normale Client-App wäre... ...ist es aber nicht!



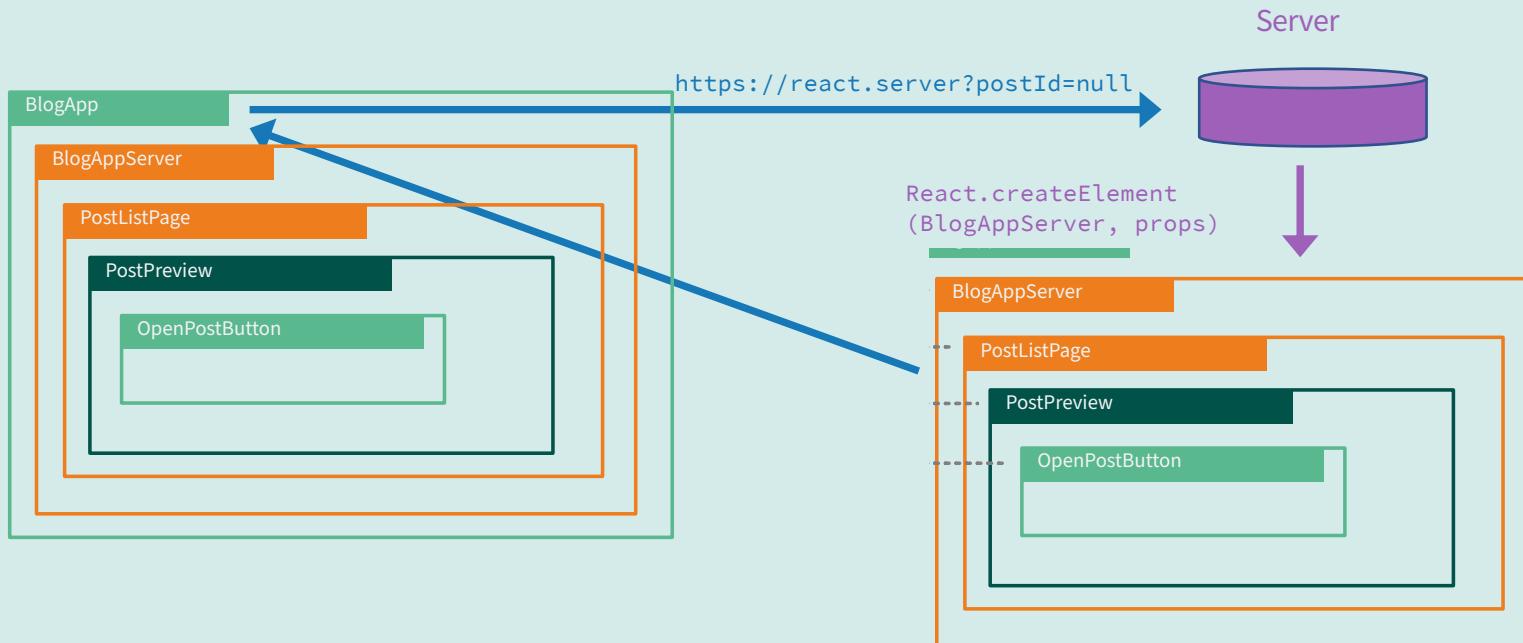
SERVER COMPONENTS

Wenn das eine normale Client-App wäre... ...ist es aber nicht!



SERVER COMPONENTS

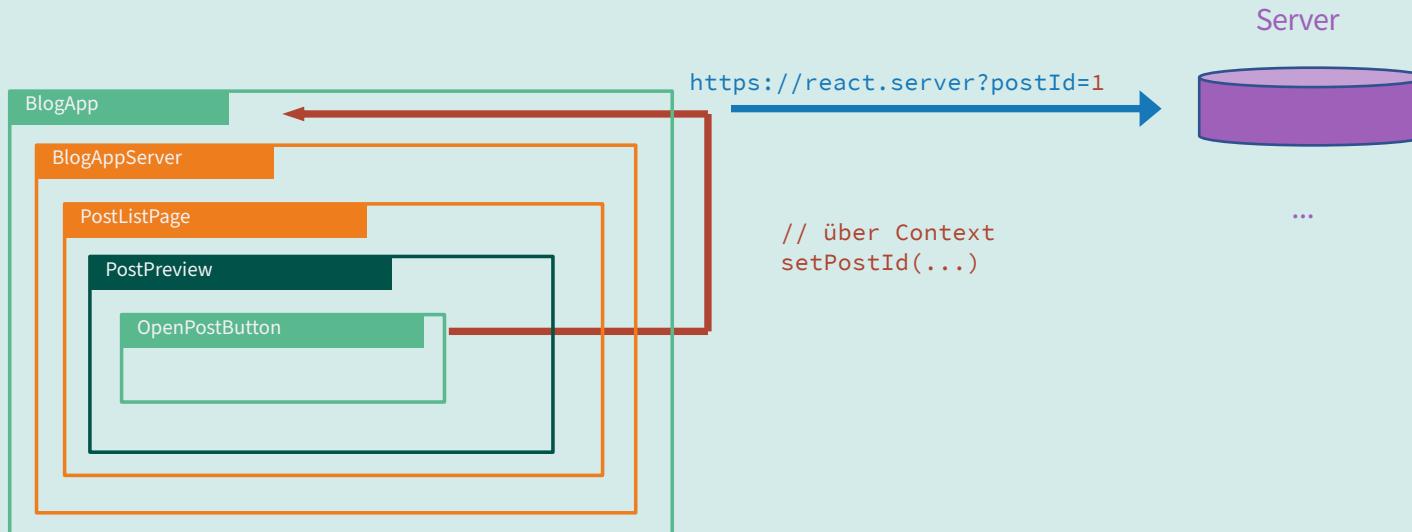
Wenn das eine normale Client-App wäre... ...ist es aber nicht!



SERVER COMPONENTS

Wenn das eine normale Client-App wäre... ...ist es aber nicht!

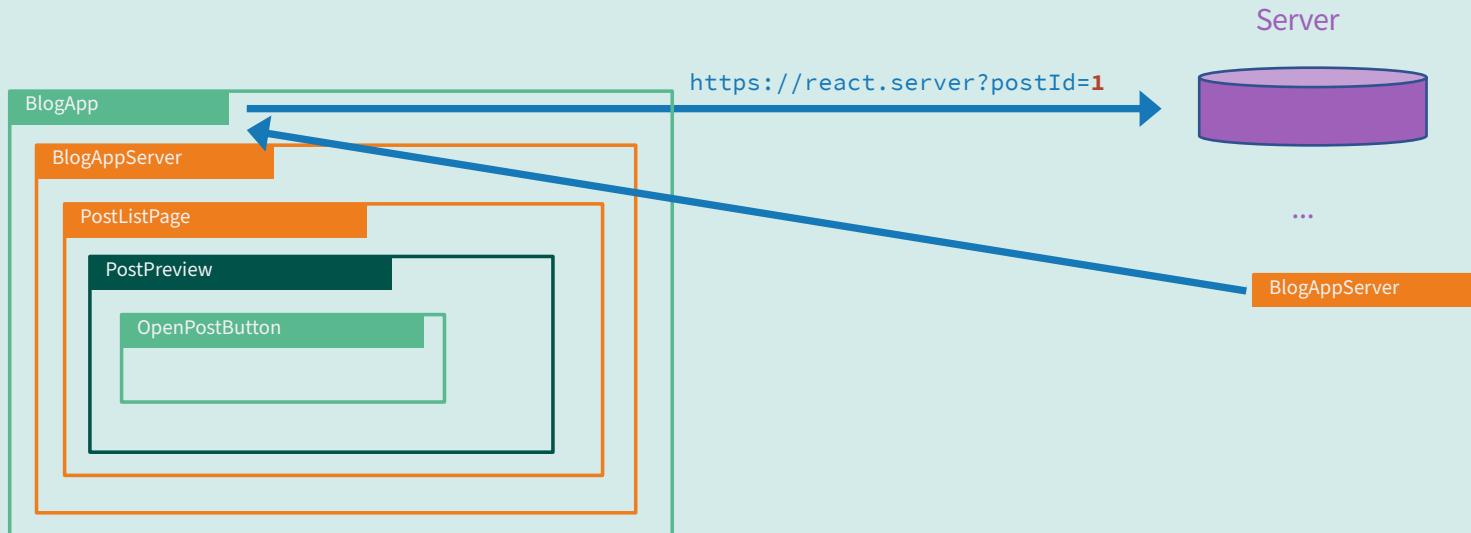
Kommunikation zurück nach oben



SERVER COMPONENTS

Wenn das eine normale Client-App wäre... ...ist es aber nicht!

Kommunikation zurück nach oben



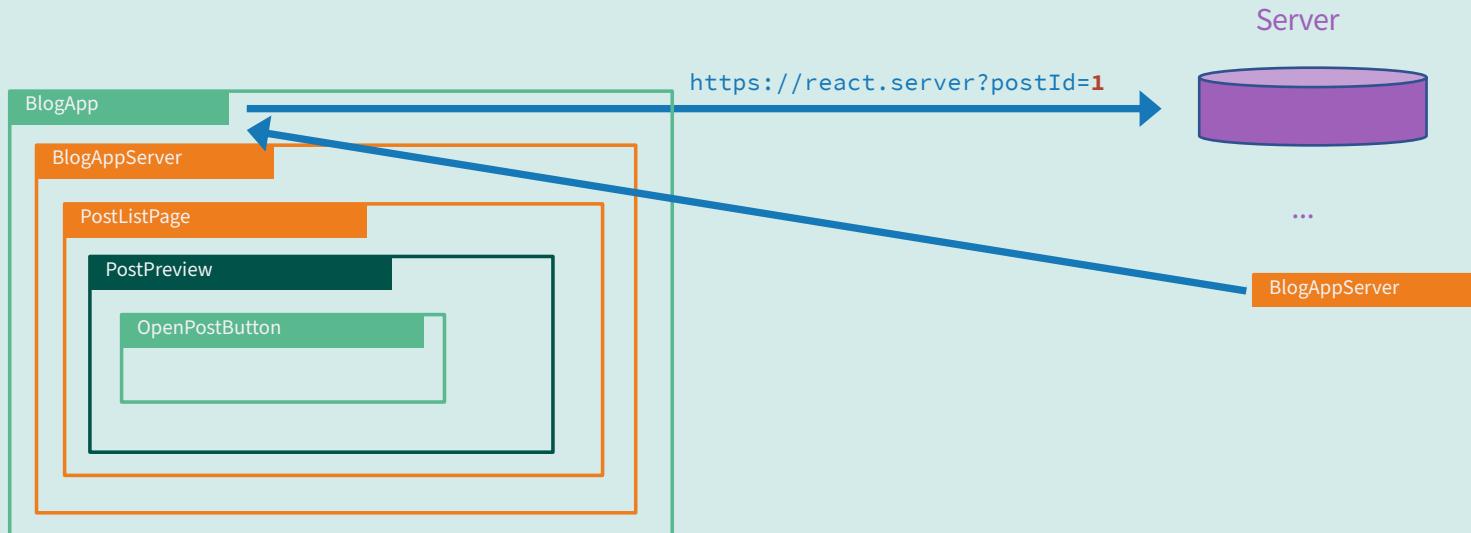
Programmfluss "fast" wie in normalen React-Anwendung,
"nur" mit Server-Aufruf dazwischen 😊

- Uni-directional data flow

SERVER COMPONENTS

Wenn das eine normale Client-App wäre... ...ist es aber nicht!

Kommunikation zurück nach oben



Programmfluss "fast" wie in normalen React-Anwendung,
"nur" mit Server-Aufruf dazwischen 😊

- Uni-directional data flow
- State bleibt nach Server Roundtrip erhalten

SERVER COMPONENTS

Beispiel: State bleibt erhalten

The screenshot shows a web browser window displaying a blog application. The main content area is titled "Blog Posts". It lists two posts:

- Post 1:** Date: 10.01.2021, Title: "Keep calm and learn React!", Content: "Pommy ipsum air one's dirty linen fork out plum pudding the lakes, naff gallivanting around ridicule knows bugger all about nowt. Jolly full English b...", Action: "Read this Post".
- Post 2:** Date: 17.04.2020, Title: "Increasing React developer experience", Content: "Tweeting a baseball. Sit on human they not getting up ever make it to the carpet before i vomit mmmmmm for cats are cute dismember a mouse and then req...", Action: "Read this Post".

To the right of the posts is a sidebar titled "Tags" with a list of tags: Marzipan, URL, Bootstrap, WebDev, DX, Tutorial, Routing, Best Practice, CSS, JavaScript, Context, State, React, and Redux.

A red box highlights the first post, and a green box highlights the comment editor. A blue arrow points from the "Order by date Asc" button in the header to the text "Button löst Server Request aus, rendert PostList neu". Another blue arrow points from the "CommentEditor (client)" area to the text "Client-Komponente mit (use)State".

Button löst Server Request aus, rendert PostList neu

Client-Komponente mit (use)State



Demo

- PostPreview: CommentEditor hinzufügen
- Kommentar eingeben
- Sortierung ändern

Konsequenzen

- PostList ist nicht als Komponente auf dem Client vorhanden
- Die Posts sind folglich ebenso nicht auf dem Client vorhanden
- Nach dem Hinzufügen eines Kommentars (CommentEditor-Komponente) haben wir keinen State zum Verändern 😢
- Wir brauchen aktualisierte UI vom Server 😱

Konsequenzen

- PostList ist nicht als Komponente auf dem Client vorhanden
- Die Posts sind folglich ebenso nicht auf dem Client vorhanden
- Nach dem Hinzufügen eines Kommentars (CommentEditor-Komponente) haben wir keinen State zum Verändern 😢
- Wir brauchen aktualisierte UI vom Server

SERVER COMPONENTS

Demo: UI aktualisieren

The screenshot shows a web browser window with the title "React Training Blog" and the URL "localhost:4001". The main content area is titled "Blog Posts" and contains two blog posts:

- Post 1:** Date: 10.01.2021, Title: "Keep calm and learn React!", Content: "Pommy ipsum air one's dirty linen fork out plum pudding the lakes, naff gallivanting around ridicule knows bugger all about nowt. Jolly full English b...", Action: "Read this Post".
- Post 2:** Date: 17.04.2020, Title: "Increasing React developer experience", Content: "Tweeting a baseball. Sit on human they not getting up ever make it to the carpet before i vomit mmmmmm for cats are cute dismember a mouse and then req...", Action: "Read this Post".

To the right of the posts is a sidebar titled "Tags" with the following categories:
Marzipan URL
Bootstrap
WebDev DX
Tutorial Routing
Best Practice CSS
JavaScript Context
State React
Redux

At the bottom left, there is a "CommentEditor (client)" section with a text input field containing "Nice article!" and a "Save" button.

Gesendet (HTTP POST) werden Daten, gelesen wird UI



Demo

- Kommentar hinzufügen -> Netzwerk-Tab (JS & XHR)

Aktualisierte UI

- HTTP POST Request sendet UI Code zurück (kein JSON o.ä.)
- Es gibt einen globalen für UI-Fragmente
- Komponenten können den Cache mit neuen Fragmenten aktualisieren
- Alles unstable API deshalb kein Code-Beispiel 😢

Data Fetching

DATEN LADEN

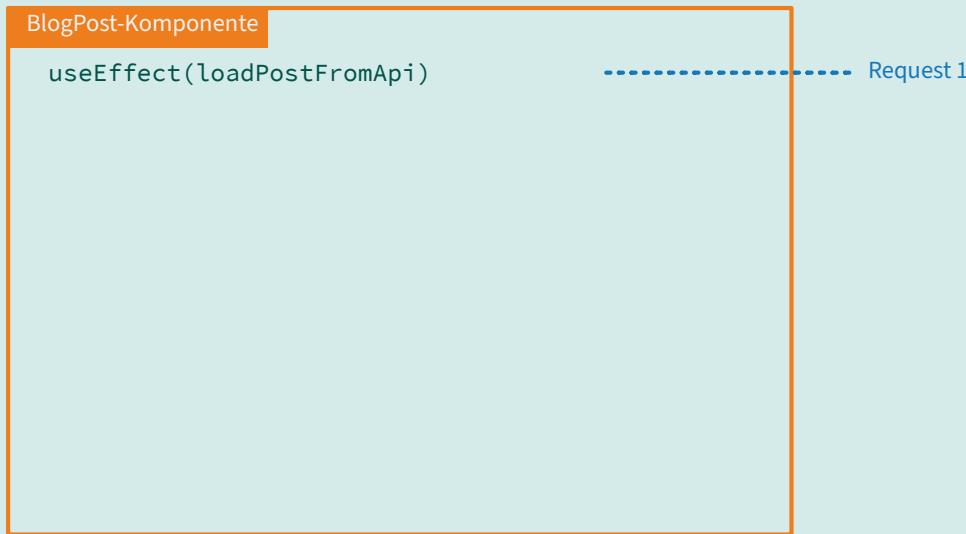
Mögliches Problem: Laden von Daten auf dem Client

- Eine Komponente lädt ihre Daten, Unterkomponenten müssen warten

DATEN LADEN

Laden von Daten auf dem Client

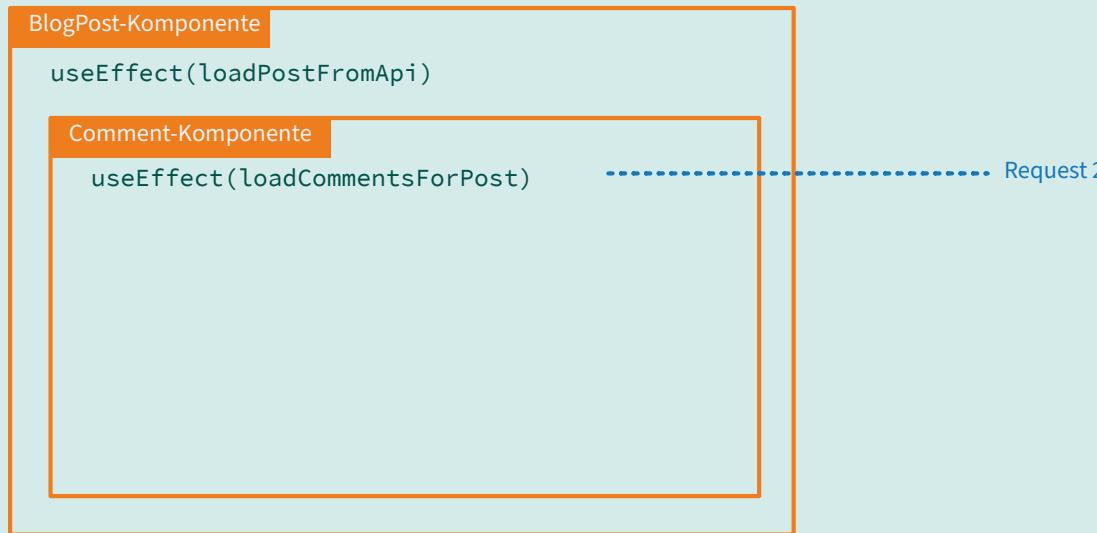
- Eine Komponente lädt ihre Daten, Unterkomponenten müssen warten



DATEN LADEN

Laden von Daten auf dem Client

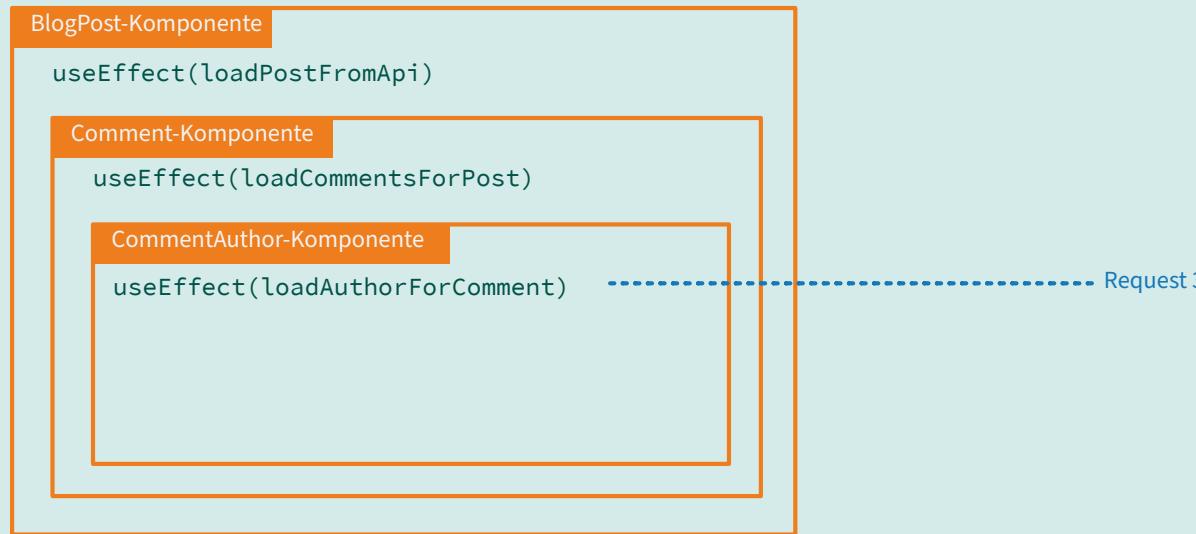
- Eine Komponente lädt ihre Daten, Unterkomponenten müssen warten



DATEN LADEN

Laden von Daten auf dem Client

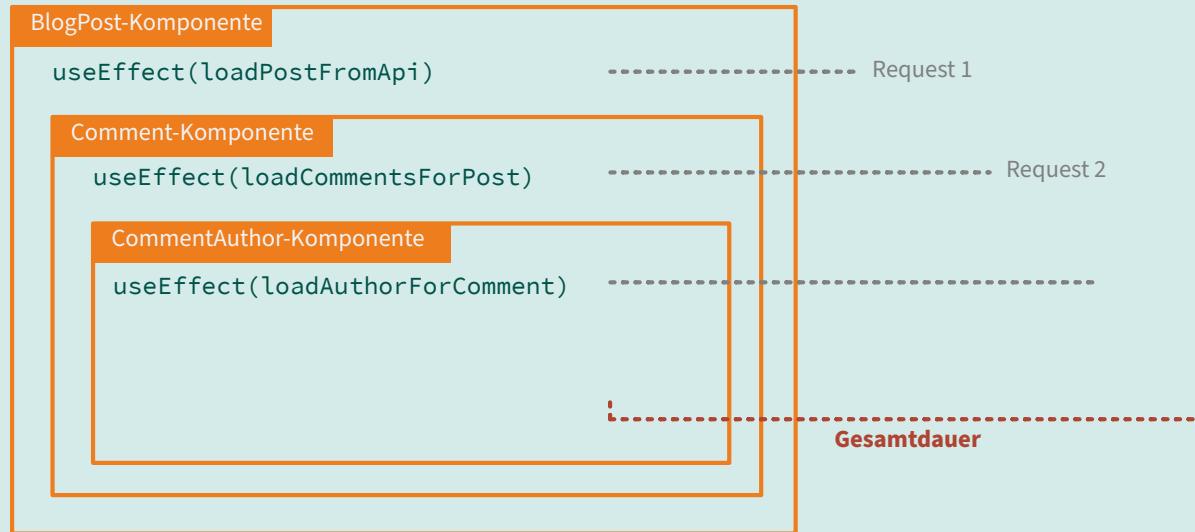
- Eine Komponente lädt ihre Daten, Unterkomponenten müssen warten



DATEN LADEN

Laden von Daten auf dem Client

- Eine Komponente lädt ihre Daten, Unterkomponenten müssen warten



😓 Wasserfall...

SERVER COMPONENTS

Idee

- Komponenten, die Daten laden, können das direkt *auf dem Server* tun
- Kann Latenz sparen und bessere Performance bringen

👉 "No *Client-Server* Waterfalls"

SERVER COMPONENTS

Beispiel: Eine Server Komponente

SUSPENSE

Beispiel: Eine Server Komponente

```
import { db } from "./db.server";

export default function PostComments({ post }) {
  const comments = db.query("select id, comment from comments where post_id = $1", [post.id]);

  return (
    <div className="Container">
      <h1>Comments</h1>
      {comments.rows.map((comment) => (
        <p key={comment.id}>{comment.comment}</p>
      ))}
    </div>
  );
}
```

Datenbank-Zugriff in Komponente 🤯

Beispiel: Eine Server Komponente

```
import { db } from "./db.server";

export default function PostComments({ post }) {
  const comments = db.query("select id, comment from comments where post_id = $1", [post.id]);

  return (
    <div className="Container">
      <h1>Comments</h1>
      {comments.rows.map((comment) => (
        <p key={comment.id}>{comment.comment}</p>
      ))}
    </div>
  );
}
```

- Server Komponenten können direkt DB-Queries ausführen, auf das Filesystem zugreifen etc.
 - (Alles was "echte" Backend-Services auch können)
- Client Komponenten können hier zum Beispiel fetch-Requests ausführen
- *Was machen wir, bis die Daten vorhanden sind, während der Query läuft?*

SUSPENSE

Suspense: React kann das Rendern von Komponenten unterbrechen, während (asynchron) Daten geladen werden

- Funktioniert aktuell für **Code Splitting** (Client)
 - Code Splitting in Server-Komponenten eingebaut

SUSPENSE

Suspense: React kann das Rendern von Komponenten unterbrechen, während (asynchron) Daten geladen werden

- Funktioniert aktuell für **Code Splitting** (Client)
 - Code Splitting in Server-Komponenten eingebaut
- **In der Zukunft** auch zum **Laden von beliebigen Daten (Client und Server)**
 - "That will likely come after the 18.0 release, but we're hoping that to have something during the next 18.x minor releases." (<https://github.com/reactwg/react-18/discussions/47#discussioncomment-847004>)

Hintergrund: Suspense for Data Fetching

- Eine Komponente kann auf "etwas" warten
- React weiß, dass die Komponente auf etwas wartet
- Solange gewartet wird, wird eine Fallback-Komponente gerendert
- Die Fallback-Komponente wird oberhalb mit Suspense festgelegt
 - Wie ein try-catch-Handler für ausstehende Daten

SERVER COMPONENTS

Beispiel: Daten laden auf dem Server

```
import db from "./blog-db";  
  
function PostList() {  
  const posts = db.readPosts(); -----  
  
  return ...; // render Posts  
}  
  
function PostListPage() {  
  return <Suspense fallback={<LoadingIndicator />}>  
    <PostList />  
  </Suspense>;  
}
```

"Suspense for Data Loading"

- Zugriff auf "etwas", das Daten lädt (Datenbank + FS in Server-Komponenten möglich, fetch z.B. würde auch im Client gehen)
- Aufruf blockiert bis Daten da sind

SERVER COMPONENTS

Beispiel: Daten laden auf dem Server

```
import db from "./blog-db";

function PostList() {
  const posts = db.readPosts();

  return ...; // render Posts
}

function PostListPage() {
  return <Suspense fallback={<LoadingIndicator />}>
    <PostList />
  </Suspense>;
}
```

Suspense-Komponente

- "Sollbruchstelle", wenn unterhalb in der Anwendung auf "etwas" gewartet wird, wird fallback angezeigt
- Eine Art try-cache für ausstehende Daten
- Wird es wohl so auch auf dem Client geben

Suspense for Data Fetching

- Es gibt Wrapper, die bekannte APIs (z.B. Postgres, NodeJS fs) für Suspense zur Verfügung stellen
- Über diese Wrapper weiß React, dass eine Komponente noch auf Daten wartet
- Solange kann dann die Fallback-Komponente dargestellt werden
- Für Client-seitige Ressourcen gilt das analog (Wrapper um fetch)

- Die Wrapper APIs können später wohl von der Community implementiert und zur Verfügung gestellt werden

SERVER COMPONENTS

Demo: Suspense for Data Fetching



- Delay für PostList und TagCloud aktivieren (delay.server.js)
- Daten bleiben gecached (Home => Post => Home)
- Suspense in PostListPage verschieben
- Delay für Post aktivieren
- Post aufrufen

Transitions

- Transitions erlauben es, eine neue Komponente im Hintergrund zu rendern
- Die alte bleibt solange sichtbar, bis die neue fertig gerendert ist
- Neuer Hook in React 18: useTransition

```
function OpenPostButton({ post, openPost }) {
  const [isPending, startTransition] = useTransition();

  return (
    <button onClick={() => {
      startTransition(() => openPost(post.id));
    }}
    >
      { isPending ? <LoadingIndicator secondary /> : label }
    </button>
  );
}
```

SERVER COMPONENTS

Demo: Transitions



Demo

- Delay für Post aktivieren
- Post aufrufen
- OpenPostButton zeigen (evtl.)

Abgrenzung

Serverseitiges

Rendern

Abgrenzung: Serverseitiges Rendern (SSR)

1. Bei SSR wird die Anwendung auf dem Server ausgeführt

Abgrenzung: Serverseitiges Rendern (SSR)

1. Bei SSR wird die Anwendung auf dem Server ausgeführt
2. Der Server schickt **fertiges HTML** zum Client
 - Gut: Client braucht HTML nur anzuzeigen (schnell!)
 - Gut: Suchmaschinen können HTML indizieren

Abgrenzung: Serverseitiges Rendern (SSR)

1. Bei SSR wird die Anwendung auf dem Server ausgeführt
2. Der Server schickt **fertiges HTML** zum Client
 - Gut: Client braucht HTML nur anzuzeigen (schnell!)
 - Gut: Suchmaschinen können HTML indizieren
3. Ebenfalls wird der **Anwendungscode** zum Client geschickt
 - Wenn vom Browser geladen, ist die Anwendung interaktiv
 - Danach in der Regel keine Server Round-trips mehr

Serverseitiges Rendern - Zusammenfassung

-  Schnelle erste Darstellung
-  Interaktion: kein Gewinn! JS-Code muss weiterhin komplett zum Client
-  Kompletter Anwendungscode muss auf den Client (Bandbreite! Performance!)
-  Anwendungscode muss auf Client *und* Server funktionieren

Serverseitiges Rendern - Zusammenfassung

- Denkbar: SSR und Server Components kombinieren:
 - Server schickt fertiges HTML 
 - Server schickt JS-Code... 
 - ...aber nur den auf dem Client benötigten 
- **Suspense auf dem Server kann SSR vereinfachen**
 - Server weiß nun, auf welche Komponenten er noch warten muss

Fazit

Server Components

Aktueller Stand: Experimentell...

- Es gibt eine offizielle Beispiel App, die aus instabilen APIs besteht
- Unklar, wie Server aussehen werden
- Unklar, wie Serverkommunikation aussehen wird (Protokoll und APIs)
- Unklar, wie Tooling aussieht (Build, React DevTools, TypeScript ...)
- Weitere große Baustellen offen (Integration Libraries)

"While we're busy with React 18, Server Components is on hold. This means that it's still in very early research phase."

(<https://github.com/reactwg/react-18/discussions/98>)

Ausblick

- Wird wohl als erstes für Frameworks wie NextJS oder Gatsby zur Verfügung gestellt
 - Diese haben bereits eine Server-Infrastruktur
 - Unklar wie Server dann für uns aussehen werden
 - Pro-Tipp: nutzt die Zeit, um Eure Ops davon zu überzeugen, NodeJS-Server zu betreiben 😈
- Für Apps mit viel statischem Content
- Integration dann auch mit SSR

Einschätzung

- Erfahrungen mit anderen Technologien, die "Misch-Betrieb" erlauben, sind eher durchwachsen
 - Architektur gerät schnell aus dem Ruder ("was läuft wo?")
 - Kommunikation mit dem Server gewöhnungsbedürftig (Daten hin, UI zurück)
 - Properties müssen immer über Server gehen
- Das ist auf jeden Fall nichts für **jede** Anwendung
 - eher im Gegenteil: Spezial-Fall?
- Man muss JavaScript-Ausführung auf dem Server zulassen

SERVER COMPONENTS

"Getting Started" – Links

- Blog Post
<https://reactjs.org/blog/2020/12/21/data-fetching-with-react-server-components.html>
- Data Fetching with React Server Components (Intro Video)
<https://www.youtube.com/watch?v=TQQPAU21ZUw>
- RFC mit FAQ und Diskussionen
<https://github.com/reactjs/rfcs/pull/188>
- Meine Beispiel-Anwendung
<https://github.com/nilshartmann/server-components-blogexample>

NILS HARTMANN
<https://nilshartmann.net>



vielen Dank!

Slides: <https://react.schule/wjax2021-server-components>

Fragen & Kontakt: nils@nilshartmann.net

Twitter: [@nilshartmann](https://twitter.com/nilshartmann)