Technische Hochschule Nürnberg

Fakultät Elektrotechnik Feinwerktechnik Informationstechnik

Studiengang Media Engineering

Bachelorarbeit von

Stefan M o h r

Matrikelnummer 358 1391

Analyse von Page Speed beeinflussenden Technologien in Frontend JavaScript Frameworks

SoSe 2024

Abgabedatum: 20. Juli 2024

Erstgutachter: Prof. Dr. Matthias Hopf

Zweitgutachter: Prof. Dr. Oliver Hofmann

Technische Hochschule Nürnberg

Hinweis: Diese Erklärung ist in alle Exemplare der Abschlussarbeit fest einzubinden. (Keine Spiralbindung)

Prüfungsrechtliche Erklärung der/des Studierenden

Angaben des bzw. der Studierenden:

Name: Mohr Vorname: Stefan Matrikel-Nr.: 3581391

Fakultät: Elektrotechnik Feinwerktechnik Informationstechnik Studiengang: Media Engineering

Semester: SoSe 2024

Titel der Abschlussarbeit:

Analyse von Page Speed beeinflussenden Technologien in Frontend JavaScript Frameworks

Ich versichere, dass ich die Arbeit selbständig verfasst, nicht anderweitig für Prüfungszwecke vorgelegt, alle benutzten Quellen und Hilfsmittel angegeben sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Nürnberg, 20.07.2024 Stefan Mohr

Ort, Datum, Unterschrift Studierende/Studierender

Erklärung der/des Studierenden zur Veröffentlichung der vorstehend bezeichneten Abschlussarbeit

Die Entscheidung über die vollständige oder auszugsweise Veröffentlichung der Abschlussarbeit liegt grundsätzlich erst einmal allein in der Zuständigkeit der/des studentischen Verfasserin/Verfassers. Nach dem Urheberrechtsgesetz (UrhG) erwirbt die Verfasserin/der Verfasser einer Abschlussarbeit mit Anfertigung ihrer/seiner Arbeit das alleinige Urheberrecht und grundsätzlich auch die hieraus resultierenden Nutzungsrechte wie z.B. Erstveröffentlichung (§ 12 UrhG), Verbreitung (§ 17 UrhG), Vervielfältigung (§ 16 UrhG), Online-Nutzung usw., also alle Rechte, die die nicht-kommerzielle oder kommerzielle Verwertung betreffen.

Die Hochschule und deren Beschäftigte werden Abschlussarbeiten oder Teile davon nicht ohne Zustimmung der/des studentischen Verfasserin/Verfassers veröffentlichen, insbesondere nicht öffentlich zugänglich in die Bibliothek der Hochschule einstellen.

Hiermit ☑ genehmige ich, wenn und soweit keine entgegenstehenden Vereinbarungen mit Dritten getroffen worden sind,

☐ genehmige ich nicht,

dass die oben genannte Abschlussarbeit durch die Technische Hochschule Nürnberg Georg Simon Ohm, ggf. nach Ablauf einer mittels eines auf der Abschlussarbeit aufgebrachten Sperrvermerks kenntlich gemachten Sperrfrist

von 0 Jahren (0 - 5 Jahren ab Datum der Abgabe der Arbeit),

der Öffentlichkeit zugänglich gemacht wird. Im Falle der Genehmigung erfolgt diese unwiderruflich; hierzu wird der Abschlussarbeit ein Exemplar im digitalisierten PDF-Format an die Betreuer übermittelt. Bestimmungen der jeweils geltenden Studien- und Prüfungsordnung über Art und Umfang der im Rahmen der Arbeit abzugebenden Exemplare und Materialien werden hierdurch nicht berührt.

Nürnberg, 20.07.2024 Stefan Mohr

Ort, Datum, Unterschrift Studierende/Studierender

Datenschutz: Die Antragstellung ist regelmäßig mit der Speicherung und Verarbeitung der von Ihnen mitgeteilten Daten durch die Technische Hochschule Nürnberg Georg Simon Ohm verbunden. Weitere Informationen zum Umgang der Technischen Hochschule Nürnberg mit Ihren personenbezogenen Daten sind unter nachfolgendem Link abrufbar: https://www.th-nuernberg.de/datenschutz/

Kurzfassung

In dieser Arbeit werden in Frontend JavaScript Frameworks genutzte Technologien untersucht, die einen maßgeblichen Einfluss auf die Geschwindigkeit von Websites nehmen. Als Bewertungsgrundlage für die Website-Geschwindigkeit werden die Kategorien Ladevorgang, Interaktivität und visuelle Stabilität verwendet, die mittels der Core-Web-Vitals-Metriken gemessen werden. Für die Auswahl der betrachteten Bibliotheken und Frameworks React, Solid.js und Qwik wurde deren Verbreitung sowie ihr Einfluss auf spezielle Technologien herangezogen. Mit den Frameworks wurden je zwei Websites erstellt – eine minimalistische Website und eine Website, die einem realen Anwendungsfall für interaktive Webanwendungen entspricht. Anschließend wurden die Core-Web-Vitals-Werte der Websites mit Lighthouse, einem Bewertungsprogramm für Web Vitals, gemessen. Die Auswertung der Messungen ergibt, dass Technologien wie Server-Side Rendering einen positiven Einfluss auf die Ladegeschwindigkeit von Websites haben und diese Technologien durch neue Konzepte wie Resumability von Qwik weiter verbessert werden können. Auch für die Interaktivität von Websites zeigten neuere Technologien wie die feinkörnige Reaktivität von Solid.js einen Vorteil gegenüber der Verwendung eines virtuellen DOMs.

Abstract

This thesis analyses technologies used in frontend JavaScript frameworks that influence the speed of websites. As a basis for evaluating the website speed, the categories loading performance, interaction and visual stability were used, which are measured using the Core Web Vitals metrics. The analysed libraries and frameworks – React, Solid.js and Qwik – were selected based on their prevalence in usage and their influence on specific technologies. Two websites were created with each of the frameworks – a minimalist website and a website that corresponds to a real use case for interactive web applications. The Core Web Vitals metrics of the websites were then measured using Lighthouse, an evaluation tool for Web Vitals. The evaluation of the measurements shows that technologies such as server-side rendering have a positive influence on the loading speed of websites and that these technologies can be further improved by new concepts such as resumability from Qwik. For the interactivity of websites, newer technologies such as the fine-grained reactivity of Solid.js also showed an advantage over the use of a virtual DOM.

Inhaltsverzeichnis

1 Einleitung

Die Geschwindigkeit von Websites spielt eine bedeutsame Rolle für die Nutzererfahrung sowie die Platzierung in Suchmaschinenergebnissen und somit auch für den Erfolg von Websites [1]. Dabei ist nicht nur das initiale Laden der Seite ein wichtiger Aspekt, sondern auch die Wartezeiten während der Nutzung bei Interaktionen mit der Seite. Für Websitenutzer stellt eine längere Wartezeit als die erwartete eine störende Unterbrechung der noch nicht abgeschlossenen Interaktion dar [2]. Während einige Einflussfaktoren für die Ladezeit hardwareabhängig sind, können andere durch die Programmierung der Webseite beeinflusst werden. Google entwickelte den sogenannten Page Speed mit den Core Web Vitals als Maßzahlen für die Ladezeit und Nutzerfreundlichkeit von Webseiten. Tatsächlich konnte Google feststellen, dass die Wahrscheinlichkeit, eine Seite vor dem fertigen Laden zu verlassen, um 24 Prozent geringer ist, wenn die Website gut bewertete Core-Web-Vitals-Metriken vorzeigen kann [3]. Das Beachten von geringen Ladezeiten und guten Core Web Vitals ist für Entwickler also von hoher Relevanz.

Im Bereich der Frontend-Webentwicklung greifen Programmierer häufig auf JavaScript-Frameworks zurück. In den letzten Jahren kam eine Vielzahl neuer Frameworks für die Webentwicklung mit unterschiedlichen Technologien auf. Nach der Umfrage für das Jahr 2023 von „State of JS“ sind von den elf meistgenutzten JavaScript-Frameworks sechs Frameworks von 2020 oder jünger [4]. Nun stellt sich für Entwickler die Frage, welches dieser Frameworks sie wählen sollten. Unter den neu entwickelten Frameworks basieren einige auf innovativen Technologien und Ansätzen, die eine deutliche Verbesserung der Geschwindigkeit von Websites versprechen.
Diese Arbeit soll nun prüfen, welche Unterschiede hinsichtlich der Website-Geschwindigkeit tatsächlich vorzufinden sind. Ganz konkret wird untersucht:

Welche quantitativen Unterschiede existieren zwischen verschiedenen Frontend JavaScript-Frameworks hinsichtlich ihrer Auswirkungen auf die Erfüllung der Core Web Vitals (Largest Contentful Paint, Interaction to Next Paint, Cumulative Layout Shift) und welche Rolle spielen dabei die von den Frameworks verwendeten Technologien und Features?

Nach einer Einführung in die Technologien, werden die drei Frontend JavaScript-Bibliotheken bzw. -Frameworks React, Solid.js und Qwik mit zugehörigen Metaframeworks miteinander verglichen. Hierzu wurden jeweils zwei Webapplikationen erstellt, anhand derer die Performance der unterschiedlichen Frameworks in mehreren realistischen Szenarien gemessen werden kann. Als Messwerte wurden die Core Web Vitals herangezogen, deren Ergebnisse gegenübergestellt und die Auswirkungen der verschiedenen Technologien diskutiert.

2 Page Speed und relevante Technologien

Page Speed beschreibt in dieser Arbeit die Geschwindigkeit von Websites und Webanwendungen, die mittels einiger von Google entwickelten Metriken bewertet werden kann. Darunter zählen die Geschwindigkeit beim Aufruf und Laden einer Website sowie die Geschwindigkeit während der Nutzung. Die Bezeichnung Page Speed verwendet Google bei seinem Analyseprogramm für die Geschwindigkeit von Websites, PageSpeed Insights. Die ursprüngliche Version von PageSpeed Insights wurde 2009 unter dem Namen Page Speed für die Öffentlichkeit zugänglich gemacht und brachte die Möglichkeit Verbesserungen zur Geschwindigkeit von Websites zu ermitteln [5]. In diesem Kapitel werden die Metriken für den Page Speed erläutert, Möglichkeiten zur Messung der Metriken gezeigt und Technologien in der Webentwicklung mit Auswirkungen auf den Page Speed vorgestellt.

2.1 Webanwendungen

Das Internet hat sich seit dem Aufkommen von Websites Anfang der 1990er-Jahre von einer globalen Plattform für den Austausch von Informationen zur größten Plattform für Anwendungen entwickelt [6]. Während Websites früher statisch waren und Änderungen oft mit einem kompletten Neuladen der Seite einhergingen, setzt heute das Nutzen von Webanwendungen eine Vielzahl von Nutzereingaben und ein dynamisches Anzeigen von Inhalten voraus. Damit eine Website interaktiv nutzbar wird, wird JavaScript benötigt, das während der Laufzeit Elemente auf der Website manipulieren kann.
Auf der Plattform HTTPArchive – die seit 2010 den Aufbau einer stetig steigenden Anzahl von Websites überwacht – lässt sich der Anstieg im Nutzen von JavaScript und damit eine Zunahme des Datenvolumens von Websites deutlich erkennen. Die Menge an verwendetem JavaScript stieg seit November 2010 bis Juni 2024 im Median um 622 % auf 639,7 kB für Desktop-Websites, die Menge für Mobile-Websites liegt mit aktuell 583,2 kB darunter. In der gleichen Zeitspanne stieg die Menge an HTML im Median um 68,5 % auf 33,2 kB für Desktop-Websites und auf 31,9 kB für Mobile-Websites. [7]

Um auch Websites mit höherem Datenvolumen schnell anzeigen zu können und die Nutzerinteraktion flüssig zu halten, wurde über die Jahre eine Vielzahl von Konzepten und Technologien entwickelt, die sowohl für Entwickler als auch Nutzer von Websites Vorteile bringen.

2.1.1 Multi- und Single Page Applikationen

Webanwendungen lassen sich in sogenannte Multi Page Applications (MPA) und Single Page Applications (SPA) unterscheiden. Eine MPA folgt dem traditionellen Ansatz von Websites, bei Änderungen im Browser die Seite neu vom Server anzufordern. Es werden also während der Nutzung der Webanwendung mehrere HTML-Dokumente benötigt. Dafür wird die Anwendungslogik am Server ausgeführt und nach Interaktionen eine neue Ansicht der Website erstellt, die dann als Antwort zum Client geschickt wird. Mit der Einführung von AJAX wurde es möglich spezifische Teile einer Seite zu verändern, was die Benutzererfahrung verbesserte, da nach einer Interaktion keine Verzögerung durch ein Neuladen der Seite erfolgte. Bei einer Navigation wird dennoch ein neues HTML-Dokument an den Client geliefert. [8]

Eine SPA besteht aus nur einem HTML-Dokument. Der benötigte Anwendungscode wird beim initialen Seitenaufruf entweder komplett mitgeliefert oder während der Nutzung der Seite nachgeladen. JavaScript erstellt den gesamten Inhalt der Anwendung und verarbeitet Zustandsänderungen dynamisch im Browser. Benötigte Daten werden über den Server mittels AJAX oder der Fetch API abgerufen. Eine Navigation findet dadurch ohne ein erneutes Laden der Seite statt, was zu flüssigen Übergängen führt. [9]

2.1.2 Rendering Pfad

Unabhängig von der Art der Webanwendung müssen im Browser einige Schritte durchlaufen werden, damit die Anwendung für Nutzer sichtbar wird. Erhält ein Browser das angeforderte HTML-Dokument von einer aufgerufenen Seite, wird das HTML-Dokument geparst und die darin angegebenen Ressourcen werden angefordert. Aus dem HTML wird das DOM erstellt, aus dem CSS das CSSOM – das CSS-Pendant zu HTML und DOM. JavaScript kann DOM sowie CSSOM verändern und auslesen, weshalb die Baumstruktur für das Rendern einer Seite (Render Tree) erst erstellt werden kann, wenn alle JavaScript Änderung an DOM und CSSOM ausgeführt wurden. Der Vorgang

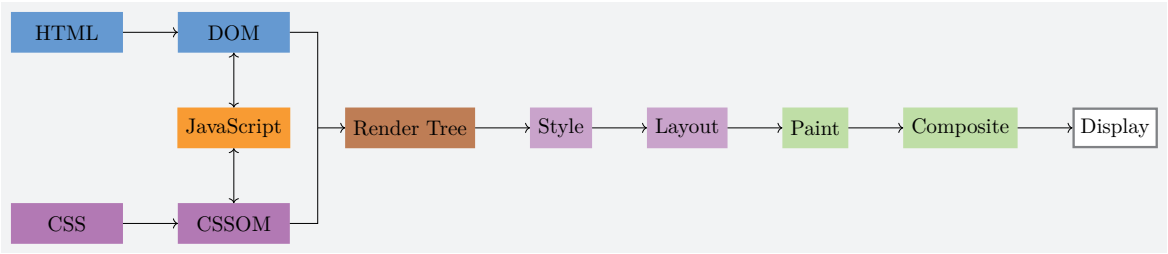

Abbildung 2.1: Rendering-Pfad von Websites im Browser (nach [10])

des Parsens des HTMLs wird durch JavaScript blockiert, das sich im HTML-`<head>` befindet und nicht als `async`, `defer` oder als `type="module"` markiert ist.

CSS blockiert das Parsen und Rendern der Seite bis das Laden und Verarbeiten des CSS vollständig abgeschlossen ist, um einen sogenannten Flash of Unstyled Content (FOUC) zu vermeiden – also ein vorzeitiges Anzeigen des Seiteninhalts ohne CSS Stile – da dies zu einer schlechten Nutzererfahrung führt [11]. Nach dem Erstellen des Render Trees werden wie in Abbildung 2.1 gezeigt, die weiteren Render-Schritte Style, Layout, Paint und Composite durchlaufen, bis die Website das erste Mal im Browser sichtbar ist. Das erste Durchlaufen des Rendering-Pfads bei einem Seitenaufruf wird als kritischer Rendering-Pfad bezeichnet, da hier die Dauer des Durchlaufs – auch aufgrund der hohen Menge an Daten für den ersten Seitenaufbau – für Websitenutzer besonders deutlich wird. Finden während der Nutzung einer Webanwendung visuelle Änderungen statt, wird der Pfad erneut durchlaufen, um die Nutzeroberfläche zu aktualisieren. Allerdings müssen dabei nicht mehr das HTML-Dokument und CSS erneut komplett verarbeitet werden, weshalb die Dauer zum Erstellen des Render Trees kürzer ausfällt.

Hier werden auch Unterschiede zwischen einer MPA zu einer SPA deutlich. Wird auf einer MPA navigiert, wird der kritische Rendering-Pfad bei jeder Navigation erneut durchlaufen, während bei einer SPA nur der initiale Seitenaufruf betroffen ist. Dafür wird eine SPA mehrere nachträgliche Durchläufe des Rendering Pfads während der Nutzung der Website ausführen müssen. Der Rendering-Pfad hat also einen hohen Einfluss auf die Ladegeschwindigkeit sowie auf die Interaktivität einer Website.

2.2 Core Web Vitals als Page Speed Metriken

Um eine einheitliche Bestimmung des Page Speeds zu gewährleisten, hat Google im Jahr 2020 die Web Vitals eingeführt. Web Vitals sind eine Sammlung von Metriken, die die Benutzererfahrung auf einer Website hinsichtlich des Seitenaufrufs, der Interaktivität und

der visuellen Stabilität messen [12]. Bei den Web Vitals wird zwischen Core Web Vitals und weiteren Web Vitals unterschieden. Die Core Web Vitals sind die relevantesten Web Vitals und bestehen aus den Messwerten Largest Contentful Paint (LCP), Interaction to Next Paint (INP) und Cumulative Layout Shift (CLS). Weitere Web Vitals sind beispielsweise First Contentful Paint (FCP), Speed Index (SI) und Total Blocking Time (TBT). In der Fachliteratur wird häufig der Wert von Time to Interactive (TTI) in Bezug auf die Ladegeschwindigkeit einer Website genannt. Dies ist ein Messwert, der angibt, ab wann eine Website tatsächlich nutzbar ist und nicht nur durch bereits gerendertes HTML nutzbar aussieht. Da der TTI durch bestimmte Netzwerkeigenschaften stark variieren kann, wurde er jedoch von Google als Messwert aus dem Messtool Lighthouse entfernt [13]. Die Core Web Vitals werden ständig von Google weiterentwickelt und angepasst. So wurde der INP im Mai 2022 als experimenteller Messwert eingeführt und ein Jahr lang getestet, bis er 2023 den Status eines ausstehenden Core-Web-Vitals-Messwerts erreichte und schließlich im März 2024 den First Input Delay als stabilen Core-Web-Vitals-Wert für die Benutzerinteraktion ablöste.

2.2.1 Largest Contentful Paint

Der Largest Contentful Paint (LCP) misst die Zeit bis zum abgeschlossenen Rendern des größten Elements mit Inhalt auf dem Bildschirm bei dem Ladevorgang einer Website. Dies spiegelt die wahrgenommene Ladegeschwindigkeit einer Website für die Benutzer wider und ist somit ein Messwert mit einer hohen Aussagekraft über die Nutzbarkeit der Seite. Die Messung des LCP beginnt bereits beim Entladevorgang der vorherigen Seite und beinhaltet dadurch auch die Verbindungseinrichtungsdauer, die Weiterleitungsdauer und die Time to First Byte (TTFB, ein Web-Vitals-Messwert, der die Servergeschwindigkeit bewertet). Zudem beeinflusst die Ladedauer der LCP-Ressource die Messung des LCPs erheblich. [14] Das Element, welches den LCP ausmacht, wird vom Browser anhand einiger Kriterien bestimmt. Der Elementtyp muss entweder ein `<img>`, `<image>` oder `<video>` sein, es muss ein Element mit einem über `url()` geladenen Hintergrundbild sein oder ein Element auf Blockebene, das Text oder Inline-Textelemente enthält. Da unter diesen Elementen jedoch auch Elemente sein können, die von den Nutzern nicht als inhaltliches Element eingestuft werden, werden für den LCP alle Elemente ausgeschlossen, die eine Deckkraft von null haben – demnach für Nutzer nicht sichtbar sind –, die den gesamten Hintergrund ausfüllen und Platzhalter sowie Bilder mit einer niedrigen Entropie, da dies darauf schließen lässt, dass es sich wahrscheinlich nicht um den eigentlichen Inhalt der Website handelt. Die Größe des LCP-Elements ist

die Größe, die im sichtbaren Bereich für Nutzer liegt. Durch CSS hinzugefügte Ränder, Abstände und Rahmen werden dabei nicht beachtet. Die LCP-Elemente entstehen in zeitlicher Abfolge nacheinander. Sie werden erst mit erfolgreichem Rendern als solche erkennbar und dann vom Browser als PerformanceEntry `largest-contentful-paint` zurückgegeben. Dies geschieht bis es zu einer Interaktion auf der Website kommt – hier zählt schon ein Scrollen als Interaktion, da sich dadurch die Wahrnehmung der Nutzer für die Ladezeit und Nutzbarkeit der Seite ändert. [15]

2.2.2 Interaction to Next Paint

Der Interaction to Next Paint (INP) misst die Reaktionsfähigkeit einer Website. Hierzu werden die Latenzen der Nutzerinteraktionen mit der Seite ermittelt. Die Latenz ist die Zeitspanne bis eine visuelle Rückmeldung auf eine Interaktion erfolgt – die Website also für den Benutzer sichtbar auf seine Interaktion reagiert. Über den gesamten Berechnungszeitraum – die Lebensdauer der Seite – wird die größte gemessene Latenz als INP-Wert herangezogen. Zwar wird nach 50 Interaktionen die größte Latenz herausgefiltert, um eine Verfälschung durch zufällige Fehler zu vermeiden, in der Realität werden 50 Interaktionen mit einer Seite jedoch selten erreicht. Dadurch kann ein Fehler während der Interaktion trotzdem einen großen Einfluss auf den INP-Wert haben. Für die endgültige Bewertung des Page Speeds fallen solche Abweichungen durch die Verwendung des 75. Perzentils für alle Seitenaufrufe nicht weiter ins Gewicht. [16] Bei der manuellen Ermittlung der Werte muss dies jedoch ebenso beachtet werden.

Die Interaktion der Benutzer mit einer Website kann auf verschiedene Arten stattfinden. Zu Interaktionen zählen neben den Klicks mit dem Cursor, Taps auf einem Touchscreen und Tastatureingaben auch `mouseovers` und Scrollen. Für den INP zählen jedoch nur die Interaktionen, die ein „click“-Event auslösen – also Mausklicks und Touchscreenberührungen – oder Tastatureingaben. [16]

2.2.3 Cumulative Layout Shift

Der Cumulative Layout Shift (CLS) bewertet die visuelle Stabilität einer Website und nutzt dafür Layoutverschiebungen. Finden Layoutverschiebungen statt, wird ein Session Window erstellt. Alle folgenden Layout-Shift-Scores innerhalb von fünf Sekunden oder bis eine Sekunde lang keine Layoutverschiebung in diesem Fenster erfolgt, stellen einen Burst dar. Der Wert des Bursts berechnet sich aus den kumulierten Einzelwerten der

Scores. Die Messung findet über die gesamte Lebenszeit der Seite statt, berücksichtigt wird jedoch nur der Wert des größten Bursts von Layout-Shift-Scores. [17]

Eine Layoutverschiebung wird als solche erkannt, wenn sich die Position (mit Punkt links-oben) eines bereits vorhandenen Elements auf der Seite zwischen zwei Frames ändert. Dieses Element wird als instabiles Element bezeichnet. Für die Berechnung des Layout-Shift-Scores werden zwei Werte verwendet – der Auswirkungsanteil (Impact Fraction) und der Entfernungsanteil (Distance Fraction). Die Multiplikation beider Werte ergibt den Layout-Shift-Score.
Impact Fraction ist der relative, sichtbare Anteil des instabilen Elements zum Viewport in den beiden betrachteten Frames.
Distance Fraction ist die relative Verschiebungsdistanz zum Viewport. Findet eine horizontale und vertikale Verschiebung des instabilen Elements statt, wird der größere Wert gewählt.
Bei mehreren instabilen Elementen zwischen zwei Frames wird das Element mit der höheren Distance Fraction zur Berechnung des Layout-Shift-Scores gewählt. Dadurch werden große Elemente – mit einer grundsätzlich hohen Impact Fraction –, die sich nur um kurze Distanzen verschieben, nicht mit einem schlechten Layout-Shift-Score bestraft. [17]

Layoutänderungen sind heutzutage – vor allem bei dynamischen Websites – oft vorzufinden und lassen grundsätzlich keinen Rückschluss auf die Nutzererfahrung zu. Da der CLS-Wert einen negativen Einfluss auf die Bewertung des Page Speeds nehmen kann, werden Layoutverschiebungen bis zu 500 ms nach einer Interaktion auf der Website nicht für die Page Speed Bewertung herangezogen [17]. Die Verschiebung nach einer Interaktion ist eine erwartete Layoutverschiebung, die nicht zu einer schlechteren Nutzererfahrung führen muss. Allerdings kann auch nach den 500 ms eine von der Interaktion initiierte Layoutverschiebung stattfinden – etwa asynchrones Laden von Suchanfragen – was am Ende doch zu einer schlechten Nutzererfahrung führt. Hier sollte darauf geachtet werden, direkt Platzhalter für die nachfolgenden Elemente für die Dauer des Ladevorgangs einzufügen [17].

2.2.4 Einordnung der Messwerte

Die Web-Vitals-Metriken geben einen genauen Messwert an, der zudem in einen Score umgerechnet wird. Der Score gibt Hinweise für Entwickler, welche Metriken verbesserungswürdig sind. Die Bewertung für die Benutzererfahrung wird in die Kategori-

en „Gut“, „Verbesserungsbedürftig“ und „Schlecht“ eingeteilt. Ein Abzug vom maximalen Score von 1,0 bedeutet nicht zwangsläufig, dass die Benutzererfahrung darunter leidet. Der LCP wird unter 2,5 Sekunden als gut gewertet und über 4,0 Sekunden als schlecht. Der INP wird unter 200 Millisekunden als gut gewertet und über 500 Millisekunden als schlecht. Der CLS wird unter dem Wert 0,1 als gut und über dem Wert 0,25 als schlecht gewertet. Zusätzlich verwendet Google bei der Bewertung das 75. Perzentil als Grenze, um eine Website für einen Wert als gut oder schlecht einzustufen. Erreichen mindestens 75 Prozent aller Website-Nutzer einen guten Wert bei einer Metrik, so stuft Google die gesamte Website in Bezug auf diese Metrik als gut ein. [18]

2.3 Messen von Page Speed

Zum Durchführen von Page-Speed-Messungen und zum Ermitteln der Core-Web-Vitals-Daten stellt Google einige Werkzeuge zur Verfügung. Jedes der Werkzeuge hat spezifische Einsatzzwecke, wodurch es möglich ist den Page Speed von der Entwicklungsphase bis zur kontinuierlichen Überwachung der veröffentlichten Website zu überprüfen. Je nach Herkunft können Messwerte in zwei Bereiche aufgeteilt werden, in Felddaten sowie in Labordaten.

Felddaten

Felddaten werden aus Messungen generiert, die direkt von den Nutzern einer Website während der Nutzung stammen und besitzen so am meisten Aussagekraft über die tatsächliche Benutzererfahrung auf der Website. Die Erfassung – auch Real User Monitoring (RUM) genannt – kann entweder selbst gesteuert werden oder durch den Bericht zur Nutzererfahrung in Chrome (Chrome User Experience, CrUX) geschehen und beinhaltet Werte zu allen Core Web Vitals. CrUX-Daten sind die offiziellen Daten, die Google im Rahmen des Web-Vitals-Programms misst. Sie unterliegen jedoch einigen Einschränkung und bilden nur einen Durchschnitt der letzten 28 Tage zum Zeitpunkt der Messung ab. So stammen die Daten ausschließlich von einer Untergruppe von Nutzern des Chrome-Browsers und werden nur auf öffentlich sichtbaren und ausreichend besuchten Websites erhoben [19].

Deshalb ist es für weniger stark besuchte Websites und zur Erhebung aussagekräftigerer Felddaten notwendig die Erfassung selbst zu führen. Dafür kann die von Google bereit-

gestellte Node.js-Bibliothek „web-vitals“ in die Webanwendung integriert werden, die alle Web-Vitals-Werte erfasst. Die Meldung, Speicherung und Auswertung der Daten muss jedoch selbst implementiert werden.

Labordaten

Mit den Tools PageSpeed Insights und Lighthouse ist es möglich, die Leistung von Websites zu simulieren und somit Labordaten zu erhalten. Dies ist bei der Entwicklung einer Webanwendung wichtig, wenn es nicht möglich ist echte Nutzerdaten zu ermitteln. Während mit PageSpeed Insights nur öffentliche Websites getestet werden können und der INP-Wert durch den Proxy TBT ersetzt wird, lassen sich mit Lighthouse Messungen im Chrome Browser über die Google DevTools durchführen. Zwar wird bei der Messung der Navigation auf eine Seite in Lighthouse ebenfalls der TBT-Wert verwendet, es ist aber auch eine Messung über eine Zeitspanne möglich, die den INP anhand realer Nutzereingaben ermittelt.

2.4 Bündeln von Webanwendungen

Bei der Entwicklung einer Webanwendung mit Frontend JavaScript Frameworks wird eine Vielzahl an JavaScript-Modulen verwendet. Würden diese Module einzeln an den Client gesendet werden, würde sich dies negativ auf die Interaktivität und damit den INP der Seite auswirken, da jedes Modul eine eigene HTTP-Anfrage benötigt. Deshalb werden Anwendungen während des Erstellungsprozesses (Build-Prozess) durch JavaScript-Bündler (Bundler) hinsichtlich ihrer Abhängigkeiten analysiert und die Module werden zu einer JavaScript-Datei – dem Bundle – zusammengefasst. Zudem wird Tree Shaking angewandt um Code, der sich zwar in den Modulen befindet, aber nicht genutzt werden wird, zu entfernen [20]. Gerade bei umfangreichen Webanwendungen können die Bündel jedoch sehr groß sein, was wiederum einen schlechten Einfluss auf die Interaktivität einer Website hat, da der im Bundle ausgeführte JavaScript-Code den Hauptthread der Anwendung für längere Zeit blockieren kann und so Nutzereingaben nicht verarbeitet werden können [21].

Code Splitting und Lazy Loading

Mit Code Splitting wird eine Anwendung während dem Bündeln in kleinere Teile (Chunks) aufgeteilt. Bei einem Aufruf der Website werden dann nur die relevanten Teile des Anwendungscodes übertragen und ausgeführt, die für den initialen Aufruf nötig sind. Die verbleibenden Teile des Codes werden dynamisch entweder erst bei Bedarf oder im Hintergrund geladen [22]. Das Laden bei Bedarf wird auch als Lazy Loading bezeichnet und kann neben JavaScript-Code auch für `<img>`- und `<iframe>`-Elemente angewandt werden. Hierbei ist darauf zu achten, dass nur JavaScript-Code, der keine blockierende Ressource darstellt, „lazy“ geladen werden kann [23]. Dadurch ist es möglich die anfängliche Ladezeit einer Website zu reduzieren.

2.5 DOM und virtueller DOM

Das Document Object Model (DOM) ist eine Programmierschnittstelle für HTML- und XML-Dokumente, die das Dokument als Baumstruktur mit Objekten als Knotenpunkte darstellt. In ihr ist der Zugriff auf die Objekte sowie die Manipulation der Objekte und des DOM-Baums definiert. Zudem identifiziert sie die Beziehungen zwischen den Objekten und das Verhalten dieser [24]. Seit 1998 ist das DOM standardisiert für die Verwaltung von HTML-Dokumenten im Browser [25]. Zugriffe oder Änderungen im DOM-Baum werden vom Browser erkannt und führen bei einer Änderung zum Aktualisieren der visuellen Darstellung. Der Browser führt einen sogenannten Reflow aus, um die Position und Geometrie der Elemente auf der Website neu zu berechnen. Im Rendering-Pfad ist dies der Layout Prozess. Während die reine Manipulation des DOM sehr schnell stattfindet, sind es die Reflows, die durch die Neuberechnung der einzelnen durch die Manipulation beeinflussten Elemente Verzögerungen bewirken. Eine hohe Anzahl nacheinander auftretender Reflows sollte demnach möglichst vermieden werden. Auch der Zugriff kann zu Leistungsproblemen führen, wenn das Element, auf welches zugegriffen werden soll, nicht klar definiert ist und so eine Suche über zu viele Knoten im Baum stattfinden muss [26].

Virtueller DOM

Damit bei komplexen modernen Webanwendungen – die stark auf ein ständiges Aktualisieren des DOMs angewiesen sind – keine Leistungseinbußen entstehen, wurde die

Technik des virtuellen DOMs (vDOM) entwickelt. Der vDOM stellt eine Kopie des eigentlichen DOMs im Arbeitsspeicher dar, mit der es möglich ist, den Zustand der Webanwendung zu jeder bestimmten Zeit zu beschreiben [27, S. 19].

Die Erstellung des vDOMs geschieht über ein JavaScript-Objekt, in dem alle DOM-Knoten als – ebenfalls – JavaScript-Objekte in einer Baumstruktur enthalten sind [28]. Findet eine Änderung in der Baumstruktur statt – sei es ein Abändern eines Elements oder ein Entfernen oder Hinzufügen von Elementen – wird eine neue, aktualisierte Version des vDOMs erstellt, die vorherige Version bleibt noch im Arbeitsspeicher erhalten. Nun kann ein Vergleich der zwei vDOM-Versionen stattfinden. Dabei kommt ein Diffing-Algorithmus zum Einsatz, der die minimale Anzahl an Knoten erkennt, die geändert werden müssen, um den gewünschten Zustand der Webanwendung im DOM zu erreichen [29]. Schließlich werden nur die Knoten aus dem Ergebnis des Diffing-Algorithmus – während dem Patch-Prozess – im richtigen DOM umgesetzt [27, S. 19]. Ein bedeutsamer Aspekt dabei ist, dass im vDOM mehrere Aktionen zusammengefasst werden und am Ende alle zusammen im DOM umgesetzt werden [28]. So kann beispielsweise bei mehreren Schreib-Lese-Schreib-Vorgängen der ständig wiederholte Reflow im Browser im Vergleich zur Ausführung im normalen DOM deutlich reduziert werden. Dadurch wird gewährleistet, dass keine Leistungseinbrüche bei der Interaktion mit der Website entstehen und sich eine bessere Nutzererfahrung ergibt.

2.6 Server-Side Rendering

Server-Side Rendering (SSR) ist eine Technologie in der Webentwicklung, mit der eine angeforderte Website vom Server an den Client ausgeliefert wird. Bei SSR wird eine Webanwendung mittels der bereitgestellten Informationen an HTML und JavaScript am Server vorgeneriert und als komplett gerendertes statisches HTML mit den notwendigen CSS- und JavaScript-Daten zum Client (Browser) übertragen [30]. Im Browser wird das HTML geparst und daraus die Nutzeroberfläche gerendert. Dadurch ist eine schnelle Darstellung der Seite möglich, die einzelnen Elemente der Seite werden jedoch erst durch das Ausführen des zugehörigen JavaScript-Codes interaktiv, da dadurch Event-Handler den Elementen zugeordnet werden können. Eine mit SSR erstellte Website bietet zudem Vorteile bei der SEO-Bewertung, da Crawler eine im Voraus generierte Seite besser indexieren können als eine Seite, deren Inhalt erst clientseitig mit JavaScript erstellt wird [31, S. 41].

2.6.1 Hydration

Um die Website für Nutzereingaben nutzbar zu machen, bedarf es der Zuordnung des JavaScript-Codes zu den HTML-Elementen im DOM. Dabei werden Event-Handler verknüpft, der Zustand der Anwendung im Browser wiederhergestellt sowie die Komponentenhierarchie erstellt [32]. Dieser Prozess wird als Hydration bezeichnet. Mit der Ausführung des JavaScript-Codes wird die Website erneut komplett gerendert – diesmal clientseitig – und der Komponentenbaum der Anwendung – falls verwendet – im vDOM gespeichert [33]. Wichtig hierbei ist die Bedingung, dass der vDOM die exakt gleiche Ausgabe erzeugt wie die durch den Server gerenderte – aktuell im DOM befindliche – Version der Webanwendung. Während der Ausführung werden Event-Handler im DOM registriert, wodurch die Website interaktiv wird und – wie eine Single Page Application – vollumfänglich im Browser agieren kann [34].

Optimierung von Hydration

Der Prozess die gesamte Website zu hydrieren, kann gerade bei größeren Anwendungen langwierig sein. Die Webanwendung sieht dann zwar nutzbar aus, bei einer Interaktion reagiert diese jedoch nicht und sie wirkt wie eingefroren[35]. Deshalb haben sich verschiedene Ansätze entwickelt, um durch die Reduzierung von JavaScript beim Seitenaufbau die Hydration nutzungsfreundlicher auszuführen. Ein Ansatz ist die Progressive Hydration, bei der die Hydration wenig relevanter Elemente der Website verzögert wird und beispielsweise erst ausgeführt wird, wenn sich das Element im Viewport des Browsers befindet oder über einen Zeitplan aufgerufen wird. Die Elemente müssen jedoch immer von der Wurzel des DOM-Baums ausgehend hydriert werden [36].

Ein weiterer Ansatz ist die Insel-Architektur (Island Architecture), mit der einzelne Teile einer Seite nicht nur verzögert, sondern unabhängig von den anderen Bereichen der Seite hydriert werden. Dies liegt daran, dass es keinen Wurzelknoten im DOM für die gesamte Webseite gibt, der hydriert werden muss. Die gesamte Seite wird als statisches HTML auf dem Server gerendert und an den Client geschickt. Bereiche, die nicht statisch bleiben – also hydriert werden müssen – sind dabei im HTML mittels Platzhalter markiert und stellen eine separate und isolierte Anwendung mit eigenem Eintrittspunkt dar, über den die Backend- und Frontend-Logik implementiert wird. [36]

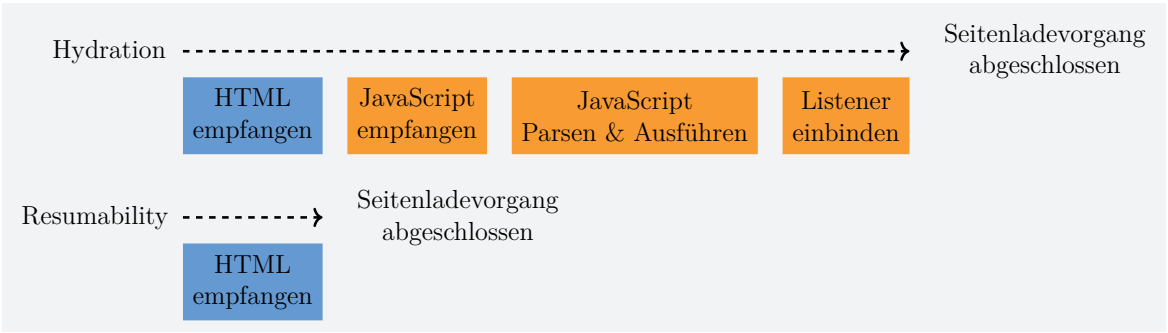

Abbildung 2.2: Gegenüberstellung von Hydration und Resumability hinsichtlich der Schritte für den Ladeorgang einer Website (nach [39])

2.6.2 Resumability

Resumability ist ein Ansatz in der Entwicklung von serverseitig gerenderten Webanwendungen, um die Seiteninitialisierung im Browser zu beschleunigen. Google verwendet diese Technik bei seinem internen JavaScript Framework Wiz, welches hinter performancekritischen Anwendungen wie beispielsweise Google Search, GMail, Google Photos und Google Drive steht [37]. Im OpenSource-Bereich findet Resumability seit 2019 im State-Management-Framework Sidewind Verwendung [6]. Als erstes öffentliches Frontend JavaScript Framework implementiert Qwik 2021 das Konzept der Resumability in seinen Rendermechanismus. Anders als bei der Hydration von servergerendertem HTML, wird die Ausführung der Webanwendung auf dem Server nicht abgeschlossen und auf der Clientseite erneut ausgeführt, sondern die Ausführung wird auf dem Server pausiert und wenn benötigt auf der Clientseite wieder aufgenommen. Zudem unterscheidet sich – wie in Abbildung 2.2 aufgezeigt – Resumability von Hydration darin, dass bei Anwendungen, die Hydration nutzen, die Elemente erst interaktiv werden, nachdem der Code des Elements auf dem Client ausgeführt wurde, während bei Resumability die gesamte Anwendung interaktiv ist, obwohl der zugehörige Code noch nicht geladen und ausgeführt wurde [38]. Es wird demnach für Resumability eine Laufzeit geschaffen, die den Anwendungscode nach Bedarf nachlädt [6, S. 9044]. Während des initialen Seitenaufrufs sind dadurch keine großen Mengen an JavaScript nötig. Findet eine Nutzerinteraktion auf der Website statt, wird dies von globalen Event-Listenern erkannt und der nötige Anwendungscode wird unabhängig von übergeordneten Elementen geladen und ausgeführt.

Serialisierung

Serialisierung ist ein Prozess, bei dem ein Objekt oder komplexe Datenstrukturen in ein mit Netzwerken übertragbares Format wie etwa JSON gebracht werden. Für Resumability ist die Serialisierung ein essentieller Bestandteil [32]. Während eine Serialisierung des Anwendungszustands auch zur Beschleunigung der Hydration eingesetzt wird, serialisieren auf Resumability bauende Frameworks zusätzlich die Komponentenhierarchie und Informationen zu den Event-Handlern. Durch das Deserialisieren der Informationen während des Parsens des HTML im Browser erlangt die Anwendung – wie beim Ausführen des Anwendungscodes bei der Hydration – ihr Frontend-Wissen. Das Deserialisieren ist jedoch deutlich schneller als ein Ausführen des Anwendungscodes [6].

2.7 Static Site Generation

Static Site Generation (SSG) verhält sich ähnlich wie SSR, mit dem Unterschied, dass das HTML einer Website während des Build-Prozesses generiert wird. Eine Website kann so nach einem Aufruf direkt vom Server an den Client geschickt werden, was zu schnellen Ladezeiten führt. SSG ist vor allem für Webseiten sinnvoll, deren Inhalt sich nicht ändert, wie es beispielsweise bei Blogs oder Dokumentationen der Fall ist. [40]

2.8 Client-Side Rendering

Eine Webanwendung, die auf Client-Side Rendering (CSR) basiert, enthält im HTML-Dokument im `<body>` lediglich einen Container in Form eines `<div>`-Elements – das als Wurzelelement dient – und agiert als SPA. Das gesamte Rendern der Anwendung findet, nachdem die JavaScript-Ressourcen geladen wurden, im Browser statt. Die Erstellung der Webseite im DOM erfolgt rein mittels JavaScript, wodurch erst nach der Ausführung des gesamten Anwendungscodes die Webseite im Browser sichtbar wird. Abhängig vom Umfang der Website und den Elementen im DOM nimmt die Menge an JavaScript zu und der initiale Rendervorgang kann sehr lange ausfallen, was einen deutlich negativen Einfluss auf die Nutzererfahrung hat. [41]

3 Frameworks

Die Webentwicklung wurde durch den Einsatz immer neuerer Technologien, wie z. B. mit der Einführung von AJAX, immer komplexer. Um dieser steigenden Komplexität zu begegnen, wurden Frameworks entwickelt. Die in Kapitel 2 vorgestellten Technologien finden in verschiedenen Frameworks Anwendung. Jedoch werden nicht alle Technologien bei jedem Framework eingesetzt, sondern nur eine Kombination einiger, wobei sich auf bestimmte Technologiekonzepte spezialisiert wird. Da jede Technologie für bestimmte Arten von Webanwendungen gut oder weniger gut funktioniert, ist es wichtig sich vor der Wahl eines Frameworks darüber bewusst zu sein, welche Eigenschaften die Webanwendung haben wird und was die Herausforderungen sein könnten.

Bei den in dieser Arbeit verwendeten Frameworks handelt es sich um deklarative JavaScript-Bibliotheken und Frameworks zum Erstellen von interaktiven Benutzeroberflächen. In diesem Kapitel werden die Bibliotheken und Frameworks React, Solid.js und Qwik sowie dazugehörige Metaframeworks vorgestellt. Das Hauptaugenmerk liegt dabei auf ihren Kerntechnologien mit Einfluss auf die Ladegeschwindigkeit und Interaktivität von Webanwendungen.

3.1 React

React ist eine JavaScript-Bibliothek zur Erstellung von interaktiven Benutzeroberflächen für Webanwendungen, die 2011 von Meta (damals Facebook) für die Social-Media-Plattform Facebook entwickelt und 2013 als Open-Source-Projekt veröffentlicht wurde [42, S. 2]. Zudem ist React das mit Abstand am meisten genutzte Frontend JavaScript im Jahr 2023 [4]

```
function Counter() {
  const [count, setCount] = useState(0);
  return <button onClick={() => setCount(count + 1)}>{count}</button>;
}
```

Listing 3.1: React Komponente mit Zustandsvariable und JSX Syntax

```
function Counter() {
  const [count, setCount] = useState(0);
  return /*#__PURE__*/ _jsx("button", {
    onClick: () => setCount(count + 1),
    children: count,
  });
}
```

Listing 3.2: Transformiertes JSX der React Komponente aus Listing 3.1

3.1.1 Komponenten mit JSX

Eine mit React erstellte Webanwendung besteht aus verschiedenen Komponenten, die wiederverwendbar sind und miteinander verschachtelt werden können. Eine Komponente stellt eine JavaScript-Funktion dar, die Anwendungslogik sowie HTML-Auszeichnung für Benutzeroberfläche enthält. Über den Funktionsnamen als HTML-Tag lassen sich Komponenten im HTML-Teil einer anderen Komponente einbinden.

React hat für die Erstellung von Komponenten JSX entwickelt – eine Syntax-Erweiterung für JavaScript [43]. Die Nutzung von JSX ist in React jedoch nicht verbindlich. Mit JSX können HTML-Elemente in JavaScript geschrieben werden, ohne auf die HTML-Syntax verzichten zu müssen (vgl. Listing 3.1). Dabei können JavaScript-Ausdrücke in das HTML eingebunden werden, wodurch eine Verbindung zwischen der Anwendungslogik und dem HTML für die Benutzeroberfläche hergestellt wird und Inhalte dynamisch angezeigt werden können. JSX wird von dem Compiler Babel zu regulärem JavaScript-Code transformiert (vgl. Listing 3.2), der für die Erstellung des HTML verantwortlich ist [44]. Da das JSX der Rückgabewert der Komponentenfunktion ist und eine JavaScript-Funktion nicht mehrere Objekte zurückgeben kann, müssen mehrere JSX-Tags stets innerhalb eines einzelnen, übergeordneten Tags verschachtelt sein [43].

Damit eine Komponente interaktiv wird, benötigt sie einen Zustand, der aktualisiert werden kann. Der Zustand wird in React mittels Zustandsvariablen erzeugt, die durch

Hooks – von React bereitgestellte Funktionen – wie `useState()` deklariert werden [45]. Dabei besteht eine Zustandsvariable immer aus dem aktuellen Wert des Zustands sowie aus einer Funktion zum Ändern des Zustands (vgl. Listing 3.1). Das Ändern über eine `set`-Funktion löst ein erneutes Rendern der Komponente aus, wobei der neue Wert des Zustands erst nach Abschluss des Renderns auslesbar ist. Denn die Renderphase wird beim Verarbeiten eines Events erst ausgeführt, nachdem der gesamte Code ausgeführt wurde, um mögliche unnötige Rendervorgänge des vDOMs zu vermeiden [46].

3.1.2 Verbesserung des vDOMs mit React Fiber

Eine Kerntechnologie von React ist der virtuelle DOM, für den React den sogenannten Reconciliation-Prozess entwickelt hat. Reconciliation ist für die Steuerung des vDOMs verantwortlich, also für das Erstellen der Anwendung als Baumstruktur, für das Vergleichen einer neuen Version des Baums bei einer Aktualisierung des Anwendungszustands mit der alten Version sowie das Ermitteln der nötigen Operationen für die Umsetzung im DOM. Dabei gibt es zwei wichtige Grundsätze, die nötig sind, um Reconciliation effizient zu gestalten. Ändert sich die Art eines Knotens im Baum, wird der gesamte Baum darunter komplett ersetzt, ohne die Unterschiede in den Unterknoten zu prüfen. Zudem müssen für Listenelemente eindeutige und beständige Schlüssel vergeben werden. [47]

Reconciliation wurde von React mit der Einführung von React Fiber weiterentwickelt, wobei die zwei Grundsätze bestehen bleiben. Hinzu kommt, dass mit React Fiber Aktualisierungen über eine Zeitspanne betrachtet werden und Änderungen an der Benutzeroberfläche Prioritäten zugeordnet werden. Zudem lassen sich schon begonnene Arbeiten im vDOM unterbrechen, wiederaufnehmen, komplett abbrechen oder wiederverwenden. [47] Dies ist möglich, da der Baum nun nicht mehr aus unveränderlichen Objekten als Knoten besteht und neu erstellt werden muss, sondern aus Fiber-Knoten. Diese Knoten enthalten den Zustand der Komponente sowie zugehörige DOM-Elemente und sind veränderbar, wodurch sie bei einer Änderung nicht komplett neu erstellt werden müssen [48]. Eine Fiber kann als Arbeitseinheit bezeichnet werden, die Komponenten zugeordnet ist und mit ihren Ausgabeeigenschaften die vorzunehmenden Änderungen im DOM beschreibt [47]. Das Ziel von React Fibers ist, den virtuellen DOM effizienter und schneller zu machen und somit auch die Interaktivität von Webanwendungen zu verbessern.

3.1.3 Next.js

Die Entwickler von React empfehlen für das Erstellen einer neuen Webanwendung mit React auf ein Metaframework zurückzugreifen. Das laut der State-of-JS-Umfrage für 2023 am häufigsten verwendete Metaframework für React ist Next.js, das von dem Unternehmen Vercel entwickelt wird [49]. Die Hauptfunktionen von Next.js sind ein verzeichnisbasierter Router und Rendering-Optionen für CSR, SSR und SSG. Next.js bietet die Wahl zwischen zwei Routern – dem Pages- und dem App-Router – wobei der App-Router eine Weiterentwicklung des Pages-Routers ist und neue React Technologien wie React Server Components (RSC) und Streaming enthält [50]. Next.js verwendet standardmäßig SSR mit clientseitiger Hydration.

React Server Components

Die React Server Components (RSC) sind eine Entwicklung von React, die bisher nicht mit React selbst, sondern nur mit wenigen Metaframeworks stabil nutzbar sind [51]. Komponenten in React können mit RSC in Server-Komponenten und Client-Komponenten unterschieden werden, wobei Server-Komponenten ausschließlich auf dem Server gerendert werden, das Rendern der Client-Komponenten jedoch serverseitig und clientseitig geschieht. Server-Komponenten sind statische Komponenten, in denen – anders als bei Client-Komponenten – keine Interaktion stattfinden kann und die keinen eigenen Zustand besitzen [51]. Deshalb muss auch kein JavaScript-Code zum Rendern der Komponenten an den Client ausgeliefert werden. Grundsätzlich sind alle Komponenten mit dem Next.js App Router Server-Komponenten. Sie werden jedoch automatisch zu Client-Komponenten, wenn sie von einer solchen importiert werden, da Client-Komponenten bei Zustandsänderungen zusammen mit allen Unterkomponenten clientseitig neu gerendert werden [52]. Dadurch, dass sich das vDOM bei der clientseitigen initialen Erstellung der Webanwendung nicht von dem auf dem Server gerenderten HTML unterscheiden darf, müssen die Daten der Server-Komponenten trotzdem clientseitig für die Erstellung des vDOM bekannt sein. Dafür werden sie serialisiert im HTML-Dokument mitgegeben [53]. Durch diese Technik lässt sich die Menge an JavaScript für den Client reduzieren und verkürzt zudem die Hydration der Webanwendung.

3.2 Solid.js

Die JavaScript-Bibliothek Solid.js wird seit 2016 von Ryan Carniato entwickelt und erreichte 2021 die Veröffentlichung in der Version 1.0. Solid.js orientiert sich in seinen Grundzügen – wie der komponentenbasierten Architektur in Verbindung mit JSX und der Deklaration von reaktiven Zustandsvariablen – an React (vgl.Listing 3.3). Solid.js nutzt einen Compiler, der es ermöglicht eine feinkörnige Änderungserkennung und Umsetzung bei sich ändernden Zuständen in der Anwendung durchzuführen. Die Änderungen werden direkt im DOM umgesetzt, wodurch auf einen virtuellen DOM verzichtet werden kann.

```
1 function Counter() {
2   const [count, setCount] = createSignal(0);
3   return (
4     <div>
5       <button onClick={() => setCount((count) => count + 1)}>{count()}</button>
6     </div>
7   );
8 }
```

Listing 3.3: Komponente in Solid.js mit einer Zustandvariablen und JSX

3.2.1 Feinkörnige Reaktivität

Zustände werden in Solid.js in Signalen gespeichert, die einen Wert, einen Getter, einen Setter sowie ein Set an sogenannten Subscribern enthalten. Mit Effekten können Funktionen erzeugt werden, die automatisch aufgerufen werden, sobald sich der Zustand – also der Wert des Signals – ändert. Wird ein Signal über seine Getter-Funktion aufgerufen, wird das aufrufende Element oder – im Fall von Effekten – die aufrufende Funktion als Subscriber gespeichert. Dadurch ist es – anders als in React – nicht nötig die Abhängigkeiten für Effekte extra zu definieren. Ändert sich der Wert des Signals über die Setter-Funktion, wird durch das Set an Subscribern iteriert und jedem Subscriber der neue Wert des Signals – also der neue Zustand – mitgeteilt [54].

Signale müssen bei ihrer Initialisierung nicht zwingend an eine Komponente gebunden sein, sie werden aber in Komponenten genutzt. Komponenten sind wiederverwendbare Strukturelemente von Solid.js-Anwendungen und beschreiben in einer Baumstruktur

aufgebaut das Aussehen einer Website. Der Lebenszyklus von Komponenten in Solid.js beschränkt sich auf ihren initialen Aufruf beim Rendern in das DOM. Wird eine Komponente im Browser gerendert, erzeugt sie ein feinkörniges reaktives System, das die für die Komponente relevanten Zustandsänderungen in Form von Signalen und Effekten überwacht. Bei einer Änderung des Zustands muss nicht die gesamte Komponente neu gerendert werden, sondern nur das betroffene Element [55]. Verantwortlich dafür ist der Compiler „Babel Plugin JSX DOM Expressions“ zusammen mit der Bibliothek „DOM Expressions“. Der Compiler erstellt aus dem statischen HTML der Komponente ein HTMLTemplateElement sowie eine Funktion der Komponente (vgl. Listing 3.4). In der `return`-Anweisung der Komponentenfunktion befindet sich eine Immediately Invoked Function Expression (IIFE) – eine anonyme Funktion, die nach ihrer Definition direkt aufgerufen wird. Dort wird das HTML-Template in eine Variable geklont und davon abgeleitet werden alle geschachtelten Elemente des Hauptelements ebenfalls jeweils Variablen zugewiesen. Wenn Elemente einen Event-Listener benötigen, wird ihnen dieser zugeteilt. Klick Events werden – um Code zu sparen und die Bundlegrößen zu reduzieren – abstrahiert mittels `$$click` zugeteilt und von der Bibliothek später umgewandelt. Über eine `_$insert`-Funktion werden Effekte für an Signale gebundene Elemente – die bei einer Zustandsänderung manipuliert werden – erzeugt, wobei anhand des Datentyps der Zustandsvariablen die Manipulationsmethode des DOMs bestimmt wird. Der Effekt überwacht den Wert der Zustandsvariablen und führt die Manipulation direkt am Knoten im DOM aus. Die Komponentenfunktion gibt am Ende die Komponente als HTML-Element zurück, das dann gerendert werden kann [56].

```
1  var _tmpl$ = /*#__PURE__*/_$template(`<div><button>`);
2  function Counter() {
3      const [count, setCount] = createSignal(0);
4      return (() => {
5          var _el$ = _tmpl$(),
6              _el$2 = _el$.firstChild;
7          _el$2.$$click = () => setCount(count => count + 1);
8          _$insert(_el$2, count);
9          return _el$;
10     })();
11 }
```

Listing 3.4: Kompilierte Solid.js Komponente durch „Babel Plugin JSX DOM Expressions“

3.2.2 SolidStart

SolidStart ist ein Metaframework für Solid.js, das als Plattform zum Integrieren verschiedener Komponenten in eine Webanwendung dient. Die Architektur von SolidStart ist minimalistisch und stellt nur die grundlegendsten Bausteine zur Verfügung, um eine Webanwendung zu entwickeln. So ist standardmäßig beispielsweise kein Router enthalten. Entwickler sind also nicht an die Nutzung vorgegebener Bibliotheken gebunden, weshalb sich SolidStart nicht als „Opinionated Framework“ versteht [57].

Rendering

SolidStart nutzt das JavaScript-SDK Vinxi, das auf Vite als Bundler und Nitro als HTTP-Server aufbaut. Mit Vinxi ist es möglich zwischen Client Side Rendering (CSR), Server Side Rendering (SSR) mit Hydration und Static Site Generation (SSG) zu wählen, ohne Änderungen am Router des Frameworks vornehmen zu müssen [58]. Für SSR und CSR gibt es spezielle Eintrittspunkte in Form von TSX-Dateien. Diese werden je nach Konfiguration beim Anfordern der Website aufgerufen, wobei darauf zu achten ist, dass der Code aus dem Server-Eintrittspunkt auch bei CSR ausgeführt wird, bei SSR jedoch nicht der Code aus dem Client-Eintrittspunkt. Im Eintrittspunkt des Servers ist es möglich die SSR Methode zu definieren. Hier wird zwischen drei Methoden unterschieden. Eine synchrone Methode rendert den Inhalt der Website synchron in einen String mit Platzhaltern für asynchrone Daten. Die asynchronen Daten werden im Browser abgerufen und gerendert. Durch das Nutzen des Servers sowie des Clients wird diese Methode auch als hybrides SSR bezeichnet. Eine asynchrone Methode wartet bis alle asynchronen Daten geladen sind, bevor der String zurückgegeben wird. Die dritte Methode rendert den Inhalt der Seite in einen Stream. Wie bei der hybriden Methode werden für asynchrone Daten Platzhalter an den Client gesendet, das Laden der asynchronen Daten findet jedoch serverseitig statt. Fertig gerenderte Daten werden – sobald sie am Server gerendert sind – nach und nach an den Client geschickt. Diese Methode skaliert mit verschiedenen Netzwerkgeschwindigkeiten. So ist sie bei einem schnellen Netzwerk performanter als die Hybridmethode und in langsamen Netzwerken identisch mit der asynchronen Methode [59].

```
export const Counter = component$(() => {
    const count = useSignal(0);
    return <button onClick$={() => count.value++}>{count.value}</button>;
});
```

Listing 3.5: Qwik Komponente mit einer Zustandsvariablen und JSX [60]

3.3 Qwik

Qwik ist ein Frontend JavaScript Framework, das von Miško Hevery, Manu Almeida und Adam Bradley entwickelt wird und im Mai 2023 die Veröffentlichungsversion 1.0 erreichte. Qwik orientiert sich im Aufbau – wie auch Solid.js – an React, weshalb die Komponenten mit JSX gleich strukturiert sind (vgl. Listing 3.5). Die von Qwik verwendeten Technologien für die Entwicklung von Webanwendungen unterscheiden sich jedoch deutlich von React, da Qwik nativ SSR nutzt und durch Resumability auf die Hydration verzichten kann. Die Philosophie von Qwik ist, das Laden von Code im Browser möglichst lange herauszuzögern, jedoch trotzdem eine performante und direkt nutzbare Webanwendung zu erstellen [61, S. 324]. Dazu wird der Code der Anwendung im Build-Prozess von dem Qwik Optimizer – ein Code Transformator – in eine hohe Anzahl kleiner Stücke, die als Symbole bezeichnet werden, aufgeteilt. Symbole lassen sich während der Laufzeit der Webanwendung im Browser individuell mittels Lazy Loading nachladen.

3.3.1 `$$$` für den Optimizer

Das `$`-Zeichen stellt eine spezielle Syntax von Qwik dar und markiert einen serialisierbaren Bereich für den Qwik-Optimizer, aus dem Symbole erstellt werden. Funktionen mit dem `$`-Suffix werden in eigene Dateien ausgelagert. Der originale Code wird so transformiert, dass ein Verweis auf die neu erstellte Datei hinterlegt wird (vgl. Listing 3.7). Hierbei kommen Qwik-URLs (QRL) im Format `./path/to/chunk.js#SymbolName` zum Einsatz, die den Pfad zur Datei sowie den Symbolnamen enthalten. Die Daten für die QRL werden in den vom Optimizer erstellten Symbolen in eine `qrl()`-Funktion gesetzt, die beim Rendern des HTMLs, wie in Listing 3.6 gezeigt, die QRL als Attribut des zugehörigen Elements ausgibt [62].

```
<button on:click="./chunk-b.js#Counter_onClick[0]">0</button>
```

Listing 3.6: Gerendertes HTML der Komponente aus Listing 3.5 mit QRL in Click Listener [63]

```
const Counter = component(qrl('./chunk-a.js', 'Counter_onMount'));

//chunk-a.js:
export const Counter_onMount = () => {
    const count = useSignal(0);
    return <button onClick$={qrl('./chunk-b.js', 'Counter_onClick', [count])}>{count
    .value}</button>;
};

//chunk-b.js:
const Counter_onClick = () => {
    const [count] = useLexicalScope();
    return count.value++;
};
```

Listing 3.7: Transformierter Code der Qwik-Komponente aus Listing 3.5 durch den Qwik-Optimizer [60]

Häufig entstehen Symbole im Zusammenhang mit Event-Handlern und den davon aufgerufenen Funktionen. Wichtig hierbei ist es den lexikalischen Gültigkeitsbereich der Closures von beispielsweise Event-Handlern wiederherstellen zu können. Dafür werden die Konstanten innerhalb einer Closure von dem Optimizer erfasst und an die QRL angehängt sowie in der generierten Closure – dem Symbol – mittels `useLexicalScope()` wiederhergestellt.

Auch Qwik-Komponenten werden in einem `component$()`-Aufruf verpackt. Komponenten besitzen dadurch automatisch Lazy-Loading-Eigenschaften und werden beispielsweise erst geladen, wenn sie sich im Viewport des Browsers befinden [64]. Sie sind also unabhängig von übergeordneten Komponenten und müssen nicht mit diesen mitgeladen werden.

3.3.2 Qwikloader

Damit die QRLs während der Laufzeit der Webanwendung verwendet werden können, benötigt Qwik den sogenannten Qwikloader. Dieser ist als Inline-JavaScript im HTML-Dokument enthalten und registriert globale Event-Listener für alle Browser-Events. Ein ausgelöstes Event wird vom Qwikloader zum auslösenden Element zurückverfolgt. Anhand des `on:...`-Attributes – welches die QRL enthält – lädt er das zugehörige Symbol und ruft es auf [63].

3.3.3 Reaktivität

Qwik nutzt für Zustandsänderungen die zu keiner Neustrukturierung des DOMs führen – wie auch Solid.js – das Prinzip der feinkörnigen Reaktivität. So werden beispielsweise Änderungen am Text eines Elements direkt am DOM-Knoten umgesetzt, ohne einen virtuellen DOM zu nutzen und die gesamte Komponente neu zu rendern. Die direkte Umsetzung von Strukturänderungen mittels feinkörniger Reaktivität ist aufgrund der Resumability nicht möglich, da der gesamte Anwendungscode im Browser beim Seitenaufruf verfügbar sein und ausgeführt werden müsste. Qwik verwendet hierfür einen vDOM. Informationen über die Hierarchie der Komponenten serialisiert Qwik während des serverseitigen Renderns im HTML. Damit ist es möglich die Grenzen von Komponenten genau zu bestimmen. [65]

Beim Ausführen einer Komponentenfunktion während der Erstellung des HTMLs auf dem Server wird für Zustandsvariablen ein Proxy erstellt, der die Lese- und Schreibvorgänge der Variablen überwacht. Ein Lesen führt zum Erstellen eines Subscribers und das Ändern zum Invalidieren der Komponente des Subscribers. Durch das Invalidieren muss die Komponente clientseitig neu gerendert werden, wodurch der neue Wert der Zustandsvariablen beachtet wird und entsprechende Manipulation am DOM stattfinden. Es wird nur die Komponente – ohne Unterkomponenten – neu gerendert, in der eine strukturelle Änderung stattgefunden hat. [66]

3.3.4 QwikCity

QwikCity ist ein Metaframework von Qwik und erweitert die auf die Erstellung von Nutzeroberflächen ausgelegte Logik von Qwik um Router- und Serverfunktionen [67]. Eine Hauptkomponente ist der verzeichnisbasierte Router, mit dem es möglich ist die

Webanwendung als Multi Page Application und als Single Page Application zu nutzen. Entwickler haben die Möglichkeit für jeden Link separat festzulegen, ob die Navigation als MPA oder SPA erfolgen soll [68].

Service Worker

Ein essentieller Faktor für die Geschwindigkeit von Qwik-Anwendungen ist die Caching-Strategie. Dadurch, dass annähernd der gesamte Anwendungscode erst bei der tatsächlichen Nutzung in kleinen Modulen geladen wird, ist es nötig dieses Nachladen effizient zu halten. Ein Anfordern des Codes vom Server, nachdem eine Interaktion getätigt wurde, kann durch schlechte Netzwerkeigenschaften eine deutliche Verzögerung der Nutzereingabe nach sich ziehen. Um eine gute Nutzererfahrung zu gewährleisten, registriert QwikCity einen Service Worker, der die Aufgabe hat Programmcode – der möglicherweise bald ausgeführt werden muss – vom Server zu laden, im Cache des Service Workers abzulegen und bei einem Anfordern durch Qwik als Antwort zurückzugeben [69]. Mit dem Service Worker ist es möglich das Vorausladen im Hintergrund durchzuführen und somit den Main Thread zum Rendern der Website freizuhalten. Zudem kennt der Service Worker den Modulgraph der Anwendung, wodurch es ihm möglich ist alle voneinander abhängigen Module mit der Anforderung des ersten Moduls an den Browser zurückzugeben [70].

4 Praktische Arbeit

In der Arbeit von Lipiński und Pańczyk [71] wurde Qwik mit React und Next.js verglichen. Dabei wurden Messungen zu den Metriken First Contentful Paint, Largest Contentful Paint, Total Blocking Time und Speed Index an drei verschiedenen Webanwendungen – die je ein bestimmtes Szenario abgedeckt haben – durchgeführt. Es wurde das Rendern verschieden hoher Anzahlen von DOM-Elementen, das Laden und Anzeigen verschieden hoher Anzahlen großer Bilder und die Durchführung rechenintensiver Operationen wie das Herunterladen und Verarbeiten externer Daten gemessen. [71] Lipiński und Pańczyk untersuchen damit ausschließlich den initialen Seitenaufruf und keine Nutzung der Website. Zudem simulierten sie Szenarien mit extrem vielen DOM-Elementen, die so vermutlich bei realen Webseiten selten vorkommen. Da viele Technologien der Frameworks einen Einfluss auf die Interaktion einer Website haben, ist es wichtig auch das Verhalten während der Nutzung zu analysieren. Vepsäläinen et al. [6, S. 9044-9045] geben an, dass es einiges an Forschung bedarf, um Resumability hinsichtlich der Leistung gegen andere Technologien wie Hydration in realen Anwendungsfällen zu testen. Deshalb wurde in dieser Arbeit der Fokus auf die Untersuchung einer realen Nutzung von Websites gelegt.

4.1 Methoden und verwendete Werkzeuge

In diesem Abschnitt werden die Methoden zur Datenerhebung für den Vergleich der verschiedenen Technologien und Frameworks beschrieben. Zudem werden die verwendeten Werkzeuge für die Erstellung der Webanwendungen und die Durchführung der Messungen vorgestellt.

4.1.1 Methoden

Um die verschiedenen Frameworks miteinander zu vergleichen, wurden mit jedem Framework zwei Webanwendungen mit unterschiedlichem Umfang erstellt – eine mini-

malistische Webanwendung und eine realitätsnahe, interaktive Webanwendung –, um daran Messungen zum Page Speed vorzunehmen. Die Ergebnisse der Messungen wurden quantitativ untersucht. Zudem wurden Technologien aufgezeigt, die einen Einfluss auf den Page Speed nehmen und die Funktionsweise der Frameworks besonders prägen.

Mit den Messungen über Lighthouse wurden Labordaten zu den Core Web Vitals erhoben. Durch die Ermittlung der Labordaten innerhalb einer gleichbleibenden Versuchsumgebung ist eine Vergleichbarkeit der Messergebnisse gegeben, da der Einfluss externer Faktoren auf das Nötigste – die Antwortzeit des Servers – minimiert wird. Eine gewisse Variabilität der Antwortzeiten des Servers, der die Website liefert, spielt auch bei realen Webanwendungen eine Rolle [72, S. 254]. Für die Messung der Core Web Vitals wurden vier unterschiedliche Netzwerkbedingungen auf einem mobilen Gerät simuliert, um eine mobile Nutzung abzubilden. Dies ist besonders aufschlussreich, da die Performance auf mobilen Geräten oft kritischer ist und die Messungen so eine realistische Einschätzung der Benutzererfahrung ermöglichen.

Es wurden je Webanwendung und je Netzwerkgeschwindigkeit zehn Messungen für zwei Aufrufszenarien durchgeführt. Das erste Aufrufszenario betrachtet einen ersten Websiteaufruf ohne gespeicherte Websitedaten im Cache. Das zweite Aufrufszenario stellt einen weiteren Seitenaufruf mit Websitedaten im Cache dar. Im Folgenden werden die Aufrufszenarien „ohne Cache“ und „mit Cache“ genannt. Die Anzahl der Messungen orientiert sich an einer Studie zur Genauigkeit von Lighthouse Messmethoden [73]. Aus diesen zehn Messungen wurde jeweils der Median ermittelt und als das Ergebnis für eine Metrik ausgewiesen. Der Median wurde verwendet, da bei ihm – im Gegensatz zum Mittelwert – Ausreißerwerte, die durch Netzwerkschwankung auftreten können, nicht ins Gewicht fallen. Somit liefert er vergleichbare Messergebnisse. Anschließend wurde für die Medianwerte der zugehörige Score der Core Web Vitals ermittelt.

Neben den Lighthouse-Messungen wurde die Abdeckung der Webanwendungen nach dem Ladevorgang sowie nach dem Ausführen von mehreren Benutzerinteraktionen auf der Website – also einer Nutzung – gemessen. Hier wurden für jeden Fall fünf Messungen durchgeführt und davon die gerundeten Mittelwerte als Ergebnis verwendet. Der Mittelwert ist hier gut anwendbar, weil es keine Ausreißerwerte gibt. Die Abdeckung gibt Aufschluss darüber, wie viel von der empfangenen Menge an JavaScript und CSS tatsächlich genutzt wird. Damit werden die Auswirkungen von Technologien geprüft, die das effiziente Bündeln und Ausbringen des Anwendungscodes betreffen.

4.1.2 Auswahl der Werkzeuge

Die Auswahl der Technologien für die Erstellung der Testanwendungen erfolgte anhand ihrer Verwendung und Verbreitung im Bereich der Webentwicklung, um reale Anwendungsfälle abzubilden. Für die Entscheidung wurden die aktuellen Umfragen für das Jahr 2023 zur Nutzung von Technologien in der Webentwicklung von Stackoverflow[1] und State of JS[2] herangezogen.

4.1.2.1 Programmiersprache - TypeScript

Für die Entwicklung der Webanwendungen in dieser Arbeit wurde TypeScript ausgewählt. TypeScript ist eine auf JavaScript basierende Programmiersprache, die statische Typisierung ermöglicht. TypeScript hat sich unter JavaScript-Entwicklern zu einer weit verbreiteten Alternative entwickelt, da durch die Typisierung Laufzeitfehler frühzeitig während der Programmierung erkannt werden, was vor allem bei großen und komplexen Anwendungen für die Entwicklung und Wartung hilfreich ist [74]. Alle in dieser Arbeit verwendeten Frameworks unterstützen die Nutzung von TypeScript von Grund auf, wobei Qwik ausschließlich die Verwendung von TypeScript vorsieht.

4.1.2.2 Frameworks

Zum Erstellen der Webanwendungen wurden die vorgestellten Metaframeworks verwendet und mittels ihrer CLIs initialisiert. Nur um CSR mit React zu nutzen wurde React zusammen mit Vite und dem React-Router-DOM aufgesetzt. Auch wenn es möglich ist Qwik mit reinem CSR – ohne serverseitiges Rendern des Inhalts der Website – zu nutzen, wurde dies in dieser Arbeit ausgeschlossen, da jegliche Kerntechnologien, die Qwik ausmachen, so nicht genutzt werden würden. Da SolidStart sowohl SSR als auch CSR ermöglicht, wurden für beide Rendermethoden die Webanwendungen erstellt. Die Auswahl von SSR oder CSR findet rein über die Konfiguration des Frameworks statt, weshalb derselbe Code verwendet werden konnte.

[1] `https://survey.stackoverflow.co/2023/`
[2] `https://2023.stateofjs.com/`

4.1.2.3 Hosting und Datenbank - Vercel und Vercel Postgres

Alle Webanwendung wurden über die gleiche Cloud-Hosting-Plattform, Vercel, für den Zugriff über das Internet bereitgestellt, um eine Beeinflussung der Messungen durch unterschiedliche Servertechniken ausschließen zu können. Vercel ist laut State of JS eine der am meisten verwendeten Hosting-Plattformen. Für die persistente Speicherung von Daten wurde die Datenbank Vercel Postgres mit dem objektrelationalen Datenbanksystem PostgreSQL ausgewählt. PostgreSQL ist laut der Stackoverflow Umfrage für das Jahr 2023 zur Nutzung von Datenbanken die meistgenutzte Datenbank. In Verbindung mit Vercel als Hosting-Plattform reduziert sich die Dauer der Datenbankzugriffe und somit die Serverantwortzeit bei der Verwendung von SSR.

4.1.2.4 ORM - Drizzle

Ursprünglich wurde Prisma für objektrelationales Mapping (ORM) in dieser Arbeit in Betracht gezogen. Prisma bietet eine deklarative Schemadefinition durch eine eigene Sprache – der Prisma Schema Language – sowie Typensicherheit mittels TypeScript [75]. Aufgrund von Schwierigkeiten bei der Einbindung zusammen mit Qwik wurde sich letztendlich für Drizzle als ORM-Tool entschieden. Drizzle verwendet TypeScript für die Erstellung des Schemas sowie für die Verwaltung der Datenbankoperationen und gewährt somit auch Typensicherheit. Drizzle bietet zudem eine minimalistische Abstraktion, die nahe an SQL-Befehlen bleibt und dadurch eine direkte Datenbankinteraktion ermöglicht.

4.1.2.5 CSS Framework - TailwindCSS

TailwindCSS wurde als CSS-Framework ausgewählt, um die Gestaltung der Benutzeroberfläche während der Entwicklung der interaktiven Webanwendung effizient umsetzen zu können. TailwindCSS folgt einem Utility-First-Ansatz, bei dem vordefinierte CSS-Klassen verwendet werden, um direkt im HTML Stile auf HTML-Elemente anzuwenden [76]. Die vordefinierten Klassen geben dabei nicht den gesamten Stil für ein Element vor, sondern meist nur eine einzelne CSS-Eigenschaft. So ist bei der Entwicklung direkt am Element sichtbar, welche Stile angewendet werden, was zusammen mit JSX einen gesamten Überblick über HTML, CSS und JavaScript einer Komponente liefert.

4.1.2.6 Bibliotheken - Chart.js, Lodash und Three.js

Für die Erstellung eines Dashboards wurde die JavaScript-Bibliothek Chart.js verwendet, mit der Diagramme in einem HTML-Canvas gezeichnet werden. Für Chart.js gibt es spezifische Module für die Nutzung mit React und Solid.js, die fertige Komponenten für unterschiedliche Arten von Diagrammen bereitstellen.
Zusätzlich wurde Lodash eingebunden, eine häufig in Webanwendungen genutzte Bibliothek, die Hilfsfunktionen für das Verarbeiten von Datentypen und -strukturen bietet.
Weiterhin wurde die Bibliothek Three.js hinzugefügt, mit der 3D-Grafiken und -Animationen auf einer Website direkt im Browser gerendert werden können. Three.js benötigt eine hohe Menge JavaScript, wodurch das Volumen der Website erhöht wird.

4.1.2.7 Page Speed Messung - Lighthouse

Lighthouse lässt sich neben der Ausführung in den Chrome DevTools auch als programmatische Variante verwenden, wodurch eine hohe Anzahl an Messungen effizient durchgeführt werden kann.
Damit die Bewertung einer gesamten Website möglich ist, wurde in dieser Arbeit die Lighthouse User Flows API verwendet, mit der mehrere Arten von Messungen zusammenhängend durchgeführt werden können. Dadurch können Nutzerinteraktionen mit einer Seite über die Zeitspannen-Messung sowie Navigationen innerhalb einer schon geladenen Seite mit Website-Daten im Cache erfasst und bewertet werden. [77]. Dies spiegelt eine reale Nutzung der Seite wider und führt zu aussagekräftigeren Core-Web-Vitals-Werten als eine Einzelmessung in den DevTools, da in den DevTools die Navigationsmessungen mit leerem Cache und leerem lokalem Speicher stattfinden.

4.1.2.8 Browser Automation - Puppeteer

Für die Messung der INP-Metrik müssen Nutzerinteraktionen mit der Webanwendung stattfinden. Um Messungen automatisiert durchführen zu können, wurde ein Tool benötigt, das im Browser auf Elemente der Webseite zugreifen und mit diesen interagieren kann. Puppeteer ist eine Node.js-Bibliothek, mit der eine automatisierte Steuerung des Chrome Browsers möglich ist. Dafür nutzt Puppeteer eine headless (fensterlose) Version von Chrome zum Aufrufen von Websites. [78] Für die Kommunikation mit dem Browser sowie für die Kommunikation mit der Website verwendet Puppeteer das Chrome DevTools Protocol. Puppeteer kann zusammen mit der Lighthouse User Flow

	Downloadrate in kBit/s	Uploadrate in kBit/s	Paketumlaufzeit in ms
Slow 3G	400	400	200
Fast 3G	1600	768	75
LTE	12000	5000	35
WiFi	250000	100000	1

Tabelle 4.1: Simulierte Netzwerkbedingungen mit Throttle

API verwendet werden und führt die Interaktionen auf der Seite aus, die dann von Lighthouse gemessen werden. Zudem wurde Puppeteer für die Messung der Abdeckung verwendet.

4.1.2.9 Netzwerkdrosselung - Sitespeed.io Throttle

Das Node.js-Modul Throttle von sitespeed.io wurde ausgewählt, um die Drosselung des Netzwerks auf Paket-Ebene durchzuführen, was von Lighthouse als realistischste Art der Drosselung empfohlen wird [79]. Durch die Verwendung von pfctl (Packet Filter Control) durch Throttle kann die Geschwindigkeit von Netzwerkverbindungen durch das Anwenden von Bandbreitenbeschränkungen und Filterregeln präzise gesteuert werden. So können realistische Netzwerkbedingungen simuliert werden. Die simulierten Netzwerkbedingungen in dieser Arbeit sind langsames 3G (Slow 3G), schnelles 3G (Fast 3G), LTE und WiFi (vgl. Tabelle 4.1), wobei die Verbindung des Messcomputers zum Internet zusätzlich über ein WLAN erfolgte, in dem die gedrosselten Bandbreiten jederzeit verfügbar waren. Als Node.js-Modul eignet sich Throttle zudem für die Integration in eine automatisierte Messanwendung, um die Drosselung des Netzwerks programmatisch zu steuern.

4.2 Implementierung

Die zwei Webanwendungen wurden mit den vier verschiedenen Frameworks in jeweils einem eigenen Projekt umgesetzt. Dazu wurden die Frameworks mittels ihres Command Line Interfaces (CLIs) initialisiert und in ihrer Grundstruktur belassen. Zum Durchführen der Messungen wurden Programme erstellt, die automatisiert die Websites aufrufen und Interaktionen darauf ausführen, um das Websiteverhalten mit Lighthouse zu messen. In diesem Abschnitt werden der Aufbau und die technische Umsetzung der einzelnen Anwendungen beschrieben.

```
...
<body>
  <div class="component-container">
    <h1>Hello World</h1>
    <h2>Welcome to React</h2>
    <button id="klick1">Hide Logo</button>
    <img
      src="/assets/react-CHdo91hT.svg"
      alt="React Logo"
      width="400"
      height="125"
    />
  </div>
</body>
...
```

Listing 4.1: HTML-Ausgabe der minimalistischen Website

4.2.1 Implementierung der minimalistischen Webanwendung

Frameworks haben immer eine bestimmte Menge an Grunddaten, die bei einem Anfordern einer Website vom Server an den Nutzer – neben dem eigentlichen Inhalt der Website – als Antwort mitgeschickt werden. Mit Betrachtung der Core Web Vitals – vor allem des Largest Contentful Paint – wird ersichtlich, dass das Datenvolumen bei einem initialen Seitenaufruf den LCP-Wert deutlich beeinflussen kann. Mit der minimalistischen Website soll geprüft werden, wie ausschlaggebend der Einfluss des reinen Frameworks auf die Leistungsmetriken einer Website ist.

4.2.1.1 Aufbau

Zum Ausführen einer Lighthouse-Messung benötigt die Website mindestens ein HTML-Element, das den Kriterien zur Auswahl eines LCP-Elements entspricht. Deswegen kann eine komplett leere Website – die auf reinen Frameworkdaten basiert – nicht getestet werden. Zudem soll eine minimalistische, aber sinnvolle Website erstellt werden, die als praxisnahes Testfeld für reale Webanwendungen dient.

Die Website (siehe Anhang B, Abbildung 1) enthält zwei Textelemente und einen Button (vgl. Listing 4.1). Bei einem Anklicken des Buttons wird darunter ein Logo des jeweils verwendeten Frameworks angezeigt. Durch den Button wird die Seite interaktiv, wodurch es mit Lighthouse möglich ist die Metrik für Interaction to Next Paint (INP)

zu messen. Das darunter angezeigte Bild führt zu einer Veränderung des Layouts, die den Messwert Cumulative Layout Shift (CLS) beeinflussen kann.

4.2.1.2 Technische Umsetzung

Aufgrund des geringen Umfangs der Website wurde auf zusätzlich erstellt Komponenten verzichtet und die gesamte Logik sowie das Markup wurden mittels JSX in einer Index-Datei beschrieben. Mit dem Button ist es möglich das Logo des Frameworks hinzu- und wieder wegzuschalten. Hierfür wurde eine Zustandsvariable erstellt, in der als Boolescher Wert gespeichert wird, ob die Anzeige des Logos aktiv ist. Zudem verfügt der Button über eine Id, die bei den Page-Speed-Messungen von Puppeteer für die Ausführung der Interaktionen genutzt wird. Das Logo wird über bedingtes Rendern eingefügt. Bei React und Qwik kommt hier der logische Und-Operator zum Einsatz. Solid.js hingegen bietet mit `<Show>` eine eingebaute Komponente für bedingtes Rendern [80]. Im Vergleich zu den Lösungen von Qwik und React ist laut Carniato dieser Ansatz beschreibender und zudem leistungsfähiger, da die in `<Show>` enthaltene Komponente nur neu gerendert wird, wenn sich der betrachtete Boolesche Zustand ändert [81]. Das Styling der Website erfolgt in allen Frameworks mittels global verfügbarem CSS. Da die Website für die Messungen über das Internet erreichbar sein soll, wurden die Anwendungen für Qwik mit SSR[3], Solid.js mit SSR[4], Next.js mit SSR[5], Solid.js mit CSR[6] und React mit CSR[7] über Vercel veröffentlicht.

4.2.2 Implementierung der interaktiven Webanwendung

Diese Webanwendung stellt eine SPA dar, die interaktiv ist und dynamisch Zustandsänderungen auf der Seite aktualisiert. Um eine reale Website abzubilden, sind elementare Bestandteile von Webanwendungen, wie Datenbankzugriffe, Benutzerinteraktionen und Routing enthalten. Da die Anwendung mit vier Frameworks erstellt wurde, werden programmiertechnische Unterschiede und Schwierigkeiten zwischen den Frameworks genauer betrachtet.

[3] `https://qwik-ssr-hello-world.vercel.app/`
[4] `https://solidjs-ssr-hello-world.vercel.app/`
[5] `https://nextjs-ssr-hello-world.vercel.app/`
[6] `https://solidjs-csr-hello-world.vercel.app/`
[7] `https://react-csr-hello-world.vercel.app/`

4.2.2.1 Aufbau

Die SPA enthält einen Header mit dem Logo des Frameworks – mit dem sie erstellt wurde – und mehreren Navigationselementen (siehe Anhang B, Abbildung 2). Die Website wurde als Gästebuch konzipiert, in dem Einträge verfasst werden können und diese in einer Datenbank gespeichert werden. Der Inhalt der Startseite besteht aus einer Komponente zur Eingabe eines Namens, der Gästebuch-Komponente sowie einer Komponente die integriert wurde, um die Verwendung von Bibliotheken auf einer Website zu simulieren – und somit einen Einfluss auf das Datenvolumen der Website nimmt – sowie eine weitere Komponente, die eine Funktion ausführt, die den Main Thread kurze Zeit blockiert. Die 50 neuesten Einträge im Gästebuch werden als Liste auf der Seite angezeigt, mit der Option noch weitere Einträge abzurufen und anzeigen zu lassen. Es ist eine SPA-Navigation zu einem Dashboard enthalten, das die Anzahl der verfassten Gästebucheinträge pro Tag und Framework anzeigt (siehe Anhang B, Abbildung 4). Das Dashboard wurde ausgewählt, da es ein häufiger Anwendungsfall für interaktive Webanwendungen ist.

4.2.2.2 Technische Umsetzung

Während QwikCity, Next.js und SolidStart automatisch mit einem Router initialisiert werden und die Pfade durch das Dateisystem bestimmt sind, musste für React das Routing zusätzlich integriert und die einzelnen Pfade für Startseite und Dashboard definiert werden. Das Layout der Seiten – bestehend aus einem Header und dem Inhalt – wurde bei allen Frameworks entweder im Router oder über eine extra Layout-Datei definiert. Die visuelle Gestaltung erfolgte mittels TailwindCSS, das für alle Frameworks als Node.js-Modul integriert wurde. Für die Komponente mit den zusätzlichen Bibliotheken wurden Lodash und Three.js integriert. Von den Bibliotheken sind Grundfunktionen in der Komponente enthalten, die während der Messungen jedoch nicht ausgeführt werden und einen ungenutzten Teil einer Website darstellen. Auch diese Websites sollten für die Messungen über das Internet erreichbar sein. Deshalb wurden die Anwendungen für Qwik mit SSR[8], Solid.js mit SSR[9], Next.js mit SSR[10], Solid.js mit CSR[11] und React

[8] `https://qwik-interactive.vercel.app/`
[9] `https://solidjs-interactive.vercel.app/`
[10] `https://nextjs-interactive.vercel.app/`
[11] `https://solidjs-csr-interactive.vercel.app/`

mit CSR[12] über Vercel veröffentlicht. Die Datenbank für die Einträge des Gästebuchs wurde ebenfalls über Vercel mit Vercel Postgres angelegt.

Header

Der Header ist ein festes Element am Seitenkopf und beinhaltet Navigationselemente für die Website. Für die Links zu Dashboard und der Startseite stellen die Frameworks Link-Komponenten bereit, die eine SPA-Navigation ermöglichen. Eine Verwendung des normalen `<a>`-Tag würde zu einer HTTP-Anfrage und somit einer Erneuerung des HTML-Dokuments führen. Um eine Nutzung auf mobilen Geräten zu gewährleisten, werden mittels „CSS Media Queries“ bei Unterschreiten einer bestimmten Fensterbreite die Navigationselemente in ein ausklappbares Menü gesetzt, dessen Anzeigestatus durch eine Zustandsvariable gesteuert wird.

Startseite

Die Startseite wurde als Komponente erstellt in der die Komponenten der einzelnen Elemente der Seite aufgerufen werden. Da der Name nach der Eingabe – zum Erstellen einer Session – in einem Cookie gespeichert wird, wird das Cookie in der Startseiten-Komponente abgerufen und als Property an Unterkomponenten weitergegeben. Mit Next.js wird der Name dafür nicht in einer Zustandsvariablen gespeichert, da die Startseiten-Komponente sonst nicht mehr als Server-Komponente gilt. Da bei Next.js Client-Komponenten nicht asynchron sein dürfen und Server-Komponenten hingegen schon, werden hier auch die Gästebucheinträge für das serverseitige Rendern von der Datenbank abgerufen und an die Gästebuch-Komponente übergeben. Bei den weiteren Frameworks findet der Datenbankzugriff erst in der Gästebuch-Komponente selbst statt.

Namenskomponente

Das erste Element auf der Startseite dient der Ausführung einiger Interaktionen und der Namenseingabe für das Gästebuch (siehe Anhang B, Abbildung 3). Durch das Klicken eines Start-Buttons wird ein Element geöffnet, mit dem einige Interaktionen an einem Counter ausgeführt werden müssen, um Zugriff auf das Formular zur Namenseingabe zu

[12] `https://react-interactive.vercel.app/`

erhalten. Damit ist es möglich eine Reihe von Benutzerinteraktionen mit unterschiedlichen Zustandsänderungen während der Lighthouse-Messungen auf der Seite auszuführen und das Verhalten hinsichtlich Interaktion und dem Nachladen von JavaScript-Code zu untersuchen. Nach der Eingabe des Namens und dem Absenden des Webformulars, wird das standardmäßige Neuladen der Website mittels `prevenDefault()` unterbunden und der aus den Formulardaten hervorgehende Name wird in einem Cookie gespeichert. Existiert ein Name im Cookie wird auch über ein erneutes Aufrufen der Website eine Begrüßung angezeigt.

Gästebuch

Das Gästebuch ist aufgeteilt in die Eingabe und die Ausgabe von Einträgen. Die Eingabe eines Beitrags erfolgt durch ein `<input type="text">`-Feld, in das durch Anklicken von Textblöcken der Text eingetragen wird und mit einem Button abgeschickt werden kann. Der Text wird mittels Drizzle ORM in der Datenbank gespeichert. Mit dem erfolgreichen Bestätigen des Datenbankeintrags wird die Liste an Einträgen aktualisiert und so der neue Beitrag angezeigt. Während der Abruf der Einträge aus der Datenbank bei allen Frameworks auf die gleiche Art über Drizzle stattfindet, unterscheidet sich der Ort der Ausführung und die Speicherung der Einträge während der Laufzeit der Website. React führt den Zugriff standardmäßig direkt im Browser aus, was ein großes Sicherheitsproblem durch das Veröffentlichen der Zugangsdaten der Datenbank darstellt. Um dies zu verhindern, müsste ein Backend-Server bereitgestellt werden, auf dem der Datenbankzugriff durchgeführt wird. Da die Webanwendungen und die Datenbank für diese Arbeit nur zu Messzwecken erstellt wurden, wurde auf diesen Backend-Server verzichtet. Bei den weiteren Frameworks ist der Datenbankzugriff über den Server bereits integriert. Mit React und Next.js werden die Einträge mit `useState()` als normaler Zustand gespeichert und die Aktualisierung erfolgt über den `useEffect()`-Hook mit einer asynchronen Funktion (vgl. Listing 4.2). Qwik stellt die `useResource$()`-Funktion zur Verfügung, die asynchron Daten abruft und die ausgeführt wird, wenn sich ein überwachter ausgewählter Zustand ändert. Zudem gibt sie einen Status über die Verfügbarkeit der abgerufenen Daten zurück. Eine darauf abgestimmte `<Resource>`-Komponente steuert wie in Listing 4.3 gezeigt die Darstellung der Daten. Mit Solid.js ist ein ähnliches Vorgehen durch `createResource()` möglich. Der Unterschied zu Qwik liegt darin, dass in Solid.js ein Setter enthalten ist, mit dem die Daten bearbeitet und somit direkt angezeigt werden können. Auch das erneute Abrufen findet nicht durch die Überwachung eines Zustands statt, sondern mit einer `refetch()`-Funktion. Der Status

```
const [entries, setEntries] = useState<GuestbookEntry[]>(initialEntries);
useEffect(() => {
    const fetchEntries = async () => {
        const fetchEntries = await getGuestbookEntries(numberOfEntries);
        setEntries(fetchEntries);
    };
    update && fetchEntries();
}, [numberOfEntries, update]);
```

Listing 4.2: Verwendung des `useEffect()`-Hooks mir Abhängigkeiten für den Abruf von Daten mit Qwik und das Speicher in der Zustandsvariablen

```
const entries = useResource$<GuestbookEntry[]>(async ({ track }) => {
  track(() => [state.refetch]);
  return await getGuestbookEntries(state.numberOfEntries);
});
return (
  <Resource
    value={entries}
    onPending={() => <Loading />}
    onResolved={() => <Entries />}
  />
);
```

Listing 4.3: Verwendung von `useResource$()` für den Abruf von Daten mit Qwik und die Verwendung durch die `<Resource>`-Komponente

```
const [entries, { mutate, refetch }] = createResource(
  () => getGuestbookEntries(numberOfEntries()) as Promise<GuestbookEntry[]>,
  { initialValue: initialEntries() || [] },
);
const listEntries = createMemo(() => {
  if (entries.loading) {
    return [];
  } else return entries.latest.slice(0, numberOfEntries());
});
return <For each={listEntries()} fallback={<Loading />}>...</For>;
```

Listing 4.4: Verwendung von `createResource$()` für den Abruf von Daten mit Solid.js und Memo-Funktion für die Anzeige von Einträgen

über die Verfügbarkeit wird wie auch in Qwik zurückgegeben, kann jedoch ohne eine spezielle Komponente in Funktionen genutzt werden.

Die Anzahl der angezeigten Einträge kann erweitert und auch wieder verringert werden. Dazu werden die Einträge nicht direkt, sondern über reaktive Memo-Funktionen – die nur die sichtbaren Einträge zurückgeben – im JSX aufgerufen. Die Memo-Funktion überwacht die Anzahl der anzuzeigenden Einträge und gibt bei Änderungen die aktualisierte Anzahl an Einträgen zurück (vgl.Listing 4.4). Somit findet nur ein Datenbankzugriff statt, wenn anzuzeigende Einträge noch nicht vorhanden sind.

In den Einträgen wird neben Eintragstext und Name das Datum der Erstellung angezeigt. Hier führte es teilweise zu Problemen während des Hydration-Vorgangs, da sich das mittels `toLocaleDateString()` am Server gerenderte Datumsformat vom Datumsformat unterschied, das während der Hydration im Browser gerendert wurde. Deshalb wurde die akzeptierte Sprache aus dem HTTP-Header verwendet, um das Datum am Server bereits im richtigen Format zu rendern.

Dashboard

Das Dashboard, das über den Pfad „/dashboard“ erreichbar ist, wurde mithilfe der Bibliothek Chart.js, bzw. den Ablegern für React und Solid.js erstellt. Von der Datenbank werden alle Einträge des Gästebuchs abgerufen und die Anzahl der geschriebenen Einträge pro Tag und Framework wird in einem Liniendiagramm angezeigt. Über zwei Datum-Eingabefelder ist es möglich das Start- und Enddatum für die Anzeige festzulegen. Da für Qwik keine Komponentenbibliothek für Chart.js existiert, wurde

ein `<canvas>`-Element als Signal referenziert, in dem das Liniendiagramm erstellt und das bei Änderungen der Zeitspanne aktualisiert wird.

Maßnahmen für Hosting

Außer bei Next.js wurden für das Ausbringen der Websites über Vercel weitere Einstellungen in den Projekten der Frameworks vorgenommen. Für React wurde eine JSON-Datei eingefügt, die für ein funktionierendes Routing benötigt wird, bei Solid.js musste Vercel als Server in der Anwendungskonfiguration eingestellt werden und für Qwik musste ein Vercel Adapter integriert sowie eine Middleware für den Einstieg erstellt werden.

4.2.3 Messanwendungen

Lighthouse-Messungen können manuell über die Chrome DevTools sowie mittels Node CLI oder programmatisch in einer Node.js-Laufzeitumgebung ausgeführt werden. Da in dieser Arbeit eine hohe Anzahl an Messungen zusammen mit der Lighthouse User Flow API nötig war und diese automatisiert durchgeführt werden sollten, wurde die programmatische Variante gewählt.

4.2.3.1 Lighthouse User Flow Messung

Für die Lighthouse-Messungen wurde in einem Node.js-Projekt ein JavaScript-Programm erstellt, das für jede Webanwendung eines jeden Frameworks zehn Messungen mit einer von vier gewählten Netzwerkbedingungen durchführt. Zusätzlich werden die Messungen für den initialen Website-Aufruf – ohne vorhandenem Cache – sowie für ein weiteres Aufrufen der Website – mit vorhandenem Cache – durchgeführt. Mittels der Kommandozeile werden nach Programmaufruf die zu testenden Frameworks und die Netzwerkbedingung mittels Angabe der zu simulierenden Download- und Upload-Geschwindigkeit in Kilobit pro Sekunde sowie der Paketumlaufzeit in Millisekunden ausgewählt, die durch Throttle angewandt wird und bis zum Beenden des Programms besteht. Im nächsten Schritt wird der fensterlose Browser mittels Puppeteer gestartet und eine leere Seite in einem Browsertab geöffnet. Der Browsertab wird der Startfunktion des Lighthouse User Flows als Messumgebung zugewiesen. Die Startfunktion erhält zusätzlich eine spezielle Lighthouse-Konfiguration, um die Standardkonfiguration zu überschreiben.

```
...
await flow.navigate(url);
await flow.startTimespan();
await page.click("#klick1");
const counterUp = await page.waitForSelector("#klick2");
for (let i = 0; i < 6; i++) {
    await counterUp.click();
}
...
await flow.endTimespan();
```

Listing 4.5: Ablauf einer Lighthouse User Flow Navigations- und Zeispannenmessung mit Website-Interaktionen duch Puppeteer

Für die Nutzung der Netzwerkdrosselung mit Throttle wurde die Drosselmethode für das Netzwerk auf `provided` gestellt, was die interne Netzwerksimulation von Lighthouse deaktiviert. Zudem wurde – zur Simulation eines mobilen Geräts – eine Drosselung der CPU angewandt. Die Höhe der CPU-Drosselung wurde unter Berücksichtigung der Rechenleistung des Messcomputers bestimmt [82]. Auch die standardmäßige Leerung des Caches vor einer Lighthouse Navigationsmessung wurde in der Konfiguration – für Messungen mit Cache – deaktiviert.

Nach dem Starten des Lighthouse User Flows wird die Website mittels Puppeteer aufgerufen, um einen Kaltstart des Servers und der Datenbank bei den Messungen zu vermeiden [83]. Qwik-HTML-Dokumente werden fünf Sekunden im Cache gespeichert, weshalb diese Zeit gewartet wird, bis die anschließende Lighthouse-Navigationsmessung für den Website-Aufruf beginnt. Nach Abschluss der Navigationsmessung startet wie in Listing 4.5 gezeigt die Zeitspannenmessung, mit der Benutzerinteraktionen erfasst werden und der INP-Wert ermittelt wird. Hier werden mittels Puppeteer Interaktionen mit den Elementen der Website ausgeführt. Die Elemente werden durch CSS-Selektoren ausgewählt und beispielsweise mit Funktionen wie `click(selector: string)` angeklickt. Führt eine Interaktion zu der Erstellung eines neuen DOM-Elements – mit dem interagiert werden soll – ist es nötig mit der Funktion `waitForSelector(selector: string)` auf das Element zu warten, da eine Interaktion mit einem nicht vorhandenen Element zum Abbruch des Programms führt.

Die Interaktionen simulieren das Verhalten eines Benutzers auf der Website und beinhalten das Klicken von Buttons, Eingaben in ein Textfeld, Absenden von Formularen und eine Navigation über das SPA-Routing. Abschließend wird eine neue Zeitspannenmessung gestartet, um das Verhalten einer langandauernden Aufgabe zu testen.

Diese letzte Zeitspannenmessung wurde zu Testzwecken eingefügt und findet in der Analyse der Ergebnisse keine Verwendung. Alle Messungen eines User Flows werden in einem Bericht zusammengefasst und als HTML sowie JSON abgespeichert. Da für die Bestimmung der Core-Web-Vitals-Werte nur die Performance-Kategorie in Lighthouse benötigt wird, wurden alle weiteren Messkategorien deaktiviert.

4.2.3.2 Messung der Abdeckung

Mit den Puppeteer–Funktionen `startJSCoverage()` und `startCSSCoverage()` wird die Abdeckung gemessen. Dafür wurde eine Anwendung erstellt, die – wie bei der Lighthouse-Messung – einen fensterlosen Browser verwendet, in dem die Website geladen wird und Puppeteer die identischen Benutzerinteraktionen durchführt. Die Coverage wird durch die Textlänge – also die Anzahl der Bytes – der JavaScript- und CSS-Dateien einer Website gemessen. Genutzter Code wird bei der Messung durch Bereiche mit Start- und Endpositionen beschrieben, aus denen sich die Anzahl der genutzten Bytes berechnen lässt.

5 Ergebnisse

In diesem Abschnitt werden die Ergebnisse der Messung der Abdeckung sowie der Core Web Vitals vorgestellt. Wie bereits erwähnt, wird hier sowohl das Szenario mit Cache als auch ohne Cache unterschieden und zusätzlich wurden Messungen in vier verschiedenen Netzwerkgeschwindigkeiten vorgenommen. Messungen, die deutliche Unterschiede aufweisen, werden in Diagrammen dargestellt. Die vollständigen Messergebnisse sind im Anhang A abgebildet.

5.1 Ergebnisse für die minimalistische Website

Abdeckung

Die Abdeckung nach dem Seitenaufruf zeigt deutliche Unterschiede zwischen den einzelnen Frameworks auf. Während die mit Qwik erstellte Website auf eine Gesamtgröße von 3,9 Kilobyte (kB) kommt, von denen 47,26 % ungenutzt bleiben, liegen die Websites, die auf React basieren, deutlich darüber. Die mit React erstellte Website hat eine Größe von 143,6 kB, von denen 72,55 % nicht genutzt werden. Die mit Next.js erstellte Website hat mit 307,3 kB sogar mehr als das doppelte Datenvolumen von React und nutzt 63,56 % davon nicht. Die mit Solid.js erstellten Websites liegen mit 46,7 kB für CSR und 48,7 kB für SSR, wovon 52,48 % und 53,74 % nicht genutzt werden, zwischen der Qwik-Website und den auf React basierenden Websites.

Nach der Nutzung der Website weist Qwik mit 54,6 kB ein höheres Datenvolumen als nach dem Seitenaufruf auf, wovon 57,36 % nicht genutzt wurden. Das Volumen der weiteren Websites veränderte sich nicht, nur die Menge der ungenutzten Bytes verringerte sich, wie in Abbildung 5.1 ersichtlich.

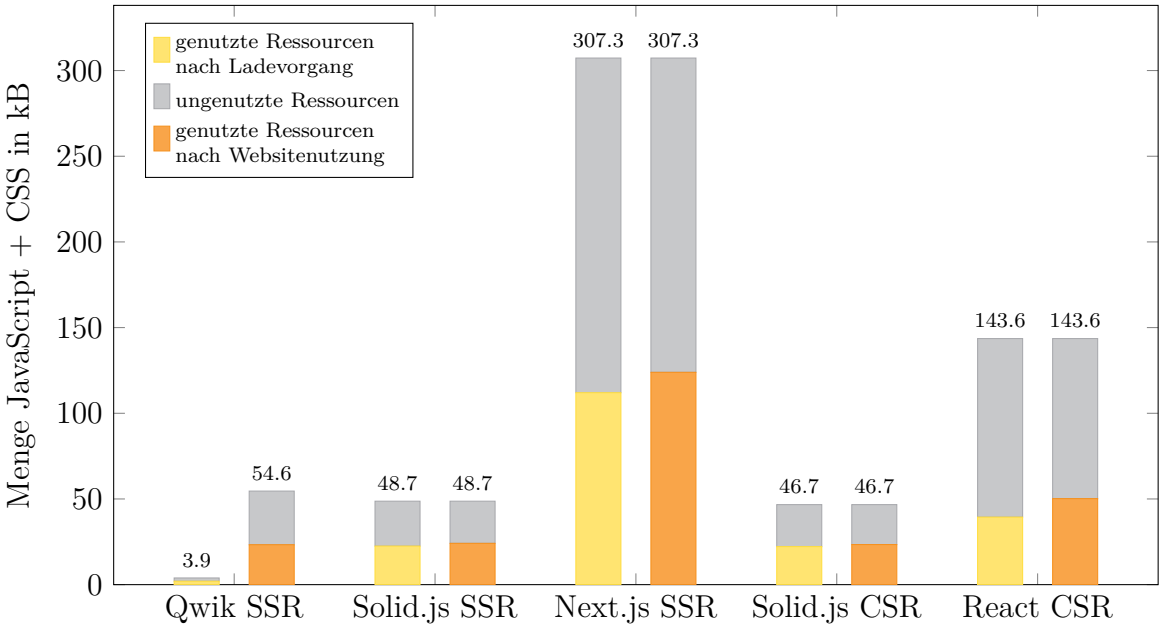

Abbildung 5.1: Messergebnisse der Abdeckung der minimalistischen Website

Largest Contentful Paint

Die Ergebnisse im Szenario der initialen Ladezeit ohne Cache bewegen sich, wie in Abbildung 5.2 zu sehen, für Qwik im Wertebereich zwischen 65,67 ms und 337,8 ms, für Next.js zwischen 195,73 ms und 1364,23 ms, für React zwischen 126,99 ms und 1 729,08 ms, für Solid.js mit SSR im Wertebereich zwischen 233,08 ms und 670,35 ms und für Solid.js mit CSR zwischen 247,4 ms und 1 377,63 ms. Bis auf React erreichen die Frameworks in allen Szenarien den vollen Score. Bei langsamen 3G beträgt der Score für den Wert von React – der höchste ermittelte Wert – 0,99.

Die Ergebnisse im Szenario der Ladezeit mit Cache bewegen sich, wie in Abbildung 5.3 zu sehen, für Qwik im Wertebereich zwischen 90,8 ms und 434,47 ms, für Next.js zwischen 65,08 ms und 277,23 ms, für React zwischen 221,54 ms und 520,36 ms, für Solid.js mit SSR im Wertebereich zwischen 194,37 ms und 425,73 ms und für Solid.js mit CSR zwischen 74,56 ms und 276,81 ms. Jeder dieser Werte entspricht dem höchsten LCP-Score von 1,0.

Interaction to Next Paint

Bei den Messungen der INP-Metrik sind die Werte über alle Szenarien und Frameworks hinweg nahezu identisch. Fast durchgängig wurde ein INP-Wert von 32 ms gemessen.

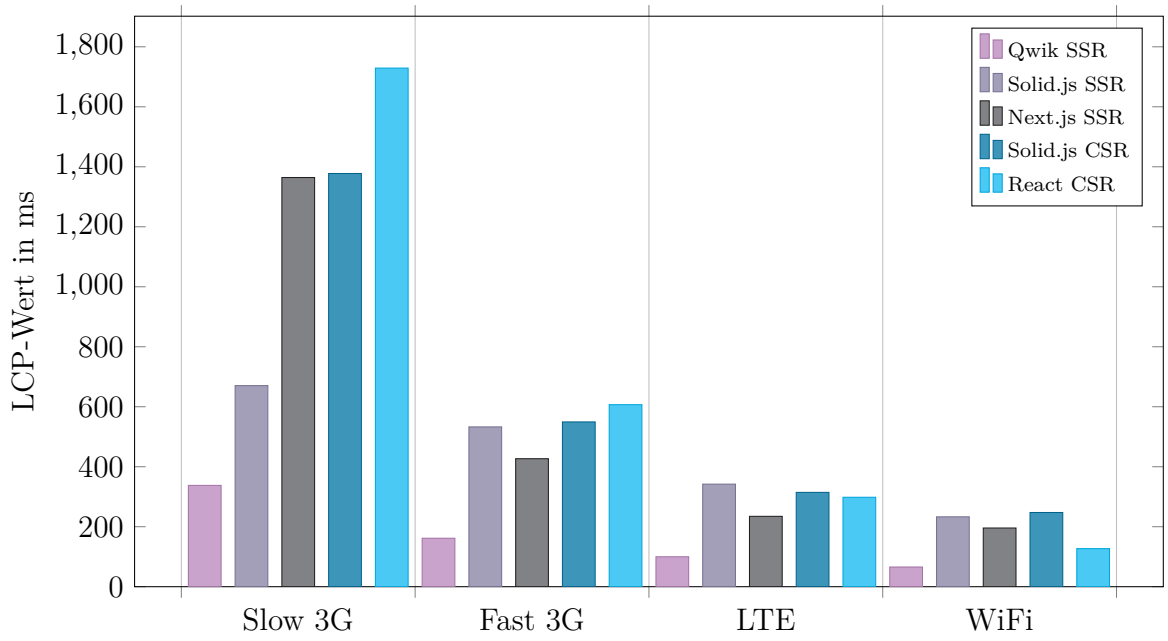

Abbildung 5.2: LCP-Messergebnisse der minimalistischen Website ohne Cache für vier Netzwerkbedingungen

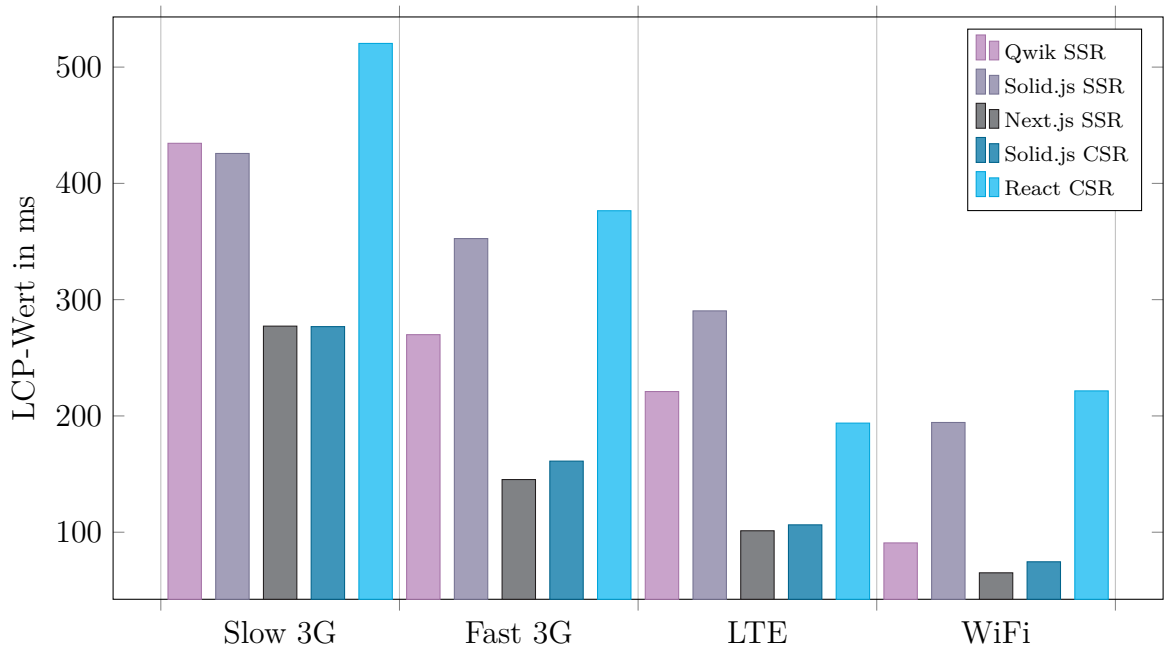

Abbildung 5.3: LCP-Messergebnisse der minimalistischen App mit Cache für vier Netzwerkbedingungen

Qwik erreicht bei zwei Messreihen kleinere Werte von 24 ms und 28 ms und Solid.js mit SSR erreicht einmal einen Median von 28 ms. Alle Frameworks haben hier in allen Szenarien den höchsten Score von 1,0.

Cumulative Layout Shift

Die CLS-Werte für die Frameworks Qwik und Solid.js liegen bei 0,05 in allen Szenarien. Bei Next.js wurde über alle Szenarien hinweg ein CLS-Wert von 0,03 gemessen, bei React ein Wert von 0,06. Bis auf Next.js entsprechen diese Werte nicht dem vollen Score.

5.2 Ergebnisse für die interaktive Website

Abdeckung

Wie auch bei der minimalistischen Webanwendung, sind bei der Abdeckung deutliche Unterschiede zwischen den einzelnen Websites der interaktiven Webanwendung erkennbar (vgl. Abbildung 5.4). Die mit Qwik erstellte Website besitzt nach dem Seitenaufruf das geringste Gesamtvolumen von 15,4 kB, die auf React basierende Website ist mit 1 130,6 kB am größten. Die beiden mit Solid.js erstellten Websites liegen bei Volumen und ungenutzten Bytes nah beieinander und ordnen sich mit ihrem Volumen zwischen Qwik und den auf React basierenden Websites ein. Der ungenutzte Anteil ist bei React mit 69,67 % am höchsten, Next.js und die Solid.js Websites liegen mit je etwa 62 % über Qwik mit 56,21 %.

Nach der Nutzung der Website weist Qwik mit 336,3 kB ein höheres Datenvolumen als nach dem Seitenaufruf auf, jedoch immer noch ein deutlich geringes als die restlichen Websites. Der Anteil der ungenutzen Bytes hat sich um rund 10 Prozentpunkte reduziert. Eine Reduzierung der ungenutzen Bytes ist bei allen Websites erkennbar, sie fällt jedoch geringer aus als bei Qwik. Im Gegensatz zu den minimalistischen Websites steigt bei den mit Solid.js und Next.js erstellten interaktiven Websites das Gesamtvolumen bei der Nutzung an. Bei der mit React erstellten Website reduziert sich nur die Menge der ungenutzten Bytes und das Gesamtvolumen bleibt gleich.

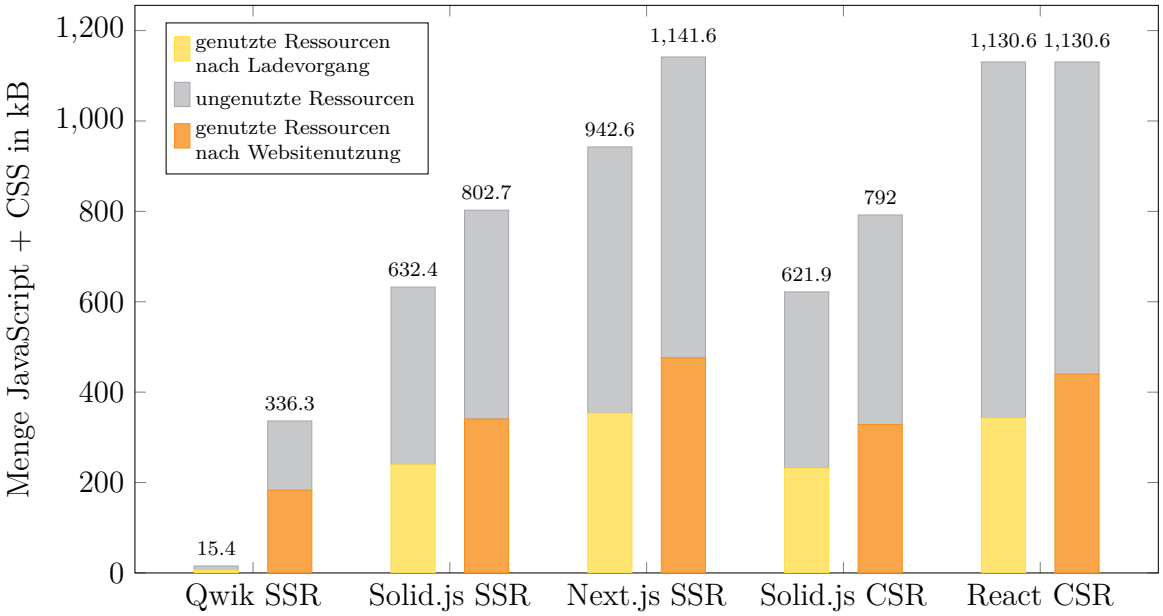

Abbildung 5.4: Messergebnisse der Abdeckung der interaktiven Website

Largest Contentful Paint

Die Ergebnisse im Szenario der initialen Ladezeit ohne Cache bewegen sich, wie in Abbildung 5.5 zu sehen, für Qwik im Wertebereich zwischen 348,87 ms und 419,83 ms, für Next.js zwischen 188,87 ms und 821,86 ms, für React zwischen 192,11 ms und 7 907,73 ms, für Solid.js mit SSR im Wertebereich zwischen 188,97 ms und 1 119,36 ms und für Solid.js mit CSR zwischen 160,19 ms und 2 333,15 ms. Die Frameworks Qwik, Next.js und Solid.js mit SSR erreichen damit in allen Netzwerkbedingungen den höchsten Score von 1,0. React zeigt eine deutliche Zunahme des LCP-Wertes bei langsamen Netzwerkbedingungen und erreicht bei schnellem 3G einen Wert von 2102,95 ms, was einem Score von 0,96 entspricht. Bei langsamen 3G entspricht der Wert von 7 907,73 ms nur noch einem Score von 0,03. Auch Solid.js mit CSR erreicht bei langsamen 3G mit 2 333,15 ms nur einen Score von 0,93, was jedoch noch als guter Wert eingestuft wird.

Die Ergebnisse im Szenario der Ladezeit mit Cache bewegen sich, wie in Abbildung 5.6 zu sehen, für Qwik im Wertebereich zwischen 102,91 ms und 413,69 ms, für Next.js zwischen 218,64 ms und 659,23 ms, für React zwischen 137,29 ms und 562,68 ms, für Solid.js mit SSR im Wertebereich zwischen 181,39 ms und 696,15 ms und für Solid.js mit CSR zwischen 163,84 ms und 511,11 ms. Jeder dieser Werte entspricht dem höchsten LCP-Score von 1,0.

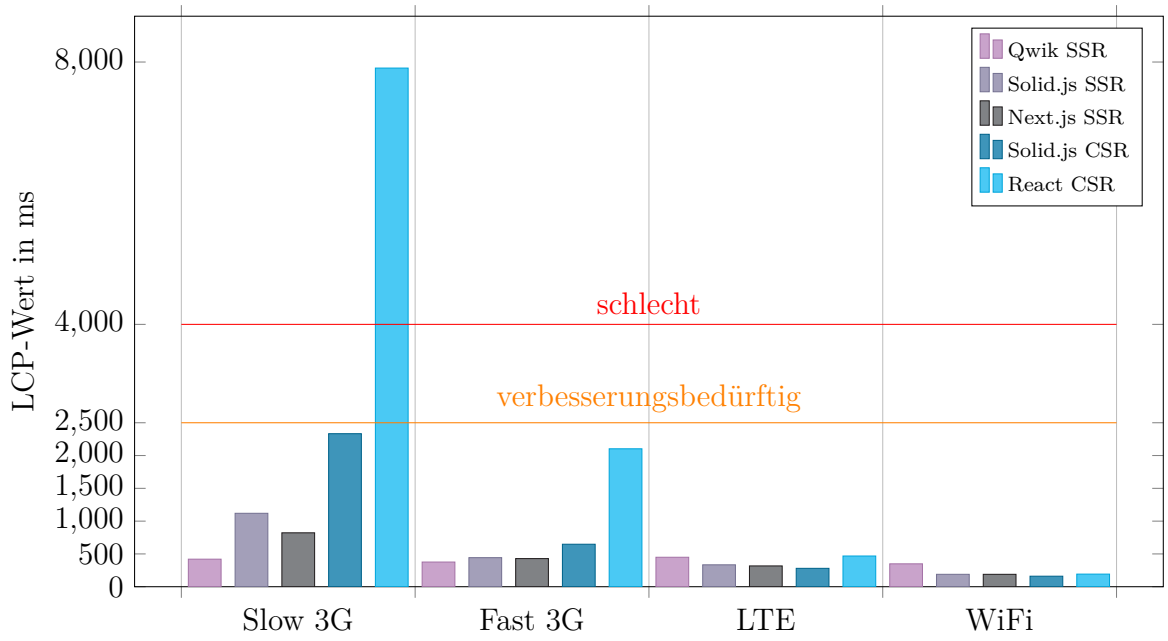

Abbildung 5.5: LCP-Messergebnisse der interaktiven Website ohne Cache für vier Netzwerkbedingungen

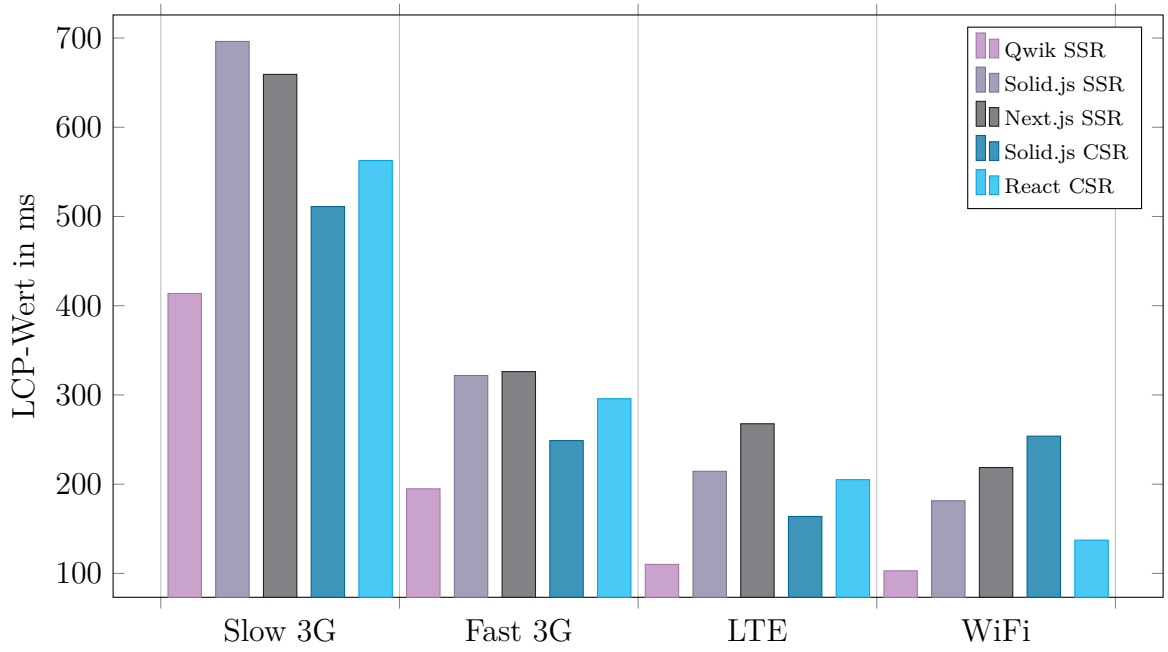

Abbildung 5.6: LCP-Messergebnisse der interaktiven Website mit Cache für vier Netzwerkbedingungen

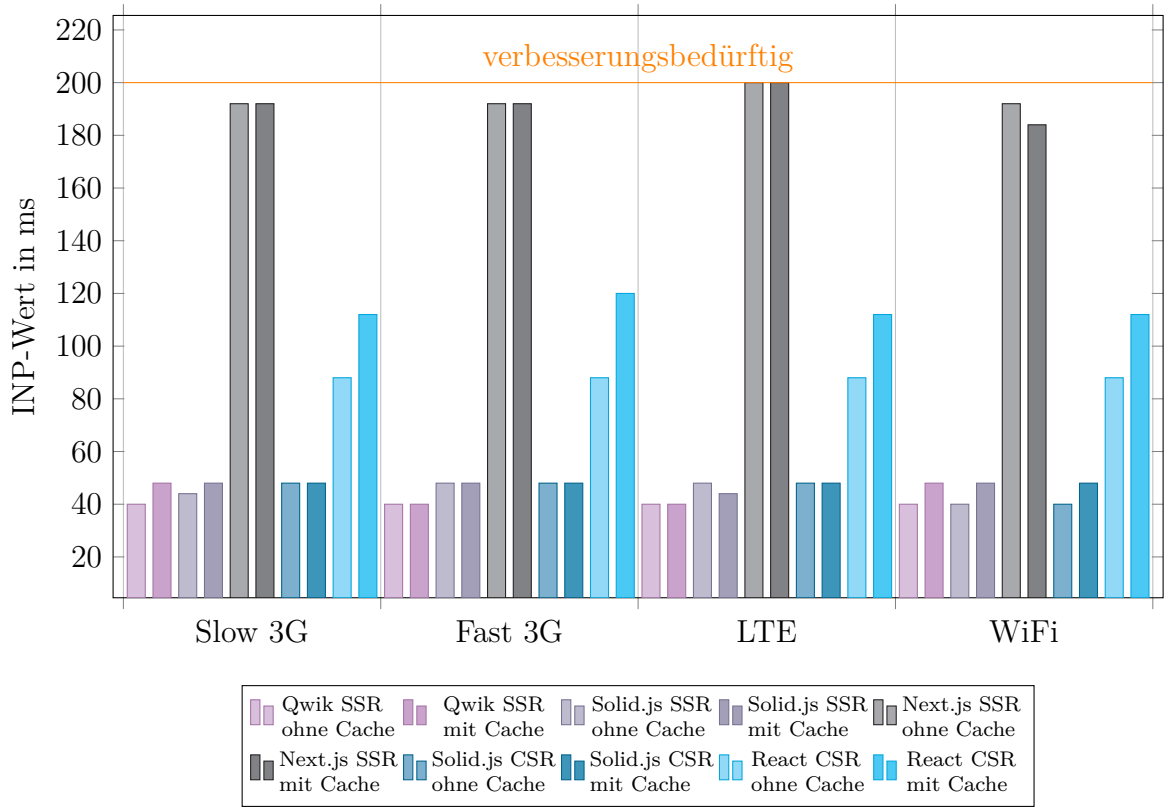

Abbildung 5.7: INP-Messergebnisse der interaktiven Website ohne Cache und mit Cache für vier Netzwerkbedingungen

Interaction to Next Paint

Bei den Messungen der INP-Metrik sind die Werte, wie in Abbildung 5.7 gezeigt, über alle Netzwerkbedingungen innerhalb eines Frameworks hinweg nahezu identisch. Auch der Vergleich zwischen einer ersten Websitenutzung ohne Cache und einer Nutzung mit Cache zeigt bei allen Frameworks außer React kaum unterschiedliche Messwerte. Qwik sowie die beiden Solid.js-Websites mit CSR und SSR erreichen für den INP Werte zwischen 40 ms und 48 ms und damit einen Score von 1,0. Next.js erreicht Werte zwischen 184 ms und 200 ms – somit Scores von 0,90 und 0,91 – und React Werte zwischen 112 ms und 120 ms mit Cache und 88 ms über alle Netzwerkbedingungen hinweg für die Websitenutzung ohne Cache, was Scores von 0,98 und 0,99 entspricht.

Cumulative Layout Shift

Die CLS-Metrik weist über alle Messungen Werte von 0,17 bis 0,24 auf, wobei der Wert 0,17 für alle Frameworks bis auf Qwik, bei allen Netzwerkbedingungen für gecachte

Aufrufe gilt. Der Wert von Qwik liegt mit 0,19 über alle Netzwerkbedingungen hinweg gering darüber. Ohne Cache variiert der Wert für Qwik zwischen 0,21 und 0,24, während er für die anderen Frameworks durchgängig bei 0,22 liegt. Der Score für die Werte liegt im Bereich zwischen 0,52 und 0,71.

6 Diskussion

Mittels Lighthouse wurde die Verwendung eines Mobilgerätes simuliert und die erzielten Ergebnisse liegen größtenteils – vor allem mit guten Netzwerkeigenschaften – im sehr guten Bereich. Ein Aufruf der Websites auf einem Desktopgerät wurde nicht gemessen, da davon ausgegangen werden kann, dass die Bewertungen dort grundsätzlich noch besser ausfallen.

Die Frameworks wurden so verwendet wie sie mittels ihrer CLIs erstellt wurden und es wurden keine expliziten Verbesserungen hinsichtlich des Bündelns des JavaScript-Codes vorgenommen. Gerade bei den auf React basierenden Frameworks wird durch die Abdeckung ersichtlich, dass viel ungenutzter JavaScript-Code nach dem Laden der Website vorhanden ist. In weiteren Arbeiten könnte untersucht werden, ob sich dieser ungenutzte Code reduzieren lässt und welche Auswirkungen dies hätte. Bei der interaktiven Website variiert die Größe zwischen dem Ladevorgang und der Nutzung neben Qwik auch bei Next.js und React. Dies deutet darauf hin, dass Lazy Loading für einige Bestandteile der Website angewandt wird und so der erste Ladevorgang verkürzt wird.

Auswirkung auf den LCP

Qwik sendet nach einem Seitenaufruf wenige Daten, da in erster Linie nur der JavaScript-Code für die globalen Event-Listener notwendig ist. Während der Nutzung wird der Code, der vorher über den Service Worker vorgeladen wurde, abgerufen. Hier zeigt sich auch, dass tatsächlich nur der benötigte Code abgerufen wird, da Qwik nach der Nutzung der interaktiven Website immer noch ein deutlich geringeres Volumen von fast der Hälfte der anderen Websites hat – die nicht genutzten Komponenten der Website machen einen hohen Anteil des Gesamtvolumens aus. Bei der minimalistischen Website, die auf die Erkennung der reinen Frameworkdaten abzielt und bei der die gesamte Website genutzt wurde, ist dieser Unterschied nicht erkennbar. Es zeigt sich somit, dass Resumabiliy einen großen Einfluss auf das JavaScript-Volumen einer Website hat. Auch

liefert Qwik prozentual betrachtet den geringsten Anteil an nicht genutzten Bytes für den Seitenaufruf. Obwohl der LCP maßgeblich von der Ladedauer der Ressource abhängt, lässt sich nicht allgemein feststellen, dass Qwik für den LCP bessere Werte als die anderen Frameworks erzielt. Herauszustellen ist hier jedoch, dass der mit Qwik erreichte LCP durch die geringe Websitegröße selbst bei schlechten Netzwerkbedingungen, wie dem gemessenen langsamen 3G, sehr gute Werte erreicht. Bei Qwik-Websites befindet sich das meiste JavaScript, das beim Laden der Seite benötigt wird, im HTML-Dokument. Da HTML-Dokumente nicht am Client gecacht werden, gibt es für den LCP bei Qwik keine nennenswerten Unterschiede bei den Seitenaufrufen mit Cache zwischen der minimalistischen Website und der größeren interaktiven Website.

Die zwei SSR-Webanwendungen neben Qwik zeigen durch den LCP auch die Nachteile von Hydration im Vergleich zu Resumability. Während das LCP-Element mit Resumability direkt nach seinem Lade- und Rendervorgang den LCP bestimmt, muss bei SSR mit Hydration erst der Hydrationsvorgang abgeschlossen sein, bevor der LCP erreicht wird. Dies lässt sich anhand der Messungen mit Cache und ohne Cache bei der minimalistischen Website sehen. Ohne Cache wurde bei Next.js und Solid.js mit SSR über alle Netzwerkbedingungen ein höherer LCP-Wert gemessen als bei Qwik, da das JavaScript erst vom Server geladen werden musste. Mit Cache erreicht Next.js sogar bessere Werte als Qwik.

Eine positive Auswirkung von SSR im Vergleich zu CSR auf den LCP ist – vor allem bei schlechten Netzwerkbedingungen – bei einem ersten Websiteaufruf erkennbar. Alle Websites mit SSR erzielen hier so gute Messwerte, dass nach den Google-Bewertungskriterien die Nutzererfahrung nicht eingeschränkt ist und auch kein Handlungsbedarf in der Programmierung der Website nötig ist. Dadurch, dass Websites mit CSR erst den gesamten JavaScript-Code für die Erstellung der Website im Browser empfangen müssen, schneiden diese bei schlechten Netzwerkbedingungen und einem ersten Seitenaufruf gerade bei der größeren betrachteten Webanwendung schlechter als Websites mit SSR ab. Vor allem React sticht hier mit besonders schlechten Werten heraus. Mit schnellem 3G ist nur noch die mit React erstellte Website von einem hohen LCP-Wert betroffen und ab LTE gibt es keine Auswirkung mehr auf die Nutzererfahrung. Hier ist es für Entwickler einer Website wichtig, die Zielgruppe – vor allem die Art der verwendeten Geräte und vorhandenen Netzwerkbedingungen – genau zu kennen und abzuwägen, welche Art des Renderns für den bestimmten Einsatzzweck die richtige ist. Da das HTML-Dokument mit SSR bei jeder Anfrage von Server neu gerendert werden muss,

kann dies zu einer längeren Antwortzeit durch den Server führen als bei Websites mit CSR. Ein Einfluss auf den LCP konnte mit den Messungen nicht ermittelt werden.

Nach einem Vergleich der Messwerte zwischen der kleinen Webanwendung und der größeren lässt sich hinsichtlich der Ladegeschwindigkeiten keine eindeutige Aussage zu einem Verhalten bei noch größeren Websites treffen. Eine mögliche Erklärung für die Schwankungen in den Werten könnte darin liegen, dass die gemessenen Zeitwerte für den LCP in einem niedrigen Wertebereich sind. Geringe Netzwerkschwankungen wirken in diesem Fall deshalb vergleichsweise stark. Die gemessenen Werte sind immer noch sehr weit von dem Grenzwert von 2,5 Sekunden entfernt, ab dem eine Website nicht mehr als Website mit guter Nutzererfahrung gesehen wird. Durch die Untersuchung der Werte für eine größere Website – beispielsweise einen Webshop – könnte ermittelt werden, ab wann der LCP einen für die Nutzererfahrung kritischen Bereich erreicht.

Auswirkung auf den INP

Für den INP spielen Technologien eine Rolle, die Einfluss auf die Interaktivität einer Website nehmen. Herausgestellt werden kann hierfür der vDOM und die, wie im Abschnitt über Solid.js gezeigt, feinkörnige Reaktivität. React und Next.js nutzen den vDOM, Qwik verwendet beide Technologien und Solid.js nutzt ausschließlich feinkörnige Reaktivität. Unterschiede der beiden Technologien sind bei der interaktiven Webanwendung zu erkennen. Während für Qwik und Solid.js durchweg ähnlich niedrige INP-Werte gemessen wurden, wies React teilweise eine Interaktionsverzögerung von 0,1 Sekunden auf und Next.js kam mit 0,2 Sekunden sogar an die Grenze zu einer beeinträchtigten Nutzererfahrung. Die hohen Werte bei Next.js fanden in Zusammenhang mit der Aktualisierung von Listenelementen statt und traten laut der Lighthouse-Auswertung wegen der Verarbeitung von JavaScript und nicht wegen der Berechnung von Reflows auf. React rendert bei einer Zustandsänderung die gesamte betroffene Komponente mit allen Unterkomponenten neu, um erkannte Änderung mittels des vDOMs im DOM umzusetzen. Diese Technik zielt zwar darauf ab, hohe Interaktionsverzögerungen durch viele Reflows zu vermeiden, die Messungen zeigen jedoch, dass die Technik auf mobilen Geräten mit schwächeren Rechenleistungen trotzdem problematisch sein kann. Hier können die Berechnungen der Komponenten und die Differenzierung im vDOM zu Verzögerungen führen.

Auswirkung auf den CLS

Der CLS-Wert wird nicht direkt von den aufgezeigten Technologien in dieser Arbeit beeinflusst. Dennoch können bei einem Seitenladevorgang, etwa wenn Elemente bei der Hydration verändert werden, oder während der Nutzung bei hohen Verzögerungen nach einer Interaktion Layoutverschiebungen auftreten, die sich negativ auf die Nutzererfahrung auswirken. Zwar weisen die Messungen der interaktiven Webanwendung CLS-Werte auf, die nicht mehr als „Gut “ einzustufen sind, allerdings zeigt die Lighthouse-Messung nicht, ob die gemessenen Layoutverschiebungen innerhalb des 500 ms-Zeitfensters nach einer Interaktion erfolgten und somit keinen Einfluss auf die Bewertung nehmen. Ein Datenbankzugriff und das asynchrone Laden von Daten kann zu einer Überschreitung des Zeitfensters führen. Da in der erstellten Webanwendung jedoch Platzhalter während des Ladens angezeigt werden und zwischen den verschiedenen Netzwerkbedingungen kein Unterschied für den CLS zu erkennen ist, kann davon ausgegangen werden, dass die gemessenen Layoutverschiebungen nicht zu einer schlechten Bewertung durch Google führen würden. Der Unterschied der Werte zwischen mit Cache und ohne Cache ist darauf zurückzuführen, dass mit Cache das Element zur Begrüßung auf der Seite von Beginn an angezeigt wird. Bei den Messungen ohne Cache wird das Namens-Cookie vor jeder Messung gelöscht und die Eingabe des Namens und das Anzeigen des Begrüßungselements sorgt somit für eine zusätzliche Layoutverschiebung.

7 Zusammenfassung und Ausblick

Ziel dieser Arbeit war es einen Überblick über Technologien in der Webentwicklung mit JavaScript zu schaffen, die hohen Einfluss auf die Nutzererfahrung von Webseiten haben. Als Bewertungsgrundlage für die Nutzererfahrung wurde der Page Speed mittels der Core-Web-Vitals-Metriken herangezogen. Die Metriken bewerten Websites hinsichtlich ihrer Ladegeschwindigkeit, der Interaktivität und der visuellen Stabilität. Die vorgestellten Technologien wurden teilweise gezielt zum Verbessern dieser Bereiche entwickelt und finden in Frontend JavaScript Frameworks Verwendung. Mit React, Next.js, Solid.js und Qwik wurden Frameworks ausgewählt, die sich auf bestimmte Technologien spezialisiert haben. Um einen Vergleich zwischen den Frameworks und somit auch den Technologien herstellen zu können, wurde eine Beispielanwendung erstellt, die einen realen Einsatzzweck einer Webanwendung in Form eines Gästebuchs darstellt. Eine zweite, minimalistische Website wurde erstellt, um die Einflüsse der Frameworks auf Webanwendungen deutlicher betrachten zu können. Für diese zwei Websites wurden Messungen zu den Core Web Vitals sowie der Größe und Codenutzung durchgeführt. Unter einer genauen Betrachtung der Messergebnisse wurden Auffälligkeiten bestimmten Technologien zugeordnet. Die Ergebnisse der Messungen zeigen, dass die neueren Technologien – wie Resumability und feinkörnige Reaktivität – zusammen mit ihren Frameworks vor allem für die Seitenladezeit und die Interaktivität Vorteile bringen können und sich so positiv auf den Page Speed auswirken. Da in der Webentwicklung viele Technologien ineinandergreifen, ist es nur schwer möglich eine Auswirkung für die Websitegeschwindigkeit gezielt einer Technologie zuzuordnen. Außer bei der Betrachtung von SSR und CSR unterscheiden sich die Ergebnisse der Technologien zudem nicht so maßgeblich, dass die Nutzererfahrung davon beeinträchtigt ist.
Die Ergebnisse zur Abdeckung der minimalistischen Website machen deutlich, dass die Datenmenge, die ein Framework standardmäßig an eine Website mit ausliefert, stark variieren kann und dass ein Großteil des JavaScript-Codes ungenutzt bleibt. Für die Entwicklung einer Webanwendung ist es sinnvoll, darauf zu achten, dass das gewählte Framework gut zur Website passt und nicht unnötig überladen ist, sodass die meisten Features tatsächlich genutzt werden. Bis auf React wurden für alle Websites Meta-

frameworks für die Entwicklung verwendet. Die davon bereitgestellten Features wie Routing, Render-Optionen und die Möglichkeiten für Datenabrufe wirkten sich positiv auf die Entwicklererfahrung aus. Auch wenn die verwendeten Frameworks grundsätzlich gut von den Erstellern dokumentiert sind, macht sich bei den neueren Frameworks Qwik und Solid.js die kleinere Community bemerkbar, was die Recherche nach Lösungen für auftretende Probleme während der Entwicklung erschwert. Next.js ermöglicht die Verwendung der React Sever Components, mit denen die Menge an clientseitig benötigtem JavaScript reduziert werden soll, um so die Dauer der Hydration und der Rendervorgänge bei Zustandsänderungen zu verkürzen. Bei der Entwicklung einer Website muss dieses Konzept bewusst eingesetzt werden und es erfordert eine darauf ausgerichtete Strukturierung von Komponenten. Vorteile der React Server Components konnten in dieser Arbeit nicht ermittelt werden, dafür enthielten die Websites zu wenig statische Inhalte. Da das Interesse nach „State of JS“ für Qwik und Solid.js für das Jahr 2023 im Vergleich zum Vorjahr abgenommen hat und die Verbreitung gering ist, bleibt abzuwarten, wie sich diese Frameworks und damit möglicherweise auch die Technologien weiterentwickeln [4].
Die Bewertungsgrundlagen für die Nutzerfreundlichkeit werden ständig von Google weiterentwickelt, weshalb eine kontinuierliche Überprüfung des Page Speeds sinnvoll ist. Auch sollten nach größeren Änderungen an einer Website Tests zum Page Speed durchgeführt werden und es sollte auf die von Lighthouse empfohlenen Verbesserungsmöglichkeiten geachtet werden.

Literatur

[1] *Understanding Google Page Experience | Google Search Central | Documentation*. Google for Developers. URL: https://developers.google.com/search/docs/appearance/page-experience (besucht am 18. 07. 2024).

[2] Robert B. Miller. „Response Time in Man-Computer Conversational Transactions“. In: *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*. AFIPS '68 (Fall, Part I). New York, NY, USA: Association for Computing Machinery, 9. Dez. 1968, S. 267–277. ISBN: 978-1-4503-7899-4. DOI: 10.1145/1476589.1476628.

[3] Amar Sagoo, Annie Sullivan und Vivek Sekhar. *The Science Behind Web Vitals*. Chromium Blog. URL: https://blog.chromium.org/2020/05/the-science-behind-web-vitals.html (besucht am 13. 07. 2024).

[4] *State of JavaScript 2023: Front-end Frameworks*. URL: https://2023.stateofjs.com/en-US/libraries/front-end-frameworks/ (besucht am 13. 07. 2024).

[5] Richard Rabbat und Bryan McQuade. *Introducing Page Speed | Google Search Central Blog*. Google for Developers. URL: https://developers.google.com/search/blog/2009/06/introducing-page-speed (besucht am 30. 06. 2024).

[6] Juho Vepsäläinen, Miško Hevery und Petri Vuorimaa. „Resumability—A New Primitive for Developing Web Applications“. In: *IEEE Access* 12 (2024), S. 9038–9046. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2024.3352891.

[7] *HTTP Archive: Page Weight*. URL: https://httparchive.org/reports/page-weight (besucht am 02. 07. 2024).

[8] Marin Kaluža, Krešimir Troskot und Bernard Vukelić. „COMPARISON OF FRONT-END FRAMEWORKS FOR WEB APPLICATIONS DEVELOPMENT“. In: *Zbornik Veleučilišta u Rijeci* 6.1 (3. Mai 2018), S. 261–282. ISSN: 1848-1299, 1849-1723. DOI: 10.31784/zvr.6.1.19.

[9] V. Solovei, Olga Olshevska und Y. Bortsova. „The Difference between Developing Single Page Application and Traditional Web Application Based on Mechatronics Robot Laboratory Onaft Application“. In: *Automation Technological and Business Processes* 10 (30. März 2018). DOI: 10.15673/atbp.v10i1.874.

[10] *Den kritischen Pfad verstehen | web.dev*. URL: https://web.dev/learn/performance/understanding-the-critical-path?hl=de (besucht am 02. 07. 2024).

[11] *Optimize Resource Loading*. web.dev. URL: https://web.dev/learn/performance/optimize-resource-loading (besucht am 02. 07. 2024).

[12] Ilya Grigorik. *Introducing Web Vitals: Essential Metrics for a Healthy Site*. Chromium Blog. URL: https://blog.chromium.org/2020/05/introducing-web-vitals-essential-metrics.html (besucht am 30. 06. 2024).

[13] Philip Walton. *Zeit bis Interaktivität (TTI) | Articles*. web.dev. URL: https://web.dev/articles/tti?hl=de (besucht am 03. 07. 2024).

[14] Philip Walton und Barry Pollard. *Optimize Largest Contentful Paint | Articles*. web.dev. URL: https://web.dev/articles/optimize-lcp (besucht am 19. 07. 2024).

[15] Philip Walton und Barry Pollard. *Largest Contentful Paint (LCP) | Articles*. web.dev. URL: https://web.dev/articles/lcp (besucht am 19. 07. 2024).

[16] Jeremy Wagner. *Interaction to Next Paint (INP) | Articles | Web.Dev*. URL: https://web.dev/articles/inp (besucht am 19. 07. 2024).

[17] Milica Mihajlija und Philip Walton. *Cumulative Layout Shift (CLS) | Articles*. web.dev. URL: https://web.dev/articles/cls (besucht am 19. 07. 2024).

[18] Philip Walton. *Web Vitals | Articles*. web.dev. URL: https://web.dev/articles/vitals (besucht am 19. 07. 2024).

[19] *CrUX-Methodik | Chrome UX Report*. Chrome for Developers. URL: https://developer.chrome.com/docs/crux/methodology?hl=de (besucht am 30. 06. 2024).

[20] *Tree Shaking - MDN Web Docs Glossary: Definitions of Web-related Terms | MDN*. 8. Juni 2023. URL: https://developer.mozilla.org/en-US/docs/Glossary/Tree_shaking (besucht am 29. 06. 2024).

[21] *Code-Split JavaScript*. web.dev. URL: https://web.dev/learn/performance/code-split-javascript (besucht am 29. 06. 2024).

[22] Ben Livshits und Emre Kiciman. „Doloto: Code Splitting for Network-Bound Web 2.0 Applications“. In: Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering. 9. Nov. 2008. DOI: 10.1145/1453101.1453151.

[23] *Lazy Loading - Web Performance | MDN*. 20. Dez. 2023. URL: https://developer.mozilla.org/en-US/docs/Web/Performance/Lazy_loading (besucht am 29. 06. 2024).

[24] Philippe Le Hégaret, Lauren Wood und Jonathan Robie. *What Is the Document Object Model?* URL: https://www.w3.org/TR/DOM-Level-2-Core/introduction.html (besucht am 16. 06. 2024).

[25] *Document Object Model (DOM) Level 1 Specification*. URL: https://www.w3.org/TR/REC-DOM-Level-1/ (besucht am 17. 06. 2024).

[26] Mark Wilton-Jones. *Dev.Opera — Efficient JavaScript*. URL: https://dev.opera.com/articles/efficient-javascript/#reflow (besucht am 17. 06. 2024).

[27] Artemij Fedosejev. *React.Js Essentials*. Packt Publishing, 2016. 194 S. ISBN: 978-1-78355-162-0.

[28] Ire Aderinokun. *Understanding the Virtual DOM*. bitsofcode. 24. Dez. 2018. URL: https://bitsofco.de (besucht am 19. 06. 2024).

[29] *Reconciliation – React*. URL: https://legacy.reactjs.org/docs/reconciliation.html (besucht am 04. 07. 2024).

[30] Aleksi Huotala, Matti Luukkainen und Tommi Mikkonen. „Benefits and Challenges of Isomorphism in Single-page Applications: Case Study and Review of Gray Literature“. In: *Journal of Web Engineering* (2022), S. 2363–2404. ISSN: 1544-5976. DOI: 10.13052/jwe1540-9589.2186.

[31] Malek Hakim. „Speed Index and Critical Rendering Path for Isomorphic Single Page Apps“. In: *Proceedings of the 16th Winona Computer Science Undergraduate Research Seminar,* 2016, S. 41–46. URL: https://www.academia.edu/24911499/Speed_Index_and_Critical_Rendering_Path_for_Isomorphic_Single_Page_Apps (besucht am 28. 06. 2024).

[32] Miško Hevery. *From Static to Interactive: Why Resumability Is the Best Alternative to Hydration*. Builder.io. URL: https://www.builder.io/blog/from-static-to-interactive-why-resumability-is-the-best-alternative-to-hydration (besucht am 29. 06. 2024).

[33] *Vue.Js*. URL: https://vuejs.org/guide/scaling-up/ssr.html#client-hydration (besucht am 07. 05. 2024).

[34] *hydrateRoot – React*. URL: https://react.dev/reference/react-dom/client/hydrateRoot#hydrating-server-rendered-html (besucht am 07. 05. 2024).

[35] Lydia Hallie und Addy Osmani. *Progressive Hydration*. URL: https://www.patterns.dev/react/progressive-hydration/ (besucht am 03. 07. 2024).

[36] Jason Miller. *Islands Architecture - JASON Format*. URL: https://jasonformat.com/islands-architecture/ (besucht am 03. 07. 2024).

[37] Jatin Ramanathan und Minko Gechev. *Angular and Wiz Are Better Together*. Medium. 29. März 2024. URL: https://blog.angular.io/angular-and-wiz-are-better-together-91e633d8cd5a (besucht am 13. 05. 2024).

[38] Miško Hevery. *Resumability vs Hydration*. Builder.io. URL: https://www.builder.io/blog/resumability-vs-hydration (besucht am 29. 06. 2024).

[39] Miško Hevery. *Hydration Is Pure Overhead*. Builder.io. URL: https://www.builder.io/blog/hydration-is-pure-overhead (besucht am 13. 05. 2024).

[40] Brian Rinaldi. *Why Static Sites?* O'Reilly Media, 2017. ISBN: 978-1-4919-6093-6. URL: https://learning.oreilly.com/library/view/working-with-static/9781491960936/ch01.html (besucht am 20. 07. 2024).

[41] Lydia Hallie und Addy Osmani. *Client-Side Rendering*. URL: https://www.patterns.dev/react/client-side-rendering/ (besucht am 03. 07. 2024).

[42] Alex Banks und Eve Porcello. *Learning React: Modern Patterns for Developing React Apps*. O'Reilly Media, Inc., 12. Juni 2020. 310 S. ISBN: 978-1-4920-5169-5.

[43] *Writing Markup with JSX – React*. URL: https://react.dev/learn/writing-markup-with-jsx (besucht am 04. 07. 2024).

[44] *What Is Babel? · Babel*. URL: https://babeljs.io/docs/ (besucht am 04. 07. 2024).

[45] *useState – React*. URL: https://react.dev/reference/react/useState (besucht am 05. 07. 2024).

[46] *Queueing a Series of State Updates – React*. URL: https://react.dev/learn/queueing-a-series-of-state-updates (besucht am 05. 07. 2024).

[47] Andrew Clark. *GitHub - Acdlite/React-Fiber-Architecture: A Description of React's New Core Algorithm, React Fiber*. GitHub. URL: https://github.com/acdlite/react-fiber-architecture (besucht am 04. 07. 2024).

[48] Karthik Kalyanaraman. *A Deep Dive into React Fiber*. LogRocket Blog. 14. März 2022. URL: https://blog.logrocket.com/deep-dive-react-fiber/ (besucht am 04. 07. 2024).

[49] *State of JavaScript 2023: Meta-Frameworks*. URL: https://2023.stateofjs.com/en-US/libraries/meta-frameworks/ (besucht am 05. 07. 2024).

[50] *Docs | Next.Js*. URL: https://nextjs.org/docs (besucht am 05. 07. 2024).

[51] *React Server Components – React*. URL: https://react.dev/reference/rsc/server-components (besucht am 05. 07. 2024).

[52] *Making Sense of React Server Components*. URL: https://www.joshwcomeau.com/react/server-components/ (besucht am 05. 07. 2024).

[53] *Rendering: Server Components | Next.Js*. URL: https://nextjs.org/docs/app/building-your-application/rendering/server-components (besucht am 05. 07. 2024).

[54] *Fine-Grained Reactivity - SolidDocs*. URL: https://docs.solidjs.com/advanced-concepts/fine-grained-reactivity (besucht am 24. 06. 2024).

[55] *Basics - SolidDocs*. URL: https://docs.solidjs.com/concepts/components/basics (besucht am 22. 06. 2024).

[56] Andreas Roth. *Eine Einführung in SolidJS*. Host Europe Blog. 12. Aug. 2022. URL: https://www.hosteurope.de/blog/eine-einfuehrung-in-solidjs-schluss-mit-framework-overhead/ (besucht am 22. 06. 2024).

[57] *Solid Docs*. URL: https://docs.solidjs.com/solid-start (besucht am 23. 06. 2024).

[58] Ryan Carniato und Saraf Nikhil. *What Is Vinxi, and How Does It Compare to Vike?* DEV Community. 5. März 2024. URL: https://dev.to/this-is-learning/what-is-vinxi-and-how-does-it-compare-to-vike-4883 (besucht am 23. 06. 2024).

[59] ryan_solid. *What's the Difference between Async SSR and Hybrid SSR?* r/solidjs. 10. Feb. 2022. URL: www.reddit.com/r/solidjs/comments/sp9tgj/whats_the_difference_between_async_ssr_and_hybrid/ (besucht am 23. 06. 2024).

[60] *Optimizer Rules | Advanced Qwik Documentation*. Qwik. URL: https://qwik.dev/docs/advanced/optimizer/ (besucht am 25. 06. 2024).

[61] Juho Vepsäläinen, Arto Hellas und Petri Vuorimaa. „The Rise of Disappearing Frameworks in Web Development“. In: *Web Engineering*. Hrsg. von Irene Garrigós, Juan Manuel Murillo Rodríguez und Manuel Wimmer. Cham: Springer Nature Switzerland, 2023, S. 319–326. ISBN: 978-3-031-34444-2. DOI: 10.1007/978-3-031-34444-2_23.

[62] *QRL | Advanced Qwik Documentation*. Qwik. URL: https://qwik.dev/docs/advanced/qrl/ (besucht am 25. 06. 2024).

[63] *Qwikloader | Advanced Qwik Documentation*. Qwik. URL: https://qwik.dev/docs/advanced/qwikloader/ (besucht am 25. 06. 2024).

[64] *Overview | Components Qwik Documentation*. Qwik. URL: https://qwik.dev/docs/components/overview/ (besucht am 24. 06. 2024).

[65] *Frequently Asked Questions | Introduction Qwik Documentation*. Qwik. URL: https://qwik.dev/docs/faq/ (besucht am 26. 06. 2024).

[66] *Reactivity | Concepts Qwik Documentation*. Qwik. URL: https://qwik.dev/docs/concepts/reactivity/ (besucht am 26. 06. 2024).

[67] *Overview | Qwik City Qwik Documentation*. Qwik. URL: https://qwik.dev/docs/qwikcity/ (besucht am 26. 06. 2024).

[68] *Routing | Qwik City Qwik Documentation*. Qwik. URL: https://qwik.dev/docs/routing/ (besucht am 26. 06. 2024).

[69] Jonathan Chen. *Service Worker Caching and HTTP Caching | Articles*. web.dev. URL: https://web.dev/articles/service-worker-caching-and-http-caching (besucht am 27. 06. 2024).

[70] *Speculative Module Fetching | Advanced Qwik Documentation*. Qwik. URL: https://qwik.dev/docs/advanced/speculative-module-fetching/ (besucht am 26. 06. 2024).

[71] Adam Lipiński und Beata Pańczyk. „Performance Optimization of Web Applications Using Qwik“. In: *Journal of Computer Sciences Institute* 28 (30. Sep. 2023), S. 197–203. ISSN: 2544-0764. DOI: 10.35784/jcsi.3672.

[72] Yingying Chen u. a. „A Provider-Side View of Web Search Response Time“. In: *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*. SIGCOMM '13. New York, NY, USA: Association for Computing Machinery, 27. Aug. 2013, S. 243–254. ISBN: 978-1-4503-2056-6. DOI: 10.1145/2486001.2486035.

[73] Patrick Hulce. *LH Variability and Accuracy Analysis*. Google Docs. URL: https://docs.google.com/document/d/1BqtL-nG53rxWOI5ROOpItSRPowZVnYJ_gBEQCJ5EeUE/edit?usp=embed_facebook (besucht am 07. 07. 2024).

[74] Fernando Cristiani und Peter Thiemann. „Generation of TypeScript Declaration Files from JavaScript Code“. In: *Proceedings of the 18th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*. MPLR 2021. New York, NY, USA: Association for Computing Machinery, 29. Sep. 2021, S. 97–112. ISBN: 978-1-4503-8675-3. DOI: 10.1145/3475738.3480941.

[75] *Prisma Schema Overview | Prisma Documentation*. URL: https://www.prisma.io/docs/orm/prisma-schema/overview (besucht am 07. 07. 2024).

[76] *Utility-First Fundamentals - Tailwind CSS*. URL: https://tailwindcss.com/docs/utility-first (besucht am 07. 07. 2024).

[77] Brendan Kenny. *Lighthouse User Flows | Articles*. web.dev. URL: https://web.dev/articles/lighthouse-user-flows (besucht am 06. 07. 2024).

[78] Kondratiuk Dario. *UI Testing with Puppeteer : Implement End-to-End Testing and Browser Automation Using JavaScript and Node.Js*. Packt Publishing, 2021. ISBN: 978-1-80020-678-6. URL: https://search.ebscohost.com/login.aspx?direct=true&db=nlebk&AN=2876277&site=ehost-live.

[79] Adam Raine. *Lighthouse/Docs/Throttling.Md at Main · GoogleChrome/Lighthouse*. GitHub. URL: https://github.com/GoogleChrome/lighthouse/blob/main/docs/throttling.md (besucht am 08. 07. 2024).

[80] *Conditional Rendering - SolidDocs*. URL: https://docs.solidjs.com/concepts/control-flow/conditional-rendering (besucht am 21. 06. 2024).

[81] Ryan Carniato. *Learn Show – Reactivity with SolidJS*. URL: https://frontendmasters.com/courses/reactivity-solidjs/show/ (besucht am 21. 06. 2024).

[82] *Lighthouse CPU Throttling Calculator*. URL: https://lighthouse-cpu-throttling-calculator.vercel.app/ (besucht am 11. 07. 2024).

[83] *Vercel Postgres Limits*. URL: https://vercel.com/docs/storage/vercel-postgres/limits (besucht am 11. 07. 2024).

Anhang A

Messergebnisse minimalistische Website

Framework	**Gesamt JS + CSS in Bytes**	**Ungenutztes JS + CSS in Bytes**	**Ungenutzter Anteil**
Qwik SSR	3.936	1.860	47,26 %
Solid.js SSR	48.714	26.177	53,74 %
Next.js SSR	307.333	195.347	63,56 %
Solid.js CSR	46.689	24.502	52,48 %
React CSR	143.647	104.210	72,55 %

Tabelle 1: Messergebnisse der Abdeckung für die minimalistische Website nach dem Seitenaufruf

Framework	**Gesamt JS + CSS in Bytes**	**Ungenutztes JS + CSS in Bytes**	**Ungenutzter Anteil**
Qwik SSR	54.648	31.348	57,36 %
Solid.js SSR	48.714	24.623	50,55 %
Next.js SSR	307.333	183.350	59,66 %
Solid.js CSR	46.689	23.257	49,81 %
React CSR	143.647	93.441	65,05 %

Tabelle 2: Messergebnisse der Abdeckung für die minimalistische Website nach der Websitenutzung

Framework	**Netzwerkbedingung**	**Cache**	**Median LCP in ms**	**Score**
Qwik SSR	Slow 3G	Mit Cache	434,47	1,0
Qwik SSR	Slow 3G	Ohne Cache	337,8	1,0
Qwik SSR	Fast 3G	Mit Cache	269,89	1,0
Qwik SSR	Fast 3G	Ohne Cache	161,62	1,0
Qwik SSR	LTE	Mit Cache	220,98	1,0
Qwik SSR	LTE	Ohne Cache	99,8	1,0
Qwik SSR	WiFi	Mit Cache	90,8	1,0
Qwik SSR	WiFi	Ohne Cache	65,67	1,0
Solid.js SSR	Slow 3G	Mit Cache	425,73	1,0
Solid.js SSR	Slow 3G	Ohne Cache	670,35	1,0
Solid.js SSR	Fast 3G	Mit Cache	352,54	1,0
Solid.js SSR	Fast 3G	Ohne Cache	532,79	1,0
Solid.js SSR	LTE	Mit Cache	290,35	1,0
Solid.js SSR	LTE	Ohne Cache	342,0	1,0
Solid.js SSR	WiFi	Mit Cache	194,37	1,0
Solid.js SSR	WiFi	Ohne Cache	233,08	1,0
Next.js SSR	Slow 3G	Mit Cache	277,23	1,0
Next.js SSR	Slow 3G	Ohne Cache	1364,23	1,0
Next.js SSR	Fast 3G	Mit Cache	145,29	1,0
Next.js SSR	Fast 3G	Ohne Cache	426,74	1,0
Next.js SSR	LTE	Mit Cache	101,24	1,0
Next.js SSR	LTE	Ohne Cache	234,63	1,0
Next.js SSR	WiFi	Mit Cache	65,08	1,0
Next.js SSR	WiFi	Ohne Cache	195,73	1,0
Solid.js CSR	Slow 3G	Mit Cache	276,81	1,0
Solid.js CSR	Slow 3G	Ohne Cache	1377,63	1,0
Solid.js CSR	Fast 3G	Mit Cache	161,14	1,0
Solid.js CSR	Fast 3G	Ohne Cache	549,35	1,0
Solid.js CSR	LTE	Mit Cache	106,33	1,0
Solid.js CSR	LTE	Ohne Cache	314,49	1,0
Solid.js CSR	WiFi	Mit Cache	74,56	1,0
Solid.js CSR	WiFi	Ohne Cache	247,4	1,0
React CSR	Slow 3G	Mit Cache	520,36	1,0
React CSR	Slow 3G	Ohne Cache	1729,08	0,99
React CSR	Fast 3G	Mit Cache	376,47	1,0
React CSR	Fast 3G	Ohne Cache	606,86	1,0
React CSR	LTE	Mit Cache	193,83	1,0
React CSR	LTE	Ohne Cache	298,18	1,0
React CSR	WiFi	Mit Cache	221,54	1,0
React CSR	WiFi	Ohne Cache	126,99	1,0

Tabelle 3: LCP-Messergebnisse der minimalistischen Website

Framework	**Netzwerkbedingung**	**Cache**	**Median INP in ms**	**Score**
Qwik SSR	Slow 3G	Mit Cache	24,0	1,0
Qwik SSR	Slow 3G	Ohne Cache	32,0	1,0
Qwik SSR	Fast 3G	Mit Cache	32,0	1,0
Qwik SSR	Fast 3G	Ohne Cache	32,0	1,0
Qwik SSR	LTE	Mit Cache	32,0	1,0
Qwik SSR	LTE	Ohne Cache	32,0	1,0
Qwik SSR	WiFi	Mit Cache	32,0	1,0
Qwik SSR	WiFi	Ohne Cache	28,0	1,0
Solid.js SSR	Slow 3G	Mit Cache	32,0	1,0
Solid.js SSR	Slow 3G	Ohne Cache	32,0	1,0
Solid.js SSR	Fast 3G	Mit Cache	28,0	1,0
Solid.js SSR	Fast 3G	Ohne Cache	32,0	1,0
Solid.js SSR	LTE	Mit Cache	32,0	1,0
Solid.js SSR	LTE	Ohne Cache	32,0	1,0
Solid.js SSR	WiFi	Mit Cache	32,0	1,0
Solid.js SSR	WiFi	Ohne Cache	32,0	1,0
Next.js SSR	Slow 3G	Mit Cache	32,0	1,0
Next.js SSR	Slow 3G	Ohne Cache	32,0	1,0
Next.js SSR	Fast 3G	Mit Cache	32,0	1,0
Next.js SSR	Fast 3G	Ohne Cache	32,0	1,0
Next.js SSR	LTE	Mit Cache	32,0	1,0
Next.js SSR	LTE	Ohne Cache	32,0	1,0
Next.js SSR	WiFi	Mit Cache	32,0	1,0
Next.js SSR	WiFi	Ohne Cache	32,0	1,0
Solid.js CSR	Slow 3G	Mit Cache	32,0	1,0
Solid.js CSR	Slow 3G	Ohne Cache	32,0	1,0
Solid.js CSR	Fast 3G	Mit Cache	32,0	1,0
Solid.js CSR	Fast 3G	Ohne Cache	32,0	1,0
Solid.js CSR	LTE	Mit Cache	32,0	1,0
Solid.js CSR	LTE	Ohne Cache	32,0	1,0
Solid.js CSR	WiFi	Mit Cache	32,0	1,0
Solid.js CSR	WiFi	Ohne Cache	32,0	1,0
React CSR	Slow 3G	Mit Cache	32,0	1,0
React CSR	Slow 3G	Ohne Cache	32,0	1,0
React CSR	Fast 3G	Mit Cache	32,0	1,0
React CSR	Fast 3G	Ohne Cache	32,0	1,0
React CSR	LTE	Mit Cache	32,0	1,0
React CSR	LTE	Ohne Cache	32,0	1,0
React CSR	WiFi	Mit Cache	32,0	1,0
React CSR	WiFi	Ohne Cache	32,0	1,0

Tabelle 4: INP-Messergebnisse der minimalistischen Website

Framework	**Netzwerkbedingung**	**Cache**	**Median CLS**	**Score**
Qwik SSR	Slow 3G	Mit Cache	0,05	0,99
Qwik SSR	Slow 3G	Ohne Cache	0,05	0,99
Qwik SSR	Fast 3G	Mit Cache	0,05	0,99
Qwik SSR	Fast 3G	Ohne Cache	0,05	0,99
Qwik SSR	LTE	Mit Cache	0,05	0,99
Qwik SSR	LTE	Ohne Cache	0,05	0,99
Qwik SSR	WiFi	Mit Cache	0,05	0,99
Qwik SSR	WiFi	Ohne Cache	0,05	0,99
Solid.js SSR	Slow 3G	Mit Cache	0,05	0,99
Solid.js SSR	Slow 3G	Ohne Cache	0,05	0,99
Solid.js SSR	Fast 3G	Mit Cache	0,05	0,99
Solid.js SSR	Fast 3G	Ohne Cache	0,05	0,99
Solid.js SSR	LTE	Mit Cache	0,05	0,99
Solid.js SSR	LTE	Ohne Cache	0,05	0,99
Solid.js SSR	WiFi	Mit Cache	0,05	0,99
Solid.js SSR	WiFi	Ohne Cache	0,05	0,99
Next.js SSR	Slow 3G	Mit Cache	0,03	1,0
Next.js SSR	Slow 3G	Ohne Cache	0,03	1,0
Next.js SSR	Fast 3G	Mit Cache	0,03	1,0
Next.js SSR	Fast 3G	Ohne Cache	0,03	1,0
Next.js SSR	LTE	Mit Cache	0,03	1,0
Next.js SSR	LTE	Ohne Cache	0,03	1,0
Next.js SSR	WiFi	Mit Cache	0,03	1,0
Next.js SSR	WiFi	Ohne Cache	0,03	1,0
Solid.js CSR	Slow 3G	Mit Cache	0,05	0,99
Solid.js CSR	Slow 3G	Ohne Cache	0,05	0,99
Solid.js CSR	Fast 3G	Mit Cache	0,05	0,99
Solid.js CSR	Fast 3G	Ohne Cache	0,05	0,99
Solid.js CSR	LTE	Mit Cache	0,05	0,99
Solid.js CSR	LTE	Ohne Cache	0,05	0,99
Solid.js CSR	WiFi	Mit Cache	0,05	0,99
Solid.js CSR	WiFi	Ohne Cache	0,05	0,99
React CSR	Slow 3G	Mit Cache	0,06	0,98
React CSR	Slow 3G	Ohne Cache	0,06	0,98
React CSR	Fast 3G	Mit Cache	0,06	0,98
React CSR	Fast 3G	Ohne Cache	0,06	0,98
React CSR	LTE	Mit Cache	0,06	0,98
React CSR	LTE	Ohne Cache	0,06	0,98
React CSR	WiFi	Mit Cache	0,06	0,98
React CSR	WiFi	Ohne Cache	0,06	0,98

Tabelle 5: CLS-Messergebnisse der minimalistischen Website

Messergebnisse interaktive Website

Framework	**Gesamt JS + CSS in Bytes**	**Ungenutztes JS + CSS in Bytes**	**Ungenutzter Anteil**
Qwik SSR	15.436	8.676	56,21 %
Solid.js SSR	632.417	393.085	62,16 %
Next.js SSR	942.613	589.416	62,53 %
Solid.js CSR	621.854	390.107	62,73 %
React CSR	1.130.614	787.717	69,67 %

Tabelle 6: Messergebnisse der Abdeckung für die interaktive Website nach dem Seitenaufruf

Framework	**Gesamt JS + CSS in Bytes**	**Ungenutztes JS + CSS in Bytes**	**Ungenutzter Anteil**
Qwik SSR	336.269	153.423	45,63 %
Solid.js SSR	802.702	462.334	57,60 %
Next.js SSR	1.141.617	666.180	58,35 %
Solid.js CSR	792.010	463.760	58,55 %
React CSR	1.130.614	691.036	61,12 %

Tabelle 7: Messergebnisse der Abdeckung für die interaktive Website nach der Websitenutzung

Framework	**Netzwerkbedingung**	**Cache**	**Median LCP in ms**	**Score**
Qwik SSR	Slow 3G	Mit Cache	413,69	1,0
Qwik SSR	Slow 3G	Ohne Cache	419,83	1,0
Qwik SSR	Fast 3G	Mit Cache	194,78	1,0
Qwik SSR	Fast 3G	Ohne Cache	376,18	1,0
Qwik SSR	LTE	Mit Cache	110,15	1,0
Qwik SSR	LTE	Ohne Cache	449,08	1,0
Qwik SSR	WiFi	Mit Cache	102,91	1,0
Qwik SSR	WiFi	Ohne Cache	348,87	1,0
Solid.js SSR	Slow 3G	Mit Cache	696,15	1,0
Solid.js SSR	Slow 3G	Ohne Cache	1119,36	1,0
Solid.js SSR	Fast 3G	Mit Cache	321,76	1,0
Solid.js SSR	Fast 3G	Ohne Cache	443,03	1,0
Solid.js SSR	LTE	Mit Cache	214,45	1,0
Solid.js SSR	LTE	Ohne Cache	333,89	1,0
Solid.js SSR	WiFi	Mit Cache	181,39	1,0
Solid.js SSR	WiFi	Ohne Cache	188,97	1,0
Next.js SSR	Slow 3G	Mit Cache	659,23	1,0
Next.js SSR	Slow 3G	Ohne Cache	821,86	1,0
Next.js SSR	Fast 3G	Mit Cache	326,15	1,0
Next.js SSR	Fast 3G	Ohne Cache	428,95	1,0
Next.js SSR	LTE	Mit Cache	267,72	1,0
Next.js SSR	LTE	Ohne Cache	316,96	1,0
Next.js SSR	WiFi	Mit Cache	218,64	1,0
Next.js SSR	WiFi	Ohne Cache	188,87	1,0
Solid.js CSR	Slow 3G	Mit Cache	511,11	1,0
Solid.js CSR	Slow 3G	Ohne Cache	2333,15	0,93
Solid.js CSR	Fast 3G	Mit Cache	248,83	1,0
Solid.js CSR	Fast 3G	Ohne Cache	647,65	1,0
Solid.js CSR	LTE	Mit Cache	163,84	1,0
Solid.js CSR	LTE	Ohne Cache	280,09	1,0
Solid.js CSR	WiFi	Mit Cache	253,81	1,0
Solid.js CSR	WiFi	Ohne Cache	160,19	1,0
React CSR	Slow 3G	Mit Cache	562,68	1,0
React CSR	Slow 3G	Ohne Cache	7907,73	0,03
React CSR	Fast 3G	Mit Cache	295,77	1,0
React CSR	Fast 3G	Ohne Cache	2102,95	0,96
React CSR	LTE	Mit Cache	205,04	1,0
React CSR	LTE	Ohne Cache	468,21	1,0
React CSR	WiFi	Mit Cache	137,29	1,0
React CSR	WiFi	Ohne Cache	192,11	1,0

Tabelle 8: LCP-Messergebnisse der interaktiven Website

Framework	Netzwerkbedingung	Cache	Median INP in ms	Score
Qwik SSR	Slow 3G	Mit Cache	48,0	1,0
Qwik SSR	Slow 3G	Ohne Cache	40,0	1,0
Qwik SSR	Fast 3G	Mit Cache	40,0	1,0
Qwik SSR	Fast 3G	Ohne Cache	40,0	1,0
Qwik SSR	LTE	Mit Cache	40,0	1,0
Qwik SSR	LTE	Ohne Cache	40,0	1,0
Qwik SSR	WiFi	Mit Cache	48,0	1,0
Qwik SSR	WiFi	Ohne Cache	40,0	1,0
Solid.js SSR	Slow 3G	Mit Cache	48,0	1,0
Solid.js SSR	Slow 3G	Ohne Cache	44,0	1,0
Solid.js SSR	Fast 3G	Mit Cache	48,0	1,0
Solid.js SSR	Fast 3G	Ohne Cache	48,0	1,0
Solid.js SSR	LTE	Mit Cache	48,0	1,0
Solid.js SSR	LTE	Ohne Cache	44,0	1,0
Solid.js SSR	WiFi	Mit Cache	48,0	1,0
Solid.js SSR	WiFi	Ohne Cache	40,0	1,0
Next.js SSR	Slow 3G	Mit Cache	192,0	0,91
Next.js SSR	Slow 3G	Ohne Cache	192,0	0,91
Next.js SSR	Fast 3G	Mit Cache	192,0	0,91
Next.js SSR	Fast 3G	Ohne Cache	192,0	0,91
Next.js SSR	LTE	Mit Cache	200,0	0,9
Next.js SSR	LTE	Ohne Cache	200,0	0,9
Next.js SSR	WiFi	Mit Cache	184,0	0,91
Next.js SSR	WiFi	Ohne Cache	192,0	0,91
Solid.js CSR	Slow 3G	Mit Cache	48,0	1,0
Solid.js CSR	Slow 3G	Ohne Cache	48,0	1,0
Solid.js CSR	Fast 3G	Mit Cache	48,0	1,0
Solid.js CSR	Fast 3G	Ohne Cache	48,0	1,0
Solid.js CSR	LTE	Mit Cache	48,0	1,0
Solid.js CSR	LTE	Ohne Cache	48,0	1,0
Solid.js CSR	WiFi	Mit Cache	48,0	1,0
Solid.js CSR	WiFi	Ohne Cache	40,0	1,0
React CSR	Slow 3G	Mit Cache	112,0	0,98
React CSR	Slow 3G	Ohne Cache	88,0	0,99
React CSR	Fast 3G	Mit Cache	120,0	0,98
React CSR	Fast 3G	Ohne Cache	88,0	0,99
React CSR	LTE	Mit Cache	112,0	0,98
React CSR	LTE	Ohne Cache	88,0	0,99
React CSR	WiFi	Mit Cache	112,0	0,98
React CSR	WiFi	Ohne Cache	88,0	0,99

Tabelle 9: INP-Messergebnisse der interaktiven Website

Framework	**Netzwerkbedingung**	**Cache**	**Median CLS**	**Score**
Qwik SSR	Slow 3G	Mit Cache	0,19	0,65
Qwik SSR	Slow 3G	Ohne Cache	0,21	0,6
Qwik SSR	Fast 3G	Mit Cache	0,19	0,65
Qwik SSR	Fast 3G	Ohne Cache	0,24	0,52
Qwik SSR	LTE	Mit Cache	0,19	0,65
Qwik SSR	LTE	Ohne Cache	0,24	0,52
Qwik SSR	WiFi	Mit Cache	0,19	0,65
Qwik SSR	WiFi	Ohne Cache	0,24	0,52
Solid.js SSR	Slow 3G	Mit Cache	0,17	0,71
Solid.js SSR	Slow 3G	Ohne Cache	0,22	0,57
Solid.js SSR	Fast 3G	Mit Cache	0,17	0,71
Solid.js SSR	Fast 3G	Ohne Cache	0,22	0,57
Solid.js SSR	LTE	Mit Cache	0,17	0,71
Solid.js SSR	LTE	Ohne Cache	0,22	0,57
Solid.js SSR	WiFi	Mit Cache	0,17	0,71
Solid.js SSR	WiFi	Ohne Cache	0,22	0,57
Next.js SSR	Slow 3G	Mit Cache	0,17	0,71
Next.js SSR	Slow 3G	Ohne Cache	0,22	0,57
Next.js SSR	Fast 3G	Mit Cache	0,17	0,71
Next.js SSR	Fast 3G	Ohne Cache	0,22	0,57
Next.js SSR	LTE	Mit Cache	0,17	0,71
Next.js SSR	LTE	Ohne Cache	0,22	0,57
Next.js SSR	WiFi	Mit Cache	0,17	0,71
Next.js SSR	WiFi	Ohne Cache	0,22	0,57
Solid.js CSR	Slow 3G	Mit Cache	0,17	0,71
Solid.js CSR	Slow 3G	Ohne Cache	0,22	0,57
Solid.js CSR	Fast 3G	Mit Cache	0,17	0,71
Solid.js CSR	Fast 3G	Ohne Cache	0,22	0,57
Solid.js CSR	LTE	Mit Cache	0,17	0,71
Solid.js CSR	LTE	Ohne Cache	0,22	0,57
Solid.js CSR	WiFi	Mit Cache	0,17	0,71
Solid.js CSR	WiFi	Ohne Cache	0,22	0,57
React CSR	Slow 3G	Mit Cache	0,17	0,71
React CSR	Slow 3G	Ohne Cache	0,22	0,57
React CSR	Fast 3G	Mit Cache	0,17	0,71
React CSR	Fast 3G	Ohne Cache	0,22	0,57
React CSR	LTE	Mit Cache	0,17	0,71
React CSR	LTE	Ohne Cache	0,22	0,57
React CSR	WiFi	Mit Cache	0,17	0,71
React CSR	WiFi	Ohne Cache	0,22	0,57

Tabelle 10: CLS-Messergebnisse der interaktiven Website

Anhang B

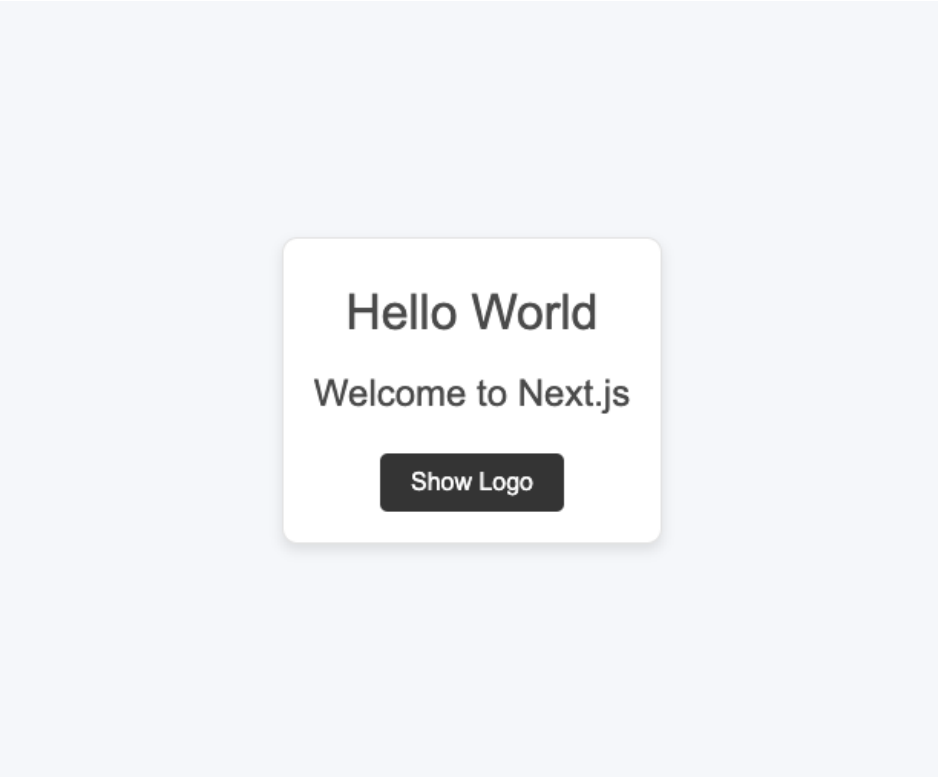

Abbildung 1: Minimalistische Website

Abbildung 2: Interaktive Website

Abbildung 3: Interaktive Website – Namenskomponente

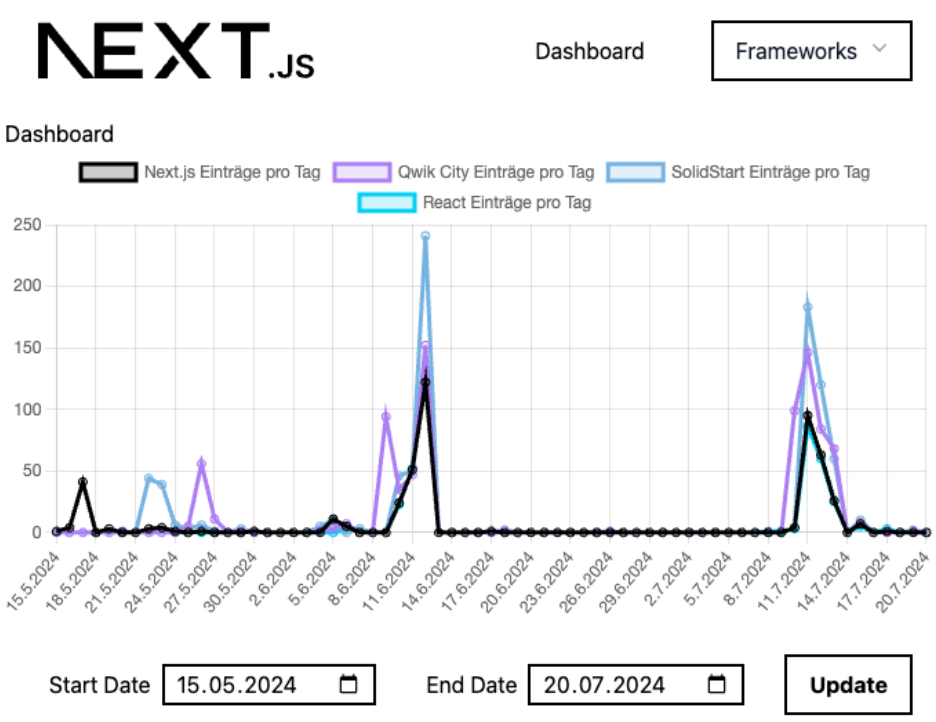

Abbildung 4: Interaktive Website – Dashboard

Anhang C

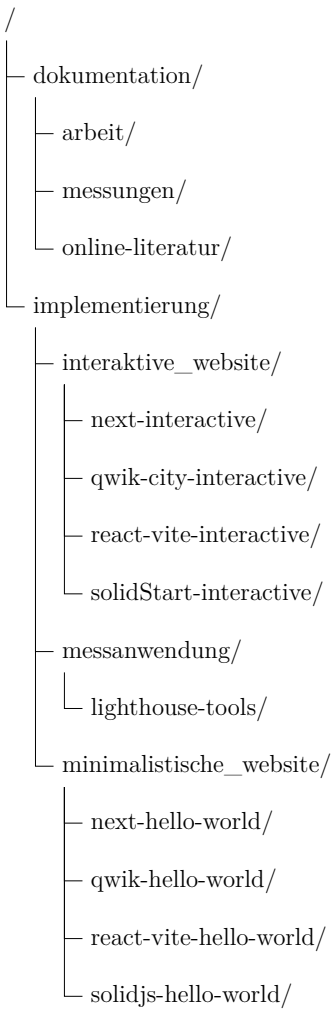

Abbildung 1: Ordnerstruktur der digitalen Abgabe