Ein breiteres Spektrum – komplexe Rendering Patterns

Nachdem im ersten Teil die Grundlagen für ein besseres Verständnis davon gelegt wurden, wie Inhalte im Web geladen und gerendert werden, wenden wir uns nun komplexeren Ansätzen zu.

Im ersten Teil dieser Artikelserie haben wir die Geschichte des Webs und die daraus resultierende Entstehung verschiedener Rendering Patterns betrachtet. Dabei wurde auf die grundlegenden Patterns wie SSG (Static Site Generation), SSR (Serverseitiges Rendering) und CSR (Clientseitiges Rendering) eingegangen. Ihre Vor- und Nachteile wurden unter anderem mit Hilfe der drei Web-Vitals-Metriken TTFB (Time to First Byte), FCP (First Contentful Paint) und TTI (Time to Interactive) unter sechs Schwerpunkten bewertet.

In diesem Teil werden darauf aufbauende Patterns behandelt, die die bisherigen Ansätze verbessern, neue Ideen hinzufügen oder Patterns kombinieren. Zuerst betrachten wir ein Pattern, das das statische Rendering verbessert. Anschließend werden wir uns mit Möglichkeiten beschäftigen, die Nachteile von clientseitig gerenderten Single-Page Applications (SPAs) zu reduzieren.

Incremental Static Regeneration

Incremental Static Regeneration (ISR), oder auch unter der Abkürzung iSSG (Incremental Static Site Generation) bekannt, ist eine erweiterte Variante des statischen Renderings (SSG), das wir bereits aus dem ersten Teil der Serie kennen. Es stellt eine Art Hybrid zwischen statischem und serverseitigem Rendering dar. ISR eliminiert einen der größten Nachteile von SSG: die lineare Skalierung der Build-Zeit mit der Anzahl der Seiten, wodurch in großen Projekten selbst kleinste

Anpassungen einer Seite einen langwierigen Build auslösen.

Stattdessen werden die einzelnen Seiten nun inkrementell erzeugt. ISR ermöglicht es, neue Seiten nach dem Build einzubinden oder bestehende Seiten zu aktualisieren, indem die statische Generierung pro Seite zur Laufzeit angestoßen wird. Wird eine Seite angefordert, die zum Zeitpunkt des Build noch nicht generiert wurde, wird bei der klassischen statischen Variante in der Regel eine Fehlerseite mit HTTP404 (Not Found) zurückgegeben. Bei ISR hingegen wird die Generierung der Webseite bei der ersten Anfrage an den Server angestoßen (**Abb. 1**). Während des Generierungsprozesses sieht der Nutzer eine Ladeanimation oder einen Platzhalter. Da die Inhalte statisch sind, ist der Generierungsprozess sehr schnell. Sobald die Generierung abgeschlossen ist, wird die Seite ausgeliefert (geringes TTFB) und gecacht. Da alle Inhalte des HTML-Dokuments bereits gerendert sind, ergibt sich ein schneller FCP und eine direkt interaktive Seite (TTI).

https://phpconference.com/session-qualification/ipc-webentwicklung/?layout=contentareafeed&widgetversion=0&utmtrackerversion=1&seriesId=XWrH7HpMZMrnZNHid
Bei der Generierung wird ein Zeitstempel hinterlegt, der angibt, wie lange die Seite aktuell ist. Jede weitere Anfrage an diese zuvor unbekannte Seite wird nun aus dem serverseitigen Cache mit der soeben generierten Seite beantwortet. Um die Aktualität der Inhalte zu gewährleisten, wird nach einer definierten Zeit (Zeitstempel plus konfigurierbarer Zeitraum) ab der nächsten Anfrage die Aktualität der Seite validiert. Dabei sollte der Zeitraum sinnvoll gewählt werden. Als Faustregel gilt: Je öfter sich der Inhalt der Seite ändert, desto kürzer sollte das Intervall sein. Ist die Seite im Cache nicht mehr aktuell ist, weil sich der Inhalt geändert hat, wird im Hintergrund eine Generierung der Seite angestoßen. In der Zwischenzeit werden Anfragen weiterhin aus dem Cache mit den veralteten Daten bedient. Dieser Ansatz wird als Stale-while-revalidate [1] bezeichnet. War die Generierung erfolgreich, wird der Cacheeintrag für

diese Seite invalidiert und durch die neue Seite ersetzt. Falls ein Fehler aufgetreten ist, können Anfragen immerhin noch mit dem alten Eintrag beantwortet werden.

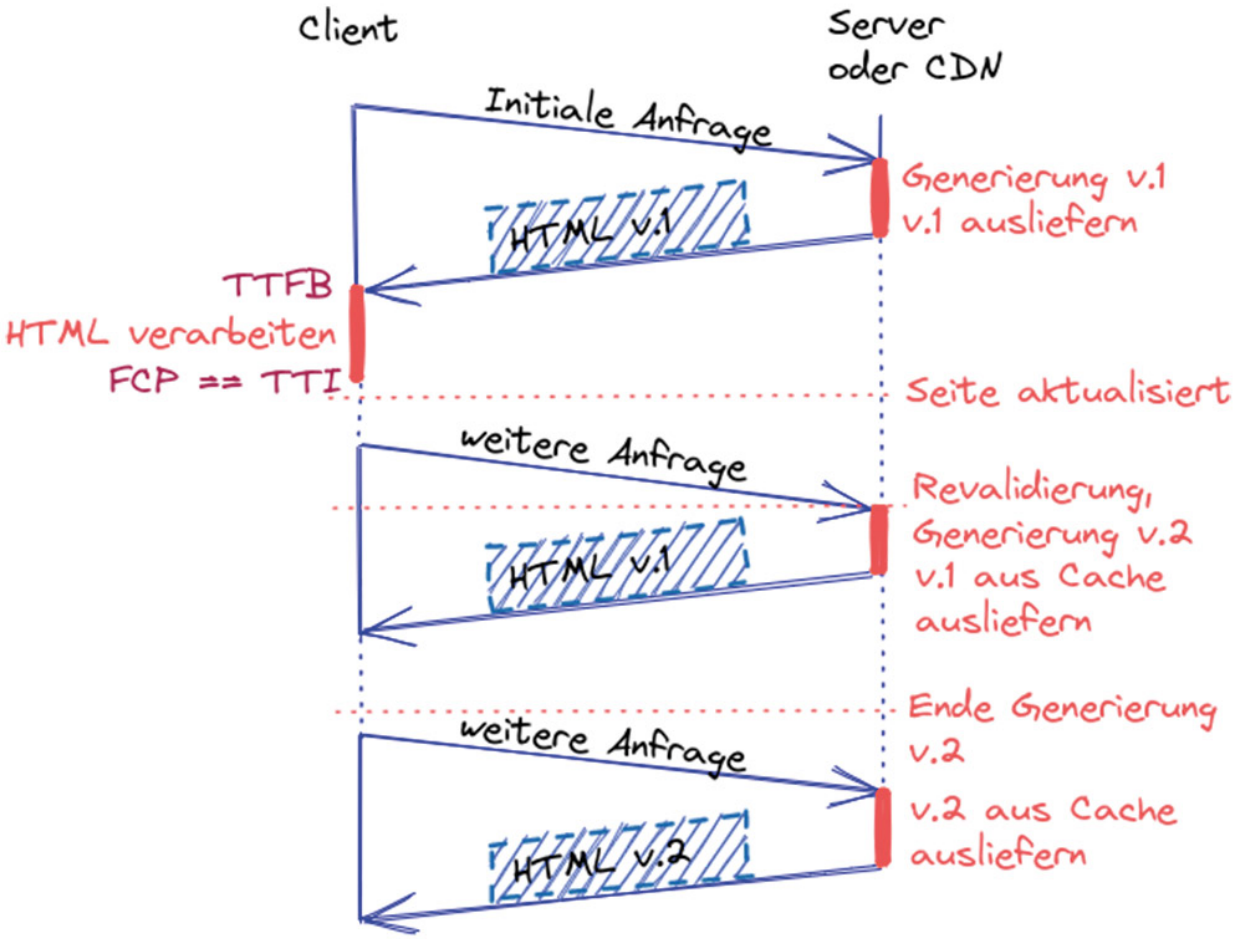

Abb. 1: Client-Server-Kommunikation von ISR mit Stale-while-revalidate-Mechanismus

Neben dieser passiven Art der Revalidierung existiert auch ein aktiver Ansatz. Dieser wird als On-Demand Revalidation bezeichnet. Damit wird es ermöglicht, den Cache manuell zu revalidieren. Dazu kann z. B. nach der Aktualisierung des Inhalts einer Webseite in einem CMS ein serverseitiges API aufgerufen werden, das die Revalidierung anstößt. Diese Variante sollte verwendet werden, wenn sich der Inhalt nur von Zeit zu Zeit ändert. Selbstverständlich können beide Varianten auch kombiniert eingesetzt werden.

Das inkrementelle Vorgehen hat den Vorteil, dass zur Build-

Zeit nicht alle Seiten auf einmal gebaut werden müssen und Anpassungen schneller produktiv sind. Das soll am Beispiel einer E-Commerce-Webseite mit 100 000 Produkten demonstriert werden. Wir nehmen an, dass die Generierung einer Produktseite im optimistischen Mittel lediglich zehn Millisekunden dauert. Ein kompletter Build würde also knapp 17 Minuten in Anspruch nehmen. Für eine Webseite, die schnell auf Preisanpassungen reagieren muss, könnte das bereits zu lang sein. Eine Lösung wäre, nur die 1 000 populärsten Produkte zum Zeitpunkt des Build mit ISR zu generieren. Die restlichen 99 000 Produkte können just in time erzeugt werden, sobald sie angefordert werden. Das würde die Build-Zeit auf wenige Sekunden reduzieren.

Ein weiterer Vorteil von ISR ist die Persistierung der Seiten zwischen verschiedenen Deployments. Das bedeutet, dass es möglich ist, sofort ein Rollback durchzuführen, ohne die generierten Seiten auszutauschen. Beispielsweise könnte nach dem Deployment mit der ID *A* auf der Webseite in Version 1 ein Tippfehler festgestellt werden. Dieser wird im CMS korrigiert. Durch die automatische Revalidierung ist kein erneutes Deployment notwendig. Es wird automatisch die aktualisierte Webseite in Version 2 erzeugt und im Cache abgelegt. Anschließend wird ein Feature entwickelt und in einem weiteren Deployment mit ID B bereitgestellt. Dieses Feature ist jedoch fehlerhaft. Daher geschieht ein Rollback zum Deployment A. Obwohl der Schreibfehler zum Zeitpunkt des Deployments A bestand, ist er nach dem Rollback nicht mehr vorhanden, da die Seite mit Version 2 unabhängig vom Deployment persistiert wurde. Dieser Mechanismus kann jedoch auch als Nachteil ausgelegt werden, da damit atomare, unveränderliche (immutable) Deployments nicht mehr garantiert werden können [2].

Ein Problem bei diesem Pattern ist, dass die ausgelieferten Seiten nicht für jeden Nutzer dieselben sind. Aufgrund des Stale-while-revalidate-Mechanismus kann es vorkommen, dass

zwischen der Änderung der Webseite, der erneuten Validierung und der Generierung der neuen Version veraltete Inhalte ausgeliefert werden. In **Abbildung 1** liefert die zweite Antwort des Servers eine veraltete Version der Seite aus. Erst mit der nächsten Anfrage wird der aktuelle Inhalt ausgeliefert. Das ist nicht optimal für den SEO-Score, tritt jedoch nur bei einem Bruchteil der Anfragen auf. Es hat aber größere negative Auswirkungen auf die User Experience und die Developer Experience, da Nutzer verschiedene Inhalte zu sehen bekommen und somit auch das Debugging erschwert wird. Schließlich erhöht dieses Pattern im Gegensatz zum klassischen statischen Rendering die Komplexität sowohl auf der Seite der Infrastruktur als auch auf der Seite der Developer Experience.

Aus diesen Gründen eignet sich Incremental Static Regeneration nicht für kleine Projekte. Es kann sogar völlig unnötig sein, wenn das Intervall für die Revalidierung größer als die gesamte Build-Zeit ist. Seine Stärken spielt es bei einer großen Anzahl statischer Seiten aus, z. B. bei E-Commerce-Webseiten. Es ist auch denkbar, dass nur stark frequentierte Seiten, wie z. B. Landing Pages, vorgerendert werden und die restlichen Seiten zur Laufzeit generiert werden. Ursprünglich stammt dieses Pattern vom auf React aufsetzenden Metaframework Next.js ab, findet jedoch auch Einzug in weitere Frameworks, wie beispielsweise dem Pendant Nuxt im Vue-Umfeld. Zusammengefasst sehen Sie die Vor- und Nachteile von ISR in Tabelle 1 und **Abbildung 2**.

Vorteile	**Nachteile**
performant	wenig Dynamik
niedrige TTFB	keine native User Experience
TTI = FCP	erhöhte Komplexität
SEO-freundlich	potenziell veraltete Seiten

Vorteile	Nachteile
CDN-fähig (Skalierbarkeit, Cache)	keine atomaren Deployments
mit deaktiviertem JS nutzbar	
wenige Angriffsvektoren	
kürzerer Build-Prozess als SSG	
geringerer Ressourcenverbrauch als SSG	
Fallback durch Cache	
Persistierung der Seiten unabhängig vom Deployment	

Tabelle 1: Vor- und Nachteile von ISR

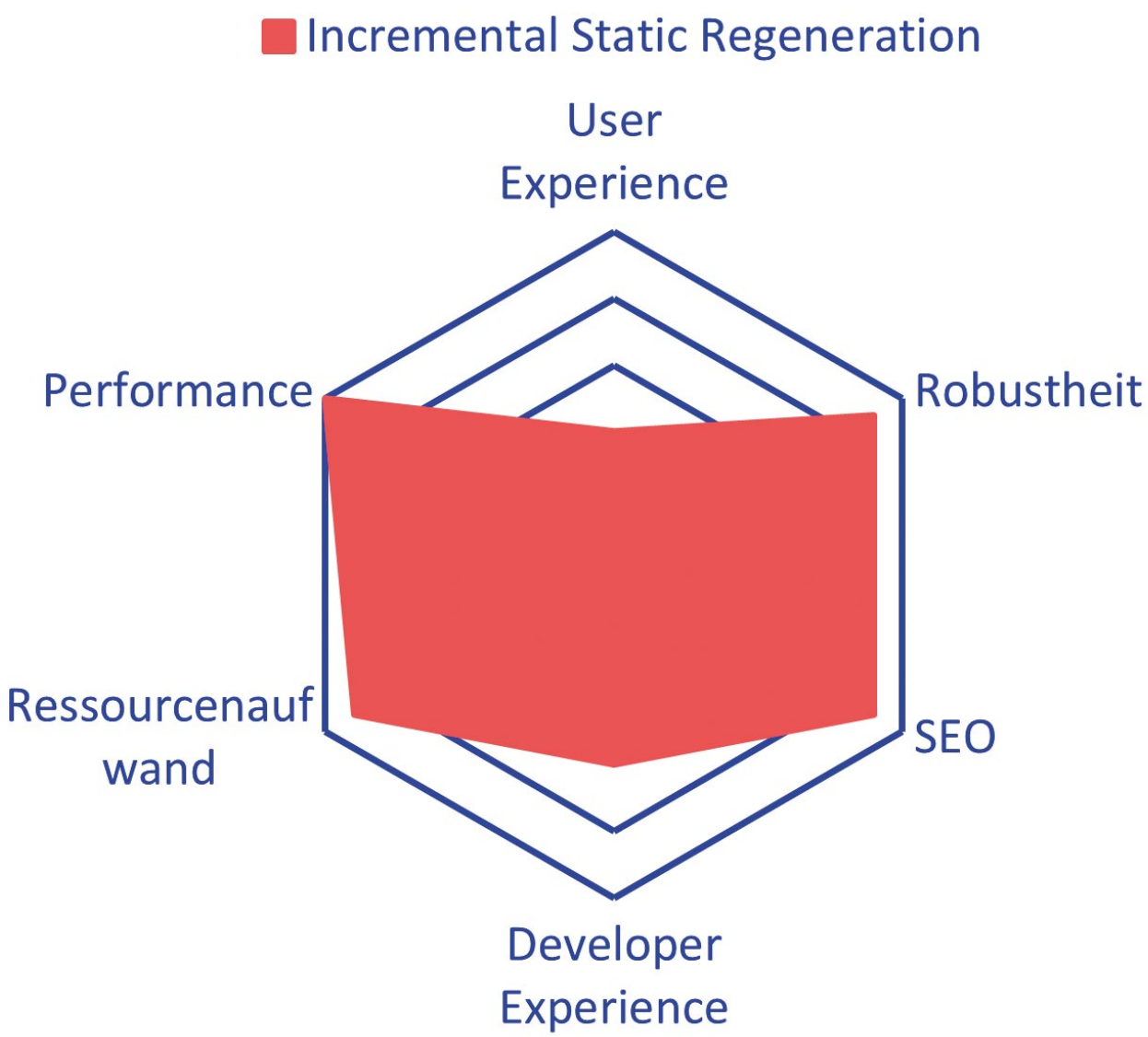

Abb. 2: Bewertung ISR: Developer und User Experience steigen

im Vergleich zu SSG

Clientseitiges Rendering mit Prerendering

Einer der größten Nachteile von clientseitig gerenderten SPAs ist ihre schlechte SEO-Unterstützung. Das initiale HTML-Dokument enthält nur eine leere Hülle mit einem Einstiegspunkt für die JavaScript-Applikation. Crawler können daher keinen auswertbaren Inhalt finden. Um den SEO-Score zu verbessern, müsste bei der initialen Antwort des Servers mehr interpretierbarer Inhalt im Dokument geliefert werden.

Daher gibt es sogenannte Prerender Services oder -Tools, die als Middleware oder Plug-in in die bestehende Anwendungsinfrastruktur integriert werden können. Sie nutzen den Code der SPA, um daraus eine statische Version zu rendern. Dieses statische HTML wird dann initial ausgeliefert (**Abb. 3**). Anschließend wird der Java-Script-Code geladen, der die Kontrolle übernimmt.

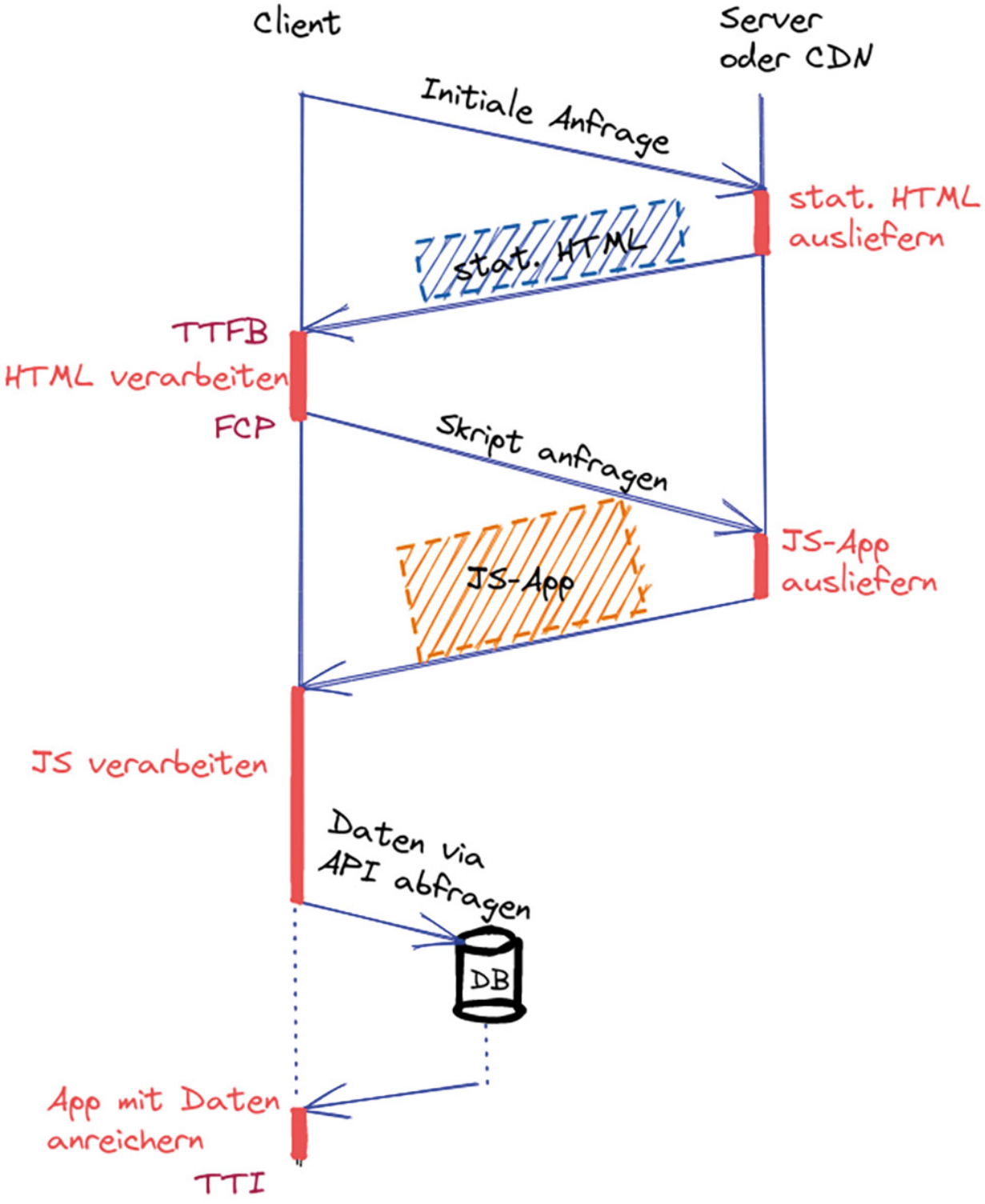

Abb. 3: Zuerst wird das statische HTML ausgeliefert, anschließend die SPA geladen

Das Prerendering kann zu unterschiedlichen Zeitpunkten erfolgen. Entweder vorab zur Build-Zeit oder automatisiert in einem definierten Intervall. Bei der Variante zur Build-Zeit wird die SPA einmalig vorgerendert. Dadurch wird der Build-

Prozess aufwendiger und die Inhalte können nur aktualisiert werden, wenn die Anwendung neu gebaut wird. Dieser Nachteil wird abgemildert, wenn die Anwendung stattdessen in einem festgelegten Intervall neu gerendert wird.

Da es sich beim Prerendering-Ansatz um eine Kombination aus dem CSR- und dem SSG-Pattern handelt, gelten größtenteils deren Vor- und Nachteile. So ist es beispielsweise nicht möglich, dynamische Inhalte vorab zur Build-Zeit zu rendern.

Prerendering steigert den SEO-Score. Darüber hinaus verbessert es die Zeit zum FCP, da die ersten Inhalte direkt nach der Interpretation des HTML-Dokuments gerendert werden, anstatt eine leere Seite anzuzeigen. Außerdem wird die Seite dadurch resistenter, da auch ohne JavaScript Inhalte gerendert werden und Inhalte nicht pro Anfrage gerendert werden müssen.

Durch die Verwendung eines externen Rendering-Diensts entsteht jedoch eine weitere Abhängigkeit, die außerhalb der eigenen Kontrolle liegt. Zusätzlich kann die Generierung zur Laufzeit zu einer längeren Antwortzeit (TTFB) führen, abhängig davon, wie schnell der Prerender Service ist.

Dieses Pattern kann zusätzlich um eine Laufzeitkomponente erweitert werden. Abhängig vom anfragenden User Agent wird unterschieden, ob die clientseitig gerenderte oder die statische Webseite ausgeliefert werden soll (**Abb. 4**). So erhält ein Crawler, für den JavaScript unerheblich ist, die statische Variante und ein normaler Nutzer die statische Seite mit dem SPA-Code. Das wird auch Dynamic Rendering [3] genannt.

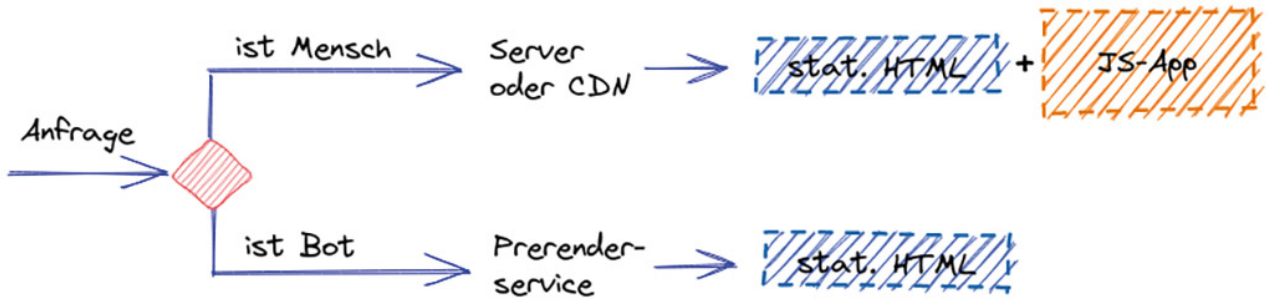

Abb. 4: Dynamic Rendering mit Prerender Service

Prerendering eignet sich, wenn SEO-Verbesserungen an einer bestehenden SPA vorgenommen werden sollen und eine Migration zu einer serverseitigen Lösung nicht im Verhältnis zum Aufwand steht. Vor allem dann, wenn die Seiten statische Daten beinhalten und sich deren Inhalte nur wenig ändern. Ein Dienst, der dieses Pattern anbietet, ist prerender.io [4]. In Tabelle 2 und **Abbildung 5** sehen Sie die Vor- und Nachteile von Prerendering auf einen Blick.

Vorteile	**Nachteile**
gute User Experience	TTI > FCP (verbesserte Zeit zum FCP gegenüber CSR)
Trennung zwischen Client- und Servercode	verteiltes mentales Modell
CDN-fähig	potenziell längere TTFB als bei CSR
SEO-freundlich	viele Angriffsvektoren
nachrüstbar	Abhängigkeit von externen Services
mit deaktiviertem JS eingeschränkt nutzbar	keine Dynamik

Tabelle 2: Vor- und Nachteile von Prerendering

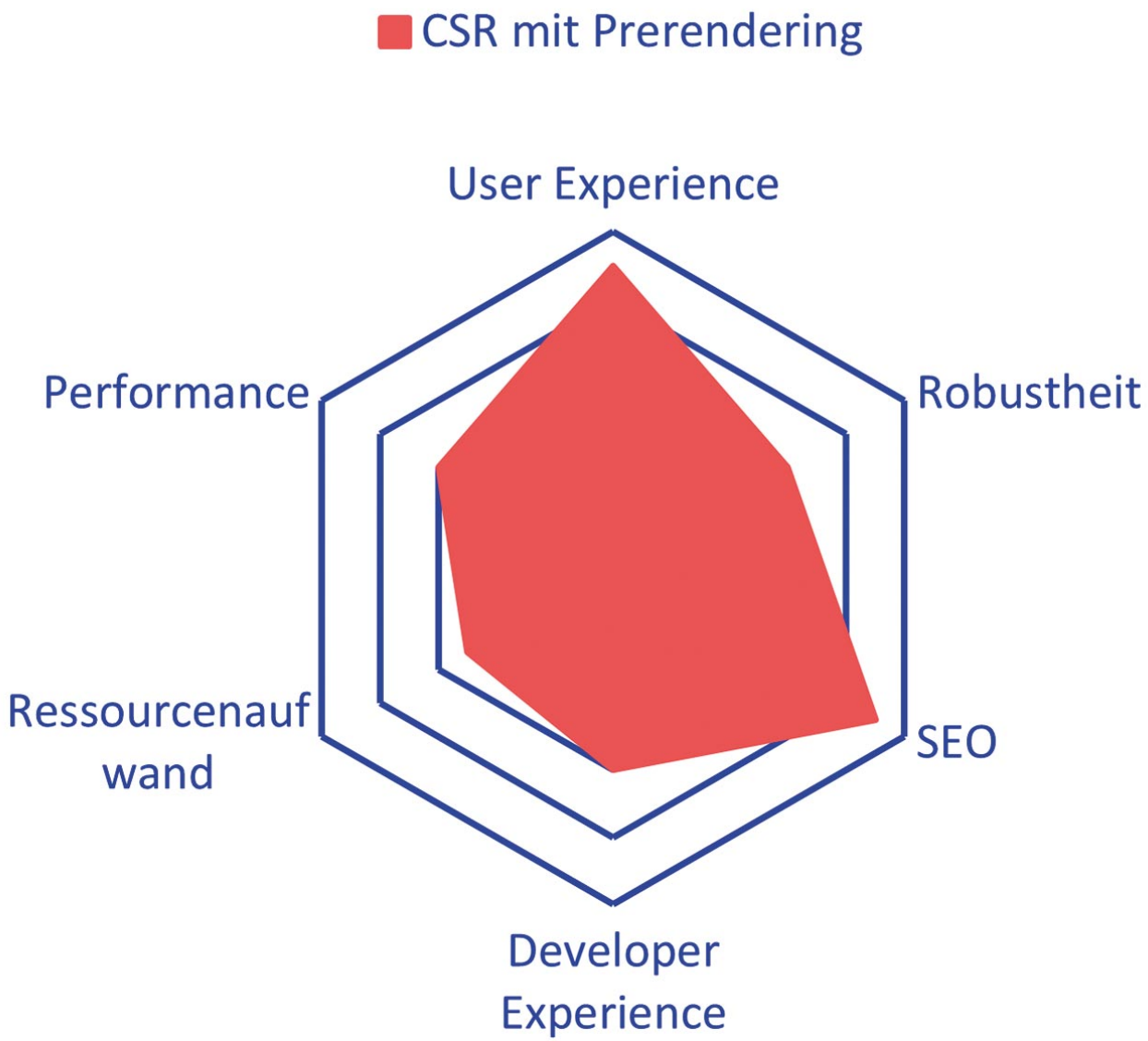

Abb. 5: Prerendering verbessert den SEO-Score erheblich

Serverseitiges Rendering mit Hydration

Neben dem Prerendering-Ansatz zur Build-Zeit gibt es noch eine weitere Möglichkeit, die Probleme von clientseitig gerenderten SPAs zu lösen: serverseitiges Rendering mit einem Hydrationsschritt. Hier findet das Rendering zur Laufzeit statt. Oft wird dieser Prozess auch als Hydration oder isomorphes bzw. universales Rendering bezeichnet. Das Pattern ist eine Kombination aus SSR und CSR.

Die Bezeichnung isomorph ist deshalb zutreffend, weil sowohl auf dem Server als auch auf dem Client der SPA-Code gerendert

wird. Begonnen wird dabei auf der Serverseite (**Abb. 6**). Dazu werden serverseitig die notwendigen Daten geladen und das DOM bzw. der Komponentenbaum der Anwendung damit angereichert. Anschließend wird daraus das HTML-Dokument gerendert und an den Client ausgeliefert. So erhält der Client schnell eine Seite mit interpretierbarem Inhalt, anstatt eine leere Seite anzuzeigen.

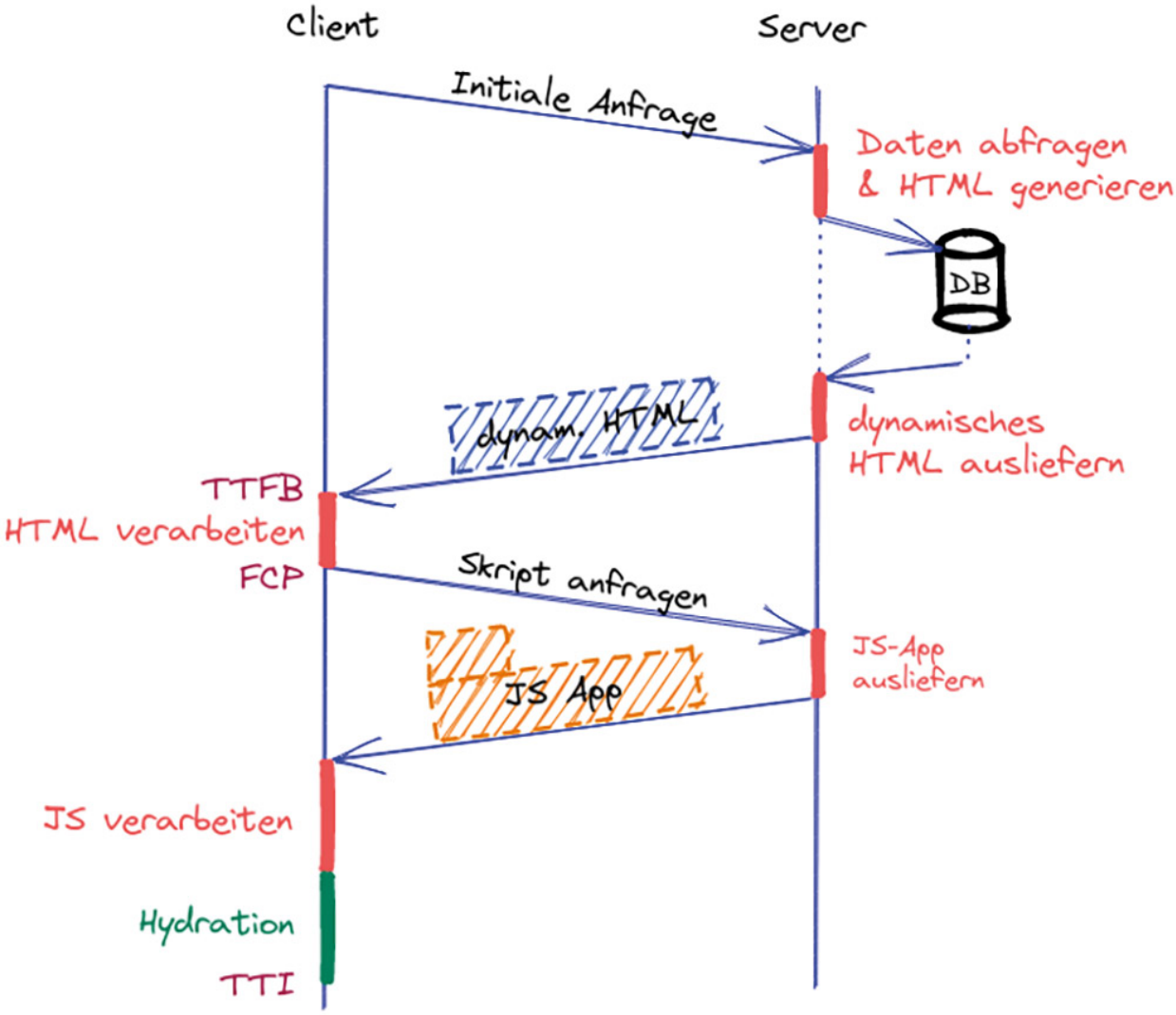

Abb. 6: Client-Server-Kommunikation – neu ist der Hydration-Schritt am Ende

Im zweiten Schritt wird clientseitig der JavaScript-Code für die SPA geladen und der Komponentenbaum erzeugt. Im Anschluss startet die Hydration, die die servergerenderte Anwendung auf dem Client wiederherstellt. Dazu wird die Initialisierung der gesamten Anwendung erneut abgespielt. Dabei wird das DOM mit

Event Handlern versehen und Zustände einzelner Komponenten wiederhergestellt. Es ist, als würde das zuvor „trockene“ HTML mit dem „Wasser“ der Interaktivität und des Event Handling hydriert werden [5]. Sobald dieser Prozess für alle Komponenten abgeschlossen ist, verhält sich die Anwendung wie eine klassische SPA, da ab diesem Zeitpunkt der JavaScript-Code die Kontrolle übernimmt.

Die Vorteile dieses Ansatzes liegen auf der Hand. Statt wie bei einer rein clientseitig gerenderten SPA können dynamische Inhalte schneller angezeigt werden und sind auch für Suchmaschinen indizierbar. Die Größe des Java-Script Bundle hat kaum Einfluss auf die FCP-Metrik, da bereits vor dem clientseitigen Laden des Bundles serverseitig generierte Inhalte gerendert werden. Nach dem Laden bestehen weiterhin die Vorteile des clientseitigen Rendering, wie z. B. eine native User Experience. Somit haben wir alle unsere Probleme gelöst. Oder? Der Schein trügt. Hydration ist nämlich ein ineffizienter und fehleranfälliger Prozess. Große Teile der Webseite sind doppelt vorhanden. Einmal im servergerenderten HTML-Dokument und einmal im JavaScript Bundle für den Client. Diese Doppelung erhöht den Datenverbrauch und die Auslastung des Clients, da sowohl das HTML als auch das JavaScript geladen, interpretiert und ausgeführt werden müssen. Je nach Endgerät kann dieser Vorgang längere Zeit in Anspruch nehmen.

Da die clientseitige Anwendung den vom Server generierten Code übernimmt, muss der Komponentenbaum des Servers genau mit dem des Clients übereinstimmen. Um das zu gewährleisten, muss vor der Hydration abgewartet werden, bis das JavaScript vollständig geladen wurde. Ansonsten kann es im besten Fall zu einer Verlangsamung der Applikation führen, im schlimmsten Fall könnten beispielsweise Event Handler für das falsche Element initialisiert werden [6]. Auch können Situationen auftreten, bei denen das initial vom Server erzeugte DOM zerstört wird und komplett neu gerendert werden muss.

Das größte Problem stellt das sogenannte Uncanny Valley dar.

Solange die SPA clientseitig nicht initialisiert ist (d. h., der Hydration-Schritt für alle Komponenten abgeschlossen ist), kann der Nutzer nicht mit der statischen Seite interagieren, obwohl diese bereits durch das serverseitig generierte HTML interaktiv erscheint.

Aus diesen Gründen ist Hydration ein komplexes Thema [7] und es werden inkrementelle Verbesserungen vorgenommen. Beispielsweise arbeitet das React-Team seit einigen Jahren an der Suspense-Architektur, die es erlaubt, nur Teile des Komponentenbaums zu rendern und zu hydrieren. Dieser Ansatz ist unter [5] sehr gut beschrieben. In den folgenden Abschnitten werden daher zwei weitere Varianten des Hydration-Patterns betrachtet, wodurch die Probleme reduziert werden sollen: progressive und partielle Hydration.

Hydration ist heutzutage bei vielen Frontend-Frameworks der Standard für die Entwicklung von Webapplikationen. Durch die dynamische Berechnung zur Laufzeit ist es weitläufig einsetzbar. Trotzdem ist es keine One-size-fits-all-Lösung. Für überwiegend statische Seiten könnte beispielsweise ISR genutzt werden. Alle großen JavaScript-Frameworks wie React oder Vue unterstützen Hydration. In der Regel wird jedoch auf die darauf aufbauenden Metaframeworks wie Next.js oder Nuxt zurückgegriffen, da diese Frameworks den gesamten Prozess rund um Hydration automatisieren. Zusammengefasst sehen Sie die Vor- und Nachteile von Hydration in Tabelle 3 und **Abbildung 7**.

Vorteile	**Nachteile**
performant	potenziell hohe TTFB
SEO-freundlich	TTI > FCP (Uncanny Valley)
gute User Experience	initiales Rendering abhängig von Konnektivität
dynamische, personalisierte Inhalte	Performance abhängig von Nutzerzahl und Serverstandort

Vorteile	Nachteile
mit deaktiviertem JS eingeschränkt nutzbar	ineffizient und fehleranfällig
	mangelnde CDN-Fähigkeit
	viele Angriffsvektoren

Tabelle 3: Vor- und Nachteile von Hydration

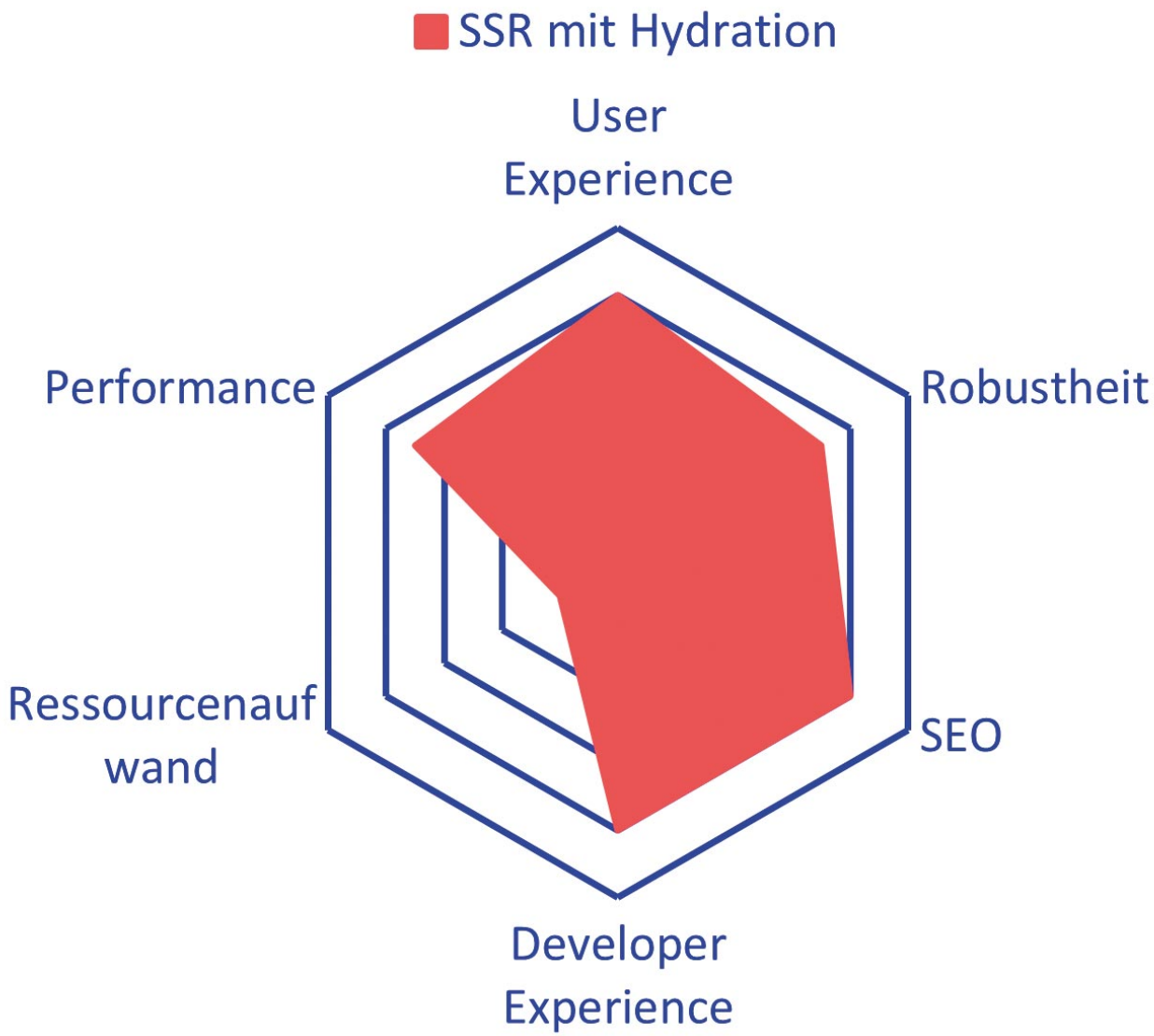

Abb. 7: Hydration ist ein guter Allrounder, wäre da nicht die verschwenderische Ressourcennutzung

Progressive Hydration

Wie wir gesehen haben, stellt die initiale Hydration aller Komponenten ein großes Problem dar. Es ist daher naheliegend

zu versuchen, diesen Aufwand zu reduzieren. Ein Ansatz dazu kann Lazy Loading sein. Diese Idee wurde bereits 2016 in einem Blogbeitrag [8] unter dem Namen „Progressive Booting“ beschrieben.

Anstatt den gesamten Komponentenbaum auf einmal zu hydrieren, werden zunächst nur die nötigsten Komponenten initialisiert. Die Hydration von weniger relevanten Zweigen im Komponentenbaum kann anfangs ausgesetzt und zu einem späteren Zeitpunkt angestoßen werden. Als Auslöser können hierfür der Viewport, die Interaktionswahrscheinlichkeit, oder die aktuelle Auslastung des Clients dienen. Beispielsweise kann eine Footer-Komponente, die sich außerhalb des Viewports befindet, erst zu dem Zeitpunkt hydriert werden, wenn sie sichtbar wird (**Abb. 8**). Hierfür wird der JavaScript Chunk für den Footer erst zu diesem Zeitpunkt angefordert.

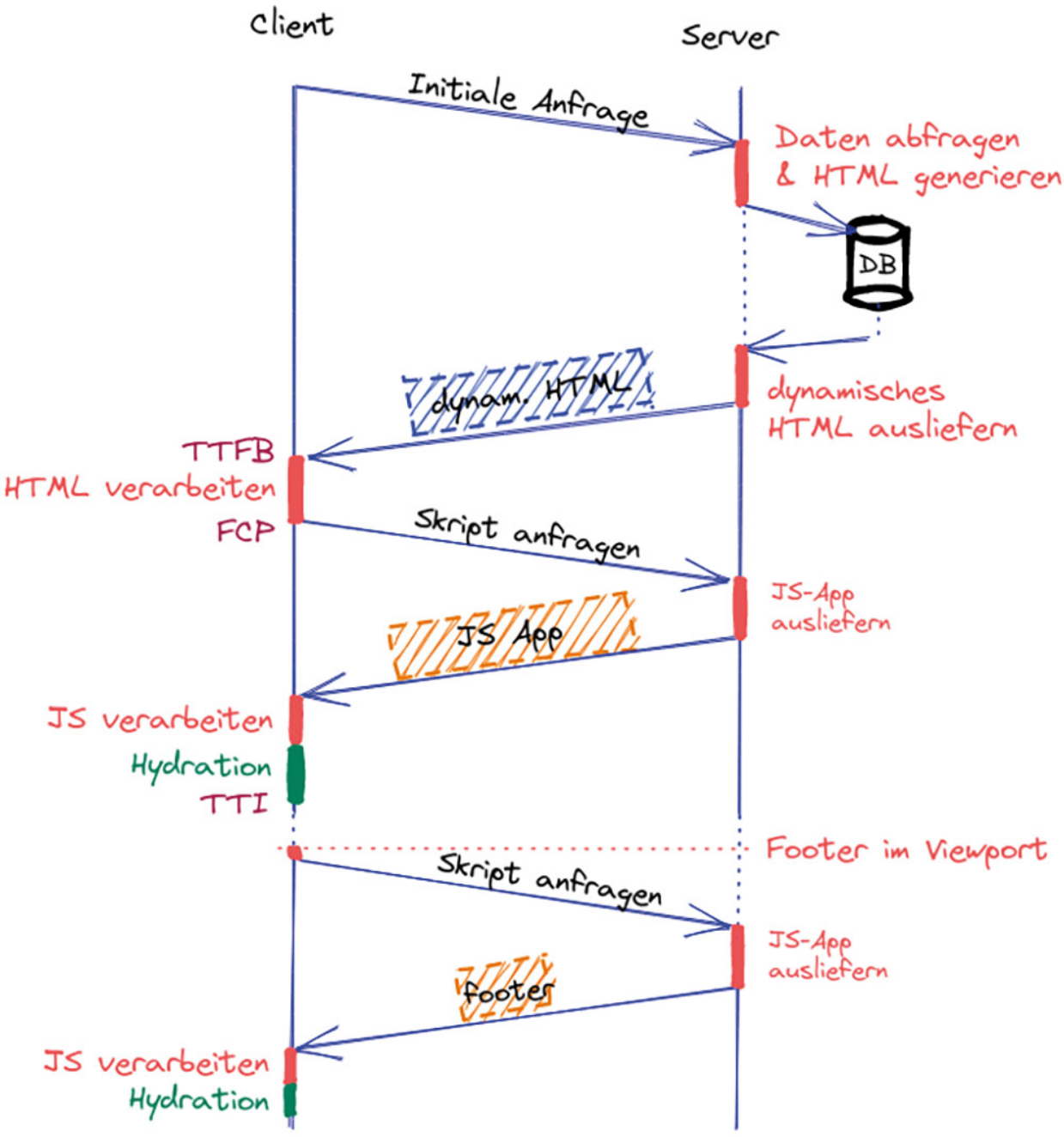

Abb. 8: Die Hydration für den Footer findet erst zu einem späteren Zeitpunkt statt

Für die Aufteilung der Applikation in einzelne Chunks müssen geeignete Grenzen und Prioritäten definiert werden. Diese können unter anderem vom Entwickler vorgegeben werden. Ein Beispiel hierfür sind Astros Clientdirektiven [9], die pro Komponente definiert werden können. Durch diesen progressiven Ansatz verringert sich die initiale Größe des JavaScript-Bundles, da bei der ersten Anfrage nur noch das Notwendigste versendet wird. Somit reduziert sich auch das Uncanny Valley,

da es die TTI für die Hauptkomponenten verbessert.

Wahre progressive Hydration ist aufgrund der vielen Fallstricke und Komplexität schwer zu erreichen. Ein Großteil der heutigen Frameworks basiert immer noch auf einem Top-down-Ansatz bei der Hydration, da diese Frameworks ursprünglich dazu gedacht waren, eine SPA zu bauen. Dadurch ist nur ein einzelner Einstiegspunkt für die Hydration gegeben. Somit muss ein großer Teil des Codes geladen und ausgeführt werden, selbst wenn lediglich eine Komponente am Ende des Komponentenbaumes hydriert werden muss. Für den progressiven Ansatz wären jedoch mehrere unabhängige Einstiegspunkte besser. Optimalerweise genau an der Stelle im Komponentenbaum, die hydriert werden soll.

Außerdem gestaltet es sich in der Praxis als schwierig, die richtige Grenze für die Chunks zu ziehen. Aufgrund der Eltern-Kind-Beziehungen kann es sein, dass die Grenze an einer Stelle gezogen wird, die von einem anderen Teil des Baumes abhängig ist. Daher ist es notwendig, auch diesen Teil zu hydrieren. Neben diesen Punkten gibt es noch weitere, die eine gute Umsetzung von progressiver Hydration erschweren. Für interessierte Leser empfehle ich die Quellen [7] und [10].

Progressive Hydration kann überall dort eingesetzt werden, wo auch Hydration in Frage kommt und die verbesserte Performance und Ressourcennutzung notwendig sind. Ein Framework, das progressive Hydration implementiert, ist das bereits erwähnte Astro. Aber auch andere Frameworks aus dem JavaScript-Ökosystem bieten diesen Ansatz an. Die Vor- und Nachteile sehen Sie zusammengefasst in Tabelle 4 und **Abbildung 9**.

Vorteile	**Nachteile**
siehe Tabelle 3	siehe Tabelle 3
effizienter durch kleineres initiales JavaScript Bundle	erhöhte Komplexität

Vorteile	Nachteile
geringere TTI, Reduktion des Uncanny Valley	

Tabelle 4: Vor- und Nachteile von progressiver Hydration

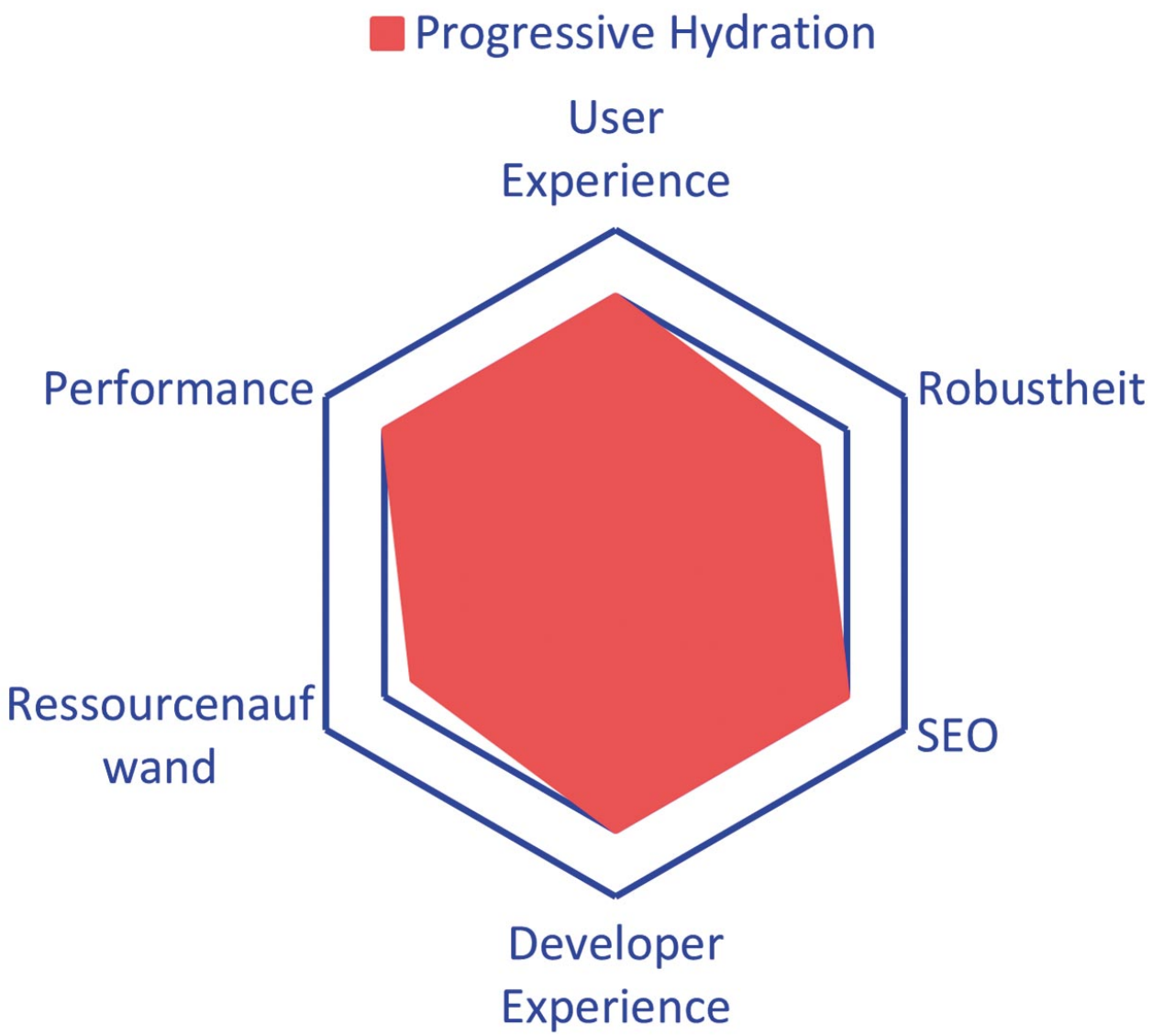

Abb. 9: Progressive Hydration reduziert den initialen Ressourcenaufwand

Partielle Hydration und Island Architecture

Partielle Hydration und Island Architecture werden häufig miteinander verwechselt oder als Synonyme verwendet. Die

Abgrenzung dieser beiden Begriffe ist nicht klar definiert. Das Ergebnis ist jedoch in beiden Fällen das gleiche. Im Gegensatz zum progressiven Ansatz, der sich auf den Zeitpunkt der Hydration auswirkt, ist es mit Hilfe der partiellen Hydration möglich, nur ausgewählte Teile des Komponentenbaums zu hydrieren.

Stellen Sie sich eine Anwendung vor, bei der nur zwei unabhängige kleine Teile interaktiv sind. Der Rest ist statisch. Wie zuvor bereits erläutert wurde, muss für diesen kleinen interaktiven Teil der gesamte Baum hydriert werden. Das ist ineffizient. Eine Lösung wäre, nur genau diese dynamischen Teile zu hydrieren (**Abb. 10**).

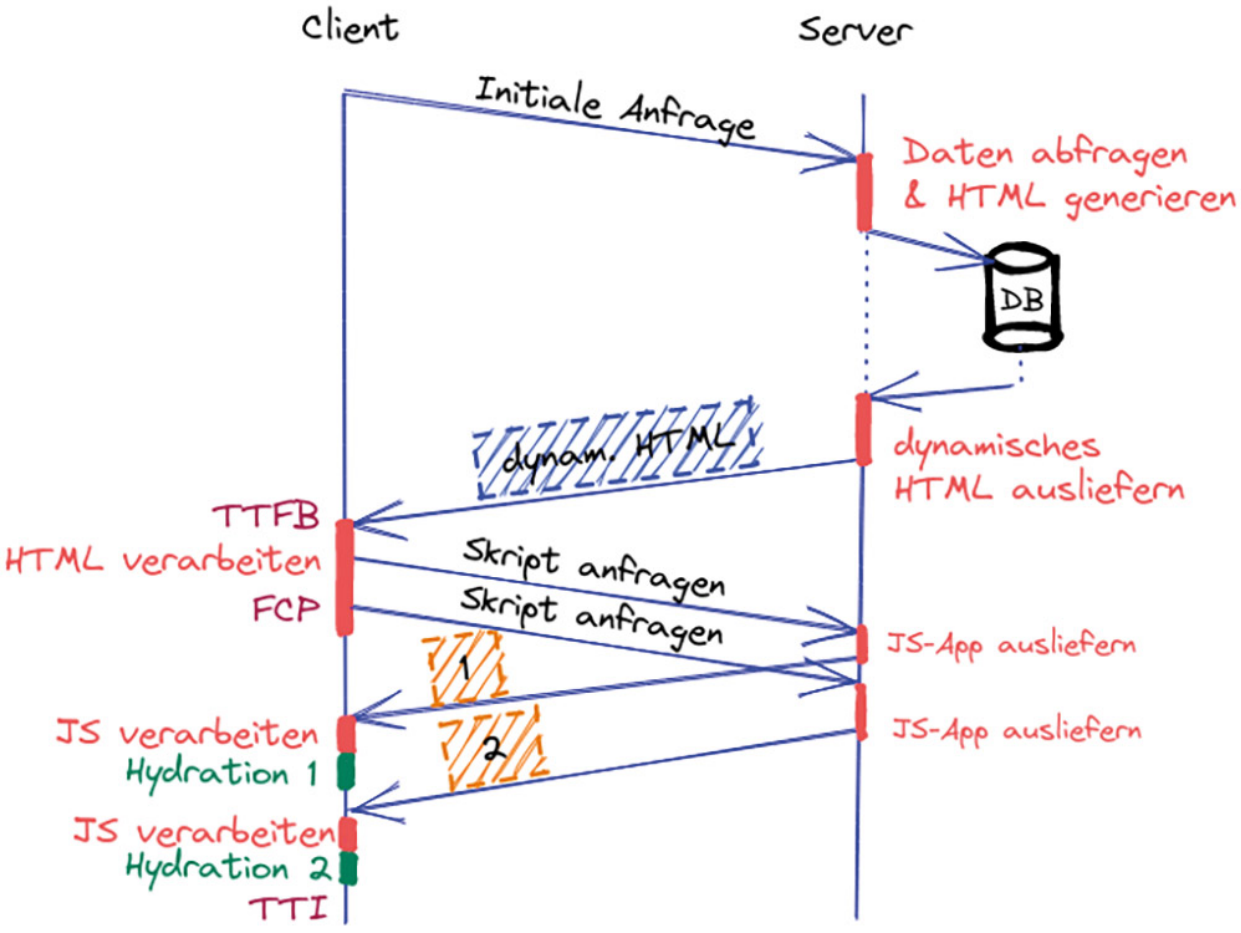

Abb. 10: Die Inseln werden unabhängig angefragt und verarbeitet

Dazu müssen geeignete Grenzen definiert werden. Das Ergebnis

ist einerseits ein statischer Teil aus HTML und andererseits dynamische Bereiche bestehend aus HTML und JavaScript. Letztere werden als Inseln bezeichnet. Damit verschwindet der zuvor bestehende Top-down-Ansatz des Komponentenbaumes, da nur noch unabhängige Teile mit verschiedenen Einstiegspunkten existieren. In unserem Beispiel hätten wir jetzt zwei unabhängige Inseln.

Für die Definition der Grenzen gibt es verschiedene Ansätze. Einige sind manueller Natur, andere arbeiten automatisch. Das bereits erwähnte Astro-Framework überlässt diese Aufgabe dem Entwickler. Dieser muss anhand der Clientdirektiven definieren, welche Teile nicht statisch sind. Andere Frameworks wie Fresh lösen das Problem, indem der Code für die Inseln in einem speziellen Ordner abgelegt werden muss. Wieder andere (z. B. Marko.js) erkennen die Inseln automatisch über einen Compiler.

Ähnlich wie beim progressiven Ansatz sinkt auch hier die Größe des clientseitigen Codes. Dadurch, dass sehr viel weniger teuer auszuführendes JavaScript ausgeliefert wird, ist die Anwendung schnell interaktiv. Das vermindert den Uncanny-Valley-Effekt. In Kombination mit progressiver Hydration kann dieser sogar ganz entfallen, da initial kein JavaScript mehr ausgeliefert werden muss. Die Seite ist somit wie beim statischen oder klassischen serverseitigen Rendering sofort interaktiv.

Da es sich um isolierte Inseln handelt, können sie auch unabhängig vom Rest der Seite geladen werden. Im Gegensatz zum Top-down-Ansatz verzögert keine Elternkomponente die Initialisierung einer Insel. Diese Vorteile werden jedoch durch eine erhöhte Komplexität erkauft. Das manuelle Setzen von Grenzen kann fehleranfällig und kompliziert sein [11]. Einen Compiler zu schreiben, der dieses Problem löst, ist eine große Herausforderung, da die Architektur des Frameworks dies berücksichtigen muss.

Ein bereits erwähnter Vorteil der Inseln ist ihre Unabhängigkeit vom Rest der Seite, erschwert ist jedoch die Kommunikation zwischen den Inseln, z. B. um den globalen Anwendungszustand zu teilen. Im klassischen Ansatz können Daten über Props ausgetauscht werden. Da die Komponenten jedoch isoliert voneinander sind, ist dieser Mechanismus nicht mehr möglich. Daher wird eine zusätzliche Vermittlungsschicht benötigt, z. B. ein globaler Speicher für den Anwendungszustand, der von allen Inseln angesprochen werden kann.

Die partielle Hydration kann in ähnlicher Weise wie die progressive Hydration verwendet werden. Es ist jedoch zu bedenken, dass dieses Pattern die Developer Experience verringern kann, da es mehr Komplexität mit sich bringt. Außerdem ist dieses Pattern nicht für interaktionslastige Seiten geeignet, da solche sehr viele Inseln erfordern würden. Genutzt wird dieses Pattern bei den bereits erwähnten Frameworks Astro und Fresh. Die Vor- und Nachteile finden Sie in Tabelle 5 und **Abbildung 11** zusammengefasst.

Vorteile	**Nachteile**
siehe Tabelle 3	siehe Tabelle 3
effizienter durch kleineres JavaScript Bundle	erhöhte Komplexität
geringere TTI, Reduktion des Uncanny Valley	erschwerter Austausch zwischen Inseln
isolierte Komponenten	

Tabelle 5: Vor- und Nachteile partieller Hydration

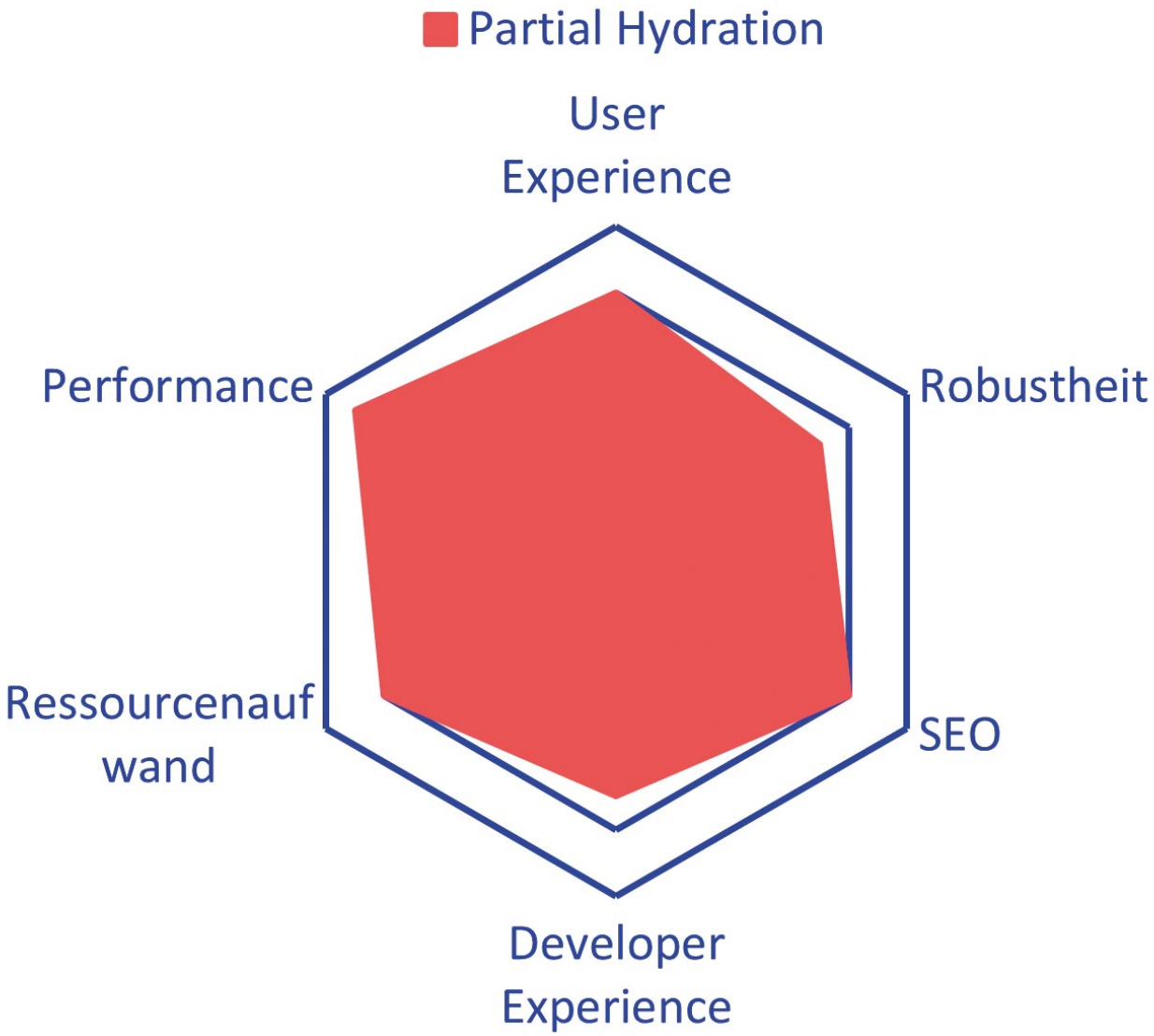

Abb. 11: Partielle Hydration steigert die Performance und reduziert unnötige Daten

Fazit

Wie wir gesehen haben, versuchen alle diese Patterns, die Schwächen der drei grundlegenden Patterns zu mildern, indem sie verschiedene Ansätze daraus kombinieren und neue Ideen einbringen. Anhand der Bewertungen der jeweiligen Patterns ist zu erkennen, dass diese sich immer weiter verbessern. Besonders hervorzuheben sind die Sprünge in den Bereichen Developer Experience und User Experience. Das deckt sich auch mit der Entwicklung des Webs, die wir bereits im ersten Teil der Serie betrachtet haben.

Leider nimmt auch die Komplexität der Lösungen stetig zu. Das ist vor allem bei den Hydrations-Ansätzen zu beobachten, die heutzutage häufig den Standard in der Entwicklung mit JavaScript-Frameworks bilden. In letzter Zeit tauchen aber auch vermehrt neuartige Ansätze auf, die teilweise gänzlich ohne Hydration auskommen, wie z. B. Resumability. Unter anderem auf dieses Pattern werden wir im kommenden und letzten Teil der Serie eingehen.

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.

Links & Literatur

[1] https://web.dev/stale-while-revalidate/

[2] https://www.netlify.com/blog/2021/03/08/incremental-static-regeneration-its-benefits-and-its-flaws/

[3] https://developers.google.com/search/docs/crawling-indexing/javascript/dynamic-rendering

[4] https://prerender.io

[5] https://github.com/reactwg/react-18/discussions/37

[6] https://www.joshwcomeau.com/react/the-perils-of-rehydration/

[7] https://dev.to/this-is-learning/why-efficient-hydration-in-javascript-frameworks-is-so-challenging-1ca3

[8] https://aerotwist.com/blog/when-everything-is-important-nothing-is/

[9] https://docs.astro.build/en/reference/directives-reference/#client-directives

[10] https://www.builder.io/blog/why-progressive-hydration-is-harder-than-you-think

[11] https://www.theguardian.com/info/2022/mar/25/react-islands-on-theguardiancom