CANCOM

FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Fachhochschule Aachen

Fakultät
Technomathematik und Medizintechnik

Studium
Angewandte Mathematik und Informatik B.Sc.

Anforderungsanalyse zur Erstellung einer automatisch generierten und isolierten Integrationstestumgebung zur Qualitätssicherung in der Feature-Entwicklung

---

Seminararbeit
Frederik Naeven
Matrikelnummer: 3529766

Aachen, 26. Januar 2024

Abstract

Kontinuierliches und automatisiertes Testen spielt in der agilen Softwareentwicklung eine besonders wichtige Rolle bei der Qualitätssicherung. Deshalb soll eine automatisierte Integrationstestumgebung dafür sorgen, dass einzelne Softwarekomponenten wie erwartet miteinander interagieren und bei jeder Entwicklungsiteration für eine grundlegende Zuverlässigkeit und Qualität der Software sorgen. Mit modernen Anforderungsanalysetechniken wurden hier die Anforderungen an eine solche Integrationstestumgebung herausgearbeitet. Dies beinhaltet auch eine ausführliche Befragung der Stakeholder.
Die zu testende Software ist eine umfangreiche Webanwendung, die aus mehreren Komponenten besteht und wiederum selbst an weitere externe Services und Komponenten angebunden ist. Die Implementierung soll in einer Gitlab-Pipeline stattfinden und bei jeder Neuerung auf dem entsprechenden Feature-Branch die Integrationstests ausführen und die Ergebnisse festhalten. Mit dem Framework Cypress sollen die Tests erstellt werden. Zentrale Qualitätsanforderungen sind ein einfaches Datenmanagement der Testdatenbank, eine einfache Wartbarkeit der Integrationstests, eine annehmbare Reaktionszeit und eine hohe Zuverlässigkeit bei der Ausführung der Tests. Funktionale Anforderungen wurden ebenfalls ermittelt und mit Unterstützung des Kanomodells priorisiert.
Abschließend wurden mögliche Lösungsansätze erfasst und diskutiert. Die Testumgebung könnte entweder mit der Docker-in-Docker Technologie in dem Container der GitLab Pipeline aufgebaut werden oder in der in einem externen System. Die Docker-in-Docker Lösung hätte einen geringeren Implementationsaufwand. Jedoch könnte es zu Performance Problemen kommen. Um die optimale Lösung zu finden empfiehlt sich als nächster Schritt vor der Implementation zunächst eine Machbarkeitsstudie.

Inhaltsverzeichnis

1. Einleitung

1.1 Projektziele und -zweck

Die zu testende Software ist ein Web-Portal, dass unter Anderem für das Identity-Management und die User-Rechte Verwaltung von Organisationen verantwortlich ist. Die Software setzt sich aus mehreren Webapplikationen zusammen, wobei die meisten aus einem Spring-Java-Backend und einem Angular-Frontend bestehen. Ziel der Automatisierung ist den Arbeitsprozess der Entwickler zu entlasten und gleichzeitig die Qualität der Entwicklung zu steigern durch automatisierte Integrationstests. Zum Zeitpunkt dieser Analyse existiert für das zu testende Projekt bereits eine minimale CI/CD Pipeline, die Unit-Tests ausführt und überprüft, sowie einige QA-Tools. Integrationstests werden bisher allerdings nur von Hand ausgeführt. Durch die Automatisierung soll der Arbeitsprozess um einige dieser manuellen Tests verkürzt werden und die Qualität mit der automatischen Erstellung von Testberichten gesichert werden. Erfolgreich ist die Automatisierung dann, wenn sie den Entwicklungsprozess beschleunigt, sowie die Qualität der Entwicklung gesteigert wird.

Integrationstests

Integrationstests sind Tests, die dann ausgeführt werden, nachdem Unit-Tests die korrekte Ausführung von individuellen Komponenten in Isolation verifiziert haben. Ziel ist es mit Integrationstests sicherzustellen, dass Module und Interface auch untereinander korrekt zusammenarbeiten, denn es kann passieren, dass Module, die in Isolation funktionieren in Zusammenarbeit mit anderen Modulen versagen. Entwickler können daraufhin den Grund für die fehlerhafte Zusammenarbeit analysieren und so lange nachbessern, bis das System einwandfrei funktioniert [vgl. 8, S. 50].

Integrationstests können auf unterschiedlichen Ebenen ausgeführt werden. Hier unterscheidet man zwischen Komponenten-Level-, System-Level- und Ende-zu-Ende-Integration. Dabei werden bei der Komponenten-Level Integration nur einzelne Software-

Komponenten innerhalb einer Applikation untereinander getestet. Im Gegensatz dazu werden bei der System-Level-Integration mehrere Komponenten von Subsystemen untereinander getestet. Bei Ende-zu-Ende-Integration wird die Integration der gesamten Applikation, inklusive aller Subsysteme und externen Systeme, mit denen die Applikation interagiert, getestet [vgl. 7, S. 9-10].

Ziel der hier angestrebten automatisierten Integrationstestumgebung ist System-Level-Integration und nicht Ende-zu-Ende-Integration. Es sollen Teilsysteme, wie zusammengehörige Backends und Frontends mit den entsprechenden Datenbanken, getestet werden. Eine Ende-zu-Ende-Integration ist hier nicht vorgesehen, da diese in ihrer Gesamtheit auf einem externen Testsystem stattfindet, das auch an verwendete Drittsysteme angebunden ist.

2. Grundlagen der Anforderungsanalyse

2.1 Was sind Anforderungen?

> „Anforderungen sind im Grunde nichts anderes als die Beschreibung eines IT-Systems bevor es existiert.“ [siehe 4, S. 2]

Bevor ein System entwickelt werden kann, muss es beschrieben werden. Eine Beschreibung durch Anforderungen erfordert zwei Perspektiven, einmal aus dem Problemraum und einmal aus dem Lösungsraum. Anforderungen aus dem Problemraum beschreiben ein Problem, dass von den Experten mit dem neuen System gelöst werden sollen. Die Anforderungen aus dem Lösungsraum sind die Anforderungen an die technische Lösung [vlg. 4, S. 3].

2.2 Sinn und Bedeutung der Anforderungsanalyse

Die Anforderungsanalyse (auch „Requirements Engineering“) soll ein Problemverständnis für Projekte schaffen und macht es auch möglich die Qualität und Vollständigkeit der Umsetzung messbar und testbar zu machen. Neben diesen wesentlichen Eigenschaften ist sie ebenfalls notwendig zur ersten Kostenabschätzung während der Projektplanung. Requirements Engineering vermindert das Risiko ein fehlerhaftes System zu entwickeln oder den Kundenwunsch zu verfehlen [vlg. 9, S. 7]. Außerdem können widersprüchliche Anforderungen von Stakeholdern durch die Analyse noch vor der Entwicklung aufgedeckt werden [vlg. 4, S. 14].

2.3 Quellen von Anforderungen

Es gibt vier Quellen für Anforderungen: Dokumente, Stakeholder, bestehende IT-Systeme und das Projektumfeld, beziehungsweise den Systemkontext [vgl. 10, S. 46]. Verfügbare Dokumente können die Grundlage der Anforderungen bilden. Daraus kann man Normen, Standards, Gesetze, Firmenvorgaben, Wissen aus Fachliteratur, technische Spezifikationen und noch mehr ermitteln [vgl. 4, S. 36]. Baut man auf einem bereits bestehenden System auf oder hat das System einen Vorgänger, ermittelt man daraus Informationen über den aktuellen Zustand des Systems und aus bereits bestehenden ähnlichen Systemen lassen sich ebenfalls Anforderungen ableiten. Aus dem Systemkontext leiten sich unter anderem Anforderungen an die Systemschnittstellen, Risiken und Restriktionen ab. Die wichtigste Quelle jedoch sind die Stakeholder [vgl. 10, S. 46ff].

Stakeholder

> „Ein Stakeholder ist eine Person oder Organisation, die Einfluss auf die Anforderungen des Systems hat oder auf die das System Auswirkungen hat.“ [Siehe 5, „Stakeholder“]

Stakeholder sind damit Quellen von Anforderungen und möglicherweise auch Teil der System-Umgebung [vlg. 4, S. 54]. Um also möglichst genaue und vollständige Anforderungen zu ermitteln, ist es zwingend notwendig, dass man zunächst alle potenziellen Stakeholder identifiziert.
Potenzielle Stakeholder lassen sich mit der Checkliste aus folgenden Kategorien [siehe 4, S. 55] ermitteln:

- direkte Benutzer des Systems,
- indirekte Nutznießer, die das System selbst nicht benutzen,
- Sicherheitsbeauftragter, Betriebsrat
- Händler und Vertriebspartner
- Gegner, Konkurrenten, Hacker,
- Trainer, Administratoren, Benutzerunterstützung (Hotline),
- Entwicklungs- und Wartungsteam,

- Gesetzgeber,
- Presse, öffentliche Meinung.

2.4 Arten von Anforderungen nach dem Kano-Modell

Das Kano-Modell, nach dem japanischen Professor Noriaki Kano [1, vgl.], teilt Anforderungen in drei unterschiedliche Kategorien ein:

- Basisfaktoren
- Leistungsfaktoren
- Begeisterungsfaktoren

Dabei sind die **Basisfaktoren** Anforderungen, die selbstverständlich erscheinen und zwingend notwendig sind. Diese Faktoren müssen vollständig erfüllt sein, um die Stakeholder zufrieden stellen zu können. Bei der Ermittlung von Basisfaktoren von Stakeholdern kann es dabei oft zu Schwierigkeiten kommen, da diese Faktoren häufig als so selbstverständlich angesehen werden, sodass die Stakeholder diese schon gar nicht mehr erwähnen. Daher muss man sich bei der Ermittlung von Anforderungen dieser Faktoren stets bewusst sein und gegebenenfalls nachfragen [vgl. 10, S. 59]. Bereits bestehende Systeme eignen sich sehr gut, um diese Art von Anforderungen zu ermitteln [vgl. 4, S. 33].

Am einfachsten zu erheben sind die **Leistungsfaktoren** von Stakeholdern. Es können funktionale sowie qualitative Anforderungen an das System sein. Sie werden explizit von den Stakeholdern genannt und sind der Grund, warum das Projekt überhaupt beauftragt wird [vgl. 4, S. 28].

Es ist unmöglich **Begeisterungsfaktoren** von Stakeholdern zu ermitteln, denn sie sind positive Überraschungen, die den Anwender begeistern. Sie können das System außergewöhnlich machen. Begeisterungsfaktoren kann man zum Beispiel mit kreativen Erhebungstechniken erlangen oder sich bei innovativen Konkurrenz Produkten abschauen [vgl. 10, S. 60] & [vgl. 9, S. 126].

Priorisierung von Anforderungen

Bei der Priorisierung von Anforderungen gibt es zwei mögliche Ansätze, die Ad-hoc- und die analytische Priorisierungstechnik. Ein wesentlicher Unterschied ist der Aufwand der für die diese Arten der Priorisierungstechnik benötigt wird. In vielen Projekten sind die Techniken der Ad-hoc Priorisierung mit geringerem Aufwand ausreichend. Zu diesen Techniken gehören üblicher Weise zum Beispiel die Kano-Analyse oder der ähnlichen MoSCoW-Priorisierung (Must have, Should have, Could have, Won't have) [vgl. 9, S. 207]. Sind Priorisierungen mit diesen Techniken nicht ausreichend nachvollziehbar sollte versuchte werden, diese mit analytischen Methoden zu verbessern. Ein Beispiel dafür wäre eine Kosten-Wert-Analyse [vgl. 9, S. 207].

Auf eine nähere Ausführung der Kosten-Wert-Analyse wird an dieser Stelle verzichtet. Sie kann in [6] nachgelesen werden.

2.5 Methodik dieser Anforderungsanalyse

Diese Anforderungsanalyse basiert vorwiegend auf den Methoden des Lehrbuches *Grundlagen der Anforderungsanalyse* [4]. Die Struktur und der Aufbau dieses Dokuments folgt, angepasst an den Rahmen dieser Seminararbeit, der Vorlage des IREB [siehe 4, S. 132 ff.]. Sie ist eine

> „leichtgewichtige Spezifikation [...], die alles Wichtige enthält, ohne Inhalte zu doppeln."

[siehe 4, S. 132].

Ermittlung der Anforderungen von Stakeholdern

Zur Ermittlung der Anforderungen von Stakeholdern sind übliche Methoden das Interview oder Workshops [siehe 4, S. 63 ff]. Strukturierte Workshops eignen sich besonders gut zur Ermittlung von Anforderungen von Stakeholdern und werden gegenüber Interviews empfohlen. Da die funktionalen Anforderungen in diesem Fall allerdings von einer recht kleinen Gruppe von Stakeholdern, dem Entwickler-Team, vertreten durch die Lead-Developer, und der Softwaretester kommt, werden diese Anforderungen aus zeitlichen Gründen zunächst in Interviews ermittelt.

3. Übersicht

In diesem Kapitel wird eine kurze Übersicht über das System, die Stakeholdern und die Anforderungen, die in den folgenden Kapiteln näher beschrieben werden, gegeben.

3.1 Systemumfang

Das System umfasst den automatisierten Aufbau einer Serverumgebung, auf der die jeweiligen Frontends und Backends und deren benötigten Komponenten, wie zum Beispiel Datenbanken, deployed werden und daraufhin automatische Integrationstests ausgeführt werden. Das Ergebnis dieser Tests soll dokumentiert werden und die Umgebung daraufhin wieder abgebaut werden. Die Tests müssen vorher definiert werden und deren Erstellung ist nicht Teil des Systems. Die Automatisierung findet im Rahmen der Pipeline-Ausführung im Git-Lab statt.

3.2 Stakeholder

Name	Position (in der Firma)	Rolle (im Projekt)	Kontaktdaten
M*	Software-architekt	Auftraggeber & Nutzer	m*@cancom.de
B*	Software-entwickler	Auftraggeber & Nutzer	b*@cancom.de
S*	Application Support Specialist	Nutzer	s*@cancom.de
Frederik Naeven	Software-entwickler	Entwickler	frederik.naeven @cancom.de

Tabelle 3.1: Stakeholder-Übersicht, zensiert für die öffentliche Version

3.3 Systemarchitektur

Das System besteht aus mehreren Komponenten. Die automatischen Integrationstests sollen während einer Pipeline in Gitlab ausgeführt werden. Die Integrationstests müssen so abgespeichert sein, dass sie für die Integrationstestumgebung und ebenfalls für ein externes Testsystem, dass nicht zu diesem System gehört, genutzt werden können.

Da die zu testenden Software Webapplikationen sind und das Entwickler-Team bereits Erfahrungen damit gesammelt haben, sind für die Ausführung der Integrationstests *Cypress*-Tests vorgesehen.

3.4 Systemkontext, Randbedingungen, Annahmen

Die Automatisierung findet für jeden Merge-Request statt und die Integrationstests müssen für ein externes generelles Testsystem wiederverwendbar sein. Die ausgeführten Tests sind *Cypress*-Tests.

Das folgende Kontextdiagramm stellt die Schnittstellen der Automatisierung dar:

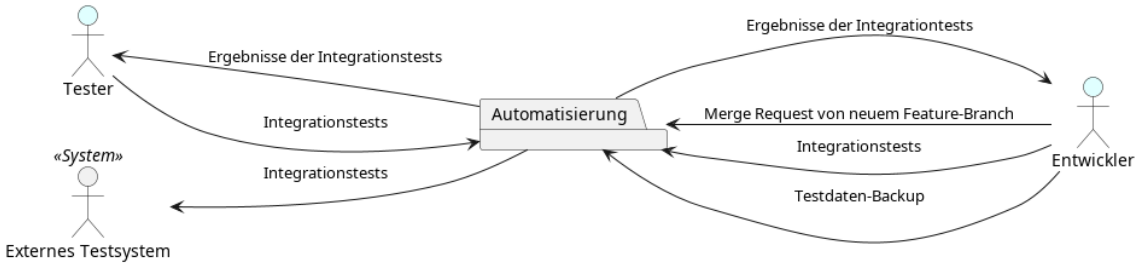

Abbildung 3.1: Systemkontext

Die zu testenden Komponenten der Webanwendungen werden bereits für das Deployment auf dem externen generellen Testsystem als Docker-Image gebaut. Dazu wurden bereits in den Projekten entsprechend benötigte Dockerfiles angelegt. In der Integrationstestumgebung sollen ebenfalls Images getestet werden. Da Images bisher nur bei der Erstellung eines neuen Release-Tags generiert werden, müssen vor dem Einrichten der Testumgebung die entsprechenden Komponenten zunächst als Docker-Images erstellt werden.

Cypress Test-Framework

Das *Cypress Test-Framework* ermöglicht die Automatisierung von Tests, um sicherzustellen, dass Webanwendungen wie erwartet funktionieren. Es ist in der Lage Benutzerinteraktionen zu simulieren und die Ergebnisse zu überprüfen. Zusätzlich erleichtern diese Tests unter Anderem den Debugging-Workflow durch automatisch aufgezeichnete Screenshots, Videos und Test-Replays [vgl. 2, Absatz "Features"].

3.5 Nutzer und Zielgruppen

Name der Rolle	**Entwickler**
Beschreibung	Entwickelt Features für die Webapplikationen
Ziel der Rolle	Möchte schnell Feedback über seine entwickelten Features
Wissen/Erfahrung/Fähigkeiten	Professionelle Kenntnisse mit Programmieren und Gitlab

Name der Rolle	**Tester**
Beschreibung	Testet Features für die Webapplikationen
Ziel der Rolle	Möchte vor dem Testen die Ergebnisse der automatisierten Tests kennen und möchte eventuell manche Tests automatisieren.
Wissen/Erfahrung/Fähigkeiten	Professionelle Kenntnisse mit Tests und Gitlab

Name der Rolle	**Externes Testsystem**
Beschreibung	Allgemeine Testumgebung für das gesamte Portal mit sämtlichen Komponenten auf dem auch manuell getestet wird.
Ziel der Rolle	Möchte, dass vor dem manuellen Testen durch Tester alle automatisierten Integrationstests ebenfalls in dieser Umgebung ausgeführt werden und benötigt deshalb Zugriff auf alle Integrationstests, die für die Integrationstests-Umgebung angelegt worden sind.
Fähigkeiten	Ist in der Lage die Integrationstests selber auszuführen.

3.6 System-Funktionalität

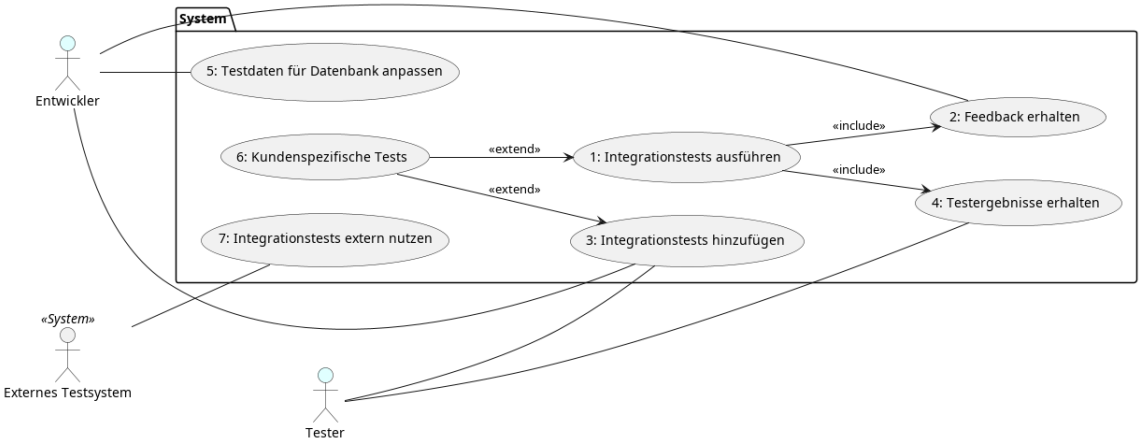

Abbildung 3.2: Use-Case-Diagramm

Im Folgenden werden mit Hilfe von User Stories die einzelnen Use Cases kurz beschrieben, um eine kurze Orientierung zu diesen zu ermöglichen. Die Use Cases werden in den folgenden Kapiteln ausführlicher ausgearbeitet.

Use Case 1: Integrationstests ausführen

Als Entwickler

möchte ich, dass automatisch Integrationstests zu meinem neue entwickelten Feature ausgeführt werden,

damit ich diese nicht selber ausführen muss und so Zeit spare.

Use Case 2: Feedback zum neu entwickelten Feature erhalten

Als Entwickler

möchte ich ein automatisiertes Feedback, ob die Integrationstests fehlerfrei durchlaufen oder welche Integrationstests fehlschlagen, sobald ich einen Merge-Request stelle,

damit ich Fehler möglichst schnell erkenne und daraufhin beheben kann.

Use Case 3: Integrationstests hinzufügen

Als Entwickler und als Tester

möchte ich Integrationstests, die automatisch ausgeführt werden sollen, hinzufügen oder bearbeiten können,

damit neu entwickelte oder angepasste Features in Zukunft automatisch getestet werden können.

Use Case 4: Testdaten für Datenbank anpassen

Als Entwickler

möchte ich möglichst einfach das Datenbank-Backup für die Testdatenbank anpassen können,

damit ich auf mögliche Datenbank Struktur Änderungen jeder Zeit reagieren kann.

Use Case 5: Kundenspezifische Tests (zukünftig)

Als Entwickler

möchte ich, dass bei kundenspezifischen Features auch kundenspezifische Integrationstests ausgeführt werden können,

damit ich mir auch bei unterschiedlichen Kunden sicher sein kann, dass weiterhin alle Features ordnungsgemäß funktionieren.

Use Case 6: Integrationstests für externes Testsystem benutzen können

Als Entwickler und Tester

möchte ich, dass die Integrationstests wiederverwendbar sind für ein Testsystem des gesamten Projekts, außerhalb dieser Automatisierung,

damit die Tests konsistent sind und nicht mehrmals erstellt werden müssen.

3.7 Ausschlüsse

Das Testen des gesamten Projekts oder mehrerer Frontends und Backends gemeinsam soll nicht in dieser Automatisierung stattfinden. Dafür sollen nur die hier bereitgestellten Integrationstests wiederverwendbar sein. Ebenfalls ausgeschlossen von der Automatisierung ist das automatische erstellen von Testdaten. Die für die Tests

benötigten Backups werden vom Entwickler Team bereitgestellt, diese müssen allerdings leicht austauschbar sein. Die bereitgestellten Backups sind bereits Datenschutz konform und sollen nicht mehr vom System überprüft werden.

3.8 Prioritäten

Den Use Cases wurde zunächst eine Kano Kategorie zugeordnet und anschließend nach Kritikalität für das System bewertet. Dabei sind die Use Cases 1 bis 4 absolute Basis-Faktoren und ebenfalls von hoher Kritikalität, da ohne sie das System gar nicht funktionieren oder zumindest das Ziel der Qualitätssicherung der Software verfehlen würde. Die Kritikalität der Use Cases 5 und 6 ist geringer, da sie nicht zwingend notwendig sind, um die Aufgabe des Systems zu erfüllen, steigern aber beim Vorhandensein den Wert der Anwendung und gehören deshalb in die Kano-Kategorie Leistung. Mit den Auftraggebern wurde außerdem vereinbart Use Case 5, die kundenspezifischen Tests, nicht mit in die erste Version aufzunehmen.

Use Case Nr.	**In Version 1 enthalten?**	**Kano-Kategorie**	**Prioritäten**	**Kritikalität**
1	Ja	Basis	hoch	hoch
2	Ja	Basis	hoch	hoch
3	Ja	Basis	hoch	hoch
4	Ja	Basis	hoch	hoch
5	Nein	Leistung	mittel	mittel
6	Ja	Leistung	mittel	mittel

Tabelle 3.2: Priorisierung der Use Cases

4. Anforderungen

4.1 Funktionsperspektive

Im Folgenden werden die Use Cases ausgearbeitet. Dabei werden die einzelnen benötigten Arbeitsschritte möglichst kleinschrittig dargestellt. Sie sind die granularen funktionalen Anforderungen an das System.

Use Case 1

Da Use Case 1 die Hauptfunktionalität des Systems ist, wird diese hier sowohl als Aktivitätsdiagramm, sowie in Textform dargestellt. Davon wird bei den weiteren Use Cases abgesehen und sich jeweils auf eine Art der Darstellung beschränkt.
Aus der jeweiligen Darstellung sind die funktionalen Anforderungen der einzelnen Arbeitsschritte zu entnehmen, die für eine erfolgreiche Implementierung notwendig sind.

Use Case 1 als Aktivitätsdiagramm:

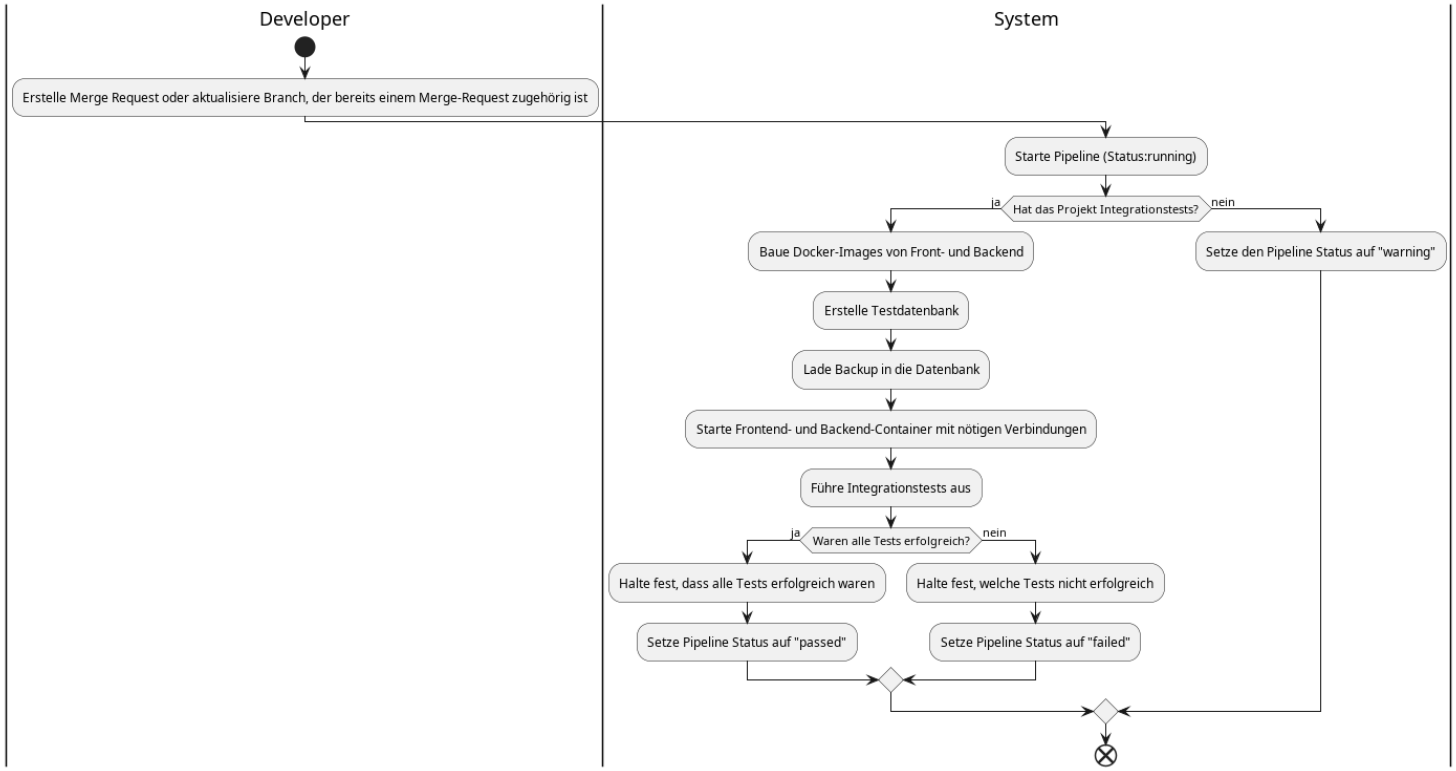

Abbildung 4.1: Aktivitätsdiagramm

Use Case 1 in textualisierter Darstellung:

Use Case Nr. 1		
Name	Integrationstests ausführen	
Quelle	Entwickler-Interview vom 21.11.2023	
Priorität	hoch	
Kritikalität	hoch	
Aktor	Entwickler	
Vorbedingung(en)	- Das System kann Komponenten in den richtigen Versionen einander zuordnen. - Es existieren bereits ausführbare Integrationstests für dieses Projekt.	
Auslösendes Ereignis	Entwickler fügt neuen Merge-Request für eine Feature-Branch hinzu oder Entwickler aktualisiert den Inhalt eines Branches für den bereits ein Merge-Request besteht.	
Beschreibung		
Zeitlicher Ablauf	Entwickler	System
10	Entwickler stellt Merge-Request oder aktualisiert Branch mit bestehendem Merge-Request	-
20	-	Baue die Docker-Images von Frontend und Backend.
30	-	Überprüfe, ob Integrationstests für dieses Projekt bestehen, falls ja fahre fort, ansonsten siehe Alternativszenarien

40	-	Starte die Testdatenbank und lade Testdaten von einem Backup in die Datenbank.
50	-	Starte Frontend und Backend des Projekts und stelle nötigen Verbindungen untereinander und mit der Datenbank her.
60	-	Führe Integrationstests auf dieser Umgebung aus.
70	-	Halte Ergebnisse der Integrationstests fest.
80	-	Beendet Integrationstests ausführen.

Alternativszenarien	20a: Keine Integrationstests vorhanden. Warne den Entwickler und beende den Prozess.
Ausnahmeszenarien	- Timeout: Der Prozess benötigt länger als erlaubt. Breche den Prozess ab, und lasse die Gitlab-Pipeline fehlschlagen.
Ergebnis	Die Integrationstests wurden in der gewünschten Umgebung ausgeführt. Sie sind entweder alle erfolgreich gewesen und die Gitlab-Pipeline hat den Status *passed* oder ein oder mehrere Integrationstests waren nicht erfolgreich und die Pipeline hat den Status *failed* und hält fest, welche Tests nicht funktionsfähig waren.
Nachbedingungen	Der Pipeline-Job hat den Status *passed*.

Tabelle 4.1: Use Case 1 in tabellarischer Darstellung, die einzelnen Ablaufschritte entsprechen den granularen funktionalen Anforderungen, die von der Implementation umzusetzen sind.

Die weiteren ausgearbeiteten Use Cases wurden für einen besseren Lesefluss in den Anhang verschoben und sind hier zu finden: siehe S. 24, 6.1.

4.2 Verhaltensperspektive

Zustandsdiagramm für die Pipeline:

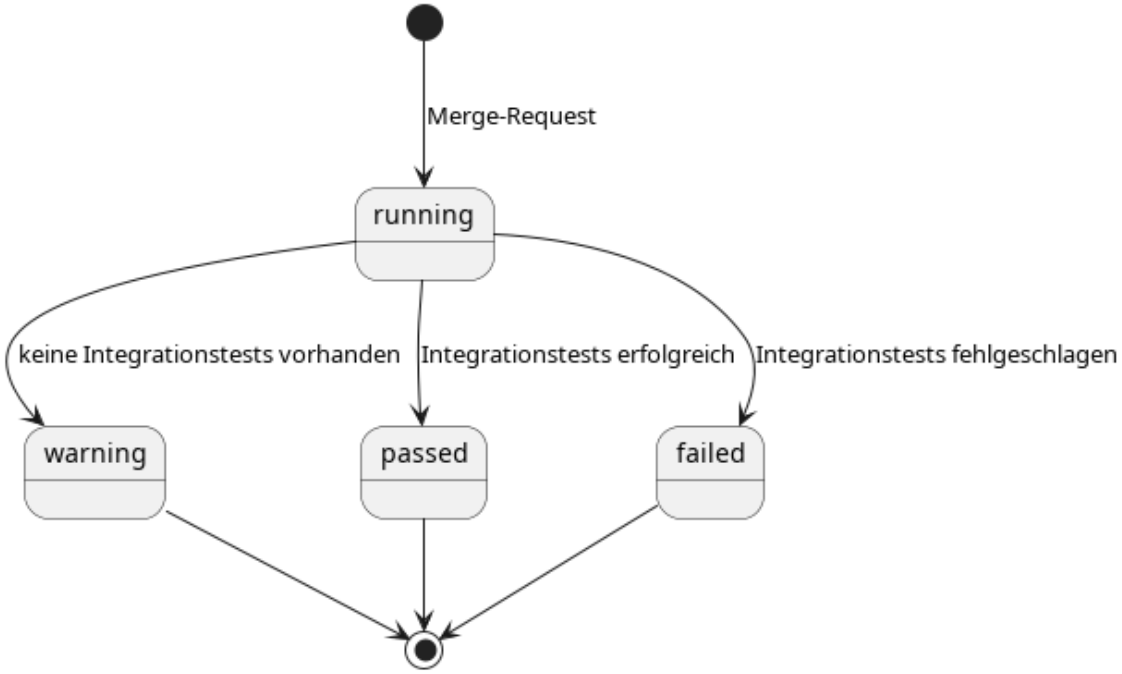

Abbildung 4.2: Zustandsdiagramm Pipeline

4.3 Qualitätsanforderungen

Die folgende Tabelle stellt die wichtigsten Qualitätsanforderungen an die Automatisierung dar und ordnet sie den entsprechenden Use Cases zu.

Qualitätsattribut	**Misuse Case / unerwünschter Zustand: was soll nicht passieren?**	**Metriken: Wie messen / testen Sie, dass die Anforderungen erfüllt ist? Welchen Wert wollen Sie erreichen?**	**Bezug zu Use Case(s)**
Reaktionszeit	Der Entwickler wartet zu lange auf das Feedback aus der Pipeline und verliert zu viel Zeit.	Der Entwickler erhält das Ergebnis der Integrationstests innerhalb von 10 Minuten.	UC 1, UC 2
Wartbarkeit	Es ist zu kompliziert neue Integrationstests hinzufügen und Entwickler vernachlässigen deshalb das Hinzufügen neuer Tests.	Alle Entwickler finden es intuitiv und einfach neue Tests hinzuzufügen.	UC 3
Datenmanagement	Das Austauschen der Datenbank-Backups für die Testdatenbank dauert zu lange oder ist zu kompliziert und Entwickler haben einen zu hohen Arbeitsaufwand.	Entwickler können das Testdaten-Backup in weniger als 5 Minuten austauschen.	UC5

Zuverlässigkeit	Die Pipeline schlägt fehl, obwohl alles korrekt ist und verunsichert so den Benutzer.	Die Pipeline funktioniert in über 99 % der Fälle.	UC 1-6

5. Diskussion von Lösungsansätzen

Typischerweise sind Lösungsansätze kein Bestandteil einer Anforderungsanalyse. Im Rahmen dieser Seminararbeit sollen jedoch einige kurze Lösungsansätzen zu den wichtigsten Anforderungen genannt und diskutiert werden, um einen Ausblick auf die zukünftige Entwicklung des Systems zu geben.

5.1 Platzierung der Testumgebung

Bevor man die Integrationstestumgebung überhaupt aufbauen kann, muss entschieden werden, an welcher Stelle dies optimaler Weise stattfindet. Dabei werden hier zwei Lösungsansätze diskutiert:

- Docker-in-Docker (DinD) in der GitLab-Pipeline
- Ein Pipeline externes System für Integrationstests

5.1.1 Docker-in-Docker (DinD)

DinD ist eine Technik, bei der Container innerhalb eines anderen Container ausgeführt werden können [vgl. 3]. Da eine GitLab Pipeline innerhalb eines Containers abläuft könnte man dort die benötigten Container der zu testenden Softwarekomponenten starten. Im Vergleich zu einem externen System wäre die Konfiguration voraussichtlich weniger komplex und erfordert auch keine zusätzliche Infrastruktur. Allerdings gibt es möglicherweise Performance Probleme bei der Ausführung von Containern in Containern. Während die Komplexität verglichen mit einer externen Infrastruktur wahrscheinlich geringer ist, steigt bei diesem Ansatz vermutlich die Komplexität der Pipelinekonfiguration selbst.

5.1.2 Pipeline externes System

Ein externes dediziertes System ist verglichen mit Docker in Docker leistungsstärker. Bei steigenden Anforderungen, zum Beispiel durch die steigende Zahl von Integrationstests, ist es auch einfacher zu skalieren.

Nachteilhaft ist die erhöhte Komplexität in der Bereitstellung und der Konfiguration dieses Systems. Es treten außerdem zusätzliche Kosten durch das externe System auf.

Ein externes Testsystem müsste außerdem so konzipiert werden, dass mehrere Testumgebungen parallel darauf ablaufen können, denn sonst könnten sich Pipelines aus unterschiedlichen Subprojekten gegenseitig blockieren.

5.1.3 Abwägung

Sollte die Performance in einer DinD basierten Lösung ausreichend sein, um die Anforderungen des Systems zu erfüllen, sollte diese Lösung bevorzugt werden. Ein externes System würde nicht nur durch die Bereitstellung höhere Kosten verursachen, sondern möglicherweise auch durch die komplexere Implementierung. Außerdem wäre bei geschickter Implementierung mit DinD Lösung ein Wechsel auf ein externes System möglich, sollte dies zu einem späteren Zeitpunkt doch nötig werden.

Es empfiehlt sich eine Machbarkeitsstudie für eine DinD-Lösung mit Performance Messung.

5.2 Speicherort der Cypress-Tests

Der Speicherort der Cypress Tests spielt eine wichtige Rolle in der Implementierung. Üblicherweise werden Cypress-Tests im Frontend-Projekt selber abgespeichert [vlg. 2]. Das hat auch den Vorteil, dass im aktuellen Feature-Branch immer die passenden Integrationstests liegen. Wählt man einen zentralen und externen Speicherort muss dort die Versionierung der Tests noch dem Subprojekt und dem Feature-Branch zugeordnet werden. Zusätzlich bietet eine Test-Ablage im Frontend-Repository dem Entwickler eine einfache Möglichkeit, die Tests während der Entwicklung auch lokal leichter auszuführen. Ein externer Speicherort erleichtert jedoch das Teilen der Tests mit dem externen Test-System.

5.3 Zusammenhörigkeit von Komponenten

Bevor die Softwarekomponenten gestartet werden, muss festgestellt werden welche Versionen der Komponenten miteinander getestet werden sollen. Bei der Feature-Entwicklung gibt es die Möglichkeit, dass gleichzeitig Anpassung am Backend und am Frontend vorgenommen werden, oder, dass jeweils nur am Frontend oder nur am Backend Anpassungen gemacht werden. In beiden Szenarien muss dem System bekannt sein, welche Feature Branches zusammengehören. Ein Lösungsansatz ist die gleiche Benennung der Feature Branches im Backend und Frontend Projekt bei Zusammenhörigkeit. Sollte ein Feature nur eines der beiden Komponenten betreffen, und es existiert kein gleichnamiger Branch im entsprechenden Projekt der Komponente, so könnte man einen Fallback-Branch definieren, beispielsweise den Development- oder Main-Branch.

5.4 Fazit

Bevor weitere Entwicklungsschritte eingeleitet werden, sollte zunächst ein Machbarkeitsstudie zu der DinD-Lösung veranlasst werden. Um eine Entscheidung zum Speicherort der Cypress-Tests zu finden, empfiehlt sich eine erneute Befragung des Entwicklerteams. So können eventuelle Vorlieben mit in die Entscheidung einfließen.

6. Anhang

6.1 Darstellung weiterer Use Cases

Use Case 2

Use Case Nr. 2	
Name	Feedback zum neu entwickelten Feature erhalten
Quelle	Entwickler-Interview vom 21.11.2023
Priorität	hoch
Kritikalität	hoch
Aktor	Entwickler
Vorbedingung(en)	Integrationstests wurden ausgeführt nach Use Case 1
Auslösendes Ereignis	Der Entwickler schaut sich das Ergebnis der Pipeline an
Beschreibung	Das System soll folgende Informationen ausgeben: - Welche Versionen der Softwarekomponenten verwendet wurden - Welche Version des Datenbankbackups verwendet wurde - Anzahl der ausgeführten Integrationstests - Anzahl der erfolgreichen Tests - Anzahl der nicht erfolgreichen Tests - Welche Tests nicht erfolgreich ausgeführt worden sind

Use Case 3

Use Case Nr. 3		
Name	Integrationstests hinzufügen	
Quelle	Entwickler-Interview vom 21.11.2023	
Priorität	hoch	
Kritikalität	hoch	
Aktor	Entwickler	
Vorbedingung(en)	-	
Auslösendes Ereignis	Entwickler fügt dem Projekt einen neuen Integrationstest hinzu.	
Beschreibung		
Zeitlicher Ablauf	Entwickler	System
10	Entwickler fügt dem Projekt einen neuen Integrationstest hinzu	-
20	-	Speichere den Test ab, sodass er während der Integrationstest Pipeline ausgeführt werden kann und auch für das generelle Testsystem verwendet werden kann.
30	-	Beende Prozess.

Use Case 4

Use Case Nr. 4		
Name	Testdaten für Datenbank anpassen	
Quelle	Entwickler-Interview vom 21.11.2023	
Priorität	hoch	
Kritikalität	hoch	
Aktor	Entwickler	
Vorbedingung(en)	-	
Auslösendes Ereignis	Entwickler möchte neues Testdaten-Backup nutzen.	
Beschreibung		
Zeitlicher Ablauf	Entwickler	System
10	Entwickler lädt ein neues Testdaten-Backup hoch.	-
20	-	Ersetze die alten Testdaten durch die neuen.
30	-	Beende Prozess.

Use Case 5

Use Case Nr. 5 (zukünftig)	
Name	Kundenspezifische Tests
Quelle	Entwickler-Interview vom 21.11.2023
Priorität	mittel
Kritikalität	mittel
Aktor	Externes Testsystem
Vorbedingung(en)	Es existieren kundenspezifische Integrationstests.
Auslösendes Ereignis	Entwickler fügt neuen Merge-Request für einen Feature-Branch hinzu oder Entwickler aktualisiert den Inhalt eines Branches für den bereits ein Merge-Request besteht.
Beschreibung	Dieser Use Case ist eine Erweiterung der UC1 und UC3. Er läuft gleich ab, jedoch werden kundenspezifische Integrationstests pro Kunde ausgeführt oder kundenspezifische Integrationstests hinzugefügt.

Use Case 6

Use Case Nr. 6		
Name	Integrationstests für generelles Testsystem bereitstellen	
Quelle	Entwickler-Interview vom 21.11.2023	
Priorität	hoch	
Kritikalität	mittel	
Aktor	Externes Testsystem	
Vorbedingung(en)	Im Projekt existiert mindestens ein Integrationstest	
Auslösendes Ereignis	Die Integrationstests sollen jeder Zeit für das externe Testsystem zugänglich sein.	
Beschreibung		
Zeitlicher Ablauf	Entwickler	System
10	Benötigt Zugriff auf die Integrationstests	-
20	-	Stellt Zugriff auf alle Integrationstests frei.
30	-	Beende Prozess.

Literatur

[1] u. A. Kano. „Attractive Quality and Must-be Quality“. In: *Journal of the Japanese Society for Quality Control* 14 (1984). URL: https://web.archive.org/web/20160303204206/http://ci.nii.ac.jp/Detail/detail.do?LOCALID=ART0003570680&lang=en.

[2] Cypress. *Why Cypress?* Besucht: 01.12.2023. 2023. URL: https://docs.cypress.io/guides/overview/why-cypress.

[3] *GitLab CI/CD with Docker*. Zugriff am: 06.12.2023. URL: https://docs.gitlab.com/ee/ci/docker/using_docker_build.html.

[4] Andrea Herrmann. *Grundlagen der Anforderungsanalyse*. Wiesbaden, Germany: Springer Fachmedien Wiesbaden GmbH, 2022. ISBN: 978-3-658-35459-6, 978-3-658-35460-2. DOI: 10.1007/978-3-658-35460-2. URL: https://doi.org/10.1007/978-3-658-35460-2.

[5] IREB. *IREB CPRE Glossary*. Zugriff am: 05.12.2023. URL: https://www.ireb.org/en/cpre/glossary/.

[6] J. Karlsson und K. Ryan. „A Cost-Value Approach for Prioritizing Requirements“. In: *IEEE Software* 14 (1997), S. 67–74. DOI: 10.1109/52.605933. URL: http://dx.doi.org/10.1109/52.605933.

[7] Panagiotis Leloudas. *Introduction to Software Testing: A Practical Guide to Testing, Design, Automation, and Execution*. New York, NY, USA: Apress Media LLC, 2023. ISBN: 978-1-4842-9513-7, 978-1-4842-9514-4. DOI: 10.1007/978-1-4842-9514-4. URL: https://doi.org/10.1007/978-1-4842-9514-4.

[8] Gerard O'Regan. *Undergraduate Topics in Computer Science: Concise Guide to Software Testing*. Cham, Switzerland: Springer Nature Switzerland AG, 2019. ISBN: 978-3-030-28493-0, 978-3-030-28494-7. DOI: 10.1007/978-3-030-28494-7. URL: https://doi.org/10.1007/978-3-030-28494-7.

[9] Chris Rupp und Klaus Pohl. *Basiswissen Requirements Engineering*. ger. 5. Aufl. https://content-select.com/de/portal/media/view/603e7138-368c-40c9-88ac-05bfb0dd2d03. dpunkt.verlag, 2021. ISBN: 9783969102473. URL: http://www.content-select.com/index.php?id=bib_view&ean=9783969102473.

[10] Hansruedi Tremp. *erfolgreich studieren: Agile objektorientierte Anfroderungsanalyse*. ISSN 2524-8693, ISSN 2524-8707 (electronic). Springer Fachmedien Wiesbaden GmbH, 2022. ISBN: 978-3-658-37193-7. DOI: 10.1007/978-3-658-37194-4. URL: https://doi.org/10.1007/978-3-658-37194-4.

Glossar

Angular-Frontend Entwicklung von Benutzeroberflächen mit dem Angular-Framework. 1

Backend Serverseitiger Teil einer Webapplikation für Datenverarbeitung und Geschäftslogik. 2

CI/CD Eine Softwareentwicklungspraxis, die die kontinuierliche Integration von Codeänderungen und deren automatisierte Bereitstellung in Produktionsumgebungen ermöglicht. 1

Container Ein isolierter und eigenständiger Softwarebehälter, der Anwendungen und ihre Abhängigkeiten kapselt und konsistent ausführt. 21

Debugging-Workflow Der systematische Prozess zur Identifizierung, Analyse und Behebung von Fehlern in der Softwareentwicklung. 9

Deployment Bereitstellung und Ausführung von Softwareanwendungen in einer Zielumgebung. 9

Dockerfile Textdatei mit Anweisungen zur Erstellung eines Docker-Containers. 9

Frontend Clientseitiger Teil einer Webapplikation für Benutzeroberfläche und Interaktion. 2

Git-Lab Eine webbasierte Plattform zur Versionskontrolle und Zusammenarbeit für Softwareentwicklungsprojekte, die Funktionen für Repository-Management, CI/CD und mehr bietet. 7

Image Eine Standbild-Repräsentation eines Systems oder einer Anwendung, häufig verwendet in der Containerisierung. 9

Integrationstest Ein Testprozess, bei dem verschiedene Softwarekomponenten oder Module kombiniert und als Ganzes getestet werden, um sicherzustellen, dass sie ordnungsgemäß miteinander interagieren. 1

Interface Schnittstelle zwischen Softwarekomponenten oder Software und Hardware. 1

IREB Eine internationale Organisation, die sich auf die Förderung von Best Practices im Requirements Engineering konzentriert und Zertifizierungen für Requirements Engineers anbietet. 6

Merge-Request Die Anfrage, Codeänderungen von einem Entwicklerzweig in einen anderen zu überführen und zu integrieren, typischerweise in Versionskontrollsystemen. 11

Modul Eigenständige, wiederverwendbare Einheit in der Softwareentwicklung. 1

Pipeline Ein automatisierter Arbeitsablauf, der verschiedene Phasen der Softwareentwicklung integriert, einschließlich Build, Test und Bereitstellung, oft implementiert mithilfe von CI/CD-Tools. 1, 7

Pipeline-Job Eine automatisierte Aufgabe innerhalb eines Continuous Integration / Continuous Deployment (CI/CD) Pipelinesystems. 17

QA-Tool Software oder Anwendungen, die verwendet werden, um den Qualitätsstandards und -richtlinien eines Projekts zu entsprechen, einschließlich Testautomatisierung und Fehlerverfolgung. 1

Release-Tag Ein markiertes Snapshot einer Softwareversion für eine stabile Veröffentlichung. 9

Spring-Java-Backend Serverseitige Anwendungsentwicklung mit dem Spring Framework in Java. 1

Unit-Tests Kleine, automatisierte Tests, die einzelne Einheiten oder Module einer Software auf ihre korrekte Funktionalität überprüfen. 1

Use Case Eine detaillierte Beschreibung eines bestimmten Anwendungsszenarios oder Interaktionsfalls in einem Softwaresystem. 11

Webapplikation Eine über den Webbrowser zugängliche Anwendungssoftware. 1