Rendering Patterns in der Webentwicklung – Teil 1

Spektrum der Rendering Patterns

SSG, SSR, CSR, ISR, Prerendering, Hydration, Streaming, Resumability. Einige dieser Begrifflichkeiten begegnen einem Entwickler, wenn er oder sie sich mit der Frage beschäftigt, wie Inhalte im Web geladen und dargestellt werden können. Vielen ist dabei nicht klar, was sich dahinter verbirgt und welche Stärken bzw. Schwächen die Patterns haben. Bringen wir Licht ins Dunkel.

von Julian Schäfer

Wie im Technology Radar von thoughtworks vermerkt [1] wird zu Projektbeginn oftmals direkt ein Framework ausgewählt. Diese trendgetriebene Entscheidung kann bereits gravierende Auswirkungen auf die Entwicklung der Anwendung haben. Der Schritt der Validierung, welcher Architekturansatz für die jeweilige Anwendung der passendste ist, wird dabei in der Regel übersprungen.

Artikelserie

Teil 1: Das Spektrum der Rendering Patterns
Teil 2: Fortgeschrittene Patterns im Detail
Teil 3: Zurück zum Server mit neuartigen Patterns

Ein wichtiger Aspekt einer Anwendungsarchitektur ist es, wie Daten performant geladen und angezeigt (gerendert) werden können. Wie die Begrifflichkeiten im einleitenden Satz sowie die diesjährige Umfrage der State of JS [2] (**Abb. 1**) zeigen, gibt es dafür zahlreiche Möglichkeiten.

Doch welche sind das? Wie unterscheiden sie sich voneinander und sind diese Unterschiede auch messbar? Welche Vor- bzw. Nachteile hat das jeweilige Pattern? In dieser Artikelserie werden die in der Meinung des Autors kennenswertesten Patterns beleuchtet und es wird versucht, die gestellten Fragen zu beantworten.

Um das Spektrum der Patterns nachvollziehen zu können, blicken wir anfangs zurück auf die Entwicklung des Webs und definieren den Begriff Rendering Pattern genauer. Anschließend betrachten wir die traditionellen Ansätze wie statisches Rendering, clientseitiges und

serverseitiges Rendering. Nachdem diese Grundlagen verstanden wurden, wenden wir uns den fortgeschrittenen Patterns zu und wagen auch einen Blick auf neue Trends. Am Ende dieser Artikelserie soll ein grundlegendes Verständnis jedes Patterns aufgebaut worden sein, damit dessen Auswirkungen und Einsatzmöglichkeiten bewertet und untereinander diskutiert werden können.

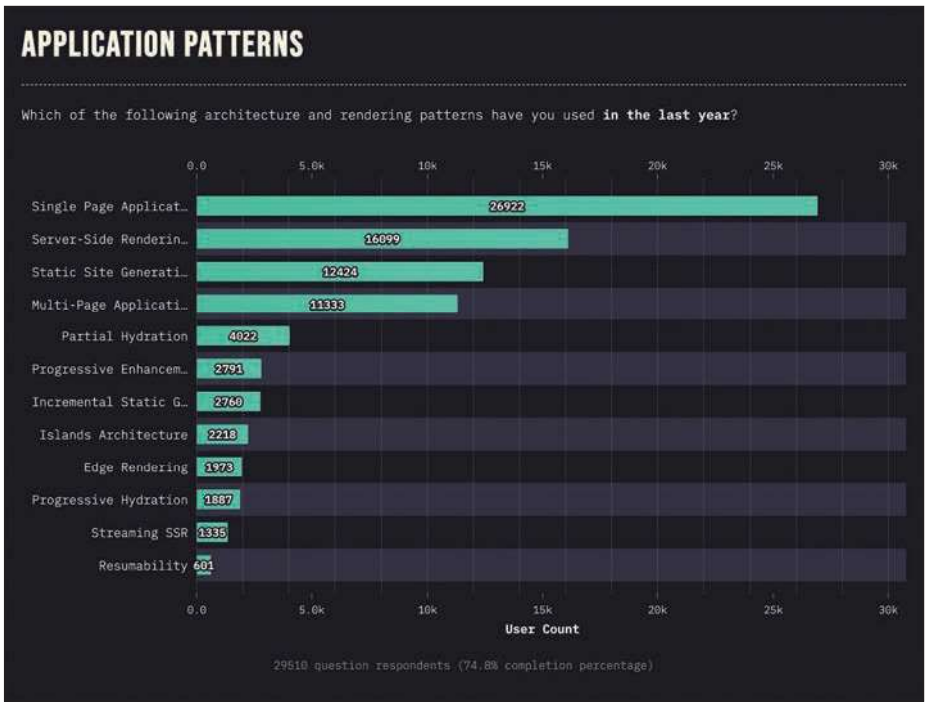

Abb. 1. Nur wenige kennen sich mit der Vielzahl der Patterns aus

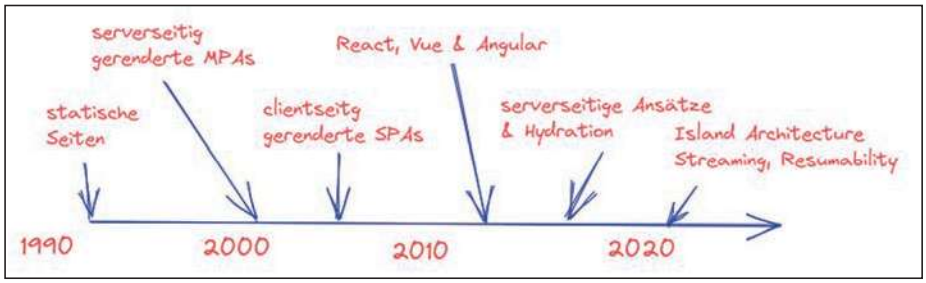

Abb. 2: Das Web hat eine bemerkenswerte Entwicklung hinter sich

Die Entstehung der Patterns

Um die jetzige Entwicklung des Webs und die daraus resultierenden Rendering Patterns besser zu verstehen, lohnt es sich, einen Blick in dessen Vergangenheit (**Abb. 2**) zu werfen.

In den 90er Jahren war das Web noch sehr einfach gestrickt und bestand überwiegend aus statisch gerenderten Inhalten. Eines der bekanntesten Beispiele ist die Webseite für den Film „Space Jam“ [3].

Um die Jahrtausendwende rückten anstatt einfacher Webseiten vermehrt Webapplikationen in den Fokus. Dazu musste das Web dynamischer und persönlicher werden. Für schnelles UI-Feedback wurde clientseitig JavaScript eingesetzt. Die Anzeige und Bearbeitung personalisierter Inhalte erfolgte mit Hilfe von Anfrageparametern und HTML-Formularen, die auf der Serverseite entsprechend interpretiert wurden. Um auf eine andere Seite navigieren zu können (im Folgenden Routing genannt) oder die Daten auf einer Seite anzupassen, muss eine weitere Anfrage an den Server gemacht werden, damit dieser ein neues HTML-Dokument ausliefert. Diesen Ansatz nennt man Multi Page Application (MPA).

Aufgrund des kompletten Roundtrips zwischen Client und Server leidet die User Experience, da die Seite neu geladen werden muss. Somit war UI-Feedback an die Nutzer nur schlecht sichtbar. Beispielsweise konnte man einen Ladevorgang nur anhand der Ladeanimation am Tab sehen, jedoch nicht dort, wo die Aktion vom Benutzer ausgelöst wurde, da bei Anzeige die Seite bereits neu geladen war.

Das änderte sich ab 2005 mit dem Aufkommen von Asynchronous JavaScript and XML (AJAX). Hierdurch konnten erstmals Daten zwischen Client und Server ausgetauscht und aktualisiert werden, ohne die Seite komplett neu laden zu müssen. Damit wurde es möglich, JavaScript nicht nur für die Präsentationslogik, sondern auch für einen Teil der Anwendungslogik zu nutzen. Die erste Generation von Bibliotheken und Frameworks war geboren, darunter etwa jQuery oder Backbone.js. Somit konnte das Routing erstmals komplett auf dem Client stattfinden. Das ermöglichte komplett clientseitig gerenderte Single Page Applications, wie z. B. Gmail oder Google Maps, womit die User Experience einen erheblichen Schritt nach vorne machte.

Jedoch litt diese Generation von Frameworks unter einer schwachen Developer Experience (DX). Sowohl auf dem Client als auch auf dem Server musste ein Teil der Anwendungslogik implementiert werden, wodurch Code dupliziert werden musste und es kein einheitliches mentales Modell mehr gab. Dadurch waren die Anwendungen fehleranfällig und schlecht skalierbar.

Mit dem Aufkommen von React, Vue und Angular ab 2013 wurde die Developer Experience verbessert. Die zweite Generation clientseitiger Frameworks übernahm bewährte Konzepte ihrer Vorfahren. Zusätzlich etablierte sie neue Ansätze wie die deklarative und komponentenbasierte Programmierung, wodurch Skalierungsprobleme und Duplikationen reduziert werden konnten. Daneben wurde das Tooling verbessert, wodurch beispielsweise das Set-up von Projekten mit Hilfe von CLIs vereinfacht wurde. Besonders hervorzuheben ist, dass JavaScript immer stärker in den Fokus rückte und neben dem Routing und der Geschäftslogik bei-

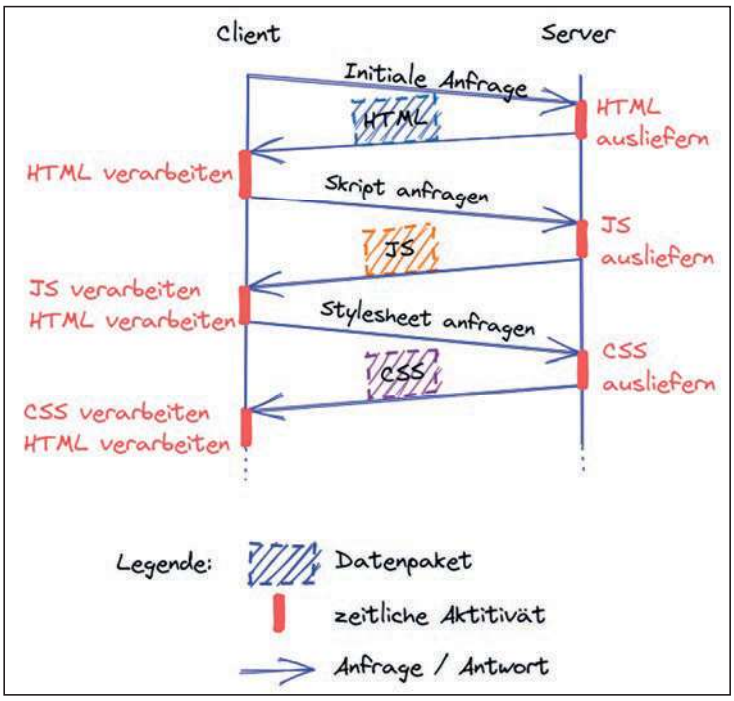

Abb. 3: Während des Renderings werden weitere Ressourcen nachgeladen

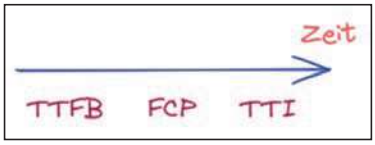

Abb. 4: Zeitliche Abfolge der Web Vitals

spielsweise auch das DOM abgebildet wurde.

Die stetig wachsende Größe und Komplexität des JavaScript-Bundles wirkten sich nachteilig auf die Performance aus, weil immer mehr Daten auf den Client übertragen werden mussten. Damit stieg auch der Aufwand, den Endgeräte für das Parsing und die Interpretation aufwenden mussten. State Management sowie die Wasserfall-Problematik (eine Abfolge sequenzieller Anfragen vom Client zum Server, verursacht durch die Eltern-Kind-Beziehung der Komponenten) waren weitere Probleme.

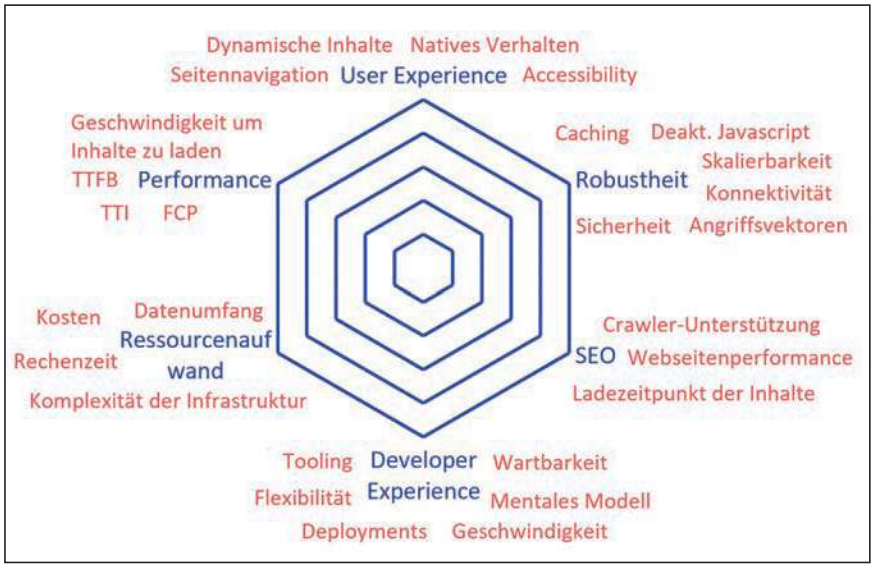

Abb. 5: Bewertung eines Patterns anhand von sechs Schwerpunkten und deren Aspekten

Daher orientieren sich die ursprünglich clientzentrierten Frameworks wieder zurück in Richtung Serverseite. So ist es heutzutage problemlos möglich, React- oder Vue-Apps auf dem Server vorzurendern, an den Client auszuliefern und anschließend als SPA zu initialisieren. Dieser als Hydration bezeichnete Prozess bringt jedoch eigene Herausforderungen mit sich, wie wir im nächsten Teil der Serie sehen werden.

Heutzutage befinden wir uns in einer Übergangsphase zur dritten Generation an Frameworks, die noch ausgefeiltere Ansätze bieten, um diese Problematiken zu lösen. Konzepte wie Streaming, Island Architecture oder Resumability lassen die Grenzen zwischen MPA und SPA zunehmend verschwimmen und erweitern das Spektrum an Patterns. Die Hoffnung besteht, dass am Ende dieser Phase sowohl eine gute User Experience als auch Developer Experience performant in einem Pattern vereinbar werden.

Der Rendering-Prozess

Mit dem Begriff „Rendering“ lässt sich in diesem Kontext erklären, wo und wann Daten abgefragt werden und eine Webseite inklusive dieser Daten generiert bzw. angereichert wird. Dieser Prozess kann entweder auf dem Client oder auf einem Server geschehen. Wie wir später sehen werden, kann es bei Letzterem sogar zu zwei unterschiedlichen Zeitpunkten geschehen: zur Laufzeit oder bereits im Vorfeld. Somit sind Rendering Patterns verschiedene Ansätze, um eine Webseite sowie deren Inhalt im Web zu laden und anzuzeigen.

Um zu verstehen, wie es dazu kommt, dass die Daten im Browser angezeigt werden, schauen wir uns diesen Rendering-Prozess einmal oberflächlich an (**Abb. 3**, Grafikdesign inspiriert durch [4]): Ein Nutzer gibt in seinem Browser einen URL ein. Dieser wird aufgelöst und die Anfrage an den Webserver weitergeleitet. Der Server liefert das HTML-Dokument aus, das vom Browser heruntergeladen und interpretiert wird.

Anschließend traversiert der Browser durch die Elemente des Dokuments und baut daraus das DOM (Document Object Model) auf. Während des ganzen Vorgangs kann der Browser auf weitere verlinkte Assets stoßen, wie beispielsweise Stylesheets, Skripte oder Bilder. Diese versucht er ebenfalls zu laden und anzuwenden.

Durch Stylesheets wird ein CSSOM (CSS Object Model) aufgebaut. Da sowohl DOM als auch CSSOM durch JavaScript manipuliert werden können, kann deren Aufbau pausiert werden, bis das JavaScript geladen, kompiliert und ausgeführt wurde. Anschließend werden DOM und CSSOM zum Render Tree kombiniert. Nachdem dieser aufgebaut wurde, beginnen die Layout- und die Zeichenphase.

Wie wir sehen können, hängt das Rendering im Browser stark von den angefragten

Ressourcen ab. Daher zielen viele Best Practices darauf ab, das Laden dieser Ressourcen so lange wie möglich hinauszuzögern (z. B. durch asynchrones Nachladen der Skripte) bzw. den Umfang der Ressourcen so minimal wie nötig zu halten (z. B. durch kritisches CSS).

Vergleichbarkeit der Patterns

Um Rendering Patterns miteinander vergleichen zu können, können verschiedene Themenblöcke herangezogen werden. Ein wichtiger ist die Performance. Um diese zu messen, können die folgenden drei Web Vitals herangezogen werden (**Abb. 4**):

- TTFB – Time to First Byte
- FCP – First Contentful Paint
- TTI – Time to Interactive

Die TTFB ist vereinfacht gesagt die Zeit, die zwischen dem Anfordern einer Ressource und dem ersten Byte der Antwort des Servers vergeht. Der Zeitpunkt, zu dem der Nutzer den ersten wertvollen Inhalt, beispielsweise Texte oder Bilder, auf dem Bildschirm sieht, wird durch die FCP-Metrik dargestellt. In unserem Fall wäre dies eine der ersten Painting-Phasen des Rendering Trees.

Als letzte Metrik gibt die TTI an, wann eine Seite nicht nur fertig gerendert, sondern auch tatsächlich interaktiv ist. Die Messung beginnt zum Zeitpunkt des First Contentful Paint. Der Endzeitpunkt ist etwas komplizierter zu beschreiben. Er lässt sich jedoch darauf reduzieren, dass in einem definierten Zeitfenster keine größeren Aufgaben, wie beispielsweise das Parsen von JavaScript, mehr durchgeführt werden. Daher kann die TTI auch gleich der FCP sein.

Neben der Performance können Rendering Patterns auch bezüglich weiterer Schwerpunkte beurteilt werden. Dazu gehören die User Experience und die Developer Experience. Außerdem kann das Pattern daraufhin untersucht werden, wie robust, resilient und sicher es sich unter verschiedenen Umwelteinflüssen verhält (z. B. keine Internetverbindung oder unter starker Last). Oder wie groß der Ressourcenaufwand ist, um die Webseite zu rendern. Ein nicht zu vernachlässigender Punkt ist schließlich die Sichtbarkeit und Auffindbarkeit der Webseite im Netz (SEO, Search Engine Optimization).

Diese Schwerpunkte wurden in **Abbildung 5** in hexagonaler Form angeordnet. Pro Schwerpunkt wurden hierfür weitere Aspekte (jedoch bei weitem nicht alle) in die Grafik eingearbeitet, die für die Bewertung herangezogen werden können. Es sollte berücksichtigt werden, dass eine abweichende Einteilung auch möglich wäre, da die Schwerpunkte nicht unabhängig voneinander sind. So ist es offensichtlich, dass eine gute User Experience und der SEO-Score von der Performance beeinflusst werden.

Im weiteren Verlauf versucht der Autor, für jedes der vorgestellten Patterns eine objektive Bewertung der Schwerpunkte vorzunehmen. Das Resultat stellt eine Fläche innerhalb des Hexagons dar, welches die Stärken (größere Ausprägung) und Schwächen (kleinere Ausprägung) des Patterns im Vergleich zu den übrigen repräsentiert. Begonnen wird mit den klassischen Patterns wie statisches, serverseitiges sowie clientseitiges Rendering.

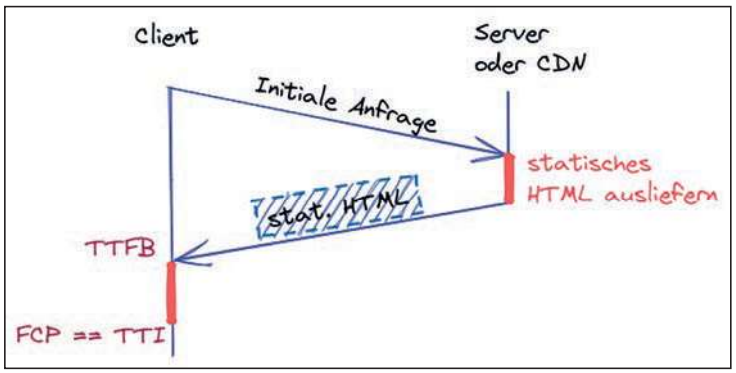

Abb. 6: Client-Server-Kommunikation beim statischen Rendering

Statisches Rendering

Wie bereits erwähnt, war das Web ursprünglich statisch. Das bedeutet, dass jede Anfrage mit demselben HTML beantwortet wird. Früher war dazu viel manueller Aufwand notwendig.

Mittlerweile existieren jedoch Static Site Generators (SSG), die Entwickler:innen einiges an Arbeit abnehmen. So ist es möglich, komponentenbasierte Frameworks und Templatesprachen zu verwenden. Der Generator reichert die Templates während des Build-Prozesses (Ahead of Time) mit statischen Inhalten (z. B. aus Markdown-Dateien) und Daten an und generiert daraus eine Webseite. Das Routing zur Seite wird in der Regel anhand der Ordnerstruktur aufgelöst (File-based Routing).

Da die Seite keine zur Laufzeit generierten Daten enthält, kann sie oft automatisiert auf ein weltweites CDN deployt werden. Das bringt auch dessen Vorteile wie flexible Skalierbarkeit, Caching und niedrige Hostingkosten mit sich. Darüber hinaus wird die Robustheit und Resilienz gesteigert, da bei einem Serverausfall oder Konnektivitätsproblemen Anfragen aus dem Cache oder von einem anderen Server des CDN bedient werden können. Außerdem kann durch ein CDN die Sicherheit, zum Beispiel gegen DDoS-Attacken, erhöht werden. Generell bietet statisches Rendering weniger Angriffsvektoren, da die Infrastruktur weniger komplex ist.

Die Anfragen werden von dem Server beantwortet, der dem Endnutzer geografisch am Nächsten liegt. Wie **Abbildung 6** zeigt, wird auf dem Server nur wenig Arbeit verrichtet, da lediglich das vorgerenderte HTML-Dokument ausgeliefert werden muss. Somit liefert statisches Rendering eine schnelle Antwortzeit (geringe Time to First Byte). Außerdem ermöglicht es einen schnellen First Contentful Paint und eine direkt interaktive Anwendung. Sofern keine synchronen Ressourcen (z. B. JavaScript) im HTML-Dokument hinterlegt sind, die den Rendering-Prozess blockieren würden, kann das Dokument direkt vom Browser geparst und gerendert werden. Dadurch können auch ressourcenschwache Clients eine statisch gerenderte Webseite schnell anzeigen.

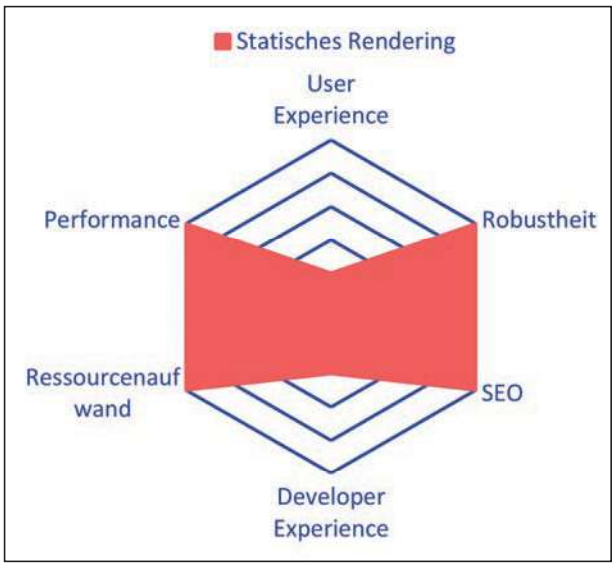

Abb. 7: Bewertung des statischen Renderings

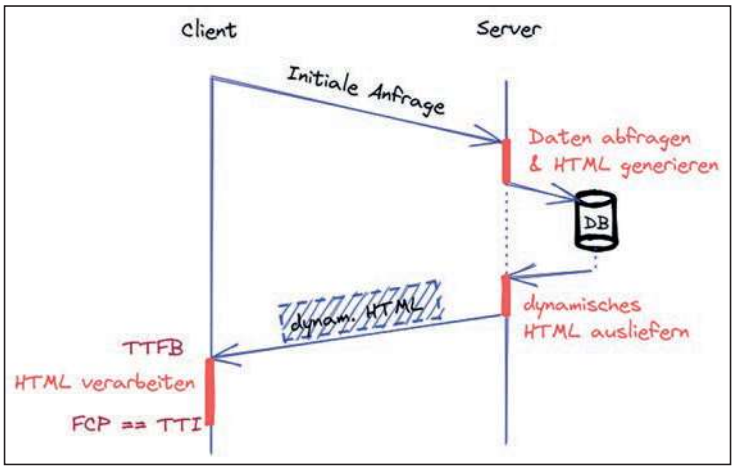

Abb. 8: Client-Server-Kommunikation beim serverseitigen Rendering

Ein weiterer großer Vorteil ist, dass Inhalte in HTML leicht von Suchmaschinen-Crawlern indiziert werden können, was eine bessere SEO ermöglicht.

Statisches Rendering ist auch vorteilhaft für die User Experience, da Inhalte ohne aktives JavaScript angezeigt werden können. Jedoch kann ein schnelles, natives Nutzererlebnis mit nahtlosen Seitenaufrufen und animierten Übergängen nicht umgesetzt werden, da aufgrund des serverseitigen Routings ein neues HTML-Dokument angefragt werden muss. Mittlerweile existiert jedoch das View Transition Proposal [5], das Besserung verspricht.

Vorteile	Nachteile
Performant	keine Dynamik
niedrige TTFB	potenziell langer Build-Prozess
TTI = FCP	kompletter Rebuild bei Änderung
SEO-freundlich	keine native User Experience
CDN-fähig (Skalierbarkeit, Cache)	
mit deaktiviertem JS nutzbar	
wenige Angriffsvektoren	

Tabelle 1: Vor- und Nachteile des statischen Renderings

Ein weiterer Nachteil kann sein, dass dynamische Inhalte oder URLs mit diesem Pattern nicht einfach zu realisieren sind. Denn die Möglichkeiten, eine Benutzerauthentifizierung clientseitig umzusetzen, sind begrenzt und haben Nachteile gegenüber der serverseitigen Variante. So wird immer JavaScript benötigt, was einige Vorteile bei Resilienz und Performance wieder zunichtemacht.

Außerdem müssten alle (persönlichen) Inhalte und Daten zuzüglich der möglichen Routen bereits zur Build-Zeit bekannt sein. Daneben spielt auch die Internationalisierung der Seite eine Rolle. Sie würde zu einer Vielzahl von vorgerenderten Dateien führen. Da die Build-Zeit linear mit der Anzahl der Dateien skaliert, führt das bei einer immer größer werdenden Anwendung zu immer längeren Wartezeiten. Hinzu kommt, dass selbst kleine Änderungen, wie z. B. die Korrektur eines Tippfehlers in einer einzelnen Datei, einen kompletten Build erfordern.

Aus diesen Gründen eignet sich statisches Rendering vor allem für statische Webseiten wie Blogs, Landing Pages oder Dokumentationsseiten, da sich hier der Inhalt in der Regel nur wenig ändert. Aber auch anwendungsartigere Webseiten können damit realisiert werden. Dafür kann die auf der Serverseite fehlende Dynamik clientseitig mit Hilfe von JavaScript nachgerüstet werden. Somit können Daten mit Hilfe von API-Aufrufen nachgeladen werden. Dieser Ansatz wird unter anderem im Jamstack-Architekturpattern [6] verwendet. Bekannte Static Site Generatoren sind z. B. Gatsby, Hugo oder Eleventy. In Tabelle 1 sowie **Abbildung 7** sehen Sie nochmal die Vor- und Nachteile des statischen Renderings im Überblick.

Serverseitiges Rendering

Im Gegensatz zum statischen Rendering, bei dem das HTML auf dem Server ahead of time vorgerendert wird und zur Laufzeit nur noch ausgeliefert werden muss, gibt es beim serverseitigen Rendering (Server-side Rendering, SSR) keine vorgelagerte Build-Phase. Stattdessen wird das HTML für jede Anfrage just in time berechnet.

Der Ablauf zwischen Client und Server (**Abb. 8**) ist fast genauso wie beim statischen Rendering. Da jedoch die Daten nicht statisch vorhanden sind, sondern von anderen Systemen, beispielsweise einer Datenbank, angefragt oder berechnet werden müssen, dauert die Generierung des HTML etwas länger. Die Antwortzeit hängt von der Antwortzeit der aufgerufenen Schnittstellen und der Rechenzeit auf dem Server ab. Dadurch erhöht sich die TTFB. Die anschließende Interpretation auf dem Client verhält sich genauso wie beim statischen Rendering, da es auf Clientseite keinen Unterschied zwischen statisch oder serverseitig generiertem HTML gibt.

Die Vorteile sind also ähnlich. SSR bietet auch eine gute SEO und eine nutzbare Anwendung, selbst bei deaktiviertem JavaScript. Außerdem ist die Webseite nach dem Rendern sofort interaktiv. Das eigentliche Argument für serverseitiges Rendering ist jedoch die Möglichkeit, Anfragen dynamisch zu beantworten und individuelle URLs (z. B. mit einer Nutzer-ID) zu ver-

arbeiten. Außerdem ist eine sichere Authentifizierung möglich, wodurch der Inhalt nicht mehr frei zugänglich ist. Dadurch kann Nutzer:innen eine personalisierte User Experience geboten werden.

Ein Nachteil ist jedoch der Wegfall des CDN. Häufig sind die Serverinstanzen nicht so weitflächig verteilt wie die des CDN. So existieren meistens nur wenige Server pro Region. Dadurch wird die Skalierbarkeit erschwert und der Server stellt einen potenziellen Flaschenhals dar. Die Render-Leistung hängt nun von der Auslastung des Servers und damit von der Anzahl der Nutzer ab. Im schlimmsten Fall kann ein Server- oder Netzwerkausfall in dieser Region zu einer potenziellen Downtime der Website führen. Zusätzlich müssen die Daten einen längeren Weg zurücklegen als beim statischen Rendering. Dennoch ist SSR weiterhin performant, da der Server in der Regel potenziell näher (idealerweise im selben Netzwerk) bei den Drittsystemen liegt als der Client und über mehr Leistung verfügt.

Durch den Wegfall des CDN-Caches können Anfragen von anderen Instanzen nicht mehr bedient werden. Ein serverseitiger Caching-Mechanismus ist zwar weiterhin möglich, aber komplizierter umzusetzen als bei einem CDN. Auch ressourcenmäßig ist SSR aufwendiger. Pro Anfrage wird mehr Berechnungszeit und Energie benötigt, wodurch auch die Kosten steigen. Ebenfalls steigt gegenüber dem statischen Ansatz die Anzahl an Angriffsvektoren, da jede Anfrage vom Server verarbeitet werden muss und potenziell mehr Systeme beteiligt sind. Ein Beispiel wären Injection-Attacken.

Neben diesen Nachteilen, die spezifisch für serverseitiges Rendering sind, gibt es auch Nachteile, die bereits beim statischen Rendering erwähnt wurden. Auch hier handelt es sich bei Webanwendungen um MPAs. Daher muss bei einem Seitenaufruf die Seite immer komplett neu geladen werden und Transitionen sind beim Seitenwechsel nicht möglich.

Serverseitiges Rendering sollte für Webseiten verwendet werden, die vor allem individuell, schnell interaktiv und gut auffindbar sein sollen. Das ist beispielsweise bei Webshops oder Webseiten mit personalisierten Inhalten der Fall. Es kann auch für nicht allzu dynamische Webanwendungen genutzt werden.

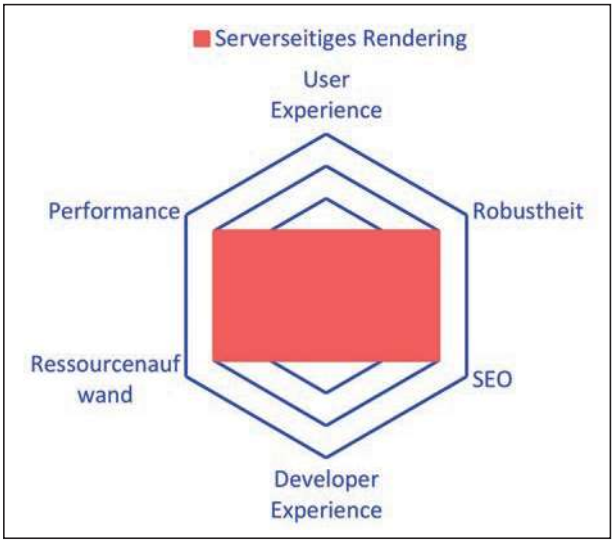

Abb. 9: Bewertung des serverseitigen Renderings

Frameworks, die SSR anbieten, gibt es viele. Die hier besprochene klassische Variante von SSR findet sich vor allem noch bei Full-Stack-Frameworks, wie beispielsweise PHP oder Spring Boot. Bei Lösungsansätzen aus dem JavaScript-Universum (z. B. React) wird SSR häufig mit einem Hydrationsschritt kombiniert. Um sich damit auseinandersetzen zu können, sollte zuvor aber clientseitiges Rendering betrachtet werden. Die Vor- und Nachteile von SSR sehen Sie nochmal in Tabelle 2 und **Abbildung 9** zusammengefasst.

Clientseitiges Rendering

Das clientseitige Rendern (Client-side Rendering, CSR) wurde mit dem Aufkommen von Single Page Applications (SPA) populär. Oft werden CSR und SPA als Synonyme verwendet, was jedoch falsch ist.

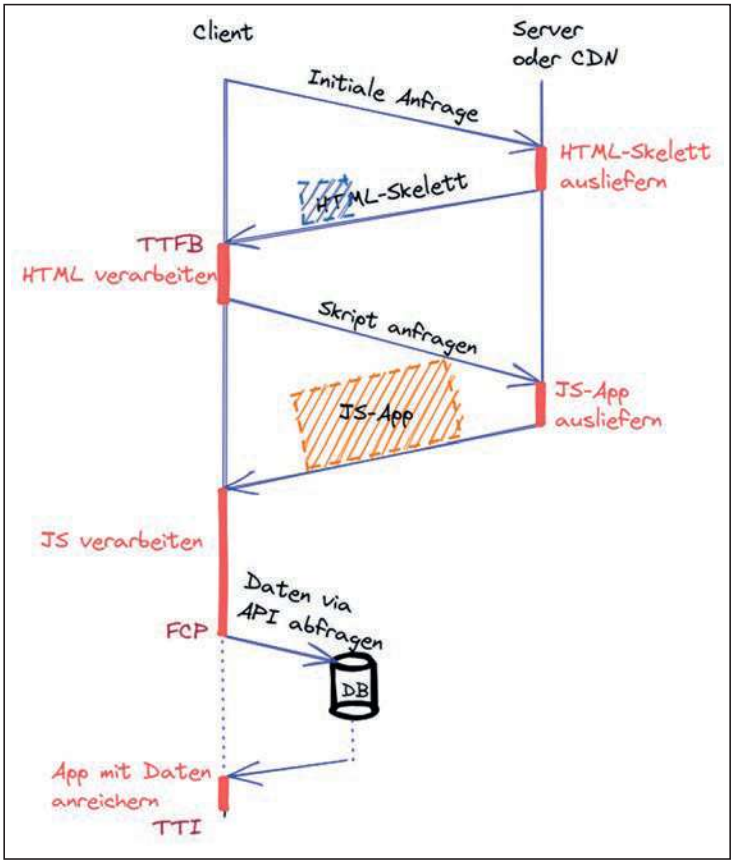

Abb. 10: Client-Server-Kommunikation beim clientseitigen Rendering

Eine SPA zeichnet sich dadurch aus, dass das Routing nur noch clientseitig erfolgt und eine neue Seite ohne Austausch des HTML-Dokuments angezeigt werden kann. Oftmals geschieht das in Kombination mit CSR, da sich beide bestens ergänzen. Jedoch könnte clientseitiges Rendering ebenfalls für Multi Page Applications eingesetzt werden, indem das Routing auf dem Server stattfindet und für jeden Seitenwechsel eine neue Anwendung auf dem Client gerendert wird. Das ist jedoch wenig sinnvoll, weil auf diese Weise die Stärke einer SPA (kein neues HTML-Dokument pro Seite) verloren geht.

Vorteile	Nachteile
Performant	potenziell hohe TTFB
TTI = FCP	Rendering abhängig von Konnektivität
SEO-freundlich	Performance abhängig von Nutzerzahl und Serverstandort
dynamische, personalisierte Inhalte	keine native User Experience
mit deaktiviertem JS nutzbar	mangelnde CDN-Fähigkeit

Tabelle 2: Vor- und Nachteile des serverseitigen Renderings

Vorteile	Nachteile
gute User Experience	geringe SEO-Unterstützung
niedrige TTFB	TTI >> FCP
Trennung zwischen Client- und Servercode	Verteiltes mentales Modell
CDN-fähig	funktionslos bei deaktiviertem JS
	viele Angriffsvektoren

Tabelle 3: Vor- und Nachteile des clientseitigen Renderings

Der Ablauf von CSR (**Abb. 10**) ist dabei wie folgt: Im Gegensatz zum statischen oder serverseitigen Rendering findet die Geschäftslogik, das Laden der Inhalte, das Templating sowie das Routing nicht auf dem Server, sondern auf dem Client statt. Daher antwortet der Server auf die initiale Anfrage lediglich mit einem HTML-Gerüst ohne Inhalt. Da das, ähnlich wie beim statischen Rendering, fast keine Arbeit auf dem Server erfordert und die Datei zudem klein ist, erhält der Client sehr schnell eine initiale Antwort (niedrige TTFB). Allerdings kann der Client nach dem Rendern noch keine sinnvollen Inhalte darstellen, sondern muss zunächst die gesamte Applikation als JavaScript-Asset herunterladen und ausführen. Ab diesem Zeitpunkt kann der erste sinnvolle Inhalt (FCP) angezeigt werden. Die Anwendung ist jedoch unter Umständen noch nicht vollständig interaktiv, da Daten und andere dynamische Inhalte erst asynchron nachgeladen werden müssen. Je nach Umfang und verfügbarer Bandbreite kann das einige Sekunden dauern.

Ist dieser Schritt abgeschlossen, übernimmt der JavaScript-Code die Arbeit und der Nutzer erhält eine nahezu native User Experience. Die Anwendung reagiert schnell und verhält sich responsiv, da für die Navigation innerhalb der Anwendung kein kompletter Seitenaufruf mehr erfolgen muss. Bei einem Seitenwechsel müssen weder HTML-Dokument noch ressourcenintensive Analyseskripte neu geladen oder ausgeführt werden. Die serverseitige Navigation wird durch einen clientseitigen Router ersetzt, der sich im JavaScript-App-Bundle befindet. Dadurch sind auch Seitentransitionen und Animationen beim Seitenwechsel möglich, wie man es von nativen Apps gewohnt ist. Außerdem ermöglicht es auch eine begrenzte Nutzung der Anwendung, falls man offline ist.

Da fast die gesamte Logik in den Client wandert, kann eine clientseitig gerenderte Single Page Application auch in ein CDN deployt werden. Das vereinfacht das Hosting, spart Ressourcen und liefert weitere Vorteile, wie es bereits im Abschnitt über das statische Rendering ausgeführt wurde.

Ein weiterer Vorteil ist, dass durch clientseitiges Rendering eine klare Trennung zwischen Client- und Servercode erreicht werden kann. In der Praxis bedeutet das, dass verschiedene Teams unabhängig von der Technologie am Backend oder Frontend arbeiten können. Durch dieses verteilte mentale Modell entsteht aber auch ein erhöhter Organisations- und Kommunikationsaufwand, da sich die Teams untereinander abstimmen müssen. Außerdem kann es zu potenzieller Codeduplizierung oder abweichender Geschäftslogik führen, wenn z. B. sowohl im Frontend als auch im Backend validiert werden soll.

Der offensichtlichste Nachteil, wenn die Webseite nur mit JavaScript gerendert wird, ist eine mangelhafte SEO-Fähigkeit. Wenn der Suchmaschinen-Crawler Java

Script nicht interpretieren kann, sieht er nur eine leere Hülle ohne nennenswerte Informationen. Und wenn doch, kann es sein, dass bei vielen oder länger andauernden clientseitigen API-Anfragen noch nicht der gesamte Inhalt geladen ist und dieser somit nicht indiziert wird. Zudem ist eine clientseitig gerenderte Anwendung bei deaktiviertem JavaScript oder schlechter Konnektivität beim initialen Aufruf nicht nutzbar.

Außerdem führt die Verlagerung von Logik, Inhalten, Templating und Routing nach JavaScript dazu, dass mit wachsender Webseite auch die Gesamtgröße des übertragenen JavaScript Bundles skaliert. Das bestätigt auch der Report vom HTTP Archive [7]. Lag der Median für den Desktop Anfang 2011 noch bei knapp 100 kB JavaScript, so ist dieser Wert mittlerweile auf über 500 kB angestiegen. Dieses Wachstum führt zu längeren Antwortzeiten. Zudem werden mehr Rechenleistung und Zeit für die Interpretation und Ausführung des Codes benötigt. Zusätzlich benötigt CSR auch mehr initiale Kommunikation zwischen Client und Server. Zum Aufbau der Seite sind mindestens zwei Anfragen notwendig (**Abb. 10**). Durch die Bundle-Größe und diese sogenannte Wasserfallproblematik steigt die TTI und schwächere Endgeräte könnten überlastet werden.

Um diese Auswirkungen so gering wie möglich zu halten, existieren verschiedene Optimierungsmöglichkeiten, damit die Bundle-Größe weiterhin gering bleibt. Dazu gehört das strikte Einhalten eines JavaScript-Budgets, z. B. durch Minification, Code Splitting und Lazy Loading, oder das Laden kritischer Ressourcen vor dem Rendern durch Preloading. Diese Optimierungen bringen jedoch ihre eigenen Probleme [8] mit sich.

Neben dem Laden von Ressourcen kann clientseitiges Rendering auch beim Laden von Daten aus der Persistenzschicht ungünstig sein. So kann sich ein Server im gleichen Netzwerk wie die Datenbank befinden. Ein Client hingegen kann sich überall auf der Welt befinden. Das erhöht die Latenz zur Datenabfrage und auch den Aufwand bzgl. des Datentransports. Schließlich ermöglicht CSR mehr Angriffsvektoren als serverseitige Varianten, da der Anwendungscode und somit die Geschäftslogik auf dem Client einsehbar und manipulierbar ist. Dadurch können sensitive Daten veröffentlicht werden.

Aus diesen Gründen eignet sich clientseitiges Rendering vor allem für interaktionslastige, dynamische und komplexe Single Page Applications, die eine native User Experience und lange Verweildauer ermöglichen. Beispiele hierfür finden sich im Social-Media-Umfeld mit Facebook von Meta oder auch bei Produktivitätsanwendungen wie Google Docs wieder. Auch Dashboards eignen sich hierfür gut. Es ist daher nicht verwunderlich, dass zwei der bekanntesten Frameworks für clientseitiges Rendering von diesen Unternehmen stammen – Angular von Google und React von Meta. Daneben existiert als dritter großer Player noch Vue. Die Vor- und Nachteile von CSR sehen Sie in Tabelle 3 und **Abbildung 11** zusammengefasst.

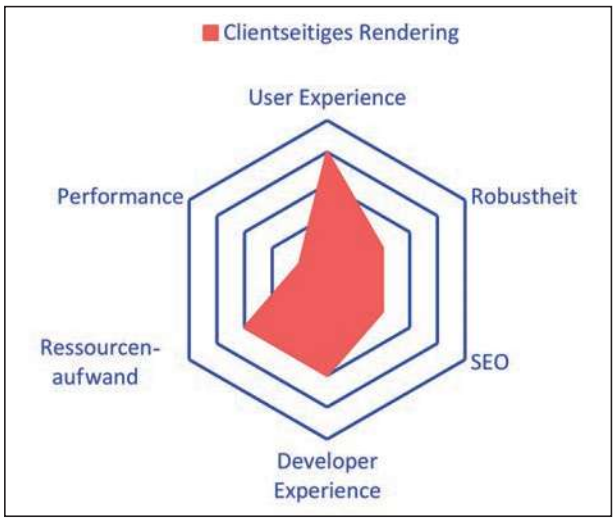

Abb. 11: Bewertung des clientseitigen Renderings

Fazit

Im ersten Teil der dreiteiligen Artikelserie konnten wir anhand der Entwicklung des Webs die Entstehung der Rendering Patterns nachvollziehen. Zudem haben wir gelernt, wie und mit welchen Schwerpunkten diese miteinander verglichen werden können. Dabei haben wir uns zuerst auf die drei grundlegenden Patterns konzentriert.

Zwischen den beiden Extremen dieses Spektrums (SSR und CSR) liegen jedoch noch fortgeschrittenere Ansätze, die auf diesen Grundlagen aufbauen oder sie sogar kombinieren. Freuen Sie sich im nächsten Teil auf Incremental Static Regeneration, Prerendering sowie Hydration-Ansätze. Falls Sie bis dahin nicht warten wollen, empfehle ich Ihnen, sich die Webseiten unter [9] und [10] bzw. den Blogbeitrag unter [11] näher anzuschauen.

Julian Schäfer arbeitet als Full-Stack-Softwareentwickler bei der synyx GmbH & Co. KG in Karlsruhe. Daneben beschäftigt er sich mit Webtechnologien und versucht, dieses Wissen auch während seines Projektalltags weiterzugeben.

@ju_schaefer

Links & Literatur

[1] SPA by default: https://www.thoughtworks.com/de-de/radar/techniques/spa-by-default

[2] The State of JS: https://2022.stateofjs.com/en-US/usage/#js_app_patterns

[3] Space-Jam-Webseite: https://www.spacejam.com/1996/

[4] Breaking Down the Web w/ Dan Jutan: https://www.youtube.com/watch?v=REXtIuAJ3dE&t=6443s

[5] Jamstack: https://jamstack.org

[6] View Transitions: https://github.com/WICG/view-transitions

[7] HTTP Archive: https://httparchive.org/reports/page-weight#bytesJs

[8] Second-guessing the modern web: https://macwright.com/2020/05/10/spa-fatigue.html

[9] Patterns.dev: https://www.patterns.dev/posts/#rendering-patterns

[10] Modern Rendering Patterns: https://www.lydiahallie.io/blog/rendering-patterns

[11] Rendering on the Web: https://web.dev/rendering-on-the-web/