pep.digital

HOCHSCHULE ESSLINGEN

Nah an Mensch und Technik.

Bachelorarbeit

Analyse des Frameworks Qwik im Vergleich zu anderen Web-Frameworks

im Studiengang
Softwaretechnik und Medieninformatik
am 15. Januar 2023

vorgelegt von

Florian Hoss

Matrikelnummer: 762472

Erstprüfer: Prof. Dr. rer. nat. Mirko Sonntag
Zweitprüfer: Prof. Dr.-Ing. Andreas Rößler

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Esslingen, den 15. Januar 2023 

Unterschrift

Lizenzen

Im Folgenden werden genutzte Komponenten mit Lizenz- und Urheberrechtshinweisen vorgestellt (vgl. GitHub, 2022, Licenses).

Whiteboard-Tool Excalidraw

Manche als *eigene Darstellung* gekennzeichnete Grafiken sind selbstständig mit einem Whiteboard-Tool namens Excalidraw erstellt. Darin enthaltene Komponenten können aus externen Bibliotheken stammen, welche unter der MIT-Lizenz stehen (vgl. Excalidraw, 2022, LICENSE).

Material Design Icons

In den verschiedenen Anwendungsbeispielen (Kapitel 3) werden unter der Apache 2.0 Lizenz stehende, als offener Quellcode veröffentlichte, Material Design Icons genutzt (vgl. Material Design, 2022, LICENSE).

Tailwind

Für vorgefertigte Cascading Style Sheet (CSS)-Klassen wird Tailwind verwendet. Das CSS-Framework steht unter der MIT-Lizenz (vgl. Tailwind Labs, Inc., 2022, LICENSE).

DaisyUI

DaisyUI ist ein Plugin für das Tailwind CSS-Framework. Damit können verschiedene, schon vorgefertigte Komponenten einfach wieder verwendet werden. Die Bibliothek steht unter der MIT-Lizenz (vgl. Saadeghi, 2022, LICENSE).

REST Countries

Für die Länder-Fallstudie (Kapitel 3.3) werden Informationen der Länder von REST Countries abgerufen und genutzt. Der Service steht unter der Mozilla Public License 2.0 (vgl. REST Countries, 2022, LICENSE).

Pexels

Für Grafiken in Fallstudien werden Bilder von Pexels genutzt. Alle Fotos und Videos auf Pexels können kostenlos heruntergeladen und verwendet werden (vgl. Pexels, 2022, LICENSE).

Go

Go ist eine kompilierbare Programmiersprache, die Nebenläufigkeit und automatische Speicherbereinigung unterstützt. Sie wird in dieser Arbeit bei fast allen vorgestellten Tests und Automationen eingesetzt. Die Programmiersprache steht unter der BSD 3-Klausel „Neu“ oder „Überarbeitet“ Lizenz (vgl. Google, 2009, LICENSE).

Node.js

Node.js ist eine plattformübergreifende Open-Source-JavaScript (JS)-Laufzeitumgebung, die JS-Code außerhalb eines Webbrowsers ausführen kann. Sie wird in dieser Arbeit bei programmatischen Tests mit Google Lighthouse verwendet (Kapitel 5.2.2). Die Programmiersprache steht unter der MIT-Lizenz (vgl. Node contributors, 2005, LICENSE).

Google Lighthouse

Google Lighthouse ist ein automatisiertes Open-Source-Tool zur Messung der Qualität von Webseiten. Sie wird in dieser Arbeit bei automatisierten Tests genutzt (Kapitel 5.2). Die Software steht unter der Apache License 2.0 Lizenz (vgl. Google, 2004, LICENSE).

Traefik

Traefik (sprich: Traffic) ist ein HTTP-Reverse-Proxy und Load Balancer. Er wird in dieser Arbeit bei der Bereitstellung der Fallstudien und zum Sammeln von Metriken genutzt (Kapitel 5.1.1). Die Software steht unter der MIT-Lizenz (vgl. Traefik Labs, 2016b, LICENSE).

Docker

Die Docker Engine wird in dieser Arbeit eingesetzt, um die geschriebene Software in einer kontrollierten Umgebung auszuführen. Docker wird unter einer proprietären Lizenz bereitgestellt und kann kostenlos für einzelne Entwickler, Bildungseinrichtungen, Open-Source-Gemeinschaften und kleine Unternehmen eingesetzt werden (vgl. Docker Inc., 2022, Pricing & Subscriptions).

Grafana

Zur Visualisierung verschiedene Metriken wird Grafana eingesetzt. Die Open-Source-Plattform für Überwachung und Beobachtbarkeit steht unter der AGPL-3.0 Lizenz (vgl. Grafana Labs, 2014, LICENSE).

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Codeverzeichnis

Abkürzungsverzeichnis

AJAX	Asynchronous JavaScript And XML
API	Application Programming Interface
ASP	Active Server Pages
CDN	Content Delivery Network
CLI	Command Line Interface
CPU	Central processing unit
CRUD	Create Read Update Delete
CSR	Client-Side Rendering
CSS	Cascading Style Sheet
DOM	Document Object Model
GUI	Graphical user interface
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IDE	Integrierte Entwicklungsumgebung
JS	JavaScript
JSON	JavaScript-Objekt-Notation
kB	Kilobyte
KiB	Kibibyte
MDX	Markdown with syntax extension
MIME	Multipurpose Internet Mail Extensions
PHP	Hypertext Preprocessor
PWA	Progressive Web App
QRL	Qwik-URL
REST	Representational State Transfer
SEO	Search Engine Optimization
SMTP	Simple Mail Transfer Protocol
SSG	Static Site Generation
SSR	Server-Side Rendering
SVG	Scalable Vector Graphics
TLS	Transport Layer Security

TSX TypeScript with syntax extension
URL Uniform Resource Locator
USD United States Dollar
UX User Experience

1 Einführung

Abbildung 1.1: Qwik Logo (Builder.io Inc, 2022)

> „Qwik[1] (Abbildung 1.1) ist ein neuartiges Web-Framework, mit dem Webanwendungen jeder Größe und Komplexität sofort geladen werden können. Ihre Webseiten und Anwendungen können mit etwa 1 Kilobyte (kB) JavaScript starten (unabhängig von der Komplexität der Anwendung) und erreichen eine konsistente Leistung im großen Maßstab.“ (Builder.io Inc, 2022)

Die Botschaft der Entwickler des Qwik Web-Framework ist klar: Webseiten laden anfänglich zu langsam. Dieses Problem möchten sie mit dem als HyperText Markup Language (HTML)-first bezeichneten Ansatz lösen. Es soll vollständig interaktive Webseiten fast ohne JavaScript (JS) laden. Während der Benutzer mit der Seite interagiert, werden nur die genutzten JS-Teile der Webseite bei Bedarf nachgeladen (vgl. Builder.io Inc, 2021).

In der folgenden Arbeit wird diese These auf die Probe gestellt. Dabei werden Unterschiede zu konkurrierenden Frontend-Web-Frameworks ermittelt und analysiert. Dazu werden mehrere Beispielanwendungen (Kapitel 3) programmiert, die verschiedene Anforderungen an die jeweiligen Frameworks stellen.

Das Qwik Web-Framework befindet sich noch in der Entwicklungsphase. Die vorgestellten Fallstudien in dieser Arbeit sind mit Qwik in der Version 0.16.1[2] umgesetzt.

Zur Zeit der Abgabe befindet sich das Framework in der Version 0.16.2[3].

1 qwik.builder.io
2 github.com/BuilderIO/qwik/releases/tag/v0.16.1
3 github.com/BuilderIO/qwik/releases/tag/v0.16.2

1.1 Motivation des Qwik Web-Framework

Qwik möchte ein Problem lösen, welches von noch keinem anderen JS-Framework gelöst werden konnte. Mit Qwik sollen Anwendungen unabhängig von der Komplexität eine sofortige Verfügbarkeit haben. Gerade in Regionen mit limitierter Netzwerkbandbreite will das Framework glänzen. Denn große Mengen an JS macht sich durch zwei Probleme bemerkbar:

- Zu Beginn der Anwendung wird ein großer Teil des Codes und damit viele Daten an den Browser übertragen
- Erst bei einer kompletten Übertragung wird der Code ausgeführt und die Anwendung dargestellt

Genau deshalb wird bei einer mit dem Qwik Web-Framework umgesetzten Anwendung *nicht* das komplette JS auf einmal an den Benutzer übertragen. Progressiven Nachladen findet auf der Grundlage von Benutzerinteraktionen statt. Das Ziel des Qwik Web-Framework soll es damit sein, Seiten so schnell wie möglich dem Benutzer zur Verfügung zu stellen. Dabei soll JS-Code und die Interaktivität der Seite nicht verloren gehen.

In den letzten Jahren wurden immer mehr Webseiten mit mobilen Geräten besucht. Seit Oktober 2016 haben diese Endgeräte die Internetnutzung per Desktop überholt (vgl. Lang, 2016). Gleichzeitig kann man bei der mobilen Breitbandmessung in den letzten Jahren nur eine geringe Verbesserung feststellen. Die Ergebnisse der Jahresberichte in Deutschland zeigen außerdem, dass nicht jeder Nutzer Zugriff über eine schnelle Datenrate hat. Bei den für den Bericht 441.223 durchgeführten Tests wurde nur bei der Hälfte der Nutzer bis zu 21.3 Mbit/s Übertragungsgeschwindigkeit gemessen (Abbildung 1.2) (vgl. Zafaco, 2022).

Zusätzlich muss der Endbenutzer für mobiles Datenvolumen ohne entsprechenden Vertrag viel Geld zahlen. In Deutschland wird für den günstigsten übertragenen Gigabyte $0,50 United States Dollar (USD) ausgegeben, im Durchschnitt sogar $2,67 USD. Bei einer Anwendung, die zum Starten viele Daten benötigt, können so dem Benutzer mehr Kosten entstehen. Selbst im günstigsten Land Israel kostet im Durchschnitt ein übertragener Gigabyte Daten immer noch $0.04 USD (vgl. Howdle, 2022).

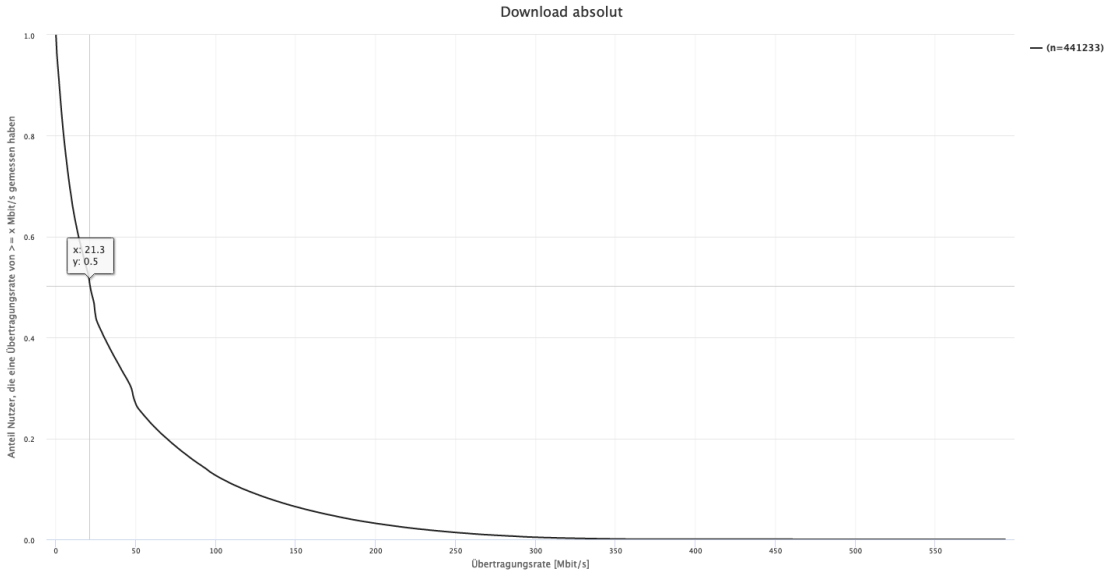


Abbildung 1.2: Breitbandmessung 2020/2021 - Jahresbericht Mobil (Zafaco, 2022)

Beim Entwickeln einer Anwendung mit einem Framework ist es die Aufgabe des Entwicklers, die Größe der Anwendung und die Ladezeiten zu verringern. Dazu kann z. B. mit Lighthouse[4] die Applikation analysiert werden (Abbildung 1.3). Vorgeschlagen wird dann u. a., dass ungenutztes CSS[5]/JS[6] entfernt werden kann, Rendering-blockierende Ressourcen beseitigt werden können[7] und statische Objekte mit einer effizienten Cache-Politik ausgeliefert werden können[8] (vgl. Chrome Developers, 2016).

Die Ansätze können bei den meisten Frameworks mit wenig Aufwand und Zeit ausgeführt werden. Beim Bauen der Anwendung können Techniken wie *Tree Shaking* (Auch *dead code elimination* genannt (deutsch: *Beseitigung von totem Code*)) genutzt werden, um JS zu entfernen. Dies funktioniert allerdings nur bei ungenutztem Code (vgl. Haczek, 2020). Das Problem der Anwendungsgröße und der damit verbunden Ladezeit, bleibt weiterhin bestehen. Wenn ein Framework renderblockierende Ressourcen laden muss, sind dem Entwickler die Hände gebunden.

4 github.com/GoogleChrome/lighthouse
5 web.dev/unused-css-rules
6 web.dev/unused-javascript
7 web.dev/render-blocking-resources
8 web.dev/uses-long-cache-ttl

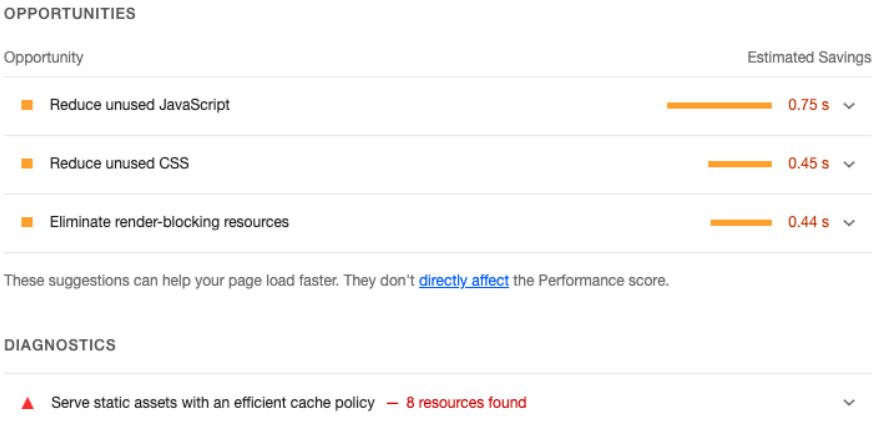


Abbildung 1.3: Google Lighthouse Vorschläge (Eigene Darstellung)

In den letzten Jahren hat zusätzlich die Menge an JS, die an Browser gesendet wird, stetig zugenommen (Abbildung 1.4). Mit steigender Komplexität der Anwendungen wird sich dieser Trend fortsetzen.

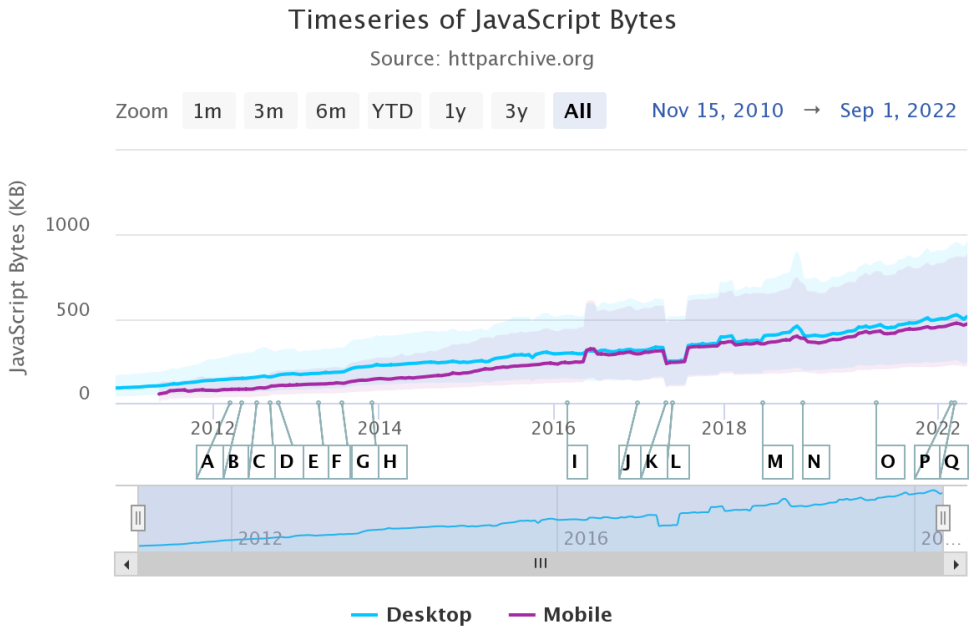
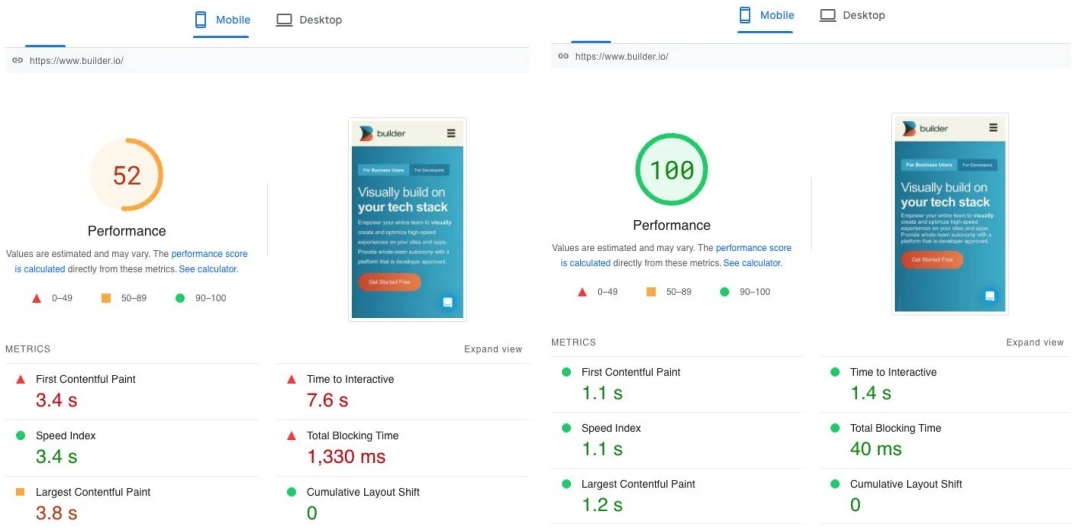


Abbildung 1.4: Übertragungsgröße von Seiten angefordertem JS (HTTP Archive, 2022a)

Doch genau hier verspricht das Qwik Web-Framework Besserung. Vor allem die Zeit zum sogenannten *First Contentful Paint* (deutsch: *Einblenden des ersten Inhalts*), welcher den Zeitpunkt markiert, an dem der erste Text oder das erste Bild angezeigt wird, kann reduziert werden. Auch die *Time to Interactive* (deutsch: *Zeit zum Interagieren*), die Zeit, die es dauert, bis die Seite vollständig interaktiv ist, kann damit erheblich gesenkt werden (Abbildung 1.5).



(a) Seitengeschwindigkeit ohne Qwik (b) Seitengeschwindigkeit mit Qwik

Abbildung 1.5: Builder.io Seitengeschwindigkeit (Hevery, 2021)

Es stellt sich trotzdem die Frage, ob die Ladezeiten von vorhandenen Frameworks, für den normalen Nutzer nicht schon ausreichend sind. Interessiert sich der Benutzer überhaupt für diesen initialen Geschwindigkeitsvorteil? Da die vorgestellten Probleme auch nur beim ersten Laden, also ohne Inhalt im Zwischenspeicher (englisch: *cache*) auftreten, lohnt sich ein anderer Ansatz überhaupt? Dazu können verschiedenen Fallbeispielen von *web.dev*[9] genannt werden (vgl. Builder.io Inc, 2022, Docs/Think Qwik):

> **Jede 100 ms schneller → 1 % mehr Umsätze:**
> Bei Mobify[10] entsprach jede Verringerung der Ladegeschwindigkeit der Homepage um 100 ms, einer Steigerung der sitzungsbasierten Umsätze um 1,11 %.

9 web.dev
10 mobify.com

Dies führte zu einer durchschnittlichen jährlichen Umsatzsteigerung von fast 380.000 USD.

850 ms schneller → 7 % mehr Umsätze
COOK[11] verkürzte die durchschnittliche Seitenladezeit um 850 Millisekunden, was die Umsätze um 7 % erhöhte, die Absprungrate um 7 % verringerte und die Anzahl der Seiten pro Sitzung um 10 % ansteigen lies.

50 % schneller → 12 % mehr Umsatz
Als AutoAnything[12] die Ladezeit einer Seite um die Hälfte reduzierte, verzeichnete das Unternehmen einen Umsatzanstieg von 12 % bis 13 %.

Eine Sekunde Verlangsamung → 10 % weniger Nutzer:
Die BBC[13] stellte fest, dass sie für jede zusätzliche Sekunde, die das Laden ihrer Website dauert, 10 % mehr Nutzer verliert.

Langsame Webseiten schrecken Besucher ab und kosten die Unternehmen Millionen. Schnell ladende Webseiten haben eine bessere Search Engine Optimization (SEO) (deutsch: *Suchmaschinenoptimierung*), eine bessere User Experience (UX) (deutsch: *Benutzererfahrung*) und sind profitabler (vgl. Pavic; Anstey; Wagner, 2022).

1.2 Zielsetzung der Bachelorarbeit

Zielsetzung dieser Arbeit ist es, das Qwik Web-Framework ausgiebig unter verschiedenen Aspekten zu untersuchen. Dabei sollen vor allem die genannten Kernpunkte des Frameworks auf Durchführbarkeit überprüft werden. Außerdem gilt es, die Notwendigkeit für solch ein Framework zu bestimmen. Dafür werden automatisierte Tests auf verschiedenen Geräten und mit verschiedenen Netzwerkgeschwindigkeiten durchgeführt. Die Bedingungen sollen dafür für alle getesteten Frameworks die gleichen sein. Zum Ende der Arbeit werden vor allem folgende Fragen beantwortet:

- Wie viel JS wird pro Framework bei den verschiedenen Beispielanwendungen (Kapitel 3) initial und insgesamt nach allen ausgeführten Aktionen geladen?
- Ist eine mit Qwik geschriebene Anwendung schneller geladen?
- Wie viel Mehraufwand steckt in der Programmierung des Frameworks Qwik im Vergleich zu den anderen Frameworks?

11 www.cookfood.net
12 www.autoanything.com
13 www.bbc.com

2 Konzepte eines Web-Framework

Um zu verstehen, welches Problem das Qwik Web-Framework lösen möchte, wird zuerst der Fokus auf die verschiedenen Konzepte der existierenden Web-Frameworks gelegt. Ein Überblick über die aktuelle Vorgehensweise bei der Web-Entwicklung zeigt, welche Techniken genutzt werden können.

Web-Anwendungen werden normalerweise auf der Grundlage von HTML-, CSS- und JS-Dateien von einem Server zu Darstellung an einen Browser übertragen. Dort kann die finale Ansicht vorbereitet oder nach der Übertragung an den Nutzer auf dem Endgerät erstellt werden.

2.1 Darstellen einer Anwendung

Die Diskussion über verschiedene Techniken des Renderns von Webseiten ist erst in den letzten Jahren aufgekommen. Früher verfolgten die Websites und Webanwendungen eine gemeinsame Strategie. Sie bereiteten den HTML-Inhalt, der an die Browser gesendet werden sollte, serverseitig vor (Server-Side Rendering (SSR) - Kapitel 2.1.1). Dabei wurden vor allem Hypertext Preprocessor (PHP)[14] und Active Server Pages (ASP)[15] genutzt (vgl. Shah, 2019).

Es entstanden Techniken wie z. B. Asynchronous JavaScript And XML (AJAX), um das Neuladen von Webseiten zu verhindern. AJAX ermöglicht es, HTTP-Anfragen durchzuführen, während eine HTML-Seite angezeigt wird, und die Seite zu verändern, ohne sie völlig neu zu laden (vgl. Mozilla Corporation, 2022a).

In der modernen Web-Entwicklung sind JS-Frameworks, welche komponentenorientierte Entwicklung unterstützen, vorherrschend. Dabei können verschiedene Teile der Webseite mit sogenannten Komponenten einfach wiederverwendet werden. React[16], Angular[17] und

14 www.php.net
15 www.w3schools.com/asp
16 reactjs.org
17 angular.io

Vue[18] dominieren die aktuelle Frontend-Web-Entwicklung (vgl. Ivanovs, 2022) und setzen auf Client-Side Rendering (CSR). Trotz der Beliebtheit und der weiten Verbreitung kommt aber immer wieder die Frage auf, welches der beiden Ansätze die bessere Wahl ist. Denn bei beiden Ansätzen müssen Kompromisse eingegangen werden. Wie z. B. die im Kapitel 1.1 angesprochene Zeit zum *First Contentful Paint*. Je umfangreicher die Anwendung und der JS-Code, desto größer die Wartezeit. Verschiedene neue Frameworks (u. a. NuxtJS[19] und NextJS[20]) versuchen deshalb mehrere Techniken gleichzeitig zu nutzen. Damit können die Vorteile der jeweiligen Konzepte mit mehr Programmieraufwand kombiniert werden.

2.1.1 Server-Side Rendering

Lange Zeit wurden auf Webseiten nur statische Bilder und Texte angezeigt, ohne viel Interaktivität zu bieten. Deswegen war die herkömmliche Methode, um HTML auszuliefern, das serverseitige Rendering.

Sobald der Benutzer die Seite besucht, wird eine Anfrage an den Server gestellt. Nachdem der Server diese bearbeitet hat, erhält der Browser das vollständig gerenderte HTML zurück. Der Inhalt des HTML hängt dabei von der jeweiligen gestellten Anfrage ab (Abbildung 2.1).

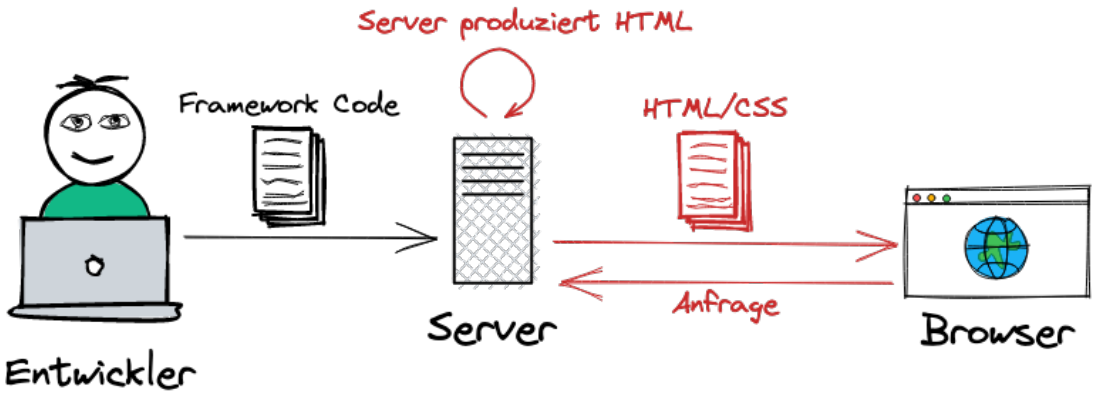


Abbildung 2.1: Server-Side Rendering (Eigene Darstellung)

18 vuejs.org
19 nuxtjs.org
20 nextjs.org

Es spielt dabei keine Rolle, ob die angeforderte Seite nur wenige oder viele Elemente enthält. Der Browser fordert immer die gesamte HTML-Seite an und stellt alles von Grund auf neu dar. Dies führt zu einer besseren SEO-Behandlung und der anfängliche Seitenaufbau ist schneller. Dadurch ist SSR ideal für statische Webseiten.

Die Anwendung muss allerdings viele Anfragen an den Server stellen. Dementsprechend fällt dort eine hohe Belastung an. Nach dem initialen Laden dauert jede Veränderung so lange, bis das HTML auf dem Server gerendert und an den Benutzer zurückgeschickt wird. Dadurch ist die Seitendarstellungen insgesamt langsamer (vgl. Vega, 2017, How server-side rendering works).

Auf Eingaben, die vom Benutzer gemacht werden, kann damit nur verzögert reagiert werden. Bei viel Interaktivität in einer Anwendung ist SSR das schlechteste Konzept.

2.1.2 Client-Side Rendering

Wenn Inhalte im Browser mithilfe von JS gerendert werden, spricht man von Client-Side Rendering (CSR). Anstatt den gesamten Inhalt fertig in einem HTML-Dokument zu erhalten, startet er mit einem einfachen HTML-Dokument und einer JS-Datei. Der Rest der Webseite wird dynamisch nachgeladen und gerendert (Abbildung 2.2). Dieser Ansatz für die Darstellung von Webseiten wurde erst populär, als JS-Frameworks begannen, ihn in ihren Entwicklungsstil zu integrieren.

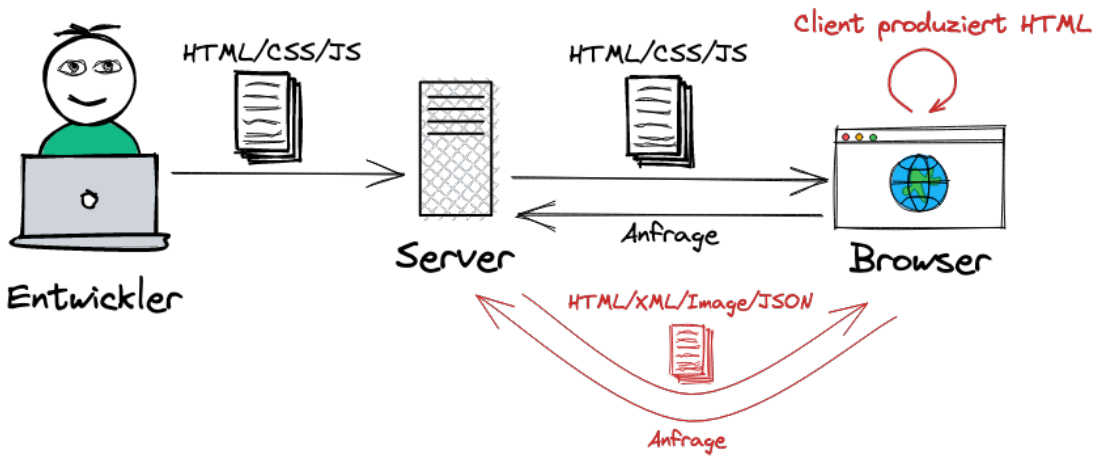


Abbildung 2.2: Client-Side Rendering (Eigene Darstellung)

Wenn Interaktionen stattfinden, stellt das Framework sicher, dass nur neue Inhalte geändert werden. Dies ist viel schneller, als die ganze Seite neu aufzubauen. Die Anwendung kann aber erst dann geladen werden, wenn das gesamte JS in den Browser heruntergeladen wurde. Mit CSR wird die Interaktivität der Anwendung gesteigert, denn Änderungen werden schneller geladen. Dadurch ist dieses Konzept ideal für Webanwendungen mit vielen Interaktionen.

Nachteilig ist allerdings eine schlechtere SEO-Behandlung im Vergleich zu SSR (Kapitel 2.1.1) und Static Site Generation (SSG) (Kapitel 2.1.3). Zum Entwickeln gern genutzte JS-Frameworks haben außerdem oft eine große Codebasis. Dadurch kann das initiale Laden mehr Zeit in Anspruch nehmen (vgl. Vega, 2017, How client-side rendering works).

Aber genau hier versprechen das Qwik Web-Framework und auch viele neue Web-Frameworks Besserung.

2.1.3 Static Site Generation

Eine der besten Möglichkeiten zur Erstellung schneller Webseiten ist die Verwendung von Static Site Generation (SSG) anstelle von Server-Side Rendering (SSR) oder Client-Side Rendering (CSR).

SSG bedeutet, dass Komponenten und Routen zur Erstellungszeit und nicht erst bei Anforderung erstellt und gerendert werden (englisch: *Pre-rendering*). Da eine Route bereits vorgerendert ist, steht der gesamte Inhalt der Route den Suchmaschinen und Clients sofort zur Verfügung, wodurch SEO und Leistung maximiert werden.

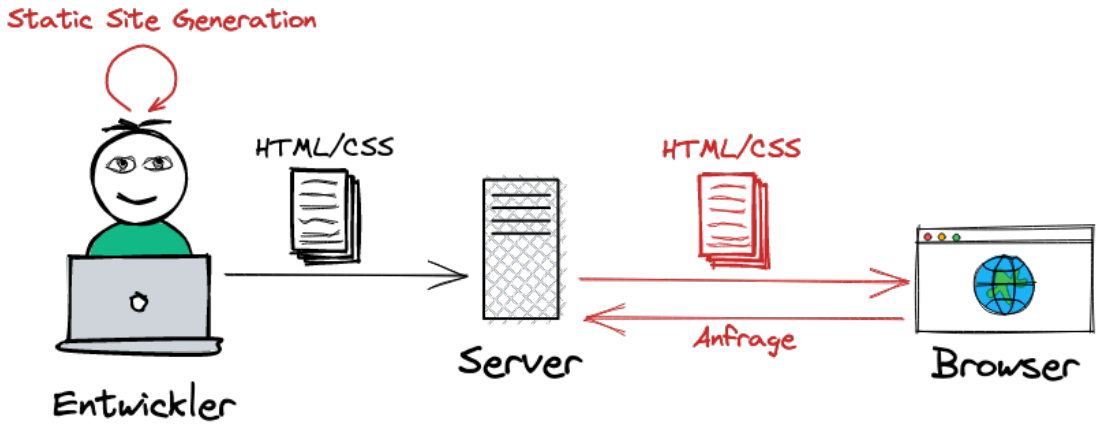


Abbildung 2.3: Static Site Generation (Eigene Darstellung)

Bei dem Benutzer kann bei statisch generierten Webseiten JS im Browser deaktiviert sein. Das Konzept bietet zusätzlich eine bessere SEO-Behandlung.

Außerdem ist eine Bereitstellung auf einem Content Delivery Network (CDN) einfach möglich. Dabei handelt es sich um ein Netzwerk von geografisch verteilten Server, die für die Bereitstellung der Inhalte zusammenarbeiten. Die Server in einem CDN speichern (englisch: *cachen*) vorübergehend Webseiteninhalte. Sie senden die zwischengespeicherten Inhalte an die Nutzer, die die Webseite laden. (vgl. Cloudflare, 2022).

Generierungswerkzeuge nehmen dabei die meiste Arbeit für den Entwickler ab. Doch bei z. B. der Darstellung einer sortierbaren Tabelle werden bei der statischen Generierung für jede Sortierung eine HTML-Seite erstellt. Damit kann je nach Interaktivität der Anwendung eine große Anzahl von Dateien entstehen. (vgl. Crutchley, 2020).

2.2 Hydrierung einer Anwendung

Wenn eine SSR/SSG-Anwendung auf einem Endgerät hochfährt, muss das genutzte Framework drei Informationen wiederherstellen, um die programmierte Interaktivität in JS zu gewährleisten.

- Ereignis-Listener im DOM (Kapitel 2.2.1)
- Komponenten-Baum (Kapitel 2.2.2)
- Anwendungsstatus (Kapitel 2.2.3)

Fast alle der aktuellen Generationen von Frontend-Web-Frameworks erfordern diese Schritte, um die Anwendung interaktiv zu machen. Dies wird auch als Hydrierung (englisch: *Hydration*) bezeichnet (Abbildung 2.4, vgl. Hevery, 2022a).

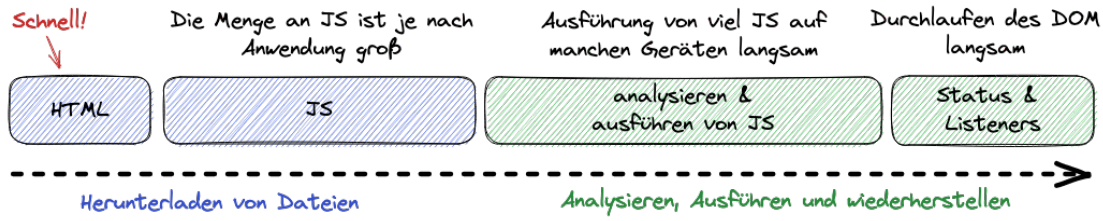


Abbildung 2.4: Ablauf der Hydrierung (Eigene Darstellung)

Es handelt sich um eine zuverlässige Art, mithilfe von Client-Seitigem JS, Interaktivität auf statischen HTML-Seiten zu erreichen. Frameworks müssen dafür aber die gesamten mit der aktuellen Seite verbundenen Skripte herunterladen. Anschließend müssen diese ausgeführt werden, um die Listener, die mit den Komponenten auf der Seite verbunden sind und den internen Komponenten-Baum, neu zu erstellen (vgl. Hevery, 2022a).

2.2.1 Das Überwachen der Ereignisse im DOM

Das Document Object Model (DOM) ist eine Programmierschnittstelle für Webdokumente. Es stellt die Seite als Knoten und Objekte dar, sodass Programmiersprachen mit der Seite interagieren können. Überwacher (englisch: *listener*) mit einem bestimmtem Ereignis-Typ[21] können an verschiedene Elemente des DOM angefügt werden. Es handelt sich dabei um Funktionen, die immer dann aufgerufen werden, wenn das Ereignis eintritt (vgl. Mozilla Corporation, 2022b). Bei statischen Anwendungen könnten manche Interaktionen zum Teil auch mit CSS erreicht werden (vgl. Builder.io Inc, 2022, Listeners).

2.2.2 Der Komponenten-Baum

Bei komponentenorientierten Frameworks können verschiedene Teile der Anwendung mit sogenannten Komponenten einfach wiederverwendet werden. Dabei können sich verändernde Eigenschaften (englisch: *props*) von sogenannten Eltern-Komponenten (englisch: *parent components*) an Kinder-Komponenten (englisch: *child components*) übergeben werden. Frameworks benötigen zu jeder Zeit ein vollständiges Verständnis des Komponenten-Baum (englisch: *Component Tree*). Nur damit können sie wissen, welche Eltern- und Kinder-Komponenten bei einer Änderung neu gerendert werden müssen. Wenn man sich die resultierenden HTML-Datei ansieht, sind die Informationen über die Grenzen der Komponenten nicht mehr zu erkennen. Um diese Informationen wiederherzustellen, führen die Frameworks die Schablonen der Komponenten erneut aus und merken sich dann die Position der Grenzen (vgl. Builder.io Inc, 2022, Component Tree).

2.2.3 Das Wiederherstellung des Anwendungsstatus

Die meisten bestehende Frameworks haben in der Regel eine Möglichkeit, den Anwendungsstatus (englisch: *Application State*) in HTML zu serialisieren, sodass der Status als Teil der Hydrierung wiederhergestellt werden kann. Eigenschaften (englisch: *props*)[22]

21 developer.mozilla.org/en-US/docs/Web/Events
22 qwik.builder.io/docs/cheat/components/#props

werden aber vorwiegend von der übergeordneten Komponente erstellt und weitergegeben. Dies führt zu einer Kettenreaktion. Denn wenn z. B. eine Komponente mit ihren Eigenschaften wiederhergestellt wird, müssen alle überliegenden Komponenten ebenfalls wiederhergestellt werden. Serialisierung kann aber nicht alle Objekttypen (z. B. DOM references, Datum, Klassen, Promises, Streams, usw...) korrekt lesen und hat damit einen erheblichen Mehraufwand mit der erneuter Kompilierung (vgl. Builder.io Inc, 2022, Application State). Eine gute Serialisierbarkeit ist deswegen direkt für die Geschwindigkeit bei dem Wiederherstellen des Anwendungsstatus verantwortlich.

2.3 Konzepte des Qwik Web-Framework

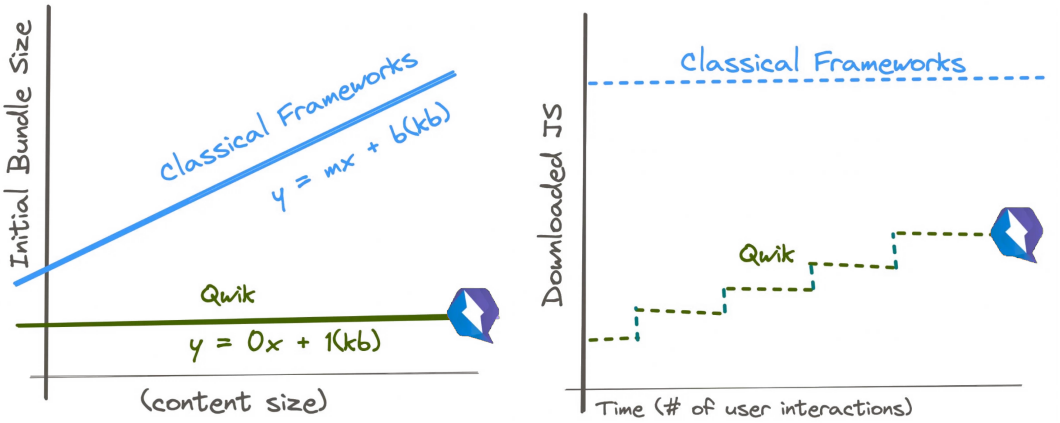
Qwik nutzt verschiedene Konzepte, um die in Kapitel 1.1 angesprochenen Ziele zu verwirklichen. Die in den nächsten Kapiteln vorgestellten Konzepte unterschieden sich grundlegend von klassischen Web-Frameworks.

2.3.1 Der O(1) Ansatz

Die Big-O-Notation ist eine Methode zur Beschreibung der Geschwindigkeit oder Komplexität eines bestimmten Algorithmus. Wenn ein Projekt einen vordefinierten Algorithmus erfordert, ist es wichtig zu wissen, wie schnell oder langsam er im Vergleich zu anderen Optionen ist (vgl. Thompson, 2021).

Jedes Jahr werden die Central processing unit (CPU)s schneller, aber der größte Teil des Geschwindigkeitszuwachses kommt von verbesserter Parallelität. Hinzu kommt, dass sich in den letzten Jahren Leistungssteigerungen pro Kern verlangsamt haben. Wir nähern uns den Grenzen, wie schnell CPU-Takte laufen können und wie viel Intelligenz die CPUs pro Taktzyklus leisten können.

Je größer die Webseiten werden und je mehr JS heruntergeladen wird, desto mehr Arbeit hat die CPU zu erledigen. Und da JS ohne erheblichen Mehraufwand des Entwicklers ein Single-Thread-Verfahren ist, fällt die gesamte zusätzliche Arbeit auf einen einzigen Kern (Siddique, 2020). Letztendlich steigt die aktuelle JS-Belastung schneller als die Leistung der Einzelnen CPU-Kerne.



(a) Anfängliche Paketgröße (b) Heruntergeladenes JS

Abbildung 2.5: Der O(1) Ansatz (Hevery, 2022b)

Eine Anwendung muss auf dem Client hochgefahren werden. Das Hochfahren setzt sich aus festen Kosten für das Framework und variablen Kosten für die Anwendung selbst (Komplexität) zusammen. Es handelt sich also um eine lineare Beziehung, die sich als

$$y = mx + b$$

ausdrücken lässt. Dabei steht das y für die resultierende Paketgröße, mx für die variable Größe der Anwendung und b für die festen Kosten des genutzten Frameworks.

Das Problem ist, dass alle Frameworks eine initiale Paketgröße von O(n) (vgl. Abbildung 2.5a - Klassische Frameworks (englisch: *Classical Frameworks*)) haben. Und selbst wenn dies komplett entfernt wird und damit b Null wird, dominiert die Anwendung die Download- und Ausführungsgröße. Bei ausreichend großen Anwendungen spielt b sozusagen keine Rolle. Mit jeder Erweiterung der Anwendung wächst die initial heruntergeladene Datenmenge (vgl. Hevery, 2022b).

Nur wenn das Framework mit progressiven Nachladen auf der Grundlage von Benutzerinteraktionen arbeitet, kann eine initiale Paketgröße von O(1) erreicht werden (vgl. Qwik Abbildung 2.5a). Damit dies mit Qwik umgesetzt werden kann, wird ausschließlich ein 1 kB großer Framework-Code und keine anfängliche Anwendungslogik initial an den Benutzer übertragen (Abbildung 2.5b). Das Nachladen von Logik setzt Qwik mit dem im Kapitel 2.3.4 vorgestellten Qwik-URL (QRL) um.

2.3.2 Der Qwik Anwendungs-Lebenszyklus

Qwik bietet unterschiedliche Methoden, mit denen zu unterschiedlichen Zeitpunkten des Lebenszyklus (englisch: *Lifecycle*), verschiedene Dinge erledigt werden können (Abbildung 2.6). Diese sogenannten Hooks können allerdings nur in einer Komponente aufgerufen werden Damit können Daten schon vor dem Rendern vorbereitet werden. Seit der Version 0.16.1 wurden die unterschiedlichen Hooks mit einem einzelnen Hook ersetzt. Damit hat der Browser auf dem Endgerät lediglich die Aufgabe, die Anwendung mit HTML und CSS darzustellen (vgl. Builder.io Inc, 2022, Lifecycle)..

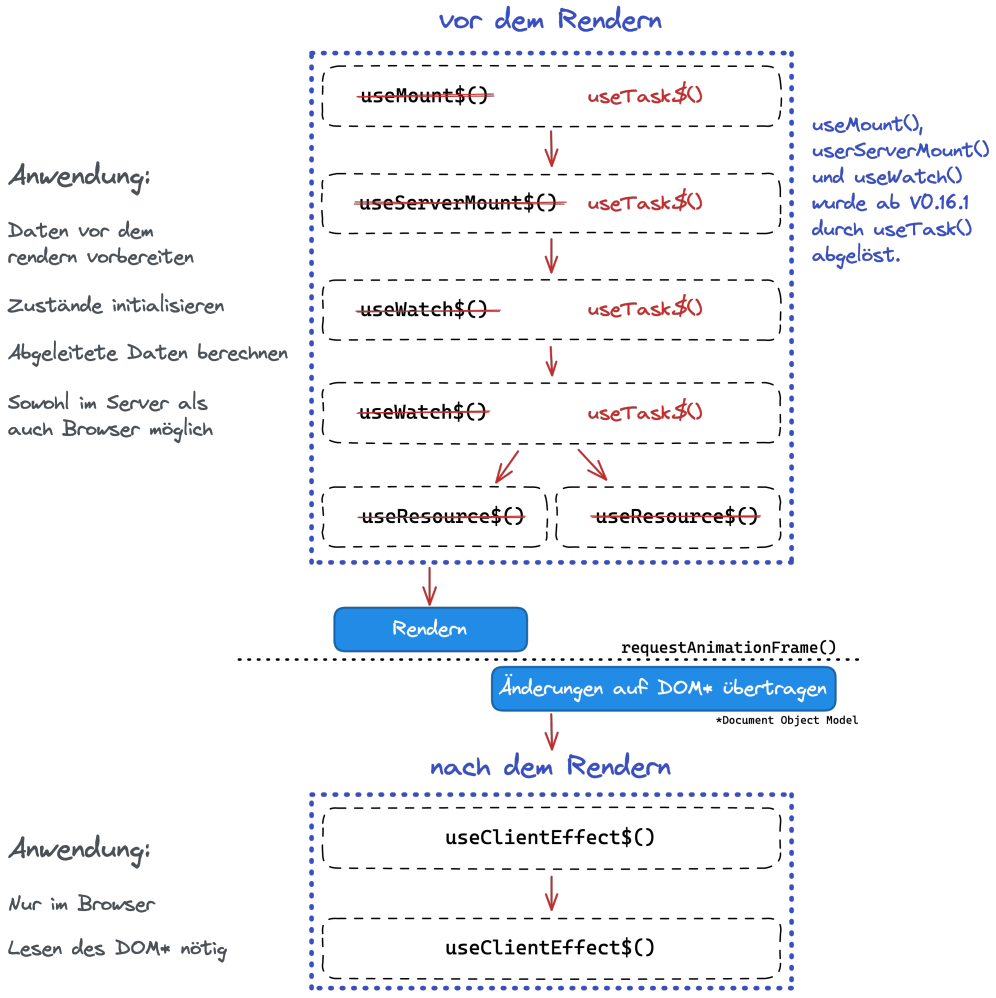


Abbildung 2.6: Qwik Lifecycle Hooks (Eigene Darstellung)

2.3.3 Der Qwikloader

Das Qwik Web-Framework ist für ein fein abgestuftes „Lazy Loading“ der Anwendung optimiert. Lazy Loading (wörtlich „faules Laden“) bezeichnet in der Softwareentwicklung ein Entwurfsmuster, bei dem Datenobjekte erst bei einer konkreten Anfrage aus der Datenquelle geholt werden.

Um Lazy-Loading zu erreichen, benötigt Qwik ein kleines Stück JS das zu Beginn geladen wird. Dieser Code weiß, wie der Rest der Anwendung bei Bedarf heruntergeladen werden kann. Er ist eingebettet in der HTML-Datei und wird als Qwikloader bezeichnet (Abbildung 2.7). Die Größe ist kleiner als 1 kB (Kapitel 2.5a). Eine Ausführung des Qwikloaders soll unter jeder Voraussetzung weniger als 5 ms dauern und beschränkt sich auf folgende Aufgaben (vgl. Builder.io Inc, 2022, Qwikloader):

- Registrieren von allen globalen Browser-Ereignissen, die benötigt werden
- Wenn ein Ereignis eintritt, wird das DOM nach dem entsprechenden Ereignisattribut durchsucht, das mit Verzögerung (QRL - Kapitel 2.3.4) geladen werden soll
- Lazy-Load und Ausführung des angehängten Codes

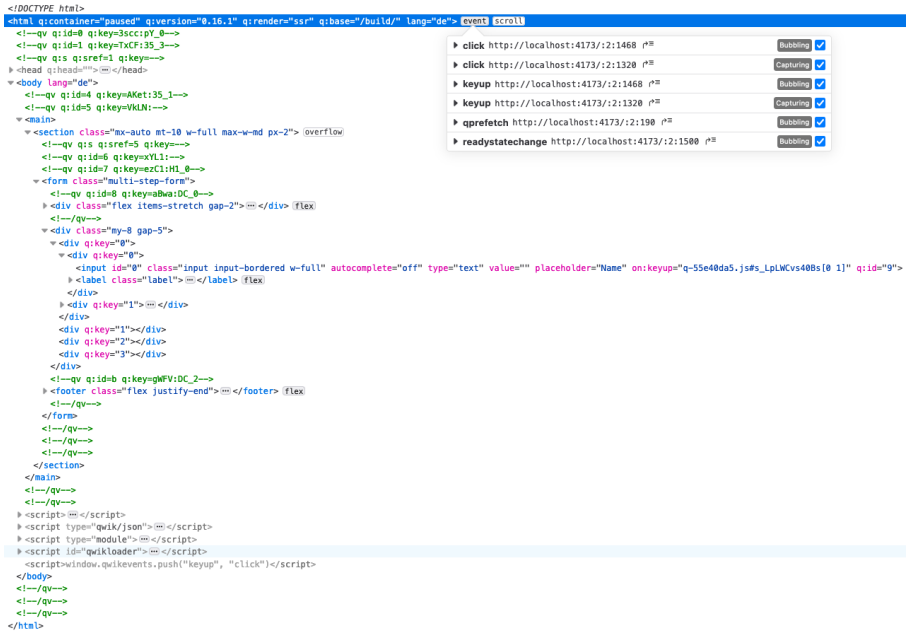


Abbildung 2.7: Übersicht einer HTML mit Annotationen, Qwikloader und dessen registrierte Events bei der Fallstudie 1: Formular (Eigene Darstellung)

2.3.4 Die Qwik-URL (QRL)

Die QRL ist eine besondere Form der Uniform Resource Locator (URL), die Qwik mithilfe des Qwikloaders (Kapitel 2.3.3) zum verzögerten Laden von Inhalten verwendet. Dazu wird die vom Entwickler geschriebene Komponente (Codefragment 2.1) mit Qwiks Optimierer in mehrere Teile zerlegt (Codefragment 2.2).

```
1  const Counter = component$((props: { step: number }) => {
2      const state = useStore({ count: 0 });
3      return <button onClick$={() => (state.count += props.step ||
           1)}>{state.count}</button>;
4  });
```

Codefragment 2.1: Die originale Qwik-Komponente

Ähnlich, wie die im Kapitel 2.3.2 vorgestellten Hooks werden diese Teile zu unterschiedlichen Zeitpunkten ausgeführt.

```
1  // Teil A: chunk-a.js
2  // -> bevor die Komponente gerendert wird
3  export const Counter_onMount = (props) => {
4      const store = useStore({ count: 0 });
5      return qrl('./chunk-b.js', 'Counter_onRender', [store, props]);
6  };
7  // Teil B: chunk-b.js
8  // -> wenn die Komponente gerendert wird
9  const Counter_onRender = () => {
10     const [store, props] = useLexicalScope();
11     return (
12         <button onClick$={qrl('./chunk-c.js', 'Counter_onClick', [store,
               props])}>{store.count}</button>
13     );
14 };
15 // Teil C: chunk-c.js
16 // -> wenn der click event abgefeuert wird
17 const Counter_onClick = () => {
18     const [store, props] = useLexicalScope();
19     return (store.count += props.step  1);
20 };
```

Codefragment 2.2: Von Qwik erstellte Teile

Nach der Optimierung ist ein resultierender HTML-Code im Browser zu sehen (Codefragment 2.3).

```
<html>
  <body q:base="/build/">
    <button q:obj="456, 123"
        on:click="./chunk-c.js#Counter_onClick[0,1]">0</button>
    <script>
      /*Qwikloader script*/
    </script>
    <script type="qwik/json">
      {...json...}
    </script>
  </body>
</html>
```

Codefragment 2.3: Im Browser dargestelltes HTML

Darin zu sehen ist, dass der Click-Listener (Kapitel 2.2.1) registriert wird. Wenn der Benutzer nun das Event abfeuert, wird die Datei `chunk-c.js` (Codefragment 2.2 - Zeile 15) nachgeladen. An diesem Punkt wird die Ausführung vom Qwikloader an den nachgeladenen Code übergeben. Dies geschieht, damit der Qwikloader so klein wie möglich sein kann, da er in den HTML-Code eingefügt wird. Die Methode `useLexicalScope()` ist dabei für das Abrufen des Speichers und der Props zuständig (vgl. Builder.io Inc, 2022, QRL)..

2.3.5 Serialisierung der Anwendung mit Qwik

Serialisierungen im Qwik Web-Framework können mit der Methode JSON.stringify() verglichen werden. Diese konvertiert einen JS-Wert in eine JSON-Zeichenkette (vgl. Mozilla Corporation, 2022c). Es ist wichtig, dessen Einschränkungen zu verstehen, wenn man Qwik-Anwendungen entwickelt. Wie Codefragment 2.4 zeigt, können Funktionen und Symbole nicht serialisiert werden. Mithilfe der in Kapitel 2.3.4 angesprochenen QRL stellt Qwik die Funktionen aber als URL-String dar. Diese können wiederum serialisiert werden.

```
console.log(JSON.stringify(
    { x: [10, new String('false'), new Boolean(false), undefined,
        function(){}, Symbol('')] }
));
// output: "{"x":[10,"false",false,null,null,null]}"
```

Codefragment 2.4: JSON.stringify()

Objekttypen, die mit Qwik serialisiert werden können:

- DOM-Referenzen
- Datumsangaben
- Funktionsabschlüsse, die in einer QRL (Kapitel 2.3.4) verpackt sind

Objekttypen, die mit Qwik *nicht* serialisiert werden können:

- Klassen
- Promises
- Streams

Das Framework stellt dem Entwickler Mechanismen zur Verfügung, mit dem er die Komponenten und Entitäten der Anwendungen so formulieren kann, dass sie serialisiert und dann fortgesetzt werden können (vgl. Builder.io Inc, 2022, Resumable vs. Hydration). Wenn z. B. die Komponente in Codefragment 2.5 analysiert wird, können verschiedenen Besonderheiten erkannt werden. Der Verweis auf exportierte Top-Level-Symbole (Codefragment 2.5 - Zeile 10) ist immer erlaubt, auch wenn der Wert nicht serialisierbar ist. Ein Verweis auf serialisierbare Inhalte ist ebenfalls erlaubt (Codefragment 2.5 - Zeile 11). Wenn allerdings auf einen nicht serialisierbaren Inhalt in der Komponente verwiesen wird (Codefragment 2.5 - Zeile 12), kommt es zu einem Error.

```
1  import { component$ } from "@builder.io/qwik";
2  export const topLevel = Promise.resolve('nonserializable data');
3
4  export const Greeter = component$(() => {
5    const captureSerializable = 'serializable data';
6    const captureNonSerializable = Promise.resolve('nonserializable data');
7
8    return (
9      <button onClick$={() => {
10         console.log(topLevel); // OK
11         console.log(captureSerializable); // OK
12         console.log(captureNonSerializable); // ERROR
13     }}/>
14   );
15 });
```

Codefragment 2.5: Grenzen der Serialisierung

$ ist eine Qwik Optimizer Markierungsfunktion, welche im Codefragment 2.6 dargestellt wird. Damit kann der Qwik Optimizer angewiesen werden, den Ausdruck in $(...) in eine von QRL (Kapitel 2.3.4) referenzierte Ressourcen zu extrahieren. Der ansonsten

nicht serialisierbare Aufruf (Codefragment 2.6 - Zeile 9) kann somit serialisiert werden (Codefragment 2.6 - Zeile 8).

```
1  import { $, component$ } from "@builder.io/qwik";
2
3  export const Greeter = component$(() => {
4    const saySomething$ = $(() => console.log("OK"));
5    const saySomething = () => (console.log("ERROR"));
6
7    return (
8      <button onClick$={saySomething$} /> // OK
9      <button onClick$={saySomething} /> // ERROR
10   );
11 });
```

Codefragment 2.6: Qwik Optimizer Markierungsfunktion $

2.3.6 Wiederaufnehmbar vs. Hydrierung

Qwik-Anwendungen basieren auf dem Schlüsselkonzept, dass sie von einem serverseitig (Kapitel 2.1.1) gerenderten Zustand aus Wiederaufnehmbar (englisch: *resumable*) sind. Die Wiederaufnahmefähigkeit lässt sich am besten mit der von Frameworks aktuell genutzten Hydrierung (Kapitel 2.2) vergleichen. Die dort beschriebene Abbildung 2.4 wird mit dem initialen Ablauf im Qwik Web-Framework ergänzt und verglichen.

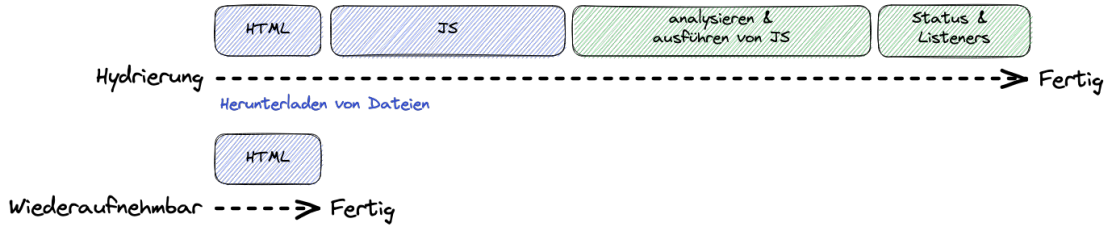


Abbildung 2.8: Wiederaufnehmbar vs. Hydrierung (Eigene Darstellung)

Bei der Wiederaufnahmefähigkeit geht es darum, die Ausführung auf dem Server anzuhalten und auf dem Client fortzusetzen, ohne die gesamte Anwendungslogik erneut abspielen und herunterladen zu müssen. Qwik-Anwendungen können zu jedem Punkt ihres Lebenszyklus serialisiert (Kapitel 2.3.5) und auf eine andere VM-Instanz (Server zu Browser) verschoben werden. Dort wird die Anwendung einfach fortgesetzt, wo die Serialisierung aufgehört hat. Es ist keine Hydrierung erforderlich. Dafür verantwortlich sind u. a. die

Annotationen, welche im gesamten HTML-Dokument zu sehen sind und von Qwik dafür genutzt werden (Abbildung 2.7). Aus diesem Grund hydrieren Qwik-Anwendungen nicht, sondern werden fortgesetzt (vgl. Builder.io Inc, 2022, Resumable vs. Hydration).

2.4 Geschwindigkeitsoptimierung

Viele Web-Frameworks stellen Tools zur Verfügung, um eine optimale Geschwindigkeit zu erreichen. So werden schon beim Kompilieren der Anwendung Textdateien komprimiert (Kapitel 2.4.2). Der Programmierer muss sich nur noch um die Bereitstellung der Daten kümmern, das Framework stellt sicher, dass alle Dateien in komprimierter Form vorliegen. Durch Plugins und Komponenten können außerdem Icons als Inline SVG in die Anwendung eingebunden werden (Kapitel 2.4.3).

2.4.1 Minimalisierung

Die für den Entwickler einfachste Optimierung ist die Minimalisierung (englisch: *Minification*). Denn diese wird beim Bau der Anwendung außer explizit deaktiviert, von jedem der Frameworks durchgeführt.

Die Verkleinerung von Textressourcen ist eine bewährte Praxis zur Verringerung der Dateigrößen. Dabei werden alle unnötigen Leerzeichen und Kommentare aus dem Quellcode entfernt, um die Übertragungsgröße zu verringern.

Ein weiterer Schritt, die sogenannte Uglifizierung (englisch: *Uglification*), wird auf JS angewandt. Dabei werden alle Variablen, Klassennamen und Funktionsnamen in einem Skript auf kürzere, unlesbare Symbole reduziert (vgl. NonStop io, 2022).

Kleinere Dateien bedeutet eine direkte Einsparung in der and den Browser zu übertragende Gesamtgröße.

2.4.2 Text Komprimierung mit GZIP

Um die Gesamtmenge der an die Benutzer übertragenen Daten zu reduzieren, sollten textbasierte Ressourcen mit Komprimierung bereitgestellt werden. GZIP ist ein Datenkomprimierungsalgorithmus, der Dateien schnell komprimieren und dekomprimieren kann. Es gibt zwei unterschiedliche Arten, die dynamische und die statische.

Dynamische Komprimierung (Abbildung 2.9) kann z. B. in einer Nginx-Konfiguration mit den Befehlen aus Codefragment 2.7 realisiert werden. Diese Art ist allerdings ein

sehr aufwendiger CPU-Intensiver Prozess. Deshalb können zwischen 1 und 9 unterschiedliche Stufen gewählt werden (Codefragment 2.7 - Zeile 3). Stufe 1 bietet die schnellste Kompressionsgeschwindigkeit, aber eine geringere Komprimierung. Für das höchste Kompressionsverhältnis mit einer niedrigeren Geschwindigkeit sollte die Stufe 9 gewählt werden.

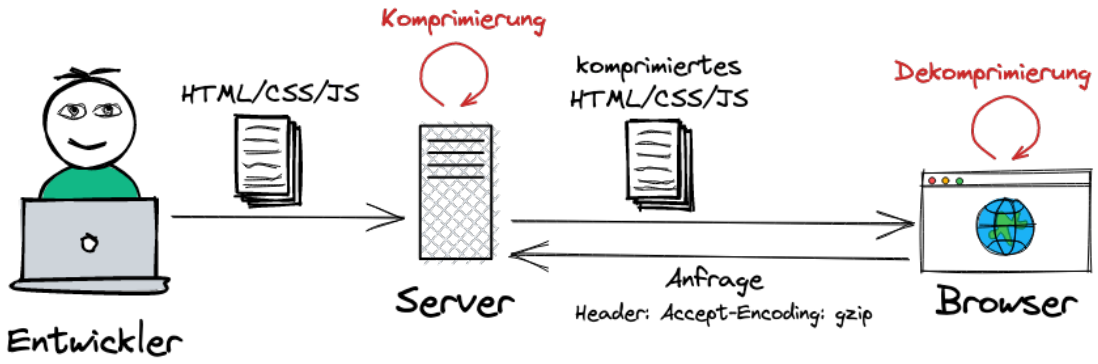


Abbildung 2.9: Dynamische Komprimierung (Eigene Darstellung)

Die minimale Länge der Antworten, die komprimiert werden sollen, kann in Bytes eingestellt werden (Codefragment 2.7 - Zeile 2). Nginx ermittelt diese aus dem *Content-Length*-Header der Antwort. Dateien, die nicht den angegebenen Multipurpose Internet Mail Extensions (MIME)-Typ haben, werden ignoriert (Codefragment 2.7 - Zeile 5). Antworten mit dem Typ *text/html* werden immer komprimiert (vgl. Nginx, 2022).

```
1 gzip            on;
2 gzip_min_length 1000;
3 gzip_comp_level 5;
4 gzip_proxied    any;
5 gzip_types      text/plain application/xml;
```

Codefragment 2.7: GZIP in Nginx Konfiguration

Es gibt zusätzlich die Möglichkeit, Dateien statisch vorzubereiten. Dabei werden mögliche Dateien schon vor der Bereitstellung auf dem Server vorkomprimiert (Abbildung 2.10). Oft wird dieser Schritt von dem jeweiligen Framework für den Entwickler erledigt. Diese Vorbereitung der Dateien benötigt mehr Speicherplatz auf dem Server. Als Vorteil kann aber im Gegensatz zur dynamischen Komprimierung die CPU entlastet werden. Bei statischen Dateien kann diese Art ohne Probleme angewandt werden. Sich ständig ändernde Ressourcen müssen nach der Erstellung auf dem Server dynamisch komprimiert werden.

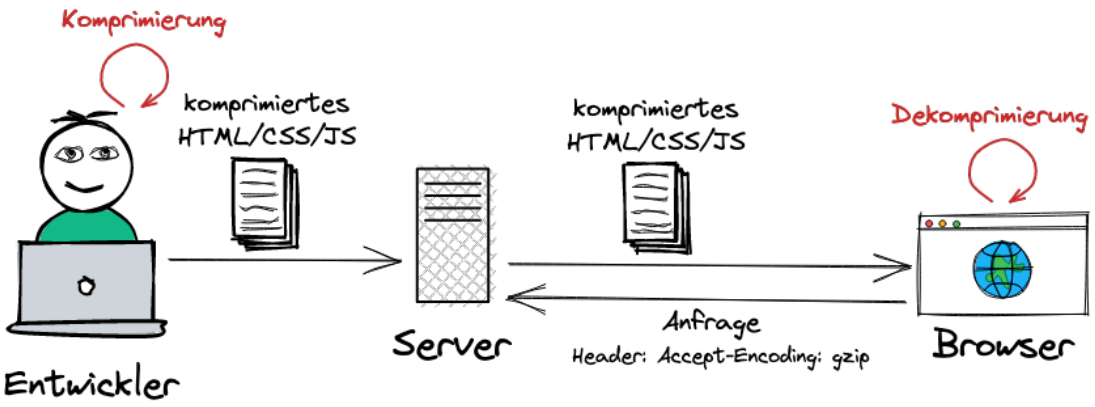


Abbildung 2.10: Statische Komprimierung (Eigene Darstellung)

Komprimierung kann vor allem bei CSS und JS Dateien von Vorteil sein. So muss z. B. bei der minimalen CSS-Datei von dem CSS-Framework Bootstrap[23] nach der Komprimierung anstatt 125.0 kB nur noch 20.7 kB übertragen werden. (vgl. SolarWinds Worldwide, 2018).

2.4.3 Eingebettete Grafiken

Eine Grafik im Scalable Vector Graphics (SVG) (deutsch: *skalierbare Vektorgrafik*)-Format kann unterschiedlich in ein HTML-Dokument eingefügt werden. Um zu verhindern, dass Grafiken mit einer zusätzlichen Anfrage nachgeladen werden, können sie direkt in das Dokument eingebettet werden (englisch: *Inline-SVG*). Nachträgliche Verschiebungen von Elementen im Dokument und Verzögerungen im Seitenaufbau können dadurch einfach verhindert werden. Dies macht sich besonders bemerkbar, wenn viele kleine Symbole auf einer Seite vorhanden sind. In den Fallstudien (Kapitel 3) werden ausschließlich eingebettete Grafiken im SVG-Format verwendet. Vor allem die Interaktion mit CSS sind wesentlich einfacher, da die Grafiken genauso behandelt werden kann wie alle anderen Elemente im Dokument. Dies ist ein entscheidender Vorteil, insbesondere für Interaktionen wie Hover-Effekte bei Buttons (vgl. Potschien, 2016).

23 getbootstrap.com

3 Vorgestellte Fallstudien

Verschiedene Anwendungen werden mit unterschiedlichen Anforderungen an die getesteten Framework programmiert. Jede Fallstudie soll dabei ein in der Praxis auftretendes Szenario nachstellen. Dabei wird darauf wert gelegt, wie aufwendig sie zu programmieren sind. Im Folgenden werden die einzelnen Fälle und deren Grundlagen vorgestellt.

3.1 Grundlagen der Fallstudien

Um gleiche Daten und eine kontrollierte Leistung für alle Frontends zu liefern, wird für alle ein selbst entwickelte Backend verwendet. Gleichzeitig wird ein reales Szenario mit Datenbank-Zugriff kreiert, in dem nicht nur auf In-Memory Daten zugegriffen wird. Das Frontend muss in manchen Fallstudien verschiedene Ressourcen aus dem Backend anfragen und je nach Anforderung direkt darstellen oder nachladen. Bei allen Fallstudien wird eine identische Konfiguration des CSS-Frameworks Tailwind[24] genutzt.

3.1.1 Kommunikation mit dem Backend

Zur Kommunikation wird im Backend eine Representational State Transfer (REST)-Application Programming Interface (API) bereitgestellt, die eine leichtgewichtige Möglichkeit der Integration mit dem Frontend bieten soll. Eine Anfrage wird vom Client an den Server in Form einer Web-URL als HTTP GET-, POST-, PUT-, PATCH- oder DELETE-Anfrage gesendet. Diese werden im folgenden unter dem Begriff Create Read Update Delete (CRUD) genannt.

Nach einer CRUD-Anfrage kommt eine Antwort vom Server in Form von JavaScript-Objekt-Notation (JSON) zurück. JSON ist zurzeit das beliebteste Format, das in Webdiensten verwendet wird (vgl. Bansal, 2022).

Das Backend wird auf dem gleichen Server wie die getesteten Frontend-Frameworks betrieben. Zusätzlich ist der Service auch mit Hypertext Transfer Protocol Secure (HTTPS)

24 tailwindcss.com

unter einer Web-Adresse zu erreichen. Dadurch können Anfragen sowohl in einem unverschlüsselten lokalen Netzwerk als auch über eine verschlüsselte Verbindung stattfinden.

3.1.2 Authentifizierung

Die Authentifizierung im Backend erfolgt durch einen geheimen Bearer-Token. Er wird bei sicherheitskritischen Anfragen im Header mitgesendet. Nur mit einem korrekten Geheimnis werden diese mit Informationen beantwortet. Andernfalls wird mit einem Fehler und dem dazugehörigen Hypertext Transfer Protocol (HTTP)-Code 401 (*Unautorisiert*)[25] geantwortet.

Diese Art der Authentifizierung ist für Produktivanwendungen *keineswegs sicher*. Denn selbst wenn HTTPS zur Verschlüsselung der gesamten Transportkommunikation eingesetzt wird, könnte ein Angreifer in Besitz des Tokens kommen. Dazu müsste er allerdings entweder die konsumierende Anwendung und/oder das API-Gateway kompromittieren (vgl. Weir; „Z“ Nemec, 2019, API bearer of token). Für das Testen der Fallbeispiele reicht diese Art der Authentifizierung aus.

3.1.3 Backend-Technologien

Für das Backend wird die Programmiersprache *Go*[26] und das Web-Framework *Gin*[27] verwendet. Persönliche Präferenz und u. a. schnelle Ausführung, Sicherheit und geringer Speicherplatzbedarf sind Gründe, das Backend in Go zu programmieren.

Zur Kommunikation mit der Datenbank wird *GORM*[28] genutzt. Als Datenbank kommt eine SQLite[29] zum Einsatz.

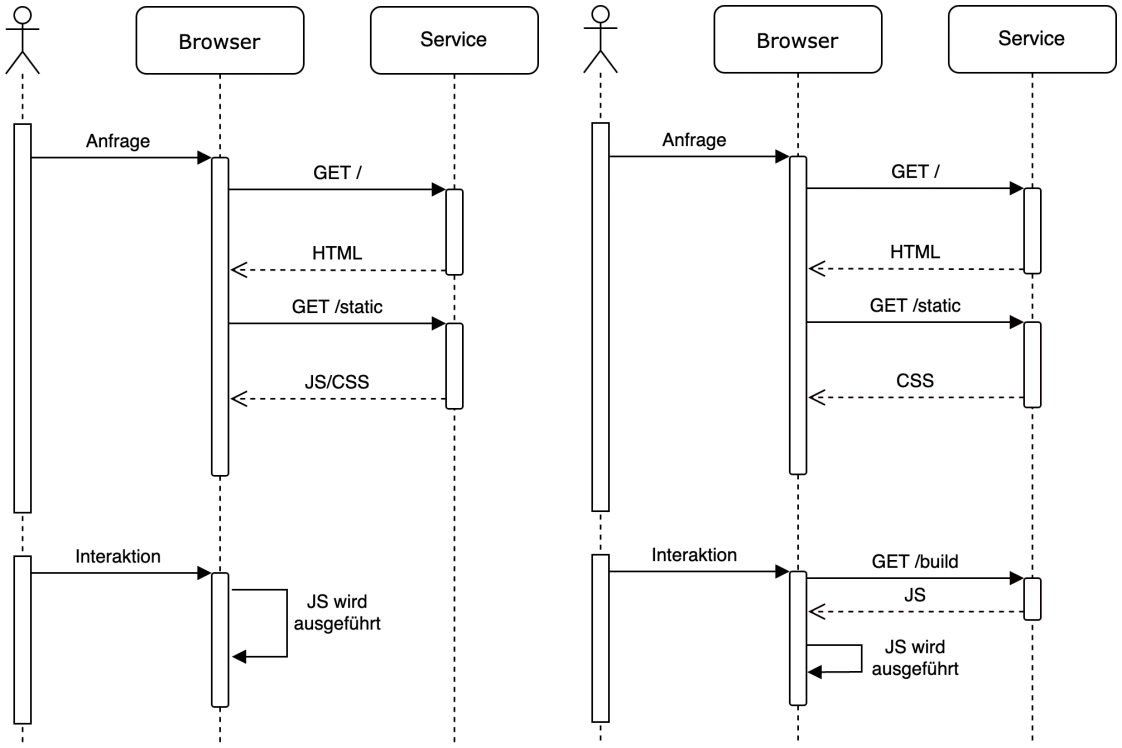
Das komplette Backend wird als Docker Container auf dem gleichen Server wie die Fallstudien bereitgestellt. Es liefert dabei Daten für sowohl Fallstudie 2: Länder (Kapitel 3.3) als auch Fallstudie 3: Blog (Kapitel 3.4).

25 developer.mozilla.org/en-US/docs/Web/HTTP/Status
26 go.dev
27 gin-gonic.com
28 gorm.io
29 sqlite.org

3.2 Fallstudie 1: Formular

Die erste Fallstudie konzentriert sich auf eine alleinstehende Frontend-Anwendung. Dadurch wird jeglicher externer Einfluss ausgeschlossen. Das Framework muss alle Dateien laden und im Browser für den Benutzer darstellen.

In einem Sequenzdiagramm kann gut gezeigt werden, welche Unterschiede zwischen Qwik und anderen Frameworks erwartet werden (Abbildung 3.1). Bei allen anderen Frameworks wird der verwendete Code beim initialen Laden im Browser heruntergeladen (Abbildung 3.1a). Im Gegensatz dazu wird bei dem Qwik Web-Framework der Code für die Validierungen in JS erst nach einer Interaktion des Nutzers nachgeladen (Abbildung 3.1b). Es wird bei Qwik eine schnellere Ladezeit der Anwendung erwartet. Das Laden von CSS unterscheidet sich zwischen den Frameworks nicht und wird kurz nach dem HTML als eine extra Datei geladen.

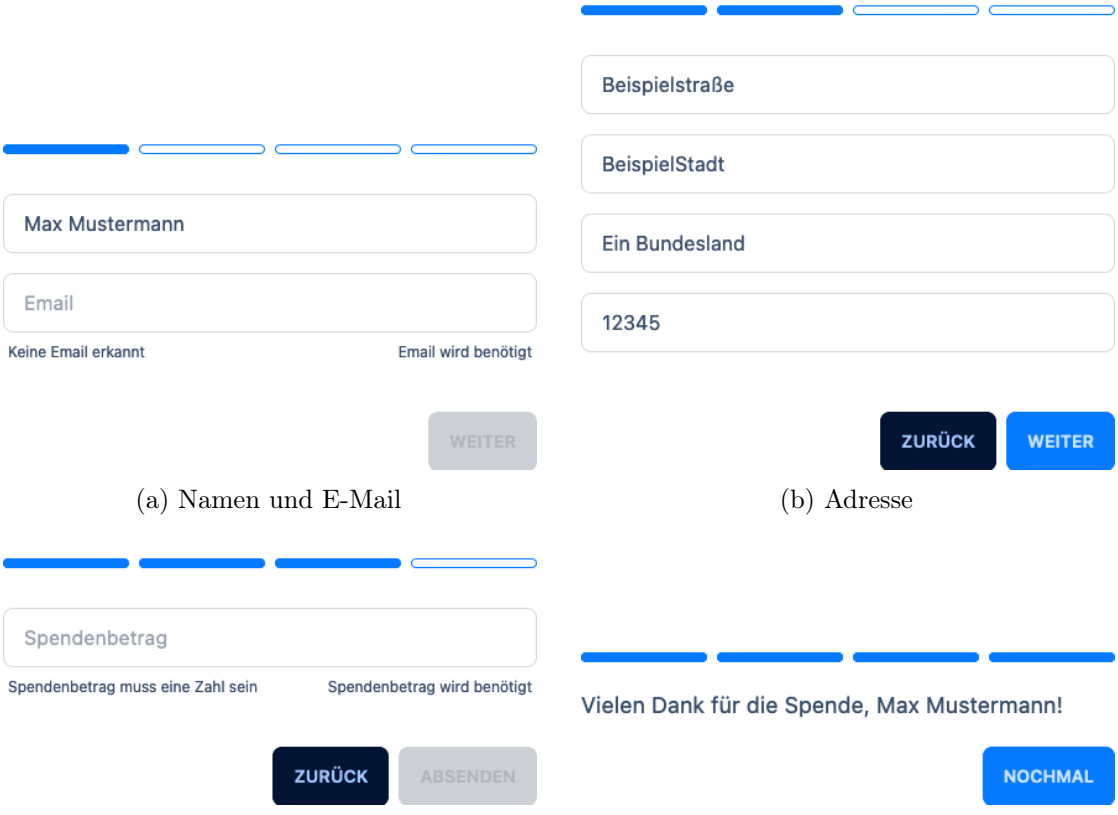


(a) Traditionelles Web-Framework (b) Qwik Web-Framework

Abbildung 3.1: UML Sequenzdiagramm Fallstudie 1: Formular (Eigene Darstellung)

Im Formular soll der Nutzer verschiedene Informationen eingeben können. Beim Eingeben werden die Inhalte auf Vollständigkeit überprüft. Die Inspiration für das Design und die Umsetzung wurde dabei aus einem Blog-Artikel für ein mehrstufiges Formular genommen (Kelly, 2022). In jeder Stufe werden Informationen abgefragt und nur bei erfolgreicher Eingabe kann der Nutzer zur nächsten Stufe gelangen (Abbildung 3.2).

In dieser Fallstudie wird vor allem die Aufmerksamkeit auf Interaktion mit der Anwendung gelegt. Die Validierung sowie die Eingabe von Daten wird dabei mit JS realisiert. Jeder vom Benutzer eingegebene Buchstabe speichert das Framework in einem zugehörigen Objekt. Bei einer Änderung müssen die jeweiligen Validierungen ausgeführt werden. Je nach Ergebnis der Validierung müssen Teile des DOM angepasst werden. Es findet keine Kommunikation mit einem separaten Backend statt.



(a) Namen und E-Mail

(b) Adresse

(c) Spendenbetrag

(d) Bestätigung

Abbildung 3.2: Darstellung des Formulars (Eigene Darstellung)

3.3 Fallstudie 2: Länder

Die zweite Fallstudie behandelt das Anzeigen von Länder-Flaggen mit zusätzlichen Informationen. Hier wird das Framework von einem separaten Backend mit einer großen Anzahl an Daten versorgt.

Diese Anzahl von Komponenten (250 Länder) sollen angezeigt werden (Abbildung 3.4). Da keine Aufteilung der Länder auf verschiedene Seiten erfolgt, muss das Framework beim Aufruf 250-mal alle Länder-Komponenten darstellen.

Nach dem Laden der Anwendung kann zusätzlich mit JS nach Ländern gesucht und die Liste alphabetisch sortiert werden.

Ohne Optimierung der Seitengröße wird hier bei jedem der getesteten Frameworks eine schlechte Leistung erwartet. Vor allem die Hydrierung der Anwendung wird hier eine Verzögerung verursachen (Kapitel 2.2).

Ein großes DOM beeinträchtigt u. a. auch die Speicher- und Laufzeitleistung. Jede Interaktion kann sich deswegen je nach Rechenleistung des Endgerätes verzögern (vgl. Chrome Developers, 2019).

Bei dem Qwik Web-Framework wird aber durch den Einsatz des Qwikloader (Kapitel 2.3.3) und keinerlei Hydrierung ein etwas besseres Ergebnis erwartet (Kapitel 2.3.6).

Das Sequenzdiagramm zeigt zu welchem Zeitpunkt die Anfragen bei Qwik stattfinden (Abbildung 3.3). Durch das Nachladen von JS im Vergleich zu anderen Frameworks kann hier möglicherweise Zeit eingespart werden.

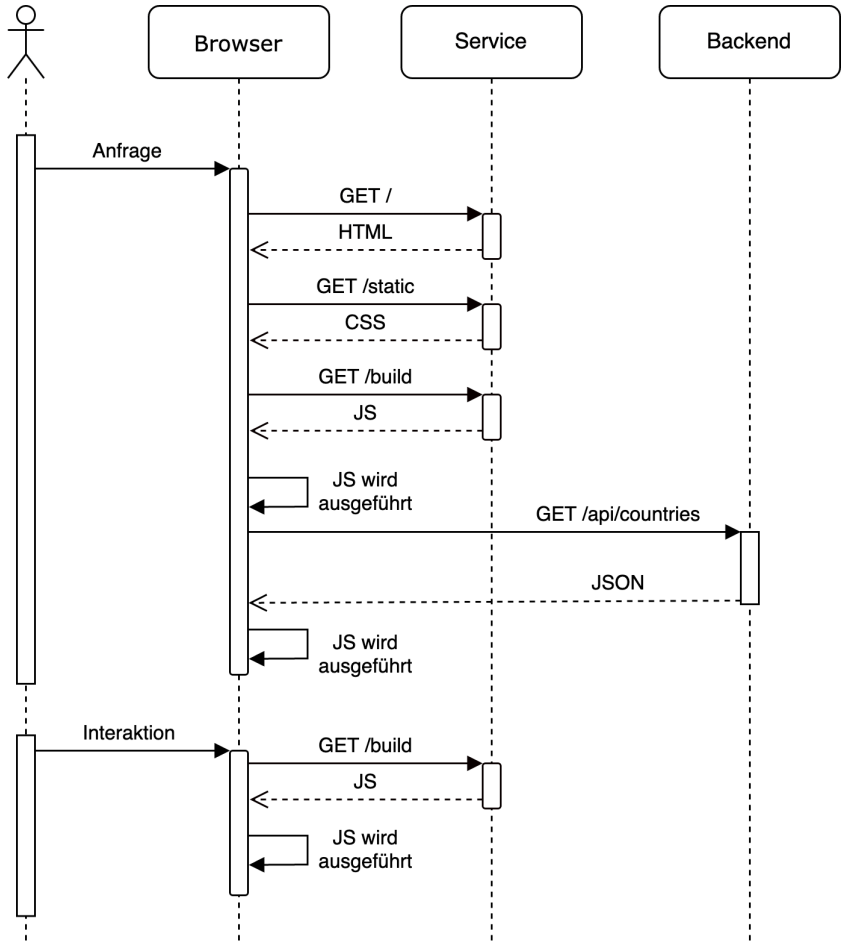
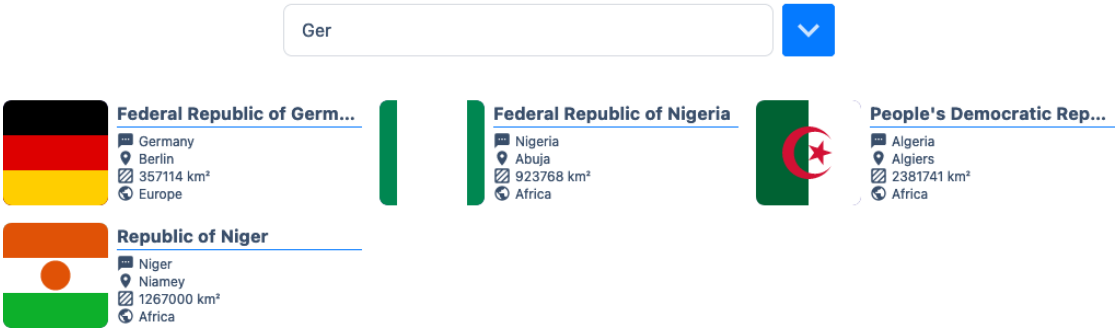


Abbildung 3.3: UML Sequenzdiagramm Fallstudie 2: Länder mit dem Qwik Web-Framework (Eigene Darstellung)

(a) Sortiert



(b) Suchen

Abbildung 3.4: Darstellung der Länder (Eigene Darstellung)

Mit dieser Fallstudie kann das Darstellen von vielen Inhalten mit Filter- und Suchfunktion getestet werden. Inhalte müssen außerdem von einer externen Quelle abgerufen und dargestellt werden. Da sich der Inhalt aus dem Backend ändern könnte, kann die Fallstudie in keine statische Seite umgewandelt werden. Das initiale Rendering soll, wenn möglich, vom Server übernommen werden. Nachträgliche Änderungen am DOM werden mit JS im Browser gerendert.

Die im Backend gespeicherte Informationen stammen von *REST Countries*[30]. Bei jedem Neustart des Backends werden diese Daten über eine REST-API aktualisiert (Codefragment 3.1).

```
1 https://restcountries.com/v3.1/all?fields=name,continents,capital,area,flags
```

Codefragment 3.1: REST Countries

Initial wird die Antwort von *REST Countries* geparst und im Speicher des Backends gehalten. Damit können unterschiedliche Antwortzeiten reduziert werden. Der vorhandene Endpunkt wird ohne Authentifizierung für jedes Framework zur Verfügung gestellt und liefert alle geparsten Länder. Jedes Land enthält den Namen, den Kontinent, die Hauptstadt, die Ländergröße und einen Link für das Anzeigen der Flagge (vgl. REST Countries, 2022).

3.4 Fallstudie 3: Blog

Als dritte Fallstudie wird ein Blog programmiert. Damit wird ein realistischer Praxiseinsatz von viel JS getestet. Es kommt Formular-Validierung, Routing, Authentifizierung und das Darstellen von externen Inhalten zum Einsatz (Abbildung 3.5 & Abbildung 3.6).

Artikel sollen angezeigt, kreiert und gelöscht werden können. Jegliche Bearbeitung des Blogs wird durch ein Bearer-Token abgesichert (Kapitel 3.1.2). Die Artikel können vorzeitig mit SSR dargestellt werden. Um Änderungen durchzuführen, wird allerdings eine erfolgreiche Authentifizierung benötigt. Mehr Details des Artikels kann auf einer eigenen Seite angezeigt werden. Auf diese Seite soll mit der Identifikationsnummer des Artikels navigiert werden. Jedes getestete Framework stellt dafür einen Router zur Verfügung.

Jede CRUD-Aktion startet eine separate Anfrage an das Backend. Nach erfolgreicher Validierung des Bearer-Tokens wird die Anfrage bearbeitet und mit JSON beantwortet (Abbildung 3.5).

Hier wird von jedem Test eine ungefähr gleiche Leitung erwartet. Qwik kann durch das verzögerte Laden von JS einen Geschwindigkeitsvorteil haben. Außerdem sollte, wie schon in der Fallstudie Länder (Kapitel 3.3), die Wiederaufnehmbarkeit das initiale Laden beschleunigen.

30 restcountries.com

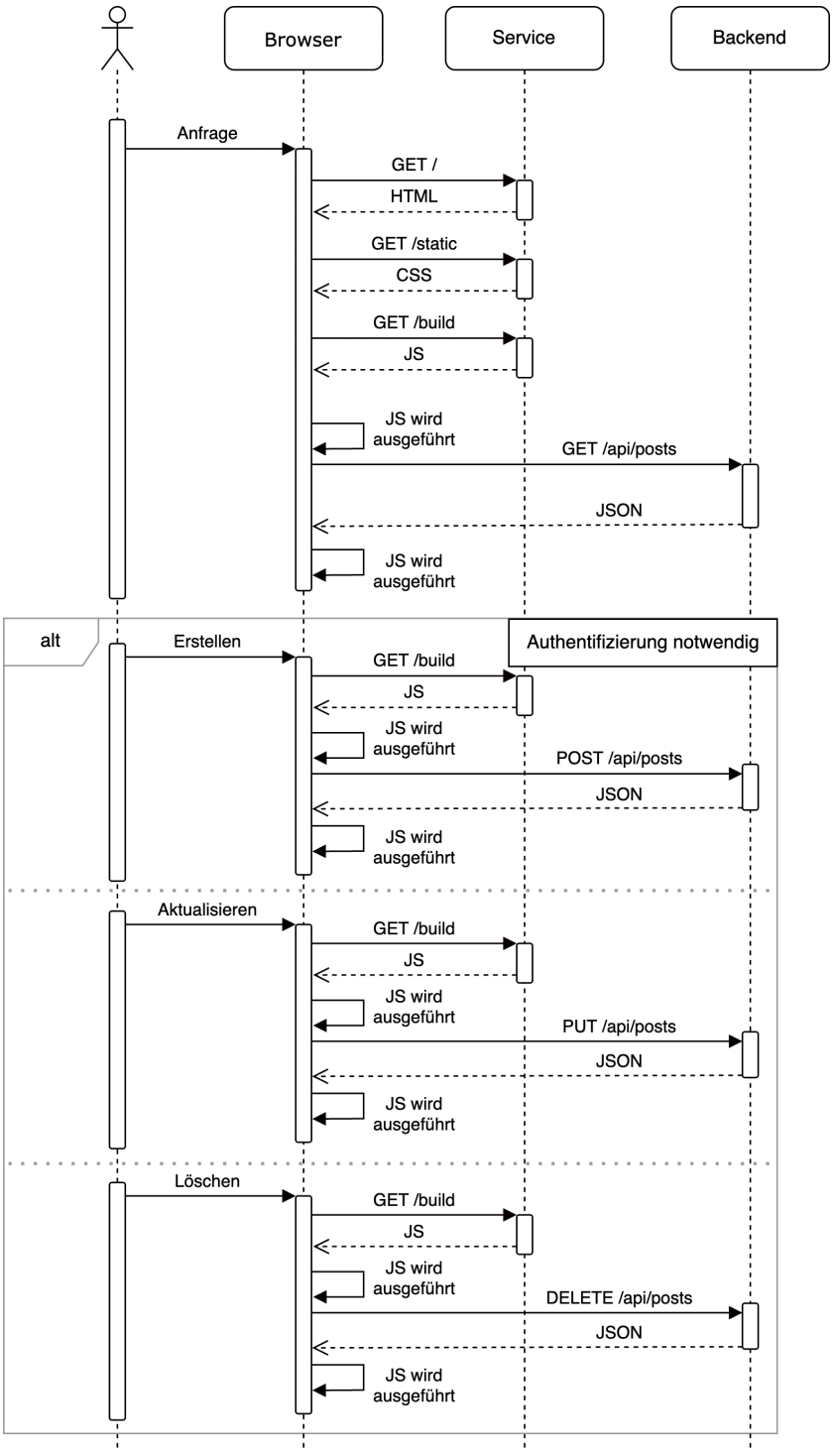
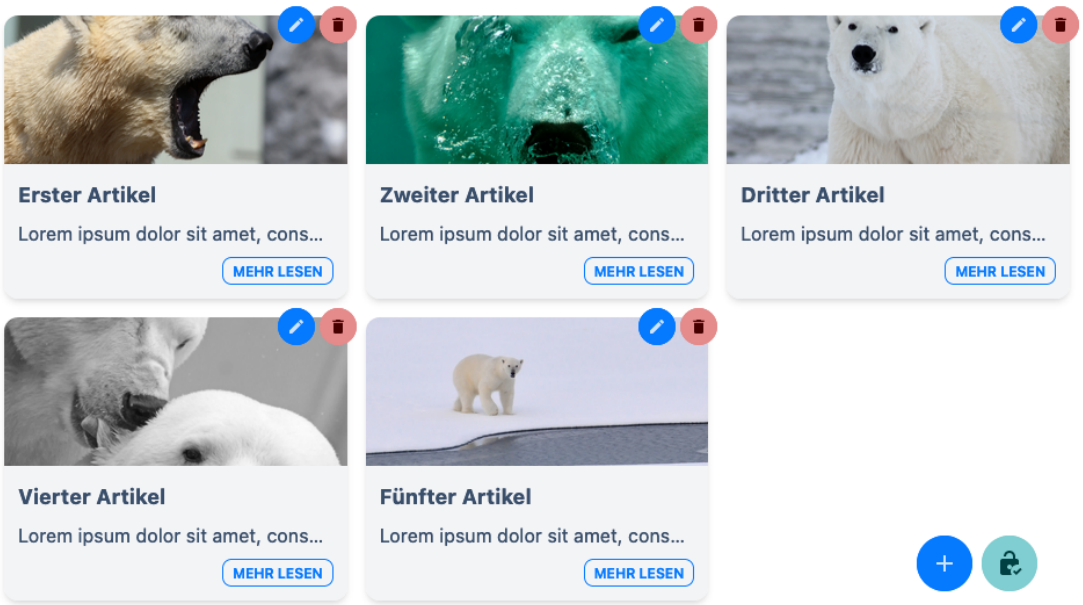
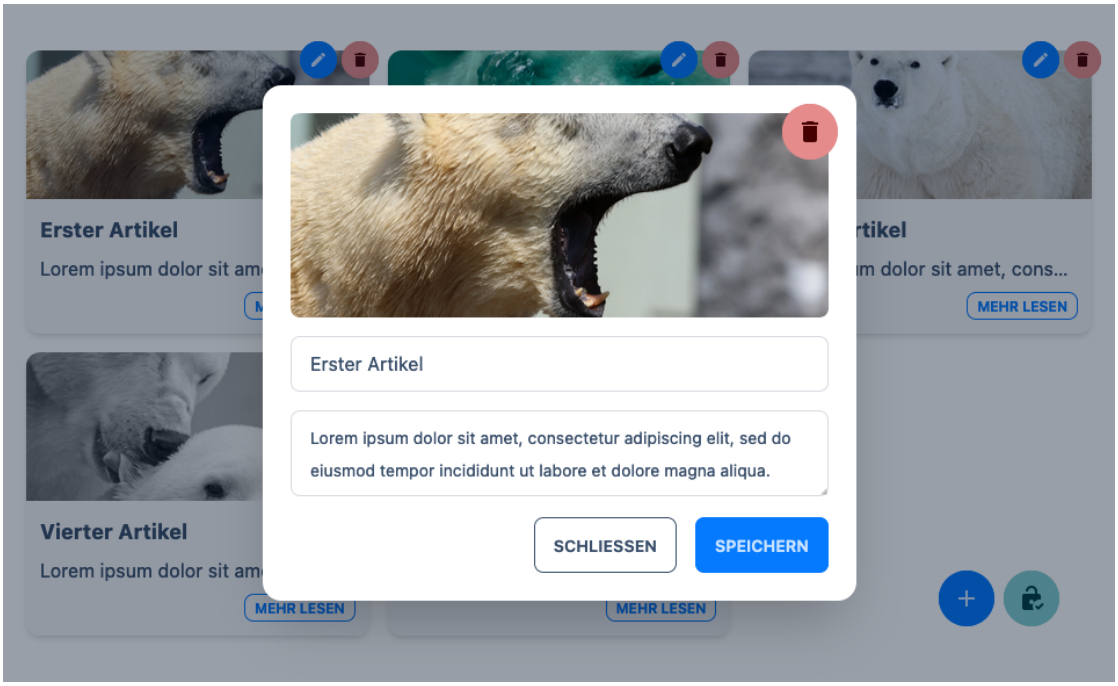


Abbildung 3.5: UML Sequenzdiagramm Fallstudie 3: Blog mit dem Qwik Web-Framework (Eigene Darstellung)



(a) Übersicht



(b) Bearbeiten eines Artikel

Abbildung 3.6: Darstellung des Blog (Eigene Darstellung)

4 Umsetzung der Fallstudien

In den folgenden Kapiteln wird besondere Aufmerksamkeit auf das Qwik Web-Framework gelegt. Bei den anderen genutzten Frameworks werden nur auffallende Merkmale erläutert. Zu jeder Umsetzung wird die dafür genutzte Version, deren Veröffentlichungsdatum und die offizielle Anleitung angegeben. Es wird eine komponentenbasierte Entwicklung angewandt.

4.1 Umsetzung mit dem Qwik Web-Framework

Version: **0.16.1**[31]

Veröffentlichungsdatum: **17.12.2022**

Anleitung: **qwik.builder.io/docs/getting-started**

Um ein Qwik Projekt zu initialisieren, wird eine Installation von Node.js[32] in der Version 16 oder höher benötigt. Zum Erstellen eines neuen Projektes wird das Qwik-Command Line Interface (CLI) verwendet (Abbildung 4.1a).

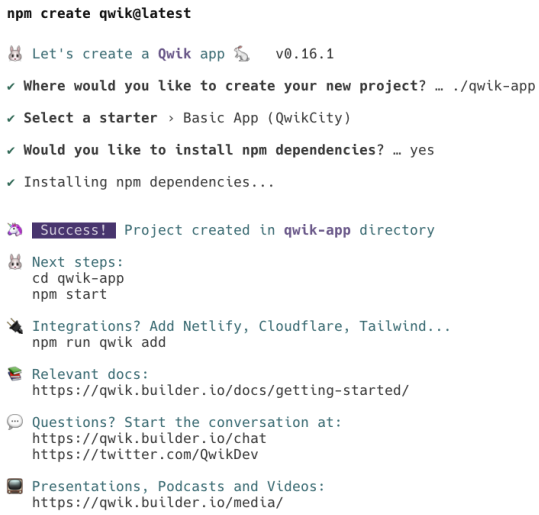
Je nach Anwendungsfall können schon bei der Initialisierung verschiedene Start-Templates ausgewählt werden (Abbildung 4.1c). Qwik City (Kapitel 4.1.1), ein begleitendes Meta-Framework, ist bei den Templates direkt integriert. Ein *Basic App*-Template ist z. B. eine Beispielseite, welche die Nutzung von Komponenten und eines Routers zeigt. Die Qwik-CLI erstellt für das gewählte Template eine initiale Ordnerstruktur mit allen nötigen Dateien (vgl. Builder.io Inc, 2022, Getting Started Qwikly).

Eine Besonderheit ist dabei, dass verschiedene Integrationen direkt über das Qwik-CLI hinzugefügt werden können (Abbildung 4.1d). Nach dem Hinzufügen von z. B. dem CSS-Framework *Tailwind* werden außerdem die notwendigen Schritte beschrieben, um die Integration vollständig nutzen zu können (Abbildung 4.1b).

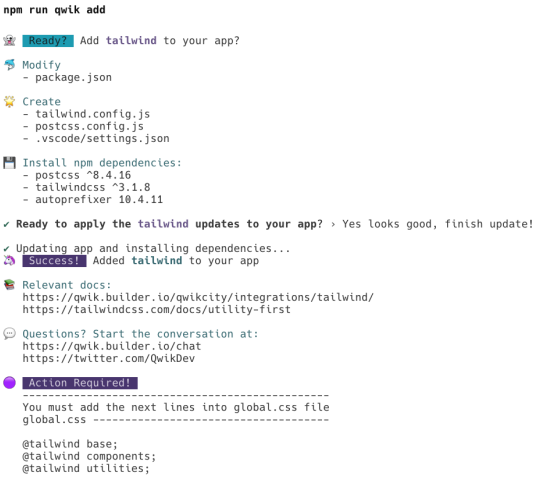
Um die Anwendung in der Entwicklung zu starten, wird ein Vite[33] Development Server

31 github.com/BuilderIO/qwik/releases/tag/v0.16.1
32 nodejs.org
33 vitejs.dev

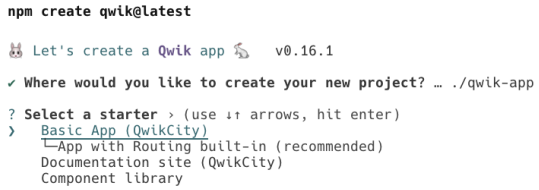
mit `npm start` ins Leben gerufen. Wenn der Port noch nicht genutzt wird, kann die Anwendung dann unter der lokalen Adresse localhost:5173 aufgerufen werden.



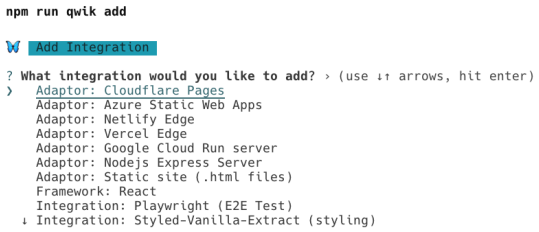
(a) Neues Qwik Projekt erstellen



(b) Hinzufügen von Tailwind



(c) Auswahl eines Templates



(d) Liste der verfügbaren Integrationen

Abbildung 4.1: Qwik-CLI (v0.16.1) (Eigene Darstellung)

4.1.1 Das Qwik City Meta-Framework

Um eine E-Commerce-Website, eine Blog-Site oder eine andere Website zu erstellen wird mit Qwik City ein begleitendes Meta-Framework benötigt. Es unterstützt Routing, Layouts, Templates und Datenabruf/-aktualisierungen. Routing kann dabei durch das Erstellen von Dateien und Ordnern realisiert werden. Dabei wird der Inhalt des Verzeichnisses *routes/* von Qwik analysiert und bei richtiger Benennung automatisch als Route hinzugefügt. Dynamische Routen werden mit eckigen Klammern (z. B. `[id]`) deklariert. Dieser Parameter kann dann in einer Qwik-Komponente ausgewertet werden. Anschließend ein Beispiel, wie so eine Struktur aussehen kann:

```
—src/
  └routes/              # Inhalt dieses Ordners wird analysiert
    ├docs/
    │ ├overview/
    │ │ └index.mdx      # https://example.com/docs/overview
    │ ├article/
    │ │ └index.tsx      # https://example.com/docs/article
    │ │ └[id].tsx       # https://example.com/docs/article/1
    │ └index.mdx        # https://example.com/docs
    ├index.tsx          # https://example.com/
    └layout.tsx         # Alle seiten nutzen dieses Layout
```

Komponenten und Seiten im Qwik Web-Framework werden normalerweise mit als TypeScript with syntax extension (TSX)-Dateien gespeichert. Eine alternative Möglichkeit zur Erstellung von Inhalten ist die Verwendung von Markdown with syntax extension (MDX)-Dateien. Diese Dateien werden wie Markdown verfasst, aber in Qwik-Komponenten kompiliert. Neben der Markdown-Syntax können MDX-Dateien auch auf andere Komponenten verweisen (vgl. Builder.io Inc, 2022, Qwik City).

4.1.2 Besonderes in der Umsetzung

Um die Fallstudien zu realisieren, wird bei der jeweiligen Initialisierung des Projektes das *Basic App*-Template genutzt. Obwohl in zwei Fallstudien (Fallstudie 1: Formular - Kapitel 3.2 und Fallstudie 2: Länder - Kapitel 3.3) kein Routing benötigt wird, gibt es kein Template, das ohne Qwik-City funktioniert.

Das mehrmalige Verwenden von Komponenten ist ohne Probleme möglich. Bei der Erstellung von Komponenten trifft man schnell auf das Problem, dass Interaktionen in wiederverwendbare Funktionen ausgelagert werden müssen. Hier wird die Entwicklung durch die Serialisierbarkeit (Kapitel 2.3.5) erschwert. Diese Funktionen müssen mit einer QRL (Kapitel 2.3.4) nachgeladen und dementsprechend angepasst werden. Damit kann die Ausführung und der Download von JS so lange wie möglich verzögert werden.

Auch wird das externe Laden von Inhalten durch das SSR der Anwendung erschwert. Die in der Dokumentation beschriebenen Methoden zur Datenbeschaffung werden von Qwik in der genutzten Version vorgerendert und somit nicht vom Backend aktualisiert. Damit werden bei der Fallstudie 2 (Kapitel 3.4) immer die gleichen Artikel geladen, obwohl sich der Blog im Backend verändert hat. Da sich dieses Verhalten in der Entwicklungsumgebung nicht nachstellen lässt, sondern nur in der final gebauten Anwendung auftritt, war dies schwer zu analysieren. Fertiggestellte Komponenten mussten nachträglich aufwendig umgebaut werden.

Im Entwicklungsmodus stellt Qwik eine Funktion im Browser bereit, um direkt in die jeweilige Komponente zu springen. Beim Halten der Taste *Alt* und einem Klick auf die Komponente öffnet und navigiert die Integrierte Entwicklungsumgebung (IDE) an die jeweilige Stelle im Code (Abbildung 4.2).

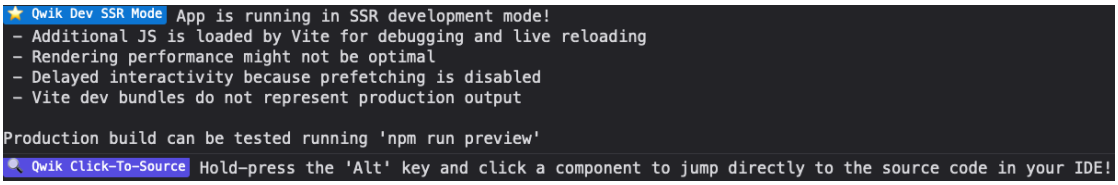


Abbildung 4.2: Qwik Entwicklungsmodus (Eigene Darstellung)

Eine weitere Besonderheit ist die Funktion eines Context. Damit können über mehrere verschachtelte Komponenten hinweg Informationen weitergegeben und angepasst werden. Ein Context kann mit einer Sammlung an globalen Variablen verglichen werden. Da es noch keine Unterstützung eines globalen Speichers (englisch: *Store*) in Qwik gibt, wurde für die Authentifizierung darauf zurückgegriffen. Bei anderen Frameworks kann dies übersichtlicher mit einem Store (z. B. React Redux[34] oder Pinia[35]) erreicht werden.

4.1.3 Bereitstellungsmöglichkeiten von Qwik

Die Qwik City Middleware verbindet Server-Rendering-Frameworks (wie Cloudflare[36], Netlify[37], Vercel[38], Express[39] usw.) mit dem Qwik City Meta-Framework. Dadurch kann die geschriebene Anwendung einfach und unkompliziert publiziert werden. Auf die einzelnen Services wird in dieser Arbeit nicht eingegangen. Für eine Bereitstellung mit Docker auf einem eigenen Server wird für diese Arbeit die SSG Middleware gewählt (vgl. Builder.io Inc, 2022, Static Site Generation Config). Das Qwik Web-Framework stellt dafür einen Adapter bereit, um die Anwendung umzuwandeln. Danach kann eine mit Qwik geschriebene Anwendung mit z. B. Nginx bereitgestellt werden (Kapitel 4.3).

34 react-redux.js.org
35 pinia.vuejs.org
36 cloudflare.com
37 netlify.com
38 vercel.com
39 expressjs.com

4.2 Umsetzung mit anderen Frameworks

Andere Frameworks werden zum Vergleich mit dem Qwik Web-Framework eingesetzt. In dieser Arbeit wird nicht auf Details der Umsetzung mit den jeweiligen Frameworks eingegangen.

4.2.1 Besonderes in der Umsetzung mit Vue

Version: **3.2.41**[40]

Veröffentlichungsdatum: **14.10.2022**

Anleitung: **vuejs.org/guide/quick-start.html**

„Vue ist ein JS-Framework zur Erstellung von Benutzeroberflächen. Es baut auf Standard-HTML, -CSS und -JS auf und bietet ein deklaratives und komponentenbasiertes Programmiermodell, das hilft, einfache oder komplexe Benutzeroberflächen effizient zu entwickeln.“ (vgl. You, 2014)

Für Vue muss jede Qwik Komponente fast neu geschrieben werden. Im Gegensatz zu React und Qwik verwendet Vue keine TSX-Dateien. Somit müssen Komponenten-Ereignisse umgeschrieben werden und for-Schleifen angepasst werden. Ebenso werden If-Abfragen mit eingebauten HTML-Tags (v-if, v-else, v-else-if, usw.) genutzt.

Wie auch Qwik unterstützt Vue *Slots*. Damit können Komponenten oder HTML an einer bestimmten Stelle in einer Komponente dazwischengeschoben werden.

Zusätzlich können Ereignisse in Vue von einem Komponenten *emittiert* werden. Bei Qwik wird dies über Props und *PropFunctions* gelöst.

Eine extra Option ist das *Computed Property* welches alle benötigten reaktiv und veränderlichen Variablen beobachtet und daraus einen neuen Wert berechnet. Dies kann in Qwik nur durch vom Programmierer manuell gesetzte Tracker nachgebaut werden (Codefragment 4.1). In den Fallstudien zeigte sich ein *Computed Property* mit viel weniger Code (Abbildung 4.2).

40 github.com/vuejs/core/releases/tag/v3.2.41

```
const exampleStore = useStore({ myVariable: 1, myComputed: true });
useTask$(({ track }) => {
    track(() => exampleStore.myVariable);
    exampleStore.myComputed = exampleStore.myVariable > 2;
}
console.log(exampleStore.myComputed); // Ausgabe: false
```

Codefragment 4.1: Der useTask$ Hook bei Qwik

```
const exampleStore = reactive({ myVariable: 1 });
const myComputed = computed(() => {
    return exampleStore.myVariable > 2;
}
console.log(myComputed.value); // Ausgabe: false
```

Codefragment 4.2: Die Computed Property bei Vue

Tailwind muss manuell über die offizielle Anleitung von Tailwind nachinstalliert werden (vgl. Tailwind Labs Inc., 2022c).

4.2.2 Besonderes in der Umsetzung mit Nuxt

Version: **3.0.0**[41]

Veröffentlichungsdatum: **16.11.2022**

Anleitung: **nuxt.com/docs/getting-started/installation**

„Nuxt ist ein Open-Source-Framework unter MIT-Lizenz zur Erstellung moderner und performanter Webanwendungen, die auf jeder Plattform mit JS eingesetzt werden können. Nuxt verwendet Vue.js als View-Engine. Alle Fähigkeiten von Vue 3 sind in Nuxt verfügbar.“ (vgl. Nuxt Project, 2022)

Nachdem alle Komponenten auf Vue angepasst sind, ist der Umstieg auf Nuxt einfach. Lediglich die Art des Routing muss angepasst werden. Nuxt setzt wie Qwik (Kapitel 4.1.1) auf eine identische dynamische Erstellung der Routen über einen Ordner (*pages/*). Routen-Parameter können so auch mit eckigen Klammern (z. B. `[id]`) im Dateinamen genutzt werden.

Wie auch schon bei Vue wird auf einen Pinia-Store[42] zurückgegriffen, um Komponenten-übergreifend Daten zu teilen.

41 github.com/nuxt/framework/releases/tag/v3.0.0
42 pinia.vuejs.org

Tailwind kann ebenso wie bei Vue, manuell mit der offiziellen Anleitung, nachinstalliert werden (vgl. Tailwind Labs Inc., 2022b).

4.2.3 Besonderes in der Umsetzung mit React

Version: **18.2.0**[43]

Veröffentlichungsdatum: **14.06.2022**

Anleitung: **reactjs.org/docs/create-a-new-react-app.html**

„React is a JavaScript library for building user interfaces.“ (vgl. Meta Platforms, Inc., 2022)

React ist von den vorgestellten Frameworks das am meisten genutzte. Laut Web Almanac 2022 wird es von 8 % aller Webseiten eingesetzt (vgl. HTTP Archive, 2022b, Libraries and frameworks).

Wie auch Qwik nutzt React TSX-Dateien. Somit können viele Konzepte und Komponenten mit wenig Anpassung übernommen werden. Qwik vergleicht die Art, Komponenten zu erstellen direkt mit React (vgl. Builder.io Inc, 2022, Qwik vs React).

QwikReact ermöglicht außerdem die Verwendung von React-Komponenten in Qwik, einschließlich des gesamten Ökosystems der Komponentenbibliotheken. Dazu kann in dem Qwik-CLI (Abbildung 4.1) React hinzugefügt werden.

Viele Dinge konnten direkt von Qwik übernommen werden. Anstelle eines Context wurde ein Redux-Store[44] integriert, um Komponentenübergreifend auf Daten zuzugreifen. Dies ist übersichtlicher und unterstützt die Verwendung von Datentypen besser.

Tailwind muss wie auch schon bei allen anderen Frameworks manuell nachinstalliert werden (Tailwind Labs Inc., 2022a).

43 github.com/facebook/react/releases/tag/v18.2.0
44 react-redux.js.org

4.3 Bereitstellung der Fallstudien

Jede Anwendung wird gebaut und in einem Docker-Container mit Nginx und identischen Einstellungen bereitgestellt (Codefragment 4.3). Dynamische Komprimierung (Kapitel 2.4.2) wird dabei in jeder Konfiguration ab 1000 Byte aktiviert (Codefragment 4.3 - Zeile 8) und ein Cache-Control Header hinzugefügt (Codefragment 4.3 - Zeile 16).

```
1  server {
2    listen 80;
3    root   /usr/share/nginx/html;
4    access_log off;
5    error_log off;
6
7    gzip on;
8    gzip_min_length 1000;
9    gzip_proxied any;
10   gzip_types text/plain application/javascript text/css;
11
12   location = / {
13     try_files $uri $uri/ /index.html;
14   }
15   location ~ \.(?!html) {
16     add_header Cache-Control "public, max-age=31536000";
17   }
18 }
```

Codefragment 4.3: Nginx Konfiguration der Fallstudien

Das dazugehörige Dockerfile nutzt die stabile offizielle auf Alpine basierende Version von Nginx. Alpine Linux ist eine sicherheitsorientierte, leichtgewichtige Linux-Distribution, die auf musl libc und busybox basiert (vgl. Alpine Linux Development Team, 2022). Je nach Framework muss hier nur der Bau-Befehl angepasst werden (Codefragment 4.4 - Zeile 9).

```
1  FROM node:alpine AS build
2  WORKDIR /frontend
3
4  COPY package.json .
5  COPY package-lock.json .
6  RUN npm install
7
8  COPY . .
9  RUN npm run build
10
11 FROM nginx:stable-alpine AS final
12
13 COPY --from=build /frontend/dist/ /usr/share/nginx/html/
14 COPY nginx.conf /etc/nginx/conf.d/default.conf
15 EXPOSE 80
16
17 CMD ["nginx", "-g", "daemon off;"]
```

Codefragment 4.4: Fallstudien Dockerfile

5 Benchmark und Analyse

Die erstellten Fallstudien (Kapitel 3) werden zuerst mit Google Lighthouse verglichen (Kapitel 5.2). Dabei werden viele unterschiedliche Leistungskennzahlen gesammelt (Kapitel 5.2.1). Entwickler des Qwik Web-Framework nutzten selbst dieses Tool, um die Vorteile des Frameworks hervorzuheben.

Danach werden in jeder Fallstudie automatisiert alle Interaktionen ausgeführt (Kapitel 5.3). Dabei soll vor allem das progressive Nachladen von Qwik getestet werden. Bei diesem Test kann u. a. die gesamte Anzahl an heruntergeladenem JS-Dateien mit anderen Frameworks verglichen werden.

Während der Durchführung werden von dem genutzten Reverse-Proxy verschiedene Metriken gesammelt (Kapitel 5.1.1).

Zusätzlich wird die aus dem Bau resultierenden Ordnergröße und die Anzahl der vom Framework erstellten Dateien analysiert (Kapitel 5.4.4).

5.1 Grundlagen und Vorbereitung

Um die Tests einfach und automatisiert durchführen zu können, werden in den folgenden Kapiteln mehrere Docker Container erstellt. Damit können über eine lange Laufzeit und in einer kontrollierten Umgebung Leistungskennzahlen gesammelt werden. Um Netzwerk-Engpässe auszuschließen, werden die Container dann auf einem identischen Server ausgeführt. Es werden sowohl praxisnahe Test über einem Reverse-Proxy (Kapitel 5.1.1) als auch Tests im lokalen Netzwerk durchgeführt. Sie laufen über einen langen Zeitraum, damit Belastungen des Servers im Durchschnitt ausgeglichen werden können. Alle Fallstudien werden somit unter identischen Voraussetzungen betrieben und analysiert.

5.1.1 Bereitstellung der Fallstudien hinter einem Reverse-Proxy

Um die Fallstudien (Kapitel 3) praxisnah analysieren zu können, werden sie auf dem gleichen Server hinter einem identischen Reverse-Proxy bereitgestellt. Dieser kann u. a. Benutzeranfragen entgegennehmen und sie an die beabsichtigte Stelle weiterleiten. Für die

Fallstudien wird ein Traefik Proxy[45] genutzt. Dabei kann auf z. B. mehrere Docker Container auf dem gleichen Server, aber unter unterschiedlicher Domain zugegriffen werden (Traefik Labs, 2022). Der Zugriff kann sowohl mit Transport Layer Security (TLS) 1.3 als auch unverschlüsselt lokal erfolgen. Wenn auf die Anwendungen über HTTPS zugegriffen wird, kümmert sich Traefik automatisch um die benötigten Zertifikate (Abbildung 5.1).

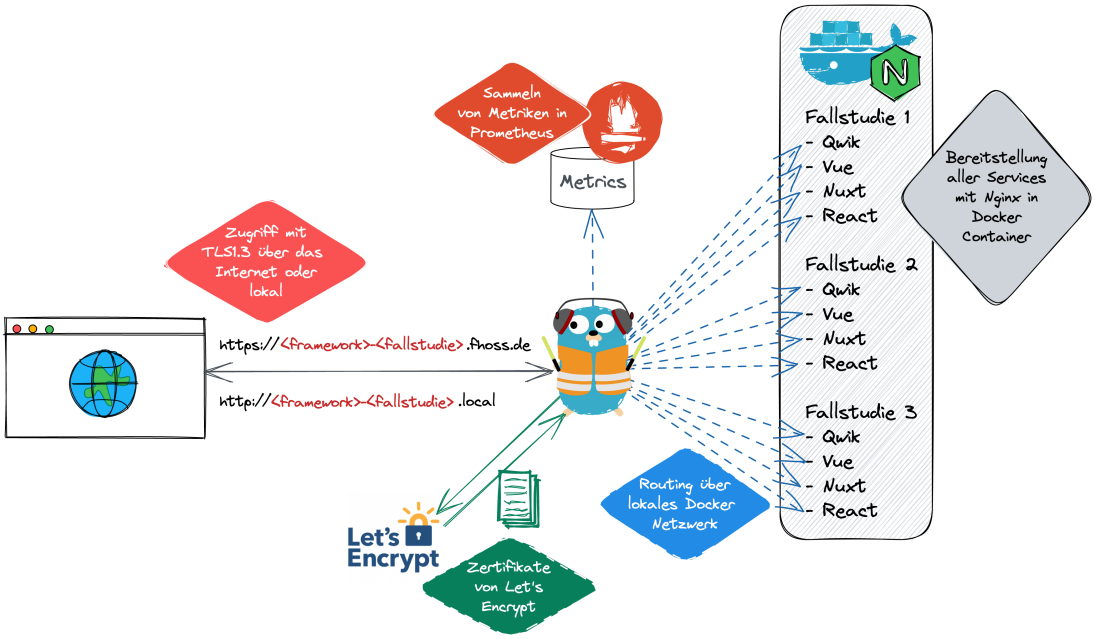


Abbildung 5.1: Traefik Reverse-Proxy Architektur (Eigene Darstellung)

Diese Art der Bereitstellung hat außerdem den Vorteil, dass Metriken darüber gesammelt werden können. Traefik unterstützt verschiedene Metrik-Backends (vgl. Traefik Labs, 2016a, Metrics). Vor allem Informationen wie z. B. die durchschnittliche Antwortzeit eines Frameworks sind dabei interessant. Dazu werden Daten mit Prometheus[46] gesammelt und mit Grafana[47] dargestellt. Für die Darstellung wird das offizielle Traefik Dashboard[48] verwendet.

45 traefik.io/traefik
46 prometheus.io
47 grafana.com
48 grafana.com/grafana/dashboards/17346-traefik-official-standalone-dashboard

5.1.2 Containerisierung

Um die in den nächsten Kapiteln vorgestellte Tests ohne Probleme auf einem Server auszuführen, werden sie in einen jeweiligen Docker Container transferiert. Es werden, wenn möglich, mehrstufige Builds (englisch: *Multi-stage builds*)[49] verwendet. Damit kann die Container-Größe optimiert werden und für die Laufzeit unnötig installierte Pakete reduziert werden. Dabei sind verschiedene Dinge zu beachten.

Wenn ein Browser für Tests benötigt wird, muss der finale Container eine Installation von Google Chrome oder Chromium beinhalten (vgl. andreasonny83, 2022). Chromium kann dabei u. a. einfach über den Package Manager des genutzten Betriebssystems installiert werden.

Es werden in den Tests unterschiedliche Programmiersprachen verwendet. Um JS-Dateien mit Node auszuführen, wird der offizielle Node Container[50] in der Version 19 als Basis genutzt. Wenn Go kompiliert werden muss, wird der offizielle Go Container[51] zum Bauen genutzt. Um die Services zu starten, wird eine jeweilige Docker-Compose[52]-Datei genutzt (Codefragment 5.1).

Als Beispiel wird im Folgenden der Start des Benchmark-Service mit einer Docker-Compose-Datei beschrieben. Sobald der Docker startet, wird ein Ordner in den Container eingebunden (Codefragment 5.1 - Zeile 42). Damit können die Ergebnisse auf dem Host-System persistent abgespeichert werden. Zu Beginn und Ende des Tests sowie bei Problemen kann eine E-Mail über den Gmail Simple Mail Transfer Protocol (SMTP)-Server versendet werden (Kumar, 2021). Dazu können die benötigten Environmentvariablen gesetzt werden. Es werden so u. a. E-Mail (Codefragment 5.1 - Zeile 11), ein Benutzername (Codefragment 5.1 - Zeile 12) und Passwort (Codefragment 5.1 - Zeile 13) in der Datei bereitgestellt.

Die zu testenden Services können als Liste in der URLS Environmentvariable gesetzt werden. Außerdem können Einstellungen wie die Anzahl an Testdurchläufe (Codefragment 5.1 - Zeile 39) und die Pause zwischen einem Durchlauf (Codefragment 5.1 - Zeile 40) eingestellt werden.

Schließlich muss die Hosts-Datei des Containers angepasst werden (Codefragment 5.1 - Zeile 43) um den Zugriff im lokalen Netzwerk (Codefragment 5.1 - Zeile 4) auf den Reverse-Proxy zu leiten (Codefragment 5.2).

49 docs.docker.com/build/building/multi-stage/
50 hub.docker.com/_/node
51 hub.docker.com/_/golang
52 docs.docker.com/compose

```yaml
version: '3.9'
networks:
  proxy:
    external: true
services:
  benchmark:
    image: ghcr.io/flohoss/bachelor-benchmark:latest
    container_name: benchmark
    environment:
      TZ: "Europe/Berlin"
      EMAIL_TO: "mail@example.de"
      EMAIL_USERNAME: "example@gmail.com"
      EMAIL_PASSWORD: "example"
      URLS:
       "http://qwik-interaction.local,
        http://vue-interaction.local,
        http://nuxt-interaction.local,
        http://react-interaction.local,
        https://qwik-interaction.fhoss.de,
        https://vue-interaction.fhoss.de,
        https://nuxt-interaction.fhoss.de,
        https://react-interaction.fhoss.de,
        http://qwik-countries.local,
        http://vue-countries.local,
        http://nuxt-countries.local,
        http://react-countries.local,
        https://qwik-countries.fhoss.de,
        https://vue-countries.fhoss.de,
        https://nuxt-countries.fhoss.de,
        https://react-countries.fhoss.de,
        http://qwik-blog.local,
        http://vue-blog.local,
        http://nuxt-blog.local,
        http://react-blog.local,
        https://qwik-blog.fhoss.de,
        https://vue-blog.fhoss.de,
        https://nuxt-blog.fhoss.de,
        https://react-blog.fhoss.de"
      AMOUNT: 100
      DELAY: 30
    volumes:
      - "./result:/app/result"
      - "./hosts:/etc/hosts:ro"
    networks:
      - proxy
```

Codefragment 5.1: Benchmark Docker-Compose

```
127.0.0.1 localhost
172.18.0.1 qwik-interaction.local qwik-countries.local qwik-blog.local
    vue-interaction.local vue-countries.local vue-blog.local
    nuxt-interaction.local nuxt-countries.local nuxt-blog.local
    react-interaction.local react-countries.local react-blog.local
```

Codefragment 5.2: Benchmark Hosts-Datei

5.2 Lighthouse-Tests

Mit Google Lighthouse können Webanwendungen einfach analysiert werden. Es wurde entwickelt, um einfach und schnell Probleme mit einer Webseite zu erkennen.

Entwickler des Qwik Web-Framework nutzten dieses Tool, um die Vorteile des Frameworks zu bewerben (Kapitel 1.1). In dieser Arbeit kommt es deshalb als Erstes zum Analysieren der Fallstudien zum Einsatz. Das Hauptargument der Entwickler von Qwik ist eine hohe Bewertung, wenn die Anwendung in Qwik geschrieben ist. Auch werden auf der offiziellen Webseite Anwendungen mit deren aktuellen, sehr guten Lighthouse-Werten dargestellt (vgl. Builder.io Inc, 2022, Showcases).

Wenn das Tool die Analyse abgeschlossen hat, wird ein Bericht mit den berechneten Werten für jede unterstützte Metrik angezeigt (Kapitel 5.2.1). Ebenso werden allgemeine und auch spezifische Empfehlungen zur Lösung dieser Probleme vorgeschlagen (vgl. Khayrullin, 2022).

Es werden verschiedene offiziell unterstützte Möglichkeiten bereitgestellt, eine Anwendung mit Lighthouse zu testen. Eine Analyse kann entweder über die Entwicklung-Tools in Chrome, die offizielle Chrome-Erweiterung oder Node durchgeführt werden (vgl. Chrome Developers, 2016). Für die programmatischen Tests in dieser Arbeit wird das Node-Modul genutzt[53].

Die Analyse wird mit einem emulierten Moto G4[54] und der Lighthouse Version 9.6.8 durchgeführt. Als Netzwerkgeschwindigkeit wird eine langsame 4G Verbindung (1.638,4 Kbps Durchsatz) simuliert. Chromium wird in der Version 108.0.5359.124[55] genutzt. Es wird immer nur das initiale Laden einer Anwendung betrachtet.

53 www.npmjs.com/package/lighthouse
54 de.wikipedia.org/wiki/Motorola_Moto_G
55 github.com/chromium/chromium/releases/tag/108.0.5359.124

5.2.1 Zusammensetzung der Lighthouse-Leistungskennzahlen

Lighthouse überprüft eine Webseite auf viele verschiedene Metriken, die in Kategorien aufgeteilt werden. Jede Leistungskennzahl hat eine unterschiedliche Gewichtung in die jeweilige Gesamtkennzahl. Folgende Kategorien werden von Lighthouse angeboten:

- Performance (deutsch: *Leistung*)
- Accessibility (deutsch: *Barrierefreiheit*)
- Best Practices (deutsch: *Bewährte Praktiken*)
- Search Engine Optimization (SEO) (deutsch: *Suchmaschinenoptimierung*)
- Progressive Web App (PWA) (deutsch: *Progressive Webanwendung*)

Die genaue Aufschlüsselung der Kategorien kann der offiziellen Konfiguration[56] entnommen werden. Bei den durchgeführten Tests können dabei die gesammelten Metriken und deren Gewichtung auf die eigenen Anforderungen angepasst werden. Für die Arbeit wichtigsten Leistungsmetriken und deren Gewichtung sind der Tabelle 5.1 zu entnehmen.

Benennung	**Beschreibung**	**Gewichtung**
First Contentful Paint (deutsch: *Einblenden des ersten Inhalts*)	markiert den Zeitpunkt, an dem der erste Text oder das erste Bild in HTML gezeichnet wird	10
Speed Index (deutsch: *Geschwindigkeitsindex*)	zeigt an, wie schnell der Inhalt einer Seite sichtbar aufgefüllt wird	10
Largest Contentful Paint (deutsch: *Einblenden des größten Inhalts*)	markiert den Zeitpunkt, an dem der größte Text oder das größte Bild in HTML gezeichnet wird	25
Time to Interactive (deutsch: *Zeit bis zum Interagieren*)	ist die Zeit, die die Seite benötigt, um vollständig interaktiv zu werden	10
Total Blocking Time (deutsch: *Gesamtsperrzeit*)	Summe aller Zeitspannen zwischen First Contentful Paint und Time to Interactive, wenn die Aufgabenlänge 50 ms überschreitet	30
Cumulative Layout Shift (deutsch: *Kumulative Layout-Verschiebung*)	misst die Bewegung der sichtbaren Elemente innerhalb des Ansichtsfensters	15

Tabelle 5.1: Lighthouse-Kennzahlen der Kategorie Leistung

Mit den ermittelten Werten und deren Gewichtung berechnet Lighthouse eine Kennzahl, die die Gesamtleistung der Webseite darstellen soll. Zusätzlich werden die Größe und Anzahl der zu Ausführung verwendete JS-Dateien angegeben (vgl. Google; Kreft, 2017).

56 github.com/GoogleChrome/lighthouse/blob/main/core/config/default-config.js

5.2.2 Programmatische Lighthouse-Tests der Fallstudien

Um das Node-Modul zu nutzen, wird eine neues Node Projekt mit Typescript[57] initialisiert. Typescript ist dabei optional, erhöht aber die Typensicherheit und kann Fehler beim Programmieren verringern. Eine zusätzliche Kompilierung ist aber erforderlich, damit eine JS-Datei in Node ausgeführt werden kann. Zusätzlich zu Typescript kann ESLint[58] installiert werden. ESLint analysiert den geschriebenen Code statisch, um Probleme zu finden. Damit können z. B. unnötige Importe gefunden werden. Eine Analyse und die Kompilierung kann durch ein Skript (Codefragment 5.3 - Zeile 6) in der *package.json* einfach durchgeführt werden (vgl. Irabor, 2019).

Um den Test mit Lighthouse durchzuführen, muss zusätzlich *lighthouse* und *chrome-launcher* installiert werden.

```
1 {
2   "name": "benchmark",
3   "version": "1.0.0",
4   "main": "dist/benchmark.js",
5   "scripts": {
6     "start": "eslint . --ext .ts && tsc && node dist/benchmark.js"
7   },
8   "devDependencies": {
9     "@typescript-eslint/eslint-plugin": "^5.43.0",
10    "@typescript-eslint/parser": "^5.43.0",
11    "chrome-launcher": "^0.15.1",
12    "eslint": "^8.27.0",
13    "lighthouse": "^9.6.8",
14    "typescript": "^4.8.4"
15  }
16 }
```

Codefragment 5.3: Programmatischer Benchmark package.json

57 www.typescriptlang.org
58 eslint.org

Um das Modul für verschiedene Szenarios nutzen zu können, werden die schon in Kapitel 5.1.2 angesprochenen Environmentvariablen genutzt. Diese können in Node direkt über *process.env* ausgelesen werden (Codefragment 5.4). Damit kann der Ablauf genau gesteuert werden und mehrere URLs gleichzeitig getestet werden. Die Environmentvariable *AMOUNT* steuert, wie oft ein Durchlauf wiederholt wird, *DELAY* stellt den Abstand zwischen den einzelnen Durchläufen in Sekunden ein.

```
1 const urls = (process.env.URLS && process.env.URLS.split(",")) || [""];
2 const amount = parseInt(process.env.AMOUNT) || 1;
3 const delay = parseInt(process.env.DELAY) || 0;
```

Codefragment 5.4: Benchmark Environmentvariablen

Der Test startet eine Instanz eines Browsers ohne Graphical user interface (GUI) (englisch: *headless*) und sammelt darin alle Leistungskennzahlen der angegebenen URLs. Diese Ergebnisse werden dann als JSON-Datei auf dem Host-System für eine spätere Analyse archiviert. Jedes Framework bekommt dabei seinen eigenen Ordner. Dieser wiederum wird in einem Ordner, der die aktuelle Uhrzeit und das Datum (z. B. 22.12.13-09-22-10) beinhaltet, gespeichert. Darin finden sich dann die resultierenden JSON-Dateien, die zur Analyse ausgewertet werden können.

Anschließend ein Beispiel, wie die Ordnerstruktur aussehen kann:

```
result/
└── 22.12.13-09-22-10/     # Start-Datum der Tests
  └── countries/           # Fallstudie
    └── qwik/              # Framework
      ├── 001.json         # Lighthouse Ergebnisse
      ├── ...
      └── 100.json
```

Zu Beginn des Tests wird eine E-Mail mit allen eingestellten Laufzeitparametern geschickt. Wenn der Test abgeschlossen ist, wird erneut eine E-Mail mit der Laufzeit des Tests verschickt.

5.2.3 Durchführung der Lighthouse-Tests

Zum Ausführen der Tests wird der in Kapitel 5.1.2 vorbereitete Docker ausgeführt. Über die Environmentvariablen wird die Anzahl der Durchgänge auf 100 Stück gesetzt. Zwischen den Durchgängen wird eine Pause von 30 Sekunden eingestellt. Somit wird eine Testdauer von ungefähr 9 Stunden erreicht. Für die Leistung der Frameworks nicht benötigte Lighthouse Kategorien (Kapitel 5.1) wie *PWA* und *Accessibility* werden nicht getestet. Es werden sowohl Tests über ein lokales Netzwerk als auch über eine öffentliche Domain im Internet durchgeführt. Da der lokale Zugriff ohne Zertifikate erfolgt und das Qwik Web-Framework sich in der Entwicklungsphase befindet, werden Metriken wie *is-on-https* (deutsch: *Ist über HTTPS erreichbar*) und *errors-in-console* (deutsch: *Fehler in der Konsole*) nicht in die *Best Practices* Kategorie mit einbezogen.

5.2.4 Analyse der Lighthouse-Tests

Die Analyse der Dateien wird programmatisch mit der Programmiersprache Go[59] durchgeführt. Lighthouse generiert pro Test eine 5395 Zeilen große JSON-Datei. Um die Anzahl an Informationen ohne Laufzeitfehler und mit Typensicherheit zu analysieren, wurde diese Sprache gewählt. Persönliche Präferenz und u. a. einfach zu nutzende Parallelität sind ebenfalls Gründe, warum die Analyse in Go programmiert wurde.

Um die Tests visualisieren zu können, wird *go-echarts*[60] genutzt. Go-echarts zielt darauf ab, eine einfache und dennoch leistungsfähige Bibliothek, basierend auf Apache ECharts[61], bereitzustellen (chenjiandongx, 2022).

Mit dem von Google bereitgestellten Lighthouse Report Viewer[62] können außerdem einzelne Ergebnisse angezeigt und analysiert werden.

59 go.dev/
60 github.com/go-echarts/go-echarts
61 echarts.apache.org/
62 googlechrome.github.io/lighthouse/viewer

Zu Beginn der Analyse muss die Struktur der JSON-Dateien in ein *Golang Struct* umgewandelt werden. Dafür kann das Transformations-Tool *json-to-go*[63] genutzt werden. Das resultierende Struct sollte allerdings manuell auf falsche Typen überprüft werden. Wenn z. B. ein Test als Ergebnis eine *0* liefert, kann ein unterschiedlicher Test anstatt eines *Integer*, ein *Float* als Datentyp liefern (Abbildung 5.2).

Nachdem fehlerhafte Typen ersetzt wurden, können die Daten in verschiedene Charts übertragen werden. Geschwindigkeiten und Ergebnisse der einzelnen Kategorien werden als Linien-Diagramme dargestellt. Größe und Anzahl von Dateien werden dann mit einem Pie-Chart angezeigt.

Abbildung 5.2: Eine Anpassung des generierten Structs (Generiert:rot - Angepasst:grün) (Eigene Darstellung)

63 github.com/mholt/json-to-go

5.3 Automatisierte Interaktionstests

Das Qwik Web-Framework lädt auf der Grundlage von Benutzerinteraktionen JS-Dateien nach. Da Lighthouse nur das initiale Laden der Anwendung unterstützt, wird zusätzlich ein eigener Test entwickelt. Darin soll jede Interaktion in jeder Fallstudie durchgeführt werden und die heruntergeladenen Dateien analysiert werden. Um dies zu realisieren wird ebenfalls ein Chromium Browsers[64] ohne GUI (englisch: *headless browser*) genutzt.

Im Gegensatz zu Lighthouse muss selbstständig eine Verbindung zum Browser aufgebaut werden und Metriken gesammelt werden. Dazu wird das Go-Paket Godet[65] genutzt. Godet ist ein Fernbedienungsprogramm für die Chromium Entwicklung-Tools. Damit kann einfach auf verschiedene Callback-Events (z. B. Netzwerk-Events oder DOM-Events) reagiert werden (vgl. Kotecha, 2019).

Zum Ausführen der Interaktionen wird dann u. a. JS genutzt um Nutzereingaben zu simulieren. Zusätzlich kann über Bildschirmfotos überprüft werden, ob alle vollständig ausgeführt wurden (Abbildung 5.3). Die gesammelten Daten werden dann mit Diagrammen ausgewertet und verglichen.

Es werden die einzelnen Metriken eines Durchlaufs in einer JSON-Datei protokolliert. Außerdem werden wie in Kapitel 5.2.4 das schon verwendete Go-Paket *go-echarts* zur Visualisierung genutzt.

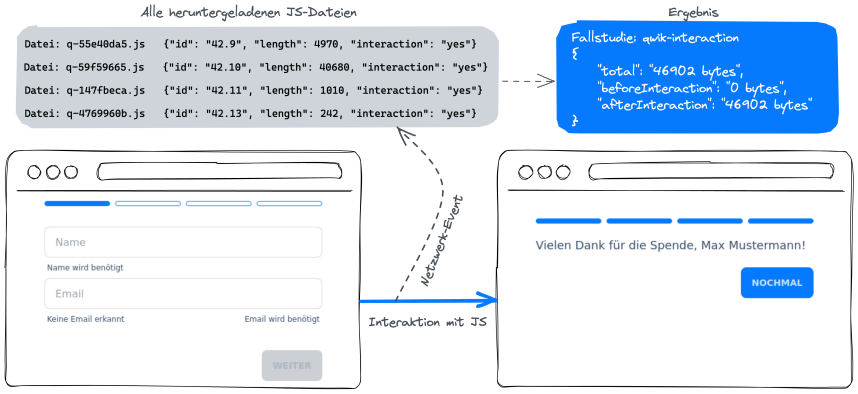


Abbildung 5.3: Automatisierter Interaktions-Test der Fallstudie 1: Formular (Eigene Darstellung)

64 www.chromium.org
65 github.com/raff/godet

5.4 Ergebnisse

Die Tests wurden an unterschiedlichen Tagen und zu unterschiedlichen Zeiten durchgeführt, um die Ergebnisse zu verifizieren. Jedes Framework wurde mit der offiziellen Anleitung initialisiert (Kapitel 4). Es wurden keine Änderungen an den Standardeinstellungen vorgenommen. Nach Optimierungen können die Daten von den hier gezeigten Ergebnissen abweichen.

5.4.1 Ergebnisse der Reverse-Proxy-Metriken

Die Daten, die der Reverse-Proxy über die Testdauer des automatisierten Lighthouse-Tests (Kapitel 5.2) gesammelt hat, zeigen u. a. die durchschnittliche Antwortzeit eines Services (Tabelle 5.2). Für die resultierenden Metriken genutzte Architektur kann der Abbildung 5.1 entnommen werden. Qwik hat bei jeder Fallstudie die schnellste Antwortzeit. React braucht im Gegensatz dazu zum Antworten immer am längsten.

Fallstudie	**Framework**	**durchschnittliche Antwortzeit (GET-Request)**
Formular (Kapitel 3.2)	Qwik	1,03 ms
	Vue	1,07 ms
	Nuxt	1,16 ms
	React	1,47 ms
Länder (Kapitel 3.3)	Qwik	893 µs
	Vue	1,12 ms
	Nuxt	1,19 ms
	React	1,56 ms
Blog (Kapitel 3.4)	Qwik	1,02 ms
	Vue	1,53 ms
	Nuxt	1,29 ms
	React	2,16 ms

Tabelle 5.2: Durchschnittliche Antwortzeit im Reverse-Proxy

Es können außerdem die am meisten angeforderten Services angezeigt werden. Da das Backend von mehr als nur einer Fallstudie und auch von allen Frameworks verwendet wird, ist es wenig verwunderlich an erster Stelle (Abbildung 5.4).

Direkt gefolgt wird es von den beiden Fallstudien Blog und Länder, die mit dem Qwik Web-Framework programmiert wurden. Eine Erklärung dafür kann sein, dass Qwik eine größere Anzahl von Dateien laden muss als andere Frameworks (Kapitel 5.3). Es wird jede angeforderte Datei über die QRL (Kapitel 2.3.4) als separate Anfrage gewertet. Vue braucht bei allen Fallstudien am wenigsten Anfragen, um sie darzustellen.

Die Reihenfolge, in der die Tests durchgeführt werden, kann der Docker-Compose (Codefragment 5.1) entnommen werden.

Abbildung 5.4: Am meisten angeforderte Services (Eigene Darstellung)

Die Abbildung 5.5 zeigt über eine Dauer von einer Stunde wie viele Daten pro Framework und Sekunde übertragen wurden. Die Werte zeigen, dass Qwik über die Zeit am wenigsten Daten überträgt. Dabei ist allerdings zu beachten, dass über den angezeigten Testzeitraum nur Lighthouse-Tests durchgeführt wurden. Deswegen können nur die initial geladenen Daten eines Frameworks in Betracht gezogen werden. Da Qwik erst mit Interaktionen JS nachlädt, werden diese hier nicht berücksichtigt. Diese Werte korrelieren mit den Ergebnissen der Interaktionstests in Kapitel 5.4.3.

So kann Qwik in der Fallstudie Formular (Kapitel 3.2) im Durchschnitt mit lediglich 454 Bytes pro Sekunde auskommen. React braucht für die gleiche Fallstudie mit 8,91 Kibibyte (KiB)/s fast 20-mal so viele Daten (Abbildung 5.5).

Außerdem ist React auch das Framework, welches für die Fallstudie Länder (Kapitel 3.3) im Durchschnitt 21,9 KiB/s lädt. Das sind fast genauso viele Daten, wie das Backend benötigt, um alle Frameworks und Fallstudien zu versorgen (25,8 KiB/s).

Die in der Einleitung angesprochenen teuren Datenvolumen bei Mobilfunkverträgen können so schnell an ihre Grenzen gelangen (Kapitel 1.1). Hier haben Qwik und Vue mit den insgesamt kleineren Antworten einen klaren Vorteil.

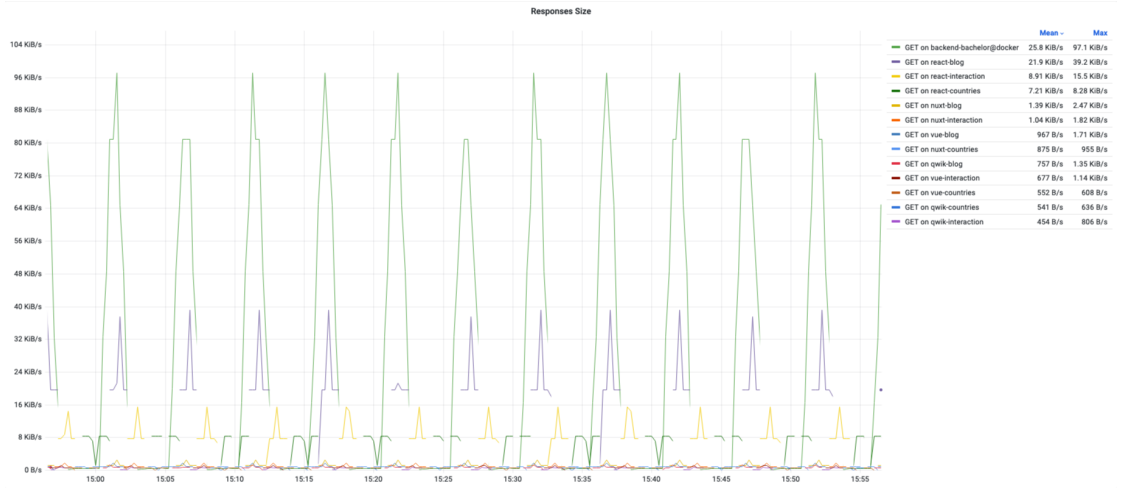


Abbildung 5.5: Antwortgröße eines GET-Request (Eigene Darstellung)

5.4.2 Ergebnisse der Lighthouse-Tests

Ergebnisse der Fallstudie 1: Formular

Jedes Framework kann in der Fallstudie Formular (Kapitel 3.2) bei den im Kapitel 5.2.1 vorgestellten Kategorien volle Punktzahl erreichen. So gibt es durchweg 100 % bei Leistung, bewährte Praktiken und Suchmaschinenoptimierung (Abbildung 5.6).

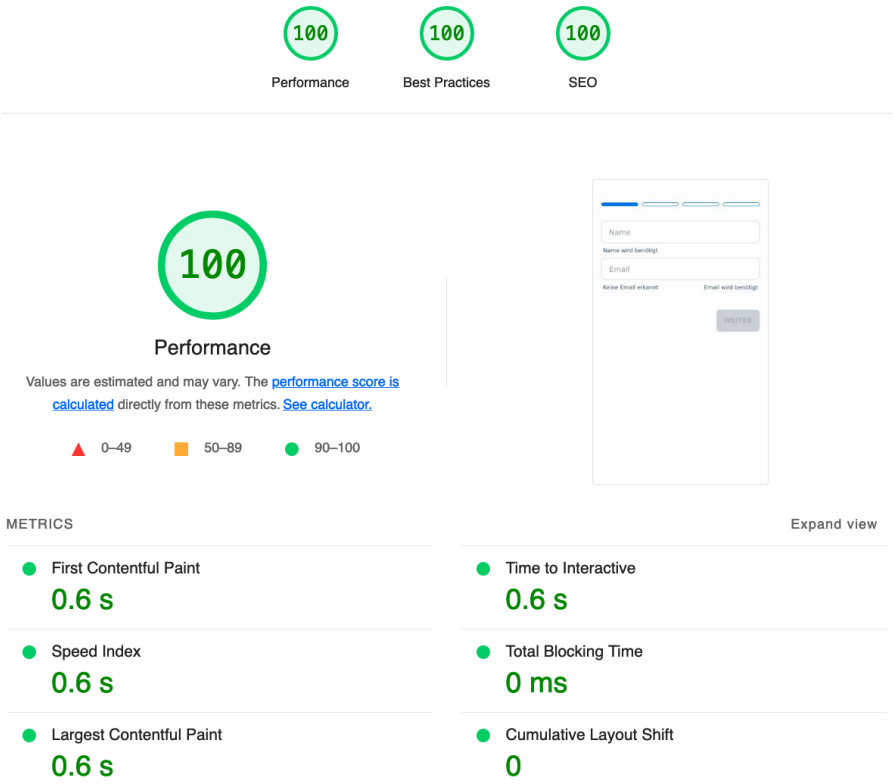


Abbildung 5.6: Eines der lokalen Qwik Ergebnisse in Lighthouse der Fallstudie 1: Formular (Eigene Darstellung)

Eine genaue Analyse der gesammelten Werte deckt aber doch deutliche Unterschiede auf. Bei der in der Tabelle 5.1 mit 25 % sehr hoch eingestuften Metrik Einblenden des größten Inhalts sticht das Qwik Web-Framework (637,5 ms) im lokalen Vergleich deutlich als schnellstes heraus (Abbildung 5.7). Trotz dieser unterschiedlichen Zeitmessungen erzielt jedes Framework 100 % als Leistungsgesamtergebnis. Somit sind die Werte von anderen Frameworks, obwohl langsamer als Qwik für Google Lighthouse sehr gut.

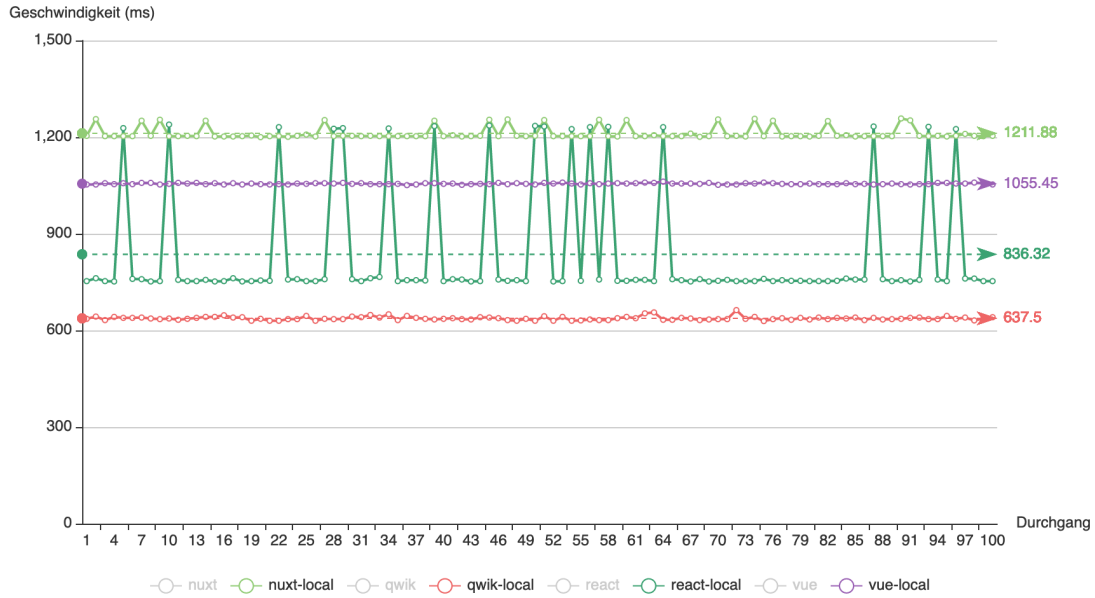
Abbildung 5.7: Zeit zur Einblendung des größten Inhalts bei der Fallstudie 1: Formular (Eigene Darstellung)

Mehr Metriken zeigen aber, wie gut Qwik bei dieser Fallstudie abschneidet.

Wenn die Werte zum Einblenden des ersten Inhalts analysiert werden, stellt sich heraus, dass Qwik die gleiche Zeit benötigt wie schon beim Einblenden des größten Inhalts (Abbildung 5.8a). Dies ist der Wiederaufnehmbarkeit zu verdanken. Qwik liefert HTML sofort und ohne nötige Ausführung von JS aus. React zeigt hier eine Inkonsistenz. Zwar wird meistens der erste Inhalt schneller eingeblendet als der größte gezeichnete Inhalt der Seite, häufig aber auch nicht.

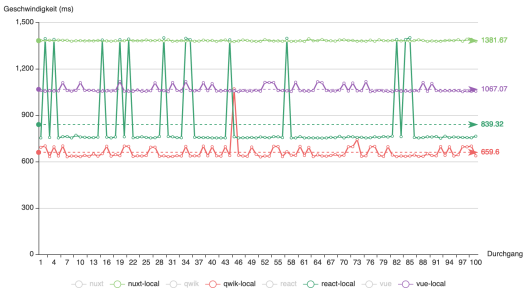
Hydrierung (Kapitel 2.2) spielt bei den Ergebnissen eine große Rolle. Wenn andere Frameworks noch mit der Wiederherstellung beschäftigt sind, ist Qwik schon mit der Darstellung fertig. Gerade bei einer Anwendung wie der Fallstudie Formular (Kapitel 3.2) wird dies deutlich.

Qwiks Erfolg zieht sich bei einem der Werte durch alle Fallstudien. Bei der Zeit zur Einblendung des ersten Inhalts ist es im Durchschnitt immer das schnellste Framework (Abbildung 5.8).



(a) Fallstudie 1: Formular

(b) Fallstudie 2: Länder



(c) Fallstudie 3: Blog

Abbildung 5.8: Zeit zur Einblendung des ersten Inhalts (Eigene Darstellung)

Anders als bei der ersten Fallstudie fallen die Gesamtergebnisse der Kategorie Leistung bei z. B. der Fallstudie Länder (Kapitel 3.3) aus. Dieser Wert schwankt bei Qwik um einen Mittelwert von 69,08 %. Bei den anderen Frameworks gibt es im Durchschnitt von Lighthouse eine bessere Bewertung. React (85,11 %) und Nuxt (85,14 %) können hier am besten abschneiden (Abbildung 5.9 und Abbildung 5.10a).

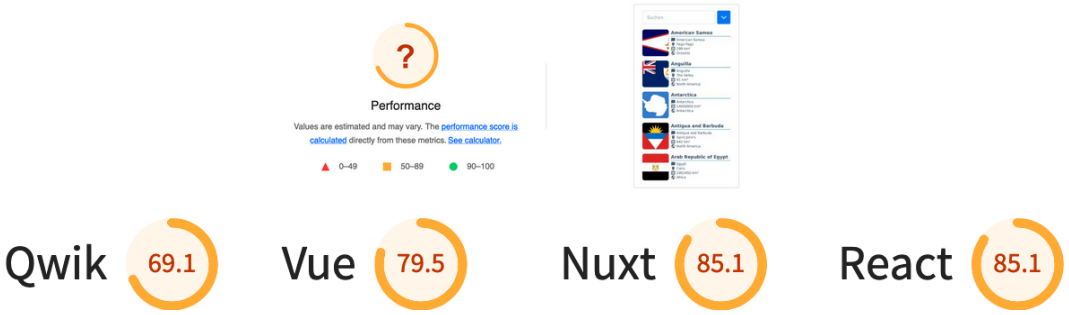
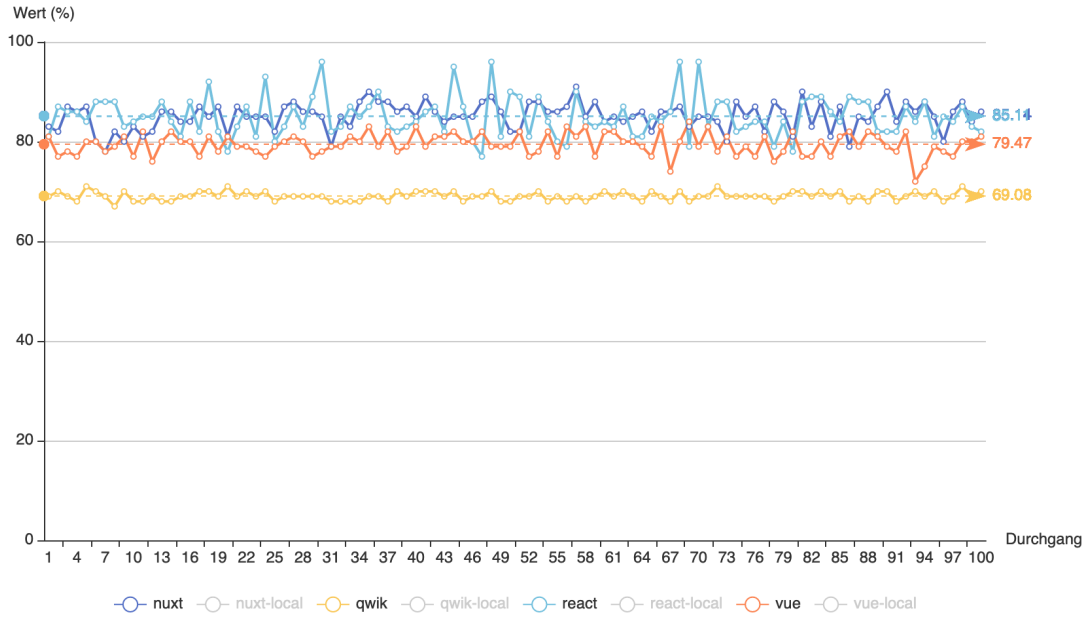


Abbildung 5.9: Durchschnitt aller Ergebnisse der Kategorie Leistung bei der Fallstudie 2: Länder bei Zugriff über TLS (Eigene Darstellung)

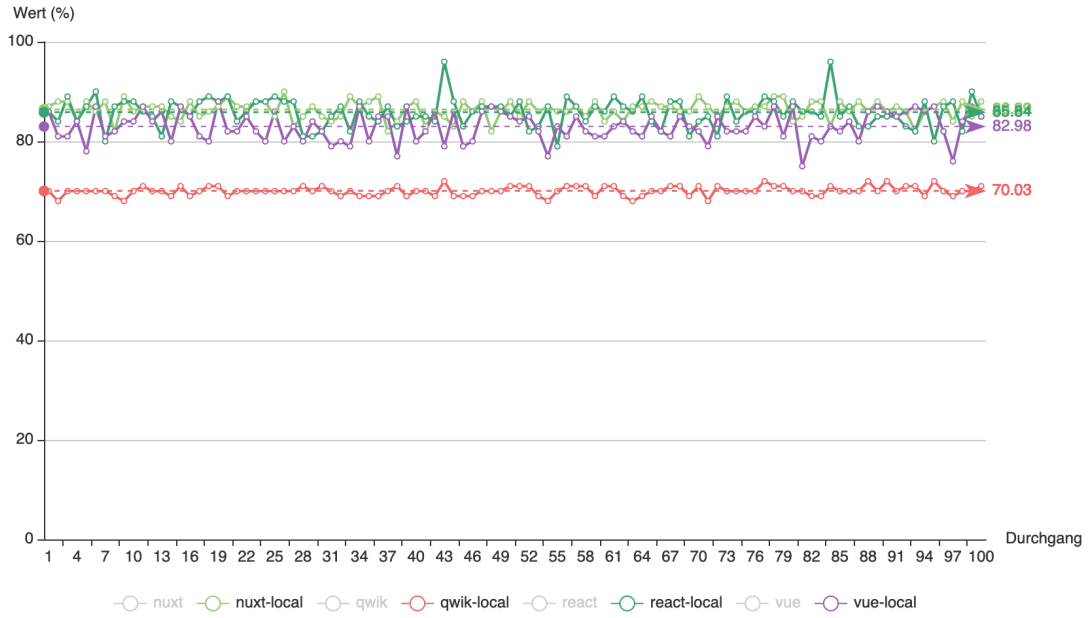
Wenn ein lokaler Zugriff erfolgt, sind vor allem die Werte von Vue (82,98 %) und Qwik (70,03 %) höher (Abbildung 5.10b). Da Qwik und Vue kleinere Dateien als React (85,84 %) und Nuxt (86,39 %) übertragen, kann eine lokale Verbindung hier von Vorteil sein.

Die Ergebnisse liegen vor allem daran, dass Qwik in den wichtigen Metriken *Einblenden des größten Inhalts* (Abbildung 5.11b) und *Gesamtsperrzeit* (Abbildung 5.11a) sehr schlecht abschließt. Viele Elemente, die dargestellt werden müssen, sind für das Qwik Web-Framework, aber auch für die anderen Frameworks eine schwierige Aufgabe. Durch Aufteilung der dargestellten Länder auf mehrere Seiten kann dieses Problem aber vom Programmierer einfach umgangen werden.

Auch könnte z. B. eine Vorbereitungen der Daten auf dem Server interessant sein. Dies wäre u. a. mit Qwik und Nuxt möglich. Da die Fallstudie aber mit jedem Framework ähnlich programmiert wurde, werden diese Vorteile nicht ausgespielt.



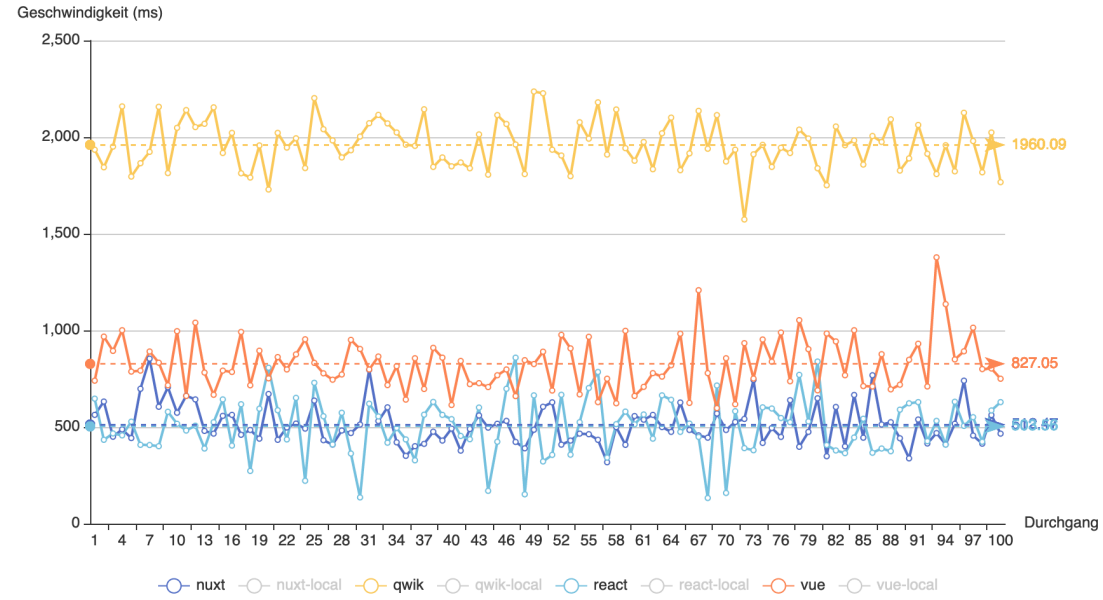
(a) Zugriff über TLS



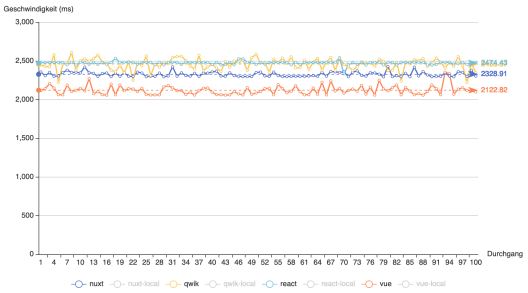
(b) Lokaler Zugriff

Abbildung 5.10: Detaillierte Ergebnisse der Kategorie Leistung bei der Fallstudie 2: Länder (Eigene Darstellung)

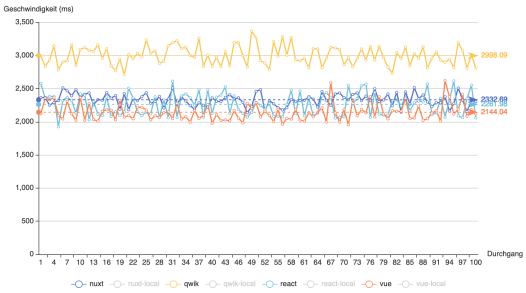
Auch wegen der fehlenden Optimierung der Anwendungen auf die Stärken des Frameworks liegt deshalb die Zeit, bis der Benutzer mit der Qwik-Anwendung interagieren kann, bei über 2,7 Sekunden. Und damit fast eine halbe Sekunde über der von Nuxt (Abbildung 5.11c).



(a) Gesamtsperrzeit



(b) Zeit zur Einblendung des größten Inhalts



(c) Zeit bis zum Interagieren

Abbildung 5.11: Wichtige Metriken für die Berechnung der Leistung bei der Fallstudie 2: Länder (Eigene Darstellung)

Die Analyse der Fallstudie Blog (Kapitel 3.4) zeigt, dass hier alle getesteten Frameworks in der Kategorie Leistung fast gleich abschließen (Abbildung 5.12). Eine Optimierung könnte wie bei der zweiten Fallstudie eine schon auf dem Server vorbereitete HTML-Seite sein. Da der JS-Teil des Blogs mit allen Frameworks identisch programmiert wurde, fallen die Ergebnisse des Qwik Web-Framework entgegen der Erwartung in Kapitel 3.4 schlecht aus. So kann es nicht den Vorteil gegenüber der von den anderen Frameworks auszuführenden Hydrierung ausspielen (Kapitel 2.3.6).

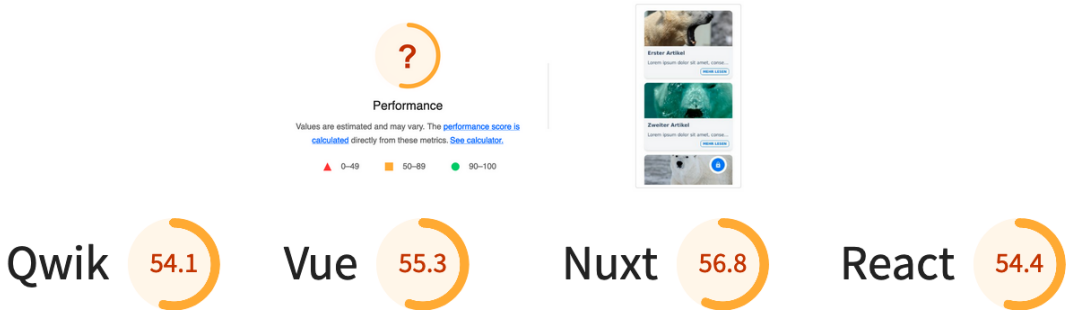


Abbildung 5.12: Durchschnitt aller Ergebnisse der Kategorie Leistung bei der Fallstudie 3: Blog bei Zugriff über TLS (Eigene Darstellung)

Bei lokalem Zugriff kann Nuxt ganz knapp die besten Werte erreichen, schwankt dabei aber am stärksten (Abbildung 5.13). Da sich das Qwik Web-Framework noch in den frühen Phasen der Entwicklung befindet, können bei der Leistung aber noch Verbesserungen erwartet werden.

Selbst Werte von im Durchschnitt 60,64 % des Frameworks Nuxt sind laut Lighthouse verbesserungsbedürftig (Abbildung 5.14).

Die Verantwortung für ein gutes Ergebnis liegt hier immer noch bei dem Programmierer. Vorhandene Optionen und Vorteile eines Frameworks sollten optimal genutzt werden. Es verdeutlicht, dass ein Framework alleine nicht reicht, um gute Ergebnisse zu erzielen.

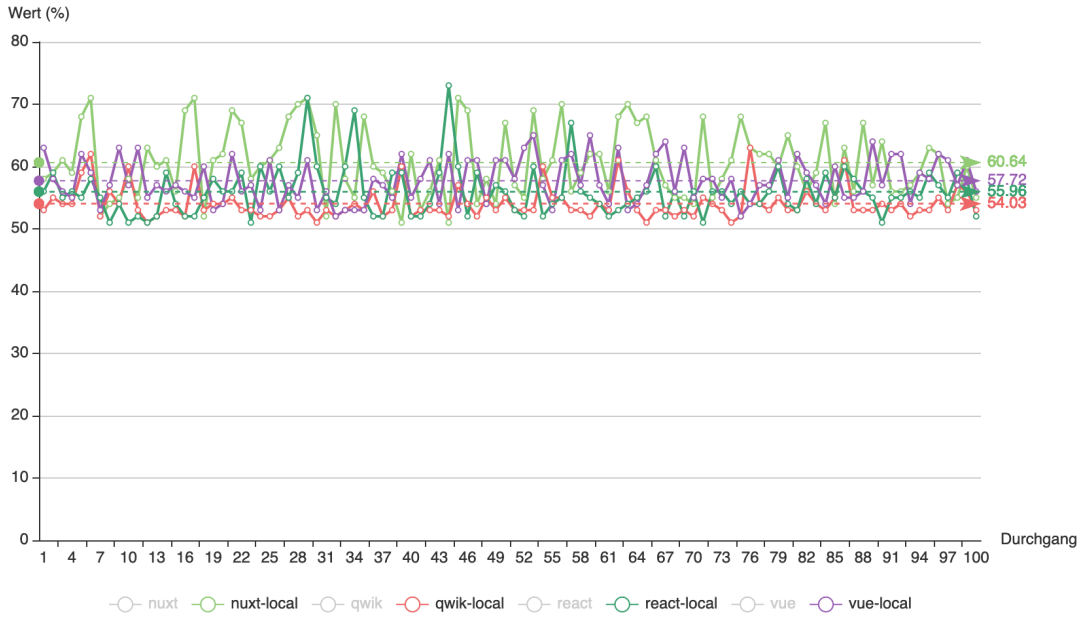


Abbildung 5.13: Kategorie Leistung bei der Fallstudie 3: Blog (Eigene Darstellung)

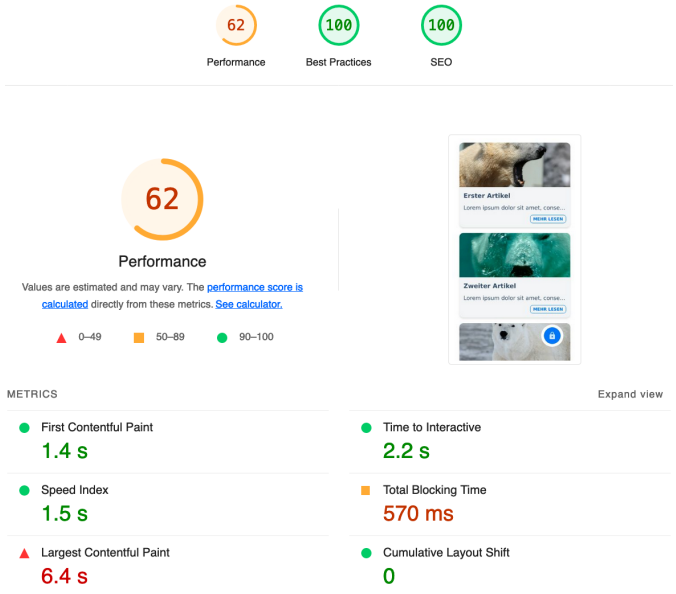


Abbildung 5.14: Eines der lokalen Nuxt Ergebnisse in Lighthouse der Fallstudie 3: Blog (Eigene Darstellung)

Bei Zugriff über TLS und einer Zeit zur Einblendung des größten Inhalts von wenig über 6 Sekunden schneidet Qwik hier knapp am besten ab. Vue kann allerdings bei lokalem Zugriff mit 6,09 Sekunden das Qwik Web-Framework (6,17 Sekunden) vom ersten Platz verdrängen (Abbildung 5.15).

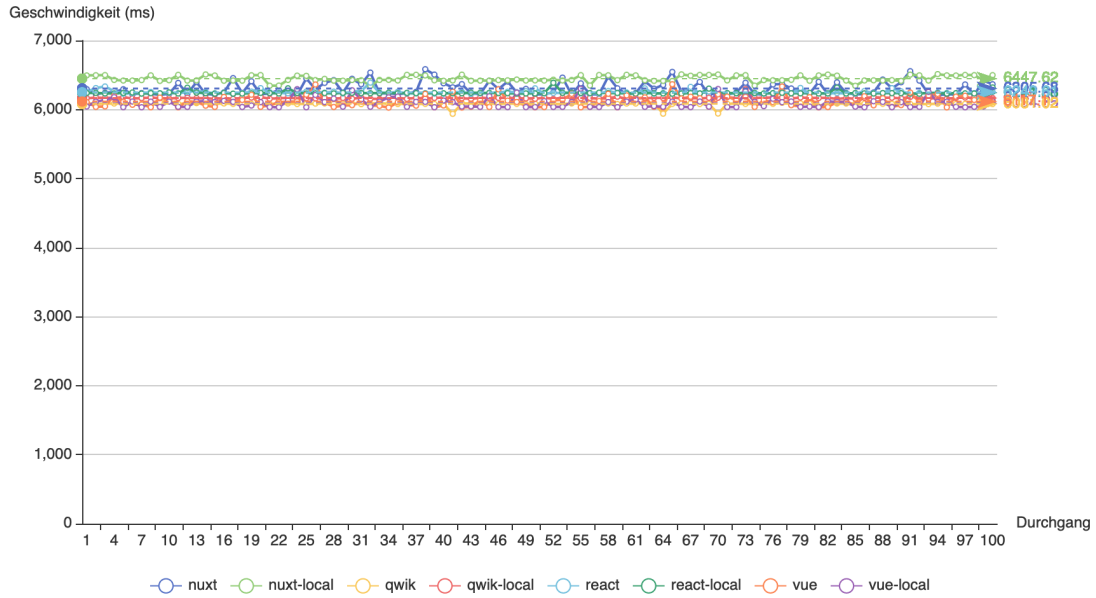
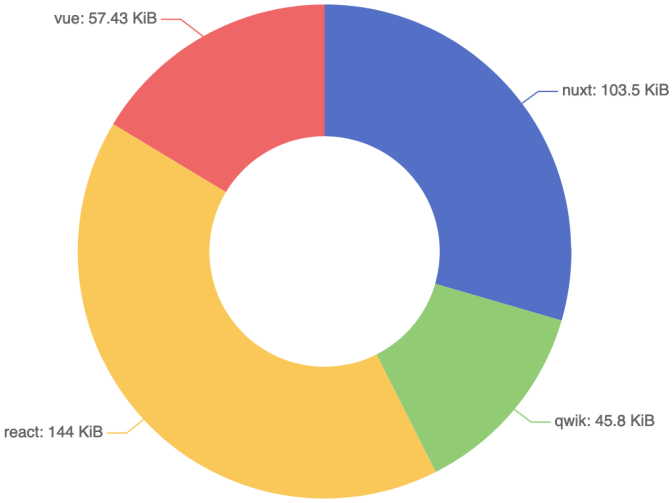


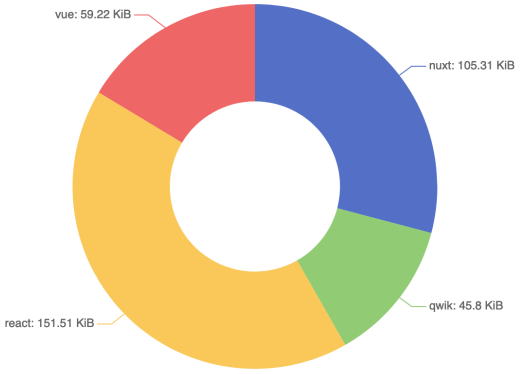
Abbildung 5.15: Zeit zur Einblendung des größten Inhalts bei der Fallstudie 3: Blog (Eigene Darstellung)

5.4.3 Ergebnisse der Interaktionstests

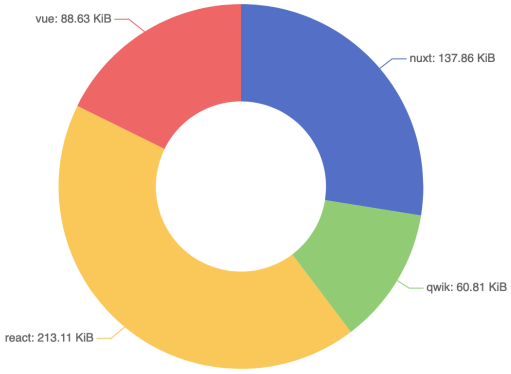
Das Qwik Web-Framework kann hier bei allen Fallstudien glänzen. Im Gegensatz zu anderen Frameworks werden anfänglich weniger große JS-Dateien an den Browser übertragen. Dies ist vor allem den in Kapitel 2.3 beschriebenen Konzepten zu verdanken. Vue braucht fast genauso wenige Daten, während React mit Abstand immer am meisten herunterlädt (Abbildung 5.16). Vor allem in der Fallstudie Blog (Kapitel 3.4), in der ein Router genutzt wird, werden viel mehr Daten benötigt (Abbildung 5.16c).



(a) Fallstudie 1: Formular



(b) Fallstudie 2: Länder



(c) Fallstudie 3: Blog

Abbildung 5.16: Heruntergeladene JS-Dateien und deren Größe nach allen Interaktionen (Eigene Darstellung)

Besonders zu beachten ist dabei, dass beim initialen Laden der Fallstudie Formular (Kapitel 3.2) bei Qwik keine externen JS-Dateien geladen werden (Tabelle 5.3). Nur nach der Interaktion der Anwendung werden über die QRL (Kapitel 2.3.4) externe Skripte nachgeladen. Dies funktioniert aber nur, wenn die Anwendung dies unterstützt. So müssen z. B. bei der Fallstudie Länder (Kapitel 3.3) beim initialen Laden alle Dateien verfügbar sein. Das liegt vor allem daran, dass die Filter- und Suchfunktionen nach dem Laden der Daten direkt ausgeführt und nicht progressiv nachgeladen werden können.

Obwohl das Qwik Web-Framework insgesamt immer die meisten JS-Dateien lädt, bleibt die Gesamtgröße aller Dateien stets geringer als bei den anderen Frameworks.

		vor Interaktionen		**nach Interaktionen**	
Fallstudie	**Framework**	Dateien	Größe (KiB)	Dateien	Größe (KiB)
Formular (Kapitel 3.2)	Qwik	-	-	4	45,8
	Vue	1	57,4	-	-
	Nuxt	1	103,5	-	-
	React	1	144,0	-	-
Länder (Kapitel 3.3)	Qwik	4	45,8	-	-
	Vue	1	59,2	-	-
	Nuxt	1	105,3	-	-
	React	1	151,5	-	-
Blog (Kapitel 3.4)	Qwik	8	51,2	5	9,6
	Vue	1	88,6	-	-
	Nuxt	3	137,9	-	-
	React	1	213,1	-	-

Tabelle 5.3: Heruntergeladene JS-Dateien und deren Gesamtgröße vor und nach allen ausgeführten Interaktionen

5.4.4 Aus dem Bau resultierende Dateien

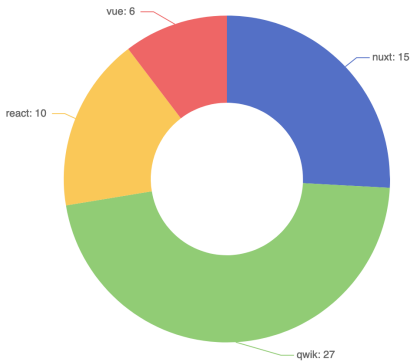
Wenn die aus dem Bau resultierenden Ordner der Frameworks analysiert werden, fallen bestimmte Dinge auf. Qwik erstellt bem Bau der Fallstudie Formular (Kapitel 3.2) viel mehr Dateien (27) als Vue (6), React (10) und Nuxt (15) (Abbildung 5.17a). Es ist gut zu erkennen, dass hier bei dem Qwik Web-Framework eine Aufteilung von JS in mehrere Dateien erfolgt. Trotzdem bleibt die Gesamtgröße des Ordners zu den anderen Frameworks relativ klein.

React erstellt mit den Standardeinstellungen immer den größten Ordner, aber weniger Dateien als Nuxt (Abbildung 5.17b). Deshalb braucht eine Bereitstellung von React mit Abstand den größten Speicherplatz. Dies kann je nach Anwendung ein entscheidender Faktor werden. Vor allem das Einbinden eines Routers hat die Größe von React auf 1280.68 KiB ansteigen lassen (Abbildung 5.17f).

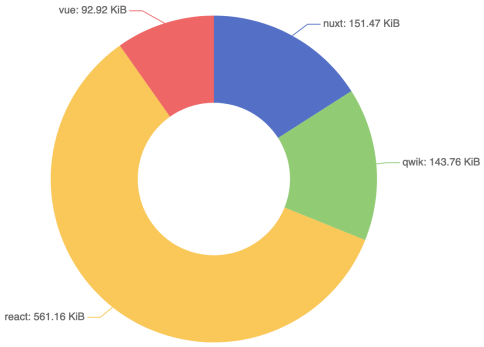
Vue kann hier immer mit der kleinsten Gesamtgröße auskommen und erstellt dabei auch immer die wenigsten Dateien (Abbildung 5.17).

Eine Optimierung im Bau-Prozess kann aber einen großen Unterschied ausmachen. So können z. B. Dateien dynamisch komprimiert werden und nicht auf als extra Dateien auf dem Server abgelegt werden (Kapitel 2.4.2).

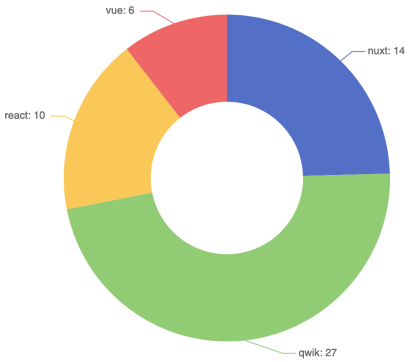
Ebenso können verschiedene, nicht genutzte Komponenten aus dem resultierend JS durch unterschiedliche Methoden entfernt und somit eine reduzierte Gesamtgröße erreicht werden (vgl. Haczek, 2020).



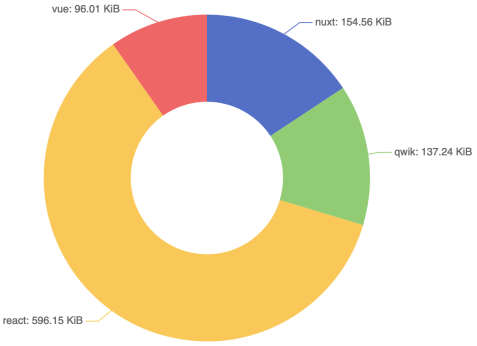
(a) Fallstudie 1: Formular - Anzahl



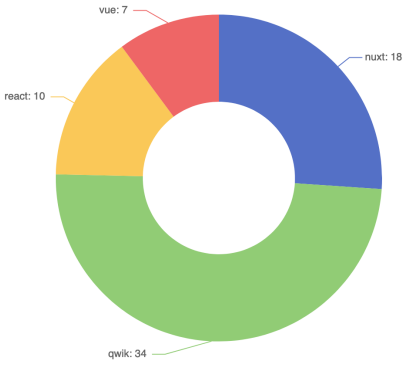
(b) Fallstudie 1: Formular - Größe



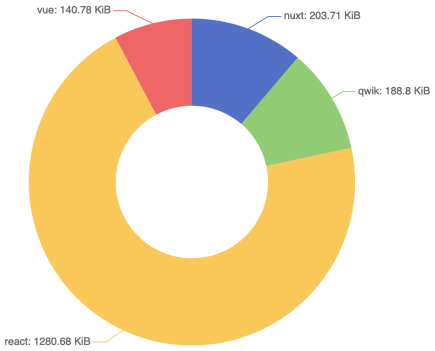
(c) Fallstudie 2: Länder - Anzahl



(d) Fallstudie 2: Länder - Größe



(e) Fallstudie 3: Blog - Anzahl



(f) Fallstudie 3: Blog - Größe

Abbildung 5.17: Aus dem Bau resultierenden Ordnergröße und Anzahl (Eigene Darstellung)

6 Fazit

Die Ergebnisse im Kapitel 5.4 zeigen, dass das Qwik Web-Framework funktioniert. Obwohl es sich in den frühen Phasen der Entwicklung befindet, kann es mit anderen Frameworks mithalten und sie sogar Übertreffen. Vereinzelte Optimierungen und Ausbesserungen können dabei in späteren Versionen immer noch hinzukommen.

Überraschend ist, wie gut das Konzept des progressiven Nachladens funktioniert. Mit sehr wenig Programmieraufwand kann damit viel Einsparung der initial übermittelten Daten erreicht werden (Kapitel 5.4.3).

Optimierungen, die zu kleineren Dateien und besseren Test-Ergebnissen führen, gibt es viele (Elrom, 2020). Interessant ist aber, dass das Qwik Web-Framework ohne große Eingriffe des Entwicklers viele dieser Optimierungen obsolet werden lässt. So wird das in Kapitel 1.1 genannte *Tree Shaking* nicht mehr benötigt, da nur genutzter Code geladen wird. Ebenso wird *Code Splitting* von Qwik für den Entwickler übernommen.

Ohne auf die Art der Programmierung zu achten, werden aber auch mit Qwik immer noch schlechte Leistungswerte erzielt. Probleme eines zu großen DOM, ineffizientes Anfragen von Daten aus dem Backend oder nicht optimierte statische Dateien können wie bei allen anderen Frameworks weiterhin zu langsamen Ladezeiten führen. Deshalb sollten Optimierungen (Kapitel 2.4) weiterhin bei der Nutzung aller Frameworks durchgeführt werden.

Speziell ist aber die Wiederaufnehmbarkeit einer mit Qwik geschrieben Anwendung (Kapitel 2.3.6). Die damit einzigartige Umsetzung der QRL (Kapitel 2.3.4) gibt dem Framework einen Geschwindigkeitsvorteil. Im Client muss kein Initialisierungscode, kein erneutes Rendern und kein Einrichten von DOM-Listeners ausgeführt werden. Der Qwikloader (Kapitel 2.3.3) kümmert sich darum, dass die serialisierbare Anwendung trotzdem interaktiv bleibt. Die Vorteile dieses Konzeptes zeigen sich vor allem in den Ergebnissen der Fallstudie Formular (Kapitel 3.2).

Die im Kapitel 1.2 gestellten Fragen wurden in dieser Arbeit mit drei unterschiedlich aufgebauten Fallstudien beantwortet. Eine genaue Analyse der Ergebnisse zeigt, dass das Qwik Web-Framework immer weniger JS als andere Frameworks laden muss (Kapitel 5.4.3). Außerdem ist eine mit Qwik geschriebene Anwendung in allen Fallstudien schneller als die Konkurrenz geladen (Zeit zur Einblendung des ersten Inhalts - Kapitel 5.4.2).

Trotz der zu Beginn etwas umständlichen Art der Programmierung entsteht kein Mehraufwand für den Programmierer. Wie diese Arbeit zeigt, kann jede Art einer Anwendung effizient im Qwik Web-Framework nachgebaut werden, wenn man sich einmal daran gewöhnt hat, *Qwik zu denken.*

Das Qwik Web-Framework hat sich während dieser Arbeit von Version *0.0.108* zur Version *0.16.2* weiterentwickelt (Abbildung 6.1). Dabei sind stetig Funktionen und Verbesserungen hinzugekommen. Die Idee und das Konzept des Frameworks stimmen, die Ergebnisse in dieser Arbeit versprechen eine gute Zukunft.

Abbildung 6.1: Entwicklung der in dieser Arbeit genutzten Versionen des Qwik Web-Framework (Eigene Darstellung)

Nun heißt es Abwarten, ob es sich als Web Frontend-Frameworks gegen andere behaupten kann. Neue Frameworks gibt es viele und trotzdem werden im Jahr 2022 die meisten Webseiten immer noch mit jQuery betrieben (vgl. HTTP Archive, 2022b, Library usage). Dies zeigt wie unglaublich schwer es ist, sich gegen fertige, schon genutzte Software durchzusetzen. Bei neuen Projekten lohnt es sich aber durchaus zu prüfen, ob Qwik denn eine produktionsreife Version zur Verfügung stellt.

Anschließend an diese Arbeit könnten weitere Tests mit ähnlicheren Frameworks durchgeführt werden. Astro[66] verspricht u. a. mit einer Aufteilung der Seite ebenfalls das anfänglich geladene JS zu reduzieren. Es setzt aber weiterhin in den verschiedenen Bereichen Hydrierung ein. Somit könnte ein direkter Unterschied zwischen Wiederaufnehmbarkeit und Hydrierung analysiert werden.

Ebenfalls sind Optimierungen der Fallstudien für die jeweiligen Frameworks interessant. Die Art der Programmierung unterscheidet sich in dieser Arbeit kaum voneinander. Qwik und Nuxt haben viele Funktionen, die so nicht genutzt werden. Es könnten z. B. Daten auf dem Server vorbereitet werden, um eine schnellere Gesamtladezeit zu erreichen.

66 astro.build

Literaturverzeichnis

Buch-Quellen

WEIR, Luis; Zdenek „Z“ NEMEC, 2019. *Enterprise API Management : Design and Deliver Valuable Business APIs.* Packt Publishing. Expert Insight. ISBN 9781787284432.

Online-Quellen

ALPINE LINUX DEVELOPMENT TEAM, 2022. *Small. Simple. Secure.* [online]. [besucht am 27. Dez. 2022]. Abger. unter: `https://alpinelinux.org/`.

ANDREASONNY83, 2022. *Lighthouse-Ci Dockerfile* [online]. [besucht am 16. Nov. 2022]. Abger. unter: `https://github.com/andreasonny83/lighthouse-ci/blob/main/demo/Dockerfile`.

BANSAL, Shubham, 2022. *REST API (Introduction)* [online]. [besucht am 6. Sep. 2022]. Abger. unter: `https://www.geeksforgeeks.org/rest-api-introduction/`.

BUILDER.IO INC, 2021. *The HTML-first framework* [online]. [besucht am 2. Sep. 2022]. Abger. unter: `https://github.com/BuilderIO/qwik`.

BUILDER.IO INC, 2022. *Qwik Framework* [online]. [besucht am 2. Sep. 2022]. Abger. unter: `https://qwik.builder.io/`.

CHENJIANDONGX, 2022. *go-echarts* [online]. [besucht am 15. Dez. 2022]. Abger. unter: `https://github.com/go-echarts/go-echarts`.

CHROME DEVELOPERS, 2016. *Overview - Lighthouse* [online]. [besucht am 6. Okt. 2022]. Abger. unter: `https://developer.chrome.com/docs/lighthouse/overview/`.

CHROME DEVELOPERS, 2019. *Avoid an excessive DOM size* [online]. [besucht am 8. Jan. 2023]. Abger. unter: `https://developer.chrome.com/docs/lighthouse/performance/dom-size/`.

CLOUDFLARE, 2022. *What is a CDN? | How do CDNs work?* [online]. [besucht am 19. Okt. 2022]. Abger. unter: `https://www.cloudflare.com/en-gb/learning/cdn/what-is-a-cdn/`.

CRUTCHLEY, Corbin, 2020. *What is Server Side Rendering (SSR) and Static Site Generation (SSG)?* [online]. [besucht am 12. Sep. 2022]. Abger. unter: `https://unicorn-utterances.com/posts/what-is-ssr-and-ssg`.

DOCKER INC., 2022. *Develop faster. Run anywhere.* [online]. [besucht am 2. Jan. 2023]. Abger. unter: `https://www.docker.com`.

ELROM, Eli Elad, 2020. *Top 10 Optimizing Techniques Every React developer should know! Tips to optimize your ReactJS App.* [online]. [besucht am 9. Jan. 2023]. Abger. unter: `https://medium.com/react-courses/optimize-react-app-best-optimzing-techniques-i-wish-i-knew-before-i-wrote-my-first-line-of-code-2b4651f45a48`.

EXCALIDRAW, 2022. *Excalidraw Libraries* [online]. [besucht am 19. Sep. 2022]. Abger. unter: `https://github.com/excalidraw/excalidraw-libraries`.

GITHUB, 2022. *Licenses* [online]. [besucht am 19. Okt. 2022]. Abger. unter: `https://choosealicense.com/licenses/`.

GOOGLE, LLC, 2004. *Lighthouse* [online]. [besucht am 2. Jan. 2023]. Abger. unter: `https://github.com/GoogleChrome/lighthouse`.

GOOGLE, LLC, 2009. *The Go Programming Language* [online]. [besucht am 2. Jan. 2023]. Abger. unter: `https://github.com/golang/go`.

GOOGLE, LLC; Sebastian KREFT, 2017. *Lighthouse Configuration* [online]. [besucht am 14. Dez. 2022]. Abger. unter: `https://github.com/GoogleChrome/lighthouse/blob/main/docs/configuration.md`.

GRAFANA LABS, 2014. *grafana/grafana: The open and composable observability and data visualization platform. Visualize metrics, logs, and traces from multiple sources like Prometheus, Loki, Elasticsearch, InfluxDB, Postgres and many more.* [online]. [besucht am 14. Jan. 2023]. Abger. unter: `https://github.com/grafana/grafana`.

HACZEK, Artur, 2020. *Angular Tree Shaking* [online]. [besucht am 19. Dez. 2022]. Abger. unter: `https://www.angular.love/en/2020/12/21/angular-tree-shaking-2/`.

HEVERY, Miško, 2021. *How we cut 99 % of our JavaScript with Qwik + Partytown* [online]. [besucht am 2. Sep. 2022]. Abger. unter: `https://dev.to/builderio/how-we-cut-99-of-our-javascript-with-qwik-partytown-3i3k`.

HEVERY, Miško, 2022a. *Hydration is Pure Overhead* [online]. [besucht am 12. Sep. 2022]. Abger. unter: `https://www.builder.io/blog/hydration-is-pure-overhead`.

HEVERY, Miško, 2022b. *Our current frameworks are O(n); we need O(1)* [online]. [besucht am 6. Okt. 2022]. Abger. unter: `https://www.builder.io/blog/our-current-frameworks-are-on-we-need-o1`.

HOWDLE, Dan, 2022. *Worldwide mobile data pricing 2022* [online]. [besucht am 15. Dez. 2022]. Abger. unter: https://www.cable.co.uk/mobiles/worldwide-data-pricing/.

HTTP ARCHIVE, 2022a. *Report: Page Weight* [online]. [besucht am 6. Okt. 2022]. Abger. unter: https://httparchive.org/reports/page-weight?start=earliest&end=latest&view=list.

HTTP ARCHIVE, 2022b. *Web Almanac HTTP Archive's annual state of the web report 2022* [online]. [besucht am 15. Dez. 2022]. Abger. unter: https://almanac.httparchive.org/en/2022.

IRABOR, Jordan, 2019. *How To Set Up a Node Project With Typescript* [online]. [besucht am 16. Nov. 2022]. Abger. unter: https://www.digitalocean.com/community/tutorials/setting-up-a-node-project-with-typescript.

IVANOVS, Alex, 2022. *The Most Popular Front-end Frameworks in 2022* [online]. [besucht am 5. Sep. 2022]. Abger. unter: https://stackdiary.com/front-end-frameworks/.

KELLY, Daniel, 2022. *Building a Multi-Step Form with Petite-Vue* [online]. [besucht am 26. Okt. 2022]. Abger. unter: https://vueschool.io/articles/vuejs-tutorials/building-a-multi-step-form-with-petite-vue/.

KHAYRULLIN, Arthur, 2022. *What is Google Lighthouse and why you should use it* [online]. [besucht am 9. Jan. 2023]. Abger. unter: https://uploadcare.com/blog/what-is-google-lighthouse/.

KOTECHA, Radha, 2019. *How to Use Headless Chrome in Golang with Godet* [online]. [besucht am 28. Dez. 2022]. Abger. unter: https://www.bacancytechnology.com/blog/use-headless-chrome-in-golang-with-godet.

KUMAR, Sanjay, 2021. *How to Send Email Using Gmail SMTP in Node Js* [online]. [besucht am 2. Jan. 2023]. Abger. unter: https://onlinewebtutorblog.com/how-to-send-email-using-gmail-smtp-in-node-js/.

LANG, Lea, 2016. *Mobile Internetnutzung weiter auf dem Vormarsch* [online]. [besucht am 13. Nov. 2022]. Abger. unter: https://www.heise.de/newsticker/meldung/Mobile-Internetnutzung-weiter-auf-dem-Vormarsch-3455624.html%3fview%3dzoom%3bzoom%3d3.

MATERIAL DESIGN, 2022. *Material Design Icons* [online]. [besucht am 26. Sep. 2022]. Abger. unter: https://github.com/Templarian/MaterialDesign/.

META PLATFORMS, INC., 2022. *React - A JavaScript library for building user interfaces* [online]. [besucht am 20. Dez. 2022]. Abger. unter: https://reactjs.org/.

MOZILLA CORPORATION, 2022a. *Ajax* [online]. [besucht am 7. Sep. 2022]. Abger. unter: https://developer.mozilla.org/en-US/docs/Web/Guide/AJAX.

MOZILLA CORPORATION, 2022b. *Introduction to the DOM* [online]. [besucht am 19. Okt. 2022]. Abger. unter: `https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction`.

MOZILLA CORPORATION, 2022c. *JSON.stringify()* [online]. [besucht am 10. Okt. 2022]. Abger. unter: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON/stringify`.

NGINX, 2022. *GZIP Module* [online]. [besucht am 14. Nov. 2022]. Abger. unter: `https://nginx.org/en/docs/http/ngx_http_gzip_module.html`.

NODE CONTRIBUTORS, 2005. *Node.js* [online]. [besucht am 2. Jan. 2023]. Abger. unter: `https://github.com/nodejs/node`.

NONSTOP IO, 2022. *Minification — A must-know tool in (web) developer's toolkit* [online]. [besucht am 9. Jan. 2023]. Abger. unter: `https://blog.nonstopio.com/minification-a-must-know-tool-in-web-developers-toolkit-72d65e3e2b04`.

NUXT PROJECT, 2022. *The Intuitive Web Framework* [online]. [besucht am 20. Dez. 2022]. Abger. unter: `https://nuxt.com/`.

PAVIC, Bojan; Chris ANSTEY; Jeremy WAGNER, 2022. *Why does speed matter?* [online]. [besucht am 6. Okt. 2022]. Abger. unter: `https://web.dev/why-speed-matters/`.

PEXELS, 2022. *Free Stock Photo and Video License* [online]. [besucht am 9. Nov. 2022]. Abger. unter: `https://www.pexels.com`.

POTSCHIEN, Denis, 2016. *Why You Should Embed SVG Inline* [online]. [besucht am 15. Dez. 2022]. Abger. unter: `https://www.jotform.com/blog/why-you-should-embed-svg-inline-98570/`.

REST COUNTRIES, 2022. *Get information about countries via a RESTful API* [online]. [besucht am 25. Sep. 2022]. Abger. unter: `https://restcountries.com`.

SAADEGHI, Pouya, 2022. *DaisyUI* [online]. [besucht am 26. Okt. 2022]. Abger. unter: `https://github.com/saadeghi/daisyui`.

SHAH, Karan, 2019. *Client-side Vs. Server-side Rendering: What to choose when?* [online]. [besucht am 6. Sep. 2022]. Abger. unter: `https://medium.com/solute-labs/client-side-vs-server-side-rendering-what-to-choose-when-dd1620fb2808`.

SIDDIQUE, Sharjeel, 2020. *What does it mean by Javascript is single threaded language* [online]. [besucht am 13. Jan. 2023]. Abger. unter: `https://medium.com/swlh/what-does-it-mean-by-javascript-is-single-threaded-language-f4130645d8a9`.

SOLARWINDS WORLDWIDE, 2018. *Can gzip Compression Really Improve Web Performance?* [online]. [besucht am 14. Nov. 2022]. Abger. unter: `https://www.pingdom.com/blog/can-gzip-compression-really-improve-web-performance/`.

TAILWIND LABS INC., 2022a. *Install Tailwind CSS with Create React App* [online]. [besucht am 20. Dez. 2022]. Abger. unter: `https://tailwindcss.com/docs/guides/create-react-app`.

TAILWIND LABS INC., 2022b. *Install Tailwind CSS with Nuxt.js* [online]. [besucht am 20. Dez. 2022]. Abger. unter: `https://tailwindcss.com/docs/guides/nuxtjs`.

TAILWIND LABS INC., 2022c. *Install Tailwind CSS with Vite* [online]. [besucht am 20. Dez. 2022]. Abger. unter: `https://tailwindcss.com/docs/guides/vite#vue`.

TAILWIND LABS, INC., 2022. *TailwindCSS* [online]. [besucht am 26. Okt. 2022]. Abger. unter: `https://github.com/tailwindlabs/tailwindcss`.

THOMPSON, Jeremy L, 2021. *Big O Notation Examples – Time Complexity and Algorithm Efficiency Explained* [online]. [besucht am 6. Okt. 2022]. Abger. unter: `https://www.freecodecamp.org/news/big-o-notation-examples-time-complexity-explained/`.

TRAEFIK LABS, 2016a. *Documentation* [online]. [besucht am 27. Dez. 2022]. Abger. unter: `https://doc.traefik.io/traefik/`.

TRAEFIK LABS, 2016b. *traefik/traefik: The Cloud Native Application Proxy* [online]. [besucht am 2. Jan. 2023]. Abger. unter: `https://github.com/traefik/traefik`.

TRAEFIK LABS, 2022. *Reverse Proxy Explained: How It Works And When You Need One* [online]. [besucht am 25. Nov. 2022]. Abger. unter: `https://traefik.io/glossary/reverse-proxy/`.

VEGA, Cristian, 2017. *Client-side vs. server-side rendering: why it's not all black and white* [online]. [besucht am 6. Sep. 2022]. Abger. unter: `https://www.freecodecamp.org/news/what-exactly-is-client-side-rendering-and-hows-it-different-from-server-side-rendering-bd5c786b340d/`.

YOU, Evan, 2014. *The Progressive JavaScript Framework* [online]. [besucht am 20. Dez. 2022]. Abger. unter: `https://vuejs.org`.

ZAFACO, GmbH, 2022. *Jahresbericht 2020/21 - Mobile Breitbandanschlüsse* [online]. [besucht am 13. Nov. 2022]. Abger. unter: `https://download.breitbandmessung.de/bbm/Breitbandmessung_Jahresbericht_2020_2021_mobil.pdf`.