Hochschule für angewandte Wissenschaften Würzburg-Schweinfurt  
Fakultät Informatik und Wirtschaftsinformatik

Bachelorarbeit

Vergleich von Rendering Strategien anhand deren Performance am Anwendungsfall einer auf Next.js basierenden Musik Streaming Web App

vorgelegt an der Hochschule für angewandte Wissenschaften  
Würzburg-Schweinfurt in der Fakultät Informatik und Wirtschaftsinformatik zum  
Abschluss eines Studiums im Studiengang E-Commerce

Kai Schäfer

Matrikelnummer: 6118066

Eingereicht am: 14. November 2019

Erstprüfer: Prof. Dr. Rolf Schillinger  
Zweitprüfer: Prof. Dr. C D



**Zusammenfassung**

TODO

**Abstract**

TODO

Inhaltsverzeichnis

[Abkürzungsverzeichnis 4](#_Toc107749701)

[1 Einführung 5](#_Toc107749702)

[2 Grundlagen 6](#_Toc107749703)

[2.1 Das Client-Server Model 6](#_Toc107749704)

[2.2 Wie Websites bereitgestellt werden 6](#_Toc107749705)

[2.3 Rendering 7](#_Toc107749706)

[2.4 Rendering Strategien 7](#_Toc107749707)

[2.4.1 Warum der Vergleich von Rendering Strategien? 8](#_Toc107749708)

[2.4.2 Wie Next.js einen genauen Vergleich möglich macht 8](#_Toc107749709)

[2.5 Performance 8](#_Toc107749710)

[2.5.1 Wichtigkeit von Performance im Web 9](#_Toc107749711)

[2.5.2 Performance Metriken 10](#_Toc107749712)

[2.6 Server Side Rendering 11](#_Toc107749713)

[2.6.1 Server Side Rendering mit Next 12](#_Toc107749714)

[2.6.2 Vorteile von Server-Side-Rendering 13](#_Toc107749715)

[2.6.3 Nachteile von Server-Side-Rendering 13](#_Toc107749716)

[2.7 Client-Side-Rendering 14](#_Toc107749717)

[2.7.1 Client-Side-Rendering mit Next 16](#_Toc107749718)

[2.7.2 Vorteile von Client-Side-Rendering 17](#_Toc107749719)

[2.7.3 Nachteile von Client-Side-Rendering 17](#_Toc107749720)

[2.8 Isomorphic Web Applications 18](#_Toc107749721)

[2.8.1 Chancen von isomorphic Web-Applications 19](#_Toc107749722)

[2.8.2 Risiken von isomorphic Web-Applications 19](#_Toc107749723)

[2.9 Static Side Generation 20](#_Toc107749724)

[2.9.1 Static Side Generation mit Next 21](#_Toc107749725)

[2.9.2 Vorteile von Static Side Generation 21](#_Toc107749726)

[2.9.3 Nachteile von Static Side Generation 21](#_Toc107749727)

[Abbildungsverzeichnis 22](#_Toc107749728)

[Literaturverzeichnis 23](#_Toc107749729)

[Eidesstattliche Erklärung 25](#_Toc107749730)

[Zustimmung zur Plagiatsüberprüfung 25](#_Toc107749731)

# Abkürzungsverzeichnis

TCP/IP = Transmission Control Protocol and Internet Protocol

HTTP = Hypertext Transfer Protocol

DNS = Domain Name System

URL = Uniform Resource Locator

DOM = Document Object Model

CSSOM = CSS Object Model

HTML = Hypertext Markup Language

CSS = Cascading Style Sheets

SEO = Search Engine Optimization

CSR = Client-Side-Rendering

SSR = Server-Side-Rendering

SSG = Static-Side-Generation

ISG = Incremental-Side-Generation

CDN = Content Delivery Network

FCP = First Contentful Paint

LCP = Largest Contentful Paint

TTI = Time To Interactive

SI = Speed Index

TTFB = Time To First Byte

TBT = Total Blocking Time

CLS = Cumulative Layout Shift

# Einführung

TODO

# Grundlagen

Um die genauen Unterschiede zwischen den Rendering-Verfahren untersuchen zu können, müssen vorerst die theoretischen Grundlagen über das Bereitstellen von Inhalten im Web geklärt werden. In den folgenden Abschnitten soll aufgezeigt werden, welche Technologien eingesetzt werden, um Inhalte, die von einem Server bereitgestellt werden, auf dem Endgerät eines Benutzers anzuzeigen.

## Das Client-Server Model

Um zu entscheiden, ob das Rendering auf dem Client oder Server geschehen soll, muss vorerst geklärt werden was diese überhaupt sind und wie sie miteinander kommunizieren. Bei dem Client-Server Model handelt es sich um eine Software-Architektur, bei welcher der Client eine bestimmte Ressource von dem Server anfragt, welcher darauf antwortet und die angeforderten Daten übermittelt (Oluwatosin, 2014, S. 67). Wenn ein User eine Website besucht, wäre in diesem Fall sein internetfähiges Endgerät, beziehungsweise der darauf installierte Browser, der Client. Als Server bezeichnet man den Service, welcher die bei sich gespeicherte Website ausliefert. Beide Parteien kommunizieren dabei über das Internet mittels des TCP/IP Protokolls. Der Client sendet einen HTTP Request an die IP-Adresse des Servers, in welchem er die angeforderten Ressourcen und weitere Informationen, wie Cookies, Autorisierung, Metadaten oder den Content Typ angibt. Wenn der Server die Anfrage empfängt, überprüft er, ob die Bedingungen einer Datenübermittlung an diesen Client erfüllt sind. Ist die Anfrage genehmigt, sendet er seine Response mit dem Status Code 200 zurück an den Client. Diese Meldung sagt aus, dass die Anfrage akzeptiert wurde, woraufhin dann der der Server die Daten übermittelt (MDN contributers, 2022).

## Wie Websites bereitgestellt werden

Neben Clients und Servern gibt es noch weitere wichtige Technologien, die bei der Bereitstellung einer Website eine Rolle spielen. Im Folgenden wird schrittweise aufgezeigt, welche Prozesse zwischen der Anfrage einer Website bis zu dessen Anzeige im Browser durchlaufen werden.

Gibt ein User eine URL in die Suchleiste seines Browsers ein, wird nicht direkt ein Request an den Web-Server gesendet, da der Client dafür vorerst die IP-Adresse der Website kennen muss. Dazu sendet er eine Anfrage an ein Domain Name System (DNS). Dieses sucht zu der gewünschten URL die dazugehörige IP-Adresse heraus und sendet sie zurück an den Client (MDN contributers, 2022).

Da der Client nun weiß wo die Daten der Website zu finden sind, sendet er den HTTP Request an den Server mit dieser IP-Adresse. Dieser prüft nun die Anfrage und antwortet entweder mit einer abweisenden Response, die einen Fehlercode und keine Daten enthält, oder einem Status Code 200, woraufhin die Übermittlung der Daten folgt. Diese Daten unterscheiden sich in Code-Files, was hauptsächlich HTML, CSS und JavaScript Dateien sind, und Assets. Dabei handelt es sich um sämtliche Multimedia-Dateien, wie Audios, Bilder, Videos oder andere Dokumente. Die Übermittlung der Daten erfolgt über sogenannte Chunks. Dies sind kleine Datenpakete, die alle einzeln durch das Web transferiert werden. Sie machen die Übertragung schneller und stabiler und ermöglichen es, dass mehrere Clients zur selben Zeit auf eine Website zugreifen können. Denn wenn alle Daten in einem großen Packet verschickt werden würden, könnte nur ein User zur selben Zeit die Website aufrufen (2022).

## Rendering

Hat der Client alle Pakete empfangen, fügt er diese zusammen und generiert daraus die fertige Website. Dazu parst der Browser zu Beginn den HTML Code, woraus sich der Document Object Model (DOM) Baum zusammenfügt. Stößt er dabei auf ein script Tag unterbricht er den Vorgang und fragt den JavaScript Code beim Server an, der dort referenziert ist. Erst, wenn er die Ressource erhalten hat, fährt er fort. Dies gilt auch für externe CSS-Stylesheets, welche über den link Tag in das HTML eingebunden werden. Das CSS wird anschließend ebenfalls geparst, woraus das CSS Object Model (CSSOM) entsteht. Und auch der JavaScript Code wird geparst und dann kompiliert (MDN contributers, 2022). Nachdem das Parsing abgeschlossen ist, werden das DOM und CSSOM zu einem Render Baum vereinigt, während das JavaScript ausgeführt wird. Bei diesem Prozess werden alle Elemente entfernt, die aufgrund eines display:none nicht auf der späteren Website angezeigt werden. Nun folgt der Layout Vorgang, bei welchem für jedes Element dessen Position auf dem Bildschirm ermittelt wird. Dazu werden Attribute wie width, height oder position verwendet. Nachdem die Elemente positioniert wurden, folgt der Paint Prozess, bei welchem Attribute, wie color, box-shadow oder border ausgelesen werden, um die Inhalte farblich auf dem Bildschirm des Users darzustellen. Der Rendering Prozess ist damit abgeschlossen und der User kann mit der Website interagieren (Saurabh Daware, 2020).

## Rendering Strategien

So wie sich das Web über die Jahre immer weiterentwickelt hat, so haben sich auch verschiedene Rendering Strategien etabliert. Es ist grundsätzlich möglich das Rendern im Browser wie auch auf dem Server selbst auszuführen. Dabei kann man ebenfalls auf unterschiedliche Weise vorgehen. Doch was begründet den Aufwand sich mit den verschiedenen Strategien zu beschäftigen?

### Warum der Vergleich von Rendering Strategien?

In der Softwareentwicklung geht es nicht nur darum eine Lösung für das bestehende Problem zu kreieren, sondern auch die richtige Architektur dabei zu wählen (Fadhilah Iskandar, Lubis, Fabrianti Kusumasari & Ridho Lubis, 2020, S. 2). Dabei ist die Entscheidung für eine Strategie im Vorzug vor einer anderen eine der ersten grundsätzlichen Schritte, wovon das spätere Vorgehen abhängt. Das Ändern dieser Strategie ist meist mit hohem Aufwand verbunden. Twitter hat beispielsweise ein Jahr lang mit vierzig Entwicklern daran gearbeitet das Rendering vom Client auf den Server zu verschieben. Dadurch haben sie nach eigenen Aussagen eine fünf Mal schnellere Ladezeit erreichen können (Brehm, 2013; Webb, 2012). An diesem Beispiel lässt sich erkennen, dass es für Unternehmen durchaus von Wichtigkeit ist, sich im Vorfeld für die richtige Strategie zu entscheiden, um bei ihrer Anwendung eine akzeptable Performance zu erreichen. Die Rendering Verfahren wirken sich ebenfalls deutlich auf weitere wichtige Faktoren, wie SEO aus (Fadhilah Iskandar et al., 2020, S. 5). Generell lässt sich Software anhand vieler Faktoren unterscheiden, wie beispielweise der Wartbarkeit, Skalierbarkeit oder dem Entwicklungsaufwand (Beke, 2018, S. 34). In dieser Arbeit wird jedoch ausschließlich der Einfluss auf die Performance untersucht. In einem späteren Kapitel wird aufgezeigt anhand welcher Metriken die Rendering Verfahren voneinander unterschieden werden.

### Wie Next.js einen genauen Vergleich möglich macht

Next.js ist ein auf React basierendes Framework, mit welchem man durch erweiterte Funktionen und Verbesserungen auf einfache Weise schnelle Web-Applikationen entwickeln kann. Next lässt sich ohne extra Konfiguration an React anbinden und bietet Lösungen für übliche Aufgaben, wie Data-Fetching, Integrationen oder Routing. Gerade die Auswahl an Rendering Verfahren, die das Framework bereitstellt, kommen der Untersuchung dieser Arbeit zugute. Next bietet nämlich die Möglichkeit individuell für jede Seite eines der folgenden Rendering Strategien zu verwenden: Client-Side-Rendering (CSR), Server-Side-Rendering (SSR), Static-Side-Generation (SSG) und Incremental-Side-Generation (ISG) (Lazuardy & Anggraini, 2022, S. 135). Untersucht man wie sich die Rendering Strategien hinsichtlich ihrer Performance voneinander unterscheiden, ist es von großer Wichtigkeit, dass die Ergebnisse nicht durch andere Faktoren verfälscht werden, welche Einfluss auf die Ladegeschwindigkeit der Seite haben. Dadurch, dass Next einfache Methoden von Haus aus zur Verfügung stellt, müssen keine größeren Umbauten am Code vorgenommen werden, wodurch mögliche Veränderungen der Ergebnisse auf das Ändern des Rendering Verfahrens zurückgeführt werden können.

## Performance

Wie bereits angemerkt, wird sich der Vergleich der Rendering Strategien in dieser Arbeit auf die Performance spezialisieren. Dabei ist zu klären, wie man diese messen kann, welche Metriken dafür sinnvoll sind und wieso eine Untersuchung in diesem Bereich von Bedeutung ist.

Es wird generell zwischen serverseitiger und clientseitiger Performance unterschieden. Bei dem Client untersucht man die initiale und die darauffolgenden Ladezeiten und bei dem Server meist den Datendurchsatz oder die Bandbreite. Auf der Client Seite ist im Webumfeld besonders zu beachten, wie oft und in welchem Ausmaß das DOM manipuliert wird. Dies hat eine wichtige Bedeutung, da Änderungen in der Baumstruktur einen weitreichenden Prozess auslösen, in welchem die Position von Elementen neu kalkuliert und die geänderten Komponenten neu gepaintet werden müssen. Das betrifft oft mehrere Elemente, die miteinander verschachtelt sind (Lindsay, 2022). Da Manipulationen des DOMs langsam sind, können Änderungen oft eine lange Zeit in Anspruch nehmen, was die Performance negativ beeinflusst (Psaila, 2008, S. 233–237).

### Wichtigkeit von Performance im Web

Google hat es sich 2017 zur Aufgabe gemacht die Absprungraten der User von diversen Websites (Bounce) in Korrelation zu deren Ladezeit zu untersuchen. Dafür haben sie ein neuronales Netz entwickelt, welches mit einem Datensatz aus Conversions, Bounces und Page Speed trainiert wurde. Dabei haben sie herausgefunden, dass sich bei einer Website, welche eine Ladezeit von einer Sekunde bis zehn Sekunden hat, die Absprungrate von Benutzern auf mobilen Endgeräten um 123 Prozent erhöht. Wie in „Abbildung 2 Korrelation von Ladezeit und Absprungrate“ zu erkennen ist, steigen die Bounces mit der Dauer der Ladezeit (An, 2018).

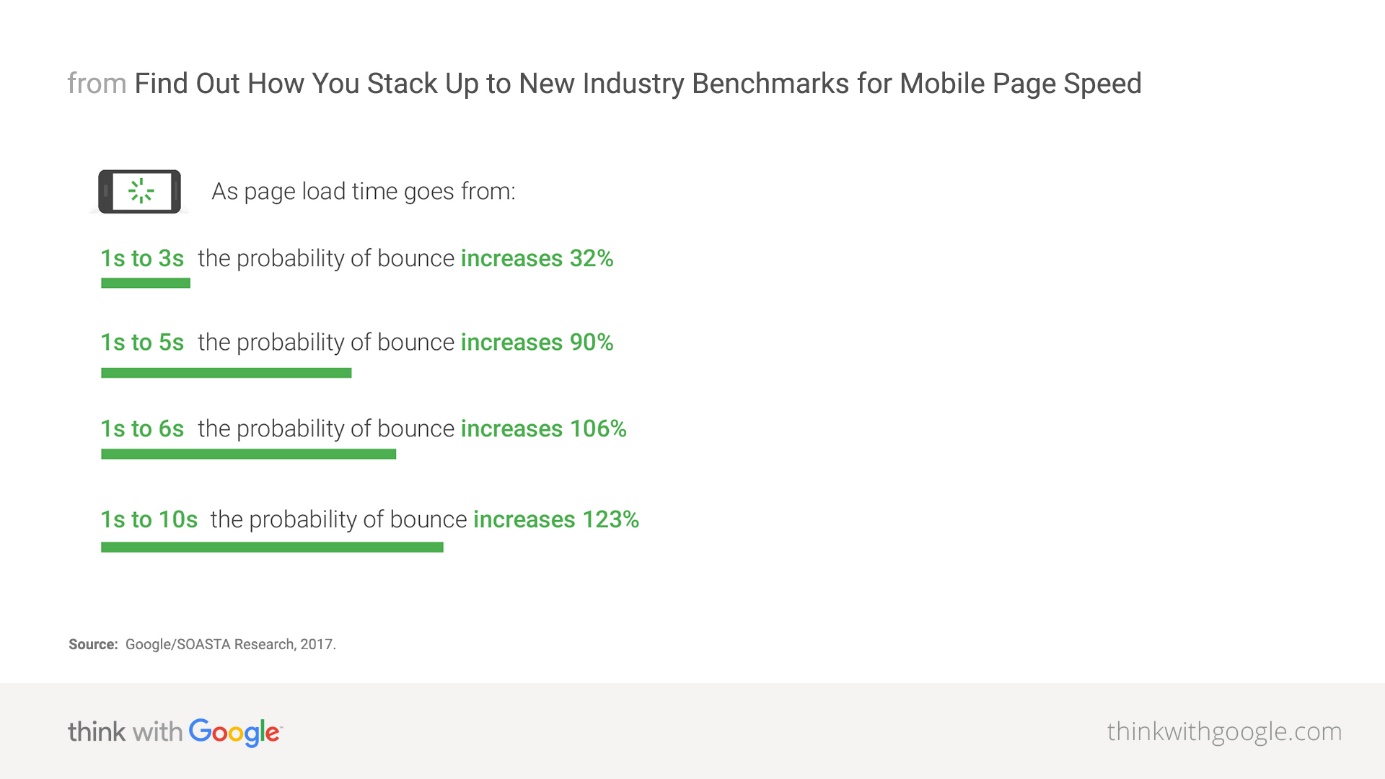


Abbildung Korrelation von Ladezeit und Absprungrate.

Quelle: Google/SOASTA Research, 2017

Darüber hinaus erläutert An, dass nach Untersuchungen von Google im Jahr 2019 die durchschnittliche Ladezeit einer mobilen Website bei fünfzehn Sekunden liegt. Dies sei eine zu hohe Zahl in Anbetracht der Zusammenhänge von der Ladezeit zu der Anzahl an Seitenbesuchen und damit der Höhe der Konversionen (2018).

Egger et al. liefern einen tiefergehenden Hinweis darauf, wie die das Warten auf das vollständige Laden einer Website von einem User wahrgenommen wird. Sie führen auf, dass Menschen die Dauer der Wartezeit subjektiv auf eine andere Weise wahrnehmen als sie objektiv tatsächlich ist. Daraus haben sich drei zeitliche Grenzen ergeben, bis zu welchen der Benutzer eine gewisse Gemütslage gegenüber der Website hat (2012, S. 87–88).

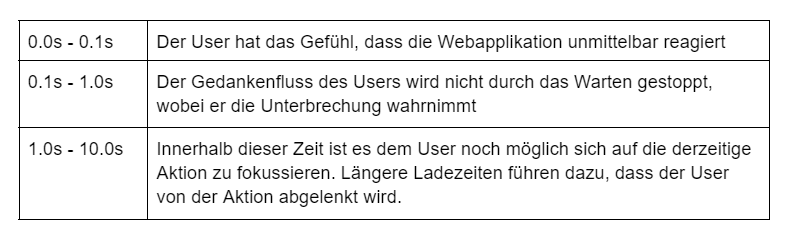


Abbildung Wartezeiten in Korrelation zum Empfinden des Users

Quelle: (Egger et al., 2012, S. 88)

Es lässt sich also erkennen, dass die Performance einer Website einen entscheidenden Faktor für die User Experience darstellt und damit ausschlaggebend für den Erfolg einer Web-Applikation ist. Lädt eine Webpage nicht schnell genug, wird der User unruhig und verlässt die Seite. Es ist demnach von großer Wichtigkeit eine gute Performance zu erreichen und sie stetig zu verbessern. Doch wie kann man dies messen und was sagen die unterschiedlichen Metriken aus?

### Performance Metriken

Es gibt mehrere Metriken, die verwendet werden können, um eine Aussage über die Performance einer Webapplikation zu machen. In dieser Arbeit werden folgende Metriken untersucht, um einen messbaren Unterschied zwischen den Rendering Strategien zu erkennen:

* First Contentful Paint (FCP): misst die Zeit, welche es braucht, bis der erste Content angezeigt wird. Darunter werden Texte, Bilder, SVGs und canvas Elemente gezählt.
* Largest Contentful Paint (LCP): misst die Zeit, welche es braucht, um den größten sichtbaren Text- oder Bild-Block innerhalb des Viewports im Browser anzuzeigen.
* Time To Interactive (TTI): misst die Zeit, welche es braucht, damit die Anwendung vollständig interaktiv ist. Das bedeutet, bis alle Event Handler vorhanden sind und die Applikation auf Eingaben des Benutzers reagieren kann.
* Speed Index (SI): misst die Zeit, welche es braucht, bis der sichtbare Content einer Seite angezeigt wird.
* Time To First Byte (TTFB): misst die Zeit, welche es braucht, bis der erste Byte der Response angekommen ist.
* Total Blocking Time (TBT): misst die Dauer, in welcher die Anwendung geblockt ist und nicht auf Usereingaben reagieren kann.
* Cumulative Layout Shift (CLS): ist eine Kennzahl, welche angibt, wie sehr sich die Inhalte auf einer Webpage während des Ladens und nach dessen Fertigstellung unerwartet, also ohne Zutun des Users, verschieben (Pavic & Brkic, 2021, S. 1756).

Mit diesen Metriken können messbare Werte miteinander verglichen und Aussagen über die Performanceunterschiede der Rendering Strategien getroffen werden. In den folgenden Abschnitten werden diese Verfahren nun erläutert.

## Server Side Rendering

Bevor JavaScript Frameworks populär wurden, war serverseitiges Rendern der Standard in der Web-Entwicklung. Programmierer verwendeten dafür meist PHP oder ASP.NET (Lazuardy & Anggraini, 2022, S. 134; Pavic & Brkic, 2021, S. 1754). Das Konzept des Server Side Renderings ist es, dass der Server die benötigte HTML Seite selbst generiert und dem Client on-demand ausliefert.

Bekommt der Server folglich den Request des Clients für eine Seite, fängt er an den HTML Code zu generieren. Dazu verwendet er mögliche Metainformationen, wie Session Daten oder Cookies, welche der Client in seiner Anfrage mitschickt, um dynamische Inhalte zu füllen. Er holt sich zudem, falls vorhanden, Inhalte aus Datenbanken oder externen Services, um alle benötigten Daten und Assets zu erlangen, sodass er das komplette HTML Dokument generieren kann. Wenn der Server nun den Prozess abgeschlossen hat, sendet er den zusammengestellten Code an den Browser, welcher nur noch den Layout-, Paint-, und Hydration-Prozess durchführen muss (Kishore & B M, 2020, S. 1–10).

Das Rendern ist demnach komplett in der Verantwortung des Servers. Das bedeutet jedoch auch, dass bei jedem Klick des Users auf einen Link, welcher zu einer neuen Seite innerhalb der Website führt, ein neuer Request an den Server und damit ein komplett neuer Render-Prozess angestoßen wird. Dabei ist es nicht von Bedeutung, ob der Inhalt sich nur marginal unterscheidet oder komplett anders ist. Sobald neuer Content gebraucht wird, muss der Server die Seite erneut rendern. Mit der Ausnahme, dass die Webpage nicht im Cache des Browsers vorhanden ist, denn dann würde sich der Client von dort den Code beschafften und muss nicht den Server dafür anfragen (Vega, 2017).

Verwendet eine Anwendung serverseitiges Rendern, spricht man von einer Multipage Application, da es mehrere Seiten gibt, welche alle einzeln ausgeliefert werden (Fadhilah Iskandar et al., 2020, S. 3). Da diese Webpages alle im Voraus vom Server gerendert werden und der Client nur noch den generierten Code anzeigen muss, wird der Vorgang auch als Pre-Rendering bezeichnet. Bei diesem Verfahren wird direkt beim initialen Laden der Seite der Content komplett angezeigt, da er schon vom Server gerendert wurde. Der User sieht demnach direkt den vollen Inhalt. Nachdem der HTML und CSS-Code gepaintet wurde, folgt der Hydration Prozess. Dabei wird das JavaScript, was der Server mitgeliefert hat ausgeführt. Es handelt sich dabei nur um den Code, der auch wirklich für diese Seite gebraucht wird, um interaktiv zu werden. Es werden bei der Ausführung Event-Handler an die HTML Elemente angefügt. Dadurch werden aus einer statischen Seite dynamische Inhalte, mit welchen der User interagieren kann. Das bedeutet also, dass es während des Hydration Prozesses ein kleines Zeitfenster gibt, in welchem die Seite komplett angezeigt wird, aber nicht interaktiv ist (Lazuardy & Anggraini, 2022, S. 139).

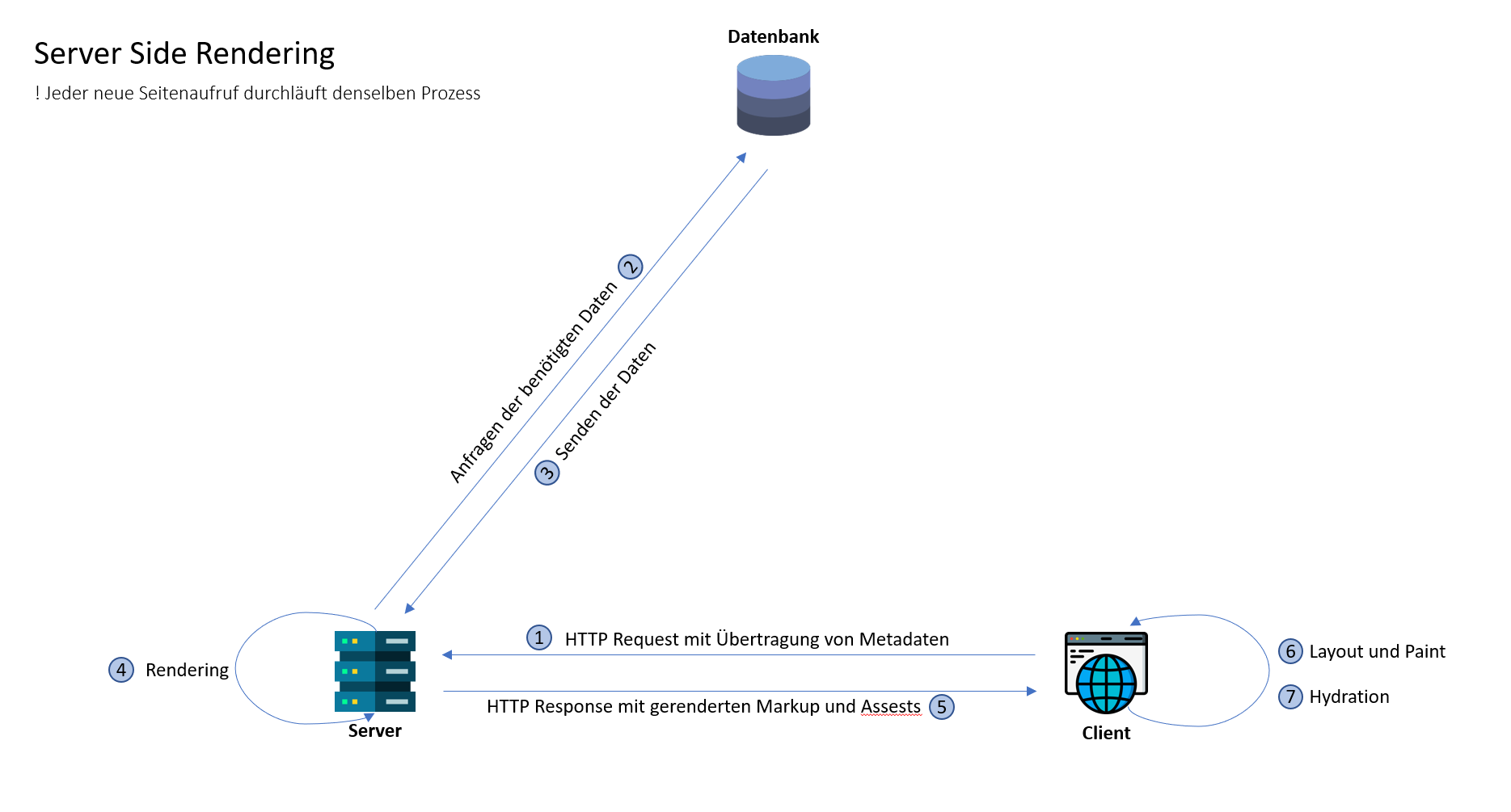


Abbildung Der Ablauf eines Seitenaufrufs mit Server Side Rendering

Quelle: Eigene Darstellung

Nachdem nun geklärt ist wie Server Side Rendering funktioniert, wird noch etwas mehr ins Detail gegangen, indem aufgezeigt wird, wie Next.js serverseitiges Rendern umsetzt.

### Server Side Rendering mit Next

Next verwendet im Gegensatz zu React serverseitiges Rendern als Standard. Wenn also nichts näher definiert ist, wird Pre-Rendering durchgeführt. Kommt es im Code zu dem Fall, dass eine API angefragt wird, verwendet man beim serverseitigen Rendern in Next dazu die getServerSideProps Funktion. Diese wird ausschließlich auf der Seite des Servers und nicht im Browser ausgeführt. Die Funktion kann jedoch nur auf Seiten-Ebene in Next und nicht in einzelnen Dateien angewendet werden. Man verwendet die getServerSideProps Funktion, wenn der Server während des Pre-Rendering Prozesses Daten braucht, um das HTML zu generieren. Kommt also der Request des Clients für eine Seite, fragt der Server die Daten an, und liefert diese dann zusammen mit dem generierten Content an den Client aus. Das Holen der Daten passiert also, bevor die Seite geladen hat. Möchte man die API erst anfragen, nachdem die Seite geladen hat, muss man die entsprechende Funktion für das clientseitige Vorgehen verwenden. Um zu entscheiden welche Funktion man wann verwenden sollte, werden nun einige Vor- und Nachteile des Server Side Renderings hinsichtlich ihrer Performance aufgezeigt (Lazuardy & Anggraini, 2022, S. 135).

### Vorteile von Server-Side-Rendering

Mit Server Side Rendering erreicht man meist einen schnellen First Paint und First Contentful Paint. Das kommt daher, dass die Seite schon im Voraus vom Server gerendert wurde. Erst wenn der Content schon auf dem Bildschirm des Users angezeigt wird, kommt der Hydration Prozess zum Einsatz. Man überspringt also die Zeit, die dieser Vorgang verbraucht und zeigt den Inhalt schon direkt an, ohne, dass das JavaScript ausgeführt wurde. Der User bekommt also bei einem Seitenaufruf schnell den gewünschten Content angezeigt. Serverseitiges Rendern eignet sich demnach besonders bei statischen Seiten, die nicht viel interaktive Elemente haben, da der Inhalt schnell da ist und es unwahrscheinlich ist, dass der User direkt nach einem Seitenaufruf eine Interaktion ausführt, welche das vollständige Laden von JavaScript benötigt. Verwendet man zudem noch optimiertes Caching, muss der Rendering Prozess nicht bei jedem Request komplett durchlaufen werden, was eine noch schnellere Ladezeit zur Folge hat (Miller & Osmani, 2019).

Das Pre-Rendering hilft ebenfalls dabei einen schnellen Time To Interactive zu erreichen. Der Server sendet dem Client nämlich nur so viel JavaScript, wie für die jeweilige Seite benötigt wird. Der Rest wird nicht übertragen, wodurch der Hydration Prozess nicht von unnötigem Code verlangsamt wird (2019).

### Nachteile von Server-Side-Rendering

Das Anfragen jeder einzelnen Seite bei dem Server bringt auch ein paar Nachteile mit sich. Zum einen hat der Server eine hohe Belastung, wenn dieser den Rendering Prozess übernehmen muss. Er muss folglich leistungsfähig und immer erreichbar sein, damit der User die Website in annehmbarer Zeit ausgespielt bekommt (Lazuardy & Anggraini, 2022, S. 141).

Hinzu kommt, dass eine Seite, welche mehrere dynamische Elemente hat und demnach mehr JavaScript laden muss, Gefahr aufläuft, dass eine sehr lange Zeit zwischen dem Paint der Seite und dem Time To Interaktive vergeht. Dadurch kann es passieren, dass der User die komplette Seite sieht und bereits mit ihr interagieren will, aber diese nicht reagiert, da das JavaScript noch nicht vollständig geladen hat (2022, S. 141).

Weiterführend kann Pre-Rendering einen schlechten Time To First Byte bewirken, da der Browser erst eine Response erlangt, wenn der Server den Render-Prozess abgeschlossen hat. Dies kann je nach Umfang der Anwendung einige Zeit dauern, in welcher der Browser keinerlei Daten erlangt (Miller & Osmani, 2019).

Trotz dieser Nachteile muss eine serverseitig gerenderte Anwendung keine schlechte Performance haben. Man kann nämlich durch optimiertes Caching den Risiken gut entgegenwirken und sowohl eine schnellere Ladezeit erreichen als auch den Server damit entlasten.

## Client-Side-Rendering

Nachdem erläutert wurde, wie serverseitiges Rendern funktioniert, werden sich die nächsten Abschnitte mit dem Rendern auf dem Client beschäftigen. Client-Side-Rendering ist im Vergleich zum Server-Side-Rendering ein eher neueres Verfahren. Durch die immer größer werdende Beliebtheit von JavaScript Frameworks und Libraries, wird auch das bei diesen verwendete clientseitige Rendern immer populärer (Lazuardy & Anggraini, 2022, S. 134). Wie es der Name erahnen lässt, wird bei diesem Verfahren das Rendern vom Client ausgeführt.

Fragt der Browser den Inhalt einer Seite per HTTP Request beim Server an, sendet dieser vorerst nur ein sehr minimiertes HTML Dokument als Grundgerüst. Darin befindet sich neben dem HTML Head und einem einzigen div Element mit der id root zwei script Tags, bei welchen eines das Framework selbst referenziert und das andere die Root-Komponente app lädt. Der Client fragt nun die verlinkten JavaScript Dateien beim Server an und führt diese nach der Übertragung aus. Mithilfe der Skripte generiert der Browser den HTML Code und fügt ihn als Kind Elemente innerhalb des div Elements mit der id root ein. An erforderlichen Stellen fragt der Client währenddessen APIs nach Daten an und benutzt sie, um schließlich die Seite zu rendern. Das bedeutet, dass die Anwendungslogik zusammen mit dem Routing und dem Data-Fetching auf der Seite des Clients ausgeführt wird und nicht auf der des Servers (Brehm, 2013).

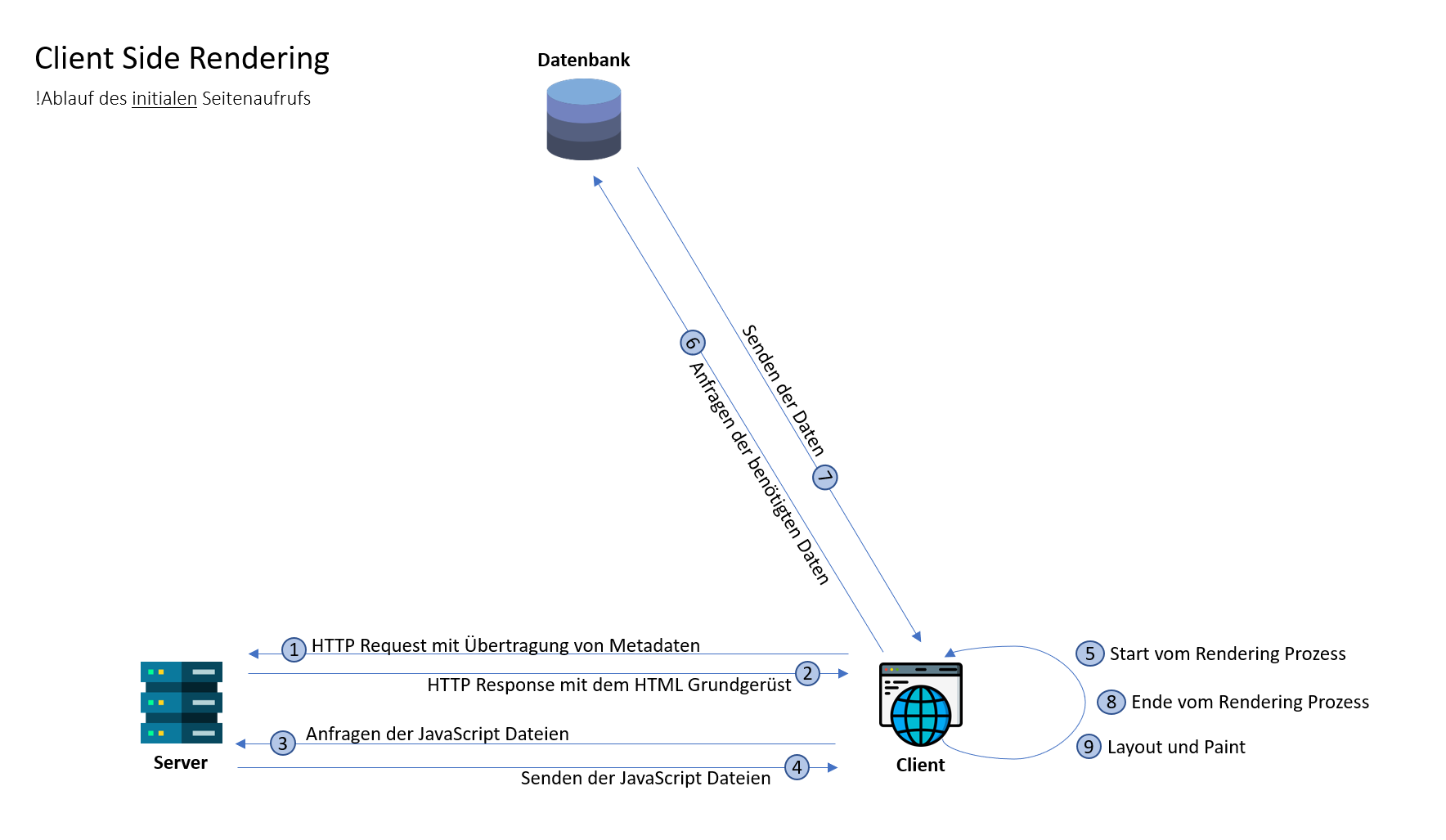


Abbildung Ablauf des initialen Seitenaufrufs mit Client Side Rendering

Quelle: Eigene Darstellung

Die Besonderheit des clientseitigen Renderns zeigt sich, wenn der User nach dem initialen Render mit der Webpage interagiert und die DOM Struktur sich ändert. Denn dann wird der neue Content nicht erneut beim Server angefragt, da er sich bereits in der Library, welche der Browser beim initialen Rendern angefragt hatte, befindet. Die Anwendung erkennt selbst, welche Inhalte der Seite sich durch die Interaktion des Users ändern werden und re-rendert die entsprechenden Komponenten, welche dadurch beeinflusst werden. Das bedeutet, dass die Seite nicht neu geladen wird. Der User befindet sich immer noch auf der initialen HTML Seite. Mit dem Unterschied, dass der Browser mithilfe des JavaScripts einzelne Komponenten neu gerendert hat (Eriksson, 2015, S. 11).

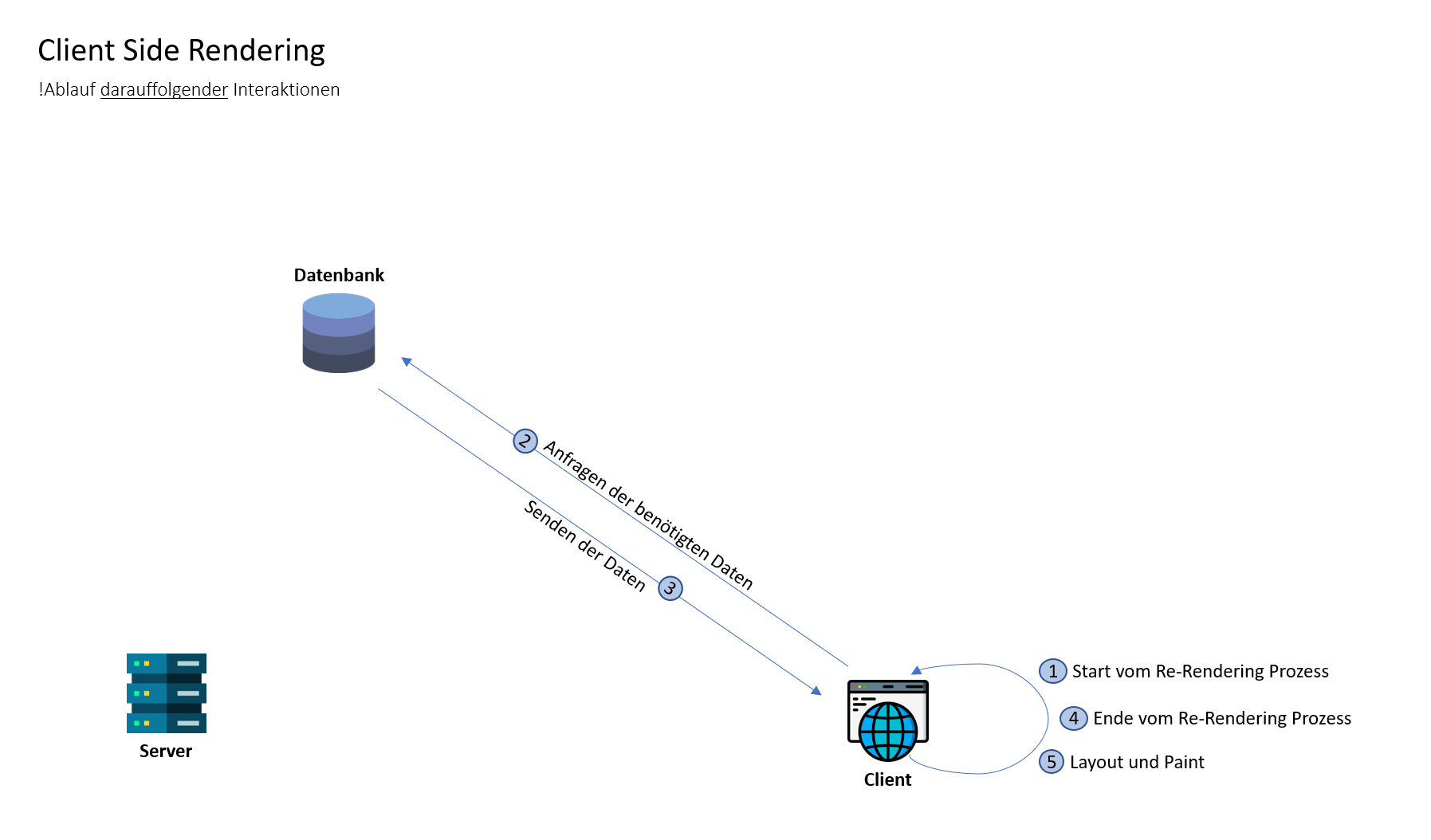


Abbildung Ablauf darauffolgender Userinteraktionen mit Client Side Rendering

Quelle: Eigene Darstellung

Es gibt also nur eine Seite, deren Komponenten sich dynamisch ändern können. Deshalb spricht man beim clientseitigen Rendern von einer Single Page Application (SPA). Diese ermöglicht, dass die Anwendung auch offline genutzt werden kann, da der Client das ganze HTML bei sich gespeichert hat und die Logik ausführt. Das funktioniert jedoch nur solange der Browser keine Anfrage an den Server senden muss. Denn dafür braucht es bekanntlich eine Verbindung zu dem Internet (2015, S. 11).

### Client-Side-Rendering mit Next

Next verwendet für das Anfragen von Daten auf der Client Seite den useEffect Hook von React. Man kann diesen sowohl auf der Page-Ebene wie auch in einzelnen Komponenten verwenden. Wenn man den Hook in eine komplette Seite einbaut, wird das Data-Fetching zur Laufzeit ausgeführt. Wenn sich dabei die Daten ändern, wird der Content der Seite neu gerendert. Führt man jedoch das Anfragen der Daten in einer Komponente selbst aus, wird das Fetching ausgeführt, wenn diese gemountet wird. Das bedeutet, dass die API angefragt wird, wenn die Komponente gerade installiert wird. Geht man also auf diese Weise vor, wird nicht die komplette Webpage, sondern nur die Komponente geupdatet (Lazuardy & Anggraini, 2022, S. 135). Das bringt einige Vorteile mit sich, welche im folgendem Abschnitt aufgezeigt werden.

### Vorteile von Client-Side-Rendering

Dadurch, dass nach dem initialen Render nur wenige Teile der Anwendung und nicht die komplette Seite neu geladen wird, benötigt jeder folgende Request des Clients nur ein Bruchteil der Zeit. Der Browser muss nämlich nur noch etwaige Daten beim Server durch die API anfragen und kann dann das DOM an den benötigten Stellen updaten. Das führt dazu, dass Interaktionen des Users sehr schnell über das vorhandene JavaScript umgesetzt werden. Die Seite reagiert demnach sofort, wodurch die User Experience stark verbessert wird. Das macht clientseitiges Rendern äußert attraktiv für dynamische Anwendungen, welche sehr viel Rich Content haben und oft geupdatet werden (Brehm, 2013).

Nach Miller und Osmani bietet das clientseitige Rendern noch eine weitere Möglichkeit die Performance immens zu verbessern, indem Komponenten, die auf vielen Seiten verwendet werden, zudem gecashet werden. Besucht der User mehrmals eine Seite mit denselben Elementen, werden diese aus dem Cache geladen, wodurch der Content schneller angezeigt wird (Miller & Osmani, 2019).

### Nachteile von Client-Side-Rendering

Wie auch das serverseitige Rendern hat die Strategie einer SPA mit Client Side Rendering ihre Nachteile. Das größte Risiko einer schlechten Performance besteht beim initialen Laden der Seite. Denn dadurch, dass der Client jede Ressource einzeln anfragen muss, kommt es vor allem bei umfänglicheren Anwendungen zu langen Ladezeiten beim Seitenaufruf (Konshin, 2018, S. 13). Hinzu kommt, dass je größer die Anwendung wird, mit ihr auch die Anzahl an JavaScript steigt. Es werden extra Bibliotheken oder anderer Third-Party-Code verwendet, welche oftmals noch vor dem Render des Contents gebraucht werden (Miller & Osmani, 2019). Das kann dazu führen, dass der User beim ersten Aufruf einer Seite für eine gewisse Zeit nur eine leere Seite oder einen Ladebildschirm sieht, bis das JavaScript geladen hat und der Content gerendert wird (Brehm, 2013). Das wirkt sich schlecht auf die User Experience aus und zeigt sich auch in der Performance Analyse mittels eines zu langen First Paint und First Contentful Paint. Ein noch viel gravierenderes Risiko besteht darin, wenn der User JavaScript in seinem Browser deaktiviert hat. Dann würde folglich nur eine leere Seite angezeigt werden, da der Content nicht gerendert werden kann (Eriksson, 2015, S. 11)

Darüber hinaus kann es vor allem bei mobilen Endgeräten oder bei schlechter Internetverbindung dazu kommen, dass die Seite sehr lange braucht um interaktiv zu werden, da das Laden des JavaScripts einige Zeit in Anspruch nimmt (Pavic & Brkic, 2021, S. 1754). Die Time To Interactive kann daher schnell in einen kritischen Bereich kommen. Dem kann man jedoch gut mit Lazy-Loading JavaScript entgegenwirken. Dazu führt der Client zuerst die Skripte aus, welche für den Render benötigt werden. Der restliche Code wird dann on-demand im Hintergrund geladen (Miller & Osmani, 2019).

## Isomorphic Web Applications

Nachdem nun aufgezeigt wurde welche Vor- und Nachteile clientseitiges und serverseitiges Rendern haben, stellt sich folglich die Frage, ob nicht beide Strategien miteinander kombiniert werden können, um von den Vorzügen beider Verfahren zu profitieren. Im Folgenden wird aufgezeigt, wie clientseitiges und serverseitiges Rendern zusammenspielen können.

Beim sogenannten Isomorphic Rendering wird der initiale Render vom Server ausgeführt, woraufhin alle folgenden Requests vom Client bearbeitet werden. Dadurch möchte man die eher langsame Ladezeit der ersten Seite beim Client Side Rendering mittels Pre-Rendering kompensieren und kann dennoch die Vorteile des clientseitigen Renderns nutzen (Arkwrite, 2017). Man setzt dabei darauf, dass sich der Rendering-Server und der API-Server sehr nah beieinander befinden, wodurch eine stabile Verbindung aufgebaut und eine große Bandbreite genutzt werden kann. Wenn sich beide Server beispielsweise im selbem Datenzentrum befinden, besteht eine deutlich schnellere und stabilere Verbindung, als zwischen dem API-Server und einem Client, welcher sich in weiterer Distanz befindet (Konshin, 2018, S. 14). Das Wort „Isomorphic“ beschreibt bei diesem Verfahren die Fähigkeit der Anwendung, sowohl auf dem Server als auch auf dem Client zu laufen (Brehm, 2013). Das bedeutet, dass der Code serverseitig wie clientseitig derselbe ist. Demnach muss die Anwendung in JavaScript geschrieben sein, da sie von beiden Parteien ausführbar sein muss. Dafür bietet sich die Verwendung von Node.js als Laufzeitumgebung an (Eriksson, 2015, S. 13).

Während der build-time von isomorphic Web Applications, wird der Code kompiliert und komprimiert, sodass zwei Pakete entstehen, wobei eines für den Client und das andere für den Server optimiert ist. Dabei ist es nötig, dass der Server eine Repräsentation des DOMs zur Verfügung hat, damit er den Content auch in einem Szenario ausliefern kann, wo der User JavaScript deaktiviert hat. Damit läuft man nicht mehr Gefahr, dass eine leere Seite angezeigt wird, wenn nur clientseitiges Rendern verwendet wird (2015, S. 14).

Der Rendering Prozess benötigt dafür einen gewissen Aufwand. Denn der Server muss vorerst bei einem Request den aktuellen Zustand der Anwendung auf dem Client ermitteln, woraufhin er das passende HTML Markup rendern und zusammen mit dem aktuellen Zustand der Komponenten an den Client übertragen kann. Der Browser paintet dann die Seite und bootet React in den aktuellen DOM Baum. Dabei verwendet er den mitgegeben Zustand durch den Server und rendert die komplette Anwendung erneuet. Dieser erneute Render wirkt sich jedoch nicht auf das Layout im Browser aus, da das vom Client gerenderte Markup dasselbe ist, welches der Server mitgegeben hat (2015, S. 14).

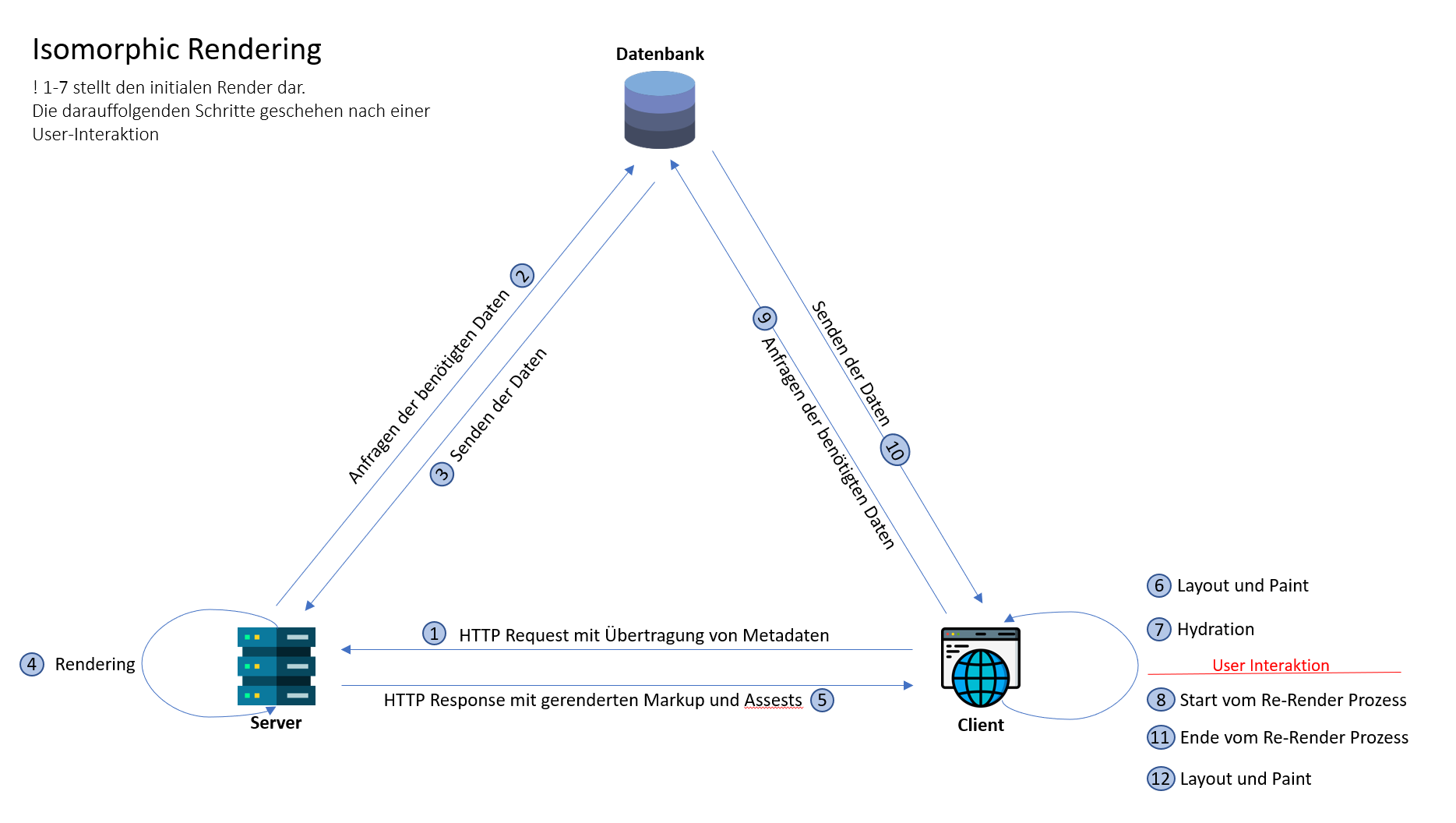


Abbildung Ablauf von Requests mit isomorphic Rendering

Quelle: Eigene Darstellung

### Chancen von isomorphic Web-Applications

Mit einer isomorphic Strategie lässt sich die hohe Auslastung des Servers beim serverseitigen Rendern durch einen clientseitigen rendering-fallback kompensieren. Dabei übernimmt der Client das Rendering, sobald der Server eine gewisse Auslastung erreicht hat. Um dies zu erreichen, programmiert man einen Zähler, welcher bei jedem neuen Request erhöht wird und bei jedem Vollenden eines Requests erniedrigt wird. Sobald der Wert über eine Gewisse grenze steigt, wird das Rendering vom Browser übernommen, bis der Server wieder Kapazitäten hat. Dadurch kann eine zu hohe Belastung des Servers vermieden werden (Arkwrite, 2017).

Einige bekannte Unternehmen haben isomorphic-rendering bereits in ihren Produkten verwendet. Netflix rendert beispielsweise deren eher statische Landingpage mittels des Servers, während im Hintergrund bereits JavaScript geladen wird, welches die Anwendung für die folgenden interaktiven Seiten verwendet. Dadurch können die dynamischen Seiten, welche viel Ressourcen brauchen, schneller geladen werden (Miller & Osmani, 2019).

### Risiken von isomorphic Web-Applications

Doch wie auch bei den bereits behandelten Strategien, kann das isomorphic Rendering negative Seiten mit sich bringen. Es kann nämlich auch bei diesem Verfahren passieren, dass eine Seite bereits vollständig angezeigt wird, aber noch nicht interaktiv ist. Denn sobald beim initialen Seitenaufruf der Content vom Server gerendert und an den Browser übergeben wurde, wird der Inhalt angezeigt. Es muss dann jedoch noch der Hydration Prozess durchgeführt werden, bei welchem das JavaScript ausgeführt wird. Dadurch kann bei schlechter Internetverbindung ein großes Zeitfenster entstehen, wo der Benutzer verärgert wird, wenn er eine Aktion ausführt und die Anwendung nicht reagiert (2019).

Ein weiterer Gefahrenpunk besteht darin den Zustand der Anwendung auf dem Server und dem Client synchron zu halten. Der Client muss wissen, welche Daten der Server verwendet hat, um das Markup zu rendern, damit er darauf aufbauen kann. Dabei muss vermieden werden, dass der Browser diese Informationen extra beim Server anfragen muss. Deshalb befinden sich in dem HTML der Response script tags, in welchen die Datenabhängigkeiten mitgegeben werden. Dieses Vorgehen beinhaltet jedoch sehr viel doppelten Code, was die Anwendung fehleranfällig macht (2019).

## Static Side Generation

Static Side Generation ist eine weitere Strategie, welche man verwenden kann, um Websites dem User bereitzustellen. Das Besondere hierbei ist, dass das HTML bereits zur build-time gerendert wird. Der Inhalt steht dann auf einem Server oder einem Content Delivery Network schon aufbereitet zur Verfügung. Folglich muss die Webpage nicht wie beim SSR on-demand vom Server gerendert werden, da sich die statischen Seiten bereits auf dem Server befinden. Es gibt demnach für jede Webpage eine eigene URL und ein eigenes HTML Dokument, welches zusammen mit den benötigten Assets an den Browser ausgeliefert wird. Das bedeutet aber auch, dass die Webpage nur zu einem geringen Maße personalisiert werden kann, da der Content unabhängig von dem jeweiligen Client gleich ist (Miller & Osmani, 2019).

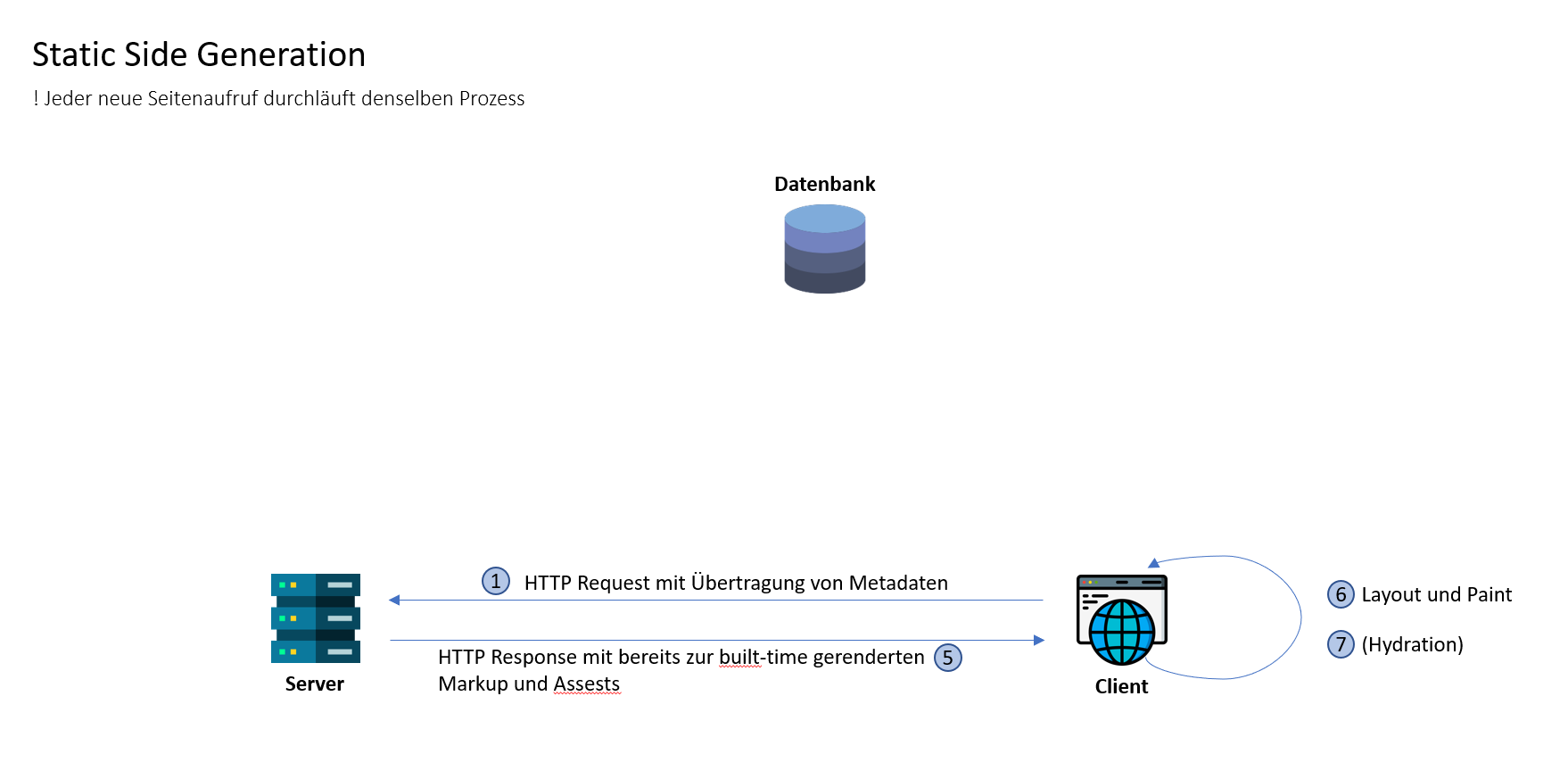


Abbildung Ablauf von Seitenaufrufen mit Static Side Generation

Quelle: Eigene Darstellung

### Static Side Generation mit Next

Next bietet sowohl Pre-Rendering mittels Server Side Rendering als auch mit Static Side Generation an. Möchte man statische Seiten generieren lassen, muss die getStaticProps Funktion verwendet werden, welche ausschließlich von Server ausgeführt werden kann. Wie auch beim serverseitigen Rendern muss das Fetchen der Daten auf Seiten-Ebene geschehen. Kommt es schließlich zum Build Prozess, wird die getStaticProps Funktion aufgerufen und die Daten werden geholt. Es bietet sich demnach an das Data-Fetching mit Static Side Generation zu wählen, wenn sich die Daten nicht oft ändern und die Seite eher statisch ist (Lazuardy & Anggraini, 2022, S. 135).

### Vorteile von Static Side Generation

Hinsichtlich der Performance bringt die Verwendung von statischem Rendering einige Vorteile mit sich. Der Server überspringt das Rendern von HTML wie auch das Fetchen der Daten und liefert die Ressourcen direkt an den Client aus. Dadurch erreicht man einen äußerst schnellen Time To First Byte, First Paint und First Contentful Paint. Hinzu kommt, dass bei statischem Content meist sehr wenig JavaScript benötigt wird, was folglich zu einem schnellem Time To Interactive führt. Ein positiver Nebeneffekt davon ist, dass die statischen Seiten meist auch durchgehend funktionsfähig sind, wenn der User JavaScript im Browser deaktiviert hat (Miller & Osmani, 2019).

Die Performance kann zudem stark verbessert werden, wenn die statischen Seiten über ein Content Delivery Network (CDN) bereitgestellt werden. Der Content wird dem Client dann von demjenigen Server übertragen, welcher sich geographisch am nächsten zu diesem befindet. Wird dies mit den passenden Caching Methoden verbunden, erreicht man eine äußerst schnelle Übertragungszeit (2019).

### Nachteile von Static Side Generation

Obwohl SSG mit sehr guter Performance eine attraktive Lösung zu sein scheint, ist das Verfahren nicht immer die beste Lösung. Der bedeutendste Grund dafür ist, dass man sich auf statische Seiten beschränken muss. Dies mag für Blogs oder weniger dynamische Websites kein Problem sein, doch sobald mehr Interaktion gebraucht wird, ist eine Umsetzung mit statischem Rendern mit hohem Aufwand und Komplexität verbunden. Es müssen nämlich für jede URL ein eigenes HTML Dokument erstellt werden. Kann man zur built-time nicht vorhersagen, wie diese URLs aussehen werden oder wie der Inhalt aussehen wird, ist statisches Rendern nicht geeignet (Miller & Osmani, 2019).

# Abbildungsverzeichnis

[Abbildung 1 Korrelation von Ladezeit und Absprungrate. 9](#_Toc107749668)

[Abbildung 2 Wartezeiten in Korrelation zum Empfinden des Users 10](#_Toc107749669)

[Abbildung 3 Der Ablauf eines Seitenaufrufs mit Server Side Rendering 12](#_Toc107749670)

[Abbildung 4 Ablauf des initialen Seitenaufrufs mit Client Side Rendering 15](#_Toc107749671)

[Abbildung 5 Ablauf darauffolgender Userinteraktionen mit Client Side Rendering 16](#_Toc107749672)

[Abbildung 6 Ablauf von Requests mit isomorphic Rendering 19](#_Toc107749673)

[Abbildung 7 Ablauf von Seitenaufrufen mit Static Side Generation 20](#_Toc107749674)

Literaturverzeichnis

An, D. (2018, Februar). *Find out how you stack up to new industry benchmarks for mobile page speed*. Zugriff am 02.07.2022. Verfügbar unter: https://www.thinkwithgoogle.com/marketing-strategies/app-and-mobile/mobile-page-speed-new-industry-benchmarks/

Arkwrite. (2017). *Scaling React Server-Side Rendering*. Zugriff am 21.05.2022. Verfügbar unter: https://arkwright.github.io/scaling-react-server-side-rendering.html

Beke, M. (2018). On the Comparison of Software Quality Attributes for Client-side and Server-side Rendering. Zugriff am 11.04.2022. Verfügbar unter: https://denbeke.be/thesis/versions/mathias-beke-final.pdf

Brehm, S. (The Airbnb Tech Blog, Hrsg.). (2013). *Isomorphic JavaScript: The Future of Web Apps,* Airbnb. Zugriff am 14.05.2022. Verfügbar unter: https://medium.com/airbnb-engineering/isomorphic-javascript-the-future-of-web-apps-10882b7a2ebc#.4nyzv6jea

Egger, S., Hossfeld, T., Schatz, R. & Fiedler, M. (2012, 5.–7. Juli). Waiting times in quality of experience for web based services. In *2012 Fourth International Workshop on Quality of Multimedia Experience* (S. 86–96). IEEE. Zugriff am 08.05.2022. Verfügbar unter: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6263888

Eriksson, M. (2015). Isomorphic web applications. Depends on how you react. Zugriff am 14.05.2022. Verfügbar unter: https://www.diva-portal.org/smash/get/diva2:825784/FULLTEXT01.pdf

Fadhilah Iskandar, T., Lubis, M., Fabrianti Kusumasari, T. & Ridho Lubis, A. (2020). Comparison between client-side and server-side rendering in the web development. *IOP Conference Series: Materials Science and Engineering*, (1). https://doi.org/10.1088/1757-899X/801/1/012136

Kishore, P. & B M, M. (Hbrp publications, Hrsg.). (2020). *Evolution of Client-Side Rendering overServer-Side Rendering*. Zugriff am 19.05.2022. Verfügbar unter: http://hbrppublication.com/OJS/index.php/RTITIA/article/view/1363

Konshin, K. (2018). *Next.js Quick Start Guide. Server-side rendering done right* (1st ed.). Birmingham: Packt Publishing Limited.

Lazuardy, M. F. S. & Anggraini, D. (2022). Modern Front End Web Architectures with React.Js and Next.Js. Zugriff am 23.04.2022. Verfügbar unter: http://scholar.googleusercontent.com/scholar?q=cache:pnVViv0guYkJ:scholar.google.com/+incremental+regeneration+next+js&hl=de&as\_sdt=0,5

Lindsay, S. (2022). *Minimizing browser reflow*. Verfügbar unter: https://developers.google.com/speed/docs/insights/browser-reflow?hl=en

MDN contributers. (2022). *How the web works,* Mozilla Developer Network. Zugriff am 14.05.2000. Verfügbar unter: https://developer.mozilla.org/en-US/docs/Learn/Getting\_started\_with\_the\_web/How\_the\_Web\_works

Miller, J. & Osmani, A. (2019). *Rendering on the Web,* web.dev. Zugriff am 14.05.2022. Verfügbar unter: https://web.dev/rendering-on-the-web/

Oluwatosin, H. S. (2014). Client-Server Model. *IOSR Journal of Computer Engineering*, (1), 57–71. https://doi.org/10.9790/0661-16195771

Pavic, F. & Brkic, L. (2021, 27. September – 1. Oktober). Methods of Improving and Optimizing React Web-applications. In *2021 44th International Convention on Information, Communication and Electronic Technology (MIPRO).* IEEE. Zugriff am 13.05.2022. Verfügbar unter: https://ieeexplore.ieee.org/document/9596762/

Psaila, G. (IEEE, Hrsg.). (2008). *Virtual DOM: an Efficient Virtual Memory Representation for Large XML Documents*. Zugriff am 02.07.2022. Verfügbar unter: https://www.diva-portal.org/smash/get/diva2:1107724/FULLTEXT01.pdf

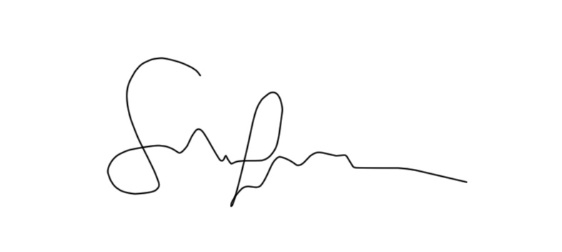
Saurabh Daware. (2020). *HTML parsing and rendering: Here's what happens when you type URL and press enter…* Zugriff am 17.05.2022. Verfügbar unter: https://dev.to/saurabhdaware/html-parsing-and-rendering-here-s-what-happens-when-you-type-url-and-press-enter-3b2o

Vega, C. (2017). *Client-side vs. server-side rendering: why it’s not all black and white,* freeCodeCamp. Zugriff am 14.05.2022. Verfügbar unter: https://www.freecodecamp.org/news/what-exactly-is-client-side-rendering-and-hows-it-different-from-server-side-rendering-bd5c786b340d/

Webb, D. (2012). *Improving performance on twitter.com*. Zugriff am 14.05.2022. Verfügbar unter: https://blog.twitter.com/engineering/en\_us/a/2012/improving-performance-on-twittercom

# Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorgelegte Bachelorarbeit selbstständig verfasst und noch nicht anderweitig zu Prüfungszwecken vorgelegt habe. Alle benutzten Quellen und Hilfsmittel sind angegeben, wörtliche und sinngemäße Zitate wurden als solche gekennzeichnet.



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

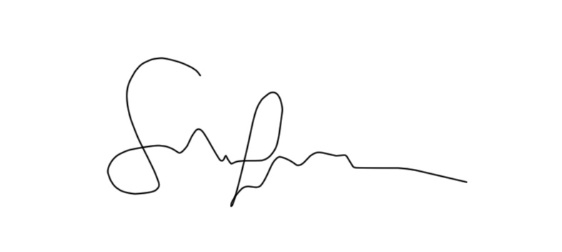
Kai Schäfer, am 14. November 2019

# 

# Zustimmung zur Plagiatsüberprüfung

Hiermit willige ich ein, dass zum Zwecke der Überprüfung auf Plagiate meine vorgelegte Arbeit in digitaler Form an PlagScan (www.plagscan.com) übermittelt und diese vorübergehend (max. 5 Jahre) in der von PlagScan geführten Datenbank gespeichert wird sowie persönliche Daten, die Teil dieser Arbeit sind, dort hinterlegt werden.

Die Einwilligung ist freiwillig. Ohne diese Einwilligung kann unter Entfernung aller persönlichen Angaben und Wahrung der urheberrechtlichen Vorgaben die Plagiatsüberprüfung nicht verhindert werden. Die Einwilligung zur Speicherung und Verwendung der persönlichen Daten kann jederzeit durch Erklärung gegenüber der Fakultät widerrufen werden.



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Kai Schäfer, am 14. November 2019