HAW HAMBURG

BACHELORTHESIS
Christopher Dührkop

Auf Cypress basierendes schichtenübergreifendes Testkonzept für Webapplikationen im Kontext von Smart-Home-Anwendungen

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Christopher Dührkop

Auf Cypress basierendes schichtenübergreifendes Testkonzept für Webapplikationen im Kontext von Smart-Home-Anwendungen

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

Betreuender Prüfer: Prof. Dr. Bettina Buth
Zweitgutachter: Prof. Dr. Stefan Sarstedt

Eingereicht am: 06. Dezember 2021

Christopher Dührkop

Thema der Arbeit

Auf Cypress basierendes schichtenübergreifendes Testkonzept für Webapplikationen im Kontext von Smart-Home-Anwendungen

Stichworte

Angular, Cypress, schichtenübergreifend, Smart-Home, Softwareschichten, teamübergreifend, Testautomatisierung

Kurzzusammenfassung

Für eine an Smart-Home-Geräte angelehnte Webapplikation wird ein schichtenübergreifendes Testkonzept erstellt und eine Testautomatisierung mit Cypress demonstriert. Die gesammelten Erfahrungen führen zu einer Bewertung, ob Cypress für schichtenübergreifende Testautomatisierung und eine teamübergreifende Nutzung empfohlen werden kann.

Christopher Dührkop

Title of Thesis

Cypress-based concept for testing all layers of a web application in the context of Smart-Home applications

Keywords

Angular, Cypress, cross-layer, Smart Home, software layers, cross-team, test automation

Abstract

A cross-layer test concept is created for a web application with a user interface based on smart home devices and its test automation is demonstrated using Cypress. The experience gained leads to an evaluation of whether Cypress can be recommended for cross-layer test automation and cross-team use.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Dieses Kapitel beschreibt die Herleitung und Relevanz des für diese Ausarbeitung gewählten Themas und die Zielsetzung, die bei der Erarbeitung verfolgt wird. Außerdem wird beschrieben, wie die Arbeit gegliedert ist.

1.1 Ausgangslage und Forschungsinteresse

Webbasierte Anwendungen werden stetig vielfältiger und komplexer. “Die Softwareentwicklung wird sich in den kommenden Jahren mit einer immer höheren Komplexität und stark steigenden Sicherheitsanforderungen konfrontiert sehen” (it-daily.net, 2021). Ein Trend, der zur Entwicklung komplexer Softwareanwendungen beiträgt, ist die steigende Nachfrage sogenannter Smart-Home-Anwendungen, also Kombinationen aus Software und Hardware, welche verschiedene Geräte im Haushalt miteinander verbinden (vgl. Vailshery, 2021) Hierzu gehören smarte Haushaltsgeräte wie Kaffeemaschinen oder Staubsaugerroboter ebenso wie mit dem Internet verbundene Anwendungen zur Gebäudesicherheit oder auch Lichtsteuerung (vgl. Brandt, 2021). Laut Statistischem Bundesamt (2021) haben im 1. Quartal 2020 in Deutschland rund 5% der Menschen ab 10 Jahren smarte Haushaltsgeräte verwendet - Das sind rund 3,3 Millionen Menschen. Unabhängig von der Art der Anwendung “[...] ist die IT-Branche weit davon entfernt, fehlerfreie Software entwickeln zu können” (Spillner & Linz, 2012, S. xviii). Somit ist davon auszugehen, dass Tests für ein jedes Projekt, und nicht zuletzt für Smart-Home-Anwendungen, von Vorteil sind, um die Software auf Fehler zu überprüfen.

Angesichts der beschriebenen Komplexität von Software ist das Testen ebendieser nicht immer ein leichtes Unterfangen. Besteht eine Anwendung aus verschiedenen Komponenten, so sollte auch jede einzelne Komponente getestet werden. Je nach Architektur des Projekts kann die Funktionalität des Gesamtsystems davon abhängen, dass alle Komponenten

einwandfrei funktionieren. Fällt eine einzige Komponente aus, von welcher andere Bereiche der Software abhängig sind, kann gegebenenfalls die gesamte Anwendung abstürzen. Dies kann für Unternehmen und Nutzer[1] je nach Anwendungsfall weitreichende Folgen haben. Daher müssen insbesondere die automatisierten Tests innerhalb der einzelnen Komponenten, die sogenannten Komponententests, ein hohes Maß an Zuverlässigkeit aufweisen. Ebenso verhält es sich für Tests von vernetzten Komponenten, die Integrationstests, als auch für die Tests des Gesamtsystems, die sogenannten Systemtests. Diese beiden Testebenen können auch mehrere Softwareschichten abdecken, wenn die zu testenden Komponenten aufgrund der Projektarchitektur über diese verteilt sind.

Ein zusätzlicher Schwierigkeitsgrad beim Testen von Software ergibt sich daraus, dass an den Komponenten eines Projekts gegebenenfalls verschiedene Entwicklerteams arbeiten. Tests können in unterschiedlichen Programmiersprachen oder Frameworks verfasst werden. Diese werden womöglich nicht in allen Teams oder von jedem Entwickler gleichermaßen verwendet. Der Fall, dass Tests auch von Kunden durchgeführt werden, wird in dieser Arbeit nicht behandelt. Der Austausch über entwickelte Tests kann in solchen Fällen aufwändig ausfallen. Dieser Effekt kann verstärkt werden, wenn Entwickler mit abweichenden Kenntnisständen aufeinandertreffen. Eine erschwerte Kommunikation kann u.a. zur Folge haben, dass die Behebung gefundener Fehler in die Länge gezogen wird, oder gesetzte Ziele aus den Augen verloren werden.

Angesichts der beschriebenen Komplexität von sowohl der Softwarearchitektur als auch der Entwicklungsprozesse und teamübergreifenden Abstimmungsschleifen wird schnell klar, was Edward Lenssen, CEO von Beech IT, mit der folgenden Äußerung meint: „Die Schere zwischen den steigenden Anforderungen an Software und den Ressourcen an Programmierern, die diese Software entwickeln sollen, geht immer weiter auseinander“ (it-daily.net, 2021).

Werden die Anforderungen an eine Software komplexer, werden es offensichtlich auch ihre Tests. Es ist daher sinnvoll, wenn Unternehmen auf eine Automatisierung von Tests setzen, um

[1] Die Verwendung der maskulinen Schreibweise eines Wortes wie z.B. Nutzer, Entwickler oder Tester, steht in dieser Ausarbeitung zum Zwecke einer besseren Lesbarkeit stets stellvertretend für alle Geschlechter.

den manuellen Arbeits- bzw. Wartungsaufwand zu reduzieren. Hierdurch können langfristig Ressourcen für komplexere Aufgaben freigesetzt werden und der „Turnus von Entwicklung und Produktivsetzung [beschleunigt]“ (Oppmann, 2019) werden. Insgesamt führt sie auch zu einer höheren Softwarequalität und -stabilität, da eine manuelle Ausführung der „immer gleichen [Tests] als repetitive Aufgabe dafür prädestiniert“ (ebd.) ist, zu Flüchtigkeitsfehlern zu führen. Es gibt verschiedene Frameworks für Testautomatisierung, die sich darin unterscheiden, inwiefern sie die Bestandteile von Software abdecken. Eines dieser Frameworks ist Cypress (vgl. Cypress.io Overview, 2021) und scheint in diesem Kontext besonders vielversprechend. Schließlich geht es über die grundlegenden Funktionen, wie das Testen von Benutzeroberflächen im Browser hinaus. Es lassen sich beispielsweise ebenso die Datenbankanbindung und API einer Anwendung, oder der Download von Dateien von verteilten Webseiten und Servern testen.

Zusammenfassend ist festzuhalten, dass Softwareentwicklung v.a. vor dem Hintergrund immer komplexer werdender Anforderungen ein vielschichtiges Unterfangen ist, bei dem Fehler nie gänzlich ausgeschlossen werden können, weshalb Tests ein elementarer Bestandteil sein sollten. Nicht nur müssen verschiedene und Komponenten orchestriert werden. Darüber hinaus kann insbesondere in schichtenübergreifenden Projekten mit mehreren involvierten Teams die Kommunikation durch variierende Vorgehensweisen in der Testentwicklung erschwert werden. Aus den genannten Gründen ist anzunehmen, dass es für Unternehmen von Vorteil ist, wenn der Großteil der Entwickler auf einer geteilten, konsistenten Ebene kommunizieren kann, da dies sowohl die Erstellung neuer Tests als auch die Behebung von gefundenen Fehlern beschleunigen kann. Hierfür ist es hilfreich, wenn mehrere Teams dieselben Tools verwenden. Wird bspw. ein einheitliches Framework zur Testautomatisierung eingeführt und über mehrere Teams hinweg als offiziell zu nutzender Standard etabliert, könnte dies die Kommunikation zwischen den Entwicklern stark vereinfachen.

Insgesamt wird vermutet, dass ein teamübergreifend festgelegtes Testautomatisierungsframework wie Cypress den Austausch zwischen Teams und Entwicklern erleichtern, Ergebnisse verbessern und die Effizienz erhöhen kann.

1.2 Zielsetzung

In dieser Arbeit soll auf explorative Weise geprüft werden, ob oder wie gut ein möglichst großer Anteil der grundlegenden automatisierbaren Tests eines schichtenübergreifenden Systems mit Cypress als Testautomatisierungsframework abgedeckt werden kann. Ziel soll es sein, basierend auf den Ergebnissen eine Empfehlung an Entwickler von Webapplikationen aussprechen zu können, inwiefern Cypress für den Einsatz in diesem Kontext geeignet ist oder nicht. Dieses Ziel soll durch die Beantwortung der folgenden Forschungsfragen erreicht werden:

- Wie gut ist Cypress für die **schichtenübergreifende** Testautomatisierung geeignet?
- Kann Cypress für die **teamübergreifende** Nutzung empfohlen werden?

Für den Kontext dieser Ausarbeitung wird von Unternehmen ausgegangen, deren Teams an verschiedenen Komponenten desselben Projektes arbeiten. Des Weiteren wird angenommen, dass die Umsetzung dieses Projektes in Form einer Schichtenarchitektur stattfindet, um die in Kapitel 1.1 erwähnten Schwierigkeiten zu berücksichtigen. Hierfür soll als Grundlage ein Demonstrationsobjekt entwickelt werden, welches sich an den Kontext von Smart-Home-Anwendungen annähert. Mit dem gewählten Testautomatisierungsframework Cypress werden anschließend automatisierte Tests geschrieben, die eine grundlegende Abdeckung von getesteten Funktionen erzielen sollen. Eine möglichst vollständige Testabdeckung ist dabei nicht Teil der Zielsetzung. Stellt sich das Ergebnis als zufriedenstellende Lösung heraus, so kann Cypress als Empfehlung an Entwicklerteams von schichtenübergreifenden Webapplikationen ausgesprochen werden. Hierbei wird sich auf die Nutzung in Kombination mit dem erstellten Testkonzept bezogen, jedoch nicht beschränkt. Es können andere Vorgehensweisen gewählt werden und gleichermaßen erfolgreich sein.

Diese Arbeit konzentriert sich auf das Testen von Präsentations-, Logik- und Persistenzschicht, um die Komplexität der Fragestellung überschaubar zu gestalten. Es werden weder Netzwerkprotokolle noch Aspekte der Informationssicherheit in dieser Ausarbeitung berücksichtigt. Im Ausblick in Kapitel 8.3 wird jedoch auf verschiedene denkbare Erweiterungen durch verwandte Thematiken eingegangen.

1.3 Aufbau der Arbeit

Um den Forschungsfragen nachgehen zu können, wird in Kapitel 2 der theoretische Rahmen gelegt. Dafür wird zunächst das verwendete Testautomatisierungsframework Cypress vorgestellt. Des Weiteren werden relevante Konzepte aus dem Bereich des Software Testings erläutert. Hierbei werden sowohl generelle als auch spezifisch auf das Testen von schichtenübergreifenden Webanwendungen bezogene Grundlagen geschaffen.

Basierend auf dieser Wissensgrundlage wird in Kapitel 3 beschrieben, welche Kriterien definiert werden, um eine fundierte Bewertungsgrundlage für Cypress zu bieten. Anschließend werden Anforderungen an das Demonstrationsobjekt gestellt. Diese definieren, über welche Funktionen die zu entwickelnde Webanwendung verfügen soll. Außerdem stecken sie ab, welche Anwendungsszenarien bei seiner Entwicklung bedacht werden müssen, um eine hinreichende Grundlage zur Beantwortung der Forschungsfragen zu bieten.

Nachdem zuvor das Demonstrationsobjekt entworfen wurde, handelt es sich bei Kapitel 4 um die Konzeption des Testansatzes. Es wird eine Auswahl von Methoden vorgestellt, welche von Cypress zur Verfügung gestellt werden. Die von der Anwendung geforderten Funktionen werden den unterschiedlichen Softwareschichten zugeordnet und die Testarten zur Abdeckung der verschiedenen Schichten erläutert. Schließlich werden einige Aspekte der Webentwicklung aufgelistet, welche bewusst aus dem Rahmen dieser Ausarbeitung ausgeschlossen sind.

Daraufhin wird in Kapitel 5 erläutert, auf welche Art und Weise und in welchem Maße die zuvor eingeplanten Konzepte letztendlich während der Entwicklung des Demonstrationsobjekts umgesetzt werden. Es wird ausgeführt, wo in der Anwendungsoberfläche die einzelnen Operationen aus der Anforderungsanalyse in Kapitel 3.4 wiederzufinden sind, um ein klares Bild vom Demonstrationsobjekt zu geben.

In Kapitel 6 wird die Umsetzung der konzeptionierten Tests beschrieben, um explorativ Erkenntnisse zu sammeln, inwiefern bei der Nutzung von Cypress besondere Herausforderungen auftreten. Hierbei wird sich an den Testarten für die verschiedenen Softwareschichten orientiert. Zu jeder umgesetzten Testart werden Beispiele aufgeführt und genauer erklärt. Zusätzlich wird auf einzelne der in Kapitel 3.2 formulierten Anforderungen

eingegangen, um zu demonstrieren, wie deren Ausprägungen unmittelbar während der Implementierung von Tests überprüft werden konnten.

Abschließend werden in Kapitel 7 alle gewonnen Erkenntnisse zusammengeführt, um in einer Evaluation die definierten Bewertungskriterien auszuwerten. Hierbei wird auf die zufriedenstellenden Erfahrungen ebenso eingegangen, wie auf die Hürden und Herausforderungen, welche die Testautomatisierung unter Cypress mit sich brachte.

Die vorliegende Arbeit schließt mit Kapitel 8, indem sie zunächst kurz zusammengefasst wird. Anschließend wird in einem Fazit die Evaluierung von Cypress destilliert, um die Forschungsfragen abschließend zu beantworten. Des Weiteren wird ein Ausblick zu offenen Forschungsfeldern gegeben.

2 Grundlagen

Im Folgenden wird eine Wissensgrundlage geschaffen, um die wichtigsten Prinzipien des Softwaretestens ebenso wie die verwendeten Technologien und für diese Arbeit relevante Begrifflichkeiten einzuordnen.

2.1 Einführung in das Testen

In diesem Kapitel soll ein Überblick darüber gegeben werden, welche Arten von Tests es im Zusammenhang mit der Entwicklung von webbasierten Anwendungen und verteilten Systemen (s. Kapitel 4.5.3) gibt, und welche im Kontext der vorliegenden Arbeit relevant sind.

2.1.1 Zielsetzung des Testens

Genau wie bei Industrieprodukten ist auch bei Softwareprodukten zu kontrollieren, ob sie den gestellten Anforderungen entsprechen und damit ihre individuelle Aufgabe wie gefordert erfüllen (vgl. Spillner & Linz, 2012, S.6). Zu diesem Zwecke werden i.d.R. Tests durchgeführt, bevor ein neues Softwareprodukt bzw. eine neue Funktion veröffentlicht wird. Im Gegensatz zu Industrieprodukten sind Softwareanwendungen nicht haptisch greifbar und die Software muss daher für die Tests auf einem Rechner zum Laufen gebracht werden, um überprüfen zu können, ob die Software sich den Anforderungen entsprechend verhält oder fehlerhaft ist (vgl. ebd). Fehler bedeuten in diesem Zusammenhang, dass Anforderungen nicht erfüllt werden und der Ist- von dem Soll-Zustand abweicht. Ein Mangel liegt bei nicht ausreichender Erfüllung von Anforderungen vor (vgl. Spillner & Linz, 2012. S.7).

Hierbei ist zwischen Fehlerwirkung (engl. failure) und Fehlerzustand (engl. bug) zu unterscheiden. Bei der Durchführung von Tests können Fehlerwirkungen festgestellt werden, also Fehler, die nach außen wahrnehmbar sind. Diese sind jeweils auf einen Ursprung, einen Fehlerzustand zurückzuführen. Hierbei wiederum handelt es sich um einen Fehler innerhalb des Systems, z.B. eine vergessene oder fehlerhaft programmierte Funktion (vgl. ebd.).

Die Durchführung von Tests ermöglicht es, Fehlerwirkungen festzustellen, bevor eine Software veröffentlicht und damit einer breiteren Masse an Nutzern zugänglich gemacht wird.

Die Fehler können rechtzeitig durch die Softwaretester an die Softwareentwickler gemeldet werden, welche daraufhin die Fehlerzustände lokalisieren können. Diesen Vorgang nennt man Debugging und er ist klar vom Testen abzugrenzen (vgl. ebd. S.8).

Es ist außerdem zu beachten, dass die Behebung von Fehlern im nächsten Schritt wiederum zu neuen Fehlern führen können, die im Code durch die vorgenommenen Änderungen entstehen. Die vorangegangenen Tests müssen daher nach der Fehlerbehebung wiederholt und ggf. erweitert werden, um neue bzw. veränderte Gegebenheiten abzudecken (vgl. ebd. S.9).

Auch wenn durch die Durchführung von Tests viele Fehler gefunden und anschließend behoben werden können, ist es dennoch grundsätzlich nicht möglich, alle Fehler und Mängel zu identifizieren. “Ein solcher vollständiger Test ist praktisch nicht durchführbar. Durch die Vielzahl kombinatorischer Möglichkeiten ergibt sich eine nahezu unbegrenzte Anzahl an Tests, die durchzuführen wären. Ein solches “Austesten” aller Kombinationen ist nicht möglich.” (vgl. ebd. S.14). "Die Anzahl und Auswahl der Tests liegt somit im Entscheidungs- und Verantwortungsbereich des Projekt- oder Qualitätsverantwortlichen.

2.1.2 Schichtenarchitektur

“Als n-Schichtenarchitektur oder -paradigma [...] bezeichnet man ein Architekturmuster, bei [dem] eine Anwendungs-Komponente in mehrere eigenständige Module unterteilt wird, die schichtenförmig angeordnet sind: Layer 1, Layer 2, ..., Layer n” (Hochschule Augsburg, 2014). Diese Schichten (engl. Layers) werden häufig auch als Tiers bezeichnet.

In dieser Arbeit wird ausschließlich Bezug auf die 3-Schichtenarchitektur genommen, wobei Teile der Inhalte ebenfalls auf Architekturen mit mehr oder weniger Schichten übertragbar sind. “Die 3-Tier-Architektur besteht aus drei Tiers. Tier 1 steht für die Darstellungs- und Eingabeschicht, Tier 2 für den Geschäftsprozess und Tier 3 für die Daten und andere Ressourcen” (ITWissen.info, 2020). Die folgende Grafik zeigt, wie die 3-Schichtenarchitektur aufgebaut ist.

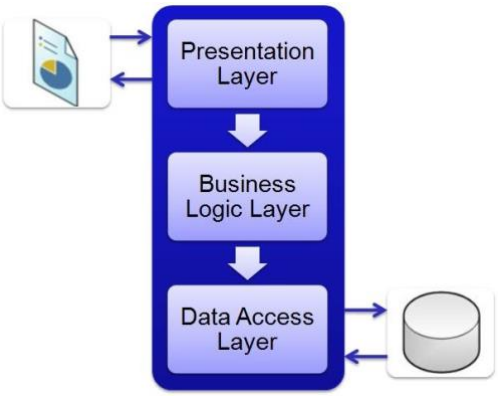

Abbildung 1: Grafik - Drei-Schichten-Modell (Patil, 2021).

Die Präsentationsschicht (hier engl. Presentation Layer) stellt die Website zur Verfügung, sodass Nutzer mit ihr interagieren können, und leitet eingegabgene Daten an die Geschäftslogik oder auch Logikschicht (hier engl. Business Logic Layer) weiter.

Microsoft definiert die Geschäftslogik, in welcher erwähnte Geschäftsprozesse ablaufen, wie folgt: “Business logic is defined as any application logic that is concerned with the retrieval, processing, transformation, and management of application data; application of business rules and policies; and ensuring data consistency and validity” (Microsoft, 2010).

Letztlich ist in der Grafik auch die Persistenzschicht (hier engl. Data Access Layer) abgebildet. Diese verwaltet die Datenhaltung und kommuniziert mit der Datenbank, in welcher die Daten der Anwendung abgelegt werden.

2.1.3 Point of Control und Point of Observation

Als Point of Control oder PoC wird die Softwareschicht oder -komponente bezeichnet, mit welcher interagiert wird, um einen Test zu initialisieren. Hierbei kann es sich gleichermaßen um einen Knopf auf einer Weboberfläche während eines GUI-Tests handeln, wie auch um eine Methode bei einem Komponententest, oder um eine Tabelle, in die Werte während eines Datenbanktests eingetragen werden.

Der Point of Observations oder PoO ist die Stelle in einer Software, welche betrachtet wird, um das Ergebnis eines Tests festzustellen. Dies kann eine Datenbanktabelle sein, in welcher

ein bestimmter Wert erwartet wird, ebenso wie der Rückgabewert einer Methode, oder ein Oberflächenelement, das sich am Ende eines Tests verändert haben sollte.

2.1.4 Testen entlang des Softwareentwicklungszyklus und Testebenen

Das V-Modell nach Boehm spielt eine wichtige Rolle, wenn es darum geht, den Stellenwert des Testens im Softwareentwicklungszyklus sowie den Ablauf der verschiedenen Testarten darzustellen. Die Testaktivitäten werden hierbei als "gleichwertig zur Entwicklung und Programmierung" (Spillner & Linz, 2012, S. 41) beschrieben. Das hier abgebildete V-Modell stellt beide Aktivitätsarten in einem chronologischen Ablaufprozess dar und zeigt, dass sie zueinander korrespondieren.

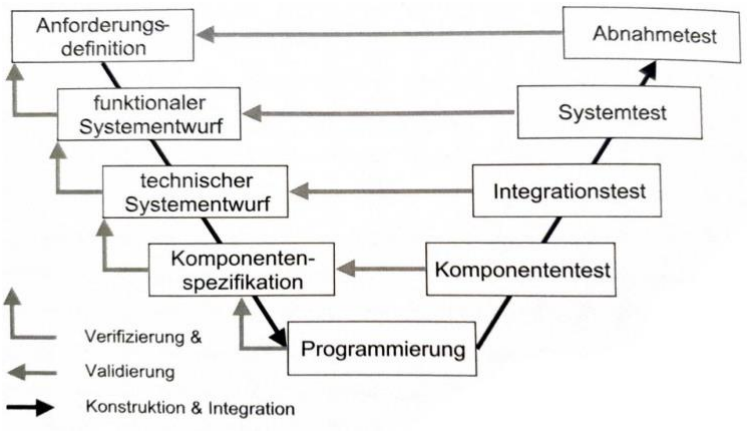

Abbildung 2: Scan - Allgemeines V-Modell (Spillner & Linz, 2012, S. 41).

Eine Definition der ersten vier Entwicklungsschritte soll nicht Teil dieser Ausarbeitung sein (vgl. Spillner & Linz, 2012, S.42). Die in der Abbildung des V-Modells auf der rechten Seite abgebildeten Testebenen sind für diese Arbeit besonders relevanten Schritte eines Entwicklungsprozesses und werden daher im Folgenden näher erläutert. Die übrigen Entwicklungsschritte können in Teilen in den Kapiteln 3 und 5 nachvollzogen werden, sind allerdings nicht eigenständiger Bestandteil dieser Arbeit.

Testebenen

Das V-Modell beschreibt nicht nur den zeitlichen Ablauf der Entwicklungsschritte. Zusätzlich wird für jede Testebene sichtbar gemacht, von welchem Entwicklungsschritt sie das Ergebnis

kontrolliert. D.h. wird während der Komponentenspezifikation ein Aspekt vergessen, fällt dies i.d.R. in den Komponententests auf. Gab es bereits in der Anforderungsdefinition Unklarheiten, werden die entstandenen Fehlerwirkungen oder Mängel meist spätestens im Abnahmetest bemerkt. Dabei unterscheiden sich die Tests nicht nur in ihrer zeitlichen Zuordnung, sondern unterscheiden sich auch technisch stark voneinander: Sie verfolgen unterschiedliche Ziele, nutzen unterschiedliche Testmethoden und Werkzeuge. Personal ist häufig auf eine dieser Arten spezialisiert (vgl. Spillner & Linz, 2012, S.44).

- **Komponententests (engl. Unit-Tests):** Hier werden erstmalig in dem jeweiligen Softwareentwicklungszyklus die zuvor programmierten Softwarekomponenten getestet. Bei diesen handelt es sich typischerweise um Programmmodule, auch Units genannt, bzw. Klassen oder Datenbankskripte (vgl. Spillner & Linz, 2012, S.45). Bei einem Unit-Test wird der jeweilig zu testende Baustein isoliert getestet und auf seine komponenteninternen Anforderungen überprüft. So sollen Einflüsse von außerhalb der jeweiligen Komponente ausgeschlossen werden, um eine mögliche Fehlerwirkung unmittelbar zuordnen zu können (vgl. ebd.).
- **Integrationstests:** Der nächsten Testebene wird vorausgesetzt, dass einzelne Komponenten getestet und anschließend zu größeren Teilsystemen verbunden worden sind. Das Zusammenspiel der Komponenten innerhalb eines Teilsystems bzw. einer Integration kann mit Hilfe von Integrationstests überprüft werden. Ziel dieser Tests ist es, Fehlerzustände in Schnittstellen und besagtem Zusammenspiel ausfindig zu machen (vgl. ebd. S.52).
- **Systemtests:** Bei der dritten Testebene wird das System als Ganzes auf Fehler bzw. Verstöße gegen die Anforderungen an das Softwareprodukt als solches überprüft. Im Gegensatz zu den vorherigen Testebenen wird nun nicht mehr in erster Linie auf technische Spezifikationen aus Sicht des Softwareherstellers geprüft. Stattdessen wird die Sicht des Nutzers eingenommen und aus dessen Sicht überprüft, ob die Anforderungen vollständig und zufriedenstellend umgesetzt sind (vgl. Spillner & Linz, 2012, S. 61).
- **Abnahmetests:** Abnahmetests kommen insbesondere dann zum Tragen, wenn es sich um Software handelt, die für einen Kunden entwickelt wird. Dies ist u.U. der einzige Test, den der Kunde nachvollziehen kann und ggf. sogar selbst durchführt und

verantwortet (vgl. ebd. S.64). Hierbei stehen die Überprüfung von Vertragsanforderungen, der Test auf Benutzerakzeptanz sowie die Akzeptanz durch den Systembetreiber im Mittelpunkt (vgl. ebd. S.66).

Es sei weiterhin angemerkt, dass Software, die einmal alle Tests erfolgreich durchlaufen hat und anschließend veröffentlicht wurde, im Nachhinein i.d.R. weiter gewartet und/oder weiterentwickelt wird. Auch in diesem Zusammenhang werden dedizierte Tests durchgeführt, welche sicherstellen, dass keine neuen Fehler eingeführt werden. Diese sogenannten **Regressionstests** (vgl. Spillner & Linz, 2012, S.77) sind an dieser Stelle jedoch nicht relevant und sollen daher nicht weiter bearbeitet werden.

2.1.5 Funktionale und nicht-funktionale Tests

Wie zuvor beschrieben, variieren Fokus und Ziele der Teststufen in einem Softwareentwicklungsprozess. Dementsprechend kommen auch unterschiedliche Testarten in unterschiedlichem Maße zum Greifen. Eine wichtige Unterscheidung ist jene zwischen funktionalen und nicht-funktionalen Tests.

- **Funktionale Tests:** Mit Hilfe funktionaler Tests wird das Eingabe- und Ausgabeverhalten von Testobjekten dahingehend überprüft, ob es mit den funktionalen Anforderungen an dieses Objekt übereinstimmt (vgl. Spillner & Linz, 2012, S.72). In Phase 1 "Anforderungsdefinition" des V-Modells wird in den namensgebenden Anforderungen das Soll-Verhalten der Software genauestens definiert. Für die anforderungsbasierten, funktionalen Tests wird dann zu jeder Anforderung mindestens ein Testfall abgeleitet. Mehr als ein Testfall ist i.d.R. jedoch empfehlenswert. Werden alle Testfälle fehlerfrei durchgeführt, gilt die Anforderung als erfolgreich erfüllt (vgl. ebd. S.73 f.).
- **Nicht-funktionale Tests:** Nicht-funktionale Tests beschreiben Attribute des Verhaltens eines Systems (vgl. Spillner & Linz, 2012, S.75). Es geht hierbei um Merkmale, die in der ISO-Norm 9126 als "Übertragbarkeit, Wartbarkeit, Effizienz, Funktionalität, Zuverlässigkeit und Benutzbarkeit/Gebrauchstauglichkeit" beschrieben werden (vgl. Johner, 2015). Performanztests, Tests der

Benutzerfreundlichkeit oder Tests der Datensicherheit sind nur drei Beispiele für nicht-funktionale Testarten (vgl. Spillner & Linz, 2012, S.75 f.).

2.1.6 Testarten

Im Folgenden werden verschiedene Arten, auf welche Tests durchgeführt werden können, kurz erläutert.

- **GUI-Tests:** Tests der grafischen Benutzeroberfläche (engl. Graphical User Interface) zeichnen sich dadurch aus, "dass sowohl der Point of Control als auch der Point of Observation die Benutzerschnittstelle des Softwaresystems ist" (Grechenig, 2010, S. 333). Die Abläufe innerhalb der betroffenen Softwarekomponenten können und sollen nicht unmittelbar nachvollzogen werden. Somit handelt es sich um ein sog. Blackbox-Verfahren.
- **Visuelle Tests:** Visuelle Tests beziehen sich ebenso wie GUI-Tests auf die Benutzeroberfläche der zu testenden Anwendung, fokussieren sich jedoch häufig darauf, den genauen Aufbau der Oberfläche zu testen (vgl. Applitools, 2021). Eine Möglichkeit für ihre Umsetzung ist der Vergleich des sichtbaren Bildes mit einer zuvor erstellten Grafik oder einem aufgenommenen Screenshot. Wenn während des Testlaufs Abweichungen des Ist- vom Soll-Zustand festgestellt werden, kann beispielsweise anhand einer zuvor festgelegten Toleranzgrenze entschieden werden, ob die Anforderungen zu genüge erfüllt sind, oder der Test als fehlgeschlagen gewertet wird.
- **API-Tests:** API ist die englische Abkürzung für Application Programming Interface und somit eine Schnittstelle von einer Software zu anderen, in welcher die Regeln zur Kommunikation definiert werden (vgl. IBM Cloud Education, 2020). API-Tests werden somit genutzt, um zu prüfen, ob die Schnittstelle zu einer Komponente oder einem (Teil-)System korrekt genutzt wird, bzw. genutzt werden kann. Unter anderem kann es notwendig sein, dass Nachrichten mit ungültigem Inhalt gesondert behandelt werden, oder auf verschiedene Anfragen stets die vorgesehenen Antworten versendet werden.

- **Datenbanktests:** Wird ein System entwickelt, welches an eine Datenbank angebunden ist, kann sichergestellt werden, dass nach dem Start des gesamten Systems oder der Datenhaltung stets der erwartete Initialzustand vorliegt. Auch nach Interaktionen mit der zu testenden Anwendung kann es von Interesse sein, zu überprüfen, wie sich die Datensätze in der verwendeten Datenhaltung verändert haben. Sowohl korrekte Antworten des Servers auf gesendete Anfragen als auch der Zustand der Datenbankwerte können Auskunft darüber geben, ob die Abläufe im Backend zu den gewünschten Ergebnissen führen. Sobald ein Test fehlschlägt, bei welchem die Benutzeroberfläche für das Versenden der Anfragen genutzt wurde, kann ggf. auf die erfolgreichen Datenbanktests verwiesen werden. Diese können darauf hinweisen, dass ein Fehler in der Implementation der Benutzeroberfläche vorliegt. Andersherum können leicht übersehbare Fehler gefunden werden, wenn die GUI-Tests erfolgreich verlaufen, die Datenbanktests jedoch fehlschlagen. In solchen Fällen bedarf es womöglich detaillierterer Analysen, welche prüfen, ob die Anforderungen der geschriebenen Tests korrekt formuliert sind, oder ggf. Fehler in der Kommunikation zwischen den Softwareschichten vorliegen.

- **End-to-End-Tests:** Die Bezeichnung der End-to-End-Tests (kurz E2E-Tests) bezieht sich darauf, dass sie die zu testende Anwendung von der Benutzeroberfläche (auch Frontend) bis hin zur Logik und Datenhaltung (zusammen auch Backend) und zurück testen (vgl. Testautomatisierung.org, o. D.). Auf diese Weise können Funktionen ganzheitlich und über das gesamte System hinweg getestet und eine Lauffähigkeit sichergestellt werden. Ein optimales Testszenario könnte wie folgt aussehen:

  1. Es wird mit der ersten/oberen Schicht, der Benutzeroberfläche bzw. **Präsentationsschicht**, interagiert.

  2. Dies führt dazu, dass eingegebene Werte oder ausgelöste Ereignisse an die zweite/mittlere Schicht, die Geschäftslogik bzw. **Logikschicht**, weitergeleitet werden. Hier kann es notwendig sein, Daten abzufragen oder zu verändern.

  3. Eine entsprechende Anfrage wird von der dritten/unteren Schicht, der **Persistenzschicht**, empfangen und ausgewertet. Hier werden die angefragten Daten bereitgestellt und in einer Antwort zurückgeliefert.

4. Die **Logikschicht** kann mit den empfangenen Werten ggf. Berechnungen durchführen und das Ergebnis auswerten.

5. Anschließend wird ein entsprechendes Resultat an die **Präsentationsschicht** zurückgesendet. Die Benutzeroberfläche kann aktualisiert werden und erhaltene Ergebnisse anzeigen.

Die folgende Grafik verbildlicht den Ablauf eines solchen E2E-Tests durch die drei Softwareschichten.

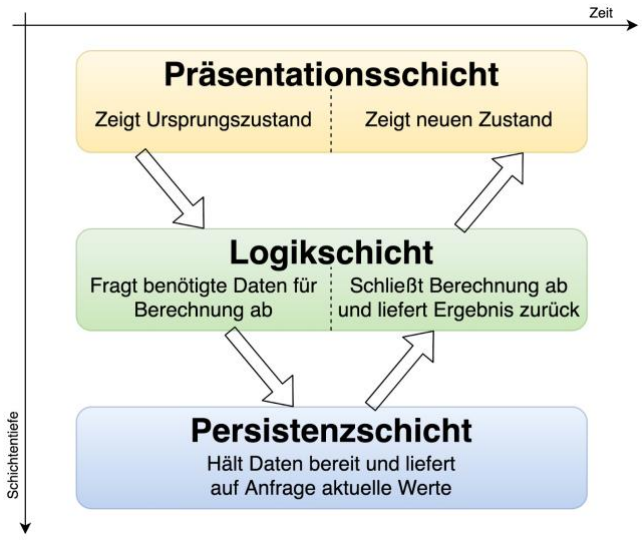

Abbildung 3: Grafik - Simplifizierte Verbildlichung des Ablaufes eines E2E-Tests anhand der drei in dieser Ausarbeitung im Fokus stehenden Softwareschichten.

2.1.7 Testautomatisierung

Die in dieser Ausarbeitung umgesetzte Testautomatisierung orientiert sich an den Grundsätzen des International Software Testing Qualifications Boards: “In software testing, test automation (which includes automated test execution) is one or more of the following tasks:

- Using purpose built software tools to control and set up test preconditions
- Executing tests
- Comparing actual outcomes to predicted outcomes” (ISTQB, 2016).

Im Gegensatz zu manuellen Tests haben automatisierte Tests den Vorteil, dass sie regelmäßig und beliebig oft ausgeführt werden können, um die Lauffähigkeit von Software sicherzustellen, während sie weniger Ressourcen binden als manuelle Tests gleicher Größenordnung es täten. „Test automation is expected to help run many test cases consistently and repeatedly on different versions of the SUT [getestetes System (engl. System under Test)]" (ebd.).

2.2 Cypress

Cypress ist ein Testautomatisierungsframework, mit Hilfe dessen Software getestet werden kann, welche in einem Browser ausgeführt wird (vgl. Cypress.io Overview, 2021). Bei den getesteten Anwendungen kann es sich somit sowohl um öffentliche als auch um unternehmensinterne Software handeln. Öffentliche Software könnten kommerzielle Webanwendungen sein, welche von einem Unternehmen entwickelt werden, um seinen Kunden den Zugriff auf Informationen und den Erwerb von Dienstleistungen oder Produkten zu gewähren. Für interne Software können Tools stehen, welche für die Nutzung innerhalb einer Firma entwickelt werden, um interne Prozesse zu digitalisieren. Bei diesen könnte es sich gleichermaßen um informationstechnische Aspekte wie Serverwartung, wie um personalwirtschaftliche Prozesse handeln.

2.2.1 Zielgruppe

Die Nutzer, auf die Cypress primär ausgerichtet ist, sind Softwareentwickler und Qualitätssicherungsingenieure von Webapplikationen. Durch eine zugängliche Syntax soll die Einstiegshürde niedrig gehalten werden, um das für eine Nutzung benötigte technische Hintergrundwissen sowie die Einarbeitungszeit zu minimieren. Eine Open-Source-Lizenz ohne Nutzungskosten macht es auch Privatpersonen zugänglich, und beschränkt den Nutzerkreis nicht auf das professionelle Umfeld oder den Unternehmenskontext (vgl. Cypress.io Overview, 2021).

2.2.2 Cypress für verschiedene Testebenen im Software-Lebenszyklus

Die Nutzung von Cypress ist nicht auf eine bestimmte Phase in der Entwicklung einer Webanwendung beschränkt. Beispielsweise kann es vor und während der Programmierung

eines ersten Prototypen eingesetzt werden, um die geplanten Funktionen und ihre korrekte Implementierung zu dokumentieren und zu kontrollieren. Doch auch die Nutzung in bestehenden, größeren Projekten ist möglich.

- **Cypress für Unit-Tests:** Insbesondere wenn es sich bei der zu testenden Komponente um eine Benutzeroberfläche handelt, können die GUI-Testmethoden von Cypress bereits früh im Entwicklungsprozess verwendet werden, um grundlegende Unit-Tests zu implementieren. Sollen hingegen einzelne Methoden und ihre Rückgabewerte getestet werden, ohne dass diese auf der Benutzeroberfläche sichtbar sind, werden fortgeschrittenere Methoden nötig.

- **Cypress für Integrationstests:** Werden mehrere Komponenten miteinander verbunden, oder wird beispielsweise ein externer Service angesprochen, kann die Kommunikation zwischen ebendiesen getestet werden. Werden hierfür HTTP-Anfragen verwendet und bestimmte Antworten erwartet, so liegen in Cypress Testmethoden vor, die speziell für diese Art von Integrationstests vorgesehen sind. Sie ermöglichen es beispielsweise, Nachrichten im json-Format anzufertigen und als HTTP-Anfrage an eine festzulegende Adresse zu schicken. Bei dem Empfänger kann es sich gleichermaßen um die Schnittstellen lokaler Dienste oder entfernter Systeme handeln. Anschließend kann die erhaltene Antwort ausgewertet werden.

- **Cypress für Systemtests:** Ebenso wie die erwähnten Integrationstests, können auch Systemtests mit Hilfe von Cypress implementiert werden. Beispielsweise können diese in der Continuous Integration Pipeline eines bestehenden Projektes integriert werden, um mit regelmäßig ausgeführten Regressionstests die Lauffähigkeit der Webanwendung zu überwachen, während neu implementierte Module und Funktionen in neuen Versionen hinzugefügt werden (vgl. Cypress.io Overview, 2021).

Den Kern der von Cypress angebotenen Funktionen bilden Methoden für sogenannte E2E-Tests. Da diese vor allem im Umfang von System- oder Integrationstests verwendet werden, ist zu erwarten, dass Cypress schwerpunktmäßig in diesen Testebenen Anwendung findet. In der Online-Präsenz von Cypress wird hauptsächlich mit besagten E2E-Tests geworben. Sie stellen somit das funktionelle Merkmal dar, welches primär in die Öffentlichkeit getragen wird,

um das Interesse potenzieller Anwender zu wecken und sich von anderen Frameworks abzuheben.

2.2.3 Nutzung von Cypress in versionierten Projekten

Eines der Einsatzgebiete für Tools zur Testautomatisierung wie Cypress stellt das automatisierte Testen von verschiedenen Versionen derselben Software dar. Da dies in schichtenübergreifenden Projekten eine gängige Praxis ist, soll dieses Thema im Folgenden erläutert werden. Die Umsetzung einer Pipeline oder einer automatisierten regelmäßigen Ausführung von Regressionstests soll jedoch nicht Teil dieser Ausarbeitung sein, da dies kein direkter Bestandteil der Forschungsfragen ist. Auch in der abschließenden Evaluation wird es somit nicht behandelt werden.

Handelt es sich bei der zu testenden Software um eine Anwendung, welche stets als ein ganzes Projekt - auch Monolith genannt - behandelt wird, gestaltet sich die Testautomatisierung verhältnismäßig simpel. In diesem Falle können die Cypress-Tests in einem eigenen Verzeichnis innerhalb des Projekts angelegt werden. Dieses kann anschließend inklusive der enthaltenen Testskripte in die Versionierung einbezogen werden. Wann immer Änderungen an der Software vorgenommen werden, können die Auswirkungen direkt beim Ausführen der Cypress-Tests betrachtet werden. Dies ist unabhängig davon, ob die Änderungen innerhalb desselben Branches vorgenommen werden, oder für die umzusetzenden Erweiterungen ein neuer Branch erstellt wird. Dadurch, dass in diesem Szenario das Cypress-Verzeichnis in der Versionierung inkludiert ist, kann stets ein aktueller Stand aufrechterhalten werden. Werden zwei Branches zusammengeführt, so werden auch die entsprechend bearbeiteten Tests der beiden Branches ineinander integriert, bzw. aktualisiert.

Alternativ zu der monolithischen Herangehensweise kann ein zu testendes Produkt auch aus mehreren separat entwickelten Modulen oder Services bestehen. Gegebenenfalls werden diese einzelnen Bestandteile des gesamten Endproduktes lediglich in ihren jeweiligen isolierten Umgebungen, also in Form von Unit-Tests (s. Kapitel 2.2.2) getestet. In diesen Fällen würde das Szenario aus dem vorhergegangenen Absatz greifen. Wird die Testabdeckung und Produktzuverlässigkeit jedoch hoch priorisiert, so können auch die Schnittstellen zur Kommunikation zwischen den einzelnen Diensten automatisiert getestet werden. In diesem

Fall könnte beispielsweise ein weiteres separates Projekt angelegt werden, welches speziell für die Testautomatisierung entwickelt wird. Dieses Testprojekt müsste fortlaufend an die Änderungen angepasst werden, die in den verschiedenen Modulen vorgenommen werden. Diese Art von Teststrategie würde aus diesem Gesichtspunkt nur erfolgreich und unter angemessenem Aufwand etabliert werden können, sofern die Arbeitsschritte der Einzelprojekte weit im Voraus geplant sind, oder ein Projekt getestet wird, dessen Entwicklung bereits weitestgehend abgeschlossen ist und nur eine geringe Zahl von zukünftigen Änderungen vorgesehen ist. Wird jedoch ein Mitarbeiter mit expliziter Verantwortlichkeit für die Testautomatisierung beauftragt, so kann dieser dafür sorgen, dass zu jedem Zeitpunkt eine hohe Testabdeckung sichergestellt wird. Diese könnte durch die Implementierung von Integrations- und Systemtests erreicht werden.

In den Best Practices zum Organisieren von Cypress wird generell empfohlen, die Cypress-Tests innerhalb des Verzeichnisses der jeweils zu testenden Komponente anzulegen (vgl. Cypress.io FAQ, 2021).

2.3 Cypress Methoden

Um die Konzeption eines Testansatzes optimal in Worte fassen zu können, kann es vorab von Vorteil sein, häufig genutzte Methoden des verwendeten Testframeworks zu erläutern. Hiermit soll ein ausreichendes Verständnis der verwendeten Vorgehensweisen vermittelt werden. Die meisten Methoden werden in Anwendungssituationen in Kapitel 6 demonstriert.

- **visit():** Um die Oberfläche einer Website testen zu können, muss ein Testautomatisierungsprogramm - ebenso wie ein Softwaretester es bei manuellen Tests täte - diese zunächst im Web-Browser aufrufen, oder auch "besuchen". In Cypress wird dies über die Methode *cy.visit()* (vgl. Mwaura, 2021, S.58) erreicht. Der Methode *cy.visit()* wird ein Argument übergeben, bei welchem es sich um die URL der Website handelt, die man besuchen möchte. Dieser Aufruf sieht wie folgt aus, möchte man die Startseite der HAW Hamburg besuchen:

```
cy.visit('https://www.haw-hamburg.de');
```

Die Ausführung dieser Methode beinhaltet einen impliziten Test. Kann die angefragte Website nicht aufgerufen werden, so wird dies von Cypress bemerkt, und der Test, welcher diesen Aufruf von *cy.visit()* beinhaltet, wird als fehlgeschlagen markiert. In diesem Falle könnte beispielsweise überprüft werden, ob ein Problem mit den Servern vorliegt, oder momentan eine Wartung durchgeführt wird, wodurch die Website nicht erreicht werden kann. Sieht der Anwender jedoch, dass der Test erfolgreich war, kann er davon ausgehen, dass während dieses ersten Schrittes, dem Aufrufen der zu testenden Website, keine Fehler vorliegen (vgl. Mwaura, 2021, S.59).

- **get():** In den meisten Tests zur Oberfläche einer Website ist nicht nur interessant, ob eine Website generell aufgerufen werden kann. Stattdessen ist vor allem von Interesse, ob die verschiedenen Elemente der Website auch sichtbar sind. Hierzu bietet Cypress die Methode *cy.get()* an. Dieser wird der Verweis auf ein Seitenelement in Form eines CSS-Selectors übergeben. Mit diesem wird auf der getesteten Website nach dem passenden Element gesucht. "When the Cypress *cy.get()* command does not find the [...] element, an error will be thrown; otherwise, Cypress will pass the test" (Mwaura, 2021, S.60). In dieser Situation erleichtert Cypress dem Anwender das Schreiben von Tests durch die sogenannten Default Assertions. "Default assertions in Cypress are built-in mechanisms that will cause a command to fail without the need for explicit assertions to be declared by the user. With these commands, Cypress takes care of the behavior of an exception if it's encountered while executing the said command" (ebd.). Der Anwender muss hier also keinen expliziten Test schreiben, um zu prüfen, ob ein Element gefunden werden konnte.

- **find():** Mit der Methode *find()* können Objekte innerhalb eines anderen lokalisiert werden (vgl. Mwaura, 2021, S.62 f.). Wird beispielsweise mit Hilfe von *get()* ein Menü lokalisiert, kann mit *find()* innerhalb dieses Menüs nach einem Textfeld oder Knopf gesucht werden.

- **contains():** Die Methode *contains()* sucht nach dem ersten Element, welches ein gegebenes Objekt beinhaltet. In dieser Ausarbeitung wird sie meistens für das Lokalisieren von Texten und Zahlen genutzt. Doch auch weitere Elemente oder reguläre Ausdrücke können als Parameter übergeben werden.

- **type():** An dieser Stelle ist es nun an der Reihe, mit dem Element, das vorab durch die Methode *get(), find()* oder *contains()* gefunden wurde, zu interagieren, um dessen Funktionalität zu überprüfen. Hierfür kann in Cypress u.a. die Methode *type()* genutzt werden (vgl. Mwaura, 2021, S.61). Sie ermöglicht es, dem Element eine Eingabe hinzuzufügen. Bei dieser kann es sich entweder um einen Text oder Zahlen handeln. Dafür kann der Tester in die Klammern der Methode den gewünschten Wert einfügen, der an das Element übermittelt werden soll. Wie bereits in Methode *cy.get()* sei auch an dieser Stelle erwähnt, dass der Tester keine weiteren manuellen Eingaben zur Überprüfung in Cypress eingeben muss. Der Test durchläuft die Methode *type()* mit Default Assertions, sodass er fehlschlägt, sollte die Eingabe blockiert sein.

- **should():** Entweder direkt beim ersten Besuch einer Website, oder nachdem Änderungen an ihr vorgenommen wurden, sollte stets geprüft werden, ob der Status der Anwendung oder ihrer Benutzeroberfläche korrekt dargestellt wird. Hierfür kann auf die Methode *should()* zurückgegriffen werden (vgl. Mwaura, 2021, S.62). Bei der Methode *should()* fügt der Tester in Klammern eine Annahme (engl. Assertion) ein, die durch den Test überprüft werden soll. So kann beispielsweise nach erfolgter Eingabe über die Methode *type()* bei einem Oberflächenobjekt geprüft werden, ob der übermittelte Eingabewert korrekt übernommen wird:

```
    .should('have.text', 'Mein Rezept');
```

  Ebenso kann bei einer Liste die Anzahl der beinhalteten Objekte überprüft werden:

```
    .should('have.length', '5');
```

  Stimmt der vorgefundene Wert mit der Annahme überein, ist der Test erfolgreich.

- **click():** Auch die Methode *click()* beinhaltet eine automatische Prüfung und stellt nicht ausschließlich eine bloße Aktion dar. Lässt sich die gewünschte Schaltfläche, meist ein Knopf, nicht betätigen, so schlägt der Testschritt fehl.

2.4 Angular

Angular ist ein Entwicklungsframework für Frontend-Anwendungen, das auf TypeScript und HTML basiert (vgl. Google, 2021). Der TypeScript-Anteil ist hierbei vollständig gegen JavaScript austauschbar, sollte dies gewünscht sein. Das Angular-Framework bietet standardmäßig eine übersichtliche Gliederung der Komponenten und des enthaltenen Quellcodes.

2.5 Node.js

Node.js ist eine Laufzeitumgebung, die es erlaubt, JavaScript-Code außerhalb eines Webbrowsers auszuführen. Node.js ist plattformunabhängig, was bedeutet, dass es auf den meisten Kombinationen von Soft- und Hardware lauffähig ist. Außerdem wird es unter einer Open-Source-Lizenz angeboten, welche es frei zugänglich für potenzielle Nutzer macht (vgl. OpenJS Foundation, 2021).

Node.js ist nicht nur eine Laufzeitumgebung, sondern bietet außerdem den Node Package Manager NPM. Mit diesem können verschiedene Entwicklungs- und Test-Frameworks und -Tools - sogenannte Node Packages - kostenfrei heruntergeladen und installiert werden. Auch das Aktualisieren und Warten von verschiedenen Versionen der genutzten Node Packages geschieht durch den Node Package Manager mit Hilfe weniger Kommandos. Auf diese Weise werden sowohl Cypress und Angular als auch einige weitere unterstützende Frameworks aus dem Angebot von NPM für diese Arbeit verwendet und koordiniert.

3 Anforderungsanalyse

Um die in Kapitel 1.2 definierten Forschungsfragen beantworten zu können, soll ein Testkonzept für die Testautomatisierung einer Webanwendung konzipiert und anschließend umgesetzt werden. Im Folgenden soll sowohl zusammengefasst werden, welche Anforderungen an das zu entwickelnde Softwareprojekt gestellt werden, und unter welchen Rahmenbedingungen es implementiert werden soll, als auch auf welcher Grundlage und mit welchem Anspruch die Bewertung des Testframeworks Cypress vorgenommen wird.

3.1 Einordnung genutzter Begrifflichkeiten

Wie in der Einleitung und Motivation in Kapitel **Error! Reference source not found.** beschrieben, beinhaltet die Zielsetzung dieser Arbeit sowohl die Absicht, die schichtenübergreifende Automatisierung von Tests unter Cypress zu bewerten, als auch eine Empfehlung für oder gegen seinen teamübergreifenden Einsatz auszusprechen. Obgleich diese Begriffe, sich im Softwareentwicklungskontext gelegentlich gegenseitig implizieren können, sind sie keineswegs gleichbedeutend. Auch der Begriff "komponentenübergreifend" ist eng mit diesen verknüpft. Aufgrund ihrer vermeintlichen Synonymität sollen diese Begriffe im Folgenden klarer voneinander abgegrenzt werden:

- **team- & komponentenübergreifend:** Je nach Unternehmensstruktur und spezifischem Projekt kann die Aufteilung der Entwickler und die Abgrenzung ihrer Verantwortlichkeiten unterschiedlich ausfallen. Grundsätzlich wird jedoch davon ausgegangen, dass verschiedene Teams zwar regelmäßig an demselben Projekt arbeiten, hingegen seltener gemeinsam an derselben Softwarekomponente. Es ist also häufig genau ein Team für genau eine Komponente verantwortlich. Aus dieser Zuweisung ergibt sich eine im Kontext dieser Ausarbeitung gültige Synonymität der Begriffe "teamübergreifend" und "komponentenübergreifend".
- **komponenten- & schichtenübergreifend:** Komponenten und Schichten sind zwei unterschiedliche Einheiten in der Softwareentwicklung. Eine Komponente ist die "kleinste Softwareeinheit, für die eine separate Spezifikation verfügbar ist, oder die

separat getestet werden kann" (Spillner & Linz, 2012, S. 253). Jede Softwareschicht kann eine oder mehrere Softwarekomponenten enthalten. Nur in Szenarien, in welchen eine Schicht lediglich aus einer Komponente besteht, könnten "komponentenübergreifend" und "schichtenübergreifend" synonym verwendet werden.

- **team- & schichtenübergreifend:** Innerhalb eines Unternehmens kann die Einteilung der Mitarbeiter so vorgenommen werden, dass jedes Team für eine oder mehrere Komponenten zuständig ist. Diese Komponenten können so ausgewählt werden, dass demselben Team ausschließlich Komponenten zugeteilt werden, welche sich innerhalb einer Softwareschicht befinden. Somit kann in einigen Projekten der Ausdruck "teamübergreifend" gleichbedeutend mit "schichtenübergreifend" sein. Wird jedoch eine andersartige Verteilung gewählt, so kann dasselbe Team für mehrere Schichten verantwortlich sein. Die Begriffe "teamübergreifend" und "schichtenübergreifend" sollten also nicht grundsätzlich synonym verwendet werden.

Sollen die beschriebenen Prinzipien und Arbeitsergebnisse auf konkrete Projekte angewendet werden, kann es hilfreich sein, die Differenzierung der aufgezählten Begrifflichkeiten maßgeschneidert neu zu spezifizieren.

3.2 Bewertungsgrundlage und Anforderungen an Cypress

Die Leitfrage dieser Ausarbeitung ist, ob Cypress für den teamübergreifenden Einsatz für die Testautomatisierung in schichtenübergreifenden Softwareprojekten empfohlen werden kann. Bei den entworfenen und begutachteten Tests sollen hauptsächlich Integrations- und Systemtests berücksichtigt werden, da diese im Fokus von Cypress Anwendungsbereichen stehen. Zu den Anforderungen gehören ausdrücklich keine Akzeptanztests zur Ausführung durch einen Nutzer oder Auftraggeber bzw. Kunden, da Cypress für diese nicht vorgesehen ist.

Für diese spezifische Fragestellung liegt kein in der Wissenschaft gängiges oder allgemein anwendbares Maß zur Bewertung vor. Um dennoch eine begründete Aussage treffen zu können, werden in den folgenden Abschnitten 3.2.1 bis 3.2.4 einige Anforderungen an die

Testautomatisierung formuliert, welche die Bewertungsgrundlage bilden werden. Nach der Realisierung der Tests wird betrachtet, ob die konzipierten Testszenarien mit einem angemessenen Aufwand realisiert, und die definierten Anforderungen erfüllt werden konnten. Werden diese Anforderungen von Cypress nicht oder nur teilweise erfüllt, wirkt sich dies negativ auf die Bewertung aus.

Eine Bewertung, zu welchem Grad Cypress teamübergreifend für die schichtenübergreifende Nutzung in einem, dem vorliegenden Demonstrationsobjekt ähnlichen, Projekt zu empfehlen ist, wird in Kapitel 0 zu finden sein. Eine derartige Evaluation kann nie vollständig sein und alle Aspekte berücksichtigen, welche bei einem realen Anwendungsfall auftreten könnten. Die vorliegende Ausarbeitung kann naturgemäß lediglich einen Überblick über jene Szenarien geben, welche während der durchgeführten Testautomatisierung betrachtet werden konnten. Einige Aspekte, welche den Rahmen dieser Arbeit gesprengt hätten, werden in Kapitel 8.3 in Aussicht gestellt.

3.2.1 Nutzerfreundlichkeit von Cypress

Zu der Evaluation, inwieweit Cypress für den teamübergreifenden Einsatz empfohlen werden kann, wird in nennenswertem Maße seine Nutzerfreundlichkeit beitragen. Nachfolgend werden einige Aspekte definiert, welche im Umfang dieser Ausarbeitung zu einer positiven Nutzerfreundlichkeit beitragen würden. Diese sind in Teilen an die Prinzipien der DIN ISO 9241-11 angelehnt (vgl. Pelz, o.D.).

Schnelle Einsatzbereitschaft (Effizienz)

Ein Tool sollte entweder umgehend nach dem Herunterladen, oder nach einer kurzen, simplen Installation einsatzbereit sein. Komplexe Abhängigkeiten zu externen Libraries oder anderen Tools, welche zusätzlich manuell eingerichtet werden müssen, stellen eine Einstiegshürde dar. Die Ressourcen, welche durch die Einrichtung eines Tools gebunden werden, bevor es sinnbringend eingesetzt werden kann, bedeuten Umstände für das betroffene Team, und Verluste für das Unternehmen.

Übersichtlichkeit in der Nutzung (Intuitivität & Erwartungskonformität)

Die Strukturierung einer Testdatei sollte den Bedürfnissen des Testers ebenso angepasst werden können, wie den Anforderungen der zu testenden Anwendung. Würde ein Testframework beispielsweise erzwingen, dass alle Testschritte eines Projektes innerhalb derselben Datei verfasst werden müssen, so würde dies der Übersichtlichkeit des Testcodes schaden. Gibt es hingegen die Möglichkeit, mehrere Verzeichnisse für die Tests verschiedener Seiten oder dedizierte Testdateien für verschiedene Anwendungsfälle anzulegen, und nach Bedarf anzuordnen, so wirkt sich dies positiv auf die Übersichtlichkeit aus.

Nachvollziehbarkeit der Testläufe (Nutzerfeedback)

Wurden einige Tests verfasst und stehen die zu testenden Komponenten bereit, so sollen für gewöhnlich umgehend Testläufe durchgeführt werden, um ihre Funktionalität sicherzustellen. Wie in der Softwareentwicklung an sich, so ist es auch in der Testentwicklung ratsam, produzierten Code regelmäßig auszuführen. Auf diese Weise kann kontrolliert werden, ob sich die bis zum aktuellen Stand entwickelten Testschritte wie vorgesehen verhalten.

Erhält der Tester bei der Ausführung von Tests lediglich die Rückmeldung, dass in einer Sammlung von Tests ein unbestimmter Schritt fehlschlägt, ohne weitere Hinweise auf den Fehlerursprung, so kann es sich als zeitaufwändig herausstellen, den Grund für den Fehlschlag zu finden. Würde hingegen die genaue Codestelle oder ein Bild der Benutzeroberfläche zum Fehlerzeitpunkt bereitgestellt, so könnte dies die Korrektur eines fehlerhaften Tests bzw. des aufgedeckten Programmierfehlers beschleunigen.

Wiederverwendbarkeit von Code

Steigt die Menge zu schreibender Tests für ein Projekt an, so kann es vorkommen, dass gewisse Situationen sich wiederholen und ähnliche oder dieselben Testschritte an mehreren Stellen ausgeführt werden müssen. In solchen Fällen verbessert es die Leserlichkeit, wenn duplizierter Code und sich wiederholende Zeilen vermieden werden können. Bietet ein Testframework die Möglichkeit, mehrere Testschritte unter einer Methode zusammenzufassen, diese zentral zu hinterlegen, und global abrufbar zu machen, würde dies einen relevanten Mehrwert - vor allem für umfangreichere Projekte - darstellen (vgl. Clarke, 2021).

3.2.2 Erweiterbarkeit der automatisierten Tests

Ein weiterer Aspekt, welcher für die Evaluation eines Testautomatisierungsframeworks als relevant definiert wird, ist die Erweiterbarkeit der geschriebenen automatisierten Tests. Diese wird beschrieben durch die Möglichkeit, die vorhandenen Tests anzupassen, auszutauschen oder auszubauen. Dies kann vor allem sinnvoll sein, nachdem ein neues Feature zu der zu testenden Anwendung hinzugefügt, oder ein bestehendes verändert wurde. Eine solche Erweiterung des Funktionsumfangs kann je nach Größe der implementierten Änderung dazu führen, dass ein größerer oder kleinerer Teil der bestehenden Tests angepasst oder gänzlich ausgetauscht werden muss.

Der für Testanpassungen benötigte Aufwand hängt zumeist von der Größe der Funktionsänderung selbst ab. Jedoch kann bei gleichem Änderungsumfang die Nutzung verschiedener Testframeworks zu variierendem Arbeitsaufwand für ähnliche Anpassungen der automatisierten Tests führen. Bestehen bei einem Framework beispielsweise zahlreiche Abhängigkeiten zwischen verschiedenen Tests, da es für das Testen von monolithisch entwickelten Anwendungen entworfen wurde, oder eine aufwendige Syntax aufweist, könnte dies zu einem größeren Aufwand führen, wenn neue Tests eingefügt oder bestehende angepasst werden. Bietet ein anderes Framework hingegen eine schnell verständliche Syntax und klar voneinander getrennte Strukturen zur Ausführung verschiedener Tests, kann dies das Anpassen bestehender Tests ebenso erleichtern, wie das Hinzufügen neuer Tests. Der Grad an Erweiterbarkeit, welcher während der Testautomatisierung für diese Arbeit festgestellt werden kann, stellt somit ein weiteres Kriterium für die Empfehlbarkeit von Cypress dar.

3.2.3 Überdeckung der Testebenen

Zusätzlich ist für die Bewertung von Cypress im vorliegenden Kontext von Interesse, inwiefern alle der drei Testebenen Unit-Test, Integrationstest und Systemtest umgesetzt werden können. Schließlich soll die Bewertung des Frameworks Aufschluss darüber geben, ob es kooperativ in verschiedenen Teams genutzt werden kann. Diese Teams können zu unterschiedlichen Zeitpunkten in der Entwicklung aktiv werden und Tests verfassen. Auch wenn mehrere Komponenten der Anwendung fertiggestellt sind, müssen diese nicht nur unabhängig nebeneinander, sondern auch im gemeinsamen Verbund getestet werden. Aus diesem Grund

ist eine Einsetzbarkeit über Testebenen hinweg von Interesse bei der hier zu treffenden Empfehlung.

3.2.4 Durchstich der Softwareschichten

Wie aus dem Titel dieser Arbeit ersichtlich ist, soll das entwickelte Testkonzept schichtenübergreifend ausfallen. Daher wird das gewählte Testautomatisierungsframework unter anderem danach bewertet, welche Möglichkeiten es bietet, um die verschiedenen Schichten einer gegebenen Webanwendung zu testen. So würde ggf. ein zufriedenstellendes Maß an GUI-Testmethoden nicht für eine positive Bewertung ausreichen, solange womöglich keine Funktionen geboten werden, um Werte aus der Datenhaltung abzufragen, da die Schichten nicht gleichermaßen ausgiebig getestet werden können.

Findet die Entwicklung einer Anwendung mit mehrschichtiger Architektur statt, so sollten die einzelnen Komponenten nicht nur isoliert in Komponententests getestet werden. Darüber hinaus ist das Testen der Interaktionen zwischen mehreren Komponenten in Integrationstests, ebenso wie das Testen des Gesamtsystems in Systemtests von hoher Wichtigkeit. Die Kommunikation zwischen Komponenten wird sich in vielen Fällen über mehrere Softwareschichten hinweg erstrecken. Daher ist es von Vorteil, wenn ein Testautomatisierungsframework zur Verfügung steht, welches schichtenübergreifend einsetzbar ist. Sind mehrere Entwickler aus verschiedenen Teams an der Implementierung von Integrations- oder Systemtests beteiligt, so kann der Austausch zwischen ebendiesen durch die Nutzung eines einheitlichen Tools erleichtert werden. Ist hingegen ein einzelner Entwickler oder ein einzelnes Team für die Testautomatisierung verantwortlich, so profitiert auch dieser/dieses von einem Testautomatisierungsframework, welches für das Testen aller Softwareschichten eingesetzt werden kann. Somit kann verhindert werden, dass mehrere Frameworks parallel eingesetzt werden müssen, um alle Testszenarien abzudecken. Diese Anforderung kann mit sogenannten E2E-Tests umgesetzt werden, welche im weiteren Verlauf dieser Ausarbeitung einen wichtigen Teil der behandelten Szenarien darstellen. Ein Werkzeug zur Testautomatisierung, welches sein Hauptaugenmerk auf die Implementierung besagter E2E-Tests legt, heißt Cypress. Aus diesem Grund wird Cypress für die Nutzung in dieser Ausarbeitung ausgewählt.

Schichtenübergreifende Tests müssen nicht grundsätzlich alle drei Softwareschichten einbeziehen. Es könnten beispielsweise Tests betrachtet werden, welche ihren Point of Control in der Geschäftslogik haben und ihren Point of Observation (s. Kapitel 2.1.3) in der Präsentationsschicht. Auch ein Test mit dem PoC in der Persistenzschicht und dem PoO in der Logikschicht wäre schichtenübergreifend und durchaus denkbar. Die in dieser Ausarbeitung betrachteten schichtenübergreifenden Tests sollen sich größtenteils über alle drei Schichten erstrecken. In diesem Fall befinden sich sowohl der PoC als auch der PoO stets in der Präsentationsschicht.

3.3 Mindestanforderungen an das Demonstrationsobjekt

Um die anzuwendenden Testprinzipien optimal demonstrieren zu können, wird für diese Ausarbeitung eine eigene, überschaubare Webanwendung entwickelt, welche nach dem Abschluss dieser Arbeit ggf. im Universitätskontext weiterverwendet werden könnte. An dieser Stelle wird beschrieben, über welche Funktionen das sogenannte Demonstrationsobjekt unbedingt verfügen soll, um eine Auswahl interessanter Szenarien abzubilden und eine angemessene Komplexität bei dem Entwerfen der Testautomatisierung zu ermöglichen. Das Ziel wird hierbei sein, einen gewissen Umfang von verschiedenen testbaren Aspekten zu generieren, während eine Generik beibehalten wird, die eine Übertragbarkeit auf verschiedene Anwendungen erlaubt. Aus diesem Grund wird eine Menge an Funktionen definiert, die einige grundlegende Operationen umfasst, welche dem Benutzer von Web- und Smart-Home-Anwendungen geboten werden können. Aus diesen ergeben sich die Mindestanforderungen für das Demonstrationsobjekt.

3.3.1 CRUD-Operationen

Die Grundlage für viele der geforderten Funktionen, bilden die vier elementaren Operationen von persistenter Datenhaltung (vgl. Sumo Logic, 2020): Anlegen/Create, Lesen/Read, Aktualisieren/Update und Löschen/Delete. Zusammengefasst werden diese Begriffe unter dem Akronym “CRUD” (vgl. ebd.). Mit Hilfe der sogenannten CRUD-Operationen können alle Veränderungen an der Datenhaltung einer Webanwendung vorgenommen werden, die bei ihrer Nutzung zu erwarten sind.

3.3.2 Zustandsabhängige Ausführung bestimmter Operationen

Als eine weitere Anforderung an das Demonstrationsobjekt wird formuliert, dass bestimmte Operationen ausschließlich möglich sein sollen, sofern vordefinierte Bedingungen erfüllt sind. Denkbare Szenarien hierfür sind, dass einer Aktion eine maximale Anzahl an täglich erlaubten Ausführungen zugewiesen wird. Alternativ könnte ihre Ausführung dem Vorhandensein von benötigten Ressourcen oder dem Besitz gewisser Nutzerrechte unterliegen. Die Prüfung auf die Erfüllung von bestehenden Bedingungen und die damit einhergehende Ausführbarkeit bestimmter Operationen stellt somit einen grundlegenden Bestandteil einer breiten Menge an möglichen Anwendungen dar und wird daher als Anforderung aufgenommen.

3.3.3 Fehlermeldungen

Ein wichtiger Aspekt für die Benutzerfreundlichkeit und Verständlichkeit von Softwareanwendungen ist Transparenz bei der Handhabung von Fehlerfällen. Ein entscheidender Bestandteil hiervon ist eine klare Kommunikation der Umstände gegenüber dem Benutzer. Werden bei dem Versuch, eine bestimmte Aktion ausführen zu lassen, eine oder mehrere der benötigten Bedingungen nicht erfüllt, so sollte beispielsweise eine klar formulierte Fehlermeldung für den Benutzer sichtbar gemacht werden. Diese kann dem Benutzer erklären, weshalb seine gewünschte Operation nicht erfolgreich ausgeführt werden kann. Aufgrund der Wichtigkeit ebendieser Anzeigen, wird auch dieser Aspekt in die Anforderungen dieser Arbeit aufgenommen.

3.3.4 API

Da es sich nicht nur bei Smart-Home-Applikationen, sondern auch bei modernen Webanwendungen im Allgemeinen häufig um verteilte Systeme handelt, wird bei ebendiesen meist eine API für die Kommunikation verwendet. Daher sollte ein Demonstrationsobjekt gewählt werden, welches die Implementierung einer simplen API motiviert oder die Nutzung einer vorgefertigten erlaubt.

3.3.5 Datenhaltung

In verteilten Systemen ist häufig mindestens eine Komponente anzutreffen, welche die Anbindung an eine Datenbank voraussetzt. Aus diesem Grund sollte auch das Demonstrationsobjekt über eine solche verfügen. Die Anbindung an eine Datenbank kann abhängig von der Art der zu entwickelnden Webanwendung und der gewählten Datenbankimplementierung variieren. Um die grundlegenden Anforderungen an eine Persistenzschicht testen zu können, wird für den Umfang dieser Ausarbeitung festgelegt, dass eine simple Datenhaltung mit überschaubarer API ausreichend sein wird. Die eigenständige Entwicklung und Anbindung einer Datenbank, bzw. die Auswahl des zu nutzenden Datenbank-Tools soll keinen entscheidenden Bestandteil dieser Ausarbeitung darstellen.

3.3.6 Komplexität

Des Weiteren ist eine gewisse Komplexität des Demonstrationsobjektes notwendig, um aussagekräftige Tests entwerfen zu können. Ein negatives Gegenbeispiel wäre ein Demonstrationsobjekt, welches ausschließlich die geforderten Funktionen anbietet, ohne in diesem Zuge ein gewisses Maß an komponenten- und schichtenübergreifenden Abhängigkeiten zu generieren. Ein solches Projekt würde kein realistisches Szenario darstellen, da davon ausgegangen wird, dass bei Webanwendungen, für welche eine Testautomatisierung in Betracht gezogen wird, stets eine gewisse Komplexität vorliegt. Somit würde bei Nutzung einer zu simplen Applikation das Ziel, einen übertragbaren Anwendungsfall zu generieren, verfehlt werden.

3.3.7 Erweiterbarkeit des Funktionsumfangs

Wie in Abschnitt 3.2.2 beschrieben, soll das Testframework auf die Möglichkeit hin analysiert werden, nach Änderungen der zu testenden Webanwendung, den Testumfang entsprechend anpassen zu können. Die Prüfung und Bewertung dieses Aspektes setzt ein Demonstrationsobjekt voraus, welches das modulare Hinzufügen neuer Funktionen ermöglicht, nachdem die Hauptfunktionalitäten implementiert und bereits einige Tests automatisiert wurden.

3.4 Eignungsprüfung und Wahl des Demonstrationsobjekts

Als Demonstrationsobjekt soll eine Smart-Home-Kaffeemaschine entwickelt werden. Um die zuvor definierten Anforderungen in Zusammenhang mit dem Demonstrationsobjekt zu bringen, werden nachfolgend einige Nutzungsszenarien geschildert, in welchen die Implementierungen der jeweiligen Anforderungen Verwendung finden. Ziel dieses Abschnitts ist es, mit Hilfe dieser Nutzungsszenarien und basierend auf den Anforderungen aus Kapitel 3.3 die Wahl der Smart-Home-Kaffeemaschine auf ihre Eignung zu prüfen.

3.4.1 Szenarien für CRUD-Operationen

Die grundlegenden Funktionen einer Smart-Home-Kaffeemaschine werden für die Zwecke dieser Arbeit wie folgt definiert und bieten die Möglichkeit, alle zuvor in Abschnitt 3.3.1 erarbeiteten CRUD-Operationen sinnvoll zu realisieren:

- Neues Rezept hinzufügen → Create
- Bestehende Rezepte und Zutatenfüllstände abfragen → Read
- Zutaten nachfüllen oder entnehmen → Update
- Kaffee nach Rezeptvorgaben zubereiten → Update
- Angelegte Rezepte löschen → Delete

3.4.2 Szenarien für Zustandsabhängige Ausführung bestimmter Operationen

Wie in Kapitel 3.3.2 beschrieben, sollen Abhängigkeiten verschiedener Operationen demonstriert werden. Auch einige der zuvor aufgelisteten Funktionen können nur unter bestimmten Umständen durchgeführt werden. Anhand dieser sollen im Folgenden einige Szenarien der zustandsabhängigen Ausführbarkeit erläutert werden. Zugehörige Wireframes dienen der Veranschaulichung der Benutzeroberfläche:

Neues Rezept hinzufügen

Die Eingabemaske zum Hinzufügen eines neuen Rezeptes kontrolliert die ausgefüllten Felder, bevor der Vorgang abgeschlossen werden kann. Ein Rezeptname sowie die Menge für mindestens eine Zutat müssen mit gültigen Eingabewerten eingetragen worden sein. Ist der Rezeptname zu kurz oder zu lang, wird dieses Feld als ungültig markiert. Solange nicht beide Bedingungen erfüllt sind, wird eine Warnung angezeigt, die den Benutzer über die fehlenden oder fehlerhaften Eingaben in Kenntnis setzt.

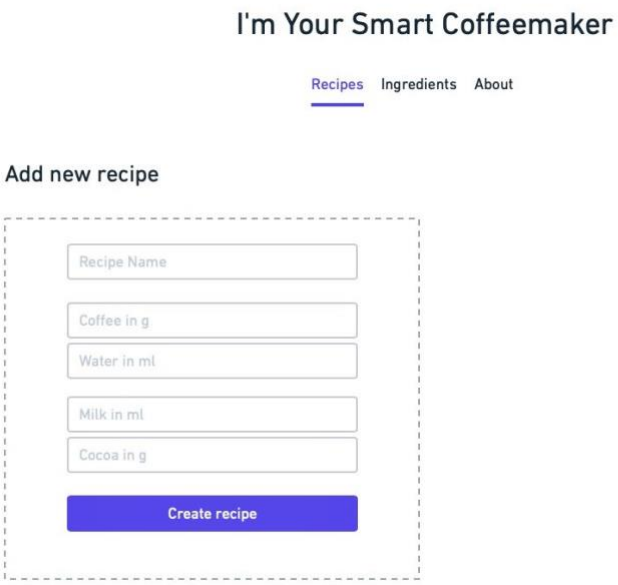

Abbildung 4: Wireframe - Neues Rezept hinzufügen.

Zutaten nachfüllen oder entnehmen

Eine Zutat kann nur aufgefüllt werden, wenn die hinzuzufügende Menge zuzüglich zu dem aktuellen Füllstand der jeweiligen Zutat nicht den in der Entwicklung fest konfigurierten Maximalfüllstand überschreitet. Soll einer Zutat eine Menge entnommen werden, so darf der Restfüllstand nicht negativ ausfallen. Wird die jeweilige Bedingung nicht erfüllt, so wird die Eingabe der zu addierenden bzw. subtrahierenden Zutatenmenge nicht akzeptiert. Auch hier informiert eine entsprechende Fehlermeldung den Nutzer über die nicht erfüllte Voraussetzung.

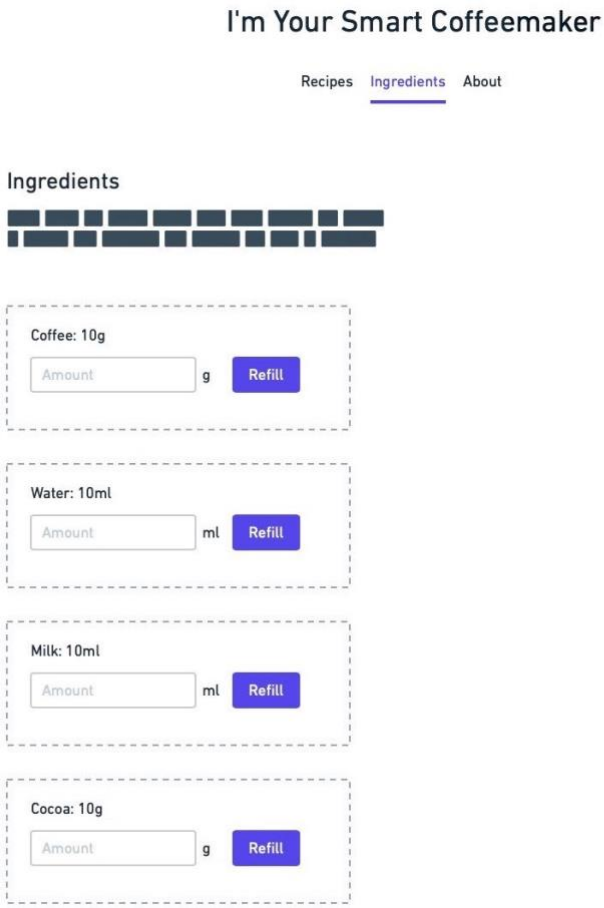

Abbildung 5: Wireframe - Zutaten nachfüllen oder entnehmen.

Bestehende Rezepte abfragen & Kaffee nach Rezeptvorgaben zubereiten

Die vorhandenen Rezepte werden in einem Gitter angezeigt. Wird ein Rezept zur Zubereitung ausgewählt, so wird überprüft, ob die derzeitigen Zutatenfüllstände hoch genug sind, um die zu verwendenden Mengen zu entnehmen, bevor die Durchführung des Kochauftrages gestartet werden kann. Ist mindestens eine der benötigten Zutaten nicht in ausreichendem Umfang vorhanden, so wird eine Benachrichtigung im Browserfenster ausgelöst. Diese teilt dem Benutzer mit, welche Zutat nachgefüllt werden muss, um eine erfolgreiche Durchführung zu ermöglichen.

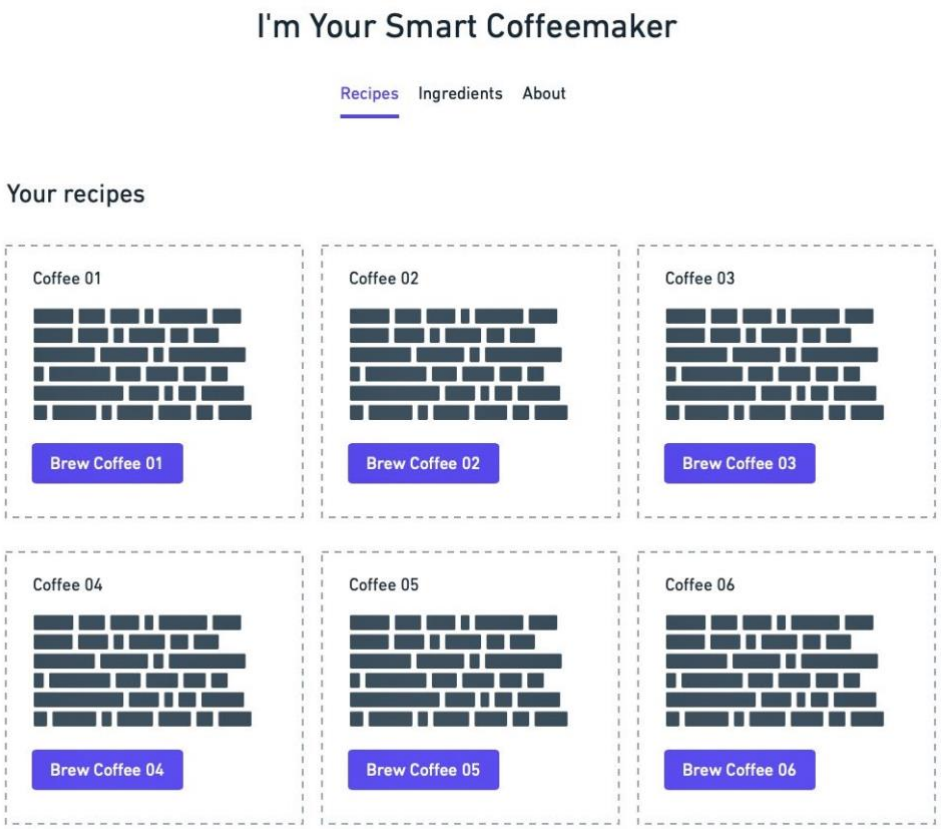

Abbildung 6: Wireframe - Rezepte anzeigen und zubereiten.

3.4.3 Szenarien für Fehlermeldungen

Wie in Kapitel 3.4.3 erwähnt, bietet die smarte Kaffeemaschine mehrere Szenarien, in welchen dem Benutzer Fehlermeldungen und Hinweistexte sinnvoll angezeigt werden können. Wenn eine Eingabemaske zum Anlegen eines neuen Rezeptes nicht korrekt ausgefüllt wird, können Hinweisfelder neben den jeweiligen Text- oder Zahlenfeldern auf ihre fehlerhaften bzw. fehlenden Eingabewerte hinweisen. Soll wie im vorigen Abschnitt ein Kaffee gekocht werden, und sind nicht ausreichend Zutaten vorhanden, so kann eine Fehlermeldung erscheinen, die den Nutzer darüber informiert, an welcher Zutat es mangelt.

3.4.4 Szenarien für eine API

Die Informationen über die aktuellen Füllstände der verschiedenen Zutaten sollen mit Hilfe von REST-Anfragen aus der Datenbank abgefragt werden können. Fehlerhafte Anfrageninhalte können mit entsprechenden Fehlercodes beantwortet werden. Die Antworten auf korrekt formatierte Anfragen können neben den verlangten Informationen außerdem mit Erfolgscodes versehen werden. Somit werden die grundlegenden Funktionen einer API abgedeckt und erlauben eine Testautomatisierung dieses Aspekts von verteilten Systemen (s. Kapitel 4.5.3).

3.4.5 Szenarien für eine Datenhaltung

Die Datenbank soll neben den aktuellen Füllständen der möglichen Zutaten ebenfalls eine Liste der gespeicherten Rezepte beinhalten. Diese Liste kann erweitert werden, wenn der Nutzer ein neues Rezept anlegt. Wird ein bestehendes Rezept gelöscht, so wird es entsprechend aus der Datenbank entfernt werden. Grundlegende Datenbankoperationen sind somit abgedeckt.

3.4.6 Fazit

In Kapitel 3.3 werden die Anforderungen an das Demonstrationsobjekt erläutert. Diese Anforderungen erfüllt die gewählte Smart-Home-Kaffeemaschine. Die Funktionen vom Nachfüllen der Zutaten bis zum Kochen eines Kaffees sind ausreichend, um eine nicht triviale Nutzung der Webanwendung zu erlauben. Auch das Testen einiger Nutzungsszenarien wird zu diesem Zeitpunkt bereits ermöglicht. Zusätzlich bietet der Kontext von Smart-Home-Lösungen die Möglichkeit, optionale Features hinzuzufügen, oder in den Ausblick zu stellen. Bei diesen könnte es sich beispielsweise um das Hinzufügen einer neuen Zutat, das Erstellen und Löschen eigener Rezepte, oder das Markieren einzelner favorisierter Rezepte handeln.

Diese günstigen Umstände haben zur Folge, dass eine realitätsnahe Situation aus der Softwareentwicklung simuliert werden kann. In dieser soll ein System, welches bereits über einige automatisierte Tests verfügt, um eine neue Funktion erweitert werden, wodurch eine Erweiterung und ggf. Anpassung der bestehenden Tests erforderlich wird. Diese Eignung trägt maßgeblich zur Wahl des Demonstrationsobjekts bei.

3.5 Mindestanforderungen an die Benutzeroberfläche

Die Präsentationsschicht des Demonstrationsobjekts soll für die Nutzung im Webbrowser eines Desktop- oder Notebook-PCs optimiert sein. Ein responsives Verhalten ist für die Zwecke des vorliegenden Forschungsinteresses nicht notwendig. Die Eignung der zu entwickelnden Webanwendung und Tests für die Ausführung auf Smartphones und anderen Mobilgeräten oder in einem Browserfenster mit verkleinerter oder vergrößerter Fenstergröße wird nicht zugesichert.

Eine Anbindung an oder Einbettung in reale Hardware ist ebenfalls nicht vorgesehen. Somit entfällt auch die Verknüpfung mit der Sensorik einer Kaffeemaschine oder ihren haptischen Bedienelementen.

Bei der Webentwicklung gibt es gelegentlich Unterschiede zwischen verschiedenen Browsern oder verschiedenen Versionen desselben Browsers zu beachten. Diese können sich auf die Kompatibilität der entwickelten Webanwendung auswirken und unter anderem die korrekte Darstellung der Oberflächenelemente oder Inhalte beeinträchtigen. Auch diese Aspekte sind nicht Teil der Anforderungen.

4 Konzeption des Testansatzes

Bei Cypress handelt es sich um ein Framework zur Testautomatisierung mit Fokus auf E2E-Tests (im Folgenden "E2E-Tests", s. Kapitel 2.1.6), welches für moderne Webanwendungen entworfen wurde (vgl. Mwaura, 2021, S.3). Aus diesem Grund darf ein Anwender dieses Frameworks erwarten, dass ein breites Spektrum von verschiedenen Nutzungsszenarien abgedeckt werden kann. Im Folgenden wird beschrieben, wie und mit welchen Zielen und Ansprüchen die Testautomatisierung für das Demonstrationsobjekt implementiert werden soll. Zunächst wird beschrieben, welche Methoden von Cypress hauptsächlich genutzt werden, und welche Funktionen diese jeweils bieten. Anschließend wird ausgearbeitet, welche Arten von Tests in diesem Testkonzept eingeplant werden. Des Weiteren werden einige beispielhafte funktionale sowie nicht-funktionale Aspekte aufgelistet, welche potenziell Teil einer zu testenden Webanwendung sein können, im Rahmen dieser Ausarbeitung jedoch nicht betrachtet werden sollen.

4.1 GUI-Tests mit Cypress

In diesem Abschnitt werden einige Voraussetzungen, Rahmenbedingungen und Ziele zur Umsetzung von GUI-Tests am Demonstrationsobjekt konzipiert.

Die Benutzerinteraktionen mit dem Demonstrationsobjekt spielen sich hauptsächlich auf den Seiten "Recipes" und "Ingredients" ab. Das Zubereiten eines oder mehrerer Rezepte auf der Recipes-Seite nimmt direkten Einfluss auf die Werte der verschiedenen Zutaten, welche auf der Ingredients-Seite abgebildet werden. Die geplanten Tests müssen dies berücksichtigen, indem diejenigen, welche die Zubereitung von Rezepten auf der Recipes-Seite beinhalten, ebenfalls die automatische Anpassung der Darstellung von verfügbaren Zutatenmengen auf der Ingredients-Seite sicherstellen.

Der relativ häufige Wechsel zwischen den beiden Hauptseiten der Anwendung mag die klare Trennung von Zuständigkeiten der betroffenen Testdateien und -methoden zwar abschwächen, simuliert jedoch realistische Verhaltensweisen von Nutzern.

Um die Unabhängigkeit der Recipe-Tests von der Darstellung der Ingredients-Seite zu erhöhen, können Datenbanktests eingebunden werden, welche im folgenden Abschnitt erläutert werden.

Des Weiteren bietet eine Auswahl von Drittanbieter-Plugins für Cypress die Möglichkeit, u.a. visuelle Tests mit Hilfe von Screenshots umzusetzen. Dies wird in Kapitel 6.3 behandelt.

4.2 Datenbanktests mit Cypress

Der Schwerpunkt von Cypress liegt auf E2E-Tests, welche die Interaktion mit GUI-Elementen beinhalten. Dennoch bietet es verschiedene Methoden, um die Kommunikation zwischen der zu testenden Anwendung und ihrer zugehörigen Datenhaltung alleinstehend zu testen. Hierzu wird hauptsächlich die Methode *cy.request()* genutzt. Diese ermöglicht das Senden von HTTP-Anfragen an die Persistenzschicht der zu testenden Anwendung. Die zurückgelieferten Antworten können auf das erwartete Format und den korrekten Inhalt überprüft werden. Eine detaillierte Beschreibung dieser Methode ist in Kapitel 6.6.1 zu finden. Die Grundlage der Tests von Datenhaltung und API werden in dieser Ausarbeitung die GET-Anfragen darstellen. Diese werden eingesetzt, um den Datenstand zum Testzeitpunkt festzustellen. Wann immer eine Änderung an den Datenbankwerten vorgenommen wird - entweder implizit durch Benutzeroberflächeninteraktion oder explizit durch eine HTTP-Anfrage - kann mit Hilfe von GET-Anfragen festgestellt werden, ob die Werte in der Datenbank den erwarteten Zustand erreichen.

Unter anderem sollen mit HTTP-Anfragen Benutzerinteraktionen simuliert werden, um die Serverkommunikation unabhängig von visuellen Schaltflächen zu testen. Auf diese Weise kann sichergestellt werden, dass ein Datenpaket vom Server korrekt verarbeitet, und eine Aktualisierung der betroffenen Daten wie erwartet umgesetzt wird, auch wenn die Benutzeroberfläche vorübergehend nicht zum Test zur Verfügung steht. In solch einem Fall können POST-, PUT- und PATCH-Anfragen verwendet werden, um die benötigten Funktionen auszulösen, ohne Eingabefelder und ähnliches zu nutzen.

Werden Tests entwickelt, welche Veränderungen an den in der Datenbank hinterlegten Werten auslösen, so ist es empfehlenswert, diese Änderungen vor dem Start des darauffolgenden Tests

stets rückgängig zu machen. Nur so kann für spätere Tests ein Stand der Datenhaltung garantiert werden, welcher ein zuverlässiges Testen erlaubt. Wird dieser Zustand nach der Ausführung eines Tests nicht wiederhergestellt, führt dies zu nicht deterministischem Verhalten und unzuverlässigen Testergebnissen. Auch hierfür können POST-, PUT- und PATCH-Anfragen hilfreich sein.

4.3 Unit-Tests mit Cypress

Unit-Tests sind unter Cypress generell umsetzbar. Hierzu können gezielt Instanzen einzelner Module erstellt und anschließend isoliert getestet werden. Die Entwickler von Cypress stellen jedoch explizit klar, dass es nicht primär für diese Art von Tests vorgesehen ist: “Cypress is not a general automation framework, nor is it a unit-testing framework for your back end services” (Cypress.io FAQ, 2021). Aus diesem Grund sollen Unit-Tests nicht im Fokus dieser Ausarbeitung stehen. Nichtsdestotrotz würden Unit-Tests einen Weg darstellen, Whitebox-Tests zu dem Testumfang des Projektes hinzuzufügen. Daher soll ein Versuch unternommen werden, Unit-Tests sinnvoll in die Testautomatisierung des Demonstrationsobjekts einzugliedern. In Kapitel 6.5 wird auf die Erfahrungen eingegangen, die bei der Umsetzung von Unit-Tests unter Cypress gemacht werden konnten, und ob diese erfolgreich war.

4.4 Erweiterbarkeit von Cypress-Tests

Wie in Kapitel 1.2 beschrieben, soll während der Testautomatisierung des Demonstrationsobjekts der Fokus nicht auf dem Erreichen einer möglichst hohen Testabdeckung liegen, sondern vor allem darauf, Stärken und Schwächen von Cypress herauszuarbeiten. Einer der Aspekte, auf welchen hin die Anwendererfahrung während der Verwendung von Cypress analysiert werden soll, stellt die in Kapitel 4.4 definierte Erweiterbarkeit der Tests dar. Diese kann mit einer Demonstration der Erweiterbarkeit des Funktionsumfangs des Demonstrationsobjekts verknüpft werden, wie sie in Kapitel 3.3.7 erläutert wird.

Um diese zu testen, soll dem bereits bestehenden, lauffähigen und testbaren Demonstrationsobjekt eine neue Charakteristik hinzugefügt werden. Diese sollte Einfluss auf jede Softwareschicht der zu testenden Anwendung haben. Zu diesem Zweck wird geplant, eine

fünfte Zutat zu implementieren, die gleichwertig zu den ursprünglichen vieren verwendet werden kann. Sobald nach der Implementierung ebendieser Funktionsänderung erneut ein stabiler Zustand der Applikation vorliegt, sollen die automatisierten Tests so erweitert oder angepasst werden, dass grob der zuvor erreichte Abdeckungsgrad erreicht wird. Alle entsprechenden Arbeitsschritte können in Kapitel 6.7 nachvollzogen werden.

4.5 Ausgeschlossene Testszenarien

Dieses Unterkapitel erläutert kurz, welche Aspekte der zu testenden Anwendung nicht abgedeckt werden sollen und listet jeweils passende Beispielanforderungen auf.

4.5.1 Funktionale Aspekte

Viele funktionale Aspekte würden die Abdeckung des hier erstellten Testkonzeptes zwar erweitern, würden das Ziel der Allgemeingültigkeit und Übertragbarkeit auf möglichst viele Anwendungen jedoch verfehlen und den Rahmen der Ausarbeitung sprengen. Hierzu gehören beispielsweise - jedoch nicht ausschließlich - die folgenden funktionalen Bereiche:

- Vernetzung von verteilten Systemen und Smart-Home-Geräten
- Verlässliche Protokolle und Middleware
- Registrierung, Anmeldung, Benutzerkontoverwaltung und Abmeldung
- Browserübergreifende Kompatibilität
- Nutzbarkeit auf mobilen Endgeräten

4.5.2 Nicht-funktionale Aspekte

Grundsätzlich als sinnvoll für die Vollständigkeit eines Testkonzeptes angesehen, jedoch im Rahmen dieser Arbeit ebenfalls nicht eingeschlossen sind unter anderem diese nicht-funktionalen Punkte:

- Lokalisation
- Performanz

- Effizienz
- Robustheit
- Informationssicherheit
- Lasttests

4.5.3 Aspekte verteilter Systeme

Die entwickelte Anwendung soll eine Grundlage zum Testen gängiger Funktionen von Webanwendungen geben. Bei diesen handelt es sich um verteilte Systeme. “Ein verteiltes IT-System umfasst Teilsysteme [...], die im Rahmen einer bestimmten Architektur miteinander gekoppelt sind und Aufgaben kooperativ abwickeln” (Fink, 2012). Die Kommunikation zwischen der entwickelten Webanwendung und einem physischen Smart-Home-Gerät stellt einen komplexen Aspekt dar, welcher im Kontext der Smart-Home-Entwicklung existiert und nicht für jede Art von Webanwendung von Interesse ist. Dieser Aspekt geht über den Anforderungshorizont dieser Arbeit hinaus und wird daher nicht in die Anforderungen aufgenommen (s. Kapitel 4.5.1). Es werden lediglich die Schnittstellen innerhalb der Webanwendung, die lokal ausgeführten Funktionen und Operationen, sowie die Kommunikation zur Datenhaltung entworfen, entwickelt und getestet. Protokolltests und ähnliches entfallen somit.

4.5.4 Datenhaltung

Die Anbindung einer robusten Datenhaltung ist eine maßgebliche Grundlage für die Entwicklung verschiedenster Applikationen. Ebenso kann die Auswahl einer Datenbanktechnologie die Entwicklung einer Anwendung und ihrer schichtenübergreifenden Tests entscheidend beeinflussen. Allerdings können Datenbanktests sehr projektspezifisch ausfallen und sich ggf. in verschiedenen Testebenen unterschiedlich gestalten. Aus diesen Gründen wird ihr Nutzen im Kontext dieser Ausarbeitung geringer bewertet als jener anderer Aspekte. Daher soll im Rahmen der hier gewählten Anwendung nicht detailliert auf die Auswahlkriterien einer passenden Datenbankstruktur eingegangen werden. Die wichtigsten Informationen werden in Kapitel 5.4 kurz zusammengefasst. Auch das Testen der

Persistenzschicht soll einen vergleichsweise geringen Teil der Testautomatisierung ausmachen und wird in Kapitel 6.6 bearbeitet.

5 Realisierung des Demonstrationsobjekts

In Kapitel 3 wurden bereits die Anforderungen an das Demonstrationsobjekt definiert, die es braucht, um eine valide Grundlage zur Erörterung der Forschungsfrage zu haben. Nach Umsetzung des Demonstrationsobjekts soll nun im folgenden Kapitel die Realisierung dieser Anforderungen beschrieben werden. Manche Anforderungen wurden wie geplant realisiert, andere konnten erst während der Umsetzung konkretisiert werden und werden nun näher beschrieben. Anschließend wird die Systemarchitektur der entwickelten Webanwendung präsentiert.

5.1 Technischer Hintergrund

Die Entwicklung der zu programmierenden Webanwendung wird mit Hilfe des Frontend-Webapplikations-Frameworks Angular entwickelt. Dieses basiert auf der Programmiersprache TypeScript in Kombination mit HTML und CSS. Die Entscheidung für Angular beruht auf einer Testphase zu Beginn der Ausarbeitung. Alle Ergebnisse dieser Ausarbeitung sind jedoch auf andersartig entwickelte Webanwendungen übertragbar.

Die Ausführung des geschriebenen Quellcodes wird standardmäßig im Standardbrowser des vom Entwickler genutzten Endgerätes getätigt. Während der Entwicklung handelte es sich bei diesem um Google Chrome, zuletzt in der Version 95.0.4638.54 (Stand 23.10.2021). Die unter Cypress automatisierten Tests wurden ausschließlich in der Headless-Variante des Webbrowsers Google Chrome in der Version 95 durchgeführt, welche von Cypress automatisch mitgeliefert wird.

5.2 Umsetzung der Anforderungen an das Demonstrationsobjekt

Dieser Abschnitt behandelt verschiedene Nutzerszenarien, welche durch die Realisierung des Demonstrationsobjekts, wie sie nun vorgenommen worden ist, ermöglicht werden. Begonnen wird hierbei mit den in Kapitel 3.4.1 definierten Anwendungsfällen. Zu jedem Szenario wird konkretisiert, welche Abläufe der Benutzer durchlebt. Des Weiteren wird ausgeführt, welche

Oberflächen, Eingabeaufforderungen, Warnhinweise und Fehlermeldungen umgesetzt wurden und ihn durch seine Nutzererfahrung führen.

5.2.1 Anlegen von Rezepten

Die Seite “Recipes” präsentiert dem Benutzer als erstes ein Feld mit einem Text- und mehreren Nummerneingabefeldern. Hierbei handelt es sich um das Eingabeformular für das Erstellen eines neuen Rezeptes. Das Textfeld ermöglicht die Benennung des zu erstellenden Rezeptes. Der Name ist hierbei beschränkt auf eine Anzahl von mindestens 2 und maximal 20 Zeichen.

Die Eingabefelder für Zutatenmengen sind so entworfen, dass nur zulässige Werte eingetragen werden können. Werte oberhalb der Maximalfüllmenge von 1.500 Einheiten werden auf ebendiese herunterkorrigiert. Negative Eingabewerte werden auf 0 korrigiert. Nur Felder, die eine Zutatenmenge von mehr als 0 beinhalten, gelten als gültig ausgefüllt. Es muss mindestens eine Zutat mit einem Wert von 1 oder höher ausgefüllt werden, um das neue Rezept speichern zu können.

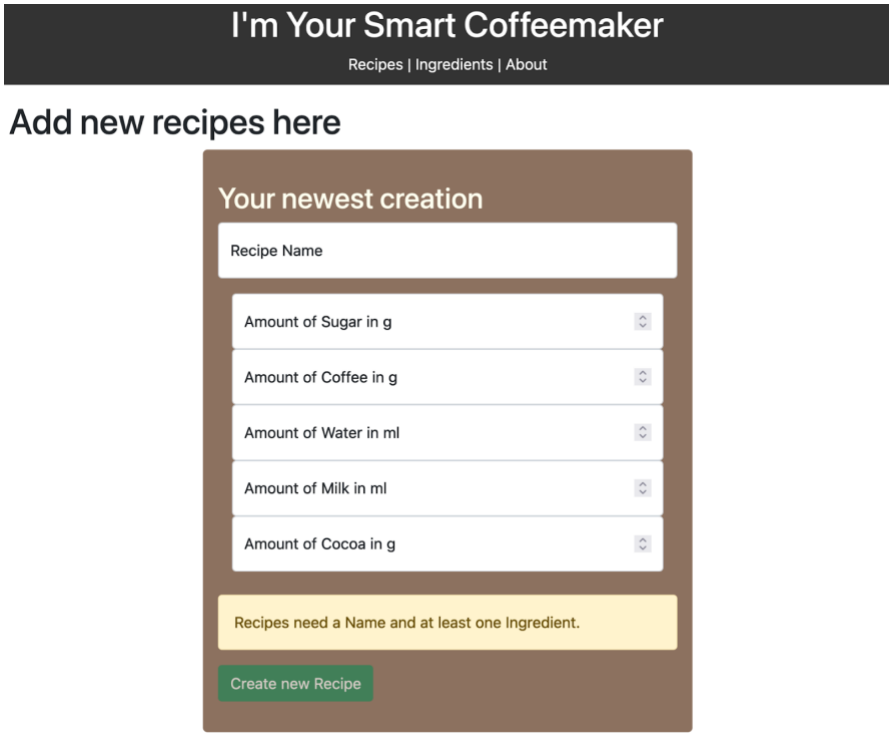

Abbildung 7: Screenshot - Neues Rezept hinzufügen.

Ist eine der Voraussetzungen nicht erfüllt, bleibt der Knopf zum Speichern des Rezeptes deaktiviert und ausgegraut. Wird der Knopf in diesem Zustand betätigt, wird kein neues Rezept angelegt. Ist dies der Fall, wird dem Nutzer ein Warnhinweis angezeigt, welcher ihn darauf hinweist, dass Rezepte sowohl einen gültigen Namen als auch mindestens eine Zutat benötigen. Zusätzlich erscheint eine Fehlermeldung an dem Texteingabefeld für den Rezeptnamen, sollte der eingegebene Name zu kurz sein. Ein zu langer Name wird von dem Textfeld durch automatische Limitierung der Zeichenanzahl vermieden. Für diesen Fall ist somit keine Anzeige vonnöten.

Hat der Benutzer alle Unzulänglichkeiten in seinen eingegebenen Werten korrigiert, so verschwinden die Hinweise umgehend. Zugleich wird der Knopf zum Speichern des Rezeptes aktiviert und der ausgegraute Effekt beseitigt. Nun kann der Benutzer das Rezept mit seinen eingegebenen Daten anlegen lassen. Wird der Knopf zum Speichern des Rezeptes in aktivem Zustand betätigt, wird die Eingabemaske umgehend geleert. Wirft der Benutzer daraufhin einen Blick auf den unteren Teil der “Recipes”-Seite, so findet er sein soeben angelegtes Rezept an hinterster Stelle der Rezeptübersicht.

5.2.2 Anzeigen und Kochen von Rezepten

Unter dem Bereich zum Anlegen neuer Rezepte befindet sich derjenige Abschnitt, der die Liste der bestehenden Rezepte anzeigt. Dabei wird zu jedem Rezept die Menge jeder Zutat aufgelistet, die beim Kochen verbraucht wird. Jedes Rezept hat außerdem einen Button, über den der Kochvorgang gestartet werden kann.

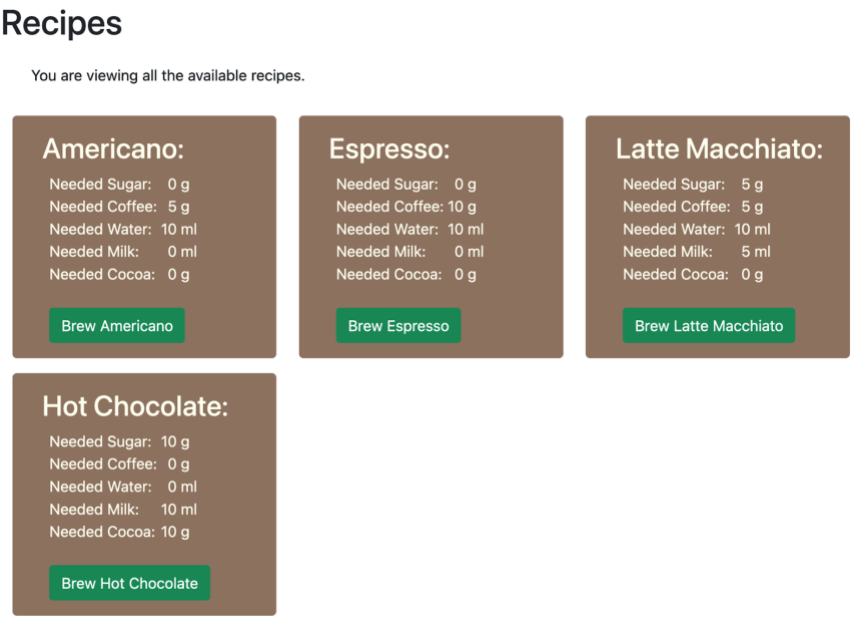

Abbildung 8: Screenshot - Rezepte anzeigen und zubereiten.

In welchem Maße Zutaten verfügbar sind, wird nicht an dieser Stelle angezeigt, sondern im Tab “Ingredients” (s. Abschnitt 5.2.4). Eine zusätzliche Implementierung der Füllstandsanzeige in der Rezeptübersicht ist für das Testszenario nicht relevant und hätte einen erhöhten Programmieraufwand bedeutet.

Der Knopf zum Zubereiten eines Rezeptes kann zu jeder Zeit angeklickt werden. Tut der Benutzer dies, wird im Hintergrund überprüft, ob von jeder benötigten Zutat eine ausreichende Menge zur Verfügung steht. Hierzu wird der aktuelle Füllstand aus der Datenbank abgefragt. Sind zum aktuellen Zeitpunkt von allen Zutaten genügend Einheiten eingetragen, so wird der virtuelle Kochvorgang durchgeführt. Hierbei werden die im Rezept angegebenen Mengen von den Füllständen der jeweiligen Zutaten abgezogen. Die Werte in der Datenbank werden in diesem Zuge umgehend aktualisiert. Anschließend wird im Browser eine Toast-Benachrichtigung ausgelöst, also eine Nachricht, die am oberen Bildschirmrand erscheint. Diese benachrichtigt den Benutzer darüber, dass das Rezept erfolgreich zubereitet wurde.

Reicht der Füllstand von einem oder mehreren Zutaten nicht zum Zubereiten des gewählten Rezeptes aus, so ändern sich die Datenbankwerte nicht. Auch in diesem Fall wird dem Benutzer eine Toast-Benachrichtigung angezeigt. Diese berichtet jedoch davon, dass das gewünschte

Rezept nicht zubereitet werden konnte und welche der Zutaten in unzureichender Menge verfügbar war. Hierbei handelt es sich um die erste negativ geprüfte Zutat. Mehrere zu geringe Füllstände gleichzeitig werden nicht angezeigt.

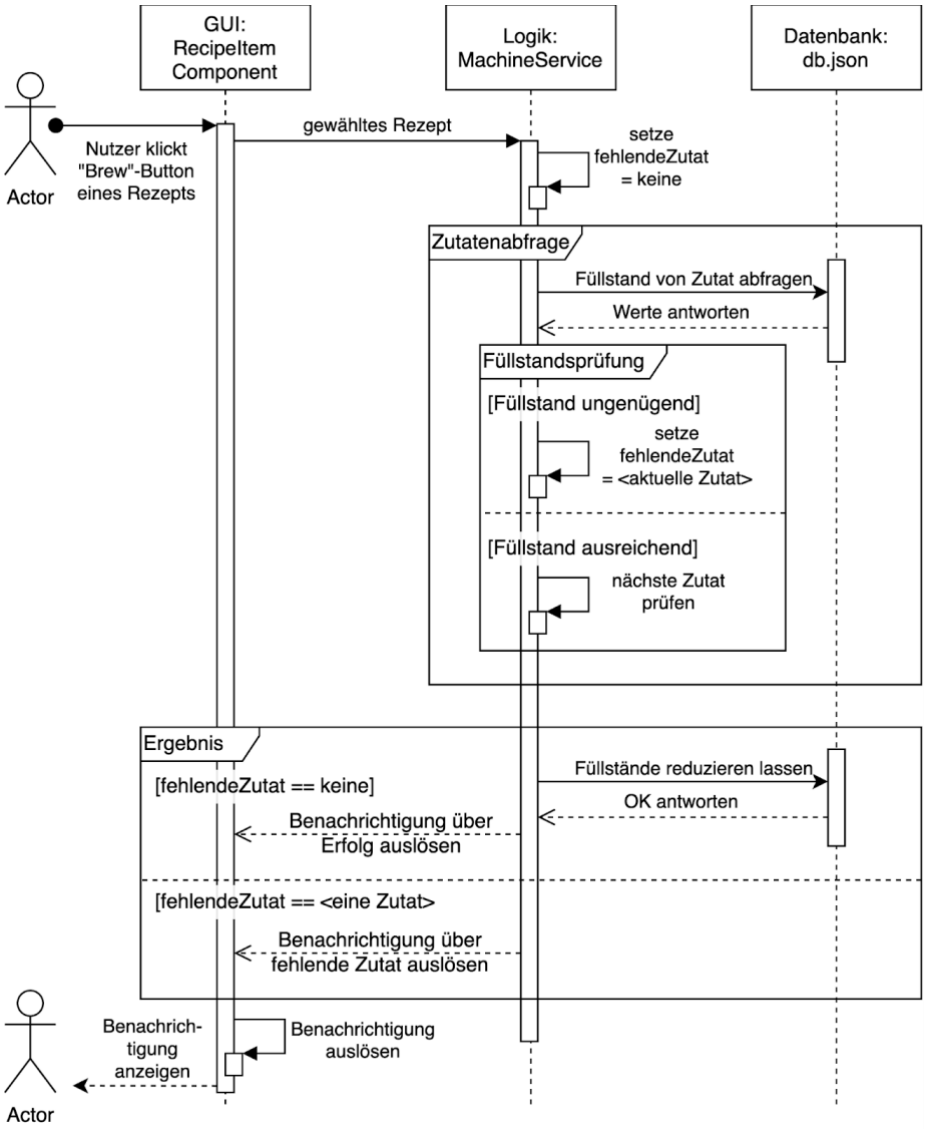

Abbildung 9: Sequenzdiagramm - Zubereitung eines Rezepts inkl. Zutatenprüfung.

Dieses Sequenzdiagramm beschreibt den Programmfluss sowohl in dem Fall, dass ein Rezept erfolgreich zubereitet werden kann, als auch jenen, dass eine Zutat nicht in ausreichendem Maße vorgefunden wird. Im folgenden Screenshot wird sichtbar, wie die Browserbenachrichtigung aus Sicht des Benutzers dargestellt wird.

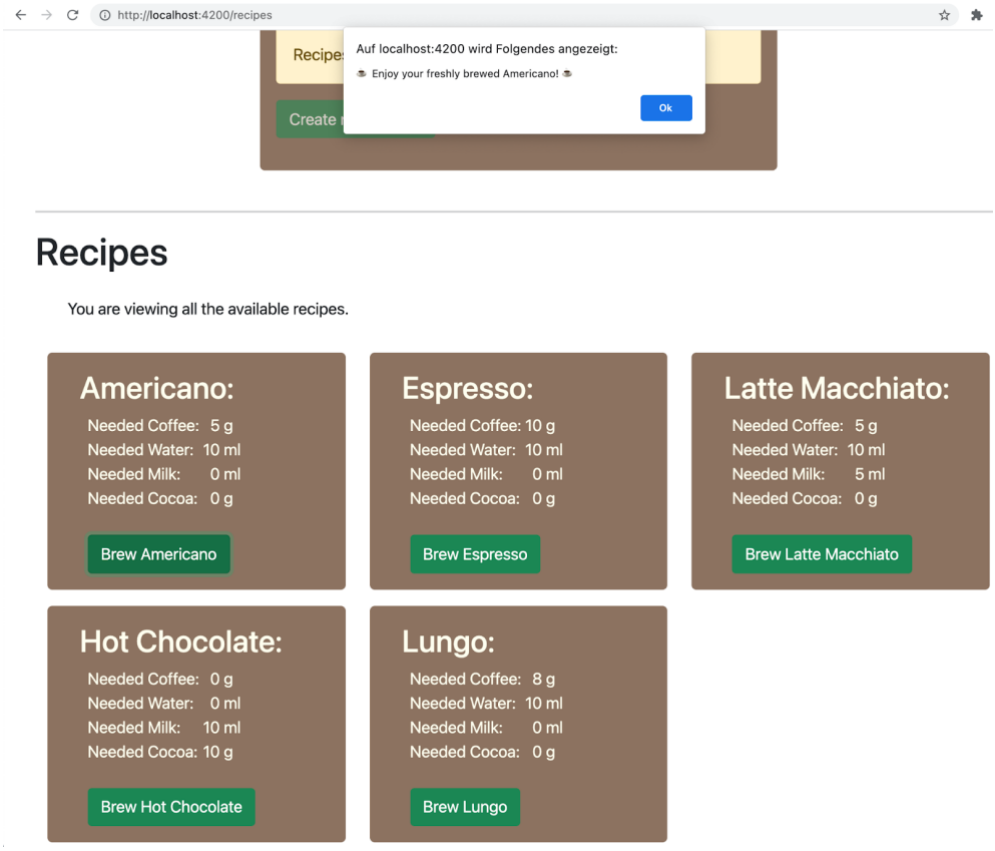

Abbildung 10: Screenshot - Browserbenachrichtigung nach erfolgreicher Rezeptzubereitung.

5.2.3 Löschen von Rezepten

Ebenfalls an dieser Stelle könnte ein Löschen-Knopf für jedes Rezept angezeigt werden. Würde dieser angeklickt, so würde das zugehörige Rezept aus der Datenbank gelöscht. Die Anzeige könnte automatisch aktualisiert werden und das gewählte Rezept wäre nicht mehr sichtbar.

Durch das Hinzufügen einer Löschfunktion zu dem Demonstrationsobjekt würde zwar eine weitere CRUD-Operation demonstriert, doch für die Forschungsfrage entstünden keine ausreichend relevanten neuen Erkenntnisse. Aus diesem Grund wurde auf diese Funktion verzichtet.

5.2.4 Anzeigen und Aktualisieren von Zutatenmengen

Im zweiten Tab “Ingredients” ist die Liste aller Zutaten sichtbar, auf die die virtuelle Smart-Home-Kaffeemaschine zugreifen kann, um Rezepte zuzubereiten. Jede Zutat wird auf einem farblich hervorgehobenen Feld abgebildet und zeigt in ihrer Überschrift den jeweiligen Füllstand an. Der Knopf zum Nachfüllen einer Zutat ist standardmäßig deaktiviert und ausgegraut. Er wird aktiviert, sobald in das Zahleneingabefeld ein gültiger positiver oder negativer Wert eingetragen wird, um den Füllstand zu erhöhen oder zu senken.

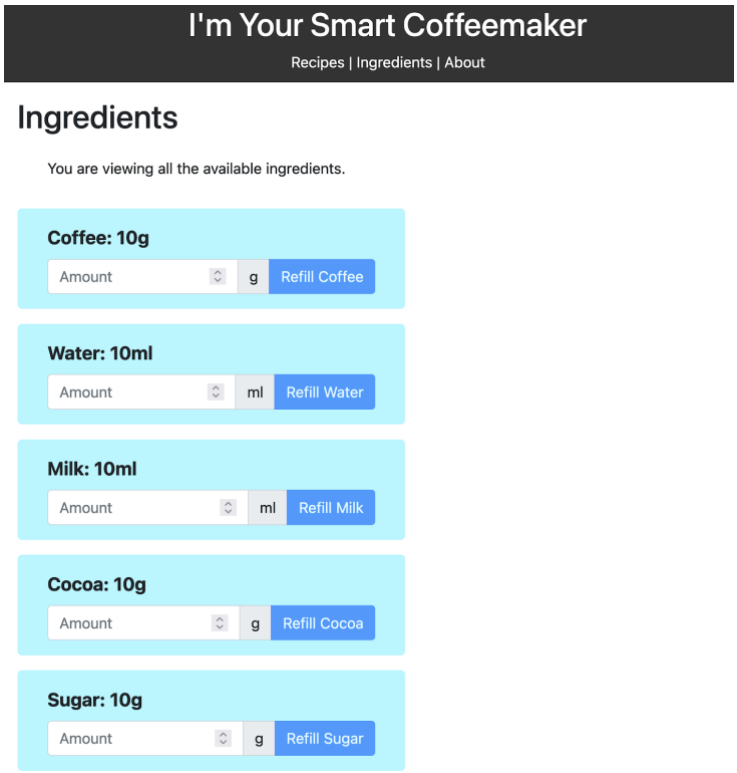

Abbildung 11: Screenshot - Zutaten nachfüllen oder entnehmen.

Warnhinweise oder Fehlermeldungen werden in folgenden Fällen eingeblendet:

- Der hinzuzufügende positive Betrag ist so hoch, dass nach seinem Hinzufügen der Gesamtbetrag den maximal zulässigen Wert 1.500 übersteigen würde. Dieser Betrag wurde festgelegt, um der Anforderung der zustandsabhängigen Ausführungen bestimmter Operationen nachzukommen. Die Fehlermeldung weist darauf hin, dass in die Kaffeemaschine maximal 1.500 Einheiten fassen kann.

- Der abzuziehende negative Betrag ist so niedrig, dass nach seiner Subtraktion der Gesamtbetrag den minimal zulässigen Wert 0 unterschreiten würde. Die Fehlermeldung weist darauf hin, dass der Füllstand nicht negativ werden darf.
- Das Ziffernfeld wird angeklickt, jedoch wieder verlassen, ohne dass ein Wert eingetragen wurde. Oder der Button wird geklickt, ohne dass zuvor ein Wert ungleich null eingegeben wurde. Der Warnhinweis weist darauf hin, dass ein gültiger Wert eingegeben werden muss.

Wird ein valider Wert in das Zahlenfeld einer Zutat eingetragen, kann der Nachfüllen-Knopf betätigt werden. Daraufhin wird der angezeigte Wert des Füllstandes der jeweiligen Zutat unverzüglich aktualisiert. Die Seite muss hierzu nicht neu geladen werden. Diese Aktion kann beliebig oft für beliebige verschiedene Zutaten wiederholt werden, solange der gewünschte Wert durch die Füllstandsveränderung nicht ungültig wird. Sollte dies doch geschehen, so wird der Nachfüllen-Knopf wieder deaktiviert und die entsprechende Fehlermeldung erneut eingeblendet.

5.2.5 About

Im dritten Tab namens “About” befindet sich die Informationsseite. Hier befindet sich ein Download-Link, der nach Klick eine im Projektordner hinterlegte Datei herunterladen lässt. Dieser Teil des Demonstrationsobjekts wurde implementiert, um das Testen von Datei-Downloads und ggf. weiteren interessanten Testfällen zu ermöglichen. Es zeigte sich jedoch, dass sich aus diesen Tests keine unmittelbar relevanten Ergebnisse für die Beantwortung der Forschungsfrage dieser Ausarbeitung ergaben. Somit wird dieser Tab und seine Testautomatisierung in den folgenden Kapiteln nicht weiter thematisiert.

5.3 Architektur

Die Softwarearchitektur des Demonstrationsobjekts orientiert sich an der Komponentenstruktur, welche bei der Erstellung eines neuen Angular-Projekts generiert und laut Darwin (2021) als Best Practice empfohlen wird. In der Präsentationsschicht wird eine Komponente für jeden Bereich erstellt, dessen Zuständigkeit klar von der restlichen Seite

abgegrenzt werden kann. Als Beispiel sollen die Komponenten der Recipes-Seite betrachtet werden.

- Die Komponente RecipesComponent ist ein Behälter für die zwei Unterklassen AddRecipeComponent und RecipeItemComponent und gibt den Seitenaufbau vor. Benannt wird sie, ebenso wie die IngredientsComponent nach dem Tab, den sie abbildet.
- Oben auf der Recipes-Seite ist die AddRecipeComponent zu sehen. Diese ermöglicht es dem Nutzer, neue Rezepte anzulegen und besteht aus einer Eingabemaske mit mehreren Feldern.
- Der untere Abschnitt der Recipes-Seite zeigt die aktuell vorhandenen Rezepte. Diese werden durch eine Liste, bzw. ein Gitter von mehreren Instanzen der RecipeItemComponent dargestellt. Jede Instanz bildet eines der standardmäßig vorhandenen oder manuell angelegten Rezepte ab. Sie halten Informationen über das jeweilige Rezept sowie einen Knopf für die Zubereitung bereit.

In der folgenden Grafik werden alle Komponenten auf die drei Softwareschichten verteilt dargestellt:

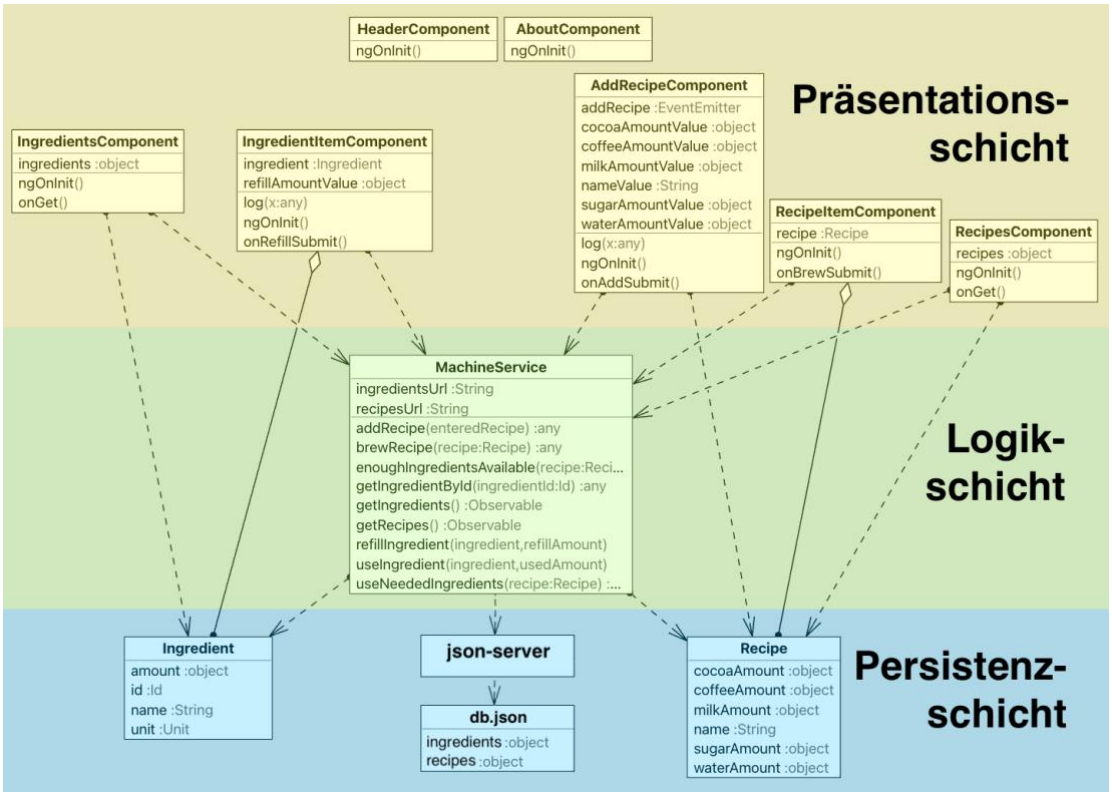

Abbildung 12: Klassendiagramm des Demonstrationsobjekts - Generiert mit dem VSCode-Extension classdiagram-ts (vgl. Shen, 2019), manuell erweitert um json-server und db.json.

Wird mit bestimmten Teilen der Präsentationsschicht interagiert, löst dies eine Kommunikation mit der Geschäftslogik aus. Diese wird in der vorliegenden Implementierung von einem zentralen Service namens MachineService verkörpert. In diesem spielen sich beispielsweise die Berechnungen ab, ob ein Rezept zubereitet werden kann, oder es an einer Zutat mangelt. Werden Zutaten verbraucht oder nachgefüllt, werden die Werte hier verarbeitet und via HTTP-Anfrage an die Datenhaltung übermittelt.

Zur Persistenzschicht zählen zunächst die Datenmodelle Ingredient und Recipe, welche für die Kommunikation zwischen den Komponenten verwendet werden. Die tatsächliche Datenbank befindet sich in der Datei db.json. Sie besteht aus einer Tabelle, welche alle Zutaten mit ihren aktuellen Füllständen bereithält, und einer Tabelle für alle eingetragenen Rezepte mit ihren

benötigten Zutatenmengen. Die Datenbank wird vom Open-Source-Tool json-server verwaltet und via REST-API verfügbar gemacht. Auf diesem Wege finden sowohl die internen Operationen des MachineService als auch externe Anfragen von Cypress statt.

5.4 Anbindung einer Datenbank

Zur Implementierung und Anbindung einer Persistenzschicht mit Datenhaltung und API wird das Tool json-server in der Version 0.16.3 verwendet. Dieses wird innerhalb des Projektes per Kommandozeilenaufruf als NPM-Package installiert. Daraufhin können die gewünschten Tabellen und Datenmodelle manuell in einer json-Datei angelegt werden. Diese Datei wird von json-server bei Start ausgelesen und unter einer URL als Datenbank zur Verfügung gestellt. Das Tool kann über ein simples Kommando via Kommandozeile gestartet werden. Die URL, auf welcher die Datenbank zur Verfügung gestellt wird, ist standardmäßig festgelegt, kann jedoch bei Bedarf angepasst werden.

6 Umsetzung der Testautomatisierung

Nachdem die Tests konzeptioniert und das Demonstrationsobjekt entwickelt wurde, soll in diesem Kapitel, nach einem kurzen Einblick in die Architektur von Cypress selbst, das Schreiben der automatisierten Tests erläutert werden. Wie in Kapitel 1.2 bereits erwähnt, ist eine möglichst hohe Testabdeckung nicht Teil der Zielsetzung. Vielmehr sollen verschiedene Aspekte von Cypress demonstriert werden, die zu einem schichtenübergreifenden Testen beitragen.

6.1 Technischer Hintergrund

Im Folgenden soll ein genereller Einblick in die Funktionsweise von Cypress gegeben werden. Das Ziel besteht hierbei nicht darin, eine umfassende Beschreibung des Zusammenspiels von Node.js, dem Browser und Cypress zu geben. Es soll vielmehr ein grundlegendes Bild geschaffen werden, um das Verständnis davon zu unterstützen, welche technischen Abläufe stattfinden, wenn ein mit Cypress verfasster Test ausgeführt wird.

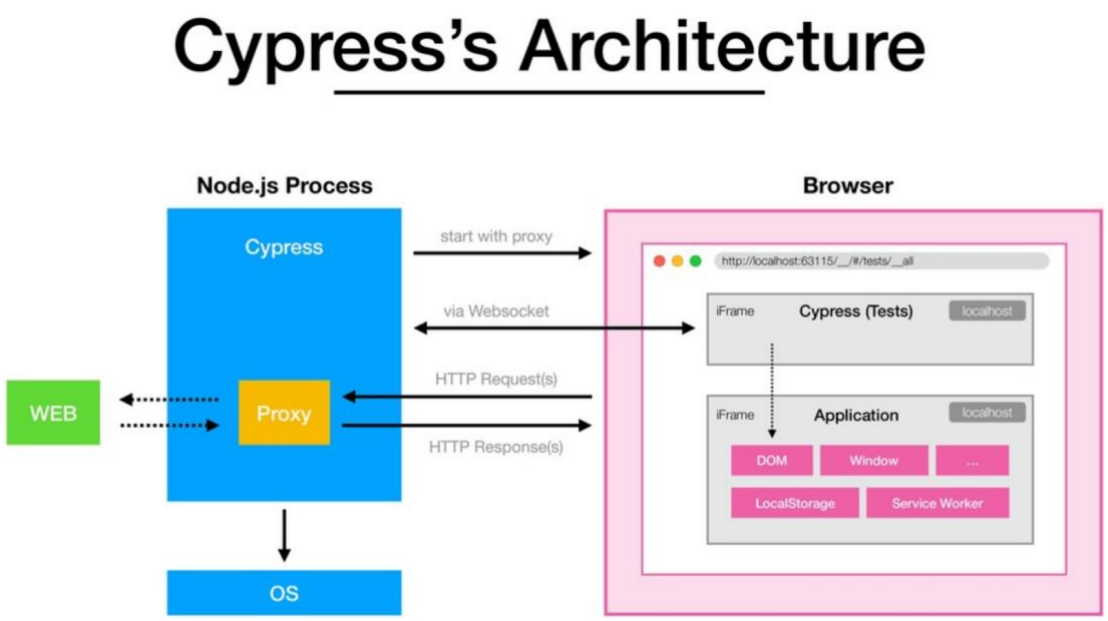

Abbildung 13: Grafik - Architektur der Testausführung durch Cypress (Elm, 2018).

Anhand dieses Diagramms wird die Architektur von Cypress in Verbindung mit der zu testenden Anwendung (hier "Application") verbildlicht. Cypress wird als ein Node.js-Prozess vom Tester gestartet. Dies findet entweder über die Benutzeroberfläche des von Cypress zur Verfügung gestellten Test Runners statt, oder per Kommandozeilenaufruf.

Wird ein Testlauf initialisiert, so wird zunächst automatisiert ein Browser gestartet. Bei diesem handelt es sich standardmäßig um Chrome, doch auch verschiedene Versionen anderer Browser können konfiguriert werden, um Tests stattdessen oder zusätzlich in diesen ausführen zu lassen, und so eine höhere Testabdeckung zu erreichen.

Anschließend werden die zuvor in JavaScript-Dateien geschriebenen Cypress-Testmethoden an den gestarteten Browser übermittelt. In diesem wird ebenfalls die zu testende Anwendung initialisiert. "Im von Cypress instrumentalisierten Browser wird eine Seite aufgerufen, die die getestete Applikation wie auch [eine] Cypress-UI als iFrame einbettet. Der Testcode wird ebenfalls im Cypress-iFrame ausgeführt, sodass Testcode und Anwendungscode im gleichen Browser-Tab und somit auch im gleichen JavaScript-Loop laufen. Der Testcode kommuniziert über WebSockets mit dem Node-Prozess." (Verhoelen, 2020). Daraufhin werden die Tests Schritt für Schritt ausgeführt und die Auswirkungen auf die Applikation dokumentiert. Während der Ausführung kann der Tester den Fortschritt der gewählten Tests in Echtzeit nachverfolgen. Hierzu stehen sowohl die Cypress-Benutzeroberfläche als auch die Log-Ausgaben auf der Kommandozeile zur Verfügung. Die Dokumentation der Testabläufe kann wahlweise ausschließlich in Textform oder zusätzlich durch Videoaufzeichnungen der Browseroberfläche erfolgen.

Solange der Cypress-Prozess nicht beendet wird, werden die zuletzt ausgeführten Tests stets neu gestartet, sobald eine Änderung an den zugehörigen Testmethoden vorgenommen wurde. Hierdurch erhält der Tester umgehend eine Rückmeldung darüber, welchen Effekt seine letzte Änderung auf die aktuell betrachteten Tests hat.

HTTP-Anfragen werden über den sogenannten Proxy im Node.js-Prozess behandelt. Proxies beschreibt TechUpdatesDaily (2020) wie folgt: "The idea of proxy is used with reference to the agent who is authorized to carry out an action on behalf of another entity. [...] The proxy server, or simply proxy, allows to register and control traffic". In diesem Fall stellt dieser die Verbindung zu externen Adressen außerhalb der zu testenden Anwendung dar. So befindet sich

in dem vorliegenden Demonstrationsbeispiel die Datenbank außerhalb der Anwendung. Der Proxy ermöglicht durch diesen Aufbau auch das Zwischenspeichern von gesendeten HTTP-Anfragen, wodurch ein automatisches Warten auf die jeweiligen Antworten umgesetzt wird. Ohne diese Funktion wäre eine sinnvolle Nutzung der Methode *cy.request()* nicht möglich gewesen, da die Cypress-Tests nach einer gesendeten Anfrage entweder unabhängig von der Antwort weiterlaufen, oder stets fehlschlagen würden.

Bei Cypress handelt es sich um ein "All-in-one testing framework" (Cypress.io, 2021). Dies bedeutet, dass beispielsweise die zu nutzende Assertion Library nicht separat vom Anwender gewählt und manuell installiert wird, sondern alle Bestandteile bei der Installation von Cypress automatisch mitgeliefert werden (vgl. Cypress.io Tools, 2021). Bei den verwendeten Bestandteilen handelt es sich u.a. um die folgenden:

- Die Syntax ist auf dem zugrundeliegenden Testframework Mocha aufgebaut.
- Als Assertion Library, oder sogenanntes Testorakel, wird Chai genutzt.
    - Die Erweiterung Chai-jquery erleichtert das Durchsuchen des DOM nach zu testenden Seitenelementen.
- Für fortgeschrittene Funktionen wie sog. Spies und Stubs (s. Kapitel 6.5) wird Sinon verwendet.
    - Die Erweiterung Sinon-Chai ermöglicht die Verbindung der beiden namensgebenden Frameworks bei der Nutzung von Cypress.

Auf tiefergehende Details zu den genannten Programmbestandteilen und Libraries soll in dieser Ausarbeitung nicht weiter eingegangen werden.

Die Anwendung des Page Object Patterns zur Verwaltung verschiedener Bestandteile der zu testenden Website (vgl. Angelov, o.D.) ist unter Cypress möglich, findet in dieser Ausarbeitung jedoch nicht statt. Ein Grund hierfür ist, dass der Umfang der Testautomatisierung hier relativ gering ist, und der Nutzen der verbesserten Übersichtlichkeit, welche mit der Umsetzung des besagten Entwicklungsmusters erreicht werden soll, somit zu vernachlässigen wäre. Ein weiterer Grund ist, dass der Mehrwert von Page Objects in der Dokumentation von Cypress

angezweifelt wird (vgl. Bahmutov, 2019). Eine Analyse oder Stellungnahme zu diesem Thema ist in dieser Ausarbeitung nicht vorgesehen.

6.2 GUI-Tests

Betrachtet man die drei verschiedenen Schichten einer Webanwendung, so decken GUI-Tests primär die Präsentationsschicht ab. Sie können jedoch auch Teile eines E2E-Tests (s. Abschnitt 2.1.6) ausmachen und komplexere schichtenübergreifende Tests mit einer Schaltflächeninteraktion beginnen, oder durch die Prüfung eines dargestellten Wertes abschließen.

Im Folgenden soll an ausgewählten Beispielen demonstriert werden, wie sich das Testen der Benutzeroberfläche unter Cypress in der vorliegenden Ausarbeitung gestaltet hat. Hierzu wird in einigen Situationen der Cypress Test Runner zur Hilfe gezogen. Dieser erlaubt eine Übersicht über die durchgeführten Tests. Betrachtet man die Nutzbarkeit in Softwareentwicklungsteams, so wird er als hilfreich für die Kommunikation über einzelne Testschritte, sowie den Zustand der Anwendung, eingeschätzt. Dies ist sowohl in der Softwareentwicklung als auch in dieser Ausarbeitung hilfreich.

6.2.1 Testen der Darstellung der Benutzeroberfläche

Um die Benutzeroberfläche des Demonstrationsobjekts zu testen, werden verschiedene Methoden genutzt, welche bereits in Kapitel 2.3 erläutert werden. Die verschiedenen Seiten Recipes und Ingredients unterscheiden sich voneinander, weisen jedoch ausreichend Ähnlichkeiten auf, sodass das Testen ihrer Oberflächen im Folgenden zu großen Teilen generalisiert werden kann.

Der erste Schritt besteht auf einer jeden Seite darin, die Anwesenheit der wichtigsten Schaltflächen zu überprüfen. Dies kann grundlegend mit Hilfe der Methode *get()* geschehen und führt im Normalfall zu einem validen Test.

```
it('Recipe creation form exists, () => {
  cy.get('#form-new-recipe');
})
```

Wird diese erfolgreich ausgeführt, steht jedoch ausschließlich fest, dass das geforderte Objekt auf der Seite geladen wird. Um konkret sicherzustellen, dass es dem Benutzer auch angezeigt wird, sollte die Methode *should()* mit dem Parameter ‘be.visible’ eingesetzt werden.

```
it('Recipe creation form is visible', () => {
  cy.get('#form-new-recipe').should('be.visible');
})
```

Ist zu genüge getestet, dass die elementaren Schaltflächen der Anwendung sichtbar sind, können zusätzlich kritische Textfelder entsprechend geprüft werden. Bei Textfeldern wie Warnungen und Fehlermeldungen ist jedoch gehobener Wert darauf zu legen, dass die Felder nicht nur angezeigt werden, sondern außerdem die vorgesehenen Informationen enthalten. Auch diese Kontrolle wird durch die Methode *should()* erleichtert, in diesem Fall jedoch in Kombination mit zwei Parametern. Bei dem ersten muss es sich um das Schlagwort ‘contain.text’ handeln, und bei dem zweiten um den gewünschten Text.

Soll die Sichtbarkeitsprüfung beibehalten werden, so können die beiden Prüfungen mit Hilfe der Methode *and()* aneinandergereiht werden (vgl. Mwaura, 2021, S.66). Diese agiert nun wie ein weiterer *should()*-Aufruf, trägt jedoch durch ihren Wortlaut zu einer verbesserten Lesbarkeit bei.

```
it('Requirements warning shown by default, including correct info', () => {
  cy.get('#requirements-warning')
    .should('be.visible')
    .and('contain.text','Recipes need a Name and at least one Ingredient.');
})
```

Auch kann die Ausgangssituation überprüft werden, die der Benutzer vorfindet, wenn er die Anwendung zum ersten Mal besucht. Diese kann beinhalten, welche Aktionen bzw. Schaltflächen dem Benutzer standardmäßig angeboten werden. Beispielsweise darf ein Rezept nicht ohne Namen und Zutat erstellt werden. Hierauf wird der Benutzer von der im obigen Codeausschnitt getesteten Warnung hingewiesen. Werden diese Anforderungen nicht erfüllt, so sollte der Knopf zum Anlegen eines neuen Rezeptes deaktiviert sein. Dies lässt sich mit einem weiteren Parameter der Methode *should()* namens ‘be.disabled’ überprüfen.

```
it('Submit button disabled before any interaction', () => {
  cy.get('#submit-new-recipe').should('be.disabled');
})
```

6.2.2 Testen der Interaktionen mit der Oberfläche

Um die Bedingungen zum Anlegen eines Rezeptes zu erfüllen, kann die Methode *type()* auf dem Eingabefeld aufgerufen werden. Hierzu übergibt man den mit dem einzutippenden Text als Parameter. Da eine Vorgabe für die Zulässigkeit des Rezeptnamens darin besteht, dass er mindestens zwei Zeichen lang sein muss, kann hier ein Negativfall abgedeckt werden. Zu diesem Zweck wird im Folgenden ein einzelnes Zeichen eingefügt, anschließend auf eine beliebige Stelle außerhalb des Eingabefeldes geklickt, um die Textfeldprüfung auszulösen, und schließlich geprüft, ob die erwartete Fehlermeldung erscheint.

```
it('Error shown if name is shorter than 2 chars', () => {
  cy.get('#name').type('A');
  cy.get('#form-new-recipe').click();
  cy.get('#error-name-length').should('be.visible');
})
```

Eine weitere Beschränkung des Rezeptnamens ist seine Maximallänge von 20 Zeichen. In der vorliegenden Implementierung wird dies erzwungen, indem jeglicher nach dem 20. Zeichen zusätzlich eingegebener Text gelöscht wird. Dieses Verhalten wird getestet, indem ein Name eingetippt wird, welcher die Länge von 20 Zeichen überschreitet. Der hier verwendete Name besitzt eine Länge von 23 Zeichen. Nach der Eingabe wird überprüft, dass der übergebene Text nur bis zu seinem 20. Zeichen in dem zu prüfenden Eingabefeld erscheint.

```
 it('Recipe name is cut at 20 chars', () => {
  cy.get('#name')
    .find('input')
    .type('Best Coffee for TestERS')
    .should('have.value', 'Best Coffee for Test')
    .and('not.have.value', 'ERS');
})
```

Nachdem die Grenzwerte des Rezeptnamens getestet sind, soll nun die Funktionalität der Zahlenfelder für die Zutatenmengen analysiert werden. Da keine negativen Werte erlaubt sind, und ein Maximum von 1.500 Einheiten nicht überschritten werden darf (s. Kapitel 5.2.1), sollen diese beiden Einschränkungen hier getestet werden.

Da diese Testfälle jenen des Rezeptnamens in gewissem Maße ähneln, sollen sie genutzt werden, um eine verkürzte Schreibweise für Tests zu demonstrieren. Zum einen ist hier die Aneinanderkettung eines Zugriffs, einer Aktion und einer Prüfung in derselben Zeile zu sehen. Des Weiteren wird nicht wie in den vorhergegangenen Tests per *find()* auf das Eingabefeld zugegriffen, sondern dieses direkt mit in den CSS-Selektor geschrieben.

```
it('Ingredient amounts are max 1.500 and never negative', () => {
  cy.get('#coffeeAmount > input').type('1501')
    .should('have.value', '1500');
  cy.get('#milkAmount > input').type('-1')
    .should('have.value', '1');
})
```

Sind die benötigten Eingaben getätigt, wird erwartet, dass der Knopf zur Werteübermittlung aktiviert ist und geklickt werden kann. Im Gegensatz zu der vergangenen Prüfung, in welcher mit *should('be.disabled')* festgestellt werden sollte, dass der Knopf deaktiviert ist, nutzen wir nun *should('be.enabled')*.

```
it('Submit button clickable after name and 1 ingredient given', () => {
  cy.get('#name').type('Test Recipe');
  cy.get('#waterAmount').type(8);
  cy.get('#submit-new-recipe').should('be.enabled').click();
})
```

Ist diese Prüfung erfolgreich, kann der besagte Knopf auch umgehend verwendet werden. Hierzu verwenden wir die Methode *cy.click()*. Auch diese kann mit Verkettung an die vorhergegangene *should()*-Prüfung angehangen werden. Auf diese Weise wird das Klick-Kommando ebenfalls auf demselben, zuvor geprüften Knopf aufgerufen, ohne dass mit einem erneuten *cy.get()*-Aufruf auf diesen zugegriffen werden müsste. Da es sich auch bei dem *click()*-Aufruf um einen Testschritt handelt, welcher in der Ausführung fehlschlagen kann,

stellt dieser eine weitere Kontrolle dar, ob der Knopf nicht nur aktiviert ist, sondern auch tatsächlich verwendet werden kann.

Nach der Übermittlung der eingegebenen Werte, z.B. durch die Betätigung des Knopfes zum Anlegen eines neuen Rezeptes, werden diese an interne Prozesse der entwickelten Anwendung übergeben und weiter verarbeitet. Bei diesen Abläufen handelt es sich um Bereiche der Geschäftslogik, welche im folgenden Kapitel behandelt werden sollen.

6.3 Visuelle Tests mit Plugin

Die Funktionen von Cypress können u.a. um die Möglichkeit erweitert werden, visuelle Tests anhand von Screenshots und Bildvergleichen durchzuführen. Um entsprechende Methoden verfügbar zu machen, existieren Plugins in Form von Node Packages von verschiedenen Anbietern. Im Folgenden wird die Einrichtung, Nutzung und Konfiguration des gewählten Plugins namens cypress-plugin-snapshots erläutert.

6.3.1 Einrichtung des Plugins cypress-plugin-snapshots

Die Installation des Plugins wird gestartet, indem innerhalb des Projektordners ein einfaches Kommando ausgeführt wird. Wie schon bei der Installation von Cypress selbst wird dies durch die Nutzung von Node mit dem Zugriff auf das entsprechende Node Package (s. Kapitel 2.5) ermöglicht.

Anschließend müssen wenige Zeilen vorgegebenen Codes in die Konfigurationsdateien cypress.json und index.js eingefügt werden. Eine Anleitung zu der Einrichtung des Plugins ist auf der entsprechenden GitHub-Seite des Anbieters zu finden (vgl. van Straalen, 2020).

Zum Zeitpunkt der Verfassung dieser Arbeit existiert ein bekanntes Problem mit dem verwendeten Plugin. Dieses verhindert, dass Tests fehlschlagen und produziert somit falsch-positive Ergebnisse. Doch auch um dies zu beheben, werden einige Zeilen Code zur Verfügung gestellt, die zusätzlich in die bereits zuvor editierte Datei cypress.json eingefügt werden müssen, um die Werte der standardmäßigen Konfiguration anzupassen.

6.3.2 Nutzung

Um das gewählte Plugin zu analysieren, wird ein simpler Test konzipiert. Dieser soll prüfen, ob die Kopfzeile des Demonstrationsobjekts über jeder Seite gleich dargestellt wird.

Während der ersten Ausführung der Testmethode wird ein initialer Snapshot des angegebenen Seitenelements als sogenannte Baseline (dt. Referenz) erstellt. Hierbei werden keine prüfenden Testschritte ausgeführt, welche fehlschlagen könnten. Der erste Testlauf ist daher stets erfolgreich. Bei einer erneuten Ausführung eines Tests, für den bereits ein Referenz-Snapshot vorliegt, werden die tatsächlichen Testschritte ausgeführt. Im Fall des beschriebenen Tests wird erst die Recipes-Seite aufgerufen, ein Snapshot der Kopfzeile angelegt und dieser mit dem Referenz-Snapshot verglichen. Stimmen diese miteinander überein, wird erst zu der Ingredients-Seite und abschließend zur About-Seite gewechselt und die Prüfungsschritte wiederholt. Werden alle Schritte fehlerfrei durchgeführt, ist der Test erfolgreich abgeschlossen.

Werden Abweichungen vom Referenz-Snapshot festgestellt, schlägt der Test fehl, sofern die Sensibilität den gewählten Grenzwert von 0,001 überschreitet. In diesem Fall wird eine Schaltfläche mit dem Schriftzug “Compare Snapshot” im Test Runner angezeigt.

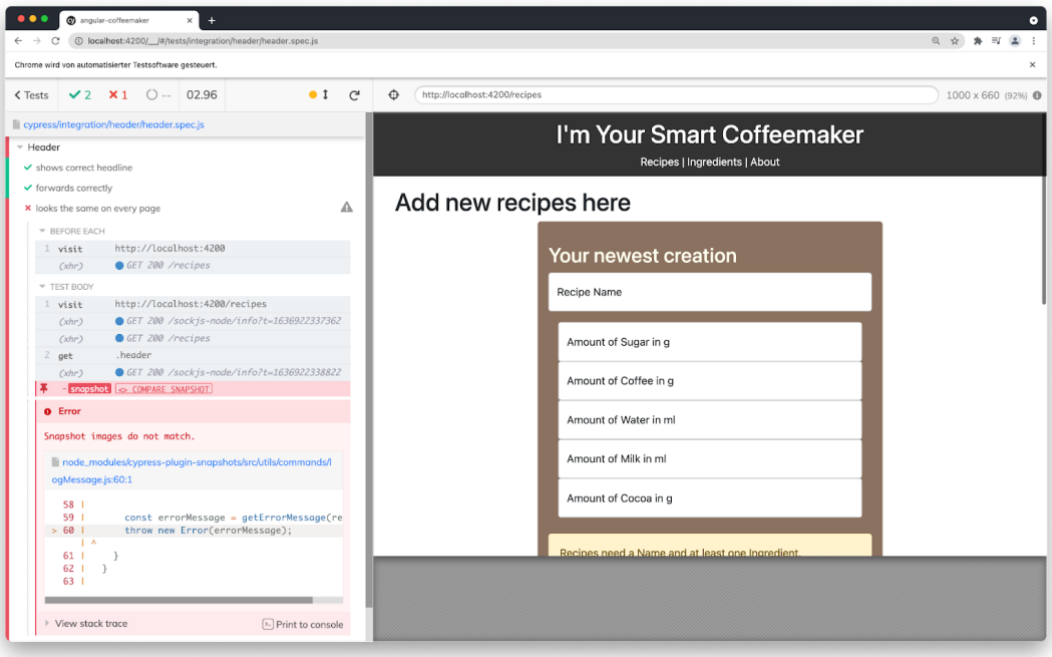

Abbildung 14: Screenshot – Fehlgeschlagener Test mit Schriftzug "Compare Snapshot" des cypress-plugin-snapshots.

Klickt man in der Fehlschlagmeldung die entsprechende Schaltfläche, wird ein neues Fenster mit einer Vergleichstabelle geöffnet. Hier wird links das “Expected result” (dt. erwartete Ergebnis) und rechts das “Actual result” (dt. tatsächliche Ergebnis) angezeigt. Die von Abweichungen betroffenen Bereiche werden in beiden Bildern gelblich hinterlegt und erkannte Elemente rot markiert.

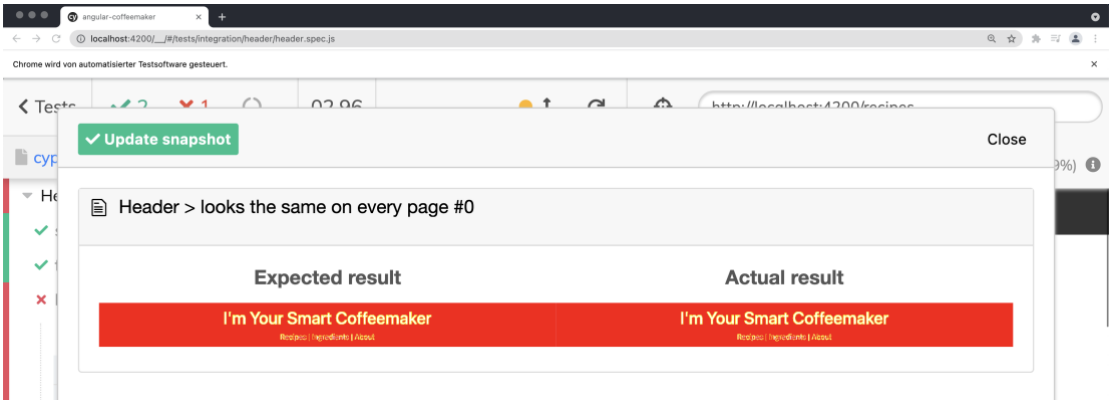

Abbildung 15: Screenshot - Vergleich der Snapshots durch cypress-plugin-snapshots.

6.3.3 Konfigurieren der Präzision

Maßgeblichen Einfluss auf die Qualität der geschriebenen visuellen Tests hat der sogenannte threshold-Wert. Direkt nach der Installation des Plugins ist dieser von Werk aus auf 0.01 gesetzt. Derselbe Wert ist in der Dokumentation des Plugins zu finden. Dort wird der Wert wie folgt beschrieben: “Amount in pixels or percentage before snapshot image is invalid" (van Straalen, 2020). Hierbei handelt es sich also um einen Toleranzwert, der bestimmt, um wie viele Pixel oder Prozent der vorgefundene Zustand des Seitenelements von dem zuvor erstellten Referenz-Snapshot abweichen darf, bevor ein Fehlerzustand erkannt wird. Für die Erkennung vieler getesteter Abweichungen ist der Standardwert nicht ausreichend. Hierbei handelt es sich u.A. um Fälle, die für das menschliche Auge offensichtlich, und somit für das GUI-Testing relevant sind. Dies wird im Folgenden durch simple Tests belegt. Hierzu wird der Snapshot-Abgleich durch verschiedene simulierte Tippfehler auf seine Zuverlässigkeit geprüft. Zu diesem Zweck wird das Wort “**About**” in der Kopfzeile des Demonstrationsobjekts genutzt. Die fehlerhafte Schreibweise “**Aboot**” stellte hierbei, wie für das menschliche Auge, so offenbar auch für den Algorithmus des Plugins, die visuell unauffälligste Abweichung dar. Hingegen waren die Varianten “**Aboxt**” und “**Abott**” leichter festzustellen. Durch eine schrittweise Verringerung des Grenzwertes um den Faktor 10 konnten die Ergebnisse leicht nachvollziehbar verbessert werden:

- Der Ausgangswert für threshold ist 0.01. Hier wird der Unterschied vom korrekten **u** zu einem fehlerhaften **t** zwar erkannt, jene zu einem **x** oder einem **o** jedoch nicht.
- Um die Präzision zu verbessern, wird der threshold-Wert anschließend auf 0.001 verringert. Hier wird der simulierte Tippfehler “Aboxt” nun korrekt erkannt. Der weniger offensichtliche Fehler “Aboot” wird weiterhin übersehen.
- Erst ab einem threshold von 0.0001 werden alle getesteten Unterschiede korrekt als Fehler gemeldet.

	Abo**o**t (unauffällig)	Abo**x**t (mittel)	Abo**t**t (auffällig)	About (korrekt)
0.01 (grob)	falsch-positiv	falsch-positiv	richtig-negativ	richtig-positiv
0.001 (mittel)	falsch-positiv	richtig-negativ	richtig-negativ	richtig-positiv
0.0001 (präzise)	richtig-negativ	richtig-negativ	richtig-negativ	richtig-positiv

Abbildung 16: Tabelle - Findung eines Tippfehlers durch cypress-plugin-snapshots.

Die beschriebenen Ergebnisse zeigen, dass der threshold-Wert stets an die Anforderungen der zu testenden Anwendung angepasst werden sollte, um die gewünschte Präzision zu erzielen. Die gefundenen Ungenauigkeiten beziehen sich ausschließlich auf die Kopfzeile des Demonstrationsobjekts. Bereiche mit größeren oder kleineren Pixelmengen können mehr oder weniger stark von niedrigen Grenzwerten profitieren.

6.4 Testen der Geschäftslogik

In diesem Kapitel sollen automatisierte Tests der Geschäftslogik beschrieben werden. Diese interagieren mit der Präsentationsschicht und lösen so Kommunikationen zur Logik- und Persistenzschicht aus. Schließlich überprüfen sie die Präsentationsschicht auf die erwarteten Effekte. Somit handelt es sich bei den folgenden Testszenarien um E2E-Tests (s. Kapitel 2.1.6).

6.4.1 Testen von Veränderungen

Zunächst soll das Anlegen eines neuen Rezeptes getestet werden. Als erstes wird vorgegeben, wie der Name des anzulegenden Rezeptes lauten soll, und welche Zutaten in welchen Mengen benötigt werden. Daraufhin wird geprüft, dass der Knopf zum Übermitteln der Daten aktiviert ist. Bis zu diesem Schritt sind alle Techniken bereits aus dem Kapitel 6.2 bekannt.

Ein für den Benutzer sichtbarer Hinweis darauf, dass die Abläufe im Backend der Anwendung erfolgreich stattgefunden haben, besteht in der Auflistung des neuen Rezeptes auf der Recipes-Seite. Hierzu wird getestet, ob das neue Rezept auf der Seite gefunden werden kann. Ist dies

der Fall, wird überprüft, ob die Überschrift des entsprechenden Seitenobjektes mit dem gewählten Rezeptnamen übereinstimmt. Außerdem gehören zu jedem Rezept die korrekten Zutatenmengen. Neben den Mengen, welche aktiv für einige Zutaten angegeben wurden, sollen auch jene Zutaten überprüft werden, für welche keine Mengen eingetragen wurden. Bei diesen könnte ein Fehler vorliegen, sollte nicht der Standardwert 0 für diese eingetragen werden.

```
it('Values saved correctly in newly created recipe', () => {
  cy.get('#name').type('Lungo')
    .get('#coffeeAmount').type(8)
    .get('#waterAmount').type(10)
    .get('#submit-new-recipe').should('be.enabled').click();

  cy.get('#recipe-Lungo > h2')
    .should('contain.text', 'Lungo:');
  cy.get('#recipe-Lungo > table > tr:nth-child(1) > td:nth-child(2)')
    .should('contain', '8');
  cy.get('#recipe-Lungo > table > tr:nth-child(2) > td:nth-child(2)')
    .should('contain', '10');
  cy.get('#recipe-Lungo > table > tr:nth-child(3) > td:nth-child(2)')
    .should('contain', '0');
  cy.get('#recipe-Lungo > table > tr:nth-child(4) > td:nth-child(2)')
    .should('contain', '0');
})
```

In diesem Beispiel wird nach dem neuen Rezept anhand der ID “recipe-<name>” gesucht, wobei <name> hier für den zuvor gewählten Rezeptnamen steht. Diese dynamische Erzeugung von IDs ist nicht in jeder Anwendung gegeben und muss von den Entwicklern aktiv implementiert werden. Sollten die Frontend-Entwickler nicht selbst von diesem Verhalten profitieren, kann es notwendig sein, dass Tester diese Form von Unterstützung explizit anfragen. Es ist also durchaus möglich, dass diese Zugänglichkeit neu erzeugter Seitenobjekte in vielen Teams nicht gegeben ist. Aus diesem Grund kann es hilfreich und ggf. zuverlässiger sein, Schaltflächen anhand von enthaltenem Text zu suchen. Ein Beispiel für dieses Vorgehen mit Hilfe der Methode *contains()* ist am Ende von Kapitel 6.4.2 zu finden.

Unabhängig davon, ob eines der standardmäßig vorhandenen Rezepte getestet werden soll, oder ein manuell neu angelegtes, gehört zu den unbedingt zu testenden Kernfunktionen des

Demonstrationsobjekts das Zubereiten ebendieser Rezepte. Für die Tests dieser Ausarbeitung wird für alle Zutaten eine Ausgangsfüllmenge von 10 der jeweiligen Maßeinheit festgelegt. Wie diese Füllstände zu jeder Zeit abgefragt werden können, wird in Kapitel 6.6.3 beschrieben. Soll ein Rezept getestet werden, welches weniger als 10 Einheiten einer Zutat benötigt, sollte dies problemlos gelingen. Soll hingegen ein Rezept zubereitet werden, welches die vorhandene Menge einer oder mehrerer Zutaten überschreitet, so sollte dieser Versuch fehlschlagen.

Um die Veränderung der Füllstände möglichst realistisch zu testen, sollen zwei Rezepte hintereinander zubereitet werden, deren Zutaten sich überschneiden. Dieser Versuch sollte fehlschlagen. Nach der Zubereitung des ersten Rezeptes “Americano” mit 10ml Wasser sollte keines mehr für das zweite Rezept „Latte Macchiato“ übrig sein.

Die Prüfung, ob ein Rezept korrekt zubereitet wurde, soll in dieser Betrachtung durch einen Abgleich der Zutatenfüllstände erfolgen. Befinden sich diese nach der versuchten Zubereitung auf den erwarteten Werten, so war die Zubereitung erfolgreich.

```
it('Brewing original recipe uses ingredients correctly', () => {
  cy.get('#recipe-Americano > #brewButton').click();
  cy.wait(500).visit(Cypress.env('appUrl_Ingredients'));

  cy.get('#form-Coffee > .form-label')
    .should('contain.text', 'Coffee: 5g');
  cy.get('#form-Water > .form-label')
    .should('contain.text', 'Water: 0ml');
  cy.get('#form-Milk > .form-label')
    .should('contain.text', 'Milk: 10ml');
  cy.get('#form-Cocoa > .form-label')
    .should('contain.text', 'Cocoa: 10g');
})
```

Eine weitere Prüfung könnte vorgenommen werden, indem, nach dem Klick auf den Knopf zum Zubereiten eines zu testenden Rezeptes, die browserseitigen Benachrichtigungen abgefangen werden. Diese könnten dann darauf geprüft werden, ob sie den Benutzer über Erfolg oder Fehlschlag informieren. Derartige Tests werden im Folgekapitel behandelt.

Wird die erfolgreiche Durchführung also durch Prüfung der Füllwerte festgestellt, soll als nächstes getestet werden, dass die Zubereitung eines "Latte Macchiato" direkt nach dem "Americano" blockiert wird, und sich die Zutatenmengen nicht verändern. Betrachten wir zunächst die theoretisch testbaren Ergebnisfüllstände nach diesem Zubereitungsversuch:

- **Wasser:** Da beide Rezepte je 10ml Wasser benötigen, stellt diese Zutat den blockierenden Faktor für den zweiten Zubereitungsprozess dar. Der korrekte Wasserfüllstand, sowohl nach der erfolgreichen ersten als auch nach der blockierten zweiten Rezeptzubereitung beträgt somit also 0ml.
- **Kaffee:** Auch in der benötigten Kaffeemenge stimmen die zwei gewählten Rezepte überein. Beide verwenden jeweils 5g Kaffee. Diese Zutat allein führt also nicht dazu, dass die Zubereitung des zweiten Rezeptes blockiert wird. Wird die zweite Zubereitung aufgrund fehlerhafter Implementierung nicht durch den Mangel an Wasser blockiert, könnte der Füllstand des Kaffees auf 0g reduziert werden. Wird die Zubereitung jedoch korrekt blockiert, wird ein übriger Füllstand von 5g erwartet.
- **Milch:** Die Milchvorräte werden von der Americano-Zubereitung nicht berührt, vom Latte Macchiato hingegen schon. Wird die Zubereitung korrekt blockiert, so sollte sich der Füllstand nicht verändern. Im Fehlerfall einer erfolgten Zubereitung hingegen würde er von 10 auf 5ml verringert.
- **Kakao:** Diese Zutat wird als einzige von keinem der beiden getesteten Rezepte verwendet. Der zugehörige Füllstand sollte also unabhängig von Erfolg oder Fehlschlag auf dem Ausgangswert von 10g verbleiben.

Nachdem die Werte auf der Ingredients-Seite vollständig ausgelesen wurden, sollen hier explizit keine Änderungen vorgenommen werden. Es wird also zurück auf die Recipes-Seite der Anwendung gewechselt. Hierfür bietet sich die Nutzung der Navigationsmethode *go()* an. Dieser wird hier der Parameter 'back' übergeben, um im Browser eine Seite zurückzugehen. Hier wird die Schaltfläche zum Zubereiten eines Latte Macchiato betätigt. Anschließend wird erneut die Ingredients-Seite besucht. Auch hier kommt *go()* zum Einsatz; dieses Mal jedoch mit dem Parameter 'forward', um den Browser eine Seite vorwärts navigieren zu lassen. Haben

sich die Zutatenmengen nicht verändert, wurde die Zubereitung des zweiten Rezeptes korrekt blockiert und der Test endet erfolgreich.

```
it('Brewing second recipe is blocked by insufficient ingredients', () => {
  cy.get('#recipe-Americano > #brewButton').click();
  cy.wait(500).visit(Cypress.env('appUrl_Ingredients'));

  cy.get('#form-Coffee > .form-label')
    .should('contain.text', 'Coffee: 5g');
  cy.get('#form-Water > .form-label')
    .should('contain.text', 'Water: 0ml');
  cy.get('#form-Milk > .form-label')
    .should('contain.text', 'Milk: 10ml');
  cy.get('#form-Cocoa > .form-label')
    .should('contain.text', 'Cocoa: 10g');

  cy.go('back');
  cy.get('#recipe-Latte\\ Macchiato > #brewButton').click();
  cy.wait(500).go('forward');

  cy.get('#form-Coffee > .form-label')
    .should('contain.text', 'Coffee: 5g');
  cy.get('#form-Water > .form-label')
    .should('contain.text', 'Water: 0ml');
  cy.get('#form-Milk > .form-label')
    .should('contain.text', 'Milk: 10ml');
  cy.get('#form-Cocoa > .form-label')
    .should('contain.text', 'Cocoa: 10g');
})
```

6.4.2 Testen von Benachrichtigungen

Zuvor wurde getestet, dass ein neues Rezept in der Liste vorhandener Rezepte auftaucht und korrekt zubereitet werden kann. Ebenso kann überprüft werden, ob Ereignisse auf der Benutzeroberfläche, erwartungsgemäße stattfinden. Ein Beispiel hierfür wird in dem vorliegenden Demonstrationsobjekt in Form von Browserbenachrichtigungen (engl. Alerts oder Notifications), implementiert (s. Kapitel 5.2.2).

Um besagte Benachrichtigungen zu testen, wird der Test aus dem vorhergegangenen Abschnitt wiederholt. Es wird also zunächst erfolgreich ein Americano und daraufhin ohne Erfolg ein Latte Macchiato zubereitet. In diesem Durchgang soll jedoch nicht erneut geprüft werden, dass sich die vorhandenen Zutatenmengen korrekt anpassen. Stattdessen wird kontrolliert, ob der Browser eine Benachrichtigung anzeigt, welche den Benutzer über Erfolg bzw. Fehlschlag informiert. Bei diesen Benachrichtigungen handelt es sich um besondere Elemente, welche nicht wie die zuvor getesteten Seitenelemente von Cypress auf unserer Website gesucht und gefunden werden können. Benachrichtigungen werden simpel ausgedrückt oberhalb der restlichen Benutzeroberfläche dargestellt und blockieren die Interaktion mit anderen Seitenelementen. Aus diesem Grund muss Cypress mit Hilfe von sogenannten Event Listenern (vgl. Cypress.io Events, 2021) aufgetragen werden, den Eintritt eines bestimmten Ereignisses zu erwarten. Sobald das angegebene Ereignis wahrgenommen wird, können beliebige Testschritte durchgeführt werden.

Dieses "Lauschen" auf bestimmte Ereignisse wird in Cypress mit der Methode *on()* in Kombination mit zwei Parametern ermöglicht (vgl. ebd.). Auf welches Ereignis gewartet werden soll, wird am Beispiel der im Demonstrationsobjekt genutzten Alerts mit *window:alert* als erstem Parameter der Methode *on()* definiert. Hierbei handelt es sich um die hier verwendeten Benachrichtigungen, welche dem Benutzer eine Information mitteilen und sichtbar bleiben, bis die einzige angebotene Schaltfläche "OK" betätigt wird.

Der zweite Parameter wird in Form einer Funktion übergeben und beschreibt die Testschritte, die beim Eintreffen des angegebenen Ereignisses durchgeführt werden sollen. Hier wird an die Funktion das gefundene Seitenelement als Variable übergeben, bei welchem es sich in diesem Fall um die zu prüfende Benachrichtigung handelt. Auf dieser Variablen werden anschließend die gewünschten Kontrollmethoden aufgerufen. Soll also bspw. die Benachrichtigung abgefangen werden, welche den Benutzer über die erfolgreiche Zubereitung des gewählten Rezeptes informiert, so könnte die erwartete Benachrichtigung als "successAlert" benannt werden. Der Aufruf sähe dann wie folgt aus:

```
cy.on('window:alert', (successAlert) => {
  <Testschritte>
});
```

Anschließend besteht der wichtigste Test darin, sicherzustellen, dass der korrekte Mitteilungstext angezeigt wird. Hierbei ist anzumerken, dass die bisher vorgestellten Kontrollmethoden aus den Kapiteln 2.3 und 6 innerhalb einer *on()*-Methode nicht genutzt werden können.

```
cy.on('window:alert', (successAlert) => {
  // Wird so von Cypress nicht erkannt:
  successAlert.should('contain.text', '<erwarteter Text>');
});
```

Aus diesem Grund muss an dieser Stelle die Methode *expect()* eingeführt werden (vgl. Mwaura, 2021, S.66). Diese ermöglicht in Verknüpfung mit der Verbindungsmethode *to*, Tests auf eine neue Art und Weise zu formulieren. Der hier benötigte Test des beinhalteten Textes lässt sich so mit Hilfe der Methode *contain()* und dem erwarteten Text als Parameter umsetzen.

```
cy.on('window:alert', (successAlert) => {
  expect(successAlert).to
    .contain('☕ Enjoy your freshly brewed Americano! ☕');
});
```

Aus der soeben beschriebenen Herangehensweise für das Testen von Browserbenachrichtigungen ergibt sich der folgende Test, wenn das erfolgreiche Zubereiten eines Americano getestet werden soll.

```
it('Original recipe can be brewn and triggers alert', () => {
  cy.get('#recipe-Americano')
    .contains('Brew Americano')
    .click();
  cy.on('window:alert', (successAlert) => {
    expect(successAlert).to
      .contain('☕ Enjoy your freshly brewed Americano! ☕');
  });
})
```

In diesem Fall wird der Knopf zum Zubereiten des Rezeptes vorab über den beinhalteten Text “Brew <Rezeptname>” gesucht. Dies wird mit Hilfe der Methode *contains()* mit dem erwarteten Text als Parameter umgesetzt.

6.5 Unit-Tests

In Kapitel 4.3 wird bereits erwähnt, dass Unit-Tests nicht zu den ausgewiesenen Kernkompetenzen von Cypress zählen. Daher wird festgelegt, dass es für den Umfang dieser Ausarbeitung genügen soll, die Geschäftslogik mit Hilfe von Blackbox-Tests abzudecken. Da es jedoch grundsätzlich möglich ist, Unit-Tests mit Cypress umzusetzen, und diese einen Mehrwert darstellen könnten, wurde der Versuch unternommen, den Testumfang um entsprechende Whitebox-Tests zu erweitern. Ein Teil dieses Experiments bestand aus dem Arbeiten mit den Methoden *cy.stub()* und *cy.spy()* (vgl. Cypress.io Stubs, 2021). Diese werden genutzt, um isolierte Instanzen der Programmkomponenten zu erstellen, welche anschließend getestet werden können. Dies stellte sich schnell als relativ komplexes Vorgehen heraus. In der für diesen Versuch eingeplanten Zeit konnte für die Komponenten des vorliegenden Demonstrationsobjekts kein lauffähiges Ergebnis produziert werden.

Da die Einstiegshürde an dieser Stelle im Vergleich zu den anderen in dieser Ausarbeitung beschriebenen Testmethoden als unverhältnismäßig hoch im Vergleich zum erwünschten Ergebnis empfunden wird, wird festgelegt, dass die Umsetzung von Unit-Tests unter Cypress für den Umfang dieser Ausarbeitung ungeeignet ist. Ob dies eine relevante Schwäche für das schichtenübergreifende Testen unter Cypress darstellt, wird in Kapitel 0 evaluiert.

6.6 Datenbanktests

Wie in den vergangenen Kapiteln 6.2 bis 6.4 beschrieben, werden die Funktionalitäten des Demonstrationsobjekts in den vergangenen Abschnitten anhand auf der Benutzeroberfläche dargestellter Schaltflächen und angezeigter Werte getestet. Im Folgenden werden einige Möglichkeiten erläutert, Datenbanken unter Cypress zu testen, um einen direkten Blick in die Persistenzschicht zu erhalten. Vorab wird die Cypress-Methode vorgestellt, welche den Kern der Datenbanktests darstellt. Anschließend wird darauf eingegangen, wie konsistente Daten vor und nach der Ausführung von Tests sichergestellt werden können.

6.6.1 Verwendung der Methode cy.request()

Um Werte aus einer zu testenden Datenbank abzufragen und auf Korrektheit zu prüfen, wird von Cypress eine Methode für das Versenden von HTTP-Anfragen zur Verfügung gestellt. Diese können u.A. zum Überschreiben von Daten genutzt werden. Somit wird gleichzeitig eine zugängliche Art und Weise geboten, das Zurücksetzen der verwendeten Datenbank vor oder nach einem Test zu handhaben. Dieser Anwendungsfall wird im folgenden Kapitel 6.6.2 beschrieben. Bei besagter Methode handelt es sich um *cy.request()*. Dieser müssen je nach Verwendungszweck ein bis drei Parameter übergeben werden:

- **method:** Der erste Parameter gibt an, welche HTTP-Methode verwendet werden soll. In der erarbeiteten Testautomatisierung werden hauptsächlich die Abfrageoperation GET und die Schreiboperation PUT verwendet.
- **url:** Bei dem zweiten Parameter handelt es sich um die URL, welche angesteuert werden soll.
- **body:** Bei Anfragen, zu welchen ein zugehöriger Datensatz erwartet wird, wird dieser als dritter Parameter übergeben. Dies geschieht in Form eines json-Objektes, welches den Körper oder auch Inhalt der zu sendenden Anfrage beschreibt.

In den nachfolgenden Abschnitten werden die verschiedenen Szenarien erläutert, in welchen die beschriebene Cypress-Methode *cy.request()* in Kombination mit verschiedenen REST-Methoden Anwendung findet.

6.6.2 Wiederherstellen eines Ausgangszustandes

Cypress bietet die Möglichkeit, gewünschte Zustände der verwendeten Datenbank als sogenannte Fixtures zu hinterlegen. Diese werden als json-Dateien in dem Projektverzeichnis angelegt und bei Bedarf aufgerufen, um den ursprünglich gespeicherten Stand der Werte in der Datenbank wiederherzustellen. Diese empfohlene Herangehensweise konnte während der Testautomatisierung dieser Ausarbeitung nicht erfolgreich umgesetzt werden. Aus diesem Grund wird eine andere Methode verwendet: Mit der Methode *cy.request()*, deren Syntax in Kapitel 6.6.1 genauer beschrieben wird, kann das Einspielen vordefinierter Werte in eine

bestehende Datenbank auf eine simple Art und Weise durchgeführt werden. Im folgenden Codeausschnitt soll die Methode zum Zurücksetzen der Zutatenfüllstände demonstriert werden.

Wie in Kapitel 5.4 beschrieben, nutzt das entwickelte Demonstrationsobjekt eine Datenbank, welche mit Hilfe des Tools json-server verwaltet wird. Dieses Tool bietet unter einer eigenen URL, welche separat von der Anwendung betrieben wird, Zugriff auf die Datenbank und die Werte, die sie beinhaltet. Bei dieser URL handelt es sich standardmäßig um die im Folgenden verwendete Adresse localhost:3000.

```
Cypress.Commands.add('resetIngredientsTable', () => {
  let ingredientsUrl = "http://localhost:3000/ingredients";
  cy.request('PUT', ingredientsUrl + '/1', default_coffee);
  cy.request('PUT', ingredientsUrl + '/2', default_water);
  cy.request('PUT', ingredientsUrl + '/3', default_milk);
  cy.request('PUT', ingredientsUrl + '/4', default_cocoa);
});
```

Bei der Implementierung der Methode resetIngredientsTable fällt auf, dass es sich nicht um eine Testmethode handelt, wie sie in den Kapiteln 6.2 bis 6.4 stets zu sehen waren. Insbesondere sticht die erste Zeile, der Methodenkopf *Cypress.Commands.add*, ins Auge. Dies ist dadurch begründet, dass sich diese Methodendefinition nicht in einer Testdatei befindet. Stattdessen liegt sie in der Datei commands.js. Diese Datei ist für das Verfassen von Funktionen vorgesehen, welche über alle Testdateien hinweg verfügbar gemacht werden sollen. In dem ersten Parameter der Methode *Cypress.Commands.add()* wird der Methodenname festgelegt, unter welchem diese zukünftig aufgerufen werden kann. Bei dem zweiten Parameter handelt es sich um die Funktion selbst, d.h. die Schritte, welche beim Aufruf ausgeführt werden sollen. Diese Art der Implementierung von global nutzbaren Methoden führt dazu, dass die hier definierte Abfolge von Anfragen für alle Tests zur Verfügung steht.

Der Aufruf der soeben definierten Funktion sollte in jeder Testdatei, welche entsprechende Testschritte beinhaltet, in einer oder mehreren der Methoden *before()*, *beforeEach()*, *after()* und *afterEach()* konfiguriert werden. Diese sorgen jeweils entsprechend ihres jeweiligen Methodennamens dafür, dass das implementierte Zurücksetzen des Datenbankzustandes, nebst

weiteren beinhalteten Aufrufen, entweder vor allen, vor jedem, nach allen oder nach jedem Test der betroffenen Testdatei ausgeführt wird. Welche dieser Vorgehensweisen oder ihrer möglichen Kombinationen am sinnvollsten ist, kann von Test zu Test entschieden werden. Der folgende Codeausschnitt zeigt die beim Testen der Zutaten verwendete Kombination.

```
beforeEach(() => {
  cy.resetIngredientsTable();
  cy.visit('http://localhost:4200/ingredients');
})
after(() => {
  cy.resetIngredientsTable();
})
```

Bei der Testautomatisierung der Ingredients-Seite werden vor jedem einzelnen Test sowie nach der vollendeten Ausführung aller Tests die Zutatenmengen zurückgesetzt. Somit kann zu jedem Zeitpunkt vor oder nach der Ausführung eines jeden Tests die Sicherheit geboten werden, dass sich die Werte innerhalb der Datenbank stets auf dem vom Tester erwarteten Stand befinden. Dies gilt sowohl durch die Nutzung von *beforeEach()* für die Tests innerhalb der betroffenen Testdatei, als auch durch das Hinzufügen von *after()* für die Tests anderer Testdateien, welche auf die Ausführung der aktuellen folgen können.

Außerdem wird die *beforeEach()*-Methode genutzt, um vor jedem Test die Ingredients-Seite aufrufen zu lassen. Hierdurch wird ein Aufruf zu Beginn jeder Testmethode vermieden.

6.6.3 Abfragen des aktuellen Status der Datenhaltung

Das tatsächliche Testen von erwarteten Werten in der Datenbank gestaltet sich nach demselben Schema wie die meisten anderen Tests:

1. Als erstes kann der aktuelle Stand des Wertes abgefragt werden, um sicherzustellen, dass er sich wie erwartet auf dem stets wiederherzustellenden Ausgangswert befindet.

```
/* Default values - Test via request */
it('Default amount is pre-set correctly in DB', () => {
  cy.request('GET', Cypress.env('dbUrl_Ingredients') + '/1')
    .then(response => {
```

```
      expect(response.status).to.eq(200);
      expect(response.body).to.have.property('name', 'Coffee');
      expect(response.body).to.have.property('amount', 10);
    });
})
```

2. Anschließend wird eine Aktion durchgeführt, welche den Zustand des zu testenden Demonstrationsobjekts verändert.

A: Befindet sich der zu entwickelnde Test im Kontext eines Integrations- oder Systemtests, so können Aktionen mit Einfluss auf die Datenbankwerte auf der Benutzeroberfläche durchgeführt werden, sofern diese zum Testzeitpunkt zur Verfügung steht.

```
/* Change values via GUI - Test via request */
it('Ingredient amount is updated after GUI interaction', () => {
  cy.get('#amount-Water').type(1);
  cy.get('#button-Water').should('be.enabled').click();
  cy.request('GET', Cypress.env('dbUrl_Ingredients') + '/2')
    .then(response => {
      expect(response.body).to.have.property('name', 'Water');
      expect(response.body).to.have.property('amount', 11);
    });
})
```

B: Soll hingegen die Persistenzschicht losgelöst von anderen Komponenten getestet werden, so kann mit Hilfe von PUT-Anfragen direkter Einfluss auf die Datenhaltung genommen werden.

```
/* Change values via request - Test via request */
it('Ingredient amount is updated after PUT request', () => {
  cy.request('PUT', Cypress.env('dbUrl_Ingredients') + '/3', empty_milk);
  cy.request('GET', Cypress.env('dbUrl_Ingredients') + '/3')
    .then(response => {
      expect(response.body).to.have.property('name', 'Milk');
      expect(response.body).to.have.property('amount', 0);
    });
})
const empty_milk = {
```

```
    "id": 3,
    "name": "Milk",
    "unit": "ml",
    "amount": 0
  }
```

Durch diese verschiedenen Vorgehensweisen, auf Datenbankinhalte zuzugreifen - entweder indirekt durch die Benutzeroberfläche oder direkt durch REST-Abfragen - kann zu jedem Zeitpunkt in der Softwareentwicklung das Testen der Persistenzschicht an die verfügbaren Schichten angepasst werden.

6.7 Testen der Erweiterbarkeit

Im Folgenden soll beschrieben werden, wie vorgegangen wird, um die Erweiterbarkeit der unter Cypress entwickelten Testautomatisierung zu analysieren. Wie in Kapitel 3.2.4 beschrieben, ist eine unkomplizierte Anpassbarkeit der entwickelten Tests v.a. in schichtenübergreifenden Projekten wichtig, da jederzeit von verschiedenen Teams Änderungen an bestehenden Komponenten vorgenommen werden können. Das Ziel dieses Kapitels besteht darin, zu erarbeiten, welche Anpassungen nach einer beispielhaften Veränderung des Demonstrationsobjekts an der bestehenden Testautomatisierung vorgenommen werden müssen, um die veränderten Aspekte angemessen zu testen. Je schneller eine Testbasis an Veränderungen angepasst werden kann, desto eher können ggf. entstandene fehlerhafte Tests korrigiert, eine ausreichende Testabdeckung erreicht und neue Fehler gefunden werden. Aus diesem Grund wird die im Folgenden getestete Flexibilität des Testframeworks sich auf die Bewertung auswirken, inwiefern Cypress für die teamübergreifende Verwendung geeignet ist.

Ebenso wird betrachtet, ob bestehende Tests fehlschlagen, sobald eine Änderung im Demonstrationsobjekt implementiert ist. Dies könnte auf eine geringe Robustheit von Cypress gegenüber sich ändernden Testobjekten hinweisen.

Es ist zu betonen, dass diese Analyse ausschließlich als ein Beispiel stehen kann. Eine Änderung an den Funktionen einer bestehenden Anwendung kann im jeweiligen Anwendungsfall zu variierenden Auswirkungen führen. Die hier gewählte beispielhafte Änderung kann somit in keinem Fall alle denkbaren Szenarien abbilden. Das nachfolgend

beschriebene Vorgehen soll lediglich eine realitätsnahe Situation im Entwicklungsprozess simulieren.

6.7.1 Funktionserweiterung

In Kapitel 4.4 wurden die Anforderungen an das in diesem Abschnitt zu wählende Vorgehen grob konzeptioniert. Unter anderem wurde festgelegt, dass die Funktionsänderung alle drei Softwareschichten des Demonstrationsobjekts beeinflussen soll. Um diese Vorgabe zu erfüllen, wird die vorzunehmende Änderung wie folgt definiert:

Zu den vier ursprünglich implementierten Zutaten soll eine fünfte hinzugefügt werden. Diese muss an allen Orten korrekt dargestellt werden, wo auch die ursprünglichen vier Zutaten sichtbar sind. Ihr Füllstand soll analog zu den bestehenden Zutaten beeinflusst werden können. Sie soll dieselben Minimal- und Maximalfüllstände besitzen und gleichwertig in Rezepten verwendet werden können.

Diese Änderung nimmt Einfluss auf die Präsentationsschicht, da mehrere Oberflächen angepasst werden müssen. Die Geschäftslogik wird beeinflusst, da neue Kontrollen eingeführt werden müssen, um die korrekte Zubereitung von Rezepten zu gewährleisten. In der Persistenzschicht muss letztlich ein weiterer Datensatz angelegt werden, welcher die Informationen über die neue Zutat bereithält. Des Weiteren müssen alle Rezepte nun einen neuen Wert beinhalten, welcher angibt, welche Menge der neuen Zutat für dieses benötigt wird. Somit werden durch diese Änderung erfolgreich alle Schichten beeinflusst.

6.7.2 Provozierte Fehlerwirkung

Nach der erfolgreichen Implementierung der Funktionserweiterung der zu testenden Webanwendung schlagen zunächst keine Tests fehl. Wenngleich dies einen positiven ersten Eindruck bezüglich der geschriebenen Tests und der Robustheit von Cypress hinterlässt, scheint die Komplexität der durchgeführten Änderung relativ trivial. Es werden bestehende Elemente der Webseite um ein Element erweitert, bestehende Strukturen und dargestellte Reihenfolgen jedoch nicht verändert. Aus diesem Grund werden die Oberflächen der Webanwendung anschließend so angepasst, dass die neue Zutat in Auflistungen nicht mehr - wie zuerst implementiert - den letzten bzw. untersten Platz, sondern den ersten bzw. obersten

Platz einnimmt. Hierdurch wird das Erscheinungsbild der Webanwendung stärker verändert. Dies soll die Wahrscheinlichkeit erhöhen, dass Tests, welche anfällig für Veränderungen auf der Benutzeroberfläche sind, fehlschlagen. Tatsächlich führt die beschriebene zweite Iteration dazu, dass ein zuvor positiver Test nun ein negatives Ergebnis zurückliefert. Somit kann ein negativer Einfluss der gewählten Funktionserweiterung nachgewiesen werden. Dies hat den gewünschten Effekt: Es gilt einen Fehler zu analysieren, eine Lösung für den provozierten Fehler zu finden, und den benötigten Aufwand zu bewerten.

6.7.3 Fehleranalyse

Bei dem Test, welcher nach der Funktionsänderung fehlschlägt, handelt es sich um den Test *Values saved correctly in newly created recipe*, welcher in Kapitel 6.4.1 näher erläutert wird. Dieser legt zunächst ein neues Rezept an und prüft anschließend, dass die eingetragenen Werte korrekt in das neue Rezept übertragen werden.

Die Zutatenmengen werden auch nach dem Hinzufügen der neuen Zutat Zucker korrekt abgespeichert. Jedoch schlägt die Prüfung der Kaffeemenge fehl. Dies wird dadurch ausgelöst, dass für die Prüfung der zugewiesenen Kaffeemenge eine fehleranfällige Teststrategie gewählt wurde. Da es sich zum Zeitpunkt der Testimplementierung bei der ersten aufgelisteten Zutat stets um Kaffee handelte, wird geprüft, ob der erste in dem neuen Rezept eingetragene Wert der erwarteten Kaffeemenge entspricht:

```
cy.get('#recipe-Lungo > table > tr:nth-child(1) > td:nth-child(2)')
    .should('contain', '8');
```

Dieser Testschritt kann wie folgt beschrieben werden: „*Navigiere in das Rezept Lungo und prüfe in der beinhalteten Tabelle* ***die zweite Spalte der ersten Zeile*** *auf den Wert 8.“*

Während der Funktionserweiterung wird der erste Platz in Auflistungen durch die neue Zutat Zucker ersetzt. Somit befindet sich die Kaffeemenge nun an zweiter Stelle und der bestehende Test schlägt fehl. Dies ist im folgenden Screenshot zu sehen.

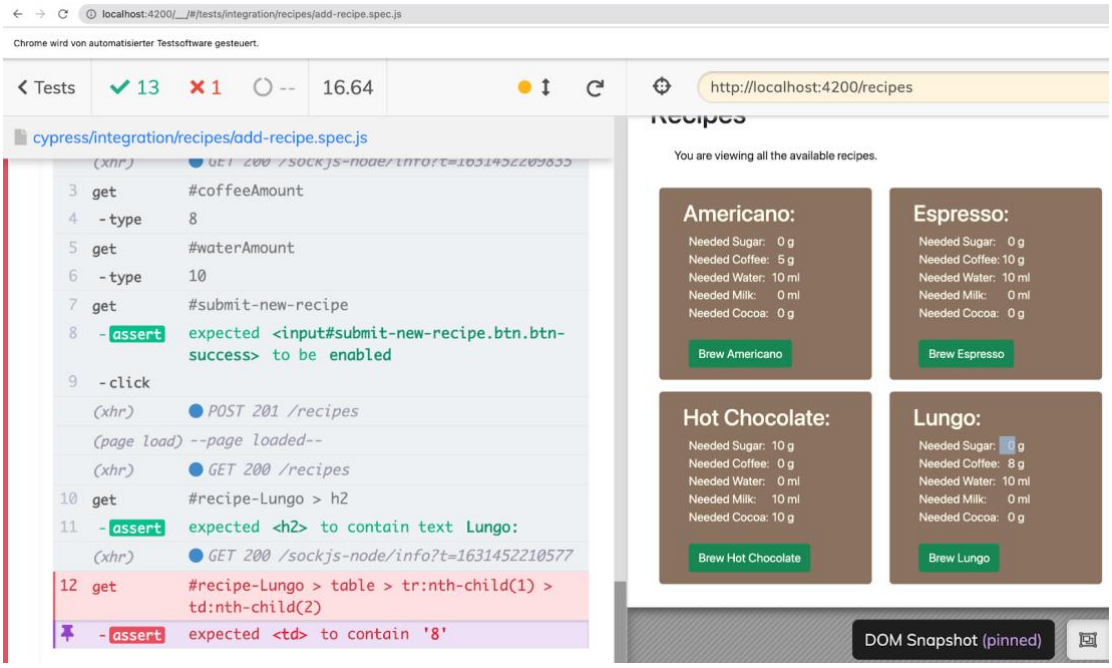

Abbildung 17: Screenshot - Fälschliche Mengenprüfung von Zucker statt Kaffee.

Im Testablauf links ist sichtbar, dass einige Schritte erfolgreich sind (graue und grüne Flächen), bevor die Prüfung auf den Wert 8 fehlschlägt (rote Flächen). Die Zelle, welche den getesteten Wert enthält, ist rechts hellblau markiert.

6.7.4 Korrektur des betroffenen Testschrittes

Das zuvor beschriebene Fehlschlagen des Tests kann vermieden werden, wird statt der absoluten Abfrage nach Zeilennummern eine relative Prüfung nach benachbarten Werten verwendet. Diese könnte wie folgt formuliert werden: *„Navigiere in das Rezept Lungo und prüfe in der beinhalteten Tabelle* ***alle benachbarten Spalten jener Zelle, in welcher "Coffee" steht,*** *auf den Wert 8."* Diese Anforderung konnte auf folgende Art und Weise umgesetzt werden:

```
cy.get('#recipe-Lungo')
  .contains('td', 'Coffee')
  .siblings()
  .should('contain', '8');
```

Zunächst wird wie zuvor auf die neu erstellte Rezeptschaltfläche zugegriffen. Innerhalb dieser wird die Zelle gesucht, welche “Coffee” enthält. Anschließend wird mit Hilfe der Methode *siblings()* eine Liste der benachbarten Zellen generiert. Innerhalb dieser kann schließlich nach dem erwarteten Wert gesucht werden. Die behandelte Testmethode wird nun wieder erfolgreich ausgeführt. Die ursprünglich formulierte Abfrage (siehe oben) wurde leicht abgeändert. Es wird nun nicht mehr präzise nach der zweiten Spalte innerhalb der gewünschten Zeile gefragt. Stattdessen werden alle Zellen der Zeile untersucht.

Mit der Anpassung dieses Tests sind die Arbeiten zur Korrektur fehlschlagender Tests nach der Implementierung der Funktionsänderung abgeschlossen.

6.7.5 Erweiterung der bestehenden Tests

Nach der vollständigen Implementierung der neuen Zutat sollen die bestehenden Tests erweitert werden. Diese Erweiterung soll die neu hinzugefügten Programmteile abdecken. Hierdurch soll die Testabdeckung lediglich auf dasselbe Niveau gehoben werden, welches vor der Funktionsänderung bestand. Eine möglichst hohe Testabdeckung zu erzielen ist weiterhin kein Ziel dieser Ausarbeitung. Das Ziel dieses Vorgehens besteht ausschließlich darin, festzustellen, ob die Erweiterung der bestehenden Tests mit unverhältnismäßig hohem Arbeitsaufwand verbunden ist. Wäre dies der Fall, würde sich dies negativ auf die Evaluation in Kapitel 0 auswirken.

Die Erweiterungen der bestehenden Tests gestalteten sich im Kontext der vorliegenden automatisierten Tests sehr übersichtlich. Alle Tests, welche alle ursprünglichen Zutaten abdeckten, wurden um die neue Zutat erweitert. Hierüber hinaus sind keine nennenswerten Aufwände entstanden. Die bearbeiteten Tests konnten umgehend ebenso erfolgreich ausgeführt werden wie vor der Funktionserweiterung.

7 Evaluation

In diesem Kapitel soll evaluiert werden, in welchem Maße verschiedene Aspekte der Testautomatisierung unter Cypress positiv oder negativ zu einer Empfehlung für die teamübergreifende Nutzung bei schichtenübergreifenden Projekten beitragen. Hierzu wird Bezug auf die Erfahrungen genommen, welche während dieser Ausarbeitung gemacht wurden. Insbesondere wird analysiert, inwieweit die zuvor in Kapitel 3.2 formulierten Anforderungen an das Testautomatisierungsframework von diesem erfüllt werden konnten. Zu diesem Zweck werden zunächst die unterschiedlichen Testarten kurz betrachtet und ihre Signifikanz im Kontext eines schichtenübergreifenden Testkonzeptes gewichtet. Anschließend wird auf den Prozess der Implementierung der entsprechenden Tests zurückgeblickt und entsprechende Bewertungen vorgenommen. Das finale Fazit wird in Kapitel 8 zusammengefasst.

7.1 Testen der Präsentationsschicht

Wie in Kapitel 2.2.2 erläutert, handelt es sich bei Cypress um ein Testautomatisierungsframework, welches mit Fokus auf E2E-Tests entwickelt wurde. An dieser Stelle soll erneut auf die Abbildung aus Kapitel 2.1.6 verwiesen werden, die ein entsprechendes Szenario verbildlicht.

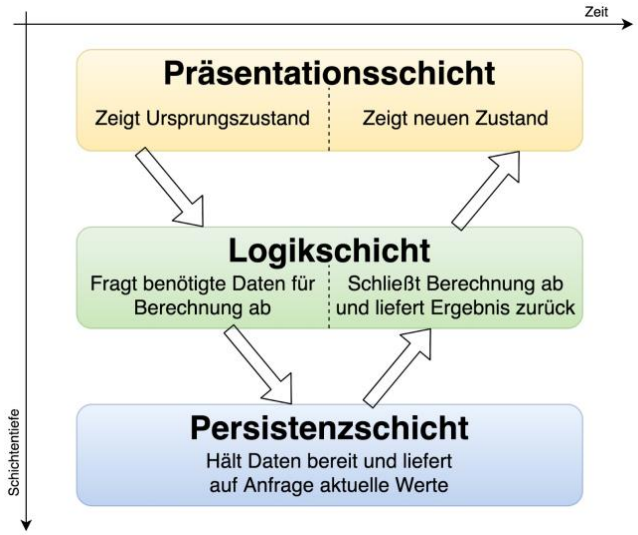

Abbildung 18: Grafik - Simplifizierte Verbildlichung des Ablaufes eines abstrakten E2E-Tests anhand der drei in dieser Ausarbeitung im Fokus stehenden Softwareschichten.

Aus diesem abstrakten Prozess wird ersichtlich, dass die Benutzeroberfläche sowohl zu Beginn eines E2E-Tests den Point of Control, als auch nach Abschluss des Programmablaufs denPoint of Observation (s. Kapitel 2.1.3) bietet. Da GUI-Tests somit den Grundstein für E2E-Tests legen, welche wiederum eine entscheidende Kompetenz für das schichtenübergreifende Testen darstellen, werden GUI-Tests in der finalen Bewertung am stärksten gewichtet.

7.1.1 Stärken

Betrachtet man die Methoden, welche Cypress zur Interaktion mit Benutzeroberflächen mit sich bringt, wird schnell klar, dass die meisten entscheidenden Aktionen von Werk aus zur Verfügung stehen. Während der Testautomatisierung dieser Ausarbeitung wurde nur selten ein Punkt erreicht, an dem eine Interaktion mit der Benutzeroberfläche vorgesehen war und keine passende Methode für die Umsetzung gefunden werden konnte (s. Kapitel 7.1.2).

Eine Stärke von Cypress Methoden für den Zugriff auf Oberflächenobjekte ist, dass für ähnliche Anforderungen mit feinen Unterschieden stets eine passende Methode oder Kombination mehrerer Methoden gefunden werden konnte. Dies soll im Folgenden kurz demonstriert werden. Für den wie folgt definierten Testschritt wurde untersucht, welche Methode hierfür am präzisesten geeignet ist:

"Navigiere in die Eingabemaske für <Zutat> und prüfe, dass das hierin enthaltene Label den Text "<Zutat>: 10" enthält."

Für diesen Testschritt wurden vier Ansätze gefunden, die in Frage gekommen sind. Die verschiedenen Zutaten werden hier lediglich zur leichteren Unterscheidung verwendet.

```
10  it('All ingredients show their current amount', () ⇒ {
11    cy.get('#form-Coffee').find('label').contains('Coffee: 10');
12    cy.get('#form-Water').find('label').should('contain.text', 'Water: 10');
13    cy.get('#form-Milk').get('label').should('contain.text', 'Milk: 10');
14    cy.get('#form-Cocoa').should('contain.text', 'Cocoa: 10');
15  })
```

Abbildung 19: Screenshot – Testcode inkl. Zeilennummerierung.

Die Methoden werden in Kapitel 2.3 bereits im Einzelnen erklärt. Die Codezeilen ähneln sich im Wortlaut ihrer Syntax zunächst stark. Einem unerfahrenen Entwickler könnten sie alle intuitiv gleichermaßen sinnvoll erscheinen. Obgleich in diesem Anwendungsfall alle vier Zeilen zu einem positiven Testergebnis führen, führt nur eine exakt die gewünschte Prüfung durch. Die konkreten Abfragen unterscheiden sich wie folgt:

- Die Methode *contains()* in Zeile 11 ist dafür vorgesehen, ein Objekt verfügbar zu machen, welches einen bestimmten Text enthält: *"Navigiere in die Eingabemaske für Kaffee und liefere das Label zurück, welches "Coffee: 10" enthält."*
- Zeile 12 führt exakt den vorgesehenen Testschritt durch.
- Die doppelte Verwendung von *get()* in Zeile 13 führt dazu, dass erneut die komplette Seite durchsucht wird: *"Navigiere zu der Eingabemaske für Milch und tue hier nichts weiter. Navigiere dann zu dem ersten auf der Website erreichbaren Label und prüfe, dass dieses den Text "Milk: 10" enthält."* Der erste Aufruf ist somit irrelevant und es ist ungewiss, wo auf der zu testenden Website sich das gefundene Objekt befindet.
- In Zeile 14 wird auf den Zugriff auf ein Label verzichtet: *"Navigiere in die Eingabemaske für Kakao und prüfe, dass diese den Text "Cocoa: 10" enthält."* Es ist somit nicht bekannt, wo in der Eingabemaske der Text zu finden ist.

Anhand dieser Beispiele wird ersichtlich, dass es möglich ist, einen erfolgreichen Test in Cypress zu schreiben, ohne zu wissen, was genau dieser im Detail prüft. Dies führt zwar zu

schnellen Erfolgserlebnissen, was sich positiv auf die teamübergreifende Eignung auswirkt, jedoch ist es ratsam, sich mit den konkreten Verwendungszwecken der verschiedenen Methoden von Cypress auseinanderzusetzen. Nur so können falsch-positive oder -negative Tests vermieden werden. Der Aufwand, die genauen Unterschiede recherchieren, ist dank Cypress übersichtlicher Dokumentation jedoch gering.

7.1.2 Schwächen

In diesem Abschnitt werden Szenarien aufgelistet, welche in der Testautomatisierungsphase dieser Ausarbeitung nicht erfolgreich umgesetzt werden konnten, und sich somit negativ auf die Evaluation auswirken. Ausnahmen von den positiven Erfahrungen während der Entwicklung von GUI-Tests unter Cypress stellen die folgenden Nutzungsszenarien dar:

- Während eines laufenden Tests konnten keine Werte von der Website abgelesen, in Variablen gespeichert, bei Bedarf verändert oder schließlich überprüft und mit anderen Werten verglichen werden.
- Die Längen sichtbarer Texte ließen sich nicht ermitteln.
- Bei dem Versuch, während eines Tests zu bestimmten Objekten oder Positionen auf der Benutzeroberfläche scrollen zu lassen, kam es vor, dass keine oder nur eine ungenügende Bewegung des Browserfensters stattfand.

Wenngleich es positiv bewertet werden kann, dass diese Testschritte, wie Recherchen ergaben, generell wohl umsetzbar sind, sofern ausreichend Zeit und/oder Expertise zur Verfügung steht, so scheint die Nutzbarkeit bzw. Zuverlässigkeit schlechter auszufallen als jene der restlichen genutzten Methoden.

7.2 Testen der Geschäftslogik

In diesem Kapitel soll evaluiert werden, ob ein Testen der Geschäftslogik in ausreichendem Maße umgesetzt werden konnte, wenngleich keine Whitebox-Unit-Tests genutzt werden konnten (s. Kapitel 6.5).

7.2.1 Stärken

Bei Cypress handelt es sich ausdrücklich um ein Framework mit Fokus auf E2E-Tests. Diese konnten genutzt werden, um relativ ausführlich alle eingeplanten Testszenarien umzusetzen. Durch diese war es wiederum - wie in Kapitel 6.4 demonstriert - möglich, die Funktionalitäten des Demonstrationsobjekts umfänglich aus der Sicht eines Anwenders zu testen. Wenngleich der Blick in die exakten Abläufe im Backend mit einer Granularität wie Whitebox-Unit-Tests sie bieten würden, auf diesem Wege verwehrt bleibt, so kann doch festgestellt werden, ob das Ergebnis, welches der Benutzer zu sehen bekommt, wie erwartet ausfällt.

Betrachtet man hauptsächlich für E2E-Tests entwickelte Testautomatisierungsframeworks wie Cypress als Ergänzung zu Unit-Tests, so ist nicht anzunehmen, dass ein dringlicher Bedarf an einer zweiten Instanz bestehen sollte, welche Whitebox-Tests zusätzlich auf der Ebene von Integrations- und Systemtests erneut durchführt, wenn diese bereits auf Unit-Test-Ebene vorhanden sind. Aus diesem Grund wird der Mangel an umsetzbaren Unit-Tests im Kontext dieser Ausarbeitung gering gewichtet.

7.2.2 Schwächen

Es wäre ein erfreulicher Zusatz gewesen, wenn simpel zu implementierende Unit-Tests ein fester Bestandteil von Cypress wären, welcher ebenso schnell und intuitiv zu integrieren ist, wie Cypress GUI-Tests. Wird Cypress teamübergreifend eingesetzt, so wäre es von Vorteil für betroffene Entwickler und Testingenieure, wenn dasselbe Tool auf allen Testebenen verwendet werden könnte. Dies ist - wie im Abschnitt 6.5 erläutert - soweit es nach Durchführung aller für den Umfang dieser Ausarbeitung vorgesehenen Experimente beurteilt werden kann, in Hinblick auf Unit-Tests nicht der Fall. Somit ist es beispielsweise nicht möglich, vollständige Grenzwertanalysen durchzuführen, um die Robustheit verschiedener Backend-Methoden zu analysieren. Ggf. sollte sichergestellt werden, was geschieht, wenn eine Zutat einen Füllstand-Wert von -1 erreicht. Dies ist weder in der Logik noch in der Datenhaltung vorgesehen. Im vorliegenden Demonstrationsobjekt werden Eingabewerte, welche einen negativen Füllstand zur Folge hätten, bereits von der Präsentationsschicht abgefangen und nicht an die Logikschicht übermittelt. Es ist also nicht zu erwarten, dass dieser Fall unter normalen Umständen eintritt.

Nichtsdestotrotz könnte es von großem Mehrwert für ein Projekt sein, wenn sichergestellt werden könnte, dass auch unerwartete Sonderfälle stets angemessen behandelt werden.

Aus den beschriebenen Gründen wird dieser Aspekt negativ in die Bewertung einfließen. Dies bedeutet jedoch nicht, dass schichtenübergreifend arbeitende Teams Cypress keinesfalls für Unit-Tests verwenden können. Es sollten lediglich die notwendigen Ressourcen eingeplant werden, um die Umsetzung unter Cypress zu recherchieren, auf Umsetzbarkeit zu erproben und ggf. die betroffenen Testingenieure und Entwickler in der optimalen Verwendung zu unterrichten.

Ein Testszenario, welches zu Schwierigkeiten führte, ohne in Zusammenhang zu fehlenden Unit-Tests zu stehen, ist das folgende: Werden Browserbenachrichtigungen wie in Kapitel 6.4.2 via *window:alert* abgefangen und anschließend z.B. auf erwarteten Inhalt kontrolliert, so schlägt dieser Testschritt nie fehl. Im Cypress Test Runner werden betroffene Testschritte stets als erfolgreich markiert. Dies kann zu falsch-positiven Ergebnissen führen. Recherchen brachten hervor, dass Wege bekannt sind, dieses Problem zu umgehen (vgl. Shehane, 2019). In den Umfang dieser Ausarbeitung wurden diese jedoch nicht aufgenommen.

7.3 Testen der Persistenzschicht

Da Cypress die Verwendung von E2E-Tests in den Fokus stellt, stellt sich im Kontext dieser Ausarbeitung die Frage, ob Cypress es ermöglicht, die Persistenz- separat von der Präsentationsschicht zu testen. Zusätzlich besteht ein Teil der Forschungsfrage darin, inwieweit Cypress für den team- bzw. komponentenübergreifenden Einsatz (s. Kapitel 3.1) geeignet ist. Hierfür wäre es von Vorteil, mit Cypress aktuelle Werte der Anwendung abfragen zu können, ohne auf die Nutzung einer Benutzeroberfläche angewiesen zu sein. Dies würde bspw. erlauben, die Datenbank zu testen, bevor die Benutzeroberfläche testbar ist.

Darüber hinaus könnten ggf. Fehler in der Geschäftslogik dazu führen, dass Werte in der Präsentationsschicht wie erwartet angezeigt werden, obwohl sie in der Persistenzschicht fehlerhaft sind. So würde ein falsch-positives Ergebnis entstehen. Ein ausschließlich auf den Anzeigen der Benutzeroberfläche basierender Test kann Datenbankwerte also nicht eindeutig validieren.

7.3.1 Stärken

Das Verfassen von Datenbankabfragen gestaltet sich in Cypress sehr zugänglich. Sind dem Anwender die Grundlagen der korrekten Nutzung der REST-Methoden bekannt, welche für die geplanten Tests benötigt werden, so fällt die Umsetzung der Tests verhältnismäßig leicht. Sind die genauen URLs für die verschiedenen Endpunkte bekannt, welche angesprochen werden sollen, und ist in der Dokumentation der zu testenden Anwendung bzw. in den Testanforderungen nachvollziehbar festgehalten, welche Inhalte die jeweiligen Abfragen enthalten müssen, um die gewünschten Antworten zu generieren, so können zügig zahlreiche Tests implementiert werden. Sehr angenehm wurde außerdem realisiert, dass nach dem Absenden einer Anfrage standardmäßig auf die erhaltene Antwort gewartet wird. Dies ermöglicht intuitiv, den Inhalt umgehend auf erwartete Inhalte oder Formate zu überprüfen.

Die in Kapitel 6.6.1 beschriebene Testmethode *cy.request()* ermöglicht es, vor und nach beliebigen Testschritten den aktuellen Stand der in der Datenbank gespeicherten Werte abzufragen (s. Kapitel 6.6.3). Auch die Veränderung von Datenbankwerten kann mit Hilfe ebendieser Methode gezielt durchgeführt werden (s. Kapitel 6.6.2). Diese Funktionen reichen aus, um alle in Kapitel 3.3.1 definierten Operationen zu testen. Des Weiteren können sie als Testschritte in E2E-Tests verwendet werden und vervollständigen somit die Möglichkeiten, den Verlauf von schichtenübergreifenden Tests sinnvoll nachzuvollziehen, gegebenenfalls zu manipulieren und schließlich zu kontrollieren.

7.3.2 Schwächen

Ein wichtiger Aspekt der Testautomatisierung konnte im Zusammenhang mit dem vorliegenden Demonstrationsobjekt, bzw. mit der genutzten Datenbankanbindung nicht erfolgreich genutzt werden. Bei diesem handelt es sich um die Verwendung von sogenannten Datenbank-Fixtures. Diese Funktion betrifft die Initialisierung und Wartung von Datenbanken und kann somit das Testen datenabhängiger Tests erleichtern. Aus diesem Grund wird davon ausgegangen, dass dieser Aspekt eine breite Menge an potenziellen Anwendern betreffen kann. Somit wird für ebendiesen Aspekt im Kontext der Evaluation in dieser Ausarbeitung eine relativ hohe Wichtigkeit festgelegt.

Es wurden Bemühungen angestellt, Fixtures zum Zurücksetzen der Datenbank zwischen verschiedenen Testmethoden zu nutzen. Diese umfassten das Anlegen von Dateien im json-Format, welche ein Abbild der Datenbank zum gewünschten Zeitpunkt darstellten. Wann immer jedoch die Methoden ausgeführt wurden, welche für das Einspielen der Fixtures in die tatsächliche Datenbank vorgesehen sind, konnte keine Veränderung an der Datenbank festgestellt werden. Womöglich ist das verwendete DB- und API-Tool json-server nicht für derartige Eingriffe in die verwendete Datenbankdatei vorgesehen.

Es ist davon auszugehen, dass es durchaus möglich ist, die Kombination aus Cypress-Fixtures und json-server wie gewünscht in einen funktionalen Zustand zu versetzen. Hierzu wäre jedoch voraussichtlich ein größerer Zeitaufwand von Nöten. Dieses spezifische Feature ist jedoch nicht als Kernfaktor für diese Ausarbeitung vorgesehen. Des Weiteren können in den Projekten, auf welche das hier entwickelte Testkonzept übertragen werden soll, zahlreiche andere Tools oder Datenbankimplementierungen verwendet werden. Aus diesen Gründen wurde dieses Vorhaben niedrig priorisiert und letztendlich nicht umgesetzt. Es stellt jedoch grundsätzlich einen Aspekt dar, welcher auch andere Datenbank-Tools als das verwendete json-server betreffen könnte. Daher wird es in den Ausblick für weitere Ausarbeitungen in Kapitel 8.3 aufgenommen.

Von den beschriebenen Schwierigkeiten mit Fixtures abgesehen, konnten während der Entwicklung von Datenbanktests mit Cypress keine negativen Aspekte festgestellt werden.

7.4 Erweiterbarkeit von Cypress-Tests

Die Erweiterbarkeit eines Testautomatisierungsframeworks ist ein Aspekt, der besonders von der Tatsache betroffen ist, dass in dem Umfang dieser Ausarbeitung naturgemäß nur Beispielszenarien mit überschaubarem Umfang genutzt werden können. Die Charakteristik der Erweiterbarkeit ist zu komplex, als dass sie mit den verfügbaren Ressourcen erschöpfend erprobt und analysiert werden könnte. Daher soll hiermit erneut darauf hingewiesen werden, dass die in Kapitel 6.7 demonstrierte Situation lediglich einen überschaubaren Einblick in die Erweiterbarkeit von schichtenübergreifenden Tests unter Cypress ermöglichen kann. Aus diesem Grund wird sie in der Bewertung auch schwächer gewichtet als andere Charakteristiken.

Der Aufwand, welcher für die programmatische Implementierung der gewählten Änderung im Demonstrationsobjekt aufgebracht werden musste, wird hierbei nicht berücksichtigt, da dieser nicht von Cypress, sondern von der genutzten Programmiersprache, bzw. dem verwendeten Entwicklungsframework - in diesem Fall Angular - abhängt.

7.4.1 Stärken

Nach den Änderungen am Demonstrationsobjekt fiel positiv auf, dass lediglich ein einzelner Test fehlschlug. Zusätzlich handelte es sich bei diesem um einen optimistisch formulierten Test, bei dessen Umsetzung kein Wert auf Robustheit oder Flexibilität gelegt wurde. Sorgfältiger implementierte Tests blieben unbeeinflusst (s. Kapitel 6.7.2).

Wie in Abschnitt 6.7.5 zusammengefasst, entstand durch die notwendigen Testanpassungen ein überschaubarer und den Funktionserweiterungen angemessener Arbeitsaufwand.

7.4.2 Schwächen

Während der Testautomatisierung im Kontext dieser Ausarbeitung konnten keine Schwächen in der Erweiterbarkeit von unter Cypress verfassten Tests festgestellt werden.

7.5 Nutzerfreundlichkeit von Cypress

Da das Empfinden zur Nutzerfreundlichkeit je nach Hintergrund des Nutzers unterschiedlich ausfallen kann, spielen besonders in diesem Kapitel der Hintergrund und die Erfahrungen des Autors eine Rolle. Um die Nutzerfreundlichkeit von Cypress bewerten und dabei möglichst objektiv vorgehen zu können, wurden in Kapitel 3.2.1 Bewertungskriterien entwickelt. In den folgenden Abschnitten wird zunächst auf ebendiese eingegangen. Sofern vorhanden werden anschließend Erfahrungen behandelt, welche über diese hinausgehen.

7.5.1 Stärken

Es würde den Rahmen dieser Arbeit sprengen, jeden positiven Aspekt der Nutzerfreundlichkeit unter Cypress ausführlich zu behandeln. Daher werden im Folgenden lediglich kurze

Erläuterungen zu einer nicht nach Wichtigkeit sortierten Auswahl an relevanten Thematiken verfasst:

Schnelle Einsatzbereitschaft (Effizienz)

Befolgt man die Anweisungen in der Dokumentation von Cypress, so kann schnell ein Punkt erreicht werden, an welchem produktiv mit diesem Testautomatisierungsframework gearbeitet werden kann. Von der Installation des Node Package Managers bis zur Ausführung der ersten automatisierten Tests sind alle Schritte leicht nachvollziehbar.

Sowohl die Installationsdauer als auch die Performanz während der Ausführung von Tests scheint außerdem nicht stark von der Hardware des genutzten Endgerätes abzuhängen. Entwickelt und getestet wurde auf einem leistungsstarken Windows-PC, einem Mittelklasse-MacBook und einem in die Jahre gekommenen Windows-Laptop. Es wurden keine auffälligen Laufzeitunterschiede bemerkt.

Zusätzlich werden standardmäßig einige Tests generiert, welche grundlegende Funktionen an einer Beispielwebsite demonstrieren. Dies ermöglicht neuen Nutzern einen schnellen Einstieg.

Übersichtlichkeit in der Nutzung

Arbeitet man an der Automatisierung von Tests und möchte ein bestimmtes Szenario umsetzen, gibt es zwei Quellen, welche bei einer schnellen Realisierung unterstützen. Als erstes sei die in dieser Ausarbeitung oft referenzierte Dokumentation auf der Cypress-Website (Cypress.io Overview, 2021) genannt. Diese bietet einen breiten Überblick über die verfügbaren Methoden. Außerdem sind diese häufig mit Empfehlungen zur optimalen Verwendung versehen**.** Zweitere sind die Beispieltests. Während der initialen Einrichtung von Cypress wird automatisch eine Sammlung verschiedener Tests zu Demonstrationszwecken generiert. Diese geben einen umfangreichen Einblick in Cypress Testmethoden. Auf Grundlage dieser können erste Tests abgeleitet werden, was den Einstieg in eine produktive Nutzung der Testautomatisierung erleichtert. Durch diese Hilfestellungen kann schnell ein Verständnis für die Nutzung von Cypress entwickelt werden, was eine intuitive Testentwicklung ermöglicht.

Nachvollziehbarkeit der Testabläufe (Nutzerfeedback)

Der Test Runner stellt eine der größten Stärken von Cypress dar. Besonders positiv aufgefallen ist die Möglichkeit, per Mauszeiger zu jedem ausgeführten Testschritt springen zu können. Dies zeigt den Zustand der Benutzeroberfläche an, der zum Zeitpunkt der Ausführung aktuell sichtbar war. Diese Funktion hat das Debugging während dieser Ausarbeitung oft stark beschleunigt. Entscheidet man sich für die Beschränkung der Ausgaben auf die Kommandozeile, zeigt diese ebenso Fortschritt, Ergebnisse und Zeilenverweise zu Fehlerursprüngen an. Detaillierter wurde diese Variante jedoch nicht analysiert. Zusätzlich können Videoaufzeichnungen der Testläufe angelegt werden. Diese kann der Tester bei Bedarf nach einem Testlauf auswerten. Wird auf diese Weise eine Fehlerwirkung aufgedeckt, kann das Video ggf. an verantwortliche Entwickler weitergeleitet werden, um eine schnelle, unkomplizierte Kommunikation zu ermöglichen.

In ihrer Gesamtheit können die genannten Stärken der Nachvollziehbarkeit besonders positiv zu der Kommunikation in einem teamübergreifenden Projekt beitragen.

Wiederverwendbarkeit von Code

Cypress unterstützt das Erstellen von globalen Testmethoden zur projektweiten Wiederverwendung, wie in Kapitel 6.6.2 demonstriert, auf verständliche Art und Weise und erfüllt somit die Anforderungen.

7.5.2 Schwächen

Zu den genannten Kriterien wie schneller Einsatzbereitschaft, Übersichtlichkeit, Nachvollziehbarkeit und Wiederverwendbarkeit konnten keinerlei gravierende Schwächen festgestellt werden. Generell fällt die Anzahl der negativen Erfahrungen mit Cypress, die über den Zeitraum dieser Ausarbeitung gesammelt werden konnten, gering aus. Der Großteil kann unter folgendem Punkt zusammengefasst werden: Sobald man einen schwer zu quantifizierenden Grad an Komplexität überschreitet, fällt die Suche nach Lösungsansätzen in der Dokumentation von Cypress exponentiell schwerer. Womöglich hat sich das Team bei der Dokumentation bisher darauf fokussiert, die simpleren Einstiegshürden besonders gut zu behandeln. Aktuell scheint daraus zu resultieren, dass in den Bereichen der gehobenen

Anforderungskomplexitäten ein Nachholbedarf in gut nachvollziehbarer und übersichtlicher Dokumentation besteht.

Des Weiteren fällt auf, dass von Cypress nativ keine Umsetzung des Capture-and-Replay-Ansatzes angeboten wird. Dies muss nicht zwingend ein Nachteil sein, da nicht jeder Tester auf diese Funktion angewiesen ist. Dennoch könnte sie von Anwendern, welche diese Art der Testschrittgenerierung nutzen wollen, vermisst werden. Es existieren Browser-Erweiterungen von Drittanbietern, welche dies ermöglichen. Bei einem Test eines Browser-Plugins namens Cypress Recorder (vgl. KabaLabs, 2021) konnte zwar ein anfängliches Testskript generiert werden, jedoch war dieses in den meisten Fällen nicht lauffähig und musste umfassend erweitert werden, um einen vollwertigen Test umzusetzen. Die Zeitersparnis war marginal und es wurde von weiterer Verwendung dieses und vergleichbarer Plugins abgesehen.

8 Fazit und Ausblick

In dem letzten Kapitel der vorliegenden Ausarbeitung soll zunächst eine Zusammenfassung des Forschungsinteresses gegeben werden, und welche wesentlichen Arbeitsschritte zur Beantwortung der Forschungsfragen geführt haben. Daraufhin werden die Ergebnisse der in Kapitel 0 erarbeiteten Evaluation zusammengefasst und eine abschließende Bewertung von Cypress für den schichten- sowie teamübergreifenden Einsatz formuliert, um damit auf die Forschungsfragen aus Kapitel 1.2 zu antworten. Zuletzt wird ein Ausblick gegeben, welche möglichen Fragestellungen oder Anwendungserweiterungen interessante Ansätze für weiterführende oder tiefergehende Arbeiten an dem Thema Testautomatisierung mit Cypress bieten könnten.

8.1 Zusammenfassung

Durch die fortschreitende Komplexität von Software, wie zum Beispiel von Smart-Home-Anwendungen, sowie die erschwerte Kommunikation zwischen Teams, die Software schichtenübergreifend entwickeln, ergibt sich eine erhöhte Relevanz von automatisierten Software-Tests und spezifischer eines einheitlichen Frameworks zur Testautomatisierung, um eine stabile Qualität der Ergebnisse sowie einen verringerten Kommunikations- und Arbeitsaufwand der Entwicklerteams zu gewährleisten. Vor diesem Hintergrund hatte die vorliegende Bachelorarbeit zum Ziel, zu prüfen, inwiefern das Framework Cypress für die schichtenübergreifende Testautomatisierung geeignet ist und ob es auch zur teamübergreifenden Nutzung empfohlen werden kann.

Um Cypress als Testautomatisierungsframework anwenden und anschließend evaluieren zu können, wurde als Demonstrationsobjekt eine simulierte Kaffeemaschine als Webanwendung entwickelt. Eine Kaffeemaschine ist einerseits ein simples Alltagsgerät, andererseits bietet sie auch das Potenzial für eine Einbindung in den Kontext zukunftsträchtiger Smart-Home-Netze. Zuletzt erfüllte sie alle Anforderungen, die für die Beantwortung der Forschungsfragen relevant sind und in den Kapiteln 3 bis 4 definiert wurden. Der Kernprozess dieser Ausarbeitung, die praktische Umsetzung der Testautomatisierung sowie das Sammeln von

empirischen Erkenntnissen findet in Kapitel 6 statt. In diesem Zuge werden einige Stärken und Schwächen von Cypress sichtbar, welche in Kapitel 7 ausgewertet werden, um hierauf basierend Cypress als Testautomatisierungsframework evaluieren zu können. Die wichtigsten Ergebnisse werden im Folgenden zusammengefasst.

8.2 Bewertung

Für die abschließende Bewertung werden zunächst die Evaluationen der in den Kapiteln 3.2 definierten und in Kapitel 0 behandelten Bewertungskriterien zusammengefasst. Anschließend wird ein Gesamtfazit formuliert, um die erarbeiteten Erkenntnisse prägnant zusammenzuführen, und damit die Forschungsfragen abschließend zu beantworten.

Testen der Präsentationsschicht mit GUI-Tests

Das Testen der Benutzeroberfläche und die Umsetzung von E2E-Tests mit Hilfe von GUI-Testmethoden gestalten sich unter Cypress erwartet angenehm. Die meisten Testszenarien konnten prompt umgesetzt werden. Die wichtigsten Testschritte konnten unter Verwendung relativ simpler Methodenkombinationen und eingängiger Syntax implementiert werden. Dies legt einen soliden Grundstein für schichtenübergreifendes Testen und sollte teamübergreifend für Akzeptanz sorgen. Um einige fortgeschrittenere Tests umsetzen zu können, müssen ausreichend Ressourcen eingeplant werden. Teamübergreifend sollten Erfolge, Erfahrungen und gefundene Lösungen zentral dokumentiert werden, um Ressourcen effizient einsetzen und Recherchezeit einsparen zu können.

Testen der Geschäftslogik mit E2E-Tests

Wenngleich Unit-Tests im Rahmen dieser Ausarbeitung nicht implementiert werden konnten, so konnte die Geschäftslogik des Demonstrationsobjekts dennoch in befriedigendem Maße durch E2E-Tests abgedeckt werden. Die Granularität eines Whitebox-Tests kann auf diesem Wege nicht erreicht werden, aus schichtenübergreifender Sicht kann jedoch eine zufriedenstelle Absicherung erreicht werden.

Testen der Persistenzschicht mit Datenbanktests

Die Möglichkeiten, die Cypress zum Testen der Datenhaltung bietet, sind überschaubar. Die meisten Vorgehensweisen, die im Umfang dieser Ausarbeitung analysiert werden konnten, beruhen auf der Methode *cy.request()*. Diese bot jedoch alle bis hierhin benötigten Operationen an. Auch da sie reibungslos in E2E-Tests eingegliedert werden kann, stellt sie einen wichtigen Bestandteil des schichtenübergreifenden Testens dar und erhöht die mögliche Testabdeckung ungemein. Wird mit dem Einsatz von Cypress begonnen, sollten die Ressourcen eingeplant werden, um teamübergreifend zu dokumentieren, wie genau Fixtures erstellt und verwendet werden sollen.

Nutzerfreundlichkeit

Wie in Kapitel 7.5 erläutert, konnte Cypress alle in Kapitel 3.2.1 definierten Anforderungen an die Nutzerfreundlichkeit erfüllen. Die Einbindung des Frameworks in ein Projekt ist gut dokumentiert und schnell umgesetzt. Lediglich bei Funktionen mit fortgeschrittener Komplexität wäre eine ausführlichere Dokumentation wünschenswert. Die Datenstruktur unterstützt eine übersichtliche Gliederung der Tests und erlaubt es, wiederverwendbare Methoden zu hinterlegen. Die Nachvollziehbarkeit der Testläufe stellt dank des intuitiv gestalteten Test Runners eine von Cypress größten Stärken dar. Die Verwendung von Cypress wurde stets als positives Nutzererlebnis wahrgenommen, da Erfolge und Fortschritte für Gewöhnlich rasch erzielt werden konnten. Dies sollte zu einer teamübergreifenden Akzeptanz beitragen.

Erweiterbarkeit der Tests

Die Erweiterbarkeit implementierter Tests wurde anhand eines überschaubaren Beispielszenarios, in welchem Änderungen an dem bestehenden Demonstrationsobjekt vorgenommen wurden, nachdem die Testautomatisierung bereits abgeschlossen war, erprobt. Fehlgeschlagene Tests gaben schnell und verständlich Auskunft über den Fehlerursprung. Dies erlaubte eine rasche Fehlerbehebung. Ein derartiges Experiment kann zwar nur einen kleinen Einblick in die generelle Erweiterbarkeit von Cypress-Tests geben, doch die gemachten Erfahrungen in Kombination mit der übersichtlichen Struktur von Testskripten im Allgemeinen

deuten darauf hin, dass es auch in größeren Änderungsszenarien in realen Anwendungsfällen eine flexible und schnelle Erweiterbarkeit bietet. Dieses Ergebnis lässt darauf schließen, dass ein team- und schichtenübergreifender Einsatz von Cypress schnelle Reaktionen auf häufige Änderungen der zu testenden Software ermöglicht.

Überdeckung der Testebenen

Es sollte stets festgelegt werden, dass die unter Cypress geschriebenen Tests als Ergänzung zu herkömmlichen Unit-Tests gesehen werden sollten. Eine jede Komponente sollte weiterhin - wie auch in Projekten ohne Testautomatisierungsframework üblich - mit ausreichend Unit-Tests abgesichert werden. Einige Komponenten können bereits auf dieser Testebene von Cypress profitieren, der größte Mehrwert wird jedoch in schichtenübergreifenden Integrations- und Systemtests gesehen.

Durchstich der Softwareschichten / E2E-Tests

Wie in Kapitel 3.2.4 beschrieben, sollten im Kontext dieser Arbeit u.a. Tests betrachtet werden, welche alle drei betrachteten Softwareschichten umfassen. Hierfür waren die durch Cypress ermöglichten E2E-Tests gut geeignet. Durch die intuitive Syntax von Cypress fällt die Interaktion mit der Benutzeroberfläche sowie der Datenhaltung gleichermaßen leicht. Tests mit einem Point of Observation in einer anderen Schicht als dem Point of Control waren ursprünglich nicht für diese Ausarbeitung geplant. Dank ihrer simplen Umsetzbarkeit konnten sie jedoch eine weitere Stärke von Cypress offenbaren (s. Kapitel 6.6.3 Punkt 2A).

Beantwortung der Forschungsfragen

Final können beide Forschungsfragen, sowohl nach schichten- als auch teamübergreifender Einsetzbarkeit, positiv beantwortet werden. Die größte gefundene Schwäche stellen die schwer umsetzbaren GUI-Testszenarien in Kapitel 7.1.2 dar. Durch seinen ansonsten sehr zufriedenstellenden Funktionsumfang ist Cypress dennoch gut für das Testen der Präsentations- und Persistenzschicht geeignet. Dank seiner E2E-Tests ist es auch für die Geschäftslogik und schichtenübergreifende Tests einsetzbar. Die Umsetzung ebendieser geht dank simpler, intuitiver Syntax und guter Nutzerfreundlichkeit leicht von der Hand. Darüber

hinaus wird der Nutzer bei der Testentwicklung und beim Debugging vom Test Runner unterstützt, was die Arbeit erleichtert und indirekt beschleunigt. Der Testumfang kann schnell und flexibel erweitert werden. Dank der übersichtlichen Struktur von Testskripten wird die teamübergreifende Kommunikation erleichtert und Fehlerursachen können schnell gefunden und behoben werden.

Des Weiteren ermöglicht Cypress die Wiederverwendung entwickelter Testmethoden, kann grundsätzlich zu jedem Zeitpunkt in der Softwareentwicklung eingesetzt werden und bietet eine ausführliche und verständlich geschriebene Dokumentation.

Es wird außerdem davon ausgegangen, dass Testingenieure und Entwickler davon profitieren, wenn Cypress in ihrem Unternehmen möglichst weit verbreitet ist. So können Herausforderungen während der Testautomatisierung gemeinsam analysiert, und Best Practices erarbeitet und zentral gesammelt werden. Dieses Potenzial kann in einer Simulation, wie sie in dieser Ausarbeitung erzeugt wurde, naturgemäß nicht ausgeschöpft werden.

Insgesamt wird Cypress als zukunftsträchtiges Framework zur Testautomatisierung bewertet, das durch seine genannten Vorzüge nicht nur die Komplexität von Software und ihren Tests schichtenübergreifend abdecken, sondern auch die Kommunikation und Kollaboration über Teams hinweg erleichtern kann. Es wird daher eine klare Empfehlung ausgesprochen, Cypress in Entwicklerteams schichtenübergreifender Webanwendungen zu nutzen.

8.3 Ausblick

Im Folgenden sollen einige Themen aufgelistet werden, welche für weiterführende Studien mit einem ähnlichen Forschungsinteresse interessant sein könnten.

- **Unit-Tests in Cypress** konnten nicht erfolgreich umgesetzt werden. In einer Arbeit, welche sich auf einzelne Detail-Bereiche der Testautomatisierung fokussiert, könnten hierfür sinnbringende Methoden analysiert werden.
- **Schichtenübergreifende Tests mit variierendem Point of Control und Point of Observation** sind ein Thema, auf welches in dieser Arbeit größtenteils verzichtet wurde. Es stellte sich jedoch heraus, dass diese in vielen Fällen ohne großen Mehraufwand umgesetzt werden können. Da sie in verschiedenen Szenarien von

Integrationstests, bei welchen nur einzelne Komponenten zur Verfügung stehen, hilfreich sein können, könnte dieses Forschungsthema einen gewissen Mehrwert bieten.

- Der Kontext von **Smart-Home-Anwendungen** bot besonders interessante Ansätze für die Entwicklung einer zeitgemäßen zu testenden Anwendung und hat das Konzept des Demonstrationsobjekts maßgeblich mitbestimmt. Die tatsächliche Umsetzung einer Smart-Home-Anwendung mit Anbindung an ein Netzwerk oder verteilte Systeme hätte den Rahmen dieser Arbeit jedoch gesprengt, und war für die Beantwortung der Forschungsfragen nicht notwendig. Für andere Arbeiten könnte es von Interesse sein, explizit die Hürden der Testautomatisierung für vernetzte Smart-Home-Anwendungen zu demonstrieren. Hierfür könnte die vorliegende smarte Kaffeemaschine weiterentwickelt werden.

- Falls kein gesteigerter Wert auf den Smart-Home-Aspekt des Demonstrationsobjekts gelegt wird, so bietet es dennoch **Potenzial für neue Funktionen**. Bspw. könnte ein Zähler implementiert werden, welcher nach X zubereiteten Getränken eine Entkalkung empfiehlt. Auch das Löschen von Rezepten wäre eine denkbare Funktionserweiterung. Solche und weitere fortgeschrittene Funktionen würden wiederum neue, interessante Anforderungen an die Testautomatisierung bedeuten.

- Auch die innere **Architektur des Demonstrationsobjekts** könnte womöglich optimiert werden. So könnte man z.B. die Geschäftslogik, welche sich momentan größtenteils im “MachineService” befindet, auf mehrere kleinere Services aufteilen, oder ein gänzlich abweichendes Architekturmuster anwenden, und anschließend kontrollieren, wie sich diese Änderungen auf die Testautomatisierung auswirken.

- Für die **Auswahl der Datenbank** wurde größter Wert auf eine schnelle Einsatzbereitschaft gelegt, was zu der Nutzung von json-server führte. Würde stattdessen eine komplexere/umfangreichere oder selbstentwickelte Lösung eingesetzt, könnten sich die Ansprüche an die Testautomatisierung verändern. Auch die **Nutzung von Fixtures** könnte in Kombination mit anderen Datenbank-Tools leichter fallen.

- Ein **Vergleich mit anderen Testautomatisierungsframeworks** könnte neue Einblicke in die Konkurrenzfähigkeit von Cypress gewähren.
- Um die verschiedenen Aspekte von Cypress zu bewerten, wurden in dieser Ausarbeitung einige Anforderungen definiert und anschließend analysiert, wie gut diese erfüllt werden konnten. Womöglich könnte man sich anderer Forschungsmethoden bedienen, oder eine maßgeschneiderte **Metrik für die Bewertung von Testautomatisierungsframeworks** entwickeln.
- Auch auf eine **Messung der Test Coverage durch Cypress** wurde verzichtet, da diese nicht im Fokus dieser Arbeit steht. Für viele andere Forschungsfragen könnte diese jedoch durchaus von Interesse sein. Die Einrichtung einer Testabdeckungs-Überwachung wird in einem dedizierten Abschnitt der Dokumentation von Cypress erläutert (vgl. Cypress.io Code Coverage, 2021).
- **Browserübergreifende Testautomatisierung mit Cypress** ist ein weiteres Gebiet, welches im Rahmen dieser Ausarbeitung keinen Platz gefunden hat, aber dennoch von Interesse sein könnte.
- Zu guter Letzt könnte noch analysiert werden, wie gut Cypress für **nicht-funktionale Tests** geeignet ist.

Literaturverzeichnis

Angelov, A. (kein Datum). *Page Object Pattern in Automated Testing*. Abgerufen am 17. 11 2021 von automatetheplanet.com: https://www.automatetheplanet.com/page-object-pattern/

Applitools. (2021). *What is Visual Testing*. Abgerufen am 15. 11 2021 von Applitools.com: https://applitools.com/automated-visual-testing-best-practices-guide/#what-is-visual-testing

Bahmutov, G. (03. 01 2019). *Stop using Page Objects and Start using App Actions*. Abgerufen am 26. 11 2021 von Cypress.io: https://www.cypress.io/blog/2019/01/03/stop-using-page-objects-and-start-using-app-actions/#page-objects-problems

Brandt, M. (06. 07 2021). *So smart sind Deutschlands Haushalte*. Abgerufen am 04. 12 2021 von Statista: https://de.statista.com/infografik/3105/anzahl-der-smart-home-haushalte-in-deutschland/

Clarke, B. (24. 06 2021). *Warum eine Investition in die Wiederverwendbarkeit von Code so hilfreich ist*. Abgerufen am 16. 11 2021 von outsystems.com: https://www.outsystems.com/de-de/blog/posts/code-reusability/

Cypress.io. (16. 08 2021). *How it works*. Abgerufen am 26. 11 2021 von cypress.io: https://www.cypress.io/how-it-works

Cypress.io Code Coverage. (27. 09 2021). *Code Coverage*. Abgerufen am 26. 11 2021 von docs.cypress.io: https://docs.cypress.io/guides/tooling/code-coverage

Cypress.io Events. (27. 09 2021). *Catalog of Events*. Abgerufen am 26. 11 2021 von docs.cypress.io: https://docs.cypress.io/api/events/catalog-of-events

Cypress.io FAQ. (16. 08 2021). *General Questions*. Abgerufen am 26. 11 2021 von docs.cypress.io: https://docs.cypress.io/faq/questions/using-cypress-faq#What-are-your-best-practices-for-organizing-tests

Cypress.io Overview. (16. 08 2021). *Why Cypress?* Abgerufen am 26. 11 2021 von docs.cypress.io: https://docs.cypress.io/guides/overview/why-cypress

Cypress.io Stubs. (03. 12 2021). *Stubs, Spies, and Clocks*. Abgerufen am 04. 12 2021 von docs.cypress.io: https://docs.cypress.io/guides/guides/stubs-spies-and-clocks

Cypress.io Tools. (16. 08 2021). *Bundled Tools*. Abgerufen am 26. 11 2021 von docs.cypress.io: https://docs.cypress.io/guides/references/bundled-tools

Darwin, P. B. (21. 07 2021). *Angular Style Guide*. Abgerufen am 04. 11 2021 von angular.io: https://angular.io/guide/styleguide

Elm, D. (07. 12 2018). *Cypress: The future of E2E testing*. Abgerufen am 26. 11 2021 von SpeakerDeck: https://speakerdeck.com/d3lm/cypress-the-future-of-e2e-testing?slide=27

Fink, A. (31. 10 2012). *Verteiltes IT-System*. Abgerufen am 16. 11 2021 von enzyklopaedie-der-wirtschaftsinformatik.de: https://www.enzyklopaedie-der-wirtschaftsinformatik.de/lexikon/is-management/Systementwicklung/Softwarearchitektur/Architekturparadigmen/Verteiltes-IT-System

Google. (28. 10 2021). *What is Angular*. Abgerufen am 15. 11 2021 von Angular.io: https://angular.io/guide/what-is-angular

Grechenig, T. (2010). *Softwaretechnik: Mit Fallbeispielen aus realen Entwicklungsprojekten.* Pearson Deutschland GmbH. Von Pearson Studium. abgerufen

Hochschule Augsburg. (06. 03 2014). *Schichtenarchitektur*. Abgerufen am 09. 10 2021 von GlossarWiki der Hochschule Augsburg (HSA): https://glossar.hs-augsburg.de/Schichtenarchitektur

IBM Cloud Education. (19. 08 2020). *Application Programming Interface (API)*. Abgerufen am 15. 11 2021 von IBM Cloud Learn Hub: https://www.ibm.com/cloud/learn/api

ISTQB. (21. 10 2016). *Test Automation Engineer.* Abgerufen am 04. 12 2021 von Certified Tester Advanced Level Syllabus Test Automation Engineer: https://www.istqb.org/certification-path-root/test-automation-engineer.html

it-daily.net. (14. 06 2021). *Software wird immer komplexer und sicherer*. Abgerufen am 09. 10 2021 von it-daily.net: https://www.it-daily.net/it-management/business-software/28958-software-wird-immer-komplexer-und-sicherer

ITWissen.info. (15. 07 2020). *Three-Tier-Architektur*. Abgerufen am 09. 10 2021 von ITWissen.info: https://www.itwissen.info/Three-Tier-Architektur-three-tier-architecture.html

Johner, C. (10. 08 2015). *ISO 25010 und ISO 9126*. Abgerufen am 09. 10 2021 von johner-institut.de: https://www.johner-institut.de/blog/iec-62304-%20medizinische-software/iso-9126-und-iso-25010

KabaLabs. (12. 04 2021). *Cypress Recorder*. Abgerufen am 16. 10 2021 von GitHub.com: https://github.com/KabaLabs/Cypress-Recorder/

Microsoft. (26. 01 2010). *Business Layer Guidelines*. Abgerufen am 14. 11 2021 von Design Fundamentals: https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ee658103(v=pandp.10)

Mwaura, W. (2021). *End-to-End Web Testing with Cypress.* Packt Publishing Ltd.

OpenJS Foundation. (13. 07 2021). *Introduction to Node.js*. Abgerufen am 09. 10 2021 von nodejs.dev: https://nodejs.dev/learn

Oppmann, D. (30. 04 2019). *Testautomatisierung – Definition und Vorteile*. Abgerufen am 05. 12 2021 von Salt Solutions: https://www.salt-solutions.de/solutions/detail/testautomatisierung-definition-und-vorteile.html

Patil, A. (09. 05 2021). *3 Layered Architecture*. Abgerufen am 09. 10 2021 von Canarys: https://www.ecanarys.com/Blogs/ArticleID/76/3-Layered-Architecture

Pelz, A. (kein Datum). *Usability & Interaktionsprinzipien*. Abgerufen am 15. 11 2021 von ux-ui-design.de: https://ux-ui-design.de/usability-interaktionprinzipien/

Shehane, J. (31. 12 2019). *Expect is failing but test is passing in cy.on confirm & alert message · Issue #6068 · cypress-io/cypress*. Abgerufen am 25. 11 2021 von GitHub: https://github.com/cypress-io/cypress/issues/6068

Shen, A. (09. 09 2019). *classdiagram-ts*. Abgerufen am 04. 11 2021 von marketplace.visualstudio.com: https://marketplace.visualstudio.com/items?itemName=AlexShen.classdiagram-ts

Spillner, A., & Linz, T. (2012). *Basiswissen Softwaretest: Aus-und Weiterbildung zum Certified Tester-Foundation Level nach ISTQB-Standard.* dpunkt. verlag.

Statistisches Bundesamt. (06. 07 2021). *3,3 Millionen Menschen nutzten 2020 smarte Haushaltsgeräte*. Abgerufen am 04. 12 2021 von DeStatis: https://www.destatis.de/DE/Presse/Pressemitteilungen/Zahl-der-Woche/2021/PD21_27_p002.html

Sumo Logic. (20. 01 2020). *CRUD*. Abgerufen am 15. 11 2021 von sumo logic: https://www.sumologic.com/glossary/crud/

TechUpdatesDaily. (26. 05 2020). *Definition Of Proxy*. Abgerufen am 16. 11 2021 von Tech Updates Daily: https://www.techupdatesdaily.com/definition-of-proxy/

Testautomatisierung.org. (kein Datum). *E2E-Testing: Was sind End-to-End-Tests*. Abgerufen am 09. 10 2021 von testautomatisierung.org: https://www.testautomatisierung.org/lexikon/e2e-testing/

Vailshery, L. S. (22. 01 2021). *Consumer spending on smart home related devices worldwide from 2019 to 2025*. Abgerufen am 09. 10 2021 von Statista: https://www.statista.com/statistics/873607/worldwide-smart-home-annual-device-sales/

van Straalen, M. (11. 08 2020). *cypress-plugin-snapshots*. Abgerufen am 08. 10 2021 von npmjs.com: https://www.npmjs.com/package/cypress-plugin-snapshots

Verhoelen, J. (30. 09 2020). *Grüne Test-Pyramiden mit Cypress – UI-Testing für die Zukunft*. Abgerufen am 26. 11 2021 von Codecentric: https://blog.codecentric.de/2020/09/ui-end2end-testing-cypress/

I Anhang

Im Anhang dieser Ausarbeitung befindet sich eine CD. Auf dieser befinden sich die folgenden Dateien:

- **Die digitale Version dieser Bachelorarbeit im PDF-Format** namens BA-Duehrkop.pdf.
- **Das zugehörige Softwareprojekt in einem ZIP-Ordner**
  Dieser beinhaltet wiederum das Verzeichnis angular-coffeemaker-main. In diesem befindet sich sowohl die Webapplikation, die das getestete Demonstrationsobjekt darstellt als auch das Unterverzeichnis cypress, in welchem sich die Testautomatisierung befindet. Die ebenfalls enthaltene Datei README.md erklärt die Einrichtung der Webapplikation mit zugehöriger Datenhaltung sowie die Ausführung der automatisierten Tests.

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

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

[redacted]

Ort	Datum	Unterschrift im Original