

NILS HARTMANN

# React

## Server Components

Slides: <https://react.schule/wdc2021-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**

# "-experimental-

```
"dependencies": {  
  "react": "0.0.0-experimental-3310209d0",  
  "react-dom": "0.0.0-experimental-3310209d0",  
  "react-fetch": "0.0.0-experimental-3310209d0",  
  "react-fs": "0.0.0-experimental-3310209d0",  
  "react-pg": "0.0.0-experimental-3310209d0",  
  "react-server-dom-webpack": "0.0.0-experimental-3310209d0",
```



**CURRENT STATE**

# "unstable"



A screenshot of a code editor's "Find in Files" search results. The search term is "unstable\_". The results show 14 matches across 6 files. The code snippets highlight the "unstable\_" prefix in yellow.

File	Line Number
Cache.client.js	9
Cache.client.js	17
Cache.client.js	25
EditButton.client.js	9
EditButton.client.js	15
NoteEditor.client.js	9
NoteEditor.client.js	21
SearchField.client.js	9

"14 MATCHES IN 6 FILES"

A screenshot of a web browser displaying a blog application titled "React Training Blog". The URL in the address bar is "localhost:4001". The page shows three blog posts:

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

To the right of the posts is a sidebar titled "Tags" containing the following categories:

- 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?

- Viel statischer Content
- Rendern des statischen Contents benötigt 3rd-Party Libs
- Minimale Benutzer-Interaktionen (PostEditor)

## EIN BEISPIEL

### Was macht die Beispiel-Anwendung aus?

- Viel statischer Content
- Rendern des statischen Contents benötigt 3rd-Party Libs
- Minimale Benutzer-Interaktionen (PostEditor)

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

## Option: Serverseitiges Rendern (SSR)

1. Bei SSR wird die Anwendung auf dem Server ausgeführt

## Option: 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

## Option: 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

## DATEN LADEN

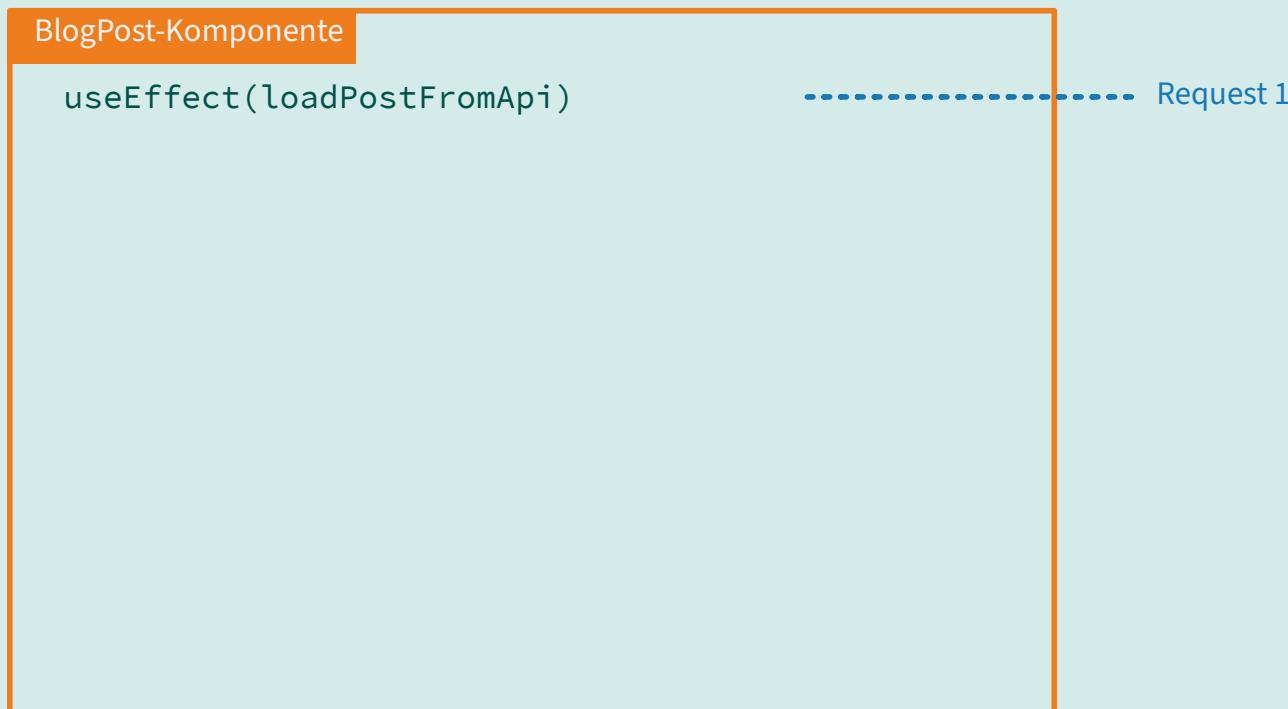
### Mögliches Problem: Laden von Daten auf dem Client

- Eine Komponente lädt ihre Daten, Unterkomponenten müssen warten

# DATEN LADEN

## Laden von Daten auf dem Client

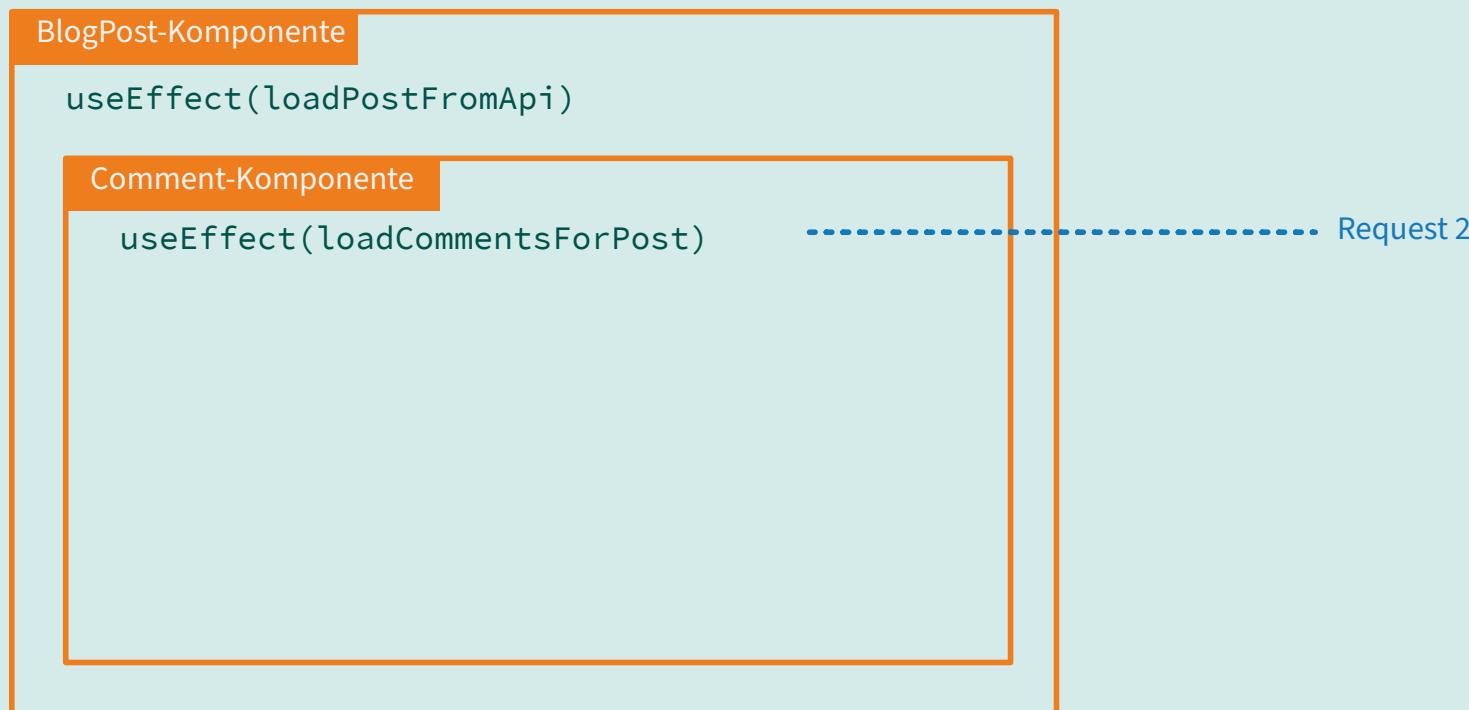
- Eine Komponente lädt ihre Daten, Unterkomponenten müssen warten



# DATEN LADEN

## Laden von Daten auf dem Client

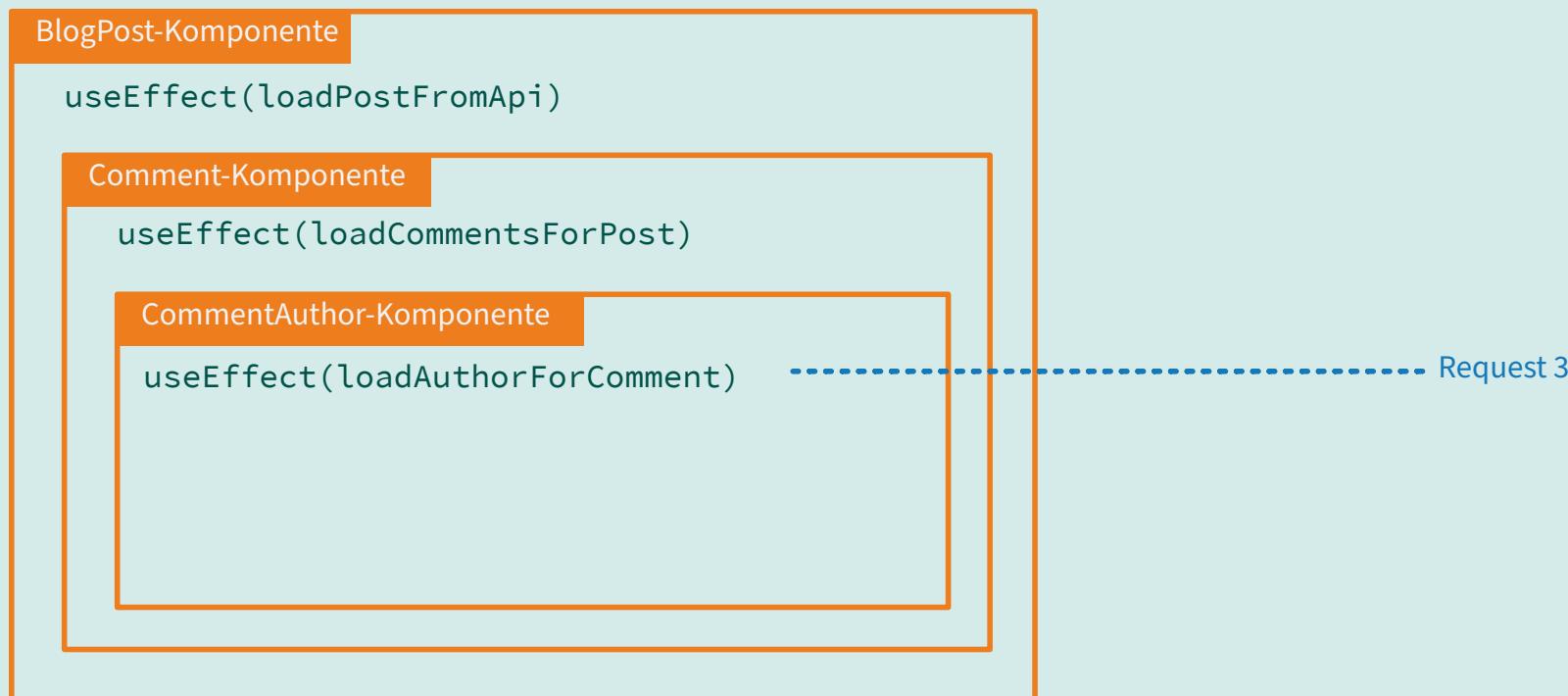
- Eine Komponente lädt ihre Daten, Unterkomponenten müssen warten



# DATEN LADEN

## Laden von Daten auf dem Client

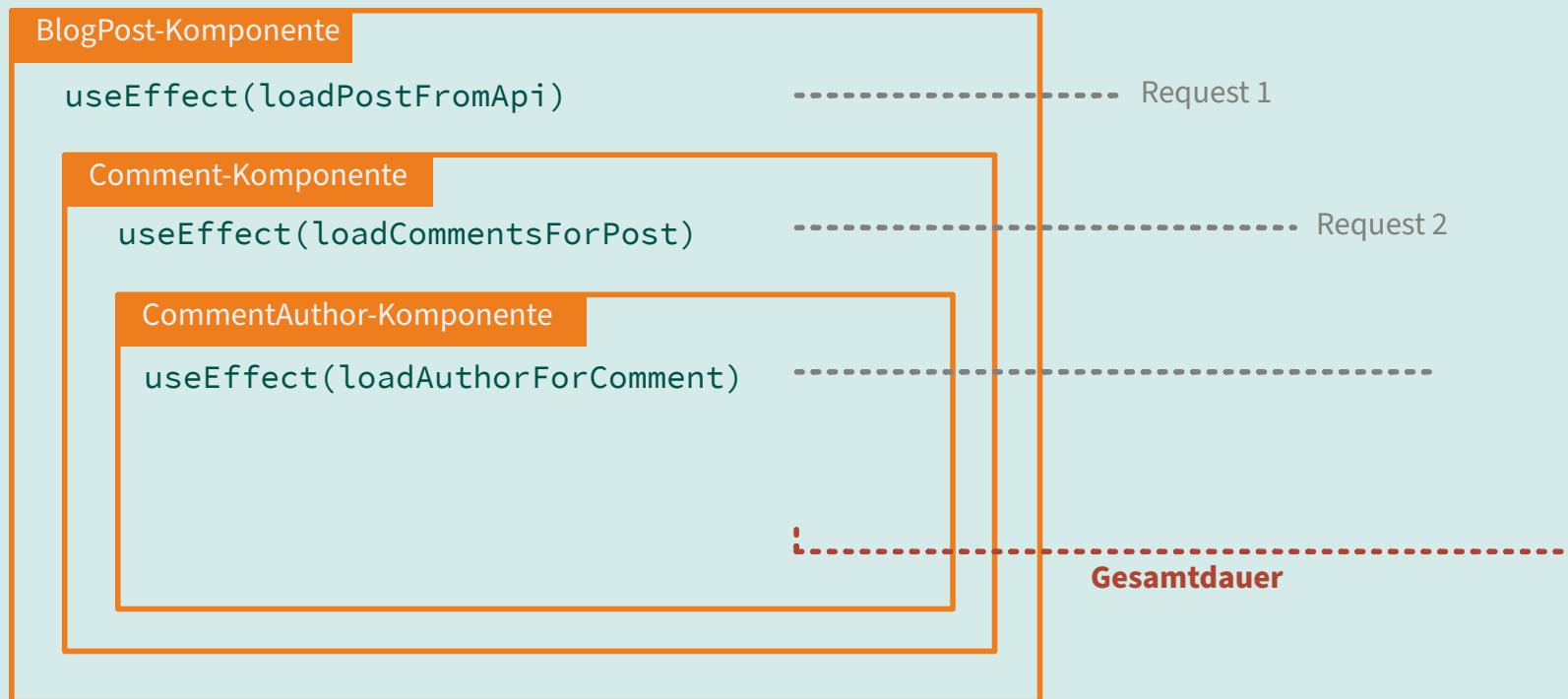
- Eine Komponente lädt ihre Daten, Unterkomponenten müssen warten



# DATEN LADEN

## Laden von Daten auf dem Client

- Eine Komponente lädt ihre Daten, Unterkomponenten müssen warten



Wasserfall...

# **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"

## 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"

🤔 Bin mir nicht sicher, ob das nicht zu viel versprochen ist

# **SERVER COMPONENTS**

**Drei Arten von Komponenten**

# SERVER COMPONENTS

## Drei Arten von Komponenten

- **Client-Komponenten**

- wie bisherige React-Komponenten
- können keine Server-Komponenten verwenden
- gilt auch für Hooks

# SERVER COMPONENTS

## Drei Arten von Komponenten

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

- werden nur auf dem Server ausgeführt
- liefern UI zum React-Client zurück
  - kein HTML!
  - proprietäres Format

## Drei Arten von Komponenten

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

- werden nur auf dem Server ausgeführt
- liefern UI zum React-Client zurück
  - kein HTML!
  - proprietäres Format
- Restriktionen: kein useState, useEffect, Browser APIs
- Können Server Umgebung und Ressourcen nutzen
  - Datenbanken
  - Filesystem

## Drei Arten von Komponenten

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

- werden nur auf dem Server ausgeführt
- liefern UI zum React-Client zurück
  - kein HTML!
  - proprietäres Format
- Restriktionen: kein useState, useEffect, Browser APIs
- Können Server Umgebung und Ressourcen nutzen
  - Datenbanken
  - Filesystem
- Werden immer bis zur ersten Client-Komponente gerendert bzw. ausgeführt
- Gilt auch für Hooks!

## Drei Arten von Komponenten

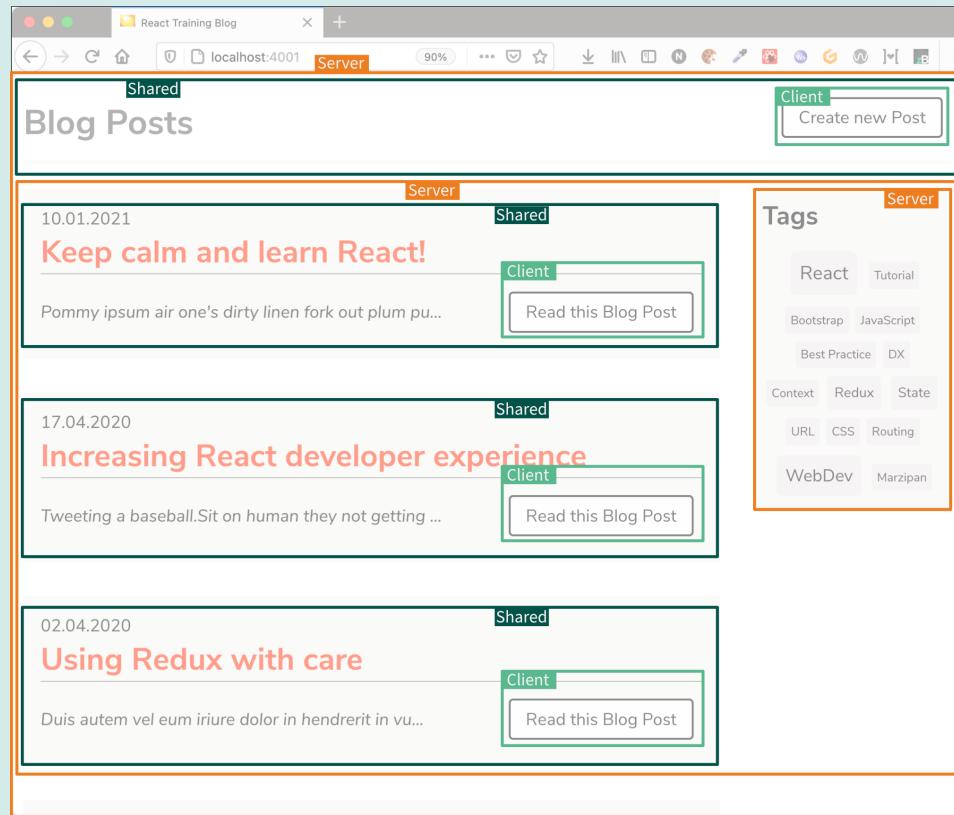
- **Shared Komponenten (Neu!)**

- werden auf dem Server und dem Client ausgeführt
- es gelten also die Restriktionen von Server und Client-Komponenten
  - keine Zugriff auf Server Umgebung
  - kein State, Effects etc.
- viele bestehende Presentation-Komponenten dürften in diese Kategorie fallen
- der entsprechende Code wird erst auf den Client übertragen, wenn er wirklich benötigt wird

# SERVER COMPONENTS

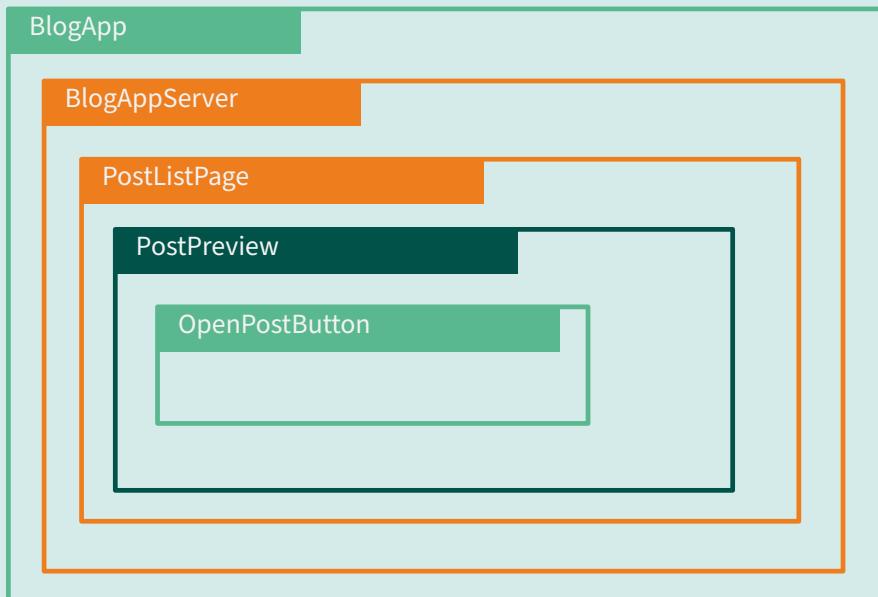
## Weiterhin ein Komponenten-Baum

- Ein Teil der Komponenten kommt jetzt jetzt vom Server...
- Der Server rendernt die Komponenten, bis er auf eine Client-Komponente trifft



# SERVER COMPONENTS

Wenn das eine normale Client-App wäre...



BlogApp würde Liste oder Einzel-Darstellung rendern

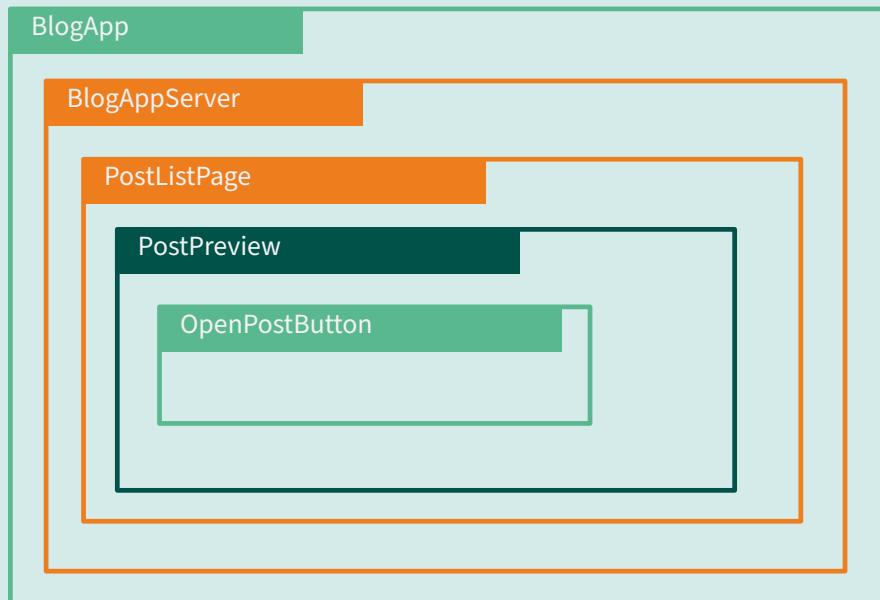
```
// Pseudo Code !!
BlogApp() {
  const [postId] = useState();

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

  return <PostListPage />
}
```

# SERVER COMPONENTS

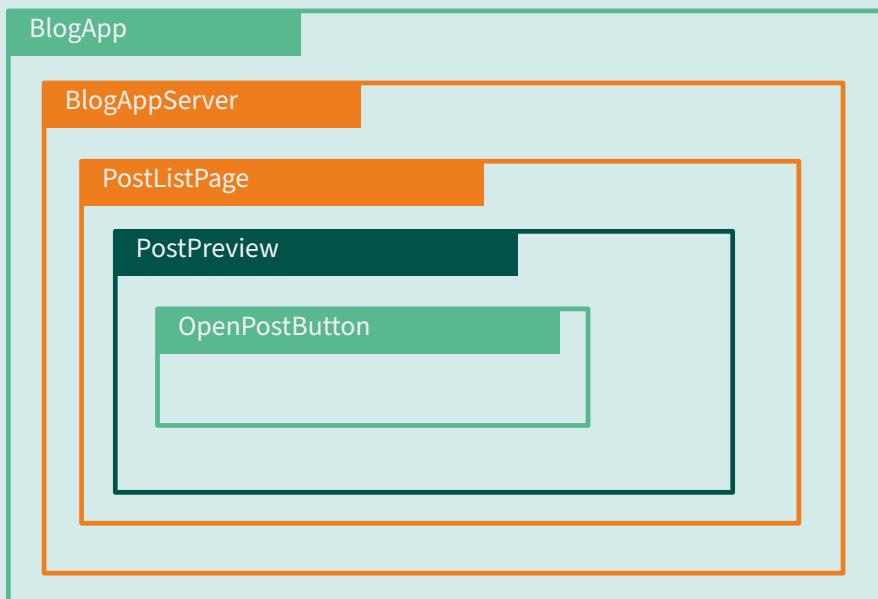
Wenn das eine normale Client-App wäre...



(BlogAppServer würde es nicht geben)

# SERVER COMPONENTS

Wenn das eine normale Client-App wäre...



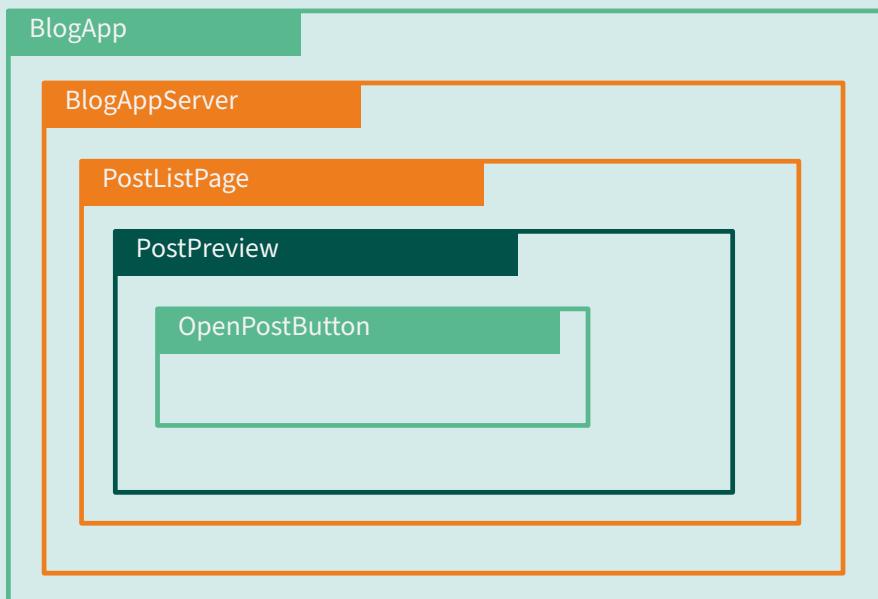
PostListPage würde Daten laden und Children rendern

```
// Pseudo Code !!
PostListPage() {
  useEffect(loadPosts());

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

# SERVER COMPONENTS

Wenn das eine normale Client-App wäre...

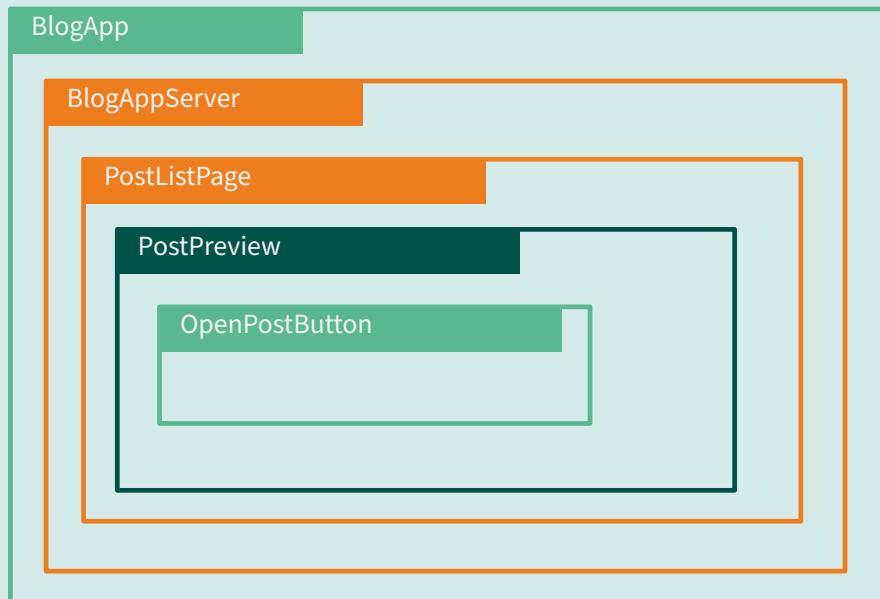


PostPreview würde Post  
darstellen und Knopf rendern  
// Pseudo Code !!

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

# SERVER COMPONENTS

Wenn das eine normale Client-App wäre...



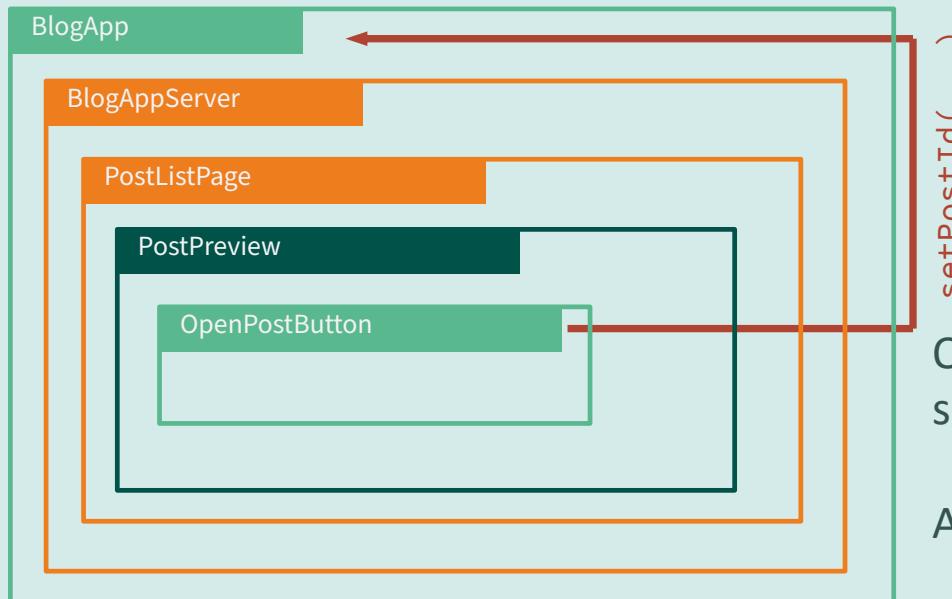
OpenPostButton würde neue Post-Id  
setzen

// Pseudo Code !!

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

# SERVER COMPONENTS

Wenn das eine normale Client-App wäre...

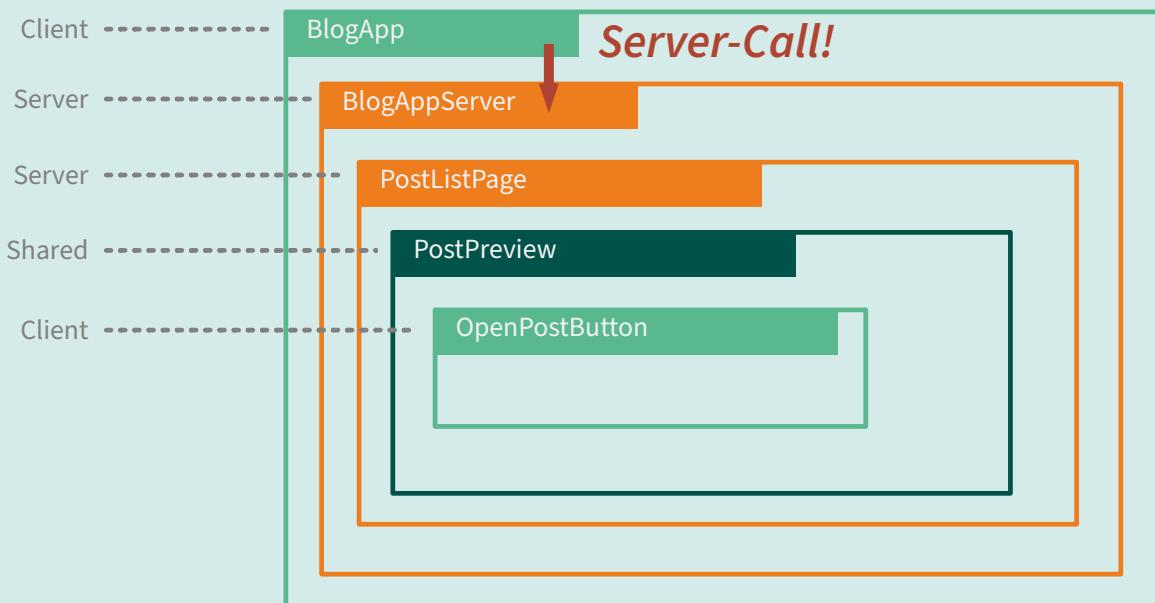


OpenPostButton würde neue Post-Id  
setzen,

App würde neu gerendert ✓

# 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!

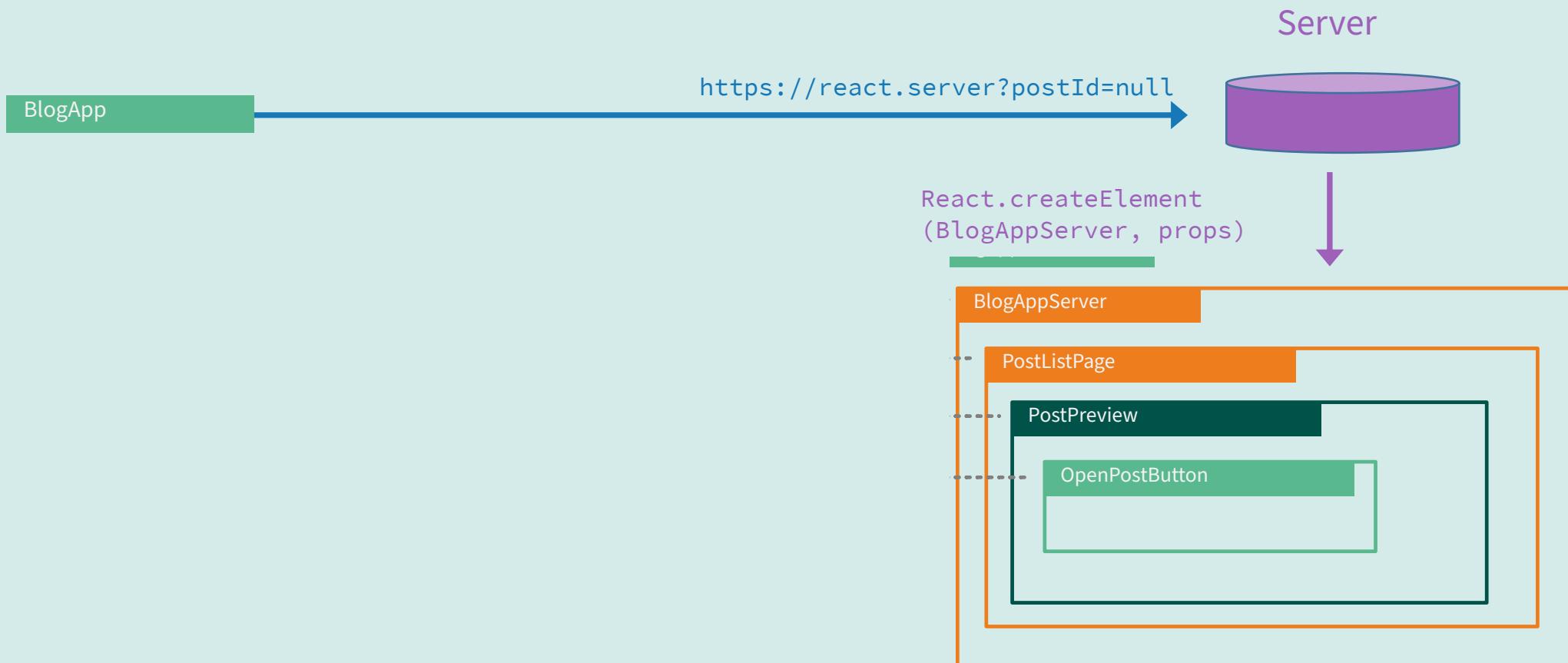


BlogApp  
- löst Server Request aus

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

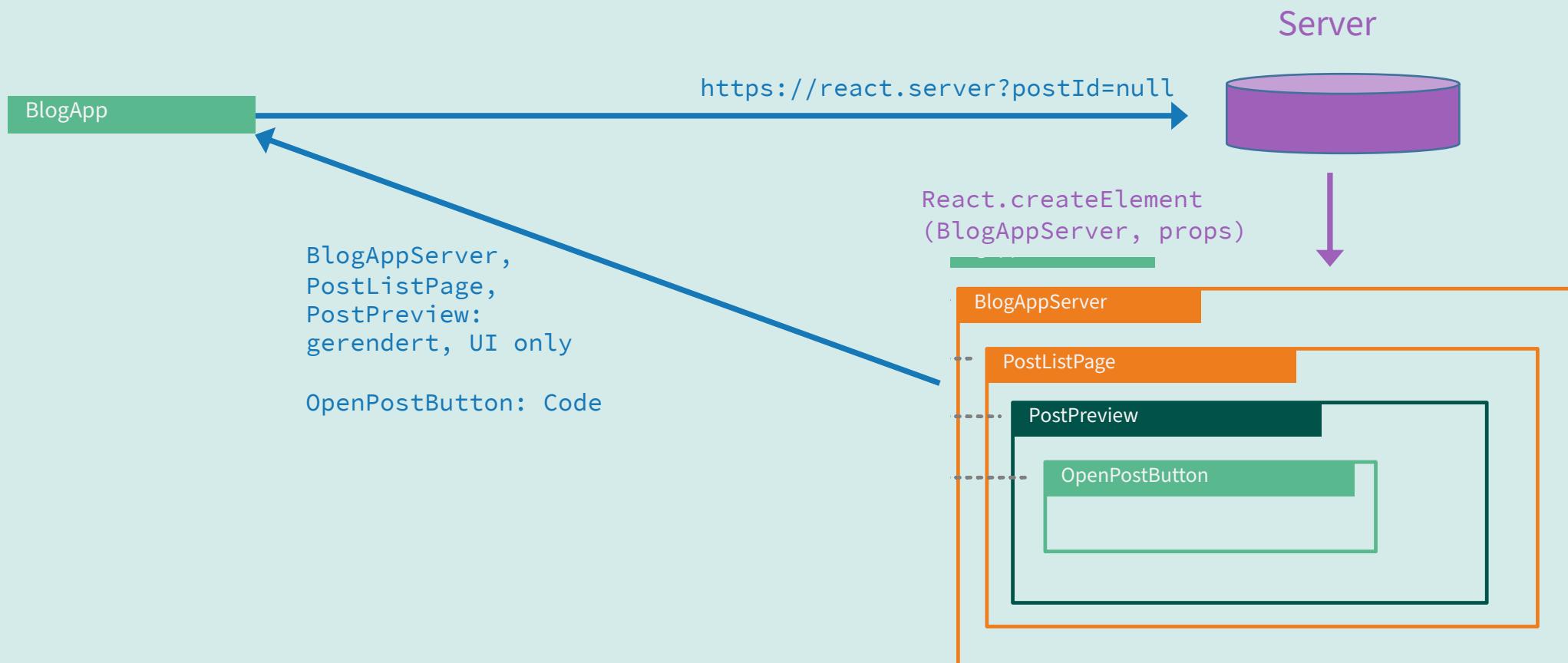
# SERVER COMPONENTS

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



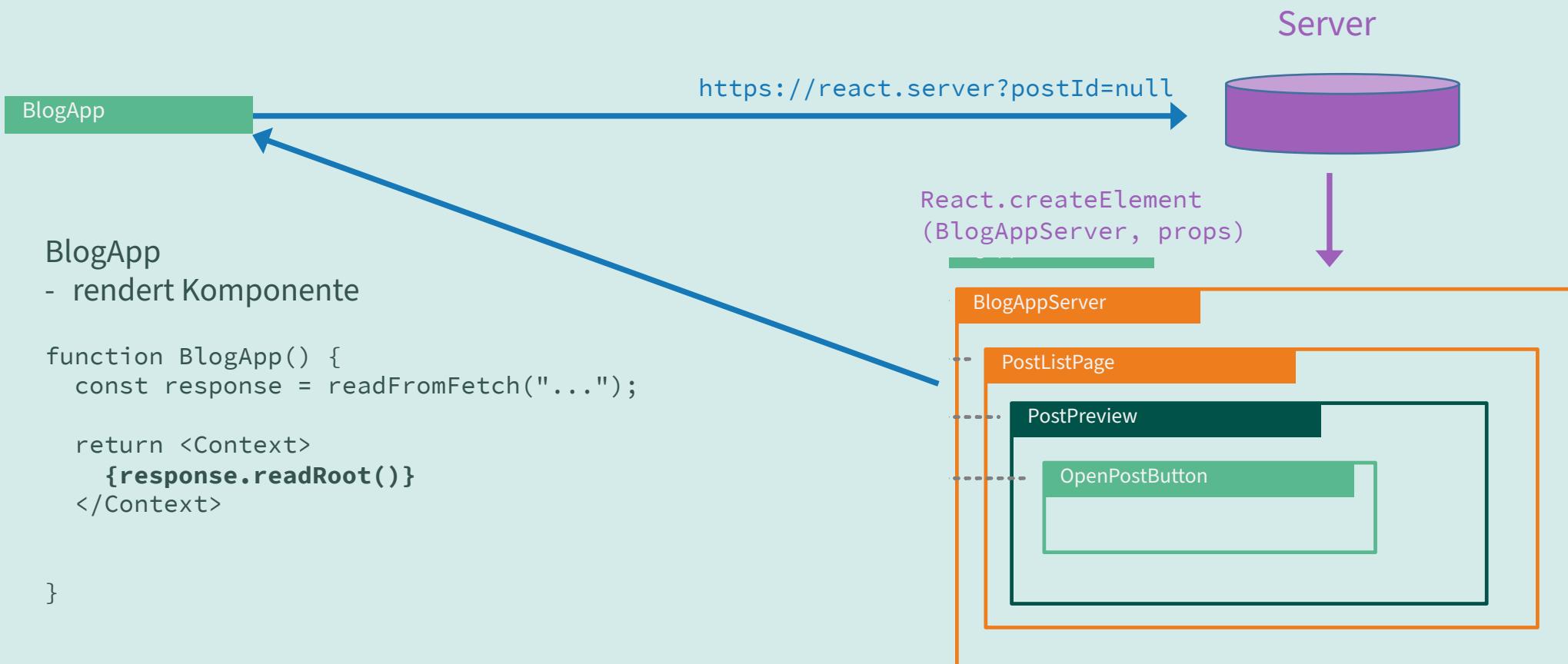
# SERVER COMPONENTS

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



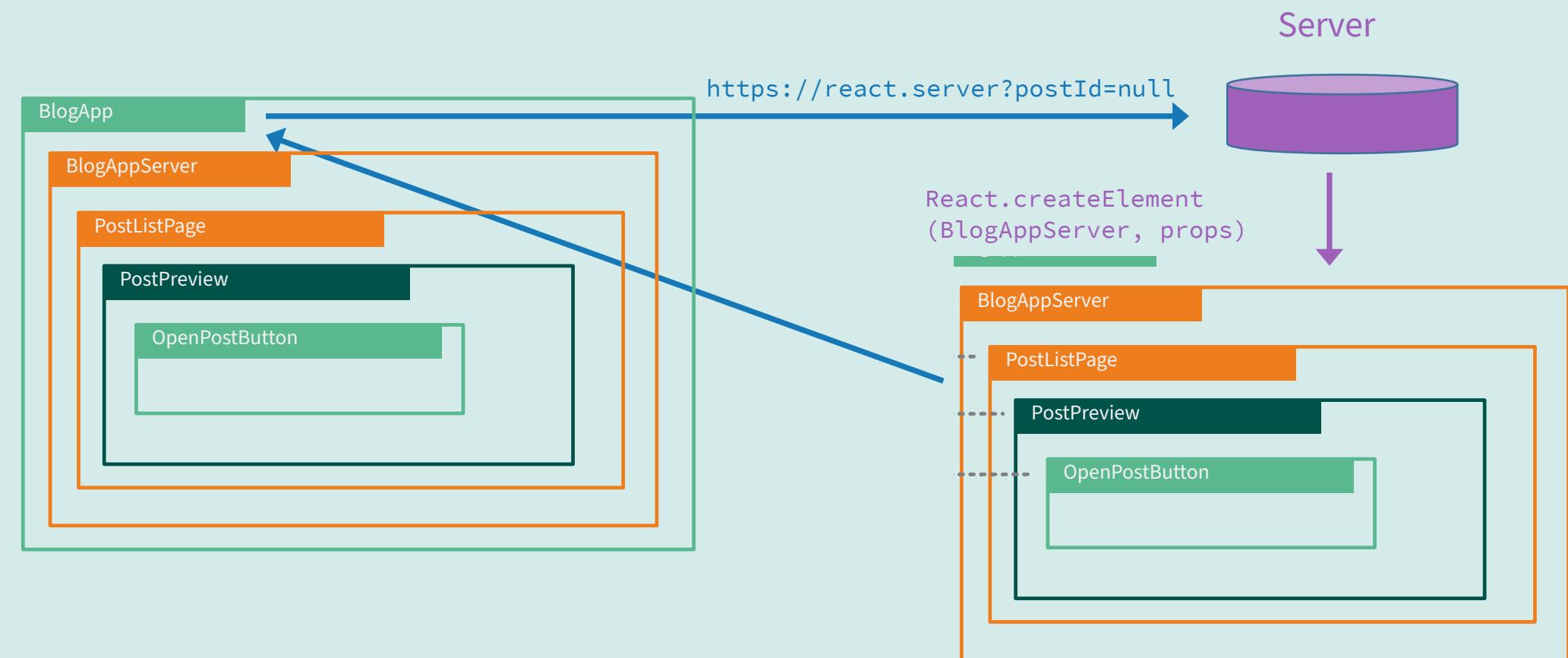
# SERVER COMPONENTS

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



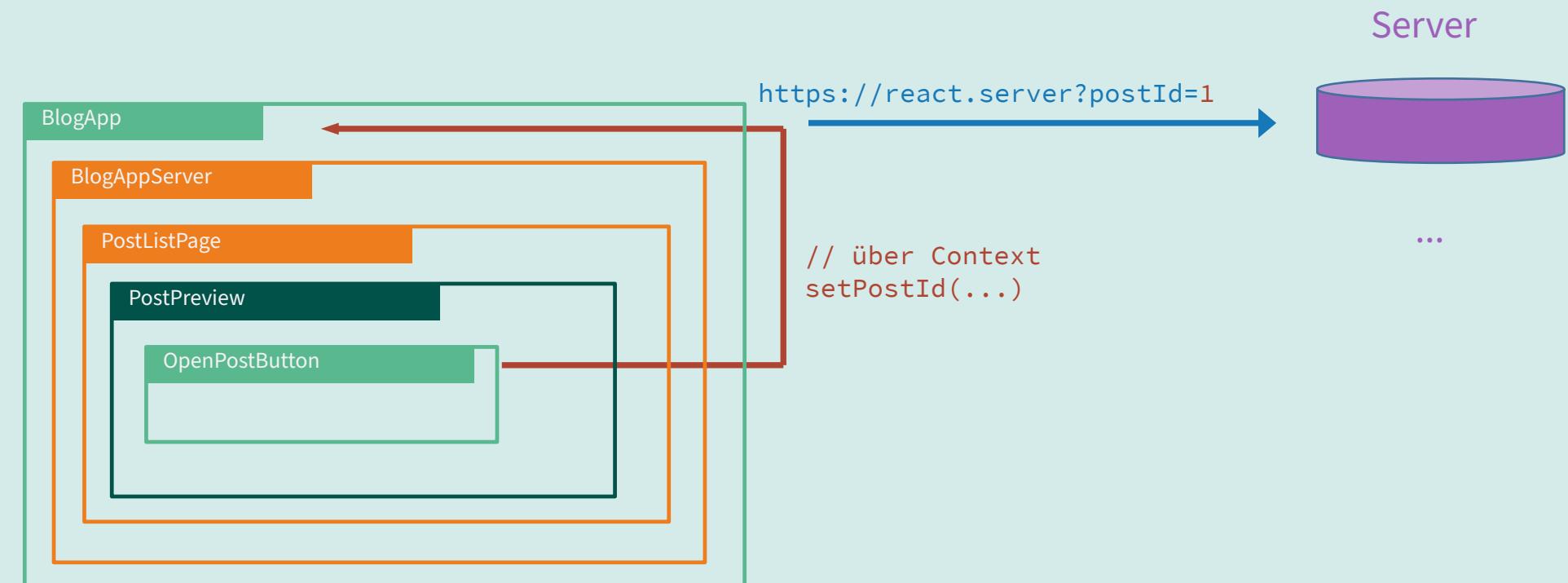
# SERVER COMPONENTS

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



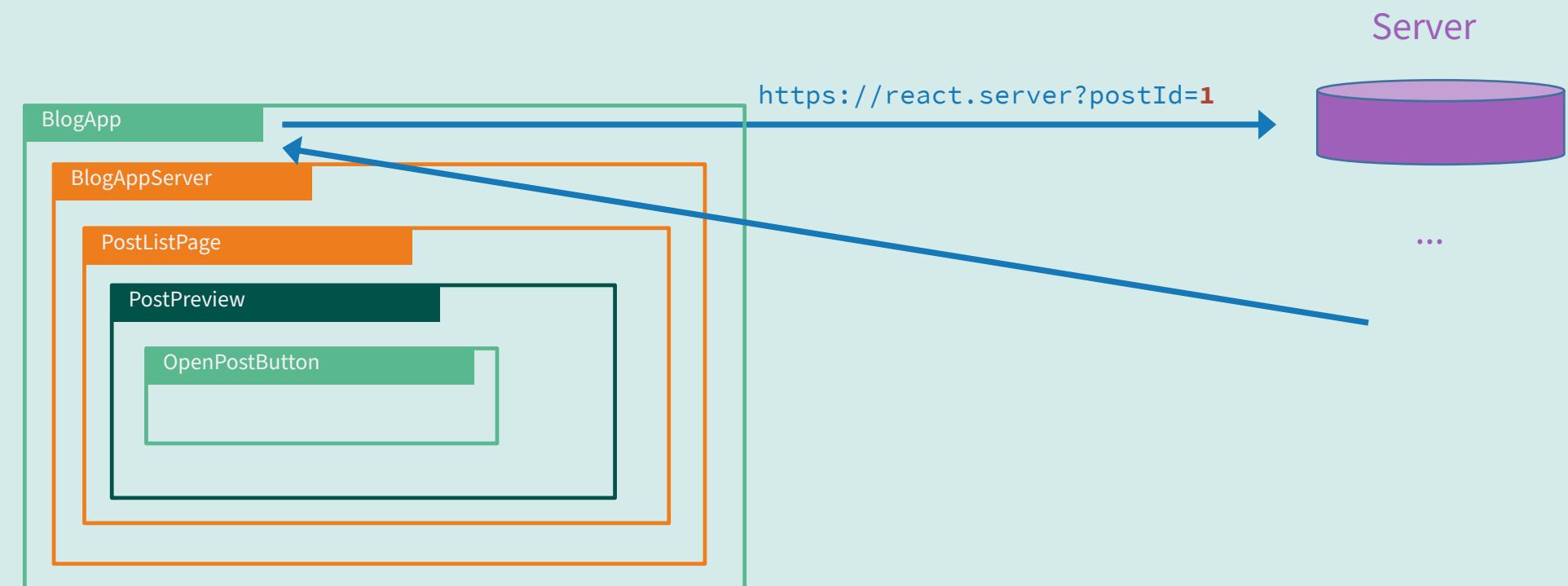
# SERVER COMPONENTS

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

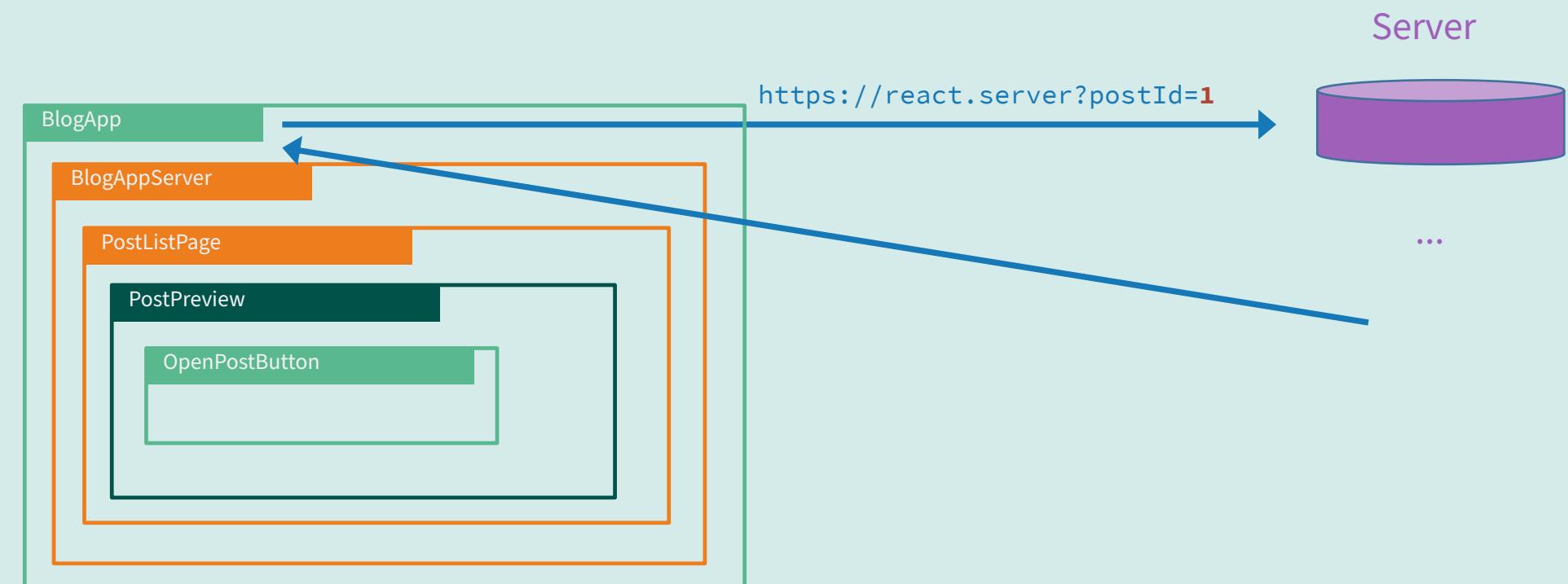


*Programmfluss "fast" wie in normalen React-Anwendungen,  
"nur" mit Server-Aufruf dazwischen 😊*

- 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 Rountrip erhalten (🕵️ Demo)

## SERVER COMPONENTS

### Beispiel: Eine Client Komponente mit State

CommentEditor.client.js

- implementieren
- in PostPreview (Shared Component) einbinden

## SERVER COMPONENTS

### Beispiel: Eine Server Komponente

PostComments.server.js

- implementieren
- in PostPage Server einbinden

## SERVER COMPONENTS

### Konsequenzen

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

## SERVER COMPONENTS

### Beispiel: Eine Client Komponente mit Server-Zugriff

CommentEditor.client.js

- fertig bauen

# SUSPENSE

## SUSPENSE

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

- Funktioniert aktuell für **Code Splitting** (Client)
  - Code Splitting in Server-Componenten eingebaut
  - **Künftig** auch zum **Laden von beliebigen Daten** (Client und Server)

# SUSPENSE

## 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 DB etc. aus Komponente möglich 😱
- 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

## SERVER COMPONENTS

### Zugriff auf Resourcen 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 Resourcen gilt das analog (Wrapper um fetch)
- Die Wrapper APIs können später wohl von der Community implementiert und zur Verfügung gestellt werden

# SERVER COMPONENTS

## Built-in Code Splitting

- Bundles für Client- und Shared Komponenten werden erst bei Bedarf auf den Client transferiert

```
import PostEditor from "./PostEditor.client";  
  
// App ist Server-Komponente  
export function App({ postId, editorOpen }) {  
  if (editorOpen) {  
    return <PostEditor />;  
  }  
  
  return (  
    <div className="App">  
      ...  
    </div>  
  );  
}
```

Bundle "PostEditor.client" wird erst zum Client geschickt, wenn editorOpen true ist!

**Fazit**

# **Server Components**

## SERVER COMPONENTS

### Aktueller Stand: Experimentell...

- Es gibt eine offizielle Beispiel App, die zur Hälfte 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
  - Suspense
  - Concurrent Mode

## SERVER COMPONENTS

### 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

## SERVER COMPONENTS

### Einschätzung

- Wird noch dauern, bis global verfügbar
- Zusammen mit Suspense und Concurrent Mode "rundes" Paket
- Nicht für alle Anwendungen geeignet und notwendig

## 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
- Man muss JavaScript-Ausführung auf dem Server zulassen
  - Wer will das?

## 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>

**NILS HARTMANN**

<https://nilshartmann.net>



# vielen Dank!

Slides: <https://react.schule/wdc2021-server-components>

Fragen & Kontakt: [nils@nilshartmann.net](mailto:nils@nilshartmann.net)

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