WEBENTWICKLUNG: TRENDS IM BEREICH FRONTEND-ARCHITEKTUR

Die Zukunft des Frontends

Aktuelle Entwicklungen auf dem Spektrum zwischen Single Page Application und Multi Page Application.

Welcher Architekturstil ist für eine neue Webapplikation angemessen? Wir unterscheiden zwischen zwei fundamentalen Stilen: Single Page Application (SPA) und Multi Page Application (MPA). Klassischerweise wird das HTML-Dokument einer Webseite von einem Server gerendert. Bei einem Seitenwechsel wird dabei eine neue Anfrage an den Server gestellt und dieser kann das HTML-Dokument für die geforderte Seite generieren. Dagegen liefert der Server bei Single Page Applications nur für die erste Seite ein HTML-Dokument – anschließend läuft eine JavaScript-Applikation auf dem Client, die Seiten rendern und zwischen Seiten wechseln kann, ohne mit dem Server zu sprechen. Zur Gegenüberstellung der beiden Architekturstile wird der klassische Ansatz heutzutage als Multi Page Application bezeichnet.

Seit dem Aufkommen der Frameworks Angular und React gewinnen Single Page Applications zunehmend an Popularität. Ursprünglich eröffneten sie neue Möglichkeiten für interaktive Webapplikationen, wie beispielsweise E-Mail-Clients, Landkarten oder Spiele. Mittlerweile werden sie häufig für alle Arten von Websites eingesetzt – von Blogs bis zu Online-Shops. Der Grund ist, dass Single Page Applications im Idealfall sowohl eine besonders gute User Experience als auch eine besonders gute Developer Experience ermöglichen. Der Preis dafür sind zusätzliche technische Komplexität und große Mengen an JavaScript. Die Herausforderung dabei ist, dass es aufgrund der technischen Komplexität und der großen Menge an JavaScript schwierig ist, die ideale User Experience und Developer Experience tatsächlich zu erreichen. Viele Teams entscheiden sich aktuell blind für eine Single Page Application, ohne Alternativen bewusst abzuwägen [1].

Wir haben nun einen Wendepunkt erreicht. Wir erkennen, dass es ungeschickt ist, alles auf den Client zu verlagern. Gleichzeitig wollen wir nicht zurück zu klassischen Multi Page Applications. Auf dem Spektrum zwischen Single Page Application und Multi Page Application gibt es Bewegung auf beiden Seiten, um die Frontend-Architektur der Zukunft zu erarbeiten. Frameworks wie React arbeiten an Ansätzen, die wieder vermehrt auf die Serverseite setzen und die Menge an JavaScript auf dem Client reduzieren. Andere Frameworks fokussieren sich dagegen auf den Multi-Page-Application-Ansatz und versuchen, die Vorzüge von Single Page Applications auf anderem Weg zu erreichen. Dieser Artikel vergleicht zunächst die beiden klassischen Architekturstile und stellt dann aktuelle Entwicklungen und Trends vor.

Architekturstil Multi Page Application (MPA)

Der Begriff Multi Page Application bezeichnet den klassischen Architekturstil für Webapplikationen – eingesetzt lange bevor es das Konzept Single Page Application gab. Dabei spielt der Server eine zentrale Rolle:

- **Server Side Rendering**: Der Server generiert das vollständige HTML-Dokument für den Browser, der damit die Seite anzeigen kann.
- **Server Side Routing:** Die Navigation von Seite zu Seite läuft immer über den Server, der die nächste Seite rendert.
- **Server Side State**: Da der Server die Benutzer:innen durch Routing und Rendering stets begleitet, kann man Zustandsinformationen einfach auf dem Server speichern und bei Page Requests verwenden.

Typische Technologien für klassische Multi Page Applications sind ASP.NET mit Razor Pages, Java mit Java Server Pages, PHP und Rails. Dazu wird JavaScript für interaktive Elemente auf der Website eingesetzt.

Listing 1 und Listing 2 zeigen das grobe Konzept einer Multi Page Application am Beispiel eines interaktiven Counters. Zuerst sendet der Browser einen Request für das HTML-Dokument. Der Browser lädt das referenzierte Stylesheet herunter und kann mit HTML und CSS die Webseite anzeigen. Zu diesem Zeitpunkt werden Zählerstand und Plus-Button zwar angezeigt, aber sie funktionieren noch nicht. Dann wird das re-

Listing 1: Server-gerendertes HTML-Dokument einer klassischen Multi Page Application

```
<html>
  <head>
    <link rel="stylesheet" href="styles.css">
    <script defer src="javascript.js"></script>
  </head>
  <body>
    <h1>My Counter</h1>
    <input id="count" type="number" value="0"></input>
    <button id="plus">+</button>
  </body>
</html>
```

ferenzierte Skript ausgeführt, um die Seite interaktiv zu machen. Das Skript implementiert das Inkrementieren durch den Button – hier mit jQuery-Syntax. Dabei hängt das Skript vom Markup ab. Die IDs müssen übereinstimmen und die Elemente müssen vorhanden sein, damit die Seite funktioniert.

Die Trade-offs von Multi Page Applications [2]:

- **Vorteil – Schnelles Laden der ersten Seite:** Wenn der Server das HTML-Dokument für den Browser generiert, kann der Browser dieses direkt anzeigen. JavaScript ist nur für interaktive Elemente nötig und kann sparsam eingesetzt werden. Die Seite wird schnell geladen.
- **Vorteil – Resilienz gegen JavaScript-Fehler:** Wenn JavaScript sparsam eingesetzt wird, funktioniert der Großteil der Website, selbst wenn die JavaScript-Ausführung fehlschlägt. Die Server-gerenderten Seiten können auch ohne JavaScript angezeigt werden. Klassische HTML-Formulare funktionieren ohne JavaScript.
- **Vorteil – Barrierefreiheit und Webkonventionen:** Benutzer:innen erwarten, dass Websites eine Reihe an Konventionen erfüllen, wie das Wiederherstellen der Scroll-Position beim Zurückgehen in der Historie oder das Öffnen von Seiten in einem neuen Tab beim Klick der mittleren Maustaste. Bei Multi Page Applications stellt der Browser sicher, dass die Konventionen eingehalten werden, ohne dass sich die Entwickler:innen darum kümmern müssen. Zur Barrierefreiheit sollten Websites zudem für Benutzer:innen mit eingeschränktem Sehvermögen per Screenreader bedienbar sein. Dazu gehört, dass der Fokus der Webseite passend gesetzt wird, wenn man auf der Seite navigiert. Wenn bei der Navigation ganze Seiten geladen werden, wird der Fokus automatisch an den Anfang der Seiten gesetzt.
- **Nachteil – Developer Experience mit zwei „Applikationen“:** Die größte Schwachstelle von klassischen Multi Page Applications ist die Developer Experience. Moderne Websites kommen selten ohne interaktive Komponenten aus. Sie benötigen JavaScript. Für die Entwicklung bedeutet das, dass man einen Tech-Stack für den Server und einen anderen Tech-Stack für die interaktiven Komponenten benötigt. Auf Serverseite kann das zum Beispiel ASP.NET mit einer Template Engine wie Razor Pages sein. Auf Clientseite kommen dann Technologien aus dem JavaScript-Ökosystem zum Einsatz. Zudem müssen die beiden Seiten miteinander interagieren: Die JavaScript-Seite verlässt sich oft auf das exakte Markup, das die Server-Seite generiert.
- **Nachteil – 3rd-Party-Skripte:** Skripte für Cookie-Einwilligung und Tracking zu Analyse- oder Marketingzwecken laufen bei Multi Page Applications auf jeder Seite. Sie können die Performance der Seite negativ beeinträchtigen.

Listing 2: JavaScript-Skript einer klassischen MPA

```
var count = parseInt($("#count").val());

$("#plus").click(function() {
  count++;
  $("#count").val(count);
})
```

Architekturstil Single Page Application (SPA)

Bei Single Page Applications handelt es sich um clientseitige Applikationen, die aus Browser-Sicht eine einzige Seite laden und diese dann bei Interaktion und Navigation manipulieren. Dabei spielt der Server eine untergeordnete Rolle:

- **Client Side Rendering:** Die clientseitige Applikation ist in der Lage, Seiten selbst zu rendern, ohne dafür mit dem Server zu kommunizieren.
- **Client Side Routing:** Die Applikation kann clientseitig zwischen Seiten navigieren, ohne dafür mit dem Server zu kommunizieren.
- **Client Side State:** Die Applikation kann Zustandsinformationen in Caches und State Stores clientseitig speichern und organisieren. Der Zustand bleibt bei Navigation innerhalb der Applikation bestehen.

Typische Technologien für klassische Single Page Applications sind Komponenten-Frameworks wie React, Angular oder Vue.js und Meta-Frameworks wie Next.js oder NuxtJS.

Listing 3 zeigt das grobe Konzept einer Single Page Application am Beispiel eines interaktiven Counters mit Vue.js. Zuerst sendet der Browser einen Request für das HTML-Dokument. Das Dokument ist nahezu leer mit einem leeren * * im *<body>*-Tag. Das Skript erzeugt die clientseitige App und mountet sie am leeren * * mit der ID *“app“*. Die App besteht aus einer Komponente, die sowohl das Markup als auch das Verhalten der Komponente beschreibt. Die Dateien liegen in derselben Codebasis und die Komponenten können interaktive wie nicht interaktive Elemente repräsentieren. ▶

Die Vor- und Nachteile von Single Page Applications [2]:

- **Vorteil – Developer Experience:** Moderne Komponenten- und Meta-Frameworks bieten exzellente Developer Experience. Das beginnt bereits beim initialen Aufsetzen des Projekts, wo umfangreiche Wizards durch eine Serie von Optionen für das Projekt führen und Libraries zum Linting und Testen automatisch konfigurieren. Anschließend erreicht man schnelle Feedback-Zyklen beim lokalen Entwickeln, da Änderungen am Quellcode sofort in die lokal laufende Entwicklungsinstanz der Applikation übernommen werden. Zudem führt die Popularität der Frameworks zu umfangreichen Ökosystemen mit einer Vielzahl an Bibliotheken und Lernmaterial. Ein entscheidender Unterschied zur klassischen Multi Page Application ist, dass Technologien sowohl die statischen Inhalte als auch die interaktiven Komponenten abdecken. Das ganze Frontend kann damit mit derselben Technologie in derselben Codebasis umgesetzt werden.
- **Vorteil – App-artige User Experience:** Single Page Applications ermöglichen es, rein clientseitig zwischen Seiten zu wechseln und mit ihnen zu interagieren – ohne Latenzen für Server-Requests. Dadurch fühlt sich die Navigation zwischen Seiten schnell an. Dabei behält die clientseitige App ihren Zustand und kann die Navigation mit Transitionseffekten aufwerten.
- **Nachteil – Langsames Laden der ersten Seite:** Während die Performance beim Navigieren zwischen Seiten gut ist, ist das initiale Laden einer Single Page Application relativ langsam. Die großen Mengen an JavaScript müssen heruntergeladen, geparst und ausgeführt werden. Mit Server Side Rendering kann der Browser die erste Seite schnell anzeigen, doch für Interaktivität muss das JavaScript ausgeführt werden.
- **Nachteil – Keine Resilienz gegenüber JavaScript-Fehlern:** Single Page Applications setzen für die Navigation zwischen Seiten und die Interaktion mit Seiten auf JavaScript. Ein Fehler beim Laden des Skripts kann dazu führen, dass die komplette Website bricht.
- **Nachteil – Barrierefreiheit und Webkonventionen:** Single Page Applications können das Standardverhalten des Webbrowsers nicht vollständig ausnutzen. Zum Beispiel wird die Navigation zwischen Seiten mittels JavaScript realisiert, ohne dass es im Browser zu einem vollen Page Load kommt. Dabei verhalten sich „Links" zwischen Seiten der App nicht wie klassische Links. Das Einhalten von Webkonventionen wird daher nicht vom Browser sichergestellt, sondern hängt von der Single Page Application ab. Es kommt häufig vor, dass Websites es nicht zulassen, Links in einem neuen Tab zu öffnen, indem man sie mit der mittleren Maustaste anklickt. Ebenso kommt es vor, dass Websites die letzte Scroll-Position verlieren, wenn man über den Zurück-Button zur vorigen Seite geht. Diese Probleme wirken sich auch auf die Barrierefreiheit aus. Ein häufiges Problem ist, dass bei der Navigation der Fokus nicht korrekt gesetzt wird. Das erschwert die Steuerung mit Tastatur und Screenreader.

Zeitenwende

In den letzten Jahren haben sich Single Page Applications zunehmend gegen Multi Page Applications durchgesetzt. Sie werden für alle Arten von Websites eingesetzt, wie beispielsweise Blogs, die wenig Interaktivität bieten. React ist das mit Abstand populärste Framework mit etwa 16 Millionen NPM-Downloads pro Woche [3]. Der Siegeszug von Single Page Applications führt dabei jedoch auch zu mehr und mehr JavaScript auf dem Client [4], was auf mobilen Geräten und in mobilen Netzwerken suboptimal ist. Die Trade-offs gegenüber Multi Page Applications werden dabei oft ungenügend berücksichtigt [1].

Listing 3: Code einer klassischen Single Page Application

```
// index.html
<html>
  <head>
    <script defer src="javascript.js"></script>
  </head>
  <body>
    <div id="app"></div>
  </body>
</html>

// main.js - Instanzierung der App
import { createApp } from 'vue';
import App from ,./App.vue';

const app = createApp(App);
app.mount('#app');
```

```
// App.vue - Komponente für einen Counter
<script>
export default {
  data() {
    return {
      count: 0
    };
  }
}
</script>

<template>
  <div>
    <h1>My Counter</h1>
    <span>{{ count }}</span>
    <button @click="count++">+</button>
  </div>
</template>
```

Listing 4: Progressive Enhancement in Remix

```
Export default function NewToDo() {
  Const transition = useTransition();
  return (
    <Form method="post">
      <input type="text" name="todo" />
      <button type="submit">
        {transition.state === 'submitting'
          ? 'Creating...' : 'Create'}
      </button>
    </Form>
  )
}
```

Hier zeichnet sich ein Umdenken ab [5][6]. Das Pendel auf dem Spektrum zwischen Single Page Application und Multi Page Application wird sich wieder in Richtung Multi Page Application bewegen. Dieses Umdenken geht nicht nur von den Kritikern von Single Page Applications aus. So schrieb Dan Abramov – Mitglied des React-Core-Teams – im Mai 2020 auf Twitter: „Client-side-only is not sustainable. We need to move more stuff to the server, but without sacrificing seamless composition of interactive pieces. […] Today, client-centered and server-centered are two different ends of the spectrum. You have to design your app in different ways. But what if you didn't have to choose between the Rails-like model and the client React model? The future is hybrid." [7]

Bessere SPA mit Progressive Enhancement

Eines der Probleme von klassischen Single Page Applications ist die fehlende Resilienz. Fehler beim Laden von JavaScript können die ganze Seite brechen. Ein neuer Trend schafft Abhilfe: Progressive Enhancement. Das Konzept dahinter ist, Dinge zu bauen, die für möglichst viele Benutzer:innen funktionieren, und sie dann für jene zu verbessern, deren Geräte mehr Potenzial haben. Für die Resilienz von Single Page Applications bedeutet das, dass die Website ohne JavaScript funktionieren sollte und dann mithilfe von JavaScript verbessert wird. Dazu muss eine Single Page Application als „Fallback" wie eine Multi Page Application funktionieren.

Die Idee ist, dass die Seiten der Website mit Server Side Rendering generiert werden können und auf HTML-Standards wie Links und Formulare setzen. Damit kann eine Website ohne JavaScript funktionieren. Für den Fall, dass JavaScript funktioniert, wird aber lediglich die erste Seite via Server Side Rendering generiert. Anschließend sorgen Client Side Rendering und Client Side Routing für die App-artige User Experience einer Single Page Application. Mehrere neue Meta-Frameworks verfolgen dieses Ziel, darunter Remix [8] und SvelteKit [9].

Als Beispiel zeigt Listing 4 eine React-Komponente im Meta-Framework Remix. Bei der Komponente handelt es sich um ein Formular, um einen neuen Eintrag auf eine To-do-Liste zu setzen. Das Formular benutzt eine <*Form*>-Komponente, die dem klassischen HTML-<*form*>-Tag nachempfunden ist [10]. Remix bietet über *useTransition()* einfach die Möglichkeit, den Zustand beim Submit des Formulars zu berücksichtigen. Während der Submit-Request läuft, wird ein anderer Text auf dem Button gezeigt. Listing 5 zeigt das Resultat im gerenderten HTML-Dokument. Aus der <*Form*>-Komponente wird ein HTML-<*form*>-Tag. Wenn JavaScript nicht fehlerfrei geladen wird, funktioniert das Formular trotzdem – ein HTML-Formular benötigt kein JavaScript. Wenn JavaScript jedoch geladen wird und fehlerfrei läuft, dann wird das Formular als „Progressive Enhancement" verbessert und Benutzer:innen bekommen direktes Feedback.

Mit Progressive Enhancement als Grundprinzip für Meta-Frameworks wie Remix und SvelteKit können Single Page Applications eine Resilienz erreichen, die zuvor Multi Page Applications vorbehalten war. Die Frameworks erreichen damit einen Vorteil gegenüber anderen populären Meta-Frameworks – es gibt jedoch keine fundamentalen Hürden für andere Frameworks, den Ansatz zu übernehmen.

Bessere SPA mit (Remix) Data Fetching

In Single Page Applications setzen sich Seiten üblicherweise aus einer Vielzahl von Komponenten zusammen. Die Elemente auf der Seite werden im Quellcode als kleine, wiederverwendbare Bausteine umgesetzt, die in sich wiederum andere Komponenten nutzen können. Die Komponenten können Markup, Verhalten und Tests der Elemente gruppieren – ideal für die Entwicklung. Eine Frage dabei ist, wo die Daten für eine Komponente herkommen.

Oft werden die Daten für einen Block auf der Webseite direkt in der Komponente geholt, die den Block repräsentiert – über einen API-Aufruf beim Laden der Komponente [11]. Der Ansatz kann die Performance der Seite beeinträchtigen: Potenziell muss zunächst JavaScript für die Seite heruntergeladen, geparst und ausgeführt werden; ein API-Aufruf wird erst gestartet, wenn die entsprechende Komponente geladen wird. Beim Öffnen einer Webseite sieht man dann zunächst häufig ein oder mehrere Spinner, die darauf hindeuten, dass die Komponenten auf Daten aus einem API-Aufruf warten. Zudem kann das Holen der Daten zu mehr clientseitigem JavaScript führen. Die Komponenten benötigen teilweise zusätzliche Bibliotheken, um die APIs zu benutzen und die Antworten auf die API-Aufrufe zu bearbeiten. Zum Beispiel könnte eine Applikation Daten von einem Content-Management-System (CMS) abfragen, das ein GraphQL-API vorschreibt und Daten im Markdown-Format liefert. In diesem Fall müsste die client- ▶

Listing 5: Formular im HTML-Dokument

```
<form method="post">
  <input type="text" name="todo" />
  <button type="submit">Create</button>
</form>
```

seitige Applikation potenziell große Bibliotheken einbinden, die mit GraphQL und Markdown umgehen können.

Neue Meta-Frameworks wie Remix bieten eine Alternative an, um Daten für eine Seite zu holen: die Trennung des Data Fetching vom Laden der Komponenten. In Remix können die Routen eine Loader-Funktion implementieren, die Daten für die Route lädt. Diese Funktion wird beim Öffnen der Route aufgerufen, bevor die Komponenten der Zielseite geladen werden. Die Funktion wird auf dem Server ausgeführt. Die clientseitige SPA bekommt dann die Daten, die die Loader-Funktion zurückgibt, und die Komponenten können beim Laden auf die Daten zurückgreifen. Listing 6 zeigt eine Loader-Funktion, die das Data Fetching aus einer Datenbank vom Laden der Komponente trennt – die Implementierung beider Bausteine bleibt nah beisammen in derselben Datei. [12]

Aus dem stärkeren Einbeziehen des Servers ergeben sich mehrere Vorteile:

- **Weniger JavaScript und Zugangsdaten auf dem Client:** Die Loader-Funktion wird auf dem Server ausgeführt. Der Code und die Bibliotheken, die benutzt werden, um die Daten zu holen und zu präparieren, werden nicht auf dem Client benötigt. Im vorigen Beispiel käme die clientseitige Applikation ohne GraphQL- und Markdown-Bibliotheken aus. Zudem werden die Datenquellen und ihre Zugangsdaten nicht dem Client exponiert.
- **Weniger Spinner:** Das Data Fetching kann beginnen, bevor die Komponenten auf der Seite geladen werden, ohne auf das Downloaden, Parsen und Ausführen von JavaScript zu warten. Damit lassen sich Ladezustände vermeiden. Beim initialen Laden einer Seite kann das mit Server Side Rendering generierte HTML-Dokument bereits alle Daten enthalten.
- **Client-Seite einfacher:** Da der Server eine größere Rolle spielt, kann die Client-Seite einfacher werden. Die clientseitige Applikation benötigt weniger Ladezustände und Zustandsdaten, da die Daten bei jedem Seitenwechsel vom Server geholt werden können und dann verfügbar sind, wenn die Komponenten geladen werden.

Listing 6: Remix Data Fetching

```
export const loader: LoaderFunction =
    async ({ context }) => {
  const todos = await getToDos(context);
  // Lädt Daten aus einer Datenbank

  return json <LoaderData> ({ todos });
};

export default function ToDos() {
  const { todos } = useLoaderData();
  // Greift auf zuvor geladene Daten zu

  return (
    <ul>
      {todos.map((todo: ToDoContent) => ...)}
    </ul>
  );
}
```

Remix-Applikationen sind noch nahe an klassischen Single Page Applications mit Client Side Rendering und Routing. Das mentale Modell beim Data Fetching macht aber einen deutlichen Schritt in Richtung Multi Page Application: Der Server liefert Daten beim Seitenwechsel. Damit werden Defizite der klassischen Single Page Application adressiert. Die Menge an JavaScript auf dem Client und die Komplexität der clientseitigen Applikation werden reduziert.

Bessere SPA mit (React) Server Components

Das React-Core-Team arbeitet seit mehreren Jahren an einem Hybridansatz zwischen Single Page Application und Multi Page Application – React Server Components. Das Ziel von Server Components ist es, Arbeit teilweise auf den Server zu verlagern und dadurch die Menge an JavaScript auf dem Client zu reduzieren. Die Idee dabei ist, dass nicht interaktive Komponenten als Server Components deklariert werden können. Die Server Components werden dann auf dem Server gerendert – der Client benötigt kein JavaScript, um die Komponenten rendern zu können. Insbesondere bei komplexen Komponenten, die beispielsweise zusätzliche Bibliotheken verwenden, um Daten zu holen und anzuzeigen, ergibt sich großes Einsparpotenzial. [13]

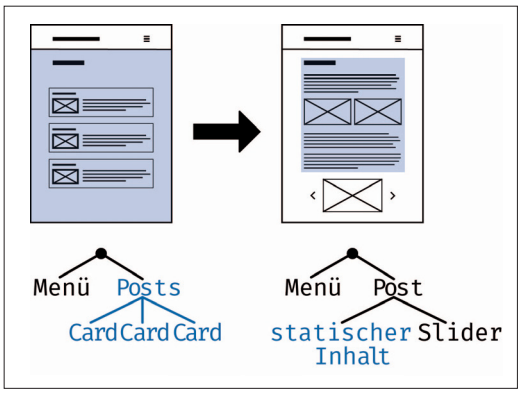
Das Konzept von Server Components ist eng verbunden mit dem Thema Routing. Bei der Navigation von einer Seite zu einer anderen benötigt die Single Page Application die gerenderten Server Components der Zielseite. Das heißt, dass eine Single Page Application mit Server Components bei der Navigation mit dem Server kommunizieren muss und von diesem gerenderte Daten erhält – fast wie bei einer Multi Page Application! [14]

Anders als bei einer Multi Page Application liefert der Server bei React Server Components keine HTML-Dokumente zurück. Eine React-Applikation arbeitet bei der Ausführung auf dem Client mit einem Virtual Document Object Model (VDOM), das den Zustand des User Interface im Speicher repräsentiert [15]. Bei React Server Components erhält die clientseitige Applikation bei der Navigation ein neues VDOM vom Server, das die gerenderten Komponenten in einem serialisierten, React-spezifischen Format enthält. React kann dann auf Clientseite das neue VDOM mit dem bisherigen VDOM „mergen“. Komponenten, die bereits im bisherigen VDOM präsent waren, werden dabei nicht überschrieben – sie behalten Zustand, Fokus und laufende Animationen. [16]

Bild 1 veranschaulicht das Konzept am Beispiel eines Blogs. Die Website bietet eine Seite zur Übersicht verschiedener Blog-Posts als Liste von Cards und Seiten für die Artikel. Die Seite besteht größtenteils aus statischen Inhalten mit Text und Bildern in einem fixen Layout. Zusätzlich gibt es zwei interaktive Elemente: ein interaktives Menü und einen Slider, der zwischen verschiedenen Bildern navigiert. Die Übersichtsseite ist bis auf das Menü vollständig statisch. Die Posts-Kompo-

nente und ihre Unterkomponenten können als Server Components umgesetzt werden. Auf der Artikelseite gibt es dagegen noch die interaktive Slider-Komponente. Dort kann nur der statische Teil des Artikels als Server Components implementiert werden. Bei der Navigation von der Übersichtsseite zum Artikel erhält die clientseitige Applikation vom Server ein serialisiertes VDOM mit den statischen Inhalten in gerenderter Form. Das Menü ändert sich nicht und behält seinen Zustand. Die clientseitige Applikation kann die Post-Komponenten in ihr bestehendes VDOM einfügen. Sie benötigt das JavaScript der Post- und Slider-Komponenten. Der Code, um die statischen Inhalte zu rendern, bleibt auf dem Server.

Mit Server Components macht React einen großen Schritt in Richtung Multi Page Application. Das Routing-Modell, bei dem der Server beim Seitenwechsel eine Repräsentation der Zielseite rendert, geht in Richtung von Server Side Rendering und Server Side Routing. Durch Server Components lässt sich die Menge an JavaScript auf dem Client reduzieren. Dies unterstützt die Performance beim Laden von Seiten und adressiert eines der Probleme von klassischen Single Page Applications – zumindest teilweise. Darüber hinaus können Server Components ähnliche Vorteile bringen wie der Remix-Ansatz zum Data Fetching: weniger Zugangsdaten auf dem Client, weniger Spinner, und die Client-Seite wird einfacher. Dennoch ist eine Single Page Application mit Server Components nach wie vor eine Single Page Application mit den damit verbundenen Vor- und Nachteilen.



Konzept von Server Components und VDOM (Bild 1)

Bessere MPA mit modernen Browsern

Die Weiterentwicklung des Webs gestaltet sich nicht nur über Frameworks – auch Webbrowser und -standards werden stets weiterentwickelt. Neue Webstandards und Browserkonventionen können Defizite der beiden grundlegenden Architekturstile angehen und Komplexität reduzieren, indem sie Funktionen direkt in den Browser integrieren, die sonst als Teil der Applikation gebaut werden müssten. Hier einige Beispiele für vergangene und zukünftige Verbesserungen:

- **Paint Holding:** In der Vergangenheit zeigte Chrome beim Laden von Seiten zuerst eine weiße Seite an. Dadurch bekamen Benutzer:innen schnell Rückmeldung, dass eine neue Seite geladen wird. Bei der Navigation zwischen Seiten derselben Website führte das jedoch zu Unruhe und gefühlt schlechter Performance. Seit 2019 benutzt Chrome „Paint Holding“, um den Effekt zu vermeiden. Während die neue Seite lädt, wird für kurze Zeit weiter die alte Seite angezeigt. Der Zwischenzustand der weißen Seite wird vermieden. [17]
- **Back/Forward Cache:** Moderne Browser benutzen einen Back/Forward Cache, der den Zustand von Seiten speichert, wenn man von den Seiten wegnavigiert. Beim Benutzen der Vor- und Zurück-Knöpfe des Browsers können die Seiten dann unmittelbar wieder im alten Zustand angezeigt werden und müssen nicht neu geladen werden. [18]
- **Shared Element Transition API:** Durch den neuen Standard für Übergänge zwischen Seiten wird es zukünftig möglich sein, komplexe Transitionen mit wenig Code deklarativ per CSS umzusetzen. So wird es beispielsweise einfach möglich sein, Elemente, die auf beiden Seiten vorkommen, durch eine Animation zu verbinden, sodass das ursprüngliche Element nahtlos in das neue Element übergeht. Der Standard soll sowohl Single Page Applications als auch Multi Page Applications unterstützen. [19]
- **Navigation API:** Das neue API richtet sich insbesondere an Single Page Applications und löst Probleme des alten History API. Damit kann die Barrierefreiheit verbessert werden: Screenreader können besser erkennen, wenn eine Navigation ausgeführt wird. Zudem können Webkonventionen wie das Wiederherstellen der Scroll-Position nach Betätigen des Zurück-Knopfes besser umgesetzt werden. [20]

Die Fortschritte bei Webbrowsern und -standards verändern die Betrachtung der Vor- und Nachteile von Single Page Applications und Multi Page Applications.

Mit neuen Standards wie dem Navigation API lässt sich der bisherige Nachteil von Single Page Applications in Bezug auf Barrierefreiheit und Webkonventionen ausgleichen. Mit modernen Browsern und Meta-Frameworks, die neue Standards benutzen, können auch Single Page Applications Webkonventionen einhalten und Barrierefreiheit unterstützen.

Mit Browser-Features wie Paint Holding und Back/Forward Cache sowie neuen Standards wie dem Shared Element Transition API können Multi Page Applications die App-artige User Experience von Single Page Applications erreichen. Eine performante Multi Page Application kann schnelle, nahtlose Seitenwechsel erreichen, und zukünftig werden selbst komplexe Transitionen zwischen den Seiten möglich.

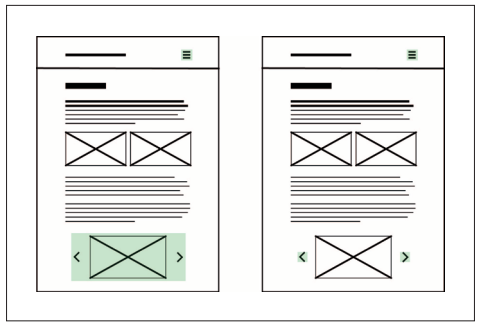
Bessere MPA mit modernen Frameworks

Eine neue Generation von Frameworks erkundet die Zukunft des Frontends vom Ausgangspunkt der klassischen Multi Page Application. Im Fokus liegen dabei Developer Experience und Performance.

Die Developer Experience von klassischen Multi Page Applications ist im Vergleich zu beliebten Komponenten- und Meta-Frameworks im Nachteil. Eine große Hürde dabei ist der Technologiebruch zwischen statischen Inhalten und den interaktiven Elementen, die mit JavaScript umgesetzt werden. Daraus ergeben sich zwei „Applikationen“ mit unterschiedlichen Technologien, die über das Markup der Seiten gekoppelt sind. Neue JavaScript-basierte Frameworks vereinheitlichen die beiden Aspekte. Dazu kommen umfangreiche Wizards zum ▶

Aufsetzen von Projekten, lokales Entwickeln mit schnellen Feedback-Zyklen und Integrationen mit gängigen Plattformen und Services.

Um schnelle Navigation zwischen Seiten zu ermöglichen, sind Multi Page Applications auf hohe Performance beim Laden der Seiten angewiesen. Kritisch dabei ist die Menge an JavaScript [4]. Webseiten kommen nur selten ohne interaktive Elemente aus. Daher laden Multi Page Applications in der Regel für jede Seite JavaScript – bei jeder Navigation. Bei Server Side Rendering benutzen JavaScript-Frameworks klassischerweise das Konzept „Hydration". Der Server führt zunächst das JavaScript der Komponenten aus, um sie zu rendern und als Teil des HTML-Dokuments an den Client zurückzugeben. Der Browser kann die Komponenten damit anzeigen, aber sie sind noch nicht interaktiv. Der Browser muss zunächst das JavaScript für die Komponenten herunterladen, parsen und ausführen, um ihren Zustand zu initialisieren und Eventhandler zu registrieren. Erst wenn die Komponenten „hydriert" sind, reagieren sie auf Eingaben der Benutzer:innen. Für die bestmögliche Performance der Website sollte die Menge an JavaScript möglichst gering sein. Die neuen Frameworks setzen auf Techniken, um diese Menge zu reduzieren. [21]



Partial Hydration auf Islands-Ebene und Subkomponenten-Ebene [43] (Bild 2)

Fortgeschrittene Hydration-Techniken sind:

- **Partial Hydration:** Der Begriff Partial Hydration bezeichnet das Konzept, nur das JavaScript zu laden, das für die interaktiven Elemente auf einer Seite tatsächlich nötig ist. Eine einfache Variante davon beschränkt JavaScript auf interaktive „Inseln" – grobe Komponenten auf der Seite – nach dem Konzept „Islands Architecture" [22]. In der fortgeschrittenen Variante erkennen Frameworks auf dem Subkomponenten-Level, welcher Code tatsächlich auf dem Client benötigt wird, um die Interaktivität zu gewährleisten. Bild 2 zeigt die beiden Variationen von Partial Hydration am Beispiel eines Blog-Posts. Die Seite besteht größtenteils aus statischen Inhalten mit Text und Bildern in einem fixen Layout. Zusätzlich gibt es zwei interaktive Elemente: ein interaktives Menü und einen Slider, der zwischen verschiedenen Bildern navigiert. Mit dem Islands-Ansatz wird die komplette Slider-Komponente hydriert. Feingranulare Hydration-Ansätze auf Subkomponenten-Ebene können bei einer Slider-Komponente vermeiden, das JavaScript der Komponente vollständig an den Client zu senden – sie können innerhalb der Komponente erkennen, welcher Code zur Interaktivität auf dem Client gebraucht wird [21].
- **Progressive Hydration:** Die Idee von Progressive Hydration ist, nicht alles JavaScript sofort beim Laden der Seite auszuführen. Wenn beispielsweise Elemente außerhalb des sichtbaren Bereichs zunächst nicht hydriert werden, benötigt das initiale Laden der Seite weniger JavaScript. Die Elemente können später hydriert werden, wenn der initiale Ladevorgang abgeschlossen ist oder wenn die Elemente sichtbar werden. [21]
- **Resumability:** Das Ziel von Resumability ist es, keinen Code auf dem Client auszuführen, der bereits auf dem Server ausgeführt wurde. Beim klassischen Hydration-Ansatz führt der Server zuerst JavaScript aus, um Komponenten zu rendern. Anschließend führt der Client das gleiche JavaScript erneut aus, um sie interaktiv zu machen. Das Konzept von Resumability ist es dagegen, dass der Server den Zustand nach dem Rendern an den Client übergibt und dieser von dort aus fortfahren kann, ohne Arbeit doppelt zu verrichten. Dazu wird der Zustand auf dem Server serialisiert und als Teil des HTML-Dokuments ausgegeben. Dadurch wird nur wenig JavaScript auf dem Client ausgeführt und die Website erreicht eine hohe Performanz mit niedriger Time to Interactive. [23]

Bemerkenswerte Frameworks aus diesem Bereich sind beispielsweise Astro, Marko und Qwik:

- **Astro:** Ein Framework für Multi Page Applications mit erstklassiger Developer Experience und hoher Performance durch Partial Hydration mit Islands-Konzept. Statische Teile von Seiten werden über Astro-Komponenten implementiert. Diese sind ähnlich zu Komponenten in Frameworks wie React. Sie bieten eine JSX-artige Syntax und Konzepte wie Props und Slots. Interaktive Teile von Seiten werden mit beliebten Komponenten-Frameworks umgesetzt – bestehende Komponenten kann man direkt mitbringen. Bisher unterstützt Astro folgende Komponenten-Frameworks: React, Vue.js, Svelte, SolidJS, Preact und Lit. Es gibt immer noch einen Bruch zwischen statischen und interaktiven Elementen – zwischen Astro-Komponenten und solchen aus ande-

Listing 7: Astro-Komponente

```
---
import Header from './Header.astro';
import Footer from './Footer.astro';
import VueComponent from '../components/Component.vue';
import SvelteComponent from '../components/
  Component.svelte';

const { title } = Astro.props
---
<div id="content-wrapper">
  <Header />
  <h1>{title}</h1>
  <VueComponent client:load />
  <SvelteComponent client:visible />
  <Footer />
</div>
```

Listing 8: Counter-Komponente in Marko

```
<let/count = 0 />
<div>${count}</div>
<button onClick() { count++ }>
  Click me!
</button>
```

ren Frameworks. Das ist aber nicht vergleichbar mit klassischen Multi Page Applications. Es gibt eine einzige Applikation und Codebasis. Die statischen und interaktiven Elemente basieren alle auf JavaScript und sind sich ähnlich. Astro hat vor Kurzem Version 1.0 erreicht. Listing 7 zeigt eine beispielhafte Astro-Komponente [24]. Die Komponente verwendet zwei weitere Astro-Komponenten, eine Vue.js-Komponente und eine Svelte-Komponente. Der Titel wird über Props in die Komponente gegeben. Die Vue.js-Komponente wird über *client:load* explizit beim Laden der Seite hydriert. Die Svelte-Komponente wird über *client:visible* explizit erst hydriert, wenn die Komponente sichtbar ist. Dadurch wird beim initialen Laden möglichst wenig JavaScript geladen [25].

- **Marko:** Ein unterbewertetes Framework von eBay für eBay. Die Arbeit an Marko startete bereits 2012 [26]. Seit Jahren laufen mehr als eine Milliarde Requests pro Tag über Marko [27]. E-Commerce-Giganten wie eBay setzen auf möglichst performante Multi Page Applications. Daher nutzt Marko bereits seit Jahren Optimierungen wie Partial Hydration und Streaming [28]. Aktuell arbeitet eBay an Version 6 des Frameworks, die erneut große Fortschritte machen wird: Mit Resumability und Partial Hydration auf Subkomponenten-Level wird so wenig JavaScript wie möglich an den Client gesendet. Damit ist Marko eines der technisch fortschrittlichsten Frameworks und auch klar produktionsreif. Statische und interaktive Elemente sind in Marko komplett einheitlich ohne Technologiebruch. Bisher sehen Marko-Komponenten ähnlich aus wie Komponenten der populären Komponenten-Frameworks wie React. Marko 6 wird zusätzlich ein neues Tags API einführen, das Skripte in Tag-Syntax zulässt. Listing 8 zeigt eine Counter-Komponente mit Tags API. Die Variable *count* wird direkt beim Markup in einem <*let*>-Tag definiert. Diese Syntax unterscheidet sich von populären Frameworks, bietet aber dafür zusätzliche Flexibilität und Optimierungsmöglichkeiten. [29][30][31][23]
- **Qwik:** Ein neues Framework von Miško Hevery, dem Schöpfer von Angular. Wie Marko reduziert auch Qwik die Menge an JavaScript auf dem Client durch Resumability und Partial Hydration auf Subkomponenten-Ebene. Anders als Marko setzt Qwik zudem standardmäßig auf eine extreme Form von Progressive Hydration. Beim initialen Laden der Seite wird nur ein Minimum an JavaScript ausgeführt. Das JavaScript für die Interaktivität wird erst geladen, wenn der Ladevorgang beendet ist oder die Benutzer:innen mit Elementen interagieren. Ein Vorteil beim Umstieg von React auf Qwik: Die Komponenten sehen sich ähnlich: „You know React? You know Qwik“ [32]. Listing 9 zeigt einen Counter mit Qwik [33]. Qwik und das dazugehörige Meta-Framework Qwik City verfolgen ambitionierte Ziele. Die Idee ist, dass das Framework von einer Multi Page Application bis hin zu einer Single Page Application skalieren kann. Das Framework hat vor Kurzem die Beta-Phase erreicht [34]. [32][35][36][37]

Die neue Generation von Frameworks macht Multi Page Applications wieder attraktiv. Sie machen große Fortschritte bei der Developer Experience und der Performance beim Seitenwechsel, und sie vermeiden dabei die höhere Komplexität von Single Page Applications. Darüber hinaus erlauben sie einen einfachen Umstieg, bei dem Entwickler:innen ihre Expertise in den populären Komponentenframeworks einbringen können.

Bessere MPA mit HTML-Fragmenten

Ein weiterer Ansatz, um Multi Page Applications zu verbessern, setzt auf HTML-Fragmente. Hierbei liefert der Server primär HTML-Dokumente – nicht nur für volle Seiten, sondern auch für Fragmente auf der Seite. So kann eine Interaktion mit einem Fragment zu einem Server-Request führen, der mit einer neuen Version des Fragments antwortet. Mit einem leichtgewichtigen JavaScript-Skript wird das neue Markup dann anstelle des alten eingefügt. Der Rest der Seite behält seinen Zustand. Diverse Frameworks verfolgen diesen Ansatz, darunter Turbo [38] und HTMX [39]. Die Firma 37signals prägte den Begriff „Hotwire“ – HTML over the wire – für das Konzept [40].

Listing 10 zeigt einen Ausschnitt aus einem HTML-Dokument, in dem Turbo zum Einsatz kommt. Ein Fragment auf der Seite zeigt eine Liste von bestehenden To-dos an und bietet ein Formular, um ein weiteres To-do hinzuzufügen. Das Frag- ▶

Listing 9: Counter-Komponente in Qwik

```
import { component$, useStore } from '@builder.io/
  qwik';

export const App = component$(() => {
  const store = useStore({ count: 0 });

  return (
    <div>
      <h1>My Counter</h1>
      <span>{store.count}</span>
      <button onClick$={() => store.count++}>+</button>
    </div>
  );
});
```

ment liegt in einem *<turbo-frame>*-Tag mit der ID *todos*. In einer klassischen MPA würde das Absenden des Formulars die Seite komplett neu laden – inklusive Skripten. Wenn jedoch das schlanke Turbo-Skript geladen ist, verhindert das Skript das Laden der vollen Seite. Turbo führt den Request anstelle des Browsers aus. Beim Absenden des Formulars erkennt Turbo, dass sich der Request auf das Turbo-Frame bezieht und nur der Inhalt des Frames ausgetauscht werden soll. Der Server antwortet mit einem HTML-Dokument, das wieder das *<turbo-frame>*-Tag mit der ID *todos* enthält. Turbo ersetzt das originale Turbo-Frame durch jenes aus der Antwort des Servers. [41]

Aus der Perspektive des Servers arbeitet die Website wie eine klassische Multi Page Application. Die Turbo-Bibliothek gibt ihr dagegen Charakteristiken einer Single Page Application – statt voller Page Loads wird die bestehende Seite mittels JavaScript manipuliert. Die Website kann weitgehend auf JavaScript verzichten und die Komplexität klassischer Single Page Applications vermeiden. Durch den Fokus auf HTML statt JavaScript werden Seiten und Fragmente schnell geladen.

Der Ansatz hat aber nicht nur Vorteile. Zum einen wird die Barrierefreiheit potenziell beeinträchtigt: Wie bei einer klassischen Single Page Application ist nicht sicher, dass Fokus-Management und Navigation für Screenreader angemessen sind [42]. Zum anderen ergibt sich stets eine Latenz, wenn Seiten und Fragmente vom Server angefordert werden – klassische SPAs können auf Interaktion sofort reagieren [2].

Fazit

Die Landschaft der Frontend-Architektur ist im Wandel. Es gibt ein Umdenken entgegen dem Trend zu klassischen Single Page Applications, um deren Defizite auszugleichen. Aktuelle Trends versprechen mehr Resilienz, weniger clientseitiges JavaScript und weniger Komplexität. Der Server spielt wieder eine größere Rolle und unterstützt insbesondere beim Routing.

Listing 10: To-do-Liste mit Turbo

```
<body>
  <header>...</header>
  <main>
    <turbo-frame id="todos">
      <h1>My ToDo's</h1>
      <div id="todo-1">ToDo 1</div>
      <div id="todo-2">ToDo 2</div>

      <form action="/todos">
        <input type="text" name="todo" />
        <button type="submit">Create</button>
      </form>
    </turbo-frame>
  </main>
  <aside>...</aside>
  <footer>...</footer>
</body>
```

Auf der anderen Seite des Spektrums bahnt sich ein Wiederaufschwung von Multi Page Applications an. Mit neuen Frameworks erreichen MPAs konkurrenzfähige Developer Experience: Sie erlauben die zusammenhängende Entwicklung von interaktiven und nicht interaktiven Elementen mit derselben Technologie in derselben Codebasis. Mit effizienten Frameworks, Webstandards und modernen Browsern können Multi Page Applications potenziell die App-artige User Experience von Single Page Applications erreichen.

Wird das Frontend der Zukunft eher auf Basis von Single Page Applications entstehen, oder bieten Multi Page Applications einen einfacheren Weg zum Ziel? Es lässt sich nicht absehen, welche Seite sich langfristig durchsetzt, und ob überhaupt eine Seite gewinnen kann. Für neue Frontend-Projekte gibt es daher keine einfachen Antworten auf die Frage nach dem Architekturstil. Je nach Anwendungsfall kann die eine oder die andere Seite des Spektrums vorteilhaft sein. Es gilt, wie so oft, die Vor- und Nachteile aller Optionen für den eigenen Kontext abzuwägen und die Entscheidung bewusst zu treffen. ■

[1] Thoughtworks, SPA by default, www.dotnetpro.de/SL2212WebFrontend1

[2] R. Harris on YouTube, Have Single-Page Apps Ruined the Web?, www.dotnetpro.de/SL2212WebFrontend2

[3] npm trends, www.dotnetpro.de/SL2212WebFrontend3

[4] T. Kadlec, The Cost of Javascript Frameworks, www.dotnetpro.de/SL2212WebFrontend4

[5] N. Lawson, The balance has shifted away from SPAs, www.dotnetpro.de/SL2212WebFrontend5

[6] T. MacWright, Second-guessing the modern web, www.dotnetpro.de/SL2212WebFrontend6

[7] D. Abramov auf Twitter, www.dotnetpro.de/SL2212WebFrontend7

[8] Remix, Philosophy, www.dotnetpro.de/SL2212WebFrontend8

[9] SvelteKit, www.dotnetpro.de/SL2212WebFrontend9

[10] Remix, Data Writes, www.dotnetpro.de/SL2212WebFrontend10

[11] R. Florence auf Youtube, Remixing React Router, www.dotnetpro.de/SL2212WebFrontend11

[12] Remix, Data Loading, www.dotnetpro.de/SL2212WebFrontend12

[13] Introducing Zero-Bundle-Size React Server Components, www.dotnetpro.de/SL2212WebFrontend13

[14] R. Carniato, The Return of Server Side Routing, www.dotnetpro.de/SL2212WebFrontend14

[15] React, Virtual DOM and Internals, www.dotnetpro.de/SL2212WebFrontend15

[16] RFC: React Server Components, www.dotnetpro.de/SL2212WebFrontend16

[17] Chrome Developers, Paint Holding, www.dotnetpro.de/SL2212WebFrontend17

[18] web.dev, Back/forward cache, www.dotnetpro.de/SL2212WebFrontend18

[19] Chrome Developers, Shared element transition API, www.dotnetpro.de/SL2212WebFrontend19

[20] WICG, Navigation API, www.dotnetpro.de/SL2212WebFrontend20
[21] R. Carniato, Why Efficient Hydration in JavaScript Frameworks is so Challenging, www.dotnetpro.de/SL2212WebFrontend21
[22] JSON Format, Islands Architecture, www.dotnetpro.de/SL2212WebFrontend22
[23] R. Carniato, Resumability, WTF?, www.dotnetpro.de/SL2212WebFrontend23
[24] Astro Docs, Components, www.dotnetpro.de/SL2212WebFrontend24
[25] Astro, Introducing Astro: Ship Less JavaScript, www.dotnetpro.de/SL2212WebFrontend25
[26] P. Steele-Idem, The Future of Marko, www.dotnetpro.de/SL2212WebFrontend26
[27] P. Steele-Idem, Why is Marko Fast?, www.dotnetpro.de/SL2212WebFrontend27
[28] P. Steele-Idem, Async Fragments, www.dotnetpro.de/SL2212WebFrontend28
[29] R. Carniato, Introducing the Marko Tags API Preview, www.dotnetpro.de/SL2212WebFrontend29
[30] Marko, www.dotnetpro.de/SL2212WebFrontend30
[31] YouTube, D. Piercey, M. Rawlings, Building Marko 6, www.dotnetpro.de/SL2212WebFrontend31
[32] Qwik, www.dotnetpro.de/SL2212WebFrontend32
[33] Qwik, Counter, www.dotnetpro.de/SL2212WebFrontend33
[34] Builder.io, Qwik and Qwik City have reached beta!, www.dotnetpro.de/SL2212WebFrontend34
[35] A first look at Qwik – the HTML first framework, www.dotnetpro.de/SL2212WebFrontend35
[36] M. Hevery, HTML-first, JavaScript last: the secret to web speed!, www.dotnetpro.de/SL2212WebFrontend36
[37] M. Hevery, Qwik: the answer to optimal fine-grained lazy loading, www.dotnetpro.de/SL2212WebFrontend37
[38] Turbo, https://turbo.hotwired.dev
[39] HTMX, https://htmx.org
[40] Hotwire, https://hotwired.dev
[41] Decompose with Turbo Frames, www.dotnetpro.de/SL2212WebFrontend38
[42] T. Hunt, Routing: I'm not smart enough for a SPA, www.dotnetpro.de/SL2212WebFrontend39
[43] M. Rawlings, Maybe you don't need that SPA, www.dotnetpro.de/SL2212WebFrontend40

Philip Schmitt
ist Senior Consultant bei TNG Technology Consulting und interessiert sich für sämtliche Facetten von Software-Engineering, Softwarearchitektur, Benutzererfahrung und Design. Sie erreichen ihn unter @philipschm1tt oder **www.linkedin.com/in/philipschm1tt**.

dnpCode	A2212WebFrontend