Rendering Patterns in der Webentwicklung – Teil 3

Zurück zum Server

Viele Rendering Patterns wurden bereits in den letzten beiden Artikeln betrachtet. Wir konnten die Entwicklung verfolgen, wie Inhalte zuerst statisch sowie serverseitig gerendert wurden, dann immer mehr Aufgaben auf den Client verlagert wurden, bevor der Trend wieder zurück zum Server ging. Die Kunst besteht heute darin, in diesem Spektrum den optimalen Bereich, den sogenannten Sweet Spot zu finden.

von Julian Schäfer

Einen Schwerpunkt des vorherigen Artikels bildete serverseitiges Rendering mit Hydration. Wie wir sehen konnten, ermöglichte die serverseitige Ausführung der Anwendung eine deutlich gesteigerte User Experience im Vergleich zu einer clientseitig gerenderten Anwendung. Hydration ist dabei jedoch nur ein Ansatz, der zeigt, dass die aktuelle Entwicklung der Rendering Patterns sich in eine Richtung orientiert: mehr Verantwortung auf den Server zu verlagern.

Zum Abschluss dieser Serie werden Patterns vorgestellt, die den Status quo und die mögliche Zukunft des Renderings von Inhalten im Web zeigen und den erwähnten Trend zum Server untermauern. Dazu zählen Streaming, React Server Components sowie Resumability.

Artikelserie

Teil 1: Spektrum der Rendering Patterns
Teil 2: Ein breiteres Spektrum – komplexere Patterns
Teil 3: Zurück zum Server

Serverseitiges Rendering mit Streaming

Dieses Pattern ist an sich nicht neu. Browser unterstützen HTML-Streaming schon seit geraumer Zeit. In den letzten Jahren hat es jedoch an Bedeutung gewonnen, um die mangelnde Effizienz des Rendering-Prozesses auszugleichen.

Wie wir bereits bei den vorherigen serverseitigen Patterns gesehen haben, müssen immer bestimmte Schritte durchgeführt werden, bevor ein Benutzer Inhalte auf der Seite sehen und mit ihnen interagieren kann. Nachdem der Client Kontakt mit dem Server aufgenommen hat, werden alle dynamischen Daten vom Server angefordert, beispielsweise durch einen Datenbankzugriff. Sind diese Daten vorhanden, erstellt der Server daraus ein HTML-Dokument und sendet es im Gesamten an den Client. Dieser interpretiert und rendert das HTML und fordert gegebenenfalls weitere Ressourcen an. Beim serverseitigen Rendering mit dem Hydration-Schritt muss noch der JavaScript-Teil angefragt werden. Anschließend erfolgt die Hydration. Erst dann ist die Anwendung vollständig gerendert und interaktiv.

Diese Schritte müssen sequenziell abgearbeitet werden, da sie aufeinander aufbauen. Ein Teil der Webseite ist jedoch statisch und kommt ohne die angefragten dynamischen Daten aus. Ein Beispiel hierfür ist der Header einer Webseite. Wenn diese Daten bereits vor der Anfrage an die Datenbank versandfertig sind, warum werden sie dann nicht direkt an den Client gesendet? HTML-Streaming macht genau das. Anstatt zu warten, bis das gesamte HTML-Dokument serverseitig generiert wurde, werden einzelne HTML-Chunks an den Client gestreamt (**Abb. 1**). Dadurch ergibt sich eine geringe und oft konstante Time to First Byte (TTFB), unabhängig von der Gesamtgröße der Webseite. Der Browser kann diese statischen HTML-Schnipsel bereits verarbeiten, obwohl auf dem Server noch keine Daten aus der Datenbank vorliegen. Dadurch wird die Seite progressiv gerendert, was einen niedrigeren FCP-Wert (First Contentful Paint) sowie eine geringere TTI (Time to Interactive) als beim Hydrationsansatz ermöglicht. Außerdem eröffnet das die Möglichkeit, weitere externe Ressourcen anzufragen, obwohl das Backend parallel noch das dynamische HTML generiert.

Da nur Teile des gesamten HTML-Dokuments zu einem Zeitpunkt verarbeitet werden, wird auch der Speicherbedarf des Servers reduziert. Somit können insgesamt mehr Anfragen verarbeitet und Kosten gespart werden. Außerdem können durch die inkrementelle Übertragung Engpässe im Netzwerk besser vermieden werden. Tritt ein solcher Fall ein, wird der Stream so lange pausiert, bis wieder genügend Kapazität vorhanden ist. Dadurch wird die Anwendung für Eingabe- bzw. Ausgabeoperationen responsiver, da größere Anfragen kleinere Anfragen nicht für längere Zeit blockieren.

Leider ist Streaming, ähnlich wie die fortgeschrittenen Hydration-Ansätze, komplex und schwer umzusetzen. Es müssen viele Fallstricke [1] beachtet werden. Zum Beispiel ist selbst die Bestimmung des HTTP-Statuscodes des HTML-Dokuments nicht trivial. Denn auch wenn unser erster HTML-Chunk erfolgreich versendet wurde, weil er nur statische Daten enthält, kann bei der Datenbankabfrage etwas schiefgehen. Daher ist es nicht möglich, als erste Antwort einen HTTP-200-Statuscode zu senden, ohne auf die Datenbankabfrage zu warten. Das führt jedoch den Streamingansatz ad absurdum. Auch SEO kann

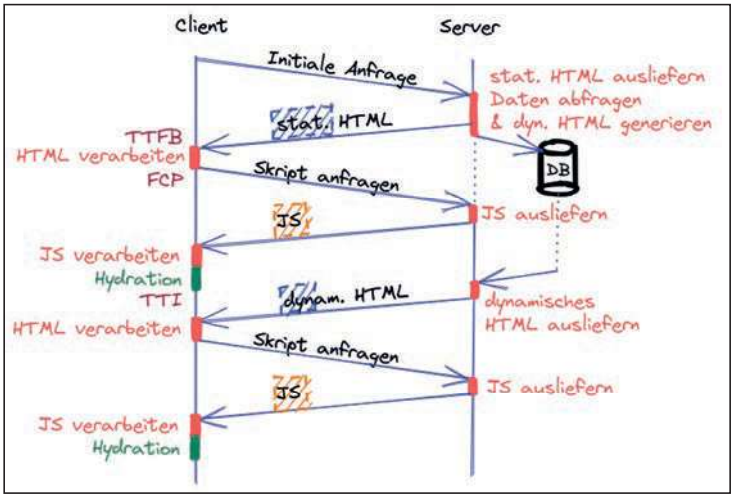

Abb. 1: Der zweite HTML-Chunk wird ohne erneute Anfrage an den Client ausgeliefert

problematisch sein, da nicht jeder Crawler mit der inkrementellen Auslieferung von HTML umgehen kann.

Streaming ist heutzutage vor allem im JavaScript-Ökosystem weit verbreitet. Neben dem Browser auf der Clientseite werden Streams auf Serverebene von Node.js gut unterstützt. Andere Ökosysteme, wie beispielsweise PHP, unterstützen zwar auch Streaming, jedoch nicht in dem Umfang [2], wie Node.js es tut.

Als eins der ersten JavaScript-Frameworks, die Streaming implementiert haben, gilt Marko.js. Mittlerweile gibt es eine Vielzahl weiterer Lösungen, die mehr oder weniger ausgereift sind. Als Beispiel sei auch hier React mit seinen Metaframeworks genannt.

Streaming spielt überall dort seine Stärken aus, wo Geschwindigkeit und eine hohe Auslastung eine Rolle spielen. Aufgrund der zusätzlichen Komplexität sollte es nur mit Bedacht eingesetzt werden. Die Vor- und Nachteile sehen Sie in Tabelle 1 und **Abbildung 2** zusammengefasst.

React Server Components

Wie der Name bereits verrät, handelt es sich um ein spezifisches Pattern für das weit verbreitete React-Framework aus dem JavaScript-Umfeld. React Server Components (RSC) wurden Ende 2020 angekündigt [3]; mit ihnen rückt das ehemals clientzentrierte Framework nun den Server stärker in den Fokus. Als Inspiration diente unter anderem die alte Facebook-Architektur,

Vorteile	Nachteile
performant und effizient	TTI > FCP (besser als bei Hydration)
niedrige und konstante TTFB	mangelnde CDN-Fähigkeit
gute User Experience	erhöhte Komplexität
dynamische, personalisierte Inhalte	geringe Laufzeitunterstützung
reduzierte Server- und Netzwerklast	potenziell problematische SEO-Unterstützung
mit deaktiviertem JS eingeschränkt nutzbar	
einheitliches mentales Modell	

Tabelle 1: Vor- und Nachteile von SSR mit Streaming

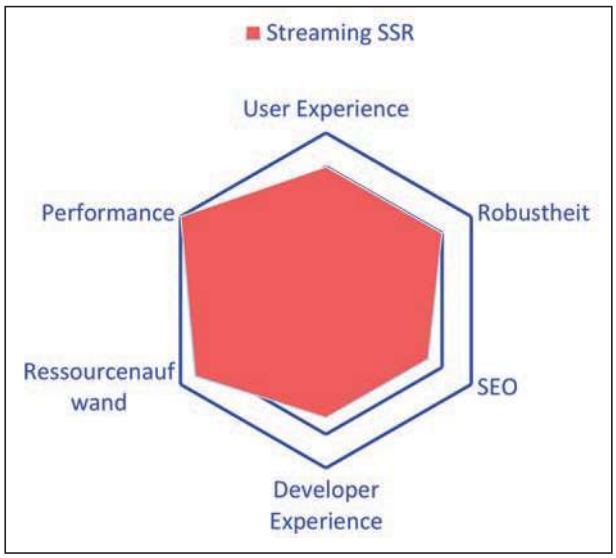

Abb. 2: Bewertung Streaming – performant und ressourcenschonend

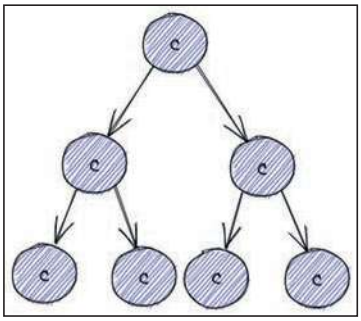

Abb. 3: Bei klassischen React-Anwendungen existieren nur Clientkomponenten

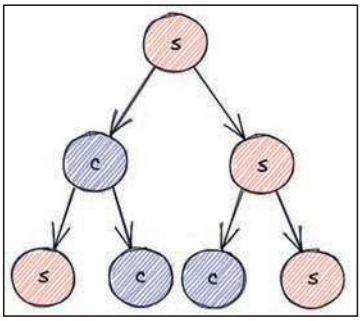

Abb. 4: Bei RSC gibt es sowohl serverseitige als auch clientseitige Komponenten

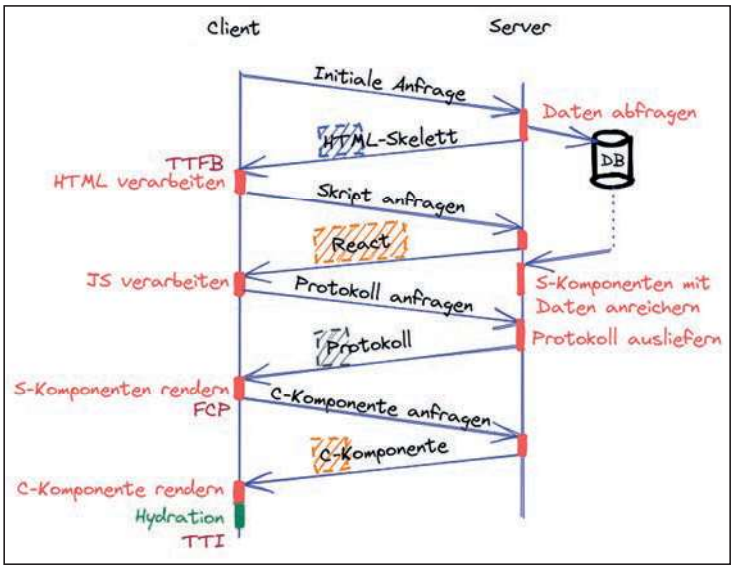

Abb. 5: Clientkomponenten werden unabhängig von der Laufzeit geladen

bei der bereits mittels SSR einige Ansätze dieses Patterns umgesetzt wurden [4].

Die Idee der Serverkomponenten ähnelt dem Ansatz der partiellen Hydration aus dem vorherigen Teil der Serie. Auch hier wird der Anwendungscode in interaktive und statische Bestandteile aufgeteilt. Hydration ist jedoch ein Ansatz, bei dem alle Komponenten auf dem Server vorgerendert und anschließend auf dem Client ausgeführt werden müssen. Daher muss ihr Code weiterhin an den Client gesendet werden. Anders ist das beim RSC-Pattern. Hier wird nur der Code versendet, der für die interaktiven Bestandteile auf dem Client benötigt wird. Komponenten ohne Interaktivität oder clientseitigen Zustand werden serverseitig ausgeführt.

Das erfordert einen Paradigmenwechsel in der Art und Weise, wie React-Anwendungen entworfen und implementiert werden. Zuvor bestand die gesamte Anwendung aus Komponenten (**Abb. 3**), die mittels JavaScript auf dem Client ausgeführt wurden.

Die Ausführung jeder Komponente auf dem Client bringt verschiedene Probleme mit sich. Beispielsweise ist das JavaScript-Bundle sehr groß, da der gesamte Code für die Komponenten einschließlich der abhängigen Pakete an den Client ausgeliefert werden muss. Das führt zu potenziell langen Ladezeiten. Dieses Problem wird z. B. durch Hydration-Ansätze zu entschärfen versucht. Allerdings können damit nicht alle Probleme gelöst werden.

Zum Beispiel erfordert das Top-down-Rendering des Komponentenbaums unnötig viel Kommunikation zwischen Server und Client. Das wird als die sogenannte Wasserfallproblematik bezeichnet. Sie entsteht, wenn eine Komponente ihre Daten vom Server anfragt und diese via Props an ihre Kindkomponenten weitergibt. Diese untergeordneten Komponenten lösen wiederum ihre eigenen Anfragen aus, was zu einem Wasserfall von Anfragen führen kann, der die Anwendung verlangsamt.

RSC versucht, diese Probleme zu lösen, indem ein Teil der Arbeit auf den Server verlagert wird. Dieser besteht überwiegend aus Datenabfragen und Vorverarbeitung der Daten. Der Code für die Interaktivität und das responsive Verhalten der Anwendung wird weiterhin auf dem Client ausgeführt. Dadurch müssen dem Client nicht mehr alle Komponenten und Abhängigkeiten bekannt sein. Stattdessen gibt es Komponenten, die nie vom Client geladen und nur auf dem Server ausgeführt werden. Der Komponentenbaum enthält somit eine Mischung aus server- und clientseitigen Komponenten (**Abb. 4**). Eine Besonderheit davon ist, dass die oberste Komponente des Baumes immer eine Serverkomponente sein muss.

Der Ablauf ist dabei ähnlich wie bei rein clientseitig gerenderten React-SPAs. Zuerst wird mit einem HTML-Skelett (niedriges TTFB) geantwortet, dann lädt der Client das JavaScript Bundle mit der React-Laufzeit (**Abb. 5**). Im Gegensatz zum rein clientseitigen Rendering wird mit dem Bundle noch kein Anwendungscode, d. h., es werden keine Clientkomponenten, ausgeliefert. Parallel zur clientseitigen Verarbeitung kann auf dem Server mit dem Aufbau und der Serialisierung des Komponentenbaums begonnen werden.

Bevor serverseitige Komponenten serialisiert werden können, müssen diese zunächst die benötigten Daten laden, z. B. durch Datenbankabfragen. Im Gegensatz zu clientseitigen Komponenten kann der Zugriff auf die Daten direkt auf der Ressource erfolgen, da keine Zwischenschicht wie ein API-Layer existiert. So kann

die Datenbank oder auch das Dateisystem des Servers direkt angesprochen werden, um die Daten zu laden. Anschließend wird die Serverkomponente in ein serialisierbares Format gebracht.

Dazu werden Informationen wie die HTML-Struktur, die geladenen Daten sowie die Eltern-Kind-Beziehungen der Komponente in ein eigenes, JSON-ähnliches Protokoll verpackt. Sobald der erste Eintrag in das Protokoll geschrieben wurde, kann die Datei bereits an den Client gestreamt werden. Durch das Streaming kann das Protokoll sukzessive um Einträge weiterer serialisierter Komponenten ergänzt werden.

Im Gegensatz zu den Serverkomponenten müssen die Clientkomponenten nicht auf dem Server ausgeführt, sondern lediglich serialisiert werden. Hierzu werden Referenzen auf das jeweilige JavaScript-Modul im Protokoll hinterlegt.

Auf dem Client werden die Einträge von der React-Laufzeit interpretiert. Da alle notwendigen Informationen zum Aufbau der serverseitigen Komponenten bereits komplett im Protokoll hinterlegt wurden, können diese verwendet werden, um den Komponentenbaum aus Serverkomponenten aufzubauen. Wie bei einer normalen React-Anwendung wird das DOM anschließend gerendert. Durch diesen progressiven Ansatz sieht der Nutzer schnell erste Inhalte (geringer FCP).

Für die Clientkomponenten werden im Komponentenbaum zunächst Platzhalter angelegt, da deren Quellcode clientseitig noch nicht vorhanden ist. Dieser muss anhand der hinterlegten Modulreferenzen nachgeladen werden. Wurde der Code geladen, kann die Clientkomponente anschließend in bekannter Weise initialisiert, gerendert und hydriert werden und der Platzhalter durch die eigentliche Komponente ersetzt werden.

Die oben beschriebene Kommunikation zwischen Server und Client hat den Vorteil, dass Clientkomponenten nach ihrer Initialisierung die benötigten Daten nicht noch einmal über das Netzwerk anfragen. Dies wird vermieden, da die Daten bereits serverseitig von den Serverkomponenten angefordert und über das JSON-ähnliche Protokoll an den Client gesendet wurden. Die benötigten Daten können dann anhand von Props von der Serverkomponente an die Clientkomponente weitergegeben werden. Das verhindert die erwähnte Wasserfallproblematik.

Im Gegensatz zu clientseitigem Rendering oder dem Hydration Pattern wird das JavaScript Bundle auf das absolut Nötigste reduziert. Es muss keinerlei Code an den Client gesendet werden, der eigentlich nur für die serverseitige Verarbeitung, wie beispielsweise Formatierung von Daten, gedacht war. Die Daten werden über das Protokoll in formatierter Form ausgeliefert. Somit ist es irrelevant, welche oder wie viele Module eine Serverkomponente nutzt, da dieser Code niemals an den Client gesendet wird. Dadurch wird die TTI massiv reduziert. Erste Tests des React-Teams ergaben eine um knapp 30 Prozent verringerte Bundle-Größe.

Durch den serverseitigen Zugriff auf die Datenressourcen kann die Anfragelatenz reduziert und eine bes-

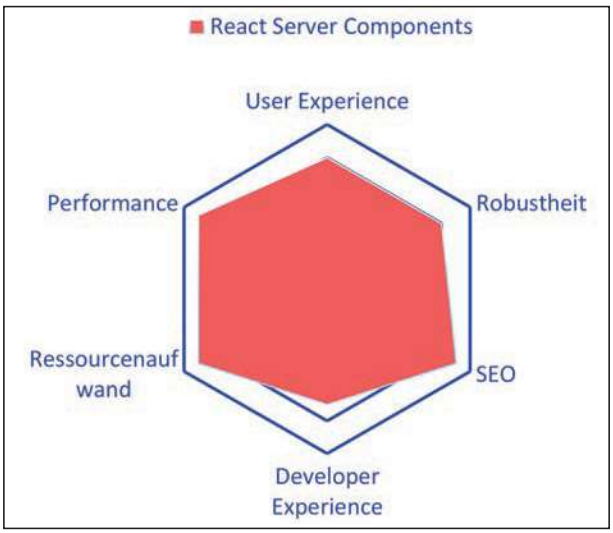

Abb. 6: Bewertung RSC – ein guter Allrounder mit Abstrichen bei der DX

sere Performance erreicht werden. Datenzugriffe können parallelisiert werden, was zu weiteren Zeiteinsparungen führt. Außerdem müssen keine sensiblen Informationen wie API-Keys an den Client gesendet werden.

Anhand des benutzerdefinierten Protokolls können bestehende UIs ohne Verlust ihres clientseitigen Zustands aktualisiert werden. Bei der Navigation zu einer anderen Seite wird, anders als beim serverseitigen Rendering, kein neues HTML-Dokument angefragt, sondern nur das Protokoll aktualisiert. Die React-Laufzeit kümmert sich anschließend darum, die neuen Daten aus dem Protokoll mit dem bereits bestehenden Zustand zu vereinen.

Da Client- und Serverkomponenten verschiedenen Zwecken dienen und vom Framework unterschiedlich behandelt werden, haben sie jeweils gewisse Beschränkungen. Beispielsweise können Serverkomponenten keine Browser-APIs verwenden. Das muss bei der Implementierung berücksichtigt werden. Diese vom bisherigen mentalen Modell abweichende Denkweise erhöht die Komplexität und erfordert eine gewisse Einarbeitungszeit.

Diese Hürde bei der Developer Experience wird in Zukunft geringer ausfallen. Der aktuelle Stand des Ökosystems ist jedoch so, dass das RSC-Pattern ohne ein darauf aufbauendes Metaframework wie Next.js nur schwer genutzt werden kann. Denn es erfordert unter anderem eine tiefe Integration mit der Routingtechnologie. Early Adopter wie Shopify mit seinem Hydrogen-Framework mussten das bereits lernen [6] und setzen mittlerweile nicht mehr darauf, da es noch zu viele offene Punkte gibt, die gelöst werden müssen.

Vorteile	Nachteile
performant und effizient	TTI > FCP (besser als bei Hydration)
niedrige TTFB	funktionslos bei deaktiviertem JS
SEO-freundlich	erhöhter Lernaufwand
gute User Experience	erhöhte Komplexität
dynamische, personalisierte Inhalte	fehleranfällig, da relativ neu
reduzierte Netzwerklast	mangelnde CDN-Fähigkeit
weniger Angriffsvektoren	abweichendes mentales Modell

Tabelle 2: Vor- und Nachteile von React Server Components

Da das React Server Component Pattern ein Opt-in-Feature ist, kann es prinzipiell in jeder bestehenden React-App eingesetzt werden. Serverkomponenten können vor allem für statische Komponenten, für Datenabfragen oder für die Ausführung von Businesslogik, die im Client nicht offengelegt werden soll, verwendet werden. Natürlich sollte darauf geachtet werden, ob die höhere Komplexität sowie der Lernaufwand die Vorteile wie eine bessere Performance rechtfertigen. Die Vor- und Nachteile sehen Sie in Tabelle 2 und **Abbildung 6** zusammengefasst.

Resumability

Viele der bisher betrachteten Patterns verwenden Hydration, um beispielsweise die lange Startzeit der clientseitigen Anwendung abzumildern und dem Nutzer die ersten sichtbaren Inhalte vorab anzuzeigen. Wie wir bereits aus den vorherigen Teilen der Serie wissen, muss dazu der Anwendungscode zweimal ausgeführt werden. Einmal auf dem Server, um das vorgerenderte HTML zu generieren und einmal auf dem Client, um die SPA zu initialisieren. Diese Dopplung ist ineffizient, jedoch notwendig, um Informationen wie den Zustand der Anwendung und des Frameworks, die Struktur des Komponentenbaumes sowie die Event Listener wiederherzustellen.

Das beschriebene Verhalten wird auch als „replayable“ bezeichnet, da der Zustand vom Server auf dem Client wiederhergestellt wird, indem die Anwendung erneut abgespielt wird. „Resumable“ hingegen bedeutet, dass der Client den zuletzt bekannten serverseitigen Zustand nahtlos wieder aufnehmen kann, ohne die Anwendung erneut ausführen zu müssen.

Metaphorisch wird das oftmals mit einer virtuellen Maschine verglichen. Deren Zustand wird serialisiert und auf einen anderen PC übertragen. Dort kann dann nahtlos weitergearbeitet werden. Ähnlich verhält es sich bei Resumability: Die Anwendung wird auf dem Server nicht nur ausgeführt, sondern auch serialisiert und anschließend an den Client übertragen. Dort kann nahtlos daran angeknüpft werden.

Ein Framework, durch das dieses Pattern bekannt wurde, ist Qwik [7]. Es wurde jüngst in Version 1.0 veröffentlicht. Mit Googles internem Framework Wiz sowie Marko.js von eBay existieren noch weitere Frameworks, die dieses Pattern implementieren bzw. eine Implementierung planen. Im Folgenden beziehen wir uns jedoch auf Qwik.

Um die Anwendung resumable zu machen, müssen bestimmte Informationen der Anwendung auf dem Server serialisiert werden. Dazu gehört, zu wissen, an welcher Stelle im DOM die Event Listener initialisiert werden müssen und welche Funktionen bei einem Event ausgeführt werden sollen. Weiterhin muss bekannt sein, wie der Komponentenbaum aufgebaut ist, welche Grenzen die einzelnen Komponenten haben und wie sie voneinander abhängig sind. Schließlich muss der Zu-

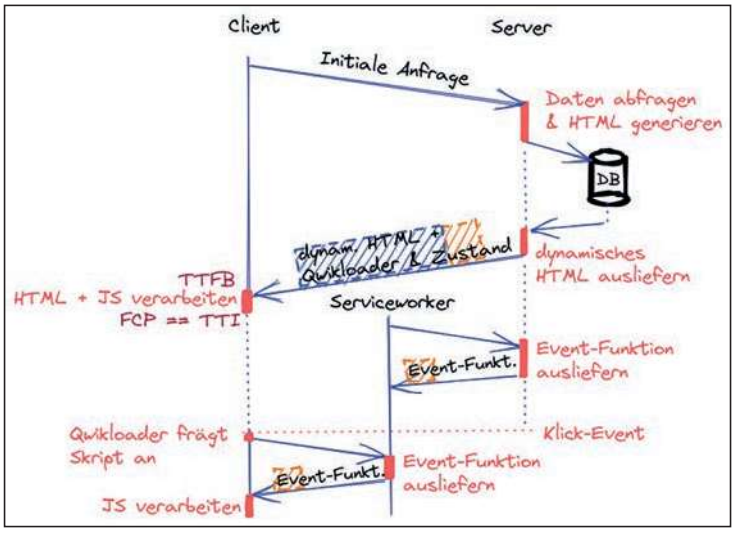

Abb. 7: Der Service Worker ist für dieses Pattern optional, verringert jedoch die Reaktionszeit der Anwendung

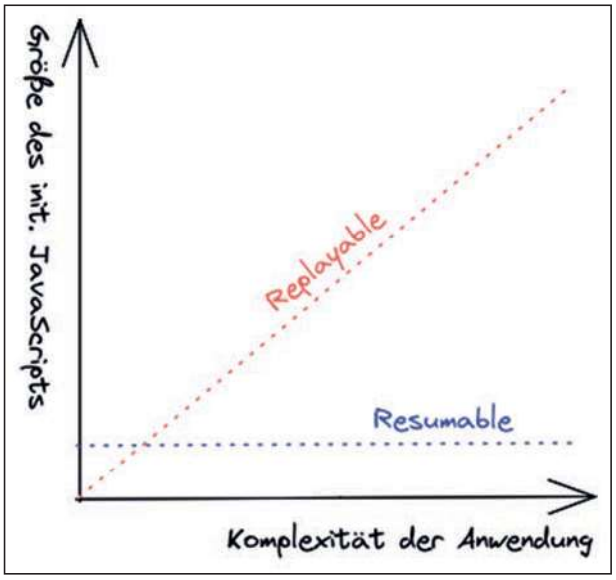

Abb. 8: Der Umfang des initial ausgelieferten JavaScripts bleibt konstant

stand der Komponenten und des Frameworks nach der serverseitigen Ausführung der Anwendung gespeichert werden.

Bei Hydration werden diese Informationen einfach mit dem JavaScript Bundle übertragen und durch Ausführung des Codes clientseitig wiederhergestellt. Bei Qwik spart man sich diese Dopplung. Das Framework teilt den Code feingranularer auf. Es ermöglicht es, eine einzelne Komponente in kleinere Bestandteile zu zerlegen und deren Informationen an verschiedenen Stellen zu speichern. Wie genau Qwik das löst, kann im Rahmen dieses Artikels nicht im Detail betrachtet werden. Im Folgenden wird nur ein grober Überblick gegeben. Nähere Informationen lassen sich unter [8] und [9] finden. So viel sei schon einmal verraten: Ganz ohne JavaScript geht es auch hier nicht.

Vorteile	Nachteile
performant und effizient	erhöhte Komplexität
niedriges TTFB	erhöhter Lernaufwand
TTI = FCP (konstante TTI)	mangelnde CDN-Fähigkeit
SEO-freundlich	Interaktivität abhängig von Konnektivität
gute User Experience	relativ neu
dynamische, personalisierte Inhalte	
reduzierte Server- und Netzwerklast	
mit deaktiviertem JS eingeschränkt nutzbar	

Tabelle 3: Vor- und Nachteile von Resumability

Wie bei jedem anderen serverseitigen Rendering Pattern werden die dynamisch generierten Inhalte im HTML-Dokument ausgeliefert. Zusätzlich wird das Dokument mit weiteren Informationen angereichert, die nicht für die Darstellung der Inhalte notwendig sind, sondern dafür, den Zustand des Servers, den Komponentenbaum oder die clientseitige Interaktivität wiederherzustellen.

Bei interaktiven DOM-Elementen, wie z. B. einem Button, werden über benutzerdefinierte HTML-Attribute die Art der Interaktion (z. B. Klick) sowie die Referenz auf ein JavaScript-Modul mit der Event-Funktion hinterlegt. Diese Attribute werden vom sogenannten Qwikloader interpretiert.

Dieses Stück Inline-JavaScript wird ebenfalls mit dem HTML-Dokument ausgeliefert. Es handelt sich um einen globalen Event Handler, der in der Lage ist, die in HTML serialisierten Informationen über die Event Listener zu interpretieren und den Code für die Event-Funktion asynchron nachzuladen (**Abb. 7**) und auszuführen. Die Vorgehensweise ist ähnlich wie beim vorhergehenden RSC-Pattern und dem Nachladen der Clientkomponenten.

Die Informationen über den Zustand der Anwendung bzw. des Frameworks zum Zeitpunkt der serverseitigen Serialisierung sowie über den Aufbau des Komponentenbaumes werden ebenfalls als Inline-JavaScript an das HTML-Dokument angehängt. Das Framework ist in der Lage, diese verteilten Informationen zusammenzuführen und sie für die clientseitige Ausführung und Interaktivität der Anwendung zu nutzen.

Der wesentliche Vorteil von Resumability ist, dass Informationen nicht mehr doppelt an den Client übertragen werden müssen. Wurden bei der Hydration die Informationen sowohl im HTML als auch im JavaScript übertragen, so sind sie nun entweder in HTML oder in JavaScript zu finden. Dadurch verringert sich die Größe des JavaScript Bundles und damit die Time to Interactive der Anwendung.

Durch Resumability ist es sogar möglich, eine nahezu konstante TTI zu erreichen, da die Größe des initialen JavaScript nicht mehr mit der Komplexität der Anwendung skaliert (**Abb. 8**). Das ausgelieferte JavaScript wächst nur noch mit den Benutzerinteraktionen oder bei bestimmten Ereignissen, da erst dann zusätzliches JavaScript durch den Qwikloader nachgeladen wird. Zum Beispiel wird die Funktion für das Inkrementie-

ren einer Zählerkomponente erst beim Klick-Event auf den Button nachgeladen. Folglich verbessert das auch den Ressourcenaufwand, da der Daten- und Energieverbrauch sinkt und die Anwendung somit auch auf schwächeren Endgeräten performant ausgeführt werden kann. Das wiederum steigert die User Experience.

Aufmerksame Leser haben vielleicht bemerkt, dass dieser Ansatz einen großen Nachteil haben kann. Was passiert, wenn das Laden des Codes für das Klick-Event sehr lange dauert? Eine reibungslose User Experience wäre dann nicht mehr gegeben, da die Interaktivität bei einem Klick nur verzögert oder gar nicht stattfinden würde. Auch für dieses Problem hat Qwik eine Lösung. Das Framework spekuliert, welche Interaktionen ausgeführt werden können und lädt Teile des Codes bereits im Hintergrund mit Hilfe eines Service Workers vor.

Ein weiterer Nachteil dieses Patterns ist die neue Denkweise. Dadurch, dass viele Dinge erst zu einem späteren Zeitpunkt nachgeladen werden, muss der Entwickler umdenken. Daher hat Resumability ähnliche Nachteile wie React Server Components, da es aufgrund des anderen mentalen Modells und den entsprechenden Konventionen komplexer ist und Lernaufwand erfordert. So kann es in bestimmten Situationen weniger flexibel sein und aufgrund der Serialisierung des serverseitigen Zustands das Testen oder Debugging erschweren.

Nichtsdestotrotz ist Resumability eines der wenigen Patterns, das eine nachhaltige Lösung für das stetige Wachstum des JavaScript Bundles verspricht. Prinzipiell kann Resumability dort eingesetzt werden, wo heute bereits Hydration-Patterns verwendet werden. Es gilt jedoch zu berücksichtigen, dass dieses Pattern noch relativ jung und nicht so weit verbreitet ist wie Hydration. Die Vor- und Nachteile sehen Sie in Tabelle 3 und **Abbildung 9** zusammengefasst.

Fazit

Mehrere der vorangegangenen Konzepte, wie das progressive Laden von JavaScript und die Aufteilung der Anwendung in statische und dynamische Teile, werden in den betrachteten Patterns wiederverwendet. Darüber hinaus wird versucht, den Workflow zu optimieren, indem Aufgabenpakete gleichzeitig auf Client und Server ausgeführt und so schnell wie möglich ausgeliefert werden. All das erhöht die Performance und die Effizienz, beides wichtige Kriterien in der heutigen Zeit. Der in der Einleitung erwähnte Trend zum Server ist daher nur logisch.

Wie bei so vielen Dingen gibt es auch in der Welt der Rendering Patterns nicht nur Schwarz oder Weiß, nein, wie das Lichtspektrum ist auch das Spektrum der Rendering Patterns vielfältig. Dabei decken die in den letzten drei Artikeln betrachteten Patterns bei weitem nicht alles ab. Es gibt noch weitere, wie z. B. Distributed Persistent Rendering, das ähnlich wie ISR funktioniert, aber den Schritt der Revalidierung eliminiert, oder Edge Rendering, das das Rendern von Inhalten direkt auf den verteilten CDN-Knoten statt auf dem Server ermöglicht.

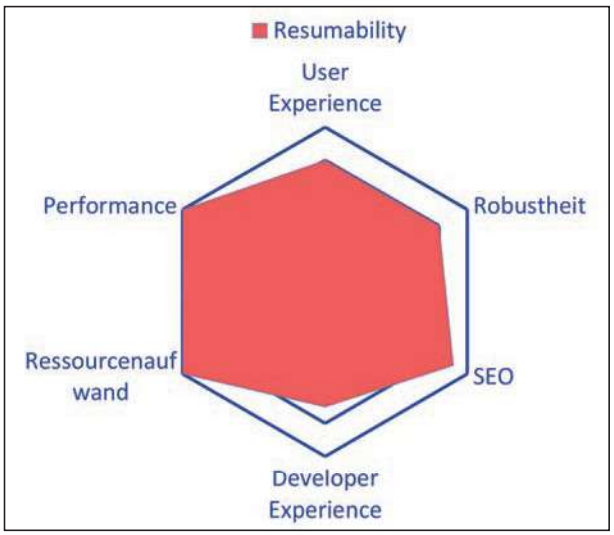

Abb. 9: Bewertung Resumability – ähnlich performant wie statische Lösungen

Dennoch hoffe ich, dass ich Ihnen einen groben Überblick über die Materie geben konnte und das Thema ohne gröbere Fehler aufgearbeitet habe. Denn wie Sie bereits ahnen können, habe auch ich noch nicht mit allen Patterns tiefergehende praktische Erfahrungen sammeln können. Dazu ist das Thema einfach zu umfangreich.

Ebenso kann es aufgrund der Vielfalt von Anforderungen an eine Anwendung kein Pattern geben, das für jeden Fall optimal ist. Anhand der aufgeführten Vor- und Nachteile sowie der (subjektiven) Bewertung jedes Patterns sollten Sie aber dennoch bereit sein, eine fundierte Entscheidung treffen zu können, wenn Sie im nächsten Projekt vor der Frage stehen: „Welches Rendering Pattern passt am besten zu meinen Anforderungen?“

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] „Streaming: is it worth it?“: https://www.builder.io/blog/streaming-is-it-worth-it

[2] „The weirdly obscure art of Streamed HTML“: https://dev.to/tigt/the-weirdly-obscure-art-of-streamed-html-4gc2

[3] „Introducing Zero-Bundle-Size React Server Components“: https://legacy.reactjs.org/blog/2020/12/21/data-fetching-with-react-server-components.html

[4] „RFC: React Server Components“: https://github.com/reactjs/rfcs/blob/main/text/0188-server-components.md

[5] „How React server components work: an in-depth guide“: https://www.plasmic.app/blog/how-react-server-components-work

[6] „What's next for Hydrogen“: https://hydrogen.shopify.dev/roadmap

[7] Qwik: https://qwik.builder.io

[8] Resumable vs. Hydration: https://qwik.builder.io/docs/concepts/resumable/

[9] „Understanding Resumability from the Ground Up“: https://www.builder.io/blog/resumability-from-ground-up