



Abschlussprüfung Sommer 2025  
Fachinformatiker für Anwendungsentwicklung  
Dokumentation zur betrieblichen Projektarbeit

---

## Implementierung End-to-End Testing in ComfortMarket Implementierung einer automatisierten Testlösung für das ERP-System

Abgabetermin: 30.04.2025

**Prüfungsbewerber:**

Ayoub Aalaou  
Ansbacher Str. 38  
28215 Bremen

**Ausbildungsbetrieb:**

ePhilos AG  
Fahrenheitstr. 7-9  
28359 Bremen



## Inhaltsverzeichnis

Abbildungsverzeichnis .....	iii
Tabellenverzeichnis .....	iii
Abkürzungsverzeichnis .....	iv
1 Einleitung .....	1
1.1 Projektumfeld .....	1
1.2 Projektziel .....	1
1.3 Projektbegründung .....	1
1.4 Projektschnittstellen .....	2
1.5 Projektabgrenzung .....	2
2 Projektplanung .....	3
2.1 Projektphasen .....	3
2.2 Abweichungen vom Projektantrag .....	3
2.3 Ressourcenplanung .....	3
2.4 Entwicklungsprozess .....	4
3 Analysephase .....	4
3.1 Ist-Analyse .....	4
3.2 Wirtschaftlichkeitsanalyse .....	4
3.2.1 Make-or-Buy Entscheidung .....	4
3.2.2 Projektkosten .....	5
3.2.3 Amortisationsdauer .....	6
3.3 Nutzwertanalyse .....	6
3.4 Anwendungsfälle .....	6
3.5 Qualitätsanforderungen .....	6
3.6 Lastenheft/Fachkonzept .....	7
4 Entwurfsphase .....	7
4.1 Zielplattform .....	7
4.2 Architekturdesign .....	7
4.3 Entwurf der Benutzeroberfläche .....	8
4.4 Datenmodell .....	8
4.5 Geschäftslogik .....	9
4.6 Maßnahmen zur Qualitätssicherung .....	10
4.7 Pflichtenheft .....	10
5 Implementierungsphase .....	10
5.1 Implementierung der Testumgebung .....	10
5.2 Implementierung der Testdaten .....	11

5.3	Implementierung der Testskripte .....	11
6	Abnahmephase .....	12
7	Dokumentation .....	13
7.1	Technische Dokumentation .....	13
7.2	Entwicklerdokumentation .....	13
8	Fazit .....	13
8.1	Soll/ist-Vergleich .....	13
8.2	Lessens Learned .....	14
8.3	Ausblick .....	15
	Quellenverzeichnis .....	16
	Anhang.....	I
A.1	Detaillierte Zeitplanung .....	I
A.2	Ressourcenplanung.....	II
A.3	Nutzwertanalyse .....	III
A.4	UML-Aktivitätsdiagramm .....	III
A.5	Lastenheft (Auszug).....	IV
A.6	Architektur von Playwright.....	V
A.7	Allgemeiner Überblick über die Durchführung von Tests .....	V
A.8	Testdatenbereinigung nach Durchführung.....	V
A.9	Pflichtenheft (Auszug).....	VI
A.10	Listing der Seeder-Klasse .....	VIII
A.11	Listing der DB-Klasse .....	IX
A.12	Listing von Beispiel-Testdaten (Fixtures) der Benutzer.....	XI
A.13	Listing mit Informationen zu Testdaten der Benutzer .....	XI
A.14	Listing des Testskripts zum Login-Prozess.....	XI
A.15	Listing: Konfigurationsobjekt für die Datenbankverbindung .....	XIV
A.16	Listing: Wiederverwendbares Login-Modul .....	XIV
A.17	Screenshot: Konfigurationsdatei für Playwright .....	XVI
A.18	Soll-/Ist-Vergleich.....	XVII
A.19	Screenshot - Anmeldemaske ausfüllen .....	XVIII
A.20	Screenshot: Login-Erfolg mit Dashboard-Weiterleitung .....	XVIII
A.21	Screenshot: Dashboard-Menü sichtbar .....	XIX
A.22	Screenshot: HTML-Report nach Testdurchführung .....	XIX

## Abbildungsverzeichnis

Abbildung 1: UML-Aktivitätsdiagramm Login-Prozess .....	III
Abbildung 2: Playwright Architektur .....	V
Abbildung 3: Illustration Überblick über Testdurchführung .....	V
Abbildung 4: Aufräumen der Testdaten nach Testende .....	V
Abbildung 5: Eingabe von Benutzernamen und Passwort .....	XVIII
Abbildung 6: Erfolgreicher Login und Weiterleitung zum Dashboard .....	XVIII
Abbildung 7: Sichtbarkeitsprüfung des Menüeintrags "Dashboard" .....	XIX
Abbildung 8: Screenshot HTML-Testreport nach Testdurchlauf .....	XIX

## Tabellenverzeichnis

Tabelle 1: Grobe Zeitplanung .....	3
Tabelle 2: Kostenberechnung .....	5
Tabelle 3: Detaillierte Zeitplanung .....	I
Tabelle 4: Nutzwertanalyse .....	III
Tabelle 5: Soll-/Ist-Vergleich .....	XVII

## Abkürzungsverzeichnis

<b>AAA</b>	Arrange, Act, Assert
<b>API</b>	Application Programming Interface
<b>CI/CD</b>	Continuous Integration/Continuous Delivery
<b>CM</b>	ComfortMarket
<b>CRUD</b>	Create, Read, Update, Delete
<b>E2E</b>	End-to-End
<b>ERP</b>	Enterprise Resource Planning
<b>HTML</b>	HyperText Markup Language
<b>JSON</b>	JavaScript Object Notation
<b>JSDoc</b>	JavaScript Documentation
<b>PHP</b>	Hypertext Preprocessor
<b>POM</b>	Page Object Model
<b>ROI</b>	Return on Investment
<b>UML</b>	Unified Modeling Language
<b>UI</b>	User Interface
<b>VS</b>	Visual Studio

# 1 Einleitung

Die folgende Projektdokumentation beschreibt den Ablauf des Abschlussprojekts, das der Autor im Rahmen seiner Prüfung zum Fachinformatiker der Fachrichtung Anwendungsentwicklung bei der ePhilos AG durchgeführt hat. Gegenstand des Projekts ist die Entwicklung und Implementierung einer *E2E*<sup>1</sup>-Testautomatisierungslösung für das *ERP*<sup>2</sup>-System ComfortMarket (*CM*).

## 1.1 Projektumfeld

Das Projekt wurde im Rahmen der Ausbildung bei der ePhilos AG durchgeführt. Die ePhilos AG ist ein Softwareunternehmen, das mit der Softwarefamilie *CM* Unternehmen bei der Beschaffung von Handels- und Verbrauchsgütern sowie Dienstleistungen unterstützt. Die Lösung ist modular aufgebaut und optimiert den gesamten Einkaufsprozess.

Das Projekt wurde in der Entwicklungsabteilung der ePhilos AG realisiert, die für die Entwicklung und Wartung des *CM*-Systems verantwortlich ist. Auftraggeber war das interne Entwicklungsteam, das den Bedarf an einer *E2E*-Testlösung identifiziert hatte, um die Softwarequalität zu verbessern und Testprozesse zu automatisieren.

Das Team bestand aus mehreren Softwareentwicklern, die gemeinsam an der Umsetzung der Anforderungen arbeiteten. Die Überwachung des Projektfortschritts wurde dabei vom Projektleiter übernommen. Eine enge Kommunikation und Zusammenarbeit innerhalb des Teams war dabei unerlässlich, um den Anforderungen gerecht zu werden.

## 1.2 Projektziel

Ziel des Projekts ist die Entwicklung und Implementierung einer *E2E*-Testautomatisierungslösung für *CM*. Diese Lösung soll die bestehenden Unit-Tests ergänzen und sicherstellen, dass alle Systemkomponenten nahtlos miteinander interagieren.

Konkret umfasst das Projektziel folgende Aspekte:

- Automatisierung von Tests der Kerngeschäftsprozessen
- Aufbau einer stabilen Testinfrastruktur mit definierter Testdatenversorgung
- Implementierung einer effektiven Datenbankbindung für die Testverwaltung
- Identifizierung von Fehlerquellen und Verbesserung der Benutzererfahrung sowie Systemstabilität

## 1.3 Projektbegründung

Die bisherige Qualitätssicherung von *CM* basiert auch auf manuellen Tests, die jedoch nicht alle Komponenten und deren Zusammenspiel systematisch abdecken können. Zudem prüfen die bestehenden Unit-Tests lediglich einzelne Komponenten und erfassen nicht das Zusammenspiel verschiedener Systemteile. Diese Einschränkungen führen dazu, dass Fehler häufig erst spät erkannt werden und die Sicherstellung der Gesamtfunktionalität aufwendig und fehleranfällig ist.

Um diese Schwächen zu beheben, ist die Entwicklung einer automatisierten Testlösung notwendig. Diese ermöglicht:

---

<sup>1</sup> "E2E": Testverfahren, bei dem die gesamte Anwendung oder ein Geschäftsprozess getestet wird, um das Zusammenspiel aller Komponenten sicherzustellen.

<sup>2</sup> "ERP" Softwarelösung zur Planung, Steuerung und Verwaltung von Unternehmensressourcen wie Material, Personal und Finanzen.

- Reduzierung manueller Testaufwände durch Automatisierung zeitintensiver Tests.
- Konsistente Qualitätssicherung durch regelmäßige, wiederholbare Prüfungen.
- Früherkennung von Fehlern durch *E2E*-Tests bereits in frühen Entwicklungsphasen.
- Effiziente Regressionstests, bei denen Änderungen im System keine unbeabsichtigten Nebeneffekte verursachen.
- Dokumentation des Systemverhaltens, da automatisierte Testfälle zugleich als Referenz für die erwartete Funktionalität dienen.

## 1.4 Projektschnittstellen

Das *E2E*-Testprojekt interagiert mit mehreren Systemkomponenten, wobei *CM* im Zentrum steht. Die automatisierten Tests greifen direkt auf die Benutzeroberfläche zu, simulieren reale Nutzeraktionen und validieren die Ergebnisse. Für die browserbasierte Automatisierung kommt das Test-Framework Playwright zum Einsatz, das als zentrale Schnittstelle zur Interaktion mit dem System dient.

Auch das Backend der Anwendung, das in *PHP* implementiert ist, wird über die Benutzeroberfläche indirekt mitgetestet, indem Anfragen verarbeitet und die entsprechenden Reaktionen überprüft werden.

Zur Sicherstellung konsistenter Testbedingungen wird eine separate MySQL-Testdatenbank verwendet, die über die MySQL2-Bibliothek<sup>3</sup> angebunden ist. Die Verwaltung der Testdaten erfolgt über eine speziell entwickelte Seeder-Klasse<sup>4</sup>, die vordefinierte Datensätze im *JSON*-Format automatisiert in die Datenbank einspielt, diese bei Bedarf bereinigt und so eine kontrollierte und reproduzierbare Testumgebung schafft. Sie unterstützt sowohl die Initialisierung standardisierter Testdaten als auch das gezielte Einrichten komplexerer Testszenarien.

Diese automatisierte Testdatensteuerung gewährleistet stabile, wiederholbare Tests und verhindert unerwünschte Seiteneffekte zwischen den Testläufen.

## 1.5 Projektabgrenzung

Da das Projekt auf einen Zeitrahmen von 80 Stunden begrenzt ist, wurden folgende Einschränkungen definiert:

- Fokus auf grundlegende Infrastruktur und begrenzten Testumfang: Im Rahmen des Projekts wird primär die grundlegende Testinfrastruktur geschaffen. Aus zeitlichen Gründen wird lediglich der Login-Prozess automatisiert getestet. Weitere Testfälle sind nicht Bestandteil dieses Projekts, können jedoch auf Basis der geschaffenen Struktur zu einem späteren Zeitpunkt ergänzt werden.
- Keine Integration in *CI/CD*-Pipeline<sup>5</sup>: Die Einbindung der Tests in eine *CI/CD*-Pipeline ist nicht Bestandteil des Projekts und wird daher nicht berücksichtigt.
- Beschränkung auf Desktop-Browser: Die Tests werden ausschließlich auf Desktop-Browsern durchgeführt und umfassen keine mobilen Endgeräte oder deren spezifische Anforderungen.

---

<sup>3</sup> MySQL2-Bibliothek: Node.js-Bibliothek zur performanten, asynchronen Anbindung einer MySQL-Datenbank.

<sup>4</sup> Seeder-Klasse: Speziell entwickelte Klasse, die vordefinierte Testdaten im *JSON*-Format automatisiert in eine Datenbank lädt und sie bei Bedarf wieder entfernt, um eine kontrollierte, reproduzierbare Testumgebung zu schaffen.

<sup>5</sup> *CI/CD*-Pipeline: Automatisierte Prozesskette zur kontinuierlichen Integration (Continuous Integration) und Auslieferung (Continuous Delivery) von Software

## 2 Projektplanung

### 2.1 Projektphasen

Für die Umsetzung des Projektes standen 80 Stunden zur Verfügung. Diese wurden vor Projektbeginn auf verschiedene Phasen verteilt, die während der Softwareentwicklung durchlaufen werden. Eine grobe Zeitplanung sowie die Hauptphasen lassen sich der Tabelle 1 entnehmen. Eine detaillierte Übersicht dieser Phasen befindet sich im Anhang A.1: Detaillierte Zeitplanung.

Projektphase	Geplante Zeit in Stunden
Analyse	6
Entwurf	13
Implementierung	43
Abnahme	7
Dokumentation	8
Summe	77

*Tabelle 1: Grobe Zeitplanung*

Die verbleibenden 3 Stunden wurden als Pufferzeit eingeplant, um bei unerwarteten Herausforderungen oder zusätzlichen Anforderungen flexibel reagieren zu können, ohne den gesamten Projektrahmen von 80 Stunden zu überschreiten.

### 2.2 Abweichungen vom Projektantrag

Im ursprünglichen Projektantrag wurde die Entwicklung bei der ePhilos AG mithilfe agiler Methodiken und Techniken beschrieben. Aufgrund der Tatsache, dass es sich um ein Einzelpersonenprojekt handelt, wurde jedoch festgestellt, dass diese Vorgehensweise wenig sinnvoll wäre. Zudem weist das Projekt eine geringe Komplexität auf und die Anforderungen sind klar definiert, was den Einsatz eines agilen Vorgehens weniger notwendig erscheinen lässt. Daher wurde entschieden, stattdessen auf das Wasserfallmodell zurückzugreifen, da dieses für ein solches Projekt besser geeignet erscheint. Das Wasserfallmodell ist ein lineares Vorgehensmodell, das in aufeinanderfolgende Projektphasen unterteilt wird und eine klare Struktur bietet.

Die Entscheidung, das Vorgehensmodell zu ändern, wurde getroffen, um die Effizienz des Projekts zu steigern und die Planung sowie die Durchführung zu vereinfachen.

### 2.3 Ressourcenplanung

Im Anhang A.2: Verwendete Ressourcen auf S. ii sind alle für das Projekt eingesetzten Ressourcen aufgeführt. Dies umfasst sowohl die verwendete Hard- und Software als auch die personellen Ressourcen. Bei der Auswahl der Software wurde besonders darauf geachtet, Open-Source-Lösungen oder bereits lizenzierte Produkte zu verwenden, um zusätzliche Kosten zu vermeiden. Auf diese Weise sollen die anfallenden Projektkosten möglichst gering gehalten werden.

## 2.4 Entwicklungsprozess

Bevor mit der Realisierung des Projektes begonnen werden konnte, wurde eine geeignete Vorgehensweise definiert. Für das Abschlussprojekt wurde das Wasserfallmodell gewählt, da die Anforderungen klar und eindeutig festgelegt waren, insbesondere der Login-Prozess, der als einfach und gut abgrenzbar betrachtet wurde.

Das Wasserfallmodell eignet sich besonders für Projekte, bei denen die Anforderungen von Anfang an genau bekannt sind und keine wesentlichen Änderungen während der Umsetzung erwartet werden. Es ermöglicht eine strukturierte Abarbeitung der Phasen in einer festen Reihenfolge. Jede Phase – von der Analyse über den Entwurf, die Implementierung bis hin zur Abnahme und Dokumentation – wurde vollständig abgeschlossen, bevor mit der nächsten Phase begonnen wurde. Diese Herangehensweise gewährleistet eine klare Aufgabentrennung und eine transparente Fortschrittskontrolle.

## 3 Analysephase

### 3.1 Ist-Analyse

CM ist eine umfangreiche ERP-Software, die Unternehmen bei der Digitalisierung ihrer Beschaffungsprozesse unterstützt. Die Softwarelösung ist modular aufgebaut und an verschiedene Kundengruppen, von Großkunden bis mittelständisches Unternehmen, angepasst. Im aktuellen Zustand erfolgt die Qualitätssicherung des Systems hauptsächlich durch Unit-Tests im Frontend und Backend, die einzelne Funktionen oder Methoden isoliert überprüfen. Diese Tests sind jedoch nicht ausreichend, um das Zusammenspiel der verschiedenen Module<sup>6</sup> und Systemschichten systematisch<sup>7</sup> zu testen. Auch die Benutzerperspektive wird in diesen Tests nicht berücksichtigt.

Zusätzlich wird die Qualitätssicherung durch manuelle Tests ergänzt. Diese manuelle Testdurchführung ist jedoch zeitaufwändig und fehleranfällig, insbesondere bei wiederholten Tests nach Systemänderungen. Fehler werden oft erst spät im Entwicklungsprozess entdeckt, was zu einem erhöhten Aufwand bei der Sicherstellung der Gesamtfunktionalität führt.

### 3.2 Wirtschaftlichkeitsanalyse

Die in den Abschnitten 1.3 (Projektbegründung) und 3.1 (Ist-Analyse) beschriebenen Mängel im bestehenden Qualitätssicherungsprozess des CM verdeutlichen die Notwendigkeit einer automatisierten E2E-Testlösung.

In den nachfolgenden Kapiteln wird untersucht, ob dieses Projektvorhaben auch aus wirtschaftlicher Perspektive tragfähig ist.

#### 3.2.1 Make-or-Buy Entscheidung

Für die Implementierung von E2E-Tests standen grundsätzlich die Eigenentwicklung einer Testlösung oder der Einsatz einer kommerziellen Testautomatisierungssoftware zur Auswahl.

---

<sup>6</sup> In diesem Kontext bezeichnet 'Module' eigenständige Funktionskomponenten der ERP-Software, wie Lagerverwalten oder Benutzerverwalten, die spezifische Aufgaben erfüllen und an Kundenbedürfnisse angepasst werden können.

<sup>7</sup> 'Systemschichten' bezieht sich auf die verschiedenen Ebenen der Systemarchitektur, wie Front-End (Benutzeroberfläche) und Back-End (Serverprozesse).

### Kommerzielle Lösungen:

Auf dem Markt gibt es verschiedene kommerzielle Tools für *E2E*-Tests, wie beispielsweise Selenium-basierte Plattformen mit erweiterten Funktionen, TestComplete oder Ranorex. Diese Lösungen bieten oft eine benutzerfreundliche Oberfläche und umfangreiche Funktionen, sind jedoch mit erheblichen Lizenzkosten verbunden.

### Eigenentwicklung:

Die Alternative besteht in der Eigenentwicklung einer Testlösung unter Verwendung von Open-Source-Frameworks wie Playwright, Cypress oder Selenium.

### Entscheidungskriterien:

1. **Kosten:** Kommerzielle Lösungen sind in der Regel mit fortlaufenden Lizenzgebühren verbunden, während bei der Eigenentwicklung vor allem interne Ressourcen und Personalkapazitäten benötigt werden.
2. **Flexibilität:** Eigenentwickelte Lösungen können besser an spezifische Anforderungen angepasst werden.
3. **Integration:** Eine Eigenentwicklung lässt sich leichter in bestehende Systeme integrieren.
4. **Know-how:** Das im Unternehmen vorhandene Know-how in JavaScript und Node.js spricht für eine Eigenentwicklung.

### Entscheidung:

Nach Abwägung dieser Faktoren wurde die Eigenentwicklung unter Verwendung des Open-Source-Frameworks Playwright gewählt. Diese Entscheidung ermöglicht eine maßgeschneiderte Lösung, die optimal auf die Anforderungen des CM zugeschnitten ist, und vermeidet langfristige Lizenzkosten. Zudem kann das bei der Entwicklung gewonnene Know-how für zukünftige Projekte genutzt werden.

### 3.2.2 Projektkosten

Die Projektkosten setzen sich aus den Personalkosten des Auszubildenden und der beteiligten Mitarbeiter sowie den Ressourcenkosten für die Bereitstellung der benötigten Arbeitsmaterialien und des Arbeitsplatzes zusammen. Da die genauen Personalkosten nicht herausgegeben werden dürfen, werden für die Kalkulation von der Personalabteilung festgelegte Pauschalsätze verwendet. Der Stundensatz eines Auszubildenden beträgt 12€, der eines Mitarbeiters 25€. Für die Ressourcennutzung, die den Büroarbeitsplatz, die Hardware- und Softwarenutzung sowie Gemeinkosten wie Strom umfasst, wurde ein pauschaler Stundensatz von 14€ festgelegt. Die detaillierte Aufschlüsselung der Projektkosten ist in Tabelle 2 dargestellt:

Vorgang	Mitarbeiter	Zeit	Personal	Ressourcen	Gesamt
Entwicklungskosten	1 x Auszubildender	77 h	924 €	1078€	2.002 €
Fachgespräch	1 x Mitarbeiter, 1x Auszubildender	2h	74€	56 €	130 €
Abnahme	2 x Mitarbeiter	7h	350 €	196€	546 €
Summe					2.678€

Tabelle 2: Kostenberechnung

### 3.2.3 Amortisationsdauer

Die Amortisationsdauer dieses Projekts ist derzeit schwer abzuschätzen, da die Implementierung momentan nur grundlegende Testfälle wie den Login-Prozess abdeckt. Die eigentlichen Vorteile werden sich jedoch erst mit der Erweiterung der Testabdeckung auf komplexere Geschäftsprozesse und regelmäßige Regressionstests zeigen.

Das Projekt stellt eine langfristige Investition in die Testinfrastruktur dar, die den Wert der Lösung kontinuierlich steigern wird, sobald der Testumfang schrittweise erweitert wird. So kann das Entwicklungsteam die Testabdeckung bei minimalem Aufwand ausbauen, was zu einem kontinuierlich steigenden  $ROI^8$  führt.

## 3.3 Nutzwertanalyse

Neben der reinen Kostenbetrachtung bietet das *E2E*-Testsystem mehrere Vorteile im Vergleich zur manuellen Testdurchführung:

1. **Zuverlässigkeit:** *E2E*-Tests liefern konsistente Ergebnisse unter gleichen Bedingungen, während bei manuellen Tests menschliche Fehler auftreten können.
2. **Reproduzierbarkeit:** Automatisierte *E2E*-Tests führen immer die exakt gleichen Schritte aus, während manuelle Tests schwer zu reproduzieren sind.
3. **Geschwindigkeit:** *E2E*-Tests können schnell und automatisch in großer Zahl ausgeführt werden, was bei manuellen Tests, besonders bei Regressionstests, zeitaufwändig ist.
4. **Dokumentation:** *E2E*-Tests erzeugen automatisch detaillierte Berichte mit Screenshots und Fehlermeldungen. Bei manuellen Tests müssen Ergebnisse manuell dokumentiert werden.
5. **Wartbarkeit:** Änderungen in der Anwendung erfordern bei manuellen Tests eine manuelle Anpassung der Testanleitungen. Bei *E2E*-Tests wirken sich zentrale Änderungen an wiederverwendbaren Komponenten auf alle Tests aus.
6. **Kosteneffizienz:** Obwohl *E2E*-Tests höhere Initialkosten haben, sinken die Kosten pro Testdurchlauf langfristig, während manuelle Tests hohe laufende Personalkosten verursachen.

Diese Vorteile sind in der Nutzwertanalyse im Anhang A.3 (S. iii) dargestellt.

## 3.4 Anwendungsfälle

Um eine strukturierte Übersicht über die zu testenden Anwendungsfälle zu erhalten, wurden im Rahmen der Entwurfsphase *UML*-Aktivitätsdiagramme erstellt. Diese Diagramme bilden die wichtigsten Geschäftsprozesse des *CM* ab, die durch das *E2E*-Testsystem überprüft werden sollen. Im Anhang A.4: *UML*-Aktivitätsdiagramm: Login-Prozess auf S. iii ist exemplarisch der Login-Prozess dargestellt, der einen von mehreren wichtigen Anwendungsfällen für die *E2E*-Tests repräsentiert. Das Diagramm visualisiert detailliert die einzelnen Schritte des Authentifizierungsprozesses, von der Eingabe der Anmeldedaten bis zur erfolgreichen Anmeldung bzw. zum Anzeigen einer Fehlermeldung bei ungültigen Zugangsdaten.

## 3.5 Qualitätsanforderungen

Zur Absicherung der Qualität und Funktionalität der Testautomatisierung wurden folgende Anforderungen festgelegt:

---

<sup>8</sup> "ROI" steht für "Return on Investment" und bezeichnet den Gewinn oder Nutzen, der im Verhältnis zu den getätigten Investitionen erzielt wird.

- Die Tests liefern konsistente und verlässliche Ergebnisse. Flaky<sup>9</sup> Tests, die unregelmäßig fehlschlagen, werden vermieden.
- Intelligente Wartemechanismen (Auto-waiting) sorgen dafür, dass unnötige Verzögerungen im Testablauf vermieden werden.
- Die Tests sind so aufgebaut, dass sie für neue Teammitglieder leicht verständlich sind.
- Funktionen und Variablen sind klar benannt.
- Die Testdateien folgen einer einheitlichen und nachvollziehbaren Struktur.

### 3.6 Lastenheft/Fachkonzept

Das Lastenheft, das die Anforderungen an das Testsystem detailliert beschreibt, wurde in Zusammenarbeit mit der Softwareentwicklern erstellt. Ein Auszug ist im Anhang A.5 Lastenheft (Auszug) auf S. iv zu finden.

## 4 Entwurfsphase

### 4.1 Zielplattform

Bei der Auswahl der Zielplattform hat sich eine Gliederung des Projektes in mehrere Bereiche ergeben.

Die Testautomatisierung wird mit Node.js umgesetzt, da diese Plattform eine einfache Integration mit Playwright, dem gewählten *E2E*-Testing-Framework, ermöglicht. Node.js bietet zudem eine umfangreiche Bibliothek von Modulen, die für die Testautomatisierung nützlich sind.

Für die Interaktion mit der Datenbank wird das mysql2-Modul verwendet, das eine effiziente und zuverlässige Verbindung zur MySQL-Datenbank ermöglicht. Die Datenbank selbst ist eine separate MySQL-Instanz, die ausschließlich für Testzwecke verwendet wird.

Als Entwicklungsumgebung wird VS Code eingesetzt, da es eine hervorragende Unterstützung für JavaScript und Node.js bietet und durch Erweiterungen an die spezifischen Anforderungen der Testautomatisierung angepasst werden kann.

Die Tests werden für die drei hauptsächlich verwendeten Browser (Chrome, Firefox und Edge) optimiert, wobei der Hauptfokus auf Chrome liegt, da dieser der im Unternehmen am häufigsten verwendete Browser ist.

### 4.2 Architekturdesign

Die Architektur der Testautomatisierungslösung basiert auf dem *POM*-Pattern<sup>10</sup>, einem bewährten Muster für die Strukturierung von *E2E*-Tests. Dieses Muster trennt die Testlogik von der Implementierung der Benutzeroberfläche, wodurch die Tests wartbarer und robuster gegenüber Änderungen in der Benutzeroberfläche werden.

Die Architektur besteht aus mehreren Schichten, die jeweils durch spezifische Ordner in der Projektstruktur abgebildet werden:

- **e2e** (Ordner): Enthält die eigentlichen Testskripte, die die Testszenarien definieren und die erwarteten Ergebnisse überprüfen.

---

<sup>9</sup> "Flaky Tests" sind Tests, die unregelmäßig fehlschlagen, selbst wenn der getestete Code korrekt ist. Dies kann durch verschiedene Faktoren wie Instabilität der Testumgebung oder Timing-Probleme verursacht werden.

<sup>10</sup> Das POM-Pattern trennt Testlogik von der Benutzeroberfläche, um Tests wartbarer und robuster gegenüber Änderungen zu machen, indem es Seitenobjekte für Interaktionen und Elemente erstellt.

- **actions** (Ordner): Enthält wiederverwendbare Funktionen, die häufig benötigte Interaktionen wie Login, Navigation oder Datenerfassung kapseln.
- **services** (Ordner): Beinhaltet Hilfsfunktionen für den Zugriff auf externe Systeme wie die Datenbank.
- **fixtures** (Ordner): Enthält Testdaten im *JSON*-Format, die für die Tests benötigt werden.
- **metadata** (Ordner): Beinhaltet Zusatzinformationen zu den Testdaten.

Diese Schichten bzw. Ordner arbeiten eng zusammen, um die Tests effektiv und strukturiert durchzuführen. Die Testskripte im *e2e*-Ordner nutzen die *POM*, um mit der Anwendung zu interagieren und sicherzustellen, dass die Testszenarien korrekt ausgeführt werden. Innerhalb der *POM* kommen die *actions* zum Einsatz, um wiederverwendbare und standardisierte Interaktionen, wie Login oder Navigation, effizient durchzuführen. Gleichzeitig stellen die *services* eine zentrale Anlaufstelle dar, die allen Schichten den Zugriff auf externe Systeme wie Datenbanken oder *APIs* ermöglicht. Die *fixtures* liefern die benötigten Testdaten, die für eine zuverlässige und konsistente Testausführung sorgen. Abschließend sorgt die *metadata* dafür, dass ergänzende Informationen und Kontext bereitgestellt werden, die für die Auswertung und Nachvollziehbarkeit der Testergebnisse notwendig sind.

Eine Veranschaulichung der Architektur kann im Anhang A.6: Architekturmuster eingesehen werden.

### 4.3 Entwurf der Benutzeroberfläche

Da es sich bei diesem Projekt um die Implementierung eines *E2E*-Testsystems handelt, wurde keine eigenständige Benutzeroberfläche entwickelt. Die Tests werden über die Kommandozeile ausgeführt und die Ergebnisse werden in Form von *HTML*-Berichten und Konsolenausgaben dargestellt. Playwright bietet einen integrierten *HTML*-Reporter, der eine übersichtliche Darstellung der Testergebnisse ermöglicht. Dieser Reporter generiert nach jedem Testlauf einen *HTML*-Bericht, der folgende Informationen enthält:

- Übersicht über alle durchgeführten Tests
- Status der Tests (bestanden/fehlgeschlagen)
- Ausführungszeit der Tests
- Detaillierte Fehlerinformationen bei fehlgeschlagenen Tests
- Screenshots von fehlgeschlagenen Tests

Ein Screenshot des *HTML*-Reporters ist im Anhang A.22: Screenshot Testausführung auf S. xix zu finden.

### 4.4 Datenmodell

Die Datenmodelle werden vor jedem Testlauf in eine separate Testdatenbank geladen, um einen definierten und stabilen Ausgangszustand zu gewährleisten. Nach Abschluss der Tests werden sie wieder entfernt, um Seiteneffekte zwischen den Testdurchläufen zu vermeiden und die Integrität der Testumgebung sicherzustellen.

Zur Verwaltung dieser Testdaten dient eine Abstraktionsschicht, die in der Datei **db.js** implementiert ist. Sie stellt grundlegende *CRUD*-Funktionen bereit und kapselt die Komplexität der direkten Datenbankzugriffe. Zur Effizienzsteigerung kommt Connection Pooling<sup>11</sup> zum Einsatz: Anstatt für jede Operation eine neue Verbindung zu öffnen, verwaltet die zentrale DB-Klasse einen Pool wiederverwendbarer Verbindungen. Dadurch wird der

---

<sup>11</sup> „Connection Pooling“: Technik zur Wiederverwendung bestehender Datenbankverbindungen zur Effizienzsteigerung und Reduktion von Verbindungs-Overhead.

Overhead der Verbindungsherstellung reduziert, da Authentifizierung und Aufbau nur einmal erfolgen müssen. Die Umsetzung erfolgt mithilfe der `mysql2/promise`-Bibliothek und der `createPool`-Methode, die die Verbindungsverwaltung automatisiert. Vor Testende wird der Pool ordnungsgemäß geschlossen, um Ressourcenlecks zu vermeiden – ein entscheidender Faktor für Performance und Stabilität bei wiederholten Datenbankzugriffen.

Die Konfiguration der Verbindungen erfolgt über die separate Datei **dbConfig.js**, was eine flexible Anpassung an verschiedene Umgebungen ermöglicht.

Für das Einspielen und Entfernen von Testdaten wird eine spezielle Seeder-Klasse verwendet. Diese automatisiert das Laden vordefinierter *JSON*-Datensätze in die Datenbank und sorgt so für eine kontrollierte, reproduzierbare Testumgebung. Sie ermöglicht sowohl die Initialisierung standardisierter Testdaten als auch die gezielte Vorbereitung komplexerer Testszenarien – was stabile und wiederholbare Abläufe sicherstellt.

Die Testdaten liegen im *JSON*-Format vor, wobei Schlüssel und Werte den jeweiligen Spalten und Inhalten der Tabellen entsprechen. Die Daten werden über die `mysql2`-Bibliothek in die *MySQL*-Testdatenbank eingespielt.

Zu den wichtigsten Entitäten gehören:

- **User:** Enthält Informationen für den Login-Prozess (z. B. `id`, `username`, `email`, `role`) und bildet die Grundlage für Authentifizierungs- und Autorisierungstests.
- **Password:** Speichert Passwort-Hashes zur sicheren Benutzerverifikation, relevant für Login- und Zugriffskontrollszzenarien.

## 4.5 Geschäftslogik

Die Geschäftslogik des *E2E*-Testsystems konzentriert sich auf die Simulation und Überprüfung der zentralen Abläufe im *CM*. Im Rahmen dieses Projekts liegt der Schwerpunkt auf dem Login-Prozess als grundlegender Funktionalität. Der Login-Prozess umfasst dabei folgende logische Schritte:

1. **Testdatenverwaltung:** Erstellung, Manipulation und Löschung von Testdaten
  - Die Seeder-Klasse lädt Testdaten aus *JSON*-Dateien und speichert sie in der Datenbank.
  - Nach Abschluss der Tests werden die Daten aus der Datenbank entfernt, um eine saubere Testumgebung zu gewährleisten.
2. **Testablaufsteuerung:** Steuerung der Testausführung
  - Jeder Test wird in einem isolierten Browserkontext ausgeführt, um sicherzustellen, dass keine Seiteneffekte zwischen den Tests entstehen.
  - Vor und nach jedem Test werden die erforderlichen Testdaten erstellt und entfernt.
3. **Testvalidierung:** Überprüfung der Testergebnisse
  - Assertions überprüfen, ob die erwarteten Ergebnisse erreicht wurden.
  - Bei Fehlern werden detaillierte Fehlerprotokolle erstellt, die eine schnelle Fehlerdiagnose ermöglichen.
4. **Reporting:** Erstellung von Testberichten
  - Nach der Testausführung wird automatisch ein Bericht generiert.
  - Der Bericht enthält umfassende Informationen zur Testdurchführung, den Ergebnissen und etwaigen Fehlern.

Zur Verifikation dieses Prozesses wurden Testfälle entwickelt, die den gesamten Ablauf – von der Eingabe der Anmeldedaten bis zur erfolgreichen Navigation zum Dashboard – abdecken. Dabei werden sowohl erfolgreiche als auch fehlgeschlagene Anmeldeversuche berücksichtigt.

Die Testlogik prüft nicht nur die oberflächliche Funktionalität, sondern validiert auch die korrekte Verarbeitung der Daten im Hintergrund. Dies umfasst die Überprüfung der Datenbankeinträge bei fehlgeschlagenen Anmeldeversuchen sowie die korrekte Darstellung der Benutzeroberfläche nach erfolgreicher Anmeldung.

Die Geschäftslogik wurde so implementiert, dass sie die tatsächlichen Nutzerabläufe in CM exakt widerspiegelt und so eine realistische Überprüfung der Systemfunktionalität ermöglicht.

## 4.6 Maßnamen zur Qualitätssicherung

Zur Absicherung der Qualität und Funktionalität der E2E-Tests wurden mehrere zentrale Anforderungen definiert. Die Testdatenverwaltung erfolgt über strukturierte JSON-Fixtures, wobei vor jedem Testlauf die Testdatenbank automatisch befüllt und nach Abschluss vollständig bereinigt wird. Dies gewährleistet konsistente Testbedingungen und verhindert Seiteneffekte zwischen verschiedenen Testdurchläufen. Für die Tests wird ein separates lokales CM-System mit eigener Datenbank verwendet, um das Entwicklungssystem nicht zu beeinträchtigen. Die Testumgebung arbeitet vollständig isoliert ohne Verbindungen zu Entwicklungssystem. Playwright erstellt automatisch Screenshots bei fehlgeschlagenen Tests, was die Fehlersuche erheblich erleichtert. Die implementierte Testlösung nutzt das POM-Pattern, wodurch Änderungen an der Benutzeroberfläche nur an einer zentralen Stelle angepasst werden müssen, was die Wartbarkeit der Tests verbessert.

## 4.7 Pflichtenheft

Als Ergebnis der Entwurfsphase wurde ein Pflichtenheft erstellt. Dieses baut auf dem Lastenheft (vgl. dazu Abschnitt 3.6 Lastenheft/Fachkonzept) auf und beschreibt, wie und womit die Anforderungen des Fachbereichs umgesetzt werden sollen. Es dient somit als Leitfaden für die Realisierung des Projektes. Ein Auszug aus dem Pflichtenheft befindet sich im Anhang A.9: Pflichtenheft (Auszug) auf S. vi.

# 5 Implementierungsphase

## 5.1 Implementierung der Testumgebung

Die Implementierung der Testumgebung begann mit der Installation der notwendigen Node.js-Pakete und der Konfiguration von Playwright. Die Konfiguration erfolgte in der zentralen Datei **playwright.config.js**, in der unter anderem die zu testende Browser, Screenshot-Einstellungen und Timeouts definiert wurden (siehe Listings Nr. 8: *playwright.config.js*).

Um eine stabile und reproduzierbare Testumgebung zu gewährleisten, wurden spezifische Versionen aller Abhängigkeiten festgelegt. Diese Versionsfixierung erfolgte in der **package.json**-Datei, wobei besonderes Augenmerk auf die Kompatibilität zwischen Playwright, Node.js und der mysql2-Bibliothek gelegt wurde. Für Playwright wurde die Version 1.35.1 verwendet, die eine hohe Stabilität bietet und gleichzeitig alle benötigten Funktionen für die E2E-Tests bereitstellt. Die mysql2-Bibliothek wurde in Version 3.2.0 eingesetzt, die eine zuverlässige Verbindung zur Testdatenbank gewährleistet. Diese präzise Versionsauswahl minimiert das Risiko von Kompatibilitätsproblemen und stellt sicher, dass die Tests über längere Zeit konsistent ausgeführt werden können, unabhängig von zukünftigen Updates der einzelnen Bibliotheken.

Anschließend wurde die im Architekturdesign definierte Verzeichnisstruktur angelegt, die die klare Trennung der verschiedenen Komponenten des Testsystems gewährleistet. Die Hauptordner umfassen actions für wiederverwendbare Aktionen, e2e für die eigentlichen Tests, fixtures für Testdaten, metadata für Testmetadaten und services für Hilfsdienste.

Neben den Standardfunktionen für *UI*-Interaktionen wurde auch die WebSocket-Unterstützung von Playwright implementiert, die für die Testautomatisierung von Echtzeit-Features des CM-Systems entscheidend ist. Die WebSocket-API ermöglicht das Abfangen, Überwachen und Analysieren von WebSocket-Verbindungen zwischen Client und Server. Diese Funktionalität wurde insbesondere bei Testfällen eingesetzt, die asynchrone Kommunikation und dynamische *UI*-Updates ohne Neuladen erfordern. Durch die Integration von WebSocket-Tests konnte sichergestellt werden, dass nicht nur die Benutzeroberfläche korrekt dargestellt wird, sondern auch der zugrunde liegende Datenaustausch in Echtzeit ordnungsgemäß funktioniert. Dies war besonders wertvoll für komplexe Szenarien wie die Aktualisierung von Dashboards oder Benachrichtigungen, bei denen die korrekte und zeitnahe Datenübertragung für die Benutzerinteraktion von wesentlicher Bedeutung ist. Die Protokollierung und Analyse der WebSocket-Nachrichten ermöglicht zudem eine detaillierte Fehlerdiagnose bei auftretenden Problemen in der Client-Server-Kommunikation.

Die zentrale Datenbankabstraktion wurde in der Datei **db.js** implementiert. Diese Klasse kapselt alle Datenbankoperationen und bietet eine einheitliche Schnittstelle für *CRUD*-Operationen, die von den Tests verwendet werden können (siehe *Listings Nr. 2: DB.js*). Die Konfiguration der Datenbankverbindung erfolgt in einer separaten Datei **dbConfig.js**, was die Anpassung an verschiedene Umgebungen erleichtert.

## 5.2 Implementierung der Testdaten

Für die Tests wurden strukturierte Testdaten in Form von *JSON*-Dateien erstellt, die die zu testenden Szenarien abbilden. Diese Testdaten umfassen beispielsweise Benutzerinformationen für die Login-Tests (siehe *Listings Nr. 3: cm\_user.json*). Aus Datenschutz- und Vertraulichkeitsgründen können keine echten Unternehmensdaten offengelegt werden. Das gezeigte Beispiel dient daher ausschließlich dazu, die Struktur und den Aufbau der verwendeten Testdaten zu veranschaulichen.

Zur Verwaltung dieser Testdaten wurde eine spezielle Seeder-Klasse entwickelt (siehe *Listings Nr. 1: seeder.js*). Diese Klasse ist für das automatisierte Laden und Entfernen der Testdaten verantwortlich und stellt sicher, dass für jeden Test ein definierter Ausgangszustand in der Datenbank hergestellt wird. Vor jedem Test werden die benötigten Daten geladen und nach Abschluss des Tests wieder entfernt, um Seiteneffekte zwischen den Tests zu vermeiden.

Die Testdaten wurden so gestaltet, dass sie verschiedene Testszenarien abdecken, wie beispielsweise gültige und ungültige Benutzeranmeldedaten oder verschiedene Artikelkategorien. Ein Beispiel für die Metadaten, die zusätzliche Informationen zu den Testdaten enthalten, ist im Anhang A.13: Beispiel für Testdaten-Metadaten auf S. xi zu finden.

## 5.3 Implementierung der Testskripte

Die Implementierung der Testskripte erfolgte nach dem *POM*-Pattern, einem bewährten Muster für die Strukturierung von *E2E*-Tests. Dieses Muster trennt die Testlogik von der Implementierung der Benutzeroberfläche, wodurch die Tests wartbarer und robuster gegenüber Änderungen werden.

Für häufig verwendete Aktionen, wie den Login-Prozess, wurden wiederverwendbare Klassen erstellt (siehe *Listings Nr. 7: login.js*). Diese Klassen kapseln komplexe Interaktionen und machen sie für alle Tests verfügbar, was die Codewiederholung reduziert und die Wartbarkeit verbessert.

Die eigentlichen Tests wurden im Verzeichnis *e2e/* organisiert und nach Funktionsbereichen gruppiert. Ein Beispiel für einen Login-Test ist im (*Listings Nr. 5: login.spec.js*) zu finden. Diese Tests folgen dem *AAA*-Muster, bei dem zuerst die Testumgebung vorbereitet, dann die zu testende Aktion durchgeführt und schließlich das Ergebnis überprüft wird.

Eine besondere Stärke bei der Implementierung ist der Einsatz der **page.evaluate()**-Methode von Playwright. Diese Methode ermöglicht die Ausführung von JavaScript-Code direkt im Kontext der Webseite und bietet damit uneingeschränkten Zugriff auf sämtliche Module und Funktionen des CM-Systems. Diese leistungsstarke Funktion öffnet zahlreiche Möglichkeiten für das Testen:

- Der Zugriff auf das Sprachausdruck-Modul gewährleistet sprachunabhängige Tests, die unabhängig von der konfigurierten Benutzeroberfläche funktionieren.
- Über das CM\_Config-Modul können Tests dynamisch an verschiedene Systemkonfigurationen angepasst werden.
- Das CM\_Rolle-Modul ermöglicht differenzierte Tests des Berechtigungskonzepts.

Darüber hinaus bietet **page.evaluate()** die Flexibilität, auf beliebige andere CM-Systemmodule und -funktionen zuzugreifen, wie z.B. Validierungsfunktionen, interne Hilfsmethoden, Geschäftslogik-Module oder Custom-Events. Diese umfassende Integrationsmöglichkeit stellt einen wesentlichen Vorteil dar, da die Tests nicht nur oberflächlich die UI prüfen, sondern auch tiefer in die Anwendungslogik integriert werden können.

Die Tests können somit das System auf einer tieferen Ebene validieren und gleichzeitig bestehende Funktionalitäten des CM-Systems wiederverwenden, was zu effizienteren, zuverlässigeren und realitätsnäheren Tests führt. Diese enge Verzahnung zwischen Testsystem und Anwendung war ein entscheidender Faktor für die hohe Qualität und Aussagekraft der implementierten Tests.

## 6 Abnahmephase

Nach Abschluss der Implementierungsphase wurde die entwickelte Testautomatisierung einem verantwortlichen Teil des Entwicklerteams zur Abnahme übergeben. Diese umfasste folgende Schritte:

- **Präsentation der Testautomatisierung:** Vorstellung der implementierten Tests, ihrer Struktur sowie ihres Zwecks
- **Live-Demonstration:** Ausführung der Tests in verschiedenen Browsern und Szenarien zur Veranschaulichung der Funktionalität
- **Code-Review:** Prüfung des Quellcodes durch einen erfahrenen Entwickler zur Sicherstellung der Codequalität
- **Validierung der Ergebnisse:** Überprüfung, ob die Tests die erwarteten Resultate zuverlässig liefern
- **Die Abnahme verlief erfolgreich:** Die verantwortlichen Teammitglieder bestätigten, dass die implementierten Tests den definierten Anforderungen entsprechen. Besonders positiv hervorgehoben wurden:
  - Modulare Architektur, die eine einfache Erweiterung und Wartung ermöglicht
  - Robuste Implementierung, die zuverlässig mit Timing-Problemen und Netzwerklatenzen<sup>12</sup> umgeht
  - Ausführliche Dokumentation, die die Nutzung sowie die Weiterentwicklung der Tests deutlich erleichtert

Im Rahmen der Abnahme wurden zudem Verbesserungspotenziale identifiziert, die außerhalb des aktuellen Projektumfangs liegen und in zukünftigen Weiterentwicklungen berücksichtigt werden könnten:

- Integration der Tests in eine CI/CD-Pipeline

---

<sup>12</sup> Netzwerklatenzen“ Verzögerungen bei der Netzwerkkommunikation, die die Performance von Anwendungen beeinflussen können.

- Erweiterung der Testabdeckung auf zusätzliche Geschäftsprozesse

## 7 Dokumentation

Für das *E2E*-Testsystem wurden zwei komplementäre Dokumentationsarten erstellt, die unterschiedliche Aspekte des Systems abdecken und verschiedenen Zielgruppen gerecht werden: eine technische Dokumentation zur Systemarchitektur sowie eine umfassende Entwicklerdokumentation. Zusammen bilden sie eine solide Wissensbasis für die Wartung und Weiterentwicklung des Testsystems.

### 7.1 Technische Dokumentation

Diese technische Dokumentation gibt einen detaillierten Überblick über die Architektur des *E2E*-Testsystems und erklärt die wesentlichen Designentscheidungen, die seiner Entwicklung zugrunde liegen. Sie legt einen besonderen Fokus auf die konzeptionelle Struktur der Testumgebung und die klare Trennung von Test- und Produktionsdaten.

Hervorgehoben werden insbesondere die Vorteile der implementierten Architektur, wie ihre Wartbarkeit, Erweiterbarkeit und Zuverlässigkeit in Bezug auf die Tests.

Zur besseren Übersichtlichkeit wurde die detaillierte Beschreibung der Projektstruktur sowie der einzelnen Komponenten in einem separaten Abschnitt dieser Dokumentation untergebracht. So wird der Fokus auf die architektonischen Grundprinzipien gewahrt.

### 7.2 Entwicklerdokumentation

Die Entwicklerdokumentation wurde systematisch mithilfe von *JSDoc*<sup>13</sup> aus dem kommentierten Quellcode generiert, was eine kontinuierliche Aktualität sicherstellt. Sie enthält detaillierte Beschreibungen aller implementierten Klassen, Methoden und deren Parameter sowie Abhängigkeiten zwischen den verschiedenen Komponenten. Ergänzt wird diese textbasierte Dokumentation durch aussagekräftige visuelle Darstellungen wie Aktivitätsdiagramme, die komplexe Interaktionen und Testabläufe anschaulich visualisieren. Diese Kombination aus präzisen Beschreibungen und visuellen Hilfsmitteln erleichtert neuen Entwicklern den Einstieg und dient erfahrenen Teammitgliedern als verlässliches Nachschlagewerk bei der Weiterentwicklung oder Anpassung des Testsystems.

## 8 Fazit

### 8.1 Soll/ist-Vergleich

Das Projektziel, *E2E*-Tests für den Login-Prozess des *CM*-Systems zu implementieren, wurde vollständig erreicht. Die entwickelten Tests decken sämtliche definierten Testfälle ab und liefern konsistente sowie verlässliche Ergebnisse.

Die geplante Projektdauer von 77 Stunden wurde eingehalten, wobei es zu Verschiebungen innerhalb der einzelnen Projektphasen kam. In der Analysephase wurde ein zusätzlicher Aufwand von zwei Stunden benötigt, um ein tieferes Verständnis der Systemstrukturen zu erlangen. Auch die Entwurfsphase erforderte mit 14 statt 13 Stunden etwas mehr Zeit als geplant, da der Entwurf der Datenbankanbindung und des Testdatenmanagements aufwendiger war als ursprünglich angenommen.

---

<sup>13</sup> JSDoc: Dokumentationsstandard zur automatischen Generierung von Beschreibungen aus kommentiertem Quellcode.

Die zusätzlich investierte Zeit in den frühen Projektphasen zahlte sich jedoch in den späteren Phasen deutlich aus. Durch die fundierte Analyse und einen klar strukturierten, gut durchdachten Entwurf verlief die Implementierungsphase deutlich reibungsloser. Viele potenzielle Probleme konnten bereits im Vorfeld gelöst werden, was zu weniger Mehraufwand während der Entwicklung führte. Dadurch konnte die Implementierung mit 42 statt 43 Stunden effizienter als geplant abgeschlossen werden.

Noch deutlicher zeigte sich der Vorteil in der Abnahmephase, die mit nur 5 statt 7 Stunden erheblich zügiger verlief. Die klare Struktur und der saubere, gut dokumentierte Code erleichterten den Code-Review-Prozess erheblich und führten zu weniger Rückfragen.

Die Dokumentationsphase wurde wie geplant mit 8 Stunden abgeschlossen.

Alle definierten Qualitätsanforderungen wurden erfüllt. Die Tests zeichnen sich durch Robustheit, Wartbarkeit und Erweiterbarkeit aus. Sie sind umfassend dokumentiert und liefern reproduzierbare Ergebnisse.

Das interne Entwicklungsteam sowie der Projektleiter zeigten sich mit dem Projektergebnis sehr zufrieden. Die implementierte Lösung überzeugte durch ihre klare Struktur, die hohe Codequalität und die stabile Testabdeckung. Eine Ausweitung der entwickelten Testinfrastruktur auf weitere Geschäftsprozesse des CM-Systems ist bereits in Planung.

Eine detaillierte Gegenüberstellung der geplanten und tatsächlichen Zeiten befindet sich im Anhang unter A.18 „Soll-/Ist-Vergleich der groben Zeitplanung“ auf Seite xvii.

## 8.2 Lessens Learned

Die Durchführung dieses Projekts stellte für mich als angehenden Fachinformatiker eine herausragende Lernchance dar. Die Implementierung von *E2E*-Tests erforderte nicht nur die Anwendung der während meiner Ausbildung erworbenen Kenntnisse, sondern auch die eigenständige Einarbeitung in neue Technologien und Konzepte.

Besonders wertvoll war die Erfahrung, ein komplexes Projekt eigenverantwortlich von der Analyse bis zur Abnahme umzusetzen. Die intensive Beschäftigung mit Testautomatisierung und dem Playwright-Framework hat mein technisches Verständnis deutlich vertieft und mein Selbstvertrauen als Entwickler gestärkt.

Die Arbeit mit Datenbanken sowie die Entwicklung eines strukturierten Testdatenmanagements boten mir wertvolle Einblicke in die Bedeutung einer sauberen Datenarchitektur. Hierbei konnte ich praxisnah erleben, wie zentral ein konsistentes und wartbares Datenfundament für automatisierte Tests ist.

Im Verlauf des Projekts wurde mir außerdem die essenzielle Bedeutung von Kommunikation in der Softwareentwicklung bewusst. Der regelmäßige Austausch mit dem Projektleiter und anderen Entwicklern war entscheidend, um ein Testsystem zu schaffen, das den tatsächlichen Anforderungen entspricht. Ohne diese kontinuierliche Abstimmung wäre es kaum möglich gewesen, die Testszenarien präzise zu definieren und effektiv umzusetzen.

Ebenso zeigte sich die Relevanz einer fundierten Planungs- und Analysephase. Die Zeit, die zu Projektbeginn in das Verständnis der Systemarchitektur und die Definition der Testanforderungen investiert wurde, zahlte sich in der späteren Umsetzung mehrfach aus. Gleichzeitig lernte ich, flexibel auf Änderungen zu reagieren und meine Vorgehensweise bei Bedarf anzupassen.

Die Entwicklung und Implementierung des *E2E*-Testsystems hat nicht nur meine technischen Fähigkeiten erweitert, sondern auch mein Verständnis für Qualitätssicherungsprozesse in der Softwareentwicklung geschärft. Die Erkenntnis, wie essenziell automatisierte Tests für die

langfristige Stabilität und Wartbarkeit eines Systems sind, wird meine zukünftige Arbeit als Entwickler nachhaltig prägen.

Insgesamt betrachte ich dieses Projekt als großen persönlichen und fachlichen Erfolg. Es hat nicht nur einen konkreten Mehrwert für die ePhilos AG geschaffen, sondern auch meine Ausbildung durch praktische Erfahrungen in einem realen Projektumfeld maßgeblich bereichert.

## 8.3 Ausblick

Die implementierte Testautomatisierung stellt eine zuverlässige Grundlage für die zukünftige Qualitätssicherung der CM-Software dar. Sie trägt bereits jetzt dazu bei, die Effizienz des Testprozesses zu steigern und die Gesamtqualität der Anwendung nachhaltig zu verbessern.

Für die Weiterentwicklung des Testsystems sind folgende Maßnahmen vorgesehen:

- **Integration in die CI/CD-Pipeline:** Die automatisierten Tests sollen künftig in den kontinuierlichen Integrations- und Deployment-Prozess eingebunden werden, um bei jeder Codeänderung automatisch ausgeführt zu werden. Dies ermöglicht eine frühzeitige Fehlererkennung und steigert die Stabilität der Softwareentwicklung.
- **Erweiterung der Testabdeckung:** Geplant ist zudem die Ausweitung der automatisierten Tests auf zusätzliche Anwendungsszenarien, um eine noch umfassendere Absicherung der Systemfunktionalität zu gewährleisten.

Diese geplanten Erweiterungen werden nicht nur die Testeffizienz weiter erhöhen, sondern auch maßgeblich zur kontinuierlichen Verbesserung der Softwarequalität beitragen.

## Quellenverzeichnis

Page Object. Abgerufen 30.04.2025, von  
<https://martinfowler.com/bliki/PageObject.html>

How to Connect to a MySQL Database Using the mysql2 Package in Node.js?. Abgerufen 30.04.2025, von  
<https://www.geeksforgeeks.org/how-to-connect-to-a-mysql-database-using-the-mysql2-package-in-node-js/>

Simple Examples of Using Playwright Evaluate Method. Abgerufen 30.04.2025, von  
<https://adequatica.medium.com/simple-examples-of-using-playwright-evaluate-method-9b00d01cad1>

Database Connection Pool. Abgerufen 30.04.2025, von  
<https://medium.com/@sujoy.swe/database-connection-pool-647843dd250b>

## Anhang

### A.1 Detaillierte Zeitplanung

Phase des Projektes	Zeit
<b>Analysephase</b>	<b>6:00:00</b>
Durchführung der Ist-Analyse	1:00:00
Ermittlung der Anforderungen für Test-Szenarien	1:30:00
Identifikation der relevanten Testumgebungen und Systeme	1:00:00
Wirtschaftlichkeitsanalyse und Nutzwertanalyse durchführen	1:00:00
Unterstützung bei der Erstellung Lastenhefts	1:30:00
<b>Entwurfsphase</b>	<b>13:00:00</b>
Erstellung von <i>UML</i> -Aktivitätsdiagrammen für Test-Szenarien	0:30:00
Festlegung der Testdaten und -strukturen	2:30:00
Vergleich und Auswahl geeigneter <i>E2E</i> -Testing-Frameworks	3:00:00
Entwurf der Datenbankbindung und Testdatenmanagement	4:00:00
Planung der Verzeichnisstruktur und Architektur	2:00:00
Erstellung des Pflichtenhefts	1:00:00
<b>Implementierungsphase</b>	<b>43:00:00</b>
Einrichtung der Entwicklungsumgebung	2:30:00
Installation und Konfiguration von Playwright	3:30:00
Implementierung der Datenbankklasse (DB.js)	5:30:00
Implementierung der Datenbankverbindung (dbConfig.js)	0:15:00
Entwicklung der Seeder-Klasse für Testdatenmanagement (seeder.js)	5:00:00
Erstellung der <i>JSON</i> -Testdaten für CM-Benutzer	2:00:00
Erstellung der <i>JSON</i> -Testdaten für CM-Kennwörter	1:00:00
Implementierung der Login-Testfälle	10:00:00
Integration der <i>CM</i> -Systemmodule und Utilities in Testskripte (z.B. Sprachausdruck, Config...)	9:00:00
Implementierung der Test-Metadaten-Verwaltung	0:30:00
Refaktorisierung und Optimierung des Testcodes	3:45:00
<b>Abnahmephase</b>	<b>7:00:00</b>
Durchführung der Tests in verschiedenen Browsern	1:00:00
Validierung der Testergebnisse mit Senior-Entwicklern	4:00:00
Behebung identifizierter Fehler und Probleme	1:00:00
Abschließende Testdurchführung und Präsentation	1:00:00
<b>Dokumentationsphase</b>	<b>8:00:00</b>
Erstellung der technischen Dokumentation	1:00:00
Dokumentation der Testfälle und -szenarien	0:30:00
Erstellung der <i>JSDoc</i> -Entwicklerdokumentation	00:45:00
Zusammenstellung der finalen Projektdokumentation	5:45:00
<b>Summe</b>	<b>77:00:00</b>

Tabelle 3: Detaillierte Zeitplanung

## A.2 Ressourcenplanung

### Hardware:

- Büroarbeitsplatz mit Lenovo ThinkBook 15 G2 ITL 20VE (Core i5 1135G7 mit 2.4 GHz, 8GB RAM)

### Software:

- Windows 11 Pro – Betriebssystem
- Visual Studio Code – Entwicklungsumgebung
- SVN – Zentrale Versionsverwaltung
- Google Chrome, Mozilla Firefox, Microsoft Edge – Browser für Webentwicklung und Tests
- Microsoft Teams – Kommunikations-Tools
- Mantis – Ticket-System
- Laragon – Lokale Entwicklungsumgebung
- MySQL – Datenbankverwaltungssystem

### Verwendete Bibliotheken und Frameworks:

- mysql2 – Node.js-Bibliothek zur Anbindung von MySQL-Datenbanken
- Playwright – Framework für automatisierte *E2E*-Tests von Webanwendungen in verschiedenen Browsern

### Team und Verantwortlichkeiten:

- Projektleiter – Überwachung des Projektfortschritts und Ansprechpartner für das Team
- Softwareentwickler – Review des Codes und Definition der Anforderungen
- Auszubildender – Durchführung des Projekts und Implementierung von *E2E*-Testing

### A.3 Nutzwertanalyse

Eigenschaft	Gewichtung	Manuelle Tests	E2E-Tests
<b>Zuverlässigkeit</b>	25	2	5
<b>Reproduzierbarkeit</b>	20	2	5
<b>Geschwindigkeit</b>	15	2	4
<b>Dokumentation</b>	10	3	4
<b>Wartbarkeit</b>	15	3	4
<b>Kosteneffizienz</b>	15	2	4
<b>Ergebnis</b>	<b>100</b>	<b>225</b>	<b>445</b>

Tabelle 4: Nutzwertanalyse

### A.4 UML-Aktivitätsdiagramm

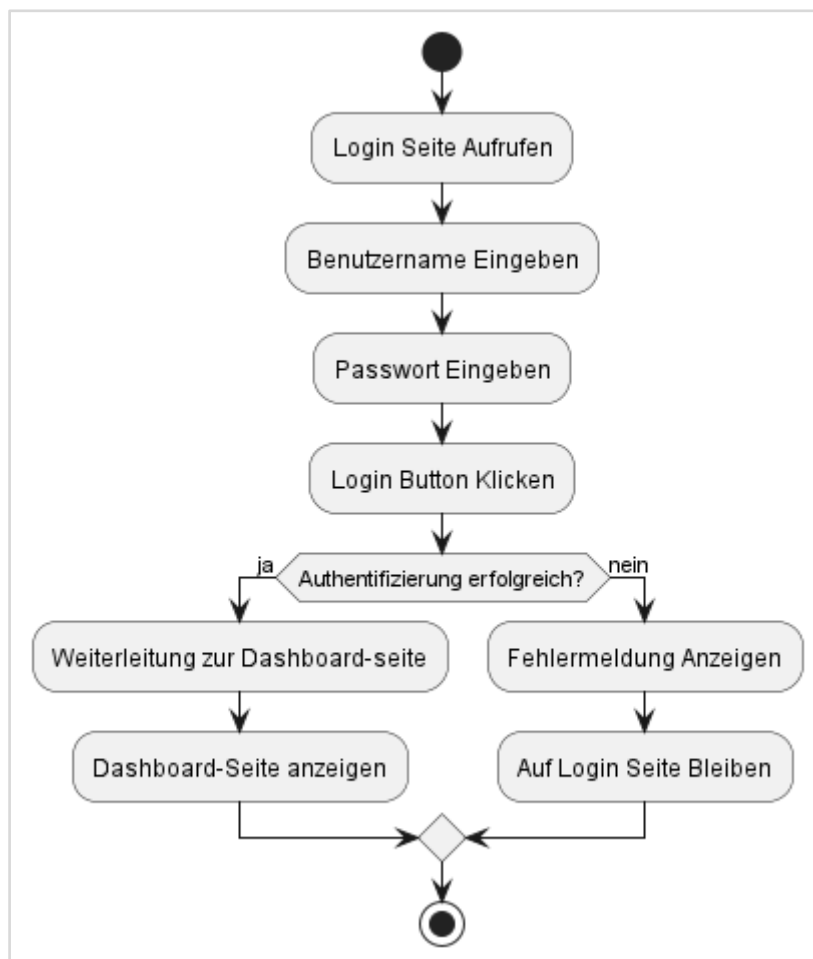


Abbildung 1: UML-Aktivitätsdiagramm Login-Prozess

## A.5 Lastenheft (Auszug)

Der folgende Auszug des Lastenheftes enthält einige der Anforderungen, welche die Stakeholder an das E2E-Testsystem für CM stellten. Zu erfüllende Anforderungen

- Tests für alle kritischen Geschäftsprozesse erstellt werden können
- Tests browserübergreifend ausführbar sein (Chrome, Firefox, Edge)
- Testergebnisse klar dokumentiert und nachvollziehbar sein
- Testdaten automatisiert in die Testdatenbank geladen werden
- Fehlersituationen durch Screenshots automatisch dokumentiert werden
- Tests unabhängig voneinander ausführbar sein
- Einfache Erweiterbarkeit um neue Testfälle
- Zuverlässige Wiederholbarkeit der Tests
- Automatisches Zurücksetzen der Testdatenbank vor jedem Testlauf
- Flexibles Anlegen von Testdaten für verschiedene Testszenarien
- Konsistente Datenhaltung über alle Tests hinweg
- Übersichtliche Darstellung der Testergebnisse
- Nachvollziehbare Fehlerdokumentation mit Screenshots

### A.6 Architektur von Playwright

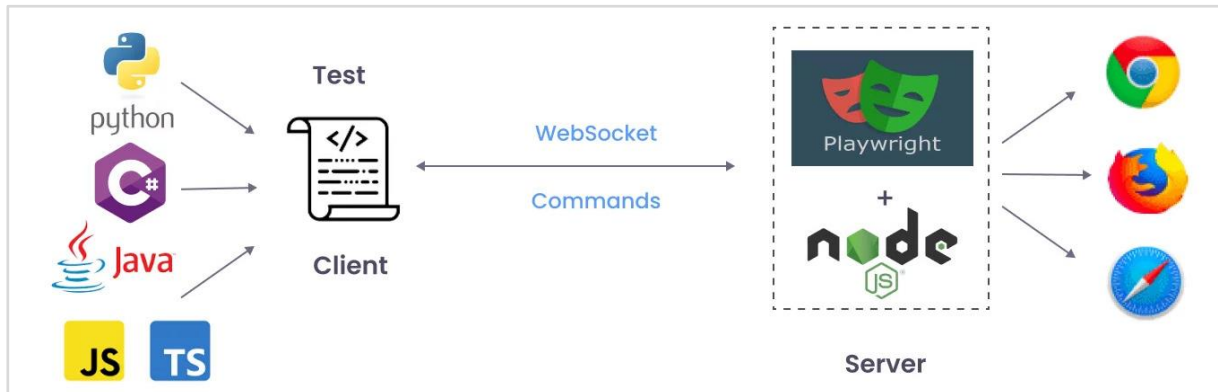


Abbildung 2: Playwright Architektur

### A.7 Allgemeiner Überblick über die Durchführung von Tests

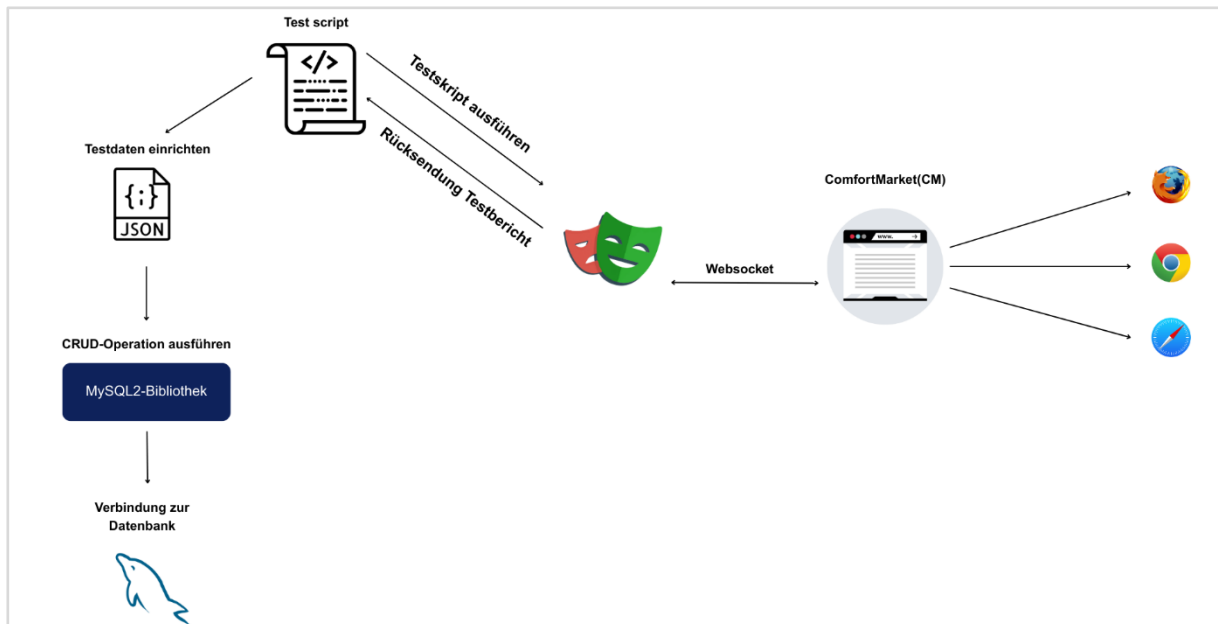


Abbildung 3: Illustration Überblick über Testdurchführung

### A.8 Testdatenbereinigung nach Durchführung

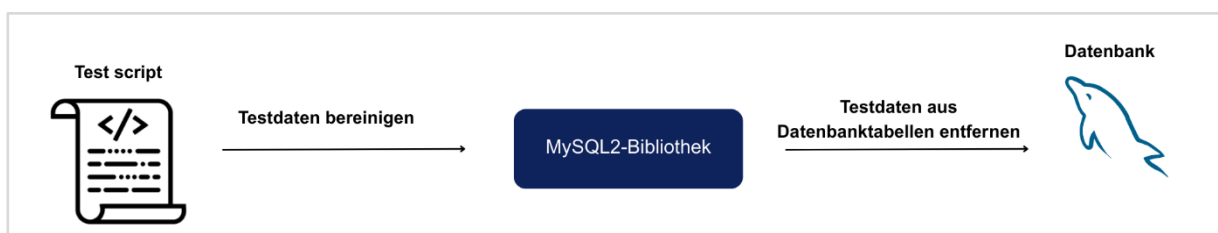


Abbildung 4: Aufräumen der Testdaten nach Testende

## A.9 Pflichtenheft (Auszug)

In folgendem Auszug aus dem Pflichtenheft wird die geplante Umsetzung der im Lastenheft definierten Anforderungen beschrieben:

### Umsetzung der Anforderungen

#### 1. Framework und Architektur

- Implementierung mit Node.js und Playwright als Test-Framework
- Strukturierung nach dem Page-Object-Model-Pattern für bessere Wartbarkeit
- Modulare Verzeichnisstruktur mit Trennung von Tests, Aktionen und Testdaten

#### 2. Browserunterstützung

- Konfiguration von Testprojekten für Chrome, Firefox und Edge in playwright.config.js
- Implementierung mit browserunabhängigen Selektoren für konsistente Ergebnisse

#### 3. Testdatenverwaltung

- Anbindung an MySQL-Testdatenbank mittels mysql2-Modul
- Testdaten in JSON-Format in /fixtures-Verzeichnissen
- Automatisierte Datenbankbefüllung und -bereinigung über die Seeder-Klasse

#### 4. Isolation und Unabhängigkeit

- Serialisierung der Tests mit test.describe.configure({ mode: 'serial' })
- Isolierte Browser-Kontexte für jeden Test
- Separate Testdatenbank mit automatischer Initialisierung und Bereinigung

#### 5. Fehlerdokumentation

- Automatische Screenshot-Erstellung bei fehlgeschlagenen Tests
- Integrierter HTML-Reporter für übersichtliche Ergebnisdarstellung
- Trace-Funktion zur detaillierten Aufzeichnung des Testablaufs

#### 6. Wiederverwendbarkeit

- Kapselung häufiger Aktionen (z.B. Login) in wiederverwendbare Klassen

- Zentrale DB-Klasse für standardisierte Datenbankoperationen
- Modularisierung der Testskripte für bessere Übersichtlichkeit

#### 7. **Datenkonsistenz**

- Transaktionale Ausführung von Datenbankoperationen
- Definierte Setup- und Teardown-Prozesse für jeden Test
- Versionskontrolle der Testdaten zur Sicherstellung der Konsistenz

#### 8. **Erweiterbarkeit**

- Standardisierte Schnittstellen für neue Testkomponenten
- Dokumentierte Teststruktur für einfache Erweiterung
- Gemeinsame Nutzung von Basisklassen und Hilfsfunktionen

Diese technische Umsetzung gewährleistet ein robustes, wartbares und erweiterbares E2E-Testsystem, das alle im Lastenheft definierten Anforderungen erfüllt.

## A.10 Listing der Seeder-Klasse

```

const fs = require('fs');
const path = require('path');
const DB = require('../db');

/**
 * @class Seeder
 * @description Klasse zur Verwaltung von Test-Daten in der Datenbank.
 * @author Ayoub Aalaou
 */
class Seeder {
  /**
   * @function seederTable
   * @description Laedt JSON-Daten dynamisch und speichert sie in einer bestimmten
   Tabelle.
   * @param {string} table - Name der Zieltabelle in der Datenbank.
   * @param {string} module - Name des Modulordners, in dem sich die Fixture befindet.
   * @param {string} fixtureName - Name der JSON-Fixtur-Datei ohne Erweiterung.
   * @returns {Promise<void>}
   * @author Ayoub Aalaou
   */
  static async seederTable(table, module, fixtureName) {
    const filePath = path.resolve(__dirname,
    `../fixtures/${module}/${fixtureName}.json`);
    const data = JSON.parse(fs.readFileSync(filePath, 'utf-8'));

    await DB.truncateTable(table); // Ensure table is clean
    for (const record of data) {
      await DB.create(table, record);
    }
  }

  /**
   * @function setupScenario
   * @description Benutzerdefinierte Seeder-Logik fuer spezifische Testszenarien.
   * @param {Array<Object>} scenarioData - Array von Objekten mit Tabellen- und
   Fixture-
   Informationen.
   * @param {string} scenarioData[table] - Name der Datenbanktabelle.
   * @param {string} scenarioData[module] - Name des Modulordners.
   * @param {string} scenarioData[fixture] - Name der Fixture-Datei.
   * @returns {Promise<void>}
   * @author Ayoub Aalaou
   */
  static async setupScenario(scenarioData) {
    for (const { table, module, fixture } of scenarioData) {
      await Seeder.seederTable(table, module, fixture);
    }
  }

  /**
   * @function teardownScenario
   * @description Aufräumlogik zur Bereinigung nach Tests.
   * @param {Array<string>} tables - Liste von Tabellennamen, die geleert werden
   sollen.
   * @returns {Promise<void>}
   * @author Ayoub Aalaou
   */
  static async teardownScenario(tables) {
    for (const table of tables) {
      await DB.truncateTable(table);
    }
  }
}

```

```

    }
  }

  module.exports = Seeder;

```

Listing 1: seeder.js

## A.11 Listing der DB-Klasse

```

const mysql = require('mysql2/promise');
const config = require('./dbConfig');

/**
 * @class DB
 * @description Klasse fuer Datenbankoperationen mit MySQL.
 * @author Ayoub Aalaou
 */
class DB {
  static connection;

  /**
   * @function initConnection
   * @description Erstellt einen Verbindungs-Pool zur Wiederverwendung.
   * @returns {Promise<void>}
   * @author Ayoub Aalaou
   */
  static async initConnection() {
    if (!DB.connection) {
      DB.connection = await mysql.createPool(config);
    }
  }

  /**
   * @function query
   * @description Fuehrt eine SQL-Abfrage aus.
   * @param {string} sql - SQL-Anweisung zur Ausfuehrung.
   * @param {Array} params - Parameter fuer die vorbereitete Anweisung.
   * @returns {Promise<Array>} Ergebniszeilen der Abfrage.
   * @author Ayoub Aalaou
   */
  static async query(sql, params = []) {
    await DB.initConnection();
    const [rows] = await DB.connection.execute(sql, params);
    return rows;
  }

  // CRUD Operations
  /**
   * @function create
   * @description Erstellt einen neuen Datensatz in der angegebenen Tabelle.
   * @param {string} table - Name der Zieltabelle.
   * @param {Object} data - Objekt mit Spaltennamen und Werten fuer den Eintrag.
   * @returns {Promise<Object>} Ergebnis der Einfuegeoperation.
   * @author Ayoub Aalaou
   */
  static async create(table, data) {
    const columns = Object.keys(data).join(', ');
    const placeholders = Object.keys(data).map(() => '?').join(', ');
    const values = Object.values(data);
    const sql = `INSERT INTO ${table} (${columns}) VALUES (${placeholders})`;
    return await DB.query(sql, values);
  }
}

/**

```

```

* @function read
* @description Liest Datensätze aus der angegebenen Tabelle.
* @param {string} table - Name der Quelltable.
* @param {string} where - WHERE-Klausel fuer die Abfrage.
* @param {Array} params - Parameter fuer die WHERE-Klausel.
* @returns {Promise<Array>} Gefundene Datensätze.
* @author Ayoub Aalaou
*/
static async read(table, where = '', params = []) {
    const sql = `SELECT * FROM ${table} ${where}`;
    return await DB.query(sql, params);
}

/**
* @function update
* @description Aktualisiert Datensätze in der angegebenen Tabelle.
* @param {string} table - Name der zu aktualisierenden Tabelle.
* @param {Object} data - Objekt mit zu aktualisierenden Spalten und Werten.
* @param {string} where - WHERE-Klausel fuer die Aktualisierung.
* @param {Array} params - Parameter fuer die WHERE-Klausel.
* @returns {Promise<Object>} Ergebnis der Aktualisierungsoperation.
* @author Ayoub Aalaou
*/
static async update(table, data, where, params) {
    const setClause = Object.keys(data).map(col => `${col} = ?`).join(', ');
    const values = [...Object.values(data), ...params];
    const sql = `UPDATE ${table} SET ${setClause} ${where}`;
    return await DB.query(sql, values);
}

/**
* @function delete
* @description Loescht Datensätze aus der angegebenen Tabelle.
* @param {string} table - Name der Tabelle, aus der geloescht werden soll.
* @param {string} where - WHERE-Klausel fuer die Loeschoperation.
* @param {Array} params - Parameter fuer die WHERE-Klausel.
* @returns {Promise<Object>} Ergebnis der Loeschoperation.
* @author Ayoub Aalaou
*/
static async delete(table, where, params = []) {
    const sql = `DELETE FROM ${table} ${where}`;
    return await DB.query(sql, params);
}

/**
* @function truncateTable
* @description Leert eine Tabelle vollstaendig.
* @param {string} table - Name der zu Leerenden Tabelle.
* @returns {Promise<Object>} Ergebnis der TRUNCATE-Operation.
* @author Ayoub Aalaou
*/
static async truncateTable(table) {
    return await DB.query(`TRUNCATE TABLE ${table}`);
}

/**
* @function closeConnection
* @description Schliesst die Datenbankverbindung.
* @returns {Promise<void>}
* @author Ayoub Aalaou
*/
static async closeConnection() {
    if (DB.connection) {
        await DB.connection.end();
    }
}

```

```
        DB.connection = null;
    }
}

module.exports = DB;
```

Listing 2: DB.js

### A.12 Listing von Beispiel-Testdaten (Fixtures) der Benutzer

```
[
  {
    "id": "1",
    "datum_erstellt": "wert",
    "benutzername": "wert",
    "email": "beispiel@gmail.com",
    ...
  }
]
```

Listing 3: cm\_user.json

### A.13 Listing mit Informationen zu Testdaten der Benutzer

```
[
  {
    "username": "wert1",
    "status": "gültig",
    "type": "Admin",
    "permissions": {
      "kann_anmelden": true,
      "kann_bearbeiten": true,
      "kann_löschen": true
    },
    "description": "Administrator-Benutzer mit vollständigem Zugriff."
    ...
  }
]
```

Listing 4: cm\_user\_metadata.json

### A.14 Listing des Testskripts zum Login-Prozess

```
const { test, expect } = require('@playwright/test');
const DB = require('../db');
const Seeder = require('../services/seeder');

const fs = require('fs');
const path = require('path');

test.describe('Login Validierung Test-Suite', () => {
  // Tests seriell ausführen, um DB-Konflikte zu vermeiden
  test.describe.configure({ mode: 'serial' });

  test.beforeEach(async () => {
    try {
      // Tabellen zuruecksetzen, um Primaerschlueselkonflikte zu vermeiden
      await Seeder.teardownScenario(['cm_kennwort', 'cm_user']);

      await Seeder.setupScenario([
        { table: 'cm_user', module: 'cm', fixture: 'cm_user' },
        { table: 'cm_kennwort', module: 'cm', fixture: 'cm_kennwort' },
      ]);
    }
  });
});
```

```

    } catch (error) {
        console.error('Fehler beim Setup:', error);
        throw error;
    }
});

test.afterEach(async () => {
    await Seeder.teardownScenario(['cm_user', 'cm_kennwort']);
});

test.afterAll(async () => {
    await DB.closeConnection();
});

test('Erfolgreiche Anmeldung mit gueltigen Benutzeranmeldedaten', async ({ page, browser }) => {
    await page.goto('http://localhost/cm_45_initial/?gui=v2#Login');
    await page.waitForLoadState('networkidle');

    const [loginButtonText, usernameText, passwordText] = await page.evaluate(() => {
        const Sprachausdruck = require('cm/Sprachausdruck');
        return [
            Sprachausdruck.get('einloggen'),
            Sprachausdruck.get('benutzername'),
            Sprachausdruck.get('kennwort')
        ];
    });

    await page.fill(`input[placeholder="${usernameText}"]`, 'admin_ayb');
    await page.fill(`input[placeholder="${passwordText}"]`, 'Start001');

    await page.click(`button:has-text("${loginButtonText}")`);
    await page.waitForURL(/.*DashBoard.*/);

    const currentURL = page.url();
    expect(currentURL).toContain('DashBoard');

    await page.locator("//span[@class='webix_icon cmi cmi-menue']").waitFor({ state: 'visible' });
    await page.locator("//span[@class='webix_icon cmi cmi-menue']").click();

    const div = page.locator('div[webix_tm_id="dashboard"]');
    await expect(div).toBeVisible();
});

test('Zugriffsverweigerung bei ungueltigem Benutzernamen', async ({ page }) => {
    await page.goto('http://localhost/cm_45_initial/?gui=v2#Login');
    await page.waitForLoadState('networkidle');

    const [loginButtonText, errorMessageText, usernameText, passwordText] = await page.evaluate(() => {
        const Sprachausdruck = require('cm/Sprachausdruck');
        return [
            Sprachausdruck.get('einloggen'),
            Sprachausdruck.get('benutzername_kennwort_falsch'),
            Sprachausdruck.get('benutzername'),
            Sprachausdruck.get('kennwort')
        ];
    });

    await page.fill(`input[placeholder="${usernameText}"]`, 'nonexistent_user');
    await page.fill(`input[placeholder="${passwordText}"]`, 'Start001');

    await page.click(`button:has-text("${loginButtonText}")`);

```

```
// Fehlermeldung pruefen
const errorDiv = await page.locator('div.webix_scroll_cont.webix_template');
await expect(errorDiv).toBeVisible();
await expect(errorDiv).toHaveText(errorMessageText);

const currentURL = page.url();
expect(currentURL).toContain('Login');
expect(currentURL).not.toContain('DashBoard');
});

test('Zugriffsverweigerung bei ungueltigem Passwort', async ({ page }) => {
  await page.goto('http://localhost/cm_45_initial/?gui=v2#Login');
  await page.waitForLoadState('networkidle');

  const [loginButtonText, errorMessageText, usernameText, passwordText] = await
page.evaluate(() => {
    const Sprachausdruck = require('cm/Sprachausdruck');
    return [
      Sprachausdruck.get('einloggen'),
      Sprachausdruck.get('benutzername_kennwort_falsch'),
      Sprachausdruck.get('benutzername'),
      Sprachausdruck.get('kennwort')
    ];
  });

  await page.fill(`input[placeholder="${usernameText}"]`, 'admin_ayb');
  await page.fill(`input[placeholder="${passwordText}"]`, 'invalid_password');

  await page.click(`button:has-text("${loginButtonText}")`);

  // Fehlermeldung pruefen
  const errorDiv = await page.locator('div.webix_scroll_cont.webix_template');
  await expect(errorDiv).toBeVisible();
  await expect(errorDiv).toHaveText(errorMessageText);

  const currentURL = page.url();
  expect(currentURL).toContain('Login');
  expect(currentURL).not.toContain('DashBoard');
});

test('Login-Button deaktiviert bei leeren Eingabefeldern', async ({ page }) => {
  await page.goto('http://localhost/cm_45_initial/?gui=v2#Login');
  await page.waitForLoadState('networkidle');

  const [loginButtonText, usernameText, passwordText] = await page.evaluate(() => {
    const Sprachausdruck = require('cm/Sprachausdruck');
    return [
      Sprachausdruck.get('einloggen'),
      Sprachausdruck.get('benutzername'),
      Sprachausdruck.get('kennwort')
    ];
  });

  const loginButton = page.locator(`button:has-text("${loginButtonText}")`);
  const usernameField = page.locator(`input[placeholder="${usernameText}"]`);
  const passwordField = page.locator(`input[placeholder="${passwordText}"]`);

  // Felder Leeren
  await usernameField.fill('');
  await passwordField.fill('');

  // Pruefung 1: Beide Felder Leer
  await expect(loginButton).toBeDisabled();
}
```

```
// Pruefung 2: Nur Benutzername
await usernameField.fill('admin_ayb');
await expect(loginButton).toBeDisabled();

await usernameField.fill('');
await expect(loginButton).toBeDisabled();

// Pruefung 3: Nur Passwort
await passwordField.fill('Start001');
await expect(loginButton).toBeDisabled();

await passwordField.fill('');
await expect(loginButton).toBeDisabled();
});
});
```

Listing 5: login.spec.js

## A.15 Listing: Konfigurationsobjekt für die Datenbankverbindung

```
/**
 * @module dbConfig
 * @description Konfigurationsobjekt fuer die Datenbankverbindung.
 * @author Ayoub Aalaou
 */
module.exports = {
  /**
   * @property {string} host - Hostname des Datenbankservers.
   */
  host: 'localhost',

  /**
   * @property {string} user - Benutzername fuer die Datenbankverbindung.
   */
  user: 'BENUTZER_PLATZHALTER',

  /**
   * @property {string} password - Passwort fuer die Datenbankverbindung.
   */
  password: 'PASSWORT-PLATZHALTER',

  /**
   * @property {string} database - Name der zu verwendenden Datenbank.
   */
  database: 'DATENBANK_NAME_PLATZHALTER',

  /**
   * @property {number} port - Port des Datenbankservers.
   */
  port: 3306
};
```

Listing 6: dbConfig.js

## A.16 Listing: Wiederverwendbares Login-Modul

```
const fs = require('fs');
const path = require('path');
const {test} = require("@playwright/test");
const Seeder = require("../services/seeder");

// Benutzerdaten aus JSON-Datei Laden
const users = JSON.parse(
  fs.readFileSync(path.resolve(__dirname, '../fixtures/cm/cm_user.json'), 'utf-8')
```

```

);

/**
 * Klasse zur Verwaltung von Login-Aktionen in Tests
 */
class LoginAction {
    /**
     * Testdaten fuer Login-Tests einrichten
     */
    static async setup() {
        try {
            await Seeder.setupScenario([
                { table: 'cm_user', module: 'cm', fixture: 'cm_user' },
                { table: 'cm_kennwort', module: 'cm', fixture: 'cm_kennwort' },
            ]);
        } catch (error) {
            console.error('Fehler beim Setup:', error);
            throw error;
        }
    }

    /**
     * Testdaten nach Tests bereinigen
     */
    static async teardown() {
        await Seeder.teardownScenario(['cm_user', 'cm_kennwort']);
    }

    /**
     * Fuehrt einen Login-Vorgang mit den angegebenen Anmeldedaten durch
     * @param {Page} page - Playwright Page-Objekt
     * @param {string} username - Benutzername fuer Login (Standard: admin_ayb)
     * @param {string} password - Passwort fuer Login (Standard: Start001)
     * @returns {Promise<boolean>} - true bei erfolgreichem Login
     */
    static async Login(page, username = 'admin_ayb', password = 'Start001') {
        try {
            const testUser = users.find(user => user.benutzername === username);

            // Zur Login-Seite navigieren
            await page.goto('http://localhost/cm_45_initial/?gui=v2#Login', {
                waitUntil: 'networkidle'
            });

            // Login-Button-Text ermitteln
            const result = await page.evaluate(() => {
                const Sprachausdruck = require('cm/Sprachausdruck');
                return Sprachausdruck.get('einloggen');
            });

            // Anmeldedaten eingeben
            await page.fill('input[placeholder="User name"]', testUser.benutzername);
            await page.fill('input[placeholder="Password"]', password);

            // Login-Button klicken und auf Navigation warten
            await Promise.all([
                page.waitForLoadState('networkidle'),
                page.click(`button:has-text("${result}")`)
            ]);

            // Auf Dashboard warten
            await page.waitForSelector('div[webix_tm_id="dashboard"]', { state:
'visible', timeout: 10000 });

```

```

        return true;
      } catch (error) {
        console.error('Login fehlgeschlagen:', error);
        throw error;
      }
    }
  }
}

module.exports = LoginAction;

```

Listing 7: login.js

## A.17 Screenshot: Konfigurationsdatei für Playwright

```

// @ts-check
const { defineConfig, devices } = require('@playwright/test');

/**
 * Read environment variables from file.
 * https://github.com/motdotla/dotenv
 */
// require('dotenv').config({ path: path.resolve(__dirname, '.env') });

/**
 * @see https://playwright.dev/docs/test-configuration
 */
module.exports = defineConfig({
  testDir: './playwright/v2/e2e/',
  /* Run tests in files in parallel */
  fullyParallel: true,
  /* Fail the build on CI if you accidentally left test.only in the source code. */
  forbidOnly: !!process.env.CI,
  /* Retry on CI only */
  retries: process.env.CI ? 2 : 0,
  /* Opt out of parallel tests on CI. */
  workers: process.env.CI ? 1 : undefined,
  /* Reporter to use. See https://playwright.dev/docs/test-reporters */
  reporter: 'html',
  /* Shared settings for all the projects below. See
  https://playwright.dev/docs/api/class-testoptions. */
  use: {
    /* Base URL to use in actions like `await page.goto('/')`. */
    // baseURL: 'http://127.0.0.1:3000',

    /* Collect trace when retrying the failed test. See
    https://playwright.dev/docs/trace-viewer */
    trace: 'on-first-retry',
  },

  /* Configure projects for major browsers */
  projects: [
    {
      name: 'chromium',
      use: { ...devices['Desktop Chrome'] },
    },

    {
      name: 'firefox',
      use: { ...devices['Desktop Firefox'] },
    },

    {
      name: 'webkit',
      use: { ...devices['Desktop Safari'] },
    }
  ]
});

```

```

    },
    /* Test against mobile viewports. */
    {
      name: 'Mobile Chrome',
      use: { ...devices['Pixel 5'] },
    },
    {
      name: 'Mobile Safari',
      use: { ...devices['iPhone 12'] },
    },
    /* Test against branded browsers. */
    {
      name: 'Microsoft Edge',
      use: { ...devices['Desktop Edge'], channel: 'msedge' },
    },
    {
      name: 'Google Chrome',
      use: { ...devices['Desktop Chrome'], channel: 'chrome' },
    },
  ],
});

```

Listing 8: playwright.config.js

## A.18 Soll-/Ist-Vergleich

Projektphase	Soll (in Stunden)	Ist (in Stunden)	Differenz
Analyse	6	8	+2
Entwurf	13	14	+1
Implementierung	43	42	-1
Abnahme	7	5	-2
Dokumentation	8	8	0
<b>Summe</b>	<b>77</b>	<b>77</b>	<b>0</b>

Tabelle 5: Soll-/Ist-Vergleich

## A.19 Screenshot - Anmeldemaske ausfüllen

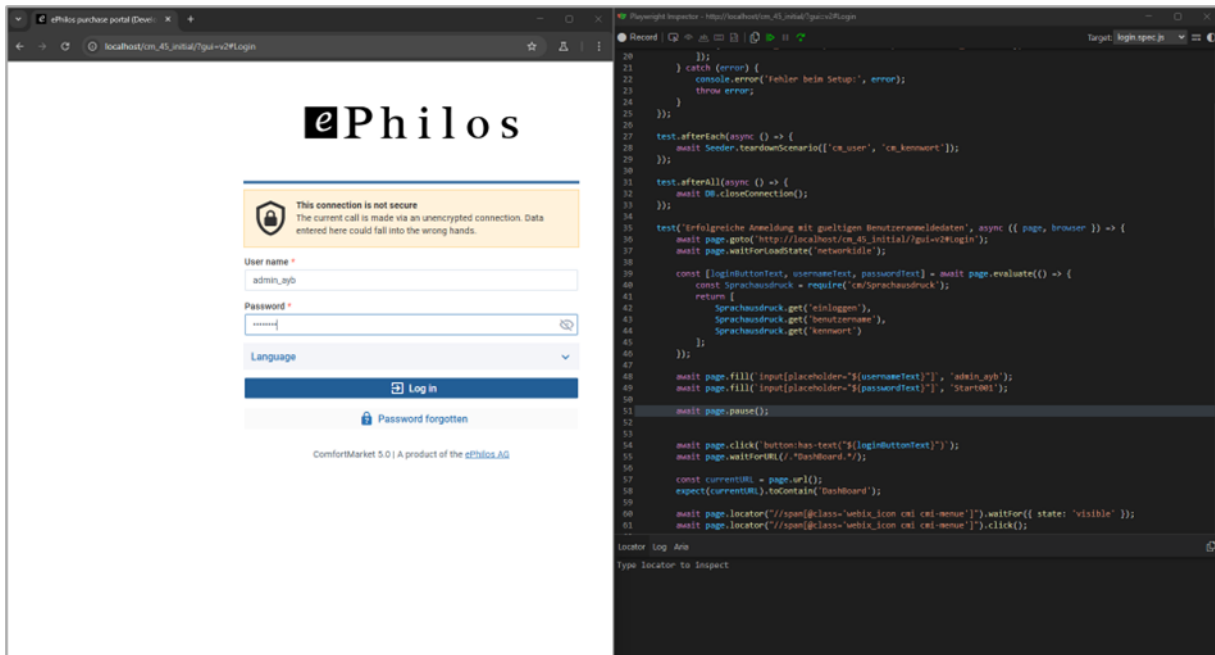


Abbildung 5: Eingabe von Benutzername und Passwort

## A.20 Screenshot: Login-Erfolg mit Dashboard-Weiterleitung

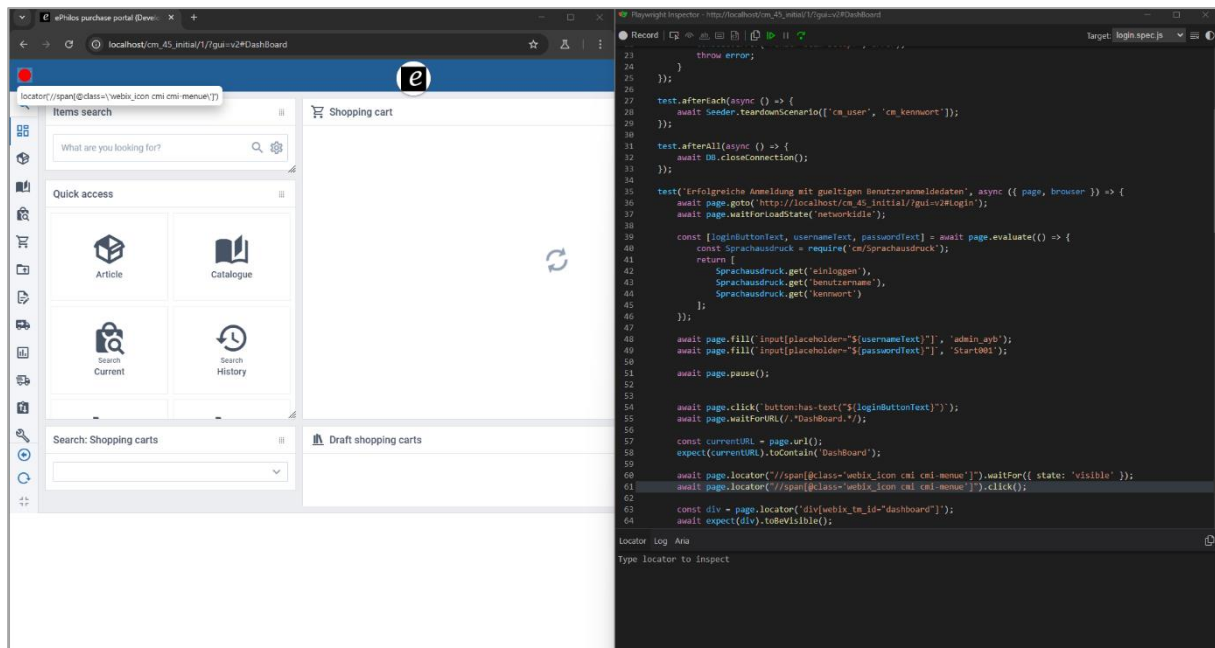


Abbildung 6: Erfolgreicher Login und Weiterleitung zum Dashboard

## A.21 Screenshot: Dashboard-Menü sichtbar

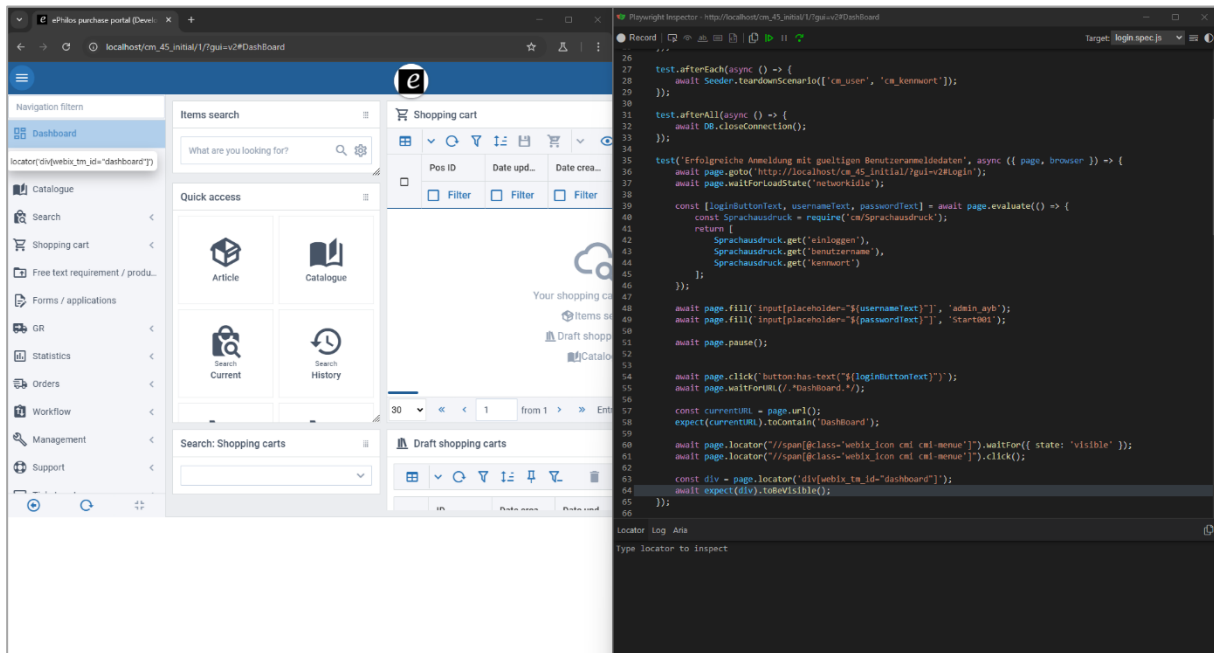


Abbildung 7: Sichtbarkeitsprüfung des Menüeintrags "Dashboard"

## A.22 Screenshot: HTML-Report nach Testdurchführung

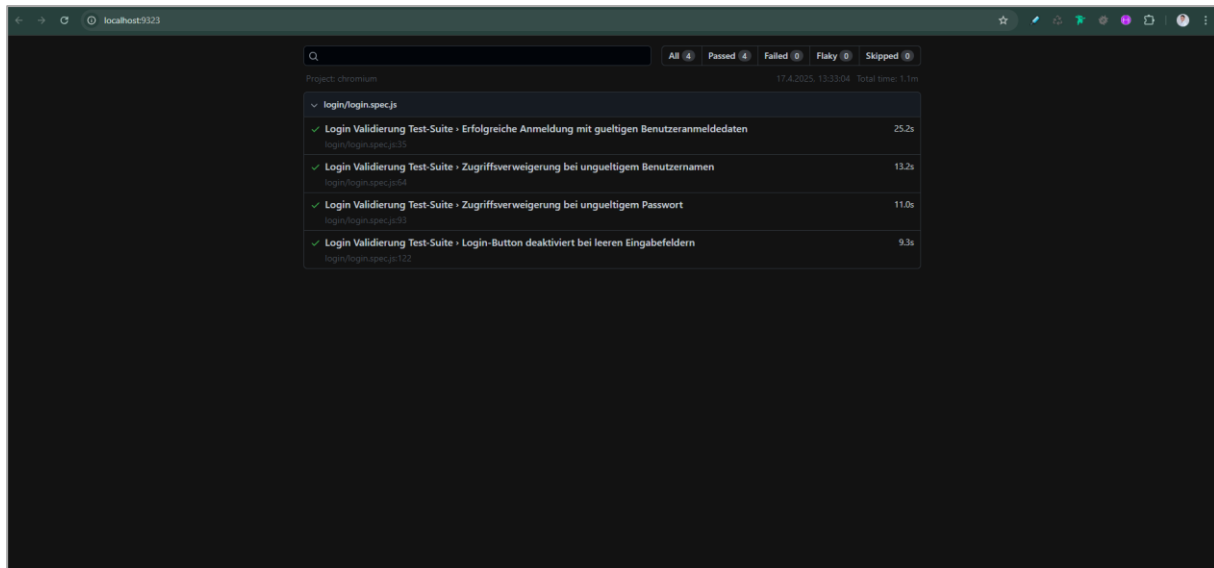


Abbildung 8: Screenshot HTML-Testreport nach Testdurchlauf