



Hochschule für Technik
und Wirtschaft Berlin

University of Applied Sciences

Gemeinsames Cloud-Deployment von Backend und Frontend aus einem Repository

Bachelorarbeit

Name des Studiengangs

Wirtschaftsinformatik

Fachbereich 4

vorgelegt von

Ivan Birkmaier

Datum:

Berlin, 05.02.2023

Erstgutachter: Prof. Dr. Arif Wider

Zweitgutachter: Prof. Dr. Axel Hochstein

Kurzbeschreibung

Diese Arbeit untersucht die Vorteile eines gemeinsamen Repository beim automatisierten Deployment mittels Cloud-Technologien. Der hierfür verwendete Prototyp ist auf GitHub öffentlich zugänglich und verwendet unter anderem Docker und Nx als Technologien. Das automatisierte Deployment wird über eine CI/CD-Pipeline umgesetzt.

Schlagworte: CI/CD-Pipeline, Mono-Repository, Cloud, Version Control System, Docker, Nx, Heroku, GitHub Actions.

Gender Erklärung

Aus Gründen der besseren Lesbarkeit wird in dieser Bachelorarbeit die Sprachform des generischen Maskulinums angewandt. Es wird an dieser Stelle darauf hingewiesen, dass die ausschließliche Verwendung der männlichen Form geschlechtsunabhängig verstanden werden soll.

Inhaltsverzeichnis

1	Einleitung	1
2	Fachliche Grundlagen	3
2.1	Cloud Computing	3
2.2	Webanwendungen	4
2.3	Repository	5
2.3.1	Das Repository im Kontext der Informatik	6
2.3.2	Repository Modelle	7
2.4	Cloud-Deployment	9
2.4.1	Virtualisierung	9
2.4.2	CI/CD-Pipelines	12
2.5	Verwandte Arbeiten	14
3	Requirements Engineering	16
3.1	Begriffe aus dem Requirements Engineering	16
3.2	Definition und Umsetzung des Requirements Engineering	18
3.2.1	Theoretische Methodik zur Durchführung des Requirements Engineering	18
3.2.2	Feststellung der Stakeholder, der Projektumgebung und einer Ermittlungstechnik zur Erhebung von Anforderungen	20
3.2.3	Erstellung einer Anforderungsschablone zur Erhebung von Anforderungen	20
3.2.4	Spezifizierung, Validierung, Priorisierung und Qualitätskontrolle der erhobenen Anforderungen	23
3.2.5	Erhebung von Abnahmekriterien	27
3.2.6	Anforderungen modellieren	29

4	Implementierung	34
4.1	Einrichtung des Version Controll Systems und der Kollaborationsplattform	34
4.1.1	Betroffene Anforderungen und Abnahmekriterien	34
4.1.2	Verwendete Technologien	35
4.1.3	Umsetzung des Arbeitsschritts	35
4.2	Erstellen eines Mono-Repositorys	36
4.2.1	Betroffene Anforderungen und Abnahmekriterien	36
4.2.2	Verwendete Technologien	37
4.2.3	Umsetzung des Arbeitsschritts	38
4.3	Implementierung einer Webanwendung	39
4.3.1	Betroffene Anforderungen und Abnahmekriterien	39
4.3.2	Verwendete Technologien	40
4.3.3	Umsetzung des Arbeitsschritts	40
4.4	Dockerisierung des Prototyps	42
4.4.1	Betroffene Anforderungen und Abnahmekriterien	42
4.4.2	Verwendete Technologien	42
4.4.3	Umsetzung des Arbeitsschritts	43
4.5	Bauen einer CI/CD-Pipeline	45
4.5.1	Betroffene Anforderungen und Abnahmekriterien	45
4.5.2	Verwendete Technologien	45
4.5.3	Umsetzung des Arbeitsschritts	45
4.6	Deployment auf Heroku	49
4.6.1	Betroffene Anforderungen und Abnahmekriterien	49
4.6.2	Verwendete Technologien	49
4.6.3	Umsetzung des Arbeitsschritts	50
5	Evaluation	54
6	Fazit	56
	Abbildungsverzeichnis	I
	Source Code Content	III
	Literaturverzeichnis	IV
	Onlinereferenzen	IX
	Eigenständigkeitserklärung	XIV

Kapitel 1

Einleitung

Ein Softwareentwickler, der seinen akademischen Werdegang beginnt, wird in der Regel damit beginnen, lauffähigen Code zu schreiben. Bei ersten Praxiserfahrungen wird dann deutlich, dass moderne Softwaresysteme weitere Lebenszyklen durchlaufen. Einer dieser Zyklen ist das *Deployment*. Unter dem Begriff Deployment versteht man im Allgemeinen den Prozess der Bereitstellung und Instandhaltung von Software für Endbenutzer [ABL15]. Hierbei ist wichtig zu beachten, dass es verschiedene Arten eines Software-Deployments gibt. So wird beispielsweise beim *On-Premise Deployment* (Vor Ort Bereitstellung), Software auf der eigenen physischen IT-Infrastruktur des Verbrauchers bereitgestellt [Ari19]. Beim sogenannten *Cloud-Deployment*, wird die Software von einer Cloud-Infrastruktur eines Cloud-Anbieters betrieben. [Hal19].

Einer der Gründe, weshalb Unternehmen von einem On-Premise Deployment ihrer Software hin zu einer Cloud-Lösung wechseln, sind die hohen finanziellen Ersparnisse bei den Kapitalkosten und den Betriebskosten. Diese sinken, da die Wartung und Instandhaltung der hauseigenen Serverlandschaften an den Cloud-Anbieter abgegeben werden [BRM21]. So basieren Cloud-Lösungen für das Bereitstellen von Software für den Verbraucher auf einem *Pay-as-you-go-model*. Dieses ermöglicht es Unternehmen, Rechnerressourcen und Infrastruktur bedarfsgerecht über das Internet zu beziehen. Der Prozess des Deployments der Software auf die Cloud-Infrastruktur bleibt jedoch ein sehr komplexer, der seitens der Unternehmen praktiziert werden muss [BRM21].

Diese Arbeit untersucht, wie sich ein derartiges Cloud-Deployment umsetzen lässt, mit welchen modernen Technologien sich das Cloud-Deployment automatisieren lässt und wie verschiedene Repository-Modelle das Cloud-Deployment unterstützen können.

Um diese Fragestellungen beantworten zu können, wurden für diese Arbeit die folgenden Ziele formuliert:

- Die ausführliche Vermittlung von fachlichen Grundlagen, die im Zusammenhang mit dem Thema dieser Arbeit stehen.
- Anwenden des Requirement Engineering als Methodik zur Erhebung und Auswertung qualitativ hochwertiger Anforderungen an eine fachliche Lösung, für das gemeinsame Cloud-Deployment von Frontend und Backend aus einem Repository.
- Die Implementierung eines Prototyps als fachliche Lösung für das vollständige und automatisierte Cloud-Deployment einer Webanwendung, bestehend aus einem Backend und einem Frontend. Hierbei soll der Quellcode des Frontends als auch des Backends in einem gemeinsamen Repository verwaltet werden. Jedoch soll bei diesem Prototyp das Cloud-Deployment als Prozess im Mittelpunkt stehen, sodass die Qualität der zu deployenden Webanwendung sich diesem unterordnet.

Die Ausarbeitung dieser Ziele wurde in verschiedene Kapitel unterteilt. Das Kapitel 2 „Fachliche Grundlagen“ ordnet die Themenbereiche, in ihren theoretischen Kontext. Das Kapitel erklärt die Begriffe und Methodiken, die notwendig sind, um die Implementierung des Prototyps zu verstehen. Im Kapitel 3 „Requirement Engineering“, werden Anforderungen erarbeitet und mit messbaren Kriterien verknüpft. Auf Basis dieser Anforderungen, wurde anschließend der Prototyp umgesetzt. Mit den messbaren Kriterien kann der Prototyp im Anschluss evaluiert werden. Das Kapitel 4 „Implementierung“ bildet den Hauptteil dieser Arbeit. Es beschreibt die elementaren Arbeitsschritte, die benötigt wurden, um aus den erhobenen Anforderungen eine konkrete fachliche Lösung abzuleiten. Bei der fachlichen Lösung handelt es sich um das gemeinsame Cloud-Deployment eines AngularJS Frontends und eines NestJS Backends aus einem Nx Mono-Repository auf die Cloud-Plattform Heroku. Kapitel 5 „Evaluation“, evaluiert diesen Prototypen. Abgerundet wird diese Arbeit, mit dem Kapitel 6 „Fazit“. Das Fazit fasst die umgesetzten Arbeitsergebnisse zusammen, betrachtet inwiefern diese die Zielstellungen erfüllen und gibt Empfehlungen, aus denen sich weiter Forschungsfragen, für kommende Arbeiten ableiten lassen können.

Kapitel 2

Fachliche Grundlagen

Die fachlichen Grundlagen legen den Grundstein für diese Arbeit. Sie sollen dazu dienen, alle Themenbereiche, die in den folgenden Kapiteln auftreten werden, in ihren fachlichen Kontext einordnen zu können. Zudem untersucht dieses Kapitel, mit dem Unterkapitel 2.5 „Verwandte Arbeiten“, ob es bereits vergleichbare Arbeiten, zum Thema dieser Arbeit gibt und beleuchtet deren Ergebnisse.

2.1 Cloud Computing

Das National Institute of Standards and Technology (NIST, Nationales Institut für Standards und Technologie) hat in den letzten Jahren die Definition von Cloud Computing stark geprägt [SK10]. Es definiert in seiner sieben seitigen Special Publication 800-145 (Sonderveröffentlichung), Cloud Computing wie folgt:

Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics, three service models, and four deployment models. [MG11, S. 2]

Eine detaillierte Ausführung der beschriebenen fünf „essential characteristics“ (wesentlichen Merkmale) und der vier „deployment models“ (Bereitstellungs Modelle) des Cloud Computings, lassen sich in der öffentlich zugänglichen Sonderveröffentlichung nachschlagen. Relevant für diese Arbeit sind jedoch die drei beschriebenen „service modelle“. Wie

das NIST-Institut diese definiert, wurde im Folgenden ausführlicher betrachtet [MG11, S. 2–3]:

- **Software as a Service (SaaS):** Beim Servicemodell SaaS (Software als Dienstleistung) wird dem Verbraucher über beispielsweise Webbrowser eine Anwendung zur Verfügung gestellt, die seitens des Anbieters in der Cloud ausgeführt wird. Ein nennenswertes Beispiel für Software-Anwendungen, die über das SaaS Model bereitgestellt werden, sind webbasierte E-Mail Postfächer.
- **Platform as a Service (PaaS):** Beim PaaS Model (Plattform als Dienstleistung) wird Verbrauchern von Cloud-Anbietern eine, nach den Bedürfnissen der Verbraucher, angepasste Cloud-Infrastruktur zur Verfügung gestellt. Auf diese können Verbraucher eigene Programmierungen oder zugekaufte Software-Anwendungen ablegen und verwalten. Jedoch hat der Verbraucher keinerlei Kontrolle oder Verwaltungsrecht über die ihm zur Verfügung gestellte Cloud-Infrastruktur.
- **Infrastructure as a Service (IaaS):** Beim IaaS Model (Infrastruktur Dienstleistung) werden Verbrauchern grundlegende Rechenressourcen zur Verfügung gestellt. Diese können nun nach Belieben des Verbrauchers verwaltet und kontrolliert werden. Auch bei diesem Servicemodell hat der Verbraucher keinerlei Einfluss auf die, den Rechnerressourcen zugrundeliegenden, Cloud-Infrastruktur. Im Gegensatz zum SaaS Modell und dem PaaS Modell kann er jedoch Betriebssysteme, Firewalls und weitere Funktionalitäten, sowie Netzwerkkomponenten auf Ebene der ihm bereitgestellten Rechnerressourcen verwalten und kontrollieren.

2.2 Webanwendungen

Bei Webanwendungen handelt es in allgemeinen um Software-Anwendungen, die sich über Browser, wie Google Chrome, oder Firefox und somit über das Internet aufrufen und benutzen lassen [Jaz07]. Webanwendungen lassen sich in zwei Teile unterteilen: Dem Backend (Server) und dem Frontend (Client). Das Backend kann hierbei für die Verarbeitung von Daten zuständig sein. Das Frontend hingegen, bietet eine Oberfläche, über die die Nutzer mit der Webanwendung interagieren können [Pav20, S. 49] Die Logik, die hinter einem Frontend steht, wird dabei innerhalb eines Client, beispielsweise dem Browser der Benutzer ausgeführt (client-side). Die Logik des Backend wird im Gegensatz dazu auf Servern und somit serverseitig (server-side) ausgeführt [Jaz07].

2.3 Repository

Philip A. Bernstein definiert in seinem Konferenzbericht von der 20. VLDB-Konferenz in Santiago Chile vom Januar 1994, den Begriff *Repository* wie folgt:

A repository is a shared database of information about engineered artifacts. [BD94, S. 705]

Untermauert wird diese grundlegende Definition des Repositorys von Prof. Dr. Frank Lehmann, der das Repository als spezielle Datenbank beschreibt, die dazu dient, systematisch Modelle und deren Bestandteile zu speichern [Ley18]. Bei einem Repository wird für ein Metadaten Management ein „Layer of control services“ (Ebene von Kontrolldiensten) über ein Speichersystem (Bsp. Datenbanksystem, Festplatten Archiv) implementiert. Diesen Layer of control services nennt man Repository Manager [BD94, S. 705]. Metadaten lassen sich als „Daten von Daten“ bezeichnen, da mit Metadaten gespeicherter Daten beschrieben werden [Kim05, S. 41]. Abbildung 2.1 bildet den strukturellen Zusammenhang zwischen einem Repository Manager, einem Speichersystem (Datenquelle) und einer Businesslogik ab. Es wird verbildlicht, wie sich das Repository als Layer of control services, zwischen Datenquelle und Businesslogik einordnet. Eine Businesslogik (syn. Domainlogik) können Algorithmen oder Regeln sein, die den Austausch von Informationen zwischen einer Datenbank und einer Benutzeroberfläche abwickelt [Fra20].



Abbildung 2.1: Repository Pattern, Quelle: [Ede11]

Bei den, in Bernsteins Definition angesprochenen, Artefakten handelt es sich um Software, Dokumente, Landkarten, Informationssysteme und diskret hergestellte Komponenten und Systeme (z. B. elektronische Schaltkreise, Flugzeuge, Automobile, Industrieanlagen). Zusätzlich lassen sich in einem Repository Informationen zu den tatsächlichen Speicherorten der Artefakte speichern. So kann über das Artefakt die folgenden Dinge gespeichert werden: Eine Revisionshistorie, zu seiner Erstellung verwendeten Werkzeuge und Prozesse, Bedingungen die es erfüllen muss, Zugriffs- und Änderungsberechtigungen, die Zuständigkeit für die Verwaltung des Objekts und seine Abhängigkeiten von anderen Objekten. Mit bestimmten Datenbank Kontrollen, kann das Repository die Integrität, die Parallelität als auch die Zugriffssteuerung der Datenquelle kontrollieren [BD94, S. 706].

2.3.1 Das Repository im Kontext der Informatik

Im Kontext der Informatik steht das Repository eng mit der Technologie des „Source code management“ (SCM, deutsch Quellcode Management) in Verbindung [Bro19, S. 1], [Maj17], [BR15], [UK18].

Das Source Code Management wird zur Steuerung der Zusammenarbeit zwischen Entwicklern genutzt. Es ermöglicht, dass mehrere Entwickler Quellcode Dateien eines Projektes gleichzeitig, gemeinsam und standortunabhängig nutzen und ändern können [UK18]. SCM wird in der Literatur unter anderem auch als „Version controll system“ (VCS, deutsch Versionskontrollsystem) bezeichnet [Maj17].

Die Kombination aus Repository und Versionskontrollsystem wird oft als Set-Up verwendet, um verteilt an gemeinsamen Projekten zu arbeiten. [BR15]

Bei Änderungen an Objekten aus einem Repository, die durch ein VCS kontrolliert werden, wird ein Schnappschuss der Objekte erstellt, um so verschiedene Versionen der Objekte zu erhalten und zu verwalten. [Maj17] So können VCSs jede Version von Objekten eines Projektes speichern [ZND18, S. 408]. VCS, wie beispielsweise Git ermöglichen es, dass Entwickler über eine Kollaborationsplattform standortunabhängig am Quellcode arbeiten können, der an einem zentralen Punkt, beispielsweise einem Haupt-Repository, gespeichert wurde [Maj17]. Eine der wohl bekanntesten Kollaborationsplattform, für eine derartige Zusammenarbeit, ist der webbasierte Code-Sharing-Dienst GitHub [Zag15]. GitHub hostete 2012, über 4,6 Millionen Repositories [Bri14]. Nach Steffan Otte, lassen sich VCSs in zwei verschiedene Ansätze unterteilen [Ott09]:

- *centralized Version Control Systems (CVCS, zentralisierte Versionskontrollsystem)*
- *distributed Version Control Systems (DVCS, verteilte Versionskontrollsystem)*

Majumdar ergänzt diese Aufgliederung um den Ansatz des *local Version Control Systems* [Maj17]. Für diese Arbeit ist jedoch eine tiefere Betrachtung dieses Ansatzes nicht weiter relevant.

Die Tabelle in Abbildung 2.2 zeigt die Charakteristiken beider Systeme im Vergleich. Sie zeigt die Unterschiede beider Systeme im Bezug auf das Verwalten von Repositories und den Zugriff auf Repositories. Zudem sind konkrete Beispiele von Werkzeugen aufgelistet, die auf den unterschiedlichen Systemen basieren. Abschließend listet die Tabelle Merkmale der Systeme auf, die bei Projektarbeiten mit einem der jeweiligen Systeme relevant sein können.

Version Control System	CVCS	DVCS
Repository	There is only one central repository which is the server.	Every user has a complete repository which is called a local repository on their local computer.
Repository Access	Every user who needs to access the repository must be connected via network.	DVCS allows every user to work completely offline. But users need a network to share their repositories with other users.
Example of VCS Tools	Subversion, Perforce Revision Control System	Git, Mercurial, Bazaar, BitKeeper
Software Characteristics that suitable	<ul style="list-style-type: none"> I. Projects that allow only several users to contribute to the software development. II. Team located in a single site. III. Teams located in multiple sites or different countries and different time zones. 	<ul style="list-style-type: none"> I. DVCS is suitable for a single or more developers because the project repository is distributed to all the developers and this ability offer a great improvement for the projects. II. III. It also can be applied for small or big software projects because it makes it less difficult for normal users to contribute to the development.

Abbildung 2.2: Vergleich von CVCS und DVCS, Quelle: [ZND18, S. 409].

2.3.2 Repository Modelle

Kapitel 2.3.1 zeigt den Zusammenhang zwischen Repositories und Versionskontrollsystemen und verdeutlicht wie eng beide Werkzeuge miteinander verstrickt sind. Darauf aufbauend beschäftigt sich dieses Kapitel nun mit verschiedenen Struktur-Modellen für Repositories. Hierbei werden die zwei Ansätze des Multi-Repository Modells und des Mono-Repository Modells betrachtet.

Der grundlegende Unterschied beider Modelle ist, dass Organisationen die Ihre Quellcode-Datenbank auf einem Multi-Repository Modells basieren, für jedes Projekt (bsp. Software-Komponenten, Bibliotheken) ein eigenes Repository anlegen. Organisationen hingegen, die ihre Quellcode-Datenbank nach dem Mono-Repository Modell strukturieren, bringen viele verschiedene Projekte in einem einzigen großen Repository unter. Beide Modelle sind in der Wirtschaft tief verankert. So verwendet der Konzern Amazon oder das Unternehmen Netflix ein Multi-Repository Modell, um ihren Quellcode zu speichern. Andere Konzerne wie Google, Facebook und Microsoft verwenden jeweils ein Mono-Repository Modell [Bro19, S. 2].

Da sich diese Arbeit mit dem gemeinsamen Cloud-Deployment von Frontend und Backend aus einem Repository auseinandersetzt, erfüllt das Modell des Mono-Repository die Anforderung des Titels maßgeschneidert.

Mono-Repository

Projekte, die innerhalb eines Mono-Repositorys abgelegt wurden, können in Verbindung stehen, müssen es aber nicht. In jedem Fall teilen sie sich jedoch die gleichen Abhängigkeiten (Dependencies). Zudem lässt sich zwischen zwei Arten von Mono-Repositories unterscheiden: Dem *monströse Mono-Repository*, welches Millionen Zeilen an Code von verschiedenen Projekten beinhaltet und dem *Projekt Mono-Repository*, welches alle Komponenten eines spezifischen Projektes beinhaltet [BTV18]. Ein nennenswertes Beispiel für ein monströses Mono-Repository, ist das Mono-Repository von Google. Es ist in den 17 Jahren von 1999 bis 2016 zu einer Gesamtgröße von 86TB (des unkomprimierten Inhalts, ohne Versionszweige) gewachsen und umfasst etwa eine Milliarde Dateien. Für Rachel Potvin der Beweis dafür, dass sich Mono-Repositories erfolgreich hoch skalieren lassen. [PL16, S. 78].

Nach Ciera Jaspan bringen Mono-Repositories fünf Charakteristiken mit sich [Jas18]:

- **Centralization:** Der gesamte Quellcode befindet sich in einem Repository.
- **Visibility:** Der gesamte Quellcode innerhalb des Mono-Repositories ist für alle berechtigten Teilnehmer (z.B. Entwickler) einer Organisation einsehbar.
- **Synchronization:** Entwicklungsprozesse sind trunk-basiert.
- **Completeness:** Alle Projekte, oder Komponenten innerhalb des Mono-Repository müssen aus den gleichen Dependencies gebaut (build) werden.
- **Standardization:** Ein gemeinsamer Satz an Werkzeugen ist durch das Mono-Repository festgelegt, mit denen berechnigte Teilnehmer (z.B. Entwickler) einer Organisation innerhalb von Entwicklungsprozessen arbeiten müssen.

Ein Vorteil, den ein Mono-Repository mit sich bringt ist, dass es eine eindeutige Versionierung aller Dateien bietet, die es beinhaltet. So agiert es als *Single Source of Truth* (einzige Quelle der Wahrheit) und es kann bei der Arbeit an Projekten nicht zu Verwirrungen kommen, wo die maßgebliche Version einer Datei gespeichert ist. Ein weitere großer Vorteil von Mono-Repository Strukturen ist, dass es bei Build-Prozessen von Software-Artefakten nicht zu sogenannten *Diamond Dependency Problems* kommen kann. In den meisten Fällen ein unlösbares Problem, was den Build-Prozess unmöglich macht [PL16, S. 84].

Ein Diamond Dependency Problem entsteht, wenn in einem Software-Artefakt zwei oder mehr Software-Bibliotheken verwendet werden, die jeweils selbst von einer dritten Software-Bibliothek abhängig sind, hierbei jedoch jeweils unterschiedliche, inkompatible Versionen

dieser dritten Software-Bibliothek verwenden. [TAM18]

Ein weiterer Vorteil eines Mono-Repositorys ist eine vereinfachte Organisation von Projekten, da diese im Mono-Repository zusammenhängend organisiert sind. Zudem sind durch die monolithische Struktur des Repositorys projektübergreifende Änderungen leichter zu realisieren. Durch eine gemeinsame Hierarchie ist die Erstellung von projektübergreifenden Werkzeugen vereinfacht [BTV18]. Auch auf die Arbeitskultur von Organisationen, haben Mono-Repository Modelle einen großen Einfluss. Da je nach Freigabe-Regelungen einer Organisation, alle Entwickler über Ihren Tätigkeitsbereich hinaus Einsicht auf den gesamten Quellcode des Repositorys der Organisation haben können. So kann eine Team-Kognition entstehen, die zur Entwicklung einer gemeinsamen Vision der Organisation beitragen und die Kommunikation und Zusammenarbeit in der Organisation erleichtern kann. [Bro19, S. 3]

Jedoch bietet ein Mono-Repository Modell auch Herausforderungen. Nennenswerte Herausforderungen sind eine komplexe Verzeichnisstruktur, vor allem bei sehr großen Mono-Repositories, lange Building-Prozesse und die Einschränkungen in der Wahl der verwendeten Entwicklungstechnologien seitens der Entwickler-Teams. [Jas18].

Die Entscheidung hin zu einem Mono-Repository Modell sollte daher in Abhängigkeit von der Organisationskultur getroffen werden. Die Vorteile von Mono-Repository Modellen kommen am besten bei Organisationen mit einer offenen und kollaborativen Kultur zum tragen. Organisationen, bei denen große Teile des Quellcodes in privaten Repositories gespeichert werden und nicht teamübergreifend einsehbar sein soll, ist von einem Mono-Repository Modell abzuraten [PL16, S. 87].

2.4 Cloud-Deployment

Dieses Kapitel beschäftigt sich ausführlich mit dem Cloud-Deployment. In Unterkapitel 2.4.1 werden hierfür technische Grundlagen erklärt, die für ein Cloud-Deployment benötigt werden. Unterkapitel 2.4.2 erklärt anschließend die einzelnen Prozessschritte, die benötigt werden um das Cloud-Deployment mit einer CI/CD-Pipeline zu automatisieren.

2.4.1 Virtualisierung

Um Software-Anwendungen über die Cloud bereitzustellen zu können, gibt es verschiedene Ansätze. Für diese Arbeit sind zwei dieser Ansätze relevant. Der Ansatz, Software-

Anwendungen über sogenannte *Virtual Machines* in der Cloud-Infrastruktur bereitzustellen und der Ansatz, Software-Anwendungen über sogenannte *Container* in der Cloud-Infrastruktur bereitzustellen [Sin19]. Beide Virtualisierungs-Technologien sind im Zusammenhang mit der Cloud essenziell, da die Cloud-Infrastruktur Virtualisierung-Technologien benötigt, um elastisch große Hardware-Ressourcen aufteilen zu können. [Pah15, S. 24] Bei der Virtualisierung wird eine Abstraktion der physischen Hardware zu virtuellen Hardwarekomponenten vorgenommen. Diese virtuellen Hardwarekomponenten lassen sich in derselben Weise nutzen, wie die zugrunde liegende physische Hardware. Durch Hardware-Virtualisierung lassen sich beispielsweise unterschiedliche virtuelle Server erstellen, die sich die gleichen Hardware-Ressourcen teilen, jedoch wie eigenständige Server agieren [FL18].

- **Definition Virtual Machines (VMs):** VMs greifen den beschriebenen Grundgedanken der Hardware-Virtualisierung auf. VMs sind Abstraktionsebenen aus der Struktur eines Softwaresystems. Die Hardware wird hierbei als „reale“ Maschine betrachtet. So ist beispielsweise das Betriebssystem, das eine Hardware initial funktionsfähig macht, die erste Abstraktionsebene eines Softwaresystems, sprich die erste virtuelle Maschine auf der „realen“ Maschine. Eine zweite Abstraktionsebene, die auf dem Betriebssystem aufbaut und dieses als virtuelle Maschine benutzt, könnte beispielsweise eine Programmiersprache sein. Nach diesem Prinzip lassen sich nun viele verschiedene Abstraktionsebene zu einem Softwaresystem zusammenbauen [LS18].
- **Definition Container:** Im Zusammenhang mit Container bezieht sich die Arbeit auf sogenannte Docker-Container. Docker ist eine open-source Plattform, die das Erstellen und Verwalten von Containern ermöglicht. Vereinfacht beinhalten Docker-Container alle erforderlichen Bestandteile, um Software-Anwendungen isoliert ausführen zu können. [BBA17].

Traditionell wurden VMs für die Server-Virtualisierung von Cloud-Infrastrukturen genutzt. Da diese jedoch die Hardware-Ressourcen der „realen“ Maschine initial partitionieren um isolierte Umgebungen zu schaffen, in denen Software-Anwendungen ausgeführt werden können. Kann es zu ineffizienter Nutzung der Hardware-Ressourcen kommen. Bei Docker Container werden im Gegensatz hierzu die Hardware-Ressourcen nicht initial, also im Vorfeld partitioniert, sondern erst zur Laufzeit der Software-Anwendung zugewiesen [BRM21]. Abbildung 2.3 zeigt die grundlegenden Unterschiede beider Virtualisierungsansätze.

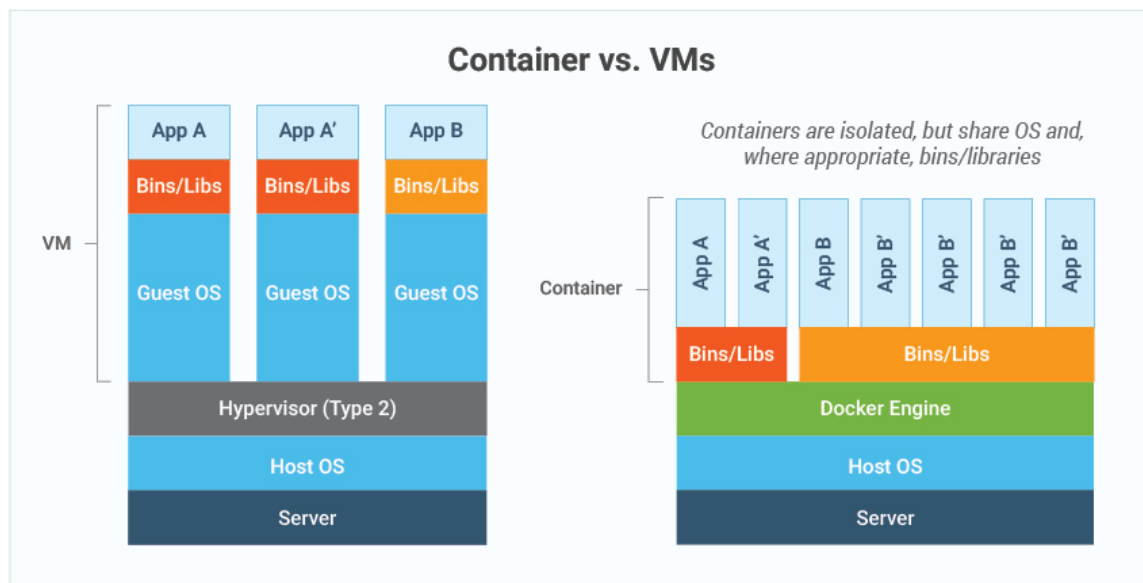


Abbildung 2.3: Virtuelle Maschine im Vergleich zu Containern, Quelle: [War22]

Bei den untersten Ebenen, „Server“ und „Host OS“, gibt es keinen Unterschied beider Ansätze. Die Ebene „Server“ stellt hierbei die Hardware dar und die Ebene „Host OS“ das Betriebssystem, das vom Cloud-Anbieter auf die Hardware aufgespielt wurde. Bei der Virtualisierung mit VMs baut auf der Ebene des Host OS die Ebene des „Hypervisor (Type 2)“ auf. Der Hypervisor (Type 2) ist hierbei eine Software, die für die Partition der Hardware-Ressourcen zuständig ist [BRM21]. Oberhalb der Ebene des „Hypervisor (Type 2)“ sind drei identische VMs abgebildet. Die Ebene „Guest OS“ stellt das Betriebssystem dar, das in den einzelnen VMs läuft. Die Ebene „Bins/Libs“ stellt die verschiedenen *Bins* und *Libaries* dar, die in den VMs vorhanden sind. Mit der Ebene „Apps“ werden die verschiedenen Software-Anwendungen (Applications, Apps) abgebildet, die in den VMs laufen. App A' stellt hierbei eine neue Version der App A dar.

Beim Ansatz mit Containern folgt anstelle der Ebene des „Hypervisor (Type 2)“, die Ebene der „Docker Engine“. Darüber abgebildet ist eine Vielzahl von Containern zu sehen. Jeder Container beinhaltet seine eigene App. Wobei auffällt, dass sich im Gegensatz zu den VMs, mehrere Container die gleichen *Bins* und *Libaries* teilen. Die Ebene der Docker Enging ist dafür zuständig, dass die einzelnen Apps in den Docker-Containern virtualisiert und ausführbar gemacht werden. Ein weiter großer Unterschied beider Ansätze ist, dass Docker-Container kein eigenes Betriebssystem (Guest OS) beinhalten, was Auswirkungen auf die Leistung hat. [BBA17]

Die in Abbildung 2.4 dargestellte Tabelle, vergleicht diese Leistungsunterschiede und listet zusätzlich noch strukturelle Unterschiede auf. Abschließend lässt sich sagen, dass

die Virtualisierung von Cloud-Infrastruktur über Container-Technologien, stark auf die Bereitstellung von Software abzielen. Daher ist diese Art der Virtualisierung eher auf das Cloud-Service-Modell PaaS ausgelegt. Der Schwerpunkt der Virtualisierung durch VMs liegt hingegen bei der Zuweisung und Verwaltung von Hardware-Ressourcen. Weshalb sie eher zu dem Cloud-Service-Model IaaS passen. Wobei Container-Technologien durch Ihre Eigenschaft der Isolation und der gemeinsamen Nutzung von Hardware-Ressourcen, ebenfalls für IaaS in Betracht gezogen werden können [Pah15, S. 24].

	Virtual Machines	Docker Container
Isolation Process Level	Hardware	Operation System
Operation System	Separated	Shared
Boot up time	Long	Short
Resources usage	More	Less
Pre-built Images	Hard to find and manage	Already available for home server
Customized preconfigured images	Hard to build	Easy to build
Size	Bigger because they contain whole OS underneath	Smaller with only docker engines over the host OS
Mobility	Easy to move to a new host OS	Destroyed and recreated instead of moving
Creation time	Longer	Within seconds

Abbildung 2.4: Leistungsunterschiede, Virtual Machines zu Docker Container, Quelle: [Pot20, S. 1422]

2.4.2 CI/CD-Pipelines

Der Deployment-Prozess kann sehr komplex und zeitaufwendig sein und sich trotz Teilautomatisierung über mehrere Tage ziehen. Es wird dennoch dazu geraten, den Deployment-Prozess im frühen Stadium der Entwicklungsphase zu automatisieren, da ein automatisierter Prozess Arbeitszeit spart und dabei Hilft, frühzeitig Probleme im Prozess zu entdecken [HRN06].

Eine Herangehensweise, um eine derartige Automatisierung umzusetzen, ist die Implementierung von CI/CD-Pipelines [Zam21, S. 471]. Eine CI/CD-Pipeline kann in unterschiedlichen Teilprozesse unterteilt werden, die unterschiedlichen Prozessschritten ablaufen [Ran20]:

- **Continuous Integration (CI)** setzt nach dem Übertragen von Quellcode-Änderungen in ein Repository ein. Diese Übertragung der Quellcode-Änderungen hat das automatische Ausführen von *Unit-Test* und einen automatisch ausgeführten *Build* der

neuen Software-Anwendung zufolge. CI umfasst als Teilprozess der Pipeline, die *Build* und *Test-Zyklen*.

- **Continuous Delivery (CD)** führt, darauf aufbauend, automatisierte Akzeptanztests und Abnahmetests durch. Diese überprüfen, ob die neuen Änderungen an der Software-Anwendung keine der bereits bestehenden Funktionalität der Software-Anwendung negativ beeinträchtigt. Je nach Automatisierungsgrad kann diese Phase auch manuelle Test beinhalten, bevor die Software-Anwendung deployed wird.
- **Continuous Deployment (CD)** deployed die neue Version der Software-Anwendung abschließend automatisiert auf die Zielumgebung, die von einer Cloud-Infrastruktur bereitgestellt wird.

Obwohl CD entweder für „Continuous Delivery“, oder für „Continuous Deployment“ stehen kann und sich beide Teilprozesse voneinander unterscheiden, werden sie in vielen Studien als Synonyme verwendet [Sha17]. Grundlegend fungiert das Continuous Delivery als eine Art Qualitätskontrolle und stellt sicher das die Software-Anwendung, für ein anschließendes Deployment bereit sind. Continuous Deployment hingegen geht einen Schritt weiter und ergänzt die Qualitätskontrolle, durch ein automatisiertes Deployment. [WNZ16, S. 75]. Abbildung 2.5 verbildlicht die unterschiedlichen Phasen einer CI/CD-Pipeline. Zudem wird verdeutlicht, inwiefern das Continuous Deployment, das Continuous Delivery um die Automatisierung im Deployment ergänzt. Es ist zu erwähnen, dass es sich beim Prozessschritt „Deploy to staging“ um das bereitstellen der Anwendung auf eine Testumgebung in der Cloud-Infrastruktur handelt.

Eine sogenannte *Staging-Umgebung* kann eine fast identische Kopie der finalen Produktions-Umgebung sein, die für das Testen von Software-Anwendung genutzt wird [Gil18]. Das Implementieren einer CI/CD-Pipeline bringt viele verschiedene Vorteile mit sich. Drei der wichtigsten Vorteile sind: Frühzeitige Fehlererkennung, eine höhere Produktivität und schnellere Veröffentlichung Zyklen bei Software-Updates [Zam21, S. 471].

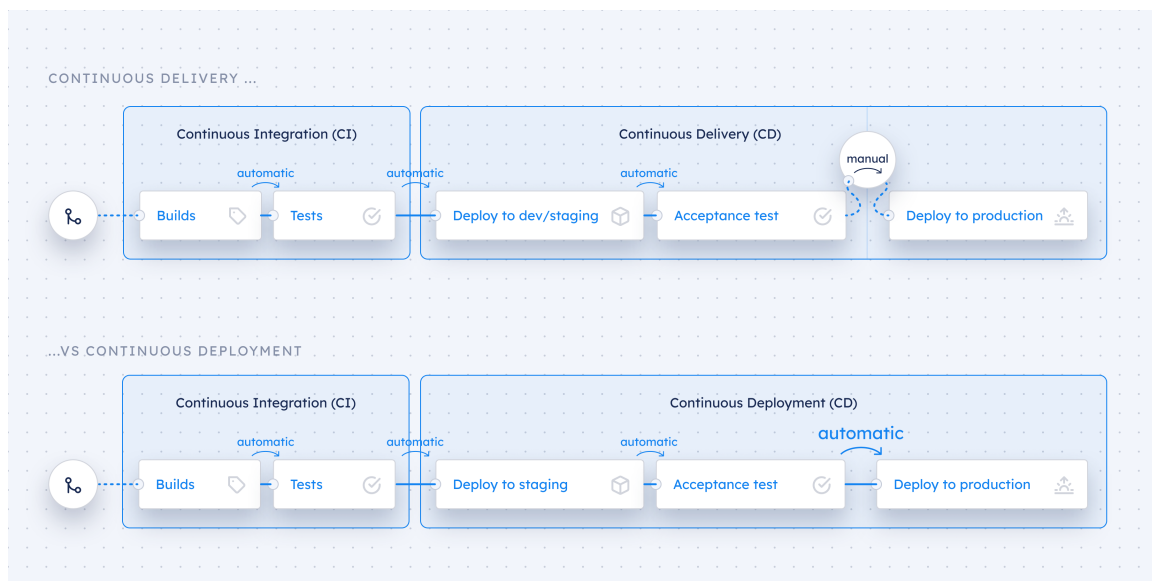


Abbildung 2.5: Continuous Delivery und Continuous Deployment im Vergleich, Quelle: [War20]

2.5 Verwandte Arbeiten

Der Deployment-Prozess lässt sich als eine der wichtigsten Kernprozesse des gesamten Software-Lebenszyklus bezeichnen. Denn erst durch die erfolgreiche Bereitstellung von Software, wird diese für den Verbraucher nutzbar [BRM21]. Die Implementierung von CI/CD-Pipelines für automatisierte Deployment-Prozesse, wird von einer wachsenden Anzahl an Unternehmen umgesetzt [Sha17].

Obwohl Mono-Repositories in großen Konzernen Anwendung finden und dies im Falle von Google schon seit Jahrzehnten (siehe Kapitel 2.3.2), ist der Diskurs über Mono-Repositories ein frischer. Daher gibt es zu diesem Thema noch keine breite Palette an akademischen Forschungsergebnissen [BTV18]. Auch bei der Recherche zu dieser Arbeit wurde nur eine verwandte Arbeit gefunden, in der vergleichbare Themen behandelt wurden und sich mit dem Deployment via CI/CD-Pipeline aus einem Mono-Repository beschäftigt wurde.

Iván Kubinyi analysiert in seiner Masterarbeit aus dem Jahr 2021 einen existierenden Deployment-Prozess, bei dem mit Hilfe einer CI/CD-Pipeline, Plugins für ein Software configuration management, aus einem Mono-Repository deployed werden. Zielstellung seiner Arbeit ist die Ausarbeitung von Verbesserungen am existierenden Prozess. Das Software configuration management und die Plugins, wurden auf AWS deployed. Als Kollaborationsplattform und zum hosten des Mono-Repositorys wurde GitHub verwendet. Git

wurde als Versionkontrollsystem genutzt. Bei Kubinyis Analyse stellte sich heraus, dass eine große Herausforderung beim Deployment aus einem Mono-Repository das Managen der Build-Prozesse ist. So werden diese im analysierten System über Bash-Skripte gemanaged. Diese Bash-Skripte ermöglichten es, Building-Prozesse nur für Plugins auszuführen, die durch Bearbeitung von konkreten Quellcodeänderungen betroffen waren. Jedoch kamen die Bash-Skripte schnell an ein Limit. Wenn beispielsweise ein Plugin bearbeitet wurde, von dem weitere Plugins abhängig waren, wurde beim Re-Build des überarbeiteten Plugins, alle Plugins im Mono-Repository neu gebildet und nicht nur die in Abhängigkeit stehenden. Kubinyis kommt zu dem Fazit, dass es bei einem Mono-Repository schwierig ist herauszufinden welche Teile des Quellcodes von Änderungen betroffen sind und wie der Quellcode miteinander in Verbindung steht, um zeitaufwändige Buildprozesse zu optimieren. [Kub21].

Eine Schlussfolgerung, die sich mit den in Kapitel 2.3.2 beschriebenen Herausforderungen eines Mono-Repository Modells vereinbaren lässt. Diese Arbeit versucht an Kubinyis Arbeit anzuknüpfen und wird mit der Erstellung des Prototyps einen Lösungsansatz für das effektive Management von Build-Prozessen und Dependencies (Abhängigkeiten) ausarbeiten. Zudem lässt sich wie oben beschrieben behaupten, dass die Themen die diese Arbeit behandelt allesamt sehr relevant im Kosmos der Informatik sind und dass die Ausarbeitung eines Prototyps, wie in Kapitel 1 vorausgesetzt, ein guter Beitrag sein kann, der die schmale Palette an akademischen Forschungsergebnissen im Bereich Mono-Repositorys, als auch die anderen Themenbereiche bereichern kann.

Kapitel 3

Requirements Engineering

Das Requirements Engineering beschäftigt sich im Allgemeinen mit dem Erfassen, Analysieren, Entwickeln, Strukturieren und Dokumentieren von Anforderungen an ein zu entwickelndes Software-Produkt [Bro07, S. 130].

Für die Umsetzung des Requirements Engineerings ist dieses Kapitel in zwei Unterkapitel aufgeteilt. In Kapitel 3.1 werden alle Begriffe erklärt, die für das Requirements Engineering relevant sind. Kapitel 3.2 erarbeitete darauf aufbauend eine Methodik, auf deren Basis einzelne Arbeitsschritte zur Umsetzung des Requirements Engineering definiert werden. Es ist zu betonen, dass sich die Aussagen die in den folgenden zwei Kapiteln getroffen werden, hauptsächlich auf drei Sachbücher stützen. Genauer, auf Helmut Balzerts 2009 erschienene dritte Auflage seines Lehrbuchs der Softwaretechniken: „Basiskonzepte und Requirments Engineering“, auf Chris Rupps 2007 erschienene vierte Auflage seines Sachbuches: „Requirements Engineering und -Management“ und auf Klaus Pohls ebenfalls 2007 erschienene erste Auflage seines Sachbuches: „Requirements engineering“.

Um die Nachvollziehbarkeit der folgenden Aussagen zu erleichtern, wurde diese mit den jeweiligen, zugehörigen Seitenzahlen zu gekennzeichnet.

3.1 Begriffe aus dem Requirements Engineering

Helmut Balzert definiert den Begriff „Anforderungen“ wie folgt:

Anforderungen (requirements) legen fest, was man von einem Softwaresystem als Eigenschaften erwartet [Bal09, S. 453].

Wobei es sich bei „man“ um sogenannten Stakeholder handelt. Die genannten „Eigenschaften“ lassen sich über einen Verbund aus den folgenden Punkten herausarbeiten [Bal09, S. 455]:

- **Versionen und Ziele** die an das Softwaresystem gestellt werden. Sie sollten vor der Ausarbeitung von Anforderungen festgelegt werden [Bal09, S. 456–459].
- Gegebene organisatorische und technische **Rahmenbedingungen**, die das Softwaresystem und den Entwicklungsprozess beschränken. Organisatorische Rahmenbedingungen legen den Anwendungsbereich des Softwaresystems fest und für welche Zielgruppe dieses entwickelt wird. Technische Rahmenbedingungen legen Software, Hardware und Orgware-Grundlagen fest, mit denen das zu entwickelnde Softwaresystem arbeiten muss [Bal09, S. 459–460].
- Der **Kontext**, in dem sich das Softwaresystem in seine materielle Umgebung (z.B. Sensoren, Personen) und immaterielle Umgebung (z.B. Software Schnittstellen, Internet) eingliedert [Bal09, S. 461].

Ein Beispiel für einen derartigen Verbund findet sich in Kapitel 3.2.3 wieder. Anforderungen an ein Softwaresystem lassen sich in funktional Anforderungen und nicht funktional Anforderungen aufgeteilt [Bal09, S. 456].

- **Funktionale Anforderungen** legen vom Softwaresystem bereitgestellte Services oder Funktionen fest [Bal09, S. 456]. Sie definieren die Statik (Struktur), die Dynamik (Verhalten) und die Logik des Softwaresystems [Bal09, S. 99]. Abgekürzt legen sie somit fest, „was“ ein Softwaresystem tun soll [Bal09, S. 465].
- **Nicht funktionale Anforderungen** lassen sich in Qualitätsanforderungen oder unterspezifizierte funktionale Anforderungen unterteilen [Poh07, S. 16]. Sie beschreiben meistens Anforderungen, die alle oder viele funktionalen Anforderungen betreffen [Bal09, S. 463]. Verallgemeinert beschreiben nicht funktionale Anforderungen, „wie gut“ Services oder Funktion seine technischen Anforderungen umsetzen soll [Bal09, S. 465]. Wobei zu erwähnen ist, dass sich unterspezifizierte funktionale Anforderungen, durch eine detailliertere Spezifizierung entweder in funktionale Anforderungen, oder in Qualitätsanforderungen umwandeln lassen [Bal09, S. 456].

Häufig werden Anforderungen in natürlicher Sprache geschrieben. Dies erhöht die Verständlichkeit und es erleichtert abstrakte Zusammenhänge darzustellen [Bal09, S. 481]. Eine Herangehensweise, um qualitativ hochwertige Anforderungen in natürlicher Sprache

zu schreiben, geht über das Benutzen von syntaktischen Anforderungsschablonen (Templates). Diese haben die Funktion, klare Regeln vorzugeben, wie Anforderungen formulieren werden müssen [Rup07, S. 227-228].

Zur Unterstützung der Anforderungsanalyse, sollten Anforderungen zudem mit Anforderungsattributen ausgestattet werden. Diese helfen bei der Verwaltung und beim Verständnis von Anforderungen. Beispiele für derartige Attribute könnten der Anforderungstyp (funktional, nicht funktional) und die Anforderungs ID (eindeutige Nummer) sein. Zudem existieren genormte Qualitätsstandards, auf die Anforderungen überprüft werden können (siehe Kapitel 3.2.4). [IEE18].

Um zu überprüfen, ob das implementierte Softwaresystem die gestellten Anforderungen erfüllt hat, lassen sich für Anforderungen Abnahmekriterien festzulegen [Poh07, S. 224–225]. Diese können ähnlich wie Anforderungen in natürlicher Sprache geschrieben werden [Rup07, S. 328-330].

3.2 Definition und Umsetzung des Requirements Engineering

Dieses Kapitel ist wie folgt aufgebaut. In Unterkapitel 3.2.1 wird eine theoretische Methodik definiert, anhand der konkrete Arbeitsschritte abgeleitet werden können, die für das Umsetzen eines Requirements Engineering benötigt werden. Die folgenden Unterkapitel setzen diese Arbeitsschritte um und präsentieren die erzeugten Ergebnisse.

3.2.1 Theoretische Methodik zur Durchführung des Requirements Engineering

Für die Erarbeitung einer theoretischen Methodik zur Durchführung des Requirements Engineering orientiert sich diese Arbeit an den von Balzert festgelegten Kernaktivitäten, die in einem Requirements Engineering durchgeführt werden sollten [Bal09, S. 443–444]. Ergänzt mit dem, in Kapitel 3.1 erzeugten Wissen, lassen sich die folgenden konkreten Arbeitsschritten ableiten:

- **Feststellung der Stakeholder, der Projektumgebung und einer Ermittlungstechnik zur Erhebung von Anforderungen:** Das Requirements Engineering

startet mit dem Definieren der Stakeholder und dem abstecken der Projektumgebung. Die Projektumgebung besteht hierbei aus allen Elementen, die ein Projekt von außen beeinflussen oder einschränken können. Stakeholder stellen alle Personen dar, die von dem entwickelten System betroffen sind, es nutzen oder Interesse daran haben. Nach der Identifizierung der Stakeholder und erfolgreichem Abstecken der Projektumgebung, können mit Hilfe von Ermittlungstechniken die Anforderungen der Stakeholder ermittelt werden [Bal09, S. 500–509].

- **Erstellung einer Anforderungsschablone zur Erhebung von Anforderungen:** Das Erstellen einer Anforderungsschablone vor der Erhebung von Anforderungen erhöht die Qualität der Anforderungserhebung [Rup07, S. 227]. Anhand der festgelegten Ermittlungstechniken können nun die Anforderungen der Stakeholder, mit Hilfe der Anforderungsschablone erhoben werden.
- **Spezifizierung, Validierung, Priorisierung und Qualitätskontrolle der erhobenen Anforderungen:** Bei der Anforderungsspezifikation werden für die erhobenen Anforderungen, Anforderungsattribute festgelegt [Bal09, S. 509]. Die Validierung überprüft, ob die spezifizierten Anforderungen im Sinne der Visionen und der Ziele dienlich sind. Validierte Anforderungen, können formal abgenommen werden und es kann eine Priorisierung der einzelnen Anforderungen durchgeführt werden [Bal09, S. 513–514]. Für die Qualitätskontrolle werden die erhobenen Anforderungen mit den Qualitätsmerkmalen abgeglichen, die im ISO/IEC/IEEE 29148:2018 definiert wurden [IEE18, S. 10–11].
- **Erhebung von Abnahmekriterien:** Es handelt sich bei Abnahmekriterien um Tests, mit denen das Erfüllen einer Anforderung bestätigt werden kann. Abnahmekriterien können daher genutzt werden, um den implementierten Prototyp zu evaluieren [Rup07, S. 324]. In Kapitel 5 Evaluation wird diese Evaluation des Prototyps mit Hilfe der erhobenen Abnahmekriterien durchgeführt.
- **Anforderungen modellieren:** Die Modellierung der Anforderungen ist der erste Schritt zur Erarbeitung einer fachlichen Lösung. Diese lässt sich unter anderem mit der Modellierungssprache „Unified Modeling Language“ (UML) modellieren [Bal09, S. 547].

3.2.2 Feststellung der Stakeholder, der Projektumgebung und einer Ermittlungstechnik zur Erhebung von Anforderungen

Da es sich bei dieser Arbeit um eine Einzelarbeit handelt, die nicht in Zusammenarbeit mit einer Organisation geschrieben wurde, lassen sich zwei Einschränkungen des Projekts durch die Projektumgebung feststellen.

- Der Autor dieser Arbeit ist der einzige Stakeholder für den zu implementierenden Prototyp.
- Aus der Recherche nach standardisierten Ermittlungstechniken zur Erhebung von Anforderungen, hat sich ergeben, dass die meisten auf eine Gruppe von mehreren Stakeholder abzielen und nicht im Rahmen dieser Arbeit verwendbar sind. [Rup07, S. 115-134].

Um diesen Einschränkungen etwas entgegenzuwirken, wurden den Erwartungen die der Autor an den Prototypen hat, verschiedenen fiktiven Rollen zugeordnet. So sollen sich verschiedene Stakeholder simulieren lassen. Die Tabelle in Abbildung 3.1, bildet die Liste der simulierten Stakeholder ab. Als Ermittlungstechnik werden Anforderungen mit Hilfe einer syntaktischen Anforderungsschablone erhoben. Hierbei werden einzelne Anforderungen jeweils einem der simulierten Stakeholder zugewiesen. Dies soll eine spätere Priorisierung der einzelnen Anforderungen erleichtern.

Stakeholder	Erwartung	Einfluss
Projektleiter	Interesse an einem vollständigen und funktionsfähigen Prototypen.	5
DevOps-Team	Erhofft sich automatisierte, vereinfachte Deployment-Prozess	3
Entwickler	Wünscht sich schnelle automatisierte Testing und Building-Prozesse	2
Legende		5 = hoch 1 = niedrig

Abbildung 3.1: Liste aller Stakeholder, in Anlehnung an [Bal09, S. 504]

3.2.3 Erstellung einer Anforderungsschablone zur Erhebung von Anforderungen

Syntaktische Anforderungsschablonen (Templates) können aus nur sechs konkreten Bestandteilen bestehen. Sie bilden das Regelwerk und die Basis für das Schreiben von

Anforderungen in natürlicher Sprache [Rup07, S. 228]. Abbildung 3.2 zeigt die von Rupp definierte Vorlage einer syntaktischen Anforderungsschablone [Rup07, S. 234].

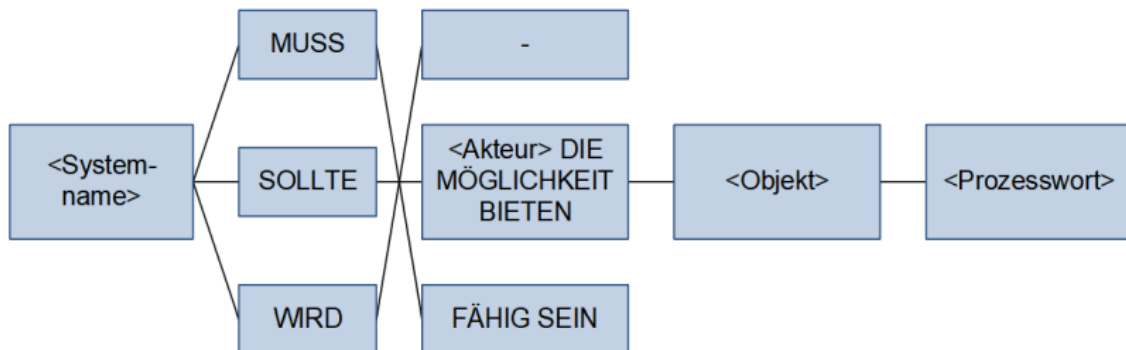


Abbildung 3.2: Vollständige Anforderungsschablone, Quelle: [Pet21]

Eine Beispielanforderung, die nach dieser Schablone formuliert wurde, sieht wie folgt aus:

Falls auf dem Mobilteil die Option „Rechnung erwünscht ausgewählt“ ist, muss das System dem Rezeptionisten die Möglichkeit bieten, eine Rechnung auf dem Netzwerkdrucker zu drucken [Rup07, S. 233].

Der erste Teilsatz der Beispielanforderung stellt eine Bedingung dar, die für die Anforderung eintritt (Bestandteil 1). Im zweiten Teilsatz wird durch die Schablone die Verbindlichkeit der Anforderungen an das System festgelegt (Bestandteil 2,3,4). Der dritte und letzte Teilsatz der Anforderung definiert das Objekt, das am System beteiligt ist und welcher Prozessschritt (drucken) durch das System ausgeführt werden soll (Bestandteil 5,6). Bei diesen Beispiel kommen die bereits beschriebenen Begriffe „Rahmenbedingungen“ und „Kontext“, aus dem Kapitel 3.1 zum tragen. So könnten die Rezeptionisten und die Schnittstelle, zwischen Mobilteil und Drucker, als „Kontext“ interpretiert werden, in dem sich das System einfindet. Die Hardware und Software beider Endgeräte könnten als „Rahmenbedingungen“ interpretieren werden.

Bei der Schablone, die in dieser Arbeit verwendet wurde, wurde der Begriff „das System“ zu „der Prototyp“ geändert. Zur Einhaltung der vorgeschriebenen Syntax der Schablone, wurde ein Google-Docs Dokument erstellt, in das eine Tabelle eingefügt wurde. Jede Spalte steht hierbei für einen einzelnen Bestandteil der Schablone. Zudem wurden für Bestandteil 2 und Bestandteil 4 Drop-Down-Menüs eingefügt, um keinen Platz für Interpretationsspielraum zu lassen. Abbildung 3.3 zeigt die erhobenen Anforderungen auf Basis der erstellten Schablone.

Wann? Unter welchen Bedingungen?				Objekt & Ergänzungen des Objekts	Prozesswort
Für das Erstellen des Repositories	Muss ▾	der Prototyp	- ▾	Nx als Technologie	verwenden.
Für die Dateien und Ordner Struktur des Repositories	Soll ▾	der Prototyp	- ▾	eine von bereitgestellte Nx Vorlage	verwenden.
Für das Erstellen des Backends	Muss ▾	der Prototyp	- ▾	NestJS als Technologie	verwenden.
Für die Containerisierung der Anwendung, für das spätere Deployment	Muss ▾	der Prototyp	- ▾	Docker als Technologie	verwenden.
Für das Speichern von Docker-Images in <u>DockerHub</u>	Muss ▾	der Prototyp	fähig sein ▾	Docker Images nach <u>DockerHub</u>	automatisch zu pushen.
Für die Implementierung des Backends	Soll ▾	der Prototyp	- ▾	eine, von Nx bereitgestellte, <u>NestJS</u> Vorlage	verwenden.
Daten aus dem Backend	Muss ▾	der Prototyp	- ▾	innerhalb eines Services	erheben.
Bei der Bereitstellung von Daten aus dem Backend an das Frontend	Soll ▾	der Prototyp	fähig sein ▾	REST-Calls zu	verarbeiten.
Für das Deployen der Anwendung	Wird ▾	der Prototyp	fähig sein ▾	von Frontend und Backend Docker Images zu	erstellen.
Für das lokale Testen von Docker Images	Soll ▾	der Prototyp	- ▾	eine Docker-Compose.yml	beinhalten.
Für das Erstellen des Frontends	Muss ▾	der Prototyp	- ▾	AngularJS als Technologie	verwenden.
Für die Implementierung des Frontend	Soll ▾	der Prototyp	- ▾	eine von bereitgestellte Nx AngularJS Vorlage	verwenden.
Backend-Code und Frontend-Code	Wird ▾	der Prototyp	- ▾	in einem gemeinsamen Nx Repository	speichern.
Backend-Code und Frontend-Code	Soll ▾	der Prototyp	- ▾	in unterschiedlichen Ordnern, innerhalb des Nx Repositories	speichern.
Für Backend-Code und Frontend-Code	Soll ▾	der Prototyp	- ▾	in einem gemeinsamen Nx Repository die Dependencies	verwalten.
Für das remote Hosting des Nx Repository	Muss ▾	der Prototyp	- ▾	die Hosting-Plattform GitHub	verwenden.
Als VCS-System	Muss ▾	der Prototyp	- ▾	git als Technologie	verwenden.
Als Standard-Branch, für Pull-Request	Soll ▾	der Prototyp	Entwick... ▾	von einem Develop-Branch Quellcode in lokale Branches	zu pullen
Als release-Branch	Soll ▾	der Prototyp	DevOps... ▾	von einem Main-Branch	zu deployen
Um zu verhindern, dass auf Release-Branch und auf Develop-Branch gepusht werden kann	Muss ▾	der Prototyp	- ▾	beide Branches mit Einschränkungen	versehen
Nach einem push auf von einem lokalen Branch zu einem remote Branch,	Muss ▾	der Prototyp	fähig sein ▾	linting, testing (Unit-Test) und building nur für den modifizierten Quellcode	automatisch durchzuführen.
Beim Linting, Testing und Building	Soll ▾	der Prototyp	Entwick... ▾	den gesamten, oder nur den modifizierten Quellcode, des Repositories	zu verwenden
Falls ein push auf den remote Branch durchgeführt wurde,	Muss ▾	der Prototyp	fähig sein ▾	Status-Benachrichtigungen	automatisch zu senden
Wenn ein Pull Request auf den Develop Branch geschlossen wird,	Soll ▾	der Prototyp		ein Deployment der Quellcode-Änderung auf die Testumgebung	automatisch durchzuführen
Wenn ein Pull Request auf den Develop Branch geschlossen wird,	Soll ▾	der Prototyp	fähig sein ▾	linting, testing und building des gesamten Quellcodes	durchzuführen
Wenn ein Pull Request auf den Main Branch geschlossen wird,	Soll ▾	der Prototyp	fähig sein ▾	ein Deployment der Quellcode-Änderung auf die Produktionsumgebung	automatisch durchzuführen
Als Cloud-Plattform	Muss ▾	der Prototyp	- ▾	Heroku als Technologie	verwenden
Für eine Testumgebung und Produktionsumgebung	Muss ▾	der Prototyp	- ▾	verschiedene Heroku-Apps	bereitstellen
Das Backend und Frontend	Wird ▾	der Prototyp	- ▾	in unterschiedlichen Testumgebungen und Produktionsumgebungen	deployen
Für das Erstellen einer CI/CD-Pipeline	Muss ▾	der Prototyp	- ▾	GitHub Actions als Technologie	verwenden
Die Geschwindigkeit in den Build-Prozessen	Soll ▾	der Prototyp	- ▾	durch das Verwenden von Nx als Building-System	beschleunigen

Abbildung 3.3: Anforderungsschablone mit erhobenen Anforderungen

3.2.4 Spezifizierung, Validierung, Priorisierung und Qualitätskontrolle der erhobenen Anforderungen

Für die Erhebung von Anforderungsattributen und die Erhebung von Qualitätsmerkmalen orientiert sich diese Arbeit am ISO/IEC/IEEE 29148:2018 Standard. Dieser definiert eine Liste von Beispielattributen, die zur Spezifizierung einzelner Anforderungen verwendet werden können [IEE18, S. 14–15]. Die in Abbildung 3.4 abgebildete Tabelle listet alle erhobenen Anforderungsattribute auf, beschreibt diese und zeigt, wie sie in Verbindung zu benannten Beispielattributen stehen.

Neben den Beispielattributen wird im ISO/IEC/IEEE 29148:2018 Standard eine Aufzählung von Qualitätsmerkmalen definiert, die für eine Qualitätskontrolle einzelner Anforderungen verwendet werden kann [IEE18, S. 10]. Die Tabelle in Abbildung 3.5 listet diese Qualitätsmerkmale auf und übersetzt sie ins Deutsche.

Priorisiert werden die Anforderungen durch die von Balzert beschriebene Top-Ten-Methode [Bal09, S. 545]. Hierbei werden eine Anzahl von n Anforderungen in eine Rangfolge sortiert. In dieser Arbeit werden für die Priorisierung zwei Variablen betrachtet, aus denen eine dritte Variable x für Wert der Priorisierung abgeleitet wird. Variable a steht für den Wert „Einfluss“ eines Stakeholders, der in Tabelle 3.1 festgelegt wurde und kann einen der drei eingetragenen Wert einnehmen. Variable b ist abhängig vom „Typ“ der Anforderung (siehe Tabelle 3.4). Handelt es sich bei einer Anforderung um eine Anforderung vom „Typ = nicht funktional“, dann nimmt Variable b den Wert -2 an. Handelt es um eine Anforderung vom „Typ = funktional“, dann nimmt Variable b den Wert 3 an. Um den Wert für die Priorisierung zu ermitteln, wird Variable a mit Variable b addiert.

$$\textit{Priorisierungs Formel} : a + b = x \quad (3.1)$$

Durch diese Herangehensweise kann garantiert werden, dass funktionale Anforderungen stets über nicht funktionale Anforderungen priorisieren werden. Ziel dieser Arbeit ist die Umsetzung eines technischen Prototyps und hierbei sollen die Funktion, also das „was der Prototyp macht“ im Mittelpunkt stehen. „Wie gut“ der Prototyp die funktionalen Anforderungen umsetzt soll zweitrangig bleiben. Durch die Priorisierungsrechnung wird ermöglicht das eine funktionale Anforderung eines Entwicklers, trotz geringeren „Einfluss“ höher priorisiert wird, als eine nicht funktionale Anforderung eines Projektleiter.

$$\textit{Funktionale Anforderung des Entwicklers} : 2 + 3 = 5 = \textit{Priorisierungswert} \quad (3.2)$$

Nicht funktionale Anforderung des Projektleiters : $5 + (-2) = 3 = \text{Priorisierungswert}$
(3.3)

Im Folgenden wurden sechs Arbeitsschritte durchgeführt, um die in Kapitel 3.2.3 erhobenen Anforderungen zu spezifizieren, validieren, priorisieren und eine Qualitätskontrolle durchzuführen:

1. Es wurde eine neue Google-Docs Tabelle erstellt, bei der die einzelnen Spalten für jeweils eins der definierten Attribute verwendet wurden. Anschließend wurden alle erhobenen Anforderungen aus Abbildung 3.3 in die neue Tabelle übertragen.
2. Eine Validierung der erhobenen Anforderungen wurde im Sinne der in Kapitel 3.2.1 beschriebenen Vorgehensweise durchgeführt. Es ist zu erwähnen, dass die Validierung der Anforderungen durch eine subjektive Einschätzung des Autors vorgenommen wurde. Anforderungen, die erfolgreich validiert wurden, weisen somit das erste Qualitätsmerkmal aus Tabelle 3.5 auf. Anforderungen, die nicht erfolgreich validiert wurden, wurden aus der Tabelle entfernt.
3. Jede Anforderung wurde bezüglich der Qualitätsmerkmale 2, 3 und 4 überprüft (siehe Tabelle 3.5). Hierbei zahlte sich das Arbeiten mit der syntaktischen Anforderungsschablone aus, da diese die Erfüllung der Qualitätsmerkmale an die erhobenen Anforderungen nahezu erzwangen. Qualitätsmerkmal 5 wurde durch das Erheben von Abnahmekriterien in Kapitel 3.2.5 erreicht.
4. Allen Anforderungen wurden Stakeholder zugeordnet. Auch dies geschah durch eine subjektive Einschätzung des Autors.
5. Den einzelnen Anforderungen wurde ein „Typ“ zugewiesen. Bei der Zuweisung des „Typ“ wurde darauf geachtet, in welchem Bezug verwendete Adjektive zum Inhalt der Anforderung stehen. Waren keine Adjektive vorhanden oder bezogen sich die vorhandenen Adjektive auf den „Prozessschritt“ (siehe Abbildung 3.2) und hatten einen technischen Bezug, wurden die Anforderungen mit dem „Typ“ funktional versehen. Anforderungen die diese Bedingung nicht erfüllten wurden als nicht funktional versehen.
6. Abschließend wurden alle Anforderungen priorisiert, die Tabelle nach der Priorisierung umstrukturiert und den einzelnen Anforderungen wurden, von oben nach unten aufzählend, „IDs“ zugewiesen.

Abbildung 3.6 zeigt das Ergebnis der durchgeführten Arbeitsschritte.

Anforderungs-attribut	ISO/IEC/IEEE 29148	Beschreibung
ID	<i>Identification</i>	Nummer zur eindeutigen Identifikation einzelner Anforderungen
Typ	<i>Type</i>	Ob einer Anforderung funktional, oder nicht funktional ist
Priorisierung	<i>Stakeholder Priority</i>	Zeigt an welche Attribute vorrangig behandelt
Rolle		Einer der simulierte Stakeholder
Validiert		Ja/Nein, ob Anforderung gegenüber der Zielstellung dieser Arbeit validiert wurde
Abgenommen		Ja/Nein, ob Anforderung Qualitätsmerkmal 2, 3 und 4 erfüllt
Anforderung		Anforderungstext übernommen aus Anforderungsschablone

Abbildung 3.4: Tabelle der erhobenen Anforderungsattribute

	ISO/IEC/IEEE 29148 quality features	Übersetzung
1	<i>It shall be met or possessed by a system to solve a problem, achieve an objective or address a stakeholder concern.</i>	Sie muss von einem System erfüllt werden oder vorhanden sein, um ein Problem zu lösen, ein Ziel zu erreichen oder eine Interessengruppe anzusprechen.
2	<i>It is qualified by measurable conditions.</i>	Sie wird durch messbare Bedingungen eingeschränkt.
3	<i>It is bounded by constraints.</i>	Sie ist durch Einschränkungen begrenzt.
4	<i>It defines the performance of the system when used by a specific stakeholder or the corresponding capability of the system, but not a capability of the user, operator or other stakeholder.</i>	Sie definiert die Leistung des Systems, wenn es von einem bestimmten Stakeholder genutzt wird, oder die entsprechende Fähigkeit des Systems, nicht aber eine Fähigkeit des Nutzers, Betreibers oder anderen Stakeholders.
5	<i>It can be verified (e.g., the realization of the requirement in the system can be demonstrated).</i>	Sie kann verifiziert werden (z. B. kann die Realisierung der Anforderung im System nachgewiesen werden).

Abbildung 3.5: Qualitätsmerkmale nach ISO/IEC/IEE 29148:2018 Standard.

ID	Typ	Priorisierung	Rolle	Validiert	Abgenommen	Anforderung
1	funktional	8	Projektleiter	JA	JA	Für das Erstellen des Repositories muss der Prototyp Nx als Technologie verwendet werden.
2	funktional	8	Projektleiter	JA	JA	Für das Erstellen einer CI/CD-Pipeline muss der Prototyp GitHub Actions als Technologie verwendet werden
3	funktional	8	Projektleiter	JA	JA	Für die Dateien und Ordner Struktur des Repositories, soll der Prototyp eine von bereitgestellte Nx-Vorlage verwenden.
4	funktional	8	Projektleiter	JA	JA	Als Cloud-Plattform muss der Prototyp Heroku als Technologie verwenden
5	funktional	8	Projektleiter	JA	JA	Backend-Code und Frontend-Code wird der Prototyp in einem gemeinsamen Nx Repository speichern
6	funktional	8	Projektleiter	JA	JA	Für das Erstellen des Frontends, muss der Prototyp AngularJS als Technologie verwenden
7	funktional	8	Projektleiter	JA	JA	Für das Erstellen des Backends muss der Prototyp NestJS als Technologie verwendet werden.
8	funktional	8	Projektleiter	JA	JA	Als VCS-System muss der Prototyp git als Technologie verwenden
9	funktional	8	Projektleiter	JA	JA	Für die Containerisierung der Anwendung, für das spätere Deployment, muss der Prototyp Docker als Technologie verwendet werden.
10	funktional	8	Projektleiter	JA	JA	Für das remote Hosting des Nx Repository, muss der Prototyp die Hosting-Plattform GitHub verwenden
11	funktional	6	DevOps	JA	JA	Für das Speichern von Docker-Images in DockerHub muss der Prototyp fähig sein Docker Images nach DockerHub automatisch zu pushen.
12	funktional	6	DevOps	JA	JA	Für eine Testumgebung und Produktionsumgebung muss der Prototyp verschiedene Heroku-Apps bereitstellen
13	funktional	6	DevOps	JA	JA	Für das Deployen der Anwendung, wird der Prototyp fähig sein von Frontend und Backend Docker Images zu erstellen
14	funktional	6	DevOps	JA	JA	Das Backend und Frontend wird der Prototyp in unterschiedlichen Testumgebungen und Produktionsumgebungen deployen
15	funktional	6	DevOps	JA	JA	Als Release-Branch soll der Prototyp DevOps die Möglichkeit bieten von einem Main-Branch zu deployen
16	funktional	6	DevOps	JA	JA	Wenn ein Pull Request auf den Develop Branch geschlossen wird, soll der Prototyp fähig sein ein Deployment der Quellcode-Änderung auf die Testumgebung automatisch durchzuführen
17	funktional	6	DevOps	JA	JA	Wenn ein Pull Request auf den Main Branch geschlossen wird, soll der Prototyp fähig sein ein Deployment der Quellcode-Änderung auf die Produktionsumgebung automatisch durchzuführen
18	funktional	6	DevOps	JA	JA	Wenn ein Pull Request auf den Develop Branch geschlossen wird, soll der Prototyp fähig sein linting, testing und building des gesamten Quellcodes durchzuführen
19	funktional	6	DevOps	JA	JA	Falls ein push auf den remote Branch durchgeführt wurde, muss der Prototyp fähig sein Status-Benachrichtigungen automatisch zu senden
20	funktional	5	Entwickler	JA	JA	Für die Implementierung des Backends, soll der Prototyp eine, von Nx bereitgestellte, NestJS Vorlage verwenden.
21	funktional	5	Entwickler	JA	JA	Daten aus dem Backend, muss der Prototyp innerhalb eines Services erheben.
22	funktional	5	Entwickler	JA	JA	Bei der Bereitstellung von Daten aus dem Backend an das Frontend, soll der Prototyp fähig sein REST-Calls zu verarbeiten
23	funktional	5	Entwickler	JA	JA	Für das lokale Testen von Docker Images, soll der Prototyp eine Docker-Compose.yml beinhalten
24	funktional	5	Entwickler	JA	JA	Für die Implementierung des Frontend, soll der Prototyp eine von bereitgestellte Nx AngularJS Vorlage verwenden
25	funktional	5	Entwickler	JA	JA	Backend-Code und Frontend-Code soll der Prototyp in unterschiedlichen Ordnern, innerhalb des Nx Repositories speichern
26	funktional	5	Entwickler	JA	JA	Für Backend-Code und Frontend-Code soll der Prototyp in einem gemeinsamen Nx Repository die Dependencies verwalten
27	funktional	5	Entwickler	JA	JA	Als Standard-Branch, für Pull-Request soll der Prototyp Entwicklern die Möglichkeit bieten von einem Develop-Branch Quellcode in lokale Branches zu pullen
28	funktional	5	Entwickler	JA	JA	Um zu verhindern, dass auf Release-Branch und auf Develop-Branch gepusht werden kann, muss der Prototyp beide Branches mit Einschränkungen versehen
29	funktional	5	Entwickler	JA	JA	Nach einem push auf von einem lokalen Branch zu einem remote Branch, muss der Prototyp fähig sein linting, testing (Unit-Test) und building nur für den modifizierten Quellcode automatisch durchzuführen
30	funktional	5	Entwickler	JA	JA	Beim Linting, Testing und Building des soll der Prototyp Entwicklern die Möglichkeit bieten den gesamten, oder nur den modifizierten Quellcode, des Repositories zu verwenden
31	nicht fun...	3	Projektleiter	JA	JA	Die Geschwindigkeit in den Build-Prozessen soll der Prototyp durch das Verwenden von Nx als Building-System beschleunigen

Abbildung 3.6: Spezifizierte Abforderungen

3.2.5 Erhebung von Abnahmekriterien

Ähnlich wie bei der Erhebung von Anforderungen hilft es, die Erhebung von natürlich-sprachigen Abnahmekriterien zu standardisieren [Rup07, S. 326–328]. Dementsprechend empfiehlt Rupp eine Gliederung von Abnahmekriterien in drei Teile:

- **Ausgangssituation des Test:** Beschreibt die Ausgangssituation des zu testenden Artefakts und legt die Testumgebung fest [Rup07, S. 328].
- **Ereignis des Test:** Legt den „Trigger“ fest, der das erwünschte Verhalten des zu testenden Artefakts auslöst, das durch die Anforderungen definiert wurde [Rup07, S. 328].
- **Erwartetes Ergebnis:** Legt den Soll-Zustand fest, der nach korrektem Verhalten des zu testenden Artefakts eintreten soll [Rup07, S. 329].

Auf Basis dieser Gliederung wurde ein Google-Docs Dokument mit einer Tabelle erstellt. Ähnlich wie in Kapitel 3.2.3, soll diese Tabelle eine Struktur bei der Erhebung von Abnahmekriterien schaffen. Die Tabelle beinhaltet drei Spalten, die die beschriebenen Gliederungspunkte in der Kopfzeile tragen. Zudem wurden der Tabelle drei weitere Spalten hinzugefügt: „ID“, „Anforderungs ID“ und „Status“. Es soll gewährleistet werden, dass Abnahmekriterien eindeutig identifiziert werden können und über die Anforderungs ID der jeweiligen Anforderung zugeordnet werden können. Abnahmekriterien und Anforderungen können hierbei in einer n zu n Beziehung stehen. Zudem kann für die Evaluation des Prototyps dem Kriterium ein Status gesetzt werden. Dieser Status bestätigt das Abnahmekriterium entweder als erfolgreich erfüllt, oder bewertet es als durchgefallen. Abbildung 3.7 zeigt diese Tabelle und die erhobenen Abnahmekriterien.

ID	Anforderungs ID	Ausgangssituation	Ereignis des Test	Erwartetes Ergebnis	Status
1	1/3	Projektstart, kein Repository vorhanden.	Nx Repository wird erstellt.	Nx Repository lokal vorhanden.	Offen
2	1	Projektstart, Repository lokal vorhanden.	Nx Repository wird auf GitHub gepusht	Nx Repository auf den remote Branches vorhanden	Offen
3	2	Repository ohne GitHub Actions	GitHub-Actions werden eingerichtet	.github/workflows Ordner im Repository vorhanden. YML-Dateien angelegt	Offen
4	4	Kein Heroku-Account vorhanden	Heroku Account anlegen, betriebsbereit machen	Heroku-Account ist angelegt, Zahlungsmittel wurden hinterlegt	Offen
5	5	Backend und Frontend Quellcode noch nicht geschrieben	Erstellung der Quellcodedateien	Quellcodedateien befinden sich in einem gemeinsamen Repository.	Offen
6	6/24	Frontend wurde noch nicht erstellt.	Frontend wird erstellt mit Nx Package: @nrwl/angular	Erstelltes Frontend ist einem eigen Ordner mit dem Dateipfad ./apps/frontend als AngularJS App	Offen
7	7/20/25	Backend wurde noch nicht erstellt.	Backend wird erstellt mit Nx Package: @nrwl/nest	Erstelltes Backend ist einem eigen Ordner mit dem Dateipfad ./apps/backend als NestJS App	Offen
8	8/10	Git auf Rechner lokal installiert	Konsolenbefehl: git --Version ausgeführt	Git Version 2.30, oder darüber, wird in der Kommandozeile angezeigt	Offen
9	9/13	Backend und Frontend vollständig entwickelt. Docker Engine installiert	Docker Image wird gebaut. Konsolenbefehl: docker images ausgeführt	Gebaute Docker-Images werden angezeigt	Offen
10	10	Git ist installiert	GitHub Account anlegen	Github Account zugänglich.	Offen
11	11/13	Docker Engine installiert, Docker Images gebaut	DockerHub Account anlegen, Docker Images in DockerHub Repository pushen	Docker Images in DockerHub Repository vorhandenen	Offen
12	12/14	Heroku-Account erstellt.	Heroku-Apps werden angelegt	Verschiedene Heroku-Apps sind ansprechbar	Offen
13	14/15/17	CI/CD-Pipeline erstellt, Anwendung Dockerisiert	Pull-Request auf Main Branches wird geschlossen	Anwendung wird automatisch auf die Produktionsumgebung deployed und ist über das Internet erreichbar	Offen
14	14/15/16	CI/CD-Pipeline erstellt, Anwendung Dockerisiert	Pull-Request auf Develop Branches wird geschlossen	Anwendung wird automatisch auf die Testumgebung deployed und ist über das Internet erreichbar	Offen
15	18	Pull Request auf Develop offen	Pull-Request auf Develop Branches wird geschlossen	Lint, Test, Build der gesamten Anwendung wird ausgeführt	Offen
16	19/29/31	Lokale Änderungen am Quellcode vorhanden, CI/CD-Pipeline implementiert	Push der Änderungen wird von lokalen Branch zu Remote branch durchgeführt	Lint, Test, Build der modifizierten Quelldateien wird automatisch ausgeführt, Entwickler wird über erfolg oder misserfolg per Mail benachrichtigt	Offen
17	21/22	Backend vollständig implementiert, mit Service und Controller	Backend-Test und Build werden ausgeführt	Service Test, Controller Test und Build erfolgreich durchgelaufen	Offen
18	23	Docker Engine installiert, Docker Images gebaut, Docker Container laufen lassen	Erstellen einer docker-compose.yml file	Konsolenbefehl: docker-compose up baut alle vorhandenen Docker Images und führt die daraus entstehenden Docker Container aus	Offen
19	26/30	Backend und Frontend vollständig implementiert	Konsolenbefehl: 1. npx nx run-many --target=lint --all 2. npx nx run-many --target=test --all 3. npx nx run-many --target=build --all	Alle Apps, die sich im Nx Repository befinden, führen gleichzeitig Lint, Test und Build aus.	Offen
20	27/28	Git auf Rechner lokal installiert, GitHub Account anlegen	Branch-Regeln für Main und Develop festgelegt, Develop als Standard-Branch festlegen	Develop ist als Standard-Branch festgelegt, auf beide Branches, Develop, als auch Main kann nicht mehr von einem lokalen Branch gepusht werden.	Offen
21	30/31	Backend und Frontend vollständig implementiert	Konsolenbefehl: 1. npx nx affected --target=lint 2. npx nx affected --target=test 3. npx nx affected --target=build	Alle Apps, die sich im Nx Repository befinden und von den Auswirkungen des modifizierten Quellcodes betroffen sind, führen gleichzeitig Lint, Test und Build aus.	Offen

Abbildung 3.7: Liste der erhobenen Abnahmekriterien

3.2.6 Anforderungen modellieren

Für die Modellierung einer fachlichen Lösung des Prototyps, wurde vorerst ein frei gestaltetes Schaubild erstellt. Das Schaubild sollte dabei helfen, erste Ideen und Ansätze visuell festzuhalten. Im Anschluss wurde mit der Modellierungssprache UML dieser freie Ansatz konkretisiert.

UML unterteilt 13 standardisierte Diagrammarten in zwei Kategorien. Die Kategorie der „Strukturdiagramme“ und die Kategorie der „Verhaltensdiagramme“. Diagramme der Kategorie „Strukturdiagramme“ werden dazu genutzt, um die Struktur der modellierten fachlichen Lösung darstellen. Diagramme der Kategorie „Verhaltensdiagramme“ werden dafür genutzt das Verhalten der fachlichen Lösung zu modellieren [Bal09, S. 101–103]. Für die Modellierung einer fachlichen Lösung ist es jedoch nicht notwendig, alle 13 Diagrammarten umzusetzen. Viel eher muss sich für einen bestimmten Satz der 13 Diagrammarten entschieden werden [MH06, S. 12]. In dieser Arbeit wurde die Entscheidung für einen Satz aus zwei Diagrammen getroffen:

- **Aktivitätsdiagramm**, das sich als Verhaltensdiagramm [Bal09, S. 103] dazu eignet, den Ablauf von Prozessen zu modellieren. [MH06, S. 43]. Es ist somit perfekt geeignet, die einzelnen Schritte der zu implementierenden CI/CD-Pipeline abzubilden.
- **Deployment Diagramm**, das sich als Strukturdiagramm dazu verwenden lässt, die physikalische Sicht auf den Prototypen abzubilden. So kann modelliert werden, wie die zu implementierende Software auf Hardware, wie beispielsweise Server, abzulegen werden soll [MH06, S. 224]

Es wurde sich bewusst gegen ein weiteres Strukturdiagramm zum Modellieren der Webanwendung entschieden, da der Fokus des Prototyps nicht auf der zu implementierenden Webanwendung liegt, sondern auf dem Cloud-Deployment als Prozess. Daher erschien es nicht notwendig, die Struktur der Webanwendung im Detail zu planen und zu modellieren.

Freie Visualisierung der fachlichen Lösung

Bei der freien Visualisierung der fachlichen Lösung wurde versucht, die in den Anforderungen definierten Technologien in einen ersten Kontext zueinander zu stellen. Zudem wurde versucht zu visualisieren, wie der zu implementierende Deployment-Prozess aussehen könnte. Die Leserichtung des Schaubilds ist von rechts nach links. Es wurde versucht

darzustellen, wie der Quellcode des zu implementierenden Frontend und Backend, von den lokalen Endgeräten der Entwickler bis zu den Endgeräten der Verbraucher kommt. Zudem wurde versucht abzubilden, an welcher Stelle die GitHub Action Skripte der CI/CD-Pipeline eingreift und was die Skripte automatisiert umsetzen sollen.

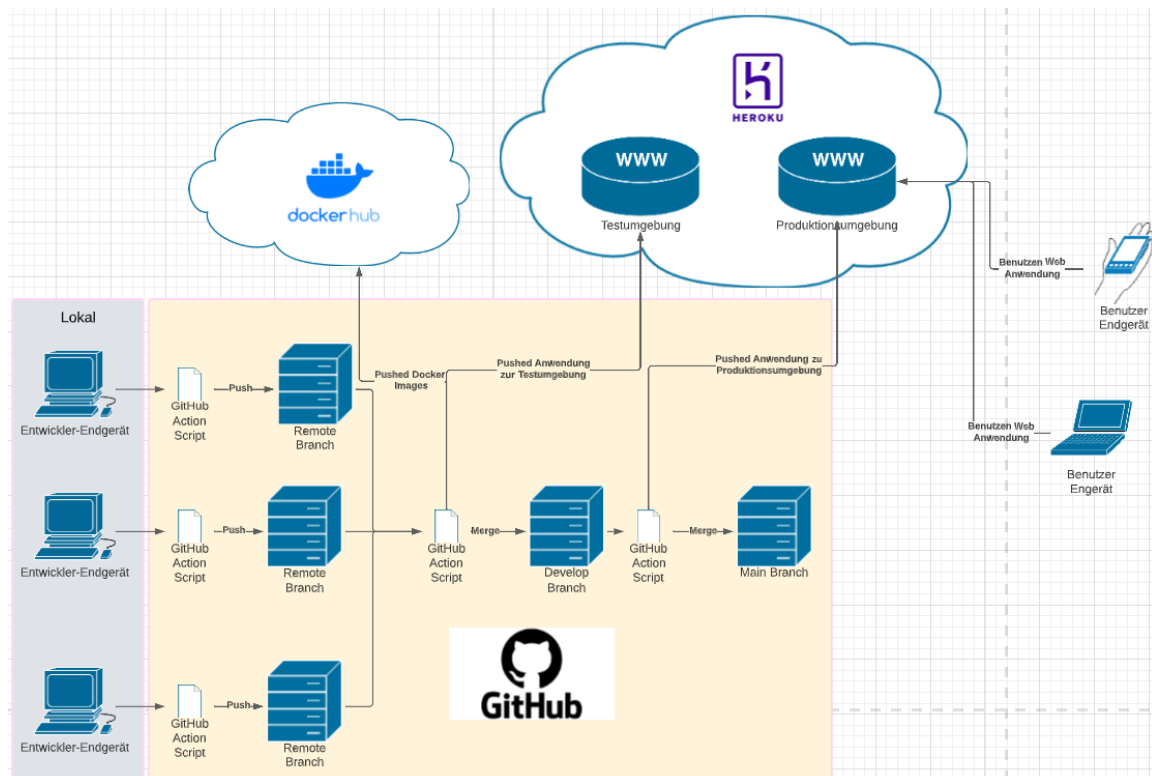


Abbildung 3.8: Frei gestaltete Visualisierung der fachlichen Lösung

Aktivitätsdiagramm

Mit den erhobenen Anforderungen wurde definiert, „was“ der Prototyp tun soll und „wie gut“ der Prototyp dies tun soll. Mit der Erstellung des Aktivitätsdiagramms wird im Anschluss geklärt, „wie“ der Prototyp die Anforderungen umsetzen soll.

Bei einem Aktivitätsdiagramm werden durch Verkettung einzelner Aktionen und Aktivitäten Prozesse dargestellt, die zur Umsetzung von Anforderungen benötigt werden [MH06, S. 43]. Aktivitäten bestehen aus einer Vielzahl einzelner Aktionen und werden innerhalb des Schaubilds durch einen Rahmen gekennzeichnet. Der Ablauf des Aktivitätsdiagramms und somit die Leserichtung, wird vom Startpunkt ausgehend durch Verbindungspfeile gekennzeichnet. Diese verbinden einzelne Aktionen oder Aktivitäten miteinander und bilden somit die Schrittfolge des Prozesses. Während des Prozesses, kann es zu „Entweder-Oder“-

Entscheidungen kommen. Derartige Entscheidungen werden durch ein diamantförmiges Element gekennzeichnet, von dem zwei Pfeile ausgehen. Die Pfeile stellen unterschiedliche Pfade dar, in die sich der Prozess im Folgenden aufteilt. Unterschiedliche Entscheidungspfade werden abschließend, mit einem weiteren diamantförmigen Element wieder zusammengeführt. Es können auch verschiedene Beteiligte innerhalb eines Aktivitätsdiagramm dargestellt werden. Hierfür werden sogenannte Schwimmbahnen modelliert, wobei jede Bahn ein beteiligtes Individuum darstellt. Das Ende eines Aktivitätsdiagramms bildet ein Endpunkt [MH06, S. 44–61].

Abbildung 3.9 zeigt das erstellte Aktivitätsdiagramm. Es wurden hierbei drei Aktivitäten modelliert, die jeweils für eins, der in Kapitel 3.2.6 modellierten GitHub Action Skripte steht. Zudem wurde der dreiteilige Gesamtprozess der CI/CD-Pipeline modelliert. Hierbei ist zu erwähnen, dass es sich bei diesem um die Verkettung der einzelnen Aktivitäten handelt.

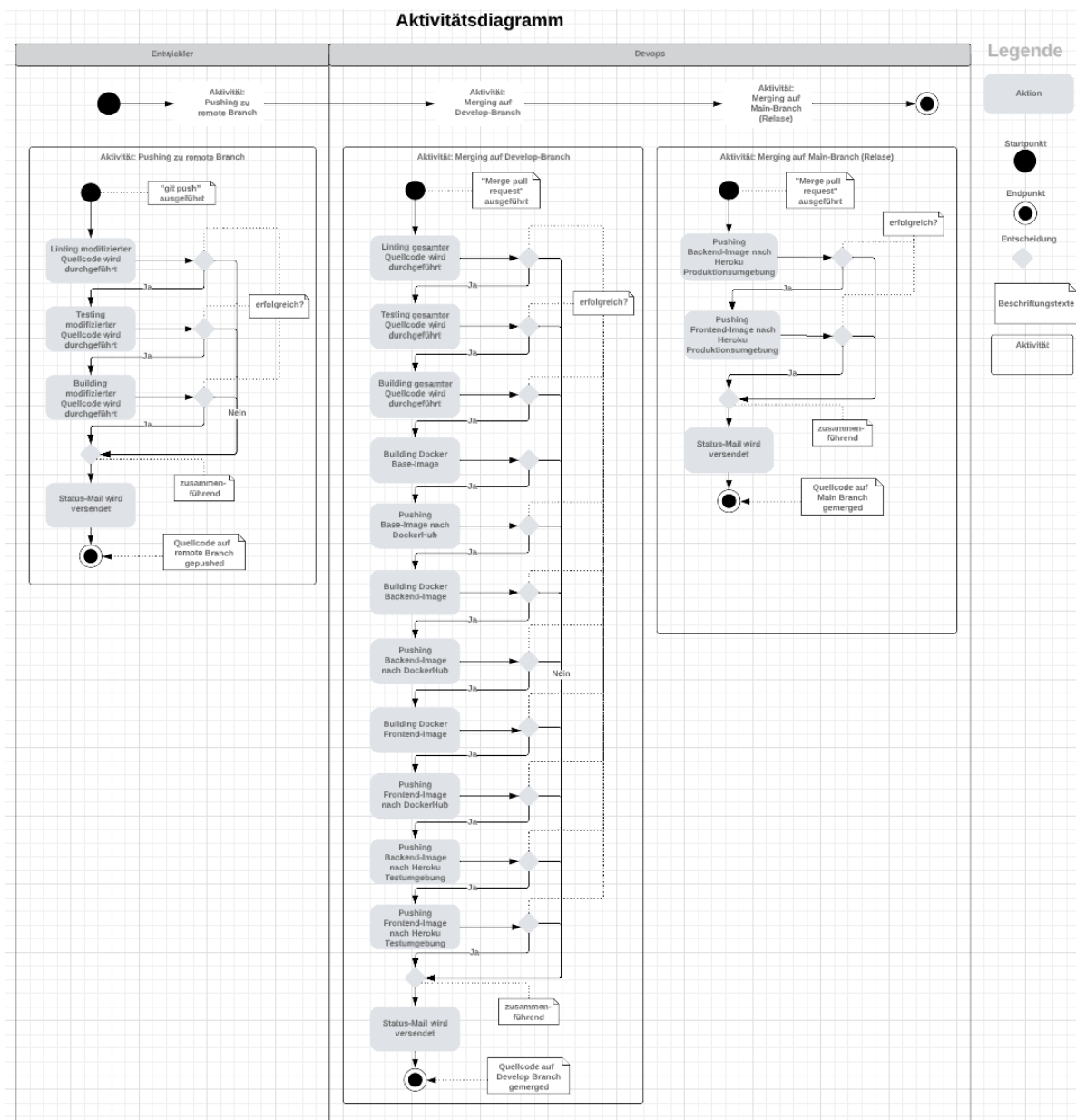


Abbildung 3.9: Aktivitätsdiagramm

Deployment Diagramm

Bei einem Deployment Diagramm wird Hardware als würfelförmiges Element dargestellt. Dieses Element wird über einen Typ definiert (Bsp. Webserver). Software Dateien, wie beispielsweise die Executable einer Software, werden als „Artefakte“ dargestellt. Werden diese Artefakte innerhalb der modellierten Hardware abgebildet, befindet sich die Software Datei auf der Hardware. Software, wie beispielsweise Webanwendungen, werden in der

UML auch als eine Gruppe von zusammenhängenden Komponenten dargestellt. Um Komponenten in einem Deployment Diagramm darstellen zu können, muss ein Artefakt eine Komponente manifestieren. Vereinfacht gesagt: um eine Komponente, wie beispielsweise „das Backend“, in einem Deployment Diagramm darstellen zu können, muss abgebildet werden, welche Software Datei hinter dem Backend steht. Innerhalb eines Hardware-Elements lassen sich sogenannte Ausführungsumgebungen (Execution Enviroments) darstellen. Das könnten zum Beispiel Host-Betriebssysteme sein, wie in Kapitel 2.4.1 näher beschrieben. Mehrere Hardware-Elemente kommunizieren über verschiedene Protokolle miteinander. Diese Kommunikation wird abgebildet, indem zwei Hardware-Elemente mit einer Linie verbunden werden. Die Linie wird mit dem jeweiligen Protokoll beschriftet [MH06, S. 224–232]. Abbildung 3.10 zeigt das erstellte Deployment Diagramm. Das Diagramm zeigt die von Heroku bereitgestellte Cloud-Infrastruktur. Auch die Virtualisierung von Heroku ist abgebildet. Zudem wurde das Endgerät eines symbolischen Verbrauchers modelliert und wie dieses Zugriff auf die Webanwendung bekommt.

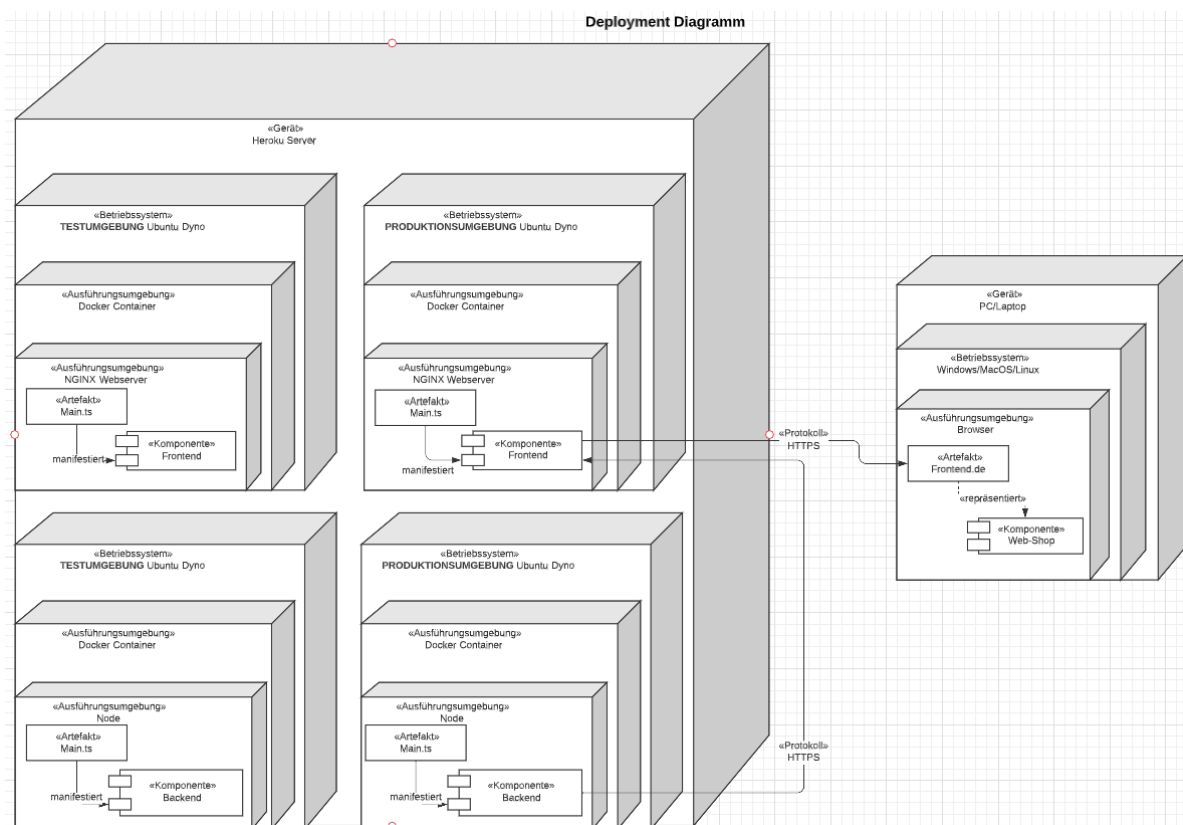


Abbildung 3.10: Deployment Diagramm

Kapitel 4

Implementierung

Das Kapitel Implementierung beschäftigt sich mit der konkreten Umsetzung des Prototyps. Die folgenden Unterkapitel behandeln jeweils essentielle Arbeitsschritte, die für die Umsetzung des Prototyps notwendig waren. Für jeden Arbeitsschritt wurden die verwendeten Technologien erklärt und die betroffenen Anforderungen und Abnahmekriterien aufgelistet. Ziel dieser Herangehensweise war es sicherzustellen, dass bei der Umsetzung des Prototyps keine der erhobenen Anforderungen übersehen wurden. Außerdem sollte nachvollziehbar gemacht werden, an welcher Stelle in der Implementierung welche Abnahmekriterien erfüllt wurden.

4.1 Einrichtung des Version Control Systems und der Kollaborationsplattform

Dieser Arbeitsschritt bildet die Grundlage für alle kommenden Arbeitsschritte und das Fundament des Prototyps.

4.1.1 Betroffene Anforderungen und Abnahmekriterien

- IDs der betroffenen Anforderungen: 8, 10, 27, 28
- IDs der betroffenen Abnahmekriterien: 8, 10, 20

4.1.2 Verwendete Technologien

- **Github:** Für das Hosten des Mono-Repositories wurde GitHub als Code-Sharing-Dienst verwendet. GitHub ist hierbei einer der bekanntesten Kollaborationsplattformen, die die standortunabhängige Zusammenarbeit von Entwicklern unterstützt (siehe Kapitel 2.3.1). Um GitHub als Kollaborationsplattform nutzen zu können, muss als Git als Versionskontrollsystem verwendet werden [Git23k]
- **Git:** Bei Git handelt es um ein DVCS wie in Kapitel 2.3.1 näher beschrieben. Git ist eine open-source Technologie und damit für den privaten, als auch kommerziellen Gebrauch kostenfrei verfügbar [Git23a].

4.1.3 Umsetzung des Arbeitsschritts

Über die offizielle Webseite von GitHub wurde ein GitHub-Account angelegt. Anschließend wurde über die offizielle Webseite von Git [Git23a], „git“ für Windows 10 heruntergeladen. Um zu überprüfen, ob dieser Download ordnungsgemäß funktioniert hat wurde der folgende Befehl in die Kommandozeile (CMD-Befehl) eingegeben:

```
1 | $ git --version
2 | git version 2.31.1.windows.1
```

Code snippet 4.1: Lokale Git-Version überprüfen

Die Ausgabe bestätigte, dass das lokale System über die Git-Version 2.31.1 für Windows verfügte. Im nächsten Schritt wurde über die Weboberfläche von GitHub ein Repository erstellt. In der hierfür bereitgestellten Eingabemaske ließ sich der Name des Repositorys eintragen, sowie ein paar Grundeinstellungen festlegen. So wurde festgelegt, dass das erstellte Repository öffentlich einsehbar ist. Zudem wurde angeklickt, dass eine „README“ Datei und eine „gitignore“ Datei automatisch hinzugefügt wird. „README“ Dateien werden genutzt um das Repository zu beschreiben. Alles was in diese Datei hineingeschrieben wird, wird automatisch als Beschreibungstext dem Repository hinzugefügt [Git23g]. Die „gitignore“ Datei wird verwendet um zu verhindern, dass bestimmte Dateien, die lokal Verwendung finden, nach GitHub überführt werden können [Git23f]. Nach der initialen Erstellung des Repositorys wurde, neben dem automatisch erstellten „Main“-Branch, ein „Develop“-Branch erstellt. Unter dem Button „main“ ließ sich mit einem Klick auf „Create new Branch“, ein neuer Branch erstellen.

Abbildung 4.1 bildet diesen Schritt ab. Nach dem der Develop-Branch erstellt war, konnten unter den Einstellungen des Repositories für beide Branches Regeln festgelegt werden. Es wurde jeweils die Regel „Lock branch“ [Git23i] eingestellt, die verhindert, dass auf beide Branches gepushed werden kann. Zudem wurde der Develop-Branch als Standard-Branch (Default-Branch) für das Repository festgelegt [Git23e]. Abschließend ließ sich das erstellte und konfigurierte Repository mit dem folgenden CMD-Befehlen als Ordner in ein beliebiges Verzeichnis auf dem lokalen System „klonen“.

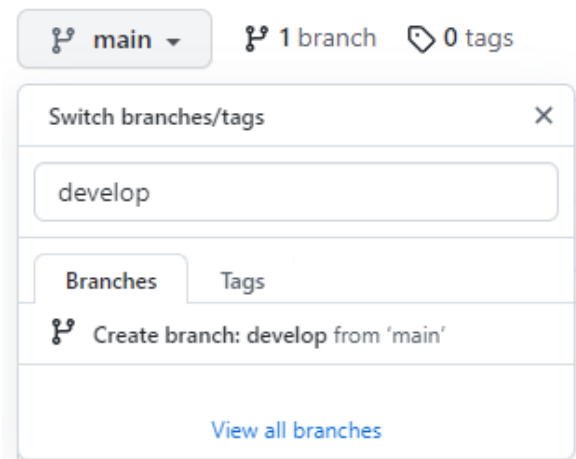


Abbildung 4.1: Erstellung des Develop-Branches

```
1 | $ git clone https://github.com/IvanBirkmaier/praxisteil_bachelor_arbeit.git
```

Code snippet 4.2: Remote Repository klonen

Dateien, die sich innerhalb dieses Ordners befanden konnten nun per „push“ [Git23j] vom lokalen Repository zum remote Repository übertragen werden. Mit dieser initialen Einrichtung des Projektes konnten folgende Abnahmekriterien für die folgenden Anforderungen erfüllt werden: Abnahmekriterium 8 für die Anforderungen 8 und 10, Abnahmekriterium 10 für Anforderung 10 und Abnahmekriterium 20 für Anforderung 27 und 28.

Link zum erstellten Repository: https://github.com/IvanBirkmaier/praxisteil_bachelor_arbeit

4.2 Erstellen eines Mono-Repositorys

Nach der Einrichtung des VCS und der Kollaborationsplattform wurde im zweiten Arbeitsschritt mit Hilfe der Nx-Technologie ein Mono-Repository erstellt.

4.2.1 Betroffene Anforderungen und Abnahmekriterien

- IDs der betroffenen Anforderungen: 1, 3
- IDs der betroffenen Abnahmekriterien: 1, 2

4.2.2 Verwendete Technologien

- **Nx, Nx Cloud:** Nx ist ein Build-Framework, das sich wunderbar für die Architektur und Wartung von Mono-Repositories einsetzen lässt. Hierbei unterstützt Nx als Verwaltungs-Tool alle in Kapitel 2.3.2 beschriebenen Kerneigenschaften eines Mono-Repositories [Rea21]. Darüber hinaus bietet Nrwl mit der Nx Cloud weitere sehr praktische Funktionalitäten, die das Arbeiten mit einem Nx Mono-Repository unterstützen. So lassen sich beispielsweise Ergebnisse von durchgeführten Tests und Builds in der Cloud speichern und verteilen [Nx23j]. Dies hat zur Folge, dass nicht jeder beteiligter Entwickler alle Anwendungen innerhalb des Nx Mono-Repositories immer neu builden muss, sondern vom gemeinsamen Speicher (Cloud-Cache) aller profitieren kann, was die Build- und Testzyklen drastisch beschleunigt [Nx23i]. Eine weitere Funktion die Nx mitbringt ist eine Code-Änderungs-Analyse. Diese ermöglicht es, bestimmte Aktionen wie das Testen, nur für den Quellcode auszuführen, der von Quellcode-Änderungen auch wirklich betroffen ist. Zudem wird analysiert, ob es Abhängigkeiten im Quellcode gibt und ob diese auch von Quellcode-Änderungen betroffen sind. Ist dies der Fall, wird nur der in Abhängigkeit stehende Quellcode erneut gebuildet, jedoch nicht der gesamte Quellcode des Mono-Repository [Nx23e]. Mit Nx lassen sich zudem Dependency Graphen erstellen, die das Dependency Management innerhalb des Mono-Repositorys unterstützen [Nx23c]. Der Hersteller Nrwl verfügt zudem über eine große Auswahl an Plugins, mit denen sich innerhalb weniger Arbeitsschritte die Grundstrukturen für Webanwendungen automatisch erstellen lassen [Nx23h].
- **Npm:** Um Nx als Technologie verwenden zu können, muss vorerst Npm auf dem lokalen System installiert werden [Nx23d]. Npm ist ein open-source Package-Manager, über den Entwickler JavaScript Packages teilen und benutzen können [npm23a]. Bei einem Package handelt es sich um einen Ordner, der über eine sogenannte „packages.json“ Datei, ein Programm beschreibt, das innerhalb des Packages existiert [npm23b]. Npm besteht aus drei Kernelementen. Der offiziellen Webseite über die existierende Packages eingesehen werden können, einem lokalen Npm Kommandozeilen-Interface und einer großen Datenbank. Über das lokale Npm Kommandozeilen-Interface lassen sich CMD-Befehle ausführen. Npm Packages lassen sich über verschiedene Wege in ein Softwareprojekt einbauen. Einer davon geht über den Npx CMD-Befehl. Mit diesem lässt sich die Software, die sich in einem Npm-Packages befindet ausführen, ohne dass das Packages davor heruntergeladen werden muss [npm23a].

4.2.3 Umsetzung des Arbeitsschritts

Npm wurde, wie in der offiziellen Dokumentation geraten [npm23c], über den „Node version manager“ installiert [But23]. Mit folgendem CMD-Befehl ließ sich überprüfen, ob Npm korrekt installiert wurde [npm23c]:

```
1 | $ npm -v
2 | 9.3.0
```

Code snippet 4.3: Lokale Npm Version überprüfen

Die Ausgabe bestätigte, dass das lokale System über die Npm-Version 9.3.0 verfügte. Nach der erfolgreichen Installation von Npm konnte anschließend ein Nx Mono-Repository erstellt werden. Im ersten Schritt musste hierfür ein Nx-Workspace erstellt werden. Dieser ließ sich mit dem folgenden CMD-Befehl einrichten [Nx23g]:

```
1 | $ npx create-nx-workspace@latest
2 | Need to install the following packages:
3 |   create-nx-workspace@15.6.3
4 | Ok to proceed? (y) y
```

Code snippet 4.4: Nx Workspace

Im sich darauf öffnenden Auswahlmenü wurde sich für die Nx Mono-Repository Vorlage „Integrated monorepo“ entschieden. Diese Vorlage ist auf das Erstellen neuer Projekte ausgelegt [Nx23f]. Anschließend ließ sich das „Integrated monorepo“ für das Bauen von Apps spezifizieren. Über das Ermächtigen von „distributed caching“, wurden das lokale Nx Mono-Repository mit der Nx-Cloud verbunden. Dadurch konnten die in Kapitel 4.2.2 beschriebenen Funktionen der Nx-Cloud genutzt werden. Code snippet 4.5 zeigt den hierfür benötigten Konsolendialog.

```
1 | Choose what to create integrated
2 | What to create in the new workspace apps
3 | Repository name praxisteil_bachelor_arbeit
4 | ? Enable distributed caching to make your CI faster ...
5 | Yes I want faster builds
6 | No
7 | > NX Nx is creating your v15.6.3 workspace.
```

Code snippet 4.5: Erstellung eines Nx Monorepositories

Das generierte Nx Mono-Repository wurde automatisch unter dem Verzeichnis C:\Users\...\praxisteil-bachelor-arbeit, im lokalen System abgespeichert. Um das erstellte Mono-Repository auf GitHub bereitzustellen, musste der Ordner mit dem erstellten Nx Mono-Repository in den Order des, in Kapitel 4.1.3 erstellten GitHub Repositories gezogen werden. Abschließend ließ sich mit der folgenden Reihe an CMD-Befehlen, das erstellte Mono-Repository nach GitHub hochladen:

```
1 | $ git add .
```

Code snippet 4.6: Git add

Git-Add fügte alle neuen Dateien, die in den Ordner des lokalen GitHub Repository hineingezogen wurden zum Verzeichnis des lokalen Repositories hinzu [Git23b]

```
1 | $ git commit -m "Initial commit"
```

Code snippet 4.7: Git commit

Git-Commit zeichnete die hinzugefügten Dateien als Änderungen im lokale Repository auf [Git23c].

```
1 | $ git push
```

Code snippet 4.8: Git push

Git-Push lud anschließend das lokale Repository mit seinen Änderungen zum remote Repository auf GitHub hoch [Git23d].

Mit Vollendung dieses Arbeitsschritts konnten die Abnahmekriterien 1 und 2 erfüllt werden und somit die Anforderungen 1 und 3 umgesetzt werden.

4.3 Implementierung einer Webanwendung

Dieser Arbeitsschritt behandelt die Implementierung der Webanwendung. Hierbei wurde sich bewusst für ein AngularJs Frontend und ein NestJs Backend entschieden, da sich beide Frameworks vom Aufbau ähneln. [Nes23].

4.3.1 Betroffene Anforderungen und Abnahmekriterien

- IDs der betroffenen Anforderungen: 5, 6, 7, 20, 21, 22, 24, 25, 26, 30, 31

- IDs der betroffenen Abnahmekriterien: 5, 6, 19, 21

4.3.2 Verwendete Technologien

- **AngularJS:** Bei AngularJS handelt es sich um ein Framework, mit dem sich dynamische Webanwendungen erstellen lassen, die komplett im Browser des Nutzers (client-side) ausgeführt werden können (siehe Kapitel 2.2) [Ang23].
- **NestJS:** Bei NestJS handelt es sich um ein Framework, mit dem sich serverseitige (server-side) Anwendungen umsetzen lassen (siehe Kapitel 2.2) [Nes23].

4.3.3 Umsetzung des Arbeitsschritts

Für die Erstellung des AngularJS Frontends und des NestJS Backends wurden die in Kapitel 4.2.2 beschriebenen Nrwl-Plugins verwendet [Nx23b], [Nx23a]. Im ersten Schritt mussten beide Plugins als Dependencies dem Nx Mono-Repository hinzugefügt werden:

```
1 $ npm install -D @nrwl/nest
2 added 748 packages, and audited 906 packages in 56s
3
4 $ npm install -D @nrwl/angular
5 added 29 packages, and audited 935 packages in 13s
```

Code snippet 4.9: Installation der NestJS und AngularJS Dependencies

Mit den folgenden CMD-Befehlen wurden dann, beide Anwendungen automatisch im Verzeichnis `.\praxisteil_bachelor_arbeit\apps` generiert.

```
1 $ npx nx generate @nrwl/nest:app backend
2 > NX Generating @nrwl/nest:application
3 added 179 packages, removed 2 packages, changed 3 packages, and audited
  1112 packages in 27s
4
5 $ npx nx g @nrwl/angular:app frontend
6 > NX Generating @nrwl/angular:application
7 added 379 packages, changed 1 package, and audited 1491 packages in 2m
```

Code snippet 4.10: Erstellung von Frontend und Backend

Abbildung 4.2 zeigt einen Teilausschnitt des Repository-Verzeichnis, mit beiden erstellten Anwendungen. So konnte das Abnahmekriterium 5 erfüllt werden und somit Anforderung 5 umgesetzt werden. Auch Abnahmekriterium 6 wurde erfüllt, wodurch Anforderungen 6 und 24 umgesetzt wurden und Abnahmekriterium 7 konnte für die Anforderungen 7, 20 und 25 erfüllt werden. Im Anschluss an das automatische generieren der beiden Anwendungen wurden diese getestet, gebuildet und ausgeführt. Somit wurde überprüft, ob das Generieren korrekt funktioniert hat. Hierbei wurden die vielen Vorteile offensichtlich, die die Nx Technologie und die Nx Cloud für die Arbeit mit Mono-Repositories mitbringen. Es wurden verschieden Nx CMD-Befehle ausprobiert. Im ersten Schritt wurden für beide Anwendungen die generierten Tests gleichzeitig ausgeführt. Dieser Schritt wurde anschließend wiederholt um zu testen, ob das Caching der Testergebnisse funktioniert. Code sippet 4.11 zeigt die Ergebnisse dieser Durchläufe.

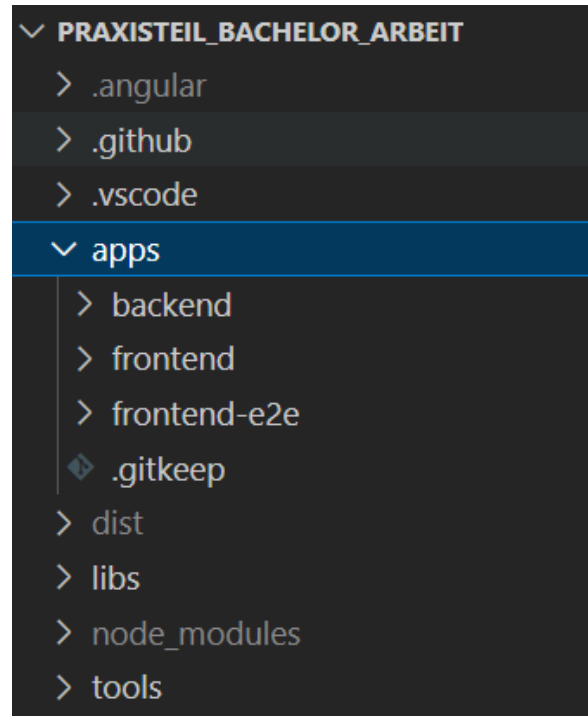


Abbildung 4.2: Verzeichnisstruktur Repository

```

1 | $ npx nx run-many --target=test
2 |     nx run backend:test (6s)
3 |     nx run frontend:test (6s)
4 |
5 | $ npx nx run-many --target=test
6 | > NX Successfully ran target test for 2 projects (103ms)
7 | Nx read the output from the cache instead of running the command for 2 out
   | of 2 tasks.
```

Code snippet 4.11: Nx run Tests

Es wurde ersichtlich, dass sich die Durchlaufzeit durch das Caching drastisch reduzierte. Ähnliche Ergebnisse ergaben sich für das Builden beider Anwendungen. Um die Funktionalität der Code-Änderungs-Analyse (Kapitel 4.2.2) zu testen, wurden Änderungen am generierten Backend-Quellcode vorgenommen. Durch den Nx Dependency-Graph konnte

sich ein Überblick geschaffen werden, welcher Quellcode von den Änderungen am Backend betroffen war. Abbildung 4.3 bildet diesen Graph ab. Der anschließende CMD-Befehl (siehe Kapitel 4.5.3), führte die Tests wie erwartet, nur für das Backend durch, anstatt für Frontend und Backend. Durch die Implementierung des AngularJS Frontends, des NestJS Backends und das Testen der Nx Kernfunktionen, ließen sich die folgenden Abnahmekriterien für die folgenden Anforderungen erfüllen: Abnahmekriterium 17 für Anforderung: 21 und 22, Abnahmekriterium 19 und 21 für Anforderung 30, Abnahmekriterium 21 für Anforderung 31 und Abnahmekriterium 19 für Anforderung 26.

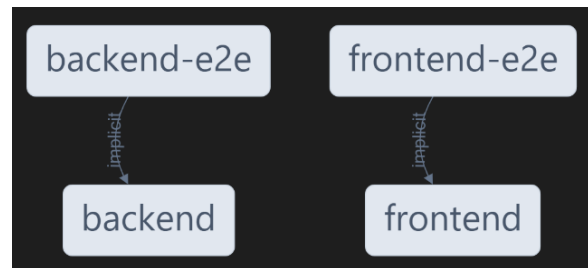


Abbildung 4.3: Nx Dependency Graph

4.4 Dockerisierung des Prototyps

Für die Dockerisierung des Prototyps wurde sich an Kristan Uccellos Blogbeitrag vom 10.12.2020 orientiert [Ucc10]. Wobei zu erwähnen ist, dass Uccello innerhalb seines Nx Mono-Repositorys andere Frameworks verwendete.

4.4.1 Betroffene Anforderungen und Abnahmekriterien

- IDs der betroffenen Anforderungen: 9, 11, 13, 23,
- IDs der betroffenen Abnahmekriterien: 9, 11, 18

4.4.2 Verwendete Technologien

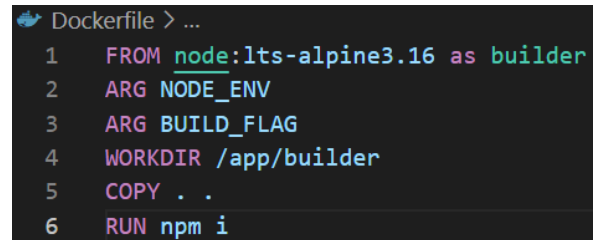
- **Docker Desktop:** Docker Desktop ist eine Anwendung, die das Einrichten der Docker-Technologie stark erleichtert. Mit dem Installieren von Docker Desktop werden alle benötigten Technologien für die Arbeit mit Docker auf einem Klick installiert. Unter diesen Technologien befindet sich beispielsweise die in Kapitel 2.4.1 näher beschriebene „Docker-Engine“, oder die im Folgenden beschriebene „Docker Compose“ [Doc23c].

- **Docker Images:** Um einen Docker-Container, wie er in Kapitel 2.4.1 näher beschrieben wurde, erstellen und ausführen zu können, muss im Vorfeld ein Docker Images erstellt werden. Docker Images lassen sich über sogenannte Dockerfiles erstellen. Dockerfiles müssen in einem Verzeichnis immer auf gleicher Ebene wie die „package.json“-Datei abgelegt werden. Dockerfiles beinhalten alle Befehle, die Docker benötigt, um Docker-Container zu bauen und auszuführen. Dafür wird erst ein Dockerfile geschrieben, anschließend wird ein Docker Image gebaut und mit abschließenden Ausführen des Docker Images, wird ein Docker Container gebaut [f2dockertech.20230202].
- **Docker Container:** Mit dem Bauen des Docker-Containers wird die dockerisierte Anwendung innerhalb des Docker-Containers ausgeführt (siehe Kapitel 2.4.1) [f2dockertech.20230202].
- **Docker Compose:** Die Docker Compose ist zusammengefasst eine YAML-Datei, die es ermöglicht, mehrere Docker Container über eine zentrale Schnittstelle zu bauen und zu verwalten [Doc23b].
- **DockerHub:** DockerHub ist ein Service, über den erstellte Docker Images in einem remoten DockerHub-Repository gehostet werden können. DockerHub ermöglicht es dadurch Docker Images, die sich auf einem lokalen System befinden, für Außenstehende zugänglich zu machen [Doc23a].

4.4.3 Umsetzung des Arbeitsschritts

Für die Dockerisierung des Frontends und Backends wurde im ersten Schritt ein Docker Images gebaut, das bis auf ein paar Ausnahmen das gesamte Verzeichnis des Mono-Repositorys in sich kopierte und alle Npm-Packages installierte. Dieses Base-Image fungierte im nächsten Schritt als Basis für die Docker Images des Frontend und des Backends. Der Grund für diese Herangehensweise ist leicht erklärt. Um das implementierte Frontend und Backend builden und ausführen zu können, müssen diese vollen Zugriff auf die komplette Struktur des Mono-Repositorys haben [Ucc10]. Wenn diese Anwendungen nun in Docker Container gebildet und ausgeführt werden sollen, muss dementsprechend das Docker Images, aus dem der Container entsteht, Zugriff auf die komplette Struktur des Mono-Repositorys haben. Durch das Kopieren und Vererben des gesamten Verzeichnisses, mit samt aller installierten Npm-Packages, konnte dieser Zugriff gewährleistet werden. Der Quellcode des Dockerfiles für das Base-Images sieht wie folgt aus:

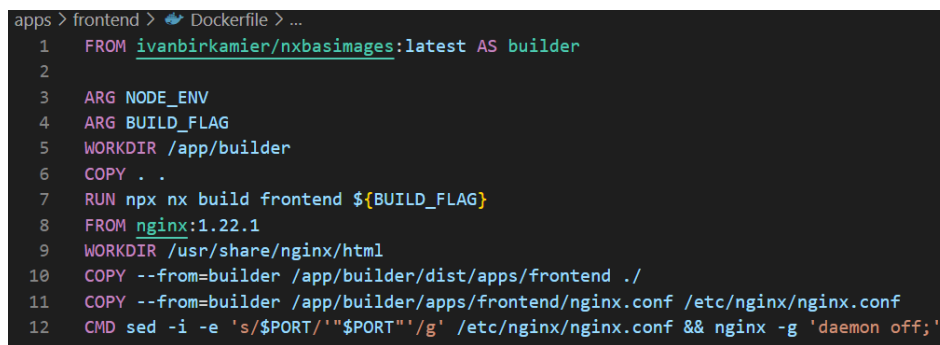
Der Befehl FROM in Zeile 1 von Abbildung 4.4, ermöglicht es, dass innerhalb des Base Images npm Befehle, wie der in Zeile 6 ausgeführt werden können. Mit dem Befehl WORKDIR wird innerhalb des Images ein Verzeichnis erstellt. In das erstellte Verzeichnis kopiert der Befehl COPY das ganze Verzeichnis des Mono-Repositorys. RUN



```
Dockerfile > ...
1 FROM node:lts-alpine3.16 as builder
2 ARG NODE_ENV
3 ARG BUILD_FLAG
4 WORKDIR /app/builder
5 COPY . .
6 RUN npm i
```

Abbildung 4.4: Base Images

npm i installiert schlussendlich alle Npm-Packages des Mono-Repositorys. Nach dem Erstellen des Base Images, konnte dieses nach DockerHub gepushed werden. Anschließend wurden die Docker Images für Frontend und Backend gebaut. Da diese sich vom Aufbau ähneln, wird im Folgenden nur auf das Dockerfile für das Frontend Docker Images eingegangen:



```
apps > frontend > Dockerfile > ...
1 FROM ivanbirkamier/nxbasimages:latest AS builder
2
3 ARG NODE_ENV
4 ARG BUILD_FLAG
5 WORKDIR /app/builder
6 COPY . .
7 RUN npx nx build frontend ${BUILD_FLAG}
8 FROM nginx:1.22.1
9 WORKDIR /usr/share/nginx/html
10 COPY --from=builder /app/builder/dist/apps/frontend ./
11 COPY --from=builder /app/builder/apps/frontend/nginx.conf /etc/nginx/nginx.conf
12 CMD sed -i -e 's/$PORT/'"$PORT"/g' /etc/nginx/nginx.conf && nginx -g 'daemon off;'
```

Abbildung 4.5: Frontend Images

In diesem Fall ermöglicht der FROM Befehl den Zugriff auf das erstellte Base Image. Aus diesem Base Image werden die benötigten Dateien, als auch Npm-Packages geerbt, um den Build des Frontends durchzuführen. Mit dem Befehl RUN npx nx build frontend wird das Frontend gebuildet. Anschließend wird die gebuildete Anwendung auf einen Nginx Webserver abgelegt. Dieser ermöglicht es HTTP-Anfragen zu handeln und das Frontend an Browser auszuliefern [NGI23]. Mit den letzten beiden Befehlen des Dockerfiles, wurden die Konfigurationen des Nginx Webserver angepasst. Auf diese Konfiguration wird in Kapitel 4.6.3 tiefer eingegangen. Um die erstellten Docker Images lokal testen zu können, wurde abschließend eine Docker Compose implementiert. Durch die Dockerisierung des Frontends und des Backends, konnten die folgenden Abnahmekriterien für die folgenden Anforderungen umgesetzt werden: Abnahmekriterium 11 für Anforderung 11 und 13, Abnahmekriterium 9 für Anforderung 9 und Abnahmekriterium 18 für Anforderung 23.

4.5 Bauen einer CI/CD-Pipeline

Bei der Implementierten CI/CD-Pipelines handelt es sich um eine Pipelines, bestehend aus Continous Integration und Continous Deployment (siehe 2.4.2).

4.5.1 Betroffene Anforderungen und Abnahmekriterien

- IDs der betroffenen Anforderungen: 2, 18, 19, 29, 31
- IDs der betroffenen Abnahmekriterien: 3, 15, 16

4.5.2 Verwendete Technologien

- **GitHub Actions:** GitHub Action ist eine CI/CD-Plattform, die von GitHub bereitgestellt wird. Mit dem Erschaffen von Workflows können alle Features der drei, in Kapitel 2.4.2 beschriebenen Bestandteile einer CI/CD-Pipeline, umgesetzt werden. Workflows sind im Repository vorhandene YAML-Dateien die automatische Prozesse ausführen. Ein Workflow wird durch ein bestimmtes Event ausgelöst und führt im Anschluss eine Reihe von Jobs aus. GitHub Action bietet zudem sogenannte Actions an, mit denen sich repetitive Prozessschritte auslagern lassen. Diese Actions können entweder selber implementiert werden, oder über den GitHub Marketplace eingebunden werden. So lassen sich mit der Plattform GitHub Actions umfangreiche und zielorientierte CI/CD-Pipelines umsetzen [Git23l].

4.5.3 Umsetzung des Arbeitsschritts

Um Workflows implementieren zu können, mussten im ersten Schritt die Ordner „.github/working“ im Verzeichnis des Mono-Repositorys angelegt werden [Git23h]. Anschließend wurden drei YAML-Dateien angelegt. Jede dieser YAML-Dateien steht für einen der drei Workflows, wie in den Abbildungen 3.8 abgebildet wurden und 3.9 modelliert wurden. Abbildung 4.6 bildet das angelegte Verzeichnis mit den erstellten YAML-Dateien ab. Mit dem Workflow „push-to-remote-branch“ wurde das automatische Ausführen von Lint, Tests und Builds, nach einem Push vom lokalen Repository auf das remote Repository umgesetzt.

Das Ziel hierbei war die Korrektheit des Quellcodes, der sich im remoten Repository befindet zu gewährleisten. Ähnlich wie in Kapitel 4.3.3, kamen erneut die Vorteile der Nx Technologie zum tragen. Auch bei der Implementierung des Workflows konnte die Code-Änderungs-Analyse von Nx eingebunden werden, um Lints, Tests und Builds nur für von Quellcode-Änderungen betroffene Anwendungen des Mono-Repository auszuführen. Dies optimierte die Performance des Workflows enorm. Als Event das den Workflow auslöst wurde git push festgelegt, wie in Kapitel 4.2.3 näher beschrieben. Abbildung 4.7 bildet die YAML-Datei des Workflows ab.

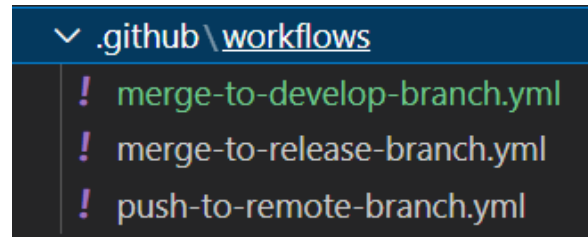


Abbildung 4.6: GitHub Action Workflows

```
.github > workflows > ! push-to-remote-branch.yml
1  name: Push to remote branch
2  run-name: Linting, testing, buliding up
3  all the affected services of your commits
4  on:
5    push:
6      branches-ignore:
7        - main
8        - develop
9  jobs:
10   lint-test-build:
11     runs-on: ubuntu-latest
12     steps:
13       - uses: actions/checkout@v2
14         with:
15           fetch-depth: 0
16       - uses: nrwl/nx-set-shas@v3
17       - run: npm ci
18       - run: npx nx affected --target=lint --parallel=3
19       - run: npx nx affected --target=test --parallel=3 --ci --code-coverage
20       - run: npx nx affected --target=build --parallel=3
```

Abbildung 4.7: Push-to-remote-branch Workflow

Mit dem Abschnitt unterhalb des „on“-Befehls wird das Event festgelegt, durch das der Workflow ausgeführt wird. Unterhalb des „jobs“-Befehls werden die einzelnen Jobs definiert, die innerhalb des Workflows abgearbeitet werden sollen. In diesem Beispiel wurde nur Job lint-test-build implementiert. Innerhalb des Jobs werden einzelne Steps abgearbeitet. Mit jedem dieser Steps wird ein CMD-Befehle ausgeführt. In Zeile 16 wurde eine GitHub Action aus dem GitHub Marketplaces eingebunden. Diese ermöglichte es, die Nx Code-Änderungs-Analyse innerhalb des Jobs anzuwenden. Durch die drei Befehle *npx nx affected*, wurden abschließend Lint, Test und Builds nur für die Anwendungen innerhalb des Mono-Repositories ausgeführt, die von Quellcode Änderungen betroffen

waren (affected).

Der Workflow „merge-to-develop-branch“ wird ausgelöst, wenn ein Pull Request auf den Develop-Branch geschlossen wird. Sprich wenn ein Merge auf den Develop-Branch durchgeführt wird. Anders als beim Workflow „push-to-remote“, wurde im darauf folgenden Job Lints, Tests und Builds für den ganzen Quellcode durchgeführt. So wurde garantiert, dass auf den Develop-Branch niemals fehlerhafter Quellcode gemerged wurde. Nach dem erfolgreichen Durchführen des „lint-test-build“ Jobs, wurden automatisch ein Base Image, ein Frontend Image und ein Backend Image gebaut und nach DockerHub gepushed. Die Images wurden für das spätere Deployment von Frontend und Backend benötigt. Zudem wurde durch diesen Job sichergestellt, dass die Docker Images fehlerfrei gebaut werden konnten. Der grundlegende Aufbau des hierfür benötigten Events und des „lint-test-build“ Jobs, ähneln dem oben beschrieben. Darum wird im Folgenden nur das automatische Builden und Pushen des Base Images ergänzt. Abbildung 4.8 zeigt den dafür zuständigen Job. Um Docker Images mit GitHub Actions builden und pushen zu können, müssen zwei Actions aus dem GitHub Marketplace eingebunden werden. Mit der ersten Action meldet man sich bei DockerHub an. Mit der zweiten Action lassen sich daraufhin die Images builden und pushen. Für die Anmeldung bei DockerHub wurden das Passwort und der Benutzername über Platzhalter-Variablen (Environment Variables) an den laufenden Workflow übergeben. Environment Variables verhindern es, dass sensible Daten für Dritte einsehbar sind und lassen sich in den Einstellungen von GitHub festlegen. Um die Images bauen zu können, wurde über den „context“, in das Verzeichnis des Mono-Repositorys navigiert, in dem sich das Dockerfile für das Docker Image befindet. Beim pushen der Images auf DockerHub müssten diese richtig benannt werden. Die richtige Syntax besteht aus einem Benutzernamen, dem DockerHub-Repository Namen, in welches das Image gepushed werden soll und einem Tag, der das Image einzigartig macht (siehe Abbildung 4.8, Zeile 45).

Die Images für Frontend und Backend wurde das gleiche Schema angewendet.

```

.github > workflows > ! merge-to-develop-branch.yml
27   building-pushing-base-image:
28     if: github.event.pull_request.merged == true
29     needs: lint-test-build-repository
30     runs-on: ubuntu-latest
31     steps:
32     - uses: actions/checkout@v2
33       with:
34         fetch-depth: 0
35     - name: Login to Docker Hub
36       uses: docker/login-action@v2
37       with:
38         username: ${ secrets.DOCKER_USER }
39         password: ${ secrets.DOCKER_TOKEN }
40     - name: Building and pushing the NX-Base-Image for all the Apps in Nx-Repository to Docker Hub
41       uses: docker/build-push-action@v2
42       with:
43         context: .
44         push: true
45         tags: ${ secrets.DOCKER_USER }}/nxbasimages:${ env.IMAGE_TAG }, ${ secrets.DOCKER_USER }}/nxbasimages:latest

```

Abbildung 4.8: Job zum Builden und Pushen des Base Images

Visuell wurde das Ausführen von Workflows durch die GitHub Weboberfläche unterstützt. Zudem wurden Beteiligte des GitHub Projektes bei Misserfolg des durchgeführten Workflows automatisch per Mail benachrichtigt. Abbildung 4.10 zeigt die GitHub Weboberfläche während eines laufenden Workflows. Abbildung 4.9 zeigt den Mailtext, der automatisch generiert und versendet wurde. Durch das Implementieren dieser ersten zwei Workflows konnten die folgenden Abnahmekriterien für die folgenden Anforderungen umgesetzt werden: Abnahmekriterium 3 für Anforderung 2, Abnahmekriterium 15 für Anforderung 18 und Abnahmekriterium 16 für Anforderung 19, 29 und 31.



[IvanBirkmaier/praxisteil_bachelor_arbeit] Push to remote branch workflow run

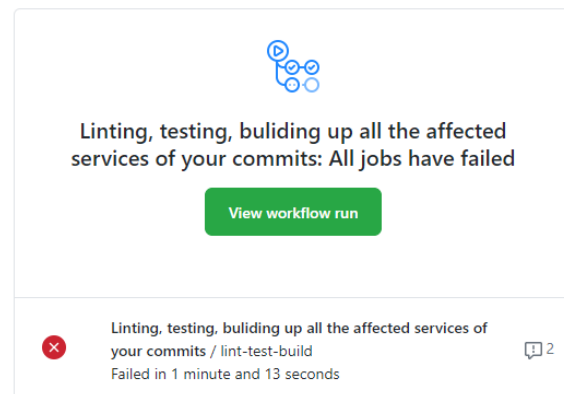


Abbildung 4.9: Mailtext bei Misserfolg eines ausgeführten Workflows

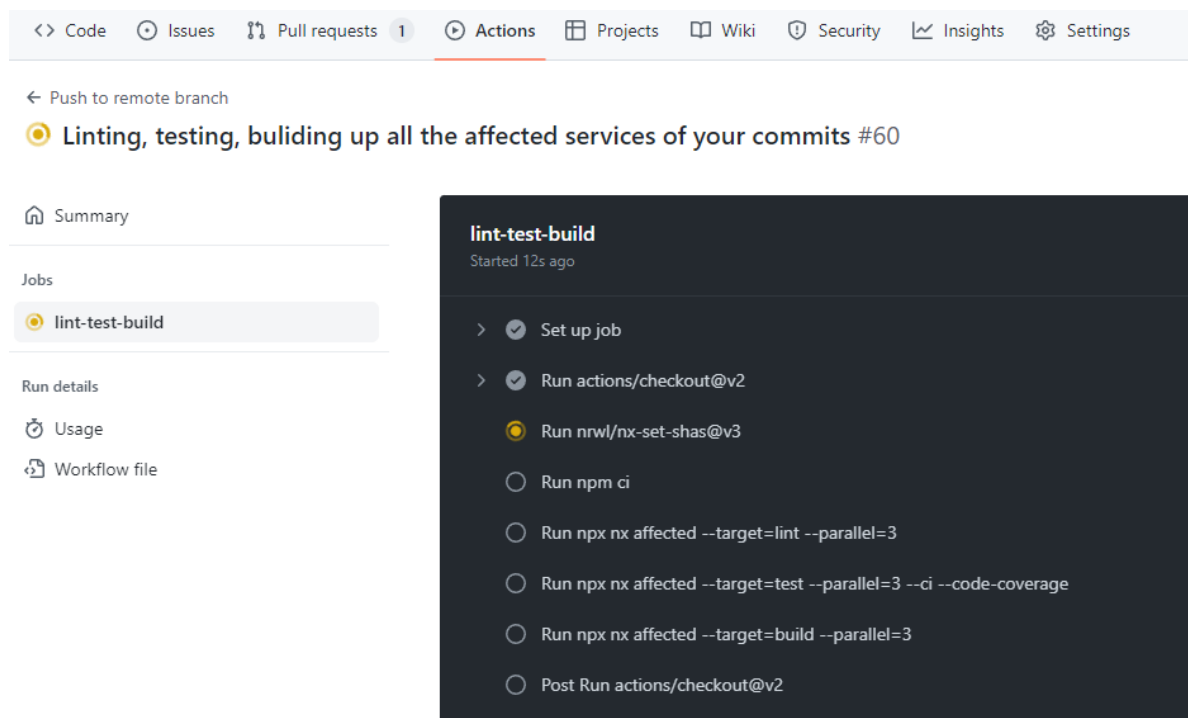


Abbildung 4.10: GitHub Weboberfläche während eines laufenden Workflows

4.6 Deployment auf Heroku

Für das Deployment auf Heroku wurde in den Anforderungen festgelegt, dass Backend und Frontend in unterschiedliche Test- und Produktionsumgebungen deployed werden sollen.

4.6.1 Betroffene Anforderungen und Abnahmekriterien

- IDs der betroffenen Anforderungen: 4, 12, 14, 15, 16, 17
- IDs der betroffenen Abnahmekriterien: 4, 12, 13, 14

4.6.2 Verwendete Technologien

- **Heroku:** Heroku ist eine PaaS Cloud Plattform [Her23]. Anwendungen die auf Heroku deployed werden, werden innerhalb sogenannter Dynos ausgeführt [Her22c]. Dynos sind leichtgewichtige Linux Container, die sich in zwei Klassen unterteilen

lassen. Für diese Arbeit ist nur die Klasse der Web-Dynos relevant, da nur diese HTTP-Aufrufe entgegennehmen können [Her22b].

4.6.3 Umsetzung des Arbeitsschritts

Um Heroku als Cloud-Plattform nutzen zu können, musste im ersten Schritt ein Account erstellt werden und diesem Kreditkarteninformationen hinterlegt werden. Nach Einrichtung des Heroku Accounts, ließen sich über die Weboberfläche Heroku-Apps erstellen. Diese Apps stellen die verschiedenen Test- und Produktionsumgebung dar. Jeder dieser Apps verfügt über eigene Dynos. Abbildung 4.11 zeigt die vier erstellten Heroku-Apps. Für das automatische Deployment des dockerisierten Frontends und Backends in die jeweiligen Testumgebungen, wurde der in Kapitel 4.5.3 näher beschriebene Workflow „merge-to-develop-branch“ in seiner Logik erweitert. Der Job für das automatische Deployment benötigte eine Action aus dem GitHub Marketplaces. Abbildung 4.12 zeigt den Deployment Job mit allen seinen Steps.

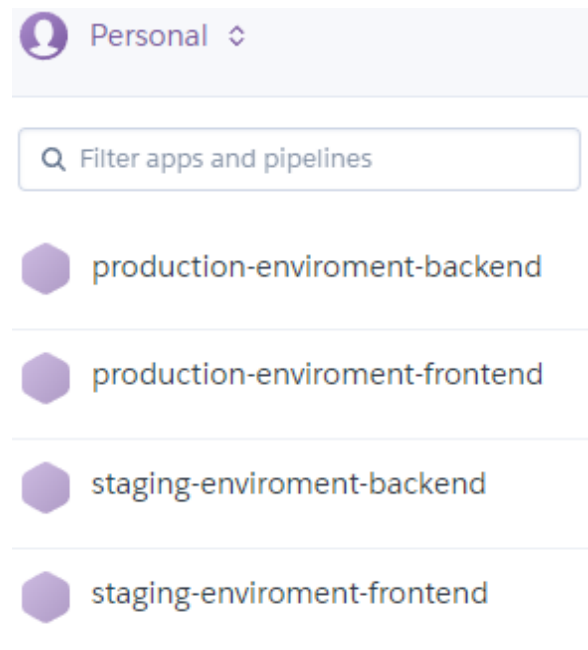


Abbildung 4.11: Erstellte Heroku Apps

```
github > workflows > ! merge-to-develop-branch.yml
71   building-deploying-to-Heroku:
72     if: github.event.pull_request.merged == true
73     needs: building-pushing-app-images
74     runs-on: ubuntu-latest
75     steps:
76     - uses: actions/checkout@v2
77     - name: Register and deploy NestJS Images to Heroku
78       uses: gonuit/heroku-docker-deploy@v1.3.3
79       with:
80         email: ${ secrets.HEROKU_ACC }
81         heroku_api_key: ${ secrets.HEROKU_API_KEY }
82         heroku_app_name: ${ secrets.STAG_B }
83         dockerfile_directory: ./apps/backend
84         process_type: web
85     - name: Register and deploy AngularJS Images to Heroku
86       uses: gonuit/heroku-docker-deploy@v1.3.3
87       with:
88         email: ${ secrets.HEROKU_ACC }
89         heroku_api_key: ${ secrets.HEROKU_API_KEY }
90         heroku_app_name: ${ secrets.STAG_F }
91         dockerfile_directory: ./apps/frontend
92         process_type: web
```

Abbildung 4.12: GitHub Action Job für das Deployment

Im Grunde ähnelt sich der Aufbau dieses

Jobs sehr mit dem, in Kapitel 4.5.3 beschriebenen, Job zum builden und pushen von Docker Images. Im Ersten Schritt musste sich über Enviroments Variablen bei Heroku angemeldet werden. Außerdem musste der Name der Heroku-App festgelegt werden, in die das Docker Images deployed werden soll. Anschließend wurde über den Befehl „dockerfile_directory“ festlegt, in welchem Verzeichnis sich das Dockerfile für das zu deployende Docker Image befindet. Mit dem Befehl „process_type“ wurde abschließend festgelegt, auf welchen Dyno die Anwendung innerhalb der Heroku-App deployed werden soll.

Das Backend ließ sich so ohne weitere Probleme deployen. Für das Frontend musste, wie in Kapitel 4.4.3 schon angesprochen, der Nginx Webserver so konfiguriert werden, dass ihm dynamisch ein Port zugewiesen werden konnte. Grund hierfür ist, dass Heroku für Web-Dynos zufällige Ports verteilt, die nach dem deployen der Anwendung von dieser übernommen werden müssen. Mit diesen Port macht der Web-Dyno die Anwendung über das Internet aufrufbar. Wenn der Nginx Server mit dem gebuildeten Frontend, innerhalb des Docker Container, den Port 80 abhört, Webaufrufe auf den Web-Dyno aber über Port 435 eintreffen, kann das Frontend nicht beim Client nicht geladen werden. In diesem Beispiel müsste der Nginx Server ebenfalls Port 435 abhören, um die Webanwendung bei eintreffendem Aufruf, an den Client ausliefern zu können [Sal20]. Nach Konfiguration des Nginx Webserver, konnte das Frontend über das Internet aufgerufen werden, und es konnten REST-Calls an das Backend gesendet werden.

Anschließend wurde der Workflow „merge-to-release-branch“ implementiert. Mit diesem

Workflow wurden das Backend und das Frontend zur finalen Produktionsumgebung deployed. Hierfür wurde als Event der Merge von Develop-Branch auf Main-Branch gewählt. Die Logik für diesen Workflow unterscheidet sich nicht von der oben beschriebenen. Der einzige Unterschied liegt darin, dass die App-Namen geändert wurden, sodass Backend und Frontend in ihre jeweilige Produktionsumgebung deployed wurden. Nach erfolgreichen Deployment des Frontends und des Backend auf Produktionsumgebung, ließ sich die Webanwendung über die folgende Url aufrufen: <https://production-enviroment-frontend.herokuapp.com/>.

Über den Button „Hier klicken!“ wird ein REST-Call an das Backend gesendet, das mit einem JSON-Objekt antwortet. Abbildung 4.14 zeigt das erstellte Frontend und Abbildung 4.13 das JSON-Objekt, das das Backend zurücksendet. Durch das automatisierte Deployment von Frontend und Backend auf Heroku, konnten die folgenden Abnahmekriterien für die folgenden Anforderungen umgesetzt werden: Abnahmekriterium 4 für Anforderung 4, Abnahmekriterium 12 für Anforderung 12 und 14, Abnahmekriterium 13 für Anforderung 14, 15 und 17 und Abnahmekriterium 14 für Anforderung 14, 15 und 16.

```
{"message": "NestJS Backend"}
```

Abbildung 4.13: JSON-Objekt des Backends

Willkommen zum,
praktischen Teil der Bachelorarbeit
👉

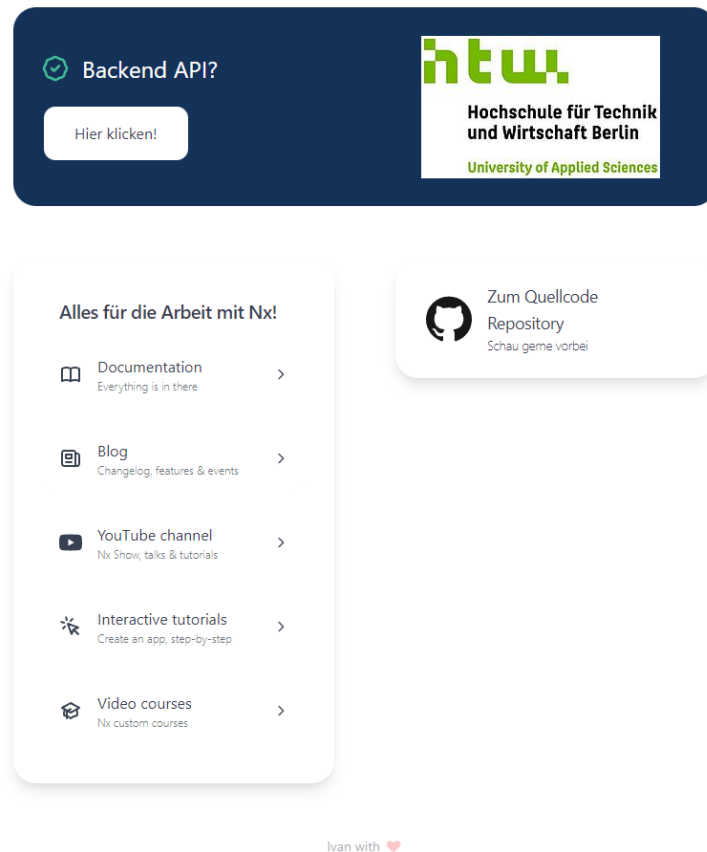


Abbildung 4.14: Frontend im Browser

Kapitel 5

Evaluation

Für alle Anforderungen, die an den Prototyp gestellt wurden, konnten die Abnahmekriterien im Laufe der Implementierung erfüllt werden. Siehe Tabelle in Abbildung 5.1.

ID	Anforderungs ID	Ausgangssituation	Ereignis des Test	Erwartetes Ergebnis	Status
1	1/3	Projektstart, kein Repository vorhanden.	Nx Repository wird erstellt.	Nx Repository lokal vorhanden.	Bestanden
2	1	Projektstart, Repository lokal vorhanden.	Nx Repository wird auf GitHub gepusht	Nx Repository auf den remote Branches vorhanden	Bestanden
3	2	Repository ohne GitHub Actions	GitHub-Actions werden eingerichtet	.github/workflows Ordner im Repository vorhanden. YML-Dateien angelegt	Bestanden
4	4	Kein Heroku-Account vorhanden	Heroku Account anlegen, betriebsbereit machen	Heroku-Account ist angelegt, Zahlungsmittel wurden hinterlegt	Bestanden
5	5	Backend und Frontend Quellcode noch nicht geschrieben	Erstellung der Quellcodedateien	Quellcodedateien befinden sich in einem gemeinsamen Repository.	Bestanden
6	6/24	Frontend wurde noch nicht erstellt.	Frontend wird erstellt mit Nx Package: @nrwl/angular	Erstelltes Frontend ist einem eigen Ordner mit dem Dateipfad ./apps/frontend als AngularJS App	Bestanden
7	7/20/25	Backend wurde noch nicht erstellt.	Backend wird erstellt mit Nx Package: @nrwl/nest	Erstelltes Backend ist einem eigen Ordner mit dem Dateipfad ./apps/backend als NestJS App	Bestanden
8	8/10	Git auf Rechner lokal installiert	Konsolenbefehl: git --Version ausgeführt	Git Version 2.30, oder darüber, wird in der Kommandozeile angezeigt	Bestanden
9	9/13	Backend und Frontend vollständig entwickelt. Docker Engine installiert	Docker Image wird gebaut. Konsolenbefehl: docker images ausgeführt	Gebaute Docker-Images werden angezeigt	Bestanden
10	10	Git ist installiert	GitHub Account anlegen	Github Account zugänglich.	Bestanden
11	11/13	Docker Engine installiert, Docker Images gebaut	DockerHub Account anlegen, Docker Images in DockerHub Repository pushen	Docker Images in DockerHub Repository vorhanden	Bestanden
12	12/14	Heroku-Account erstellt.	Heroku-Apps werden angelegt	Verschiedene Heroku-Apps sind ansprechbar	Bestanden
13	14/15/17	CI/CD-Pipeline erstellt, Anwendung Dockerisiert	Pull-Request auf Main Branches wird geschlossen	Anwendung wird automatisch auf die Produktionsumgebung deployed und ist über das Internet erreichbar	Bestanden
14	14/15/16	CI/CD-Pipeline erstellt, Anwendung Dockerisiert	Pull-Request auf Develop Branches wird geschlossen	Anwendung wird automatisch auf die Testumgebung deployed und ist über das Internet erreichbar	Bestanden
15	18	Pull Request auf Develop offen	Pull-Request auf Develop Branches wird geschlossen	Lint, Test, Build der gesamten Anwendung wird ausgeführt	Bestanden
16	19/29/31	Lokale Änderungen am Quellcode vorhanden, CI/CD-Pipeline implementiert	Push der Änderungen wird von lokalen Branch zu Remote branch durchgeführt	Lint, Test, Build der modifizierten Quelldateien wird automatisch ausgeführt, Entwickler wird über erfolg oder misserfolg per Mail benachrichtigt	Bestanden
17	21/22	Backend vollständig implementiert, mit Service und Controller	Backend-Test und Build werden ausgeführt	Service Test, Controller Test und Build erfolgreich durchgelaufen	Bestanden
18	23	Docker Engine installiert, Docker Images gebaut, Docker Container laufen lassen	Erstellen einer docker-compose.yml file	Konsolenbefehl: docker-compose up baut alle vorhandenen Docker Images und führt die daraus entstehenden Docker Container aus	Bestanden
19	26/30	Backend und Frontend vollständig implementiert	Konsolenbefehl: 1. npx nx run-many --target=lint --all 2. npx nx run-many --target=test --all 3. npx nx run-many --target=build --all	Alle Apps, die sich im Nx Repository befinden, führen gleichzeitig Lint, Test und Build aus.	Bestanden
20	27/28	Git auf Rechner lokal installiert, GitHub Account anlegen	Branch-Regeln für Main und Develop festgelegt, Develop als Standard-Branch festlegen	Develop ist als Standard-Branch festgelegt, auf beide Branches, Develop, als auch Main kann nicht mehr von einem lokalen Branch gepusht werden.	Bestanden
21	30/31	Backend und Frontend vollständig implementiert	Konsolenbefehl: 1. npx nx affected --target=lint 2. npx nx affected --target=test 3. npx nx affected --target=build	Alle Apps, die sich im Nx Repository befinden und von den Auswirkungen des modifizierten Quellcodes betroffen sind, führen gleichzeitig Lint, Test und Build aus.	Bestanden

Abbildung 5.1: Evaluation der erhobenen Abnahmekriterien

Kapitel 6

Fazit

Für das gemeinsame Cloud-Deployment von Backend und Frontend aus einem Repository wurde ein Docker und Nx basierter Prototyp entwickelt, der mittels einer CI/CD-Pipeline ein Deployment automatisiert umsetzt. Aufgrund der allgemeinen Bauweise kann der Prototyp für weitere Deployments als Vorlage verwendet werden. Im Zuge des Entwicklungsprozesses wurde außerdem ein ausführliches und umfangreiches Requirement Engineering angewandt. So konnten qualitativ hochwertige Anforderungen und Abnahmekriterien erhoben werden.

Nx als Technologie ermöglichte es zwei der größten Herausforderungen bei der Arbeit mit Mono-Repositorys leicht zu bewältigen. Zum einen ließen sich durch die Code-Änderungs-Analyse Build-Prozesse effizient managen. Zum anderen ließen sich die Dependencies innerhalb des Nx Mono-Repositorys übersichtlich verwalten. Wodurch viel Zeit im Entwicklungsprozess eingespart werden kann. Indem der Prototyp einen Lösungsansatz für das effektive Managen von Build-Prozessen und Dependencies bietet, konnten die, in Kapitel 2.5 Verwandte Arbeiten gestellten, Vorsätze an die Arbeit umgesetzt werden. Die Kombination von GitHub als Kollaborationsplattform und GitHub Actions für das Implementieren einer CI/CD-Plattform, ermöglichte es einen benutzerfreundlichen und schnellen Deployment-Prozess umzusetzen. Der gewählte Ansatz zur Dockerisierung aller Anwendungen, hat den Vorteil, dass er sich leicht auf kommende Anwendungen im Repository übertragen lässt. Limitierend für diesen Ansatz ist zu nennen, dass das Base-Image in Relation zu Repository mitwächst. Bei einem sehr großen Mono-Repository würde das Bauen und Pushen des Base Images deshalb sehr viel Zeit in Anspruch nehmen, wodurch in der Praxis verwendete Softwaresysteme schwierig wartbar werden können. Bei der Implementierung des Prototyps hat sich zudem herausgestellt, dass Heroku als PaaS-Plattform Docker zwar unterstützt, jedoch sollten diese nur bei notwendigen Bedarf verwendet werden, da Heroku durch die bereitgestellten Dynos schon eigene Container

bereitstellen [Her22a]. Im Prototyp hingegen war das Deployment mit Docker leicht umsetzbar. Es ist aufgefallen, dass es schwierig war, nicht funktionale Anforderungen an den Prototyp zu erheben, da bei es bei der Planung keine konkreten Referenzpunkte gab. Beispielsweise hätte der Vergleich zu einem bereits existierenden System genutzt werden können, um abzuleiten wie gut der Prototyp dieses mit seiner Performanz übertreffen kann. Durch aktuell komplexer werdende Software wird auch zukünftig das Managen des Quellcodes, die Auslegung von verschiedenen Repository-Modellen und das automatisieren des Deployments relevanter und folglich weiter in den Forschungsfokus rücken.

Abbildungsverzeichnis

2.1	Repository Pattern, Quelle: [Ede11]	5
2.2	Vergleich von CVCS und DVCS, Quelle: [ZND18, S. 409].	7
2.3	Virtuelle Maschine im Vergleich zu Containern, Quelle: [War22]	11
2.4	Leistungsunterschiede, Virtual Machines zu Docker Container, Quelle: [Pot20, S. 1422]	12
2.5	Continuous Delivery und Continuous Deployment im Vergleich, Quelle: [War20]	14
3.1	Liste aller Stakeholder, in Anlehnung an [Bal09, S. 504]	20
3.2	Vollständige Anforderungsschablone, Quelle: [Pet21]	21
3.3	Anforderungsschablone mit erhobenen Anforderungen	22
3.4	Tabelle der erhobenen Anforderungsattribute	25
3.5	Qualitätsmerkmale nach ISO/IEC/IEEE 29148:2018 Standard.	25
3.6	Spezifizierte Abforderungen	26
3.7	Liste der erhobenen Abnahmekriterien	28
3.8	Frei gestaltete Visualisierung der fachlichen Lösung	30
3.9	Aktivitätsdiagramm	32
3.10	Deployment Diagramm	33
4.1	Erstellung des Develop-Branche	36
4.2	Verzeichnistruktur Repository	41
4.3	Nx Dependency Graph	42
4.4	Base Images	44
4.5	Frontend Images	44
4.6	GitHub Action Workflows	46
4.7	Push-to-remote-branch Workflow	46
4.8	Job zum Builden und Pushen des Base Images	48
4.9	Mailtext bei Misserfolg eines ausgeführten Workflows	48
4.10	GitHub Weboberfläche während eines laufenden Workflows	49
4.11	Erstellte Heroku Apps	50

4.12	GitHub Action Job für das Deployment	51
4.13	JSON-Objekt des Backends	52
4.14	Frontend im Browser	53
5.1	Evaluation der erhobenen Abnahmekriterien	55

Source Code Content

4.1	Lokale Git-Version überprüfen	35
4.2	Remote Repoditory klonen	36
4.3	Lokale Npm Version überprüfen	38
4.4	Nx Workspace	38
4.5	Erstellung eines Nx Monorepositories	38
4.6	Git add	39
4.7	Git commit	39
4.8	Git push	39
4.9	Installation der NestJS und AngularJS Dependencies	40
4.10	Erstellung von Frontend und Backend	40
4.11	Nx run Tests	41

Literaturverzeichnis

- [ABL15] Jean-Paul Arcangeli, Raja Boujbel und Sébastien Leriche. „Automatic deployment of distributed software systems: Definitions and state of the art“. In: *Journal of Systems and Software* 103 (2015), S. 198–218. ISSN: 0164-1212. DOI: 10.1016/j.jss.2015.01.040. URL: <https://www.sciencedirect.com/science/article/pii/S0164121215000308> (besucht am 04.02.2023).
- [Bal09] Helmut Balzert. *Lehrbuch der Softwaretechnik: Basiskonzepte und Requirements Engineering*. 3. Aufl. Heidelberg: Spektrum Akademischer Verlag, 2009. ISBN: 978-3-8274-1705-3. DOI: 10.1007/978-3-8274-2247-7. (Besucht am 04.02.2023).
- [BBA17] Babak Bashari Rad, Harrison Bhatti und Mohammad Ahmadi. „An Introduction to Docker and Analysis of its Performance“. In: *IJCSNS International Journal of Computer Science and Network Security* 173 (2017). URL: https://www.researchgate.net/profile/harrison-bhatti/publication/318816158_an_introduction_to_docker_and_analysis_of_its_performance (besucht am 04.02.2023).
- [BD94] Philip A. Bernstein und Umeshwar Dayal. „An Overview of Repository Technology“. In: *Proceedings of the 20th VLDB Conference*. 1994, S. 705–713. URL: https://www.researchgate.net/profile/philip-bernstein/publication/221310590_an_overview_of_repository_technology (besucht am 03.02.2023).
- [BR15] Peter Brichzin und Thomas Rau. „Repositories zur Unterstützung von kollaborativen Arbeiten in Softwareprojekten“. In: *Gallenbacher, J. (Hrsg.), Informatik allgemeinbildend begreifen*. 2015, S. 73–82. URL: <https://dl.gi.de/handle/20.500.12116/2027> (besucht am 04.02.2023).

- [Bri14] Caius Brindescu u. a. „How do centralized and distributed version control systems impact software changes?“ In: *ICSE 2014: Proceedings of the 36th International Conference on Software Engineering*. New York, NY, USA: Association for Computing Machinery (ACM), 2014, S. 322–333. ISBN: 9781450327565. DOI: 10.1145/2568225.2568322. URL: <https://dl.acm.org/doi/10.1145/2568225.2568322> (besucht am 04.02.2023).
- [BRM21] Sejal Bhavsar, Jimit Rangras und Kirit Modi. „Automating Container Deployments Using CI/CD“. In: *Data Science and Intelligent Applications*. Springer Singapore, 2021, S. 423–429. ISBN: 978-981-15-4474-3. DOI: 10.1007/978-981-15-4474-3_{\textunderscore}47. URL: https://link.springer.com/chapter/10.1007/978-981-15-4474-3_47 (besucht am 04.02.2023).
- [Bro07] Manfred Broy u. a. „Ein Requirements-Engineering-Referenzmodell“. In: *Informatik-Spektrum* 30.3 (2007), S. 127–142. ISSN: 1432-122X. DOI: 10.1007/s00287-007-0149-5. URL: https://idp.springer.com/authorize/casa?redirect_uri=https://link.springer.com/article/10.1007/s00287-007-0149-5&%20casa_token=nqh9j1n8vegaaaaa:4fnqmkp-d-hxelm3rlyamjcexae7nhk8zk9eynrkjc-nhrbg-zi2dbhhlx (besucht am 04.02.2023).
- [Bro19] Nicolas Brousse. „The issue of monorepo and polyrepo in large enterprises“. In: *Programming '19: Companion Proceedings of the 3rd International Conference on the Art, Science, and Engineering of Programming*. Association for Computing Machinery, 2019, S. 1–4. ISBN: 9781450362573. DOI: 10.1145/3328433.3328435. (Besucht am 04.02.2023).
- [BTV18] Gleison Brito, Ricardo Terra und Marco Tulio Valente. „Monorepos: A Multivocal Literature Review“. In: *CoRR* 2018 (22.10.2018). DOI: 10.48550/arXiv.1810.09477. URL: <https://arxiv.org/pdf/1810.09477.pdf> (besucht am 04.02.2023).
- [HRN06] Humble Jez, Chris Read und Dan North. „The Deployment Production Line“. In: *AGILE 2006 (AGILE'06)*. IEEE, 2006. DOI: 10.1109/AGILE.2006.53. (Besucht am 04.02.2023).
- [IEE18] IEEE. *ISO/IEC/IEEE International Standard - Systems and software engineering – Life cycle processes – Requirements engineering*. Piscataway, NJ, USA, 2018. URL: <https://ieeexplore.ieee.org/document/8559686> (besucht am 04.02.2023).

- [Jas18] Ciera Jaspan u. a. „Advantages and Disadvantages of a Monolithic Repository: A Case Study at Google“. In: *ICSE '18: 40th International Conference on Software Engineering*. New York, NY, USA: Association for Computing Machinery (ACM), 2018, S. 225–234. ISBN: 978-1-4503-5659-6. DOI: 10.1145/3183519.3183550. (Besucht am 04.02.2023).
- [Jaz07] Mehdi Jazayeri. „Some Trends in Web Application Development“. In: *Future of Software Engineering FOSE '07*. IEEE, 2007, S. 199–213. ISBN: 0-7695-2829-5. DOI: 10.1109/FOSE.2007.26. (Besucht am 03.02.2023).
- [Kim05] Won Kim. „On Metadata Management Technology: Status and Issues“. In: *The Journal of Object Technology* 4.2 (2005), S. 41–47. ISSN: 1660-1769. DOI: 10.5381/jot.2005.4.2.c4. URL: https://www.jot.fm/issues/issue_2005_03/column4.pdf (besucht am 03.02.2023).
- [Kub21] Iván Kubinyi. „Building an effective CI/CD pipeline for a content management system“. Master’s Programme in Information Networks. Aalto University. School of Science, 2021. URL: <https://aaltodoc.aalto.fi/handle/123456789/112632> (besucht am 04.02.2023).
- [Maj17] Rana Majumdar u. a. „Source code management using version control system“. In: *2017 6th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO)*. IEEE, 2017, S. 278–281. DOI: 10.1109/icrito.2017.8342438. (Besucht am 04.02.2023).
- [MG11] Peter Mell und Timothy Grance. *The NIST definition of cloud computing*. NIST Special Publications (NIST SP) 800-145. Gaithersburg, MD: National Institute of Standards and Technology, 28. Sep. 2011. DOI: 10.6028/NIST.SP.800-145. URL: <https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf> (besucht am 03.02.2023).
- [MH06] Russ Miles und Kim Hamilton. *Learning UML 2.0*. 1. Aufl. Sebastopol, Calif.: O’Reilly Media, 2006. ISBN: 0596009828.
- [Ott09] Stefan Otte. *Version Control Systems*. 2009. URL: <https://citeseerx.ist.psu.edu/pdf/5093c4b55ad419379ad22e0c60008f133a2b6e3b> (besucht am 04.02.2023).
- [Pah15] Claus Pahl. „Containerization and the PaaS Cloud“. In: *IEEE Cloud Computing* 2.3 (2015), S. 24–31. ISSN: 2325-6095. DOI: 10.1109/mcc.2015.51. (Besucht am 04.02.2023).

- [Pav20] Andrey Pavlenko u. a. „Micro-frontends: application of microservices to web front-ends“. In: *Journal of Internet Services and Information Security* (2020), S. 49–66. DOI: 10.22667/JISIS.2020.05.31.049. URL: <https://jisis.org/wp-content/uploads/2022/11/jisis-2020-vol10-no2-04.pdf> (besucht am 03.02.2023).
- [PL16] Rachel Potvin und Josh Levenberg. „Why Google stores billions of lines of code in a single repository“. In: *Communications of the ACM* 59.7 (2016), S. 78–87. ISSN: 0001-0782. DOI: 10.1145/2854146. URL: <https://dl.acm.org/doi/pdf/10.1145/2854146> (besucht am 04.02.2023).
- [Poh07] Klaus Pohl. *Requirements engineering: Grundlagen, Prinzipien, Techniken*. 1. Aufl. Heidelberg: dpunkt.verl., 2007. ISBN: 9783898643429.
- [Pot20] Amit M. Potdar u. a. „Performance Evaluation of Docker Container and Virtual Machine“. In: *Procedia Computer Science* 171 (2020), S. 1419–1428. ISSN: 1877-0509. DOI: 10.1016/j.procs.2020.04.152. URL: <https://www.sciencedirect.com/science/article/pii/S1877050920311315> (besucht am 04.02.2023).
- [Ran20] Thorsten Ragnau u. a. „Continuous Security Testing: A Case Study on Integrating Dynamic Security Testing Tools in CI/CD Pipelines“. In: *2020 IEEE 24th International Enterprise Distributed Object Computing Conference*. Piscataway, NJ: IEEE, 2020, S. 145–154. ISBN: 978-1-7281-6473-1. DOI: 10.1109/edoc49727.2020.00026. (Besucht am 04.02.2023).
- [Rup07] Chris Rupp. *Requirements-Engineering und -Management: Professionelle, iterative Anforderungsanalyse für die Praxis*. 4., aktualisierte und erw. Aufl. München und Wien: Carl Hanser, 2007. ISBN: 3446405097.
- [Sha17] Mojtaba Shahin u. a. „Beyond Continuous Delivery: An Empirical Investigation of Continuous Deployment Challenges“. In: *11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. Piscataway, NJ: IEEE, 2017, S. 111–120. ISBN: 978-1-5090-4039-1. DOI: 10.1109/esem.2017.18. (Besucht am 04.02.2023).
- [Sin19] Charanjot Singh u. a. „Comparison of Different CI/CD Tools Integrated with Cloud Platform“. In: *Proceedings of the 9th International Conference on Cloud Computing, Data Science and Engineering - Confluence 2019*. Piscataway, NJ: IEEE, 2019. ISBN: 978-1-5386-5933-5b. DOI: 10.1109/confluence.2019.8776985. (Besucht am 04.02.2023).

- [SK10] Ilango Sriram und Ali Khajeh-Hosseini. *Research Agenda in Cloud Technologies*. arXiv, 2010. DOI: 10.48550/arXiv.1001.3259. (Besucht am 03.02.2023).
- [TAM18] Yudai Tanabe, Tomoyuki Aotani und Hidehiko Masuhara. „A Context-Oriented Programming Approach to Dependency Hell“. In: *Proceedings of the 10th International Workshop on Context-Oriented Programming Advanced Modularity for Run-time Composition*. ACM Other conferences. New York, NY: Association for Computing Machinery (ACM), 2018, S. 8–14. ISBN: 9781450357227. DOI: 10.1145/3242921.3242923. (Besucht am 04.02.2023).
- [Ucc10] Kristan Uccello. „Nx, NestJs, React — Docker Deploys - The Startup - Medium“. In: *The Startup* 2020 (2020-12-10). URL: <https://medium.com/swlh/nx-nestjs-react-docker-deploys-928a55fc19fd> (besucht am 05.02.2023).
- [UK18] Serhat Uzunbayir und Kaan Kurtel. „A Review of Source Code Management Tools for Continuous Software Development: Bitte das ohne Text zitieren“. In: *2018 3rd International Conference on Computer Science and Engineering (UBMK)*. IEEE, 2018, S. 414–419. DOI: 10.1109/UBMK.2018.8566644.
- [WNZ16] Ingo Weber, Surya Nepal und Liming Zhu. „Developing Dependable and Secure Cloud Applications“. In: *IEEE Internet Computing* 20.3 (2016), S. 74–79. ISSN: 1089-7801. DOI: 10.1109/mic.2016.67. (Besucht am 04.02.2023).
- [Zag15] Alexey Zagalsky u. a. „The Emergence of GitHub as a Collaborative Platform for Education“. In: *CSCW '15: Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing*. New York, NY, USA: Association for Computing Machinery (ACM), 2015, S. 1906–1917. ISBN: 9781450329224. DOI: 10.1145/2675133.2675284. (Besucht am 04.02.2023).
- [Zam21] Fiorella Zampetti u. a. „CI/CD Pipelines Evolution and Restructuring: A Qualitative and Quantitative Study“. In: *2021 IEEE International Conference on Software Maintenance and Evolution*. Piscataway, NJ: IEEE, 2021, S. 471–482. ISBN: 978-1-6654-2882-8. DOI: 10.1109/ICSME52107.2021.00048. (Besucht am 04.02.2023).
- [ZND18] Nazatul Nurlisa Zolkifli, Amir Ngah und Aziz Deraman. „Version Control System: A Review“. In: *Procedia Computer Science* 135 (2018), S. 408–415. ISSN: 1877-0509. DOI: 10.1016/j.procs.2018.08.191. (Besucht am 04.02.2023).

Onlinereferenzen

- [Ang23] AngularJS. *Developer Guide: Introduction*. 2023. URL: <https://docs.angularjs.org/guide/introduction> (besucht am 05.02.2023).
- [Ari19] Jagath Ariyaratne. *On-premise, Cloud or Hybrid?* 2019. URL: <https://medium.com/@jagathsisira/on-premise-cloud-or-hybrid-5708f5e1fd1a> (besucht am 04.02.2023).
- [But23] Corey Butler. *nvm-windows: A node.js version management utility for Windows. Ironically written in Go*. 2023. URL: <https://github.com/coreybutler/nvm-windows> (besucht am 05.02.2023).
- [Doc23a] Docker. *What is Docker Hub? | Docker*. 2023. URL: <https://www.docker.com/products/docker-hub/> (besucht am 05.02.2023).
- [Doc23b] Docker Docs. *Overview*. 2023. URL: <https://docs.docker.com/compose/> (besucht am 05.02.2023).
- [Doc23c] Docker Documentation. *Docker Desktop*. 2023. URL: <https://docs.docker.com/desktop/> (besucht am 05.02.2023).
- [Ede11] Norbert Eder. *Das Repository Pattern anhand eines Beispiels inkl. Tests*. 2011. URL: <https://norberteder.com/das-repository-pattern-anhand-eines-beispiels-inkl-tests/> (besucht am 03.02.2023).
- [FL18] Christoph Fehling und Frank Leymann. *Hardware-Virtualisierung*. 2018. URL: <https://wirtschaftslexikon.gabler.de/definition/hardware-virtualisierung-53364/version-276457> (besucht am 04.02.2023).
- [Fra20] Jake Frankenfield. *Business Logic: Definition, Benefits, and Example*. Hrsg. von Investopedia. 2020. URL: <https://www.investopedia.com/terms/b/businesslogic.asp> (besucht am 03.02.2023).
- [Gil18] Alexander Gillis. *staging environment*. 2018. URL: <https://www.techtarget.com/searchsoftwarequality/definition/staging-environment> (besucht am 04.02.2023).

- [Git23a] Git. *Git*. 2023. URL: <https://git-scm.com/> (besucht am 05.02.2023).
- [Git23b] Git. *git-add Documentation*. 2023. URL: <https://git-scm.com/docs/git-add> (besucht am 05.02.2023).
- [Git23c] Git. *git-commit Documentation*. 2023. URL: <https://git-scm.com/docs/git-commit> (besucht am 05.02.2023).
- [Git23d] Git. *git-push Documentation*. 2023. URL: <https://git-scm.com/docs/git-push> (besucht am 05.02.2023).
- [Git23e] GitHub Docs. *Changing the default branch - GitHub Docs*. 2023. URL: <https://docs.github.com/en/repositories/configuring-branches-and-merges-in-your-repository/managing-branches-in-your-repository/changing-the-default-branch> (besucht am 05.02.2023).
- [Git23f] GitHub Docs. *Ignorieren von Dateien - GitHub Docs*. 2023. URL: <https://docs.github.com/de/get-started/getting-started-with-git/ignoring-files> (besucht am 05.02.2023).
- [Git23g] GitHub Docs. *Informationen zu README-Dateien - GitHub Docs*. 2023. URL: <https://docs.github.com/de/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/about-readmes> (besucht am 05.02.2023).
- [Git23h] GitHub Docs. *Informationen zu Workflows*. 2023. URL: <https://docs.github.com/de/actions/using-workflows/about-workflows> (besucht am 05.02.2023).
- [Git23i] GitHub Docs. *Managing a branch protection rule - GitHub Docs*. 2023. URL: <https://docs.github.com/en/repositories/configuring-branches-and-merges-in-your-repository/defining-the-mergeability-of-pull-requests/managing-a-branch-protection-rule> (besucht am 05.02.2023).
- [Git23j] GitHub Docs. *Pushing commits to a remote repository*. 2023. URL: <https://docs.github.com/en/get-started/using-git/pushing-commits-to-a-remote-repository> (besucht am 05.02.2023).
- [Git23k] GitHub Docs. *Set up Git - GitHub Docs*. 2023. URL: <https://docs.github.com/en/get-started/quickstart/set-up-git> (besucht am 05.02.2023).
- [Git23l] GitHub Docs. *Understanding GitHub Actions*. 2023. URL: <https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions> (besucht am 05.02.2023).

- [Hal19] David Hallinan. *Cloud vs. On-Premise Software Deployment – What’s Right for You?* 2019. URL: <https://dzone.com/articles/cloud-vs-on-premise-software-deployment-whats-righ> (besucht am 04.02.2023).
- [Her22a] Heroku Dev Center. *Building Docker Images with heroku.yml*. 2022. URL: <https://devcenter.heroku.com/articles/build-docker-images-heroku-yml> (besucht am 05.02.2023).
- [Her22b] Heroku Dev Center. *Dynos and the Dyno Manager*. 2022. URL: <https://devcenter.heroku.com/articles/dynos> (besucht am 05.02.2023).
- [Her22c] Heroku Dev Center. *How Heroku Works*. 2022. URL: <https://devcenter.heroku.com/articles/how-heroku-works> (besucht am 05.02.2023).
- [Her23] Heroku. *What is Heroku?* 2023. URL: <https://www.heroku.com/about#:~:text=Heroku%20is%20a%20container%2Dbased,getting%20their%20apps%20to%20market> (besucht am 05.02.2023).
- [Ley18] Frank Leymann. *Repository*. 2018. URL: <https://wirtschaftslexikon.gabler.de/definition/repository-52691/version-275809> (besucht am 03.02.2023).
- [LS18] Richard Lackes und Markus Siepermann. *virtuelle Maschine*. 2018. URL: <https://wirtschaftslexikon.gabler.de/definition/virtuelle-maschine-48523/version-271775> (besucht am 04.02.2023).
- [Nes23] NestJS Docs. *Introduction*. 2023. URL: <https://docs.nestjs.com/> (besucht am 05.02.2023).
- [NGI23] NGINX. *What is NGINX? - NGINX*. 2023. URL: <https://www.nginx.com/resources/glossary/nginx/> (besucht am 05.02.2023).
- [npm23a] npm Docs. *About npm / npm Docs*. 2023. URL: <https://docs.npmjs.com/about-npm> (besucht am 05.02.2023).
- [npm23b] npm Docs. *About packages and modules / npm Docs*. 2023. URL: <https://docs.npmjs.com/about-packages-and-modules> (besucht am 05.02.2023).
- [npm23c] npm Docs. *Downloading and installing Node.js and npm / npm Docs*. 2023. URL: <https://docs.npmjs.com/downloading-and-installing-node-js-and-npm> (besucht am 05.02.2023).
- [Nx23a] Nx. *@nrwl/angular / Nx*. 2023. URL: <https://nx.dev/packages/angular> (besucht am 05.02.2023).

- [Nx23b] Nx. *@nrwl/nest / Nx*. 2023. URL: <https://nx.dev/packages/nest> (besucht am 05.02.2023).
- [Nx23c] Nx. *Explore the Graph*. 2023. URL: <https://nx.dev/core-features/explore-graph> (besucht am 05.02.2023).
- [Nx23d] Nx. *Getting Started with Package-Based Repos*. 2023. URL: <https://nx.dev/getting-started/package-based-repo-tutorial> (besucht am 05.02.2023).
- [Nx23e] Nx. *How Affected Works*. 2023. URL: <https://nx.dev/concepts/affected> (besucht am 05.02.2023).
- [Nx23f] Nx. *Integrated Repos vs. Package-Based Repos vs. Standalone Apps*. 2023. URL: <https://nx.dev/concepts/integrated-vs-package-based> (besucht am 05.02.2023).
- [Nx23g] Nx. *Node Tutorial - Part 1: Code Generation*. 2023. URL: <https://nx.dev/getting-started/node-tutorial> (besucht am 05.02.2023).
- [Nx23h] Nx. *Nx Plugins Registry*. 2023. URL: <https://nx.dev/community#plugin-directory> (besucht am 05.02.2023).
- [Nx23i] Nx. *Share Your Cache*. 2023. URL: <https://nx.dev/core-features/share-your-cache> (besucht am 05.02.2023).
- [Nx23j] Nx. *What is Nx Cloud?* 2023. URL: <https://nx.dev/nx-cloud/intro/what-is-nx-cloud> (besucht am 05.02.2023).
- [Pet21] Horst Peterjohann. *Satzschablonen*. 2021. URL: <https://www.peterjohann-consulting.de/satzschablonen/> (besucht am 04.02.2023).
- [Rea21] Ben Read. *6 reasons why we chose Nx as our monorepo management tool*. 2021. URL: <https://medium.com/purplebricks-digital/6-reasons-why-we-chose-nx-as-our-monorepo-management-tool-1fe5274a008e> (besucht am 05.02.2023).
- [Sal20] Anniina Sallinen. *Deploying containerized NginX to Heroku - how hard can it be?* 2020. URL: <https://dev.to/levelupkoodarit/deploying-containerized-nginx-to-heroku-how-hard-can-it-be-3g14> (besucht am 05.02.2023).
- [Ucc10] Kristan Uccello. „Nx, NestJs, React — Docker Deploys - The Startup - Medium“. In: *The Startup* 2020 (2020-12-10). URL: <https://medium.com/swlh/nx-nestjs-react-docker-deploys-928a55fc19fd> (besucht am 05.02.2023).

-
- [War20] Chris Ward. *Continuous Integration (CI) vs. Continuous Delivery (CD) vs. Continuous Deployment (CD)*. Hrsg. von humanitec. 2020. URL: <https://humanitec.com/blog/continuous-integration-vs-continuous-delivery-vs-continuous-deployment> (besucht am 04.02.2023).
- [War22] Abhilash Warriar. *Containers vs Virtual Machines (VMs)*. 2022. URL: <https://www.eginnovations.com/blog/containers-vs-vms/> (besucht am 04.02.2023).

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel verfasst habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Berlin, den 05.02.2023

A handwritten signature in black ink, appearing to read 'I. Birkmaier' with a checkmark-like flourish at the end.

Ivan Birkmaier