

Das neue JavaScript Framework Qwik.js - Mit Resumability zur optimalen Performance im Web?

1. Einleitung

In einer zunehmend vernetzten und digitalen Ära ist das Web zu einem integralen Bestandteil unseres täglichen Lebens geworden. Dabei hat die kontinuierliche Evolution des Internets die Anforderungen der User an heutige Webseiten und -anwendungen in Bezug auf Funktionalität, Geschwindigkeit und Reaktionsfähigkeit erheblich gesteigert. Eine entscheidende Herausforderung bei der Entwicklung der Anwendungen liegt somit auch in der Optimierung der Performance, um eine nahtlose, effiziente Interaktion zwischen Nutzern und Anwendung sowie eine positive User Experience zu gewährleisten.

Aufgrund des mittlerweile riesigen Angebots und der Menge an Mitbewerbern in nahezu jeder Branche im Web sind die Konsumenten der Inhalte anspruchsvoller als je zuvor und der Page Speed ist ein entscheidender Erfolgsfaktor für jedes Online-Unternehmen. Webseiten, die schnell laden und ohne Verzögerung auf Nutzerinput reagieren, halten die User nicht nur länger auf der Seite, sondern verbessern nachweislich auch die Conversion Rate, wie beispielsweise eine Anfang diesen Jahres veröffentlichte Case Study von Rakuten zeigt. [1] Die intensiven Bemühungen des Online-Retailers in die Optimierung der Performance ihres Shops lässt sich dabei in Handfesten zahlen darstellen: [2]

- 53,37 % mehr Umsatz pro Besucher
- 33,13 % höhere Conversion-Rate
- 15,20 % Steigerung des durchschnittlichen Bestellwerts
- Reduzierung der Exit-Rate um 35,12 %

Ein entscheidender Schlüssel für diesen signifikanten Anstieg war für Rakuten dabei neben der Bildoptimierung und Anpassung der CSS-Dateien vor allem auch eine Verbesserung und Reduktion des JavaScript-Codes der Seite. [2]

Mit Blick auf Abbildung 1 scheint diese Maßnahme wichtiger denn je zu sein. Heutige Webseiten- und Applikationen bieten Usern mehr Möglichkeiten als noch vor 10 oder 15 Jahren und ein nie zuvor dagewesenes Maß an Interaktivität und Funktionalität im Web. Diese Entwicklung resultiert jedoch in einer deutlichen Zunahme des benötigten JavaScript-Codes, tendenziell weiter steigend, wodurch sich im Bereich der Webentwicklung mittlerweile immer mehr die Frage stellt: Wie kann JavaScript reduziert und die Auslieferung an den Browser verbessert werden? [3]

In dieser Arbeit wird nach einem Überblick über Trends und Ansätze des Web Developments der letzten Jahre und den damit entstandenen Problemen das Framework Qwik in den Fokus gerückt. Dessen neuer Ansatz der Resumability soll dabei im Kontext der Performance von Webapplikationen genauer beleuchtet und in einer prototypischen, praktischen Anwendung mit bisherigen Standards verglichen werden.

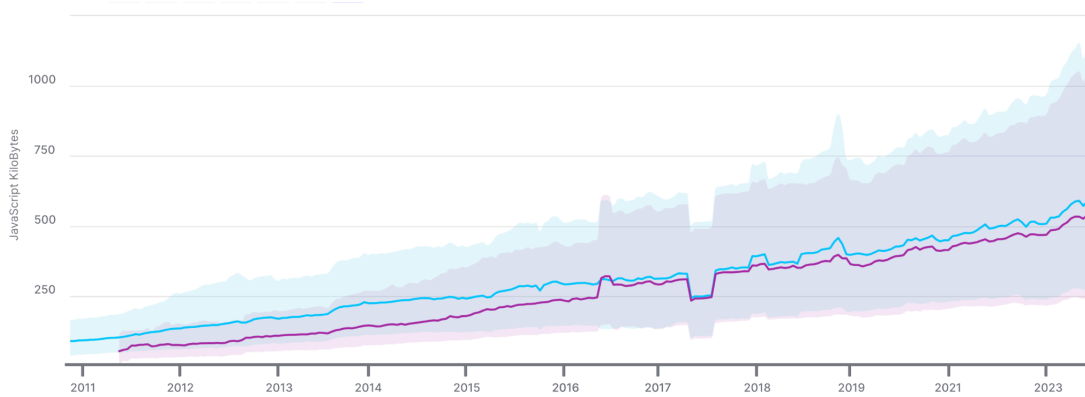


Abbildung 1: Durchschnittliche Menge an JavaScript-KiloBytes einer Webseite im Verlauf der Zeit

2. Ein Überblick über die Geschichte der Webentwicklung

In seinen anfänglichen Jahren war das Web noch weit entfernt von den hochinteraktiven Anwendungen, die wir heute nutzen. Statische Inhalte wurden als HTML-Dokumente auf den Servern bereitgestellt und auf Anfrage des Clients ausgeliefert. Erst mit der Einführung des Document Object Models als Schnittstelle zwischen zu den HTML-Dokument und dem Aufkommen von JavaScript, konnten nun auch mit Hilfe von Programmiersprachen wie PHP oder Java dynamische und personalisierte HTML-Inhalte auf der Serverseite erzeugt werden, die zudem auch noch im Browser auf Eingaben der Nutzer reagierten. [4]

Mit der wachsenden Komplexität von Webanwendungen und dem Bedarf an effizienter Codeverwaltung und Strukturierung schreiben immer mehr Entwickler kleine, hilfreiche Tools in wiederverwendbaren Paketen. Aus den losen Ansammlungen von verschiedenen Funktionen in Bibliotheken entstanden schließlich feste Programmiergerüste, sogenannte Frameworks, die es den Entwicklern erleichtern sollten, moderne und interaktive Webanwendungen zu erstellen. [5]

Mit dem Durchbruch von Frameworks wie Angular, React oder Vue hat sich der HTML-Rendering-Prozess von der Server- auf die Clientseite verschoben. Bei diesen sogenannten Single Page Apps lädt die initiale Anfrage lediglich eine minimale HTML-Datei, die in der Regel nur ein leeres Root-Element sowie Links zu den entsprechenden JavaScript-Files enthalten. Diese werden nun angefragt, kompiliert, ausgeführt und so die HTML-Elemente dynamisch erzeugt und auf dem Display für den User ausgegeben. [6]

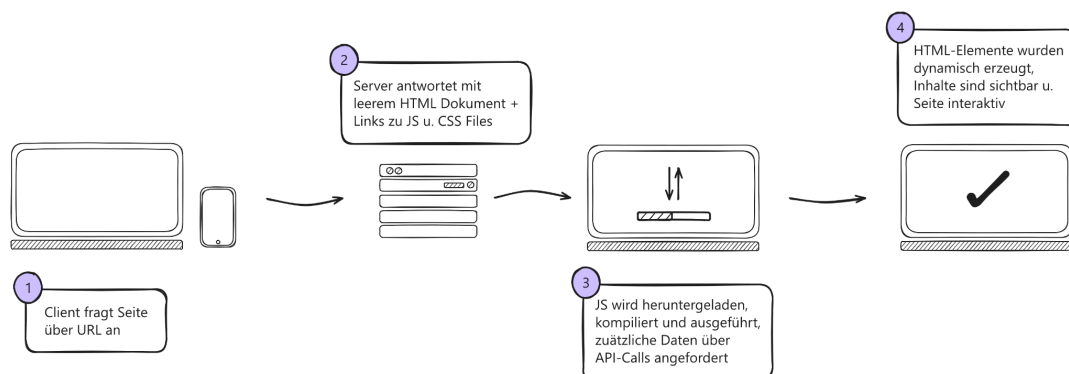


Abbildung 2: Schritte des Client Side Renderings vom Request bis zur Fertigstellung der Seite

Single Page Applications mit clientseitigem Rendering können zum einen die Server entlasten. Zudem sind diese nach dem ersten Ladevorgang sehr flüssig und fühlen sich für die Nutzer oftmals wie eine native App an. Da bei erstmaligem Aufruf der Seite zunächst alle zusätzlichen JavaScript- und CSS-Dateien heruntergeladen und verarbeitet werden müssen, sieht der Nutzer bei dem ersten Ladevorgang für einige Zeit nur einen leeren Screen. Aufgrund dessen, dass die ungerenderte HTML-Seite zunächst nur ein leeres Element enthält, haben auch die Crawler der Suchmaschinen Probleme, den Inhalt der Seite richtig zu erfassen, was sich negativ auf die SEO auswirkt. [7]

3. Meta-Frameworks und das Problem der Hydration

Aus diesen Gründen wurden gegen Ende der 2010er Jahre Erweiterungen und neue Technologieansätze entwickelt, die die Nachteile der Single Page Apps ausgleichen sollten. Diese sogenannten Meta-Frameworks wie beispielsweise Next.js für React oder Nuxt.js für Vue bauen auf den Grundlagen der SPA-Frameworks auf, bieten jedoch neue Funktionalitäten und Möglichkeiten für das Erstellen von Multi-Page-Anwendungen mit Server Side Rendering (SSR) oder Static Site Generation (SSG). Hier werden die HTML-Inhalte nicht erst auf dem Client erstellt, sondern ähnlich wie in den Anfangszeiten des Webs nach eingehendem Request auf dem Server gerendert bzw. bei der Static Site Generation als statische Assets gehostet und anschließend ausgeliefert. [4]

Anstatt leeres HTML zu senden, das dann im Client von Grund auf neu gebaut werden muss, sehen die User so nach dem Seitenaufruf zum einen deutlich schneller die Inhalte und auch die Crawler der Suchmaschinen können den Content besser indexieren. Diese Rendering-Technologie löst jedoch nur einen Teil des Problems. Zwar scheint die Seite schnell geladen zu sein, der User kann jedoch noch nicht mit ihr interagieren. So muss zusätzlich JavaScript-Code geladen und mit dem serverseitig erzeugten HTML verbunden werden, damit die Webanwendung auch wirklich interaktiv ist. [4]

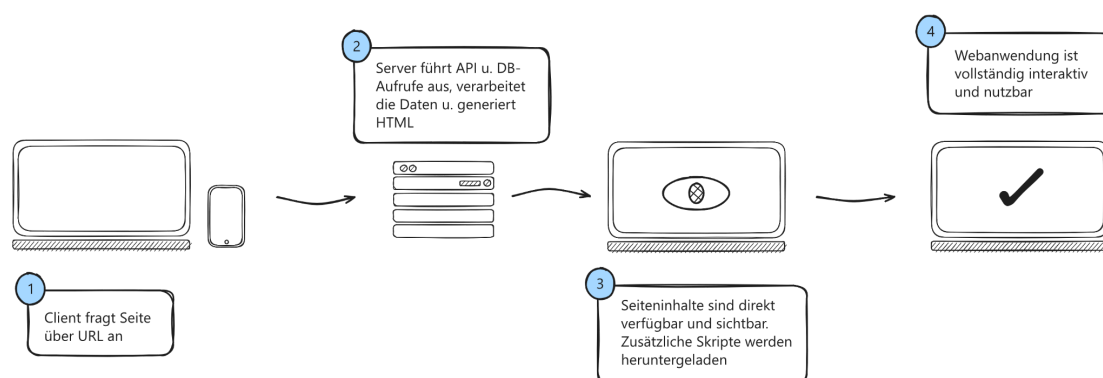


Abbildung 3: Schritte des Server Side Renderings vom Request bis zur Fertigstellung der Seite

3.1 Was versteht man unter Hydration?

Um dieses Ziel zu erreichen, wird die sogenannte Hydration (dt. *Hydratation*) als Lösung genutzt, um dem vom Server gesendeten, statischen HTML wieder Event Handler hinzuzufügen und den State der Anwendung initialisieren, sodass diese wieder im Browser dynamisch und interaktiv ist. [8]

Die technische Herausforderung bei der Hydration besteht nun jedoch darin, zu wissen, welche Event Handler benötigt werden und an welchen Stellen sie an das HTML angehängt werden müssen. Dies ist besonders komplex, da die Event Handler als innere Funktionen, sogenannte Closures auf den Kontext, in dem sie erstellt wurden, zugreifen müssen (z.B. den State der Anwendung, Variablen oder andere Funktionen). Da die Closures meist tief in die Komponenten eingebettet sind und nicht auf oberster Ebene exportiert werden können, muss nun zum Wiederherstellen des States und dem Anbringen der Handler der Code aller Komponenten des aktuellen HTML-Dokuments noch einmal heruntergeladen und ausgeführt werden. Der Prozess der Hydration läuft dabei typischerweise in vier Phasen ab: [9]

- Alle JavaScript-Files vom Server abrufen
- JavaScript parsen und ausführen
- State der Anwendung und die zugehörigen Event Handler wiederherstellen
- Verknüpfen der Event Handler mit den DOM-Elementen

3.2 Das Problem der Hydration

Besonders die ersten drei, der oben beschriebenen Phasen, in denen der Code der Komponenten heruntergeladen, ausgeführt und die State-Informationen wiederhergestellt werden, sind recht zeitaufwendig. Da der Wiederherstellungsprozess proportional zur Komplexität der zu hydrierenden Anwendung ist, kann dies vor allem bei größeren Seiten zu einer schlechten Performance führen und beispielsweise auf mobilen Geräten bis zu 10s dauern. [9]

Es entsteht redundanter Aufwand, da Informationen, die im Backend bereits während des serverseitigen Renderings gesammelt und verworfen wurden, später vom Client wiederhergestellt werden müssen. Nachdem zunächst das gerenderte HTML übertragen wird, erfordert die Hydration ein zweites Senden der Anwendung als JavaScript an den Client und das erneute Ausführen des Framework-Codes. Dies führt zu "Overhead" und einer unnötigen Duplizierung der Arbeit von Server- und Browser. [9]

Die Hydration bietet somit zwar eine Lösung, die HTML-Elemente des DOM bei serverseitig gerenderten Anwendungen wieder mit dem JavaScript des Frameworks zu verknüpfen. Allerdings wird dieser Ansatz aufgrund der zuvor beschriebenen Nachteile von vielen Entwicklern problematisch gesehen. So beschreibt Misko Hevery die Hydration als "schreckliche Umgehung, da die Frameworks nicht berücksichtigen wie Browser tatsächlich funktionieren" [10] und Ryan Carniato kritisiert, dass der JavaScript-Payload dadurch oftmals erhöht wird und es meist sogar länger als bei einem clientseitigen Rendering dauert, bis die Anwendung vollständig geladen und interaktiv ist. [8]

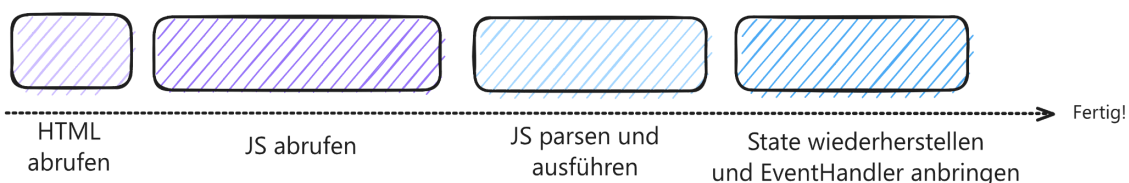


Abbildung 4: Einzelne Schritte der Hydration beim Laden der Seite

4. Das Framework Qwik.js

Im Jahr 2021 versuchen die Verantwortlichen hinter dem CMS-Tool Builder.io unter der Leitung von Misko Hevery, früher Teil des Teams von Angular und einer der Pioniere der modernen Webentwicklung, dieses Problem zu lösen und veröffentlichen das neue JavaScript-Framework Qwik in einer ersten Version. [4]

Ein übergeordnetes Ziel von Qwik ist es, performante und direkt interaktive Anwendungen zu liefern, unabhängig von deren Größe oder Komplexität und ohne dass sich der Entwickler dabei darüber Gedanken machen muss. Laut CTO Hevery liegt die eigentliche Stärke des Frameworks jedoch nicht in seiner Geschwindigkeit selbst, sondern darin, "dass es gekonnt jegliche Arbeit vermeidet". Möglich wird dies dadurch, dass Qwik nicht wie viele andere SSR-Frameworks Hydration nutzt, sondern auf das neue Konzept der Resumability setzt. [10]

4.1 Was versteht man unter Resumability?

Der Begriff Resumability lässt sich am besten mit "Wiederaufnahmefähigkeit" übersetzen und beschreibt im Kontext von Qwik, die Fähigkeit die Ausführung der Anwendung nach Übertragung vom Server auf den Client, dort wieder aufzunehmen und fortzusetzen, ohne dass wie bei der Hydration die gesamte Anwendungslogik erneut heruntergeladen und ausgeführt werden muss. Durch diesen Ansatz sind Qwik-Applikationen nach der Übertragung des HTML und einer 1kB kleinen JavaScript-Funktion, dem QwikLoader, fertig geladen. [11]

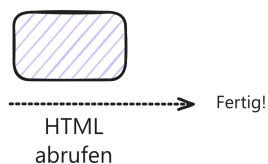


Abbildung 5: Prozess der Resumability beim Laden der Seite

Um die Anwendung interaktiv zu machen, müssen auch hier wieder, wie bei allen serverseitig gerenderten Anwendungen, die Event Listener mit den Elementen des DOM verknüpft werden. Während dies bei anderen Frameworks typischerweise durch die in 3.1 beschriebenen Schritte der Hydration erreicht wird, löst Qwik diese Aufgabe, indem alle dafür benötigten Informationen beim SSR-Prozess gesammelt und im HTML als sogenannte QLRs (Qwik URL) serialisiert werden (siehe Abbildung 6). [11]

```
<button on:click="./chunk.js#handler_symbol">click me</button>
```

Abbildung 6: Serialisierung der Event Handler in das HTML

Diese lassen sich als spezielle Form von URLs beschreiben, die aus einem Pfad zur Ressource und einem Symbolnamen hinter der Raute bestehen, die zusätzliche Informationen zum State oder Referenzfunktionen beinhalten. Der QwikLoader dient nun dabei als globaler Event Listener für alle Browser Events. Tritt ein Event durch eine

Nutzerinteraktion ein, durchsucht der QwikLoader das DOM für das entsprechende Attribut mit der zugehörigen QRL, sodass der einzelne Chunk geladen und ausgeführt werden kann. Auf diese Weise weiß die serverseitig gerenderte Qwik-Applikation auch im Client, wo Events registriert sind und welcher Code beim Eintreten ausgeführt werden muss. Während bei der Hydration der vollständige Komponenten-Baum der Anwendung ab der Wurzel geladen werden muss, genügt bei Qwik zu Beginn durch die Serialisation in das HTML nur der 1kB große QwikLoader wodurch ein redundantes Ausführen der kompletten Anwendung im Client nicht mehr nötig ist. [12]

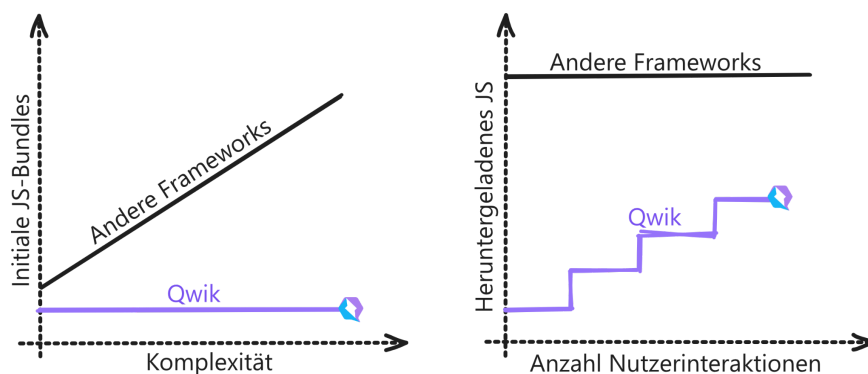


Abbildung 7: Vergleich der initialen Bundle-Größe und dem heruntergeladenen JavaScript zwischen Qwik und herkömmlichen Meta-Frameworks

Bei kleineren, serverseitig gerenderten Webanwendungen stellt die anschließende, zusätzliche Hydration normalerweise kein Problem dar und die Ladeperformance wird nur marginal und für den User nicht merklich beeinflusst. Anhand der oberen Abbildung links lässt sich erkennen, dass bei den herkömmlichen Meta-Frameworks mit zunehmender Komplexität und mehr Komponenten jedoch immer mehr JavaScript-Code heruntergeladen werden muss, um diese zu hydrieren. Bei Qwik-Anwendungen hingegen steigt die initiale JavaScript-Bundlegröße nicht proportional zur Größe und Komplexität der Anwendung, sondern bleibt konstant, wodurch die Performance sehr gut skalierbar ist. [13]

Durch das Prinzip der Resumability lässt sich in Qwik wie auf der rechten Seite der Abbildung Lazy-Loading umsetzen, bei dem nach und nach bei jeder Interaktion des Nutzers mit einer Komponente das entsprechende JavaScript heruntergeladen und ausgeführt wird. So muss auch nur der wirklich benötigte Code abgerufen werden, während bei herkömmlichen Frameworks der aller Komponenten benötigt wird, ganz egal ob der User später mit diesen interagiert oder nicht. [13]

4.2 Syntax in Qwik

Betrachtet man ein Stück Code einer Qwik-Anwendung, so wird dies zumindest Entwicklern mit Erfahrung in React bekannt vorkommen, da Qwik ebenfalls auf funktionale Komponenten und JSX setzt. Auf den zweiten Blick fällt jedoch ein `$`-Suffix an bestimmten Stellen der Syntax auf. Dies ist zum einen für den Entwickler als auch den QwikOptimizer ein Zeichen, dass an dieser Stelle die Applikation in kleinere Chunks aufgesplittet wird, die dann später nach dem Prinzip der Resumability "lazy-loadable" sind. [14] Der Optimizer übersetzt beim Build den Code so, dass die Referenzen und States für die Closures wiederhergestellt sind und so die Anwendung im Client direkt fortgesetzt werden kann. Beim Build-Prozess der

Anwendung wird dann das `$`-Suffix als Indiaktor erkannt und die Funktion als sogenanntes Symbol auf oberster Ebene unter einer eigenen QLR (*siehe oben*) extrahiert wodurch automatisch ein sehr fein-granulares Code-Splitting erreicht wird, ohne dass sich der Entwickler selbst darum kümmern muss. [14]

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

export default component$(() => {

  const count = useSignal(0)

  return(
    <button onClick$={() => count.value++}>Actual count: {count.value}</button>
  )
});
```

Abbildung 8: Komponenten-Syntax in Qwik mit EventListener und lokalem State

In diesem Beispiel erkennt man, dass das `$`-Zeichen hinter der Komponente und dem `onClick`-Event steht. Der Optimizer kann nun daraus zwei separate Qwik-Symbole generieren, die nun auf User-Request geladen werden.

5. Performance-Vergleich anhand einer prototypischen Anwendung

Die Performance des Frameworks und ob das oben beschriebene Konzept der Resumability auch in der Praxis einen wirklichen Geschwindigkeitsvorteil bietet, soll nun anhand einer prototypischen Testapplikation überprüft und mit einer “Hydration”-Anwendung verglichen werden.

5.1 Vorgehen und Aufbau der Anwendungen

Für den Performance-Vergleich wird dafür zum einen eine Qwik-Anwendung und zusätzlich eine weitere Applikation mit Next.js entwickelt, das als Meta-Framework von React die Seite serverseitig rendert und schließlich hydriert. Die beiden Anwendungen sollen für eine bestmögliche Gegenüberstellung nahezu identisch aufgebaut und implementiert werden. So bestehen sowohl die Qwik- als auch die Next.js- App aus einer einzigen Seite mit einer übergeordneten, vom Framework bereitgestellten Layout-Komponente, die einen simplen Header und Footer beinhaltet, sowie eine globale CSS-Datei einbindet. Der Hauptinhalt der Seite setzt sich aus verschiedenen einzelnen Qwik- bzw. React-Komponenten zusammen, die in einem separaten Ordner mit jeweils zugehörigen CSS-Files für das Design liegen und dabei verschiedene, typische Aspekte einer modernen Webanwendung abdecken sollen.

- Header mit SVG-Logo und externen Link
- Landing-Komponente mit Text und großem Bild als WebP-Datei
- Interaktiver Counter mit Live-Anzeige und lokalem State
- Interaktiver Card-Slider mit wechselndem Text
- Liste mit Child-Components um serverseitig abgerufene JSON-Placeholderdaten von einer API darzustellen
- Footer mit clientseitig gerendeter, aktueller Uhrzeit

Für die anschließende Überprüfung werden nun noch beide Anwendungen über den Cloud-Hosting-Anbieter Vercel deployed und sind unter den URLs <https://qwik-app-one.vercel.app/> bzw. <https://next-app-psi-one.vercel.app/> öffentlich aufrufbar.

5.2 Vergleichender Test

Nach dem Deployment der Testapplikationen werden diese mit dem Tool "WebPageTest" von Catchpoint hinsichtlich ihrer Ladeperformance überprüft. Für jede Seite werden dabei zwei Testläufe unter verschiedenen Bedingungen durchgeführt.

Bei dem ersten Durchgang wird im Tool eine Konfiguration gewählt, bei der die Seiten von einem Desktoprechner mit Firefox-Browser in Frankfurt mit einer 5 Mbps LAN-Verbindung und einer Latenz von 28ms aufgerufen werden, sodass dieser Test unter nahezu optimalen Bedingungen durchgeführt wird. Im zweiten Teil soll nun außerdem überprüft werden, wie die beiden Anwendungen unter schlechteren Voraussetzungen abschneiden. Dabei wird der WebPageTest von einem Motorola-Mittelklasse-Smartphone mit Chrome-Browser in Indien mit einer 1,6 Mbps 3G-Verbindung und einer höheren Latenz von 300ms simuliert.

Für die Evaluierung der Ladeperformance der Seiten werden anschließend folgende Metriken betrachtet:

- Time to first Byte (*TTFB*): Zeitraum zwischen Aufruf der Seite und dem ersten, vom Webserver geladenen Byte
- First Contentful Paint (*FCP*): Zeitspanne bis das erste für den User relevante Element gerendert ist
- Speed Index: Gibt an, wie schnell Inhalte beim Seitenaufbau visuell dargestellt werden. Es wird dabei ein Video des Seitenaufbaus im Browser aufgezeichnet und der visuelle Fortschritt zwischen den Frames berechnet
- Total Time (*TT*): Zeit bis das Dokument und alle Inhalte vollständig geladen und gerendert sind
- Page Weight: Gesamtanzahl der heruntergeladenen Bytes
- Total Requests: Anzahl der an den Server für das Laden der Seite gesendeten Requests

Frankfurt, Desktop, Firefox, 5 Mbps LAN						
	TTFB	FCP	Speed Index	Page Weight	TT	Total Requests
Qwik	0,181 s	0,304 s	0,373 s	38 kb	0,428 s	4
Next	0,181 s	0,407 s	0,474 s	149 kb	0,647 s	13

Betrachtet man die in der oberen Tabelle gegenübergestellten Werte des ersten Tests beider Anwendungen, so fällt auf, dass die Serverreaktionszeit bis zum ersten Byte bei beiden Seiten sehr schnell ist und in einem exzellenten Bereich liegt. Leichte Unterschiede hingegen lassen sich beim First Contentful Paint und dem Speed Index feststellen. Hier liegen die Werte der Qwik-Anwendung ungefähr 0,1s vor der mit Next erstellten Seite.

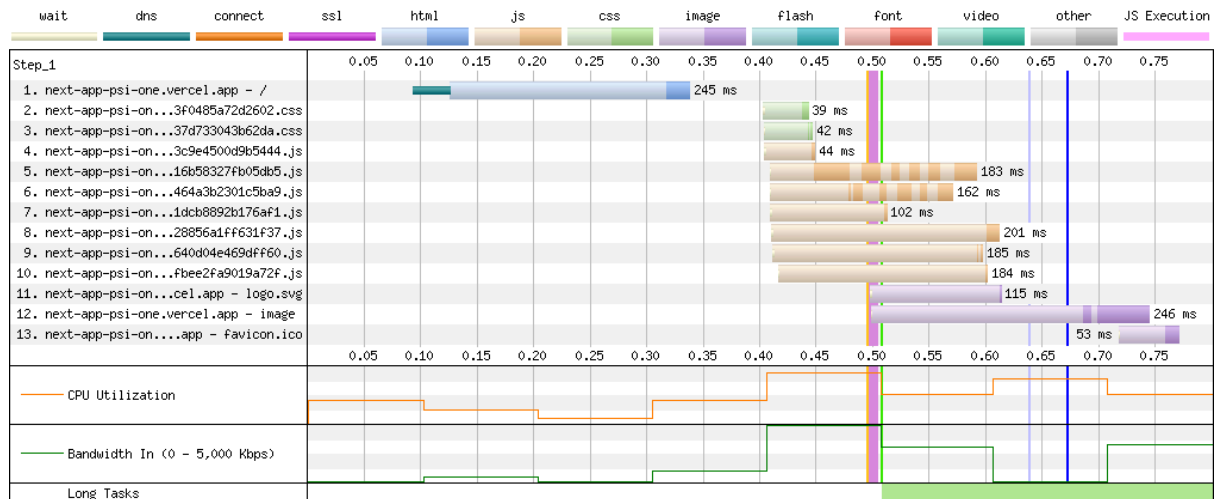


Abbildung 9: Wasserfalldiagramm des Ladevorgangs der Next.js-Anwendung

Auffällig ist die Differenz des gesamten initialen Page Weights und der Anzahl der gesendeten Requests. Diese werden, wie sich im Wasserfalldiagramm erkennen lässt, bei der Next-Applikation dafür benötigt, die zusätzlichen JavaScript-Bundles der Komponenten für die Hydration herunterzuladen. Bei Qwik hingegen werden neben dem initialen HTML nur noch das SVG-Logo, das WebP-Bild und eine Manifest.json-Datei angefragt, wodurch hier insgesamt weniger Kilobytes abgerufen werden müssen und so auch die Gesamtzeit zum vollständigen Laden mit 0,438 s leicht schneller ist als die 0,674 s der Next.js-Seite. Da zusätzlich beim Build-Prozess das CSS der Seite automatisch "Inline" in das HTML geschrieben wird, können noch einmal zusätzliche Requests gespart werden.

Zwar lassen sich anhand dieser Zahlen minimale Geschwindigkeitsvorteile der Qwik-Applikation ableiten, für einen User in der Praxis wird dies jedoch höchstwahrscheinlich kaum einen merklichen Unterschied machen, da auch die Next-Anwendung in diesem Testfall sehr gute Performance-Werte und eine optimale User Experience aufweist.

Mumbai, Smartphone, Chrome, 1,6 Mbps 3G						
	TTFB	FCP	Speed Index	Page Weight	TT	Total Requests
Qwik	2,492 s	2,720 s	2,836 s	38 kb	3,972 s	4
Next	2,795 s	3,478s	3,919 s	149 kb	5,765 s	13

Deutlicher hingegen sind die Geschwindigkeitsdifferenzen der beiden Testapplikationen im zweiten Durchlauf unter weniger guten Netzwerkbedingungen. Aufgrund der schlechteren Verbindung mit langsamer Datenrate, kostet hier jeder zusätzliche Request sowie das Herunterladen weiterer Dateien mehr Zeit, was sich in deutlicheren Unterschieden bei den Werten für das First Contentful Paint (2,720s zu 3,478s) und dem Speed Index (2,836s zu 3,919s) niederschlägt. Bis die Anwendung hier vollständig geladen ist, dauert es so bei Qwik knapp 1,8s kürzer, was nun auch durchaus in der Praxis für den User einen deutlichen Unterschied darstellt.

6. Fazit

Qwik verfolgt mit seinem neuen Ansatz der Resumability das Ziel, performante und interaktive Anwendungen unabhängig von ihrer Größe oder Komplexität zu liefern. Im Gegensatz zur Hydration, bei der die gesamte Anwendungslogik erneut heruntergeladen und ausgeführt werden muss, ermöglicht Resumability durch Serialisierung der Event Listener in das HTML die Wiederaufnahme und Fortsetzung der Anwendung nach der Übertragung vom Server auf den Client. Somit ruft Qwik durch das Lazy-Loading des JavaScripts nur den wirklich benötigten Code ab, während andere SSR-Frameworks den Code aller Komponenten laden müssen, unabhängig davon, ob der Nutzer später mit ihnen interagiert oder nicht.

Auch im vergleichenden Test zweier prototypischer Anwendungen konnte so eine verbesserte Performance bei Qwik im Gegensatz zu einer gleich aufgebauten Applikation auf Next.js-Codebasis gemessen werden. Da die beiden Seiten recht simpel aufgebaut sind und nur wenige interaktive Komponenten enthalten, scheint hier der Geschwindigkeitsvorteil unter guten Netzwerkbedingungen für einen tatsächlichen User vernachlässigbar.

Deutlich relevanter wird dies bei der Übertragung eines solchen Szenarios auf mögliche Praxisanwendungen in der Realität, die häufig noch weit komplexer sind und viele verschiedene Komponenten beinhalten. Während hier der abgerufene JavaScript-Anteil proportional ansteigt, bleibt die Größe des initialen JavaScripts bei Qwik auch mit zunehmender Komplexität konstant. Gerade unter einer gegebenenfalls eingeschränkten Internetverbindung auf mobilen Geräten führt diese reduzierte Größe der Qwik-App zu sichtbar schnelleren Ladezeiten und einer besseren User Experience.

7. Ausblick

Mittlerweile scheinen sich immer mehr Entwicklerteams hinter aktuellen Frameworks dem Problem des kontinuierlichen Anstiegs von JavaScript und dessen Auswirkung auf die Ladeperformance bewusst zu sein. Es wird an neuen Lösungswegen und Möglichkeiten gearbeitet, um auf der einen Seite weiterhin Features wie Server Side Rendering und Interaktivität zu bieten und gleichzeitig vermehrt nur statisches HTML an den Client zu senden und die Auslieferung von JavaScript auf ein Minimum zu beschränken.

Mit der Reduktion von JavaScript gegen Null sprechen mittlerweile einige Entwickler wie Juho Vepsäläinen und Arto Hellas von "Disappearing Frameworks". Während manche dieser Frameworks wie Astro oder Marko dabei versuchen, entweder nur einzelne Teile der Seite oder die Anwendung nach und nach progressiv zu hydrieren und Next.js in seiner neuesten Version mit sogenannten Server Components diesen aufwändigen Prozess effizienter gestalten möchte, setzt Qwik hingegen mit der Resumability dafür auf ein gänzlich neues Konzept. [15] Man darf durchaus gespannt sein, wohin der Weg der JavaScript-Frameworks in einer mittlerweile so schnelllebigen Welt des Web Developments gehen wird und mit welchen Verbesserungen oder neuen Ansätzen in Zukunft versucht wird eine optimale Suchmaschinenoptimierung, Ladeperformance und Developer Experience zu vereinen.

Literaturverzeichnis

- [1] "Why does speed matter". Web.dev. [Online] <https://web.dev/learn/performance/why-speed-matters> (abgerufen am 12.12.2023)
- [2] "How Rakuten 24's investment in Core Web Vitals increased revenue per visitor by 53.37% and conversion rate by 33.13%". Web.dev. [Online] <https://web.dev/case-studies/rakuten> (abgerufen am 12.12.23)
- [3] WeAreDevelopers (15.06.2022). "WWC22 - Qwik + Partytown: How to remove 99% of JavaScript from main thread". YouTube [Onlinevideo] <https://www.youtube.com/watch?v=0dC11DMR3fU> (abgerufen am 14.12.2023)
- [4] X.A. Cao (2023). "Headless CMS and Qwik Framework and their practicalities in the future of application development". Vaasa University of Applied Sciences [Thesis] https://www.theseus.fi/bitstream/handle/10024/795367/Cao_Xuan-An.pdf (abgerufen am 16.12.2023)
- [5] "Introduction to client-side frameworks". MDN Webdocs. [Online] https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/Introduction (abgerufen am 16.12.2023)
- [6] M. Thakkar. *Building React Apps with Server-Side Rendering: Use React, Redux, and Next to Build Full Server-Side Rendering Applications*. Vadodara, Indien: Apress, 2020
- [7] M. Riva. *Real-World Next.js: Build scalable, highperformance, and modern web applications using Next.js, the React framework for production*. Birmingham, England: Packt Publishing, 2022
- [8] R. Carniato (3.02.2022). "Why Efficient Hydration in JavaScript Frameworks is so Challenging". Dev.to. [Online] <https://dev.to/this-is-learning/why-efficient-hydration-in-javascript-frameworks-is-so-challenging-1ca3> (abgerufen am 16.12.2023)
- [9] M. Hevery (21.04.2022). "Hydration is Pure Overhead". Builder.io [Online] <https://www.builder.io/blog/hydration-is-pure-overhead> (abgerufen am 20.12.2023)
- [10] M. Hevery (18.10.2022). "Die Magie von Qwik liegt nicht in seiner Geschwindigkeit, sondern darin, dass es gekonnt jegliche Arbeit vermeidet." Entwickler.de [Online] <https://entwickler.de/javascript/qwik-hydration-resumability> (abgerufen am 20.12.2023)
- [11] "Resumable vs. Hydration". Qwik Docs [Online] <https://qwik.builder.io/docs/concepts/resumable> (abgerufen am 20.12.2023)
- [12] "Qwikloader". Qwik Docs [Online] <https://qwik.builder.io/docs/advanced/qwikloader/> (abgerufen am 23.12.2023)
- [13] M. Hevery (17.05.2022). "Our Current Frameworks are $O(n)$; We Need $O(1)$ ". Builder.io [Online] <https://www.builder.io/blog/our-current-frameworks-are-on-we-need-o1> (abgerufen am 23.12.2023)
- [14] "The dollar \$ sign". Qwik Docs [Online] <https://qwik.builder.io/docs/advanced/dollar> (abgerufen am 28.12.2023)
- [15] J. Vepsalainen, A. Hellas % P. Vuorimaa (2023). "The State of Disappearing Frameworks in 2023". Aalto University [Paper] <https://arxiv.org/pdf/2309.04188.pdf> (abgerufen am 07.01.2024)