HAW HAMBURG

Bachelorarbeit

Paul Höppner

Herleitung und Anwendung einer Methode zur Entwicklung eines Minimal Viable Products

Fakultät Technik und Informatik
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

Paul Höppner

Herleitung und Anwendung einer Methode zur Entwicklung eines Minimal Viable Products

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Angewandte Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Olaf Zukunft
Zweitgutachterin: Prof. Dr. Birgit Wendholt

Eingereicht am: 18.08.2022

Paul Höppner

Thema der Arbeit

Herleitung und Anwendung einer Methode zur Entwicklung eines Minimal Viable Products

Stichworte

MVP, Softwarearchitektur, Softwareentwicklung, Webanwendung, Frontend

Kurzzusammenfassung

Als Minimal Viable Product (MVP) wird die Iteration einer Software beschrieben, die die minimalen Anforderungen erfüllt, um die Software zu veröffentlichen. Dies ist vor allem hilfreich, um in einem frühen Stadium des Projektes Feedback von Kunden einzuholen. In dieser Arbeit wird beschrieben, wie aus der Analyse eines MVPs eine Methode abgeleitet werden kann, mit der MVPs einfacher entwickelt werden können.
Der MVP ist Teil der Webanwendung **Bandrecording Hamburg**, eines Konfigurators zum Buchen von professionellen Tonstudioaufnahmen. Im Anschluss soll anhand der Methode auf ein MVP für ein weiteres Fallbeispiel angewendet werden. Abschließend wird die Methode an den Ergebnissen der Entwicklung Überprüft.

Paul Höppner

Title of Thesis

Derivation and application of a method for the development of a minimal viable product

Keywords

MVP, software architecture, software development, web application, Frontend

Abstract

A Minimal Viable Product (MVP) is described as the iteration of a software that meets the minimum requirements to release the software. This is particularly useful for obtaining

feedback from customers at an early stage of the project. In this paper, it is described how the analysis of an MVP can be used to derive a method to develop MVPs more easily. The MVP is part of the web application **Bandrecording Hamburg**, a configurator for booking professional recording studios. Subsequently, the method will be applied to an MVP for another case study. Finally, the method will be verified on the results of the development.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Das Entwickeln einer Softwareanwendung ist ein Prozess, der klassisch in diskreten Phasen, nach dem sogenannten Wasserfallmodell abläuft. Dabei werden die Phasen nacheinander durchlaufen. Wie in einem Wasserfall kann nicht in eine der vorherigen Phasen zurückgesprungen werden [16]. Ein großes Problem des Wasserfallmodells ist es, dass das Testen der entwickelten Anwendung erst zum Ende hin durchgeführt wird. Dadurch kann es dazu kommen, dass viele Ressourcen zum Entwickeln einer Anwendung aufgewendet wurden, die nicht den sich möglicherweise stets ändernden Ansprüchen der Kund:innen entspricht.
Um das Problem der geringen Einbeziehung der Kund:innen zu lösen, wird heutzutage der Ansatz der agilen Softwareentwicklung verwendet. Bei diesem wird die Anwendung in kleinere Teile zerlegt und diese in kurzen Zeiträumen umgesetzt. Nach der Umsetzung eines solchen Abschnittes können die Kund:innen mit einbezogen werden. Dadurch ist eine kontinuierliche Absprache mit den Kund:innen und damit verbunden eine bessere Reaktion auf Veränderungen möglich [29].

1.1 Motivation

In Zeiten der agilen Softwareentwicklung wird in Iterationen vorgegangen, wovon die Erste häufig zunächst eine Art Prototyp, dem Minimal Viable Product (im Folgenden als MVP abgekürzt) darstellt [3]. Mithilfe dieses MVPs ist es möglich in einer frühen Entwicklungsphase Kundenfeedback einzuholen. Der MVP muss nicht die finale Architektur des Produktes aufweisen, kann aber zum Entwurf dieser herangezogen werden. Die Erstellung eines MVPs weist jedoch einige Herausforderungen, wie beispielsweise die Auswahl der zu implementierenden Anforderungen, auf.
Auf diese Herausforderungen wird im Rahmen der Forschungsarbeit eingegangen und ein Ansatz vorgeschlagen, um diese zu meistern.

1.2 Ziele

Das Ziel dieser Arbeit ist es, einen Ansatz zu entwickelt, mit dem ein Minimal Viable Product möglichst effizient und zielgerichtet entwickelt werden kann. Dazu wird der MVP der Webanwendung **Bandrecording Hamburg** (im Folgenden als BRHH abgekürzt) analysiert und eine Methode zur Entwicklung von MVPs erstellt. Um die Methode zu überprüfen, soll anhand dieser ein MVP für ein zweites Fallbeispiel, einen Blog, entwickelt werden. Anhand der Entwicklung des zweiten MVPs soll die Tauglichkeit der Methode geprüft werden, sowie weitere Verbesserungsmöglichkeiten abgeleitet werden. Innerhalb der Methode sollen zur Umsetzung der Methode hilfreiche Heuristiken vorgestellt werden. Der Fokus der Arbeit soll dabei auf Webanwendungen gelegt werden, um den Forschungsrahmen adäquat zu begrenzen.

1.3 Related Work

In der Literatur existieren bereits Methoden zur Entwicklung eines MVPs. Amit Manchanda stellt in seinem Artikel „A Step-by-Step Guide to Build a Minimum Viable Product (MVP)“ [28] eine Möglichkeit vor ein MVP zu entwickeln. Er bezieht sich jedoch auf den gesamten Zyklus des MVPs und beschreibt auch die Auswertung, weshalb die Beschreibung der Entwicklungsphase kürzer ausfällt, als es in dieser Arbeit geplant ist.
In dem Artikel „Minimum Viable Product – Die Anleitung für dein nächstes MVP“ [20] beschreibt Andreas Diehl, unter anderem, wie ein MVP entwickelt werden kann. Er konzentriert sich dabei auch eher auf den ganzen Zyklus und beschreibt den Entwicklungsprozess nur kurz. Zusätzlich führt er jedoch noch einige hilfreiche Tools, zur Entwicklung eines MVP auf, wie zum Beispiel die in Kapitel 2.1.5 beschriebene Kano-Methode.
Keine der aufgeführten Methoden legt den Fokus auf die Entwicklungsphase des MVPs. Außerdem wird wenig Hilfe zur Auswahl der Art des MVPs und des Architekturstils gegeben. Diese Punkte sollen in der Forschungsarbeit behandelt werden.

1.4 Struktur der Arbeit

Die Arbeit ist in acht Kapitel unterteilt. Zunächst werden die für die Entwicklung der Methode und den Kontext der Forschungsarbeit nötigen Grundlagen beschrieben. Das darauf folgende Kapitel befasst sich mit dem ersten Fallbeispiel, Bandrecording Hamburg,

und stellt dieses vor. Als Nächstes wird die Entwicklung des MVPs von Bandrecording Hamburg analysiert und daraus die Methode zur Entwicklung von MVPs abgeleitet sowie diese näher beleuchtet. Darauf folgend wird die entwickelte Methode auf das zweite Fallbeispiel, den Blog, angewendet und danach beschrieben, wie der MVP des Blogs implementiert worden ist. Danach wird die Methode im Fazit bewertet. Abschließend wird eine Zusammenfassung der Arbeit sowie ein Ausblick auf weiterführende Forschung gegeben.

2 Grundlagen

Um zunächst wichtige Begriffe und Grundlagen zu klären, werden in diesem Kapitel für diese Arbeit relevante Grundlagen der Softwarearchitektur und Webanwendungen aufgeführt. Im ersten Teil (Kapitel 2.1) werden verwendete Begriffe und in der Arbeit verwendete Methoden beschrieben. Im zweiten Teil (Kapitel 2.2) geht es um für die Softwarearchitektur wichtige Prinzipen, die nötig sind, um die im dritten Teil (Kapitel 2.3) beschriebenen Architekturstile zu verstehen.

2.1 Begriffsklärung und grundlegende Methoden

In dem ersten Teil der Grundlagen soll zunächst der Begriff der Softwarearchitektur sowie weitere in dieser Arbeit verwendete Begriffe eingeführt werden. Außerdem werden im späteren Teil der Arbeit verwendete Methoden zur Priorisierung von Features vorgestellt.

2.1.1 Softwarearchitektur

Softwarearchitektur ist ein sehr umfangreiches Gebiet und lässt sich schwer mit einer kurzen und prägnanten Definition beschreiben. Len Bass und Rick Kazman haben folgende Definition vorgeschlagen [15]: „The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.“ [31].
‚Software components‘ lassen sich hierbei als Software Bausteine übersetzen. Mit Software-Bausteinen sind die Teile gemeint, aus denen sich eine Software zusammensetzt. Hierbei können die Bausteine grob in drei Kategorien unterteilt werden [31]:

1. fachliche Bausteine

2. technische Bausteine

3. Plattform-Bausteine

Der Übergang zwischen den Kategorien ist fließend. Fachliche Bausteine repräsentieren den Problembereich der Software und somit unter anderem die an die Software gestellten funktionalen Anforderungen [31].
Auf fachlicher Ebene ist der Grad der Abstraktion sehr hoch und die Software-Bausteine haben wenige Abhängigkeiten zur Plattform. Unter der fachlichen Ebene liegt die technische Ebene, welche die nicht-funktionalen Anforderungen abdeckt. Ein Beispiel für einen technischen Baustein ist die Datenbank als Persistenzlösung. Im Vergleich zur fachlichen Ebene sinkt der Grad der Abstraktion der technischen Bausteine und die Abhängigkeit zur Plattform steigt. Technische Bausteine können grobe fachliche Bausteine feiner unterteilen. Vogel führt hier als Beispiel den fachlichen Baustein der Auftragsverwaltung an, der durch Verwendung des MVC-Musters (Näheres hierzu in Kapitel 2.3.2), einer Art der technischen Architektur, in drei einzelne Bausteine, den Modell-, View- sowie Controller-Baustein aufgespalten wird.
Außerdem beschreibt die Softwarearchitektur wie diese Bausteine miteinander in Beziehung stehen und welche Teile von ihnen nach außen sichtbar sind [31].

2.1.2 Lean Startup

Unter Lean Startup versteht man eine Methodik, ein Startup zu führen, die von Steve Blank entwickelt und von Eric Ries popularisiert worden ist [3]. Das Ziel der Methodik ist es, das Scheitern von Startups zu verhindern, indem Produktentwicklungszyklen verkürzt werden und Ideen vor der Umsetzung empirisch geprüft werden.
Das Grundprinzip der Lean Startup Methode ist es, jede Idee als Hypothese zu sehen, die empirisch bewiesen werden muss. Dadurch soll Zeit, die für irrelevante Punkte aufgewendet wird, minimiert werden und somit kein Produkt veröffentlicht werden, für das kein Markt existiert.
Eine Methode, um solche Hypothesen zu validieren ist das Minimal Viable Product [3].

2.1.3 Minimal Viable Product

Das Minimal Viable Product bezeichnet die Iteration eines Produktes, welches die minimalen Ansprüche des Kunden erfüllt [2]. Es ist besonders nützlich, um in einem frühen

Stadium des Projektes Feedback von Kunden einzuholen.
„According to Lean Startup, every startup should start with building a Minimum viable product (MVP), and use it to validate their hypotheses about customer needs“ [22]. Laut Blank und Ries sollte jedes, nach der Lean Startup Methode geführte, Startup immer mit der Entwicklung eines MVP beginnen, um die Hypothese bezüglich Kundenerwartungen zu validieren.
Die Entwicklungsphase eines MVP ähnelt dem Pareto Prinzip. Dieses besagt, dass mit 20 % des Aufwands 80 % der Ergebnisse erreicht werden [30]. Genau wie bei dem Pareto Prinzip sollte sich bei der Entwicklung von MVPs an diesem Verhältnis orientiert werden, um den Release des MVP nicht unnötig zu verzögern.
Es sind verschiedene Arten eines MVPs für Webanwendungen denkbar, wie zum Beispiel die Folgenden [22]:

1. Eine Landingpage, also eine Website, auf der ein Kunde landet, sobald die Website aufgerufen wird und die das Design sowie einige Grundfunktionalitäten präsentieren kann.

2. Ein Modell (Mockup) des Produktes, welches die Anwendung auf Papier oder einem ähnlichen Medium darstellt.

3. Ein „Wizard of Oz“-MVP, bei dem der Kunde den Eindruck hat, als würde er ein fertiges Produkt bedienen, obwohl nur das Interface funktional ist, während die Logik händisch ausgeführt wird.

4. Ein Frontend MVP, bei dem die gesamte Logik des Produktes im Frontend ausgeführt wird und nur ein minimales bis kein Backend vorhanden ist.

5. Ein Backend MVP, bei dem kein Wert auf das Frontend gelegt wird, sondern dieses nur alle Eingaben erfüllt, die unbedingt notwendig sind. Im Backend kann ein Algorithmus oder die Anbindung an einen anderen Service, wie eine Datenbank geprüft werden.

Welche Art des MVP gewählt werden sollte, hängt stark von den Anforderungen an das Produkt und dem zu validierenden Teil des Produktes ab. So können sich MVPs, die die Businesslogik testen sollen, stark von solchen, die einen Algorithmus testen, unterscheiden. Je nachdem, angestrebtem Ziel des MVPs, müssen die Features, die implementiert werden, anders priorisiert werden.
Soll die Bedienoberfläche getestet werden, bietet sich beispielsweise ein „Wizard of Oz“

oder ein Frontend MVP an, bei denen ein großer Teil der Logik nicht implementiert sein muss.

Ein MVP hat viele Gemeinsamkeiten mit einem Prototyp. Anders als beim MVP ist der Prototyp nicht nur für die Endkund:innen, also im Fall einer Website die Anwender:in, sondern auch für die Kommunikation innerhalb des Projektes gedacht. Hierbei kann es sich sowohl um die Kommunikation zwischen Auftraggeber:in und Entwickler:in, die Kommunikation zwischen Entwickelnden oder auch die Kommunikation zur Endkund:in handeln. Protoyping beschreibt sehr viel allgemeiner den Vorgang, mithilfe eines Teilproduktes die Kommunikation zu vereinfachen. Hierbei geht es auch um die Kommunikation innerhalb des Entwickler:innenteams [24].
Floyd beschreibt in ihrem Paper ‚A systematic look at Prototyping' den, in dieser Arbeit genauer betrachteten, Frontend Prototypen, den sie als „Human Interface (Front End) Simulation" bezeichnet. Sie beschreibt hier einen Prototyp, der ein finales Frontend präsentiert, aber dessen andere Funktionalitäten, wie zum Beispiel die Evaluierung der eingegebenen Daten, nicht vollständig, sondern nur als Mockup implementiert sind [24].

2.1.4 API

APIs (Application Programming Interfaces) sind Schnittstellen, mit denen Programme untereinander kommunizieren können. Dadurch lässt sich eine, von der Implementierung der Programme unabhängige Kommunikation aufbauen. Über APIs können diverse Informationen, wie zum Beispiel Dokumente oder Antworten auf Rechenanfragen, übertragen werden. In einer sogenannten API-Specification wird festgelegt, wie über die API mit dem Programm kommuniziert werden kann. Damit dies einfach und transparent ist, ist eine gute Dokumentation von APIs besonders wichtig. Durch APIs können Anwendungen einfacher erweitert werden, da zusätzliche Module nicht zwingend in den alten Quelltext eingepflegt werden müssen, sondern über die API mit der Anwendung integriert werden können [9].

2.1.5 Priorisierung von Anforderungen

Um auswählen zu können, welche Features in einen MVP aufgenommen werden sollen, ist es sehr hilfreich, alle Anforderungen des Produktes zu priorisieren, um diese zu ordnen. Einige der viele Möglichkeiten, diese Priorisierung durchzuführen werden hier kurz dargestellt.

Stack-Ranking

Bei der Stack-Ranking-Methode wird jedem Feature eine Zahl zugewiesen, welche die Priorität darstellt. Die Nummer Eins stellt hierbei die höchste Priorität dar, während n (bei n Features) die niedrigste Priorität darstellt. Ein Vorteil des Stack Rankings ist, dass es nur ein Feature auf Platz eins geben kann. Dadurch kann vermieden werden, dass vielen Anforderungen gleichzeitig eine zu hohe Priorität zugeordnet wird. Dieser Vorteil kann aber auch gleichzeitig ein Nachteil sein, wenn Features vorhanden sind, die genau gleich hoch priorisiert werden müssten. Ein weiterer Vorteil ist der Fakt, dass eine Nummerierung sehr einfach und schnell zu verstehen ist und so zu weniger Missverständnissen bei der Kommunikation führen kann [4].
Für den MVP bietet es sich an, die Features, die am Anfang der Liste stehen zu implementieren. Wie weit die Liste hierbei abgearbeitet wird, hängt vom geplanten Umfang des MVP sowie dem Umfang der Features ab.

Kano-Modell

Das Kano-Modell wurde von Professor Noriaki Kano entwickelt. Bei diesem Modell werden die Features in die folgenden fünf Kategorien eingeteilt [4]:

1. **Must-Be**: Diese Features werden von den Kund:innen erwartet. In der Anwendung müssen sie implementiert sein, da das Produkt sonst als unfertig angesehen wird.

2. **Attractive**: Features, die nicht fehlen, wenn sie nicht vorhanden sind, aber einen echten Mehrwert bringen, wenn sie vorhanden sind.

3. **One-Dimensional**: Features, die den Nutzer:innen gefallen, wenn sie vorhanden sind, ihnen aber auch fehlen, wenn sie nicht vorhanden sind.

4. **Indifferent**: Features, die keinen direkten Einfluss auf das Nutzungserlebnis der Nutzer:innen haben, wie zum Beispiel die Dokumentation der Codebasis, aber für die Entwickler:innen im Laufe der Entwicklung hilfreich sind.

5. **Reverse**: Features, die die Kund:innen stören, aber notwendig sind. Als Beispiel ist hier der Zwang zur zwei-Faktor-Authentisierung vorstellbar, da vielen Nutzer:innen die gewonnene Sicherheit egal ist, sie sich aber an dem zusätzlichen Schritt stören.

Mithilfe dieser Kategorien können nun Features für den MVP leichter gewählt werden. Es bietet sich an, viele Must-Be und Attractive-Features im MVP zu implementieren, während Indifferent- und Reverse-Features erst im weiteren Laufe der Entwicklung implementiert werden sollten [20]. Bei One-Dimensional-Features sollte man pro Feature abwägen, ob dies so implementiert werden kann, dass es nicht zu Unzufriedenheit der Nutzer:innen führt.

MoSCoW-Methode

Die MoSCoW-Methode weist Ähnlichkeiten in der Kategorisierung der Features zur Kano Methode vor. Diese Kategorien unterscheiden sich jedoch voneinander und sind deshalb nicht untereinander austauschbar. Die Features werden in folgenden vier Kategorien eingeteilt [4]:

1. **Must-Have**: Ohne diese Features kann das Produkt nicht veröffentlicht werden, da entweder elementare Funktionen fehlen oder die Anwendung nicht legal wäre (zum Beispiel nicht DSGVO konform). Sobald es eine Möglichkeit gibt, das Produkt ohne eins dieser Features auszuliefern, handelt es sich nicht mehr um ein Must-Have, sondern ein Should-Have Feature.

2. **Should-Have**: Diese Features sind äußerst wichtig für den Erfolg des Produktes. Es führt zu Unzufriedenheit der Nutzer:innen, wenn sie nicht implementiert werden, sie könnten aber rein theoretisch auch weggelassen werden.

3. **Could-Have**: Features, die weggelassen werden könnten, aber zu erhöhter Zufriedenheit der Nutzer:innen führen, wenn sie implementiert werden. Sie sind weniger wichtig als Should-Have Features.

4. **Won't-Have**: Diese Features werden in dem nächsten Release nicht implementiert, können aber im Backlog behalten werden, um diese zu einem späteren Zeitpunkt zu implementieren.

Der Name der Methode setzt sich aus den Anfangsbuchstaben der Kategorien wie folgt zusammen: **M**ust-Have, **S**hould-Have, **Co**uld-Have, **W**on't-Have.

Welche der aufgeführten Methoden verwendet wird, ist zweitrangig und hängt von den persönlichen Vorlieben der Entwickler:innen sowie dem Grund der Priorisierung ab. Wichtig ist jedoch in jedem Fall, dass eine klare Priorisierung der Features erfolgen kann und diese gut zu kommunizieren ist.

2.1.6 Statische und dynamische Websites

Bei Websites wird zwischen statischen und dynamischen Websites unterscheiden. Statische Websites bestehen aus einer festen Menge an Inhalten, die sich erst ändern, wenn der Quellcode angepasst wird. Sie bestehen aus HTML, Javascript und CSS Inhalten, welche von dem Webserver zur Verfügung gestellt werden. Unabhängig davon, wer die Website aufruft, wird sie bei gleichen Voraussetzungen (Art des Endgerätes, Browser etc.) immer den gleichen Inhalt darstellen. Statische Websites können, mithilfe von Javascript, auf unterschiedlichen Endgeräten durchaus unterschiedlich dargestellt werden [17]. Die typische persönliche Website aus den 90er Jahren ist ein gutes Beispiel für eine statische Website.
Dynamische Websites haben ein Backend auf dem Webserver, welches dafür sorgt, dass auch ohne Änderungen am Quellcode neue Inhalte angezeigt werden können. Dies ist zum Beispiel möglich, indem User sich anmelden und für sie bestimmte Inhalte angezeigt bekommen, die der Server aus einer Datenbank lädt [17]. Ein gutes Beispiel für eine dynamische Website ist Facebook.

2.1.7 User Stories

User Stories sind eine vor allem in der agilen Softwareentwicklung verwendete Methodik, um Software-Features einfach und verständlich darzustellen. Dazu wird aus der Sicht der Benutzer:in ein Anwendungsbeispiel kurz beschrieben. User Stories vermitteln den Entwickelnden einen Kontext zu den jeweiligen Features, was die Kommunikation vereinfacht [13]. Meist werden User Stories als einfacher Satz aufgebaut und weisen folgendes Schema auf: „Als [Kundentyp] [möchte] ich, [damit]“ [13].
Der [Kundentyp] beschreibt dabei, welche Art von Kunde in der User Story gemeint ist. Beispiele dafür sind bei einem Blog Autor:innen oder Leser:innen. Das [möchte] beschreibt die Absicht der Kund:in. Ein Beispiel dafür ist, dass eine Leser:in Artikel lesen

möchte. Unter dem [damit] Teil der User Story wird der Nutzen für die Benutzer:in beschrieben [13]. Um das vorherige Beispiel fortzuführen: ‚Als Leser:in möchte ich Artikel lesen können, damit ich immer auf dem aktuellen Stand bleibe‘.

2.2 Architekturprinzipien

Um die Motivation hinter verschiedenen Architekturmustern nachzuvollziehen, ist es notwendig, einige grundlegende Prinzipien zu verstehen. Die dafür wichtigsten sind in diesem Kapitel aufgeführt.

2.2.1 Prinzip der losen Kopplung und hohen Kohäsion

Unter der Kopplung eines Systems versteht man die Abhängigkeiten von Systembausteinen untereinander. Es ist zwischen unterschiedlich starken Abhängigkeiten zu unterscheiden. Greifen Klassen beispielsweise auf die privaten Daten einer anderen Klasse direkt zu, besitzen diese Klassen eine hohe Kopplung, da zur Änderung der einen Klasse Wissen über die internen Strukturen der anderen Klasse notwendig ist. Ist der Datenzugriff im Gegensatz dazu über einen Methodenaufruf realisiert, spricht man zwar immer noch von einer Kopplung, diese ist aber wesentlich loser als in dem vorher genannten Beispiel, da keine Kenntnis der inneren Struktur der anderen Klasse notwendig ist. In einem System ist eine lose Kopplung erstrebenswert, da dies die Wartung und Anpassung von Bausteinen sowie des Gesamtsystems erleichtert [31].
Ein weiteres Prinzip, auf das in Zusammenhang mit loser Kopplung geachtet werden sollte, ist das Vermeiden von zirkulären Abhängigkeiten. Dies liegt vor, wenn Komponente A von Komponente B abhängig ist, welche dann wiederum von Komponente C abhängig ist. Komponente C ist jedoch von Komponente A abhängig, wodurch eine zirkuläre Abhängigkeit entsteht (siehe Abbildung 2.1). Diese zirkulären Abhängigkeiten sollten vermieden werden, da sie zu einer hohen Kopplung führen. Veränderungen in einer Komponente führen dazu, dass gegebenenfalls alle Komponenten des Zyklus angepasst und getestet werden müssen.

Kohäsion beschreibt die interne Abhängigkeit der Komponente beziehungsweise, wie gut die Komponente ihre spezifische Aufgabe erfüllt. Mit einer losen Kopplung entsteht häufig eine hohe Kohäsion. Es sollte also darauf geachtet werden, dass für eine Aufgabe nicht

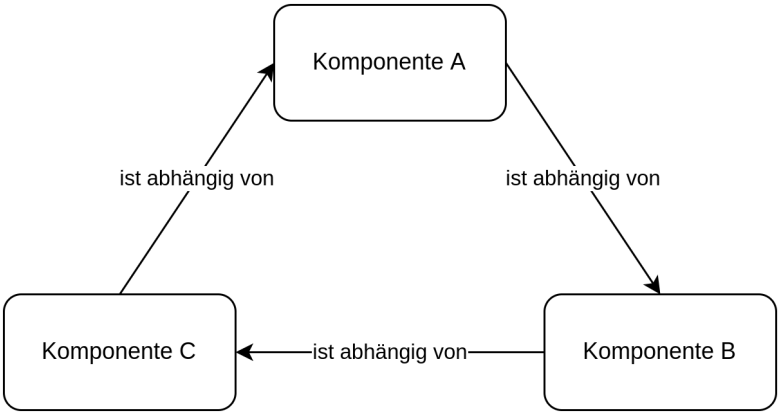

Abbildung 2.1: Darstellung zirkulärer Abhängigkeiten (eigene Darstellung)

mehrere Komponenten geschrieben werden, die dann wiederum einen hohe Kopplung besitzen, sondern die Funktionalität in einer Komponente gebündelt wird [31].

2.2.2 Separation of Concerns

Separation of Concerns (engl. für Trennung von Belangen) ist ein Prinzip, das neben der Softwarearchitektur auch in vielen Bereichen der Softwareentwicklung Anwendung findet. Es beschreibt die Praxis, Probleme in kleinere, weniger komplexe Teilbereiche aufzutrennen und folgt damit dem römischen Prinzip „Teile und Herrsche“ (oft auch als „Divide and Conquer“ bezeichnet) [31].
Das Prinzip lässt sich auf beliebigen Abstraktionsebenen anwenden. So sieht es vor, dass einzelne Module nur wenige Aufgaben haben, aber auch auf Klassen- und Methodenebene nur jeweils ein bestimmtes Belangen umgesetzt wird [23] [31].
Wird das Prinzip konsequent umgesetzt, führt es zu einer losen Kopplung sowie einer hohen Kohäsion und ermöglicht eine einfache Wartung des Systems. Auftretende Probleme können entsprechend geteilt und dann von mehreren Teams parallel gelöst werden [31].

2.2.3 Entwurf für Veränderung

Da sich Software in einem ständigen Wandel befindet, handelt es sich bei Anwendungen selten um statische Produkte. Werden zukünftig nötige Veränderungen bereits während des Entwurfsprozesses antizipiert, spricht man vom sogenannten Entwurf für Veränderung (englisch: Design for Change) [31].

Das Prinzip ist sehr allgemeingültig und sagt im Grunde nur aus, dass die Architektur bestmöglich auf spätere Veränderungen vorbereitet sein sollte. Dabei ist zwischen erwartbaren und nicht erwartbaren Änderungen zu unterscheiden. Während erwartbare Änderungen explizit vorbereitet werden können, muss das System für nicht erwartbare Änderungen so entworfen werden, dass es generell erweiterbar ist.
Wird während des Entwurfs auf eine lose Kopplung geachtet, ist der Entwurf für Veränderung häufig leichter umzusetzen. Ein Nachteil dieses Prinzips ist, dass änderbare Architekturen häufig weniger performant sind als statische, weshalb immer zwischen Leistung und Änderbarkeit abgewogen werden muss [31].

2.2.4 Information-Hiding-Prinzip

Für das Verstehen von Anwendungen können zu viele Informationen schnell kontraproduktiv sein. Dieses Problem soll das Information-Hiding-Prinzip lösen. Es besagt, dass Klient:innen nur die Informationen präsentiert bekommen, die unbedingt notwendig sind. Alle restlichen Informationen werden vor den Klient:innen verborgen. Das Prinzip ist von großer Bedeutung hinsichtlich des Verständnisses von großen Anwendungen und sollte deswegen unbedingt beachtet werden [31].
Eine Anwendung des Information-Hiding-Prinzips findet sich in der Modularisierung, weil dadurch die Module nach außen nur durch Schnittstellen Informationen austauschen. Sind keine Informationen über die inneren Strukturen bekannt, spricht man von sogenannten Black-Boxes [31].
Durch Information-Hiding wird eine losere Kopplung erzwungen, da für eine hohe Kopplung Kenntnisse über innere Strukturen notwendig sind.

2.3 Architekturstile

Da Softwarearchitekturen komplex sein können, ist es an diesem Punkt nicht hilfreich, einzelne Architekturen vorzustellen. Stattdessen soll es in dem folgenden Kapitel um Architekturstile gehen. Unter einem Architekturstil versteht man eine Menge von architektonischen Lösungen für Probleme, welche sich zu einer nicht vollständig spezifizierten Architektur zusammensetzen. Der Stil beinhaltet Vorschläge für zu verwendende Muster und Topologien der Komponenten [15].

Verschiedenste Architekturmuster, an denen man sich während der Softwareentwicklung orientieren kann, sind über die Jahre entstanden. Einige davon sollen hier kurz beleuchtet werden, um einen Überblick zu schaffen.

2.3.1 Monolith

Der einfachste zu denkende Architekturstil wird als Monolith bezeichnet. Hierbei wird das gesamte System in einer einzigen Komponente realisiert, was dafür sorgt, dass keine Separation of Concerns erreicht wird. Monolithen entstehen oft eher aus historisch gewachsenen Systemen, als dass sie aktiv geplant werden. Eine lose Kopplung ist nicht zu erreichen, da keine gekoppelten Bausteine vorhanden sind, deshalb kann man weder von loser noch enger Kopplung sprechen [31].
Monolithen zeichnen sich dadurch aus, dass die Anpassung sowie das Testen aufwendig sind. Meist gibt es nur wenige Entwickler, die das gesamte System ausreichend gut verstehen, um Änderungen umzusetzen. Fluktuation im Unternehmen führt deshalb zu einer erschwerten Wartung monolithischer Systeme. Generell sollte ein klassischer Monolith nicht das Ergebnis einer Architekturplanung sein [31].

2.3.2 Model View Controller

Das Model View Controller Muster (im Folgenden als MVC bezeichnet) unterteilt die Verantwortlichkeiten bei Benutzerschnittstellen in drei Rollen. Die drei Rollen sind das Model, welches für die Verwaltung der darzustellenden Daten verantwortlich ist, die View, die für die Darstellung sorgt und der Controller, welcher Benutzereingaben verwaltet und Daten ändert. Der Controller sorgt außerdem für die Aktualisierung der View. In Abbildung 2.2 werden diese Abhängigkeiten dargestellt.
Vorteile des Musters sind unter anderem, dass die Benutzeroberfläche verändert werden kann, ohne dass das Model davon betroffen ist. Dadurch ist dies sogar zur Laufzeit möglich. Außerdem kann der Änderungsaufwand präziser eingeschätzt werden, da klar ist, welche Teile betroffen sind. Ein Nachteil ist, dass der Implementierungsaufwand höher ausfallen könnte, da Schnittstellen zwischen den Komponenten umgesetzt werden müssen. Generell beruht das Model View Controller Muster auf dem Prinzip der Trennung der Verantwortlichkeiten [23].

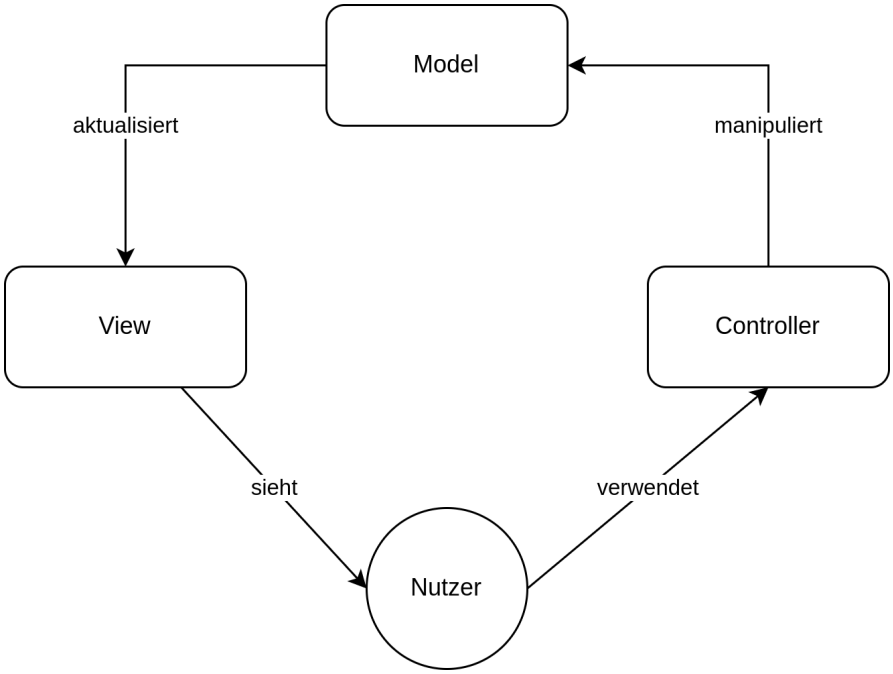

Abbildung 2.2: Abhängigkeiten im Model View Controller Muster (eigene Darstellung nach [23])

2.3.3 Model View ViewModel

Bei dem Model View ViewModel (im Folgenden als MVVM bezeichnet) Muster handelt es sich um eine Variation des MVC Musters. Es dient dazu, sowohl die Geschäftslogik als auch die Präsentationslogik von der View zu trennen. Die View und das Model übernehmen die gleichen Funktionalitäten wie im MVC-Muster. Der Unterschied liegt darin, dass kein Controller verwendet wird, sondern ein ViewModel in Verbindung mit einem Binder. Dabei handelt es sich bei dem ViewModel um eine Abstraktion der View. Das ViewModel beinhaltet die Präsentationslogik der View und kommuniziert mit dem Model. Durch den Binder wird die View an Eigenschaften in dem ViewModel gebunden, wodurch eine implizite Kommunikation zwischen dem ViewModel und der View möglich ist. Außerdem wird die Ausführung der Geschäftslogik durch das ViewModel im Model angestoßen [27].
In Abbildung 2.3 wird der beschriebene Zusammenhang der Komponenten dargestellt.

Das Ziel des Designmusters ist es, dass die Präsentationslogik unabhängig, von der grafischen Benutzeroberfläche (im Folgenden auch als User-Interface oder UI bezeichnet), ist und dieses dadurch austauschbar wird. In dem ViewModel werden alle Daten gehalten, die

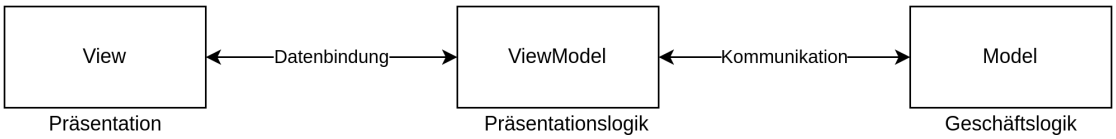

Abbildung 2.3: Darstellung des MVVM-Musters (eigene Darstellung nach [27])

durch die View veränderbar sind. Dazu zählen nicht nur die Inhalte von Eingabefeldern, sondern auch beispielsweise der Status von Schaltern (an- oder abgewählt) [5]. Ein großer Vorteil des MVVM-Musters ist, dass sich das ViewModel einfach testen lässt. Es lässt sich ohne aufwendige, automatisierte User-Interface Tests testen. Ein Nachteil des Musters ist, dass es nicht immer einfach ist, im Voraus ein ViewModel zu entwerfen, welches allgemein genug ist, um alle Möglichkeiten der Benutzeroberfläche abzudecken [26].

2.3.4 Client-Server-Modell

In Webanwendungen ist ein Client-Server-Modell heutzutage der etablierte Standard. Hierbei wird auf einem Rechner eine Anwendung, der Client, betrieben welcher auf zentral verwaltete Ressourcen, den Server, zugreift. Dabei können mehrere Clients auf einen Server zugreifen, es können aber auch mehrere Server für verschiedene Ressourcen zur Verfügung stehen. Der Hauptvorteil einer solchen Architektur ist, dass Anwendungen nicht als ganzes immer vollständig an die Anwender:innen übertragen werden müssen, sondern nach und nach, mittels Anfragen an den Server, bereitgestellt werden [31].

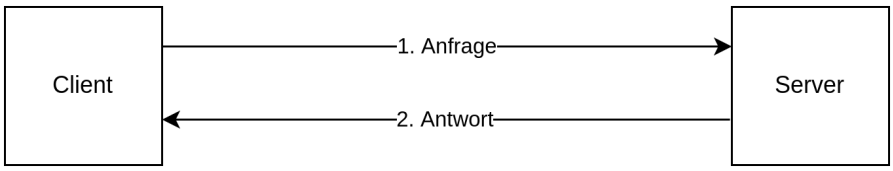

Abbildung 2.4: Informationsaustausch im Client-Server-Modell (eigene Darstellung nach [31])

Welche der, im Folgenden aufgeführten, Variationen der Client-Server-Architektur gewählt werden sollte, hängt davon ab, wie die Arbeit zwischen Client und Server aufgeteilt werden soll.

Thin Client

Als Thin Client wird ein schlanker Client bezeichnet, der möglichst wenig Arbeit übernimmt. Hier ist meist ein einfacher Webbrowser ausreichend, um die auf HTML, Javascript und CSS basierten Anwendungen auszuführen [25].
Dadurch, dass der Server den Großteil der Berechnungen übernimmt, müssen mehr Daten ausgetauscht werden. Das belastet das Netzwerk und den Server. Aus diesem Grund bieten sich Thin Client Architekturen für Anwendungen in einer Umgebung mit stabiler Netzanbindung an.

Rich Client

Ein Rich Client beinhaltet alle Funktionen des Thin Clients, kann aber sowohl eine dedizierte Benutzeroberfläche bieten [25] als auch Daten speichern [31]. Werden Daten lokal gespeichert, bietet dies den Vorteil, dass der Client weniger abhängig von der Netzanbindung ist. Diese Daten müssen aber bei Änderungen synchronisiert werden, um aktuell gehalten zu werden. Softwareupdates werden ebenfalls über das Netzwerk ausgeliefert [31].

Fat Client

Klassische Desktopanwendungen werden als Fat Client bezeichnet. Auch diese können nutzerspezifische Daten über Webservices von einem Server beziehen, werden aber lokal auf einem Rechner installiert [25].
Fat Clients sind eine Möglichkeit zur Skalierung von Anwendungen, um bei ausreichend Rechenkapazitäten des Clients den Server zu entlasten. In den meisten Fällen haben Fat Clients wenig Abhängigkeiten vom Netzwerk und können, wenn auch meist mit Einschränkungen der Funktionen, ohne Netzanbindung betrieben werden [25].

2.3.5 Peer-to-Peer

Als Gegensatz des zentralisierten Client-Server-Modells existiert die Peer-to-Peer Architektur (auch häufig als P2P bezeichnet). Hier kommunizieren die Klienten direkt miteinander bzw. dezentral, anstatt mit einem zentralen Server. Die kommunizierenden Klienten

werden als Peers bezeichnet und gleichwertig behandelt. Dienste, die das System anbietet, und der Zustand des Systems sind zwischen den einzelnen Peers verteilt [31].
Ein Nachteil der P2P Architektur ist, dass eine Aktualisierung von Informationen aufwendiger ist, da das gesamte Netzwerk synchronisiert werden muss. Im Gegensatz dazu lässt sich in einer Client-Server-Architektur die Information auf dem Server hinterlegen. Dies kann dazu führen, dass eine P2P-Architektur weniger leistungsfähig als eine Client-Server-Architektur ist.

Es kann ein zentraler Server verwendet werden, um neue Peers in das System einzupflegen. In diesem Fall wird die Architektur als Hybrid zwischen P2P und dem Client-Server-Modell bezeichnet [31].

2.3.6 Schichten Architektur

Bei einer Schichtenarchitektur ist die Software in diskrete Schichten unterteilt. Diese Schichten können je nach Abstraktionsgrad als Funktionalitäten, Komponenten oder Klassen betrachtet werden.
Bei einer geschlossenen Schichtenarchitektur kommunizieren die Schichten nur mit der jeweils nächsten Schicht, was dafür sorgt, dass Schnittstellen zwischen den Schichten klar definiert und begrenzt sind. Die Schichtenarchitektur setzt damit das Information-Hiding-Prinzip um. Dies führt zu einer effizienten Wartbarkeit, da die Abhängigkeiten der einzelnen Schichten dadurch gering gehalten werden [31].
Können Schichten übersprungen werden, spricht man von einer offenen Schichtenarchitektur [18]. Dies kann dazu führen, dass die Anwendung eine bessere Performance aufweist, da Daten bei einer geschlossenen Schichtenarchitektur alle Schichten passieren müssen, die zwischen zwei Schichten liegen, auch wenn diese dort nicht verarbeitet werden müssen. Allerdings führt die offene Architektur zu einer hohen Kopplung, da die einzelnen Schichten im Extremfall zu jeder anderen Schicht Abhängigkeiten besitzen.
Eine Schichtenarchitektur in einem Client-Server-Modell ist hilfreich, um hohe Lasten verarbeiten zu können. Die Benutzeroberfläche läuft hier auf dem Rechner der Anwender:in, während der Server auf mehrere Schichten und auch Rechnern verteilt sein kann. Dadurch kann die erste Serverschicht beispielsweise für die Lastverteilung auf die Server danach zuständig sein. Die folgenden Server könnten dann auf einen Datenbankserver, der als eigene Schicht realisiert ist, zugreifen. Durch eine solche Schichtenarchitektur lassen sich die einzelnen Schichten gut skalieren [31].

2.3.7 REST

REST steht für ‚Representational State Transfer' (auch als RESTful-API bezeichnet) und ist ein Architekturstil, der bezüglich der Übertragung von Daten zwischen Anwendungen über das Netzwerk Vorschläge enthält. Es handelt sich nicht um einen Standard oder ein Protokoll, sondern vielmehr einen Architekturstil, der auf unterschiedliche Weisen implementiert werden kann [10]. Dabei werden die Daten über die existierenden Web-Standards HTTP und URI übertragen. An dieser Stelle ist nur wichtig zu wissen, dass es sich bei HTTP (Hypertext Transfer Protocol) um das im Web verwendete Übertragungsprotokoll [7] und bei URIs (Uniform Resource Identifier) um Zeichenfolgen zur Identifizierung einer Ressource handelt [6]. In HTTP sind verschiedene Anfragetypen definiert, die an dieser Stelle jedoch nicht alle weiter beschrieben werden. Zum besseren Verständnis folgt nur eine kurze Beschreibung der GET-Anfrage. Es handelt sich um eine Anfrage, bei der als Antwort der Inhalt der angefragten Ressource zurückgegeben wird [7].
Rest-Schnittstellen sind HTTP-Requests an eine, je nach Schnittstelle definierte URI [31]. Ein Beispiel hierfür wäre eine GET-Anfrage an die URI:

```
www.api.haw-hamburg.de/studenten/2435451
```

um den Studenten mit der Matrikelnummer 2435451 zu erhalten.
Diese Daten werden meist im JSON-Format (Javascript Object Notation) zurückgeliefert, da es sich hierbei um ein, von vielen Programmiersprachen verarbeitbares, Format handelt, das auch für Menschen gut lesbar ist [10].
Zusätzlich zu den bereits beschriebenen Kriterien müssen RESTful-APIs außerdem zustandslos sein. Von zustandslosen Prozessen wird gesprochen, wenn diese keinerlei Kenntnis von vergangenen Transaktionen besitzen und sich bei subsequenten Ausführungen nicht verändern. Es werden also keine Zustände des Systems gespeichert oder verwendet [12].

2.3.8 Service Oriented Architecture

Die serviceorientierte Architektur (im Folgenden als SOA bezeichnet) ähnelt einem modularen Ansatz in vielen Teilen, erweitert diesen jedoch, um komplexere Systeme realisieren zu können. Bausteine werden hier als verteilte, lose gekoppelte Dienste (englisch: Services) umgesetzt. Die Dienste werden über den sogenannten Enterprise Service Bus (ESB) zur Verfügung gestellt. Über den ESB können die Dienste auf Funktionen von anderen Diensten im Unternehmensnetzwerk zugreifen und Funktionen von diesen verwenden. Dadurch

wird vermieden, dass Funktionen mehrmals implementiert werden müssen.
Um eine optimale Zusammenarbeit der Services zu gewährleisten, ist es ideal, wenn diese idempotent, zustandslos sowie transaktional abgeschlossen sind. Als idempotent werden Vorgänge bezeichnet, die immer zu dem gleichen Ergebnis führen, wenn sie mit der gleichen Eingabe durchgeführt werden, unabhängig davon, wie oft sie ausgeführt werden [31]. Transaktional abgeschlossene Vorgänge gehen von einem zulässigen Zustand wieder in einen zulässigen Zustand des Systems. Ist kein zulässiger Zustand nach der Transaktion gegeben, wird diese rückgängig gemacht, um das System wieder in den Zustand vor der Transaktion zu überführen [31].
SOA Architekturen haben gegenüber monolithischen Architekturen einige im Folgenden beschriebene Vorteile. Durch den modularen Aufbau lassen sich Teile von alten Anwendungen für neue verwenden. Dies ist besonders in einem Unternehmenskontext hilfreich, in dem viele ähnliche Anwendungen entwickelt werden. Der Kern dieser Anwendungen kann übernommen werden und die Entwickler müssen nicht wie bei monolithischen Anwendungen jedes Mal von vorne beginnen [11].
Da die einzelnen Services übersichtlicher als ein Monolith sind, lassen sie sich einfacher warten. Die Skalierbarkeit einer SOA ist einfacher zu erreichen, da meist nicht alle Teile einer Anwendung skalieren müssen und sich die Entwickler durch die getrennten Services auf jene, die skalieren müssen, konzentrieren können.

2.3.9 Microservices

Microservices bezeichnen einen Architekturstil, bei dem Funktionalitäten in voneinander getrennte, austauschbare Services aufgeteilt werden. Das Ziel ist es, eine leichte Ersetzbarkeit und damit eine bessere Wartbarkeit einzelner Bausteine zu gewährleisten. Die Services werden meist von kleinen Teams entwickelt. Das ist möglich, da die Kommunikation der Services über vorher definierte APIs realisiert wird und dadurch eine lose Kopplung erreicht wird. Außerdem handelt es sich bei Microservices um kleinere Bausteine, die durch ihren Umfang leicht zu überblicken sind.

Im Folgenden werden zunächst einige Vorteile aufgelistet [21]:

1. Die Auswahl der verwendeten Technologien erfolgt auf Microserviceebene und bietet dadurch einen hohen Freiheitsgrad.

2. Durch die geringe Größe der einzelnen Services ist weniger Planung und Organisation nötig, als bei einem monolithischen Projekt. Dies ermöglicht den Einsatz von simpleren, agilen Projektmanagementmethodiken, wie zum Beispiel Scrum anstelle von traditionellen, umfangreicheren Modellen.

3. Nicht funktionale Anforderungen können pro Service einzeln und damit feingranularer definiert werden.

4. Treten Fehler auf, können diese meist schnell lokalisiert und damit auch schneller behoben werden und andere Services sind nur von dem Fehler betroffen, wenn sie Abhängigkeiten zu der fehlerhaften Komponente haben.

Folgende Nachteile sollten bei dem Entwurf eines Microservices berücksichtigt werden:

1. Microservices sind als verteiltes System zu sehen und bringen damit Herausforderungen mit sich. Die Fehlersuche wird dadurch erschwert, dass die Kommunikation über das Netzwerk stattfindet und somit schwerer zu verfolgen ist. Dies führt außerdem dazu, dass Aufrufe zwischen den Services durch externe Fehler, wie etwa Paketverlust, schlechte Internetverbindungen oder Ähnlichem fehlschlagen können.

2. Ändern sich Schnittstellen, müssen die Anpassungen meist mehrere Teams umsetzen, was zu Fehlern führen kann.

3. Ein hohes Maß an Automatisierung ist nötig, um Microservices effizient zu testen und zu deployen.

Generell sind mehrere Variationen der Realisierung von Microservices vorstellbar. Im Folgenden werden zwei grundlegende vorgestellt.

Horizontale Microservices

Bei horizontalen Microservices setzt die Benutzeroberfläche auf einer Menge von Microservices auf und kommunizieren über eine API mit ihnen. Diese Kommunikation wird üblicherweise durch ein API-Gateway realisiert. Untereinander kommunizieren die Services über die Service-Discovery-Komponente (siehe Abbildung 2.5). Welche Aufgaben beide Komponenten jeweils übernehmen, ist nicht vorgeschrieben und kann sich von Implementierung zu Implementierung unterscheiden. Horizontale Microservices bieten sich an, wenn mehrere UI-Komponenten, wie zum Beispiel eine App- und eine Webansicht, geplant sind [21].

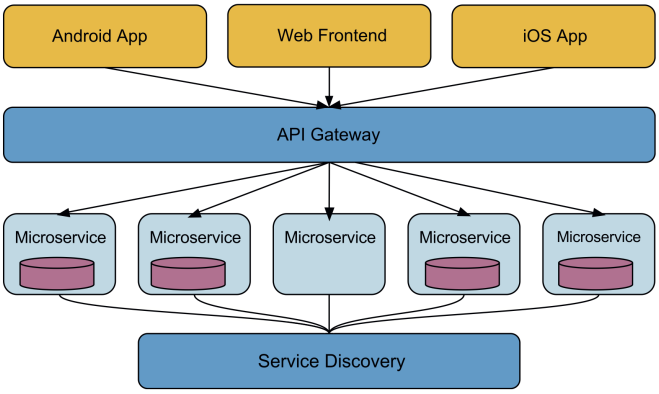

Abbildung 2.5: Horizontale Variante von Microservices (Abbildung aus [21])

Vertikale Microservices

Im Gegensatz zu der horizontalen Variante werden bei der vertikalen Variante die UI-Komponenten mit in die jeweiligen Services aufgenommen. Die Services werden dann nach entsprechenden Subdomänen klar getrennt (siehe Abbildung 2.6). Durch diese Variante wird die Kopplung auf ein Minimum reduziert. Es bietet sich an, die vertikale Variante immer zu verwenden, wenn nichts speziell für die Horizontale Variante spricht [21].

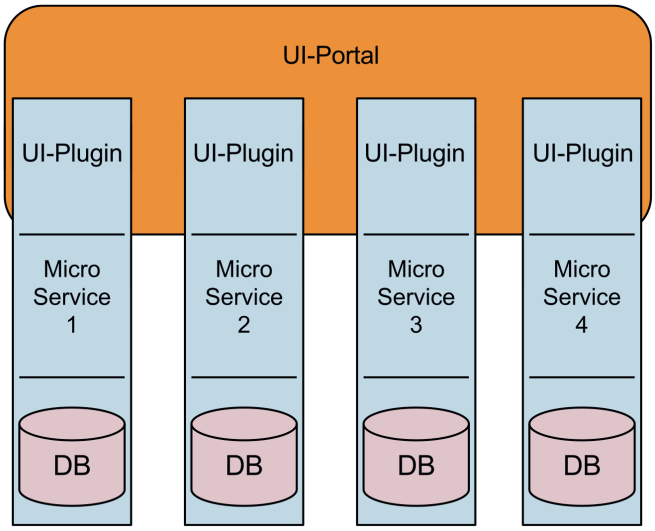

Abbildung 2.6: Vertikale Variante von Microservices (Abbildung aus [21])

2.3.10 Makroservices

In großen Projekten können die Nachteile von Microservices ein großes Hindernis darstellen, weshalb Mischformen von Monolithen und Microservices verwendet werden. Man spricht auch von Makroservices [1]. Hierbei werden einzelne Microservices zu Einheiten gebündelt, die dann wiederum einen eigenen Service darstellen [31].
Es bietet sich vor allem an, hybride Architekturen zu verwenden, wenn ein monolithisches Softwareprodukt erweitert werden soll. Der Monolith muss dann nur um eine Schnittstelle zu dem neuen Service erweitert werden.

3 Erstes Fallbeispiel (Bandrecording Hamburg)

Bei Bandrecording Hamburg handelt es sich um eine Webanwendung, mit der eine Aufnahme in einem professionellen Tonstudio konfiguriert werden kann, sodass sie den individuellen Anforderungen der Musiker:innen entspricht. Dies erhöht die Transparenz bezüglich des Preises für eine Aufnahme (auch Recording genannt) deutlich. Auch für das Tonstudio bietet die Plattform einen Mehrwert, indem die Anforderungen an die Aufnahme und deren Preisgestaltung automatisiert werden. Die Preisgestaltung erfolgt bei Tonstudios in der Regel individuell und nur auf explizite Anfrage von Künstler:innen. Im folgenden Schritt werden dann feingranular die Anforderungen und der Umfang der Aufnahme skizziert, um einen realistischen Preisvorschlag zu unterbreiten. Bei der bisher gängigen Vorgehensweise zum Erhalt eines Preisvorschlags, kontaktieren die Musiker:innen die Tonstudios per E-Mail oder Telefon. Dabei muss auf die Zusammensetzung der Band eingegangen werden sowie deutlich werden, was aufgenommen werden soll. Dies stellt insofern ein Problem dar, dass gerade unerfahrene Musiker oft nicht wissen, welche Informationen das Studio benötigt. Dieses Problem wird durch den Konfigurator von BRHH adressiert und gelöst, da die Kund:innen nur noch ihre Daten in den Konfigurator eingeben müssen und daraufhin einen Preis angezeigt bekommen.
Der MVP von Bandrecording Hamburg ist bereits online und über die URL `www.bandrecording-hamburg.de` zu erreichen.

3.1 MVP des Produktes

Der Kern des Projektes ist bereits als Frontend-MVP umgesetzt worden. In der Oberfläche der Anwendung kann jeder Aufnahmeschritt individuell konfiguriert werden, was sich in einer Echtzeitanpassung des Gesamtpreises widerspiegelt (siehe Abbildung 3.1). Der Konfigurator umfasst dabei die, im Folgenden beschriebenen, fünf Schritte.

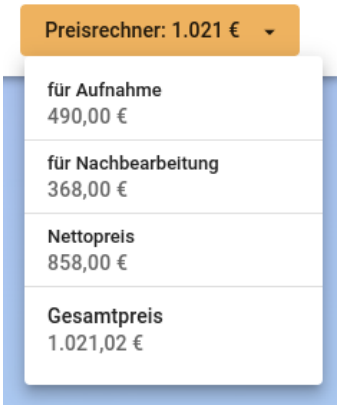

Abbildung 3.1: Preiskomponente des Konfigurators (eigene Darstellung)

3.1.1 Die Auswahl der Aufnahmeart

Im ersten Schritt (siehe Abbildung 3.2) wird die grundlegende Art der Aufnahme festgelegt. Dabei wird zwischen Live und Overdub entschieden. Fällt die Wahl auf eine Liveaufnahme, spielen die Musiker:innen die Songs, wie bei einem Konzert, gleichzeitig ein. Beim Overdub spielen die Musiker:innen nacheinander einzeln ihre Instrumente und diese werden in der Nachbearbeitung zu einem Song zusammengelegt.

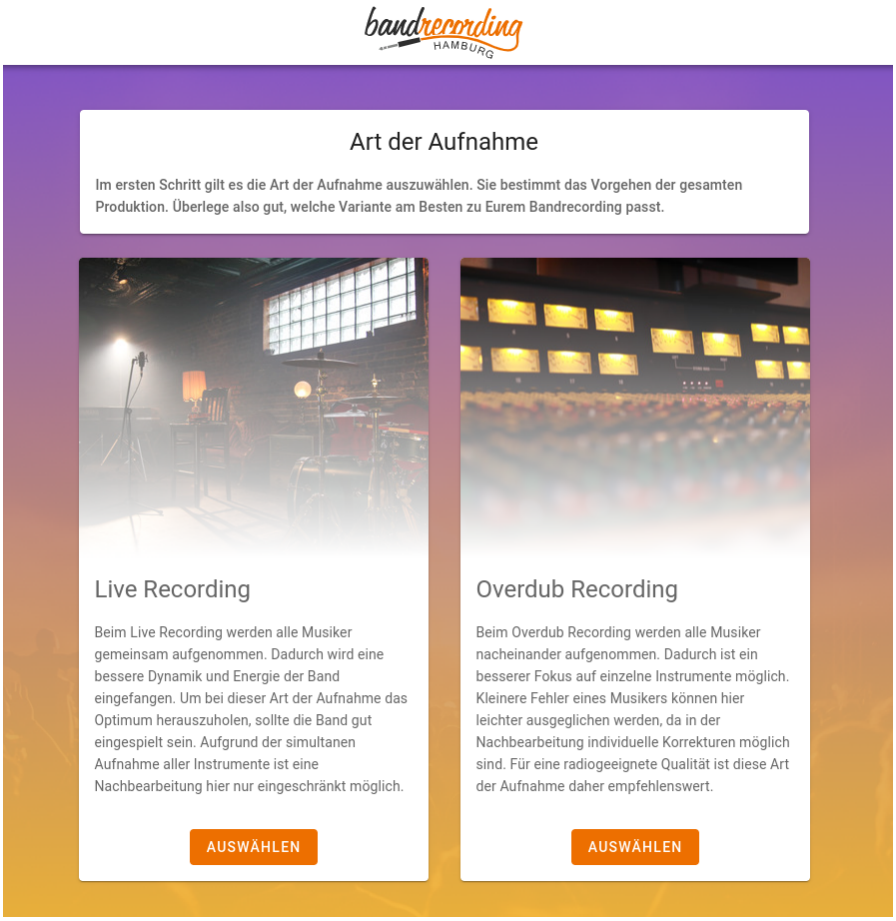

Abbildung 3.2: Auswahl der Aufnahmeart im ersten Schritt des Konfigurators (eigene Darstellung)

3.1.2 Die Konfiguration der Musiker:innen

In diesem Schritt (siehe Abbildung 3.3) werden die einzelnen Musiker:innen der Band angelegt. Jedem Bandmitglied können dabei mehrere Instrumente zugeordnet werden. Nachfolgend wird jedem Mitglied die Studioerfahrung, in den drei Abstufungen **keine**, **mittel** oder **viel**, zugeordnet, um den Zeitaufwand der Aufnahme konkreter einschätzen zu können. Des Weiteren können den Bandmitglieder:innen Namen zugewiesen werden, um sie einfacher auseinander zu halten. Weitere Musiker:innen können über einen **Plus**-Button hinzugefügt werden.

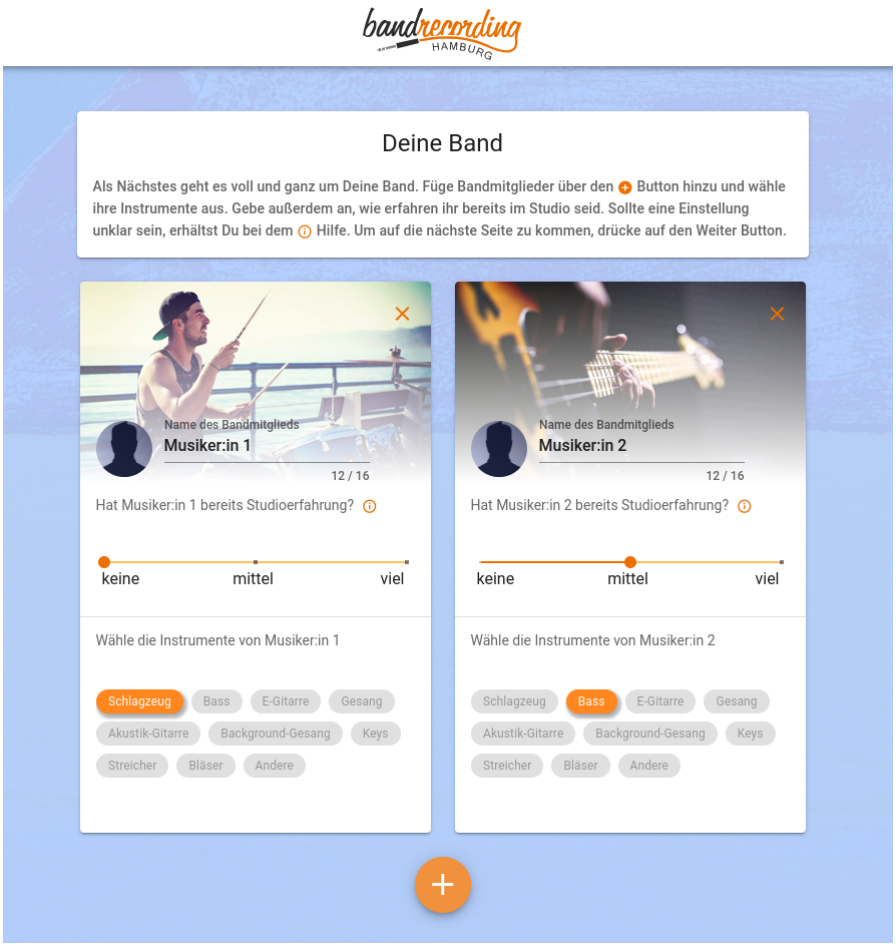

Abbildung 3.3: Konfiguration der Musiker:innen im zweiten Schritt des Konfigurators (eigene Darstellung)

3.1.3 Die Konfiguration der Songs

In diesem Schritt (siehe Abbildung 3.4) wird die Dauer des jeweiligen Songs in Minuten festgelegt. Dabei wird zwischen den drei Kategorien **unter 3**, **3 - 5** oder **über 5** unterschieden. Außerdem wird festgelegt, welche Instrumente der Song beinhaltet. Wie den Musikern:innen können auch den Songs Namen zugewiesen werden. Weitere Songs können über einen **Plus**-Button hinzugefügt werden. Ab diesem Schritt wurden genug Informationen gesammelt, um einen Preis anzeigen zu können. Dieser verändert sich bei jeder Eingabe entsprechend. Geht die Nutzer:in, nachdem mindestens ein Song hinzugefügt wurde, zurück zu einem der vorherigen Schritte, wird der Preis weiterhin angezeigt.

So können die Nutzer:innen bei jeder Option detailliert nachvollziehen, was diese am Preis der Aufnahme ändert.

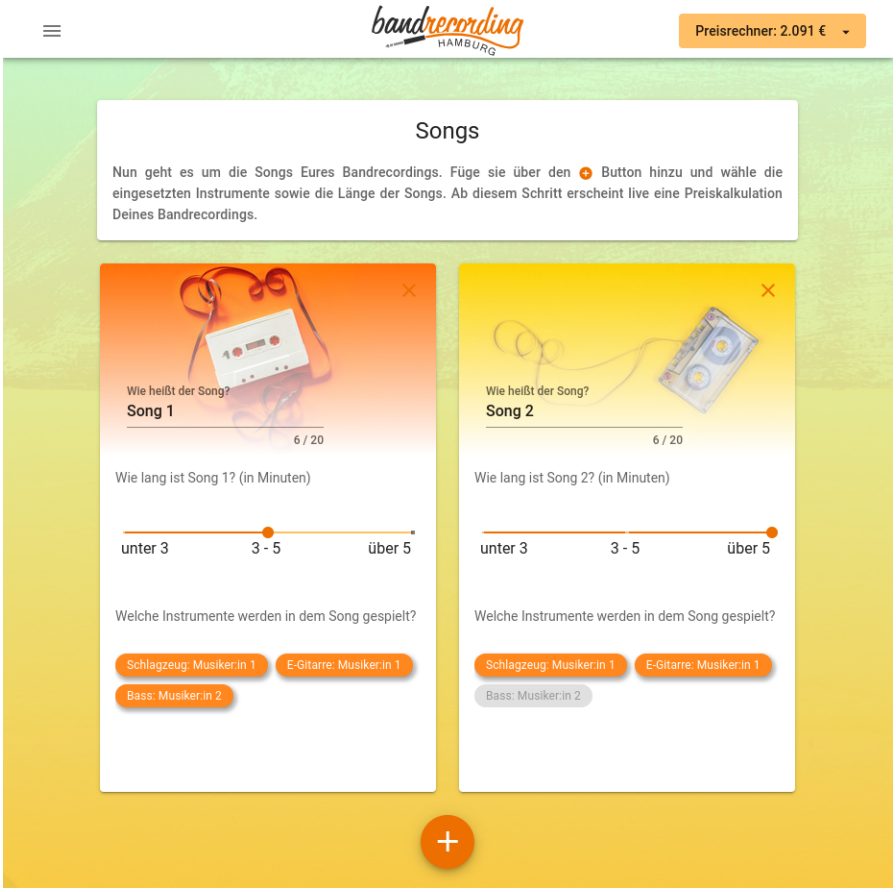

Abbildung 3.4: Konfiguration der Songs im dritten Schritt des Konfigurators (eigene Darstellung)

3.1.4 Die Konfiguration der Nachbearbeitung

In diesem Abschnitt (siehe Abbildung 3.5) wird der Umfang der Nachbearbeitung (engl. Post Production) definiert. Es kann ausgewählt werden, ob ein Edit, ein Mix und oder ein Master gewünscht sind. Außerdem können Nachbearbeitungsoptionen in den Abstufungen **Gold**, **Platin** oder **Diamond** gebucht werden. Diese Abstufungen leiten sich aus den in der Musikbranche vergebenen und als Schallplatten bezeichnete Preise für verkaufte Einheiten ab [14]. Genaueres zu diesen Nachbearbeitungsschritten ist an die-

ser Stelle nicht relevant. Außerdem können einige Zusatzoptionen, wie zum Beispiel die Tonkorrektur des Gesangs, dazu gebucht werden.

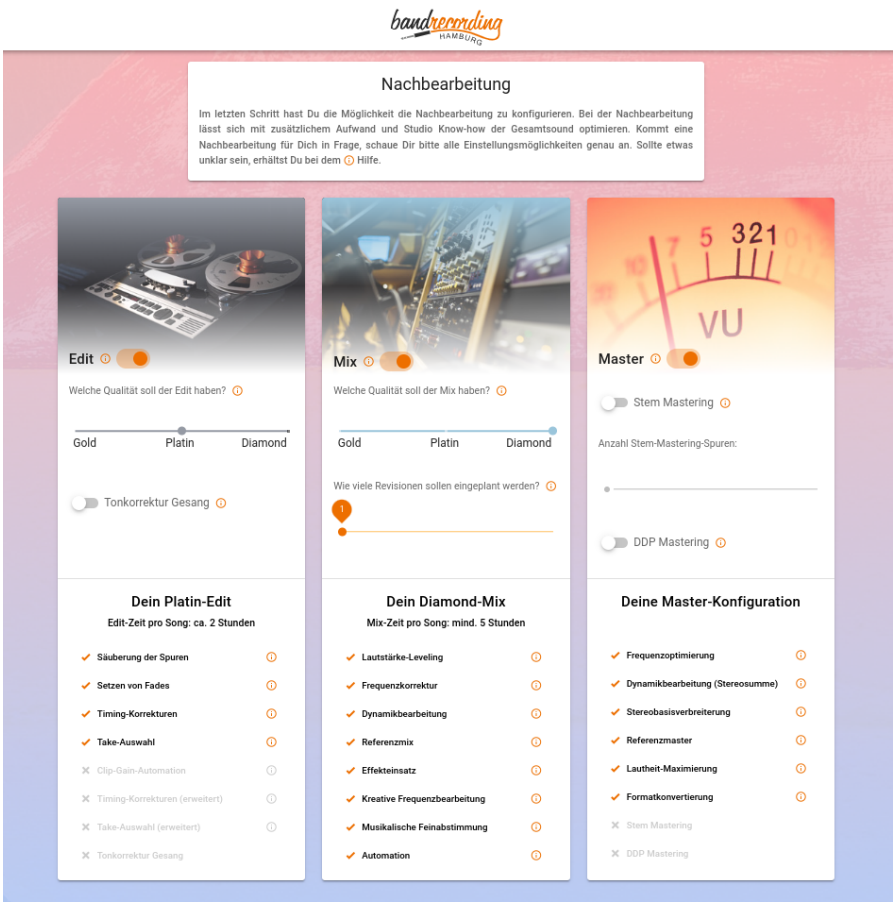

Abbildung 3.5: Konfiguration der Nachbearbeitung im vierten Schritt des Konfigurators (eigene Darstellung)

3.1.5 Zusammenfassung der Konfiguration

Im letzten Schritt (siehe Abbildungen 3.6 und 3.7) wird die Konfiguration abschließend übersichtlich dargestellt und alle Einstellungen aus den vorherigen Schritten können noch einmal angepasst werden. Außerdem können weitere Zusatzoptionen gebucht werden. Ein Beispiel ist der Flex-Discount, durch den die Aufnahme durch flexiblere Aufnahmezeiten, zum Beispiel auch nachts, günstiger wird. Abschließend wird noch der Name der Kund:in sowie die E-Mail-Adresse abgefragt.

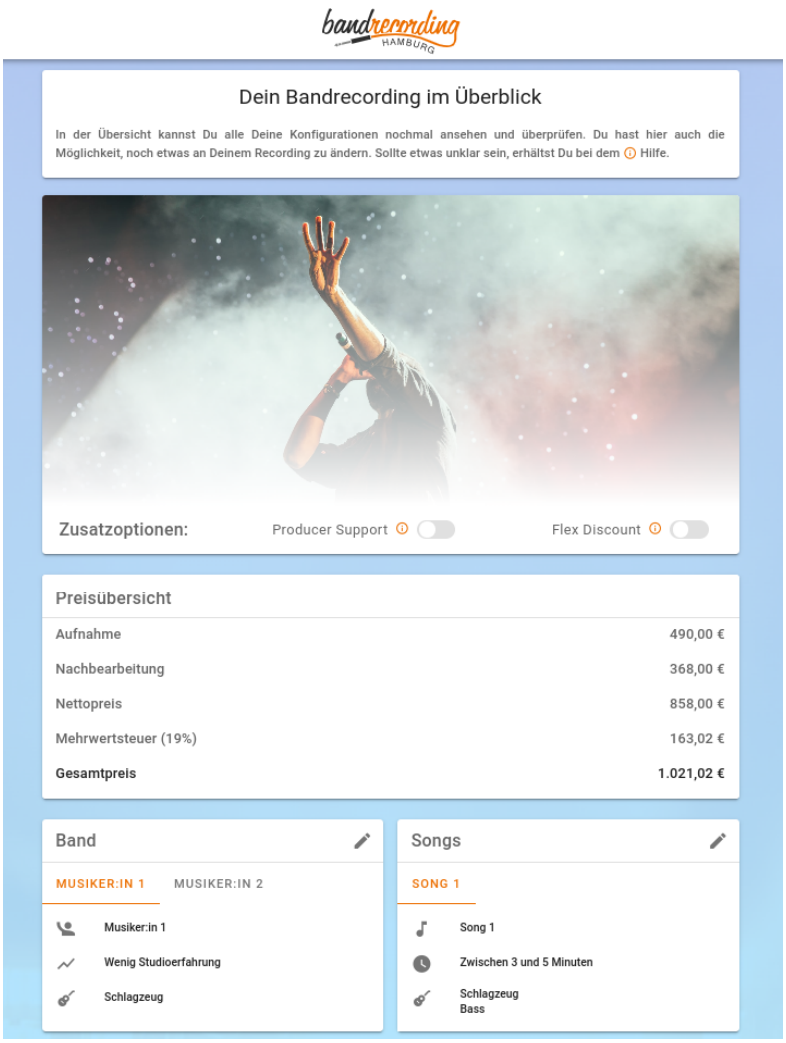

Abbildung 3.6: Zusammenfassung der Konfiguration (Teil 1, eigene Darstellung)

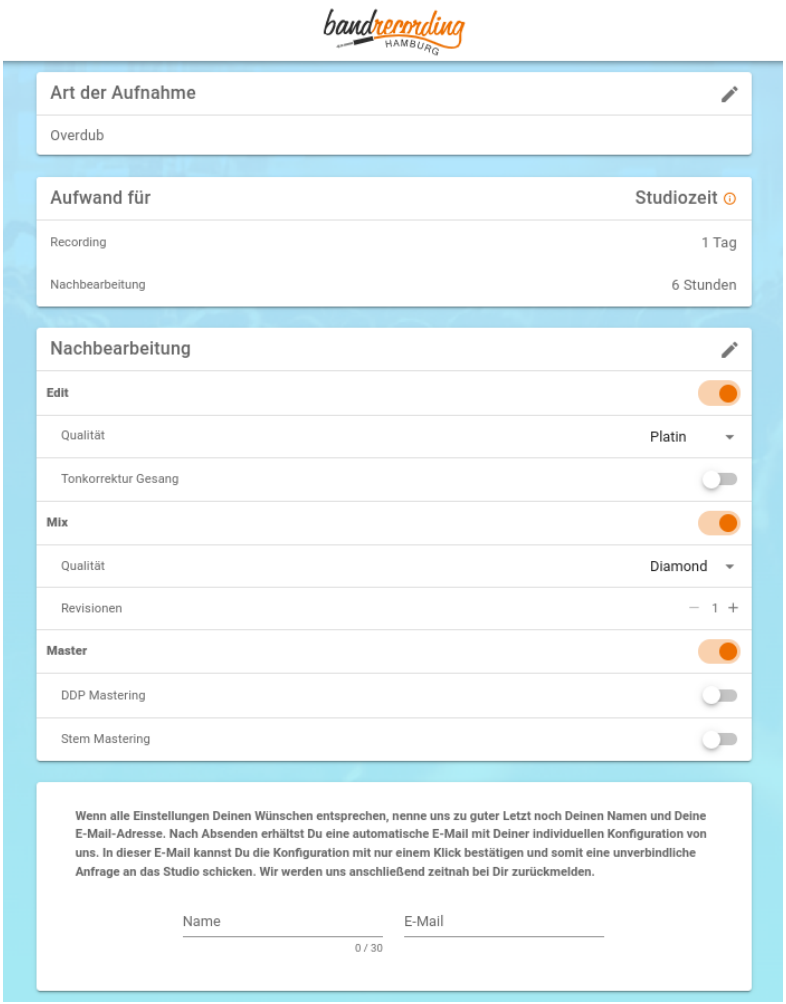

Abbildung 3.7: Zusammenfassung der Konfiguration (Teil 2, eigene Darstellung)

Nach Abschluss des Konfigurators wird eine automatisch generierte E-Mail an die Kund:innen versendet, in der diese sich noch einmal einen Überblick über ihr Angebot verschaffen können und dieses dann über einen Button in der Mail bestätigen können. Nach der Bestätigung wird eine Auftragsbestätigung per E-Mail an das Tonstudio versendet, welche die Details des Angebotes sowie die Kontaktdaten der Kund:in enthält.

3.2 Anforderungen des finalen Produktes

Für das finale Produkt fehlen noch einige funktionale sowie nicht funktionale Anforderungen, die umgesetzt werden sollen. Diese werden im Folgenden kurz beschrieben. Die Anforderungen, die noch umgesetzt werden müssen, lassen sich in drei Bereiche Aufteilen:

1. Ein Login-Bereich für Kund:innen
2. Ein Login-Bereich für Tonstudios
3. Die Erweiterbarkeit des Systems

Diese werden in den folgenden Kapiteln kurz beschrieben, um einen Eindruck davon zu vermitteln, wohin sich BRHH entwickeln soll.

3.2.1 Login-Bereich für Kund:innen

Für die Kund:innen soll ein Login-Bereich entwickelt werden, in dem diese ihre kommende Studioaufnahme verwalten können. Dazu soll es zunächst möglich sein, direkt über die Website Kontakt mit dem Studio aufzunehmen. Diese Kontaktaufnahme soll über einen integrierten Messenger erfolgen und den Kontakt erleichtern, damit immer beiden Parteien direkt klar ist, um welche Aufnahme es sich handelt.
Außerdem soll es möglich sein, das Angebot zur Aufnahme detailliert einzusehen, bei Bedarf zu ändern sowie es final zu bestätigen und über die Website unter Anbindung diverser Zahlungsdienstleister zu bezahlen. Es soll darüber hinaus möglich sein, Details, wie zum Beispiel Tempo, Tonart, etc. zu den einzelnen Songs der Aufnahme zu ergänzen. Durch diese zusätzlichen Parameter ist es dem Studio möglich, die Aufnahme bestmöglich zu planen und umzusetzen.
Die Terminfindung für die Aufnahme, soll durch einen in die Webapplikation integrierten Kalender möglich sein, über den freie Termine des Studios einzusehen sind und ein Termin für die Aufnahme gebucht werden kann. Dafür soll es dem Studio möglichst einfach gemacht werden, einen Kalender einzubinden, in dem offene Termine eingetragen werden können.
Ein Schlüsselaspekt des Konzeptes ist es, die Musiker:innen optimal auf die Aufnahme vorzubereiten. Im Rahmen einer Gamification soll immer ein Fortschrittsbalken im

Login-Bereich einsehbar sein. Dieser Fortschrittsbalken kann dadurch werden, indem kleine Aufgaben, wie beispielsweise für die Gitarrist:in das Aufziehen neuer Gitarrensaiten, erledigt werden. Um die Band zu motivieren, sollen die Fortschrittsbalken der einzelnen Musiker:innen für alle Bandmitglieder:innen einsehbar sein. Dadurch soll eine Art Wettbewerb zwischen den Musiker:innen in der Vorbereitung gefördert werden. Durch, von BRHH bereitgestellte, Anleitungsvideos wird den Musiker:innen eine Hilfestellung bei der Vorbereitung auf die Aufnahme gegeben.
Ein weiteres Feature des Login-Bereiches soll eine Art Dateiverwaltung sein, mithilfe derer Musiker:innen Audiodaten untereinander oder mit dem Studio teilen können, um im Voraus Unklarheiten zu klären.

3.2.2 Login-Bereich für die Studios

Um sicherzustellen, dass die Preise immer aktuell gehalten werden können, ist ein Login-Bereich für die Studios nötig. Über diesen sollen alle Werte, die festlegbar sind, geändert werden können. Des Weiteren bietet der Login-Bereich die Möglichkeit, über den integrierten Messenger, mit den Kunden zu korrespondieren. Das Studio muss außerdem auf geteilte Audio Daten zugreifen können. Um die Verwaltung der Angebote möglichst einfach zu gestalten, soll auch dies im Login-Bereich möglich sein. Weitere Funktionen sind für das Produkt auf Seite der Studios zunächst nicht vorgesehen.

3.2.3 Erweiterbarkeit der Anwendung

Das Team verfolgt das Ziel, die Anwendung in Zukunft weiterzuentwickeln, daher ist es erforderlich, dass die Erweiterbarkeit der Anwendung von Beginn an in der Grundarchitektur berücksichtigt wird. Außerdem muss sichergestellt werden, dass eine Einarbeitung neuer Entwickler:innen unkompliziert erfolgen kann, um auf lange Sicht Zeit zu sparen. Dazu muss bereits beim Entwurf der Architektur auf eine gute Dokumentation Wert gelegt werden. Des Weiteren soll darauf geachtet werden, dass das Produkt weitgehend modular entworfen wird.

3.3 Deployment des Produktes

Die Anwendung läuft aktuell auf einem ‚Shared Server Space' eines Internetanbieters. Dadurch wird die Art der Anwendung stark eingeschränkt und auch die Skalierbarkeit ist nicht gegeben. Um die Erweiterbarkeit und Skalierbarkeit der Anwendung sicherzustellen, könnte es in Zukunft sinnvoll sein, die Anwendung in der Cloud zu betreiben oder einen Linux Server zu verwenden. Dies soll im Rahmen der Arbeit und des MVPs jedoch nicht weiter behandelt werden.

3.4 Übernahme von Teilen des MVPs in das finale Produkt

Der MVP ist bereits mithilfe des Frontend-Frameworks ‚Vue' entwickelt worden. Dies soll auch für die finale Anwendung verwendet werden, damit große Teile des Frontends weiter verwendet werden können. Dadurch, dass der MVP auch visuell bereits sehr weit fortgeschritten ist, lässt sich die Designsprache des MVPs aufgreifen und weiterverwenden.
Im Frontend wird aktuell die gesamte Preisberechnung durchgeführt. Diese soll bestehen bleiben, da sie sehr performant ist und auf dem Endgerät des Nutzers ausgeführt wird. So kann sowohl der Server als auch das Netzwerk entlastet werden. Im finalen Produkt soll diese um eine Preisberechnung im Backend ergänzt werden. Das ist zwingend erforderlich, um clientseitige Manipulationen des Preises zu verhindern.

3.5 Differenz zwischen MVP und finalem Produkt

Im Gegensatz zum finalen Produkt ist bei dem MVP noch keinerlei Backendanbindung vorhanden, was zur Folge hat, dass keine Daten persistiert werden können. Eine Persistenzlösung ist für die in Kapitel 3.2 beschriebenen Funktionen aber unbedingt nötig.
Des Weiteren handelt es sich bei dem Backend aktuell um einen PHP-Server, der eine einzige REST-Anfrage verarbeiten, aus dieser eine E-Mail erstellen und diese dann verschicken kann. Um die Preisberechnung sicher und stabil zu gestalten, ist ein Backend nötig. Außerdem ist ein Backend für die Erweiterbarkeit des Produktes durchaus hilfreich.

4 Methode

Anhand der Entwicklung des MVP für Bandrecording Hamburg soll analysiert werden, was für die Entwicklung eines MVP hilfreich ist und welche Teile verbesserungswürdig sind. Anschließend an die Analyse, werden aus den Ergebnissen eine Methode hergeleitet und diese vorgestellt.

4.1 Analyse des Fallbeispiels Bandrecording Hamburg

Zur Analyse wird zunächst beleuchtet, wie die Anforderungen an den MVP ausgewählt wurden und dies bewertet. Anschließend werden Punkte, die bei der Entwicklung positiv verlaufen sind und solche, die negativ verlaufen sind, aufgeführt. Aus der Analyse werden Schlüsse gezogen, was bei der Entwicklung eines MVPs zu beachten ist und anhand dieser eine Methode vorgeschlagen, die bei der Entwicklung eines MVPs behilflich ist.

4.1.1 Auswahl der Anforderungen

Um zu entscheiden, welche Features im MVP von BRHH umgesetzt werden sollen, wurde mit dem Team besprochen, welche Features für die Grundfunktionalität wichtig sind. Hierbei wurden folgende funktionale Anforderungen ausgewählt:

- Eine Aufnahme soll vollständig konfigurierbar sein.
- Der Preis der Aufnahme soll während der Konfiguration jederzeit einsehbar sein.
- Nach Abschluss der Konfiguration soll automatisch eine Mail erstellt werden, in der das Angebot angefragt werden kann.
- Die einzelnen Einstellungsmöglichkeiten der Aufnahme sollen erklärt werden und für jeden verständlich sein.

Folgende nicht-funktionale Anforderungen wurden ausgewählt:

- Der Konfigurator soll einfach zu bedienen sein.
- Die Anwendung soll als statische Website, ohne komplexes Backend, deployed werden können.
- Die Anwendung soll sowohl auf dem Desktop als auch auf mobilen Endgeräten gleich gut bedienbar sein.

Die Anforderungen wurden ohne Verwenden einer Methode ausgewählt und priorisiert, was im Verlauf der Entwicklungsphase dazu geführt hat, dass die Priorisierung nicht für alle Entwickler:innen klar war. Der MVP von BRHH wurde entwickelt, bevor eine finale Architektur ausgewählt worden ist. Aus diesem Grund wurde der MVP so entworfen, dass er als eigenständiges Frontend für das finale Produkt als Service verwendbar ist.

4.1.2 Positive Aspekte der Entwicklung

Bei der Entwicklung des BRHH MVP wurde sich bei der Implementierung der meisten Anforderungen an das Pareto Prinzip (siehe Kapitel 2.1.3) gehalten, wodurch die Ressourcen effizient für das Entwickeln weiterer Features eingesetzt werden konnte.
Sobald auffiel, dass zu viel Zeit in ein Feature investiert wurde, wurde dies schnellstmöglich abgeschlossen. Dadurch wurden für den MVP unnötige Details ausgelassen und Zeit gespart.

Die Features, die in den MVP aufgenommen werden sollten, wurden vor der Entwicklung ausgewählt. Dazu wurde zunächst die Zielstellung für die Entwicklung des MVP identifiziert und anschließend eine entsprechende Sortierung der Features vorgenommen. Die Zielsetzung für die Entwicklung des MVP war hierbei, zunächst die Überprüfung der Funktionalität des Konfigurators und der Bedienbarkeit für die Anwender:innen. Um diese Ziele zu erreichen, muss der Konfigurator funktionsfähig sein und die Benutzeroberfläche intuitiv bedienbar sein.

Die Entwicklung des MVPs wurde über ein Kanban-Board organisiert. Bei der Kanban-Board-Methode werden die einzelnen Aufgaben der Entwickler:innen in die Kategorien **Backlog**, **in Arbeit** oder **Fertig** eingeteilt und je nach Zustand der Aufgabe in die jeweils nächste Kategorie verschoben. Das Kanban-Board ist dabei für das gesamte Team

einsehbar. Dadurch konnte jede Entwickler:in einzelne kleine Aufgaben erfüllen, die größtenteils unabhängig voneinander waren, wodurch eine gute parallele Arbeitsweise erreicht werden konnte.

Der MVP wurde von Anfang an so geplant, dass dieser als Frontend für das finale Produkt weiter verwendet werden kann. Um dieses Ziel zu erreichen, wurde das Frontend als Microservice entwickelt, welcher über REST-Schnittstellen an ein beliebiges Backend angebunden werden kann. Das verwendete Backend ist sehr rudimentär und erfüllt nur die unbedingt für die Grundfunktionalität notwendigen Aufgaben.

4.1.3 Negative Aspekte der Entwicklung

Der Hauptteil der Entwicklungszeit wurde für das visuelle Design des Frontends aufgewendet. In diesem Teil wurde sich nicht an das Pareto Prinzip gehalten. Der MVP war bereits funktionsfähig, er wurde aber noch nicht veröffentlicht, da das Entwicklerteam nicht mit dem Aussehen der UI zufrieden war. In diesem Punkt ist das Team nicht zielorientiert vorgegangen und hat unnötig viele Ressourcen aufgewendet. Während der Entwicklung hätte mehr Wert auf die Priorisierung der Anforderungen gelegt werden sollen und diese hätte, wenn nötig, angepasst werden sollen. Außerdem wurde die Priorisierung der Anforderungen nicht klar genug an alle Entwickler:innen kommuniziert, was dazu geführt hat, dass Zeit für nicht zielführende Aufgaben aufgewendet worden ist.
Viele Teile des MVPs wurden mit zu wenig Planung implementiert, was zum Teil der Arbeit mit den, für die Entwickler:innen neuen, Frameworks geschuldet ist. Diese vorschnelle Implementierung hat dazu geführt, dass einige Teile mehrmals verändert werden mussten, nachdem den Entwickelnden aufgefallen ist, dass etwas fehlt.
Am Anfang der Entwicklung des MVP wurde viel Zeit dafür aufgewendet, ein für den MVP passendes Framework zu finden. Zunächst wurde Vue3[1] ausgewählt und die Entwicklung mit diesem begonnen. Nach einer Weile hat sich jedoch herausgestellt, dass das Framework Vuetify[2] verwendet werden soll, um die UI ansprechender zu gestalten. Vuetify war jedoch zum Zeitpunkt der Entwicklung nur mit Vue2[3] verwendbar. Dies hat dazu geführt, dass viele Teile des MVPs von Vue3 auf Vue2 portiert werden mussten.

[1] Siehe https://vuejs.org/
[2] Siehe https://vuetifyjs.com/en/
[3] Siehe https://v2.vuejs.org/

4.1.4 Ergebnis der Analyse des MVPs

Aus der Analyse der Entwicklung des MVPs von BRHH lassen sich Schlüsse ziehen, aus denen die Schritte der Methode hergeleitet werden. Von den Punkten, die positiv liefen, lässt sich vor allem die gute Auswahl von Features sowie die Organisation über das Kanban-Board herausstellen. Ein Punkt, der negativ verlaufen ist, aber zu demselben Bereich zählt, ist, dass die intransparente Priorisierung der Features. Daraus wird der Schluss gezogen, dass zur Organisation der Entwicklung eines MVP zunächst die Anforderungen notiert und klar priorisiert werden müssen.
Ein weiterer Punkt, aus dem eine Lehre gezogen werden kann, ist die unnötig aufgewendete Zeit für das visuelle Design des Frontends. Um dies zu vermeiden, wäre es wichtig gewesen, das Ziel des MVPs vor der Entwicklung klar zu definieren. Durch die Definition eines klaren Ziels des MVPs ist es einfacher, die Anforderungen auszuwählen und zu priorisieren.
Die Architektur des MVPs wurde so gewählt, dass dieser sich mit wenig Arbeit in das finale Produkt integrieren lässt. Dieser Punkt sollte in der Methode unbedingt beachtet werden, da dadurch die Zeit vom MVP zur nächsten Iteration des Produktes verkürzt werden kann.

4.2 Herleitung der Methode

Aus dem Ergebnis der Analyse stellen sich vor allem folgende Punkte als Ziele der Methode heraus:

- Klare Definition des Ziels und Umfangs des MVP
- Priorisierung der Anforderungen
- Erleichterung der Wahl eines Frameworks und einer Architektur
- Wiederverwendbarkeit des MVP in der Architektur des finalen Produktes
- Transparente Organisation der Entwicklung

Um diese Punkte abzudecken, werden im folgenden Unterkapitel die Schritte der Methode aufgeführt sowie diese näher erklärt.

4.2.1 Schritte der Methode

1. User Stories herausarbeiten
2. Funktionale und nicht-funktionale Anforderungen aus User Stories herausarbeiten
3. Erarbeiten der Zielsetzung des MVPs
4. Methodische Priorisierung der Anforderungen
5. Auswahl der Anforderungen, die im MVP umgesetzt werden sollen
6. Auswahl der Art des MVPs
7. Auswahl der Architektur des MVP
8. Auswahl der zu verwendenden Technologien (Frameworks etc.)

Schritte eins bis drei sowie Schritt fünf decken die klare Definition von Ziel und Umfang des MVPs ab. Mit dem vierten Schritt wird die Priorisierung der Anforderungen abgedeckt. Die Schritte sechs bis acht dienen zusammen zur Auswahl der Frameworks und der Architektur und sollen dafür sorgen, dass der MVP nach Möglichkeit weiter verwendet werden kann. Um die transparente Organisation zu gewährleisten, müssen alle Schritte für alle Entwickler:innen zu jeder Zeit einsehbar dokumentiert werden.

User Stories herausarbeiten

Um einfach und übersichtlich Features zu finden, bietet es sich an die Anwendung in User Stories (siehe 2.1.7) aufzuteilen. Dabei geht es in diesem Schritt noch nicht um die User Stories, die im MVP umgesetzt werden sollen, sondern um die User Stories des Endprodukts. Dieser Schritt ist wichtig, damit im MVP keine Aspekte des finalen Produktes übersehen werden.

Funktionale und nicht-funktionale Anforderungen aus User Stories herausarbeiten

Aus den User Stories müssen nun die funktionalen sowie nicht-funktionalen Anforderungen herausgearbeitet werden. Dazu werden die User Stories durchgegangen und alle in ihnen beinhalteten Anforderungen notiert. Als Beispiel ist eine funktionale Anforderung, die aus der User Story: ‚Als Musiker:in möchte ich während der Konfiguration den Preis ständig im Blick haben können' folgt eine sich stetig aktualisierende Ansicht des Preises der Aufnahme.

Erarbeiten der Zielsetzung des MVPs

Damit die Anforderungen im nächsten Schritt priorisiert werden können, muss zunächst klargestellt werden, was das Ziel des MVPs ist. Einige mögliche Ziele eines MVPs sind:

- Test eines Konzeptes
- Test des Designs der Anwendung
- Test des Business-Modells des Unternehmens
- Test der Skalierbarkeit der Anwendung
- Test der Usability der Anwendung

Um das Ziel des MVPs auszuwählen, muss sich das Team darüber im Klaren sein, wie die Ergebnisse des Feedbacks in dem finalen Produkt verwendet werden sollen.

Priorisierung der Anforderungen

Nachdem das Ziel des MVPs bestimmt worden ist, lassen sich nun die Anforderungen priorisieren. Dazu bietet es sich an, eine der in Kapitel 2.1.5 beschriebenen Methoden zu verwenden. Welche Methode verwendet wird, hängt dabei von den persönlichen Vorlieben der Entwickler:innen und der Art des MVPs ab, weshalb an dieser Stelle keine spezifische Methode empfohlen wird. Generell lassen sich mit allen vorgestellten Methoden die Anforderungen genügend priorisieren.

Auswahl der Anforderungen, die im MVP umgesetzt werden sollen

Nach der Priorisierung muss nun entschieden werden, welche Anforderungen im MVP umgesetzt werden sollten. Hierbei sollte neben der Priorisierung vor allem auf folgende Dinge geachtet werden:

- Wie viel Zeit nimmt das Umsetzen einzelner Anforderungen in Anspruch und lohnt es sich diese bereits im MVP umzusetzen?
- Lassen sich Anforderungen, die umgesetzt werden, im Endprodukt weiter verwenden?
- Sind die Features wichtig, um das Ziel des MVPs zu erreichen?
- Was macht für den MVP nicht viel Mehrarbeit, hilft aber für das Endprodukt?

Es sollte unbedingt darauf geachtet werden, dass nicht zu viele Anforderungen ausgewählt werden, um die Entwicklungszeit nicht unnötig in die Länge zu ziehen. Wie viele Anforderungen ausgewählt werden sollten, lässt sich an dieser Stelle jedoch nicht pauschal sagen, da sich Anwendungen sehr stark voneinander unterscheiden können.

Auswahl der Art des MVPs

Wurden nun alle Anforderungen ausgewählt, die umgesetzt werden sollen, muss ausgewählt werden, welche Art eines MVP (siehe Kapitel 2.1.3) verwendet werden soll. Die Auswahl hängt dabei von der Zielsetzung sowie den ausgewählten Anforderungen ab. Wichtig ist, dass alle ausgewählten Anforderungen umgesetzt werden können. Wurde beispielsweise eine Anforderung ausgewählt, für die eine Datenbank verwendet werden muss, bietet es sich nicht an, ein Frontend-MVP zu wählen, der wenig bis kein Backend aufweist.
In der Tabelle 4.1 werden einige Arten von MVPs für Webanwendungen zusammen mit Auswahlkriterien für diese aufgeführt. Dabei werden die jeweiligen Arten aufgelistet und ihnen jeweils ein Wert von null bis fünf zu dem jeweiligen Ziel zugeteilt. Null repräsentiert dabei, dass sich die Art gar nicht eignet, während fünf bedeutet, dass sich die Art sehr gut eignet. Wie sich die Werte ergeben wird im Folgenden kurz geschildert.
Im Frontend MVP kann die UI vollständig realisiert werden. Dadurch kann diese in dem Stadium getestet werden, in dem sie auch im finalen Produkt sein wird. Ein Algorithmus lässt sich zum Teil testen, indem er im Frontend implementiert wird. Komplexere oder

zeitintensive Algorithmen lassen sich allerdings oft nicht im Rahmen eines Frontend-MVPs testen, da sich zum Umsetzen dieser die verfügbaren Sprachen und Werkzeuge nicht gut eignen. Frontend-MVPs lassen sich häufig schneller entwickeln als Applikationen mit vollem Front- und Backend, benötigen dennoch einige Zeit für den Feinschliff der UI. Wird das Produkt so realisiert, dass das Frontend als eigener Service aufgebaut ist, lassen sich Frontend-MVPs mit wenig Arbeit an ein Backend anbinden, was für eine sehr gute Weiterverwendbarkeit sorgt.
Bei einem ‚Wizard of Oz' MVP wird die Logik des Frontends manuell ausgeführt. Dadurch lässt sich ein Großteil der UI testen. Teile, die zwingend eine Automatisierung benötigen, da sie sonst nicht schnell genug ausgeführt werden können, lassen sich aber nicht testen. Einfache Algorithmen, die manuell durchführbar sind, können getestet werden. Alle anderen lassen sich jedoch nicht testen. Da kaum Logik entwickelt werden muss, lässt sich diese Art des MVPs schnell entwickeln. Wird der MVP so umgesetzt, dass das manuell bediente Frontend mit Logik versehen werden kann, können Teile des MVPs im Endprodukt verwendet werden.
Wird als MVP ein Mockup erstellt, können nur visuelle Teile des Frontends ohne jegliche Logik bewertet werden. Ein Algorithmus kann im Rahmen eines Mockups nicht getestet werden. Die Entwicklung eines Mockups kann innerhalb weniger Stunden erfolgen, es kann aber nicht ins Endprodukt eingebunden werden.
Auf einer Landing-Page lassen sich Elemente der UI testen, komplexere UI-Elemente sind jedoch schwer zu testen, vor allem wenn sie zur Navigation zwischen Teilen der Anwendung vorgesehen sind. Zum Test eines Algorithmus ist eine Landing-Page nicht geeignet, aber auch nicht vorgesehen. Dadurch, dass keine Navigationslogik und generell wenig Bedienungslogik umgesetzt werden muss, lässt sich eine Landing-Page schnell entwickeln. Da die meisten Webanwendungen eine Landing-Page benötigen, lässt sich der MVP in vielen Fällen vollständig weiterverwenden.
Der Fokus eines Backend-MVPs liegt nicht auf dem Frontend, weshalb nur einfachste UI-Elemente getestet werden. Durch das komplexe Backend lassen sich Algorithmen vollumfänglich testen. Je nach Ausmaß der im Backend umzusetzenden Logik kann die Entwicklung des Backend-MVPs viel Zeit in Anspruch nehmen. Teile des Backends lassen sich meist im Endprodukt weiterverwenden, das Frontend muss aber überarbeitet werden.

Die Tabelle stellt dabei nur eine Hilfestellung dar, mit der ein schneller Überblick über einige Arten von MVPs möglich ist. An dieser Stelle ist es wichtig zu erwähnen, dass die Tabelle nicht als vollständig und gänzlich objektiv gesehen werden sollte, da es viele

Art des MVPs	**Test der UI**	**Test eines Algorithmus**	**Schnelle Entwicklung**	**Weiterverwendung im Produkt**
Frontend MVP	5	2	3	4
Wizard of Oz	4	1	4	2
Mockup	2	0	5	0
Landing-Page	2	0	4	5
Backend MVP	1	5	2	3

Tabelle 4.1: Auswahl der Art des MVP

weitere Möglichkeiten gibt, einen MVP umzusetzen und auch weitere Kriterien zur Auswahl der Art vorstellbar sind. Eine vollständige Tabelle würde aber den Rahmen dieser Arbeit überschreiten und ist, in der sich ständig im Wandel befindenden Umgebung der Softwareentwicklung, auch eher schwer zu verwirklichen.

Auswahl der Architektur des MVP

Im nächsten Schritt muss eine Architektur für den MVP ausgewählt werden. Dazu bietet es sich an, folgende Fragen zu beantworten:

- Wie viel Zeit kostet die Ausarbeitung der Architektur?
- Muss der MVP skalierbar sein?
- Wie komplex ist die Architektur?
- Welche Architektur bietet sich für den MVP an und kann man diese dann für das Endprodukt verwenden?

Anhand der Ergebnisse dieser Fragen und den Anforderungen an den MVP kann nun in Orientierung an einem der in Kapitel 2.3 genannten Architekturstile eine Architektur für den MVP entworfen werden. Dabei ist es sicherlich einfacher, vor allem Frontend-MVPs, in Microservice basierte Architekturen zu integrieren als in Monolithen. Es muss darauf geachtet werden, ob mit der ausgewählten Architektur alle Anforderungen an den MVP abgedeckt werden können.
Außerdem schließen sich die Architekturstile nicht gegenseitig aus. Einzelne Microservices könnten zum Beispiel mit einer MVC-Architektur umgesetzt werden.
In der Tabelle 4.2 wird dargestellt, wie sich die Architekturen bezüglich der Punkte Entwicklungszeit, Skalierbarkeit und Komplexität zur Entwicklung eines MVP eignen. In der

Spalte Entwicklungszeit geht es dabei um die Entwicklungszeit, die zur Umsetzung der architekturspezifischen Aspekte aufgewendet werden muss. Damit ist zum Beispiel die Kommunikation zwischen Microservices gemeint, die in einer monolithischen Architektur wegfallen. Aspekte, die unabhängig von der Architektur umgesetzt werden müssen, werden hier nicht betrachtet. Es kann also sein, dass auch die Entwicklung eines Monolithen viel Zeit in Anspruch nimmt, obwohl diese in der Tabelle mit **kurz** angegeben ist.
Die Spalte Skalierbarkeit gibt an, wie gut sich einzelne Architekturstile für skalierbare Anwendungen eignen.
In der Spalte Komplexität wird angegeben, wie komplex das Zusammenspiel der Architekturbausteine ist. Anhand der Komplexität lässt sich außerdem ablesen, wie schwer es ist, die Architektur zu verstehen sowie diese umzusetzen.
Die Tabelle stellt eine Hilfestellung dar, die die Orientierung vereinfachen soll, hat aber nicht den Anspruch darauf, eine vollständige Architekturübersicht zu sein. In vielen Punkten können außerdem schwer klare Aussagen getroffen werden, da viele Variablen eine Rolle spielen.
Ein Monolith lässt sich schnell umsetzen, da kaum Architekturplanung nötig ist, was außerdem zu einer geringen Komplexität führt. Dadurch, dass die ganze Anwendung in einem Baustein umgesetzt wird, muss die gesamte Anwendung skaliert werden, was zu einer schlechten Skalierbarkeit führt.
Die Entwicklungszeit einer Client-Server-Architektur ist ein wenig länger als die eines Monolithen, da zwei separate Teile der Anwendung umgesetzt werden müssen. Lässt sich der Server skalieren, skaliert diese Architektur sehr gut, da viele Clients auf einen Server zugreifen können. Die Komplexität erhöht sich im Vergleich zum Monolithen ein wenig, da die Kommunikation zwischen Client und Server umgesetzt werden muss.
Da bei einer P2P-Anwendung die Peers die Aufgaben von Clients und Servern übernehmen, muss nur eine Anwendung entwickelt werden, was die Entwicklungszeit verringern kann. Die Komplexität einer P2P-Anwendung ist durch die Kommunikation sehr ähnlich zu einer Client-Server-Anwendung.
Für eine Schichten-Architektur muss mehr Entwicklungszeit als für einen Monolithen gerechnet werden, da die Schichten untereinander kommunizieren müssen. Wie gut eine Schichten-Architektur skaliert hängt davon ab, wie gut die einzelnen Schichten skalieren. Sie hat jedoch den Vorteil, Aspekte, die skalieren müssen, in einzelne Schichten auslagern zu können. Bei der Schichten-Architektur handelt es sich um einen Architekturstil, der je nachdem, wie viele Schichten die Architektur aufweist und wie viel die Schichten untereinander kommunizieren, deutlich an Komplexität zugewinnen kann.
Die benötigte Zeit, um eine einzelne REST-Schnittstelle umzusetzen, ist sehr kurz, es

Architekturstil	**Entwickungszeit**	**Skalierbarket**	**Komplexität**
Monolith	kurz	schwer	gering
Client/ Server	mittelmäßig	über Server	mittelmäßig
P2P	oft kürzer als Client/ Server	sehr gut	mittelmäßig
Schichten-Architektur	mittelmäßig	von Schichten abhängig	von Kommunikation zwischen Schichten abhängig
REST	pro Schnittstelle kurz	sehr gut	pro Schnittstelle gering
SOA	meist kürzer als Microservices aber länger als Monolith	wesentlich besser als Monolith	höher als Monolith
Microservices	lang	sehr gut	sehr hoch
Makroservices	mittelmäßig	gut	hoch

Tabelle 4.2: Auswahl der Architektur des MVP

sind aber eventuell viele Schnittstellen nötig. Dieser Aspekt sollte bei der Auswahl der Architektur unbedingt beachtet werden. REST-Architekturen skalieren sehr gut, da es durch die Zustandslosigkeit einfach ist, diese als verteiltes System zu realisieren. Die Komplexität pro Schnittstelle ist sehr gering.
Um eine SOA umzusetzen, wird mehr Zeit als zur Umsetzung eines Monolithen benötigt, da die Anwendung in Module aufgeilt werden muss. Eine SOA kann, anders als Microservices, einfach auf gemeinsame Ressourcen zugreifen, was ein Punkt ist, der vergleichend die Entwicklungszeit verkürzen kann. Durch die Trennung in Services lassen sich gezielt die Elemente, die skaliert werden müssen skalieren, was zu einer guten Skalierbarkeit führt. Ebendiese Trennung hat jedoch eine hohe Komplexität zur Folge.
Es ist herausfordernd Microservices zu entwickeln, da die Kommunikation zwischen den zahlreichen kleinen Bausteinen zu einer sehr hohen Komplexität und damit zu einer langen Entwicklungszeit führt. Die Skalierbarkeit von Microservices ist sehr gut, da sich jeder Service theoretisch auf einer eigenen Plattform laufen kann, wodurch sich Services, die viel Last tragen, gut skalieren lassen.
Da die Services bei einer Makroservice-Architektur grobgranularer sind als die einer Microservice-Architektur, wird die Komplexität und die Entwicklungszeit ein wenig verringert. Die Skalierbarkeit ist durch die Trennung der Services gut.

Auswahl der zu verwendenden Technologien

Als letzten Schritt vor der Umsetzung des MVP sollte nun, anhand der vorher gewonnen Informationen, eine Technologie-Suite zusammengestellt werden. Dazu werden für die Entwicklung notwendige Frameworks ausgewählt und festgelegt. Dadurch werden alle Entwickler:innen auf den gleichen Stand gebracht, um eine aufwendige Integration verschiedener Frameworks zu vermeiden. Bei der Auswahl der Technologien sollte auf folgende Dinge geachtet werden:

- Wie groß ist die Community hinter der Technologie?
- Haben die Entwickler:innen Erfahrung mit den Technologien?
- Wird die Technologie noch weiterentwickelt/ unterstützt?
- Sind für den MVP Features aus der neuesten Version eines Frameworks nötig oder kann auch eine stabile Version des Frameworks verwendet werden?
- Handelt es sich bei dem Framework um eine sehr neue Technologie?
- Lässt sich die Technologie im finalen Produkt verwenden?
- Wird durch das Einsetzen des Frameworks im MVP voraussichtlich Zeit gespart?
- Wie hoch sind die Kosten, um die Technologie zu verwenden?

Ist die Community, die hinter einer Technologie oder einem Framework steht sehr groß, ist die Wahrscheinlichkeit hoch, dass während der Entwicklungsphase auftretende Probleme, bereits von Mitglieder:innen der Community gelöst worden sind. Außerdem ist bei einer größeren Community mit schnellerer Hilfe zu rechnen.
Wenn die Entwickler:innen bereits Erfahrung mit einer Technologie haben, kann Zeit, für die Einarbeitung in diese gespart werden. Diesem Punkt sollte ein besonderes Gewicht zugesprochen werden, wenn die Zeit zur Entwicklung knapp bemessen ist.
Technologien, die nicht mehr gepflegt werden, sollten nur verwendet werden, wenn diese absolut notwendig und nicht zu ersetzen sind. Wird eine Technologie nicht mehr weiterentwickelt, ist es meist schwer, Unterstützung bei Problemen zu erhalten.
Viele Technologien werden schnell weiterentwickelt und es ist verlockend, immer die aktuellste Version zu verwenden. Dabei sollte aber unbedingt auch darauf geachtet werden, ob es sich bei der verwendeten Version um eine stabile Version handelt. Frameworks, die sich noch in der Beta-Phase befinden, werden möglicherweise von anderen Technologien,

die verwendet werden sollen, nicht unterstützt. Aus diesem Grund sollten vor Start der Entwicklung alle zu verwendenden Technologien sowie ihre Versionen festgehalten werden. Außerdem sollten diese auf eventuelle Kompatibilitätsprobleme überprüft werden.
Handelt es sich bei einem Framework um eine sehr neue Technologie, sollte abgewogen werden, ob es sich lohnt, diese zu verwenden. Bei neuen Technologien sind die Communities häufig noch sehr klein.
In Technologien, die im MVP bereits verwendet wurden müssen sich Entwickelnde im Laufe der weiteren Entwicklung nicht neu einarbeiten, weshalb es sich lohnen kann die Technologien des MVP mit Weitsicht auf das finale Produkt auszuwählen. Dadurch kann die Zeit zwischen dem MVP und der nächsten Iteration des Produktes verringern.
Dabei sollte aber auch darauf geachtet werden, ob durch die Verwendung des Frameworks bereits in der MVP-Phase Zeit gespart wird. Wird im Gegenteil voraussichtlich mehr Zeit aufgewendet, um sich in das Framework einzuarbeiten, die Arbeitsumgebung aufzusetzen und den MVP mit dem Framework umzusetzen, als ohne Verwendung des Framework bietet es sich unter Umständen an auf das Framework zu verzichten. Da sich ein Unternehmen bei der Entwicklung des MVPs meist in der Anfangsphase befindet, sollten Technologien, die mit hohen Kosten verbunden sind, mit Vorsicht verwendet werden. Hat ein Unternehmen schon vor Auswertung des MVP finanzielle Probleme, war dieser nicht erfolgreich. Es bietet sich also an Open Source Technologien zu verwenden, die keine Kosten erzeugen.

5 Anwendung der Methode auf das zweite Fallbeispiel (Blog)

Um die Methode zu überprüfen, soll sie für die Entwicklung des MVPs eines zweiten Fallbeispiels verwendet werden. Das zweite Fallbeispiel ist ebenfalls eine Webanwendung, welche im Folgenden kurz beschrieben wird. Es handelt sich um einen Blog, auf dem Inhalte von mehreren Autor:innen veröffentlicht werden können sollen. Der Blog ist vor allem dafür gedacht, Reiseerinnerungen der Autor:innen an einer zentralen Stelle festzuhalten, um die Reisen im Nachhinein einfacher nachvollziehen zu können. Außerdem sollen über den Blog Fotos und Videos einfach mit Mitreisenden und Menschen, die auf den Reisen getroffen wurden, geteilt werden können.
In diesem Kapitel werden die in Kapitel 4.2.1 beschriebenen Schritte abgearbeitet und beschrieben.

5.1 User Stories und Anforderungen herausarbeiten

Im Folgenden werden User Stories und zugehörige Features, die für das finale Produkt gedacht sind, aufgeführt. Damit werden die ersten beiden Schritte der Methode in einem Schritt dargestellt, um die Anwendung der Methode übersichtlicher zu gestalten. Bei den in Tabelle 5.1 aufgeführten Anforderungen handelt es sich außer bei den beiden letzten, als nicht-funktionale Anforderungen markierte, um funktionale Anforderungen.

User Story	**Anforderung**
Als Autor:in möchte ich Artikel schreiben können.	Schreiben von Artikeln
Als Autor:in möchte ich Artikel veröffentlichen können.	Einfaches veröffentlichen von Artikeln
Als Benutzer:in möchte ich Artikel lesen können.	Anzeigen von Artikeln
Als Benutzer:in möchte ich zwischen Artikeln navigieren können.	Navigation zwischen Artikeln
Als Benutzer:in möchte ich Artikel durchsuchen können, damit ich spezielle Artikel finden kann.	Durchsuchen von Artikeln
Als Autor:in möchte ich verschiedene Ansichten für Artikel auswählen können, damit ich die für meinen Inhalt optimale Ansicht verwenden kann.	Unterschiedliche Ansicht von Artikeln, je nach Inhalt
Als Benutzer:in oder Autor:in möchte ich mich auf der Website anmelden können, um Artikel zu veröffentlichen oder zu kommentieren.	Login-Bereiche für Benutzer:innen und Autor:innen
Als Benutzer:in möchte ich alle Artikel einer Autor:in angezeigt bekommen, damit ich mir einen Überblick verschaffen kann.	Gruppieren von Artikeln nach Autor:innen
Als Autor:in möchte ich die Möglichkeit haben, mich auf der Website kurz vorzustellen, damit Benutzer:innen wissen, wer ich bin.	Kurzbeschreibung der Autor:innen
Als Autor:in möchte ich alle meine geschriebenen Artikel auf der Website bearbeiten können.	Content-Management-System für Autor:innen
Als Benutzer:in möchte ich sowohl auf Desktopcomputern, als auch auf mobilen Endgeräten auf die Website zugreifen können.	Mobil- und Desktopansicht (nicht funktional)
Als Benutzer:in möchte ich, dass die Website schnell reagiert, egal wie viele Nutzer gleichzeitig auf diese zugreifen.	Skalierbarkeit des Blogs (nicht funktional)

Tabelle 5.1: User Stories mit zugeordneten Anforderungen

5.2 Zielsetzung des MVP

Das Ziel dieses MVPs ist es, die Funktion des Blogs vor allem aus Sicht der Autor:innen zu testen. Außerdem ist es ein Ziel, zu testen, ob der Blog tatsächlich dabei hilft, die Erinnerungen einfacher zu sammeln oder ob er so viel mehr Arbeit macht, dass er nicht verwendet wird. Sollte Letzteres eintreffen, ist es außerdem Ziel, in Erfahrung zu bringen was geändert werden müsste, damit der Blog verwendet wird. Zu den Zielen gehört nicht das Erreichen eines breiten Publikums.

5.3 Priorisierung der Anforderungen

Die Anforderungen werden mithilfe der MoSCoW Methode (siehe Kapitel 2.1.5) priorisiert. Diese Methode wurde gewählt, da die Unterteilung sich, aus persönlicher Sicht, für die Entwicklung eines MVP anbietet. Es ist klar, welche Anforderungen unbedingt umgesetzt werden sollten, welche optional sind sowie welche Anforderungen nicht umgesetzt werden sollten.

Must-Have Anforderungen

- Anzeigen von Artikeln
- Schreiben von Artikeln
- Navigieren zwischen Artikeln

Damit der Blog seine Grundfunktionalität erhält, ist es vor allem nötig, dass Artikel geschrieben und gelesen werden können. Außerdem müssen Benutzer:innen Artikel auswählen können, die sie lesen wollen.

Should-Have Anforderungen

- Durchsuchen von Artikeln
- Einfaches Veröffentlichen von Artikeln
- Unterschiedliche Ansicht von Artikeln je nach Inhalt

Um einfacher Artikel zu finden, ist eine Suchfunktion sehr hilfreich und verbessert die Benutzbarkeit des Blogs. Ein besonders einfaches Veröffentlichen von Artikeln ist nicht für die Grundfunktionalität des Blogs notwendig, erspart aber sowohl den Entwickler:innen sowie den Autor:innen Arbeit. Eine optimale Darstellung der Inhalte wird, je nachdem was für Inhalte in den Artikeln sind, durch unterschiedliche Layouts realisiert.

Could-Have Anforderungen

- Gruppieren der Artikel nach Autor:innen
- Kurzbeschreibung der Autor:innen
- Mobil- und Desktopansicht

Nicht unbedingt notwendig, aber hilfreich, um Artikel zu durchsuchen, ist das Gruppieren der Artikel nach Autor:innen. So kann sich schnell ein Überblick darüber verschafft werden, welche Artikel von welchen Autor:innen stammen. Damit diesen Autor:innen auch ‚ein Gesicht' zugeordnet werden kann, ist es hilfreich, zu jedem Autor eine Kurzbeschreibung mit einem Bild bereitzustellen. Separate Ansichten für mobile Endgeräte und Desktopcomputer führen dazu, dass die Benutzbarkeit des Blogs sich, vor allem unterwegs, verbessert.

Won't-Have Anforderungen

- Content-Management-System für die Autor:innen
- Login-Bereiche für Benutzer:innen und Autor:innen
- Kommentieren von Artikeln
- Skalierbarkeit des Blogs

Ein Backend mit Content-Management-System (im Folgenden als CMS bezeichnet) überschreitet den Umfang des MVPs deutlich. Auch für einen Login-Bereich ist ein Backend und viel damit verbundene Arbeit nötig, wodurch sich die Entwicklungszeit des MVPs deutlich verlängern würde. Eine Kommentarfunktion für Artikel ist für den MVP nicht notwendig, da sich die Anzahl der User voraussichtlich auf einige wenige beschränken wird. Die Skalierbarkeit des Blogs ist für den MVP nicht von Bedeutung, da nicht mit

vielen Benutzer:innen gerechnet wird und ein Ausfall des Servers nicht von großer Bedeutung ist.

5.4 Auswahl der Anforderungen, die im MVP umgesetzt werden

Um diese Ziele des MVP zu erreichen, müssen nun aus den vorher priorisierten Anforderungen solche ausgewählt werden, die im MVP umgesetzt werden sollen.
Aus der **Must-Have** Kategorie werden alle Anforderungen ausgewählt, da ohne diese Anforderungen das Produkt nicht die Basisfunktionalität gewährleisten kann.

Aus den **Should-Have** Anforderungen wird das Durchsuchen von Artikeln sowie die unterschiedliche Ansicht entsprechend dem Artikelinhalt implementiert, da dies voraussichtlich keine große Mehrarbeit mit sich bringt, aber die Benutzbarkeit deutlich verbessert. Für ein besonders einfaches Veröffentlichen von Artikeln ist ein CMS und damit auch ein Backend notwendig. Um die Arbeit am Backend zu reduzieren, wird im MVP ein Git basiertes CMS verwendet, für das minimales Verständnis von Markdown und Git notwendig ist. Dadurch ist es für die Autor:innen einfacher, Artikel zu veröffentlichen als, wenn diese in HTML im Quelltext eingepflegt werden müssten. Es ist aber immer noch mit mehr Arbeit verbunden, als es im finalen Produkt der Fall sein sollte.

Aus der **Could-Have** Kategorie werden sowohl die Kurzbeschreibung der Autor:innen als auch das Gruppieren der Artikel umgesetzt, da diese mit wenig Aufwand implementiert werden können. Die Mobil-Ansicht des Blogs wird im MVP noch nicht implementiert, da diese für die erste Zielsetzung nicht unbedingt nötig ist, aber viel Arbeit beim Anpassen der Ansichten mit sich bringt.

Aus der **Won't-Have** Kategorie werden keine Anforderungen im MVP implementiert, da diese nicht für die Ziele des MVPs nötig sind. Sie sollen aber in einer der nächsten Iterationen des Produktes, sofern der MVP erfolgreich ist, implementiert werden.

Zusammenfassend wurden folgende Anforderungen ausgewählt:

- Anzeigen von Artikeln
- Schreiben von Artikeln
- Navigieren zwischen Artikeln

- Durchsuchen von Artikeln
- Unterschiedliche Ansicht von Artikeln je nach Inhalt
- Gruppieren der Artikel nach Autor:innen
- Kurzbeschreibung der Autor:innen

5.5 Auswahl der Art des MVPs

Der Blog soll als Frontend-MVP umgesetzt werden, da dieser als statische Website veröffentlicht werden kann und dadurch wenig Zeit bei der Einrichtung eines Webservers verloren geht. Außerdem steht mit Beachtung der Zielsetzung des MVPs die Bedienung des Blogs im Vordergrund. Das einfache Veröffentlichen von Artikeln, wofür ein größeres Backend nötig wäre, ist erst im nächsten Schritt wichtig, wenn mehr Artikel veröffentlicht werden sollen.
Im finalen Produkt kann der MVP nahtlos als Frontend übernommen und weitergepflegt werden. Es müssen keine Algorithmen getestet werden und da kein großer Zeitdruck besteht, ist eine kurze Entwicklungszeit nicht die erste Priorität.

5.6 Auswahl der Architektur des MVP

Der MVP des Blogs soll in der nächsten Iteration des Blogs verwendet werden können. Deshalb bietet es sich an, das Frontend als Microservice umzusetzen und über eine REST-Schnittstelle an ein Backend anzubinden. Die Kommunikation mit dem Webserver wird nach dem Client-Server-Architekturstil umgesetzt. Da es sich bei dem Blog um eine Browseranwendung handelt, wird das Frontend als Thin Client umgesetzt. In der Phase des MVPs wird es jedoch kein dediziertes Backend geben, sondern nur einen Webserver, der die statische Website zur Verfügung stellt, weshalb in dieser Entwicklungsphase keine komplexen REST-Schnittstellen notwendig sind.
Für das Endprodukt soll eine Microservice-Architektur verwendet werden, bei der das Frontend, das Backend, die Datenbank sowie mögliche Erweiterungen als eigene Services realisiert werden sollen. Das Frontend wird dabei nicht in einzelne Services unterteilt,

sondern stellt einen der Microservices dar. Es bietet sich eine horizontale Microservice-Architektur (siehe Kapitel 2.3.9) an, da das Frontend zu einem großen Teil im MVP fertiggestellt wird. Der Frontend-Service soll nach dem MVVM-Muster (siehe Kapitel 2.3.3) umgesetzt werden.

5.7 Auswahl der zu verwendenden Technologien

Für den MVP des Blogs bietet es sich an mit Vue zu arbeiten, da bereits viel Erfahrung mit dem Framework gesammelt wurde. Bei Vue handelt es sich um ein Javascript Framework mithilfe dessen Frontends einfacher und schneller entwickelt werden können, als wenn nur HTML, Javascript und CSS verwendet wird. Vue implementiert das MVVM-Muster (siehe Kapitel 2.3.3). Vue Anwendungen lassen sich zu HTML, Javascript und CSS Dateien kompilieren, um diese auf einem Webserver zu deployen [8]. Vue wurde außerdem ausgewählt, da es sich um ein Framework handelt, welches seit 2014 entwickelt wird [8] und eine dementsprechend große Community aufweist. Näheres zu dem Framework wird in dieser Arbeit nicht beleuchtet, ist aber in der Dokumentation von Vue[1] nachzulesen.
Zusätzlich zu Vue soll das Framework Nuxt.js[2] verwendet werden. Bei Nuxt handelt es sich um eine Art Meta-Framework, welches mehrere Frameworks (Vue, Node.js, Webpack, und Babel.js) bündelt und das Nutzen dieser Frameworks erleichtert [19]. Nuxt soll für den MVP des Blogs verwendet werden, da es das Aufsetzen von Vue-Applikationen stark vereinfacht und im späteren Projektverlauf auch für Teile des Backends verwendet werden kann.
Außerdem wird, als sehr einfaches Content-Management-System, das Content Plugin[3] von Nuxt verwendet. Hierbei handelt es sich um ein auf Git basierendes CMS, mithilfe dessen dynamisch Artikel aus Markdown Dateien geladen werden können.
Der MVP soll als statische Website auf einem Webserver deployed werden. Die dafür nötigen Dateien lassen sich mithilfe von Nuxt aus den Vue-Dateien und dem Content-Plugin kompilieren.
Alle verwendete Technologien lassen sich im finalen Produkt weiter verwenden. Außerdem treten keine Kosten für die Verwendung der Frameworks auf, da beide Projekte kostenlos zur Verfügung gestellt werden.

[1] https://vuejs.org/guide/introduction.html
[2] https://nuxtjs.org/
[3] https://content.nuxtjs.org/

6 Implementierung des MVPs

Nachdem der MVP, wie im vorherigen Kapitel beschrieben geplant wurde, wurde er umgesetzt. In diesem Kapitel wird zunächst auf die Organisation der Entwicklung eingegangen und danach kurz beschrieben, wie die ausgewählten Anforderungen umgesetzt worden sind.

6.1 Organisation der Entwicklung

Um die Entwicklung übersichtlich zu organisieren wurden alle Anforderungen in ein Kanban-Board eingepflegt und dort entsprechend ihres Status verschoben. Das Projekt wird mithilfe von Git als Versionskontrolle geführt[1].

6.2 Aufsetzen des Projektes

Das zur Entwicklung des Blogs verwendete Betriebssystem war Linux. Da die Installation der Frameworks für diese Arbeit nicht relevant ist, wird diese an dieser Stelle als gegeben betrachtet und nicht weiter beschrieben. Dank Nuxt ist es sehr einfach, ein neues Projekt aufzusetzen. Dazu wird durch den Befehl `npm init nuxt-app <project-name>` der Dialog um ein neues Nuxt-Projekt aufzusetzen in einem Terminal ausgeführt. In dem Dialog werden alle benötigten Einstellungen (wie zum Beispiel die Installation des Content-Plugins) getätigt. Nachdem der Dialog ausgeführt wurde, kann die Anwendung im Entwicklungsmodus mit dem Befehl `npm run dev` gestartet werden.

[1] Repository des Projektes: https://git.haw-hamburg.de/acq833/blog-bachelor-thesis

6.3 Implementierung der Anforderungen

Auf der Website von Nuxt existiert ein Tutorial, in dem ein Blog mit dem Content-Plugin aufgesetzt wird[2]. Orientiert an diesem Tutorial, wurden die Grundfunktionalitäten des Blogs implementiert. Es wurde Teile des Quellcodes aus dem Tutorial übernommen, da das Tutorial eine Art Erweiterung der Dokumentation des Content-Plugins darstellt. Im Folgenden wird aufgeführt, wie die ausgewählten Anforderungen umgesetzt wurden und, wenn nötig, anhand von Screenshots aus der Anwendung präsentiert.
Zur besseren Nachvollziehbarkeit der Verzeichnisstrukturen werden die wichtigsten Verzeichnisse in Abbildung 6.1 dargestellt. Für die gesamte Verzeichnisstruktur steht das Projekt unter `https://git.haw-hamburg.de/acq833/blog-bachelor-thesis` zur Verfügung. In dem Verzeichnis **components** werden Vue-Komponenten abgelegt, die dank Nuxt überall im Projekt ohne einen expliziten Import verwendet werden können. Das Verzeichnis **content** beinhaltet die Unterverzeichnisse **articles** und **authors**. In diesen befinden sich respektive Artikel als Markdown-Dateien und Autor:innen als YAML-Dateien, aus denen das Content-Plugin dynamisch Inhalt laden kann. Durch die Struktur im **pages** Verzeichnis wird die Navigationsstruktur der Website und die Struktur der URL festgelegt. Die URL bekommt dadurch beispielsweise für die Seite, auf der alle Autor:innen aufgelistet werden folgenden Form: `basis-url.de/blog/authors`. Die restlichen Verzeichnisse sind an dieser Stelle unwichtig und werden deshalb nicht weiter beleuchtet.

6.3.1 Anzeigen von Artikeln

Zum Anzeigen der Artikel wird eine dynamische Komponente erstellt, die über das Content-Plugin Artikel, die in dem **content** Verzeichnis abgelegt werden, anzeigt. Die Komponente ist in der Datei `_slug.vue` realisiert. Das Content-Plugin stellt für das Laden von Inhalten die asynchrone Methode `fetch()` zur Verfügung. Mit der in 6.1 dargestellten Zeile wird beispielsweise im Javascript-Teil der Vue-Datei nicht-blockierend der in der URL angegebene Artikel als Javascript Objekt in die Konstante `article` geladen.

```
1 const article = await $content('articles', params.slug).fetch();
```

Listing 6.1: Asynchrones Laden der Artikel

[2] https://nuxtjs.org/tutorials/creating-blog-with-nuxt-content

Abbildung 6.1: Wichtige Verzeichnisstrukturen des Blogs (eigene Darstellung)

Dabei wird der letzte Teil der URL (zum Beispiel in der URL `basis-url.de/blog/test` der Teil **test**) über `params.slug` an das Content-Plugin übergeben. Dieser Teil der URL wird im Folgenden als Slug bezeichnet. Das Plugin sucht nun aus dem entsprechenden Verzeichnis die **test** benannte Datei heraus. Das `$content` Objekt stellt dabei die Schnittstelle zum Plugin dar, über das die Methoden des Plugins verwendet werden können. Im HTML-Teil der Datei kann nun über die Konstante `article` auf den Artikel zugegriffen werden. Es kann auf die Attribute des Artikels, wie zum Beispiel das Datum, an dem er erstellt wurde, zugegriffen werden. Mit der in 6.2 gezeigten Zeile kann nun der Artikel so angezeigt werden, wie er als Markdown formatiert ist.

```
1 <nuxt-content :document="article" />
```

Listing 6.2: Zeile zur Darstellung der Artikel

6.3.2 Schreiben von Artikeln

Artikel müssen im Markdown-Format geschrieben werden. Dieses bietet viele grundlegende Formatierungsoptionen, wie beispielsweise hierarchische Überschriften, Aufzählungen und das Einfügen von Bildern. Um einem Artikel Metadaten, wie zum Beispiel den Namen der Autor:in zuzuweisen, können diese im YAML-Format vor dem eigentlichen Artikel eingefügt werden. Dazu werden die Metadaten mit jeweils einer Zeile mit dem Inhalt `---` vor und nach den YAML-Daten umklammert. So können Werte als Key-Value in den

in Kapitel 6.3.1 beschriebenen Artikel-Objekten mit übergeben werden. Ein Beispiel für den Metadaten-Teil eines Artikels ist in dem Listing 6.3 angegeben.

```
---
title: 'Test'
description: 'This is a short test article'
image: 'https://some.image.link'
author: 'Paul'
---
```

Listing 6.3: YAML-Metadaten eines Artikels

6.3.3 Navigieren zwischen Artikeln

Um zwischen den Artikeln zu navigieren, werden am Ende der Artikel zwei Buttons angezeigt, über die zu dem zeitlich gesehenen nächsten und vorherigen Artikel gesprungen werden können. Dazu müssen, zusätzlich zu dem eigentlichen Artikel, auch diese Artikel geladen werden. Mit der in Listing 6.4 gezeigten Funktion werden alle Artikel geladen, davon jedoch nur der Titel und der Slug gespeichert, aufsteigend nach Datum sortiert und der Artikel, vor und nach dem eigentlichen Artikel, in ein die Konstanten `prev` und `next` gespeichert.

```
const [prev, next] = await $content('articles')
    .only(['title', 'slug'])
    .sortBy('createdAt', 'asc')
    .surround(params.slug)
    .fetch();
```

Listing 6.4: Asynchrones Laden des vorherigen und folgenden Artikels

Die Navigation wird nun über zwei, in der `PrevNext` Komponente definierte, Buttons durchgeführt.

6.3.4 Durchsuchen von Artikeln

Zum Durchsuchen der Artikel stellt das Content-Plugin die asynchrone `search()` Methode zur Verfügung. Durch diese lassen sich die Artikel nach einem oder mehreren Stichworten durchsuchen. Das Stichwort wird dabei über ein Eingabefeld, das immer angezeigt wird, eingegeben. Sobald etwas eingeben wird, werden Ergebnisse, wie in Abbildung 6.2 dargestellt, angezeigt. Wird auf einen Artikel geklickt, wird dieser aufgerufen und das

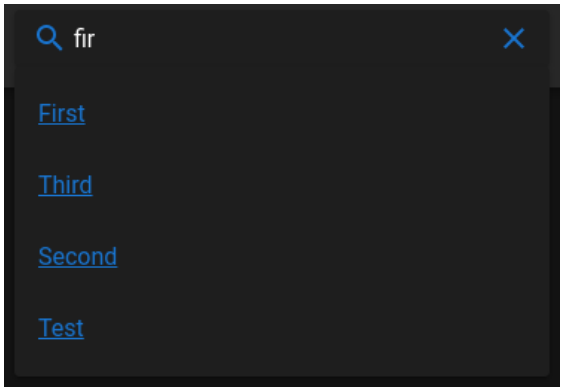

Abbildung 6.2: Suchleiste des Blogs während einer Suche (eigene Darstellung)

Eingabefeld geleert. Die Artikel werden wie in Listing 6.5 gezeigt durchsucht. Dabei wird die Suche auf die ersten 6 Artikel limitiert, damit das Suchfeld übersichtlich bleibt. Die Variable `searchQuery` wird dabei mit dem Text aus dem Eingabefeld belegt.

```
async searchQuery(searchQuery) {
    if (!searchQuery) {
        this.articles = [];
        return;
    }
    this.articles = await this.$content('articles')
        .limit(6)
        .search(searchQuery)
        .fetch();
},
```

Listing 6.5: Durchsuchen von Artikeln

Wird, während das Suchfeld im Fokus ist und etwas beinhaltet, auf Eingabe gedrückt, öffnet sich eine Seite, auf der alle Artikel, auf die die Suche zutrifft, angezeigt werden (siehe Abbildung 6.3). Das Stichwort wird in der URL an die neue Seite wie in Listing 6.6 dargestellt übergeben und das Eingabefeld geleert. Auf der neuen Seite wird nun analog zur ersten Suche eine neue durchgeführt, diese aber nicht limitiert.

```
fullSearch() {
    this.$router.push({
        path: '/blog/search?keyword=' + this.searchQuery,
    });
    this.clearQuery();
},
clearQuery() {
```

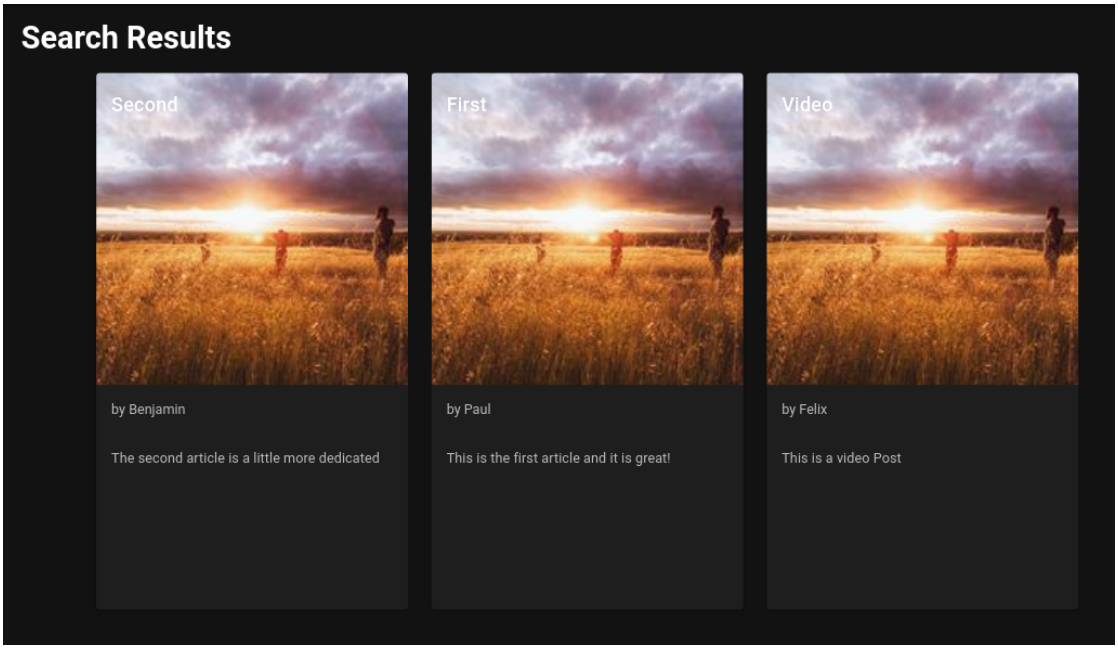

Abbildung 6.3: Suchergebnisse nach aus der Suchleiste ausgeführter Suche des Blogs (eigene Darstellung)

```
8       this.searchQuery = '';
9   },
```

Listing 6.6: Volle Suche

6.3.5 Unterschiedliche Ansicht von Artikellayouts

Die unterschiedliche Ansicht von Artikeln, je nach Inhalt, wird in der in Kapitel 6.3.1 beschriebenen `_slug` Komponente realisiert. Vue bietet mit If-Bedingungen versehene Blöcke im HTML-Teil an, die nur angezeigt werden, wenn bestimmte Bedingungen erfüllt werden. Als Beispiel wurde eine Einbettung von YouTube-Videos implementiert, die nur angezeigt wird, wenn in dem YAML-Teil des Artikels eine Video-URL unter dem Schlüssel `videoURL` angegeben wird. Der zugehörige Quellcode wird in Listing 6.7 dargestellt. Zur Einbettung des Videos wird das Plugin „vue-youtube-embed“[3] verwendet.

[3]Siehe https://openbase.com/js/vue-youtube-embed

```
<YoutubeEmbed
    v-if="article.videoURL"
    :videoURL="article.videoURL"
></YoutubeEmbed>
```

Listing 6.7: Dynamische Einbettung von YouTube-Videos

6.3.6 Gruppieren der Artikel nach Autor:innen

Zur Gruppierung der Artikel nach Autor:innen wurden für jede Autor:in, der in dem **content** Verzeichnis in dem **authors** Unterverzeichnis als YAML-Datei abgelegt ist, dynamisch eine Seite angelegt. Diese Komponente (`_author`) wurde analog zu der in Kapitel 6.3.1 beschriebenen `_slug` Komponente erstellt. Der Unterschied ist, dass statt Artikeln Autor:innen dynamisch geladen werden können.
Um alle Artikel einer Autor:in anzuzeigen, wird die `where()` Methode des Content-Plugins verwendet. Mit dieser lassen sich Inhalte über reguläre Ausdrücke (RegEx) filtern. In dem Listing 6.8 werden alle Artikel so gefiltert, dass nur solche übrig bleiben, die von der aktuell angezeigten Autor:in geschrieben worden sind. Mit der `without()` Methode können Teile der Artikel-Objekte weggelassen werden.

```
const articles = await $content('articles')
    .where({
        author: {
            $regex: [params.author, 'i'],
        },
    })
    .without('body')
    .sortBy('createdAt', 'asc')
    .fetch();
```

Listing 6.8: Filtern von Artikeln nach Autor:in

Die so gefilterten Artikel werden nun, wie in Listing 6.9 gezeigt, mithilfe eines For-Loops dargestellt. Vue stellt diesen als `v-for` bereit, um im HTML-Teil durch Sammlungen aus dem Javascript-Teil iterieren zu können. Gibt es von der Autor:in keine Artikel, wird das ganze Element durch ein `v-if` ausgeblendet.

```
<v-container v-if="articles.length > 0" class="mx-auto px-0 px-sm-auto">
    <v-row align="center" justify="center">
        <v-col
            v-for="article of articles"
            :key="article.slug"
            cols="auto"
        >
            <article-preview :article="article"></article-preview>
        </v-col>
    </v-row>
</v-container>
```

Listing 6.9: Anzeigen der gefilterten Artikel

6.3.7 Kurzbeschreibung der Autor:innen

Auf der in Kapitel 6.3.6 beschrieben Seite für Autor:innen wird eine Kurzbeschreibung dieser angezeigt. Die Kurzbeschreibung der Autor:in wird in der jeweiligen YAML-Datei unter dem Schlüssel `bio` abgelegt. Außerdem haben Autor:innen die Möglichkeit unter dem Schlüssel `image` ein Link zu einem Bild von sich anzugeben. Dieses Bild wird dann auch auf der Seite der Autor:in angezeigt.

6.4 Fazit zur Implementierung

Durch die gute Dokumentation von Nuxt war es möglich, den Blog innerhalb einer Woche so umzusetzen, dass er als MVP veröffentlicht werden kann. Durch den Befehl `npm run generate` lassen sich aus dem Projekt die notwendigen Dateien erstellen, um die Anwendung als statische Website auf einem Webserver zu deployen.
Aufgrund von Vorerfahrungen mit Vue war es einfach, sich in das Nuxt Framework einzuarbeiten. Die Verwendung von Nuxt hat, im Vergleich zur Entwicklung von BRHH ohne Nuxt, viele Punkte stark vereinfacht. Ein aufzuführender Punkt ist das Routing der Website, welches Nuxt automatisiert umsetzt[4] und einem so viel Arbeit erspart. Insgesamt hat es sich gelohnt, Nuxt zu verwenden und nicht nur mit Vue zu arbeiten.
Dadurch, dass dank der in der Arbeit beschriebenen Methode die Entwicklung gut geplant wurde, verlief diese reibungslos.

[4] Mehr dazu in der Dokumentation von Nuxt zum Punkt Routing https://nuxtjs.org/docs/get-started/routing

7 Bewertung der Methode

In diesem Kapitel soll die Methode, anhand der gesammelten Erfahrung aus der Anwendung auf das zweite Fallbeispiel, bewertet werden. Dazu wird zunächst die Struktur der Methode bewertet. Darauf folgend werden die Heuristiken, die die Methode zur Auswahl der Art des MVPs, der Architektur sowie der Frameworks bereitstellt, bewertet. Abschließend wird ein Fazit gezogen, bei dem vor allem die Entwicklung mit der des MVPs von BRHH verglichen wird. Aufgrund der Tatsache, dass in dieser Arbeit die Methode von der gleichen Person entwickelt und getestet wurde besteht eine gewisse positive Voreingenommenheit gegenüber der Methode. Die Methode wurde des Weiteren für die eigene Problemstellung des Autors entwickelt. Um eine vollständige Objektivität zu gewährleisten, wäre es nötig, dass die Methode von einer zweiten, unabhängigen Person getestet wird. Außerdem ist zu betonen, dass die Methode nicht die einzig richtige Möglichkeit darstellt einen MVP zu entwickeln, sondern vor allem die Vorgehensweise des Autors vereinfacht.

7.1 Struktur der Methode

Durch die schrittweise Anwendung der Methode war der Entwicklungsprozess des MVPs klar strukturiert. Es war stets offensichtlich, welche Aufgaben noch ausgeführt werden müssen und welche bereits abgeschlossen sind. Die deutliche Definition der Zielsetzung des MVPs hat die Priorisierung und Auswahl der Anforderungen für den MVP beschleunigt und einfacher gestaltet. Durch die Priorisierung mit der MoSCoW-Methode konnten einige Anforderungen direkt für den MVP ausgewählt werden, was den Auswahlprozess vereinfacht hat.
Da die User Stories für das finale Produkt und nicht für den MVP erstellt worden sind, wurde verhindert, dass weitere Anforderungen im Laufe der Entwicklung dazukommen. Insgesamt ist die Entwicklungsphase sehr sauber nach der Methode verlaufen und ein

Springen zwischen den Schritten war nicht nötig, was für die Struktur der Methode und Reihenfolge der Schritte spricht.

7.2 Heuristiken der Methode

Durch die in Tabelle 4.1 dargestellte Hilfestellung zur Auswahl der Art des MVPs wurde sich schnell für den Frontend-MVP entschieden, da dieser die Anforderungen sehr gut abdeckt. An dieser Stelle muss jedoch auch eine Voreingenommenheit des Autors erwähnt werden, da dieser bereits mit Frontend-MVPs Erfahrungen aufweist. Unabhängig davon ist ein Frontend-MVP dennoch die richtige Wahl für das Ziel des MVPs und spiegelt sich in den Werten der Tabelle wider. Näheres dazu wurde bereits in Kapitel 5.5 beschrieben.
Die Architektur wurde anhand der Tabelle 4.2 gewählt. Es wurde sich für eine, auf REST basierende Microservice-Architektur entschieden. Durch die Wahl von Vue wurde das MVVM-Muster für das Frontend verwendet. Das Frontend kommuniziert über eine Client/ Server-Architektur mit dem Webserver. Die Heuristik war insofern hilfreich, dass ein guter Überblick über die infrage kommenden Architekturstile geschaffen werden konnte. Da die Entwicklungszeit nicht das Hauptkriterium war, wurde eine Architektur verwendet, die einfach in dem finalen Produkt weitergeführt werden kann. Insgesamt könnte die Heuristik im Rahmen einer weiteren Arbeit um einige Kriterien erweitert werden. Darüber hinaus könnten zusätzlich weitere Architekturstile aufgenommen werden, um eine größere Auswahl zu bieten.
Die Hilfestellungen zur Auswahl der Frameworks haben die Entscheidung für Vue stark unterstützt. An dieser Stelle ist zu erwähnen, dass einer der Hauptpunkte, der zur Auswahl des Frameworks Vue geführt hat, die Erfahrung mit dem Framework ist. Auch die anderen in Kapitel 4.2.1 aufgeführten Punkte werden abgedeckt. Zur Auswahl des Frameworks Nuxt hat vor allem der Punkt der Zeitersparnis geführt. Generell hat sich herausgestellt, dass die Hilfestellung zur Auswahl der Frameworks sehr hilfreich war.
Die Heuristiken waren sehr hilfreich, sind aber vor allem im Bereich der Auswahl der Architektur noch erweiterbar.

7.3 Abschließendes Fazit und Vergleich zu Bandrecording Hamburg

Im Vergleich zum Entwicklungsprozess von BRHH war stets offensichtlich, welcher Schritt als Nächstes auszuführen ist. Bei BRHH wurde während der Entwicklungsphase viel zwischen Entwurfs- und Implementierungsphase gewechselt. Dies hat dazu geführt, dass in vielen Aspekten zu früh mit der Implementierung begonnen wurde, sodass mehrmals Teile der Anwendung nachträglich angepasst werden mussten. Bei der Entwicklung des Blogs wurde die Umsetzung der Anforderungen im Voraus gut genug geplant, um dies zu verhindern.
Des Weiteren gab es bei dem Blog, anders als bei BRHH keinen Punkt, in den zu viel Zeit in Details investiert wurde.
Durch die Verwendung von Nuxt wurde die zum Aufsetzen des Projektes benötigte Zeit, im Vergleich zu BRHH, deutlich minimiert.
Zur Reliabilität, also der Wiederholbarkeit der Methode lassen sich nur Vermutungen aufstellen, da diese bisher nur einmal getestet wurde. Man kann jedoch davon ausgehen, dass mit der Methode auch bei der Durchführung durch eine andere Person schnell ein passender MVP entstehen würde. Dieser würde sich jedoch auf jeden Fall von dem im Rahmen dieser Arbeit entwickelten Methode unterscheiden, da es zu viele Variablen gibt, die mit den persönlichen Vorlieben der Entwickler:innen zusammenhängen. Die Auswahl der Frameworks sowie der Methode zur Priorisierung der Anforderungen sind nur einige Beispiele dafür.
Ein umfangreicher Test der Methode könnte im Rahmen einer weiteren Arbeit durchgeführt werden, indem mehrere Entwickler:innen mit der Methode einen MVP für das gleiche Produkt entwickeln und diese dann untereinander verglichen werden.
Insgesamt wurden die Ziele der Forschungsarbeit erreicht. Es wurde eine umfangreiche Methode vorgestellt, die die Entwicklung eines MVPs einer Webanwendung effizienter gestaltet hat, als es ohne die Methode der Fall gewesen wäre. Außerdem haben die Heuristiken dazu beigetragen, dass die Unterschritte der Methode zielgerichtet ausgeführt werden konnten. Durch die Anwendung der Methode zur Entwicklung des Blogs wurde diese überprüft und als hilfreich bewertet.

8 Zusammenfassung und Ausblick

Abschließend wird die Arbeit zur besseren Übersicht zusammengefasst und die Ergebnisse knapp dargestellt. Außerdem wird ein Ausblick auf Möglichkeiten zur Weiterführung der Forschungsarbeit gegeben. Dazu werden Anstöße für weiterführende Forschung vorgestellt.

8.1 Zusammenfassung

Im Rahmen der Arbeit wurde, nachdem alle erforderlichen Grundlagen beschrieben worden sind die Anwendung Bandrecording Hamburg zusammen mit ihrem abgeschlossenen MVP vorgestellt. Der MVP der Anwendung sowie die Entwicklung dieses wurde kritisch begutachtet und analysiert. Aus den Ergebnissen der Analyse wurde anschließend eine Methode erarbeitet, mithilfe derer die Entwicklung von MVPs für weitere Anwendungen vereinfacht und beschleunigt werden soll. Der Fokus der Methode liegt dabei auf der Entwicklung von MVPs für Webanwendungen. Die Methode besteht aus acht Unterschritten, die die aus der Analyse hervorgegangenen zu beachtenden Punkte bei der Entwicklung eines MVPs abdecken. Als wichtige Punkte wurde die klare Definition des Ziels des MVPs, die Priorisierung der Anforderungen, eine fundierte Wahl der zu verwendenden Technologien, die Wiederverwendbarkeit des MVPs im finalen Produkt sowie die transparente Organisation der Entwicklungsphase herausgearbeitet.
Die erarbeitete Methode wurde an einem zweiten Fallbeispiel, dem eines Reiseblogs, angewendet und dadurch getestet. Nachdem kurz beschrieben wurde, wie der Blog umgesetzt worden ist, wurde die Methode bewertet.
Das Ergebnis der Arbeit ist, dass durch Anwenden der Methode die Entwicklung des MVPs sehr gut strukturiert war und zu jedem Schritt klar war, was noch zu erledigen ist bis der MVP fertig ist. Außerdem wurden die vorgestellten Heuristiken innerhalb der Methode als hilfreich bewertet. Damit wurde das Hauptziel der Forschungsarbeit, die Entwicklung der Methode und die Überprüfung dieser erreicht.

8.2 Ausblick

Um die Methode weiterführend zu testen, müssten weitere MVPs nach anhand dieser entwickelt werden. Da eine Entwicklung eines MVP umfangreich ist, würden weitere Entwicklungen den Rahmen dieser Arbeit überschreiten.
Um die Reliabilität der Methode zu prüfen, wäre es nötig, dass sie von andere Entwickler:innen angewendet und bewertet wird.
Da die Methode aus der Erfahrung eines einzelnen MVPs entwickelt wurde, kann diese nach dem Sammeln weiterer Erfahrung und anschließendem Anpassen der Methode verbessert werden.
Durch das umfangreiche Testen aller in Kapitel 4.2.1 beschriebenen Methoden zur Priorisierung der Anforderungen könnten ausführlichere Empfehlungen zur Auswahl einer der Methoden bereitgestellt werden. Außerdem könnten die in Kapitel 4.2.1 beschriebenen Heuristiken verbessert werden, indem mehr Erfahrung mit den einzelnen Architekturstilen und verschiedenen Arten von MVPs gesammelt wird. Um die Heuristiken zu erweitern, könnten weitere Architekturstile und Arten von MVPs mit aufgenommen werden.
In dieser Arbeit wurde der Fokus vor allem auf die Entwicklungsphase des MVPs gelegt. Als Thema einer weiteren Forschungsarbeit wäre es interessant, die Veröffentlichung sowie den Feedbackprozess genauer zu untersuchen und auch hierfür eine Methode zu erstellen, die bei beiden Vorgängen hilfreich ist.
Da sich diese Forschungsarbeit vor allem mit MVPs für Webanwendungen befasst, ist ein weiterer Forschungsansatz die Generalisierung der Methode um diese für jegliche Art von Anwendung verwenden zu können. Dazu müssten vor allem die Bereiche der Arten des MVPs sowie der Architekturstile erweitert werden. Viele Teile der Methode, wie zum Beispiel die Priorisierung der Anforderungen ließen sich jedoch übernehmen.

Literaturverzeichnis

[1] *Macroservices – Nicht kleine Teile, sondern das große Ganze | Informatik Aktuell.* https://www.informatik-aktuell.de/entwicklung/methoden/macroservices-nicht-kleine-teile-sondern-das-grosse-ganze.html. – [Online; accessed on 21. Feb. 2022]

[2] *Minimum Viable Product | SyncDev.* https://web.archive.org/web/20160525101214/http://www.syncdev.com:80/minimum-viable-product/. – [Online; accessed on 13. Jan. 2022]

[3] *The Lean Startup | Methodology.* Oktober 2017. – URL http://theleanstartup.com/principles. – [Online; accessed 13. Jul. 2022]

[4] *4 Product Backlog Prioritization Techniques That Work | Perforce Software.* April 2022. – URL https://www.perforce.com/blog/hns/4-product-backlog-prioritization-techniques-work. – [Online; accessed 2. May 2022]

[5] *Presentation Model.* Juli 2022. – URL https://martinfowler.com/eaaDev/PresentationModel.html. – [Online; accessed 16. Jul. 2022]

[6] *RFC 1630 - Universal Resource Identifiers in WWW: A Unifying Syntax for the Expression of Names and Addresses of Objects on the Network as used in the World-Wide Web.* August 2022. – URL https://datatracker.ietf.org/doc/html/rfc1630. – [Online; accessed 2. Aug. 2022]

[7] *RFC 1945 - Hypertext Transfer Protocol – HTTP/1.0.* August 2022. – URL https://datatracker.ietf.org/doc/html/rfc1945. – [Online; accessed 2. Aug. 2022]

[8] *Vue.js - The Progressive JavaScript Framework | Vue.js.* Juli 2022. – URL https://vuejs.org/guide/introduction.html. – [Online; accessed 15. Jul. 2022]

[9] *Was ist eine API?* Juli 2022. – URL https://www.redhat.com/de/topics/api/what-are-application-programming-interfaces#soas-oder-microservices. – [Online; accessed 2. Aug. 2022]

[10] *Was ist eine REST-API?* Juli 2022. – URL https://www.redhat.com/de/topics/api/what-is-a-rest-api#was-ist-eine-rest-api. – [Online; accessed 2. Aug. 2022]

[11] *Was ist SOA (Service-Oriented Architecture)?* August 2022. – URL https://www.redhat.com/de/topics/cloud-native-apps/what-is-service-oriented-architecture. – [Online; accessed 3. Aug. 2022]

[12] *Zustandsbehaftet oder zustandslos?* August 2022. – URL https://www.redhat.com/de/topics/cloud-native-apps/stateful-vs-stateless. – [Online; accessed 2. Aug. 2022]

[13] Atlassian: *User Storys | Beispiele und Vorlage | Atlassian.* Juli 2022. – URL https://www.atlassian.com/de/agile/project-management/user-stories. – [Online; accessed 14. Jul. 2022]

[14] Autoren der Wikimedia-Projekte: *Goldene Schallplatte – Wikipedia.* Februar 2004. – URL https://de.wikipedia.org/w/index.php?title=Goldene_Schallplatte&oldid=222180038. – [Online; accessed 7. Aug. 2022]

[15] Bass, Len ; Clements, Paul ; Kazman, Rick: *Software Architecture In Practice*, 01 2003. – ISBN 978-0321154958

[16] Brandt-Pook, Hans ; Kollmeier, Rainer: *Softwareentwicklung kompakt und verständlich.* Springer, 2015

[17] Bulkar, Mangesh: The Difference Between Static and Dynamic Websites? In: *LinkedIn* (2018), Januar. – URL https://www.linkedin.com/pulse/difference-between-static-dynamic-websites-mangesh-bulkar

[18] Buschmann, Frank ; Henney, Kevlin ; Schmidt, Douglas C.: *Pattern-oriented software architecture, on patterns and pattern languages.* Bd. 5. John wiley & sons, 2007

[19] Contributors to Wikimedia projects: *Nuxt.js - Wikipedia.* Juli 2022. – URL https://en.wikipedia.org/w/index.php?title=Nuxt.js&oldid=1097159688. – [Online; accessed 15. Jul. 2022]

[20] Diehl, Andreas: Minimum Viable Product - Die Anleitung für dein nächstes MVP - Andreas Diehl (#DNO). In: *Andreas Diehl (#DNO)* (2022), Juni. – URL https://digitaleneuordnung.de/blog/mvp-minimum-viable-product/#dein-mvp-entwickeln

[21] Dowalil, Herbert 1.: *Grundlagen des modularen Softwareentwurfs der Bau langlebiger Mikro- und Makro-Architekturen wie Microservices und SOA 2.0.* Hanser, 2018 (Hanser eLibrary). – URL https://www.hanser-elibrary.com/doi/book/10.3139/9783446456006

[22] Duc, Anh N. ; Abrahamsson, Pekka: Minimum Viable Product or Multiple Facet Product? The Role of MVP in Software Startups. In: Sharp, Helen (Hrsg.) ; Hall, Tracy (Hrsg.): *Agile Processes, in Software Engineering, and Extreme Programming.* Cham : Springer International Publishing, 2016, S. 118–130. – ISBN 978-3-319-33515-5

[23] Eilebrecht, Karl ; Starke, Gernot: *Patterns kompakt: Entwurfsmuster für effektive Softwareentwicklung.* Springer-Verlag, 2018

[24] Floyd, Christiane: A systematic look at prototyping. In: *Approaches to prototyping.* Springer, 1984, S. 1–18

[25] Horn, Thorsten: *Architekturen für Webanwendungen.* https://www.torsten-horn.de/techdocs/webanwendungen.htm. – [Online; accessed on 30. March 2022]

[26] kexugit: *Advantages and disadvantages of M-V-VM.* Juli 2022. – URL https://docs.microsoft.com/en-us/archive/blogs/johngossman/advantages-and-disadvantages-of-m-v-vm. – [Online; accessed 16. Jul. 2022]

[27] kexugit: *Patterns - WPF Apps With The Model-View-ViewModel Design Pattern.* Juli 2022. – URL https://docs.microsoft.com/en-us/archive/msdn-magazine/2009/february/patterns-wpf-apps-with-the-model-view-viewmodel-design-pattern. – [Online; accessed 16. Jul. 2022]

[28] Manchanda, Amit: A Step-by-Step Guide to Build a Minimum Viable Product (MVP). In: *Insights - Web and Mobile Development Services and Solutions* (2022), April. – URL https://www.netsolutions.com/insights/how-

to-build-an-mvp-minimum-viable-product-a-step-by-step-guide/#conclusion

[29] Meyer, Albin: *Softwareentwicklung: Ein Kompass für die Praxis*. De Gruyter Oldenbourg, 2018. – URL https://doi.org/10.1515/9783110578379. – ISBN 9783110578379

[30] Newman, M. E. J.: *Power laws, Pareto distributions and Zipf's law*. September 2019. – URL https://arxiv.org/PS_cache/cond-mat/abs/0412/0412004v3.pdf. – [Online; accessed 29. Jun. 2022]

[31] Vogel, Oliver (Hrsg.): *Software-Architektur Grundlagen - Konzepte - Praxis*. 2. Auflage. Spektrum Akademischer Verlag, 2009. – URL http://site.ebrary.com/lib/alltitles/docDetail.action?docID=10277696

Erklärung zur selbstständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original