

NILS HARTMANN

React

Server Components

Slides: <https://react.schule/ejs2021-server-components>

NILS HARTMANN

nils@nilshartmann.net

Freiberuflicher Entwickler, Architekt, Trainer aus Hamburg

Java

JavaScript, TypeScript

React

GraphQL

Trainings & Workshops



<https://reactbuch.de>

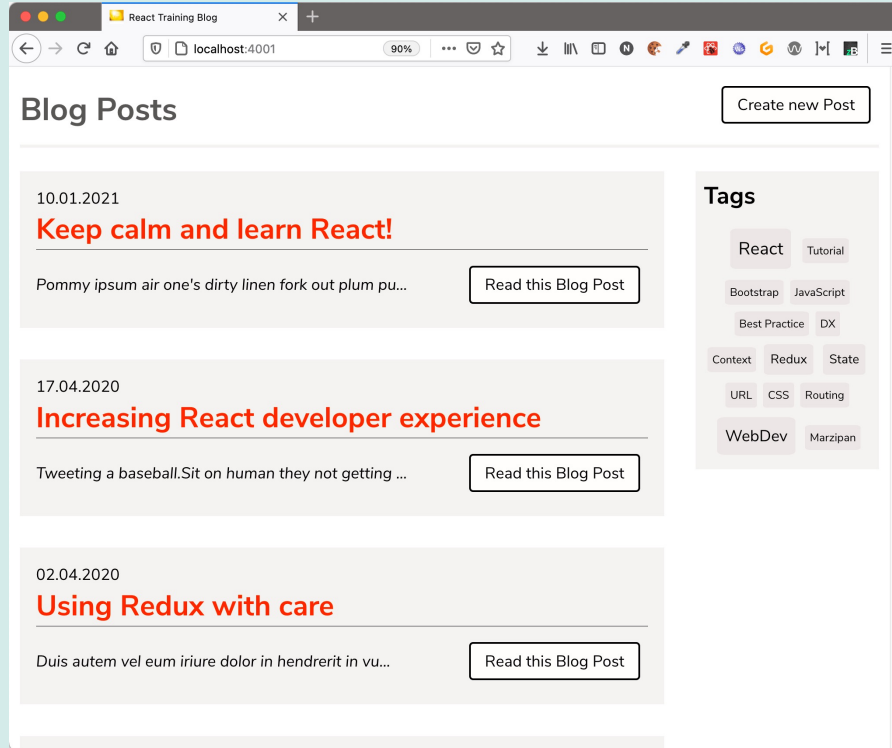
[HTTPS://NILSHARTMANN.NET](https://nilshartmann.net)

"-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



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

EIN BEISPIEL...

Was macht die Beispiel-Anwendung aus?

- Viel statischer Content
- Rendern des statischen Contents benötigt 3rd-Party Libs
 - viel JavaScript-Code (Bandbreite!)
- Minimale Benutzer-Interaktionen (PostEditor)

Was macht die Beispiel-Anwendung aus?

- Viel statischer Content
- Rendern des statischen Contents benötigt 3rd-Party Libs
 - viel JavaScript-Code (Bandbreite!)
- Minimale Benutzer-Interaktionen (PostEditor)

👉 Für Besucher des Blogs sollen die Artikel schnell zur Verfügung stehen!

Zero-Bundle-Size

Server Components

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"

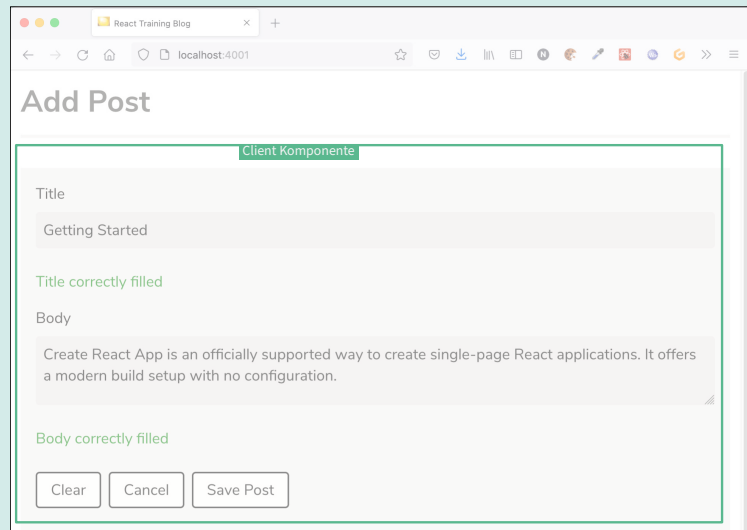
Drei Arten von Komponenten

SERVER COMPONENTS

Drei Arten von Komponenten

- **Client-Komponenten**

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



Drei Arten von Komponenten

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

Drei Arten von Komponenten

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

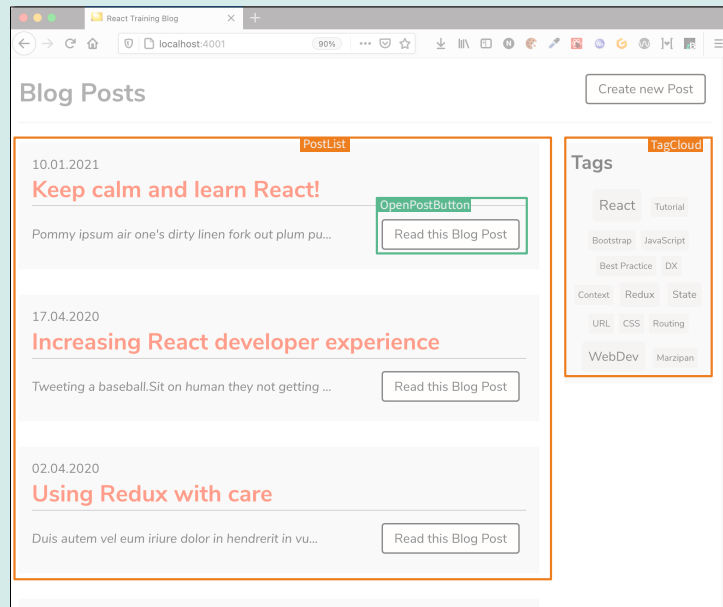
- werden *nur* auf dem Server ausgeführt
- liefern UI (!) zum React-Client zurück
- API: "normale" React-Komponenten (JS, JSX, ...)

- Restriktionen: kein useState, useEffect, Browser APIs
- aber: können Server Umgebung und Ressourcen nutzen (!)
 - Datenbanken
 - Filesystem

SERVER COMPONENTS

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 sind nicht auf dem Client vorhanden!**



SERVER COMPONENTS

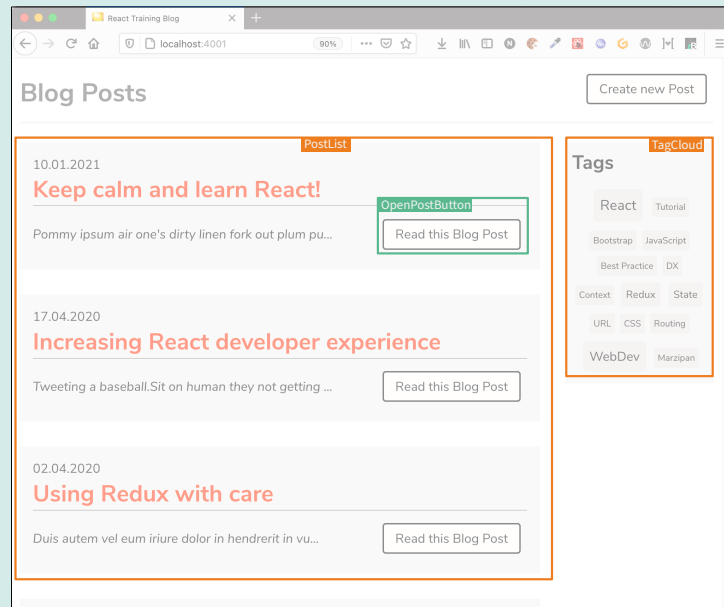
Weiterhin ein Komponenten-Baum

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



Demo

- Server-Komponenten "PostList" und "TagCloud" gibt es als Komponenten, aber nicht auf dem Client (-> React Dev Tools)



Drei Arten von Komponenten

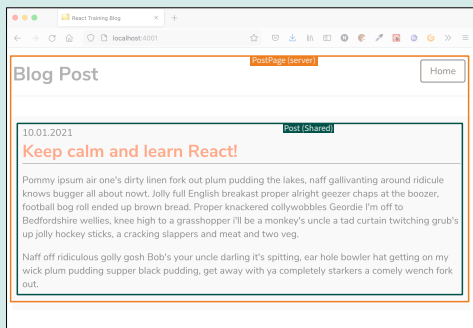
- **Shared Komponenten (Neu!)**

- 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

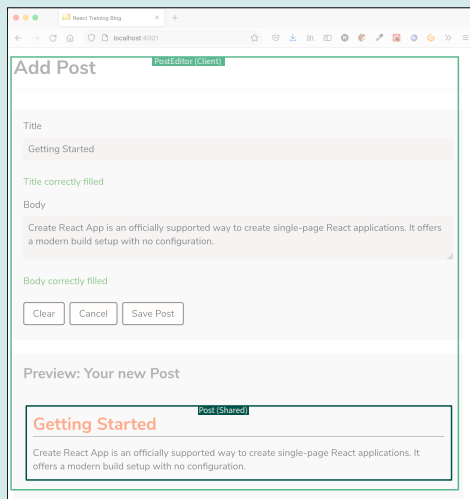
SERVER COMPONENTS

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



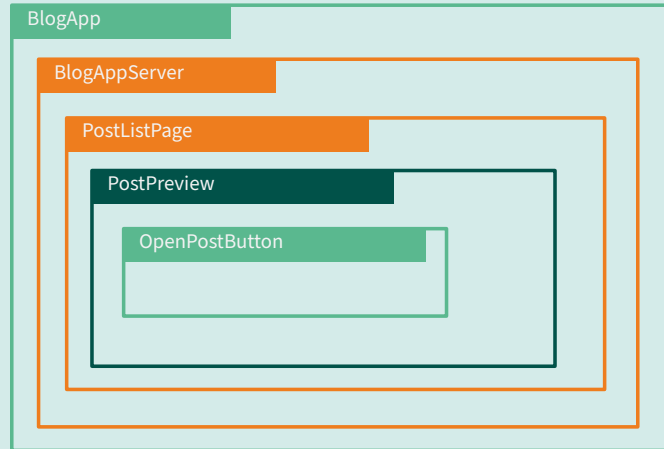
Demo

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

Rendering und Updates

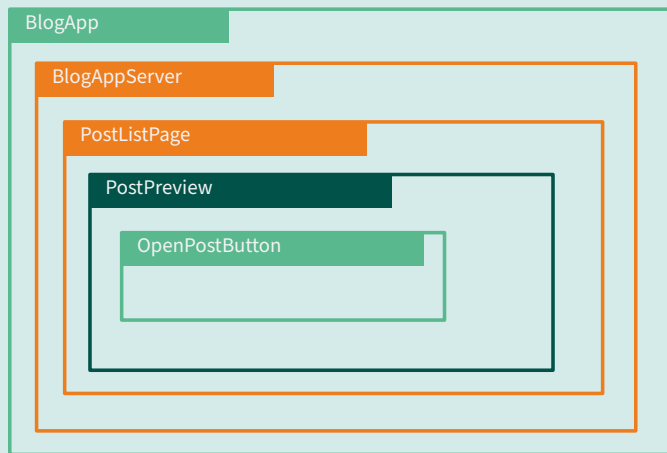
SERVER COMPONENTS

Wenn das eine normale Client-App wäre...



SERVER COMPONENTS

Wenn das eine normale Client-App wäre...



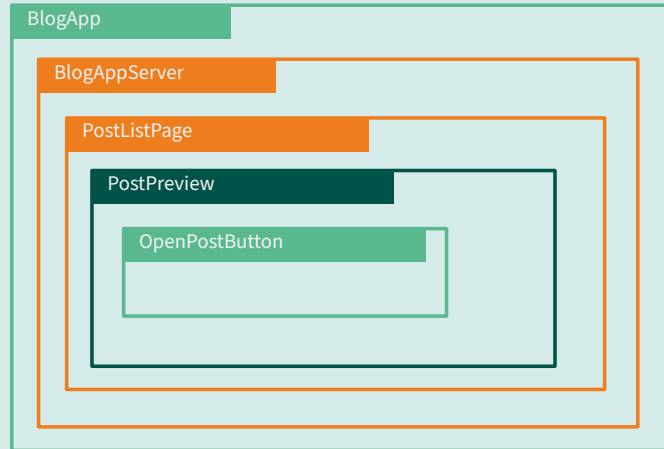
BlogApp würde Liste oder Einzel-Darstellung rendern

// Pseudo Code !!

```
BlogApp() {  
  const [postId] = useState();  
  
  if (postId) {  
    return <PostPage id={postId} />  
  }  
  
  return <PostListPage />  
}
```

SERVER COMPONENTS

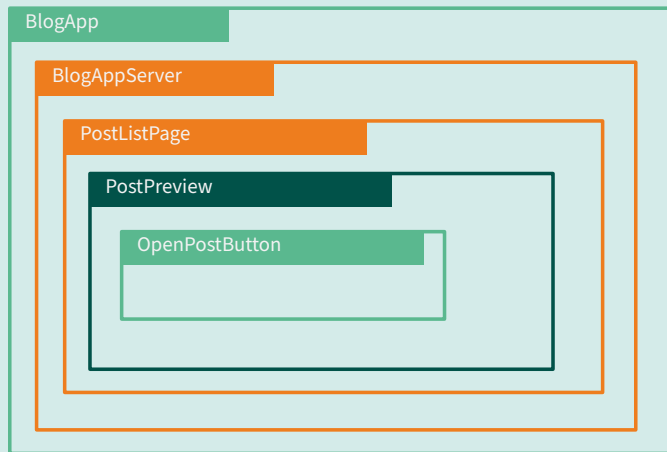
Wenn das eine normale Client-App wäre...



(BlogAppServer würde es nicht geben)

SERVER COMPONENTS

Wenn das eine normale Client-App wäre...

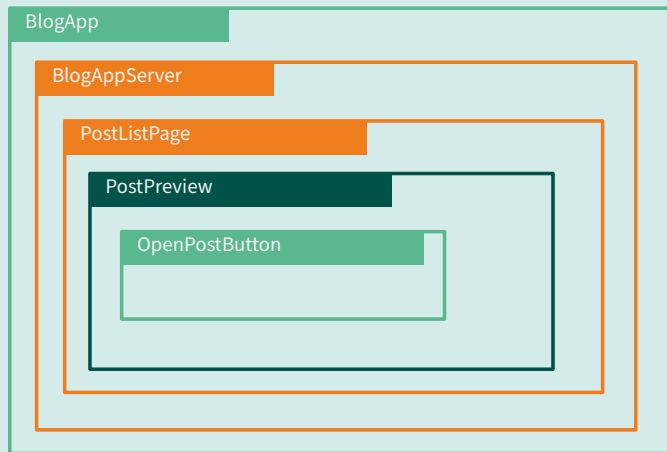


PostListPage würde Daten laden
und Children rendern

```
// Pseudo Code !!  
PostListPage() {  
  useEffect(loadPosts());  
  
  return {posts.map(  
    p =><PostPreview post={p} />  
  )}  
}
```

SERVER COMPONENTS

Wenn das eine normale Client-App wäre...

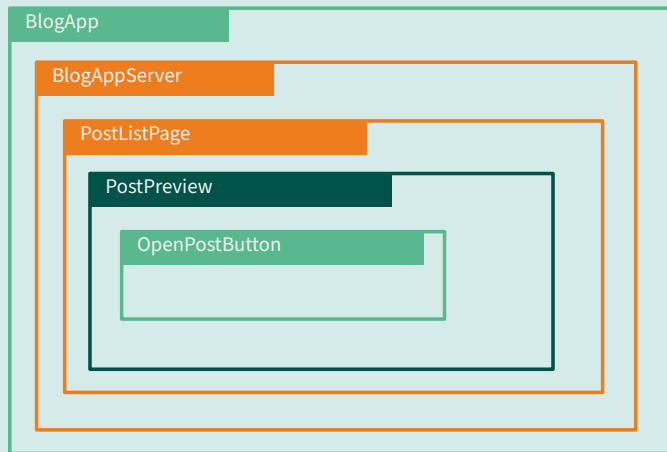


PostPreview würde Post darstellen und Knopf rendern
// Pseudo Code !!

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

SERVER COMPONENTS

Wenn das eine normale Client-App wäre...



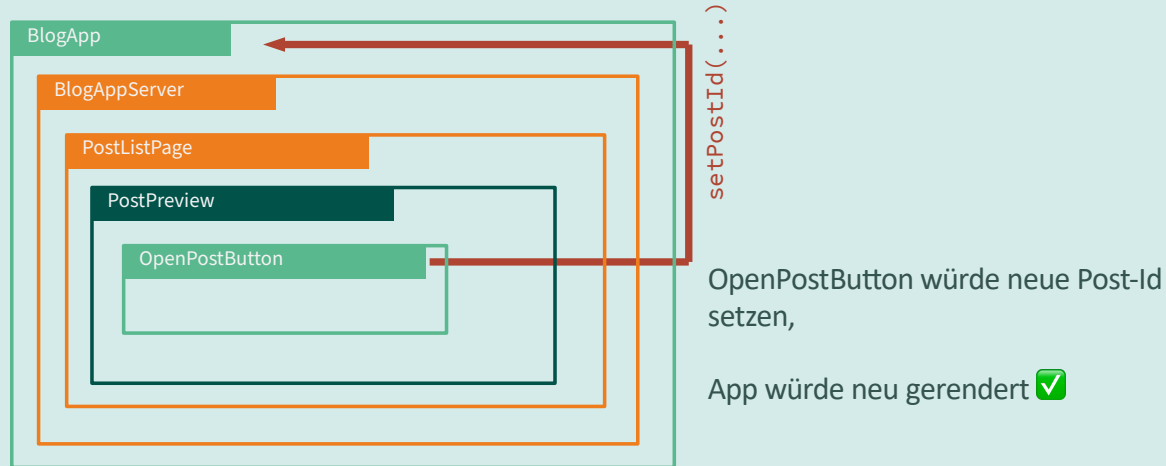
OpenPostButton würde neue Post-Id setzen

// Pseudo Code !!

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

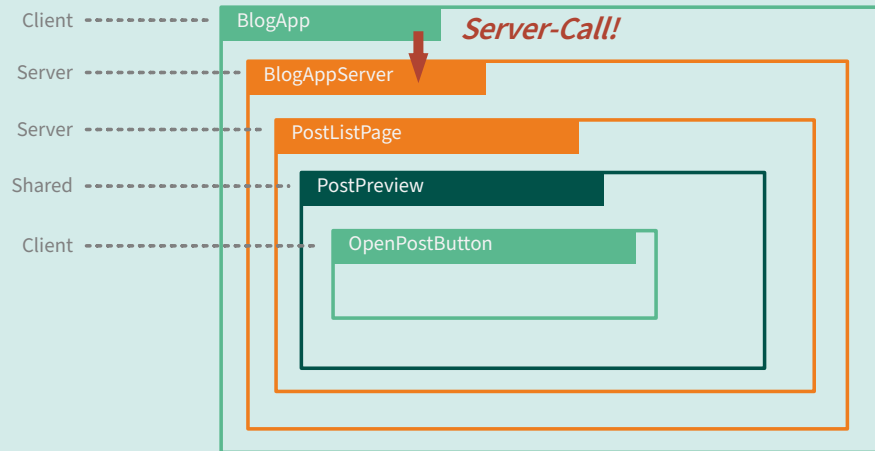
SERVER COMPONENTS

Wenn das eine normale Client-App wäre...



SERVER COMPONENTS

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



BlogApp will **Server**-Komponenten darstellen

SERVER COMPONENTS

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



BlogApp

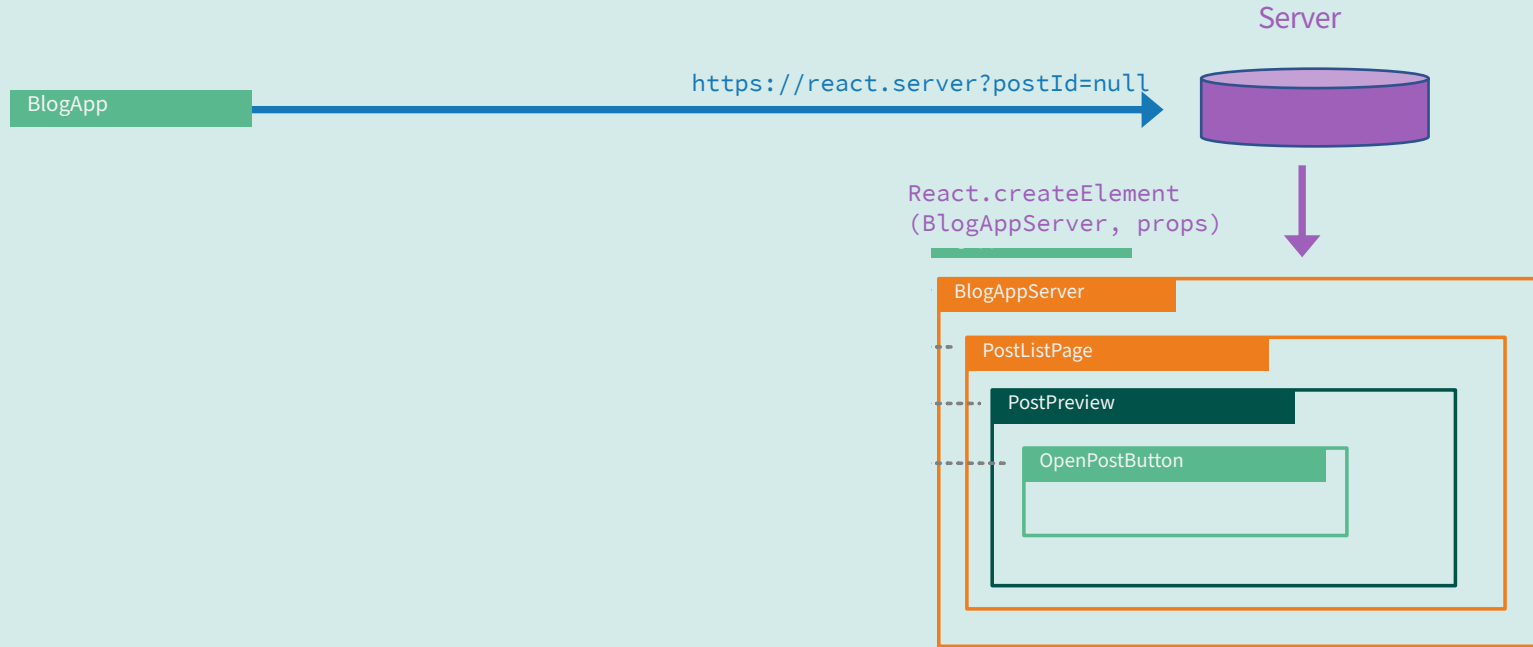
- löst Server Request aus

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



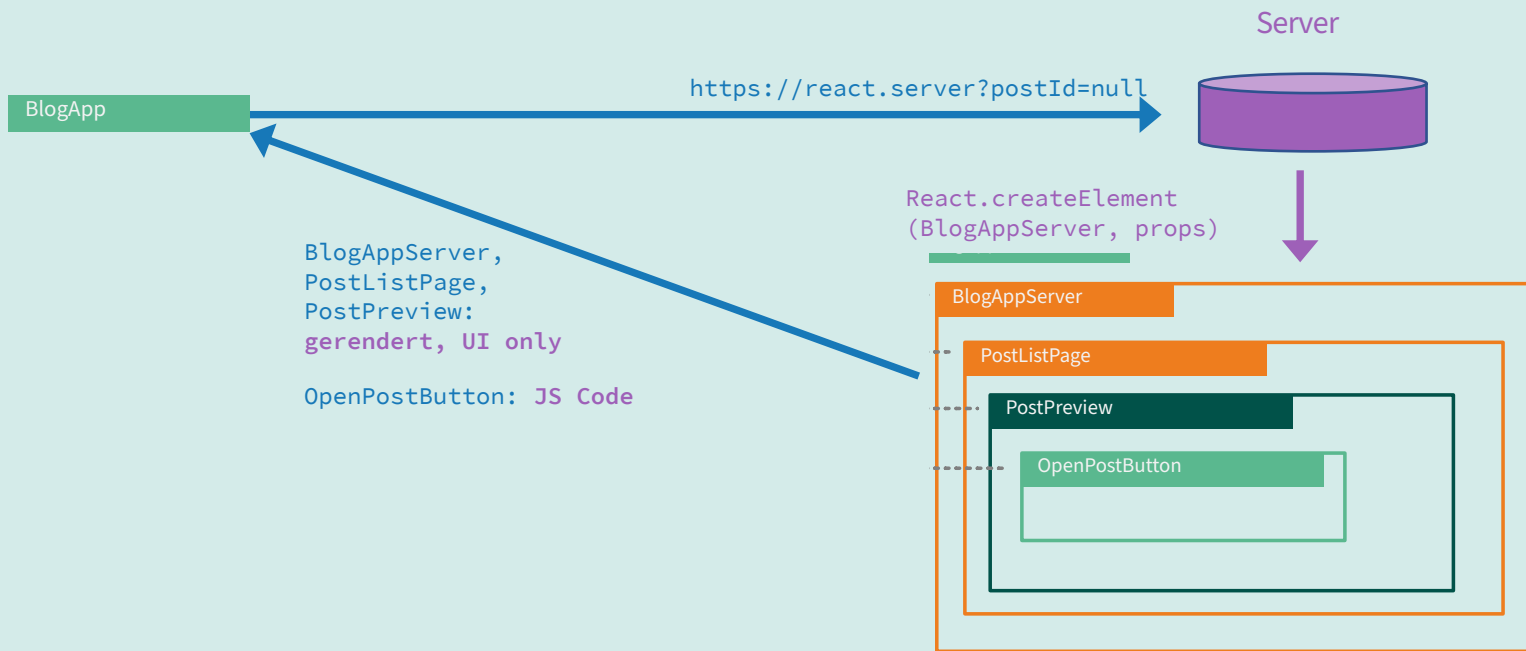
SERVER COMPONENTS

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



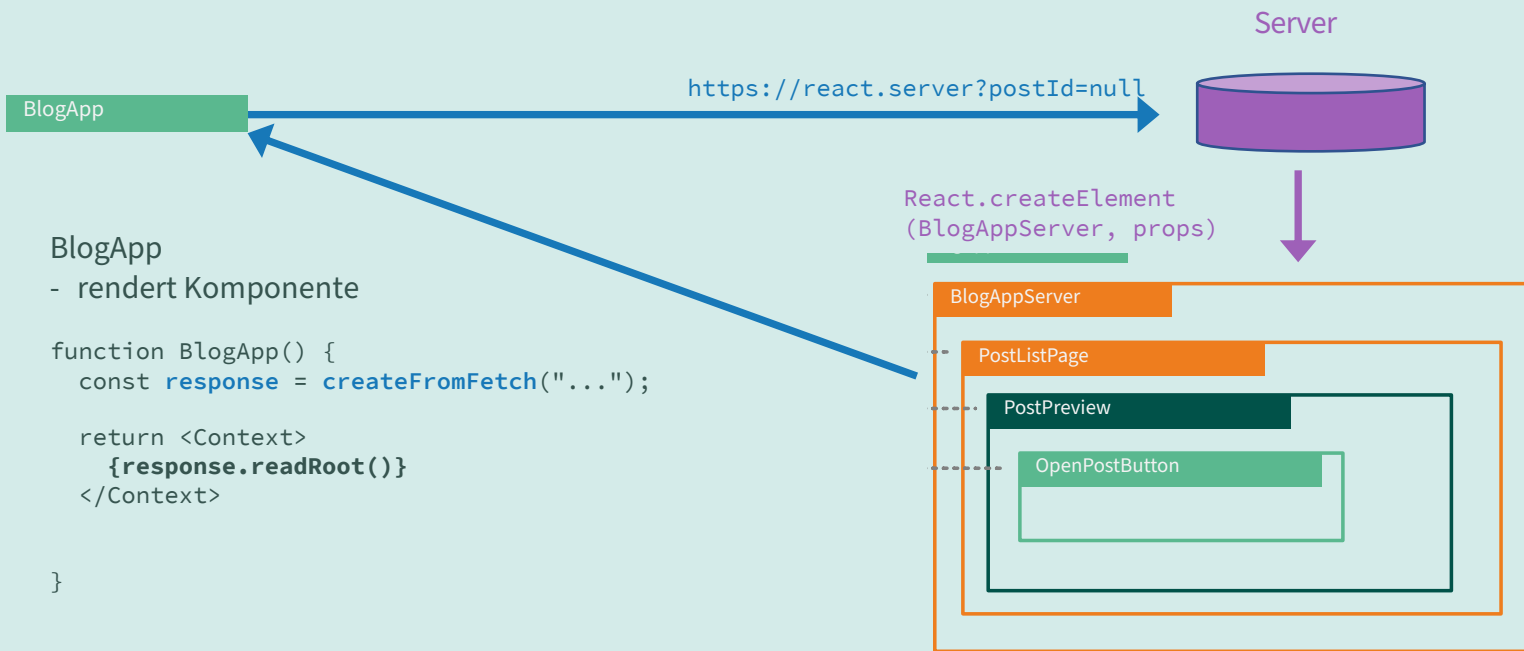
SERVER COMPONENTS

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



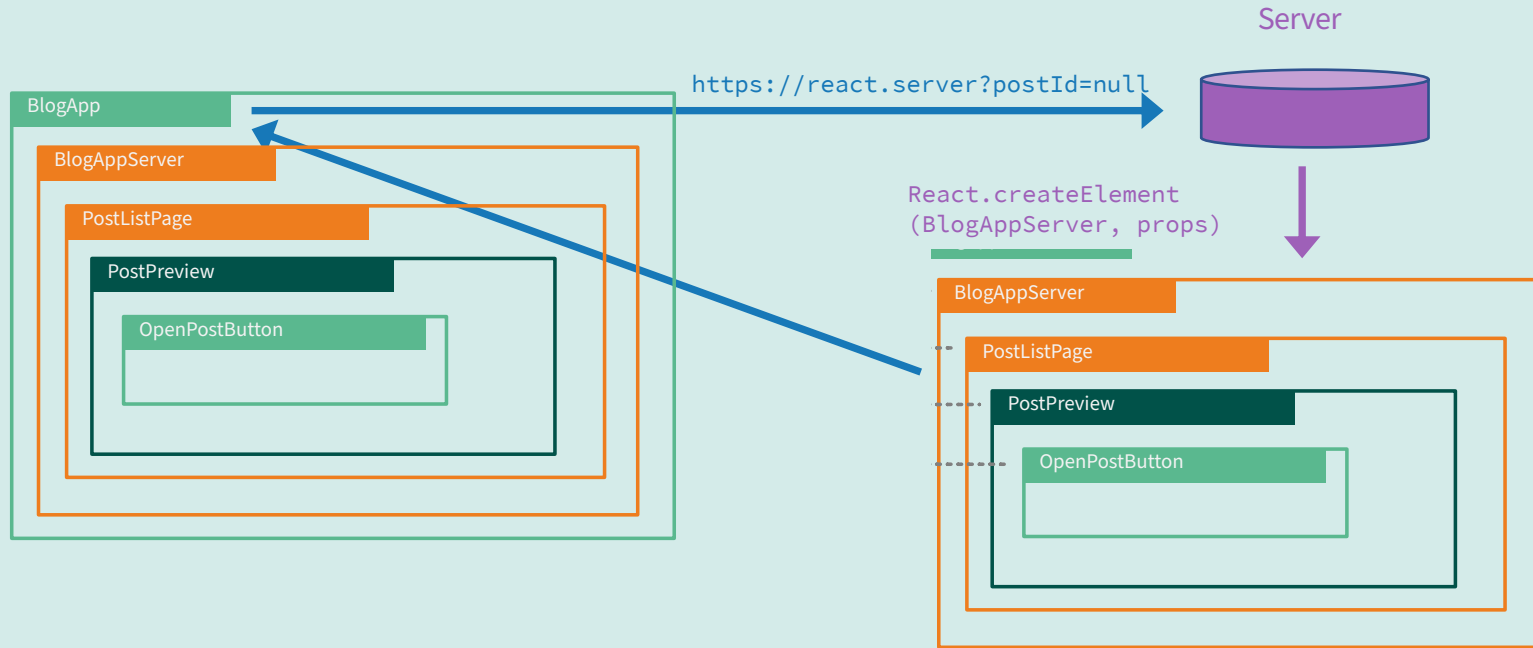
SERVER COMPONENTS

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



SERVER COMPONENTS

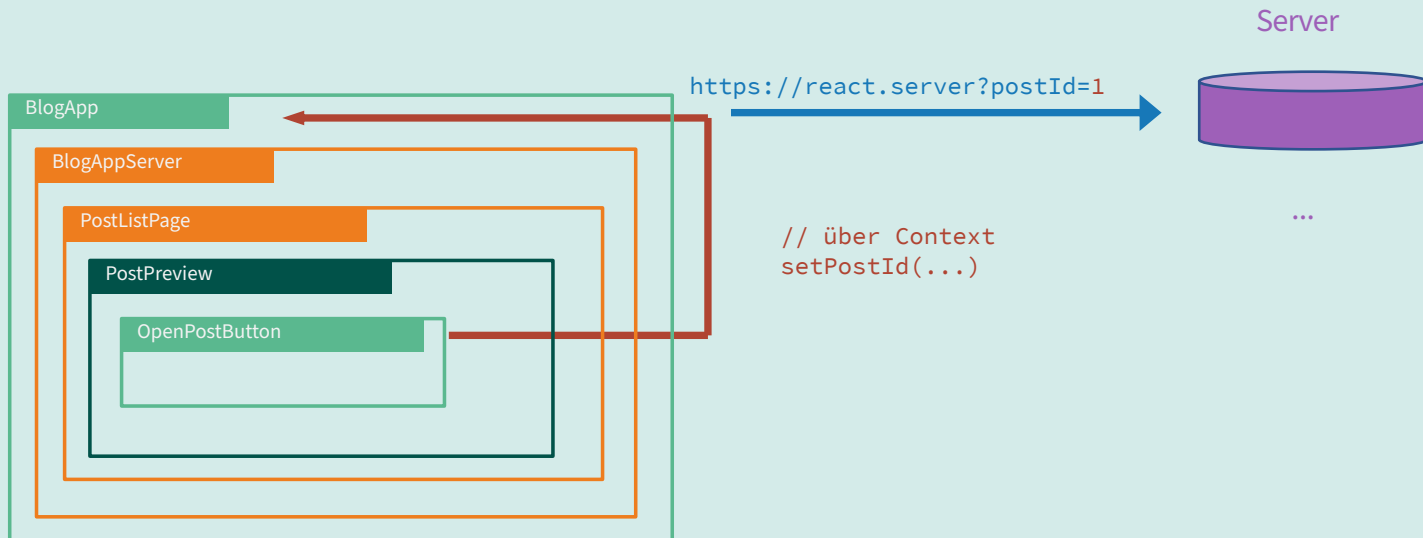
Wenn das eine normale Client-App wäre... ...ist es aber nicht!



SERVER COMPONENTS

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

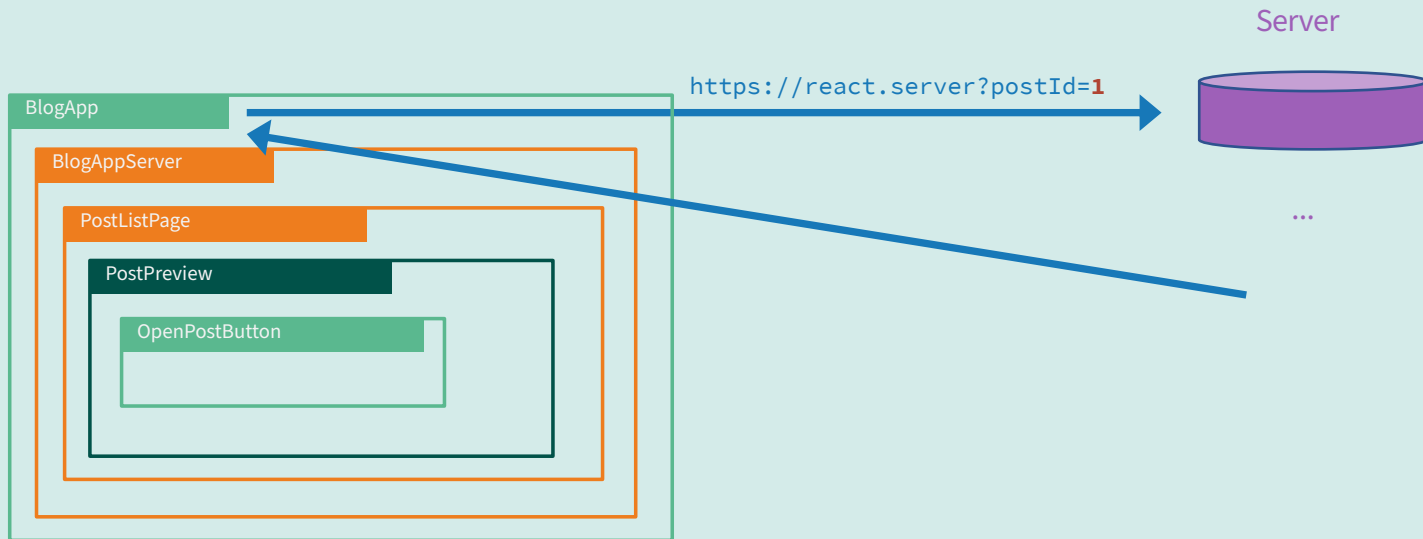
Kommunikation zurück nach oben



SERVER COMPONENTS

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

Kommunikation zurück nach oben



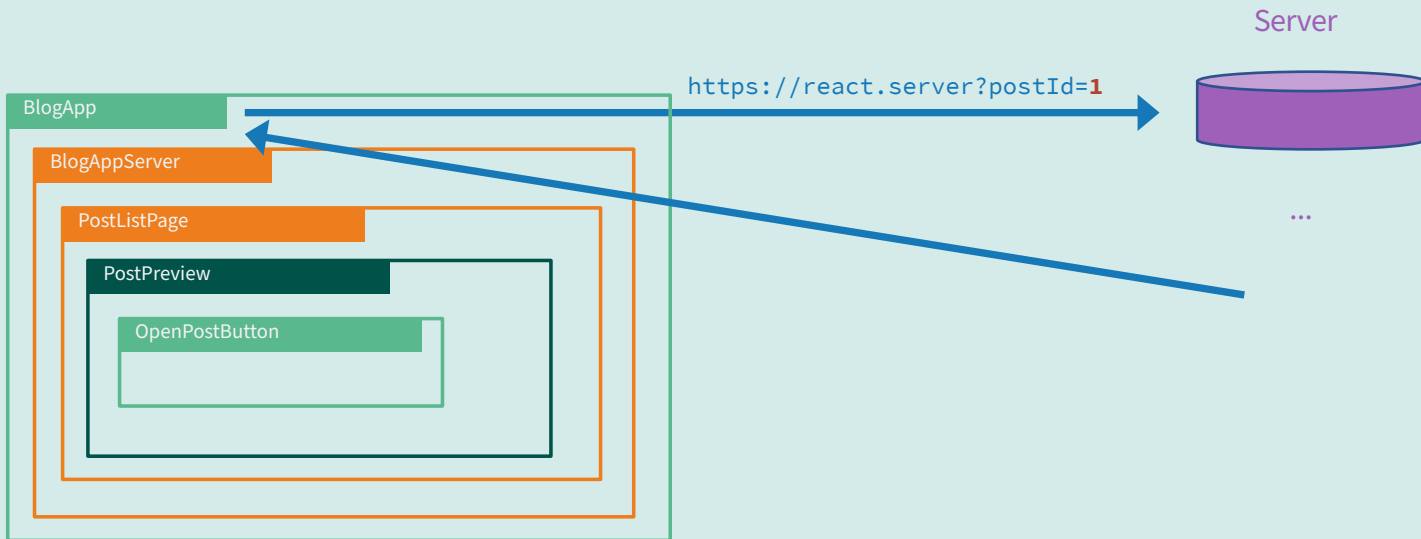
Programmfluss "fast" wie in normalen React-Anwendung,
"nur" mit Server-Aufruf dazwischen 🤖

- Uni-directional data flow

SERVER COMPONENTS

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

Kommunikation zurück nach oben

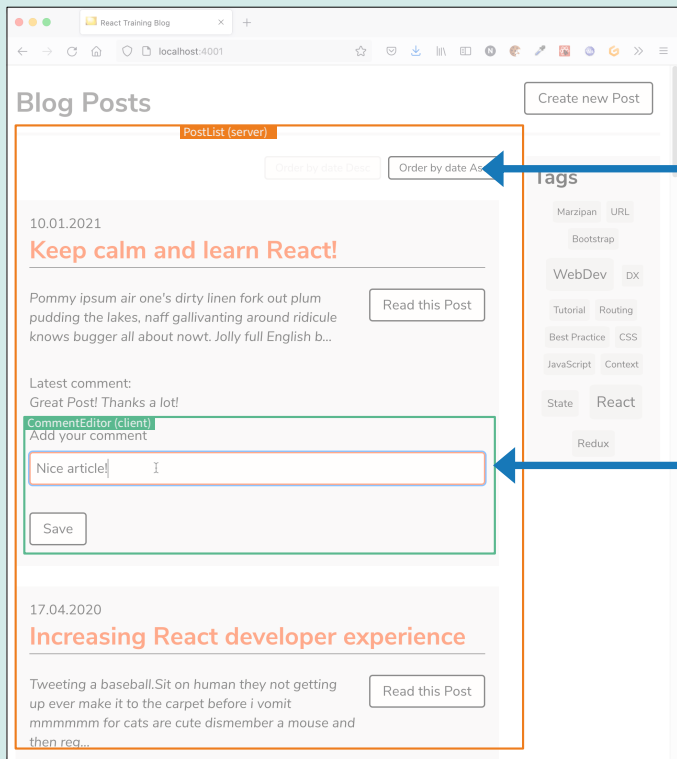


Programmfluss "fast" wie in normalen React-Anwendung,
"nur" mit Server-Aufruf dazwischen 🤖

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

SERVER COMPONENTS

Beispiel: State bleibt erhalten



Button löst Server Request aus, rendert PostList neu

Client-Komponente mit (use)State



Demo

- PostPreview: CommentEditor hinzufügen
- Kommentar eingeben
- Sortierung ändern

Konsequenzen

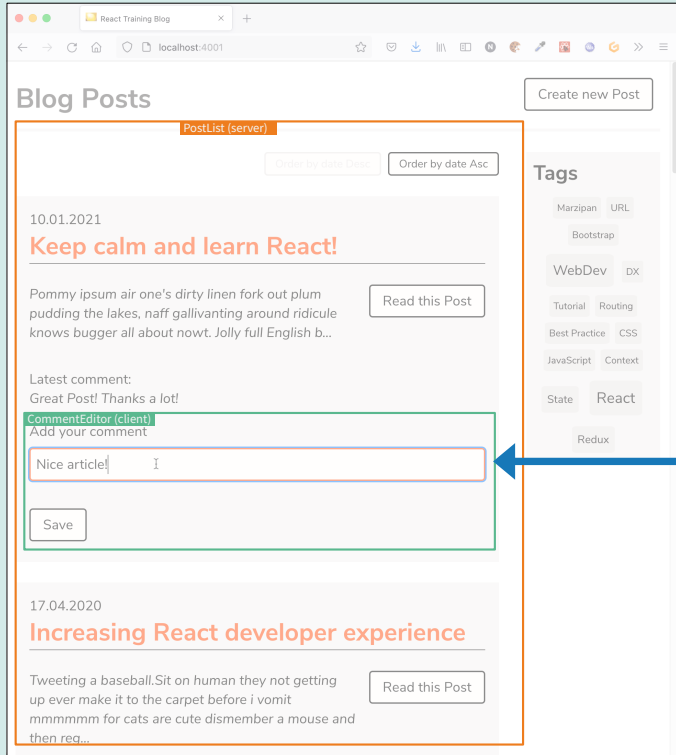
- PostList ist nicht als Komponente auf dem Client vorhanden
- Die Posts sind folglich ebenso nicht auf dem Client vorhanden
- Nach dem Hinzufügen eines Kommentars (CommentEditor-Komponente) haben wir keinen State zum Verändern 😓
- Wir brauchen **aktualisierte UI vom Server**

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**
- Es gibt einen globalen Cache (unstable API!)
- Der Cache hält die UI-Fragmente und kann mit neuen UI Fragmenten (zzt. manuell) aktualisiert werden

SERVER COMPONENTS

Demo: UI aktualisieren



Gesendet (HTTP POST) werden Daten, gelesen wird UI



Demo

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

Data Fetching

DATEN LADEN

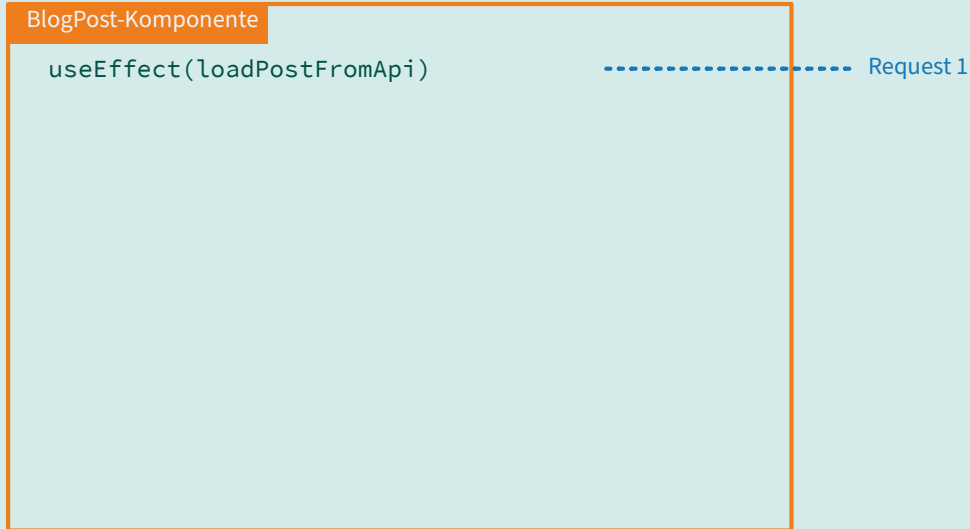
Mögliches Problem: Laden von Daten auf dem Client

- Eine Komponente lädt ihre Daten, Unterkomponenten müssen warten

DATEN LADEN

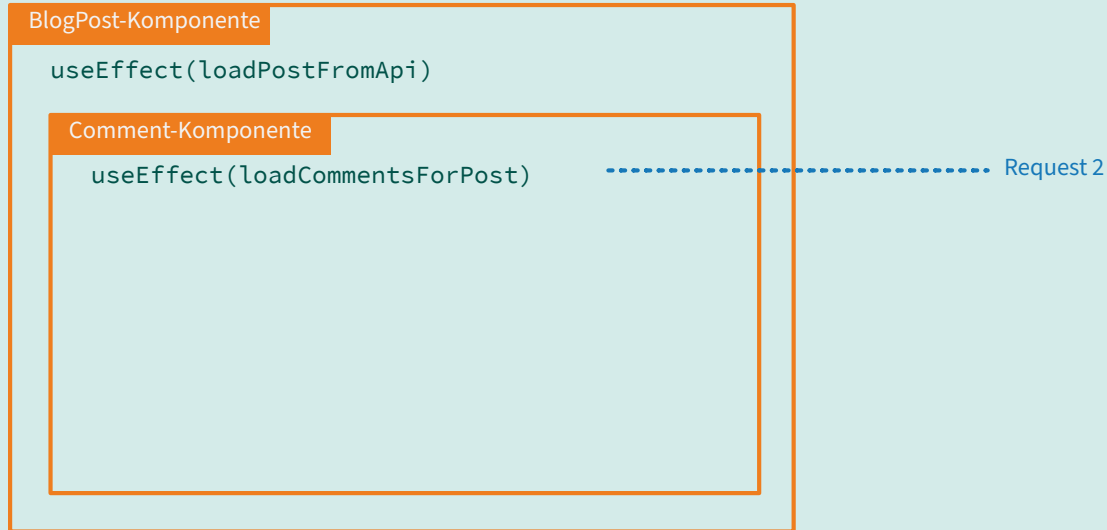
Laden von Daten auf dem Client

- Eine Komponente lädt ihre Daten, Unterkomponenten müssen warten



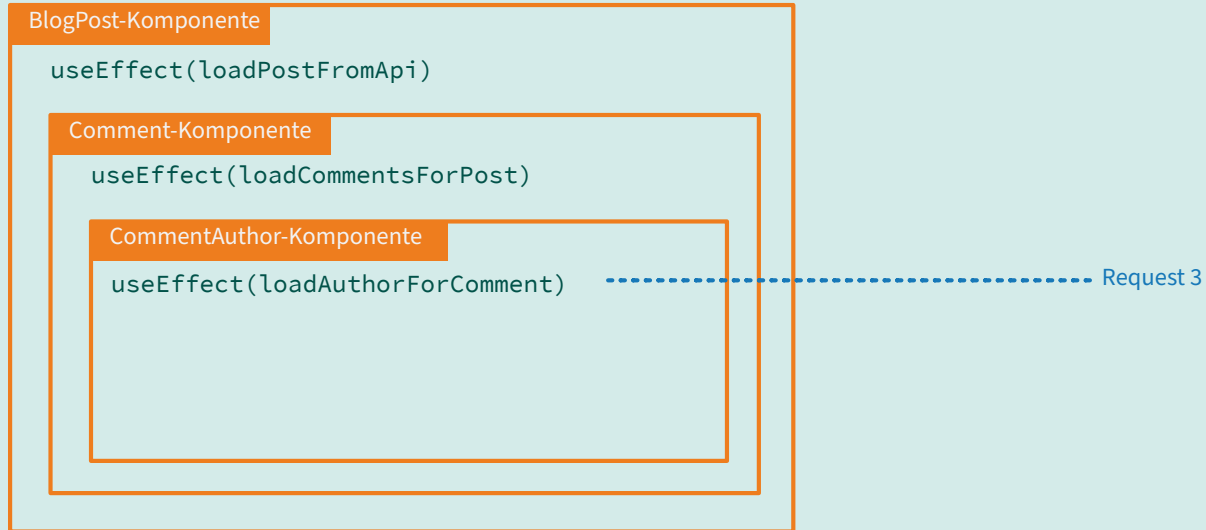
Laden von Daten auf dem Client

- Eine Komponente lädt ihre Daten, Unterkomponenten müssen warten



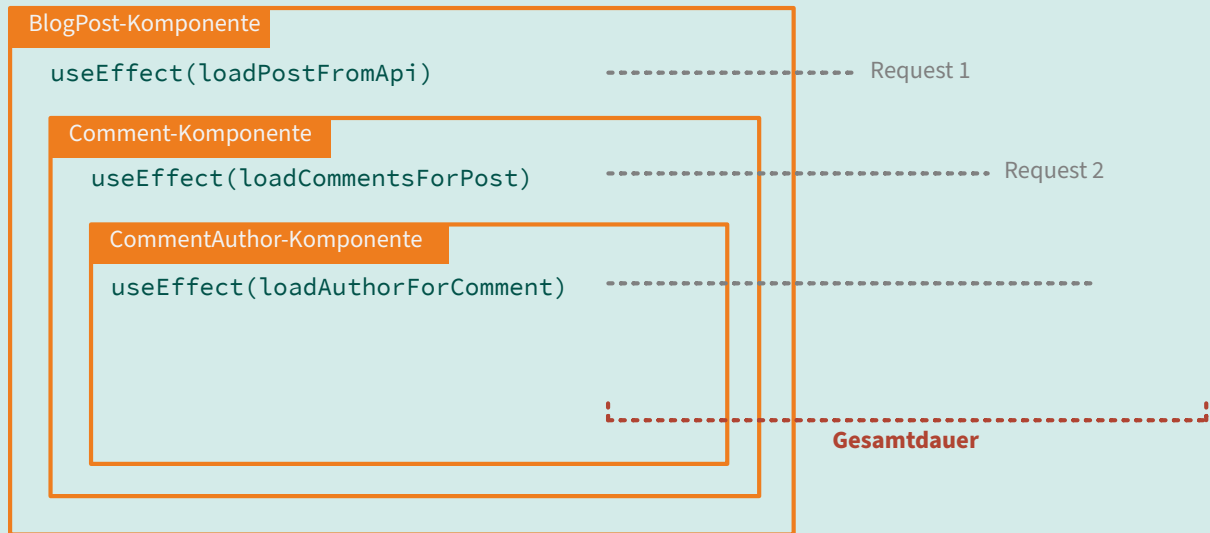
Laden von Daten auf dem Client

- Eine Komponente lädt ihre Daten, Unterkomponenten müssen warten



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"

Beispiel: Eine Server 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>
  );
}
```

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
- **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

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

Zugriff auf Ressourcen im Server

- 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

Zugriff auf Ressourcen im Server



Demo (falls noch Zeit ist)

- 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

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
- 😓 Kein Gewinn, bis Anwendung im Client auch *interaktiv* ist
- 😓 Kompletter Anwendungscode muss auf den Client (Bandbreite! Performance!)
- 😐 Anwendungscode muss auf Client *und* Server funktionieren

Fazit

Server Components

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
- Für Apps mit viel statischem Content
- Integration dann auch mit SSR

Einschätzung

- Erfahrungen mit anderen Technologien, die "Misch-Betrieb" erlauben, sind eher durchgewachsen
 - 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
- Man muss JavaScript-Ausführung auf dem Server zulassen

"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>

NILS HARTMANN

<https://nilshartmann.net>



Vielen Dank!

Slides: <https://react.schule/ejs2021-server-components>

Fragen & Kontakt: nils@nilshartmann.net

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