Technische Hochschule
Augsburg Technical University of
Applied Sciences

Bachelorthesis

Fakultät für
Informatik

Studienrichtung
Wirtschaftsinformatik

Michael Mertl

Ist die Trennung zwischen Frontend und Backend in webbasierten Systemen noch zeitgemäß?

Erstprüfer: Prof. Dr. Anja Metzner
Zweitprüfer: Prof. Dr.-Ing. Christian Märtin
Abgabe der Arbeit am: 02.02.2024

Technische Hochschule Augsburg
Augsburg Technical University
of Applied Sciences

An der Hochschule 1
D-86161 Augsburg

Telefon +49 821 55 86-0
Fax +49 821 55 86-3222
www.hs-augsburg.de
info@hs-augsburg.de

Fakultät für Informatik
Telefon +49 821 5586-3450
Fax +49 821 5586-3499

Verfasser der Bachelorthesis:
Michael Mertl
Obere Ortsstraße 31
86576 Schiltberg
michael.mertl@hs-augsburg.de

Kurzfassung

Zu Beginn eines neuen Webprojekts ist es gängige Praxis, eine Architektur auszuwählen, die als Grundlage für die Entwicklung der jeweiligen Problemstellung dient. Diese Bachelorarbeit zielte darauf ab, diese Entscheidungsfindung zu unterstützen, indem Kriterien gesammelt wurden, die für oder gegen die Multi-Page-Architektur, Single-Page-Architektur und die hier als Fullstack-Architektur bezeichnete Architektur sprechen. Die Fullstack-Architektur wird durch *Blazor Server* repräsentiert, ein *Framework*, das die Entwicklung von *Frontend* und *Backend* ohne Trennung ermöglicht. Das Ziel bestand darin zu untersuchen, unter welchen Bedingungen ein solcher Ansatz geeignet ist.

Die Arbeit umfasste eine umfangreiche Literaturrecherche, praxisorientierte Experteninterviews bei der beauftragenden Firma XITASO und durchgeführte Experimente. Die gewonnenen Ergebnisse wurden genutzt, um einen Kriterienkatalog zu erstellen, der zukünftig bei architektonischen Problemstellungen im Bereich der Webarchitekturen als Leitfaden dienen kann, um eine passende Architektur abzuleiten und darüber zu diskutieren. Die Erkenntnisse dieser Arbeit tragen dazu bei, die Entscheidungsfindung im Bereich der Webarchitektur zu erleichtern und bieten einen Beitrag zur aktuellen Diskussion über effektive Entwicklungsansätze.

Schlagwörter: Webanwendungsarchitekturen, Fullstack-Webentwicklung, Single-Page-Architektur, Multi-Page-Architektur, Fullstack-Architektur, *Blazor Server*

Abstract

When starting a new web project, it is common to choose a software architecture that serves as the basis for solving the problem at hand. This thesis aimed to support this decision-making process by gathering criteria that speak in favour of or against the multi-page architecture, single-page architecture and the architecture referred to here as fullstack. The fullstack architecture is represented by *Blazor Server*, a *framework* that enables the development of *frontend* and *backend* in one software component without separation. The goal was to analyse the conditions under which it makes sense to choose this software architecture.

The thesis contained an extensive literature review, practice-orientated expert interviews and realised software experiments. The work was created in cooperation with the client of the work XITASO. The results obtained were used to create a catalogue of criteria that can serve as a guide for future architectural problems in the area of web architectures. The outcome of this work contributes to the decision-making in the field of web architectures and provides a contribution to the ongoing discussion on effective development approaches.

Keywords: Web-application-architectures, fullstack web-development, single-page-architecture, multi-page-architecture, fullstack-architecture, *Blazor Server*

Erklärung zur Abschlussarbeit

Hiermit versichere ich, die eingereichte Abschlussarbeit selbständig verfasst und keine andere als die von mir angegebenen Quellen und Hilfsmittel benutzt zu haben. Wörtlich oder inhaltlich verwendete Quellen wurden entsprechend den anerkannten Regeln wissenschaftlichen Arbeitens zitiert. Ich erkläre weiterhin, dass die vorliegende Arbeit noch nicht anderweitig als Abschlussarbeit eingereicht wurde.

Das Merkblatt zum Täuschungsverbot im Prüfungsverfahren der Hochschule Augsburg habe ich gelesen und zur Kenntnis genommen. Ich versichere, dass die von mir abgegebene Arbeit keinerlei Plagiate, Texte oder Bilder umfasst, die durch von mir beauftragte Dritte erstellt wurden.

Augsburg, 02.02.2024
Ort, Datum

Unterschrift des/der Studierenden

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
ASP	Active Server Page
CSS	Cascading Style Sheets
CSV	Comma Separated Values
DOM	Document Object Model
FCP	First Contentful Paint
HTML	Hypertext Markup Language
INP	Interaction to next Paint
JS	JavaScript
JSON	JavaScript Object Notation
MPA	Multi-Page-Architektur
MVC	Model View Controller
NET	Network Enabled Technologies
OAuth 2.0	Open Authorization 2.0
SEO	Suchmaschinenoptimierung
SPA	Single-Page-Architektur
XML	Extensible Markup Language

Vorwort

Die vorliegende Arbeit entstand im Rahmen einer externen Bachelorarbeit im Bereich Fullstack-Webentwicklung an der Technischen Hochschule Augsburg. Die Zusammenarbeit mit der XITASO GmbH war nicht nur motivierend, sondern auch entscheidend für die Ausrichtung und Tiefe dieser Bachelorarbeit. Mein aufrichtiger Dank gilt meinen Betreuern, Jan-Philipp Steghöfer und Alexander Rampp sowie den Experten von XITASO, die durch ihre Teilnahme an den Experteninterviews einen bedeutenden Beitrag zur Qualität dieser Arbeit geleistet haben.
Besondere Anerkennung gebührt Prof. Dr. Anja Metzner, meiner betreuenden Professorin an der Technischen Hochschule Augsburg, für ihre fachkundige Anleitung und Unterstützung während des gesamten Forschungsprozesses.
Um die Lesbarkeit zu fördern, werden alle Fachbegriffe, die in dieser Arbeit kursiv geschrieben sind, im Glossar erläutert.
Es ist wichtig zu beachten, dass die Betrachtung des *Frameworks Blazor Server* von Microsoft auf der siebten Version der *.NET-Umgebung* basierte. Die achte Version, die während der Bearbeitung veröffentlicht wurde, wird im Kapitel 8.3 („Ausblick“) berücksichtigt.
Die Bearbeitung dieser Arbeit war eine informative und lehrreiche Reise und ich hoffe, dass sie nicht nur einen Beitrag zum bestehenden Wissensstand leistet, sondern auch als Ausgangspunkt für zukünftige Arbeiten dient.

Ich wünsche Ihnen viel Freude beim Lesen dieser Bachelorarbeit.

Michael Mertl

Kapitel 1

Einleitung

Das einleitende Kapitel bietet einen Überblick über die Entwicklung von Webanwendungsarchitekturen und zeigt dabei auf, dass neue Architekturansätze die Grenzen zwischen dem *Frontend* und *Backend* verschwimmen lassen. Dabei wird für diese Arbeit der Inhalt und das Vorgehen genauer definiert. Außerdem bietet das aktuelle Kapitel einen Überblick über das Ergebnis der Arbeit. Es handelt sich dabei um einen Kriterienkatalog der, der Arbeit beiliegt (siehe Anhang F: „Finale Version des Kriterienkatalogs nach den Experimenten“). Des Weiteren zeigt die Einleitung, für welche Zielgruppe diese Arbeit geeignet ist.

1.1 Ausgangslage und Unternehmen

Über 30 Jahre existiert das Web bereits und die Forschung an verschiedenen Webanwendungsarchitekturen. Dabei wurden viele neue Technologien entwickelt, um neue Architekturen zu ermöglichen. Doch wie kann die Entwicklung von modernen Webanwendungen beschleunigt werden?
Die Architektur von Webanwendungen hat sich seit ihren Anfängen in den 1990er Jahren kontinuierlich weiterentwickelt. Ursprünglich bestanden Webseiten aus statischen Dokumenten. Diese wurden, wie in Abbildung 1 gezeigt, von dem Client (Nr. 1 in Abbildung 1) angefragt (Nr. 2 in Abbildung 1) und vom Server (Nr.

3 in Abbildung 1) zurückgegeben (Nr. 4 in Abbildung 1). Der Server hat lediglich diese Anfragen zur Bereitstellung und Übertragung der Dokumente bearbeitet (vgl. Kulesza u. a. 2020: 3). Diese Architektur wird als Multi-Page-Architektur (Abk.: MPA) bezeichnet, bei der, der Server bei jedem Seitenwechsel eine neue HTML-Seite bereitstellt (vgl. Schmitt 2022) (siehe auch Kapitel 2.2: „Die Multi-Page-Architektur im Überblick“).

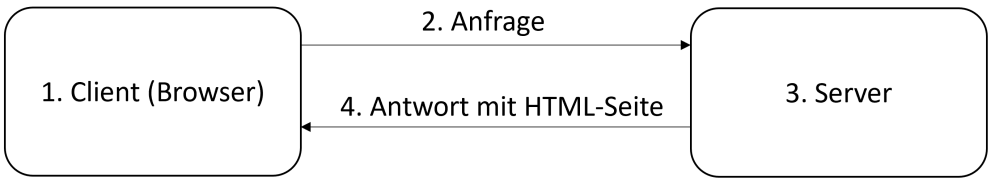


Abbildung 1: Vereinfachte Client-Server Architektur der MPA (eigene Darstellung)

Im Laufe der Zeit haben sich Webanwendungen erheblich weiterentwickelt und sind zunehmend komplexer geworden. Moderne Webanwendungen setzen sich aus zahlreichen Komponenten zusammen, darunter Datenbanken, *(Cloud-)Services* und sogenannte *Content Delivery Networks*. Außerdem sind Webseiten in der Lage, clientseitig Code auszuführen, was zu einer Vielzahl von Technologien wie *Frameworks, APIs* und Bibliotheken für neue Architekturen geführt hat (vgl. Kulesza u. a. 2020: 3f).

Eine aktuell weitverbreitete Architektur für moderne Webanwendungen ist die sogenannte Single-Page-Architektur (Abk.: SPA), wie sie von Sun (Sun 2019: 141f) beschrieben wurde. Bei dieser Architektur erfolgt die Verwaltung des Anwendungsstatus und der Logik hauptsächlich am Client. SPAs rufen bei Bedarf dynamische Daten über Anfragen an eine *Web-API* in Datenübertragungsformaten, wie dem *JSON-Format*, vom *Backend* ab. Dadurch kann die Webanwendung Seiten ohne vollständiges neu Laden aktualisieren und ermöglicht ein nahtloses Benutzererlebnis (siehe Kapitel 2.3: „Erklärung der Single-Page-Architektur“).

Viele Softwarehersteller beschäftigen sich zu Beginn eines Webanwendungsprojektes intensiv mit der Architektur. Eine Statistik (vgl. Stack Overflow 2023) aus dem Jahr 2023 zeigt die am häufigsten von Entwicklern verwendeten Web-*Frameworks*. React, Angular und Vue.js sind *Frontend-Frameworks* zur Entwicklung von Single-Page-Architekturen. Diese sind in der Statistik unter den Top 10 zu finden, neben den möglichen zugehörigen *Backend-Frameworks* wie *ASP.NET Core 7.0*. Es lässt sich also ableiten, dass aus aktuell genannter Statistik, Single-Page-Architekturen gegenüber anderen Webanwendungsarchitekturen bevorzugt werden. Trotz der hohen Flexibilität, die die Verwendung der SPA bietet, geht diese auch mit erheblicher Komplexität einher. Bei der Entwicklung ergeben sich diesbezüglich verschiedene Herausforderungen (vgl. Shilpi 2019; GetDevDone Team 2021):

1. Eine erste Herausforderung ergibt sich aus dem Einsatz unterschiedlicher Programmiersprachen und *Framework*s im *Frontend* und *Backend*, was Fachkenntnisse in beiden Bereichen erfordert.

2. Eine zweite Herausforderung besteht darin, eine reibungslose Kommunikation zwischen *Frontend* und *Backend* sicherzustellen. Hierbei können Schwierigkeiten mit dem *Typsystem* auftreten, da Konzepte in einer Programmiersprache nicht immer nahtlos in einer anderen umsetzbar sind.

3. Eine dritte Herausforderung entsteht bei der Implementierung von *Autorisierung*, da diese für beide Anwendungen (*Frontend* und *Backend*) separat berücksichtigt werden muss.

All diese Aspekte tragen, aus Autorsicht, zu einer erhöhten Komplexität bei, die durch die derzeit favorisierte strikte Trennung von *Frontend* und *Backend* verursacht wird.

Es existieren bereits Ansätze, bei denen eine strikte Trennung zwischen *Frontend* und *Backend* nicht zwingend erforderlich ist und diese Grenzen verschwimmen lassen, indem alles auf einer Seite läuft und einheitlich gebaut wird. Bei diesen

Ansätzen treten viele der genannten Probleme nicht auf. Als Beispiel zeigt Vermeir (Vermeir 2022: 125) das *Framework Blazor Server* auf, welches es ermöglicht, die Programmiersprache C# im *Frontend* zu verwenden und direkte Aufrufe an Hardwarekomponenten (z.B. Maschinen, Datenbanken) zu stellen ohne die Nutzung von *APIs*. Allerdings gehen solche *Framework*s mit gewissen Einschränkungen in Bezug auf die Flexibilität, Schnelligkeit und Netzwerklast im Vergleich zu SPAs oder MPAs einher. So ist es beispielsweise mit *Blazor Server* derzeit nicht möglich, eine offlinefähige Version bereitzustellen, da eine ständige Netzwerkverbindung erforderlich ist (vgl. Gingter 2020: 46:52 – 47:45). Dennoch könnte ein solcher sogenannter „Fullstack-Ansatz“ in einigen Projekten ausreichend sein und die Gesamtkomplexität der Softwareerstellung reduzieren.

Der Softwaredienstleister XITASO entwickelt seit 2011 individuelle Softwarelösungen für Kunden aus der Maschinenbau- und Medizintechnik-Branche. Bei der Architekturauswahl im Bereich der Webarchitekturen für neue Softwareprojekte fällt oft auf, dass sowohl die Entwickler der Kunden als auch die Entwickler von XITASO die verschiedenen Alternativen für das Projekt evaluieren. Dennoch neigen die meisten Teams am Ende oft dazu, auf vertraute Vorgehensweisen zurückzugreifen, wie beispielsweise die Entscheidung für die SPA, bei der sich das Team sicher fühlt und bereits Erfahrung hat. Dabei bleibt jedoch unklar, ob diese Architekturentscheidungen immer die besten Lösungen für die jeweiligen Problemstellungen sind. Eine umfassende und detaillierte Übersicht der Kriterien, die für und gegen eine Architektur sprechen, sind nicht an einem einheitlichen Punkt in der Literatur beschrieben und nicht leicht zu finden. Daher möchte diese Arbeit einen Beitrag zur besseren Entscheidungsfindung leisten, indem die wichtigsten Kriterien gefunden und niedergeschrieben wurden.

Das genau Ziel der Arbeit wird hierbei im nächsten Kapitel (Kap. 1.2: „Ziel der Arbeit“) aufgezeigt.

1.2 Ziel der Arbeit

Die vorliegende Arbeit hatte als Ziel die Schwierigkeit der Architekturentscheidungen, die in Kapitel 1.1 („Ausgangslage und Unternehmen“) beschrieben wurde, anhand des Ablaufplans in Abbildung 2 aus Kapitel 1.4 („Beschreibung des Vorgehens“), in der Softwareentwicklung bei Web-Projekten zu untersuchen. Dabei wurden die verschiedenen Webanwendungsarchitekturen betrachtet und Kriterien definiert, die für oder gegen die jeweiligen Ansätze sprechen. Diese Kriterien wurden in einen Kriterienkatalog aufgenommen und dann ausgewählten Personen aus der Software-Engineering *Community* von XITASO in Form eines Experteninterviews vorgestellt und dabei bewertet. Außerdem wurden für ausgewählte Architekturen (SPA, MPA, Fullstack) Software Prototypen entwickelt, die die definierten Kriterien auf die Probe gestellt haben, um die vergebenen Punktzahlen für die Architekturen zu testen. Nach der Bewertung dieser Experimente wurde der Kriterienkatalog (siehe Anhang F: „Finale Version des Kriterienkatalogs nach den Experimenten“) vervollständigt, um zukünftig in neuen Projekten als Instrument bei der Architekturentscheidung dienen zu können. Außerdem enthält die Arbeit eine Forschungsfrage und hat eine Annahme aufgestellt. Diese wurden während der Bearbeitung und der Erstellung des Kriterienkatalogs beantwortet.
Nachfolgend im Kapitel 1.2.1 („Konkretisierung der Forschungsfrage“) und im Kapitel 1.2.2 („Aufstellung der Annahme“) werden diese beiden Teile konkret aufgezeigt.

1.2.1 Konkretisierung der Forschungsfrage

Die Möglichkeit den Code für eine ganze Webanwendung an einer Stelle in der Architektur zu schreiben, wie es aus Kapitel 1.1 („Ausgangslage und Unternehmen“) hervorgeht, stellt die klassische Trennung in *Frontend* und *Backend* in Frage.

Daher stellte diese Arbeit die folgende Forschungsfrage auf: „Ist die Trennung zwischen *Frontend* und *Backend* in webbasierten Systemen noch zeitge-

mäß?“.
Mit der Entwicklung des Kriterienkatalogs für die verschiedenen Architekturen, sollte ebenfalls herausgefunden werden, ob in Zukunft mehr Architekturen und *Framework*s entwickelt werden sollten, die keine klare Trennung von *Frontend* und *Backend* erfordern. Die Beantwortung der Forschungsfrage findet in Kapitel 8.1 („Antwort auf die Forschungsfrage aus Kapitel 1.2.1) statt.

1.2.2 Aufstellung der Annahme

Die Herausforderungen, die in Kapitel 1.1 („Ausgangslage und Unternehmen“) für die SPA beschrieben wurden, zeigen, dass das erforderliche Fachwissen in *Frontend* und *Backend* zu zusätzlichem Aufwand und größerer Komplexität führt, wenn eine strenge Trennung dieser Komponenten eingehalten wird.

Diese Arbeit stellte daher die Annahme auf, dass genau eine solche Trennung zu unnötiger Komplexität in der Entwicklung in einigen Projekten führen kann und es deswegen wichtig ist zu prüfen, ob die Fullstack-Architektur ausreichen könnte und so den Entwicklungsaufwand vermindert und die Komplexität reduziert.
Die genaue Definition einer Fullstack-Architektur wird in Kapitel 2.5 („Erklärung der Fullstack-Architektur ohne direkte Trennung von *Frontend* und *Backend*“) aufgezeigt. Außerdem wird die Annahme in Kapitel 8.2 („Beurteilung der Annahme aus Kapitel 1.2.2“) beantwortet.

Nachfolgend wird im Kapitel 1.3 („Abgrenzung der Arbeit“) aufgezeigt, welche Architekturen genau in dieser Arbeit behandelt wurden und in welchem Umfang die Software Prototypen dieser beschrieben werden.

1.3 Abgrenzung der Arbeit

Es existieren eine Vielzahl von verschiedenen Webanwendungsarchitekturen. Harsh (Harsh 2023) zeigt in seinem Beitrag zum Thema, was eine Webanwendungsar-

chitektur ist, die verschiedenen, nachfolgend Aufgelisteten, um zwei ergänzt, auf:

- vorgerenderte Anwendungsarchitektur
- serverseitige gerenderte Architektur
- Multi-Page-Architektur (ergänzt)
- Single-Page-Architektur
- isomorphe Anwendungsarchitektur
- progressive Webanwendungsarchitektur
- Fullstack-Architektur (orientiert an *Blazor Server* in dieser Arbeit) (ergänzt)
- serviceorientierte Architektur
- Microservice-Architektur
- serverlose Architektur

Außerdem zeigt er die verschiedenen Schichten in einer Webanwendungsarchitektur auf, die alle Architekturen, an verschiedenen Stellen und in verschiedenen Komponenten, beinhalten. Hierbei erläutert er die Präsentations-, Geschäfts- und Persistenz-Schicht. Eine Diskussion aller Webanwendungsarchitekturen mit den unterschiedlichen Anordnungen der Schichten würde jedoch von der abstrakten Diskussion der *Makroarchitekturen* ablenken, die die Arbeit führen wollte. Daher wurde die Betrachtung der Architekturen, die die *Mikroarchitektur*ebene betrachten, wie die Serviceorientierte-, die Microservices- und die Serverlose-Architektur, ausgeschlossen. Auch eine progressive Webanwendungsarchitektur wurde nicht extra betrachtet, da diese nur eine Erweiterung der SPA um eine plattformübergreifende Funktion ist. Die Arbeit fokussierte sich also, mit einem **hohen Abstraktionsblickwinkel**, auf die **Multi-Page-Architektur** (Multi-Page-Architektur, serverseitige gerenderte Architektur und vorgerenderte Anwendungsarchitektur), die in Kapitel 2.2 („Die Multi-Page-Architektur im Überblick“) aufgezeigt wird, die **Single-Page-Architektur** (Single-Page-Architektur und progressive Architektur), die in Kapitel 2.3 („Erklärung der Single-Page-Architektur“)

aufgezeigt wird und die **selbst benannte Fullstack-Architektur** (Fullstack-Architektur und isomorphe Anwendungsarchitektur), die in Kapitel 2.4 („Erklärung der Fullstack-Architektur ohne direkte Trennung von *Frontend* und *Backend*“) aufgezeigt wird.
Des Weiteren werden nur die wichtigsten Funktionen der für diese Arbeit entwickelten Software Prototypen erläutert, um die in Kapitel 5.1 („Erklärung der Anforderungen an die Experimente“) definierten Anforderungen zu diskutieren. So wurde sich auf die Anforderungen der zu erstellenden Kriterien fokussiert. Daher wurde darauf verzichtet, den kompletten Aufbau und den Umfang der Prototypen zu beschreiben und zu untersuchen. Es wurde daher nicht der gesamte Code der Software abgebildet, ist aber auf GitHub (MIT-Lizenz) einsehbar unter folgendem Link: `https://github.com/Murtl/bachelorarbeit-mertl`. Außerdem ist der Code auf einer DVD abgespeichert, die dieser Arbeit beiliegt (siehe Anhang G: „DVD mit Code der Experimente“).

Nach dieser Eingrenzung der Architekturen und der Experimente, folgt im nächsten Kapitel (Kap. 1.4: „Beschreibung des Vorgehens“) eine genaue Beschreibung des gesamten Vorgehens der Arbeit unter Berücksichtigung der genannten Abgrenzungen.

1.4 Beschreibung des Vorgehens

Die Motivation dieser Arbeit wurde im aktuellen Kapitel 1.1 bis 1.3 beschrieben. Dabei zeigte sich, dass eine strikte Trennung von *Frontend* und *Backend* in Frage gestellt wurde. Außerdem wurde die Abgrenzung der Arbeit aufgezeigt und nach der Beschreibung des Vorgehens wird ebenfalls das Ergebnis in Kapitel 1.5 („Ergebnis der Arbeit“) kurz vorgestellt und die Zielgruppe in Kapitel 1.6 („Definition der Zielgruppe“) definiert.

Um die in Kapitel 1.2.1 („Konkretisierung der Forschungsfrage“) aufgeworfene Forschungsfrage zu beantworten und die in Kapitel 1.2.2 („Aufstellung der Annahme“) formulierte Annahme zu bewerten, wurde in dieser Arbeit ein Kriterienkatalog (siehe Anhang F: „Finale Version des Kriterienkatalogs nach den Experimenten“) entwickelt. Die schrittweise Entwicklung dieses Kriterienkatalogs wurde gemäß dem in Abbildung 2 dargestellten Ablaufplan durchgeführt.

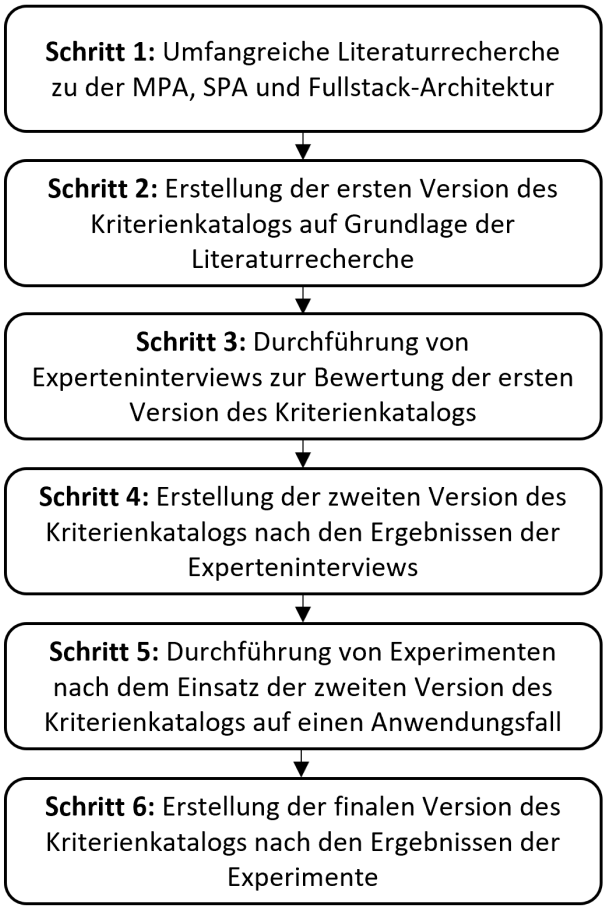


Abbildung 2: Ablaufplan der vorliegenden Arbeit (eigene Darstellung)

Nachfolgend werden den folgenden Kapiteln verschiedene Schritte des Ablaufplans zugeordnet. Die genaue Einteilung wird durch kurze Beschreibungen der Kapitel aufgezeigt und ist auch zu Beginn jedes Kapitels zu finden.

Im zweiten Kapitel („Beschreibung der Webanwendungsarchitekturen“) wird die Single-Page-Architektur, die Multi-Page-Architektur und die selbst benannte Fullstack-Architektur beschrieben, da ein abstrakter Blick auf die *Makroarchitekturen* geworfen wurde, wie aus Kapitel 1.3 („Abgrenzung der Arbeit“) hervorgeht. Das zweite Kapitel soll daher maßgeblich zum Verständnis dieser Arbeit beitragen und Grundlagen zusammenfassen. Für die Ermittlung des Inhaltes dieses Kapitels wurde eine umfassende und möglichst internationale Recherche durchgeführt, um relevante wissenschaftliche und technische Literatur zu den drei Architekturansätzen (SPA, MPA, Fullstack) zu identifizieren. Ziel war es, die Architekturen tiefgreifend zu verstehen und relevante Kriterien für und gegen diese einzelnen Architekturen zu definieren. Dieses Kapitel zeigt den Schritt 1 im Ablaufplan (siehe Abbildung 2).

Im dritten Kapitel („Aufstellung des Kriterienkatalogs zur Ermittlung der passenden Architektur für ein Projekt“) wird gezeigt, wie die erste Version des Kriterienkatalogs (siehe Anhang A: „Erste Version des Kriterienkatalogs nach der Literaturrecherche“) für die unterschiedlichen Architekturen aufgestellt wurde. Dazu diente die vorherige erfolgte Literaturrecherche als Grundlage. Außerdem wurde ein Punktesystem entwickelt, welches es ermöglicht, eine Webarchitektur anhand der vorliegenden Problemstellung abzuleiten. Dieses Kapitel zeigt den Schritt 2 im Ablaufplan (siehe Abbildung 2).

Das vierte Kapitel („Evaluation des Kriterienkatalogs mit Experteninterviews“) beschäftigt sich mit der durchgeführten Evaluation des Kriterienkatalogs durch den Einsatz von Experteninterviews. Dabei wurden drei Mitglieder der Software-Engineering *Community* von XITASO zu der ersten Version des aufgestellten Kriterienkatalogs befragt. So wurde die zweite Version (siehe Anhang C: „Zweite Version des Kriterienkatalog nach den Experteninterviews“) auf die Bedürfnisse und Vorstellungen der *Community* zugeschnitten und verfeinert. Ebenfalls wurde die Aussagekraft des Kriterienkatalogs durch die praxisorientierte Qualitätskontrolle erhöht. Dieses Kapitel zeigt die Schritte 3 und 4 im Ablaufplan (siehe Abbildung 2).

Das fünfte Kapitel („Erläuterung der durchgeführten Experimente“) erläutert die Durchführung der Experimente. Dabei wurden zuerst Anforderungen in Form eines Anwendungsfalles definiert, um festzuhalten, welche Punkte genau untersucht werden sollten. Die Experimente lieferten ein direkter Vergleich der drei Ansätze und brachten Abweichungen bei den ermittelten Kriterien der zweiten Version des Kriterienkatalogs hervor. Konkret wurde eine Beispielanwendung mit *ASP.NET Core 7.0* MVC (MPA), eine mit *Vue 3.3* & *ASP.NET Core 7.0* (SPA) und eine mit *Blazor Server* (Fullstack-Architektur) entwickelt. Dieses Kapitel zeigt den Schritt 5 im Ablaufplan (siehe Abbildung 2).

Im sechsten Kapitel („Ergebnis der drei Experimente“) werden die Ergebnisse der durchgeführten Experimente vorgestellt. Diese ermöglichten eine systematische und objektive Beurteilung der Leistungsfähigkeit der Ansätze und lieferten Erkenntnisse darüber, welcher Ansatz unter welchen Bedingungen besser geeignet ist. Zusammengefasst wurden die Ergebnisse in einer tabellarischen Übersicht. Zudem sind die Erkenntnisse in die zweite Version des Kriterienkatalogs eingeflossen und dieser wurde final (siehe Anhang F: „Finale Version des Kriterienkatalogs nach den Experimenten“) fertiggestellt. Dieses Kapitel zeigt den Schritt 6 im Ablaufplan (siehe Abbildung 2).

Das siebte Kapitel („Diskussion“) zeigt die Diskussion über die Ergebnisse und den entstandenen Kriterienkatalog der Bachelorarbeit. Dabei wurden die Ergebnisse interpretiert und Stärken und Schwächen der Arbeit aufgezeigt und kritisch betrachtet.

Im achten Kapitel („Fazit“) wird das Fazit der Arbeit aufgezeigt. Dabei wird die Antwort auf die Forschungsfrage aus Kapitel 1.2.1 („Konkretisierung der Forschungsfrage“) und die Beurteilung der Annahme aus Kapitel 1.2.2 („Aufstellung der Annahme“) dargestellt. Außerdem wird ein Ausblick über den zukünftigen Einsatz des Kriterienkatalogs und über mögliche Folgearbeiten gegeben.

Im nächsten Kapitel 1.5 („Ergebnis der Arbeit“) wird das Ergebnis der Arbeit aufgezeigt, welches durch das aufgezeigte Vorgehen erzielt wurde.

1.5 Ergebnis der Arbeit

Mit dieser Bachelorarbeit wurde ein Kriterienkatalog (siehe Anhang F: „Finale Version des Kriterienkatalogs nach den Experimenten“) entwickelt, der bei Architekturentscheidungen in neuen Webprojekten als Leitfaden dienen kann. Mithilfe dieses Katalogs können Softwareentwickler anhand der geforderten Anforderungen und des gewünschten Flexibilität Niveaus besser abschätzen, ob eine strikte Trennung von *Frontend* und *Backend* angebracht ist, in Form der Single-Page-Architektur oder der Multi-Page-Architektur oder ob eine alternative Architektur in Form des Fullstack-Ansatzes im konkreten Anwendungsfall eine besser geeignete Option darstellt. Die weiteren Vorteile, die sich durch die Anwendung des Kriterienkatalogs ergeben, werden in Kapitel 3.3 („Darstellung der Vorteile durch den Kriterienkatalog“) beschrieben. Der entwickelte Kriterienkatalog soll nicht nur als eigenständiges Ergebnis dieser Arbeit dienen, sondern kann und darf auch als Grundlage für weitere Abschlussarbeiten dienen (siehe Kap. 8.3: „Ausblick“). Durch kontinuierliche Weiterentwicklung und Aktualisierung können neue entstehende Architekturen und Kriterien in den Katalog aufgenommen werden. Auf diese Weise kann der Katalog als dynamisches Instrument dienen, das sich den ständig weiterentwickelnden Anforderungen und dem Fortschritt der Technik in Webprojekten anpasst.

Im nächsten Kapitel (Kap. 1.6: „Definition der Zielgruppe“) wird beschrieben, für welchen Leserkreis diese Arbeit gedacht ist, um zu zeigen, wer genau einen Vorteil aus dem entwickelten Kriterienkatalog ziehen kann.

1.6 Definition der Zielgruppe

Die vorliegende Arbeit richtet sich an alle Informatiker und Software-Architektur-Interessierten, die eine Webanwendung entwickeln wollen und sich zuvor informieren wollen, welche Webarchitektur für ihre Problemstellung am besten geeignet sein könnte. Der entwickelte Kriterienkatalog kann als Instrument herangezo-

gen, um einen Webarchitekturvorschlag und eine fundierte Diskussionsgrundlage für die jeweilige vorliegende Problemstellung zu liefern.

1.7 Zusammenfassung

Das erste Kapitel („Einleitung“) gibt in Kapitel 1.1 („Ausgangslage und Unternehmen“) einen Überblick über die Entwicklung der Webanwendungsarchitekturen Multi-Page-Architektur (Abk.: MPA) und Singe-Page-Architektur (Abk.: SPA). Dabei wurde gezeigt, dass diese Ansätze eine klare Trennung von *Frontend* und *Backend* verfolgen und dadurch die Komplexität, durch z.B. nötiges Fachwissen in beiden Komponenten, erhöhen. Deshalb wurde das *Framework Blazor Server* vorgestellt, welches es ermöglicht den gesamten Code auf dem Server zu schreiben in der Programmiersprache C#. Dieser Ansatz wurde in dieser Arbeit als dritte Variante gesehen und als „Fullstack-Architektur“ bezeichnet.

In Kapitel 1.2 („Ziel der Arbeit“) wurde als Ziel der Arbeit die Entwicklung eines Kriterienkatalogs aufgezeigt. Dieser definiert Kriterien für die Auswahl zwischen den drei genannten Architekturen und hinterfragt dabei ganz bewusst die Notwendigkeit einer strikten Trennung von *Frontend* und *Backend* in Webanwendungen. Des Weiteren wurde erläutert, wie die Forschungsfrage konkretisiert und die Annahme für diese Bachelorarbeit formuliert wurde.

In Kapitel 1.3 („Abgrenzung der Arbeit“) erfolgte eine Eingrenzung der Arbeit. Dabei wurde erörtert, dass ein abstrakter Blickwinkel auf die *Makroarchitekturen* (MPA, SPA, Fullstack-Architektur) geworfen wurde und nur essenzielle Teile der Experimente beschrieben werden.

Das Kapitel 1.4 („Beschreibung des Vorgehens“) erläuterte die Struktur der Arbeit und zeigte den Einsatz unterschiedlicher Methodiken (d.h. Literaturrecherche, Experteninterviews, Experimente, Bewertung) auf, um den Kriterienkatalog zu entwickeln. In Abbildung 2 wurde der Ablaufplan dieser Arbeit präsentiert, wobei die verschiedenen Schritte den nachfolgenden Kapiteln zugeordnet wurden.

Der Kriterienkatalog (siehe Anhang F: „Finale Version des Kriterienkatalogs

nach den Experimenten“) wurde dann in Kapitel 1.5 („Ergebnis der Arbeit“) als Ergebnis der Arbeit vorgestellt. Mithilfe dieses Katalogs können Softwareentwickler anhand der geforderten Anforderungen und des gewünschten Flexibilität Niveaus besser einschätzen, ob die SPA, MPA oder Fullstack-Architektur für das jeweilige Projekt geeignet sein könnte. Die weiteren Vorteile, die sich durch die Anwendung des Kriterienkatalogs ergeben, werden in Kapitel 3.3 („Darstellung der Vorteile durch den Kriterienkatalog“) beschrieben.

Nicht nur Softwareentwickler wurden in Kapitel 1.6 („Definition der Zielgruppe“) als Zielgruppe beschrieben. Vielmehr richtet sich diese Arbeit an alle Informatiker und Software-Architektur-Interessierte und der Kriterienkatalog eignet sich immer dann, wenn eine Diskussionsgrundlage gewünscht wird.

Kapitel 2

Beschreibung der Webanwendungsarchitekturen

Das zweite Kapitel beschreibt zu Beginn die möglichen verschiedenen Webanwendungsarchitekturen und zeigt nochmal auf, warum im weiteren Verlauf des Kapitels nur die Multi-Page-Architektur, die Single-Page-Architektur und die selbst benannte Fullstack-Architektur beschrieben wird. Diese drei Architekturen werden hier näher erläutert und mit Vor- und Nachteilen aufgezeigt, damit ein einheitliches Verständnis dieser Architekturen geschaffen wird und der aufgestellte Kriterienkatalog in Kapitel 3 („Aufstellung des Kriterienkatalogs zur Ermittlung der passenden Architektur für ein Projekt“) besser verstanden werden kann. Das zweite Kapitel zeigt den Schritt 1 im Ablaufplan (siehe Abbildung 2 in Kapitel 1.4: „Beschreibung des Vorgehens“).

2.1 Webanwendungsarchitekturen in der Übersicht

Die vorliegende Arbeit dreht sich um die Untersuchung und Bewertung verschiedener Webanwendungsarchitekturen. Doch bevor drei ausgewählte Architekturen detailliert beschrieben werden, muss zuerst geklärt werden, was überhaupt eine Webanwendungsarchitektur ist. „Einfach ausgedrückt, ist die Architektur ei-

ner Webanwendung eine Übersicht darüber, wie die verschiedenen Komponenten deiner Webanwendung miteinander interagieren“ (Harsh 2023). Ein einfaches Beispiel wurde in Abbildung 1 im Kapitel 1.1 („Ausgangslage und Unternehmen“) aufgezeigt. In dieser sind zwei Komponenten, einmal der Client (Nr. 1 in Abbildung 1) und einmal der Server (Nr. 3 in Abbildung 1), schematisch dargestellt worden. Eine moderne Architektur kann aber in der Praxis viel komplexer sein und einen Schwarm von *Container-Backend-Server*, *Load Balancern*, *API-Gateways* und benutzerorientierten Single-Page-*Frontend*s besitzen (vgl. Harsh 2023). Die unterschiedliche Zusammensetzung dieser Komponenten bilden verschiedene Architekturen. Eine Übersicht der Arten der Webanwendungsarchitekturen wird in Tabelle 1 gegeben, welche zum Teil an die Beschreibungen von Harsh (Harsh 2023) angelehnt sind.

Web-Architektur	**Kurzbeschreibung**
Vorgerenderte Anwendungsarchitektur	Alle Dateien der Anwendung werden vorgeneriert und auf dem Server abgelegt. Der Client kann sich diese bei Bedarf dadurch mit minimalen Ladezeiten holen.
Serverseitig gerenderte Architektur	Der Client holt sich hier ebenfalls fertig gerenderte Dateien vom Server, diese werden aber erst gerendert, wenn sie angefordert werden.
Multi-Page-Architektur	Der Client bekommt bei jeder Anfrage eine neue HTML-Seite. Diese liegen entweder vorgerendert auf dem Server (vorgerenderte Anwendungsarchitektur) oder werden zur Laufzeit auf diesem gerendert (Serverseitig gerenderte Architektur) (siehe Kap. 2.2: „Die Multi-Page-Architektur im Überblick“).

Web-Architektur	**Kurzbeschreibung**
Single-Page-Architektur	Der Client bekommt zum Start der Anwendung nur eine Seite, deren Teilbereiche sich während der Laufzeit laufend anpassen (siehe Kap. 2.3: „Erklärung der Single-Page-Architektur“).
Isomorphe Anwendungsarchitektur	Ist eine hybride Architektur, da diese eine Mischung aus der SPA und der Serverseitig gerenderten Architektur darstellt.
Progressive Webanwendungsarchitektur	Das Prinzip ist hier wie bei der Single-Page-Architektur, aber diese Architektur erlaubt es, durch die Verwendung von *Service Worker*n, die Anwendung auf allen Plattformen anzubieten.
Fullstack-Architektur (orientiert an *Blazor Server* in dieser Arbeit)	Der Client bekommt zum Start der Anwendung nur eine Seite, deren Teilbereiche sich während der Laufzeit durch neuen HTML-Code vom Server anpassen. Hier werden, wie bei der isomorphen Anwendungsarchitektur, Oberflächenelemente auf dem Server gerendert und, im Fall von *Blazor Server*, über eine sogenannte SignalR-Verbindung an den Client gesendet (siehe Kap. 2.4: „Erklärung der Fullstack-Architektur ohne direkte Trennung von *Frontend* und *Backend*“).
Serviceorientierte Architektur	Diese Architektur unterteilt die Webanwendung in Dienste, welche gekoppelt werden, die jeweils eine funktionale Einheit im Unternehmen darstellen.
Microservices-Architektur	Hier werden kleine modulare Komponenten entwickelt, die zu einer Webanwendung zusammengefügt werden. Im Gegensatz zur serviceorientierten Architektur sind Microservices darauf ausgerichtet kleine und kontextbezogene Einheiten zu halten, was die Abhängigkeiten minimiert.

Web-Architektur	Kurzbeschreibung
Serverlose Architektur	Hier wird die Webanwendung in Funktionen zerlegt, welche auf Funktion-als-Service Plattformen betrieben werden. Dadurch müssen Entwickler sich nicht selbst um die serverseitige Infrastruktur kümmern.

Tabelle 1: Übersicht der Arten der Webanwendungsarchitekturen (eigene Darstellung)

Die Betrachtung jeder aufgezeigten Architektur aus Tabelle 1, wie auch bereits in Kapitel 1.3 („Abgrenzung der Arbeit“) erwähnt, würde von der abstrakten Diskussion der *Makroarchitektur*en ablenken, die die Arbeit führen wollte. Daher wurde die Betrachtung der Architekturen, die die *Mikroarchitektur*ebene betrachten, wie die Serviceorientierte-, die Microservices- und die Serverlose-Architektur, ausgeschlossen. Die Arbeit fokussierte sich also, mit einem **hohen Abstraktionsblickwinkel**, auf die **Multi-Page-Architektur** (Multi-Page-Architektur, vorgerenderte Anwendungsarchitektur und serverseitig gerenderte Architektur), die in Kapitel 2.2 („Die Multi-Page-Architektur im Überblick“) aufgezeigt wird, die **Single-Page-Architektur** (Single-Page-Architektur und progressive Architektur), die in Kapitel 2.3 („Erklärung der Single-Page-Architektur“) aufgezeigt wird und die **selbst benannte Fullstack-Architektur** (Fullstack-Architektur und isomorphe Architektur), die in Kapitel 2.4 („Erklärung der Fullstack-Architektur ohne direkte Trennung von *Frontend* und *Backend*“) aufgezeigt wird.

Für alle drei Architekturen werden die folgenden Merkmale, in unterschiedlicher Reihenfolge, diskutiert:

- Benutzererfahrung
- Browseranforderung
- Entwicklererfahrung

- Entwicklungszeit
- Erweiterbarkeit
- Offlinefähigkeit
- Performanz
- Sicherheit
- Serverbelastung
- Skalierbarkeit
- Suchmaschinenoptimierung (Abk.: SEO)

Eine tabellarische Übersicht über die diskutierten Merkmale für die einzelnen Architekturen wird in Kapitel 2.5 („Vor- und Nachteile der vorgestellten Webanwendungsarchitekturen in der Übersicht“) gegeben.

Als erste Architektur wird im nächsten Kapitel (Kap. 2.2: „Die Multi-Page-Architektur im Überblick“) die Multi-Page-Architektur aufgezeigt, da diese zeitlich gesehen als Erstes entstanden ist.

2.2 Die Multi-Page-Architektur im Überblick

Die Mutli-Page-Architektur war die erste webbasierte Architektur überhaupt. Mit der Entstehung von Webanwendungen in den 1990er Jahren, ist diese erstmalig dokumentiert worden. Das Hauptmerkmal dieser Architektur ist es, dass diese aus mehreren HTML-Seiten besteht. Die verschiedenen HTML-Seiten liegen hierbei statisch oder dynamisch auf dem Server und werden vom Client bei Bedarf angefragt und angezeigt (vgl. Kulesza u. a. 2020: 3). In dieser Client-Server-Architektur wird das *Frontend* und *Backend* in einer Anwendung entwickelt, dort

aber klar abgetrennt durch eine Codetrennung. Der genaue Prozess und die Architektur dahinter wird in Abbildung 3 schematisch dargestellt und wird nachfolgend mit der Definition von statischen und dynamischen Seiten aufgezeigt.

Architektur und Funktionsweise

Sobald eine Webanwendung im Browser (Client (Nr.1 in Abbildung 3)) aufgerufen wird, die nach der Multi-Page-Architektur entwickelt wurde, wird eine HTTP-Anfrage (Nr. 2 in Abbildung 3) an den Server (Nr. 3 in Abbildung 3) geschickt. Diese Anfrage wird vom Server (Nr. 3 in Abbildung 3) bearbeitet, indem die entsprechende HTML-Seite zu der Anfrage rausgesucht und über eine HTTP-Antwort (Nr. 4 in Abbildung 3) zurückgeschickt wird. Abschließend zeigt der Client (Nr. 1 in Abbildung 3) die HTML-Seite an. Dieser HTTP-Anfrage/Antwort Prozess hat seinen eigenen Aufbau und verschiedene Statuscodes, die von Ackermann (Ackermann 2021: 164 - 174) erläutert werden, aber hier nicht weiter betrachtet werden.

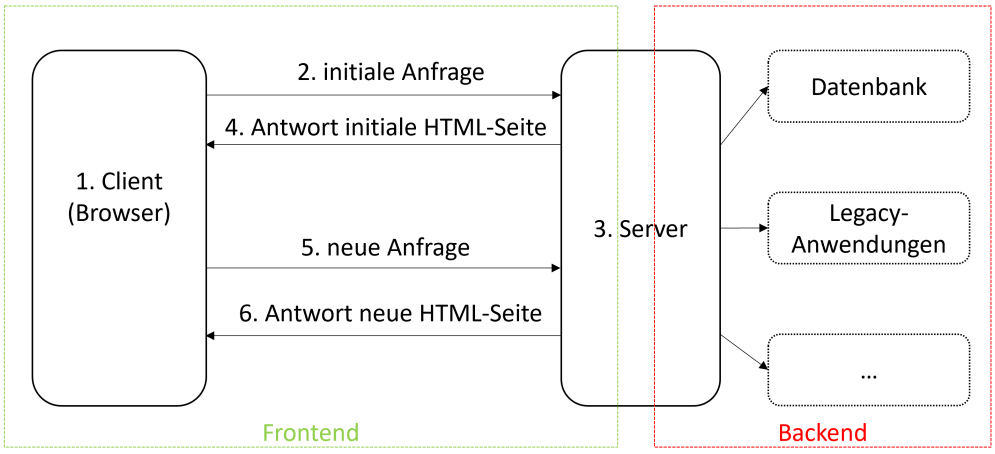


Abbildung 3: Beispielhafte Multi-Page-Architektur (eigene Darstellung)

Sobald ein Nutzer, auf seiner aktuell angezeigten HTML-Seite in seinem Browser

(Client (Nr. 1 in Abbildung 3)), auf einen Link klickt, um auf eine andere Ansicht zu wechseln, wird eine neue HTML-Seite vom Server angefordert (Nr. 5 in Abbildung 3). Der Server (Nr. 3 in Abbildung 3) sucht die jeweilige HTML-Seite wieder raus und sendet diese in Form einer HTTP-Antwort (Nr. 6 in Abbildung 3) zurück an den Client (Nr. 1 in Abbildung 3). Dort sieht ein Nutzer dann die neue Seite (vgl. Himschoot 2022: 351). Während diesem Prozess kann der Server entweder statische HTML-Seiten, diese sind bereits vollständig gerendert und zusammengebaut auf dem Server oder dynamische HTML-Seiten, welche erst zum Zeitpunkt der Anfrage zusammengebaut werden, zurückgeben. Der Prozess, der bei einer dynamischen HTML-Seite auf dem Server ausgeführt wird, wird als **Server-Side-Rendering** bezeichnet und wird im Folgenden näher betrachtet.

Server-Side-Rendering

Eine dynamische HTML-Seite wird beim Server-Side-Rendering erst zum Zeitpunkt der Anfrage zusammengebaut. Dabei wird auf dem Server (Nr. 3 in Abbildung 3) Code ausgeführt und es besteht die Möglichkeit, wie in Abbildung 3 schematisch dargestellt, auf eine Datenbank oder auf Legacy-Anwendungen zuzugreifen, um die notwendigen Daten zu holen und die neue HTML-Seite programmatisch zu generieren. Sobald dieser Prozess abgeschlossen ist, wird die neue Seite wieder über eine HTML-Antwort (Nr. 6 Abbildung 3) an den Client (Nr. 1 in Abbildung 3) zurückgeschickt und dort angezeigt. Bei jeder neuen Anfrage durch einen Nutzer, muss dieser Prozess zur Generierung einer neuen HTML-Seite, aber wiederholt werden (vgl. Himschoot 2022: 351).
Die Anwendung der Multi-Page-Architektur bringt verschiedene Vor- und Nachteile mit sich, welche nachfolgend aufgelistet werden.

Vorteile

Eine Übersicht über die verschiedenen Vorteile der MPA ist in Tabelle 2 aufgelistet.

Vorteil	**Beschreibung**
Browseranforderung	Die MPA läuft auf jedem Browser, da JavaScript nicht notwendig ist für die Entwicklung (vgl. Schwichtenberg 2020).
Entwicklungszeit	Die MPA hat in der Regel eine kürzere Entwicklungszeit als die SPA. Es muss nur eine Anwendung entwickelt werden und es kann auf die wie bei der SPA nötige *API* für die Kommunikation verzichtet werden (vgl. Schwichtenberg 2020).
Erweiterbarkeit	Die MPA lässt sich gut erweitern, erfordert aber bei einer Erweiterung, eine Anpassung in *Frontend* und *Backend*, da diese beiden Komponenten stärker verknüpft sind (vgl. Harris 2021: 1:43 - 2:10).
Performanz -> initialer Ladezyklus	Lädt initial schneller als die SPA, da nur eine HTML-Seite (diese kann statisch oder dynamisch sein) geladen werden muss (vgl. Patadiya 2023).
Sicherheit -> ungewolltes Offenlegen von Daten	Bei der MPA können vertrauliche Daten auf dem Server gespeichert werden (vgl. Microsoft 2023g). Durch die Codetrennung von *Frontend* und *Backend* in der Anwendung ist es leichter, keine vertraulichen Daten beim Entwickeln am Client zu verarbeiten oder offenzulegen.
Skalierbarkeit	Bei der MPA kann der Client oder auch der Server die Zustandsverwaltung übernehmen (vgl. Microsoft 2023g). Hier besteht also die Möglichkeit eine *horizontale* oder auch eine *vertikale Skalierung* problemlos durchzuführen.

Suchmaschinen-optimierung (SEO)	Die MPA ist durch ihre vielen Seiten gut für SEO geeignet (vgl. Patadiya 2023).

Tabelle 2: Vorteile der Multi-Page-Architektur (eigene Darstellung)

Nachteile

Eine Übersicht über die verschiedenen Nachteile der MPA ist in Tabelle 3 aufgelistet.

Nachteil	**Beschreibung**
Benutzererfahrung	Jede Benutzerinteraktion führt zum neuen Laden der gesamten Seite, welche die neuen Inhalte dann bereitstellt (vgl. Patadiya 2023).
Entwicklererfahrung	Es ist Fachwissen im *Frontend* (HTML, CSS und JavaScript) und im *Backend* (z.B. C#) nötig (vgl. Harris 2021: 1:43 - 2:10). Diese Teile werden durch eine Codetrennung in der Anwendung getrennt.
Offlinefähigkeit	Die MPA ist nicht offlinefähig, da immer wieder neue HTML-Seiten vom Server geladen werden müssen und dafür eine Internetverbindung nötig ist (vgl. Himschoot 2022: 351).
Performanz -> nachladen neuer Daten	Der Client fordert bei jeder Benutzerinteraktion eine neue HTML-Seite (diese kann statisch oder dynamisch sein) vom Server an. Der Server stellt bzw. rendert die neue HTML-Seite und gibt diese an den Client zum Anzeigen zurück. Dies beeinträchtigt die Performanz stark (vgl. Patadiya 2023).

Nachteil	Beschreibung
Serverbelastung	Jede Benutzerinteraktion fordert eine neue HTML-Seite an, der Server ist also dauerhaft in Benutzung und wird belastet (vgl. Patadiya 2023). Bei der MPA kann der Client oder auch der Server die Zustandsverwaltung übernehmen. Dadurch kann der Server zusätzlich belastet werden (vgl. Microsoft 2023g).

Tabelle 3: Nachteile der Multi-Page-Architektur (eigene Darstellung)

All diese Nachteile und vor allem der, das jede Benutzerinteraktion ein komplettes neu Laden der Seite verursacht, haben zu einer Weiterentwicklung der Webanwendungsarchitekturen beigetragen. Diesen Nachteil hat das Gegenstück zu der Multi-Page-Architektur, die Single-Page-Architektur nicht, weshalb die SPA aktuell eine beliebte Alternative zu der MPA darstellt.

Im nächsten Kapitel (Kap. 2.3: „Erklärung der Single-Page-Architektur“) wird beschrieben, wie die Single-Page-Architektur entstanden ist und wie sie funktioniert.

2.3 Erklärung der Single-Page-Architektur

Im Jahre 1995 erfand Brendan Eich JavaScript. Dies war der Startschuss für die Entstehung der Single-Page-Architektur. Mit JavaScript haben Webseiten die Möglichkeit bekommen, den Kontrollbaum des Browsers, auch bekannt als „*Document Object Model (Abk.: DOM)*“, zu manipulieren (vgl. Himschoot 2022: 351f). Der Anwendungsstatus und die Logik einer Webanwendung werden hierbei hauptsächlich am Client (z.B. Browser) verwaltet. Im Gegensatz zu der MPA existiert in der SPA nur eine HTML-Seite, deren Seitenteile nur bei Bedarf neu gerendert

werden. Es gibt also kein ganzseitiges neu Laden mehr wie bei der in Kapitel 2.2 („Die Multi-Page-Architektur im Überblick“) beschriebenen MPA (vgl. Sun 2019: 141). In dieser Client-Server-Architektur stellt der Client das *Frontend* und der Server das *Backend* dar. Der genaue Prozess des Seitenaufbaus und der Seitenaktualisierung wird in Abbildung 4 schematisch dargestellt und nachfolgend aufgezeigt.

Architektur und Funktionsweise

Sobald eine Webanwendung im Browser (Client (Nr.1 in Abbildung 4)) aufgerufen wird, die nach der Single-Page-Architektur entwickelt wurde, wird eine HTTP-Anfrage (Nr. 2 in Abbildung 4) an den Server (Nr. 3 in Abbildung 4) geschickt. Diese initiale Anfrage wird vom Server (Nr. 3 in Abbildung 4) bearbeitet, indem die entsprechende HTML-Seite zu der Anfrage rausgesucht und über eine HTTP-Antwort (Nr. 4 in Abbildung 4) zurückgeschickt wird. Hierbei wird initial die gesamte Anwendung übertragen, da nur eine HTML-Seite insgesamt vorliegt bei der SPA. Abschließend zeigt der Client (Nr. 1 in Abbildung 4) die HTML-Seite an. Dieser HTTP-Anfrage/Antwort Prozess hat seinen eigenen Aufbau und verschiedene Statuscodes, die von Ackermann (Ackermann 2021: 164 - 174) erläutert werden, aber hier nicht weiter betrachtet werden.

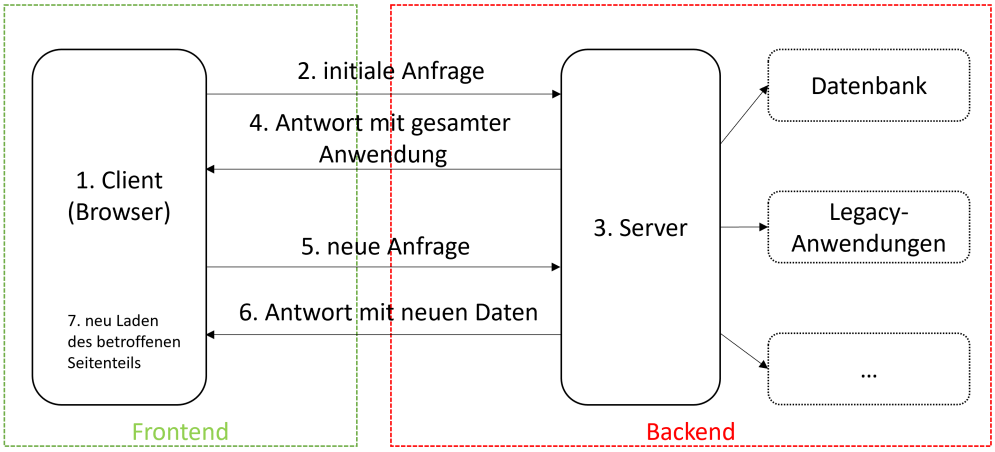


Abbildung 4: Beispielhafte Single-Page-Architektur (eigene Darstellung)

Interagiert ein Nutzer nun auf dieser initial geladenen HTML-Seite z.B. mit einer Schaltfläche, um auf eine andere Ansicht zu gelangen, sendet der Client (Nr. 1 in Abbildung 4) eine neue HTTP-Anfrage (Nr. 5 in Abbildung 4) an den Server (Nr. 3 in Abbildung 4). Damit eine Webanwendung im Hintergrund Daten mit dem Server austauschen kann, ohne dass die komplette Seite neu geladen werden muss, sondern nur ein Teil davon, ist die Verwendung der Technologie „*Asynchronous JavaScript und XML (Abk.: AJAX)*“ nötig. Außerdem wird hier eine sogenannte *Web-API* benötigt, die Serverendpunkte definiert, an denen die *AJAX*-Anfragen (Nr. 5 in Abbildung 4) für neue Daten geschickt werden (vgl. Ackermann 2021: 229 - 235). Der Server (Nr. 3 in Abbildung 4) bearbeitet die eingegangene Anfrage (Nr. 5 in Abbildung 4) indem die neuen Daten zusammengesammelt werden. Dabei hat der Server (Nr. 3 in Abbildung 4) die Möglichkeit auf eine Datenbank oder auch Legacy-Anwendungen zuzugreifen. Sobald alle nötigen Daten vorhanden sind, sendet der Server (Nr. 3 in Abbildung 4) diese über eine HTTP-Antwort (Nr. 6 in Abbildung 4) zurück an den Client (Nr. 1 in Abbildung 4). Die Daten werden dabei häufig in einem *JSON-Format* übergeben. Jedoch gibt es auch andere Formate, wie z.B. „Comma Separated Values (Abk.: CSV)“ oder „Extensible Markup Language (Abk.: XML)“, die für die Datenübertragung genutzt werden können (vgl. Ackermann 2021: 189 - 201). Die neuen Daten werden dann vom

Client (Nr. 1 in Abbildung 4) entgegengenommen und verarbeitet. Als letztes wird der Teil der Seite aktualisiert, der durch die neuen Daten ersetzt wird, ohne die Seite komplett neu zu Laden (Nr. 7 in Abbildung 4). Da nur Teile der Seite neu geladen werden entsteht ein Geschwindigkeitsvorteil der für den Benutzer meist spürbar ist (vgl. Sun 2019: 141).
Die verbesserte Benutzererfahrung ist nicht der einzige Vorteil der SPA. Die verschiedenen Vor- und Nachteile dieser Architektur werden nachfolgend aufgelistet.

Vorteile

Eine Übersicht über die verschiedenen Vorteile der SPA ist in Tabelle 4 aufgelistet.

Vorteil	**Beschreibung**
Benutzererfahrung	Beim Navigieren in der Anwendung finden keine ganzen Seitenaufrufe statt, dies fühlt sich für den Benutzer daher an wie eine Desktopanwendung. Außerdem besteht die Möglichkeit, in der Zeit, in der Daten vom Server angefragt werden, optisch ansprechende Ladeanimationen einzubauen, für eine noch bessere Benutzererfahrung (vgl. Sun 2019: 141).
Erweiterbarkeit	Die SPA ermöglicht es das *Frontend* und *Backend* isoliert voneinander zu erweitern, solange die Endpunkte für die Kommunikation einheitlich bleiben, da zwei getrennte Anwendungen vorliegen (vgl. Schwichtenberg 2020; Thattil 2018). So besteht die Möglichkeit, dass zwei getrennte Teams an der Webanwendung arbeiten.

Vorteil	Beschreibung
Offlinefähigkeit	Sobald die Anwendung einmal vollständig geladen ist, kann diese offline ohne Probleme genutzt werden, solange keine neue Daten-Anfrage angestoßen wird (vgl. Thattil 2018).
Performanz -> nachladen neuer Daten	Die SPA fordert neue Daten über einen HTTP-Anfrage an und rendert bei Erhalt den betroffenen Teil der Seite neu. Dies bietet einen deutlichen Geschwindigkeitsvorteil gegenüber der MPA (vgl. Sun 2019: 141).
Serverbelastung	Der Zustand der Anwendung wird am Client gespeichert, so kann der Server zustandslos betrieben werden und wird nur durch kleine Datenanfragen belastet (vgl. Sun 2019: 141).
Skalierbarkeit	Der Client eines neuen Benutzers verwaltet jeweils den Anwendungsstatus und die Logik (vgl. Sun 2019: 141), also den Zustand. Die SPA eignet sich also für eine *horizontale* oder auch eine *vertikale Skalierung*.

Tabelle 4: Vorteile der Single-Page-Architektur (eigene Darstellung)

Nachteile

Eine Übersicht über die verschiedenen Nachteile der SPA ist in Tabelle 5 aufgelistet.

Nachteil	**Beschreibung**
Browseranforderung	Die SPA erfordert einen modernen Browser, wie z.B. Google Chrome, Firefox, Safari oder Edge, der aktuelles JavaScript unterstützt (vgl. Thattil 2018).
Entwicklererfahrung	Es werden zwei Anwendungen mit einer Schnittstelle dazwischen entwickelt. Es ist also Fachwissen in *Frontend* (HTML, CSS und JavaScript) und im *Backend* (z.B. C#) nötig (vgl. Schwichtenberg 2020; Thattil 2018). Zudem müssen Entwickler besonders eng zusammenarbeiten, damit keine Probleme bei der Kommunikation zwischen den Komponenten auftreten (z.B. unterschiedliche *Typsysteme*) (vgl. GetDevDone Team 2021; Shilpi 2019).
Entwicklungszeit	Die SPA benötigt eine längere Entwicklungszeit verglichen mit der MPA und der Fullstack-Architektur, da zwei Anwendungen geschrieben werden und eine *API* für die Kommunikation entwickelt werden muss (vgl. Schwichtenberg 2020; Thattil 2018).
Performanz -> initialer Ladezyklus	Der erste Ladezyklus einer SPA kann sehr viel Zeit in Anspruch nehmen, da die komplette Anwendung geladen wird. Außerdem benötigt eine SPA viel Rechenkapazität am Client, da viel Code im Browser ausgeführt wird (vgl. Basse 2021).

Nachteil	Beschreibung
Sicherheit -> ungewolltes Offenlegen von Daten	Die SPA speichert sehr viel am Client und ist daher anfällig für schädliches JavaScript, das eingeschleust werden kann (vgl. Basse 2021). Die Entwickler müssen also beim Entwickeln aufpassen, dass keine vertraulichen Daten am Client preisgegeben werden.
Suchmaschinen-optimierung (SEO)	Die SEO ist ein großer Nachteil der SPA, da bei der SPA nur eine HTML-Seite vorliegt. Suchmaschinen bewerten verschiedene Webseiten und das ist bei der SPA nicht möglich (vgl. Patadiya 2023).

Tabelle 5: Nachteile der Single-Page-Architektur (eigene Darstellung)

All diese Nachteile, vor allem die, die durch die Trennung von *Frontend* und *Backend* entstehen, wie z.B. das nötige Fachwissen in beiden Bereichen, haben zu einer Weiterentwicklung der Webanwendungsarchitekturen beigetragen. Es wurden neue Ansätze entwickelt, die es ermöglichen ohne eine strikte Trennung von *Frontend* und *Backend*, Webanwendungen zu entwickeln.

Im nächsten Kapitel (Kap. 2.4: „Erklärung der Fullstack-Architektur ohne direkte Trennung von *Frontend* und *Backend*“) wird *Blazor Server* als einer dieser Ansätze aufgezeigt und in die Fullstack-Architektur eingeordnet.

2.4 Erklärung der Fullstack-Architektur ohne direkte Trennung von *Frontend* und *Backend*

Die Fullstack-Architektur bietet die Möglichkeit das *Frontend* und das *Backend* zusammen zu entwickeln, ohne eine strikte Trennung dieser Komponenten, wie sie bei der klassischen Multi-Page-Architektur, durch Codetrennung oder bei der

Single-Page-Architektur, durch Anwendungstrennung, üblich ist. Es existieren bereits unterschiedliche aktuelle Web-*Frameworks*, wie z.B. *Blazor Server*, *Vaadin* oder auch *Remix.run*, die jeweils auf ihre ganz eigene Art, die Vermischung von *Frontend* und *Backend* umsetzen. Als Vertreter für die Fullstack-Architektur in dieser Arbeit dient das von Microsoft entwickelte *Framework Blazor Server*, da dieses in der Vergangenheit bereits bei XITASO in Projekten abgelehnt wurde, aufgrund einer fehlenden Übersicht wichtiger Kriterien gegenüber anderen Architekturen. Die genaue Architektur, angelehnt an die von Schwichtenberg (Schwichtenberg 2020) und die Funktionsweise von *Blazor Server* wird in Abbildung 5 schematisch dargestellt und nachfolgend erläutert.

Architektur und Funktionsweise

Ein Punkt warum bei *Blazor Server* die Vermischung von *Frontend* und *Backend* möglich ist, ist der, dass *Blazor Server* die Möglichkeit bietet, C# anstatt JavaScript für das *Frontend* zu verwenden. Es wird nur eine Anwendung entwickelt, bei der, der gesamte Code auf dem Server läuft. Daher können *Frontend* Komponenten auch *Backend* Logik beinhalten und darauf zugreifen (vgl. Schwichtenberg 2020).
Eine weitere Besonderheit ist die Kommunikation zwischen Client und Server in dieser Architektur. Im Gegensatz zu einem HTTP-Anfrage/Antwort Verfahren nutzt *Blazor Server* eine sogenannte **SignalR**-Verbindung für die Kommunikation. SignalR ist ein *Framework*, welches es ermöglicht in **Echtzeit** zwischen Client und Server zu kommunizieren. Der zugrunde liegende Mechanismus ist hierbei die Verwendung von ***WebSockets*** (vgl. Vermeir 2022: 144f). Es werden daher keine Anfragen von der Client-Anwendung benötigt, um neue Daten anzufordern vom Server. Sobald die Verbindung zwischen Client und Server hergestellt ist, kann der Server dem Client alle Informationen in Echtzeit zur Verfügung stellen, ohne dass der Client selbst Anfragen stellen muss (vgl. Redaktion 2020).

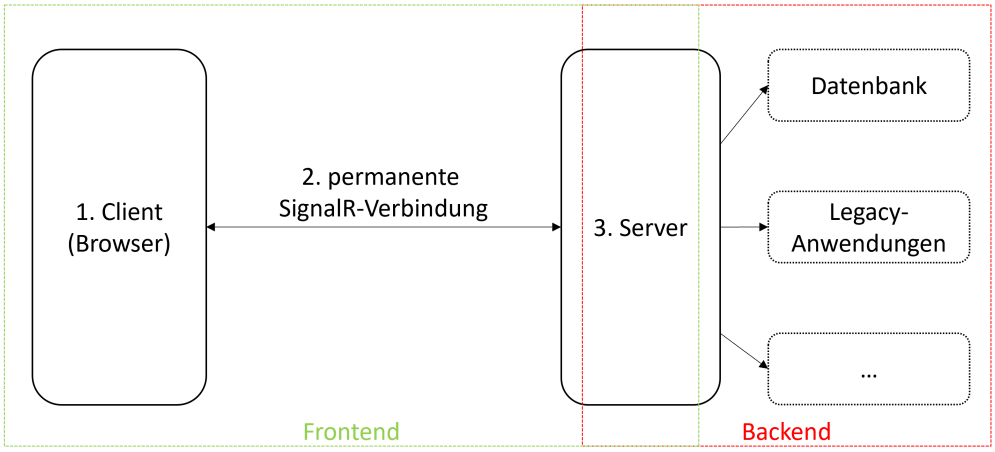


Abbildung 5: Beispielhafte Fullstack-Architektur am Beispiel *Blazor Server* (eigene Darstellung)

Sobald eine Webanwendung im Browser (Client (Nr.1 in Abbildung 5)) aufgerufen wird, die mit *Blazor Server* entwickelt wurde, wird eine SignalR Verbindung (Nr. 2 in Abbildung 5) zwischen Client (Nr. 1 in Abbildung 5) und Server (Nr. 3 in Abbildung 5) aufgebaut. Beim ersten Aufruf überträgt der Server (Nr. 3 in Abbildung 4) eine komplette HTML-Seite über die SignalR-Verbindung (Nr. 2 in Abbildung 5) an den Client (Nr. 1 in Abbildung 5), damit dieser die Seite anzeigen kann. Dennoch merkt sich der Server das initial zum Browser gesendete HTML in Form eines Abbilds des Browserinhalts, dem *DOM*-Baum. Zusätzlich erhält der Client (Nr. 1 in Abbildung 5) eine **JavaScript-Bibliothek** von Microsoft names **„blazor.server.js“** vom Server (Nr. 3 in Abbildung 5) beim ersten Laden. Diese Bibliothek ist dafür verantwortlich Benutzerinteraktionen über die SignalR-Verbindung (Nr. 2 in Abbildung 5) an den Server (Nr. 3 in Abbildung 5) zu übertragen. Der Server (Nr. 3 in Abbildung 5) führt dann die nötigen Änderungen an dem auf dem Server gespeicherten Abbild des *DOM*-Baums, an der betroffenen Stelle, durch und hat dabei die Möglichkeit für notwendige Daten, unter vollen Zugriff auf die *.NET-Umgebung*, auf eine Datenbank oder Legacy-Anwendungen zuzugreifen. Als nächstes werden die durchgeführten *DOM*-Änderungen über die SignalR-Verbindung (Nr. 2 in Abbildung 5) an den Client (Nr. 1 in Abbildung 5) gesendet und dort an der entsprechenden Stelle ausgetauscht. Als letztes wird der

Teil der Seite auf dem Client (NR. 1 in Abbildung 5) aktualisiert, der durch den neuen *DOM*-Teil ersetzt wurde, ohne die Seite komplett neu zu laden. Dies ist möglich, da SignalR *AJAX* nutzt für die Kommunikation im Hintergrund. Dieses Verhalten erinnert an die SPA, da bei *Blazor Server* ebenfalls nur insgesamt eine HTML-Seite existiert, in der immer wieder Teile neu gerendert werden (vgl. Schwichtenberg 2020).
Dieser Ansatz behält also den großen Vorteil einer SPA, genauer gesagt die verbesserte Benutzererfahrung bei. Die genauen Vor- und Nachteile von *Blazor Server* werden nachfolgend aufgelistet.

Vorteile

Eine Übersicht über die verschiedenen Vorteile von *Blazor Server* ist in Tabelle 6 aufgelistet.

Vorteil	**Beschreibung**
Benutzererfahrung	Beim Navigieren in der Anwendung finden keine ganzen Seitenaufrufe statt, dies ist wie bei der SPA. Außerdem besteht die Möglichkeit, in der Zeit, in der auf neue Daten vom Server gewartet wird, optisch ansprechende Ladeanimationen einzubauen, für eine noch bessere Benutzererfahrung (vgl. Schwichtenberg 2020).
Entwicklererfahrung	Bei *Blazor Server* wird nur eine einzige Anwendung entwickelt und es ist daher kein Wissen in JavaScript für das *Frontend* notwendig. Das *Frontend* und *Backend* werden auf dem Server in der Programmiersprache C# geschrieben. Der Entwickler muss aus diesem Grund lediglich Wissen in HTML, CSS und C# besitzen (vgl. Schwichtenberg 2020).

Vorteil	**Beschreibung**
Entwicklungszeit	Bei *Blazor Server* ist die Entwicklungszeit schneller als bei der SPA, da nur eine Anwendung entwickelt werden muss und auf die nötige *API* für die Kommunikation wie bei einer SPA verzichtet werden kann (vgl. Schwichtenberg 2020).
Performanz -> initialer Ladezyklus	Bei dem ersten Ladezyklus einer *Blazor Server* Anwendung wird einmalig, ähnlich wie bei der MPA, eine komplette statische HTML-Seite übertragen. Zusätzlich wird hier jedoch initial, zur Ermöglichung der Echtzeit-Kommunikation der Benutzerinteraktionen zwischen *Frontend* und *Backend*, eine JavaScript-Bibliothek von Microsoft übergeben (vgl. Schwichtenberg 2020).
Performanz -> Nachladen neuer Daten	Neue Daten werden am Server geholt und dort im virtuellen *DOM* ausgetauscht. Der geänderte Teil des *DOM*s wird dann über die SignalR-Verbindung an den Client gesendet und auch dort im *DOM* ausgetauscht und neu gerendert. Liegt die Netzwerklatenz eines Nutzers hierbei unter 250 Millisekunden fühlt sich dieser Austausch sehr flüssig an (vgl. Microsoft 2023e).
Skalierbarkeit	Bei Blazor Server übernimmt der Server die Zustandsverwaltung. Eine *vertikale Skalierung* ist problemlos möglich (vgl. Microsoft 2023e). Eine *horizontale Skalierung* ist jedoch aufgrund dessen und der nötigen Verwaltung der SignalR-Verbindungen über mehrere Serverinstanzen hinweg, eine größere Herausforderung als bei der MPA oder SPA (vgl. Microsoft 2023a).

Tabelle 6: Vorteile von *Blazor Server* (eigene Darstellung)

Nachteile

Eine Übersicht über die verschiedenen Nachteile von *Blazor Server* ist in Tabelle 7 aufgelistet.

Nachteil	**Beschreibung**
Browseranforderung	*Blazor Server* erfordert einen modernen Browser, wie z.B. Google Chrome, Firefox, Safari oder Edge, der aktuelles JavaScript unterstützt (vgl. Schwichtenberg 2020).
Erweiterbarkeit	Es wird eine Anwendung entwickelt, in der keine saubere Codetrennung zwischen *Frontend* und *Backend* eingehalten werden muss, wie es bei der MPA oder der SPA der Fall ist. Dies kann die Erweiterbarkeit erschweren, da es möglicherweise unübersichtlich werden kann (vgl. Gingter 2020: 56:30 - 58:30).
Offlinefähigkeit	*Blazor Server* ist nicht offlinefähig, da eine ständige Netzwerkverbindung notwendig ist. Sobald diese unterbrochen ist, stoppt die Anwendung (vgl. Gingter 2020: 46:52 - 47:45).
Serverbelastung	Bei *Blazor Server* liegen alle Nutzer-Sitzungen nebeneinander auf dem Server. Es werden viele Megabyte an Arbeitsspeicher am Server benötigt, da der Zustand dauerhaft gehalten werden muss (vgl. Gingter 2020: 48:35 - 49:42). Somit liegt eine hohe Grundbelastung des Servers vor. Neue Datenanfragen hingegen, belasten den Server nur wenig, da die neuen Daten auf dem Server gesucht werden und die *DOM*-Änderungen zurück an den Client geschickt werden (vgl. Schwichtenberg 2020).

Nachteil	Beschreibung
Sicherheit -> ungewolltes Offenlegen von Daten	Durch die Vermischung von *Frontend* und *Backend*, kann es sein, dass beim Entwicklungsprozess unbeabsichtigt vertrauliche Daten auf dem Client verarbeitet werden, da die Anwendung unübersichtlich werden kann (vgl. Gingter 2020: 56:30 - 58:30).
Suchmaschinen-optimierung (SEO)	Die SEO ist ein großer Nachteil von *Blazor Server*, da wie bei der SPA nur eine HTML-Seite vorliegt. Suchmaschinen bewerten verschiedene Webseiten und das ist bei *Blazor Server* nicht möglich (vgl. Patadiya 2023).

Tabelle 7: Nachteile von *Blazor Server* (eigene Darstellung)

Die Vermischung von *Frontend* und *Backend*, die *Blazor Server* bietet, bringt wie gezeigt eigene Vor- und Nachteile mit. Daher ist dieser Ansatz nicht für jedes Projekt geeignet und muss mit Bedacht eingesetzt werden. Dies wurde im Kriterienkatalog, der in Kapitel 3.1 („Definition der Kriterien anhand der Vor- und Nachteile der vorgestellten Webanwendungsarchitekturen“) aufgezeigt wird, berücksichtigt.

Im nächsten Kapitel (Kap. 2.5: „Vor- und Nachteile der vorgestellten Webanwendungsarchitekturen in der Übersicht“) folgt eine Auflistung aller Vor- und Nachteile der MPA, SPA und Fullstack-Architektur.

2.5 Vor- und Nachteile der vorgestellten Webanwendungsarchitekturen in der Übersicht

Die drei vorgestellten Webanwendungsarchitekturen, nämlich die MPA, die SPA und die Fullstack-Architektur, besitzen jeweils eigene Vor- und Nachteile, welche

in den Kapiteln 2.2 („Die Multi-Page-Architektur im Überblick“), 2.3 („Erklärung der Single-Page-Architektur“) und 2.4 („Erklärung der Fullstack-Architektur ohne direkte Trennung von *Frontend* und *Backend*“) beschrieben wurden. In diesem Kapitel fasst die Tabelle 8 diese Vor- und Nachteile verkürzt und übersichtlich gegenüberstellt zusammen. Die Farben rot und grün werden genutzt, um anhand eines Kriteriums zu zeigen, wie gut oder schlecht der jeweilige Ansatz diesen Punkt erfüllt.

Kriterium	**Multi-Page-Architektur**	**Single-Page-Architektur**	**Fullstack-Architektur (*Blazor Server*)**
Benutzererfahrung	Nachteil: Jede Benutzerinteraktion führt zum neu Laden der gesamten Seite.	Vorteil: Benutzeraktionen führen keine ganzen Seitenaufrufe aus, dies fühlt sich für den Benutzer an wie eine Desktop-Anwendung.	Vorteil: Benutzeraktionen führen keine ganzen Seitenaufrufe aus, dies fühlt sich für den Benutzer an wie eine Desktop-Anwendung.
Browseranforderung	Vorteil: Läuft auf jedem Browser.	Nachteil: Moderne Browser nötig, die JavaScript fähig sind.	Nachteil: Moderne Browser nötig, die JavaScript fähig sind.

Kriterium	**Multi-Page-Architektur**	**Single-Page-Architektur**	**Fullstack-Architektur (*Blazor Server*)**
Entwicklererfahrung	Nachteil: Fachwissen in *Frontend* und *Backend* nötig, aber es wird nur eine Anwendung mit einer Codetrennung entwickelt.	Nachteil: Fachwissen in *Frontend* und *Backend* mit enger Zusammenarbeit (z.B. wegen *Typsysteme* bei Kommunikation) nötig, da zwei Anwendungen mit einer Schnittstelle zur Kommunikation entstehen.	Vorteil: Kein Wissen in JavaScript notwendig. *Frontend* und *Backend* werden zusammen in C# in einer Anwendung entwickelt.
Entwicklungszeit	Vorteil: Schnell, da nur eine Anwendung entwickelt wird.	Nachteil: Langsam, da zwei Anwendungen mit einer *API* für die Kommunikation entwickelt werden müssen.	Vorteil: Schnell, da nur eine Anwendung entwickelt wird.

Kriterium	**Multi-Page-Architektur**	**Single-Page-Architektur**	**Fullstack-Architektur (*Blazor Server*)**
Erweiterbarkeit	Vorteil: Leicht erweiterbar, *Frontend* und *Backend* sind aber etwas stärker verknüpft, da diese in einer Anwendung sind.	Vorteil: Leicht erweiterbar, *Frontend* und *Backend* können isoliert erweitert werden, da zwei Anwendungen vorliegen.	Nachteil: Keine saubere Codetrennung zwischen *Frontend* und *Backend* in der einen Anwendung wird erzwungen, dies erschwert die Erweiterbarkeit.
Offlinefähigkeit	Nachteil: nein	Vorteil: ja	Nachteil: nein
Performanz -> initialer Ladezyklus	Vorteil: schnell	Nachteil: langsam	Vorteil: schnell

Kriterium	**Multi-Page-Architektur**	**Single-Page-Architektur**	**Fullstack-Architektur (*Blazor Server*)**
Performanz -> Nachladen neuer Daten	Nachteil: Neue Datenanfragen fordern immer ganze HTML-Seiten vom Server an.	Vorteil: Neue Datenanfragen stoßen Anfragen an den Server an. Dieser liefert die neuen Daten zurück und daraufhin wird der betroffene Teil auf der Seite neu gerendert.	Vorteil: Neue Datenanfragen werden über die SignalR-Verbindung dem Server bekannt gemacht. Dieser holt die neuen Daten und tauscht diese im virtuellen Abbild des *DOM*s auf dem Server aus. Der geänderte Teil des *DOM*s wird dann an den Client gesendet und auch dort im *DOM* ausgetauscht und angezeigt.

Kriterium	**Multi-Page-Architektur**	**Single-Page-Architektur**	**Fullstack-Architektur (*Blazor Server*)**
Sicherheit -> ungewolltes Offenlegen von Daten	Vorteil: Nicht so anfällig wie die SPA oder Fullstack-Architektur.	Nachteil: Anfällig für bösartiges JavaScript, welches eingeschleust werden kann.	Nachteil: Durch die unübersichtliche Trennung von *Frontend* und *Backend*, kann es vorkommen, dass vertrauliche Daten auf dem Client verarbeitet werden.
Serverbelastung	Nachteil: hoch	Vorteil: niedrig	Nachteil: mittel
Skalierbarkeit	Vorteil: Sehr gut *horizontal* oder *vertikal skalierbar*, kommt auf die Implementierung der Zustandshaltung an.	Vorteil: Sehr gut *horizontal* oder *vertikal skalierbar*, da der Zustand auf dem Client gespeichert wird.	Vorteil: Sehr gut *vertikal skalierbar*. Ebenfalls *horizontal skalierbar*, da der Zustand aber am Server gespeichert wird, etwas schwieriger umzusetzen.
Suchmaschinen-optimierung	Vorteil: gut geeignet	Nachteil: nicht geeignet	Nachteil: nicht geeignet

Tabelle 8: Übersicht der Vor- und Nachteile der MPA, SPA und Fullstack-Architektur (eigene Darstellung)

Diese Übersicht diente als Grundlage für die erste Version für den für diese Arbeit erstellten Kriterienkatalog, der in Kapitel 3.1 („Definition der Kriterien anhand der Vor- und Nachteile der vorgestellten Webanwendungsarchitekturen“) aufgezeigt wird.

2.6 Zusammenfassung

Das zweite Kapitel („Beschreibung der Webanwendungsarchitekturen“) gibt in Kapitel 2.1 („Webanwendungsarchitekturen in der Übersicht“) einen Überblick über die verschiedenen Architekturen (siehe Tabelle 1), die existieren, um eine Webanwendung zu entwickeln. Dabei wurde aufgezeigt, dass sich diese Arbeit auf die Multi-Page-Architektur, die Single-Page-Architektur und die selbst benannte Fullstack-Architektur fokussierte, da die Diskussion der *Makroarchitekturen* im Vordergrund stand. Dabei wurden für alle drei Architekturen die folgenden Merkmale, in unterschiedlicher Reihenfolge, diskutiert:

- Benutzererfahrung
- Browseranforderung
- Entwicklererfahrung
- Entwicklungszeit
- Erweiterbarkeit
- Offlinefähigkeit
- Performanz
- Sicherheit
- Serverbelastung
- Skalierbarkeit

- Suchmaschinenoptimierung (Abk.: SEO)

In Kapitel 2.2 („Die Multi-Page-Architektur im Überblick“) wurde die Multi-Page-Architektur als erste Architektur, die entstanden ist, vorgestellt. Dabei zeigte das Kapitel die Architektur dieser, mit der Besonderheit der mehreren HTML-Seiten, schematisch auf. Außerdem wurde die Trennung in *Frontend* und *Backend* gezeigt. Ebenfalls präsentierte Tabelle 2 die Vorteile der MPA, wie z.B. die Skalierbarkeit, die Erweiterbarkeit oder die Suchmaschinenoptimierung. Neben den Vorteilen beleuchtete Tabelle 3 die Nachteile, wie z.B. die Benutzererfahrung, die Entwicklererfahrung oder die Offlinefähigkeit.

In Kapitel 2.3 („Erklärung der Single-Page-Architektur“) wurde die Single-Page-Architektur, die aufgrund der Entwicklung von JavaScript entstanden ist, vorgestellt. Dabei zeigte das Kapitel die Architektur dieser, mit der Besonderheit nur einer einzigen HTML-Seite, schematisch auf. Außerdem wurde die Trennung in *Frontend* und *Backend* gezeigt. Ebenfalls demonstrierte Tabelle 4 die Vorteile der SPA, wie z.B. die Benutzererfahrung, die Erweiterbarkeit oder die Offlinefähigkeit. Neben den Vorteilen zeigte Tabelle 5 die Nachteile, wie z.B. die Entwicklererfahrung, die Entwicklungszeit oder die Sicherheit, auf.

In Kapitel 2.4 („Erklärung der Fullstack-Architektur ohne direkte Trennung von *Frontend* und *Backend*“) wurde die Fullstack-Architektur, die in dieser Arbeit durch das *Framework Blazor Server* von Microsoft widergespiegelt wurde, vorgestellt. Dabei zeigte das Kapitel die Architektur dieser, mit der Besonderheit, dass alles in der Programmiersprache C# auf dem Server geschrieben wird, schematisch auf. Außerdem wurde die Vermischung von *Frontend* und *Backend* gezeigt, die dieser Ansatz ermöglicht. Ebenfalls präsentierte Tabelle 6 die Vorteile, wie z.B. die Benutzererfahrung, die Entwicklererfahrung oder die Entwicklungszeit. Neben den Vorteilen beleuchtete Tabelle 7 die Nachteile, wie z.B. die Skalierbarkeit, die Erweiterbarkeit oder die Offlinefähigkeit.

Das Kapitel 2.5 („Vor- und Nachteile der vorgestellten Webanwendungsarchitekturen in der Übersicht“) bietet einen Überblick und eine Zusammenfassung über alle diskutierten Vor- und Nachteile der Multi-Page-Architektur, der Single-Page-Architektur und der Fullstack-Architektur. Dabei wurden diese in Tabelle

8 sehr verkürzt und übersichtlich gegenübergestellt. Dazu wurden die Farben rot und grün genutzt, um für ein Kriterium zu dem jeweiligen Ansatz beschreiben zu können, wie gut oder schlecht, dieser diesen Punkt erfüllt.

Kapitel 3

Aufstellung des Kriterienkatalogs zur Ermittlung der passenden Architektur für ein Projekt

Das dritte Kapitel zeigt die Aufstellung der ersten Version des für diese Arbeit erstellten Kriterienkatalogs. Für die erste Version wurden die Vor- und Nachteile der Architekturen, die in Kapitel 2.5 („Vor- und Nachteile der vorgestellten Webanwendungsarchitekturen in der Übersicht“) beschrieben wurden, als Grundlage genutzt. Außerdem wird demonstriert, welches System der Kriterienkatalog beinhaltet, damit für webbasierte Projekte eine passende Architektur abgeleitet werden kann. Das dritte Kapitel zeigt den Schritt 2 im Ablaufplan (siehe Abbildung 2 in Kapitel 1.4: „Beschreibung des Vorgehens“).

3.1 Definition der Kriterien anhand der Vor- und Nachteile der vorgestellten Webanwendungsarchitekturen

In Kapitel 1.1 („Ausgangslage und Unternehmen“) wurde gezeigt, dass eine umfassende und detaillierte Übersicht der Kriterien, die für und gegen eine Webarchitektur sprechen, nicht an einem einheitlichen Punkt in der Literatur beschrieben wurde und dementsprechend nicht leicht zu finden ist. Das aktuelle, vorliegende Kapitel zeigt nachfolgend den Prozess auf, wie diese Kriterien gesammelt und in eine Übersicht in Form eines Kriterienkatalogs für die drei betrachteten Webarchitekturen (Multi-Page-Architektur, Single-Page-Architektur und Fullstack-Architektur) gebracht wurden.

Aufbau des Kriterienkatalogs

Die Kriterien wurden aus der selbst zusammengestellten Übersicht der Vor- und Nachteile der MPA, SPA und Fullstack-Architektur in Tabelle 8 aus Kapitel 2.5 („Vor- und Nachteile der vorgestellten Webanwendungsarchitekturen in der Übersicht“) übernommen. Zusätzlich wurde zu jedem Kriterium eine kleine Kurzbeschreibung hinzugefügt, damit ein Anwender sofort erkennen kann, was sich dahinter verbirgt und diskutiert, wie qualitativ die Architektur dieses Kriterium erfüllt. Diese qualitative Bewertung beruht hierbei auf der durchgeführten Literaturrecherche, die in Kapitel 2 („Beschreibung der Webanwendungsarchitekturen“) beschrieben wurde.

Weiteres Kriterium für den Kriterienkatalog: *Autorisierung* mit OAuth 2.0

Ein weiteres Kriterium, welches zuvor nicht explizit durch die Literaturrecherche behandelt wurde, sondern dessen Problemstellung kurz in Kapitel 1.1 („Ausgangslage und Unternehmen“) angesprochen wurde, wurde dem Kriterienkatalog

hinzugefügt. Konkret handelt es sich dabei um die *Authentifizierung* und die *Autorisierung*. Bei XITASO benötigt jede Webanwendung ein ordentliches Anmelde- und Zugriffsystem, daher wurde dieses Kriterium zusätzlich mit aufgenommen. Es gibt unterschiedliche Verfahren zur Durchführung der *Authentifizierung* und *Autorisierung*, wie z.B. die sogenannte „Basic Authentication“, die sogenannte „Session Based Authentication“ oder die sogenannte „Token Based Authentication“ (vgl. Ackermann 2021: 540 - 544). Ein Standard, der sich mittlerweile bezüglich der „Token Based Authentication“ etabliert hat, ist die tokenbasierte *Authentifizierung* und *Autorisierung* mit „Open Authorization 2.0 (Abk.: OAuth 2.0)“. Dabei wird ein Nutzer auf die OAuth 2.0 Login Seite weitergeleitet und authentifiziert sich dort. Nachdem dies erfolgreich war, erhält er ein sogenanntes *Zugriffstoken*. Dieses muss dann von der aufrufenden Webanwendung überprüft werden, ob es autorisiert ist die Daten anzufragen und dementsprechend die Ansicht anpassen (vgl. Redaktion 2022).
Bei der MPA und der Fullstack-Architektur erfolgt eine solche Implementierung jeweils nur in der einen vorliegenden Webanwendung. Bei der SPA jedoch, muss der Entwickler die *Autorisierung* in beiden Anwendungen (dem getrennten *Frontend* und *Backend*) implementieren, was zusätzlich den Aufwand erhöht.

Ausschnitt aus dem aufgestellten Kriterienkatalog

Der komplett aufgestellte Kriterienkatalog liegt in erster Version aus Platzgründen der Arbeit unter Anhang A („Erste Version des Kriterienkatalogs nach der Literaturrecherche“) bei. In Tabelle 9 werden zwei Kriterien aus der ersten Version des aufgestellten Kriterienkatalogs dargestellt, um diesen kurz zu veranschaulichen.

Kriterium	**Multi-Page-Architektur**	**Single-Page-Architektur**	**Fullstack-Architektur (*Blazor Server*)**
***Autorisierung* mit OAuth 2.0** -> ist die Umsetzung der *Autorisierung* mit OAuth 2.0 in der Architektur einfach?	Die *Autorisierung* muss nur in einer Anwendung umgesetzt werden, da bei der MPA nur eine gebaut wird.	Die *Autorisierung* muss in beiden Anwendungen (dem getrennten *Frontend* und *Backend*) umgesetzt werden. So entsteht ein Mehraufwand.	Die *Autorisierung* muss nur in einer Anwendung umgesetzt werden, da bei *Blazor Server* nur eine gebaut wird.
Benutzererfahrung -> bietet die Architektur eine angenehme Benutzererfahrung beim Laden neuer Inhalte?	Jede Benutzerinteraktion führt zum neuen Laden der gesamten Seite, welche die neuen Inhalte dann bereitstellt.	Benutzerinteraktionen lösen keine ganzen Seitenaufrufe mehr aus und es können Animationen genutzt werden, beim Warten auf neue Inhalte.	Benutzerinteraktionen lösen keine ganzen Seitenaufrufe mehr aus und es können Animationen genutzt werden, beim Warten auf neue Inhalte.

Tabelle 9: Ausschnitt aus dem Kriterienkatalog (eigene Darstellung)

Das Aufstellen der Kriterien allein reicht aber nicht aus, um zu einer besseren Entscheidungsfindung beizutragen, da so keine Webarchitektur als am Besten ge-

eignet für eine Problemstellung ermittelt werden kann. Daher wird im nächsten Kapitel (Kap. 3.2: „Erläuterung der Ableitung auf die Architekturen“) aufgezeigt, welches System in den Kriterienkatalog eingebaut wurde, um eine Webarchitektur, auf Grundlage der vorliegenden Problemstellung, abzuleiten.

3.2 Erläuterung der Ableitung auf die Architekturen

Im vorangehenden Kapitel (Kap. 3.1: „Aufstellung des Kriterienkatalogs zur Ermittlung der passenden Architektur für ein Projekt“) wurde der Aufbau der ersten Version des aufgestellten Kriterienkatalogs vorgestellt. Allerdings fehlt zu dieser Aufstellung noch ein System, welches es ermöglicht, eine Architektur abzuleiten, um zur Entscheidungsfindung beizutragen. Das eigens dafür entwickelte System wird nachfolgend vorgestellt.

Darstellung des Punktesystems des Kriterienkatalogs

Jede Architektur erfüllt die aufgestellten Kriterien anders. Auf Grundlage der Literaturrecherche und der Vor- und Nachteile der einzelnen Architekturen, wurde eine Punktzahl abgeleitet. Dabei existieren drei verschiedene Werte:

- 6 Punkte: Die Architektur erfüllt das Kriterium vollständig.
- 3 Punkte: Die Architektur erfüllt das Kriterium nur teilweise.
- 1 Punkt: Die Architektur erfüllt das Kriterium überhaupt nicht bzw. sehr schlecht.

Diese Werte wurden entsprechend für die Architekturen vergeben. Außerdem wurden in den Kriterienkatalog zwei weitere Spalten eingefügt. Eine mit dem Titel „Gewollt?“ und eine mit dem Titel „Gewicht“. In Tabelle 10 ist ein Ausschnitt aus

dem Kriterienkatalog abgebildet, welcher vereinfacht wurde, um das Punktesystem zu verdeutlichen.

Kriterium	**Multi-Page-Architektur**	**Single-Page-Architektur**	**Fullstack-Architektur (*Blazor Server*)**	**Gewicht**	**Gewollt?**
***Autorisierung* mit OAuth 2.0** -> ...	... -> Punktzahl: 6	... -> Punktzahl: 1	... -> Punktzahl: 6	Gewicht: MPA: SPA: Fullstack:	
Erweiterbarkeit -> ...	... -> Punktzahl: 6	... -> Punktzahl: 6	... -> Punktzahl: 3	Gewicht: MPA: SPA: Fullstack:	

Tabelle 10: Vereinfachte Darstellung des Systems des Kriterienkatalogs (eigene Darstellung)

In der Spalte „Gewollt?“ kann ein Anwender des Kriterienkatalogs ankreuzen, ob das jeweilige Kriterium für seine eigene architektonische Problemstellung wichtig ist. So wird eine Ableitung genauer, da Kriterien, die möglicherweise schlechter erfüllt, aber nicht notwendig sind, nicht mit in die Wertung einfließen. Außerdem hat ein Anwender in der Spalte „Gewicht“ die Möglichkeit, für ein gewolltes Kriterium, ein zur eigenen Problemstellung passendes Gewicht zu vergeben. Dabei muss beachtet werden, dass alle gewollten Kriterien zusammen, ein Gewicht von 100% haben müssen. Wichtig zu erwähnen ist, dass das Punktesystem für alle Versionen des Kriterienkatalogs gleich ist und jeweils nur bei Bedarf die vergebenen Punkte und Beschreibungen sich in den verschiedenen Versionen veränderten.

Darstellung der Funktionsweise des Kriterienkatalogs

Nach der Aufstellung der ersten Version des Kriterienkatalogs und der Darstellung des Punktesystems ist dieser einsatzbereit. Die genaue Funktionsweise wird nachfolgend in Abbildung 6 schrittweise dargestellt:

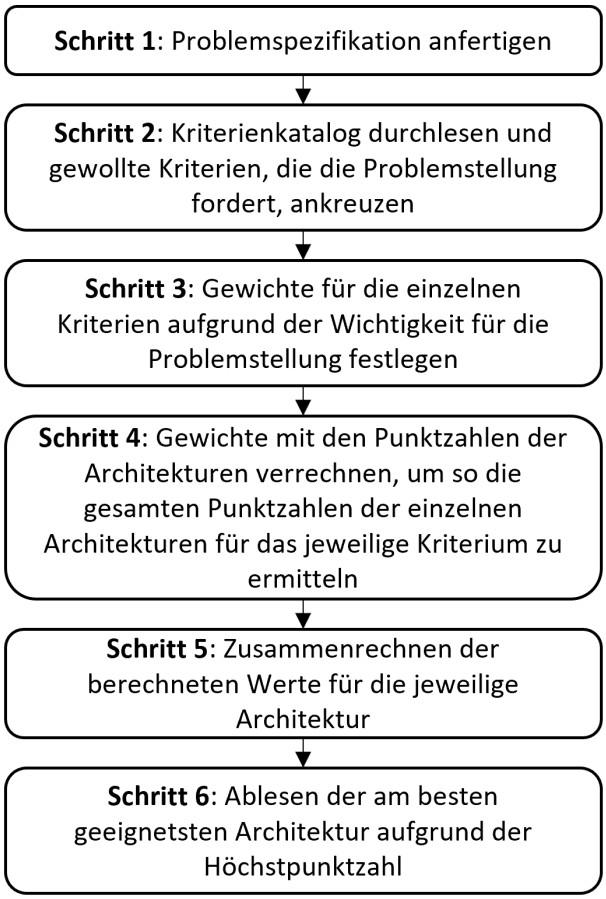


Abbildung 6: Funktionsweise des Kriterienkatalogs (eigene Darstellung)

Ein konkretes Beispiel für die Funktionsweise des Kriterienkatalogs wird in Kapitel 5.1 („Erklärung der Anforderungen an die Experimente“) aufgezeigt. Es wurde ein Anwendungsfall definiert und für diesen die zweite Version des Kriterienkatalogs angewandt. Die Funktionsweise ist hierbei in allen Versionen des Kriterienkatalogs dieselbe. Die ausgefüllte zweite Version des Kriterienkatalogs dazu,

liegt der Arbeit unter Anhang D („Ausgefüllte zweite Version des Kriterienkatalogs anhand des Beispielfalls für die Experimente“) bei.

Nachdem die Aufstellung der ersten Version des Kriterienkatalogs und dessen Funktionsweise erklärt wurde, folgt im nächsten Kapitel (Kap. 3.3: „Darstellung der Vorteile durch den Kriterienkatalog“) eine Auflistung der verschiedenen Vorteile, die sich durch den Einsatz des Katalogs, in egal welcher Version, für einen Anwender ergeben.

3.3 Darstellung der Vorteile durch den Kriterienkatalog

In den vorangehenden Kapiteln wurde die Aufstellung der erste Version des Kriterienkatalogs und die Funktionsweise dessen erläutert. Durch den Einsatz von diesem Kriterienkatalog ergeben sich verschiedene Vorteile für einen Anwender. Diese werden nachfolgend in Tabelle 11 aufgezeigt.

Vorteil	**Beschreibung**
Effizienz	Der Kriterienkatalog hilft dabei den Entscheidungsprozess effizient zu gestalten, da er eine klare Struktur vorgibt. Somit können unnötige Diskussionen über irrelevante Aspekte vermieden werden.

Vorteil	**Beschreibung**
Objektivität	Der Kriterienkatalog wurde anhand einer Literaturrecherche (siehe Kapitel 2: „Beschreibung der Webanwendungsarchitekturen“), einer Expertenbefragung (siehe Kapitel 4: „Evaluation des Kriterienkatalogs mit Experteninterviews“) und von Experimenten (siehe Kapitel 5: „Erläuterung der durchgeführten Experimente“ und Kapitel 6: „Ergebnis der drei Experimente“) entwickelt. Er schafft eine objektive Basis für Entscheidungen und minimiert eine mögliche Voreingenommenheit.
Strukturierte Entscheidungsfindung	Der Kriterienkatalog bietet eine strukturierte Herangehensweise an die Entscheidungsfindung. Er stellt sicher, dass wichtige Aspekte berücksichtigt werden und unterstützt einen systematischen Entscheidungsprozess.
Transparenz	Die Verwendung des Kriterienkatalogs macht den Entscheidungsprozess transparent für alle beteiligten Parteien. Jeder kann leicht nachvollziehen, welche Kriterien genau verwendet und wie diese gewichtet wurden.
Vergleichbarkeit	Durch die Bewertung anhand der gleichen Kriterien wird ein direkter Vergleich zwischen den verschiedenen Architekturen ermöglicht. Dabei werden die Stärken und Schwächen dieser leicht identifiziert.

Vorteil	Beschreibung
Zeitersparnis	Die beteiligten Parteien müssen sich nicht selbst auf die Suche nach Kriterien für die Entscheidungsfindung machen und diese für die unterschiedlichen Ansätze bewerten. Der Kriterienkatalog bietet eine zentrale Sammlung dieser Aspekte und spart einsetzenden Parteien viel Zeit.

Tabelle 11: Vorteile des Kriterienkatalogs (eigene Darstellung)

All diese Vorteile sprechen dafür, dass jeder Informatiker oder Software-Architektur-Interessierte, der ein neues Webprojekt zu bewältigen hat, diesen Kriterienkatalog einsetzen sollte, um sich die Arbeit zu erleichtern. Wichtig zu erwähnen ist, dass die erste Version des Kriterienkatalogs gemäß dem Ablaufplan (siehe Abbildung 2 in Kapitel 1.4: „Beschreibung des Vorgehens“) weiterentwickelt wurde. Diese Entwicklung wird im Laufe der Arbeit dargestellt. Die Vorteile bleiben jedoch gleich und gelten für alle Versionen des Kriterienkatalogs.

3.4 Zusammenfassung

Das dritte Kapitel („Aufstellung des Kriterienkatalogs zur Ermittlung der passenden Architektur für ein Projekt“) zeigte die Aufstellung des für diese Arbeit entstandenen Kriterienkatalog. Dabei wurde der Aufbau, die Struktur und die Berechnungsmodalitäten von diesem dargestellt.

In Kapitel 3.1 („Definition der Kriterien anhand der Vor- und Nachteile der vorgestellten Webanwendungsarchitekturen“) wurde erläutert, dass die aufgestellte Tabelle (Tabelle 8) aus Kapitel 2.5 („Vor- und Nachteile der vorgestellten Webanwendungsarchitekturen in der Übersicht“) die Grundlage für die erste Version für den für diese Arbeit aufgestellten Kriterienkatalog bildete. Dabei wurden alle Kriterien übernommen und um eine Kurzbeschreibung erweitert, um die Verständlichkeit zu erhöhen. Zusätzlich wurde der Kriterienkatalog um das Kriterium „*Autorisierung* mit OAuth 2.0“ erweitert. OAuth 2.0 ist heute ein Standard für

Anmelde- und Zugriffsverwaltung und diese ist in Webanwendungen sehr wichtig. Außerdem erhielten die untersuchten Architekturen eine Spalte zur Einschätzung, wie gut oder schlecht, sie das jeweilige Kriterium erfüllen. Dies ist in einem Ausschnitt aus dem Kriterienkatalog in Tabelle 9 zu sehen oder direkt der ersten Version des beiliegendem Kriterienkatalogs zu entnehmen (siehe Anhang A: „Erste Version des Kriterienkatalogs nach der Literaturrecherche“).

In Kapitel 3.2 („Erläuterung der Ableitung auf die Architekturen“) wurde das System hinter dem Kriterienkatalog gezeigt, welches dafür sorgt, dass nach dem ausfüllen eine Architektur abgeleitet werden kann. Dabei wurde erläutert, dass jede der drei betrachteten Architekturen für jedes aufgestellte Kriterium eine Punktzahl, die sich aus der Diskussion der Literatur ergeben hat, erhalten hat. Dazu wurden drei verschiedene Werte genutzt:

- 6 Punkte: Die Architektur erfüllt das Kriterium vollständig.
- 3 Punkte: Die Architektur erfüllt das Kriterium nur teilweise.
- 1 Punkt: Die Architektur erfüllt das Kriterium überhaupt nicht bzw. sehr schlecht.

Außerdem stellte das Kapitel dar, dass der Kriterienkatalog um zwei Spalten erweitert wurde. Einmal um die Spalte „Gewicht“, in der ein Anwender angeben kann, wie wichtig das Kriterium für seine eigene Problemstellung ist und einmal um die Spalte „Gewollt“. In dieser kann ein Anwender entscheiden, ob das Kriterium überhaupt wichtig ist für seine eigene Problemstellung oder weggelassen werden sollte für die Ableitung einer Architektur. Dies ist in einem vereinfachten Ausschnitt aus der ersten Version des Kriterienkatalogs in Tabelle 10 oder direkt in dem beiliegendem Kriterienkatalog (siehe Anhang A: „Erste Version des Kriterienkatalogs nach der Literaturrecherche“) zu sehen. Außerdem wurde die Funktionsweise des Kriterienkatalogs schrittweise in Abbildung 6 aufgezeigt und auf die konkrete Anwendung dessen in Kapitel 5.1 („Erklärung der Anforderungen an die Experimente“) verwiesen. Des Weiteren wurde erwähnt, dass das Punktesystem und die Funktionsweise für alle Versionen des Kriterienkatalogs gelten.

In Kapitel 3.3 („Darstellung der Vorteile durch den Kriterienkatalog“) wurden die Vorteile des für diese Arbeit entwickelten Kriterienkatalogs dargestellt. Dabei hat das Kapitel Aspekte wie Effizienz, Transparenz und Zeitersparnis aufgezeigt, die durch den Kriterienkatalog verbessert werden. Ebenfalls wurde erwähnt, dass jeder Informatiker oder Software-Architektur-Interessierte, der ein neues Webprojekt zu bewältigen hat, diesen Kriterienkatalog einsetzen sollte, um sich die Arbeit zu erleichtern. Des Weiteren wurde darauf hingewiesen, dass die Vorteile nicht nur für die erste Version des Kriterienkatalogs gelten, sondern für alle Versionen.

Kapitel 4

Evaluation des Kriterienkatalogs mit Experteninterviews

Das vierte Kapitel zeigt die durchgeführte praxisorientierte Qualitätsprüfung des Kriterienkatalogs anhand von Experteninterviews bei der Firma XITASO auf. Der Ablauf des Kapitels ist wie folgt:

- Im ersten Schritt wird die Notwendigkeit der Experteninterviews betont. Dabei wird der Handlungsablauf der Interviews aufgezeigt. Außerdem wird begründet, wieso die Experten von XITASO stammten.
- Im zweiten Schritt werden die Ergebnisse der Experteninterviews tabellarisch dargestellt. Dabei werden nur die Ergebnisse dargestellt, die einen Einfluss auf die Verbesserung des Kriterienkatalogs hatten.
- Im dritten Schritt wird gezeigt, wie die zuvor dargestellten relevanten Ergebnisse genutzt wurden, um den Kriterienkatalog anzupassen und zu verbessern. Dadurch ist eine neue verbesserte Version von diesem entstanden, welcher der Arbeit unter Anhang C („Zweite Version des Kriterienkatalogs nach den Experteninterviews“) beiliegt.

Das vierte Kapitel zeigt die Schritte 3 und 4 im Ablaufplan (siehe Abbildung 2 in Kapitel 1.4: „Beschreibung des Vorgehens“).

4.1 Darstellung des Ablaufs der Experteninterviews

In Kapitel 3.2 („Erläuterung der Ableitung auf die Architekturen“) wurde das entwickelte Punktesystem für den Kriterienkatalog vorgestellt. Hierbei wurde betont, dass die Punktevergabe auf einer fundierten Literaturrecherche sowie der Identifikation von Vor- und Nachteilen der Architekturen basierte. Um die Praxisrelevanz der vergebenen Punkte zu steigern und somit die Aussagekraft des Katalogs zu erhöhen, wurde eine praxisorientierte Qualitätsprüfung durch Experteninterviews durchgeführt. In diesem Zusammenhang wurden drei verschiedene Mitglieder der Software Engineering-*Community* von XITASO befragt, da diese zukünftig hauptsächlich mit dem Kriterienkatalog arbeiten werden. Konkret wurden Interviews mit Herrn Hirschmüller, Herrn Winkelmann und Herrn Brunner geführt. Vor der Vorstellung des Ablaufs der Experteninterviews bietet Tabelle 12 eine kurze Übersicht über die Experten und ihre jeweilige Expertise, um die Auswahl dieser zu begründen:

Experte	**Allgemeine Informationen**	**Mitgebrachte Expertise**
Josef Hirschmüller	Firma: XITASO GmbH Adresse: Austraße 35 Ort: 86153 Augsburg Position: Softwareentwickler E-Mail: josef.hirschmueller @xitaso.com	- seit 2017 im Webbereich tätig - 15 (20% MPA und 80% SPA) Webprojekte umgesetzt - Branchen: Maschinenbau - seit 2020 in Kontakt mit *Blazor Server* - bereits Einblicke in *Blazor Server* Projekte bekommen

Experte	Allgemeine Informationen	Mitgebrachte Expertise
Maximilian Winkelmann	Firma: XITASO GmbH Adresse: Austraße 35 Ort: 86153 Augsburg Position: Softwareentwickler E-Mail: maximilian.winkelmann@xitaso.com	- seit 2016 im Webbereich tätig - 10 (30% MPA und 70% SPA) Webprojekte umgesetzt - Branchen: Maschinenbau, Medizintechnik, Logistik - seit 2017 in Kontakt mit *Blazor Server* - bereits ein Projekt mit *Blazor Server* umgesetzt
Michael Brunner	Firma: XITASO GmbH Adresse: Austraße 35 Ort: 86153 Augsburg Position: Softwareentwickler E-Mail: michael.brunner@xitaso.com	- seit 2013 im Webbereich tätig - 7 (15% MPA und 85% SPA) Webprojekte umgesetzt - Branchen: Maschinenbau, E-Commerce, Geräte-Hersteller - seit 2019 in Kontakt mit *Blazor Server* - bereits vier Projekte mit *Blazor Server* umgesetzt

Tabelle 12: Beschreibung der Experten (eigene Darstellung)

Die Struktur jedes Gesprächs orientierte sich an einem im Voraus definierten Handlungsablauf, welcher ausführlich im Anhang B („Handlungsablauf der Experteninterviews“) der Arbeit dargelegt ist.
Der Einstieg jedes Experteninterviews ist in Abbildung 7 visualisiert:

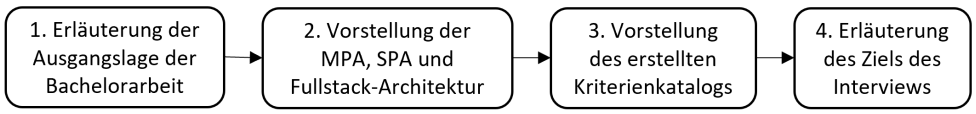


Abbildung 7: Ablauf des einführenden Teils der Experteninterviews (eigene Darstellung)

Nach Abschluss des einführenden Teils war jeder Experte dazu aufgefordert, vordefinierte Aussagen zu den verschiedenen Kriterien für jede Architektur anhand einer *Likert-Skala* zu bewerten.

Aufbau der *Likert-Skala*

Eine *Likert-Skala* stellt eine Methode zur Bewertung von Meinungen oder Einstellungen dar, bei der die Teilnehmer ihre Zustimmung oder Ablehnung zu Aussagen auf einer gestuften Skala ausdrücken (vgl. Prof. Dr. Klaus Wübbenhorst 2018). In Tabelle 13 ist ein Ausschnitt aus der erstellten Tabelle zu sehen, der die Umsetzung der *Likert-Skala* veranschaulicht. Die erste Spalte präsentiert das Kriterium mit einer kurzen Beschreibung in Form eines Statements. Die fünf folgenden Spalten boten dem Experten fünf unterschiedliche Möglichkeiten das Statement zu bewerten (von „Stimme überhaupt nicht zu“ bis „Stimme voll und ganz zu“). Pro Kriterium und Zeile durfte der Experte hierbei nur eine Bewertung vornehmen.

Kriterium	**Stimme überhaupt nicht zu**	**Stimme nicht zu**	**Stimme weder zu noch lehne ich ab**	**Stimme zu**	**Stimme voll und ganz zu**
Performanz -> Die Anwendung lädt initial schnell.					

Tabelle 13: Ausschnitt der *Likert-Skala* für die Experteninterviews (eigene Darstellung)

Die erstellte Tabelle wurde jeweils für jede der drei zu analysierenden Architekturen mit jedem Experten durchgeführt. Dies geschah einmal zur Bewertung der Statements für die Multi-Page-Architektur, einmal für die Single-Page-Architektur und einmal für die Fullstack-Architektur. Um die Auswertung der Fragebögen zu ermöglichen, wurde jeder Bewertungsmöglichkeit eine Punktzahl zugeordnet:

- 1 Punkt: Stimme überhaupt nicht zu
- 2 Punkte: Stimme nicht zu
- 3 Punkte: Stimme weder zu noch lehne ich ab
- 4 Punkte: Stimme zu
- 5 Punkte: Stimme voll und ganz zu

Diese Punktzuordnung ermöglichte die numerische Zusammenfassung der in einem Interview gegebenen Antworten. Durch die Summierung und anschließende Division durch die Anzahl der geführten Interviews entstanden Durchschnittswerte, die die Tendenz der Antworten widerspiegelten.

Aufbau des restlichen Experteninterviews

Nachdem die Experten die *Likert-Skala* für alle drei Architekturen ausgefüllt hatten, wurden ihnen Fragen zu ausgewählten Kriterien gestellt. Dabei wurden architektonische Aspekte abgefragt, die die Kriterien selbst hinterfragten und beeinflussen konnten. Nachfolgend wird eine von insgesamt sechs Fragen, wie sie im beigefügten Handlungsablauf (siehe Anhang B: „Handlungsablauf der Experteninterviews“) aufgeführt sind, exemplarisch präsentiert. Ziel ist es, die Auswirkungen der gestellten Fragen auf die Kriterien kurz zu verdeutlichen:

- Wie oft kommt es in Ihren Augen vor, dass in einem Projekt keine Authentifizierung und Autorisierung benötigt wird? (Einfluss auf Relevanz von Kriterium: Autorisierung mit OAuth 2.0)

Nach der Beantwortung dieser Fragen hatte der befragte Experte im abschließenden Schritt die Gelegenheit, Feedback zum durchgeführten Interview zu geben. Dadurch konnten nicht abgedeckte Aspekte sowie zusätzliche Meinungen über den Kriterienkatalog eingefangen werden.

Im nächsten Kapitel (Kap. 4.2: „Ergebnisse der durchgeführten Experteninterviews“) werden die für diese Arbeit relevanten Ergebnisse nach der Durchführung der drei Experteninterviews zusammengefasst und dargestellt.

4.2 Ergebnisse der durchgeführten Experteninterviews

Die praxisorientierten Experteninterviews wurden mit drei Experten aus der beauftragenden Firma XITASO durchgeführt, wie bereits im vorangegangenen Kapitel (Kap. 4.1: „Darstellung des Ablaufs der Experteninterviews“) kurz vorgestellt. Im Folgenden werden die relevanten Ergebnisse der drei Experteninterviews zusammengefasst und präsentiert, welche einen Einfluss auf den Kriterienkatalog

hatten. Nicht aufgeführte Ergebnisse der Experteninterviews hatten keinen Einfluss auf die Optimierung des Kriterienkatalogs, da sie mit bereits integrierten Elementen und Informationen übereinstimmten. Daher werden diese nicht explizit erwähnt. Die im folgenden dargestellten relevanten Erkenntnisse werden in Kapitel 4.3 („Einarbeitung der Experteninterview-Ergebnisse in den Kriterienkatalog“) aufgegriffen, um zu zeigen, wie diese in die zweite, verbesserte Version des Kriterienkatalogs eingearbeitet wurden.

Darstellung der relevanten Ergebnisse der *Likert-Skala*

Nach der Bildung von Durchschnittswerten für die Antworten der Experten auf die Fragebögen erfolgte ein Vergleich mit den vergebenen Punkten in der ersten Version des Kriterienkatalogs nach der Literaturrecherche. Dabei wurde eine Abweichung in zwei Kriterien festgestellt. Die übrigen berechneten Werte entsprachen den zuvor vergebenen Punkten für die Kriterien der Architekturen. Aus diesem Grund werden diese nicht separat aufgeführt, da ihre Ergebnisse keinen Einfluss auf die Optimierung des Kriterienkatalogs hatten. Tabelle 14 zeigt die relevanten berechneten Durchschnittswerte für die beiden Kriterien, wobei der Rechenweg zusätzlich dargestellt wird.

Kriterium mit Statement	**Multi-Page-Architektur** Durchschnittswert	**Single-Page-Architektur** Durchschnittswert	**Fullstack-Architektur (*Blazor Server*)** Durchschnittswert
***Autorisierung* mit OAuth 2.0** -> Die Umsetzung der *Autorisierung* mit OAuth 2.0 ist mit der Architektur einfach.	(5+4+5) / 3 = 4,67	(3+5+4) / 3 = 4	(5+4+5) / 2 = 4,67

Kriterium mit Statement	**Multi-Page-Architektur** Durch-schnittswert	**Single-Page-Architektur** Durch-schnittswert	**Fullstack-Architektur (*Blazor Server*)** Durch-schnittswert
Serverbelastung -> Der Server wird nicht stark belastet.	(2+2+2) / 3 = 2	(4+3+2) / 3 = 3	(3+4+2) / 3 = 3

Tabelle 14: Darstellung der relevanten Ergebnisse der *Likert-Skala* der durchgeführten Experteninterviews (eigene Darstellung)

Die in diesem Kapitel berechneten Werte werden im nächsten Kapitel (Kap. 4.3: „Einarbeitung der Experteninterview-Ergebnisse in den Kriterienkatalog“) aufgegriffen, um zu zeigen, wie diese den Kriterienkatalog optimiert haben.

Auswertung der anschließend gestellten konkreten Fragen an die Experten

Die sechs konkret gestellten Fragen können dem beiliegenden Handlungsablauf der Experteninterviews (siehe Anhang B: „Handlungsablauf der Experteninterviews“) entnommen werden. Die Antworten der drei Befragten stimmten bei 5 von 6 Fragen bereits mit den integrierten Elementen in der ersten Version des Kriterienkatalogs überein. Daher werden diese Ergebnisse nicht separat aufgeführt, da diese nicht zu einer Veränderung des Kriterienkatalogs geführt haben. Die Antwort von Herrn Brunner auf die zweite Frage (Frage 2 lautete: „Gibt es in Ihren Augen noch andere Aspekte der jeweiligen Architekturen, die Einfluss auf eine angenehme „Benutzererfahrung“ haben?“) spielte jedoch eine Rolle. Der Experte betonte, dass für eine angenehme Benutzererfahrung auch das Verhalten der Browsertasten (z.B. vor- und zurück-Schaltflächen) bei einer Architektur berück-

sichtigt werden sollte. Dieser Aspekt wird im nächsten Kapitel (Kap. 4.3: „Einarbeitung der Experteninterview-Ergebnisse in den Kriterienkatalog“) aufgegriffen, um zu zeigen, wie dieser in die zweite, verbesserte Version des Kriterienkatalogs integriert wurde.

Neben dem Ausfüllen der Fragebögen und den spezifischen Fragen zur Erweiterung der Kriterien hatten die Experten die Gelegenheit, Feedback zum Aufbau des Kriterienkatalogs zu geben. Dabei haben alle drei Experten betont, dass die *Autorisierung* mit OAuth 2.0 in der SPA zwar aufwändig, aber einfach ist. Herr Winkelmann erwähnte außerdem, dass die Serverbelastung bei der SPA nur gering ist, wenn *overfetching* und *underfetching* vermieden werden. Diese Informationen werden im nächsten Kapitel (Kap. 4.3: „Einarbeitung der Experteninterview-Ergebnisse in den Kriterienkatalog“) aufgegriffen, um zu zeigen, wie diese in der zweiten, verbesserten Version des Kriterienkatalogs integriert wurden.
Ein weiterer Aspekt, der von allen drei Experten hervorgehoben wurde und in Kapitel 7.1 („Interpretation der Ergebnisse“) behandelt wird, bezieht sich darauf, dass sie alle ein großes Interesse daran zeigen, den Kriterienkatalog in der praktischen Anwendung zu erleben. Die Experten bewerteten die Idee äußerst positiv und teilten mit, dass sie bereits bestehende Instrumente nutzen, um zu Beginn eines Webprojekts eine geeignete Architektur auszuwählen. Trotzdem betonten sie, dass der Kriterienkatalog einen signifikanten Mehrwert bieten würde, indem er Informationen zentral sammelt und die Möglichkeit bietet, eine Diskussionsgrundlage zu schaffen.

Im folgenden Kapitel (Kap. 4.3: „Einarbeitung der Experteninterview-Ergebnisse in den Kriterienkatalog“) werden die hier präsentierten relevanten Ergebnisse der Experteninterviews aufgegriffen, um zu zeigen, wie diese Erkenntnisse in die zweite Version des Kriterienkatalogs integriert wurden, um dessen Qualität und Aussagekraft zu verbessern.

4.3 Einarbeitung der Experteninterview-Ergebnisse in den Kriterienkatalog

Die relevanten Ergebnisse der drei durchgeführten Experteninterviews wurden im vorangehenden Kapitel (Kap. 4.2: „Ergebnisse der durchgeführten Experteninterviews“) zusammengefasst. In diesem Kapitel werden diese Ergebnisse nun aufgegriffen, um zu zeigen, wie diese in den Kriterienkatalog eingearbeitet wurden, um eine zweite, verbesserte Version nach der praxisorientierten Qualitätsprüfung zu erstellen. Diese zweite Version liegt der Arbeit im Anhang C („Zweite Version des Kriterienkatalogs nach den Experteninterviews“) bei. Alle Änderungen, die im Folgenden beschrieben werden, sind in der neuen verbesserten Version gelb markiert.

Einarbeitung der Ergebnisse der *Likert-Skala*

Die Antworten der Experten auf den Fragebogen wurden in Tabelle 14 ausgewertet und übersichtlich mit Durchschnittswerten dargestellt. Im Folgenden werden diese Durchschnittswerte unter Berücksichtigung des im Kapitel 4.2 („Ergebnisse der durchgeführten Experteninterviews“) präsentierten Feedbacks der Experten aufgriffen, um zu zeigen, wie diese Informationen genutzt wurden, um den Kriterienkatalog zu verbessern:

- ***Autorisierung* mit OAuth 2.0**: Die Auswertung der Likert-Skala ergab für die Architekturen 4,67 (MPA), 4 (SPA) und 4,67 (*Blazor Server*). Alle Experten betonten, dass die *Autorisierung* mit OAuth 2.0 bei der SPA zwar aufwändiger ist, jedoch trotzdem als einfach bewertet wurde. Aufgrund dieser Einschätzung wurde die Punktzahl der SPA in diesem Kriterium auf 3 festgelegt.
- **Serverbelastung**: Herr Winkelmann wies darauf hin, dass die Serverbelastung bei der SPA in der Regel niedrig ist. Dies gilt jedoch nur, wenn *overfetching* und *underfetching* vermieden werden, da andernfalls der Server

unnötig belastet wird. Daher wurde die Beschreibung der SPA im Kriterium „Serverbelastung“ erweitert, um Anwender auf die möglichen Probleme durch *overfetching* und *underfetching* hinzuweisen. Gleichzeitig wurde die Punktzahl der SPA von 6 auf 3 reduziert.

Einarbeitung der Antworten auf die konkreten Fragen

Die Antworten der Experten auf 5 von 6 Fragen deckten sich bereits mit der bestehenden ersten Version des Kriterienkatalogs und sind daher bereits im Katalog abgedeckt. Allerdings hatte die Antwort von Herrn Brunner auf Frage 2 (Frage 2 lautete: „Gibt es in Ihren Augen noch andere Aspekte der jeweiligen Architekturen, die Einfluss auf eine angenehme „Benutzererfahrung“ haben?“) einen Einfluss, der nachfolgend erläutert wird und zu einer Erweiterung des Kriteriums „Benutzererfahrung“ führte:

- **Frage 2**: Herr Brunner betonte, dass das Verhalten der Browsertasten, wie beispielsweise der vor- und zurück-Schaltflächen, einen Einfluss auf das Kriterium „Benutzererfahrung“ hat und daher berücksichtigt werden sollte. Das Kriterium „Benutzererfahrung“ wurde deshalb erweitert, um das Verhalten der Architektur bei der Betätigung von Browsertasten (z.B. Betätigung der vor- und zurück-Schaltfläche) zu berücksichtigen, da dieser Aspekt in allen drei Architekturen unterschiedlich gehandhabt wird. Hierbei ergibt sich für die drei diskutierten Architekturen folgende Darstellung:
  - **MPA**: Bei der MPA existieren viele HTML-Seiten, wodurch der Browser automatisch sicherstellt, dass die verschiedenen Browsertasten, wie z.B. die Vorwärts- und Rückwärts-Schaltflächen, ihre Funktionen richtig erfüllen. In der Regel muss sich der Entwickler hierbei keine Gedanken machen (vgl. Harris 2021: 7:10 - 7:20; Patadiya 2023).
  - **SPA**: In der SPA erfolgt die Navigation zwischen verschiedenen Ansichten mit JavaScript auf einer Seite. Daher kann das Standardverhalten des Browsers nicht vollständig genutzt werden und ein Entwickler

muss dies in der SPA selbst implementieren. Wenn dies nicht berücksichtigt wird, kann ein Nutzer die letzte Position auf der Seite verlieren, wenn er über die Zurück-Schaltfläche des Browsers zur vorherigen Seite geht (vgl. Harris 2021: 4:05 - 4:45; Patadiya 2023).

- ***Blazor Server***: Bei *Blazor Server* muss der Entwickler, ähnlich wie bei der SPA, aktiv werden, um sicherzustellen, dass die Anwendung mit dem Standardverhalten des Browsers umgehen kann. Allerdings bietet die *.NET-Umgebung* eine Klasse namens „NavigationManager“, die die Umsetzung dieses Aspekts erleichtert (vgl. Hilton 2023).

Nach der Integration der Interview-Ergebnisse in den Kriterienkatalog entstand die zweite Version von diesem. Diese ist der Arbeit aus Platzgründen im Anhang C („Zweite Version des Kriterienkatalogs nach den Experteninterviews“) beigefügt. Es ist von besonderer Bedeutung zu betonen, dass diese Version des Kriterienkatalogs, dank der Experteninterviews, eine höhere Aussagekraft und Praxisrelevanz aufweist und sich nicht ausschließlich auf eine Literaturrecherche stützt. Die zusätzlichen Vorteile, die der Kriterienkatalog bietet, wurden bereits in Kapitel 3.3 („Darstellung der Vorteile durch den Kriterienkatalog“) detailliert erläutert.

Im Kapitel 5.1 („Erklärung der Anforderungen an die Experimente“) wird gezeigt, dass diese Version des Katalogs auf die Probe gestellt wurde, indem ein Anwendungsfall definiert wurde, für den schließlich Experimente in allen drei Architekturen durchgeführt wurden.

4.4 Zusammenfassung

Im Kapitel 4.1 („Darstellung des Ablaufs der Experteninterviews“) wurde herausgestellt, dass die erste Version des Kriterienkatalogs auf Grundlage der erörterten Vor- und Nachteile der durchgeführten Literaturrecherche entwickelt wurde. Eine praxisorientierte Qualitätsprüfung erfolgte durch Experteninterviews, um die erste Version des Kriterienkatalogs zu evaluieren und zu verbessern. Die Interviews

wurden mit drei ausgewählten Mitgliedern der Software Engineering-*Community* von XITASO durchgeführt, da diese zukünftig hauptsächlich mit dem Kriterienkatalog arbeiten werden. Konkret wurden Interviews mit Herrn Hirschmüller, Herrn Winkelmann und Herrn Brunner durchgeführt, deren Expertise Tabelle 12 präsentierte. Der Handlungsablauf der Experteninterviews ist im Anhang B („Handlungsablauf der Experteninterviews“) zu finden, wobei auch die verwendete *Likert-Skala* für die Interviews erläutert wurde. Zusätzlich wurden die spezifisch gestellten Fragen aufgezeigt, da diese architektonische Aspekte der Kriterien beleuchtet und deren Relevanz überprüft haben.

In Kapitel 4.2 („Ergebnisse der durchgeführten Experteninterviews“) wurden die relevanten Ergebnisse der drei Experteninterviews kompakt präsentiert. Es ist wichtig zu erwähnen, dass die Ergebnisse, die sich mit den bereits integrierten Elementen und Informationen deckten oder diese bestätigten und keine Veränderung des Katalogs veranlassten, nicht explizit dargestellt wurden. Es wurde die Ermittlung der Durchschnittswerte der *Likert-Skala* und dessen Vergleich mit den Punkten in der ersten Kriterienkatalog-Version aufgezeigt. Dabei zeigt das Kapitel, dass dabei lediglich bei den Kriterien „*Autorisierung* mit OAuth 2.0“ und „Serverbelastung“ Abweichungen festgestellt wurden, welche Tabelle 14 veranschaulichte. Unter den konkreten Fragen für die Experten stimmten 5 von 6 bereits mit den integrierten Katalogelementen in der ersten Version überein. Die Antwort von Herrn Brunner auf die zweite Frage („Gibt es in Ihren Augen noch andere Aspekte der jeweiligen Architekturen, die Einfluss auf eine angenehme „Benutzererfahrung“ haben?“) wurde besonders hervorgehoben, da diese eine Rolle spielte.

In Kapitel 4.3 („Einarbeitung der Experteninterview-Ergebnisse in den Kriterienkatalog“) wurde gezeigt, wie die zusammengefassten relevanten Ergebnisse aus Kapitel 4.2 („Ergebnisse der durchgeführten Experteninterviews“) in die erste Version des Kriterienkatalogs integriert wurden. Dabei zeigt das Kapitel, dass zunächst die abweichenden Punktzahlen in der ersten Version des Kriterienkatalogs mit den ermittelten Durchschnittswerten nach der *Likert-Skala* aus Tabelle 14 harmonisiert wurden. Besonders wurde hervorgehoben, dass die Antwort von Herrn Brunner auf die zweite Frage („Gibt es in Ihren Augen noch andere Aspekte der

jeweiligen Architekturen, die Einfluss auf eine angenehme „Benutzererfahrung“ haben?“) zu einer Erweiterung des entsprechenden Kriteriums führte. Die Ergebnisse wurden erfolgreich in die erste Version des Kriterienkatalogs eingearbeitet, was zur Entwicklung der zweiten Version führte. Diese überarbeitete Ausgabe des Katalogs ist im Anhang C („Zweite Version des Kriterienkatalogs nach den Experteninterviews“) beigefügt. Wichtig ist zu erwähnen, dass alle Änderungen im Vergleich zur ersten Version gelb markiert wurden. Diese zweite Version bietet zahlreiche Vorteile, die bereits in Kapitel 3.3 („Darstellung der Vorteile durch den Kriterienkatalog“) ausführlich erläutert wurden.

Kapitel 5

Erläuterung der durchgeführten Experimente

Im fünften Kapitel werden die durchgeführten Experimente näher erläutert. Dabei wurde zunächst ein Anwendungsfall definiert, der dazu diente, die zweite Version des Kriterienkatalogs einer weiteren Qualitätsprüfung zu unterziehen. Es ist von Bedeutung zu betonen, dass sämtliche Experimente auf der siebten Version der *.NET-Umgebung* von Microsoft basierten. Die achte Version der *.NET-Umgebung* wurde während der Erstellung dieser Arbeit veröffentlicht und bringt Veränderungen für das *Blazor Server Framework* mit. Die relevanten Anpassungen werden in Kapitel 8.3 („Ausblick“) angesprochen. Das fünfte Kapitel zeigt den Schritt 5 im Ablaufplan (siehe Abbildung 2 in Kapitel 1.4: „Beschreibung des Vorgehens“).

5.1 Erklärung der Anforderungen an die Experimente

Im Zuge von Kapitel 4.3 („Einarbeitung der Experteninterview-Ergebnisse in den Kriterienkatalog“) wurde die zweite Version des Kriterienkatalogs erstellt, welche als Anhang C („Zweite Version des Kriterienkatalogs nach den Experteninterviews“) der Arbeit beigefügt ist. Dieses Kapitel zeigt, wie die Gültigkeit der

zweiten Version des Kriterienkatalogs auf die Probe gestellt wurde. Zu diesem Zweck wurde ein Anwendungsfall definiert, bei dem die zweite Version des Kriterienkatalogs zur Identifikation der am besten geeigneten Architektur verwendet wurde. Der definierte Anwendungsfall, die Anwendung der zweiten Version des Kriterienkatalogs und die genutzten Bewertungskennzahlen für die Experimente werden nachfolgend aufgezeigt.

Darstellung des Anwendungsfalls

Für diesen Anwendungsfall wurde die fiktive Firma „Schalk Maschinen GmbH“ als Beispiel inszeniert. Es ist wichtig zu erwähnen, dass dieser Anwendungsfall auf einem früheren Projekt der Firma XITASO basiert und in Zusammenarbeit mit dem technischen Betreuer dieser Arbeit, Alexander Rampp, von der Firma XITASO vereinfacht für diese Arbeit formuliert wurde. Die Schalk Maschinen GmbH ist ein mittelständischer Maschinenhersteller mit 1000 Mitarbeitern und benötigt eine Webanwendung zur Verwaltung und Vereinfachung von Anmeldungen für anstehende Schulungen und Workshops. Bisher erhielten alle Mitarbeiter E-Mails von den Organisatoren der Veranstaltung und mussten darauf antworten, um sich anzumelden. Dieser Prozess ist zeitaufwendig und soll durch die neue Webanwendung optimiert werden. Die Schalk Maschinen GmbH hat folgende Anforderungen gestellt:

- Die Mitarbeiter sollen sich über ihre firmeneigenen Microsoft Accounts anmelden können (ohne Anmeldung kein Zugriff).
- Angemeldete Mitarbeiter (Teilnehmer) sollen aktuelle Schulungen und Workshops einsehen können, damit sich sich dafür anmelden können.
- Angemeldete Mitarbeiter (Teilnehmer) sollen eine übersichtliche Liste über ihre angemeldeten Veranstaltungen einsehen können, welche zeitlich aufsteigend sortiert ist. Ebenfalls soll die Möglichkeit bestehen, dass sich die Mitarbeiten von den Veranstaltungen wieder abmelden können.

- Ausgewählte angemeldete Mitarbeiter (Organisatoren) sollen neue Veranstaltungen erstellen und löschen können. Ebenfalls sollen diese sehen, wer sich bereits für welche Veranstaltung angemeldet hat.
- Jeder Mitarbeiter soll nur seine eigenen Daten einsehen können, nicht die von anderen.

Anwendung der zweiten Version des Kriterienkatalogs auf den Anwendungsfalls

In Kapitel 3.2 („Erläuterung der Ableitung auf die Architekturen“) wurde in Abbildung 6 die schrittweise Anwendung des Kriterienkatalogs dargestellt. Der erste Schritt ("Problemspezifikation festlegen") wurde bereits aufgezeigt. Danach wurden alle weiteren Schritte auf den definierten Anwendungsfall angewendet, um durch die zweite Version des Kriterienkatalogs eine geeignete Architektur abzuleiten. Es ist wichtig zu betonen, dass die Auswertung der zweiten Version des Kriterienkatalogs anhand des Anwendungsfalls von dem technischen Betreuer dieser Arbeit, Alexander Rampp, durchgeführt wurde. Dies erfolgte, da er den Anwendungsfall mitformuliert hat und über das erforderliche Wissen aus dem als Basis geltenden vergangenen XITASO-Projekt verfügte. Die vollständig ausgefüllte zweite Version des Kriterienkatalogs für den definierten Anwendungsfall ist im Anhang D („Ausgefüllte zweite Version des Kriterienkatalogs anhand des Beispielfalls für die Experimente“) zu finden. Ein Auszug daraus wird in Tabelle 15 gezeigt.

Kriterium	**Multi-Page-Architektur**	**Single-Page-Architektur**	**Fullstack-Architektur (*Blazor Server*)**	**Gewicht**	**Gewollt?**
***Autorisierung* mit OAuth 2.0** -> ...	... -> Punktzahl: 6	... -> Punktzahl: 3	... -> Punktzahl: 3	Gewicht: 10% MPA: 0,6 SPA: 0,3 Fullstack: 0,3	X
Entwicklungszeit -> ...	... -> Punktzahl: 3	... -> Punktzahl: 3	... -> Punktzahl: 6	Gewicht: 30% MPA: 0,9 SPA: 0,3 Fullstack: 1,8	X
...	...	...	...	...	...
Gesamtpunktzahl	3,55	3	5,55	100%	

Tabelle 15: Ausschnitt aus der ausgefüllten zweiten Version des Kriterienkatalogs durch den Anwendungsfall (eigene Darstellung)

Es werden zwei Kriterien dargestellt, die für die Umsetzung von Bedeutung waren: Die „*Autorisierung* mit OAuth 2.0“, gewichtet mit 10% und die „Entwicklungszeit“, gewichtet mit 30%. Die berechneten Gesamtpunktzahlen für die verschiedenen Architekturen zeigten, dass die Fullstack-Architektur, konkret *Blazor Server*, am besten für den gegebenen Anwendungsfall geeignet war. Trotzdem wurde der Anwendungsfall mit allen drei Architekturen umgesetzt, um einen direkten Vergleich dieser Ansätze zu ermöglichen. Dadurch wurde die zweite Version des Kriterienkatalogs evaluiert, um herauszufinden, ob die abgeleitete Architektur tatsächlich die optimalste Wahl war.

Aufzeigen der Bewertungskennzahlen für die Kriterien der zweiten Version des Kriterienkatalogs

Der Anwendungsfall stellte den Großteil der Kriterien der zweiten Version des Kriterienkatalogs auf die Probe. Kriterien, die nicht durch den Anwendungsfall überprüft wurden, wie beispielsweise die Offlinefähigkeit der Architekturen, bedurften keiner weiteren Evaluierungsrunde, da die Punktzahlen und Beschreibungen dieser sich nicht mehr veränderten oder durch den experimentellen Teil nicht ordentlich bewertet werden konnten. Alle anderen Kriterien wurden während der Durchführung der Experimente erneut verfeinert. Dafür werden im Folgenden verschiedene Kennzahlen aufgezeigt, die während der Umsetzung des Anwendungsfalls angewandt wurden:

- ***Autorisierung* mit OAuth 2.0**: Für die Implementierung der *Autorisierung* in allen drei Architekturen wurde die Zeit gestoppt und die Anzahl der Code-Zeilen festgehalten. So konnte annähernd überprüft werden, ob die Implementierung der *Autorisierung* einfach ist. Hier ist wichtig zu erwähnen, dass für die Experimente noch keine Vorkenntnisse bei der Implementierung der *Autorisierung* in OAuth 2.0 vorhanden waren.
  Folge für den Kriterienkatalog: Die Ergebnisse ermöglichten einen direkten Vergleich zwischen den drei Architekturen. So konnten die Werte im Kriterienkatalog bei Abweichungen angepasst werden.

- **Benutzererfahrung**: Die Benutzererfahrung wurde mithilfe der in der Praxis bewährten Chrome-Erweiterung „*Lighthouse*“ getestet, die eine Punktzahl für die zu untersuchende Webanwendung zurückgab.
  Folge für den Kriterienkatalog: Die Ergebnisse ermöglichten einen direkten Vergleich zwischen den drei Architekturen. So konnten die Werte im Kriterienkatalog bei Abweichungen angepasst werden.

- **Entwicklererfahrung**: Es wurde überprüft, ob *Blazor Server* tatsächlich am wenigsten Fachwissen im *Frontend* und *Backend* erforderte. Dies wurde erreicht, indem die verschiedenen Technologien und Programmiersprachen

festgehalten wurden, für die Fachwissen erforderlich war, um den Anwendungsfall technisch umzusetzen.
Folge für den Kriterienkatalog: Die Ergebnisse ermöglichten einen direkten Vergleich zwischen den drei Architekturen und zeigten, für welche am meisten Fachwissen benötigt wurde. So konnten die Werte im Kriterienkatalog bei Abweichungen angepasst werden.

- **Entwicklungszeit**: Die Zeit für die Entwicklung der verschiedenen Architekturen wurde gemessen, um einen Vergleich zwischen den dreien zu ermöglichen. Hier ist wichtig zu erwähnen, dass für die Experimente bereits Vorkenntnisse in den Programmiersprachen Java, JavaScript, TypeScript, CSS und HTML vorhanden waren. Ebenso ist darauf hinzuweisen, dass die unterschiedlichen Anforderungen in den Experimenten in unterschiedlicher Reihenfolge implementiert wurden. Dies geschah, um annähernd vergleichbare Entwicklungszeiten zu erzielen. Auf diese Weise sollte vermieden werden, dass eines der Experimente übermäßig von bereits erlangtem Implementierungswissen in einem anderen Prototypen profitierte.
  Folge für den Kriterienkatalog: Die Ergebnisse ermöglichten einen direkten Vergleich der Architekturen. So konnten die Werte im Kriterienkatalog bei Abweichungen angepasst werden.

- **Erweiterbarkeit**: Für jeden Ansatz wurde dokumentiert, wo im Code Anpassungen für neu hinzukommende Anforderungen erforderlich sind.
  Folge für den Kriterienkatalog: Die Ergebnisse ermöglichten einen direkten Vergleich zwischen den drei Architekturen und zeigten, bei welcher Architektur am meisten angepasst werden muss, bei neu hinzukommenden Anforderungen. So konnten die Werte im Kriterienkatalog bei Abweichungen angepasst werden.

- **Performanz**: Die Performanz wurde mithilfe der in der Praxis bewährten Chrome-Erweiterung „*Lighthouse*“ getestet, die eine Punktzahl für die zu untersuchende Webanwendung zurückgab.
  Folge für den Kriterienkatalog: Die Ergebnisse ermöglichten einen direk-

ten Vergleich zwischen den drei Architekturen. So konnten die Werte im Kriterienkatalog bei Abweichungen angepasst werden.

Die Resultate der Architekturen unter Berücksichtigung der unterschiedlichen Kennzahlen werden im sechsten Kapitel (Kap. 6: „Ergebnis der drei Experimente“) präsentiert.

Im Anschluss werden in den folgenden Kapiteln (Kap. 5.2: „Experiment 1: *ASP.NET Core 7.0* MVC (MPA)“, Kap. 5.3: „Experiment 2: *Vue 3.3* & *ASP.NET Core 7.0* (SPA)“, Kap. 5.4: „Experiment 3: *Blazor Server* (Fullstack-Architektur)“) die durchgeführten Experimente detailliert beschrieben. Dabei wird für jedes ein kurzer Steckbrief der verwendeten Technologien, ein Auszug aus der Projektstruktur und ein Komponentendiagramm aufgezeigt. Es ist wichtig zu erwähnen, dass die konkrete Umsetzung des Anwendungsfalls durch die drei Experimente nicht explizit in dieser Arbeit beschrieben wird. Diese kann jedoch auf GitHub unter dem folgenden Link eingesehen werden: `https://github.com/Murtl/bachelorarbeit-mertl`. Der Code ist auch auf der beiliegenden DVD (siehe Anhang G: „DVD mit Code der Experimente“) verfügbar. Das nächste Kapitel (Kap. 5.2: „Experiment 1: *ASP.NET Core 7.0* MVC (MPA)“) zeigt dabei das Experiment für die Multi-Page-Architektur.

5.2 Experiment 1: *ASP.NET Core 7.0* MVC (MPA)

Für die Umsetzung des in Kapitel 5.1 („Erklärung der Anforderungen an die Experimente“) beschriebenen Anwendungsfalls im Kontext der Multi-Page-Architektur wurde das von Microsoft entwickelte *Framework ASP.NET Core 7.0* Model-View-Controller (Abk.: MVC) verwendet. Nachfolgend wird in Tabelle 16 ein Steckbrief für dieses Experiment aufgezeigt, der die wichtigsten Fakten übersichtlich darstellt.

Steckbrief MPA Experiment	
Framework:	*ASP.NET Core 7.0* Model-View-Controller (MVC)
Entwicklungsumgebung:	*Visual Studio*
Programmiersprachen *Frontend*:	JavaScript, CSS, HTML
Programmiersprache *Backend*:	C#
Datenbank:	*JSON*-Datei
Programmiervorlage:	*Visual Studio* stellte eine *.NET*-Programmiervorlage mit dem Namen „ASP.NET Core-Web-App (Model View Controller)“ bereit.

Tabelle 16: Steckbrief des Experiments für die Multi-Page-Architektur (eigene Darstellung)

Dieses umfassende *Framework* ermöglichte die Erstellung einer Webanwendung unter Anwendung des Model-View-Controller-Entwurfsmusters. Die Entwicklungsumgebung war *Visual Studio*, diese stellte auch eine *.NET*-Programmiervorlage mit dem Namen „ASP.NET Core-Web-App (Model View Controller)“ für das *Framework* bereit. Die Programmiersprache für das *Backend* war C#, während JavaScript, HTML und CSS für das *Frontend* verwendet wurden. Die Anwendung wurde gemäß dem MVC-Entwurfsmuster strukturiert, das die Codetrennung in Modelle (Models), Ansichten (Views) und Controller (Controllers) vorsah. Die Projektstruktur, die durch die .NET-Programmiervorlage erzeugt und für das Experiment genutzt wurde, wird in Abbildung 8 dargestellt.

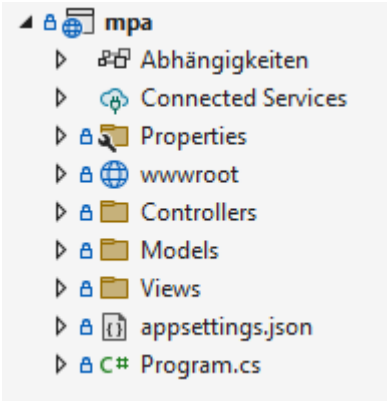


Abbildung 8: Struktur der *ASP.NET Core 7.0* MVC Anwendung (eigene Darstellung)

Die Abbildung verdeutlicht die Aufteilung der Anwendung in Modelle, Ansichten und Controller, wobei die Ansichten das *Frontend* repräsentieren und die Modelle sowie Controller das *Backend*. Neue Komponenten wurden durch den Anwendungsfall in diesen Hauptkomponentengruppen hinzugefügt. Weitere Informationen zur Programmiervorlage und dem MVC-Entwurfsmuster bietet Microsoft (vgl. Microsoft 2023i). Nachfolgend wird in Abbildung 9 ein Komponentendiagramm aufgezeigt, um das Zusammenspiel der verschiedenen Komponenten dieses Experiments zu verdeutlichen.

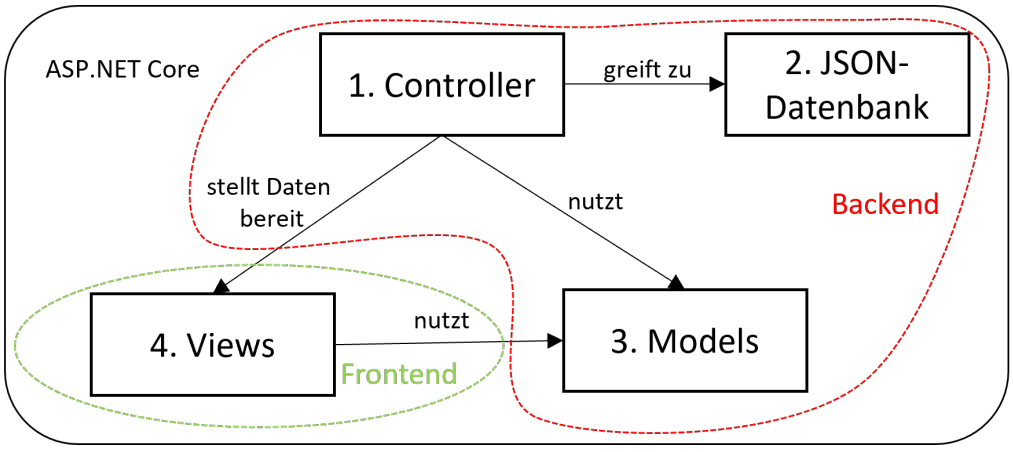


Abbildung 9: Komponentendiagramm des MPA Experiments (eigene Darstellung)

Die Abbildung verdeutlicht, dass die Controller (Nr. 1 in Abbildung 9) auf die *JSON*-Datenbank (Nr. 2 in Abbildung 9) zugreifen, um Daten zu erhalten. Zudem verwenden die Controller (Nr. 1 in Abbildung 9) die Models (Nr. 3 in Abbildung 9), um Daten zu speichern und diese den Views (Nr. 4 in Abbildung 9) zur Verfügung zu stellen. Die Views (Nr. 4 in Abbildung 9) greifen ebenfalls auf die Models zu und können auf diese Weise die bereitgestellten Daten durch die Controller (Nr. 1 in Abbildung 9) abrufen und anzeigen. Diese Views (Nr. 4 in Abbildung 9) werden einem Anwender am Client angezeigt. Der genaue Prozess, der erforderlich ist, um eine View am Client anzuzeigen, wurde bereits in Kapitel 2.2 („Die Multi-Page-Architektur im Überblick“) beschrieben. Die Ergebnisse des MPA Experiments werden in Kapitel 6.1 („Experiment 1: *ASP.NET Core 7.0* MVC-Anwendung (MPA)“) präsentiert.

Im nächsten Kapitel (Kap. 5.3: „Experiment 2: *Vue 3.3* & *ASP.NET Core 7.0* (SPA)“) wird das Experiment für die Single-Page-Architektur, ähnlich wie in diesem Kapitel für die Multi-Page-Architektur, vorgestellt.

5.3 Experiment 2: *Vue 3.3* & *ASP.NET Core 7.0* (SPA)

Für die Umsetzung des in Kapitel 5.1 („Erklärung der Anforderungen an die Experimente“) beschriebenen Anwendungsfalls im Kontext der Single-Page-Architektur wurden das von Microsoft entwickelte *Framework ASP.NET Core 7.0* für das *Backend* und das von Evan You entwickelte *Framework Vue 3.3* für das *Frontend* verwendet. Zusätzlich wurde eine *API* für die Kommunikation zwischen den beiden Anwendungen benötigt, wofür *Swagger (OpenAPI)* genutzt wurde. Nachfolgend wird in Tabelle 17 ein Steckbrief für dieses Experiment aufgezeigt, der die wichtigsten Fakten übersichtlich darstellt.

Steckbrief SPA Experiment	
Framework:	*Vue 3.3* und *ASP.NET Core 7.0*
Entwicklungsumgebung:	*Visual Studio*
Programmiersprachen ***Frontend***:	TypeScript, CSS, HTML
Programmiersprache ***Backend***:	C#
Datenbank:	*JSON*-Datei
Programmiervorlage:	*Visual Studio* stellte eine *.NET*-Programmiervorlage mit dem Namen „Vue and ASP.NET Core“ bereit.

Tabelle 17: Steckbrief des Experiments für die Single-Page-Architektur (eigene Darstellung)

Die Entwicklungsumgebung war *Visual Studio*, diese stellte auch eine *.NET*-Programmiervorlage für die benötigten Technologien bereit und verknüpfte die zwei Anwendungen miteinander. Die Projektstruktur, die durch die .NET-Programmiervorlage erzeugt und für das Experiment genutzt wurde, wird in Abbildung 10 dargestellt (vgl. Microsoft 2023d).

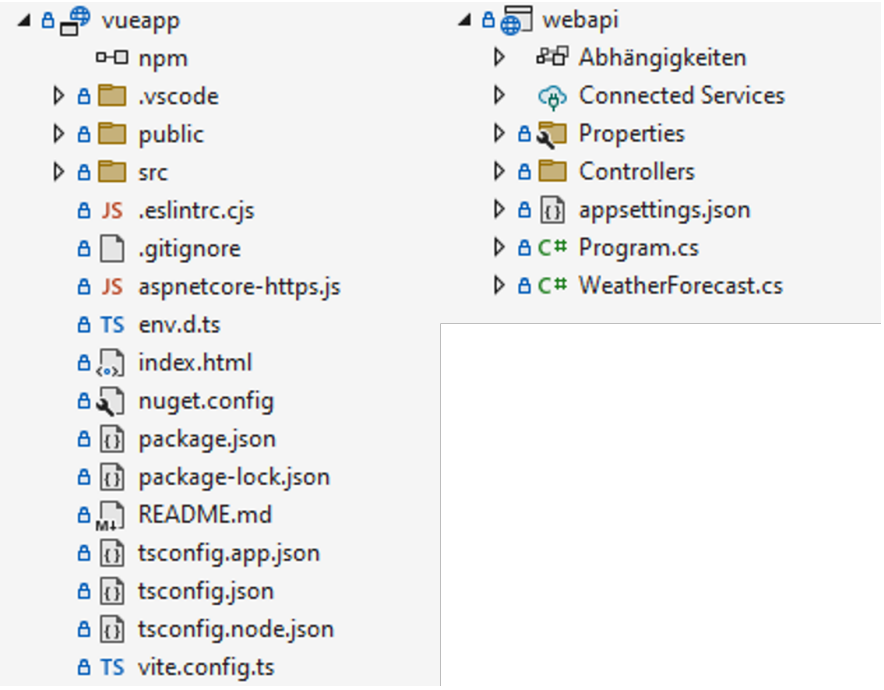


Abbildung 10: Struktur der *Vue 3.3 & ASP.NET Core 7.0* Anwendung (eigene Darstellung)

In der Abbildung ist der linke Teil dem *Vue 3.3 Frontend* zugeordnet, erkennbar am Namen des Oberordners „vueapp“. In diesem Teil wurde TypeScript, HTML und CSS verwendet, um die Benutzeroberfläche zu erstellen und interaktiv zu gestalten. Der rechte Teil repräsentiert das *ASP.NET Core 7.0 Backend*, identifizierbar am Namen des Oberordners „webapi“. Hier wurde C# verwendet, um mit den HTTP-Anfragen des *Frontend*s umzugehen und neue Daten über eine HTTP-Antwort bereitzustellen. Die API, die mit *Swagger (OpenAPI)* erstellt wurde, ist ebenfalls im *Backend* integriert und enthält die Definition und Behandlung verschiedener Endpunkte für Anfragen. Eine detaillierte Erläuterung der Vorlage und zusätzliche Informationen zur Programmierung sind bei Microsoft verfügbar (vgl. Microsoft 2023j). Durch den Anwendungsfall wurden jeweils neue Komponenten im *Frontend* und *Backend* hinzugefügt. Nachfolgend wird in Abbildung 11 ein Komponentendiagramm aufgezeigt, um das Zusammenspiel der verschiedenen Komponenten dieses Experiments zu verdeutlichen.

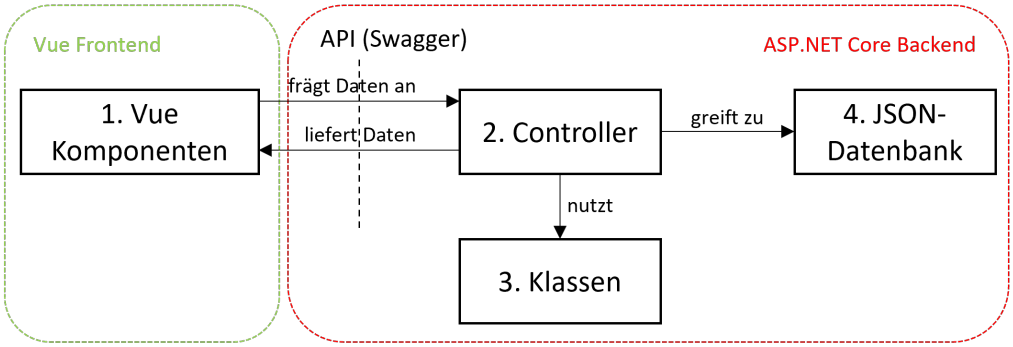


Abbildung 11: Komponentendiagramm des SPA Experiments (eigene Darstellung)

Die Abbildung verdeutlicht, dass die *Vue 3.3* Komponenten (Nr. 1 in Abbildung 11) über eine *API* (in dieser Arbeit: *Swagger (OpenAI)*) Daten bei den Controllern (Nr. 2 in Abbildung 11) anfragen. Diese verwenden Klassen (Nr. 3 in Abbildung 11), um die Daten zu strukturieren und greifen dabei auf die *JSON*-Datenbank (Nr. 4 in Abbildung 11) zu. Sobald alle Daten vorhanden sind, werden sie an die *Vue 3.3* Komponenten (Nr. 1 in Abbildung 11) zurückgeliefert. Die *Vue 3.3* Komponenten werden einem Anwender am Client angezeigt. Der genaue Prozess, der erforderlich ist, um die neuen Daten am Client anzuzeigen, wurde bereits in Kapitel 2.3 („Erklärung der Single-Page-Architektur“) beschrieben. Die Ergebnisse des SPA Experiments werden in Kapitel 6.2 („Experiment 2: *Vue 3.3* & *ASP.NET Core 7.0* Anwendung (SPA)“) präsentiert.

Im nächsten Kapitel (Kap. 5.4: „Experiment 3: *Blazor Server* (Fullstack-Architektur)“) wird das Experiment für die Fullstack-Architektur, ähnlich wie in diesem Kapitel für die Single-Page-Architektur, vorgestellt.

5.4 Experiment 3: *Blazor Server* (Fullstack-Architektur)

Für die Umsetzung des in Kapitel 5.1 („Erklärung der Anforderungen an die Experimente“) beschriebenen Anwendungsfalls im Kontext der Fullstack-Architektur wurde das von Microsoft entwickelte *Framework Blazor Server* verwendet. Dieses wurde bereits ausführlich in dieser Arbeit in Kapitel 2.4 („Erklärung der Fullstack-Architektur ohne direkte Trennung von *Frontend* und *Backend*“) beschrieben. Nachfolgend wird in Tabelle 18 ein Steckbrief für dieses Experiment aufgezeigt, der die wichtigsten Fakten übersichtlich darstellt.

Steckbrief Fullstack-Architektur Experiment	
Framework:	*Blazor Server*
Entwicklungsumgebung:	*Visual Studio*
Programmiersprachen *Frontend*:	C#, CSS, HTML
Programmiersprache *Backend*:	C#
Datenbank:	*JSON*-Datei
Programmiervorlage:	*Visual Studio* stellte eine *.NET*-Programmiervorlage mit dem Namen „Blazor Server App“ bereit

Tabelle 18: Steckbrief des Experiments für die Fullstack-Architektur (eigene Darstellung)

Die Entwicklungsumgebung war *Visual Studio*, diese stellte auch eine *.NET*-Programmiervorlage für das *Framework* bereit. Nach Verwendung der *.NET*-Programmiervorlage zur Projekterstellung wurde eine Struktur ersichtlich, wie in Abbildung 12 dargestellt.

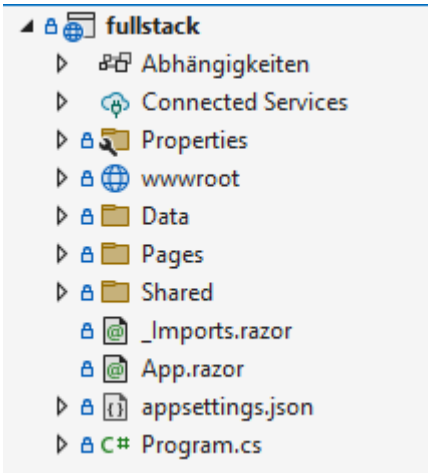


Abbildung 12: Struktur der *Blazor Server* Anwendung (eigene Darstellung)

In dieser Anwendung wurden Komponenten entwickelt, die keine strikte Trennung von *Frontend* und *Backend* beinhalten. Das *Framework Blazor Server* setzt C#, HTML und CSS für die Entwicklung der gesamten Webanwendung ein. Eine detaillierte Erklärung der Vorlage und zusätzliche Informationen zur Programmierung bietet Microsoft (vgl. Microsoft 2023c). Durch den Anwendungsfall sind neue Komponenten hinzugekommen. Nachfolgend wird in Abbildung 13 ein Komponentendiagramm aufgezeigt, um das Zusammenspiel der verschiedenen Komponenten dieses Experiments zu verdeutlichen.

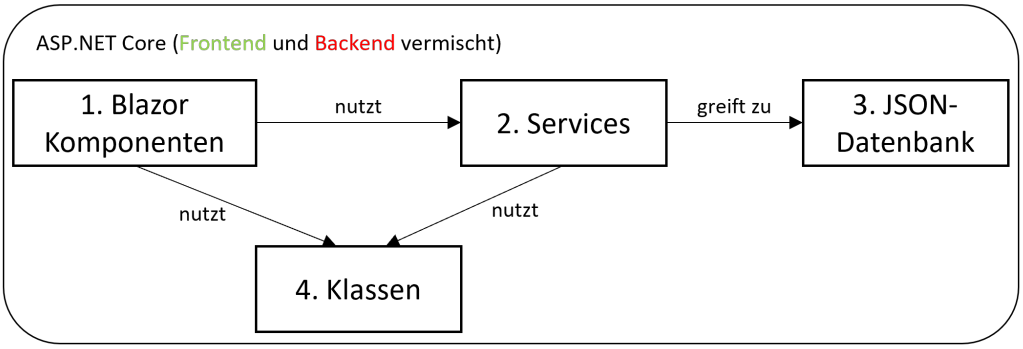


Abbildung 13: Komponentendiagramm des Fullstack-Architektur Experiments (eigene Darstellung)

Die Abbildung verdeutlicht, dass die Blazor Komponenten (Nr. 1 in Abbildung 13) Services (Nr. 2 in Abbildung 13) nutzen, um neue Daten zu erhalten. Die Services greifen dabei auf die *JSON*-Datenbank (Nr. 3 in Abbildung 13) zu. Sowohl die Blazor Komponenten (Nr. 1 in Abbildung 13) als auch die Services (Nr. 2 in Abbildung 13) verwenden die gleichen Klassen (Nr. 4 in Abbildung 13), um die Daten einheitlich zu strukturieren. Die Blazor Komponenten (Nr. 1 in Abbildung 13) werden einem Anwender am Client angezeigt. Der genaue Prozess, der erforderlich ist, um die neuen Daten am Client anzuzeigen, wurde bereits in Kapitel 2.4 („Erklärung der Fullstack-Architektur ohne direkte Trennung von *Frontend* und *Backend*“) beschrieben. Die Ergebnisse des Fullstack-Architektur Experiment werden in Kapitel 6.3 („Experiment 3: *Blazor Server* Anwendung (Fullstack-Architektur)“) präsentiert.

5.5 Zusammenfassung

Kapitel 5.1 („Erklärung der Anforderungen an die Experimente“) zeigte, wie die Gültigkeit der zweiten Version des Kriterienkatalogs auf die Probe gestellt wurde. Zu diesem Zweck wurde ein Anwendungsfall definiert, der die zweite Version des Kriterienkatalogs durch die Simulation der „Schalk Maschinen GmbH“ auf die Probe gestellt hat. Die zweite Version des Kriterienkatalogs, ausgefüllt anhand der Anforderungen dieses Anwendungsfalls, befindet sich im Anhang D („Ausgefüllte zweite Version des Kriterienkatalogs anhand des Beispielfalls für die Experimente“). Es ist wichtig zu erwähnen, dass dieser Anwendungsfall auf einem früheren Projekt der Firma XITASO basierte und in Zusammenarbeit mit dem technischen Betreuer dieser Arbeit, Alexander Rampp, von der Firma XITASO vereinfacht für diese Arbeit formuliert wurde. Ebenfalls wurde die zweite Version des Kriterienkatalogs von Alexander Rampp ausgefüllt, da er über das nötige Wissen über das vergangene XITASO-Projekt verfügte. Der Kriterienkatalog zeigte die Fullstack-Architektur (konkret *Blazor Server*) als Lösung für diesen Anwendungsfall auf. Trotzdem wurde der Anwendungsfall in allen drei Ansätzen implementiert, um einen direkten Vergleich zu ermöglichen und dabei anhand

von zuvor definierter und ebenfalls dargestellten Bewertungskennzahlen (z.B. das Stoppen der Zeit) untersucht. Die Ergebnisse dieser Experimente werden in Kapitel 6 („Ergebnis der drei Experimente“) vorgestellt.

In Kapitel 5.2 („Experiment 1: *ASP.NET Core 7.0* MVC (MPA)“) wurde für die Realisierung des Anwendungsfalls, repräsentativ für die Multi-Page-Architektur, das *Framework ASP.NET Core 7.0* Model-View-Controller (MVC) von Microsoft dargestellt. Das MVC-Entwurfsmuster unterteilte die Anwendung in drei Hauptkomponentengruppen (Modelle, Ansichten und Controller). Dabei wurde erklärt, dass die Ansichten das *Frontend* repräsentieren und die Modelle sowie Controller das *Backend*. Die Entwicklung erfolgte in der Entwicklungsumgebung *Visual Studio*, die auch eine *.NET*-Programmiervorlage für das *Framework* bereitstellte. Die Struktur der Anwendung, die von der *.NET*-Programmiervorlage erzeugt und erweitert wurde, stellte die Abbildung 8 dar. Das Zusammenspiel der verschiedenen Komponenten, die in dem MPA Experiment entstanden sind und genutzt wurden, präsentierte Abbildung 9.

In Kapitel 5.3 („Experiment 2: *Vue 3.3* & *ASP.NET Core 7.0* (SPA)“) wurde für die Realisierung des Anwendungsfalls, repräsentativ für die Single-Page-Architektur, das *Framework ASP.NET Core 7.0* für das *Backend* und das von Evan You entwickelte *Framework Vue 3.3* für das *Frontend* dargestellt. Die Kommunikation zwischen den beiden Teilen erfolgte über *Swagger (OpenAPI)*. Die Entwicklungsumgebung *Visual Studio* wurde genutzt, welche eine *.NET*-Programmiervorlage bereitstellte und die Struktur für beide Anwendungen erstellte: eine für das *Frontend* in *Vue 3.3* und eine für das *Backend* in *ASP.NET Core 7.0*. Die *Swagger (OpenAPI) API* wurde erstellt, um beide Komponenten direkt zu verbinden. Die Struktur der Anwendung, die von der *.NET*-Programmiervorlage erzeugt und erweitert wurde, veranschaulichte Abbildung 10. Das Zusammenspiel der verschiedenen Komponenten, die in dem SPA Experiment entstanden sind und genutzt wurden, präsentierte Abbildung 11.

In Kapitel 5.4 („Experiment 3: *Blazor Server* (Fullstack-Architektur)“) wurde für die Umsetzung des Anwendungsfalls, repräsentativ für die Fullstack-Architektur, das von Microsoft entwickelte *Framework Blazor Server* dargestellt, welches be-

reits in Kapitel 2.4 („Erklärung der Fullstack-Architektur ohne direkte Trennung von *Frontend* und *Backend*“) ausführlich beschrieben wurde. Die Entwicklungsumgebung *Visual Studio* wurde eingesetzt, wobei auch hier eine *.NET*-Programmiervorlage für das *Framework* zur Verfügung gestellt wurde. Die Struktur der Anwendung, die von der *.NET*-Programmiervorlage erzeugt und erweitert wurde, zeigte Abbildung 12. Das Zusammenspiel der verschiedenen Komponenten, die in dem Fullstack-Architektur Experiment entstanden sind und genutzt wurden, präsentierte Abbildung 13.

Kapitel 6

Ergebnis der drei Experimente

Im sechsten Kapitel werden die Ergebnisse der durchgeführten Experimente präsentiert, basierend auf dem zuvor beschriebenen Anwendungsfall in Kapitel 5.1 („Erklärung der Anforderungen an die Experimente“). Die Resultate werden mithilfe der in Kapitel 5.1 („Erklärung der Anforderungen an die Experimente“) beschriebenen Kennzahlen vorgestellt und tabellarisch gegenübergestellt. Im Gegensatz zu den Ergebnissen der Experteninterviews (siehe Kap. 4.2: „Ergebnisse der durchgeführten Experteninterviews“) werden in diesem Kapitel alle Ergebnisse der Experimente aufgezeigt, um die praktische Anwendung des Kriterienkatalogs und die daraus resultierenden Erkenntnisse nachvollziehbar darzustellen. Ebenfalls wird beschrieben, wie diese Ergebnisse genutzt wurden, um die finale Version des Kriterienkatalogs zu erstellen. Das sechste Kapitel zeigt den Schritt 6 im Ablaufplan (siehe Abbildung 2 in Kapitel 1.4: „Beschreibung des Vorgehens“).

6.1 Experiment 1: *ASP.NET Core 7.0* MVC Anwendung (MPA)

In Kapitel 5.2 („Experiment 1: *ASP.NET Core 7.0* MVC (MPA)“) wurde das Experiment zur Multi-Page-Architektur erläutert. In diesem Kapitel werden die Ergebnisse nach der Durchführung anhand der in Kapitel 5.1 („Erklärung der Anfor-

derungen an die Experimente“) vorgestellten Kennzahlen tabellarisch aufgezeigt. Nachfolgend folgt in Tabelle 19 eine Übersicht über die Ergebnisse mit erklärenden Beschreibungen.

Kriterium	**Kennzahlergebnis**
Autorisierung mit OAuth 2.0	Für die Implementierung der *Autorisierung* mithilfe von OAuth 2.0 wurde die Plattform *Microsoft Azure* verwendet. Der Gesamtaufwand für die Umsetzung belief sich auf 3 Stunden, einschließlich der Zeit, die für das Erlernen des Verfahrens benötigt wurde. Die Anzahl der Codezeilen für diese Aufgabe betrug 40.
Benutzererfahrung	Die Benutzererfahrung der MPA-Anwendung wurde unter Zuhilfenahme der Google-Chrome-Erweiterung *Lighthouse* bewertet (siehe Anhang E: „Ergebnisse der Bewertung der Experimente durch *Lighthouse*“). Für die Beurteilung der Benutzererfahrung wurde die Metrik „*Interaction to next Paint (Abk.: INP)*“ herangezogen. Der *INP* des Experiments betrug 30 Millisekunden.
Entwicklererfahrung	Die Realisierung des Experiments für die Multi-Page-Architektur erforderte Expertise in der *.NET-Umgebung* sowie umfassendes Fachwissen in den Programmiersprachen C#, HTML und CSS. Darüber hinaus war Kenntnis in JavaScript notwendig, um die Interaktivität zu verbessern. Insbesondere wurde JavaScript eingebunden, um in diesem Kontext ein Fenster für das Hinzufügen einer neuen Veranstaltung auf der Seite zu integrieren.

Kriterium	**Kennzahlergebnis**
Entwicklungszeit	Die Gesamtdauer für die Durchführung des Experiments belief sich auf 12 Stunden und 5 Minuten. Dies umfasst die Implementierung der *Autorisierung*, sämtlicher Anforderungen sowie das Hinzufügen von Kommentaren im Code. Es ist zu betonen, dass in diesem Rahmen lediglich ein Prototyp entwickelt wurde.
Erweiterbarkeit	Dank der implementierten Codetrennung gemäß dem Model-View-Controller-Entwurfsmuster weist die Anwendung eine übersichtliche Struktur auf. Bei neuen Anforderungen, wie beispielsweise der Integration neuer Ansichten, sind die Erstellung neuer Controller und entsprechender Models für die Daten notwendig. Eine Erweiterung erfordert somit Eingriffe an mehreren Stellen im Code.
Performanz	Die Performanz der MPA-Anwendung wurde unter Zuhilfenahme der Google-Chrome-Erweiterung *Lighthouse* bewertet (siehe Anhang E: „Ergebnisse der Bewertung der Experimente durch *Lighthouse*“). Zur Beurteilung der Performanz wurden, wie bereits bei der Bewertung der Benutzererfahrung, die Metrik „*Interaction to next Paint (Abk.: INP)*“ und zusätzlich die Metriken „*Speed Index*“ und „*First Contentful Paint (Abk.: FCP)*“ berücksichtigt. Die Ergebnisse für das Experiment sind wie folgt: Der *Speed Index* betrug 2,8 Sekunden, der *INP* lag bei 30 Millisekunden und der *FCP* betrug 1,4 Sekunden.

Tabelle 19: Ergebnis aus dem Experiment der Multi-Page-Architektur (eigene Darstellung)

Die präsentierten Ergebnisse des Experiments für die Multi-Page-Architektur dienen dazu, einen direkten Vergleich mit den anderen beiden Experimenten zu ermöglichen. Eine klare Zusammenfassung und Gegenüberstellung der Resultate der drei Experimente erfolgt in Kapitel 6.4 („Ergebnisse der Experimente in der Übersicht“). Im nachfolgenden Kapitel (Kap. 6.2: „Experiment 2: *Vue 3.3* & *ASP.NET Core 7.0* Anwendung (SPA)“) werden die Ergebnisse des Experiments für die Single-Page-Architektur genauso ausführlich präsentiert wie die Resultate des Multi-Page-Architektur-Experiments in diesem Kapitel.

6.2 Experiment 2: *Vue 3.3* & *ASP.NET Core 7.0* Anwendung (SPA)

In Kapitel 5.3 („Experiment 2: *Vue 3.3* & *ASP.NET Core 7.0* (SPA)“) wurde das Experiment zur Single-Page-Architektur erläutert. In diesem Kapitel werden die Ergebnisse nach der Durchführung anhand der in Kapitel 5.1 („Erklärung der Anforderungen an die Experimente“) vorgestellten Kennzahlen tabellarisch aufgezeigt. Nachfolgend folgt in Tabelle 20 eine Übersicht über die Ergebnisse mit erklärenden Beschreibungen.

Kriterium	**Kennzahlergebnis**
Autorisierung mit OAuth 2.0	Für die Implementierung der *Autorisierung* mithilfe von OAuth 2.0 wurde die Plattform *Microsoft Azure* verwendet. Der Gesamtaufwand für die Umsetzung belief sich auf 5 Stunden, einschließlich der Zeit, die für das Erlernen des Verfahrens benötigt wurde. Die Anzahl der Codezeilen für diese Aufgabe betrug 220.

Kriterium	**Kennzahlergebnis**
Benutzererfahrung	Die Benutzererfahrung der SPA-Anwendung wurde unter Zuhilfenahme der Google-Chrome-Erweiterung *Lighthouse* bewertet (siehe Anhang E: „Ergebnisse der Bewertung der Experimente durch *Lighthouse*“). Für die Beurteilung der Benutzererfahrung wurde die Metrik „*Interaction to next Paint (Abk.: INP)*“ herangezogen. Der *INP* des Experiments betrug 30 Millisekunden.
Entwicklererfahrung	Die Umsetzung des Experiments für die Single-Page-Architektur erforderte Kenntnisse in der *.NET-Umgebung* sowie in *Vue 3.3*. Zusätzlich waren umfassende Fachkenntnisse in den Programmiersprachen C#, HTML, CSS und TypeScript (typisiertes JavaScript) erforderlich. In diesem Kontext ist Fachwissen in zwei *Frameworks* (.NET und Vue) notwendig, da die Entwicklung von zwei unterschiedlichen Anwendungen erfolgte.
Entwicklungszeit	Die Gesamtdauer für die Durchführung des Experiments belief sich auf 14 Stunden und 55 Minuten. Dies umfasst die Implementierung der *Autorisierung*, sämtlicher Anforderungen sowie das Hinzufügen von Kommentaren im Code. Es ist zu betonen, dass in diesem Rahmen lediglich ein Prototyp entwickelt wurde.

Kriterium	**Kennzahlergebnis**
Erweiterbarkeit	Hier wurden zwei Anwendungen entwickelt, die das *Frontend* und *Backend* voneinander trennen. Bei neuen Anforderungen, beispielsweise der Integration neuer Ansichten, genügt es, das *Vue 3.3 Frontend* anzupassen und vorhandene Schnittstellen für Daten zu nutzen. Eine Erweiterung des *ASP.NET Core Backends* wird erst dann erforderlich, wenn neue Endpunkte für zusätzliche Daten benötigt werden. In diesem Fall ist eine Anpassung in beiden Anwendungen notwendig.
Performanz	Die Performanz der SPA-Anwendung wurde unter Zuhilfenahme der Google-Chrome-Erweiterung *Lighthouse* bewertet (siehe Anhang E: „Ergebnisse der Bewertung der Experimente durch *Lighthouse*“). Zur Beurteilung der Performanz wurden, wie bereits bei der Bewertung der Benutzererfahrung, die Metrik „*Interaction to next Paint (Abk.: INP)*“ und zusätzlich die Metriken „*Speed Index*“ und „*First Contentful Paint (Abk.: FCP)*“ berücksichtigt. Die Ergebnisse für das Experiment sind wie folgt: Der *Speed Index* betrug 4,5 Sekunden, der *INP* lag bei 30 Millisekunden und der *FCP* betrug 1,6 Sekunden.

Tabelle 20: Ergebnis aus dem Experiment der Single-Page-Architektur (eigene Darstellung)

Die vorgestellten Resultate des Experiments für die Single-Page-Architektur ermöglichen einen direkten Vergleich mit den beiden anderen durchgeführten Experimenten. Eine klare Zusammenfassung und Gegenüberstellung der Resultate aller drei Experimente findet sich in Kapitel 6.4 („Ergebnisse der Experimente in

der Übersicht“). Im nachfolgenden Kapitel (Kap. 6.3: „Experiment 3: *Blazor Server* Anwendung (Fullstack-Architektur)“) werden die Resultate des Experiments für die Fullstack-Architektur genauso ausführlich präsentiert wie die Erkenntnisse des Experiments zur Single-Page-Architektur in diesem Kapitel.

6.3 Experiment 3: *Blazor Server* Anwendung (Fullstack-Architektur)

In Kapitel 5.4 („Experiment 3: *Blazor Server* (Fullstack-Architektur)“) wurde das Experiment zur Fullstack-Architektur erläutert. In diesem Kapitel werden die Ergebnisse nach der Durchführung anhand der in Kapitel 5.1 („Erklärung der Anforderungen an die Experimente“) vorgestellten Kennzahlen tabellarisch aufgezeigt. Nachfolgend folgt in Tabelle 21 eine Übersicht über die Ergebnisse mit erklärenden Beschreibungen.

Kriterium	**Kennzahlergebnis**
Autorisierung mit OAuth 2.0	Für die Implementierung der *Autorisierung* mithilfe von OAuth 2.0 wurde die Plattform *Microsoft Azure* verwendet. Der Gesamtaufwand für die Umsetzung belief sich auf 3 Stunden, einschließlich der Zeit, die für das Erlernen des Verfahrens benötigt wurde. Die Anzahl der Codezeilen für diese Aufgabe betrug 50.

Kriterium	**Kennzahlergebnis**
Benutzererfahrung	Die Benutzererfahrung der Fullstack-Architektur-Anwendung wurde unter Zuhilfenahme der Google-Chrome-Erweiterung *Lighthouse* bewertet (siehe Anhang E: „Ergebnisse der Bewertung der Experimente durch *Lighthouse*“). Für die Beurteilung der Benutzererfahrung wurde die Metrik „*Interaction to next Paint (Abk.: INP)*“ herangezogen. Der *INP* des Experiments betrug 20 Millisekunden.
Entwicklererfahrung	Die Realisierung des Experiments für die Fullstack-Architektur erforderte Expertise in der *.NET-Umgebung* sowie umfassendes Fachwissen in den Programmiersprachen C#, HTML und CSS. Hier konnte komplett auf JavaScript verzichtet werden, da mit C# alle nötigen Funktionen umgesetzt werden konnten.
Entwicklungszeit	Die Gesamtdauer für die Durchführung des Experiments belief sich auf 10 Stunden und 40 Minuten. Dies umfasst die Implementierung der *Autorisierung*, sämtlicher Anforderungen sowie das Hinzufügen von Kommentaren im Code. Es ist zu betonen, dass in diesem Rahmen lediglich ein Prototyp entwickelt wurde.

Kriterium	**Kennzahlergebnis**
Erweiterbarkeit	Für das Experiment mit der Fullstack-Architektur wurde eine *Blazor Server*-Anwendung entwickelt, bei der keine klare Trennung zwischen *Frontend* und *Backend* implementiert wurde. Bei neuen Anforderungen, wie beispielsweise der Integration neuer Ansichten, sind die Entwicklung neuer Blazor-Komponenten erforderlich. Diese müssen entweder auf vorhandene Services oder direkt auf die benötigten Daten zugreifen.
Performanz	Die Performanz der Fullstack-Architektur-Anwendung wurde unter Zuhilfenahme der Google-Chrome-Erweiterung *Lighthouse* bewertet (siehe Anhang E: „Ergebnisse der Bewertung der Experimente durch *Lighthouse*“). Zur Beurteilung der Performanz wurden, wie bereits bei der Bewertung der Benutzererfahrung, die Metrik „*Interaction to next Paint (Abk.: INP)*“ und zusätzlich die Metriken „*Speed Index*“ und „*First Contentful Paint (Abk.: FCP)*“ berücksichtigt. Die Ergebnisse für das Experiment sind wie folgt: Der *Speed Index* betrug 0,9 Sekunden, der *INP* lag bei 20 Millisekunden und der *FCP* betrug 0,9 Sekunden.

Tabelle 21: Ergebnis aus dem Experiment der Fullstack-Architektur (eigene Darstellung)

Die dargestellten Ergebnisse des Experiments für die Fullstack-Architektur dienen dazu, einen direkten Vergleich mit den anderen beiden Experimenten zu ermöglichen. Eine klare Zusammenfassung und Gegenüberstellung der Resultate der drei Experimente erfolgt im nachfolgendem Kapitel 6.4 („Ergebnisse der Experimente

in der Übersicht“).

6.4 Ergebnisse der Experimente in der Übersicht

In den vorherigen Kapiteln wurden die Ergebnisse der jeweiligen Experimente für die Multi-Page-Architektur (Kap. 6.1: „Experiment 1: *ASP.NET Core 7.0* MVC-Anwendung (MPA)“), Single-Page-Architektur (Kap. 6.2: „Experiment 2: *Vue 3.3* & *ASP.NET Core 7.0* Anwendung (SPA)“) und Fullstack-Architektur (Kap. 6.3: „Experiment 3: *Blazor Server* Anwendung (Fullstack-Architektur)“) ausführlich präsentiert. Im Folgenden wird in Tabelle 22 eine vereinfachte tabellarische Übersicht dieser Ergebnisse gegeben.

Kriterium	**Multi-Page-Architektur**	**Single-Page-Architektur**	**Fullstack-Architektur (*Blazor Server*)**
Autorisierung mit OAuth 2.0	Codezeilen: 40 Zeitaufwand: 3h	Codezeilen: 220 Zeitaufwand: 5h	Codezeilen: 50 Zeitaufwand: 3h
Benutzererfahrung	*Interaction to Next Paint*: 30ms	*Interaction to Next Paint*: 30ms	*Interaction to Next Paint*: 20ms
Entwickler-erfahrung	Fachwissen nötig in: *.NET*, C#, HTML, CSS, JavaScript	Fachwissen nötig in: *.NET*, C#, HTML, CSS, TypeScript, *Vue 3.3*	Fachwissen nötig in: *.NET*, C#, HTML, CSS
Entwicklungszeit	12h 5min	14h 55min	10h 40min

Kriterium	**Multi-Page-Architektur**	**Single-Page-Architektur**	**Fullstack-Architektur (*Blazor Server*)**
Erweiterbarkeit	Durch die übersichtliche Codetrennung in der Anwendung ist die Struktur klar gegliedert. Dennoch erfordern beispielsweise neue Ansichten Anpassungen sowohl im *Frontend* als auch im *Backend*.	Durch die Trennung von *Frontend* und *Backend* in zwei eigenständige Anwendungen gestaltet sich die Anpassung bei z.B. neuen Ansichten einfach und erfolgt ausschließlich im *Frontend*.	In der Anwendung besteht keine klare Codetrennung, weshalb beispielsweise die Einführung neuer Ansichten neue Blazor-Komponenten erfordert. Diese müssen in einem nicht klar strukturierten Umfeld zugeordnet werden und haben die Möglichkeit, entweder auf vorhandene Services zuzugreifen oder direkt auf die Datenbank zuzugreifen. Letzteres beeinträchtigt jedoch die Übersichtlichkeit der Struktur.

Kriterium	**Multi-Page-Architektur**	**Single-Page-Architektur**	**Fullstack-Architektur (*Blazor Server*)**
Performanz	*Speed Index*: 2,8s *First Contentful Paint*: 1,4s *Interaction to Next Paint*: 30ms	*Speed Index*: 4,5s *First Contentful Paint*: 1,6s *Interaction to Next Paint*: 30ms	*Speed Index*: 0,9s *First Contentful Paint:* 0,9s *Interaction to Next Paint*: 20ms

Tabelle 22: Übersicht der Ergebnisse der Experimente (eigene Darstellung)

Die in Tabelle 22 dieses Kapitels gegenübergestellten Ergebnisse bieten einen schnellen Überblick über die verschiedenen Erfüllungsgrade der Kriterien für die drei unterschiedlichen Architekturen. Dabei zeigt sich, dass die Fullstack-Architektur bessere Ergebnisse erzielte als die beiden anderen Architekturen. Beispielsweise wies die Fullstack-Architektur die geringste Entwicklungszeit auf und zeigte die beste Performanz im Test mithilfe der Chrome-Erweiterung *Lighthouse*. Die Ableitung durch die zweite Version des Kriterienkatalogs nach der Anwendung auf den in Kapitel 5.1 („Erklärung der Anforderungen an die Experimente“) gezeigten Anwendungsfall (siehe Anhang D: „Ausgefüllte zweite Version des Kriterienkatalogs anhand des Beispielfalls für die Experimente“) wurde somit bestätigt, da die Fullstack-Architektur für diesen spezifischen Anwendungsfall als die am besten geeignete Architektur herausragte.
Dennoch zeigten die Ergebnisse in den Kriterien „Benutzererfahrung“ und „Performanz“ Abweichungen gegenüber den integrierten Informationen in der zweiten Version des Kriterienkatalogs. Daher wird im nachfolgenden Kapitel (Kap. 6.5: „Einarbeitung der Ergebnisse in den Kriterienkatalog“) gezeigt, wie diese Erkenntnisse in die finale Version des Kriterienkatalogs integriert wurden.

6.5 Einarbeitung der Ergebnisse in den Kriterienkatalog

Die Ergebnisse der drei durchgeführten Experimente wurden übersichtlich im vorangehenden Kapitel (Kap. 6.4: „Ergebnisse der Experimente in der Übersicht“) dargestellt. In diesem Kapitel werden die Ergebnisse genauer präsentiert, die von den zuvor festgelegten Informationen und Punkten in der zweiten Version des Kriterienkatalogs abwichen. Zunächst ist zu betonen, dass die Ergebnisse für die definierten Kennzahlen bezüglich der Kriterien *Autorisierung* mit OAuth 2.0, Entwicklererfahrung, Entwicklungszeit und Erweiterbarkeit mit den bereits gepflegten und integrierten Informationen in der zweiten Version des Kriterienkatalogs übereinstimmten. Daher bedurften diese Kriterien keiner Anpassungen. Im Gegensatz dazu zeigten die Ergebnisse für die Kennzahlen der Kriterien Benutzererfahrung und Performanz Abweichungen. Im Folgenden werden diese Abweichungen detailliert beschrieben und gezeigt, wie diese in die zweite Version des Kriterienkatalogs integriert wurden, um die finale Version von diesem zu erstellen. Alle Änderungen im Vergleich zur zweiten Version sind in der finalen beiliegenden Version des Kriterienkatalogs (siehe Anhang F: „Finale Version des Kriterienkatalogs nach den Experimenten“) gelb markiert.

Einarbeitung der Ergebnisse des Kriteriums „Benutzererfahrung“

Für die Bewertung der Benutzererfahrung wurde der Messwert „*Interaction to next Paint (Abk.: INP)*“ von *Lighthouse* verwendet, welcher angibt, wie schnell eine Webanwendung auf Nutzerinteraktionen reagieren kann. In den Experimenten ergab sich für die Fullstack-Architektur der beste Wert von 20 Millisekunden. Daher blieb der Punktewert für die Fullstack-Architektur in diesem Kriterium in der finalen Version des Kriterienkatalogs unverändert bei 6. Für die SPA wurde dieser Wert von 6 auf 3 reduziert, da die Fullstack-Architektur schneller auf Nutzerinteraktionen reagierte und somit eine bessere Benutzererfahrung bietet. Der

Wert für die MPA blieb hingegen bei 1, da trotz der gemessenen 30 Millisekunden jede Anfrage nach neuen Daten zum Neuladen der gesamten Seite führte. Im Gegensatz dazu war dies im Experiment der SPA nicht erforderlich und konnte mit ansprechenden, benutzerfreundlichen Animationen überbrückt werden. Diese Informationen wurden der Beschreibung in der finalen Version des Kriterienkatalogs hinzugefügt.

Einarbeitung der Ergebnisse des Kriteriums „Performanz“ für das initiale Laden

Für die Bewertung des initialen Ladens bei der Performanz wurden die Messwerte „*First Contentful Paint (Abk.: FCP)*“ und „*Speed Index*“ von *Lighthouse* verwendet. Der *FCP* gibt an, wann ein Nutzer auf einer Webanwendung erstmalig visuellen Inhalt wahrnehmen kann. Der *Speed Index* gibt an, wie schnell der gesamte Inhalt einer Webanwendung visuell dargestellt werden kann. Im Experiment der Fullstack-Architektur wurden hier in beiden Messwerten die besten Werte mit 0,9 Sekunden für den *Speed Index* und ebenfalls 0,9 Sekunden für den *FCP* gemessen. Daher blieb der Punktewert für die Fullstack-Architektur in diesem Kriterium in der finalen Version des Kriterienkatalogs unverändert bei 6. Im Experiment der SPA wurden hier in beiden Messwerten die schlechtesten Werte mit 4,5 Sekunden für den *Speed Index* und 1,6 Sekunden für den *FCP* gemessen. Daher blieb der Punktewert für die SPA in diesem Kriterium in der finalen Version des Kriterienkatalogs unverändert bei 1. Im Experiment der MPA wurden hier in beiden Messwerten mittlere Werte mit 2,8 Sekunden für den *Speed Index* und 1,4 Sekunden für den *FCP* gemessen. Dadurch änderte sich der Punktewert für die MPA in diesem Kriterium in der finalen Version des Kriterienkatalogs auf 3. Die Beschreibung wurde ebenfalls um die Information erweitert, dass die Fullstack-Architektur schneller lädt.

Einarbeitung der Ergebnisse des Kriteriums „Performanz“ für das Nachladen neuer Inhalte

Für die Bewertung des Nachladens neuer Inhalte bei der Performanz wurde, wie bei der Benutzererfahrung, der Messwert „*Interaction to next Paint (Abk.: INP)*“ von *Lighthouse* verwendet, welcher angibt, wie schnell eine Webanwendung auf Nutzerinteraktionen reagieren kann. In den Experimenten ergab sich für die Fullstack-Architektur der beste Wert von 20 Millisekunden und für die MPA und SPA jeweils 30 Millisekunden. Dadurch reduzierte sich die Punktzahl der SPA von 6 auf 3 in der finalen Version des Kriterienkatalogs, da diese beiden Architekturen gleich schnell auf Nutzerinteraktionen reagierten. Die Beschreibung der SPA wurde um die Information erweitert, dass die Fullstack-Architektur schneller lädt und die MPA gleich schnell.

Nach der Durchführung, Bewertung und Berücksichtigung der Experimente für den Kriterienkatalog wurde dieser vollständig erstellt und liegt aus Platzgründen in seiner finalen Version (siehe Anhang F: „Finale Version des Kriterienkatalogs nach den Experimenten“) der Arbeit bei. Die Experimente trugen wesentlich dazu bei, den Katalog praktisch anzuwenden und die Theorie in die Praxis umzusetzen. Somit ist der Kriterienkatalog nun einsatzbereit und bietet zahlreiche Vorteile, die bereits in Kapitel 3.3 („Darstellung der Vorteile durch den Kriterienkatalog“) ausführlich erläutert wurden.

6.6 Zusammenfassung

Das sechste Kapitel („Ergebnis der drei Experimente“) präsentierte die Ergebnisse der durchgeführten Experimente für die Multi-Page-Architektur, Single-Page-Architektur und Fullstack-Architektur. Dabei wurden die Ergebnisse mithilfe der in Kapitel 5.1 („Erklärung der Anforderungen an die Experimente“) beschriebenen Kennzahlen vorgestellt. Es ist wichtig zu erwähnen, dass im Gegensatz zu

den Ergebnissen der Experteninterviews (siehe Kap. 4.2: „Ergebnisse der durchgeführten Experteninterviews“) in diesem Kapitel alle Ergebnisse der Experimente aufgezeigt wurden, um die praktische Anwendung des Kriterienkatalogs und die daraus resultierenden Erkenntnisse nachvollziehbar darzustellen.

In den Kapiteln 6.1 („Experiment 1: *ASP.NET Core 7.0* MVC Anwendung (MPA)“), 6.2 („Experiment 2: *Vue 3.3* & *ASP.NET Core 7.0* Anwendung (SPA)“) und 6.3 („Experiment 3: *Blazor Server* Anwendung (Fullstack-Architektur)“) wurden die Ergebnisse der drei durchgeführten Experimente detailliert tabellarisch aufgezeigt. Dabei präsentierte Tabelle 19 (siehe Kap. 6.1: „Experiment 1: *ASP.NET Core 7.0* MVC Anwendung (MPA)“) die Ergebnisse des Experiments für die Multi-Page-Architektur, Tabelle 20 (siehe Kap. 6.2: „Experiment 2: *Vue 3.3* & *ASP.NET Core 7.0* Anwendung (SPA)“) die Ergebnisse des Experiments für die Single-Page-Architektur und Tabelle 21 (siehe Kap. 6.3: „Experiment 3: *Blazor Server* Anwendung (Fullstack-Architektur)“) die Ergebnisse des Experiments für die Fullstack-Architektur.

In Kapitel 6.4 („Ergebnisse der Experimente in der Übersicht“) wurden die Ergebnisse der drei Experimente in Tabelle 22 vereinfacht und übersichtlich dargestellt, um einen schnellen Vergleich zu ermöglichen. Es wurde betont, dass die Ergebnisse des Experiments für die Fullstack-Architektur im Vergleich zu den beiden anderen Architekturen besser ausfielen. Daher wurde deutlich gemacht, dass die Ableitung der zweiten Version des Kriterienkatalogs für den definierten Anwendungsfall aus Kapitel 5.1 („Erklärung der Anforderungen an die Experimente“) sich als korrekt erwiesen hat (siehe Anhang D: „Ausgefüllte zweite Version des Kriterienkatalogs anhand des Beispielfalls für die Experimente“).

Kapitel 6.5 („Einarbeitung der Ergebnisse in den Kriterienkatalog“) zeigte, wie die Ergebnisse der Experimente in die zweite Version des Kriterienkatalogs eingearbeitet wurden. Dabei zeigte das Kapitel, dass die Ergebnisse für die Kriterien *Autorisierung* mit OAuth 2.0, Entwicklererfahrung, Entwicklungszeit und Erweiterbarkeit mit den bereits gepflegten und integrierten Informationen in der zweiten Version des Kriterienkatalogs übereinstimmten und somit keiner Anpassungen bedurften. Die Ergebnisse für die Kriterien Benutzererfahrung und Perfor-

manz, die sich unter dem Einsatz von *Lighthouse* ergeben hatten, wurden jedoch berücksichtigt. Die Abweichungen wurden detailliert beschrieben und das Kapitel zeigte, wie diese in die zweite Version des Kriterienkatalogs eingearbeitet wurden. So ist die finale Version des Kriterienkatalogs entstanden, die der Arbeit unter Anhang F („Finale Version des Kriterienkatalogs nach den Experimenten“) beiliegt. Wichtig zu erwähnen ist, dass alle Änderungen im Vergleich zur zweiten Version gelb markiert wurden. Diese finale Version ist nun einsatzbereit und bietet zahlreiche Vorteile, die bereits in Kapitel 3.3 („Darstellung der Vorteile durch den Kriterienkatalog“) ausführlich erläutert wurden.

Kapitel 7

Diskussion

Das siebte Kapitel widmet sich der eingehenden Diskussion der Ergebnisse, die im Verlauf des Ablaufplans (siehe Abbildung 2 in Kapitel 1.4: „Beschreibung des Vorgehens“) erarbeitet wurden. Zunächst erfolgt eine detaillierte Interpretation der Ergebnisse, um ihre Bedeutung für architektonische Untersuchungen im Bereich der Webarchitekturen zu verdeutlichen. Anschließend werden die identifizierten Stärken und Schwächen der durchgeführten Bachelorarbeit aufgezeigt, die bei weiteren Betrachtungen berücksichtigt werden sollten.

7.1 Interpretation der Ergebnisse

Die vorliegende Arbeit hat durch die systematische Abarbeitung des Ablaufplans (siehe Abbildung 2 in Kapitel 1.4: „Beschreibung des Vorgehens“) einen umfassenden Kriterienkatalog erarbeitet, welcher im Anhang F („Finale Version des Kriterienkatalogs nach den Experimenten“) vorliegt. Dieser Katalog kann in architektonischen Fragestellungen zu Webarchitekturen innerhalb von Projekten genutzt werden, um Architekturvorschläge auf Ebene der *Makroarchitekturen* abzuleiten. Die Erstellung dieses Kriterienkatalogs erfolgte durch eine sorgfältige Literaturrecherche, praxisorientierte Experteninterviews bei der Firma XITASO und durchgeführte Experimente. Bei den Experimenten wurden drei Software Prototy-

pen unter Verwendungen passender *Frameworks* für die jeweiligen Architekturen entwickelt (siehe Kap. 5: „Erläuterung der durchgeführten Experimente“).

Interpretation der Ergebnisse der Literaturrecherche

Die Literaturrecherche stützte sich auf eine Vielzahl unterschiedlicher Quellen, um eine umfassende Übersicht über die relevanten Kriterien für Architekturen zu schaffen. Dabei fiel besonders auf, dass zuvor keine einheitliche Zusammenstellung dieser Kriterien existierte. Der entwickelte Kriterienkatalog leistet somit einen signifikanten Beitrag zu architektonischen Untersuchungen im Bereich der Webarchitekturen, indem er eine fundierte Sammlung von Kriterien mit Beurteilungen für die Multi-Page-Architektur, Single-Page-Architektur und die spezielle Fullstack-Architektur (in dieser Arbeit: *Blazor Server*) bereitstellt.

Interpretation der Ergebnisse der Experteninterviews

Die praxisorientierten Experteninterviews bei der Firma XITASO dienten dazu, den Kriterienkatalog zu optimieren, indem er drei Experten (siehe Tabelle 12 in Kapitel 4.2: „Darstellung des Ablaufs der Experteninterviews“) von der Firma XITASO vorgestellt wurde. In diesen Prozess floss die langjährige Erfahrung der Experten im Bereich Webentwicklung in die Entwicklung des Kriterienkatalogs ein. Während der Interviews stellte sich heraus (siehe Kap. 4.2: „Ergebnisse der durchgeführten Experteninterviews“), dass die Experten die Idee des Kriterienkatalogs sehr positiv bewerteten. Sie äußerten, dass sie es kaum erwarten können, diesen in der Praxis einzusetzen, um zu Beginn eines Webprojekts eine solide Diskussionsgrundlage zu haben. Obwohl bereits Instrumente zur Auswahl der passenden Architektur verwendet werden, betonten die Experten, dass der Kriterienkatalog einen wertvollen Beitrag leisten würde, insbesondere durch die zentrale Sammlung der relevanten Informationen.

Interpretation der Ergebnisse der Experimente

Im Rahmen der Experimente wurden drei Software Prototypen für einen zuvor definierten Anwendungsfall (siehe Kapitel 5.1: „Erklärung der Anforderungen an die Experimente“) entwickelt. Dabei wurden folgende *Frameworks* für die Entwicklung der Prototypen in den unterschiedlichen Webarchitekturen genutzt:

- Multi-Page-Architektur: *ASP.NET Core 7.0* Model-View-Controller
- Single-Page-Architektur: *Vue 3.3* und *ASP.NET Core 7.0*
- Fullstack-Architektur: *Blazor Server*

Ziel der Experimente war es, den Kriterienkatalog auf die Probe zu stellen und weiter zu optimieren. Anzumerken ist, dass die Ergebnisse der Experimente (siehe Kap. 6: „Ergebnis der drei Experimente“) auf Prototypen basierten und nicht auf Endanwendungen. In der Praxis sind webbasierte Systeme oft umfangreicher, weshalb berücksichtigt werden muss, dass echte Endanwendungen möglicherweise unterschiedliche Ergebnisse geliefert hätten. Es ist ebenfalls von Bedeutung zu beachten, dass während der Durchführung der Experimente dieselben Anforderungen in allen drei Software-Prototypen umgesetzt wurden, wobei die Zeit für jede Umsetzung gemessen wurde. Trotz der unterschiedlichen Reihenfolge, in der die Anforderungen in den Software-Prototypen umgesetzt wurden (siehe Kap. 5.1: „Erklärung der Anforderungen an die Experimente“), muss berücksichtigt werden, dass die gemessenen Zeiten davon beeinflusst wurden.
Dennoch war die Durchführung der Experimente aus Autorsicht entscheidend, um zu überprüfen, ob die Ableitung des Kriterienkatalogs im Hinblick auf die am besten geeignete Architektur korrekt war. Die Ableitung wurde durch die Experimente weitgehend bestätigt, dennoch wurden Anpassungen vorgenommen, um den Kriterienkatalog zu vervollständigen. Die Anpassungen waren nötig, da durch den Einsatz der Chrome-Erweiterung *Lighthouse* Abweichungen in den Kriterien „Benutzererfahrung“ und „Performanz“ bei den drei Experimenten festgestellt wurden und diese eingearbeitet werden mussten (siehe Kap. 6.5: „Einarbeitung

der Ergebnisse in den Kriterienkatalog“).

Im anschließenden Kapitel 7.2 („Stärken und Schwächen der Bachelorarbeit“) werden die Schritte des Ablaufplans (siehe Abbildung 2 in Kapitel 1.4: „Beschreibung des Vorgehens“) erneut betrachtet, wobei sowohl die Stärken als auch die Schwächen der Arbeit hervorgehoben werden.

7.2 Stärken und Schwächen der Bachelorarbeit

Im vorherigen Kapitel (Kap. 7.1: „Interpretation der Ergebnisse“) wurden die Ergebnisse der vorliegenden Arbeit schrittweise interpretiert, um ihre Bedeutung für architektonische Untersuchungen im Bereich der Webarchitekturen zu verdeutlichen. Im Folgenden werden nun Stärken und Schwächen der Arbeit aufgezeigt, die wichtig und erwähnenswert sind.

Stärken der Arbeit

Die Bachelorarbeit konzentrierte sich darauf, einen abstrakten Blickwinkel auf die Multi-Page-Architektur, die Single-Page-Architektur und die Fullstack-Architektur (in dieser Arbeit: *Blazor Server*) zu werfen und diese Ansätze zu vergleichen, um die Forschungsfrage „Ist die Trennung von *Frontend* und *Backend* in webbasierten Systemen noch zeitgemäß?“ zu beantworten. Hierbei wurde ein Kriterienkatalog auf Grundlage eines Ablaufplans (siehe Abbildung 2 in Kapitel 1.4: „Beschreibung des Vorgehens“) entwickelt, wobei **solide wissenschaftliche Methoden** wie Literaturrecherche, Experteninterviews und Experimente verwendet wurden. Die **Ergebnisse** der verschiedenen Methodiken **wurden klar und übersichtlich in den Kapiteln dargestellt**. Darüber hinaus **bietet der entwickelte Kriterienkatalog eine übersichtliche, vergleichbare, transparente und objektive Darstellung der verschiedenen Kriterien und ihrer qualitativen Erfüllung durch die Multi-Page-Architektur, die Single-Page-Architektur und die**

Fullstack-Architektur. Diese Vorteile des Kriterienkatalogs wurden bereits detailliert in Kapitel 3.3 („Darstellung der Vorteile durch den Kriterienkatalog“) aufgezeigt.

Schwächen der Arbeit

Neben den beschriebenen Stärken weist die Bachelorarbeit auch Schwächen auf, die berücksichtigt werden müssen. Die Arbeit hat einen abstrakten Blickwinkel auf die Multi-Page-Architektur, die Single-Page-Architektur und die Fullstack-Architektur (in dieser Arbeit: *Blazor Server*) geworfen. Als Vertreter für die Fullstack-Architektur wurde *Blazor Server* ausgewählt. Es ist wichtig zu betonen, dass **andere Fullstack-Architekturen** wie zum Beispiel *Vaadin* oder *Remix.run* **völlig andere Vor- und Nachteile aufweisen und daher zu ganz anderen Ergebnissen führen könnten**. Daher ist die Untersuchung für andere *Frameworks* ratsam und wird auch in Kapitel 8.3 („Ausblick“) als Gegenstand möglicher Folgearbeiten aufgezeigt.
Ebenso ist zu beachten, dass in der Praxis viele Nachteile z.B. der Single-Page-Architektur im Bereich der Suchmaschinenoptimierung durch zusätzliche Technologien und Erweiterungen behoben werden können, wodurch sie nutzbar werden. Diese zusätzlichen **Technologien und Erweiterungen, die viele Nachteile der Architekturen beheben können, wurden nicht gesondert erwähnt und berücksichtigt**. Dennoch ist es wichtig an dieser Stelle darauf hinzuweisen, dass sie existieren und die verschiedenen Architekturen verbessern können. Eine vertiefte Untersuchung der Technologien und Erweiterungen für die unterschiedlichen Webarchitekturen ist also ratsam, falls die Erfüllung einzelner Kriterien durch die Webarchitekturen in Frage gestellt wird.

7.3 Zusammenfassung

In Kapitel 7.1 („Interpretation der Ergebnisse“) wurden die Ergebnisse der Bachelorarbeit, insbesondere der Literaturrecherche, der praxisorientierten Expertenin-

terviews bei der Firma XITASO und der durchgeführten Experimente, im Hinblick auf ihre Bedeutung für architektonische Untersuchungen im Bereich der Webarchitekturen interpretiert. Im Kontext der Literaturrecherche wurde betont, dass der entwickelte Kriterienkatalog eine solide Sammlung von Kriterien mit qualitativen Bewertungen für die Multi-Page-Architektur, die Single-Page-Architektur und die spezifische Fullstack-Architektur (in dieser Arbeit: *Blazor Server*) bereitstellt. Für die Experteninterviews wurde festgestellt, dass bei der Firma XITASO zwar bereits Instrumente zur Beurteilung geeigneter Architekturen im Einsatz sind, der Kriterienkatalog jedoch dazu beitragen könnte, Diskussionsgrundlagen leicht abzuleiten und die Entscheidungsfindung zu unterstützen. Hinsichtlich der Experimente wurde darauf hingewiesen, dass diese dazu dienten, den Kriterienkatalog zu prüfen, wobei zu beachten ist, dass lediglich Prototypen und keine Endanwendungen erstellt wurden.

In Kapitel 7.2 („Stärken und Schwächen der Bachelorarbeit“) wurden die Stärken und Schwächen der Arbeit aufgezeigt. Zu den Stärken zählen die Anwendung solider wissenschaftlicher Methoden, die klare und übersichtliche Darstellung der Ergebnisse sowie die Vorteile des entwickelten Kriterienkatalogs, wie bereits in Kapitel 3.3 („Darstellung der Vorteile durch den Kriterienkatalog“) erläutert. Hinsichtlich der Schwächen wurde betont, dass *Blazor Server* als Repräsentant der Fullstack-Architektur ausgewählt wurde, während andere Fullstack-Architekturen wie *Vaadin* oder *Remix.run* unterschiedliche Vor- und Nachteile aufweisen könnten. Zudem wurde darauf hingewiesen, dass es bereits viele Technologien und Erweiterungen gibt, die potenzielle Nachteile der Architekturen, wie die Suchmaschinenoptimierung bei der SPA, ausgleichen können. Diese wurden in dieser Bachelorarbeit nicht speziell erwähnt und berücksichtigt, da der Fokus auf einem abstrakten Blickwinkel lag.

Kapitel 8

Fazit

Im achten Kapitel werden die in den vorherigen Kapiteln aufgezeigten und diskutierten Ergebnisse der Arbeit genutzt, um die Forschungsfrage „Ist die Trennung zwischen *Frontend* und *Backend* in webbasierten Systemen noch zeitgemäß?“ aus Kapitel 1.2.1 („Konkretisierung der Forschungsfrage“) zu beantworten. Zugleich wird die Annahme, dass eine strikte Trennung von *Frontend* und *Backend* zu unnötiger Komplexität führt, aus Kapitel 1.2.2 („Aufstellung der Annahme“) bewertet. Nach dieser Analyse folgt ein Ausblick auf weitere Forschungsarbeiten und zukünftige Entwicklungen im Bereich der Webanwendungsarchitekturen.

8.1 Antwort auf die Forschungsfrage aus Kapitel 1.2.1

Zu Beginn dieser Arbeit wurde die Problemstellung (siehe Kapitel 1.1: „Ausgangslage und Unternehmen“) dargelegt, die die Grundlage für die Forschungsfrage „**Ist die Trennung zwischen *Frontend* und *Backend* in webbasierten Systemen noch zeitgemäß?**“ (siehe Kapitel 1.2.1: „Konkretisierung der Forschungsfrage“) bildete. Im Verlauf der Bachelorarbeit wurde ein Kriterienkatalog (siehe Anhang F: „Finale Version des Kriterienkatalogs nach den Experimenten“) anhand eines zuvor definierten Ablaufplans (siehe Abbildung 2 in Kapitel 1.4: „Beschreibung des Vorgehens“) entwickelt, um diese Forschungsfrage zu beantwor-

ten.

Der entwickelte Kriterienkatalog verdeutlicht, dass unterschiedliche Webarchitekturen für unterschiedliche architektonische Problemstellungen geeignet sind. Somit lässt sich die Forschungsfrage dank des Kriterienkatalogs mit einem klaren „**Es kommt darauf an, welche architektonische Problemstellung vorliegt**“ beantworten. Es ist wichtig zu betonen, dass der Kriterienkatalog zukünftig in Webprojekten eingesetzt werden kann, um besser zu beurteilen, für welche Anwendungsfall eine strikte Trennung von *Frontend* und *Backend* notwendig ist und wann nicht. Aufgrund des Kataloges können Interessierte selbst Wertungen einzelner Aspekte nivellieren und für konkrete Aufgabenstellungen anpassen.

Besonders hervorzuheben ist jedoch, dass die Fullstack-Architektur aus Sicht des Autors einen zukunftsfähigen Ansatz für die schnelle und effiziente Entwicklung von Webanwendungen darstellt. Die vorliegende Bachelorarbeit und der zukünftige Einsatz des entwickelten Kriterienkatalogs sind von großer Bedeutung, um mehr Entwickler neben den etablierten Webarchitekturen wie die Multi-Page-Architektur und die Single-Page-Architektur auf die Vorteile der Fullstack-Architektur aufmerksam zu machen. Dazu gehören die schnelle Entwicklungszeit, ein geringeres benötigtes Fachwissen im *Frontend* und eine gute Performanz beim Laden von Inhalten (siehe Tabelle 6 in Kap. 2.4: „Erklärung der Fullstack-Architektur ohne direkte Trennung von *Frontend* und *Backend*“). Eine aktuelle Statistik (vgl. Stack Overflow 2023), die bereits in Kapitel 1.1 („Ausgangslage und Unternehmen“) erwähnt wurde und die meistgenutzten Web-*Frameworks* des Jahres 2023 auflistet, verdeutlicht, dass Blazor bereits von 4,88% der Entwickler weltweit verwendet wird. Diese Zahl wird voraussichtlich weiter steigen und die Beliebtheit verschiedener *Frameworks*, die die Fullstack-Architektur implementieren, wird zunehmen.

Im nächsten Kapitel (Kap. 8.2: „Beurteilung der Annahme aus Kapitel 1.2.2“) wird die Annahme aus Kapitel 1.2.2 („Aufstellung der Annahme“) beurteilt, die neben der Forschungsfrage für diese Arbeit aufgestellt wurde.

8.2 Beurteilung der Annahme aus Kapitel 1.2.2

Neben der Forschungsfrage, die in Kapitel 1.2.1 („Konkretisierung der Forschungsfrage“) aufgestellt und im vorherigen Kapitel 8.1 („Antwort auf die Forschungsfrage aus Kapitel 1.2.1“) beantwortet wurde, stellte das Kapitel 1.2.2 („Aufstellung der Annahme“) die formulierte Annahme dar. Die aufgestellte **Annahme lautete, dass eine strikte Trennung von *Frontend* und *Backend* zu unnötiger Komplexität in der Entwicklung in einigen Projekten führen kann. Es wurde daher als wichtig erachtet zu prüfen, ob die Fullstack-Architektur ausreichen könnte, um den Entwicklungsaufwand zu verringern und die Komplexität zu reduzieren**.
In der Bachelorarbeit wurden drei Experimente für die drei unterschiedlichen Architekturansätze (siehe Kapitel 5: „Erläuterung der durchgeführten Experimente“) durchgeführt. Ein Anwendungsfall (siehe Kapitel 5: „Erklärung der Anforderungen an die Experimente“) wurde inszeniert, um sicherzustellen, dass jedes Experiment dieselben Anforderungen erfüllte und somit Vergleichbarkeit gewährleistet war. Die Ergebnisse der Experimente wurden übersichtlich in Tabelle 22 in Kapitel 6.4 („Ergebnisse der Experimente in der Übersicht“) dargestellt. Es fiel auf, dass die Entwicklungszeit für das Experiment der Fullstack-Architektur am geringsten war und am wenigsten Fachwissen benötigt wurde.
Die aufgestellte **Annahme bestätigte sich also während der Untersuchung**. Die Fullstack-Architektur ohne strikte Trennung von *Frontend* und *Backend* bietet eine geringere Komplexität bei der Entwicklung und einen geringeren Entwicklungsaufwand im Vergleich zur Multi-Page-Architektur oder zur Single-Page-Architektur. Dennoch ist zu berücksichtigen, dass die Übersichtlichkeit in dieser Architektur beeinträchtigt wird. Aufgrund der Vermischung von *Frontend* und *Backend* haben Entwickler Schwierigkeiten, die Übersicht zu behalten und diese Teile zu unterscheiden.
Im anschließenden Kapitel 8.3 („Ausblick“) werden die Ergebnisse und Erkenntnisse der Bachelorarbeit als Grundlage genutzt, um einen Ausblick auf weitere Entwicklungen im Bereich der Webarchitekturen zu geben. Zudem werden poten-

zielle Forschungsbereiche für zukünftige Arbeiten vorgestellt.

8.3 Ausblick

Dieses Kapitel bietet einen Ausblick, der Entwicklungen im Bereich der Webanwendungsarchitekturen erörtert, um so einen Vorausblick auf mögliche weitere Forschungsarbeiten in diesem Fachgebiet zu geben.
In dieser Arbeit wurde *Blazor Server* von Microsoft als Repräsentant der Fullstack-Architektur verwendet. Es ist wichtig zu betonen, dass die gesamte Recherche zu *Blazor Server* auf der siebten Version der *.NET-Umgebung* von Microsoft basierte, wie bereits im Vorwort erwähnt. Während der Bearbeitung dieser Bachelorarbeit veröffentlichte Microsoft am 14.11.2023 die achte Version der *.NET-Umgebung*, in der *Blazor Server* verbessert wurde, um den bekannten Nachteilen, wie der ständig erforderlichen stabilen Netzwerkverbindung, entgegenzuwirken (vgl. Schwichtenberg 2023). Eine **mögliche Folgearbeit** könnte darin bestehen, **den entwickelten Kriterienkatalog** (siehe Anhang F: „Finale Version des Kriterienkatalogs nach den Experimenten“) dieser Arbeit **auf die neueste Version der *.NET-Umgebung* zu aktualisieren, um einen aktuellen Vergleich zwischen den Architekturen zu ermöglichen**.
Eine weitere **mögliche Folgearbeit wäre die Auswahl eines anderen Repräsentanten für die Fullstack-Architektur, wie beispielsweise *Vaadin* oder *Remix.run***, um diesen anstelle von *Blazor Server* zu untersuchen und mit der Multi-Page-Architektur und Single-Page-Architektur zu vergleichen. Zudem könnte der entwickelte Kriterienkatalog (siehe Anhang F: „Finale Version des Kriterienkatalogs nach den Experimenten“) um weitere Kriterien erweitert werden, um eine noch spezifischere Diskussionsgrundlage für architektonische Problemstellungen im Bereich der Webarchitekturen zu schaffen. Sinnvolle weitere Kriterien wären zum Beispiel:

- Testing: Die Evaluierung des Testaufwands könnte die Fähigkeit jeder Architektur, sowohl Modul- als auch End-to-End-Tests effizient durchzufüh-

ren, bewerten.

- Community-Unterstützung: Dieses Kriterium könnte die Stärke der Entwicklergemeinschaft jeder Architektur anhand von Aktivität in Foren, der Anzahl aktiver Beitragenden zur Codebasis und der Qualität der verfügbaren Dokumentation und Ressourcen bewerten.

Eine **weitere Thematik, die ebenfalls als Gegenstand künftiger Arbeiten dienen könnte, wäre die Verschiebung von Code auf den Server**, wie es Microsofts *Framework Blazor Server* tut. Das React-Team arbeitet bereits an einem Ansatz, der ebenfalls wieder vermehrt auf die Serverseite setzt (vgl. Schmitt 2022). Eine Folgearbeit, die diese Entwicklungen untersucht, könnte auf den Ergebnissen dieser Bachelorarbeit aufbauen.

Anhang A: Erste Version des Kriterienkatalogs nach der Literaturrecherche

Kriterienkatalog für die Architekturauswahl

Mit diesem Kriterienkatalog sollen Software-Architekten anhand der vorliegenden Problemstellung einen webbasierten Architekturvorschlag (MPA, SPA, Fullstack-Architektur) ableiten können. Entstanden ist das Ganze im Rahmen der Bachelorarbeit „Ist die Trennung zwischen Frontend und Backend in webbasierten Systemen noch zeitgemäß?“ von Michael Mertl. Die Kriterien sind dabei aus einer umfangreichen und internationalen Literaturrecherche zu den verschiedenen Ansätzen entstanden, indem die Vor- und Nachteile dieser ermittelt wurden und dementsprechend als Grundlage dienten für die vergebenen Punkte.

Funktionsweise:
Bitte nur die Kriterien ankreuzen und gewichten, die die vorliegende Problemstellung erfordern. Am Ende die Gesamtpunktzahl ausrechnen und die Architektur mit der höchsten Punktzahl ist diejenige, die am besten geeignet ist, für die vorliegende Problemstellung. Jede Architektur hat für jedes Kriterium vorweg eine Punktzahl bekommen (1 = nicht/schlecht erfüllt, 3 = teilweise erfüllt, 6 = vollständig erfüllt), die aus der Literaturrecherche anhand der Vor- und Nachteile entstanden sind. Diese verrechnet mit dem Gewicht ergibt die Punktzahl für das jeweilige Kriterium. Die Gewichte, die für die einzelnen Kriterien vergeben werden, müssen am Ende 100% entsprechen, um eine mathematisch korrekte Berechnung zu gewährleisten.

Kriterium	Multi-Page-Architektur	Single-Page-Architektur	Fullstack-Architektur (orientiert an Blazor Server)	Gewicht	Gewollt?
Autorisierung mit OAuth 2.0 -> ist die Umsetzung der Autorisierung mit OAuth 2.0 in der Architektur einfach?	Die Autorisierung muss nur in einer Anwendung umgesetzt werden, da bei der MPA nur eine gebaut wird. -> Punktzahl: 6	Die Autorisierung muss in beiden Anwendungen (dem getrennten Frontend und Backend) umgesetzt werden. So entsteht ein Mehraufwand. -> Punktzahl: 1	Die Autorisierung muss nur in einer Anwendung umgesetzt werden, da bei Blazor Server nur eine gebaut wird. -> Punktzahl: 6	Gewicht: MPA: SPA: Fullstack:	☐
Benutzererfahrung -> bietet die Architektur eine angenehme Benutzererfahrung beim Laden neuer Inhalte?	Jede Benutzerinteraktion führt zum neuen Laden der gesamten Seite, welche die neuen Inhalte dann bereitstellt. -> Punktzahl: 1	Benutzerinteraktionen lösen keine ganzen Seitenaufrufe mehr aus und es können Animationen genutzt werden, beim Warten auf neue Inhalte. -> Punktzahl: 6	Benutzerinteraktionen lösen keine ganzen Seitenaufrufe mehr aus und es können Animationen genutzt werden, beim Warten auf neue Inhalte. -> Punktzahl: 6	Gewicht: MPA: SPA: Fullstack:	☐
Browseranforderung -> muss der Browser JavaScript fähig sein für die Webanwendung?	Nein, da die MPA auch komplett ohne JS funktionieren würde. -> Punktzahl: 6	Ja -> Punktzahl: 1	Ja -> Punktzahl: 1	Gewicht: MPA: SPA: Fullstack:	☐

2

Entwicklererfahrung -> welches Fachwissen wird für die Architektur benötigt?	Es muss nur eine Anwendung entwickelt werden. In dieser wird aber Fachwissen im Frontend (HTML, CSS und JS) und Backend (z.B. C#) benötigt. -> Punktzahl: 3	Es werden zwei Anwendungen entwickelt. Für die Kommunikation dazwischen muss zusätzlich eine API entwickelt werden. Es wird also Fachwissen im Frontend (HTML, CSS und JS) und Backend (z.B. C#) benötigt. -> Punktzahl: 3	Es muss nur eine Anwendung entwickelt werden. In dieser wird teilweise Wissen im Frontend und Backend benötigt. Hier wird kein JavaScript benötigt, lediglich Wissen in C#, HTML und CSS. -> Punktzahl: 6	Gewicht: MPA: SPA: Fullstack:	☐
Entwicklungszeit -> wie lange dauert die Entwicklung einer Webanwendung mit dieser Architektur?	In der Regel schneller als bei der SPA, da nur eine Anwendung entwickelt werden muss und auf die nötige API für die Kommunikation wie bei einer SPA verzichtet werden kann. Durch die Codetrennung von Frontend und Backend, aber in der Regel	In der Regel langsamer als bei der MPA und der Fullstack-Architektur, da zwei Anwendungen geschrieben werden und eine API für die Kommunikation entwickelt werden muss.	In der Regel schneller als bei der SPA, da nur eine Anwendung entwickelt werden muss und auf die nötige API für die Kommunikation wie bei einer SPA verzichtet werden kann.	Gewicht: MPA: SPA: Fullstack:	☐

3

	langsamer als die Fullstack-Architektur. -> Punktzahl: 3	 -> Punktzahl: 1	 -> Punktzahl: 6		
Erweiterbarkeit -> ist die Architektur in der Zukunft leicht erweiterbar (in Bezug auf neue Features)?	Leicht erweiterbar, Frontend und Backend sind aber etwas stärker verknüpft, da diese in einer Anwendung sind. -> Punktzahl: 6	Leicht erweiterbar, Frontend und Backend können isoliert erweitert werden, da zwei Anwendungen vorliegen. -> Punktzahl: 6	Es wird eine Anwendung entwickelt, in der keine saubere Codetrennung erzwungen wird, wie bei der MPA oder SPA, dies kann die Erweiterbarkeit erschweren. -> Punktzahl: 3	Gewicht: MPA: SPA: Fullstack:	☐
Offlinefähigkeit -> ist die Architektur ohne Erweiterungen in der Lage, die Webanwendung auch offline nutzbar zu machen?	Nein -> Punktzahl: 1	Ja (solange keine neue Daten-Anfrage angestoßen wird) -> Punktzahl: 6	Nein -> Punktzahl: 1	Gewicht: MPA: SPA: Fullstack:	☐
Performanz -> wie schnell ist das initiale Laden der Webanwendung?	Lädt initial schnell, da nur eine HTML-Seite (diese kann statisch oder dynamisch sein) geladen werden muss.	Erster Ladezyklus dauert sehr lang, da die gesamte Anwendung geladen wird.	Lädt initial schnell, da nur eine statische HTML-Seite und zusätzlich eine JavaScript-Bibliothek von Microsoft (diese ist verantwortlich für die Übertragung der Benutzer-	Gewicht: MPA: SPA: Fullstack:	☐

4

			interaktionen an den Server) übertragen werden muss. -> Punktzahl: 6		
	-> Punktzahl: 6	-> Punktzahl: 1			
Performanz -> wie schnell ist das Nachladen neuer Inhalte (ohne die Verwendung von zusätzlichen Caching-Mechanismen)?	Eine Anfrage nach neuen Daten fordert eine neue HTML-Seite (wieder statisch oder dynamisch) vom Server an. Der Server stellt bzw. rendert die neue HTML-Seite und gibt diese an den Client zum Anzeigen zurück. Lädt genauso schnell wie initial, aber langsamer als die SPA oder die Fullstack-Architektur. -> Punktzahl: 3	Eine Anfrage nach neuen Daten stößt eine Anfrage (z.B. HTTP-Anfrage) an den Server an. Dieser sendet dann eine Antwort (z.B. HTTP-Antwort) mit den benötigten Daten zurück. Die neuen Daten werden dann am Client verarbeitet und es wird nur der Seitenteil neu gerendert, der von den neuen Daten betroffen ist. -> Punktzahl: 6	Eine Anfrage nach neuen Daten wird von der JavaScript-Bibliothek von Microsoft über die SignalR dem Server gemeldet. Dieser beschafft sich die neuen Daten und tauscht den betroffenen Teil der HTML-Seite auf dem virtuellen Abbild des Servers aus. Dieser geänderte Teil der Seite wird über die SignalR-Verbindung an den Client geschickt, wo er ebenfalls ausgetauscht wird und somit der neue Inhalt einem Anwender angezeigt wird. -> Punktzahl: 6	Gewicht: MPA: SPA: Fullstack:	☐

5

Sicherheit -> wie anfällig ist die Architektur für die ungewollte Offenlegung von vertraulichen Daten durch einen Entwickler?	Die MPA ist wie Blazor Server in der Lage, die vertraulichen Daten auf dem Server zu speichern. Ebenfalls liegt hier beim Entwickeln eine strikte Trennung zwischen Frontend und Backend vor. Daher ist es beim Entwickeln leichter, keine vertraulichen Daten ungewollt am Client zu verarbeiten oder offenzulegen. -> Punktzahl: 6	Die SPA speichert viele Daten am Client und ist daher anfällig für bösartiges JavaScript, welches durch Eingaben eingeschleust werden kann. Der Entwickler muss also beim Entwickeln aufpassen, dass er keine vertraulichen Daten am Client preisgibt. -> Punktzahl: 1	Blazor Server speichert im Gegensatz zu der SPA die vertraulichen Daten auf dem Server. Durch die Vermischung von Frontend und Backend, kann es jedoch vorkommen, dass beim Entwicklungs-prozess unbeabsichtigt vertrauliche Daten auf dem Client verarbeitet werden, da die Anwendung unübersichtlich werden kann. -> Punktzahl: 3	Gewicht: MPA: SPA: Fullstack:	☐
Serverbelastung -> wie stark wird der Server belastet (ohne die Verwendung von zusätzlichen Caching-Mechanismen)?	Hoch, da jede Anfrage nach neuen Daten eine neue HTML-Seite anfordert. Diese muss jedes Mal neu vom Server bereitgestellt bzw.	Niedrig, da jede Anfrage nach neuen Daten nur die genau benötigte Datenmenge vom Sever anfordert und dieser somit nicht so stark belastet wird	Mittel, da jede Anfrage nach neuen Daten wie bei der SPA nur die genau benötigte Datenmenge vom Server anfordert. Die Grundbelastung des Servers ist jedoch höher	Gewicht: MPA: SPA: Fullstack:	☐

6

	generiert werden. Der Zustand und die Logik der MPA können am Client und am Server (oder auch in Kombination) verarbeitet werden. Dadurch kann der Server dann zusätzlich belastet werden. -> Punktzahl: 1	als bei der MPA. Der Zustand und die Logik in der SPA werden am Client verarbeitet und belastet somit den Server nicht. -> Punktzahl: 6	als bei der MPA oder der SPA, da bei Blazor Server die Logik und die Zustände am Server verwaltet werden müssen und diesen somit belasten. -> Punktzahl: 3		
Skalierbarkeit -> ist die Architektur in der Lage horizontal oder vertikal zu skalieren, um mit steigender Anzahl von Benutzern und Anfragen umzugehen?	Bei der MPA kann der Client oder auch der Server die Zustands-verwaltung übernehmen. Hier besteht also die Möglichkeit eine horizontale oder auch eine vertikale Skalierung problemlos durchzuführen.	Bei der SPA übernimmt der Client die Zustands-verwaltung, somit kann der Server zustandslos betrieben werden. Hier besteht also die Möglichkeit eine horizontale oder auch eine vertikale Skalierung problemlos durchzuführen.	Bei Blazor Server übernimmt der Server die Zustands-verwaltung. Eine vertikale Skalierung ist problemlos möglich. Eine horizontale Skalierung ist jedoch aufgrund dessen und der nötigen Verwaltung der SignalR-Verbindungen über mehrere Serverinstanzen hinweg, eine	Gewicht: MPA: SPA: Fullstack:	☐

7

	-> Punktzahl: 6	-> Punktzahl: 6	größere Herausforderung als bei der MPA oder SPA. -> Punktzahl: 3		
Suchmaschinen-optimierung (SEO) -> ist die Architektur in der Lage, die Webanwendung für SEO nutzbar zu machen?	Es liegen viele HTML-Seiten vor, deshalb funktioniert die SEO sehr gut. -> Punktzahl: 6	Es liegt nur eine HTML-Seite vor, deshalb funktioniert SEO nicht ohne weitere Technologien. -> Punktzahl: 1	Es liegt nur eine HTML-Seite vor, deshalb funktioniert SEO nicht ohne weitere Technologien. -> Punktzahl: 1	Gewicht: MPA: SPA: Fullstack:	☐
Gesamtergebnis				100%	

8

Anhang B: Handlungsablauf der Experteninterviews

Handlungsablauf der Experteninterviews

Interviewer:	Michael Mertl
Interviewte Person:	
Firma:	
Firmenadresse:	
Position:	
E-Mail:	
Expertise:	

Einführung

- Ausgangslage erläutern mit Ableitung auf Titel der Bachelorarbeit
- Multi-Page-Architektur, Single-Page-Architektur und Fullstack-Architektur vorstellen (anhand selbst erstellter Darstellungen)
- Einführung in den Kriterienkatalog geben mit Ableitung auf die Kriterien
- Aufzeigen, dass es im Interview um die Beurteilung und Bewertung der verschiedenen Kriterien geht
- Der Experte soll dafür jeden Ansatz für das Kriterium in Form einer Likert-Skala bewerten
- Danach werden dem Experten einige konkrete Fragen zu ausgewählten Kriterien gestellt
- Zum Schluss hat der Experte noch die Möglichkeit Feedback zu geben

Durchführung

Multi-Page-Architektur / Single-Page-Architektur / Fullstack-Architektur (orientiert an Blazor Server)

Kriterium	Stimme überhaupt nicht zu	Stimme nicht zu	Stimme weder zu noch lehne ich ab	Stimme zu	Stimme voll und ganz zu
Autorisierung mit OAuth 2.0 -> Die Umsetzung der Autorisierung mit OAuth 2.0 ist mit der Architektur einfach.					
Benutzererfahrung -> Die Architektur bietet eine angenehme Benutzererfahrung beim Laden neuer Inhalte.					
Browseranforderung -> Die entstehende Webanwendung benötigt einen JavaScript-fähigen Browser.					
Entwicklererfahrung -> Die Architektur erfordert Fachwissen im Frontend und Backend.					
Entwicklungszeit -> Die Entwicklungsdauer einer Webanwendung ist mit der Architektur schnell.					
Erweiterbarkeit -> Die Architektur ist in der Zukunft leicht erweiterbar (in Bezug auf neue Features).					

2

Offlinefähigkeit -> Die Architektur macht eine Webanwendung auch offline nutzbar.					
Performanz -> Die entstehende Webanwendung lädt initial schnell.					
Performanz -> Die entstehende Webanwendung lädt neue Inhalte schnell nach.					
Sicherheit -> Die Architektur ist nicht anfällig für die ungewollte Offenlegung von vertraulichen Daten durch einen Entwickler.					
Serverbelastung -> Der Server wird nicht stark belastet.					
Skalierbarkeit -> Die Architektur ist in der Lage horizontal zu skalieren, um mit steigender Anzahl von Benutzern und Anfragen umzugehen.					
Skalierbarkeit -> Die Architektur ist in der Lage vertikal zu skalieren, um mit steigender Anzahl von Benutzern und Anfragen umzugehen.					
Suchmaschinenoptimierung (SEO) -> Die Architektur macht die entstehende Anwendung für SEO nutzbar.					

3

Fragen

Wie oft kommt es in Ihren Augen vor, dass in einem Projekt keine Authentifizierung und Autorisierung benötigt wird? (Einfluss auf Relevanz von Kriterium: Autorisierung mit OAuth 2.0)

Das Kriterium „Benutzererfahrung“ bezieht sich nur auf die Erfahrung beim Laden neuer Inhalte. Gibt es in Ihren Augen noch andere Aspekte der jeweiligen Architekturen, die Einfluss auf eine angenehme „Benutzererfahrung“ haben? (Einfluss auf Kriterium: Benutzererfahrung)

Wie relevant halten Sie das Kriterium „Browseranforderung“ für die Bewertung der verschiedenen Ansätze? Macht man sich darüber heutzutage in einem neuen Projekt noch Gedanken, oder geht man davon aus, dass jeder Anwender JavaScript-fähige Browser nutzt? (Einfluss auf Kriterium: Browseranforderung)

4

Wie oft ist es Ihnen schon passiert, dass Sie ungewollte vertrauliche Daten offengelegt haben, indem Sie diese an einem ungewollten Punkt gespeichert haben bei der Entwicklung? (Einfluss auf Kriterium: Sicherheit)

Wie oft haben Sie persönlich schon ein Projekt entwickelt, bei dem die Skalierbarkeit keine große Rolle gespielt hat und eine feste Benutzerzahl bekannt war? (Einfluss auf Kriterium: Skalierbarkeit)

Fehlt Ihnen ein Kriterium, welches in Ihren Augen wichtig ist für den Vergleich der Architekturen?

5

Feedback

6

Anhang C: Zweite Version des Kriterienkatalogs nach den Experteninterviews

Alle Änderungen, die sich gegenüber der Vorversion (siehe Anhang A: „Erste Version des Kriterienkatalogs nach der Literaturrecherche“) ergeben haben, sind in dieser Version farblich (gelb) markiert.

Kriterienkatalog für die Architekturauswahl

Mit diesem Kriterienkatalog sollen Software-Architekten anhand der vorliegenden Problemstellung einen webbasierten Architekturvorschlag (MPA, SPA, Fullstack-Architektur) ableiten können. Entstanden ist das Ganze im Rahmen der Bachelorarbeit „Ist die Trennung zwischen Frontend und Backend in webbasierten Systemen noch zeitgemäß?“ von Michael Mertl. Die Kriterien sind dabei aus einer umfangreichen und internationalen Literaturrecherche zu den verschiedenen Ansätzen entstanden, indem die Vor- und Nachteile dieser ermittelt wurden und dementsprechend als Grundlage dienten für die vergebenen Punkte. Außerdem wurde eine praxisorientierte Qualitätsprüfung in Form von Experteninterview bei der Firma XITASO durchgeführt, um den Kriterienkatalog zu verbessern.

Funktionsweise:
Bitte nur die Kriterien ankreuzen und gewichten, die die vorliegende Problemstellung erfordern. Am Ende die Gesamtpunktzahl ausrechnen und die Architektur mit der höchsten Punktzahl ist diejenige, die am besten geeignet ist, für die vorliegende Problemstellung. Jede Architektur hat für jedes Kriterium vorweg eine Punktzahl bekommen (1 = nicht/schlecht erfüllt, 3 = teilweise erfüllt, 6 = vollständig erfüllt), die aus der Literaturrecherche anhand der Vor- und Nachteile entstanden sind und durch Experteninterviews evaluiert wurden. Diese verrechnet mit dem Gewicht ergibt die Punktzahl für das jeweilige Kriterium. Die Gewichte, die für die einzelnen Kriterien vergeben werden, müssen am Ende 100% entsprechen, um eine mathematisch korrekte Berechnung zu gewährleisten.

Kriterium	Multi-Page-Architektur	Single-Page-Architektur	Fullstack-Architektur (orientiert an Blazor Server)	Gewicht	Gewollt?
Autorisierung mit OAuth 2.0 -> ist die Umsetzung der Autorisierung mit OAuth 2.0 in der Architektur einfach?	Die Autorisierung muss nur in einer Anwendung umgesetzt werden, da bei der MPA nur eine gebaut wird. -> Punktzahl: 6	Die Autorisierung muss in beiden Anwendungen (dem getrennten Frontend und Backend) umgesetzt werden. So entsteht ein Mehraufwand. -> Punktzahl: 3	Die Autorisierung muss nur in einer Anwendung umgesetzt werden, da bei Blazor Server nur eine gebaut wird. -> Punktzahl: 6	Gewicht: MPA: SPA: Fullstack:	☐
Benutzererfahrung -> bietet die Architektur eine angenehme Benutzererfahrung beim Laden neuer Inhalte?	Jede Benutzerinteraktion führt zum neuen Laden der gesamten Seite, welche die neuen Inhalte dann bereitstellt. -> Punktzahl: 1	Benutzerinteraktionen lösen keine ganzen Seitenaufrufe mehr aus und es können Animationen genutzt werden, beim Warten auf neue Inhalte. -> Punktzahl: 6	Benutzerinteraktionen lösen keine ganzen Seitenaufrufe mehr aus und es können Animationen genutzt werden, beim Warten auf neue Inhalte. -> Punktzahl: 6	Gewicht: MPA: SPA: Fullstack:	☐
Benutzererfahrung -> bietet die Architektur eine angenehme Benutzererfahrung beim Betätigen von Browsertasten (z.B.	Bei der MPA liegen viele HTML-Seiten vor, daher stellt der Browser bereits von selbst sicher, dass die verschiedenen	Bei der SPA liegt nur eine HTML-Seite vor auf der mit JS navigiert wird. Das Standardverhalten des Browsers kann nicht direkt	Bei Blazor Server liegt dasselbe Problem wie bei der SPA vor. Die .NET-Umgebung bringt eine Klasse (namens „NavigationManager“) mit, die	Gewicht: MPA: SPA: Fullstack:	☐

2

Vorwärts/Rückwärts Schaltfläche)?	Browsertasten richtig funktionieren. -> Punktzahl: 6	genutzt werden und muss durch Programmierung nutzbar gemacht werden. -> Punktzahl: 1	es sehr einfach macht, das Standardverhalten des Browsers zu nutzen und richtig zu programmieren. -> Punktzahl: 3		
Browseranforderung -> muss der Browser JavaScript fähig sein für die Webanwendung?	Nein, da die MPA auch komplett ohne JS funktionieren würde. -> Punktzahl: 6	Ja -> Punktzahl: 1	Ja -> Punktzahl: 1	Gewicht: MPA: SPA: Fullstack:	☐
Entwicklererfahrung -> welches Fachwissen wird für die Architektur benötigt?	Es muss nur eine Anwendung entwickelt werden. In dieser wird aber Fachwissen im Frontend (HTML, CSS und JS) und Backend (z.B. C#) benötigt. -> Punktzahl: 3	Es werden zwei Anwendungen entwickelt. Für die Kommunikation dazwischen muss zusätzlich eine API entwickelt werden. Es wird also Fachwissen im Frontend (HTML, CSS und JS) und Backend (z.B. C#) benötigt. -> Punktzahl: 3	Es muss nur eine Anwendung entwickelt werden. In dieser wird teilweise Wissen im Frontend und Backend benötigt. Hier wird kein JavaScript benötigt, lediglich Wissen in C#, HTML und CSS. -> Punktzahl: 6	Gewicht: MPA: SPA: Fullstack:	☐

3

Entwicklungszeit -> wie lange dauert die Entwicklung einer Webanwendung mit dieser Architektur?	In der Regel schneller als bei der SPA, da nur eine Anwendung entwickelt werden muss und auf die nötige API für die Kommunikation wie bei einer SPA verzichtet werden kann. Durch die Codetrennung von Frontend und Backend, aber in der Regel langsamer als die Fullstack-Architektur. -> Punktzahl: 3	In der Regel langsamer als bei der MPA und der Fullstack-Architektur, da zwei Anwendungen geschrieben werden und eine API für die Kommunikation entwickelt werden muss. -> Punktzahl: 1	In der Regel schneller als bei der SPA, da nur eine Anwendung entwickelt werden muss und auf die nötige API für die Kommunikation wie bei einer SPA verzichtet werden kann. -> Punktzahl: 6	Gewicht: MPA: SPA: Fullstack:	☐
Erweiterbarkeit -> ist die Architektur in der Zukunft leicht erweiterbar (in Bezug auf neue Features)?	Leicht erweiterbar, Frontend und Backend sind aber etwas stärker verknüpft, da diese in einer Anwendung sind.	Leicht erweiterbar, Frontend und Backend können isoliert erweitert werden, da zwei Anwendungen vorliegen.	Es wird eine Anwendung entwickelt, in der keine saubere Codetrennung zwischen Frontend und Backend erzwungen wird, wie bei der MPA oder SPA, dies kann die Erweiterbarkeit erschweren.	Gewicht: MPA: SPA: Fullstack:	☐

4

	-> Punktzahl: 3	-> Punktzahl: 6	-> Punktzahl: 3		
Offlinefähigkeit -> ist die Architektur in der Lage, die Webanwendung auch offline nutzbar zu machen?	Nein -> Punktzahl: 1	Ja (solange keine neue Daten-Anfrage angestoßen wird) -> Punktzahl: 6	Nein -> Punktzahl: 1	Gewicht: MPA: SPA: Fullstack:	☐
Performanz -> wie schnell ist das initiale Laden der Webanwendung?	Lädt initial schnell, da nur eine HTML-Seite (diese kann statisch oder dynamisch sein) geladen werden muss. -> Punktzahl: 6	Erster Ladezyklus dauert sehr lang, da die gesamte Anwendung geladen wird. -> Punktzahl: 1	Lädt initial schnell, da nur eine statische HTML-Seite und zusätzlich eine JavaScript-Bibliothek von Microsoft (diese ist verantwortlich für die Übertragung der Benutzer-interaktionen an den Server) übertragen werden muss. -> Punktzahl: 6	Gewicht: MPA: SPA: Fullstack:	☐
Performanz -> wie schnell ist das Nachladen neuer Inhalte (ohne die Verwendung von zusätzlichen Caching-Mechanismen)?	Eine Anfrage nach neuen Daten fordert eine neue HTML-Seite (wieder statisch oder dynamisch) vom Server an. Der Server stellt bzw. rendert	Eine Anfrage nach neuen Daten stößt eine Anfrage (z.B. HTTP-Anfrage) an den Server an. Dieser sendet dann eine Antwort (z.B. HTTP-Antwort) mit	Eine Anfrage nach neuen Daten wird von der JavaScript-Bibliothek von Microsoft über die SignalR dem Server gemeldet. Dieser beschafft sich die neuen Daten und	Gewicht: MPA: SPA: Fullstack:	☐

5

	die neue HTML-Seite und gibt diese an den Client zum Anzeigen zurück. Lädt genauso schnell wie initial, aber langsamer als die SPA oder die Fullstack-Architektur. -> Punktzahl: 3	den benötigten Daten zurück. Die neuen Daten werden dann am Client verarbeitet und es wird nur der Seitenteil neu gerendert, der von den neuen Daten betroffen ist. -> Punktzahl: 6	tauscht den betroffenen Teil der HTML-Seite auf dem virtuellen Abbild des Servers aus. Dieser geänderte Teil der Seite wird über die SignalR-Verbindung an den Client geschickt, wo er ebenfalls ausgetauscht wird und somit der neue Inhalt einem Anwender angezeigt wird. -> Punktzahl: 6		
Sicherheit -> wie anfällig ist die Architektur für die ungewollte Offenlegung von vertraulichen Daten durch einen Entwickler?	Die MPA ist wie Blazor Server in der Lage, die vertraulichen Daten auf dem Server zu speichern. Ebenfalls liegt hier beim Entwickeln eine strikte Trennung zwischen Frontend und Backend vor. Daher ist es beim Entwickeln leichter, keine	Die SPA speichert viele Daten am Client und ist daher anfällig für bösartiges JavaScript, welches durch Eingaben eingeschleust werden kann. Der Entwickler muss also beim Entwickeln aufpassen, dass er keine vertraulichen Daten am Client preisgibt.	Blazor Server speichert im Gegensatz zu der SPA die vertraulichen Daten auf dem Server. Durch die Vermischung von Frontend und Backend, kann es jedoch vorkommen, dass beim Entwicklungs-prozess unbeabsichtigt vertrauliche Daten auf dem Client verarbeitet werden, da	Gewicht: MPA: SPA: Fullstack:	☐

6

	vertraulichen Daten ungewollt am Client zu verarbeiten oder offenzulegen. -> Punktzahl: 6	 -> Punktzahl: 1	die Anwendung unübersichtlich werden kann. -> Punktzahl: 3		
Serverbelastung -> wie stark wird der Server belastet (ohne die Verwendung von zusätzlichen Caching-Mechanismen)?	Hoch, da jede Anfrage nach neuen Daten eine neue HTML-Seite anfordert. Diese muss jedes Mal neu vom Server bereitgestellt bzw. generiert werden. Der Zustand und die Logik der MPA können am Client und am Server (oder auch in Kombination) verarbeitet werden. Dadurch kann der Server dann zusätzlich belastet werden.	Niedrig, da jede Anfrage nach neuen Daten nur die genau benötigte Datenmenge vom Sever anfordert und dieser somit nicht so stark belastet wird als bei der MPA. Der Zustand und die Logik in der SPA werden am Client verarbeitet und belastet somit den Server nicht. Dennoch stellt over- und under-fetching eine Herausforderung dar und muss vermieden werden, sonst wird der Server unnötig belastet.	Mittel, da jede Anfrage nach neuen Daten wie bei der SPA nur die genau benötigte Datenmenge vom Server anfordert. Die Grundbelastung des Servers ist jedoch höher als bei der MPA oder der SPA, da bei Blazor Server die Logik und die Zustände am Server verwaltet werden müssen und diesen somit belasten.	Gewicht: MPA: SPA: Fullstack:	☐

7

	-> Punktzahl: 1	-> Punktzahl: 3	-> Punktzahl: 3		
Skalierbarkeit -> ist die Architektur in der Lage horizontal oder vertikal zu skalieren, um mit steigender Anzahl von Benutzern und Anfragen umzugehen?	Bei der MPA kann der Client oder auch der Server die Zustands-verwaltung übernehmen. Hier besteht also die Möglichkeit eine horizontale oder auch eine vertikale Skalierung problemlos durchzuführen. -> Punktzahl: 6	Bei der SPA übernimmt der Client die Zustands-verwaltung, somit kann der Server zustandslos betrieben werden. Hier besteht also die Möglichkeit eine horizontale oder auch eine vertikale Skalierung problemlos durchzuführen. -> Punktzahl: 6	Bei Blazor Server übernimmt der Server die Zustands-verwaltung. Eine vertikale Skalierung ist problemlos möglich. Eine horizontale Skalierung ist jedoch aufgrund dessen und der nötigen Verwaltung der SignalR-Verbindungen über mehrere Serverinstanzen hinweg, eine größere Herausforderung als bei der MPA oder SPA. -> Punktzahl: 3	Gewicht: MPA: SPA: Fullstack:	☐
Suchmaschinen-optimierung (SEO) -> ist die Architektur in der Lage, die Webanwendung für SEO nutzbar zu machen?	Es liegen viele HTML-Seiten vor, deshalb funktioniert die SEO sehr gut. -> Punktzahl: 6	Es liegt nur eine HTML-Seite vor, deshalb funktioniert SEO nicht ohne weitere Technologien. -> Punktzahl: 1	Es liegt nur eine HTML-Seite vor, deshalb funktioniert SEO nicht ohne weitere Technologien. -> Punktzahl: 1	Gewicht: MPA: SPA: Fullstack:	☐

8

Gesamtergebnis				100%	

9

Anhang D: Ausgefüllte zweite Version des Kriterienkatalogs anhand des Beispielfalls für die Experimente

In Kapitel 5.1 („Erklärung der Anforderungen an die Experimente“) wurde die fiktive Firma namens „Schalk Maschinen GmbH“ als Beispiel inszeniert. Für diese Firma wurde ein Anwendungsfall definiert. Der Anhang zeigt den Einsatz der zweiten Version des Kriterienkatalogs, um für die vorliegende Problemstellung der Schalk Maschinen GmbH eine Architektur abzuleiten.

Kriterienkatalog für die Architekturauswahl

Mit diesem Kriterienkatalog sollen Software-Architekten anhand der vorliegenden Problemstellung einen webbasierten Architekturvorschlag (MPA, SPA, Fullstack-Architektur) ableiten können. Entstanden ist das Ganze im Rahmen der Bachelorarbeit „Ist die Trennung zwischen Frontend und Backend in webbasierten Systemen noch zeitgemäß?“ von Michael Mertl. Die Kriterien sind dabei aus einer umfangreichen und internationalen Literaturrecherche zu den verschiedenen Ansätzen entstanden, indem die Vor- und Nachteile dieser ermittelt wurden und dementsprechend als Grundlage dienten für die vergebenen Punkte. Außerdem wurde eine praxisorientierte Qualitätsprüfung in Form von Experteninterview bei der Firma XITASO durchgeführt, um den Kriterienkatalog zu verbessern.

Funktionsweise:
Bitte nur die Kriterien ankreuzen und gewichten, die die vorliegende Problemstellung erfordern. Am Ende die Gesamtpunktzahl ausrechnen und die Architektur mit der höchsten Punktzahl ist diejenige, die am besten geeignet ist, für die vorliegende Problemstellung. Jede Architektur hat für jedes Kriterium vorweg eine Punktzahl bekommen (1 = nicht/schlecht erfüllt, 3 = teilweise erfüllt, 6 = vollständig erfüllt), die aus der Literaturrecherche anhand der Vor- und Nachteile entstanden sind und durch Experteninterviews evaluiert wurden. Diese verrechnet mit dem Gewicht ergibt die Punktzahl für das jeweilige Kriterium. Die Gewichte, die für die einzelnen Kriterien vergeben werden, müssen am Ende 100% entsprechen, um eine mathematisch korrekte Berechnung zu gewährleisten.

Kriterium	Multi-Page-Architektur	Single-Page-Architektur	Fullstack-Architektur (orientiert an Blazor Server)	Gewicht (gesamt 100%)	Gewollt?
Autorisierung mit OAuth 2.0 -> ist die Umsetzung Autorisierung mit OAuth 2.0 in der Architektur einfach?	Die Autorisierung muss nur in einer Anwendung umgesetzt werden, da bei der MPA nur eine gebaut wird. -> Punktzahl: 6	Die Autorisierung muss in beiden Anwendungen (dem getrennten Frontend und Backend) umgesetzt werden. So entsteht ein Mehraufwand. -> Punktzahl: 3	Die Autorisierung muss nur in einer Anwendung umgesetzt werden, da bei Blazor Server nur eine gebaut wird. -> Punktzahl: 6	Gewicht: 10% MPA: 0,6 SPA: 0,3 Fullstack: 0,6	☒
Benutzererfahrung -> bietet die Architektur eine angenehme Benutzererfahrung beim Laden neuer Inhalte?	Jede Benutzerinteraktion führt zum neuen Laden der gesamten Seite, welche die neuen Inhalte dann bereitstellt. -> Punktzahl: 1	Benutzerinteraktionen lösen keine ganzen Seitenaufrufe mehr aus und es können Animationen genutzt werden, beim Warten auf neue Inhalte. -> Punktzahl: 6	Benutzerinteraktionen lösen keine ganzen Seitenaufrufe mehr aus und es können Animationen genutzt werden, beim Warten auf neue Inhalte. -> Punktzahl: 6	Gewicht: 10% MPA: 0,1 SPA: 0,6 Fullstack: 0,6	☒
Benutzererfahrung -> bietet die Architektur eine angenehme Benutzererfahrung beim Betätigen von Browsertasten (z.B.	Bei der MPA liegen viele HTML-Seiten vor, daher stellt der Browser bereits von selbst sicher, dass die verschiedenen	Bei der SPA liegt nur eine HTML-Seite vor auf der mit JS navigiert wird. Das Standardverhalten des Browsers kann nicht direkt	Bei Blazor Server liegt dasselbe Problem wie bei der SPA vor. Die .NET-Umgebung bringt eine Klasse (namens „NavigationManager“) mit, die	Gewicht: 5% MPA: 0,3 SPA: 0,05 Fullstack: 0,15	☒

2

Vorwärts/Rückwärts Schaltfläche)?	Browsertasten richtig funktionieren. -> Punktzahl: 6	genutzt werden und muss durch Programmierung nutzbar gemacht werden. -> Punktzahl: 1	es sehr einfach macht, das Standardverhalten des Browsers zu nutzen und richtig zu programmieren. -> Punktzahl: 3		
Browseranforderung -> muss der Browser JavaScript fähig sein für die Webanwendung?	Nein, da die MPA auch komplett ohne JS funktionieren würde. -> Punktzahl: 6	Ja -> Punktzahl: 1	Ja -> Punktzahl: 1	Gewicht: MPA: SPA: Fullstack:	☐
Entwicklererfahrung -> welches Fachwissen wird für die Architektur benötigt?	Es muss nur eine Anwendung entwickelt werden. In dieser wird aber Fachwissen im Frontend (HTML, CSS und JS) und Backend (z.B. C#) benötigt. -> Punktzahl: 3	Es werden zwei Anwendungen entwickelt. Für die Kommunikation dazwischen muss zusätzlich eine API entwickelt werden. Es wird also Fachwissen im Frontend (HTML, CSS und JS) und Backend (z.B. C#) benötigt. -> Punktzahl: 3	Es muss nur eine Anwendung entwickelt werden. In dieser wird teilweise Wissen im Frontend und Backend benötigt. Hier wird kein JavaScript benötigt, lediglich Wissen in C#, HTML und CSS. -> Punktzahl: 6	Gewicht: 15% MPA: 0,45 SPA: 0,45 Fullstack: 0,9	☒

3

Entwicklungszeit -> wie lange dauert die Entwicklung einer Webanwendung mit dieser Architektur?	In der Regel schneller als bei der SPA, da nur eine Anwendung entwickelt werden muss und auf die nötige API für die Kommunikation wie bei einer SPA verzichtet werden kann. Durch die Codetrennung von Frontend und Backend, aber in der Regel langsamer als die Fullstack-Architektur. -> Punktzahl: 3	In der Regel langsamer als bei der MPA und der Fullstack-Architektur, da zwei Anwendungen geschrieben werden und eine API für die Kommunikation entwickelt werden muss. -> Punktzahl: 1	In der Regel schneller als bei der SPA, da nur eine Anwendung entwickelt werden muss und auf die nötige API für die Kommunikation wie bei einer SPA verzichtet werden kann. -> Punktzahl: 6	Gewicht: 30% MPA: 0,9 SPA: 0,3 Fullstack: 1,8	☒
Erweiterbarkeit -> ist die Architektur in der Zukunft leicht erweiterbar (in Bezug auf neue Features)?	Leicht erweiterbar, Frontend und Backend sind aber etwas stärker verknüpft, da diese in einer Anwendung sind.	Leicht erweiterbar, Frontend und Backend können isoliert erweitert werden, da zwei Anwendungen vorliegen.	Es wird eine Anwendung entwickelt, in der keine saubere Codetrennung zwischen Frontend und Backend erzwungen wird, wie bei der MPA oder SPA, dies kann die Erweiterbarkeit erschweren.	Gewicht: 10% MPA: 0,3 SPA: 0,6 Fullstack: 0,3	☒

4

	-> Punktzahl: 3	-> Punktzahl: 6	-> Punktzahl: 3		
Offlinefähigkeit -> ist die Architektur in der Lage, die Webanwendung auch offline nutzbar zu machen?	Nein -> Punktzahl: 1	Ja (solange keine neue Daten-Anfrage angestoßen wird) -> Punktzahl: 6	Nein -> Punktzahl: 1	Gewicht: MPA: SPA: Fullstack:	☐
Performanz -> wie schnell ist das initiale Laden der Webanwendung?	Lädt initial schnell, da nur eine HTML-Seite (diese kann statisch oder dynamisch sein) geladen werden muss. -> Punktzahl: 6	Erster Ladezyklus dauert sehr lang, da die gesamte Anwendung geladen wird. -> Punktzahl: 1	Lädt initial schnell, da nur eine statische HTML-Seite und zusätzlich eine JavaScript-Bibliothek von Microsoft (diese ist verantwortlich für die Übertragung der Benutzer-interaktionen an den Server) übertragen werden muss. -> Punktzahl: 6	Gewicht: 10% MPA: 0,6 SPA: 0,1 Fullstack: 0,6	☒
Performanz -> wie schnell ist das Nachladen neuer Inhalte (ohne die Verwendung von zusätzlichen Caching-Mechanismen)?	Eine Anfrage nach neuen Daten fordert eine neue HTML-Seite (wieder statisch oder dynamisch) vom Server an. Der Server stellt bzw. rendert	Eine Anfrage nach neuen Daten stößt eine Anfrage (z.B. HTTP-Anfrage) an den Server an. Dieser sendet dann eine Antwort (z.B. HTTP-Antwort) mit	Eine Anfrage nach neuen Daten wird von der JavaScript-Bibliothek von Microsoft über die SignalR dem Server gemeldet. Dieser beschafft sich die neuen Daten und	Gewicht: 10% MPA: 0,3 SPA: 0,6 Fullstack: 0,6	☒

5

	die neue HTML-Seite und gibt diese an den Client zum Anzeigen zurück. Lädt genauso schnell wie initial, aber langsamer als die SPA oder die Fullstack-Architektur. -> Punktzahl: 3	den benötigten Daten zurück. Die neuen Daten werden dann am Client verarbeitet und es wird nur der Seitenteil neu gerendert, der von den neuen Daten betroffen ist. -> Punktzahl: 6	tauscht den betroffenen Teil der HTML-Seite auf dem virtuellen Abbild des Servers aus. Dieser geänderte Teil der Seite wird über die SignalR-Verbindung an den Client geschickt, wo er ebenfalls ausgetauscht wird und somit der neue Inhalt einem Anwender angezeigt wird. -> Punktzahl: 6		
Sicherheit -> wie anfällig ist die Architektur für die ungewollte Offenlegung von vertraulichen Daten durch einen Entwickler?	Die MPA ist wie Blazor Server in der Lage, die vertraulichen Daten auf dem Server zu speichern. Ebenfalls liegt hier beim Entwickeln eine strikte Trennung zwischen Frontend und Backend vor. Daher ist es beim Entwickeln leichter, keine	Die SPA speichert viele Daten am Client und ist daher anfällig für bösartiges JavaScript, welches durch Eingaben eingeschleust werden kann. Der Entwickler muss also beim Entwickeln aufpassen, dass er keine vertraulichen Daten am Client preisgibt.	Blazor Server speichert im Gegensatz zu der SPA die vertraulichen Daten auf dem Server. Durch die Vermischung von Frontend und Backend, kann es jedoch vorkommen, dass beim Entwicklungs-prozess unbeabsichtigt vertrauliche Daten auf dem Client verarbeitet werden, da	Gewicht: MPA: SPA: Fullstack:	☐

6

	vertraulichen Daten ungewollt am Client zu verarbeiten oder offenzulegen. -> Punktzahl: 6	 -> Punktzahl: 1	die Anwendung unübersichtlich werden kann. -> Punktzahl: 3		
Serverbelastung -> wie stark wird der Server belastet (ohne die Verwendung von zusätzlichen Caching-Mechanismen)?	Hoch, da jede Anfrage nach neuen Daten eine neue HTML-Seite anfordert. Diese muss jedes Mal neu vom Server bereitgestellt bzw. generiert werden. Der Zustand und die Logik der MPA können am Client und am Server (oder auch in Kombination) verarbeitet werden. Dadurch kann der Server dann zusätzlich belastet werden.	Niedrig, da jede Anfrage nach neuen Daten nur die genau benötigte Datenmenge vom Sever anfordert und dieser somit nicht so stark belastet wird als bei der MPA. Der Zustand und die Logik in der SPA werden am Client verarbeitet und belastet somit den Server nicht. Dennoch stellt over- und under-fetching eine Herausforderung dar und muss vermieden werden, sonst wird der Server unnötig belastet.	Mittel, da jede Anfrage nach neuen Daten wie bei der SPA nur die genau benötigte Datenmenge vom Server anfordert. Die Grundbelastung des Servers ist jedoch höher als bei der MPA oder der SPA, da bei Blazor Server die Logik und die Zustände am Server verwaltet werden müssen und diesen somit belasten.	Gewicht: MPA: SPA: Fullstack:	☐

7

	-> Punktzahl: 1	-> Punktzahl: 3	-> Punktzahl: 3		
Skalierbarkeit -> ist die Architektur in der Lage horizontal oder vertikal zu skalieren, um mit steigender Anzahl von Benutzern und Anfragen umzugehen?	Bei der MPA kann der Client oder auch der Server die Zustands-verwaltung übernehmen. Hier besteht also die Möglichkeit eine horizontale oder auch eine vertikale Skalierung problemlos durchzuführen. -> Punktzahl: 6	Bei der SPA übernimmt der Client die Zustands-verwaltung, somit kann der Server zustandslos betrieben werden. Hier besteht also die Möglichkeit eine horizontale oder auch eine vertikale Skalierung problemlos durchzuführen. -> Punktzahl: 6	Bei Blazor Server übernimmt der Server die Zustands-verwaltung. Eine vertikale Skalierung ist problemlos möglich. Eine horizontale Skalierung ist jedoch aufgrund dessen und der nötigen Verwaltung der SignalR-Verbindungen über mehrere Serverinstanzen hinweg, eine größere Herausforderung als bei der MPA oder SPA. -> Punktzahl: 3	Gewicht: MPA: SPA: Fullstack:	☐
Suchmaschinen-optimierung (SEO) -> ist die Architektur in der Lage, die Webanwendung für SEO nutzbar zu machen?	Es liegen viele HTML-Seiten vor, deshalb funktioniert die SEO sehr gut. -> Punktzahl: 6	Es liegt nur eine HTML-Seite vor, deshalb funktioniert SEO nicht ohne weitere Technologien. -> Punktzahl: 1	Es liegt nur eine HTML-Seite vor, deshalb funktioniert SEO nicht ohne weitere Technologien. -> Punktzahl: 1	Gewicht: MPA: SPA: Fullstack:	☐

8

Gesamtergebnis	**3,55**	**3**	**5,55**	100%	

Auswertung des inszenierten Fallbeispiels:

Die Auswertung des Kriterienkatalogs zeigt, dass für die Problemstellung der Schalk Maschinen GmbH die Fullstack-Architektur (konkret Blazor Server) am besten geeignet ist. Dieses Ergebnis wurde durch die praktische Umsetzung des Anwendungsfalls in allen drei Architekturen (MPA, SPA und Fullstack-Architektur) überprüft. Dabei wurde evaluiert, ob die Ableitung des Kriterienkatalogs korrekt war oder ob weitere Anpassungen erforderlich sind, um eine fundiertere Entscheidung zu ermöglichen. Die Ergebnisse der durchgeführten Experimente und die Anpassungen an der zweiten Version des Kriterienkatalogs sind im Kapitel 6 („Ergebnis der drei Experimente“) detailliert beschrieben

9

Anhang E: Ergebnisse der Bewertung der Experimente durch *Lighthouse*

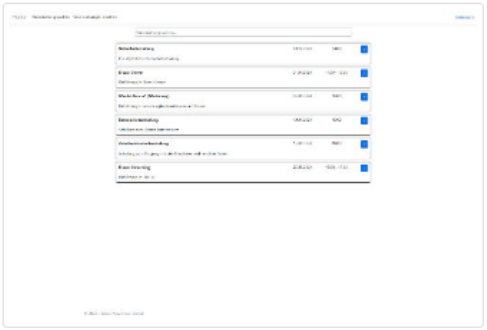
Dieser Anhang beinhaltet die Bewertungsergebnisse der Experimente durch die Chrome-Erweiterung „*Lighthouse*“.

Multi-Page-Architektur:

Performance

Values are estimated and may vary. The performance score is calculated directly from these metrics. See calculator.

▲ 0–49 ■ 50–89 ● 90–100



METRICS Expand view

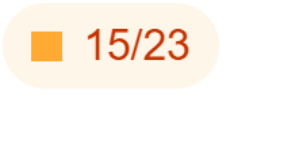
■ First Contentful Paint
1.4 s

■ Largest Contentful Paint
2.1 s

● Total Blocking Time
0 ms

● Cumulative Layout Shift
0

■ Speed Index
2.8 s



Performance

▲ 0–49 ■ 50–89 ● 90–100

METRICS Expand view

● Total Blocking Time
0 ms

● Cumulative Layout Shift
0

● Interaction to Next Paint
30 ms

Single-Page-Architektur:

Performance

Values are estimated and may vary. The performance score is calculated directly from these metrics. See calculator.

▲ 0–49 ■ 50–89 ● 90–100

METRICS Expand view

■ First Contentful Paint
1.6 s

▲ Largest Contentful Paint
2.9 s

● Total Blocking Time
0 ms

● Cumulative Layout Shift
0

▲ Speed Index
4.5 s

Performance

▲ 0–49 ■ 50–89 ● 90–100

METRICS Expand view

● Total Blocking Time
0 ms

● Cumulative Layout Shift
0

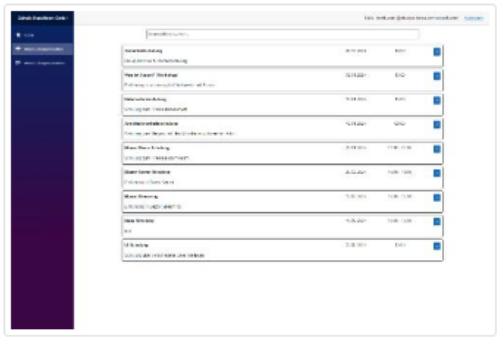
● Interaction to Next Paint
30 ms

Fullstack-Architektur:

Performance

Values are estimated and may vary. The performance score is calculated directly from these metrics. See calculator.

▲ 0–49 ■ 50–89 ● 90–100



METRICS Expand view

■ First Contentful Paint
0.9 s

● Largest Contentful Paint
1.1 s

● Total Blocking Time
0 ms

▲ Cumulative Layout Shift
0.399

● Speed Index
0.9 s

Performance

▲ 0–49 ■ 50–89 ● 90–100

METRICS Expand view

● Total Blocking Time
0 ms

● Cumulative Layout Shift
0

● Interaction to Next Paint
20 ms

Anhang F: Finale Version des Kriterienkatalogs nach den Experimenten

Alle Änderungen, die sich gegenüber der Vorversion (siehe Anhang C: „Zweite Version des Kriterienkatalog nach den Experteninterviews“) ergeben haben, sind in dieser Version farblich (gelb) markiert.

Kriterienkatalog für die Architekturauswahl

Mit diesem Kriterienkatalog sollen Software-Architekten anhand der vorliegenden Problemstellung einen webbasierten Architekturvorschlag (MPA, SPA, Fullstack-Architektur) ableiten können. Entstanden ist das Ganze im Rahmen der Bachelorarbeit „Ist die Trennung zwischen Frontend und Backend in webbasierten Systemen noch zeitgemäß?“ von Michael Mertl. Die Kriterien sind dabei aus einer umfangreichen und internationalen Literaturrecherche zu den verschiedenen Ansätzen entstanden, indem die Vor- und Nachteile dieser ermittelt wurden und dementsprechend als Grundlage dienten für die vergebenen Punkte. Außerdem wurde eine praxisorientierte Qualitätsprüfung in Form von Experteninterview bei der Firma XITASO durchgeführt, um den Kriterienkatalog zu verbessern. Ebenso wurden Experimente in den drei Architekturen durchgeführt, um einen direkten Vergleich vorzunehmen und den Kriterienkatalog zu verbessern.

Funktionsweise:
Bitte nur die Kriterien ankreuzen und gewichten, die die vorliegende Problemstellung erfordern. Am Ende die Gesamtpunktzahl ausrechnen und die Architektur mit der höchsten Punktzahl ist diejenige, die am besten geeignet ist, für die vorliegende Problemstellung. Jede Architektur hat für jedes Kriterium vorweg eine Punktzahl bekommen (1 = nicht/schlecht erfüllt, 3 = teilweise erfüllt, 6 = vollständig erfüllt), die aus der Literaturrecherche anhand der Vor- und Nachteile entstanden sind und durch Experteninterviews und Experimente evaluiert wurden. Diese verrechnet mit dem Gewicht ergibt die Punktzahl für das jeweilige Kriterium. Die Gewichte, die für die einzelnen Kriterien vergeben werden, müssen am Ende 100% entsprechen, um eine mathematisch korrekte Berechnung zu gewährleisten.

Kriterium	Multi-Page-Architektur	Single-Page-Architektur	Fullstack-Architektur (orientiert an Blazor Server)	Gewicht	Gewollt?
Autorisierung mit OAuth 2.0 -> ist die Umsetzung der Autorisierung mit OAuth 2.0 in der Architektur einfach?	Die Autorisierung muss nur in einer Anwendung umgesetzt werden, da bei der MPA nur eine gebaut wird. -> Punktzahl: 6	Die Autorisierung muss in beiden Anwendungen (dem getrennten Frontend und Backend) umgesetzt werden. So entsteht ein Mehraufwand. -> Punktzahl: 3	Die Autorisierung muss nur in einer Anwendung umgesetzt werden, da bei Blazor Server nur eine gebaut wird. -> Punktzahl: 6	Gewicht: MPA: SPA: Fullstack:	☐
Benutzererfahrung -> bietet die Architektur eine angenehme Benutzererfahrung beim Laden neuer Inhalte?	Jede Benutzerinteraktion führt zum neuen Laden der gesamten Seite, welche die neuen Inhalte dann bereitstellt. Zusätzlich werden die Inhalte im Vergleich zur Fullstack-Architektur langsamer geladen, jedoch genauso schnell wie bei der SPA. -> Punktzahl: 1	Benutzerinteraktionen lösen keine ganzen Seitenaufrufe mehr aus und es können Animationen genutzt werden, beim Warten auf neue Inhalte. Dennoch werden diese Inhalte im Vergleich zur Fullstack-Architektur langsamer geladen. -> Punktzahl: 3	Benutzerinteraktionen lösen keine ganzen Seitenaufrufe mehr aus und es können Animationen genutzt werden, beim Warten auf neue Inhalte. -> Punktzahl: 6	Gewicht: MPA: SPA: Fullstack:	☐

2

Benutzererfahrung -> bietet die Architektur eine angenehme Benutzererfahrung beim Betätigen von Browsertasten (z.B. Vorwärts/Rückwärts Schaltfläche)?	Bei der MPA liegen viele HTML-Seiten vor, daher stellt der Browser bereits von selbst sicher, dass die verschiedenen Browsertasten richtig funktionieren. -> Punktzahl: 6	Bei der SPA liegt nur eine HTML-Seite vor auf der mit JS navigiert wird. Das Standardverhalten des Browsers kann nicht direkt genutzt werden und muss durch Programmierung nutzbar gemacht werden. -> Punktzahl: 1	Bei Blazor Server liegt dasselbe Problem wie bei der SPA vor. Die .NET-Umgebung bringt eine Klasse (namens „NavigationManager“) mit, die es sehr einfach macht, das Standardverhalten des Browsers zu nutzen und richtig zu programmieren. -> Punktzahl: 3	Gewicht: MPA: SPA: Fullstack:	☐
Browseranforderung -> muss der Browser JavaScript fähig sein für die Webanwendung?	Nein, da die MPA auch komplett ohne JS funktionieren würde. -> Punktzahl: 6	Ja -> Punktzahl: 1	Ja -> Punktzahl: 1	Gewicht: MPA: SPA: Fullstack:	☐
Entwicklererfahrung -> welches Fachwissen wird für die Architektur benötigt?	Es muss nur eine Anwendung entwickelt werden. In dieser wird aber Fachwissen im Frontend (HTML, CSS und JS) und Backend (z.B. C#) benötigt.	Es werden zwei Anwendungen entwickelt. Für die Kommunikation dazwischen muss zusätzlich eine API entwickelt werden. Es wird also Fachwissen im	Es muss nur eine Anwendung entwickelt werden. In dieser wird teilweise Wissen im Frontend und Backend benötigt. Hier wird kein JavaScript benötigt, lediglich Wissen in C#, HTML und CSS.	Gewicht: MPA: SPA: Fullstack:	☐

	-> Punktzahl: 3	Frontend (HTML, CSS und JS) und Backend (z.B. C#) benötigt. -> Punktzahl: 3	-> Punktzahl: 6		
Entwicklungszeit -> wie lange dauert die Entwicklung einer Webanwendung mit dieser Architektur?	In der Regel schneller als bei der SPA, da nur eine Anwendung entwickelt werden muss und auf die nötige API für die Kommunikation wie bei einer SPA verzichtet werden kann. Durch die Codetrennung von Frontend und Backend, aber in der Regel langsamer als die Fullstack-Architektur. -> Punktzahl: 3	In der Regel langsamer als bei der MPA und der Fullstack-Architektur, da zwei Anwendungen geschrieben werden und eine API für die Kommunikation entwickelt werden muss. -> Punktzahl: 1	In der Regel schneller als bei der SPA, da nur eine Anwendung entwickelt werden muss und auf die nötige API für die Kommunikation wie bei einer SPA verzichtet werden kann. -> Punktzahl: 6	Gewicht: MPA: SPA: Fullstack:	☐
Erweiterbarkeit -> ist die Architektur in der Zukunft leicht erweiterbar	Leicht erweiterbar, Frontend und Backend sind aber etwas stärker	Leicht erweiterbar, Frontend und Backend können isoliert erweitert	Es wird eine Anwendung entwickelt, in der keine saubere Codetrennung	Gewicht: MPA: SPA:	☐

4

(in Bezug auf neue Features)?	verknüpft, da diese in einer Anwendung sind. -> Punktzahl: 3	werden, da zwei Anwendungen vorliegen. -> Punktzahl: 6	zwischen Frontend und Backend erzwungen wird, wie bei der MPA oder SPA, dies kann die Erweiterbarkeit erschweren. -> Punktzahl: 3	Fullstack:	
Offlinefähigkeit -> ist die Architektur in der Lage, die Webanwendung auch offline nutzbar zu machen?	Nein -> Punktzahl: 1	Ja (solange keine neue Daten-Anfrage angestoßen wird) -> Punktzahl: 6	Nein -> Punktzahl: 1	Gewicht: MPA: SPA: Fullstack:	☐
Performanz -> wie schnell ist das initiale Laden der Webanwendung?	Lädt initial schnell, da nur eine HTML-Seite (diese kann statisch oder dynamisch sein) geladen werden muss. Dennoch dauert das initiale Laden im Vergleich zur Fullstack-Architektur länger. -> Punktzahl: 3	Erster Ladezyklus dauert sehr lang, da die gesamte Anwendung geladen wird. -> Punktzahl: 1	Lädt initial schnell, da nur eine statische HTML-Seite und zusätzlich eine JavaScript-Bibliothek von Microsoft (diese ist verantwortlich für die Übertragung der Benutzer-interaktionen an den Server) übertragen werden muss. -> Punktzahl: 6	Gewicht: MPA: SPA: Fullstack:	☐

5

Performanz -> wie schnell ist das Nachladen neuer Inhalte (ohne die Verwendung von zusätzlichen Caching-Mechanismen)?	Eine Anfrage nach neuen Daten fordert eine neue HTML-Seite (wieder statisch oder dynamisch) vom Server an. Der Server stellt bzw. rendert die neue HTML-Seite und gibt diese an den Client zum Anzeigen zurück. Lädt genauso schnell wie initial und wie die SPA, aber langsamer als die Fullstack-Architektur. -> Punktzahl: 3	Eine Anfrage nach neuen Daten stößt eine Anfrage (z.B. HTTP-Anfrage) an den Server an. Dieser sendet dann eine Antwort (z.B. HTTP-Antwort) mit den benötigten Daten zurück. Die neuen Daten werden dann am Client verarbeitet und es wird nur der Seitenteil neu gerendert, der von den neuen Daten betroffen ist. Lädt genauso schnell wie die MPA, aber langsamer als die Fullstack-Architektur. -> Punktzahl: 3	Eine Anfrage nach neuen Daten wird von der JavaScript-Bibliothek von Microsoft über die SignalR dem Server gemeldet. Dieser beschafft sich die neuen Daten und tauscht den betroffenen Teil der HTML-Seite auf dem virtuellen Abbild des Servers aus. Dieser geänderte Teil der Seite wird über die SignalR-Verbindung an den Client geschickt, wo er ebenfalls ausgetauscht wird und somit der neue Inhalt einem Anwender angezeigt wird. -> Punktzahl: 6	Gewicht: MPA: SPA: Fullstack:	☐
Sicherheit -> wie anfällig ist die Architektur für die ungewollte Offenlegung	Die MPA ist wie Blazor Server in der Lage, die vertraulichen Daten auf dem Server zu	Die SPA speichert viele Daten am Client und ist daher anfällig für bösartiges JavaScript,	Blazor Server speichert im Gegensatz zu der SPA die vertraulichen Daten auf dem Server. Durch die Vermischung	Gewicht: MPA: SPA: Fullstack:	☐

6

von vertraulichen Daten durch einen Entwickler?	speichern. Ebenfalls liegt hier beim Entwickeln eine strikte Trennung zwischen Frontend und Backend vor. Daher ist es beim Entwickeln leichter, keine vertraulichen Daten ungewollt am Client zu verarbeiten oder offenzulegen. -> Punktzahl: 6	welches durch Eingaben eingeschleust werden kann. Der Entwickler muss also beim Entwickeln aufpassen, dass er keine vertraulichen Daten am Client preisgibt. -> Punktzahl: 1	von Frontend und Backend, kann es jedoch vorkommen, dass beim Entwicklungs-prozess unbeabsichtigt vertrauliche Daten auf dem Client verarbeitet werden, da die Anwendung unübersichtlich werden kann. -> Punktzahl: 3		
Serverbelastung -> wie stark wird der Server belastet (ohne die Verwendung von zusätzlichen Caching-Mechanismen)?	Hoch, da jede Anfrage nach neuen Daten eine neue HTML-Seite anfordert. Diese muss jedes Mal neu vom Server bereitgestellt bzw. generiert werden. Der Zustand und die Logik der MPA können am Client und am Server	Niedrig, da jede Anfrage nach neuen Daten nur die genau benötigte Datenmenge vom Sever anfordert und dieser somit nicht so stark belastet wird als bei der MPA. Der Zustand und die Logik in der SPA werden am Client verarbeitet und belastet	Mittel, da jede Anfrage nach neuen Daten wie bei der SPA nur die genau benötigte Datenmenge vom Server anfordert. Die Grundbelastung des Servers ist jedoch höher als bei der MPA oder der SPA, da bei Blazor Server die Logik und die Zustände am Server	Gewicht: MPA: SPA: Fullstack:	☐

7

	(oder auch in Kombination) verarbeitet werden. Dadurch kann der Server dann zusätzlich belastet werden. -> Punktzahl: 1	somit den Server nicht. Dennoch stellt over- und under-fetching eine Herausforderung dar und muss vermieden werden, sonst wird der Server unnötig belastet. -> Punktzahl: 3	verwaltet werden müssen und diesen somit belasten. -> Punktzahl: 3		
Skalierbarkeit -> ist die Architektur in der Lage horizontal oder vertikal zu skalieren, um mit steigender Anzahl von Benutzern und Anfragen umzugehen?	Bei der MPA kann der Client oder auch der Server die Zustands-verwaltung übernehmen. Hier besteht also die Möglichkeit eine horizontale oder auch eine vertikale Skalierung problemlos durchzuführen. -> Punktzahl: 6	Bei der SPA übernimmt der Client die Zustands-verwaltung, somit kann der Server zustandslos betrieben werden. Hier besteht also die Möglichkeit eine horizontale oder auch eine vertikale Skalierung problemlos durchzuführen. -> Punktzahl: 6	Bei Blazor Server übernimmt der Server die Zustands-verwaltung. Eine vertikale Skalierung ist problemlos möglich. Eine horizontale Skalierung ist jedoch aufgrund dessen und der nötigen Verwaltung der SignalR-Verbindungen über mehrere Serverinstanzen hinweg, eine größere Herausforderung als bei der MPA oder SPA. -> Punktzahl: 3	Gewicht: MPA: SPA: Fullstack:	☐

8

Suchmaschinen-optimierung (SEO) -> ist die Architektur in der Lage, die Webanwendung für SEO nutzbar zu machen?	Es liegen viele HTML-Seiten vor, deshalb funktioniert die SEO sehr gut. -> Punktzahl: 6	Es liegt nur eine HTML-Seite vor, deshalb funktioniert SEO nicht ohne weitere Technologien. -> Punktzahl: 1	Es liegt nur eine HTML-Seite vor, deshalb funktioniert SEO nicht ohne weitere Technologien. -> Punktzahl: 1	Gewicht: MPA: SPA: Fullstack:	☐
Gesamtergebnis				100%	

9

Anhang G: DVD mit Code der Experimente

Der Code der Experimente ist alternativ auch unter folgendem Link auf GitHub einsehbar: `https://github.com/Murtl/bachelorarbeit-mertl`.

Glossar

ASP.NET Core 7.0: ASP.NET Core 7.0 ist ein von Microsoft entwickelte plattformübergreifendes Open-Source-*Framework* zum Erstellen von Webanwendungen in der siebten Version (vgl. Microsoft 2023h).

Asynchronous JavaScript und XML (Abk.: AJAX): AJAX ist eine Technik, die es ermöglicht, Daten zwischen einem Client und einem Server im Hintergrund auszutauschen, ohne die gesamte Seite neu zu Laden (vgl. Ackermann 2021: 229 - 234).

API: Ein „Application Programming Interface“ stellt verschiedene Objekte und Methoden zur Verfügung, die wiederum in den konkreten Umsetzungen der Schnittstelle vorhanden sein müssen. Durch diese Vereinheitlichung können verschiedene Systeme miteinander interagieren (vgl. Ackermann 2021: 219).

API-Gateways: Sind Softwarekomponenten, die als zentraler Eingangspunkt dienen, um den Zugriff auf verschiedene *API*s zu steuern, verwalten und zu sichern (vgl. Tyson 2022).

Authentifizierung: Bei der Authentifizierung handelt es sich um den Prozess der Überprüfung der Identität, damit das System sicherstellen kann, dass der Nutzer auch derjenige ist, der er vorgibt zu sein (vgl. Ackermann 2021: 540 - 544).

Autorisierung: Bei der Autorisierung handelt es sich um den Prozess der Überprüfung der Zugriffsrechte, damit das System sicherstellen kann, dass der Nutzer eine bestimmte Aktion ausführen darf (vgl. Ackermann 2021: 540 - 544).

Backend: Ist der nicht sichtbare Teil einer Anwendung. Das Backend ist für die Verarbeitung von Anfragen, die Datenbankkommunikation, Geschäftslogik und die Bereitstellung von Ressourcen an das *Frontend* verantwortlich (vgl. Warnimont 2023).

Blazor Server: Blazor Server ist ein von Microsoft entwickeltes Web-*Framework*, das die Entwicklung von interaktiven, serverseitigen Webanwendungen mit C# ermöglicht (vgl. Schwichtenberg 2020).

(Cloud-)Services: Sind Dienstleistungen und Ressourcen, die über das Internet angeboten werden und keine Hardware erfordern (vgl. Red Hat 2023b).

Community: Eine sogenannte „Community of Practice“ bietet Raum für thematisch fokussierte, teamübergreifende Aktivitäten. Bei XITASO darf jeder gemäß der persönlichen Stärken und Neigungen an diesen teilnehmen, um aktiv an der Gestaltung des Unternehmens mitzuwirken (vgl. XITASO 2023).

Container-Backend-Server: Sind spezielle Server, die dafür zuständig sind, isolierte Softwareanwendungen auszuführen und zu verwalten (vgl. Red Hat 2023a).

Content Delivery Networks: Ist ein Netzwerk von Servern, das Inhalte (Webseiten, Bilder, Videos) speichert und an Nutzer in der Nähe ausliefert, um Ladezeiten zu minimieren. Dadurch wird die Leistung einer Webseite verbessert und die Verfügbarkeit der Inhalte erhöht (vgl. Akamai 2023).

Document Object Model (Abk.: DOM): Das DOM stellt die Komponenten einer

Webseite hierarchisch als Baum dar, als sogenannten DOM-Baum. Dieser setzt sich aus Knoten zusammen, die durch ihre Anordnung den Aufbau einer Webseite widerspiegeln (vgl. Ackermann 2021: 218).

First Contentful Paint (Abk.: FCP): Der „First Contentful Paint“ ist ein bedeutender Messwert von *Lighthouse* zur Einschätzung der wahrgenommenen Ladegeschwindigkeit, da er den initialen Zeitpunkt auf der Zeitleiste des Seitenaufbaus markiert, an dem der Nutzer erstmals visuellen Inhalt auf dem Bildschirm wahrnehmen kann (vgl. Walton 2023).

Framework: Ist eine vorgefertigte Struktur, die es Entwicklern ermöglicht, Anwendungen schneller zu erstellen, indem es bereits implementierte Funktionen und Bibliotheken zur Verfügung stellt (vgl. Luber 2022).

Frontend: Ist der sichtbare Teil einer Anwendung, welche im Webbrowser des Benutzers dargestellt wird. Das Frontend ist dafür verantwortlich die Benutzeroberfläche zu gestalten, Inhalte anzuzeigen und Interaktionen zu ermöglichen. Es besteht hauptsächlich aus HTML für die Struktur, CSS für das Aussehen und JavaScript für die Interaktivität (vgl. Warnimont 2023).

Horizontale Skalierung: Bei der horizontalen Skalierung wird die Leistungskapazität erhöht, indem einem System mehr Ressourcen des gleichen Typs hinzugefügt werden. Ein Beispiel hierfür ist das Hinzufügen eines weiteren Servers zu einem System (vgl. Barber 2023).

Interaction to next Paint (Abk.: INP): Der „Interaction to next Paint“ ist ein wichtiger Messwert von *Lighthouse* zur Beurteilung der Reaktionsfähigkeit einer Webanwendung auf Nutzerinteraktionen. Ein niedriger INP deutet darauf hin, dass die Seite durchgehend schnell auf alle oder den Großteil der Nutzerinteraktionen reagieren konnte (vgl. Wagner 2023).

JSON-Format: Ist ein leichtgewichtiges Datenformat, das zur Strukturierung von Daten in lesbarer Form (Schlüssel-Wert-Paare) verwendet wird, um so den Datenaustausch zwischen zwei Anwendungen zu vereinfachen (vgl. Ackermann 2021: 196 - 201).

Lighthouse: Google „Lighthouse“ ist ein Chrome-Entwicklertool, welches es Entwickler ermöglicht, die Leistung von Webanwendungen zu analysieren (vgl. Kühle 2021).

Likert-Skala: Die Likert-Skala ist eine Methode zur Bewertung von Meinungen oder Einstellungen, bei der Teilnehmer ihre Zustimmung oder Ablehnung zu Aussagen auf einer stufenweisen Skala ausdrücken (vgl. Prof. Dr. Klaus Wübbenhorst 2018).

Load Balancer: Ist eine Software oder Hardware, welche den Datenverkehr auf mehrere Server gleichmäßig verteilt (vgl. Yasar und Irei 2023).

Makroarchitektur: Bezieht sich auf die grundlegende Struktur und Organisation eines Systems auf abstrakter Ebene. Es werden hier nur wesentliche Komponenten, die Beziehungen zueinander und die Art und Weise, wie sie zusammenarbeiten beschrieben (vgl. Schleupen.CS Developer Campus 2023a).

Microsoft Azure: Microsoft Azure ist eine Cloud-Computing-Plattform von Microsoft, die eine Vielzahl von Diensten und Ressourcen für die Entwicklung, Bereitstellung und Verwaltung von Anwendungen in der Cloud bietet (vgl. Orth 2023).

Mikroarchitektur: Bezieht sich auf die detaillierte interne Struktur einer spezifischen Komponente und legt fest, wie diese auf niedriger Ebene implementiert ist

(vgl. Schleupen.CS Developer Campus 2023b).

.NET-Umgebung: Ist eine von Microsoft entwickelte plattformübergreifende Open-Source-Entwicklerplattform, die umfangreiche Hilfsmittel für die Entwicklung von z.B. Windows-Anwendungen oder Webdiensten bereitstellt (vgl. Microsoft 2023f).

Overfetching: Eine *API*-Anfrage gibt zu viele Daten zurück, die nicht verwendet werden und unnötig mitgeschickt werden (vgl. Simpson 2022).

Remix.run: Ist ein Fullstack-*Framework*, welches es ermöglicht JavaScript auch auf dem Server einzusetzen und die Entwicklung von *Frontend* und *Backend* zusammenzieht bei einer Webanwendung (vgl. Boga 2023).

Service Worker: Ein Service Worker in Web-Anwendungen ist ein JavaScript-Code, der unabhängig von der eigentlichen Anwendung im Hintergrund läuft. Er ermöglicht Funktionen wie Offline-Fähigkeit, Push-Benachrichtigungen und die Verbesserung der Anwendungsleistung (vgl. Siegrist 2021).

Speed Index: Der „Speed Index“ ist ein bedeutender und einer der aussagekräftigsten Messwerte von *Lighthouse* zur Bewertung der Geschwindigkeit einer Webanwendung. Der Speed Index misst, wie schnell der gesamte Inhalte visuell dargestellt werden kann (vgl. Khayrullin 2022).

Swagger (OpenAPI): Bei Swagger (OpenAPI) handelt es sich um eine sprachunabhängige Spezifikation für das Beschreiben von *API*s (vgl. Microsoft 2023b).

Typsystem: Legt fest, welche Arten von Werten oder Ausdrücken in einer Programmiersprache existieren und wie diese miteinander agieren dürfen. Das Typsysteme definiert die Datentypen, Operationen und Regeln für die Verarbeitung von

Daten in einem Programm (vgl. Block 2017).

Underfetching: Eine *API*-Anfrage gibt zu wenig Daten zurück, was einen weiteren Aufruf erforderlich macht (vgl. Simpson 2022).

Vaadin: Ist ein *Framework*, welches es ermöglicht moderne und interaktive webbasierte Benutzeroberflächen mit Java zu entwickeln (vgl. Tam 2017).

Vertikale Skalierung: Bei der vertikalen Skalierung wird die Kapazität einer Hardware oder Software erhöht, indem Ressourcen zu einem physischen System hinzugefügt werden. Ein Beispiel hierfür ist das Hinzufügen von Rechenleistung zu einem Server (vgl. Barber 2023).

Visual Studio: Visual Studio ist eine integrierte Entwicklungsumgebung von Microsoft, die Entwicklern Werkzeuge für das Schreiben, Debuggen und Testen von Softwareprojekten bietet (vgl. Microsoft 2023k).

Vue 3.3: Vue.js ist ein JavaScript-*Framework* für die *Frontend*entwicklung, welches von Evan You ins Leben gerufen wurde und inzwischen durch ein internationales Team um Evan You vorangetrieben wird (vgl. Kraus 2022).

Web-APIs: Sind verschiedene Schnittstellen, die vom Browser zur Verfügung gestellt und mit JavaScript gesteuert werden. Damit können Webseiten noch interaktiver und professioneller gestaltet werden (vgl. Ackermann 2021: 217).

WebSockets: Sind eine Kommunikationstechnologie, die es ermöglicht, eine dauerhafte, bidirektionale Verbindung zwischen einem Webbrowser und einem Server herzustellen, was Echtzeitinteraktionen in Webanwendungen ermöglicht (vgl. Redaktion 2020).

Zugriffstoken: Erhält ein Benutzer, nachdem er seine Identität verifiziert hat. Besteht meistens aus einer Zeichenfolge und ist für einen bestimmten Zeitraum gültig (vgl. Ackermann 2021: 543f).

Literatur

Ackermann, Philip (2021). *Webentwicklung: Das Handbuch für Fullstack-Entwickler*. 1. Auflage. Bonn: Rheinwerk Verlag. ISBN: 978-3-8362-6882-0.

Akamai, Hrsg. (2023). *Was ist ein Content Delivery Network (CDN)?* URL: https://www.akamai.com/de/glossary/what-is-a-cdn (besucht am 31. 10. 2023).

Barber, Elisabet García (2023). „Skalierbarkeit“. In: *Wilde-IT GmbH* 2023. URL: https://www.wilde-it.com/skalierbarkeit/ (besucht am 08. 11. 2023).

Basse, Leonard (2021). *Daher sollten Sie Single Page Applications nutzen! - Sortlist Blog*. URL: https://www.sortlist.de/blog/single-page-applications-web-apps-aus-einem-html-dokument/ (besucht am 29. 10. 2023).

Block, Sascha (2017). *Was ist ein Type System?* Hrsg. von INZTITUT. URL: https://inztitut.de/blog/glossar/type-system/ (besucht am 31. 10. 2023).

Boga, Alejo (2023). *What is Remix And How To Use It in Real Life*. Hrsg. von Medium. URL: https://medium.com/@alejoboga19/remix-run-bringing-the-bases-back-to-life-aa9bed254159 (besucht am 31. 10. 2023).

GetDevDone Team (2021). *A New Drupal Development Dilemma: Should You Decouple Your Website? - GetDevDone Blog*. URL: https://getdevdone.com/blog/new-drupal-development-dilemma-should-you-decouple-your-website.html (besucht am 05. 10. 2023).

Gingter, Sebastian (2020). *Blazor Server: Mögliche Architekturalternative zu SPAs?* Hrsg. von Thinktecture AG. URL: https://www.thinktecture.com/webinare/blazor-server-moegliche-alternative-zu-spas/ (besucht am 15. 10. 2023).

Harris, Rich (2021). *Have Single-Page Apps Ruined the Web? | Transitional Apps with Rich Harris, NYTimes*. URL: https://www.youtube.com/watch?v=860d8usGC0o&ab_channel=JamstackTV (besucht am 15. 10. 2023).

Harsh, Kumar (2023). „Was ist eine Webanwendungsarchitektur? Eine Webanwendung aufschlüsseln“. In: *Kinsta* 2023. URL: https://kinsta.com/de/blog/web-anwendungs-architektur/ (besucht am 24. 10. 2023).

Hilton, Jon (2023). *Blazor's New LocationChanging Events in .NET 7*. URL: https://www.telerik.com/blogs/blazor-new-locationchanging-events-dotnet-7 (besucht am 18. 11. 2023).

Himschoot, Peter (2022). „Single-Page Applications and Routing“. In: *Microsoft Blazor*. Apress, Berkeley, CA, S. 351–387. DOI: 10.1007/978-1-4842-7845-1_9. URL: https://link.springer.com/chapter/10.1007/978-1-4842-7845-1_9 (besucht am 14. 10. 2023).

Khayrullin, Arthur (2022). „What is the Lighthouse Speed Index and why should you care?“ In: *Uploadcare* 2022. URL: https://uploadcare.com/blog/what-is-the-lighthouse-speed-index/ (besucht am 20. 12. 2023).

Kraus, Chrissi (2022). „Was ist Vue.js?“ In: *Dev-Insider*. URL: https://www.dev-insider.de/was-ist-vuejs-a-5f8b41ce678a4a47c6fdb394ed8d193a/ (besucht am 24. 11. 2023).

Kühle, Markus (2021). „Google Lighthouse: Was ist das und ist das wirklich wichtig?“ In: *coodoo* 2021. URL: https://blog.coodoo.io/google-lighthouse-was-ist-das-und-ist-das-wirklich-wichtig-5d36bccb0475 (besucht am 22. 11. 2023).

Kulesza, Raoni u. a. (2020). „Evolution of Web Systems Architectures: A Roadmap“. In: *Special Topics in Multimedia, IoT and Web Technologies*. Springer,

Cham, S. 3–21. DOI: 10.1007/978-3-030-35102-1_1. URL: https://link.springer.com/chapter/10.1007/978-3-030-35102-1_1 (besucht am 02. 10. 2023).

Luber, Stefan (2022). „Was ist ein Framework?“ In: *CloudComputing-Insider* 2022. URL: https://www.cloudcomputing-insider.de/was-ist-ein-framework-a-1104630/ (besucht am 31. 10. 2023).

Microsoft, Hrsg. (2023a). *ASP.NET Core SignalR: Hosten in der Produktion und Skalieren*. URL: https://learn.microsoft.com/de-de/aspnet/core/signalr/scale?view=aspnetcore-7.0 (besucht am 10. 11. 2023).

— Hrsg. (2023b). *ASP.NET Core-Web-API-Dokumentation mit Swagger/OpenAPI*. URL: https://learn.microsoft.com/de-de/aspnet/core/tutorials/web-api-help-pages-using-swagger?view=aspnetcore-7.0 (besucht am 24. 11. 2023).

— Hrsg. (2023c). *Blazor-Projektstruktur in ASP.NET Core*. URL: https://learn.microsoft.com/de-de/aspnet/core/blazor/project-structure?view=aspnetcore-7.0 (besucht am 24. 11. 2023).

— Hrsg. (2023d). *Erstellen einer ASP.NET Core-App mit Vue - Visual Studio (Windows)*. URL: https://learn.microsoft.com/de-de/visualstudio/javascript/tutorial-asp-net-core-with-vue?view=vs-2022 (besucht am 24. 11. 2023).

— Hrsg. (2023e). *Hosten und Bereitstellen von serverseitigen ASP.NET Core Blazor-Apps*. URL: https://learn.microsoft.com/de-de/aspnet/core/blazor/host-and-deploy/server?view=aspnetcore-8.0&viewFallbackFrom=aspnetcore-3.0 (besucht am 23. 11. 2023).

— Hrsg. (2023f). *NET (und .NET Core): Einführung und Übersicht - .NET*. URL: https://learn.microsoft.com/de-de/dotnet/core/introduction (besucht am 23. 11. 2023).

Microsoft, Hrsg. (2023g). *Sitzung in ASP.NET Core*. URL: https://learn.microsoft.com/de-de/aspnet/core/fundamentals/app-state?view=aspnetcore-7.0 (besucht am 23. 11. 2023).

— Hrsg. (2023h). *Übersicht über ASP.NET Core*. URL: https://learn.microsoft.com/de-de/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-7.0 (besucht am 23. 11. 2023).

— Hrsg. (2023i). *Übersicht über ASP.NET Core MVC*. URL: https://learn.microsoft.com/de-de/aspnet/core/mvc/overview?view=aspnetcore-7.0 (besucht am 23. 11. 2023).

— Hrsg. (2023j). *Übersicht über Single-Page-Webanwendung (SPA) in ASP.NET Core*. URL: https://learn.microsoft.com/de-de/aspnet/core/client-side/spa/intro?view=aspnetcore-7.0#developing-single-page-apps (besucht am 23. 11. 2023).

— Hrsg. (2023k). *Was ist Visual Studio?* URL: https://learn.microsoft.com/de-de/visualstudio/get-started/visual-studio-ide?view=vs-2022 (besucht am 22. 11. 2023).

Orth, Björn (2023). „Was ist Microsoft Azure?“ In: *CIO*. URL: https://www.cio.de/a/was-ist-microsoft-azure,3678460 (besucht am 19. 12. 2023).

Patadiya, Jaydeep (2023). „SPA vs MPA: What to Choose for Business?“ In: *Radixweb* 2023. URL: https://radixweb.com/blog/single-page-application-vs-multi-page-application#Difference (besucht am 20. 10. 2023).

Prof. Dr. Klaus Wübbenhorst (2018). „Definition: Likert-Skalierung“. In: *Springer Fachmedien Wiesbaden GmbH* 2018. URL: https://wirtschaftslexikon.gabler.de/definition/likert-skalierung-40986 (besucht am 11. 11. 2023).

Red Hat, Hrsg. (2023a). *Was ist Docker? Welche Vorteile bieten Container?* URL: https://www.redhat.com/de/topics/containers/what-is-docker (besucht am 23. 10. 2023).

Red Hat, Hrsg. (2023b). *Was sind Cloud-Services?* URL: https://www.redhat.com/de/topics/cloud-computing/what-are-cloud-services (besucht am 31. 10. 2023).

Redaktion, Ionos (2020). „Was ist WebSocket?“ In: *IONOS* 2020. URL: https://www.ionos.de/digitalguide/websites/web-entwicklung/was-ist-websocket/ (besucht am 15. 10. 2023).

— (2022). „OAuth (Open Authorization)“. In: *IONOS*. URL: https://www.ionos.de/digitalguide/server/sicherheit/was-ist-oauth/ (besucht am 09. 11. 2023).

Schleupen.CS Developer Campus, Hrsg. (2023a). *Makroarchitektur*. URL: https://developer-campus.de/tracks/architecture/makroarchitektur/ (besucht am 31. 10. 2023).

— Hrsg. (2023b). *Mikroarchitektur Länder*. URL: https://developer-campus.de/tracks/architecture/mikroarchitektur-laender/ (besucht am 31. 10. 2023).

Schmitt, Philip (2022). „Die Zukunft des Frontends“. In: *dns* 2022. URL: https://www.dotnetpro.de/frontend/zukunft-frontends-2810967.html (besucht am 15. 10. 2023).

Schwichtenberg, Holger (2020). *Blazor Server: Der erste Blazor-Streich*. URL: https://entwickler.de/dotnet/der-erste-blazor-streich-001 (besucht am 15. 10. 2023).

— (2023). „NET 8.0 und C# 12.0 erscheinen heute: Viel Neues für Blazor und C#-Compiler“. In: *heise online* 2023. URL: https://www.heise.de/news/NET-8-0-und-C-12-0-erscheinen-heute-Viel-Neues-fuer-Blazor-und-C-Compiler-9502644.html (besucht am 22. 12. 2023).

Shilpi (2019). *Divide or Join: The Frontend Backend Dilemma | OpenSense Labs*. URL: https://opensenselabs.com/blog/articles/frontend-backend (besucht am 14. 10. 2023).

Siegrist, Katharina (2021). *Service Worker in Web-Anwendungen*. URL: https://www.informatik-aktuell.de/entwicklung/methode

n/service-worker-in-web-anwendungen.html (besucht am 16. 11. 2023).

Simpson, J. (2022). *How to Avoid Overfetching and Underfetching | Nordic APIs |*. URL: https://nordicapis.com/how-to-avoid-overfetching-and-underfetching/ (besucht am 18. 11. 2023).

Stack Overflow, Hrsg. (2023). *Stack Overflow Developer Survey 2023*. URL: https://survey.stackoverflow.co/2023/#most-popular-technologies-webframe (besucht am 04. 10. 2023).

Sun, Yiyi (2019). „Single-Page Applications“. In: *Practical Application Development with AppRun*. Apress, Berkeley, CA, S. 141–162. DOI: 10.1007/978-1-4842-4069-4_7. URL: https://link.springer.com/chapter/10.1007/978-1-4842-4069-4_7 (besucht am 04. 10. 2023).

Tam, Hanna (2017). „Einführung in die Entwicklung mit Vaadin“. In: *heise online* 2017. URL: https://www.heise.de/hintergrund/Einfuehrung-in-die-Entwicklung-mit-Vaadin-Teil-1-3594394.html (besucht am 31. 10. 2023).

Thattil, Sascha (2018). „Was ist eine Single Page Application (SPA)?“ In: *Yuhiro* 2018. URL: https://www.yuhiro.de/was-ist-eine-single-page-application-spa/ (besucht am 23. 10. 2023).

Tyson, Matthew (2022). „Was ist ein API Gateway?“ In: *COMPUTERWOCHE* 2022. URL: https://www.computerwoche.de/a/was-ist-ein-api-gateway,3551047 (besucht am 31. 10. 2023).

Vermeir, Nico (2022). „Blazor“. In: *Introducing .NET 6*. Apress, Berkeley, CA, S. 125–152. DOI: 10.1007/978-1-4842-7319-7_5. URL: https://link.springer.com/chapter/10.1007/978-1-4842-7319-7_5 (besucht am 05. 10. 2023).

Wagner, Jeremy (2023). *Interaction to Next Paint (INP)*. Hrsg. von web.dev. URL: https://web.dev/articles/inp?hl=de (besucht am 20. 12. 2023).

Walton, Philip (2023). *First Contentful Paint (FCP)*. Hrsg. von web.dev. URL: https://web.dev/articles/fcp?hl=de (besucht am 20. 12. 2023).

Warnimont, Joe (2023). „Backend vs. Frontend: Wie unterscheiden sie sich?“ In: *Kinsta* 2023. URL: https://kinsta.com/de/blog/backend-vs-frontend/#backend-vs-frontendentwicklung--hauptunterschiede (besucht am 14. 10. 2023).

XITASO, Hrsg. (2023). *Kultur < XITASO*. URL: https://xitaso.com/unternehmen/kultur/ (besucht am 04. 12. 2023).

Yasar, Kinza und Alissa Irei (2023). *Was ist Load Balancing (Lastenverteilung)?* Hrsg. von ComputerWeekly.de. URL: https://www.computerweekly.com/de/definition/Load-Balancing (besucht am 31. 10. 2023).