NILS HARTMANN
https://nilshartmann.net

Das Ende der Single-Page-Anwendungen?

Fullstack Architekturen

NILS HARTMANN

nils@nilshartmann.net

Freiberuflicher Entwickler, Architekt, Trainer aus Hamburg

Java, Spring, GraphQL, React, TypeScript

https://graphql.schule/video-kurs

https://reactbuch.de

Teil 1

Der Weg zur Single-Page-Anwendung

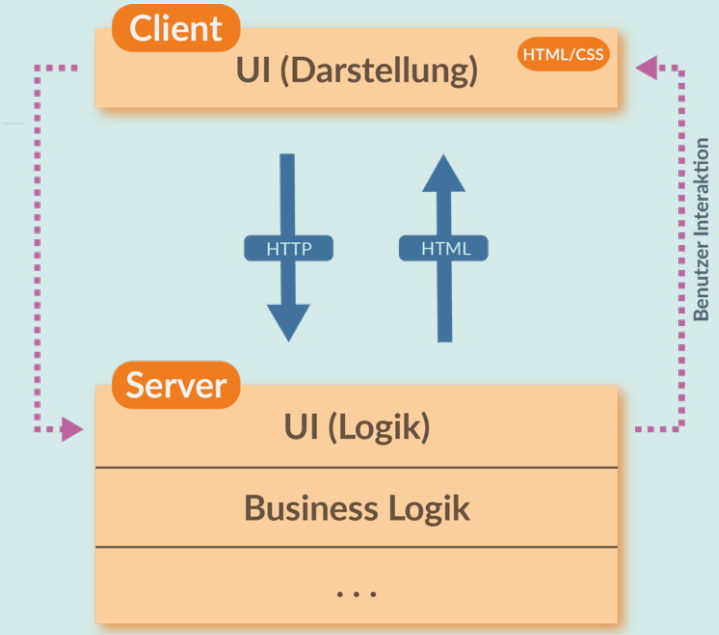
"Klassische" Web-Anwendung (Web 1.0)

Browser ist nur für die Darstellung zuständig

- UI wird auf dem Server erzeugt
  (z.B. JSP/JSF, ASP, PHP)

Konsequenz:

- Roundtrips für jede Kleinigkeit
- Dynamik / Interaktion fast unmöglich



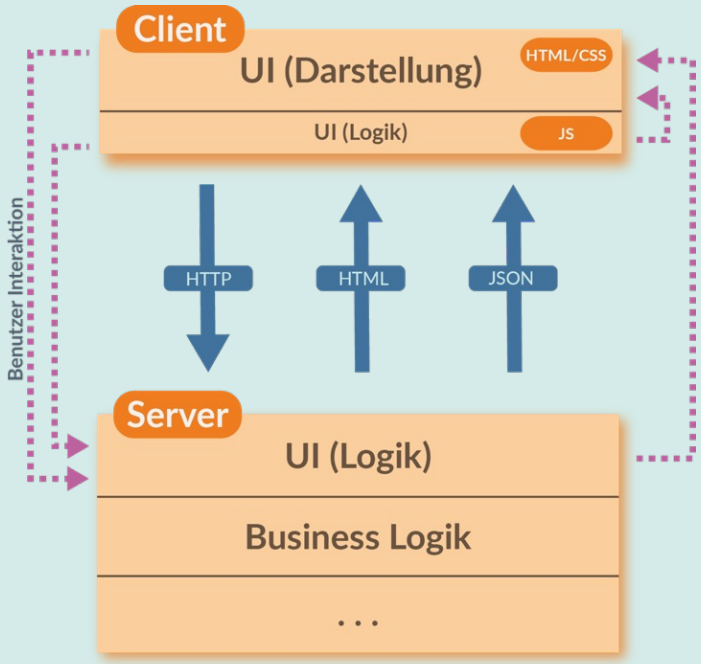
"Klassische" Web-Anwendung (Web 2.0)

Mit JavaScript-Schnippseln (früher jQuery):

- Interaktion
- Validierung
- Animationen 🤩

Konsequenz:

- Unsaubere Architektur (was ist wo?)
- Wartungshölle

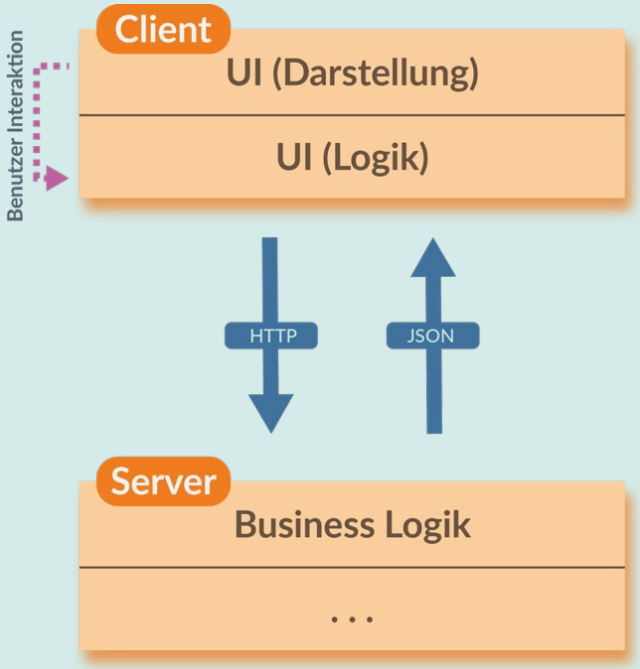


Single-Page-Anwendungen (ca. seit 2010)

- Client ist vollständig in JavaScript geschrieben
- Client läuft vollständig im Browser

Konsequenz:

- Klare Trennung zwischen Backend und Frontend
- Server "nur" für Geschäftslogik, Datenhaltung...
- Frontend "nur" für Darstellung und Interaktion
- Server muss APIs bereit stellen



Klassisch, mit JavaScript oder SPA?

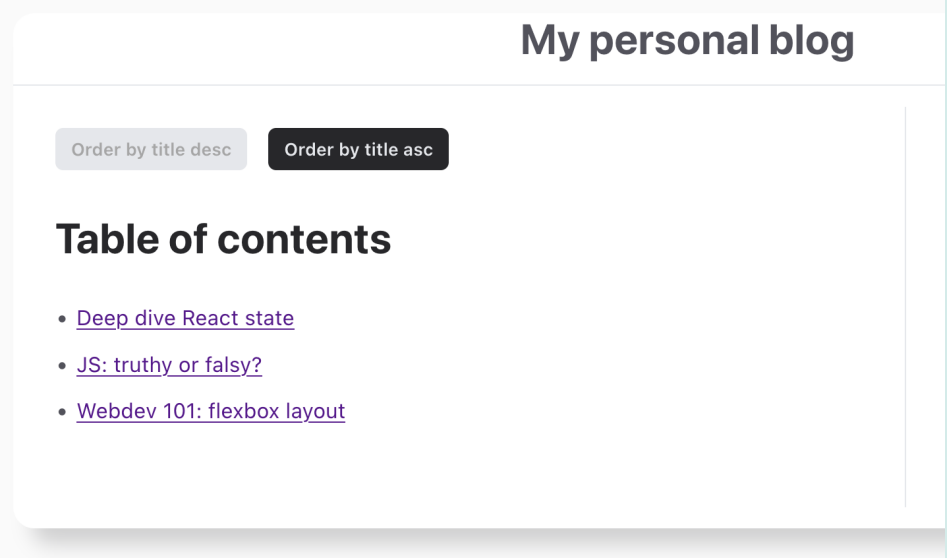
- Wir bauen Schritt-für-Schritt eine Anwendung
- Zu jedem Schritt überlegen wir, wie wir die Anwendung bauen würden:
  - **"klassisch"**: serverseitig, statisch gerenderte Anwendung (z.B. PHP, Spring WebMVC oder ASP.net)
  - "**klassisch + JS**" wie "klassisch" aber hier und da mit JavaScript-Schnippseln ("jQuery")
  - "**Single-Page-App**": Frontend komplett in JS, Daten im Backend, Kommunikation per HTTP API
- Es geht nicht (nur) darum, ob es technisch eindeutig eine klare Option gibt, sondern auch, welche Option "sinnvoll" ist
- Wenn ihr eine Anforderung für die Art der Anwendung unsinnig findet, könnt ihr das auch sagen

Klassisch, mit JavaScript oder SPA?

- Inhaltsverzeichnis für die Artikel einer Blogging-Platform

Anforderungen:

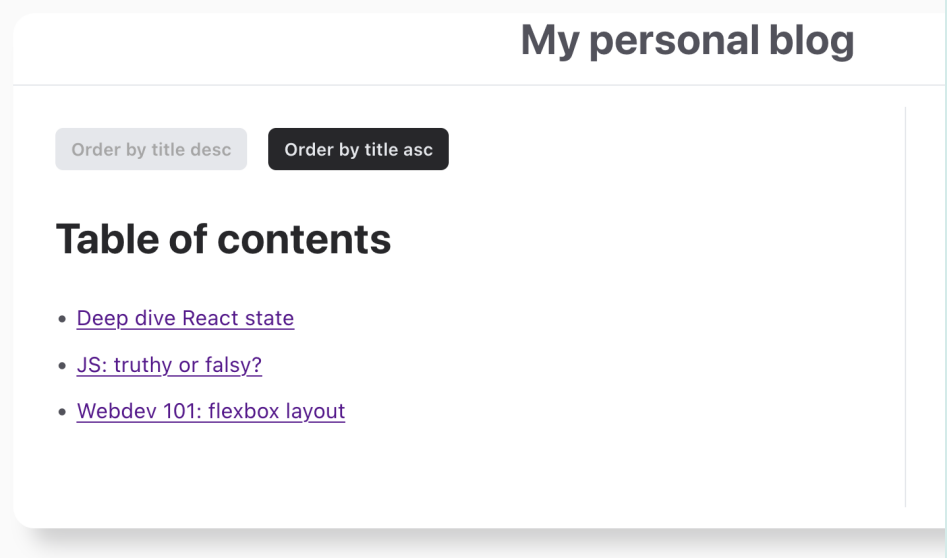
- Mit dem Klick auf einen Button wird die Liste der Titel umsortiert
- Der Button für die aktuelle Sortierung ist disabled
- Klick auf einen Blog-Titel öffnet den entsprechenden Artikel



Klassisch, mit JavaScript oder SPA?

Ohne JS möglich, klassische HTML-Seite:

- Serverseitig wird die Seite fertig als HTML gerendert
- Die Buttons zum Sortieren sind Links (a-Elemente)
- Der Server setzt entsprechende HTML- bzw. CSS-Attribute um die Button zu disablen



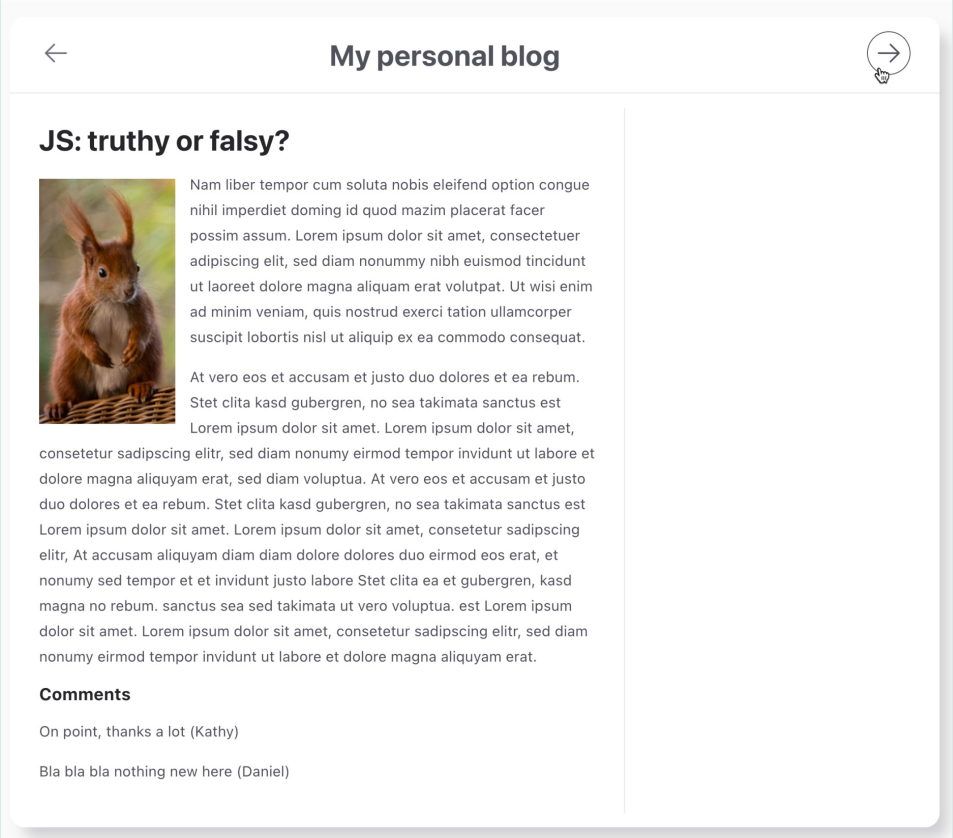
EINE WEB-ANWENDUNG

Klassisch, mit JavaScript oder SPA?

- Artikel in einer Blog-Anwendung

<u>Anforderungen:</u>

- Mit den Pfeilen kann man vorherigen/nächsten Blog-Post anzeigen
- Die einzelnen Posts sind per Link adressierbar/als Bookmark speicherbar
- Die Artikel werden von Google gefunden (SEO)

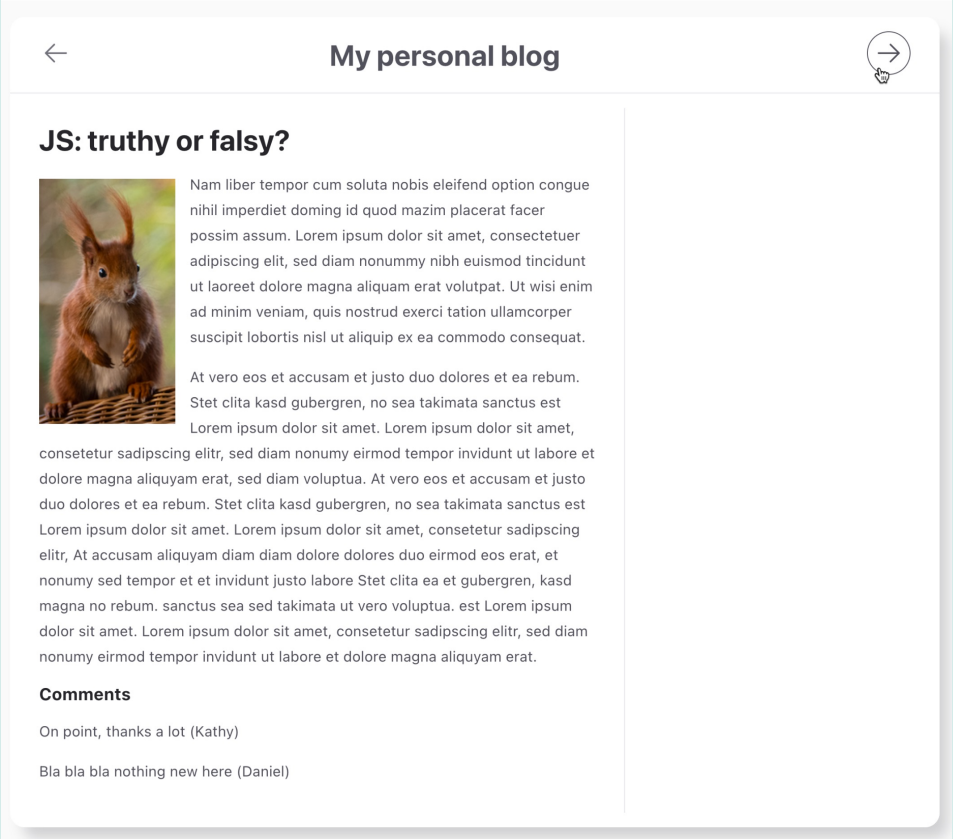


EINE WEB-ANWENDUNG

Klassisch, mit JavaScript oder SPA?

Ohne JS möglich, klassische Website

- Seite kommt fertig als HTML vom Server
- Die Id des Blog-Posts ist in der URL vorhanden und kann verlinkt werden
- Die Pfeile sind reguläre HTML-Links



step_01

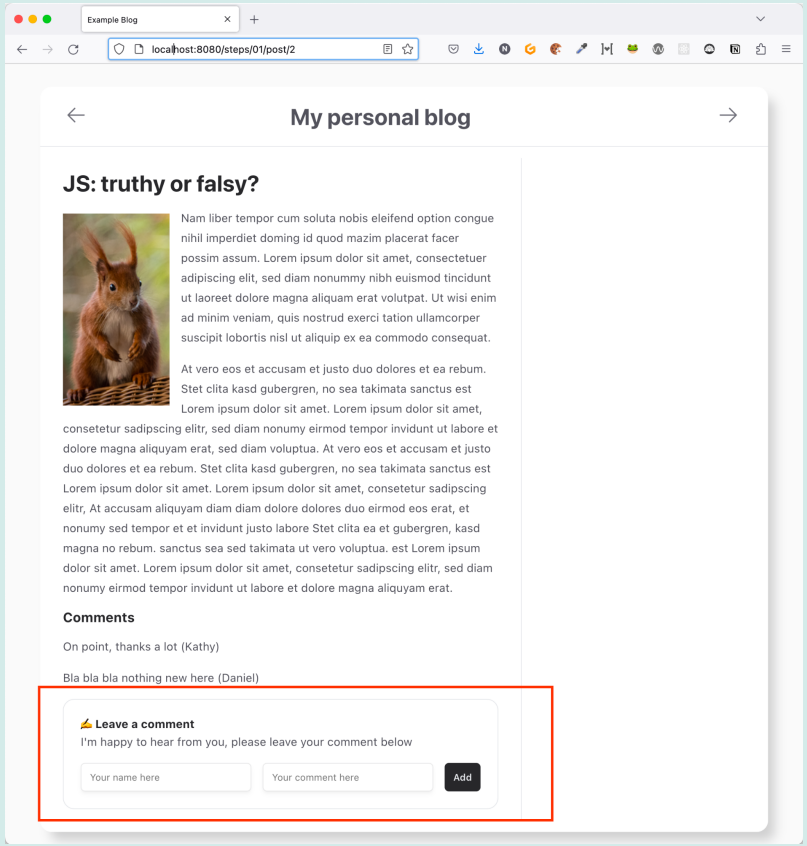
EINE WEB-ANWENDUNG

Klassisch, mit JavaScript oder SPA?

- Neu: Es gibt ein Formular für Kommentare

<u>Anforderungen:</u>

- Validierung: Beide Felder sind Pflicht. Wann und wo Fehlermeldung?
- Nach dem Speichern soll der Kommentar direkt angezeigt werden

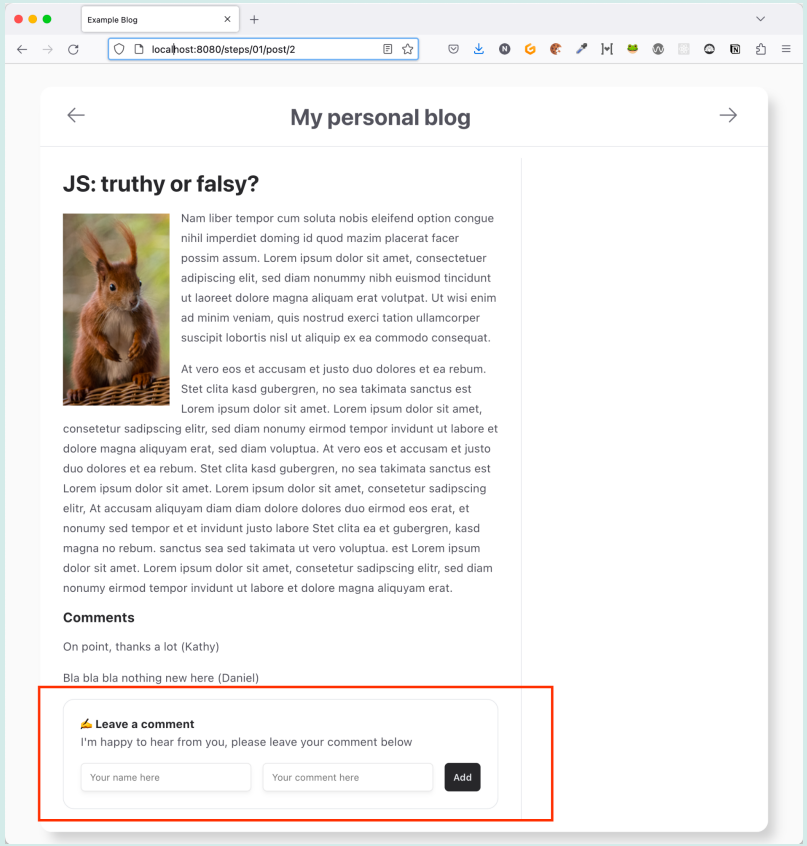



step_03

Klassisch, mit JavaScript oder SPA?

Wohl ohne JS möglich

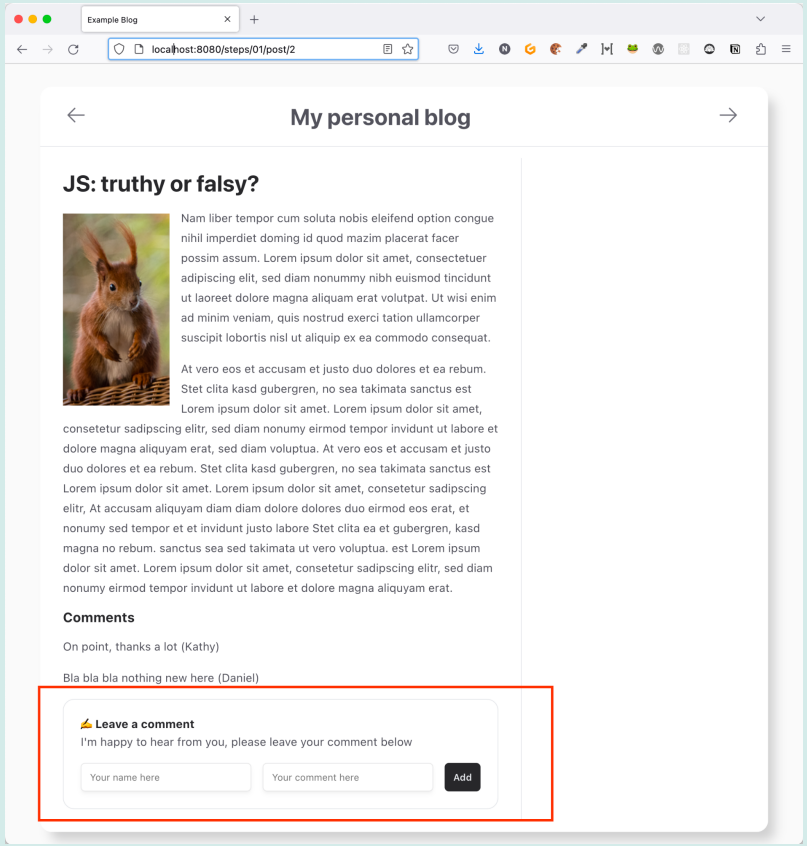
- Validierung geht mit HTML Constraint API so-la-la (kommt auf die Anforderung an)
  - Disablen des Buttons ohne JS nicht möglich
- Nach Absenden des Formulars:
  - Server muss komplette Seite neu zurückschicken
  - Ggf. mit Fehlermeldung
  - Was bedeutet das für das Datenvolumen?



step_03

Exkurs: Formvalidierung

- Validierung der Eingaben muss im Server erfolgen (Sicherheit!)
- Validierung kann im Client erfolgen (bessere UX)
- HTML Constraints API
  - z.B. semantische Felder (email, number) mit eingebauten Constraints
  - JS API für komplexe Validierungen
- HTML Constraint API deckt einige Fälle noch nicht ab




step_03

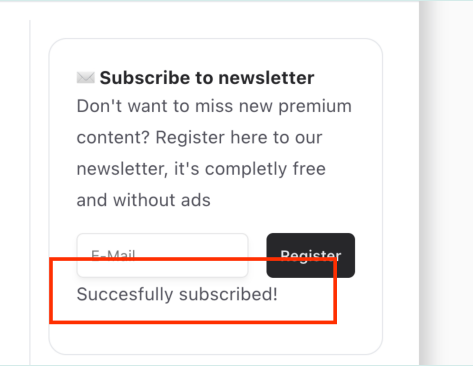
EINE WEB-ANWENDUNG

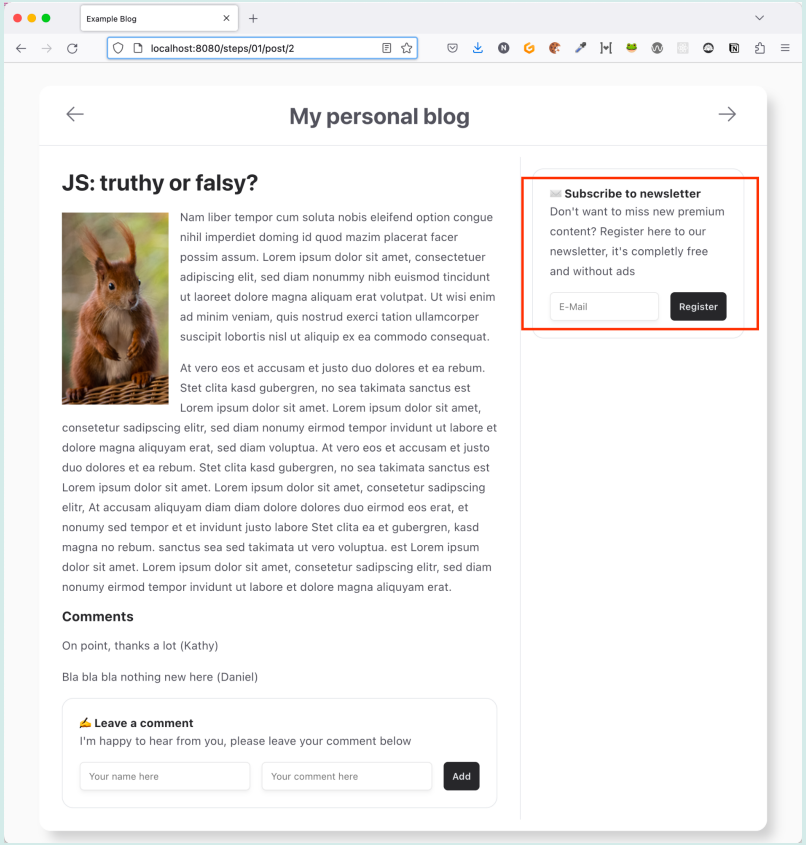
Klassisch, mit JavaScript oder SPA?

- Neu: Es gibt ein 2. Formular ("Newsletter")

Anforderungen:

- Nach dem erfolgreichen Registrieren soll eine Meldung unter dem Textfeld auftauchen ("Vielen Dank für Ihre Registrierung)



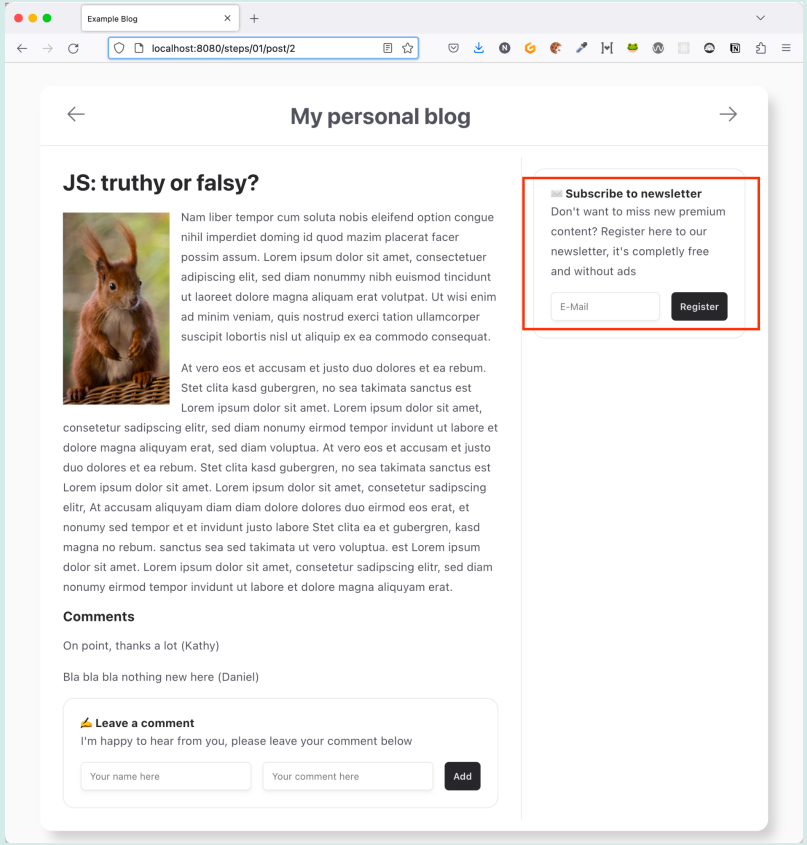


step_03

Klassisch, mit JavaScript oder SPA?

Wohl ohne JS möglich:

- Konsequenzen wie beim 1. Formular
- Backend muss jetzt aber in der Lage sein, für eine Seite mit zwei Formularen zu arbeiten
- Muss wissen, woher die Daten jeweils kommen (URL, POST Body, ...)
- Muss wissen, welche Meldungen wohin gerendert werden müssen



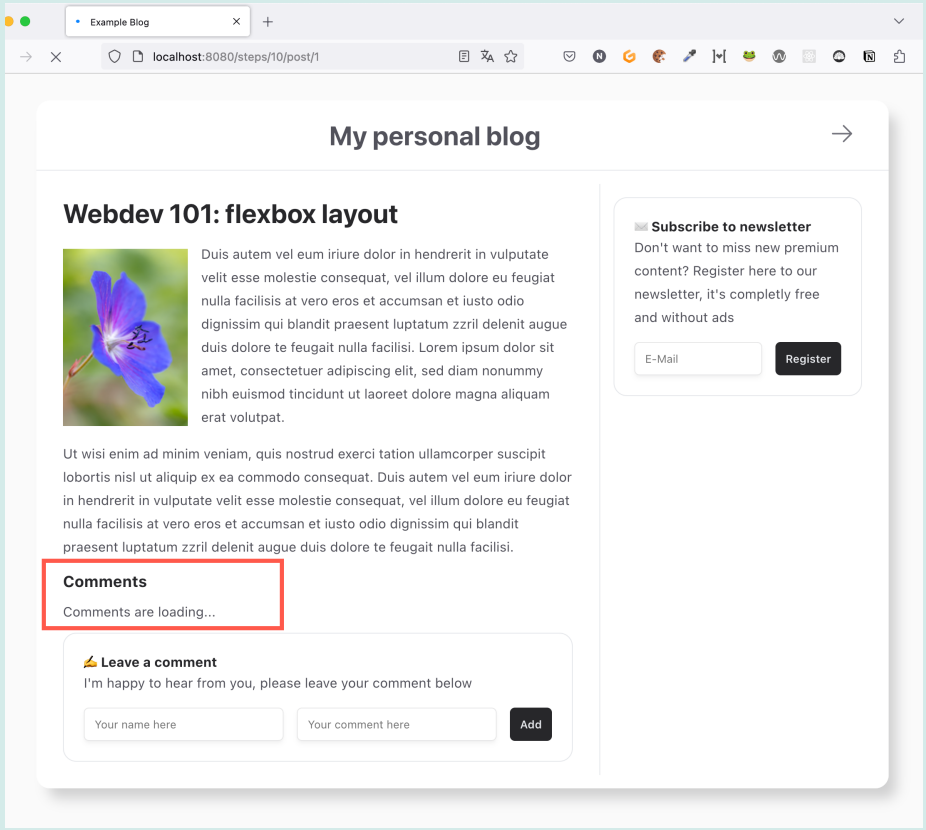
step_03

Klassisch, mit JavaScript oder SPA?

- Der Blog-Post ist das Wichtigste der Seite

Anforderungen:

- Wenn das Ermitteln/Laden der Kommentare lange dauert, soll der Rest der Seite schon angezeigt werden

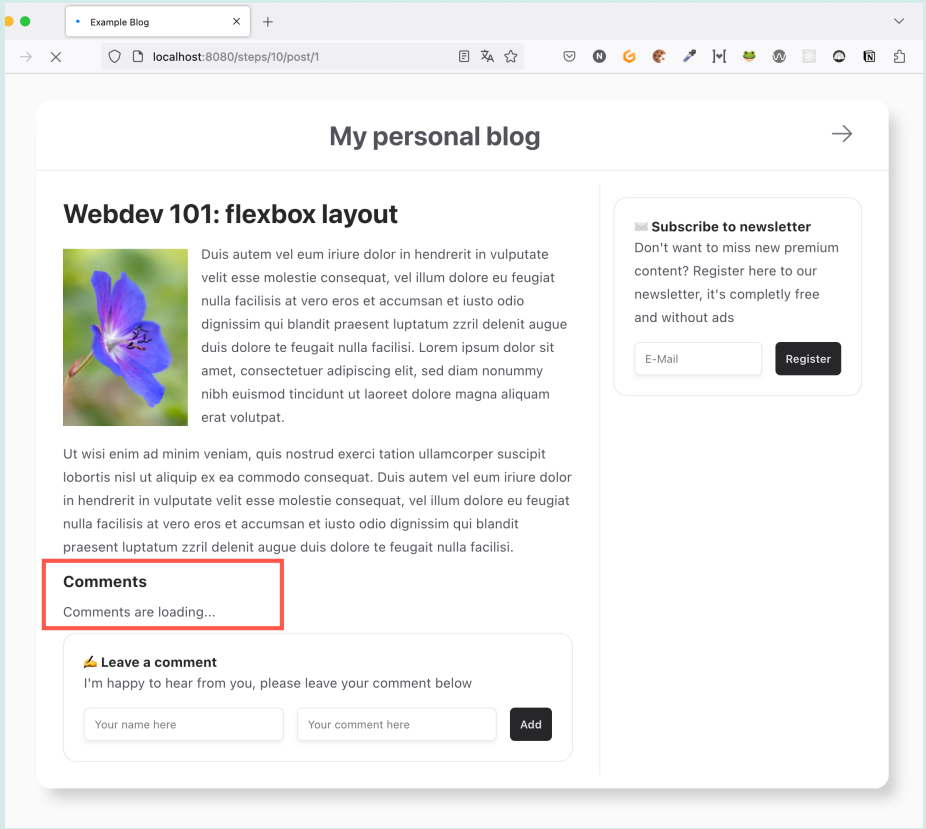


step_10

Klassisch, mit JavaScript oder SPA?

Ohne JS unmöglich (?)

- Browser muss Seite mit Platzhalter/Wartemeldung liefern
- Client muss auf fehlende Teile warten und dann in die Seite integrieren
- HTMX



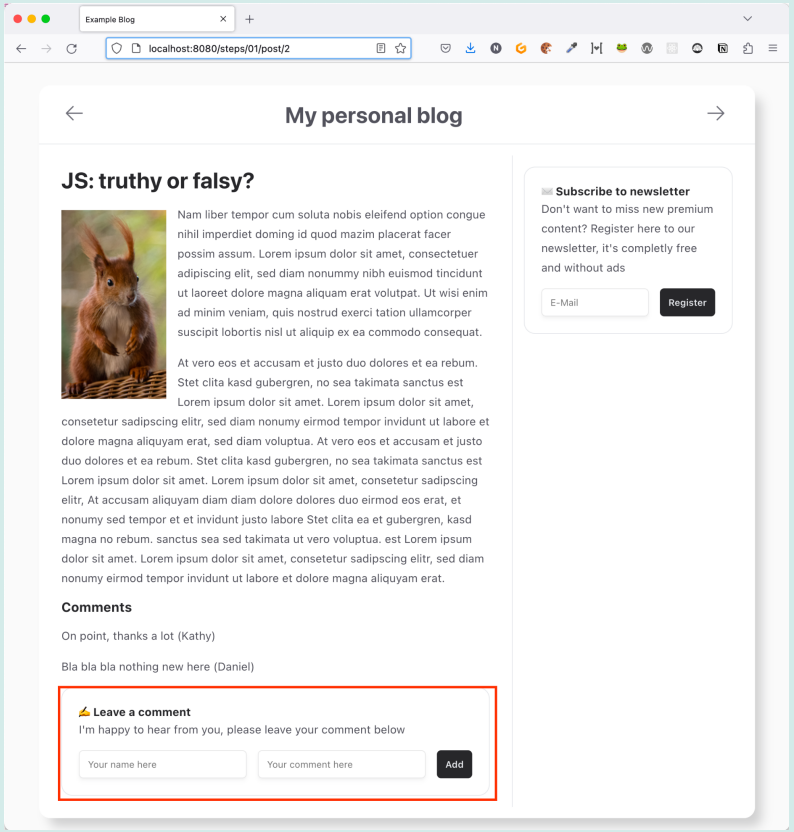
step_10

Klassisch, mit JavaScript oder SPA?

- Das Speichern des Kommentares dauert etwas länger

Anforderungen:

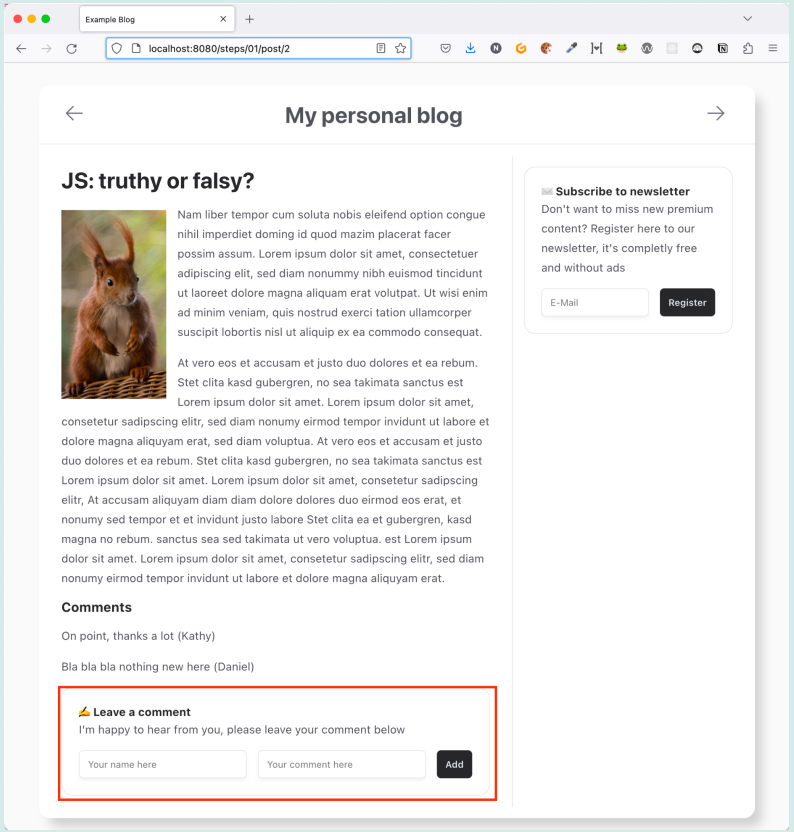
- Während der neue Kommentar gespeichert wird, soll eine Wartemeldung angezeigt werden
- Während der Kommentar gespeichert wird, soll es nicht möglich sein erneut, auf "Add"-Button zu drücken ("Double Submit")



Klassisch, mit JavaScript oder SPA?

Ohne JS nicht möglich:

- Als Reaktion auf den "Add"-Button-Klick muss neben dem Submit-Request auch unmittelbar die Darstellung angepasst werden
- und der Button muss disabled werden



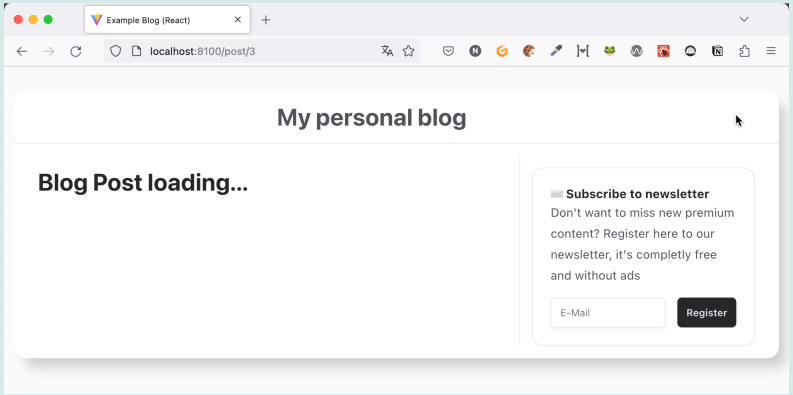
EINE WEB-ANWENDUNG

Klassisch, mit JavaScript oder SPA?

- Lahmes Internet

Anforderungen:

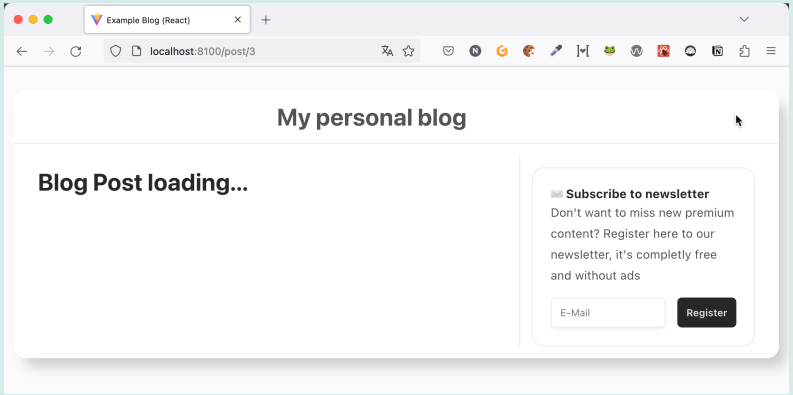
- Beim Klicken auf die Pfeile (vorheriger/nächster Blogpost) soll eine Wartemeldung angezeigt werden, bis der nächste Blog-Post dargestellt werden kann



Klassisch, mit JavaScript oder SPA?

Ohne JS nicht möglich

- Wenn der Request läuft, zeigt der Browser ein Wartesymbol an (wenn überhaupt)
- Laden der Daten muss von der Darstellung entkoppelt werden
- Was bedeutet das für unsere SEO-Anforderung?



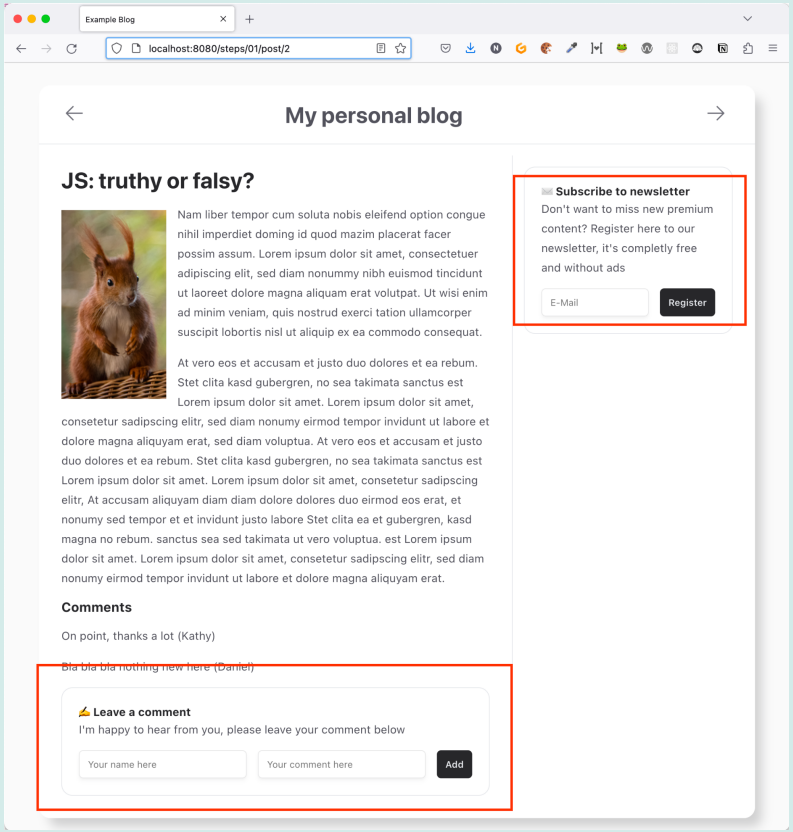
EINE WEB-ANWENDUNG

Klassisch, mit JavaScript oder SPA?

- Zwei Formulare auf einer Seite

Anforderungen

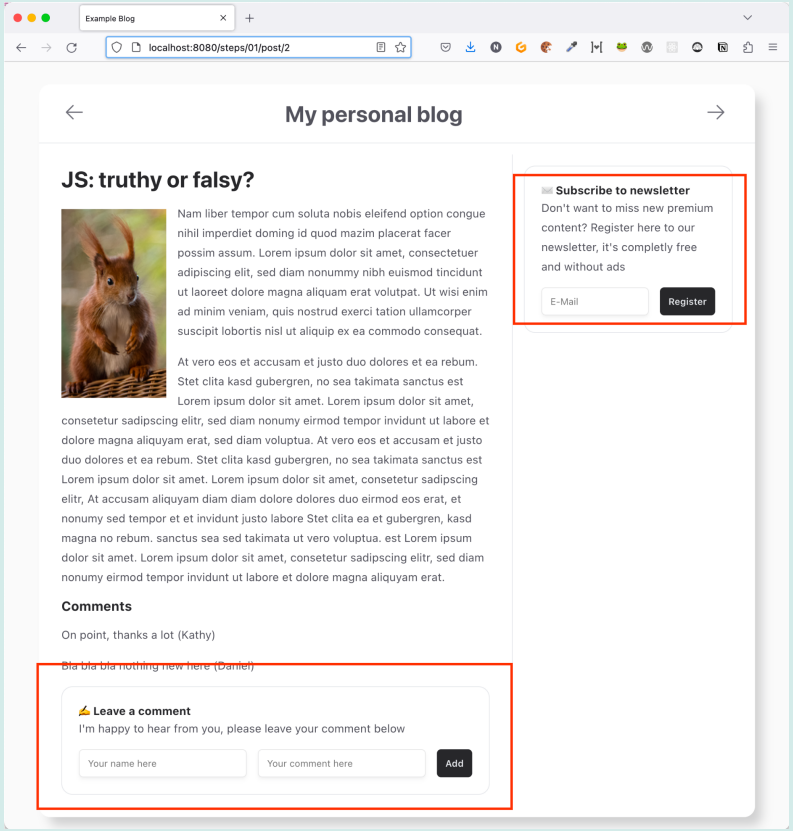
- Wenn auf den "Register"-Knopf gedrückt wird, sollen Eingaben im Kommentar-Formular erhalten bleiben



Klassisch, mit JavaScript oder SPA?

Ohne JS schwierig

- Beide Formulare müssten "eins" sein
- beide Inhalte müssten beim Klick auf den Server übertragen werden
- der Server müsste wissen, was er tun soll (Button "Add" oder Button "Register")



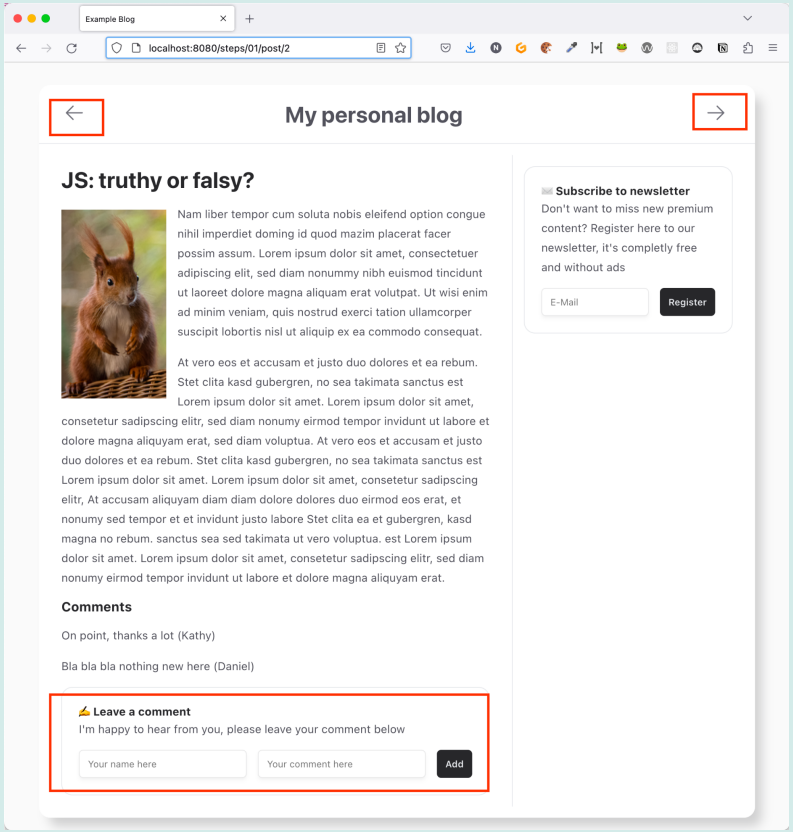
EINE WEB-ANWENDUNG

Klassisch, mit JavaScript oder SPA?

- Kein Datenverlust

Anforderungen

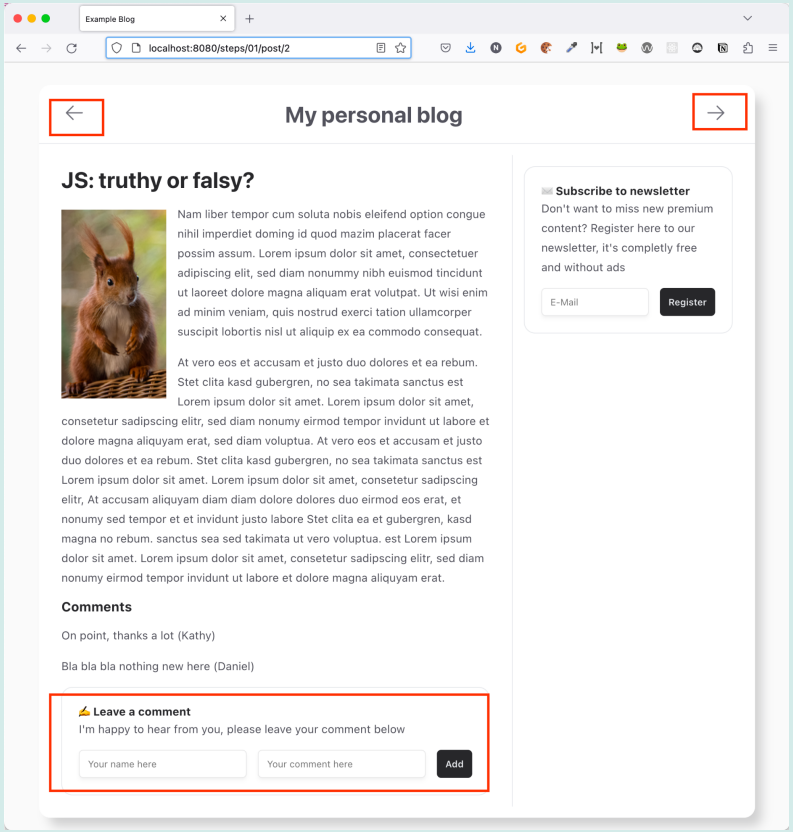
- Beim Navigieren von einer Seite zur nächsten und wieder zurück sollen die Daten im Kommentar-Formular erhalten bleiben
- Auch bei der Verwendung des Back-/Forward-Buttons!



Klassisch, mit JavaScript oder SPA?

Ohne JS unmöglich

- bei Navigation über die Links kommt immer neue Seite vom Server
- Eingaben sind dann weg
- Die Eingaben müssen also irgendwo zwischengespeichert werden (bzw. bei jedem Request mitgesendet werden)
- Klassiker in SPA: Globaler Zustand
- Wie sähe es aus, wenn die Daten sogar über Seiten reload erhalten bleiben sollen?



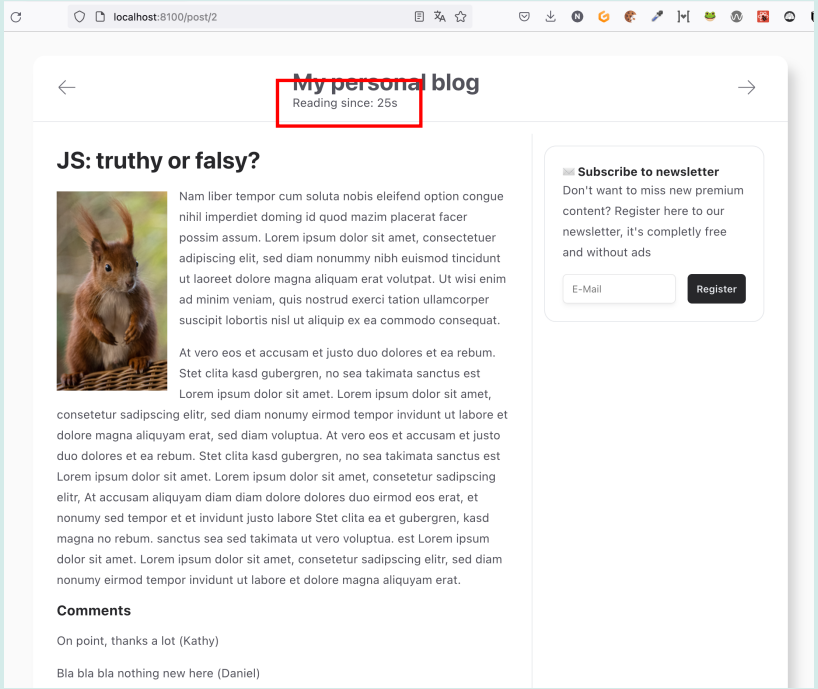
EINE WEB-ANWENDUNG

Klassisch, mit JavaScript oder SPA?

- Im Header wird ein Timer o.ä. dargestellt (z.B. Session Timeout Timer)
- oder ein Video läuft (z.B. mit Werbung)

<u>Anforderungen</u>

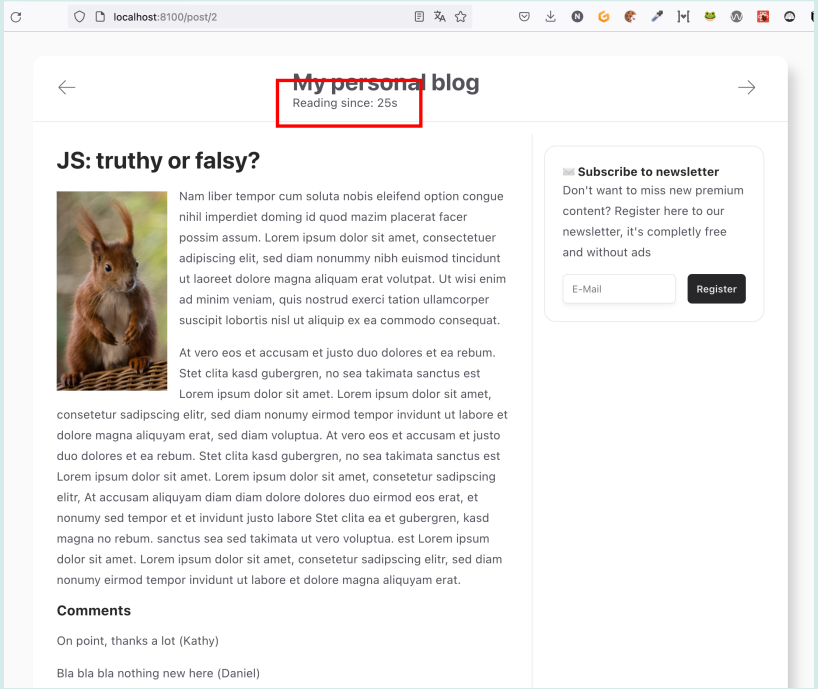
- Beim Navigieren durch die Seiten soll das Video/der Timer ununterbrochen flüssig weiterlaufen



Klassisch, mit JavaScript oder SPA?

Ohne JS unmöglich

- Beim Server Request wird die Seite komplett neu aufgebaut, Video fängt von vorne an
- Countdown/Timer ist ohne JS ohnehin nicht möglich



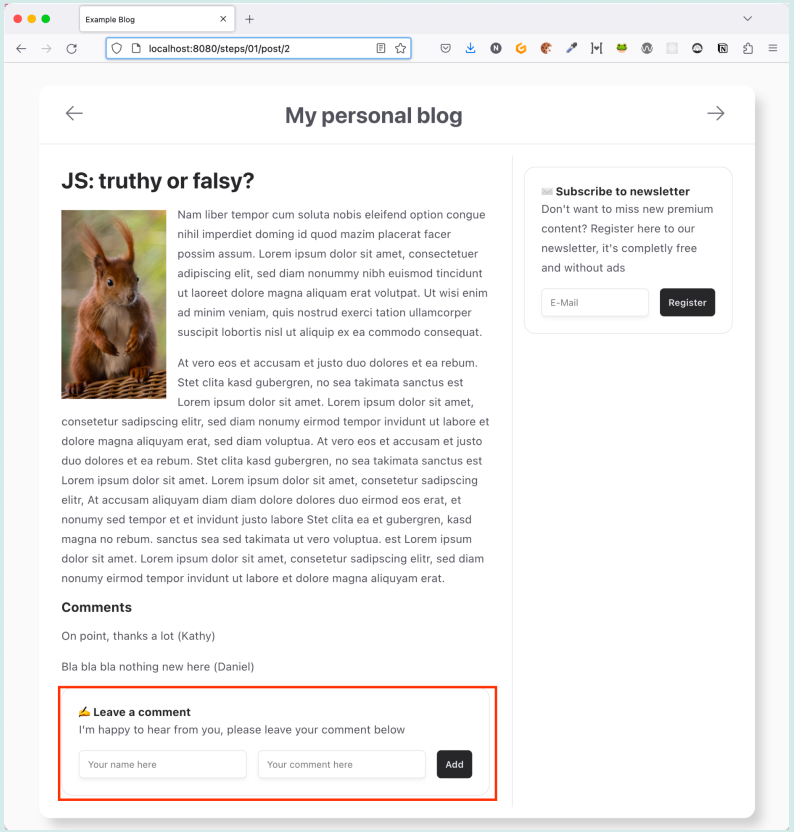
EINE WEB-ANWENDUNG

Klassisch, mit JavaScript oder SPA?

- Der CommentService wird von einem externen Dienstleister betrieben
- Wir brauchen für den Zugriff einen API-Key

<u>Anforderungen:</u>

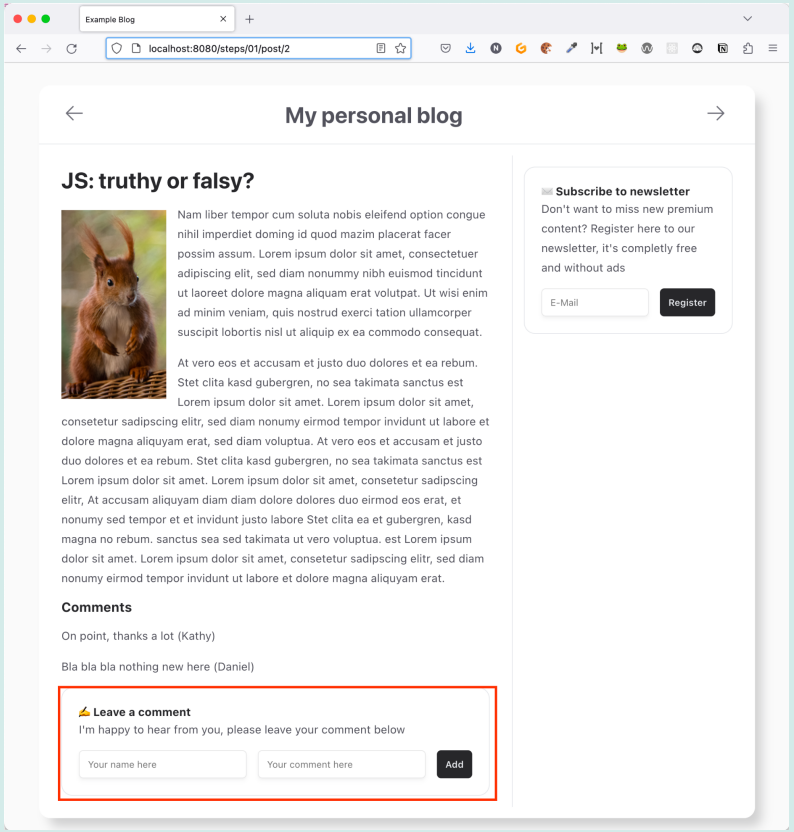
- Der API-Key ist ein Geheimnis und darf nicht für Dritte sichtbar sein



Klassisch, mit JavaScript oder SPA?

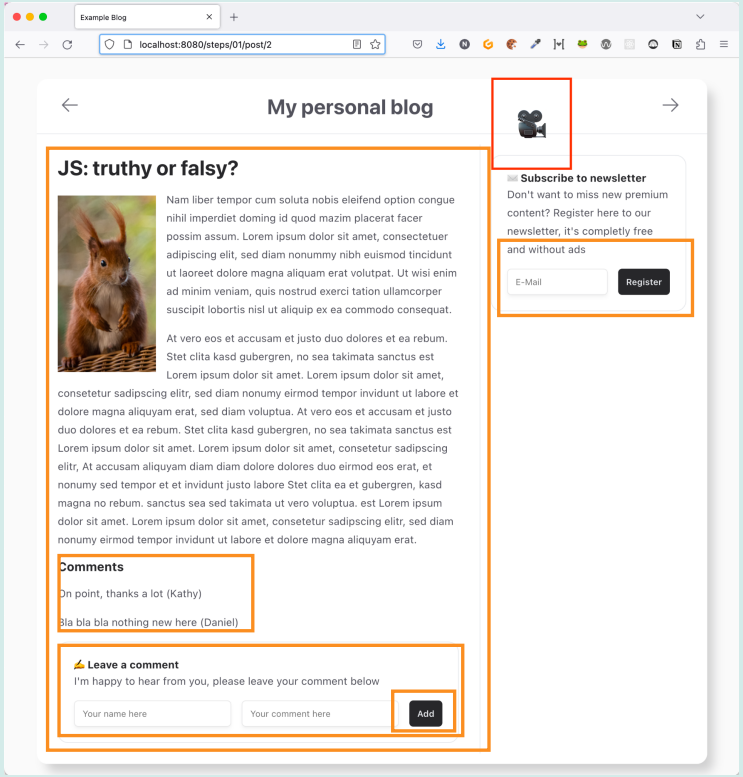
Weder noch:

- "Geheimnisse" wie Keys und Passwörter dürfen nicht in den Client gelangen
- Sofern die Blog-Anwendung API Requests mit JavaScript macht, müsste "unser" Backend dafür einen "Proxy-Dienst" anbieten
- Der Proxy-Dienst nimmt den Request vom Client, fügt den Key hinzu und leitet ihn an den Dritt-Service weiter



Wo haben wir denn jetzt überall JavaScript?

- Formulare: Button disablen, während Request läuft
- Formulare: Warte-Meldung, während Request läuft
- Kommentar-Formular: Eingaben bei Seitenwechsel nicht verlieren
- Kommentar-Formular: Eingaben bei Submit des Newsletter-Formulars nicht verlieren
- (evtl.) Formulare: Validierung
- Kommentare: Warte-Meldung, wenn Laden lange dauert
- Blog-Artikel: Warte-Meldung, wenn Laden lange dauert



Wir brauchen JavaScript

- **In realistischen Anwendungen ist JavaScript unverzichtbar**
  - Vielleicht schneller/früher als man häufig denkt
  - Es gibt Fortschritte in CSS und HTML, die JS an mehr Stellen verzichtbar machen, aber nicht an allen

Wir brauchen JavaScript

- **In realistischen Anwendungen ist JavaScript unverzichtbar**
  - Vielleicht schneller/früher als man häufig denkt
  - Es gibt Fortschritte in CSS und HTML, die JS an mehr Stellen verzichtbar machen, aber nicht an allen
- **Bei "klassischem" Ansatz besteht die Gefahr, dass UI in zwei Teile aufgeteilt wird**
  - "Das bisschen JavaScript wurschteln wir da irgendwie rein"
  - Aufteilung in Backend-Sprache und JavaScript

Ansätze von Web-Anwendungen

Serverseitig gerenderte Anwendungen vs. Single-Page-Anwendungen

Serverseitig gerenderte ("statische") Anwendungen

- Rendern auf dem Server fängt mit jedem Request wieder bei "0" an

Serverseitig gerenderte ("statische") Anwendungen

- **Rendern auf dem Server fängt mit jedem Request wieder bei "0" an**
  - Alle Informationen, die der Server braucht, müssen bei jedem Request mitgegeben werden
    - Inhalt von Eingabefeldern, welche Menüs sind offen etc.
  - Technik: URL, Request-Parameter, Cookies, Headers

Serverseitig gerenderte ("statische") Anwendungen

- **Rendern auf dem Server fängt mit jedem Request wieder bei "0" an**
  - Alle Informationen, die der Server braucht, müssen bei jedem Request mitgegeben werden
    - Inhalt von Eingabefeldern, welche Menüs sind offen etc.
  - Technik: URL, Request-Parameter, Cookies, Headers
  - Im Backend muss dann die ganze Seite zusammengebaut werden
    - Ggf. können Teile aus dem Cache kommen, aber es muss aus Browser-Sicht eine komplette Seite kommen

Single-Page-Anwendungen

- **Man muss JavaScript machen**
  - Man kann auch TypeScript nehmen. JavaScript und TypeScript werden heute synonym verwendet.

Verhalten von Single-Page-Anwendungen

- **Der Browser macht einen Request zum Server ("index.html")**
  - Diese Seite ist in der Regel leer, enthält nur Links auf JavaScript und ggf. CSS

Verhalten von Single-Page-Anwendungen

- **Der Browser macht einen Request zum Server ("index.html")**
  - Diese Seite ist in der Regel leer, enthält nur Links auf JavaScript und ggf. CSS
  - Browser muss nun weitere Requests machen, um JS (und CSS) zu laden

Verhalten von Single-Page-Anwendungen

- **Der Browser macht einen Request zum Server ("index.html")**
  - Diese Seite ist in der Regel leer, enthält nur Links auf JavaScript und ggf. CSS
  - Browser muss nun weitere Requests machen, um JS (und CSS) zu laden
  - Der Code muss über das Internet geladen werden
  - Das JavaScript muss geparst und ausgeführt werden

Verhalten von Single-Page-Anwendungen

- **Der Browser macht einen Request zum Server ("index.html")**
  - Diese Seite ist in der Regel leer, enthält nur Links auf JavaScript und ggf. CSS
  - Browser muss nun weitere Requests machen, um JS (und CSS) zu laden
  - Der Code muss über das Internet geladen werden
  - Das JavaScript muss geparst und ausgeführt werden
  - Erst jetzt ist die Anwendung sichtbar (aber auch interaktiv)

Verhalten von Single-Page-Anwendungen

- **Der Browser macht einen Request zum Server ("index.html")**
  - Diese Seite ist in der Regel leer, enthält nur Links auf JavaScript und ggf. CSS
  - Browser muss nun weitere Requests machen, um JS (und CSS) zu laden
  - Der Code muss über das Internet geladen werden
  - Das JavaScript muss geparst und ausgeführt werden
  - Erst jetzt ist die Anwendung sichtbar (aber auch interaktiv)
  - Jede weitere Navigation findet nur noch im Browser statt (keine Server Round-trips)
    - Deswegen "Single-Page-Anwendung"

Verhalten von Single-Page-Anwendungen

- **Der Browser macht einen Request zum Server ("index.html")**
  - Diese Seite ist in der Regel leer, enthält nur Links auf JavaScript und ggf. CSS
  - Browser muss nun weitere Requests machen, um JS (und CSS) zu laden
  - Der Code muss über das Internet geladen werden
  - Das JavaScript muss geparst und ausgeführt werden
  - Erst jetzt ist die Anwendung sichtbar (aber auch interaktiv)
  - Jede weitere Navigation findet nur noch im Browser statt (keine Server Round-trips)
    - Deswegen "Single-Page-Anwendung"
  - Server ist für die Geschäftslogik und die Bereitstellung der Daten zuständig

Single-Page-Anwendungen

- **Konsequente Trennung von UI und Backend**
  - Klare Architektur, klare Aufgabenverteilung
  - Löst also das "JavaScript-Schnippsel-Problem"

Single-Page-Anwendungen

- **Konsequente Trennung von UI und Backend**
  - Klare Architektur, klare Aufgabenverteilung
  - Löst also das "JavaScript-Schnippsel-Problem"
- **Schnitt der Anwendung muss nicht den Backend-Services entsprechen**
  - Oftmals ist eine UI "grober" oder einfach nur anders geschnitten als Services im Backend
  - Sachbearbeiter:in möchte z.B. Kundendaten, vorherige Einkäufe und ... in einer Ansicht sehen
  - Leser:in möchte Blog-Artikel und Kommentare und Tipps für weitere Artikel in einer Ansicht sehen

Single-Page-Anwendungen

- **Zustand ("UI State") befindet sich im Client**
  - Daten im Formular
  - Kommentar-Liste auf- oder zugeklappt?

Single-Page-Anwendungen

- **Zustand ("UI State") befindet sich im Client**
  - Daten im Formular
  - Kommentar-Liste auf- oder zugeklappt?
- **Die Daten für eine Seite könnten gezielt per API Request(s) geladen werden**

Single-Page-Anwendungen

- **Zustand ("UI State") befindet sich im Client**
  - Daten im Formular
  - Kommentar-Liste auf- oder zugeklappt?
- **Die Daten für eine Seite könnten gezielt per API Request(s) geladen werden**
- **Teile von Seiten werden neu gerendert**
  - Die Darstellung fängt nicht bei jeder Interaktion wieder bei "0" an

Probleme von Single-Page-Anwendungen

Probleme von Single-Page-Anwendungen

- Was fällt euch ein? 🤔

Probleme von Single-Page-Anwendungen

- Was fällt euch ein? 🤔
- Gängige Vorurteile? 🤬

Probleme von Single-Page-Anwendungen

- SEO geht nicht / schlecht!
- Langsam!
- Viel JavaScript muss geladen werden!
- JS-Bibliotheken ändern sich ständig!
- Tooling funktioniert nicht bzw. ist komplex!

Performance in Single-Page-Anwendungen

1. Es gibt die Zeit, die es braucht, bis eine Anwendung fertig dargestellt und interaktiv wird

Performance in Single-Page-Anwendungen

1. Es gibt die Zeit, die es braucht, bis eine Anwendung fertig dargestellt und interaktiv wird
2. Es gibt die Performance zur Laufzeit (Öffnen eines Menüs oder Drag'n'Drop funktioniert flüssig)

Performance in Single-Page-Anwendungen

1. Es gibt die Zeit, die es braucht, bis eine Anwendung fertig dargestellt und interaktiv wird
2. Es gibt die Performance zur Laufzeit (Öffnen eines Menüs oder Drag'n'Drop funktioniert flüssig)
3. Performance beim Austausch mit Server-Daten (z.B. REST API)

Performance in Single-Page-Anwendungen

1. Es gibt die Zeit, die es braucht, bis eine Anwendung fertig dargestellt und interaktiv wird
2. Es gibt die Performance zur Laufzeit (Öffnen eines Menüs oder Drag'n'Drop funktioniert flüssig)
3. Performance beim Austausch mit Server-Daten (z.B. REST API)

👉 In der Diskussion um Fullstack-Ansätze geht es meist um den ersten Punkt

Teil 2

Fullstack

Ansätze

Fullstack-Ansätze

- **Im Client nur dort JS, wo technisch notwendig**
  - wie beim JS-Schnippsel-Ansatz!

Fullstack-Ansätze

- **Im Client nur dort JS, wo technisch notwendig**
  - wie beim JS-Schnippsel-Ansatz!
- **...aber Entwicklung aus einem Guss / Stack**
  - wie beim Single-Page-Ansatz

Fullstack-Ansätze

- **Im Client nur dort JS, wo technisch notwendig**
  - wie beim JS-Schnippsel-Ansatz!
- **...aber Entwicklung aus einem Guss / Stack**
  - wie beim Single-Page-Ansatz
- **Das Beste aus beiden Welten?**

Fullstack-Ansätze

- **Fullstack JavaScript-Frameworks**
  - Insbesondere in React gerade heftig diskutiert (Next.js, Remix)
  - Vue (Nuxt)
  - SvelteKit (Svelte)

Fullstack-Ansätze

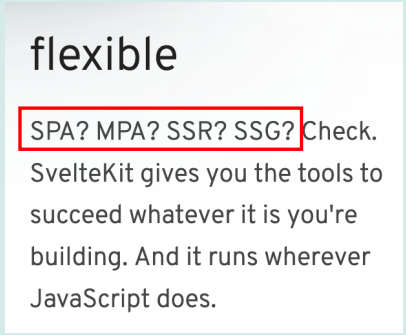
- **Fullstack JavaScript-Frameworks**
  - Insbesondere in React gerade heftig diskutiert (Next.js, Remix)
  - Vue (Nuxt)
  - SvelteKit (Svelte)
- **Astro, Qwik**

Fullstack-Ansätze

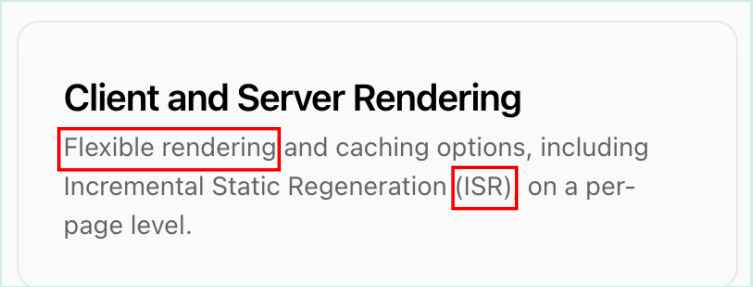
- **Fullstack JavaScript-Frameworks**
  - Insbesondere in React gerade heftig diskutiert (Next.js, Remix)
  - Vue (Nuxt)
  - SvelteKit (Svelte)
- **Astro, Qwik**
- **(HTMX)**
  - Kein Fullstack-Framework aber evtl. interessant für kleinere Anwendungen

Konzepte

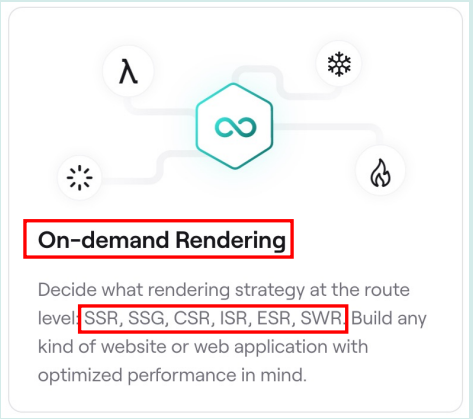
- Es gibt eine Reihe von Ideen, JS-basierte Anwendungen schneller in den Browser zu bringen....
- Umsetzung unterscheidet sich je nach Framework



https://kit.svelte.dev/



https://nextjs.org



https://nuxt.com

Begriffe

- **Multi-Page-Application (MPA)**
  - "Traditionelle" Web-Anwendung, in der der Client pro Route einen Request zum Server macht

Begriffe

- **Serversite Rendering (SSR)**
  - Seite einer JS-Anwendung wird auf dem Server gerendert
  - Fertiger HTML-Code wird zum Client geschickt

Begriffe

- **Serversite Rendering (SSR)**
  - Seite einer JS-Anwendung wird auf dem Server gerendert
  - Fertiger HTML-Code wird zum Client geschickt
- **Hydration**
  - Serverseitig gerenderte HTML-Seite wird auf dem Client durch Hinzufügen und Ausführen von JavaScript interaktiv gemacht. Erst danach ist die Anwendung interaktiv.

Begriffe

- **Serversite Rendering (SSR)**
  - Seite einer JS-Anwendung wird auf dem Server gerendert
  - Fertiger HTML-Code wird zum Client geschickt
- **Hydration**
  - Serverseitig gerenderte HTML-Seite wird auf dem Client durch Hinzufügen und Ausführen von JavaScript interaktiv gemacht. Erst danach ist die Anwendung interaktiv.
- **Partial, selective, oder progressive hydration:**
  - Es werden nur Teile des HTML-Codes interaktiv gemacht, bzw. Schritt-für-Schritt (z.B. mit Suspense in React)

Begriffe

- **Progressive Enhancement:**
  - Die Anwendung funktioniert grundsätzlich ohne JavaScript
  - JavaScript wird nur verwendet, um die Bedienbarkeit noch zu verbessern
  - Beispiel:
    - Formular kann ausgefüllt und abgeschickt werden (ohne JS)
    - Validiert wird aber nur mit JS
  - Motivation von PE: Man kann die Anwendung schon bedienen, auch wenn der JS-Code noch nicht geladen wurde

Begriffe: Render-Verhalten

- **Static Site Generation (SSG)**
  - Die Seiten der kompletten Anwendung werden schon zur Buildzeit generiert. Zum Beispiel auf Basis von Daten aus einem CMS
  - Ergebnis muss aber nicht zwangsläufig HTML sein.

Begriffe: Render-Verhalten

- **Static Site Generation (SSG)**
  - Die Seiten der kompletten Anwendung werden schon zur Buildzeit generiert. Zum Beispiel auf Basis von Daten aus einem CMS
  - Ergebnis muss aber nicht zwangsläufig HTML sein.
- **Incremental Static Regeneration (ISG)**
  - Einzelne Seiten der Anwendung werden zur Laufzeit neu generiert (z.B. wenn ein Artikel im CMS aktualisiert wurde)

Begriffe: Render-Verhalten

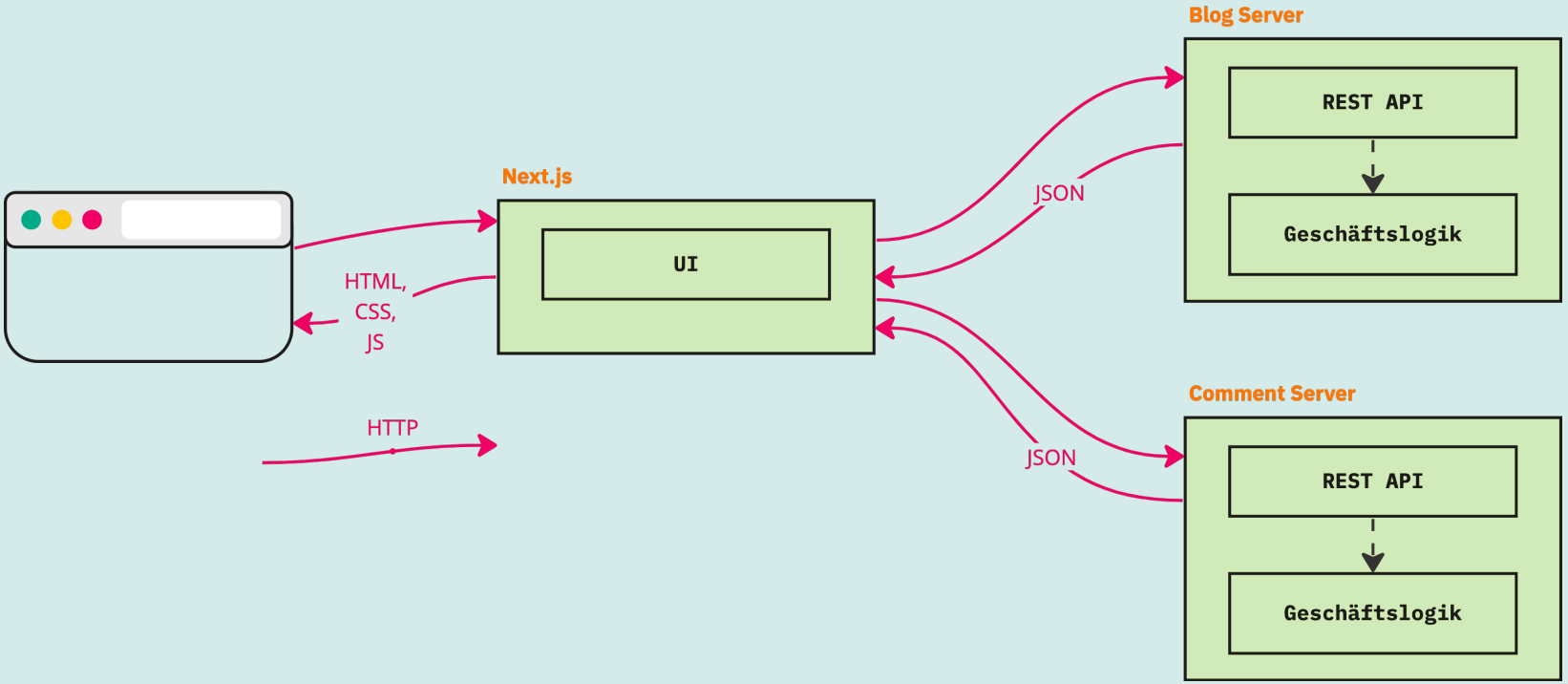
- **Static Site Generation (SSG)**
  - Die Seiten der kompletten Anwendung werden schon zur Buildzeit generiert. Zum Beispiel auf Basis von Daten aus einem CMS
  - Ergebnis muss aber nicht zwangsläufig HTML sein.
- **Incremental Static Regeneration (ISG)**
  - Einzelne Seiten der Anwendung werden zur Laufzeit neu generiert (z.B. wenn ein Artikel im CMS aktualisiert wurde)
- **Edge-Side Rendering (ESR)**
  - Der Server für das Frontend läuft in einer "Edge-Funktion" im CDN (z.B. bei Cloudflare, Vercel oder Netlify)
  - Da die Funktionen "nahe" beim Client laufen, soll die Latenz gering sein

Fullstack Web-Anwendungen am Beispiel Next.js

Next.js von Vercel (https://nextjs.org)

- **Basiert auf React**
- **Statisches und dynamisches Rendering**
- **Caching**
- **Enthält Build-Tooling und Server**
- **Neuste Features mit dem "App-Router" (ab Version 13.4)**
  - React Server Components (RSC)
  - Suspsene
  - entspricht der React "Architekturvision"

Deployment: "Backend for Frontend"



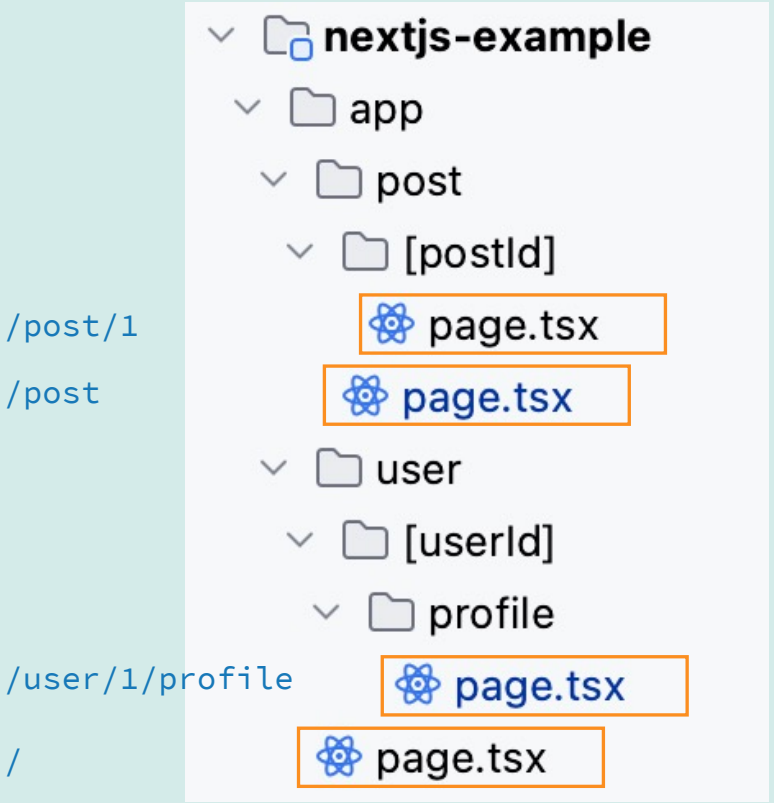
Routing

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

Routen in Next.JS

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



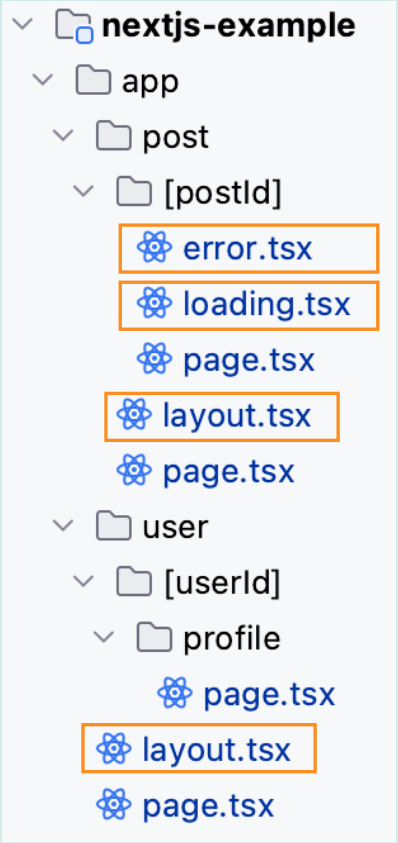
Routen in Next.JS

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

```tsx
type PostPageProps = {
  params: { postId: string }
}

export default function PostPage( {params}: PostPageProps ) {
  return <h1>Hallo Post {params.postId}!</h1>
}
```

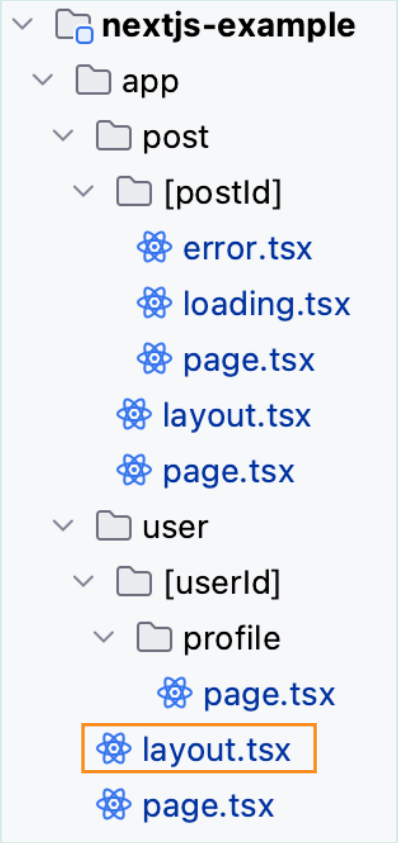
Routen in Next.JS

- 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
- Wenn eine Route keine Layout-Komponente hat, wird im Baum oberhalb nach der nächstgelegenen Layout-Komponente gesucht
- Layout-Komponenten können verschachtelt sein
- So können gemeinsame Layouts für ganze Bereiche der Anwendung gebaut werden



React

Server

Components

SERVER COMPONENTS

Idee: Es gibt Komponenten, die nicht im Client ausgeführt werden

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

Im Browser ist trotzdem JavaScript erforderlich...

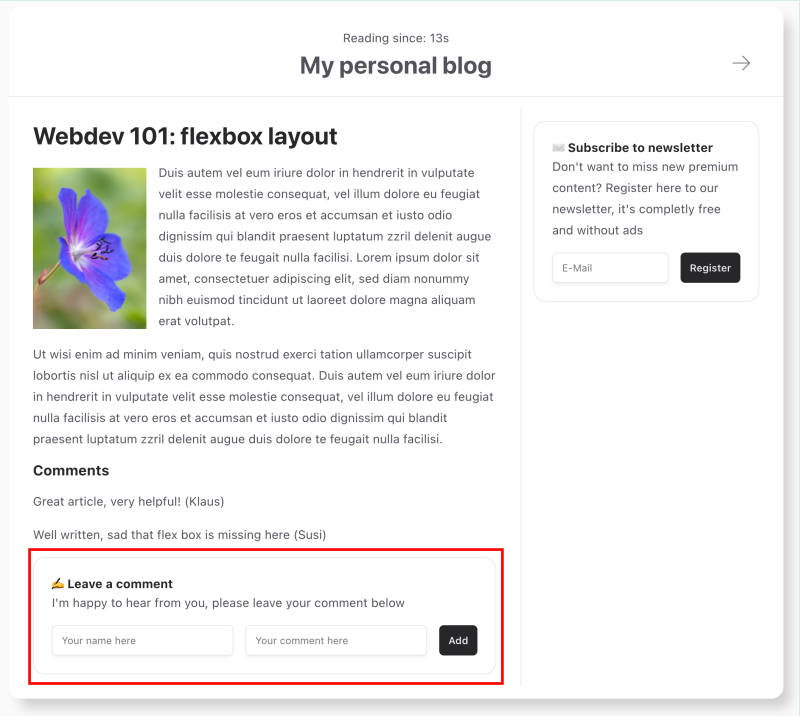
- aber "nur" für das Framework (Next.js)
- und die interaktiven Teile der Anwendung
- der benötigte JavaScript-Code wächst nicht mehr mit jedem Feature

Arten von Komponenten

ARTEN VON KOMPONENTEN

Client-Komponenten (wie bisher)

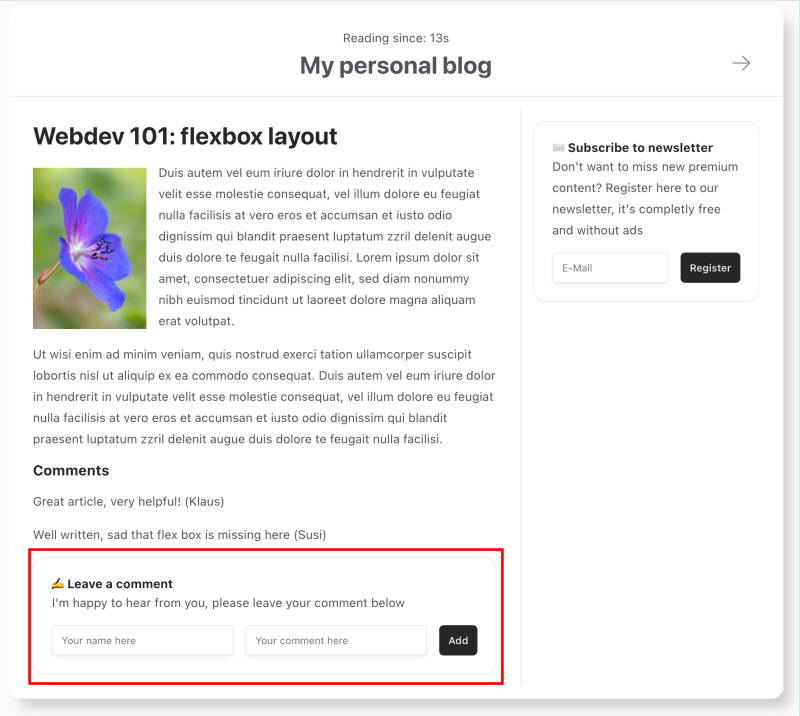
- Werden auf dem Client gerendert



ARTEN VON KOMPONENTEN

Client-Komponenten (wie bisher)

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

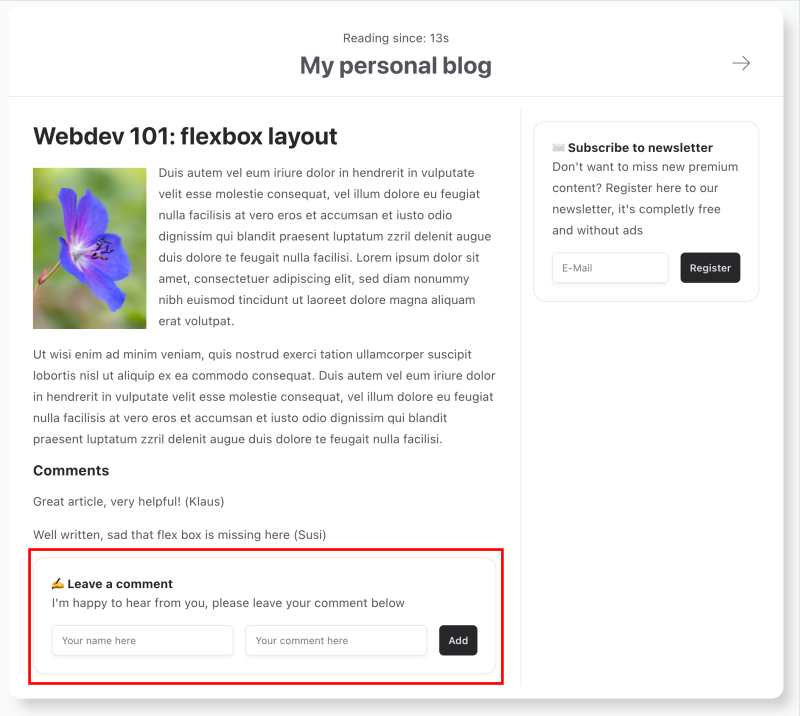


ARTEN VON KOMPONENTEN

Client-Komponenten (wie bisher)

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

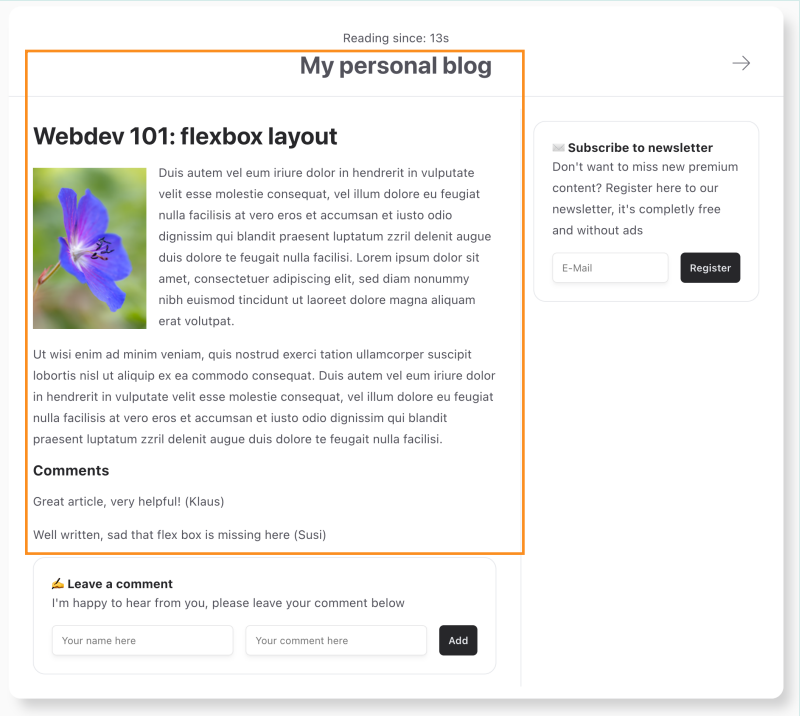
- JavaScript-Code immer zum Client gesendet
- Können deshalb interaktiv sein
- "Schippsel" aus dem klassischen Ansatz



ARTEN VON KOMPONENTEN

Neu: Server-Komponenten

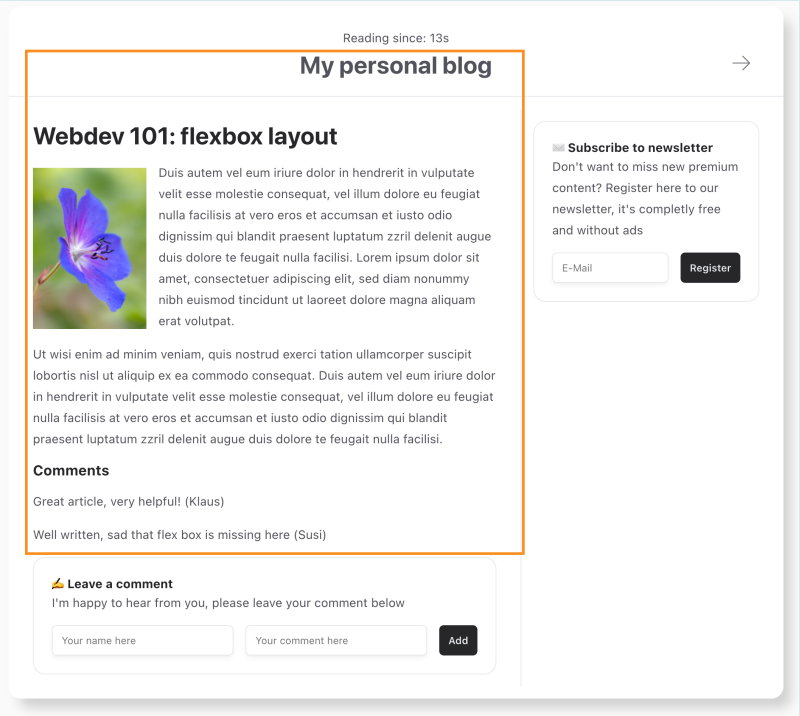
- werden auf dem Server gerendert



ARTEN VON KOMPONENTEN

Neu: Server-Komponenten

- werden auf dem Server gerendert
- oder im Build 🙄



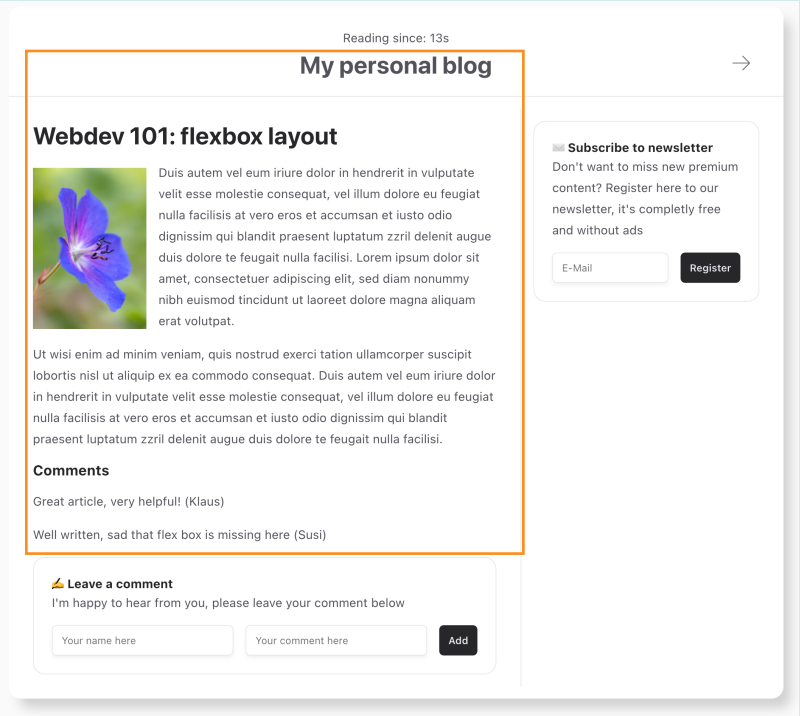
ARTEN VON KOMPONENTEN

Neu: Server-Komponenten

- werden auf dem Server gerendert
- oder im Build 🙄

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

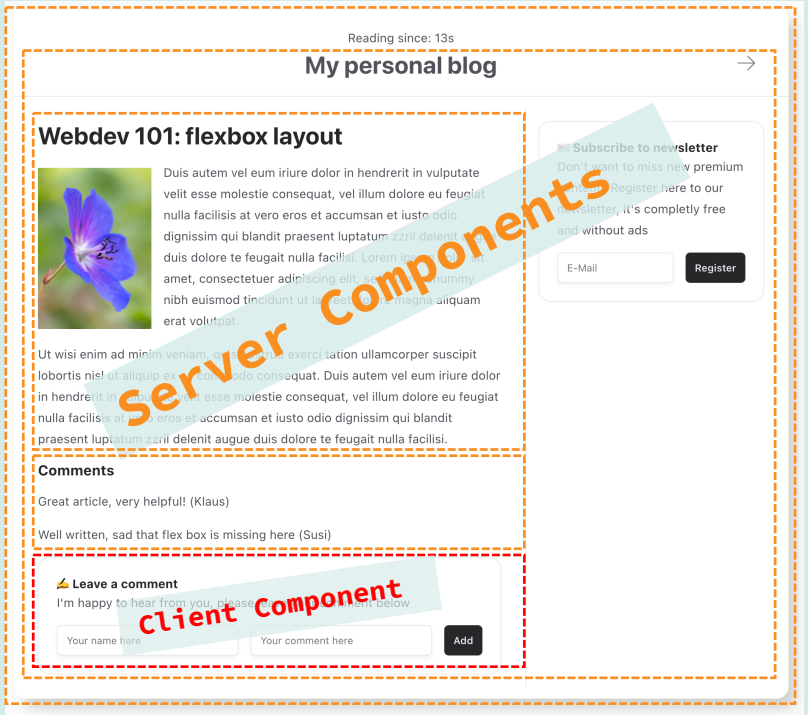
- API: "normale" React-Komponenten



ARTEN VON KOMPONENTEN

Weiterhin ein Komponenten-Baum

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



ARTEN VON KOMPONENTEN

RSC: Komponenten, die auf dem Server ausgeführt werden

JS-Code kommt nicht in den Browser

Auch nicht 3rd-Party-Code

```
import moment from "moment";
import marked from "marked";

export default function Post({ post }) {

  const date = moment(post.date).format("DD.MM.YYYY");
  const body = marked.parse(post.body);

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

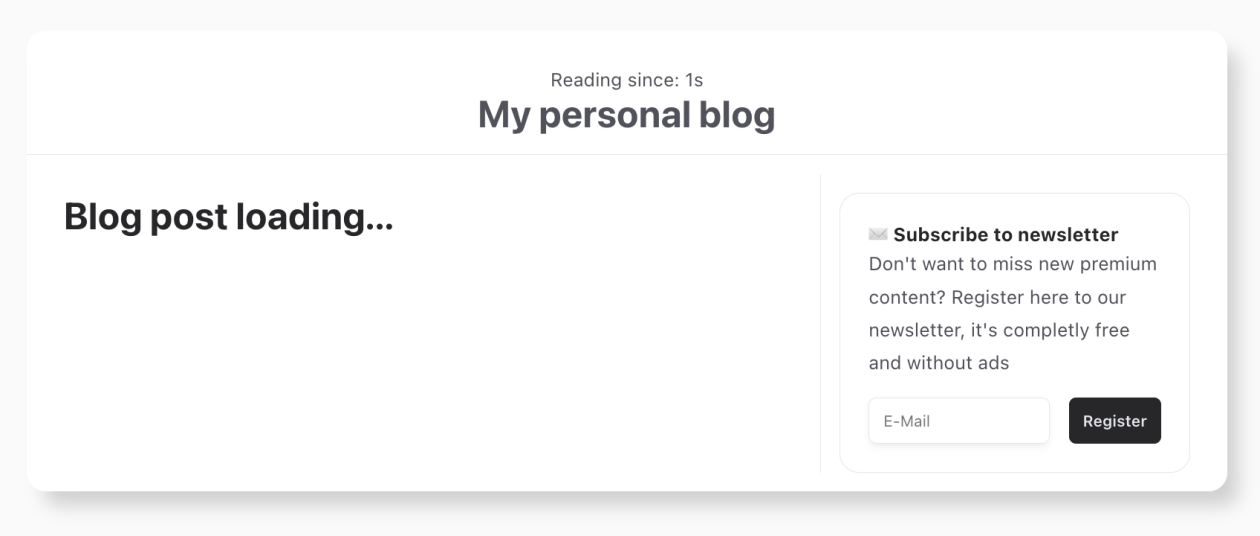
Data Fetching

ARBEITEN MIT SERVERSEITIGEN DATEN

Asynchrone Komponenten: Daten können in der Komponente geladen werden

```
export default async function BlogPostPage({ params }: BlogPostPageProps) {
  const postId = params.postId;

  const commentsPromise = loadComments(postId);

  const { post } = await loadBlogPost(postId);

  return (
    <div>
      <h1>{post.title}</h1>
      <img src={`http://localhost:8080/${post.image}`}  alt={""}/>
      <span dangerouslySetInnerHTML={{ __html: post.body }} />

      <Comments commentsPromise={commentsPromise} />

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

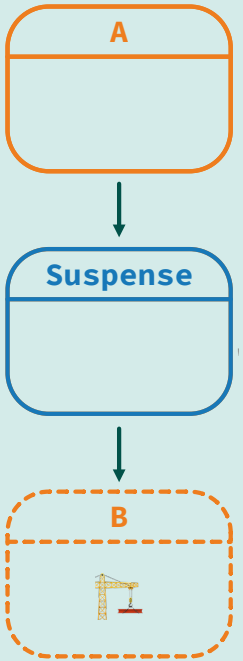
Anforderung: Beim Laden der Daten soll Hinweis ausgegeben werden



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

SUSPENSE

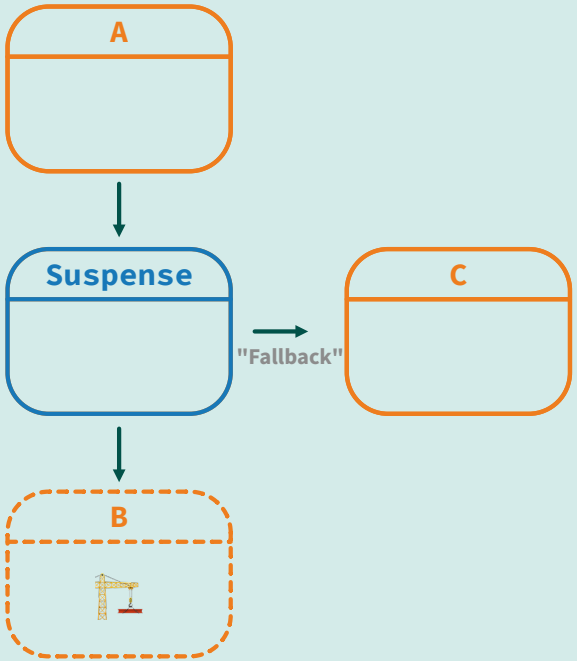
Suspense: Unterbricht das Rendern, solange "etwas" fehlt



SUSPENSE

Suspense: Unterbricht das Rendern, solange "etwas" fehlt

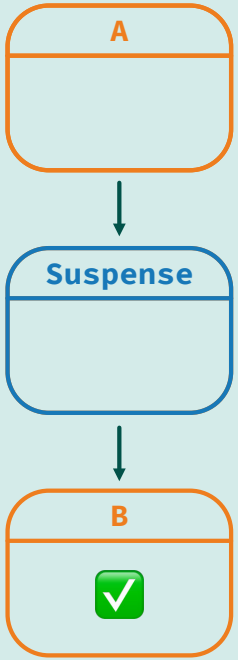
Während der Wartezeit wird im Client eine "Fallback"-Komponente angezeigt



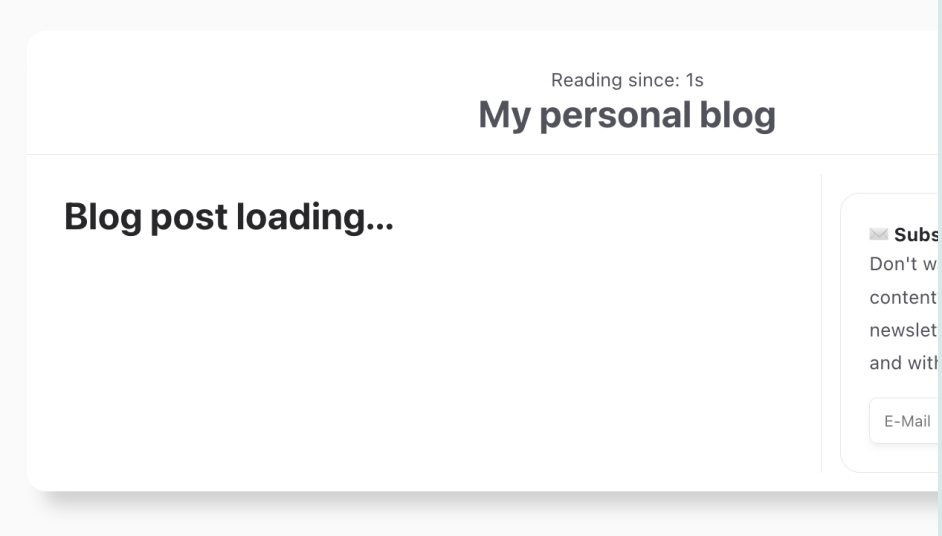
SUSPENSE

Suspense: Unterbricht das Rendern, solange "etwas" fehlt

Sobald die Komponente gerendert werden konnte, wird sie auf den Client übertragen



Suspense auf Routen-Ebene

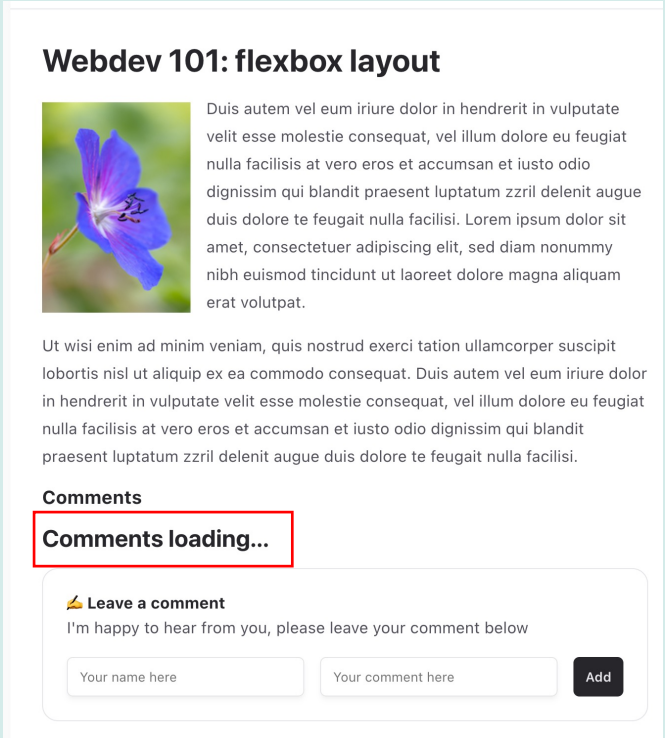



```
export default function Loading() {
  return (
    <div>
      <h1>Blog post loading...</h1>
    </div>
  );
}
```

- Wenn auf die Daten einer Route gewartet werden soll, wird eine loading.tsx-Datei angelegt
- Diese exportiert die Platzhalter-Komponente

SUSPENSE

Priorisierung: Teile der Seite können nachgeliefert werden



Priorisierung

- Requests können parallel gestartet werden (vermeidet "Wasserfälle)
- An die Unterkomponenten wird das Promise-Objekt übergeben

```
async function BlogPostPage({ params }: any) {
  const postId = params.postId;

  const commentsPromise = loadComments(postId);

  const { post } = await loadBlogPost(postId);

  return (
    <div>
      <>
        <h1>{post.title}</h1>
        <img src={`http://localhost:8080/${post.image}`} alt={""} />
        <span dangerouslySetInnerHTML={{ __html: post.body }} />
      </>

      <Comments commentsPromise={commentsPromise} />
    </div>
  );
}
```

SUSPENSE

Priorisierung

- Mit Suspense-Komponente kann das Rendern an beliebiger Stelle unterbrochen werden
- Der Client bekommt dann alles, was bis hierher fertig gerendert ist
- und die Fallback-Komponente
- der Rest wird "nachgereicht"

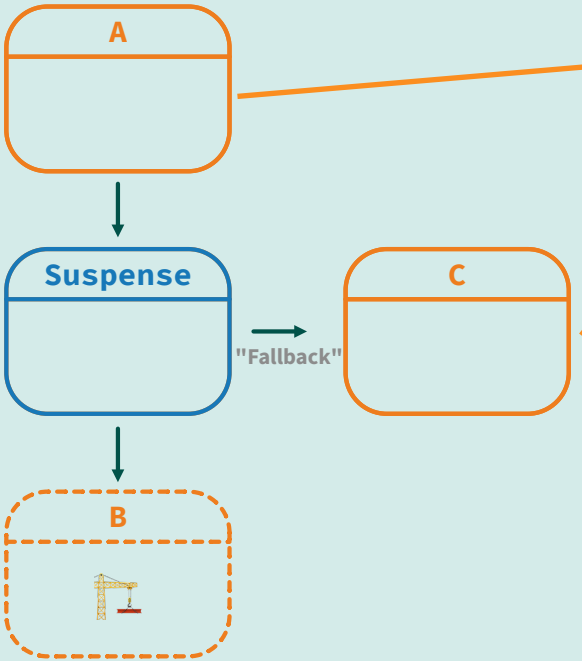
```
export default function Comments({ commentsPromise }: any) {
  return (
    <section className="Comments">
      <h3>Comments</h3>

      <Suspense fallback={<h1>Comments loading...</h1>}>
        <CommentList commentsPromise={commentsPromise} />
      </Suspense>
    </section>
  );
}
```

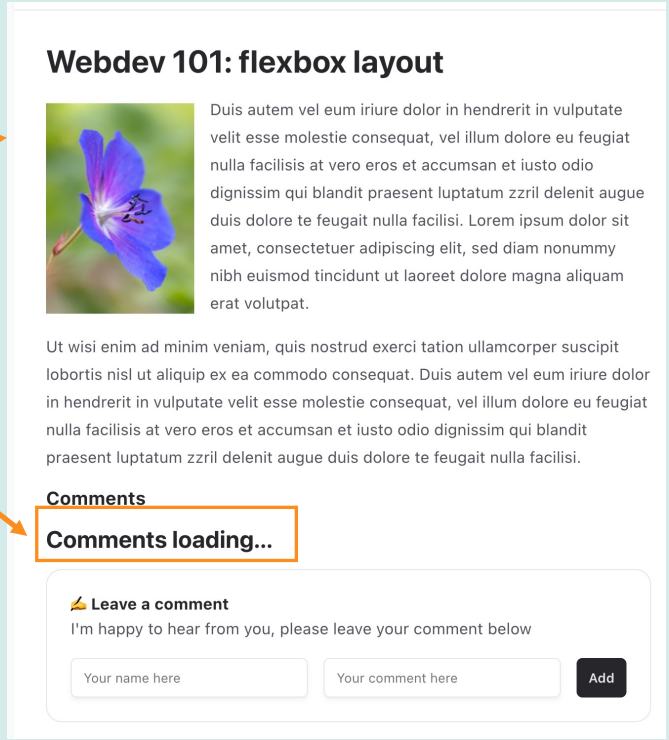
Priorisierung

- Die Kommentar-Liste wartet auf die Daten
- Währenddessen wird ggf. die Fallback-Komponente angezeigt

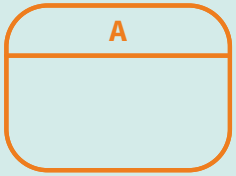
```
async function CommentList({ commentsPromise }: CommentListProps) {

  const comments = await commentsPromise;

  return (
    <>
      {comments.map((c) => (
        <p key={c.id}>
          {c.comment} ({c.name})
        </p>
      ))}
    </>
  );
}
```

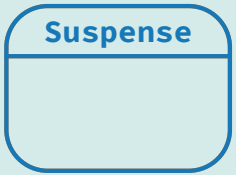
Suspense: Bis zur "Sollbruch-Stelle" werden die Daten an den Browser geschickt

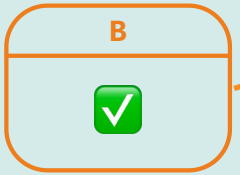


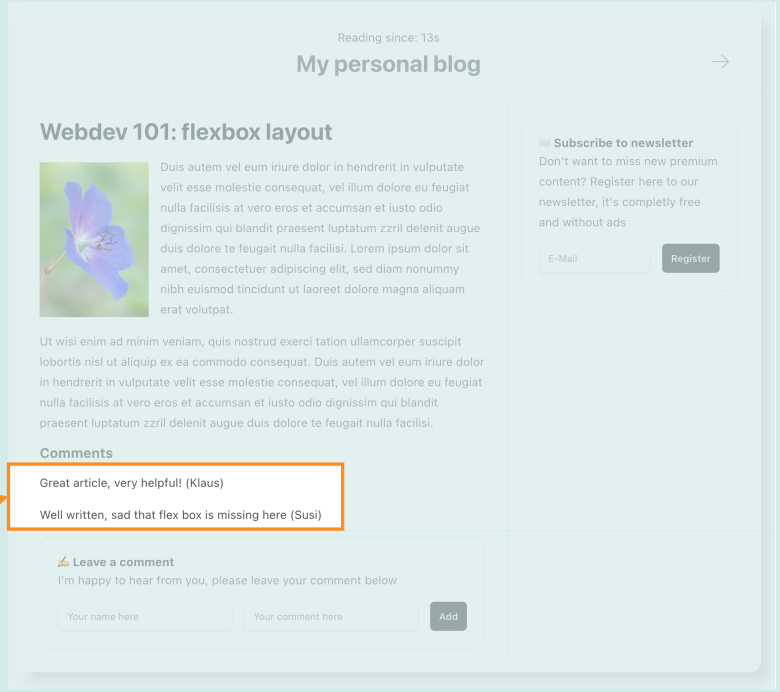


Suspense: Die fehlenden Daten werden nachgeliefert ("Streaming")

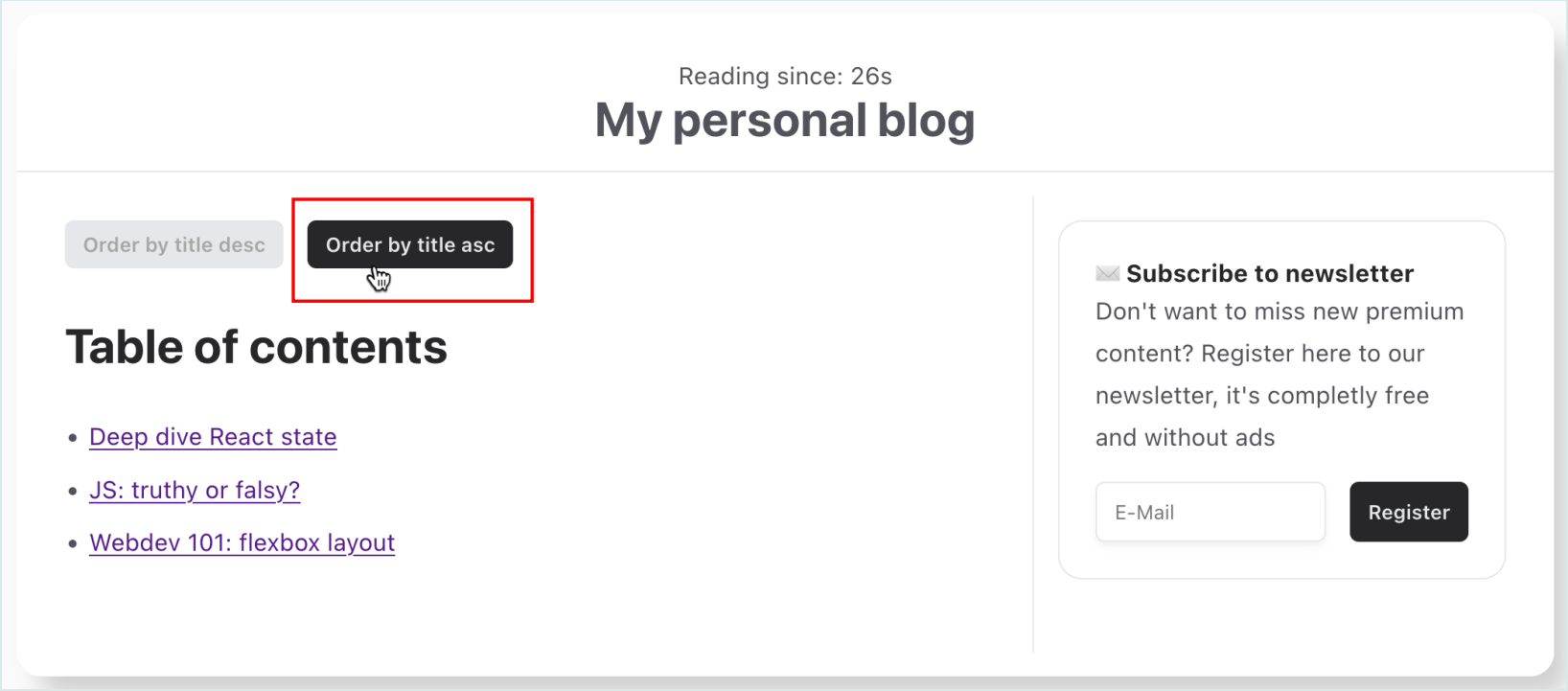








Aufteilung in Server-Client: Konsequenzen



SORTIERUNG DES INHALTSVERZEICHNISSES

```tsx
type Props = {
  posts: Post[];
  onSortOrderChange(newOrder: "asc" | "desc"): void;
};

export function ArticleList({ posts, onSortOrderChange }: Props) {
  return (
    <div>
      {posts.map((c) => (
        <p key={c.id}>
          <Link href={"..."}>{c.title}</Link>
        </p>
      ))}

      <button onClick={() => onSortOrderChange("asc")}>Asc</button>

      <button onClick={() => onSortOrderChange("desc")}>Desc</button>
    </div>
  );
}
```

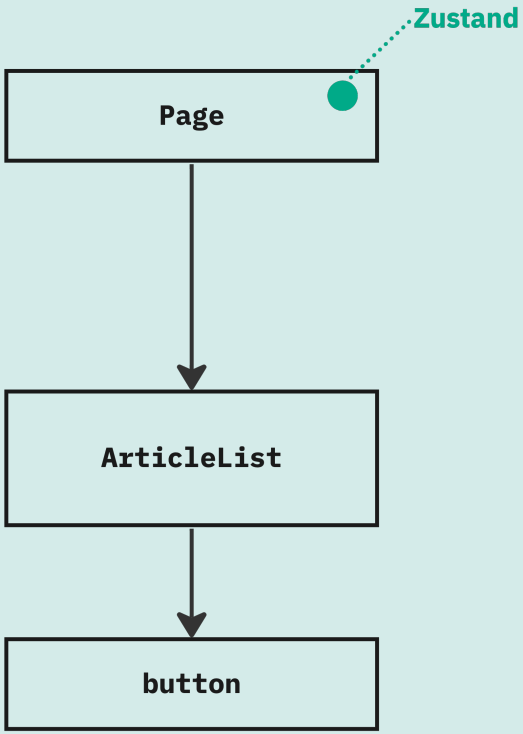
CAN YOU SPOT THE PROBLEM?

```
<button onClick={() => onSortOrderChange("asc")}>Asc</button>
```

```
⨯ Error: Event handlers cannot be passed to Client Component props.
  <button onClick={function} children=...>
                  ^^^^^^^^^^

If you need interactivity, consider converting part of this to a Client Component.
    at stringify (<anonymous>)
```

CAN YOU SPOT THE PROBLEM?

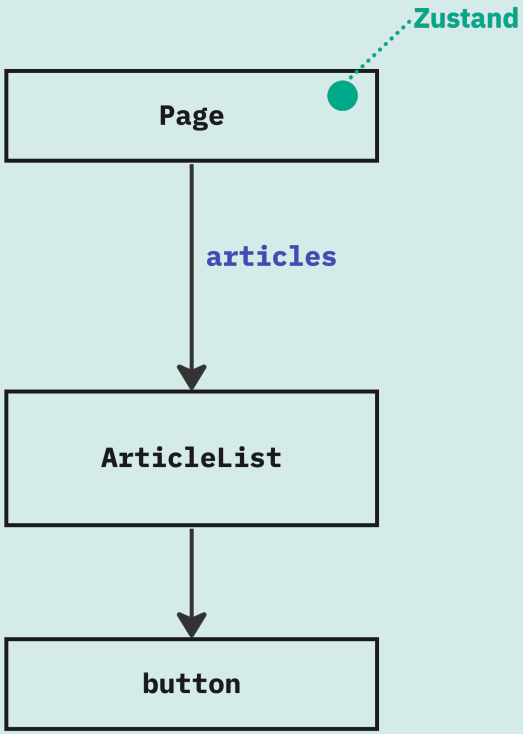
EINE REACT ANWENDUNG IM BROWSER



- Daten befinden sich im Client

Eine "normale" React-Anwendung...

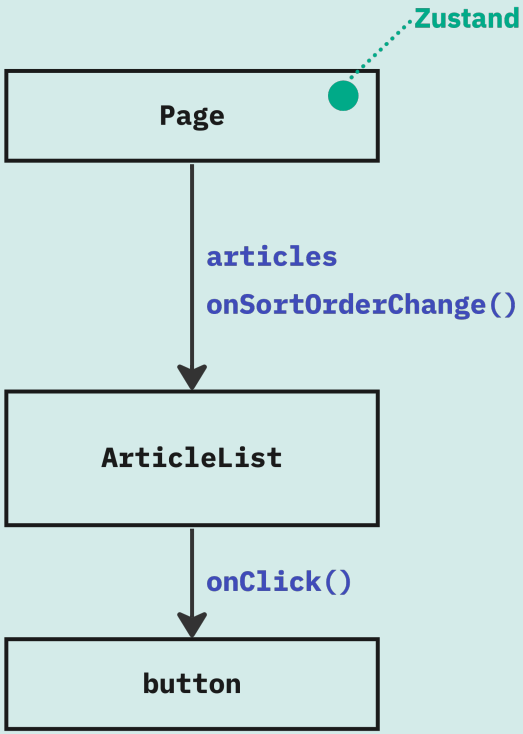
EINE REACT ANWENDUNG IM BROWSER



- Dafff befindet sich oben
- Daten werden runtergereicht ("props")

Eine "normale" React-Anwendung...

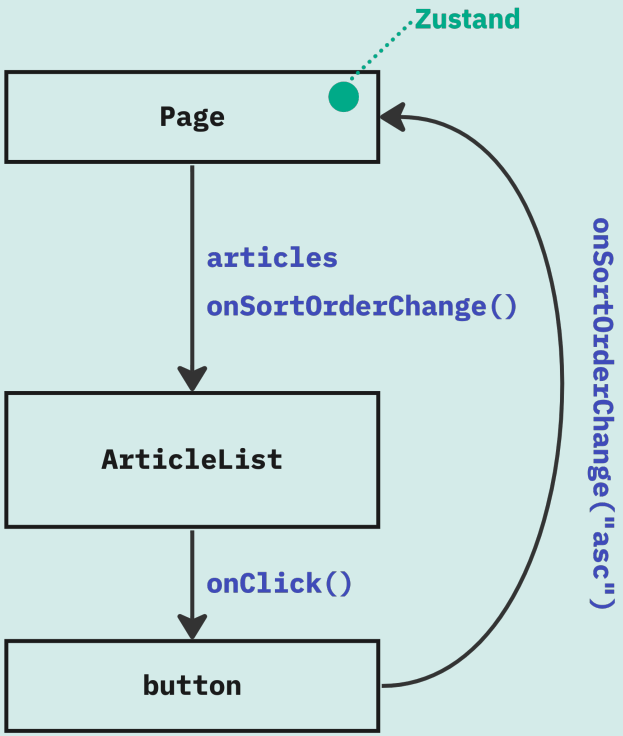
EINE REACT ANWENDUNG IM BROWSER



Eine "normale" React-Anwendung...

- State befindet sich oben
- Daten werden runtergereicht ("props")
- Callbacks werden runtergereicht

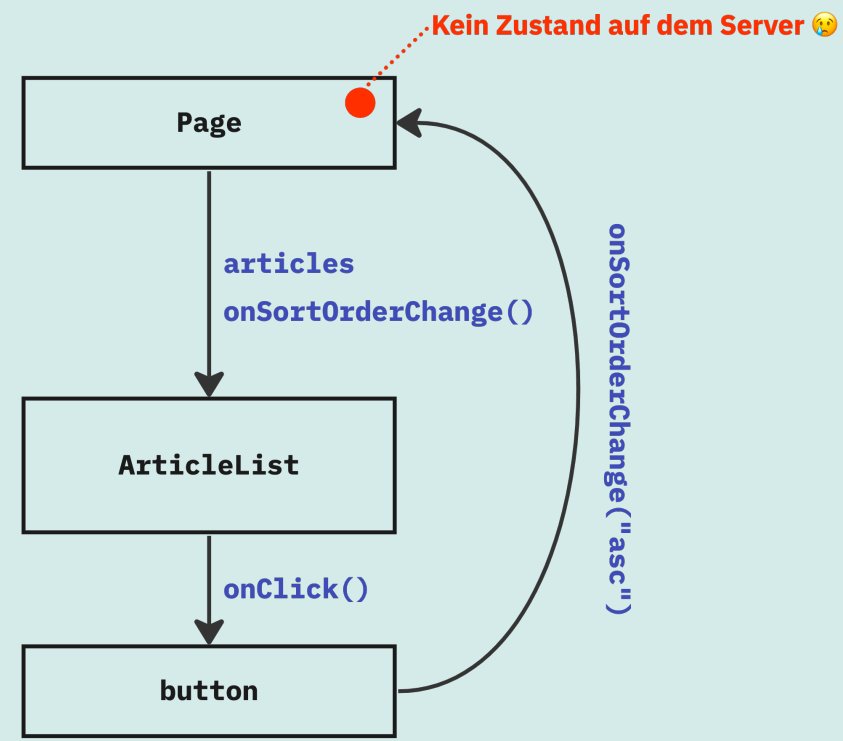
EINE REACT ANWENDUNG IM BROWSER



Eine "normale" React-Anwendung...

- State befindet sich oben
- Daten werden runtergereicht ("props")
- Callbacks werden runtergereicht
- Über Callbacks kann State-Veränderung ausgelöst werden

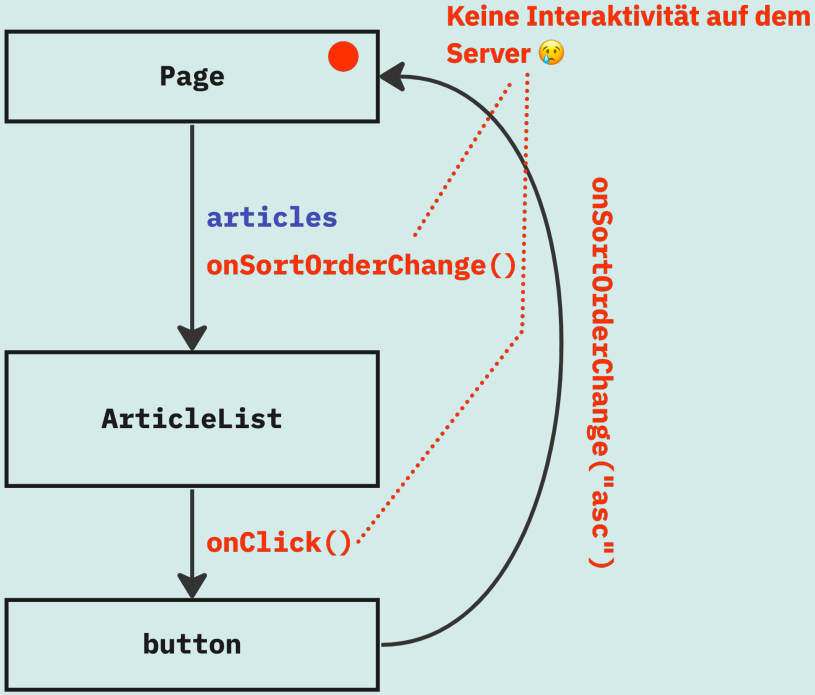
...UND AUF DEM SERVER



- Auf dem Server gibt es keinen UI-State!

Mit Next.js sind wir aber auf dem Server (by Default)

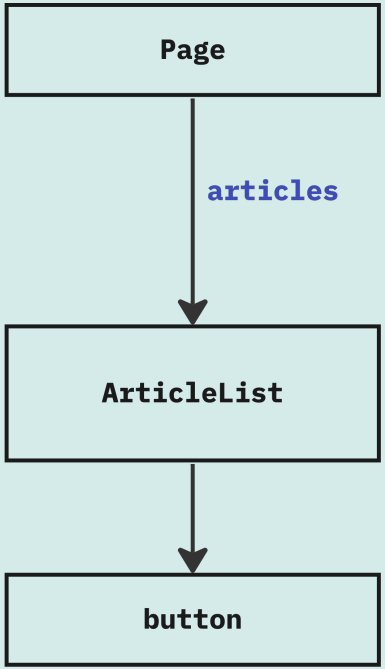
...UND AUF DEM SERVER



- Auf dem Server gibt es keinen State!
- ...und keine Interaktion

Mit Next.js sind wir aber auf dem Server (by Default)

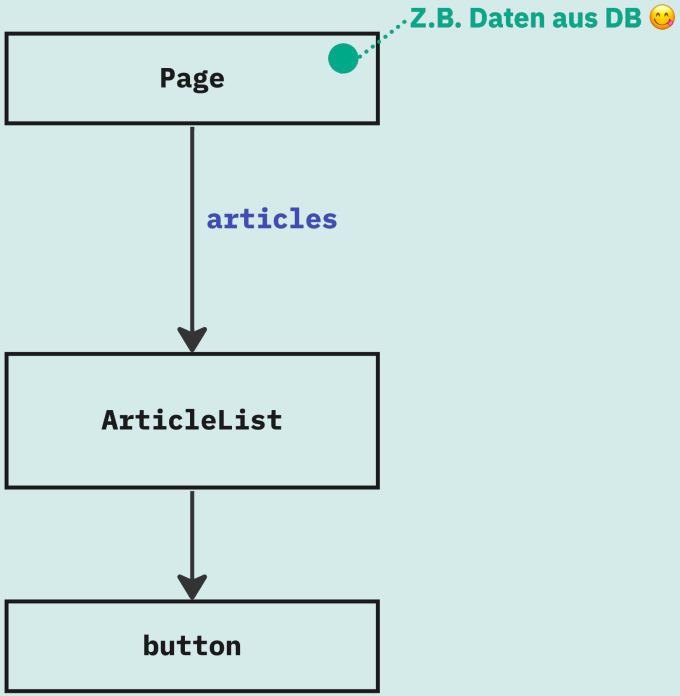
...UND AUF DEM SERVER



- Auf dem Server gibt es keinen State!
- ...und keine Interaktion
- Wir haben nur statischen Content

Mit Next.js sind wir aber auf dem Server (by Default)

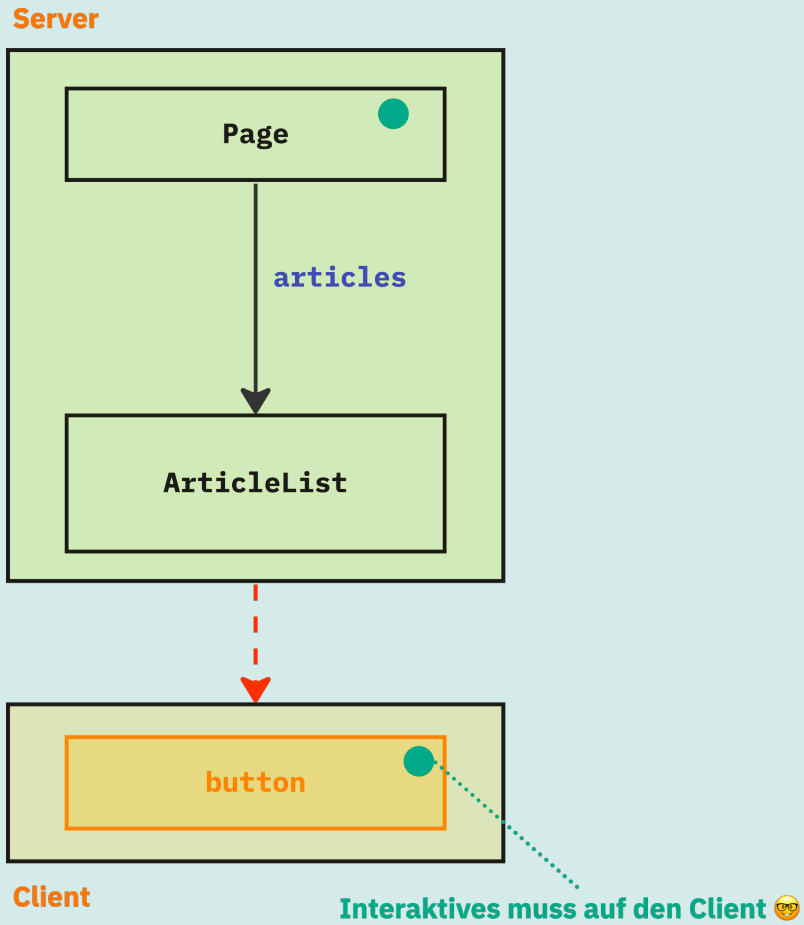
...UND AUF DEM SERVER



- Auf dem Server gibt es keinen State!
- ...und keine Interaktion
- Wir haben nur statischen Content
- Wir haben aber **Daten**
  z.B. aus DB, Microservice, Filesystem...

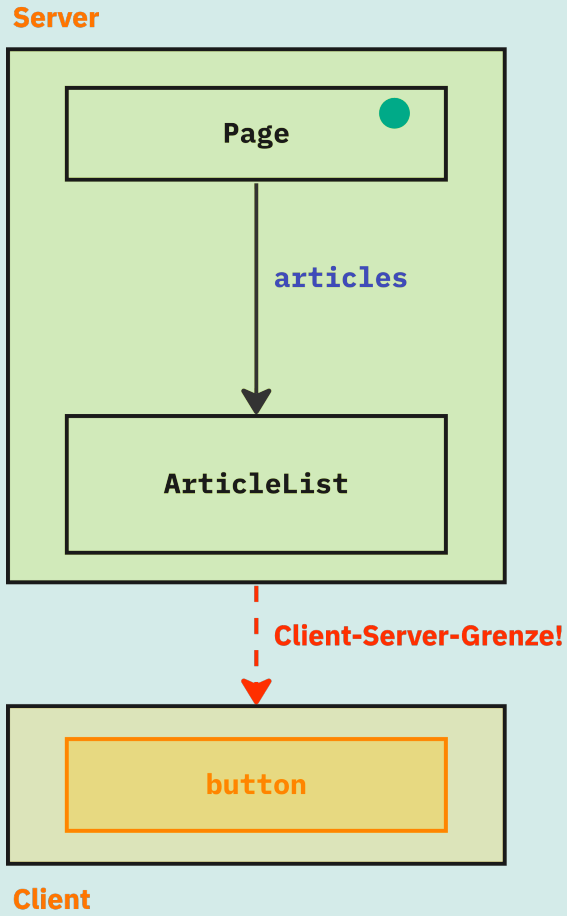
Mit Next.js sind wir aber auf dem Server (by Default)

...UND AUF DEM SERVER



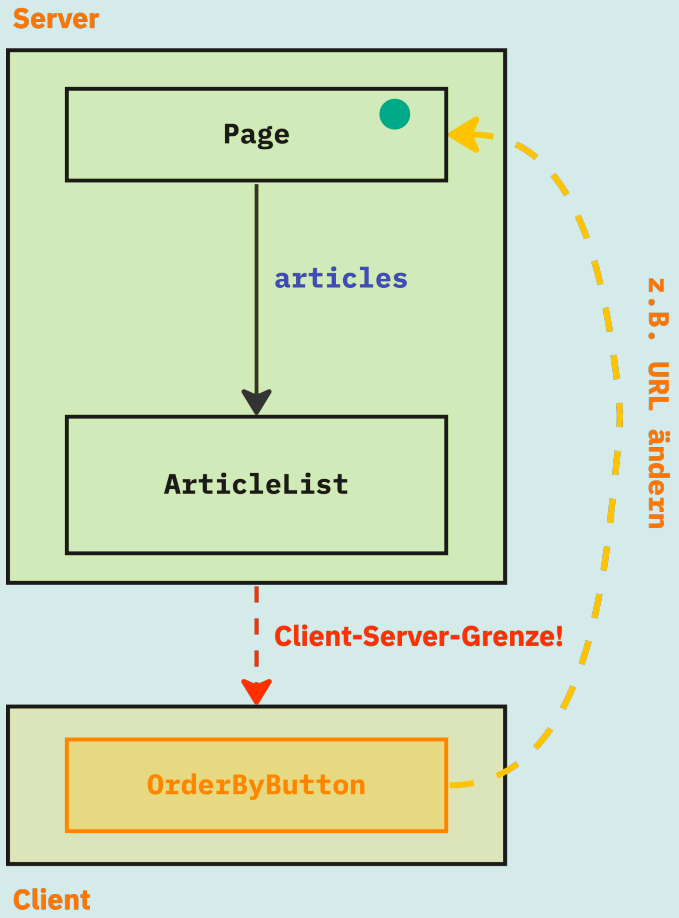
- Bestimmte Teile **müssen** auf den Client
  - Event-Handler
  - Lokaler State
  - Effekte

...UND AUF DEM SERVER



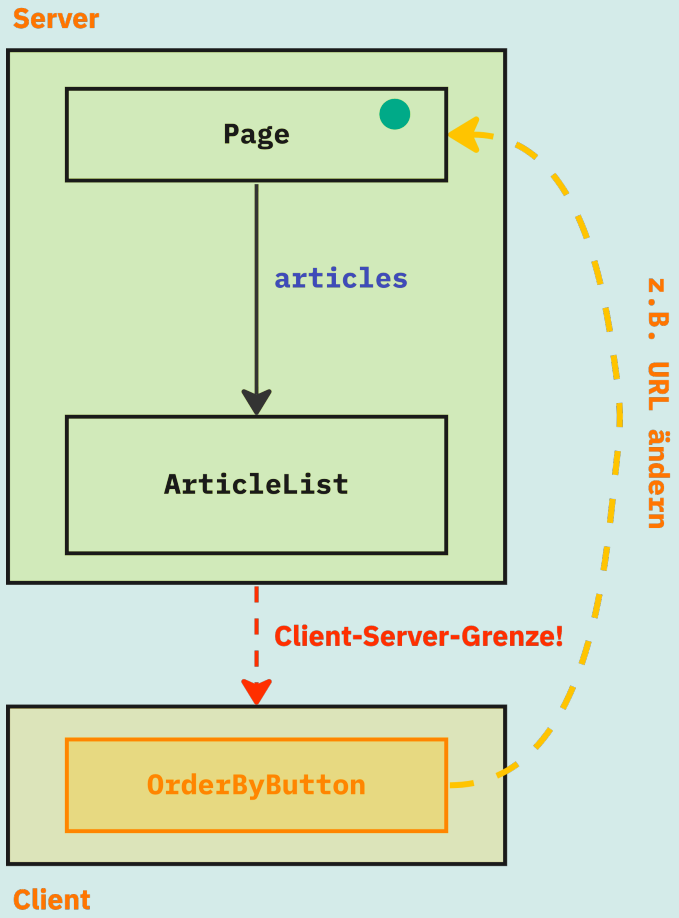
- Properties müssen hier Client-Server-Grenze überwinden
- Müssen serialisierbare Daten sein
- **Keine (Callback-)Funktionen!**

...UND AUF DEM SERVER



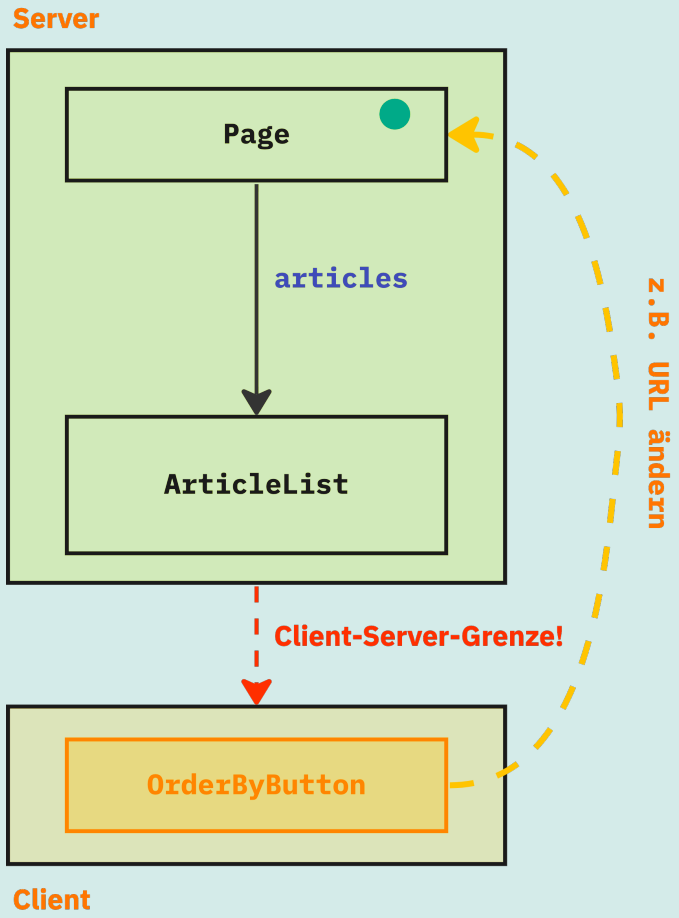
- Properties müssen Client-Server-Grenze überwinden
- Müssen serialisierbare Daten sein
- Keine (Callback-)Funktionen!
- Zur Kommunikation müssen **Server-Requests** durchgeführt werden
  - z.B. URL ändern

...UND AUF DEM SERVER



- Properties müssen Client-Server-Grenze überwinden
- Müssen serialisierbare Daten sein
- Keine (Callback-)Funktionen!
- Zur Kommunikation müssen Server-Requests durchgeführt werden
  - z.B. URL ändern
- **Server-Komponente hat Zugriff auf Request Informationen**
  - URL mit Search Params
  - Cookies
  - Headers

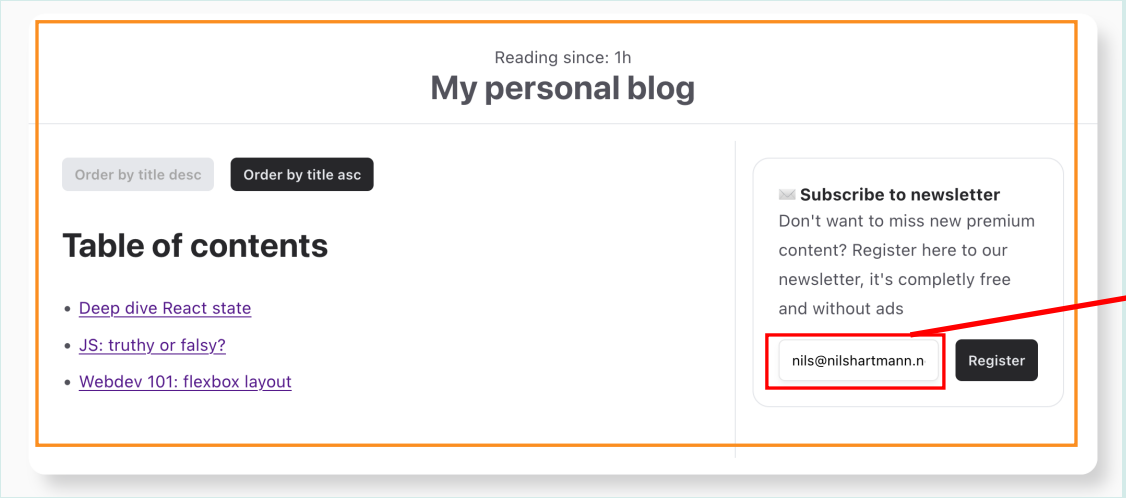
...UND AUF DEM SERVER



- **Konsequenz**
  - Code wird durch URL-Handling komplexer?
  - Wo ziehen wir Server/Client-Grenze?
    - Button? Ganzes Formular?
    - Hier werden sich Patterns entwickeln
  - Ganze Seite (oder Teile) werden neu gerendert
  - Fertiges UI kommt dafür vom Server
    - Das kann mehr Daten als bei (REST-)API-Call bedeuten!

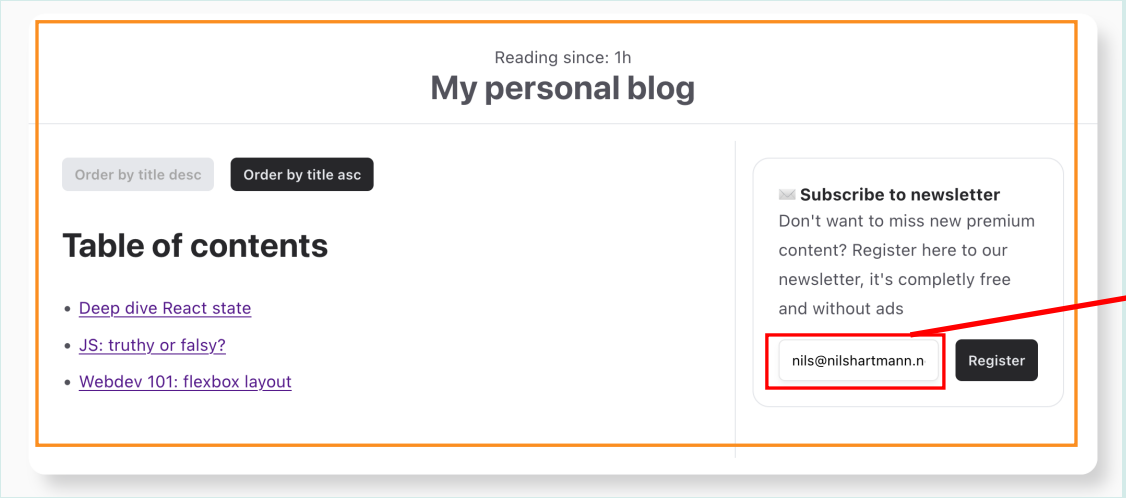
CLIENT UND SERVER DATEN

- Im Browser gibt es die statische UI (vom Server gesendet)
- Es gibt (UI) Zustand, der nur im Client ist
  - Inhalt eines Textfelds zum Beispiel
- In der Komponentenhierarchie kann das gemischt sein
- Der UI-Zustand soll nicht verloren gehen



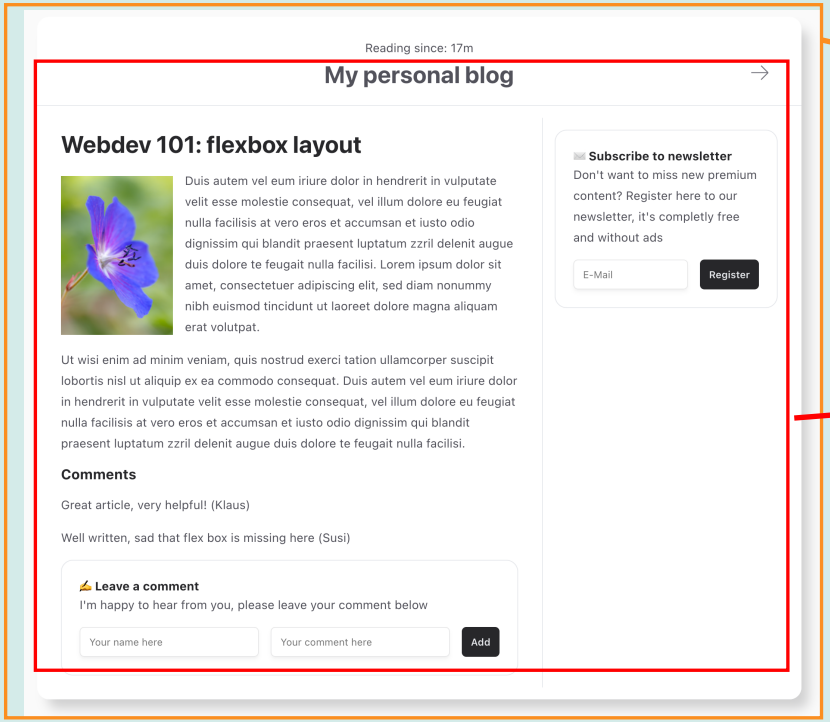
CLIENT UND SERVER DATEN

- **Zustand soll im Client erhalten bleiben**
- **In Next.JS bleibt beim "Refresh" einer Route der Client-Zustand erhalten**
  - Beispiel: Newsletter-Formular
  - Beispiel: Textfeld unter Inhaltsverzeichnis



CLIENT UND SERVER DATEN

- **Zustand soll im Client erhalten bleiben**
- **Bei Routenwechsel bleiben Daten in "unveränderten" Layouts erhalten**



Root-Layout (bleibt erhalten)

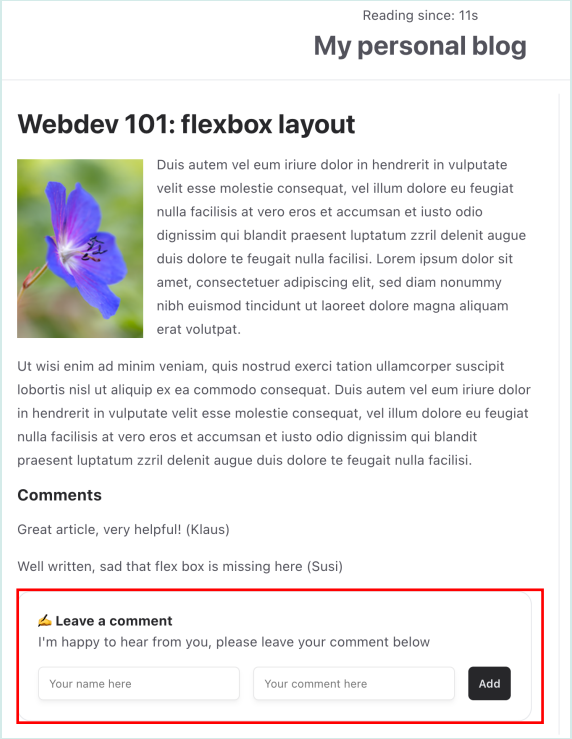
Layout für /post/postId-Route (wird neu gerendert)

🤦 Beispiel: Newsletter in Root-Layout verschieben

Data Mutations

MUTATIONS

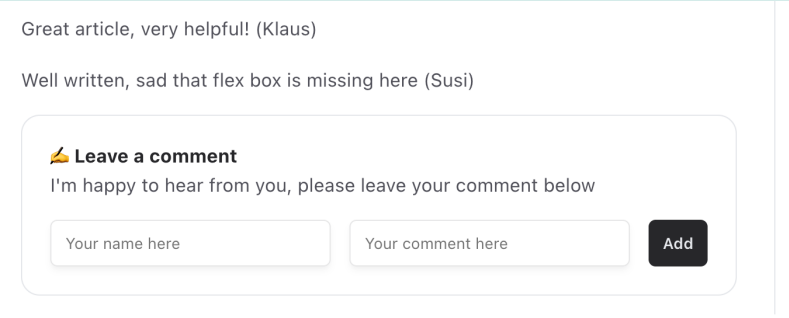
Verändern von Daten: Hinzufügen einer eines Kommentars



Verändern von Daten: Hinzufügen eines Kommentars

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 der neue Kommentar sofort in der Liste angezeigt
- Nach dem Speichern wird das Formular geleert

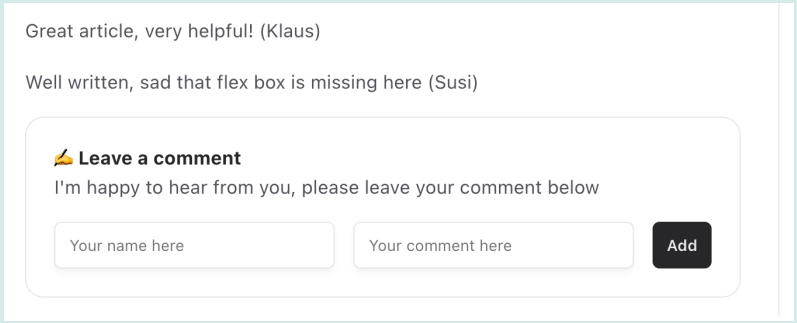


Verändern von Daten: Hinzufügen eines Kommentars

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 auf eine Fehlermeldung angezeigt
- Nach dem erfolgreichen Speichern wird der neue Kommentar sofort in der Liste angezeigt
- Nach dem Speichern wird das Formular geleert

Javascript im Client erfoderlich!



Verändern von Daten: Hinzufügen eines Kommentars

Anforderungen:

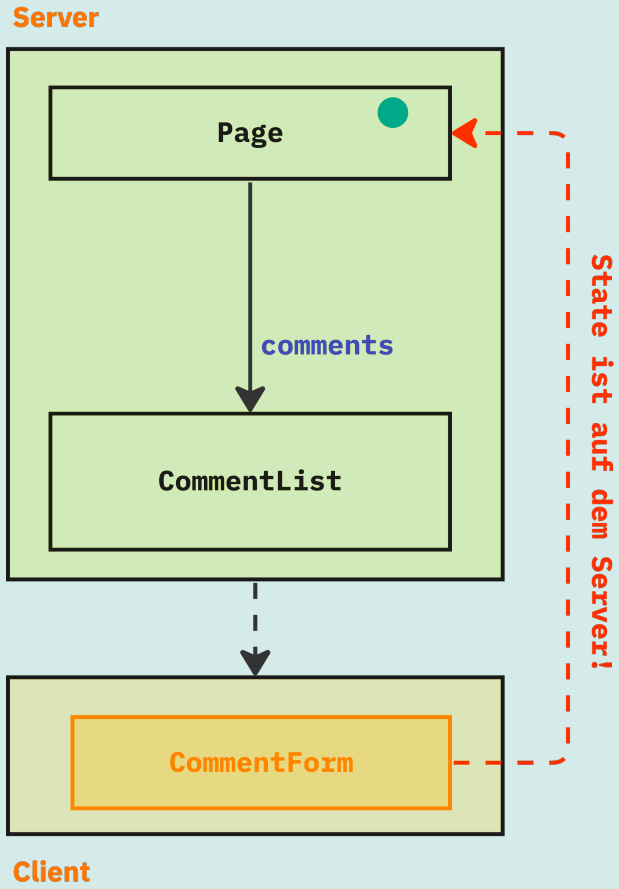
- Submit-Button ist disabled, bis alle Felder ausgefüllt sind
- Während Speicher-Request läuft, soll Submit-Button ebenfalls disabled sein und Meldung angezeigt werden
- Nach dem Submit wird ggf. eine Fehlermeldung angezeigt
- Nach dem erfolgreichen Speichern wird der neue Kommentar sofort in der Liste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

Javascript im Client erfoderlich!

Formular muss Client Komponente sein

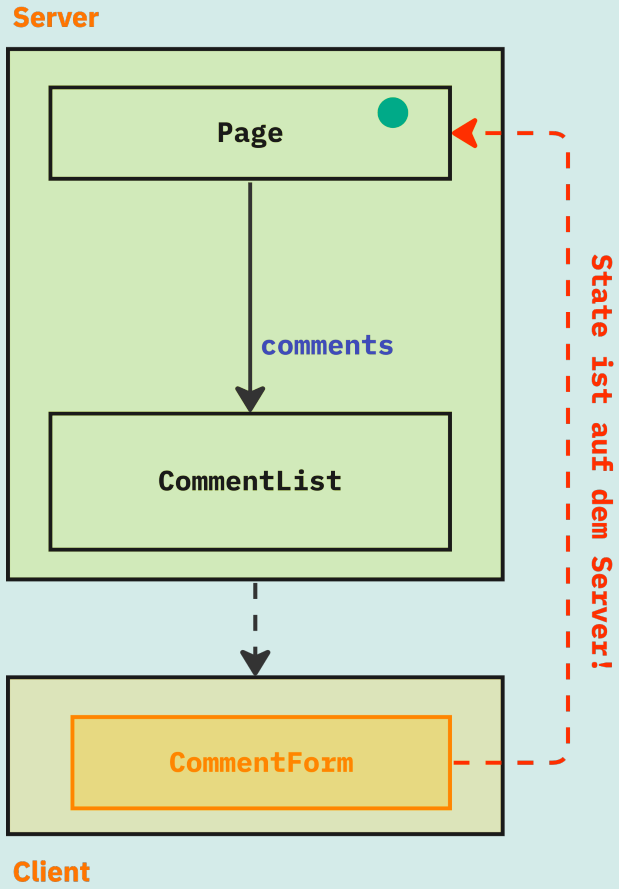
Verändern von Daten

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



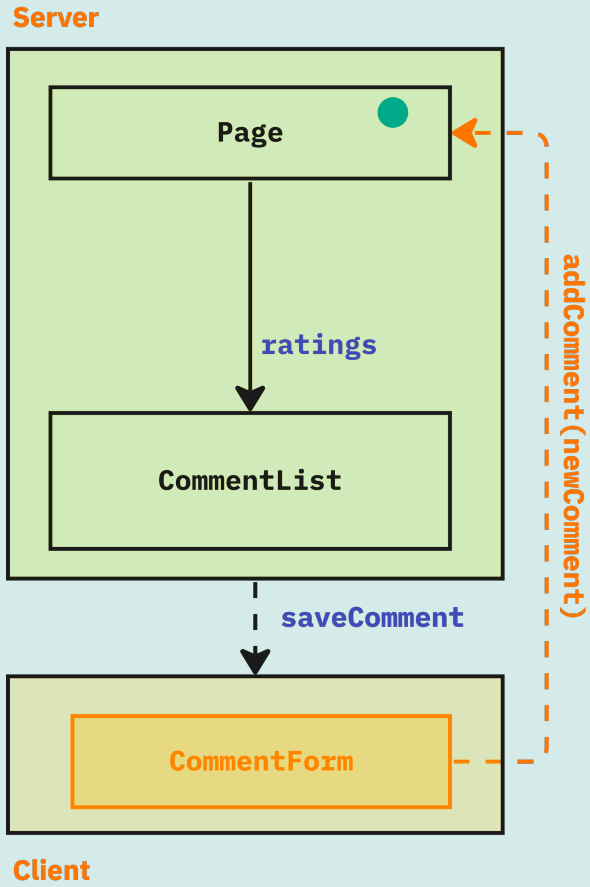
Verändern von Daten

- 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

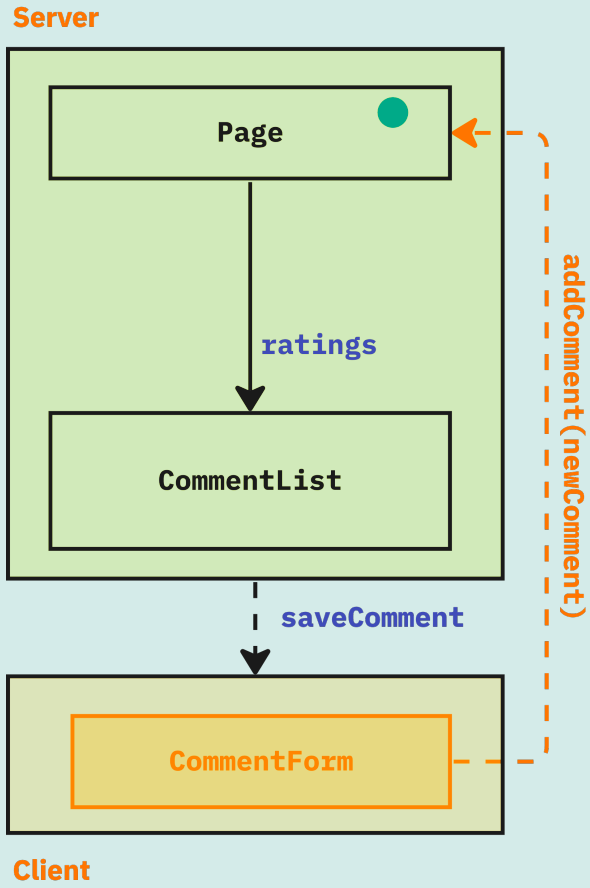
- Das ist eine Art Remote Funktion, die aus einer Server- oder Client-Komponente aufgerufen werden kann
- Next.js stellt dafür einen Endpunkt zur Verfügung
- Ausgeführt wird die Funktion auf dem Server
- Darin kann man angeben, welche Routen sich durch die Änderung der Daten verändert haben und neu gerendert werden müssen



MUTATIONS

Server Actions

```
"use server";

export async function saveComment(
  postId: string,
  newComment: AddCommentRequestBody,
) {
  const response = await fetch(`http://localhost:8081/api/comments/${postId}`, {
    method: "POST",
    body: JSON.stringify(newComment),
    headers: { "content-type": "application/json" },
  });

  if (!response.ok) {
    const err = await response.json();
    return { status: "error", err };
  }

  revalidatePath(`/post/${postId}`);

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



Formular

- Die Server Action-Funktion wird aus dem Formular heraus aufgerufen
- Das zurückgelieferte Ergebnis kann zur Aktualisierung des Formulars benutzt werden

```
export default function CommentForm({postId, mutation}: Props) {
  const [name, setName] = useState("");
  const [comment, setComment] = useState("");
  const [error, setError] = useState("");

  const handleSubmit = async (e: React.MouseEvent<HTMLButtonElement>) => {
    const response = await saveCommentForPost(postId, { name, comment })
    if (response.status === "success") {
      setName("");
      setComment("");
    } else {
      setError("Could not add comment.")
    }
  };

  return (
    <form>
      <div className="FormRow"...>
      <button
        type="submit"
        onClick={handleSubmit}
      >
        Add
      </button>

    </form>
  );
}
```

Rendering

FAZIT

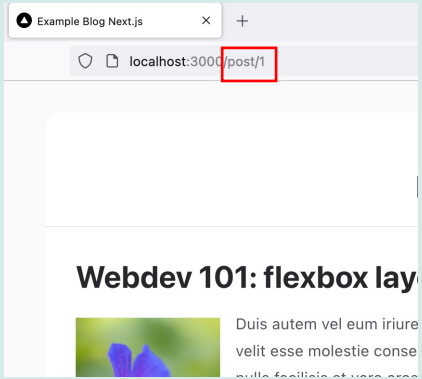
Statisches oder dynamisches Rendering

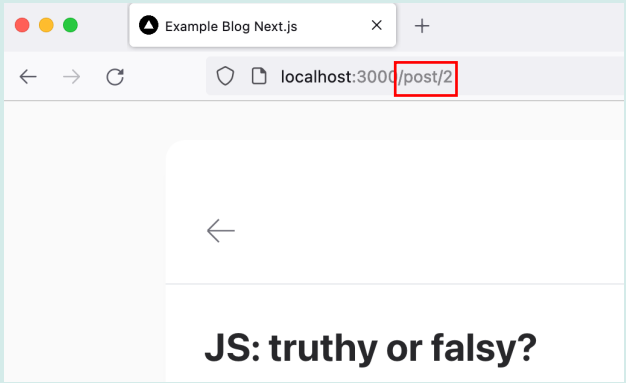
- **Statisches Rendering: Route wird zur Buildzeit gerendert**
  - Wenn die Route vom Browser abgefragt wird, ist das Verhalten fast wie bei einem klassischen Webserver: die fertige Datei kann ausgeliefert werden
  - Entspricht einer Art Cache
  - Trotzdem können weiterhin einzelne Teile auf der Website interaktiv sein (Client-Komponente)
- **Dynamisches Rendering: Route wird bei jedem Request gerendert**
  - Das ist vor allem für Teile, die sich häufig ändern relevant
  - Oder wenn Benutzer-Informationen bei jedem Request benötigt werden

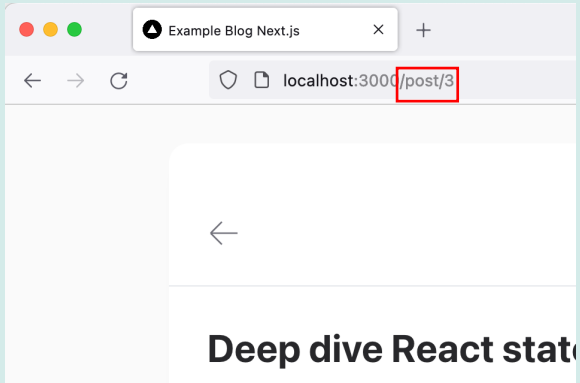
FAZIT

Statisches Rendering

- **Dynamische Informationen können im Build bereitgestellt werden**
  - Zum Beispiel die Ids für alle Blog-Posts, deren Seiten gerendert werden sollen







🕵️ Beispiel: [postId]/page.tsx, pnpm build, Log-Ausgabe

FAZIT

Statisches Rendering

- **Statische Seiten werden gecached**
  - Auf dem Server und im Browser
  - Leider (?) nur Routen-basiert
- **Invalidierung über API oder zeitgesteuert**
  - Das Verhalten ist manchmal verwirrend und leider (?) opt-out

Preloading

- **Interne Routen könnten von Next.js im Hintergrund geladen**
  - Dann geht der Seitenwechsel schneller

🕵️ **Beispiel:** Header.tsx enablePrefetch auf true stellen, pnpm build

Fazit

Die Anforderungen unserer Anwendung mit Next.js

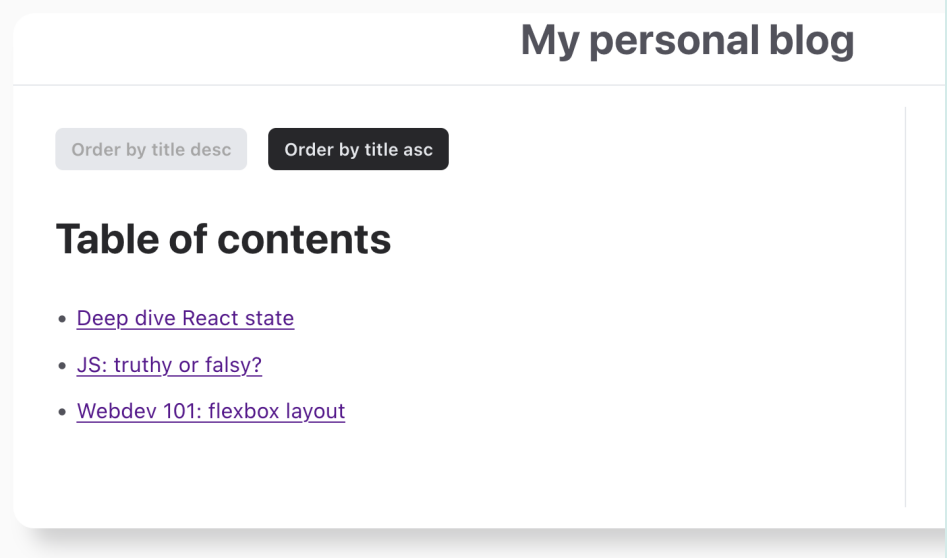
- Sinnvoll oder nicht?

JavaScript oder klassisch/statisch?

- Inhaltsverzeichnis für die Artikel einer Blogging-Platform

Anforderungen:

- Mit dem Klick auf einen Button wird die Liste der Titel umsortiert
- Der Button für die aktuelle Sortierung ist disabled
- Klick auf einen Blog-Titel öffnet den entsprechenden Artikel



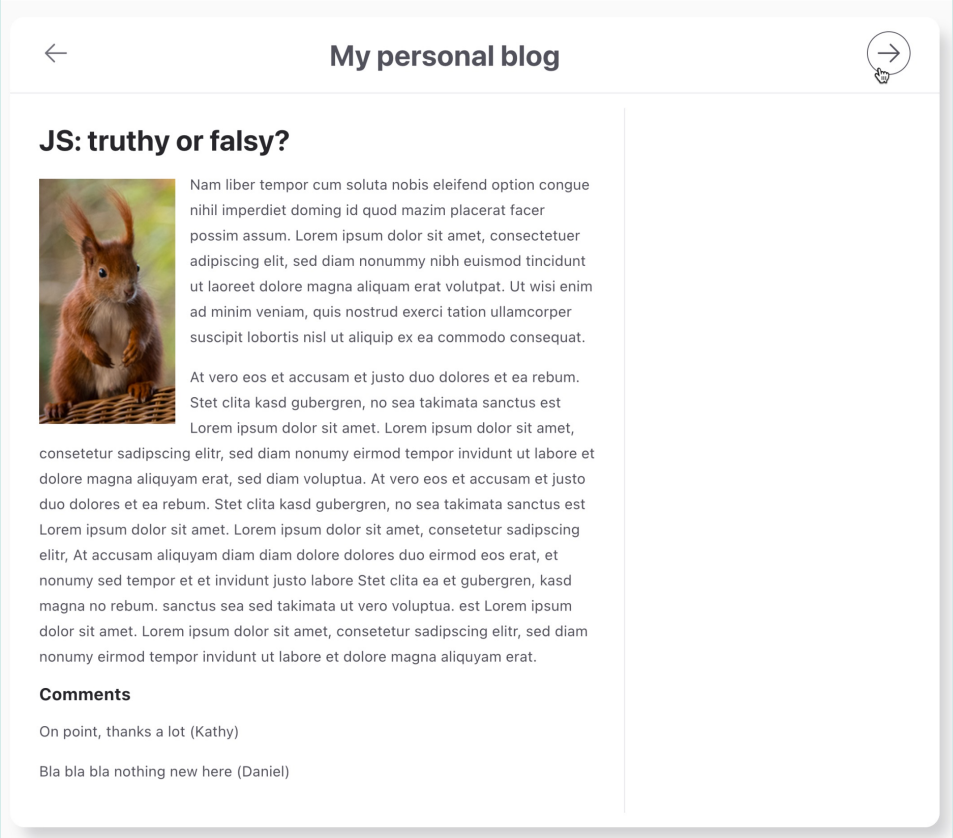
EINE WEB-ANWENDUNG

JavaScript oder klassisch/statisch?

- Artikel in einer Blog-Anwendung

<u>Anforderungen:</u>

- Mit den Pfeilen kann man vorherigen/nächsten Blog-Post anzeigen
- Die einzelnen Posts sind per Link adressierbar/als Bookmark speicherbar
- Die Artikel werden von Google gefunden (SEO)




step_01

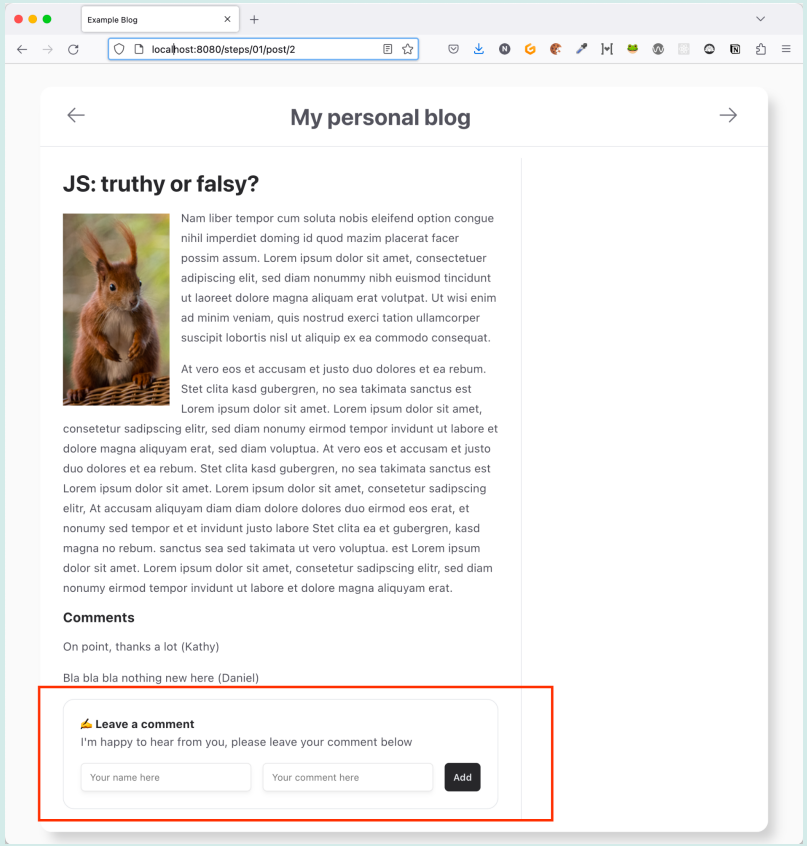
EINE WEB-ANWENDUNG

JavaScript oder klassisch/statisch?

- Neu: Es gibt ein Formular für Kommentare

<u>Anforderungen:</u>

- Validierung: Beide Felder sind Pflicht. Wann und wo Fehlermeldung?
- Nach dem Speichern soll der Kommentar direkt angezeigt werden



👨‍🏫 step_03

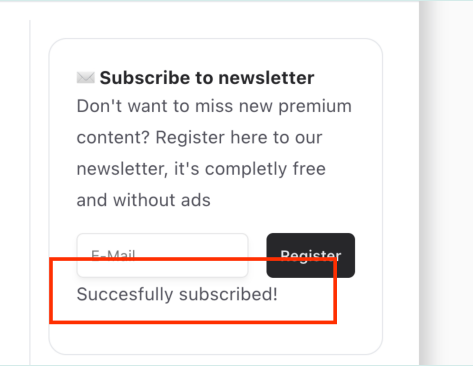
EINE WEB-ANWENDUNG

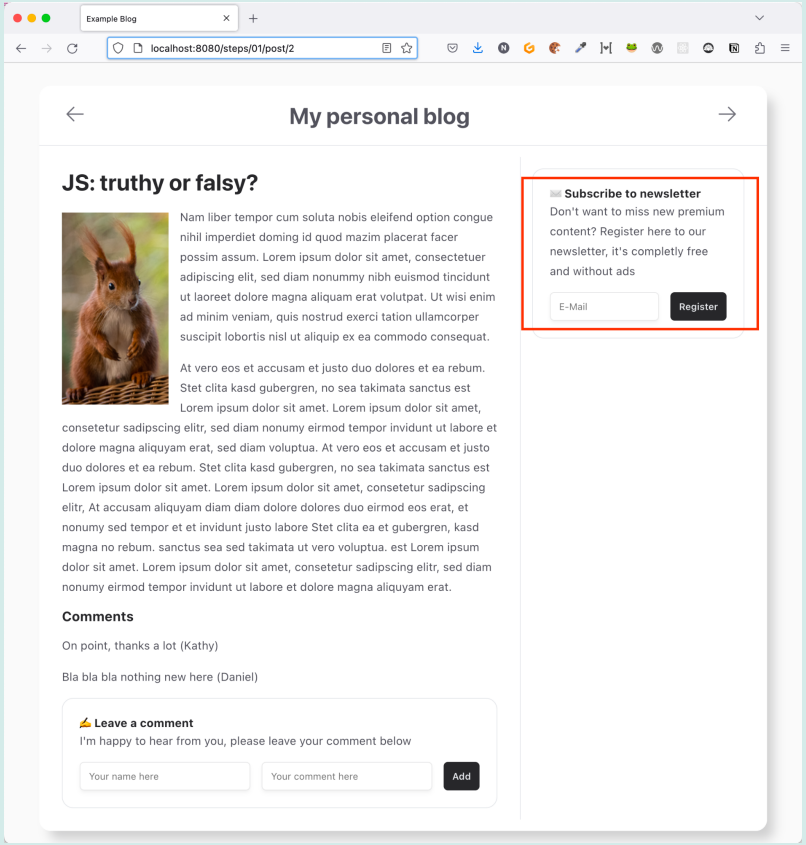
JavaScript oder klassisch/statisch?

- Neu: Es gibt ein 2. Formular ("Newsletter")

<u>Anforderungen:</u>

- Nach dem erfolgreichen Registrieren soll eine Meldung unter dem Textfeld auftauchen ("Vielen Dank für Ihre Registrierung)





step_03

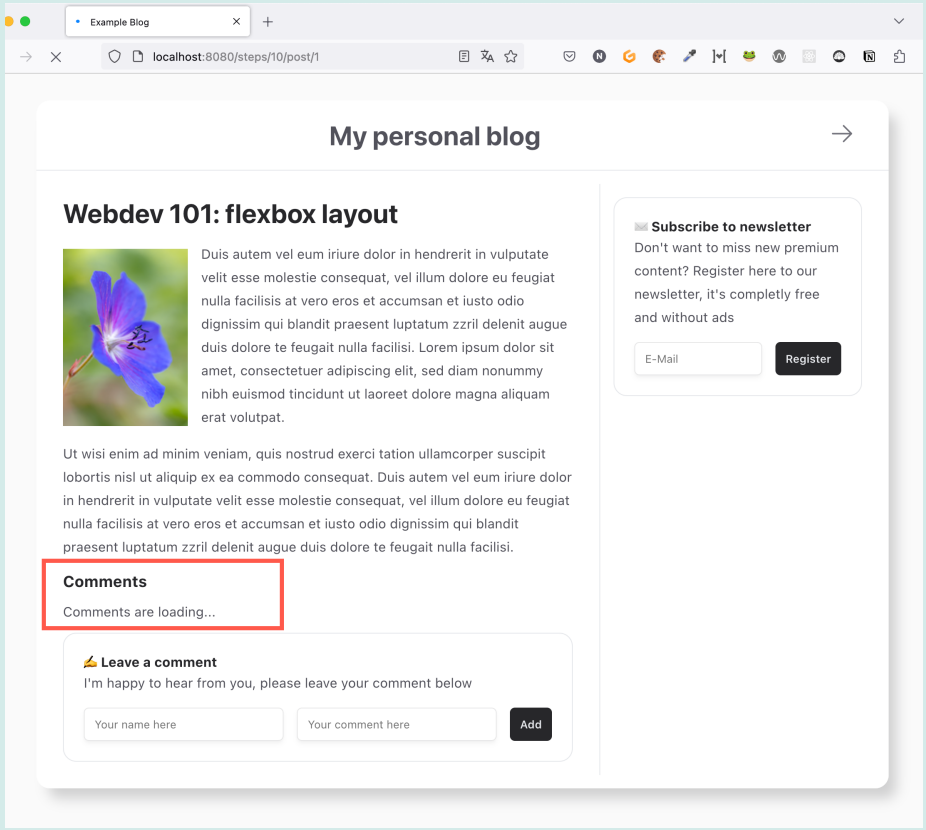
EINE WEB-ANWENDUNG

JavaScript oder klassisch/statisch?

- Der Blog-Post ist das Wichtigste der Seite

<u>Anforderungen:</u>

- Wenn das Ermitteln/Laden der Kommentare lange dauert, soll der Rest der Seite schon angezeigt werden



step_10

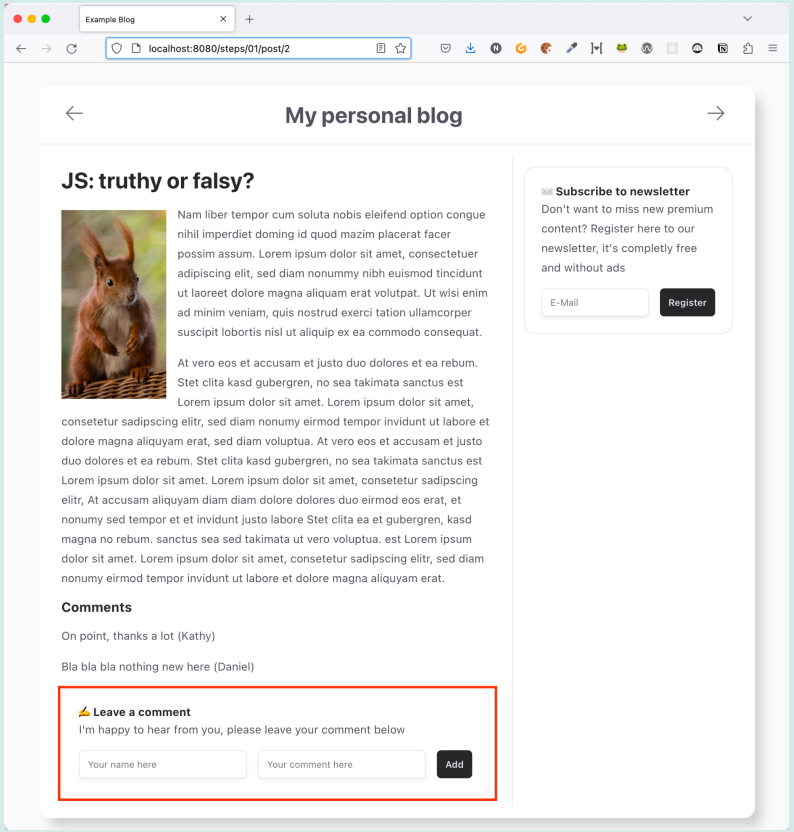
EINE WEB-ANWENDUNG

JavaScript oder klassisch/statisch?

- Das Speichern des Kommentares dauert etwas länger

Anforderungen:

- Während der neue Kommentar gespeichert wird, soll eine Wartemeldung angezeigt werden
- Während der Kommentar gespeichert wird, soll es nicht möglich sein erneut, auf "Add"-Button zu drücken ("Double Submit")

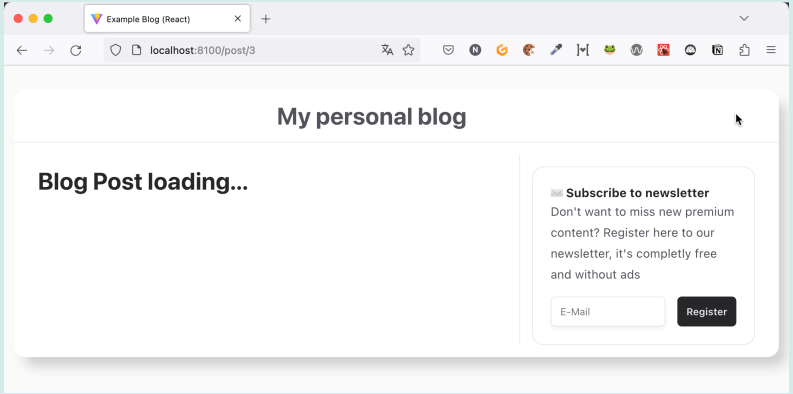


JavaScript oder klassisch/statisch?

- Lahmes Internet

Anforderungen:

- Beim Klicken auf die Pfeile (vorheriger/nächster Blogpost) soll eine Wartemeldung angezeigt werden, bis der nächste Blog-Post dargestellt werden kann



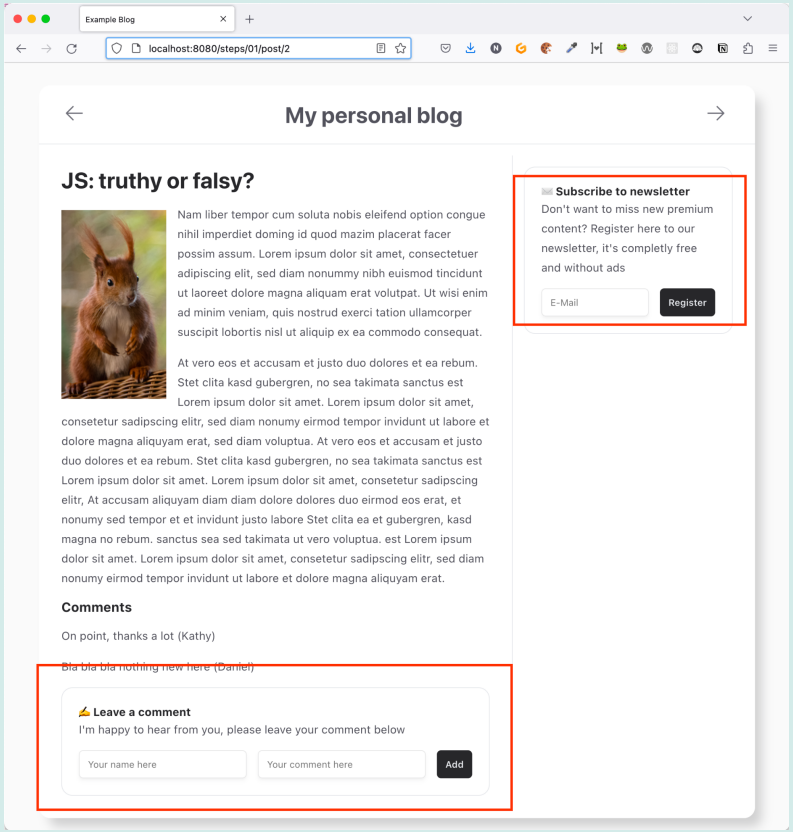
EINE WEB-ANWENDUNG

JavaScript oder klassisch/statisch?

- Zwei Formulare auf einer Seite

Anforderungen

- Wenn auf den "Register"-Knopf gedrückt wird, sollen Eingaben im Kommentar-Formular erhalten bleiben



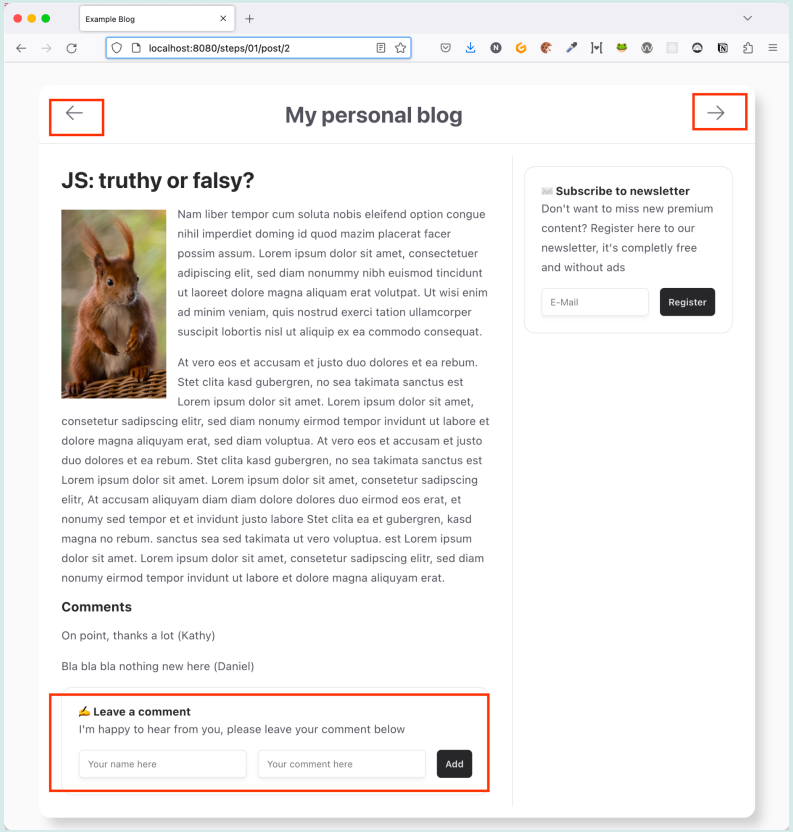
EINE WEB-ANWENDUNG

JavaScript oder klassisch/statisch?

- Kein Datenverlust

Anforderungen

- Beim Navigieren von einer Seite zur nächsten und wieder zurück sollen die Daten im Kommentar-Formular erhalten bleiben
- Auch bei der Verwendung des Back-/Forward-Buttons!



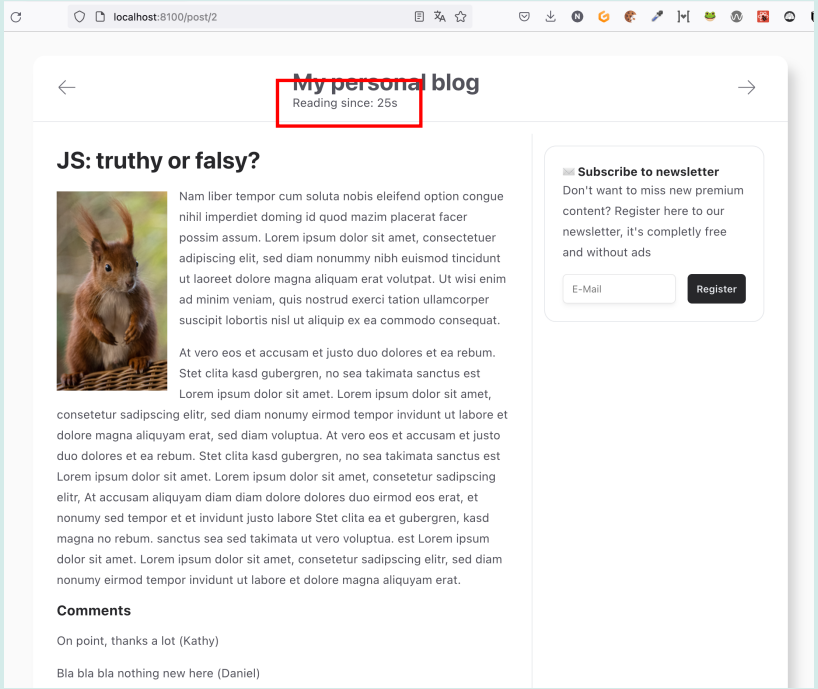
EINE WEB-ANWENDUNG

JavaScript oder klassisch/statisch?

- Im Header wird ein Timer o.ä. dargestellt (z.B. Session Timeout Timer)
- oder ein Video läuft (z.B. mit Werbung)

<u>Anforderungen</u>

- Beim Navigieren durch die Seiten soll das Video/der Timer ununterbrochen flüssig weiterlaufen



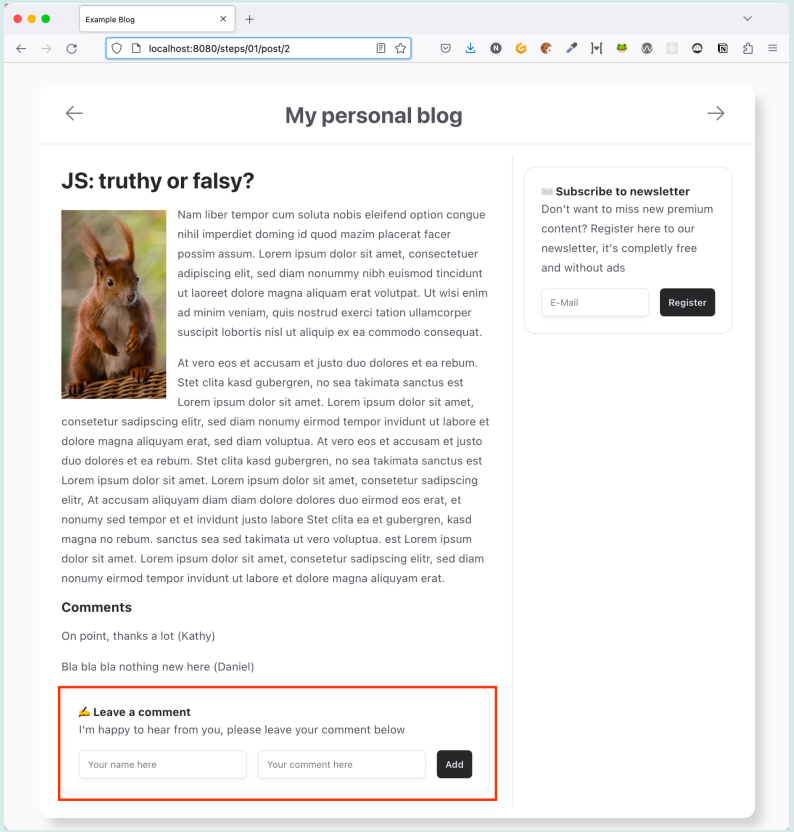
EINE WEB-ANWENDUNG

JavaScript oder klassisch/statisch?

- Der CommentService wird von einem externen Dienstleister betrieben
- Wir brauchen für den Zugriff einen API-Key

<u>Anforderungen:</u>

- Der API-Key ist ein Geheimnis und darf nicht für Dritte sichtbar sein




Fullstack
Architekturen
Das Ende der Single-Page-Anwendungen?


Fullsta
Architekturen
Nein!
Das Ende der Single-Page-Anwendungen?


Fullstack
Architekturen
Nein!
Das Ende der Single-Page-Anwendungen?
...es kommt drauf an, was ihr baut!

NILS HARTMANN
https://nilshartmann.net

Vielen Dank!

Slides: https://react.schule/saa2023

Source-Code: https://react.schule/saa2023-code

Fragen & Kontakt: nils@nilshartmann.net

Twitter: @nilshartmann

Artikel: https://www.sigs.de/autor/nils.hartmann