

NILS HARTMANN

React

Server Components

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

ENTERJS ONLINE | 30. SEPTEMBER 2021 | @NILSHARTMANN

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-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"



Find in Files 14 matches in 6 files

File mask: *.java

Q不稳定_

In Project Module Directory Scope /blogexample-server-components/src ...

```
import { unstable_getCacheForType, unstable_useCacheRefresh } from "react" Cache.client.js 9
const refreshCache = unstable_useCacheRefresh(); Cache.client.js 17
const cache = unstable_getCacheForType(createResponseCache); Cache.client.js 25
import {unstable_useTransition} from 'react';
const [startTransition, isPending] = unstable_useTransition();
import {useState, unstable_useTransition} from 'react';
const [startNavigating, isNavigating] = unstable_useTransition();
import {useState, unstable_useTransition} from 'react';
```

"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 main content area 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!

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

Drei Arten von Komponenten

- Client-Komponenten

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

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

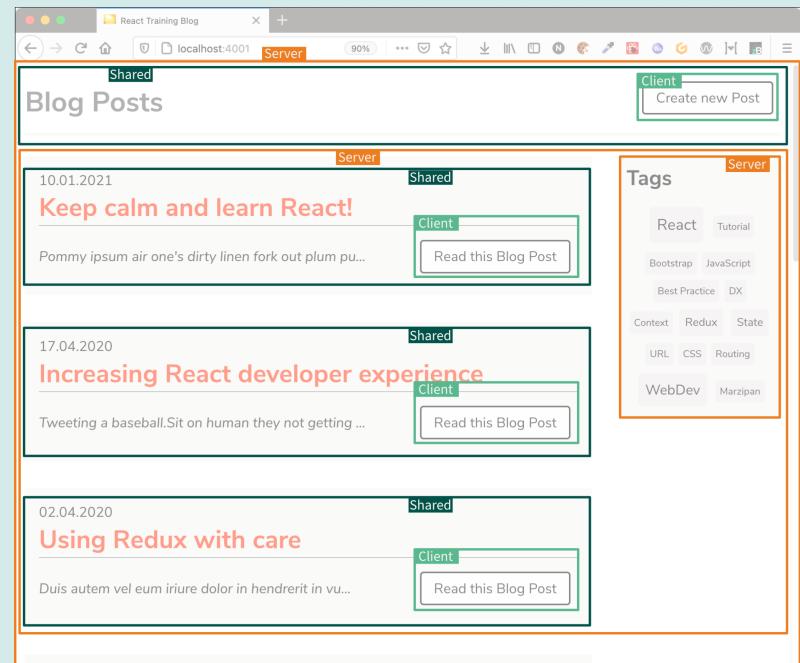
- **Shared Komponenten (Neu!)**

- 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 Komponenten dürften in diese Kategorie fallen
- der entsprechende Code wird erst auf den Client übertragen, wenn er wirklich benötigt wird

SERVER COMPONENTS

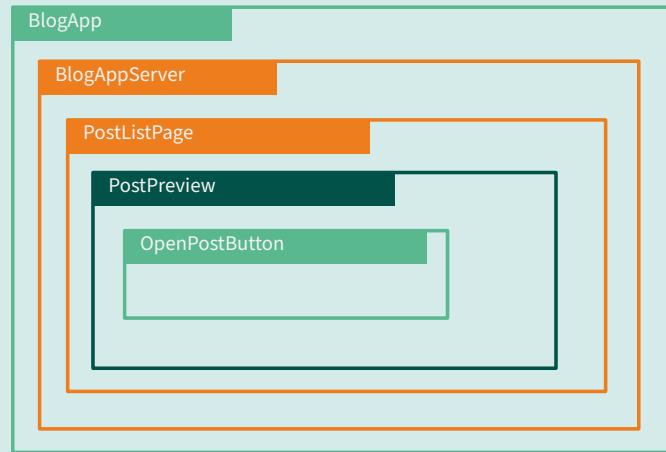
Weiterhin ein Komponenten-Baum

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



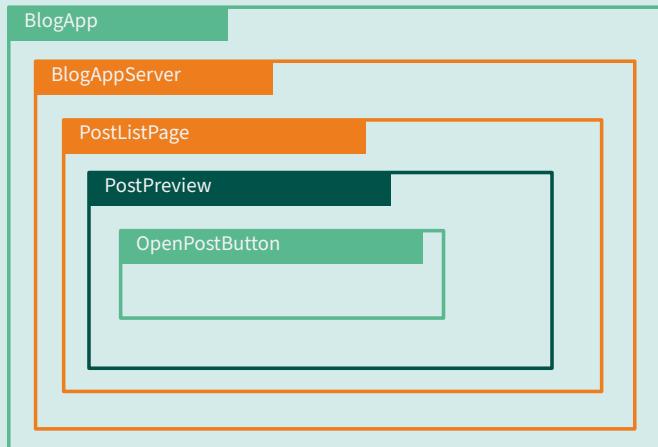
SERVER COMPONENTS

Wenn das eine normale Client-App wäre...



SERVER COMPONENTS

Wenn das eine normale Client-App wäre...



BlogApp würde Liste oder Einzel-Darstellung rendern

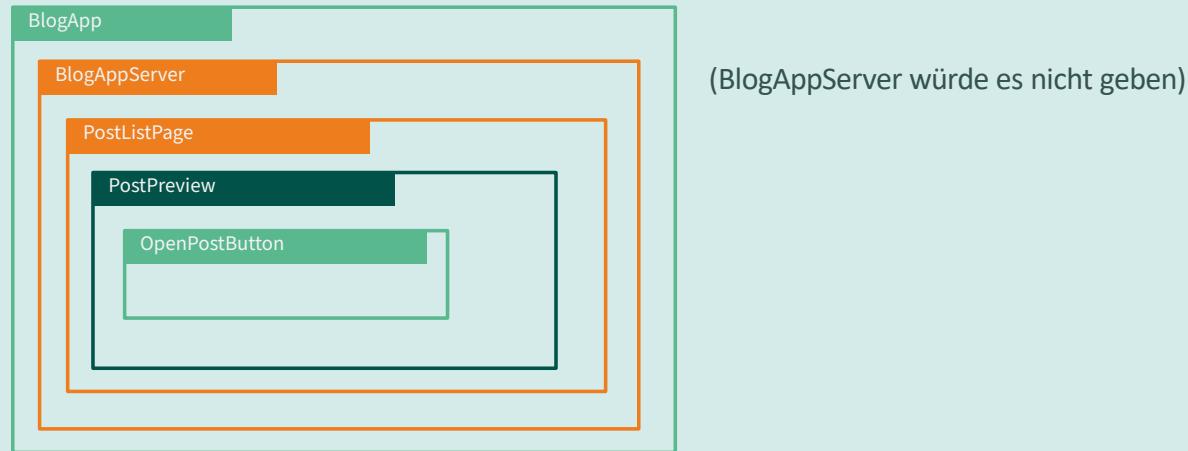
```
// Pseudo Code !!
BlogApp() {
  const [postId] = useState();

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

  return <PostListPage />
}
```

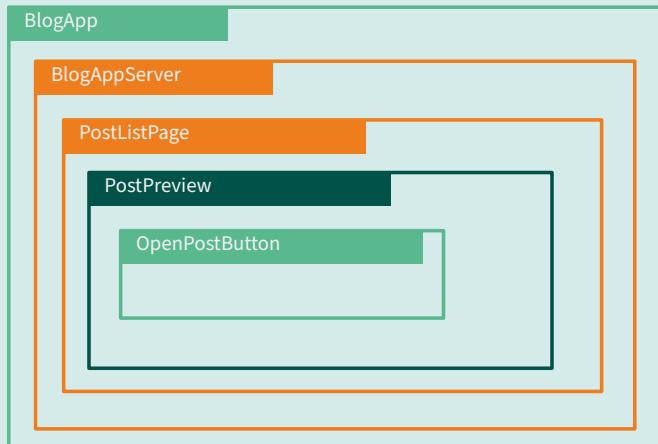
SERVER COMPONENTS

Wenn das eine normale Client-App wäre...



SERVER COMPONENTS

Wenn das eine normale Client-App wäre...



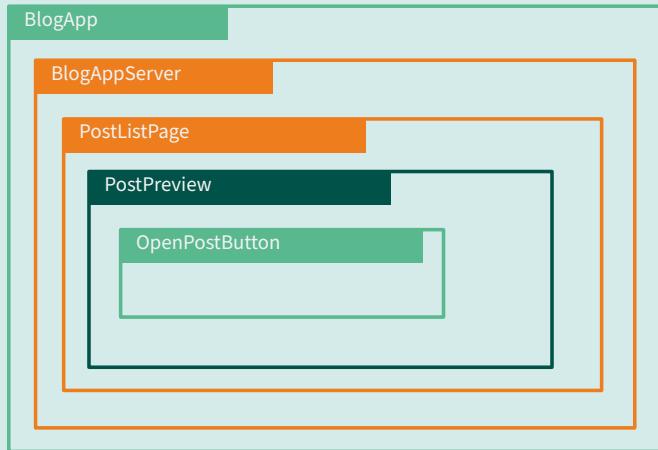
PostListPage würde Daten laden und Children rendern

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

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

SERVER COMPONENTS

Wenn das eine normale Client-App wäre...

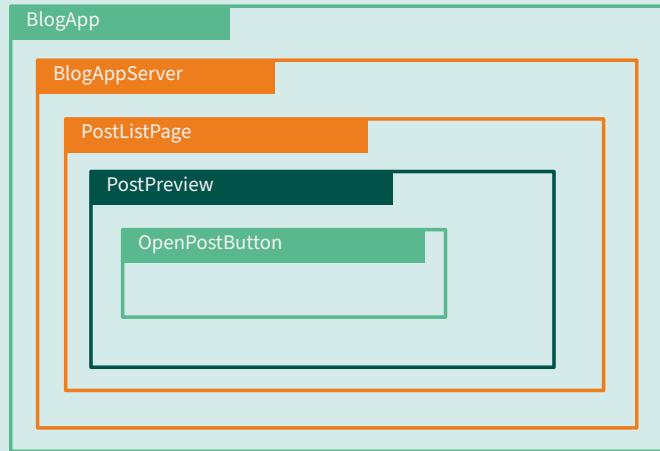


PostPreview würde Post
darstellen und Knopf rendern
// Pseudo Code !!

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

SERVER COMPONENTS

Wenn das eine normale Client-App wäre...



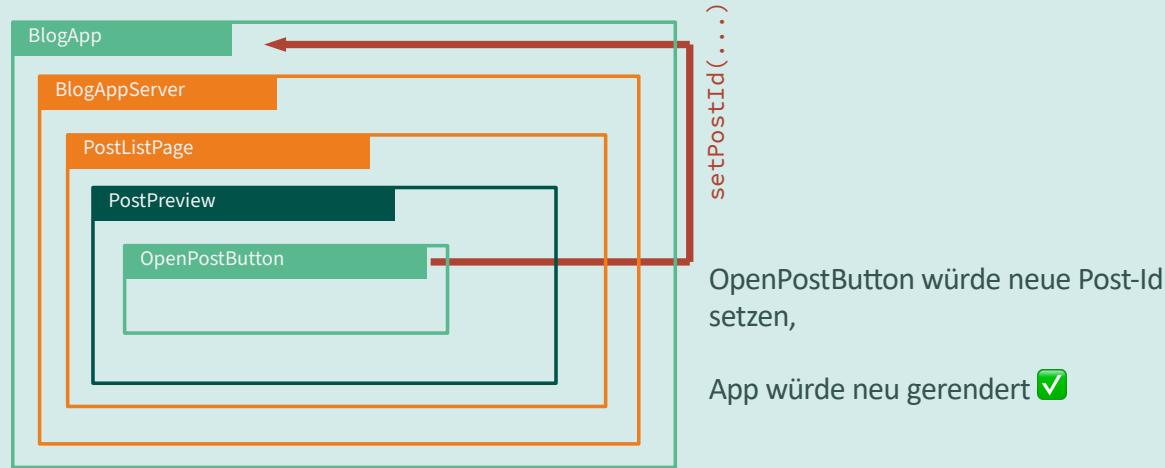
OpenPostButton würde neue Post-Id
setzen

// Pseudo Code !!

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

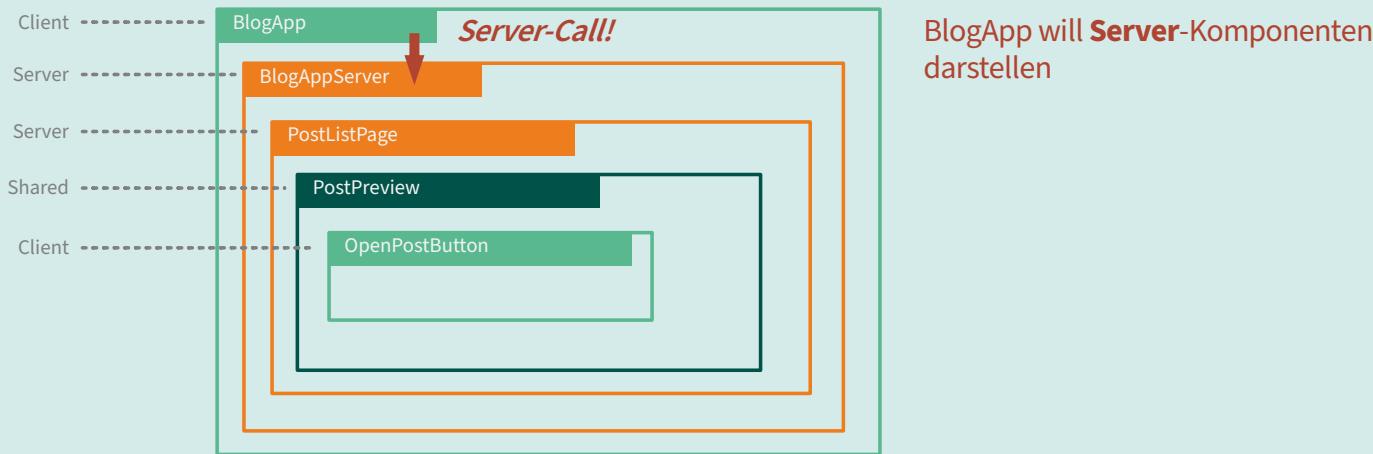
SERVER COMPONENTS

Wenn das eine normale Client-App wäre...



SERVER COMPONENTS

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



SERVER COMPONENTS

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



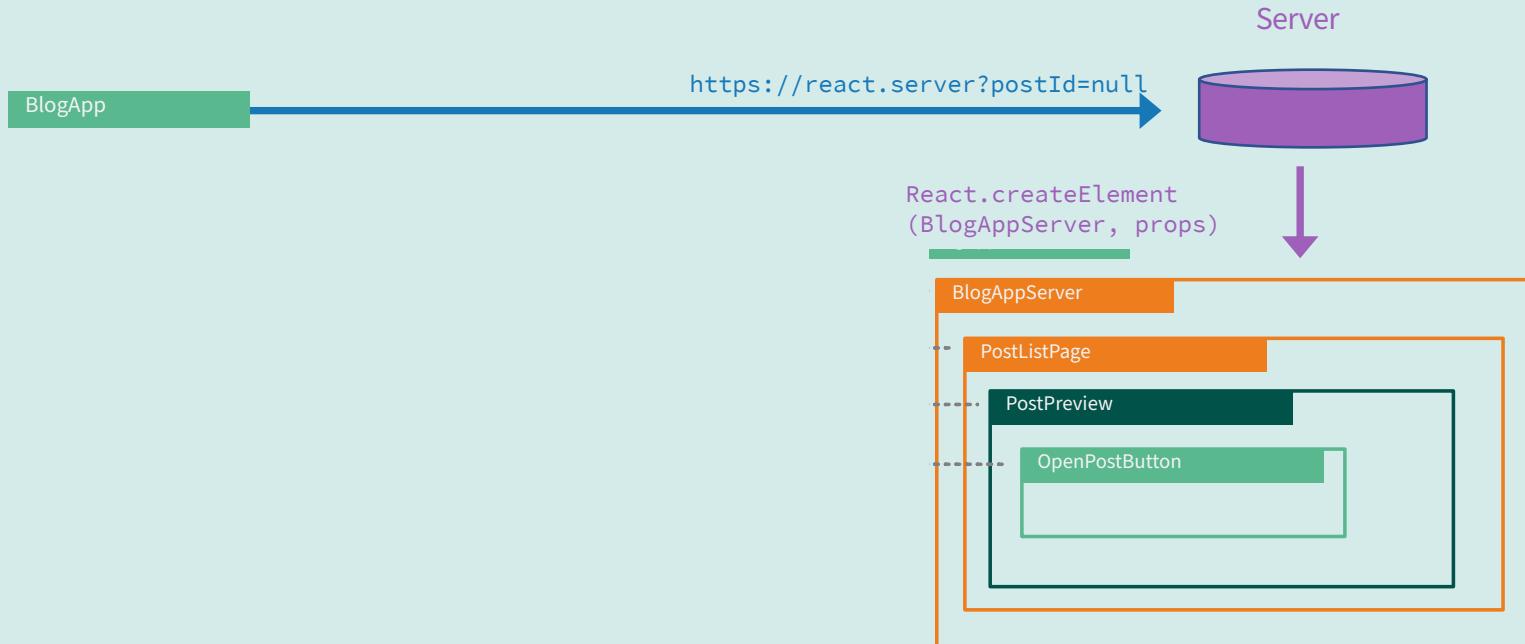
BlogApp

- löst Server Request aus

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

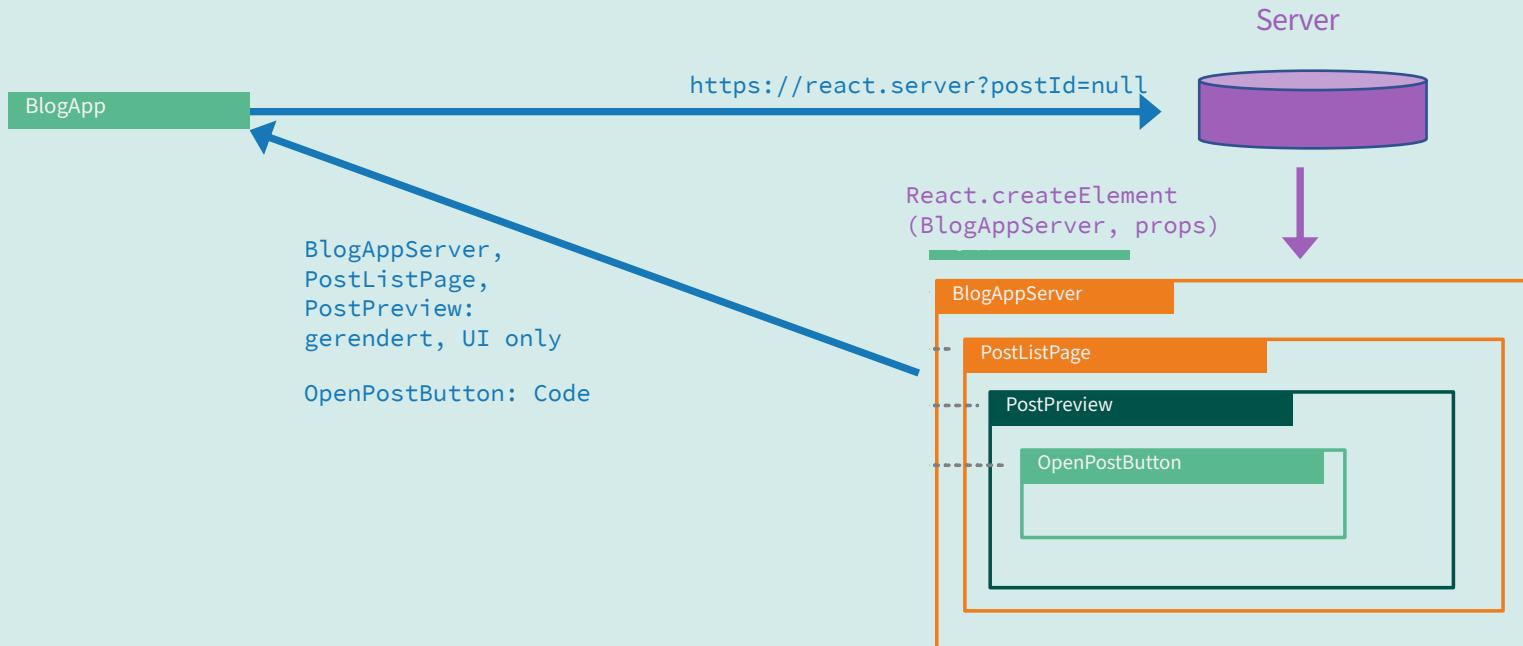
SERVER COMPONENTS

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



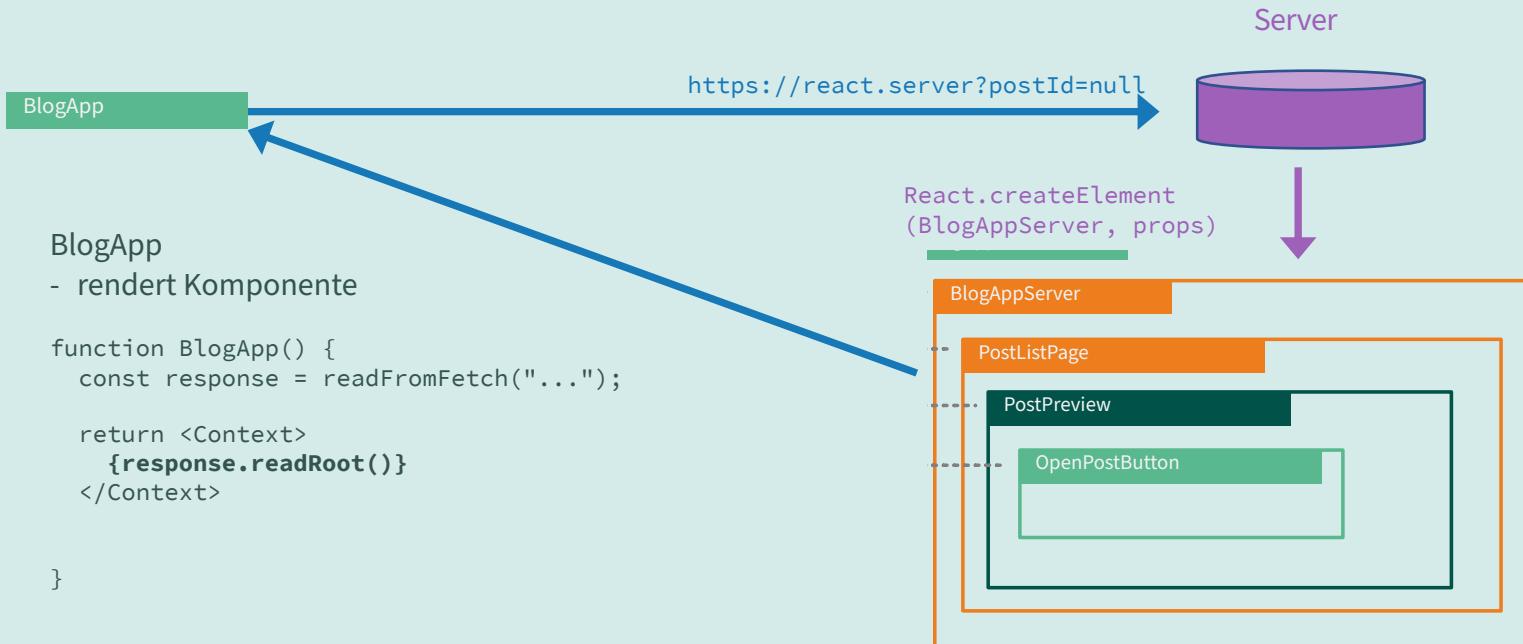
SERVER COMPONENTS

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



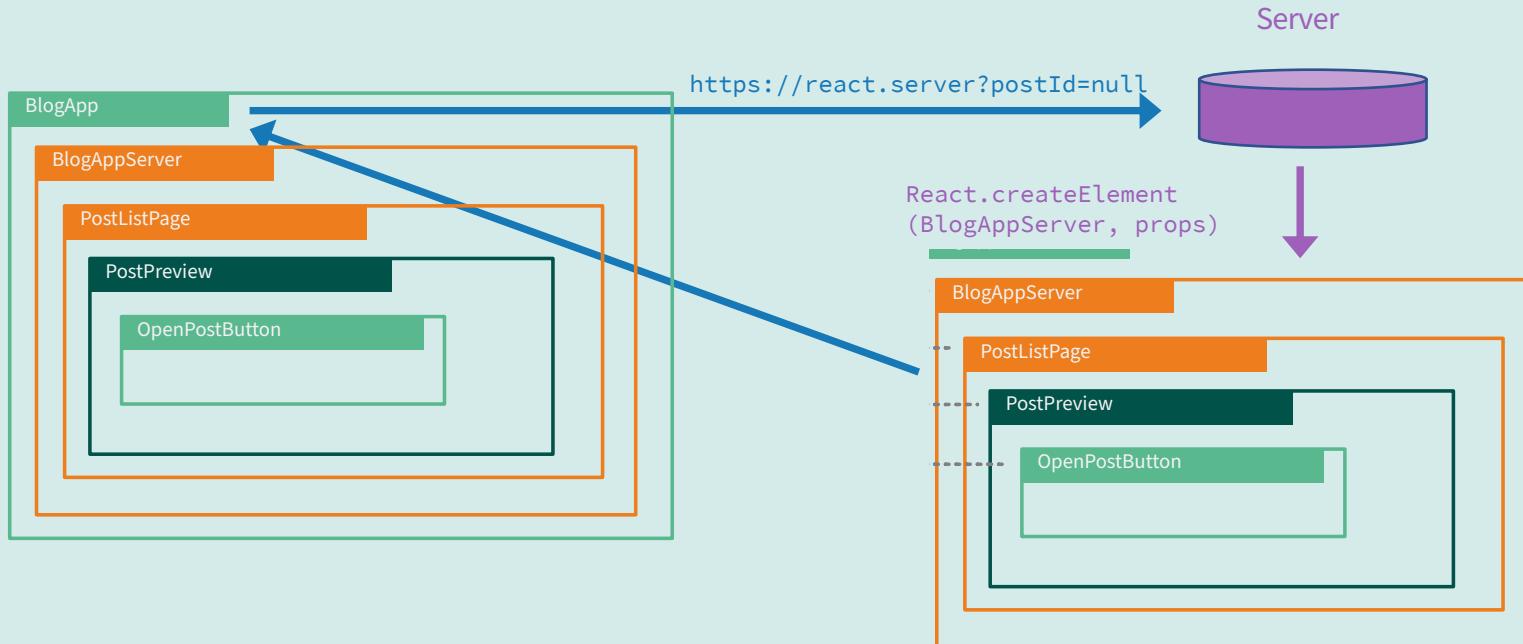
SERVER COMPONENTS

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



SERVER COMPONENTS

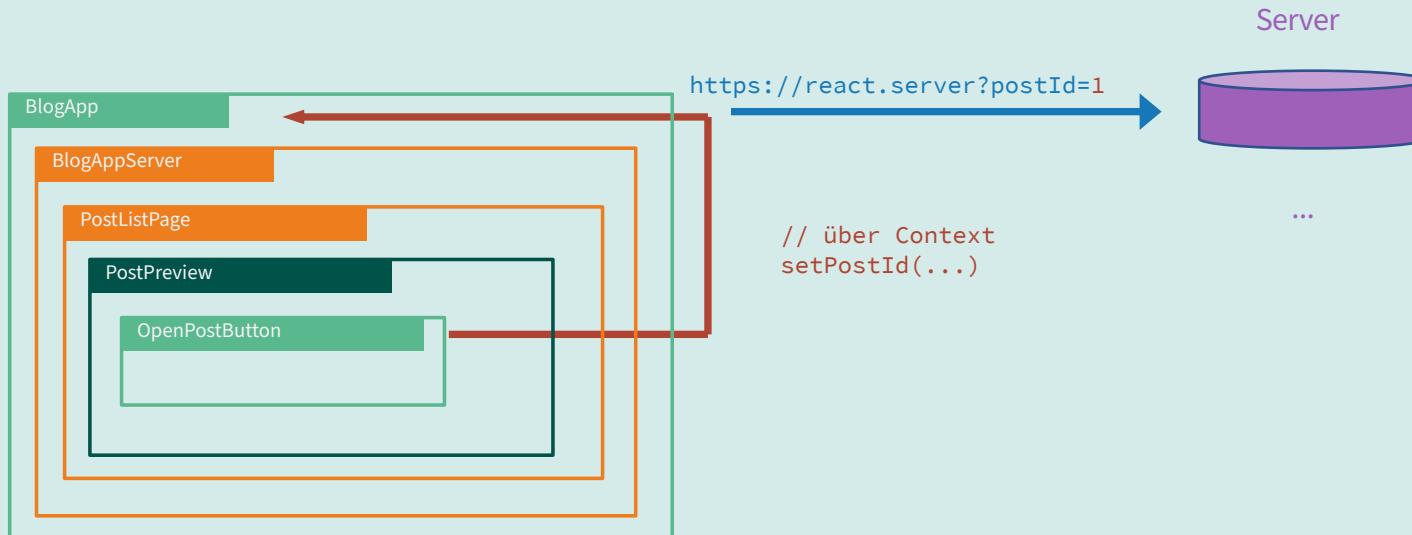
Wenn das eine normale Client-App wäre... ...ist es aber nicht!



SERVER COMPONENTS

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

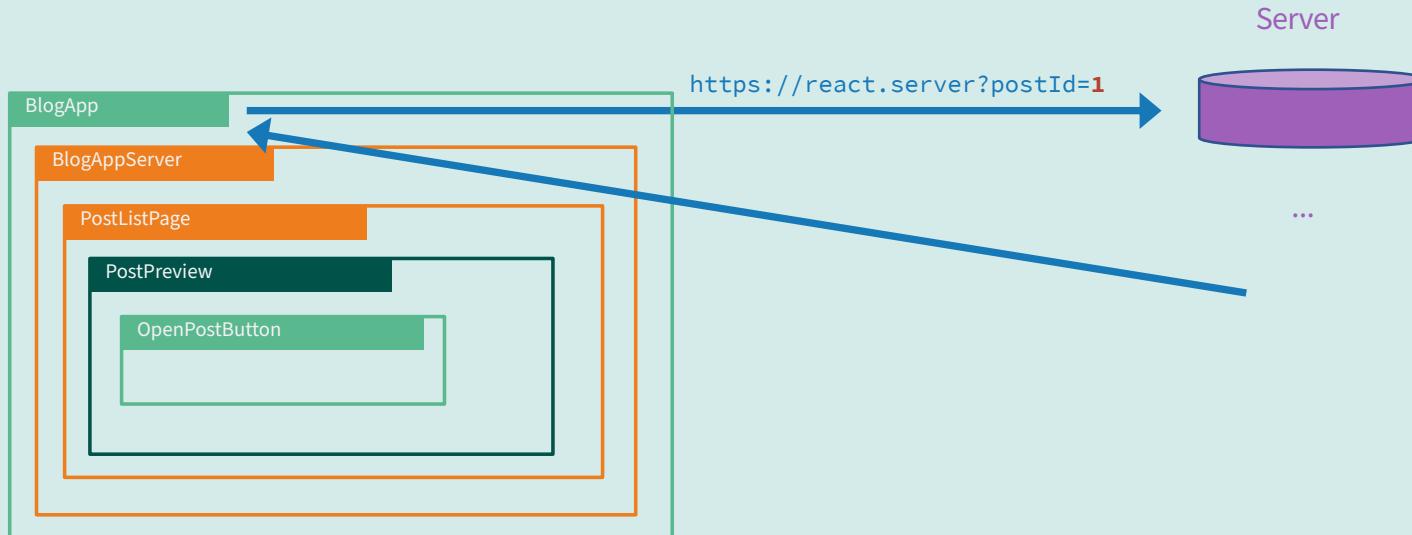
Kommunikation zurück nach oben



SERVER COMPONENTS

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

Kommunikation zurück nach oben



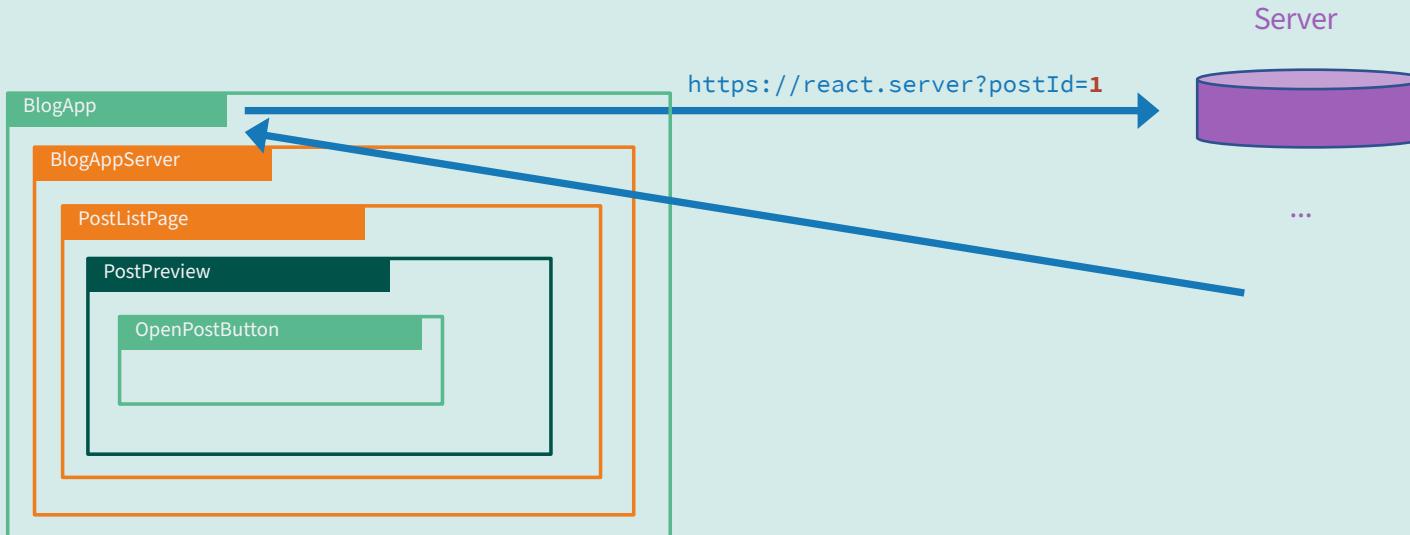
Programmfluss "fast" wie in normalen React-Anwendung,
"nur" mit Server-Aufruf dazwischen 😊

- 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 (👉 Demo)

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

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)



Demo

App.server.js und PostListPage.server.js

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

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

Fazit

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

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

- 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?

"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>



vielen Dank!

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

Fragen & Kontakt: nils@nilshartmann.net

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