

Hochschule Darmstadt

– Fachbereich Informatik –

Vergleich von End-to-End-Test-Frameworks in Flutter und React Native

Abschlussarbeit zur Erlangung des akademischen Grades
Bachelor of Science (B.Sc.)

vorgelegt von

Cedric Engler

Matrikelnummer: 767056

Referentin : Prof. Dr. Ute Trapp
Korreferent : Prof. Dr. Kai Renz

ERKLÄRUNG

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.

Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Darmstadt, 05. August 2024



Cedric Engler

ABSTRACT

This thesis investigates and compares various end-to-end test frameworks for cross-platform applications in the context of a migration from Flutter to React Native. The motivation for this work stems from the DDiT research project, initially developed in Flutter, which now needs to be migrated to React Native due to various factors. The aim of the study is to identify an end-to-end test framework that enables the creation of tests equally applicable to both Flutter and React Native. To this end, the test frameworks Appium and Maestro are evaluated and assessed based on defined criteria and a subsequent evaluation matrix. The findings of the thesis indicate that both tools exhibit specific strengths and weaknesses but are suitable for end-to-end testing of cross-platform applications in the context of a migration.

ZUSAMMENFASSUNG

Diese Arbeit untersucht und vergleicht verschiedene End-to-End-Test-Frameworks für Cross-Plattform-Anwendungen im Kontext einer Migration von Flutter zu React Native. Die Motivation für diese Arbeit entspringt dem Forschungsprojekt DDiT, das ursprünglich in Flutter entwickelt wurde, nun aber aufgrund verschiedener Faktoren auf React Native migriert werden soll. Ziel der Arbeit ist es, ein End-to-End-Test-Framework zu bestimmen, das die Erstellung von Tests ermöglicht, die sowohl für Flutter als auch für React Native gleichermaßen anwendbar sind. Hierzu werden die Testframeworks Appium und Maestro evaluiert und anhand definierter Kriterien sowie einer darauf aufbauenden Bewertungsmatrix bewertet. Die Ergebnisse der Arbeit zeigen, dass beide Tools spezifische Stärken und Schwächen aufweisen, jedoch für die End-to-End-Testung von Cross-Plattform-Anwendungen im Kontext einer Migration geeignet sind.

INHALTSVERZEICHNIS

I Thesis	
1 Einleitung	2
1.1 Motivation	2
1.2 Ziel der Arbeit	2
1.3 Gliederung der Arbeit	3
2 Grundlagen	4
2.1 Cross-Plattform-Frameworks	4
2.1.1 Flutter	4
2.1.2 React Native	5
2.1.3 Gemeinsamkeiten und Unterschiede	7
2.2 Tests	8
2.2.1 Unit Test	9
2.2.2 Integration Test	9
2.2.3 System Testing	9
2.2.4 User Acceptance Testing	9
2.2.5 End-to-End-Test	9
2.2.6 Visual Regression Testing	10
2.3 End-to-End-Test-Frameworks	10
2.3.1 Appium	10
2.3.2 Maestro	11
2.3.3 Detox	13
2.3.4 Flutter Integration Test Package	15
2.3.5 Auswahl der Kandidaten	16
3 Relevante Arbeiten	17
4 Konzept	20
4.1 Anwendung	20
4.1.1 Grundidee	20
4.1.2 Design	21
4.1.3 Implementierung in Flutter	22
4.1.4 Implementierung in React Native	23
4.1.5 Gemeinsamkeiten und Unterschiede	24
4.2 Kriterienkatalog	25
4.2.1 Kriterien	25
4.2.2 Gewichtung	26
4.3 Testszenarien	27
4.3.1 Use Cases	27
4.3.2 Use Cases zu Testszenarien	27
5 Implementierung	30
5.1 Rahmenbedingungen	30
5.2 Maestro	31
5.2.1 Installation	31

5.2.2 Konfiguration	31
5.2.3 Implementierung der End-to-End-Tests	31
5.2.4 Durchführung	37
5.3 Appium	39
5.3.1 Installation	39
5.3.2 Konfiguration	40
5.3.3 Implementierung der End-to-End-Tests	41
5.3.4 Durchführung	47
6 Ergebnisse	49
6.1 Auswertung nach den Kriterien	49
6.1.1 Verifizierung der Funktionalität	49
6.1.2 Gemeinsame Testfälle	49
6.1.3 Cross-Plattform Unterstützung	49
6.1.4 Unterstützung von UI-Elementen	50
6.1.5 Benutzerfreundlichkeit und Dokumentation	50
6.1.6 Integration mit CI/CD Pipelines	50
6.2 Auswertung der Bewertungsmatrix	51
7 Diskussion	53
8 Zusammenfassung	56
II Appendix	
A Anhang	58
Literatur	61

ABBILDUNGSVERZEICHNIS

Abbildung 4.1	UML-Diagramm – Anwendung	21
Abbildung 4.2	Wireframes für die zu verwendete Anwendung	22
Abbildung 5.1	Bildervergleich mittels ResembleJS [Rsm]	35
Abbildung 5.2	Maestro Studio – Zeichnen einer Figur	36
Abbildung 5.3	Ausführung der Maestro Tests	38
Abbildung 5.4	WebdriverIO – Konfiguration	40
Abbildung 5.5	Appium Inspector	45
Abbildung 5.6	Ausführung der Appium Tests	48
Abbildung a.1	UML-Diagramm der Flutter Anwendung	58
Abbildung a.2	Android Studio – Schritte zur Erstellung eines Emulatorn	59
Abbildung a.3	Fehlerfall – Maestro Test für Flutter	60
Abbildung a.4	Vergleichsbilder des visuellen Vergleichs – Fehlerfall a.3	60

TABELLENVERZEICHNIS

Tabelle 2.1	React Native Komponenten	6
Tabelle 2.2	Unterschiede – Flutter und React Native	7
Tabelle 2.3	Gegenüberstellung – Auswahl der Frameworks	16
Tabelle 4.1	Gewichtung der Kriterien	26
Tabelle 4.2	Use Case zu Testszenario	29
Tabelle 6.1	Bewertungsmatrix – Auswertung	51

ABKÜRZUNGSVERZEICHNIS

UI User Interface

CLI Command Line Interface

GUI Graphical User Interface

CI Continuous Integration

CD Continuous Deployment

Teil I
THESIS

EINLEITUNG

In der Anfangsphase eines Softwareentwicklungsprojekts ist es von entscheidender Bedeutung, eine strategische Entscheidung über die Zielplattformen zu treffen, auf denen die bereitzustellende Software betrieben werden soll. Angesichts des stetig wachsenden Bedarfs an plattformübergreifenden Lösungen werden vermehrt Cross-Plattform-Frameworks, wie Flutter oder React Native in Betracht gezogen [Sta], um den Entwicklungsaufwand zu minimieren und die Effizienz der Entwicklungsprozesse zu steigern. In der Softwareentwicklung ist es dabei gängige Praxis, eine vordefinierte Framework-Architektur, anhand spezifischer Kriterien zu wählen, wie Time-to-Market oder Kosteneffizienz [Rad]. Dennoch können sich Situationen ergeben, in denen ein Wechsel zu einem anderen Framework aus technischer Notwendigkeit erforderlich ist, etwa aufgrund von Unmöglichkeiten, bestimmte Faktoren innerhalb des bisherigen Frameworks umzusetzen. In einem solchen Szenario stellt sich die Frage, wie mit der bereits etablierten Infrastruktur, einschließlich Tests, umzugehen ist.

1.1 MOTIVATION

Für einen reibungslosen Migrationsprozess einer Cross-Plattform-Anwendung ist es erstrebenswert, bestehende Infrastrukturen weitestgehend mitzunehmen, die derzeit auf Flutter basieren, jedoch für eine Migration zu React Native gleichermaßen von Bedeutung sind. Die Motivation für diese Arbeit entspringt dem Forschungsprojekt DDiT, das ursprünglich in Flutter entwickelt wurde, nun aber aufgrund verschiedener Faktoren auf React Native migriert werden soll [Ddi].

Die unterschiedlichen Anwendungsfälle der zu migrierenden Anwendung sollten bereits für die vorherige Anwendung spezifiziert worden sein. Um die Funktionalität dieser Anwendungsfälle zu überprüfen, wird die Anwendung üblicherweise umfangreichen Tests unterzogen. Diese Testumgebungen sind häufig komplex konfiguriert oder basieren auf dem verwendeten Cross-Platform-Framework, was die Migration solcher Tests erschwert.

1.2 ZIEL DER ARBEIT

Ziel dieser Arbeit ist die Ermittlung eines geeigneten End-to-End-Test-Frameworks, das ohne Anpassungen der Tests für Anwendungen verwendet werden kann, die sowohl in Flutter als auch in React Native entwickelt wurden. Zu diesem Zweck wird eine exemplarische Anwendung, die parallel in Flutter und React Native erstellt wurde, als Basis herangezogen, um End-to-End-Tests in den jeweiligen Test-Frameworks zu verfassen und zu analysieren. Die Er-

gebnisse dieses Praxisbeispiels werden mittels einer Bewertungsmatrix evaluiert, deren Kriterien und Gewichtungen im Verlauf der Untersuchung definiert werden. Am Ende der Arbeit soll eine fundierte Entscheidung über ein End-to-End-Test-Framework getroffen werden, das zur Validierung der Funktionalität im Kontext einer Migration von Flutter zu React Native geeignet ist.

1.3 GLIEDERUNG DER ARBEIT

Die Arbeit beginnt mit der Erläuterung relevanter Begriffe und einer umfassenden Untersuchung des aktuellen Forschungsstands.

Der dritte Abschnitt der Arbeit widmet sich dem Konzept. Hier wird zunächst die Anwendung beschrieben, die sowohl in Flutter als auch in React Native implementiert wird. Es folgt die Entwicklung eines Kriterienkatalogs zur Bewertung der Test-Frameworks, einschließlich der Gewichtung der einzelnen Kriterien. Im Weiteren werden konkrete Testszenarien auf Grundlage definierter Use Cases entwickelt.

Im vierten Abschnitt wird die Implementierung der End-to-End-Tests in den Test-Frameworks Maestro und Appium detailliert beschrieben. Dies umfasst die Installation, Konfiguration, Testimplementierung und Durchführung der Tests.

Im fünften Abschnitt werden die Ergebnisse präsentiert. Die Auswertung erfolgt nach den zuvor definierten Kriterien und die Ergebnisse werden in einer Bewertungsmatrix zusammengefasst. Hierdurch wird eine fundierte Entscheidung über das am besten geeignete Test-Framework ermöglicht. Aufbauend darauf werden mögliche Empfehlungen für die Auswahl des Test-Frameworks formuliert.

Die Arbeit endet mit einem Fazit, in dem die wesentlichen Erkenntnisse zusammengefasst und ein Ausblick auf zukünftige Forschungsarbeiten gegeben wird.

GRUNDLAGEN

Die vorliegende Arbeit untersucht und vergleicht die Entwicklung von End-to-End-Tests in ausgewählten Test-Frameworks für eine Cross-Plattform Anwendung, die ursprünglich mit Flutter entwickelt wurde und für eine Migration zu React Native vorgesehen ist. Dieses Kapitel stellt die theoretischen Grundlagen dar, die für das Verständnis und die Bearbeitung dieser Fragestellung notwendig sind.

2.1 CROSS-PLATTFORM-FRAMEWORKS

Im nachfolgenden Abschnitt werden die für diese Arbeit relevanten Cross-Plattform-Frameworks detailliert vorgestellt und ihre Funktionsweisen dargelegt. Daraufhin erfolgt eine zusammenfassende Darstellung der wesentlichen Gemeinsamkeiten und Unterschiede dieser Frameworks.

2.1.1 *Flutter*

Flutter ist ein von Google entwickeltes plattformübergreifendes Framework, das die Entwicklung von Anwendungen für verschiedene Endgeräte wie Android, iOS, Web und Desktop auf der Grundlage einer einzigen Codebasis ermöglicht [Flue]. Dies wird durch die plattformunabhängige Programmiersprache Dart realisiert, die sowohl einen Just-In-Time (JIT) Interpreter für einen schnellen Entwicklungszyklus, als auch die Möglichkeit einer Ahead-Of-Time (AOT) Kompilierung für performante Anwendungen bietet [Flud]. Um das Rendern der Benutzeroberfläche zu realisieren, nutzt Flutter die in C/C++ geschriebene Grafik-Engine Skia, welche bereits in Android, ChromeOS oder dem Browser Google Chrome verwendet wird [Fluf]. Im Laufe der Zukunft soll diese durch die Grafik-Engine Impeller ersetzt werden und ist bereits der Status Quo für das Rendern der Benutzeroberfläche auf iOS Geräten. Impeller bietet im Vergleich zu Skia mehrere Vorteile, einschließlich vorhersehbarer Leistung durch die Vorabkompilierung von Shadern, Nutzung moderner Grafik-APIs, effizientem Ressourcenmanagement und Vorbereitung auf zukünftige 3D-Fähigkeiten. Dies soll in flüssigeren Animationen und stabileren Bildraten resultieren [Tea23].

2.1.1.1 *Konzept*

In Flutter sind Widgets die grundlegenden Bausteine der Benutzeroberfläche. Jedes visuelle, wie Texte oder Bilder, strukturelle, wie Container oder Styling, oder interaktive Element in einer Flutter-Anwendung ist ein Widget. Widgets definieren, wie das Aussehen und Verhalten des User Interface (UI)

```

class ExampleWidget extends StatelessWidget {
  const ExampleWidget({super.key});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Example App'),
      ),
      body: Center(
        child: Text('Hello, World!'),
      ),
    );
  }
}

```

Quellcode 1: Flutter Widget Beispiel

ist und können entweder statisch oder dynamisch sein [Fluh]. Es gibt zwei Haupttypen von Widgets: **StatefulWidget**s und **StatelessWidget**s [Flua].

1. **StatelessWidgets**: Diese Widgets sind unveränderlich, d.h., sie repräsentieren eine Benutzeroberfläche, die sich nicht ändert. Ein Beispiel für ein StatelessWidget könnte ein einfacher Text oder ein Bild sein, das sich nicht ändert.
2. **StatefulWidget**: Diese Widgets können sich im Laufe der Zeit ändern. Sie enthalten einen internen Zustand, der dynamisch aktualisiert werden kann. Ein Beispiel für ein StatefulWidget ist ein Textfeld, dessen Inhalt sich basierend auf Benutzereingaben ändert.

Dabei sind die Widgets hierarisch in einem sogenannten Widget-Tree angeordnet. Jedes Widget innerhalb dieses Baums repräsentiert ein Element der Benutzeroberfläche und wird durch eine `build`-Methode konstruiert, die eine hierarchische Anordnung von weiteren Widgets zurückgibt [Flug]. Dieser Baum beginnt mit einem Root Widget, das die gesamte Anwendung darstellt und in Eltern-Kind-Beziehungen gegliedert ist. Wenn sich der Zustand eines Widgets mittels der `setState` Methode eines StatefulWidget ändert, wird die `build`-Methode erneut aufgerufen, um den Widget-Tree zu aktualisieren, wobei nur die betroffenen Teile, also die Kinder des StatefulWidget, neu gerendert werden. Diese Struktur ermöglicht eine effiziente Darstellung der Benutzeroberfläche in Flutter-Anwendungen [Flua].

2.1.2 React Native

React Native ist eine Open-Source-Bibliothek, die von Meta (früher Facebook) entwickelt wurde. Diese ermöglicht die Entwicklung nativer mobiler

React Native Komponente	Android View	iOS View	Web
<View>	<ViewGroup>	<UIView>	<div>
<Text>	<TextView>	<UITextView>	<p>
<Image>	<ImageView>	<UIImageView>	
<TextInput>	<EditText>	<UITextField>	<input type="text">

Tabelle 2.1: React Native Komponenten

Apps für Android und iOS. React Native basiert hierbei auf React, einer beliebten Open-Source-Bibliothek zur Erstellung von Benutzeroberflächen mit JavaScript [Reak].

2.1.2.1 Konzept

It's views all the way down!

In einer React Native-Anwendung ist die grundlegende Einheit ein sogenanntes **View**, ein rechteckiges Element, das Text, Bilder oder Benutzereingaben darstellen oder darauf reagieren kann. Diese Views bilden die grundlegenden Bausteine einer React Native-Anwendung [Reac].

Während der Laufzeit einer React Native-Anwendung werden diese Views in native Komponenten (siehe Tabelle 2.1) umgewandelt.

Um weiter zu verstehen, wie React Native funktioniert, ist es ratsam, sich mit dem darunterliegenden React-Framework auseinanderzusetzen.

React basiert auf einer komponentenbasierten Architektur, die es Entwicklern ermöglicht, wiederverwendbare Komponenten zu erstellen. Eine Komponente in React ist eine JavaScript-Funktion, die ein React-Element zurückgibt und zur Definition der Benutzeroberfläche verwendet wird [Reai]. Ein zentrales Merkmal von React ist der virtuelle DOM, der Performance-Optimierungen durchführt, indem er nur die tatsächlich geänderten Teile des DOM aktualisiert. Der virtuelle DOM ist ein Konzept, das eine leichte Kopie des realen DOM darstellt. Änderungen werden zunächst am virtuellen DOM vorgenommen und anschließend effizient auf das reale DOM übertragen. Dies minimiert die Anzahl der notwendigen Änderungen und erhöht die Performance der Anwendung [Reaf; Reah].

JSX, eine Erweiterung von JavaScript, erlaubt das Einbetten von HTML-ähnlichem Syntax direkt in JavaScript-Code und wird dazu verwendet, die Benutzeroberfläche innerhalb einer Komponente zu definieren [Reag]. React folgt einem unidirektionalen Datenfluss, was die Datenverwaltung einfacher und vorhersehbarer macht [Ream]. Mit der Einführung von Hooks in React können Funktionskomponenten nun State und andere React-Funktionalitäten nutzen, was zur Reduktion der Komplexität von Komponenten beiträgt [Reae]. Das umfangreiche Ökosystem von React bietet zusätzliche Werkzeuge wie React Router und Redux, die die Entwicklung und Verwaltung von Anwendungen erleichtern [Reaj]. Insbesondere ermöglicht Redux als globale State-Management-Erweiterung, dass mehrere Komponenten auf einen zentralisierten globalen Zustand zugreifen und diesen verwalten können [Red].

```

import React from 'react';
import {Text, View} from 'react-native';

const YourApp = () => {
  return (
    <View
      style={{
        flex: 1,
        justifyContent: 'center',
      }}>
      <Text>Try editing me!</Text>
    </View>
  );
};

export default YourApp;

```

Quellcode 2: React Native – Komponente Beispiel

Darüber hinaus ist es wichtig, die Funktionsweise der Kommunikation zwischen React Native und nativen Modulen zu verstehen. Die JavaScript-Bridge fungiert hierbei als Vermittler für den Datenaustausch zwischen der JavaScript-Umgebung und den nativen Plattformen (iOS/Android). Beispielsweise wird bei einem JavaScript-Ereignis wie einem Button-Klick eine Nachricht über die Bridge an den nativen Code gesendet, welcher dann die entsprechende native Methode ausführt [Reab].

Aspekt	React Native	Flutter
Programmiersprache	JavaScript	Dart
Komponenten	Native Komponenten	Eigene Widgets, die direkt gerendert werden
Rendering Engine	Verwendet native UI-Elemente	Skia Graphics Engine
Ausführung	Läuft über eine JavaScript-Bridge	Direkte Kompilierung in nativen Code
Native Module Integration	Zugriff auf native Module über JavaScript Brücken	Direkter Zugriff auf native APIs und SDKs

Tabelle 2.2: Unterschiede – Flutter und React Native

2.1.3 Gemeinsamkeiten und Unterschiede

Sowohl Flutter als auch React Native sind dazu konzipiert, auf möglichst vielen Plattformen ausgeliefert zu werden. Dabei verfolgen beide Frameworks einen ähnlichen konzeptionellen Ansatz, indem sie reaktive Programmiermodelle verwenden. In Flutter wird hierfür das `StatefulWidget` eingesetzt, welches in der Lage ist, einen eigenen Zustand zu verwalten und auf dieser Basis Änderungen an der Benutzeroberfläche vorzunehmen. Im Gegensatz

dazu verwendet React ein Hook-System, welches es einer Komponente ermöglicht, ihren eigenen Zustand zu verwalten. In dieser Hinsicht weisen beide Frameworks eine wesentliche Gemeinsamkeit auf. Zudem gibt es in beiden Frameworks die Möglichkeit des Hot Reloads, die es Entwicklern ermöglicht, Änderungen am Code sofort zu sehen, ohne die App neu zu starten. Die wesentlichen Unterschiede werden in der [Tabelle 2.2](#) aufgelistet.

2.2 TESTS

Das Testen in der App-Entwicklung stellt einen fundamentalen Aspekt der Qualitätssicherung dar. Insbesondere im Kontext einer Migration von Flutter zu React Native ist es entscheidend, die Konsistenz und Integrität der Funktionalität der Anwendung zu gewährleisten. Hierbei kommt dem Testen eine zentrale Bedeutung zu.

Testen ist wie folgt definiert:

„Testen umfasst Verifikations-, Validierungs- und Explorationsaktivitäten, die dazu dienen, Informationen über die Qualität und die damit verbundenen Risiken eines Testobjekts zu sammeln. Ziel ist es, ein Maß an Vertrauen zu schaffen, dass das Testobjekt den angestrebten Geschäftswert liefern kann.“¹ [Tma]

Das Testen von Software dient dazu, die Übereinstimmung der Software mit den Anforderungen sowie die korrekte Implementierung der Funktionalität zu überprüfen und damit die Softwarequalität zu gewährleisten. Dabei ist das Testen eine sowohl konstruktive als auch destruktive Aktivität: Einerseits wird die Korrektheit der Software verifiziert, andererseits werden gezielt so viele Fehler wie möglich identifiziert. Dabei kann die Anwendung auf verschiedene Arten getestet werden. Zum einen *manuell*, welches auf der direkten Interaktion mit der Anwendung basiert. Jene Tests sind zwar durchführbar, jedoch in ihrer Skalierbarkeit begrenzt und neigen dazu, Regressionen zu übersehen.

Zum anderen *automatisierte* Tests, welche hingegen Tools verwenden, die diese Tests durchführen. Sie sind schneller, wiederholbar und liefern in der Regel früher im Entwicklungsprozess verwertbares Feedback zur Anwendung [Andd].

Das Testen von Software umfasst zudem die Definition und Gestaltung von Testumgebungen sowie die Durchführung der Tests. Die Analyse der Testergebnisse soll sicherstellen, dass die Software bereit für die Veröffentlichung ist [O9].

Im Folgenden werden die verschiedenen Testarten detaillierter beschrieben und ihre jeweiligen Ziele erläutert.

¹ Testing consists of verification, validation and exploration activities that provide information about the quality and the related risks, to establish the level of confidence that a test object will be able to deliver the pursued business value.

2.2.1 Unit Test

Unit Testing ist eine Testtechnik, die sich im Fokus auf einzelne Funktionalität von Modulen beschränkt. Das Ziel von Unit-Tests ist, zu überprüfen, ob einzelne Komponenten der Software wie vorgesehen arbeiten und Fehler frühzeitig zu erkennen. Ein Modul sollte der kleinste testbare Teil einer Software sein. Dabei wird ein Modul isoliert getestet, meist mithilfe von Stubs (Services oder Hilfsfunktionen, welche die eigentliche Implementierung simulieren) oder gemockte Daten [Lel23].

2.2.2 Integration Test

Integration Testing fokussiert sich beim Testen auf das Zusammenspiel verschiedener Komponenten oder Modulen einer Software. Das Ziel von Integration Testing ist das Verifizieren, dass die individuellen Komponenten fehlerfrei miteinander arbeiten [Lel23].

2.2.3 System Testing

Das System Testing hat als Ziel, das komplette, integrierte System auf bestimmte Kriterien zu testen. Dabei wird das System als Ganzes getestet und auf funktionale und nicht-funktionale Kriterien evaluiert [Lel23].

2.2.4 User Acceptance Testing

User Acceptance Testing ist ein Prozess, welcher sicherstellen soll, dass das System die spezifizierten Kriterien in einem realen Szenario erfüllt [Lel23].

Um Apps auf deren Funktion zu testen, ohne deren Implementierung zu kennen, wird sich in dieser Arbeit auf End-to-End-Tests fokussiert werden. Diese sind eine Art von Integration Tests, die aus Nutzersicht die Anwendung auf deren Funktion testet. Im Weiteren wird nochmals genauer dargelegt, wie End-to-End-Tests zu definieren sind.

2.2.5 End-to-End-Test

End-to-End-Tests sind eine Art der Black Box Tests, basierend auf einem Testszenario, welches verschiedene Instruktionen und Schritte innerhalb der Anwendung, in der Regel oberflächlich (auf dem Graphical User Interface (GUI)) ausführt, wie bspw. die Eingabe von Nutzerdaten in einem Login Fenster. Wie in Black Box Tests üblich, ist die innere Implementierung und Funktionsweise der Anwendung nicht bekannt. Dabei werden diese mit bereits vordefinierten Daten (bspw. `username=johndoe`) ausgeführt und einem erwarteten Ausgang und mit dem tatsächlichen Ausgang verglichen [Leo+16].

2.2.6 Visual Regression Testing

Ein weiterer wichtiger Aspekt der End-to-End-Tests, der auch in dieser Untersuchung von Relevanz ist, ist das Visual Regression Testing. Visual Regression Testing stellt sicher, dass nach Änderungen in der Anwendung, wie beispielsweise Codeänderungen, keine neuen Fehler eingeführt wurden oder Elemente falsch dargestellt werden [Gitc]. Dies wird üblicherweise durch den Vergleich von Screenshots der Anwendung erreicht. Der Vergleich basiert auf einer Baseline, die den gewünschten Zustand der UI repräsentiert, und einem aktuellen Screenshot der Anwendung. Dieser Vergleich wird häufig mit Tools durchgeführt, die die Bilder auf Pixelebene analysieren und Unterschiede identifizieren [ATY20]. In Flutter ist ein solches Tool im `golden_toolkit` integriert. Dabei wird eine `Golden File`, die den gewünschten Zustand der Benutzeroberfläche darstellt, mit dem aktuellen Zustand der Anwendung verglichen [Pub].

2.3 END-TO-END-TEST-FRAMEWORKS

Um die Frage zu beantworten, wie End-to-End-Tests für eine Cross-Plattform-Anwendung, die ursprünglich auf Flutter entwickelt und auf React Native migriert werden soll, so gestaltet werden können, dass sie für beide Frameworks gleichermaßen anwendbar sind, müssen spezifische Test-Frameworks in Betracht gezogen werden. Diese Frameworks sollten plattformunabhängig arbeiten und eine flexible Anpassung an die unterschiedlichen Architektur- und Implementierungsunterschiede zwischen Flutter und React Native ermöglichen. Dazu werden in diesem Abschnitt die in Betracht gezogenen Test-Frameworks vorgestellt, beschrieben und evaluiert.

2.3.1 Appium

Appium ist ein Open-Source-Framework zur Automatisierung von Tests mobiler Anwendungen. Es ermöglicht die plattformübergreifende Automatisierung von Testfällen sowohl für Android- als auch für iOS-Betriebssysteme sowie für Webanwendungen im Browser. Im Gegensatz zu zahlreichen anderen Automatisierungswerkzeugen fokussiert sich Appium darauf, die native Benutzeroberfläche von Anwendungen zu testen, indem es direkt mit den nativen UI-Komponenten interagiert. Appium basiert auf dem Client-Server-Architekturmödell. Der Appium-Server ist in Node.js geschrieben und kann auf einer Vielzahl von Plattformen ausgeführt werden. Der Server akzeptiert Verbindungen von einem Client, empfängt Testanforderungen, die in JSON über das WebDriver-Protokoll formuliert sind, und führt diese Anforderungen dann auf einem mobilen Gerät aus, indem er die entsprechenden Automatisierungs-Frameworks verwendet [Appd].

Die zentrale Komponente von Appium ist der Appium Server, der als HTTP-Server fungiert. Dieser Server empfängt Automatisierungsbefehle vom Client und leitet sie an die entsprechenden Plattformen weiter, indem er sage-

nannte Driver für Android oder iOS verwendet. Diese Driver interagieren direkt mit der zu testenden Anwendung [Appc]. Es existieren Implementierungen des Clients in verschiedenen Programmiersprachen, was das Schreiben von Testfällen in einer Vielzahl von Programmiersprachen ermöglicht, wie Java oder JavaScript [Appa].

Ein Test, der im WebdriverIO-Client unter Verwendung des Test-Frameworks Mocha implementiert ist, könnte folgendermaßen aussehen:

```
const {remote} = require('webdriverio');

const capabilities = {
    platformName: 'Android',
    'appium:automationName': 'UiAutomator2',
    'appium:deviceName': 'Android',
    'appium:appPackage': 'com.android.settings',
    'appium:appActivity': '.Settings',
};

const wd0pts = {
    hostname: process.env.APPIUM_HOST || 'localhost',
    port: parseInt(process.env.APPIUM_PORT, 10) || 4723,
    logLevel: 'info',
    capabilities,
};

async function runTest() {
    const driver = await remote(wd0pts);
    try {
        const batteryItem = await driver.$('//*[@text="Battery"]');
        await batteryItem.click();
    } finally {
        await driver.pause(1000);
        await driver.deleteSession();
    }
}

runTest().catch(console.error);
```

Quellcode 3: Appium – Beispiel [Webc]

2.3.2 Maestro

Maestro ist ein relativ neues Testframework [Gitb], welches keine großen Konfigurationen benötigt und auf den üblichen mobilen Endgeräten, wie bspw. Android und iOS, anwendbar ist. Dabei beschreibt sich Maestro selbst wie folgt:

“

- Eingebaute Toleranz gegen *Flakiness*². UI-Elemente sind nicht immer dort, wo du sie erwartest, Benutzereingaben funktionieren nicht immer wie geplant usw. Maestro toleriert die Instabilität von mobilen Anwendungen und Geräten und versucht, ihr entgegenzuwirken.
- Eingebaute Toleranz gegen Verzögerungen. Keine Notwendigkeit, deine Tests mit `sleep()` Aufrufen zu versehen. Maestro weiß, dass das Laden von Inhalten (z. B. über das Netzwerk) Zeit in Anspruch nehmen kann, und wartet automatisch darauf (aber nicht länger als nötig).
- Extrem schnelle Iterationen. Tests werden interpretiert, es ist keine Kompilierung erforderlich. Maestro kann deine Testdateien kontinuierlich überwachen und sie bei Änderungen neu ausführen.
- Deklarative und dennoch leistungsstarke Syntax. Definiere deine Tests in einer YAML-Datei. Einfache Einrichtung. Maestro ist ein einzelnes Binär, das überall funktioniert.

“ [Mofb]

Maestro operiert in einer Weise, die Appium ähnelt. Es nutzt ein Command Line Interface ([CLI](#)), um einen Client zu starten, der die in einer YAML-Datei definierten Befehle ausliest. Dieser Client übermittelt die Befehle an einen Driver. Im Fall von Android kommt die Android Debug Bridge (ADB) zum Einsatz, um eine Verbindung zum Emulator oder physischen Gerät herzustellen, während für iOS XCUITest verwendet wird. Diese Systeme führen die Befehle auf dem jeweiligen Gerät aus und übermitteln die Ergebnisse zurück an den Maestro-Client. Der Client analysiert die Resultate und zeigt sie schließlich in der CLI an [[Gitb](#)].

Die Installation von Maestro kann mittels eines Befehls über die Konsole installiert werden [[Mobb](#)]. Zudem wird das Maestro Studio angeboten, eine grafische Benutzeroberfläche zur Erstellung von Testabläufen [[Mobc](#)], und die Möglichkeit, Tests in der Cloud auszuführen, was die Notwendigkeit lokaler Emulatoren oder Simulatoren eliminiert und parallele Testausführungen sowie detaillierte Anwendungsprotokolle ermöglicht [[Mobe](#)].

Eine Testdatei in Maestro kann wie folgt aussehen:

² Flakiness – Unvorhersehbarkeit

```
appId: com.android.contacts
---
- launchApp
- tapOn: "Create new contact"
- tapOn: "First Name"
- inputText: "John"
- tapOn: "Last Name"
- inputText: "Snow"
- tapOn: "Save"
```

Quellcode 4: Maestro – Beispiel [Mobf]

2.3.3 Detox

Detox wurde speziell für mobile Anwendungen entwickelt, insbesondere für solche, die auf React Native basieren [Wixb]. Es ermöglicht die automatische Ausführung von Tests auf physischen Geräten oder Emulatoren, wobei reale Benutzerinteraktionen simuliert werden. Ein weiterer wesentlicher Aspekt von Detox ist seine plattformübergreifende Unterstützung, da sowohl iOS- als auch Android-Plattformen abgedeckt werden [Wixd].

Detox verwendet einen Tester, einen nativen Client und eine WebSocket-Verbindung zur Durchführung seiner Funktionen. Der Tester, der in einem Node.js-Prozess ausgeführt wird, ist für die Implementierung der Testlogik, die Verwaltung der Geräte und die Sammlung von Artefakten zuständig. Der native Client, verfügbar für iOS und Android, wird in die zu testende Applikation integriert. Dieser Client synchronisiert sich mit der Applikation, führt Benutzereingaben durch und validiert die erwarteten Ergebnisse. Die WebSocket-Verbindung, die ebenfalls in einem Node.js-Prozess betrieben wird, ermöglicht die Kommunikation zwischen Tester und Client. Diese Verbindung wird durch eine zufällig generierte Sitzungs-ID und einen verfügbaren Port initialisiert und anschließend an die Client-App übermittelt [Wixc]. Die Tests in Detox werden in JavaScript geschrieben und basiert auf dem JavaScript-Framework Jest. Während der Testausführung kann Detox Screenshots aufnehmen und detaillierte Protokolle erstellen [Wixe].

Eine typische Testdatei in Detox könnte wie folgt aussehen:

```
describe('Login flow', () => {
  beforeEach(async () => {
    await device.reloadReactNative();
  });

  it('should login successfully', async () => {
    await element(by.id('email')).typeText('john@example.com');
    await element(by.id('password')).typeText('123456');

    const loginButton = element(by.text('Login'));
    await loginButton.tap();

    await expect(loginButton).not.toExist();
    await expect(element(by.label('Welcome'))).toBeVisible();
  });
});
```

Quellcode 5: Detox – Beispiel [Wixb]

2.3.4 Flutter Integration Test Package

Dies ist das interne Test-Framework von Flutter. Hier simuliert ein *Driver* Benutzereingaben in der bereits kompilierten nativen Anwendung auf dem jeweiligen Endgerät.

Der Testprozess beginnt mit der Initialisierung des `Test Drivers`, einer Dart-Datei, die die Testumgebung konfiguriert und startet. Die `IntegrationTestWidgetsFlutterBinding`-Klasse erweitert das herkömmliche `WidgetsFlutterBinding`, um spezielle Funktionen für Integrationstests bereitzustellen. Dies wird durch den Aufruf von

`IntegrationTestWidgetsFlutterBinding.ensureInitialized()` aktiviert. Während der Tests werden Benutzerinteraktionen wie Tippen und Scrollen mittels Methoden des `WidgetTester`-Objekts simuliert [Flub; Fluc].

Eine Testdatei könnte wie folgt aussehen:

```
import 'package:flutter/material.dart';
import 'package:flutter_test/flutter_test.dart';
import 'package:integration_test/integration_test.dart';
import 'package:introduction/main.dart';

void main() {
  IntegrationTestWidgetsFlutterBinding.ensureInitialized();

  group('end-to-end test', () {
    testWidgets('tap on the floating action button, verify counter',
        (tester) async {
      // Load app widget.
      await tester.pumpWidget(const MyApp());

      // Verify the counter starts at 0.
      expect(find.text('0'), findsOneWidget);

      // Finds the floating action button to tap on.
      final fab = find.byKey(const Key('increment'));

      // Emulate a tap on the floating action button.
      await tester.tap(fab);

      // Trigger a frame.
      await tester.pumpAndSettle();

      // Verify the counter increments by 1.
      expect(find.text('1'), findsOneWidget);
    });
  });
}
```

Quellcode 6: Flutter Integration Test Package – Beispiel [Flub]

End-to-End-Test-Framework	Geeignet für beide Frameworks	Grund
Appium	Ja	Unterstützung für Flutter und React Native
Maestro	Ja	Unterstützung für Flutter und React Native
Detox	Nein	Konzipiert für React Native und nicht verwendbar für andere Anwendungen
Flutter Integration Test Package	Nein	Konzipiert ausschließlich für Flutter

Tabelle 2.3: Gegenüberstellung – Auswahl der Frameworks

2.3.5 Auswahl der Kandidaten

Appium und Maestro sind beide darauf ausgerichtet, plattformübergreifende Tests für mobile Anwendungen auf Android und iOS zu unterstützen. Appium basiert auf einer Client-Server-Architektur, bei der der Server Befehle an einen Driver sendet, der diese Befehle auf einem Emulator oder physischen Gerät ausführt. Testanforderungen werden im JSON-Format über das WebDriver-Protokoll empfangen und durch einen Driver umgesetzt. Maestro verwendet eine CLI-basierte Interaktion, bei der Tests in YAML-Dateien definiert werden. Diese Tests senden ebenfalls mittels eines Clients Befehle an einen Driver, der sie auf den jeweiligen Emulatoren oder Endgeräten ausführt, ähnlich zu Appium.

Detox und das Flutter Integration Test Package sind auf spezifische Anwendungsfälle ausgerichtet. Detox ist hierbei für React Native-Anwendungen optimiert. Das Flutter Integration Test Package ist in das Flutter-Ökosystem eingebunden und optimiert für das native Testen von Flutter-Widgets. Es operiert direkt in der kompilierten Anwendung auf dem jeweiligen Endgerät.

Alle genannten Frameworks unterstützen sowohl Android- als auch iOS-Plattformen und automatisieren Benutzereingaben sowie die Validierung von UI-Komponenten. Sie unterscheiden sich jedoch in ihrer Implementierung und den Konfigurationsanforderungen. Appium und Maestro bieten flexible, plattformübergreifende Lösungen, wobei Maestro durch seine einfache Konfiguration und schnellen Iterationszyklen hervorsticht. Detox und das Flutter Integration Test Package bieten spezialisierte Lösungen, wobei Detox auf die React Native-Integration fokussiert ist und das Flutter Integration Test Package eine nahtlose Testumgebung für Flutter-Anwendungen bereitstellt.

Die Auswahl der Testkandidaten in dieser Arbeit basiert auf deren Anwendbarkeit sowohl für in Flutter als auch in React Native kompilierte Anwendungen. Dabei wurden Appium und Maestro berücksichtigt, da beide Frameworks Unterstützung für Flutter und React Native bieten [Mobj; Moba; Appb]. Das Flutter Integration Test Package und Detox wurden hingegen von der Analyse ausgeschlossen, da ersteres ausschließlich für Flutter und letzteres zwar primär für React Native konzipiert ist, theoretisch jedoch auch für andere Anwendungen verwendet werden könnte. Zudem gibt es bei Detox noch Kompatibilitätsprobleme mit Android [Wixa] (siehe Tabelle 2.3).

3

RELEVANTE ARBEITEN

In diesem Abschnitt werden die relevanten wissenschaftlichen Arbeiten und Studien vorgestellt, die für das Verständnis und die Bearbeitung der Fragestellung dieser Arbeit von Bedeutung sind. Ziel ist es, einen Überblick über den aktuellen Stand der Forschung zu geben und aufzuzeigen, welche Erkenntnisse und Methoden bereits entwickelt wurden, die zur Lösung der in dieser Arbeit behandelten Problematik beitragen können.

In dieser Arbeit [Sai+21] wurde durch eine quantitative Studie herausgearbeitet, welche Hauptziele in GUI Tests verfolgt werden. Dabei werden diese vier Hauptziele genannt:

- Funktionalität
 - Die Anwendung funktioniert korrekt unter den Bedingungen der Spezifikation.
- Verlässlichkeit
 - Die Anwendung funktioniert fehlerfrei in einer gewissen Zeit unter bestimmten äußereren Faktoren.
- Leistung
 - Überprüft, ob die Anwendung unter höherer Last richtig funktioniert.
- Sicherheit
 - Die Anwendung stellt die Authentizität und Integrität der verwendeten Daten sicher.

In einer anderen Arbeit [Kon+18] werden Testziele und die zu testenden Events, jedoch nur für Android Anwendungen herausgearbeitet. Daraus ergeben sich folgende Testziele:

- Parallelität
 - In modernen mobilen Anwendungen ist es üblich, dass verschiedene Events asynchron, also parallel verlaufen. Dabei soll auf verschiedene Race Conditions geachtet werden.
- Sicherheit
 - Die Anwendung soll auf Datenlecks, fehlerhaftes und schadhaftes Verhalten gegenüber dem Nutzer überprüft werden.
- Performance

- Die Anwendung soll auf Reaktionszeit getestet werden und eventuelle Ausnahmebehandlungen überprüfen. Dazu kann in einem Test die Reaktionszeit bewusst verlängert werden, um solche Szenarien für teure Prozesse der Anwendung zu testen [YYR13].
- Energie
 - Da mobile Anwendungen in der Regel unter Batteriebetrieb laufen, der während der App-Nutzung verbraucht wird, sollen in Tests gewisse Energie Schwerpunkte gefunden werden. Dabei soll jede weitere Energienutzung, die über die bereits bekannte Energienutzung des mobilen Endgerätes hinaus geht, als ungewöhnlich betrachtet werden [Wan+15].
- Kompatibilität
 - Unter Umständen kann es passieren, dass durch Updates des Frameworks oder Nutzen externer Bibliotheken Inkompatibilitäten entstehen können.
- Bug/Defekt
 - Anwendung sind oftmals unbeabsichtigt fehlerhaft, durch neue Implementierungen oder nicht weitreichend konzipiertes Verhalten der Anwendung.

Weitergehend werden in dieser Arbeit folgende Events für Android Anwendungen bestimmt, die für GUI Tests relevant sind:

- GUI/Events
 - Android arbeitet auf einer Event-basierten Architektur, dass, wenn bspw. ein Knopf betätigt wird, ein weiteres Event ausgelöst wird.
- IAC/ICC
 - Inter-App Communication (IAC) oder Inter-Component Communication (ICC) sind Aktivitäten einer Android-Anwendung, die eine andere Anwendung (wie bspw. eine Karte) integrieren.

Eine weitere Arbeit [MB23] beschäftigt sich mit der Übersetzung von User Stories in End-to-End Testfälle. Dabei erarbeiten Sie ebenfalls die Best Practices für End-to-End Tests. Diese lassen sich wie folgt beschreiben:

- *Auf den wichtigsten Workflow fokussieren:*
Der wichtigste Workflow sollte als Erstes getestet werden. Zum Beispiel das Erfüllen einer Aufgabe in einer To-Do App. Hier sollte bspw. das Abhaken und dann Verschwinden der Aufgabe der auf Benutzeroberfläche getestet werden.
- *Große Workflows sollten in Kleinere aufgeteilt werden:*
Große und komplexe Workflows können sich zur Fehlersuche kompliziert gestalten. Daher sollte der Test in kleinere Schritte aufgeteilt, bis der ganze Workflow abgebildet wurde.

- *Low-Level Tests vermeiden:*

Wenn es um End-to-End Testing geht, sollte man möglichst vermeiden, einzelne Funktionalitäten von kleineren Modulen zu testen, um deren Funktionalität sicherzustellen. Um einzelne Module zu testen, eignen sich hierfür Unit-Tests.

- *Tests für alle möglichen Workflows:*

Eine Anwendung sollte auf möglichst allen Interaktionen des Nutzers getestet werden.

Weitere Arbeiten, wie [CM23] vergleichen Testframeworks im Kontext der CI. Dabei liegt der Fokus auf Testframeworks für mobile Anwendungen wie Apium, Detox, Maestro und Calabash. Diese werden anhand Kriterien, wie Popularität, Benutzerfreundlichkeit sowie die Dauer der Testausführung, miteinander verglichen. [AAS21] entwickeln ein mehrkriterielles Auswahlverfahren für Testframeworks, das unter anderem Benutzerfreundlichkeit und Integrationsfähigkeit, sowie die Funktionalität berücksichtigt. [Ama+17] bietet umfassende Frameworks zum Vergleich von Testtechniken für mobile Apps und Integrationstests, während [Che16] die Qualität der Testdaten als entscheidenden Erfolgsfaktor für End-to-End-Tests beschreibt. [SS21] vergleichen verschiedene Testframeworks hinsichtlich ihrer Robustheit gegenüber unvorhersehbaren Ereignissen während der Testausführung.

Obwohl es umfangreiche Arbeiten zu GUI-Tests und End-to-End-Tests für mobile Anwendungen gibt, fehlt eine spezifische Untersuchung darüber, wie diese Tests im Kontext einer Migration von einer Plattform zu einer anderen sinnvoll eingesetzt werden können. Besonders für Multi-Plattform-Frameworks wie Flutter und React Native existiert bislang keine spezifische Forschung, die End-to-End-Tests darauf ausrichtet, beide Plattformen zu unterstützen und eine Migration reibungslos zu gestalten. Jedoch können wir Erkenntnisse, wie die Hauptziele für GUI-Tests [Sai+21; Kon+18] sowie die Kriterien der Benutzerfreundlichkeit und der Integration von CI/CD Pipelines [CM23] nutzen, um einen Kriterienkatalog zu erstellen. Des Weiteren können Best Practices und der Leitfäden aus User Stories geeignete Testfälle zu spezifizieren herangezogen werden [MB23].

Um den Umfang dieser Arbeit nicht zu überschreiten, konzentriert sich diese Untersuchung im Wesentlichen auf die *funktionalen Ziele*. Der Schwerpunkt liegt dabei auf der Überprüfung der Funktionalität und Zuverlässigkeit spezifischer Anforderungen der Anwendung. *Nicht-funktionale Ziele*, wie *Sicherheit, Leistung und Parallelität*, werden in dieser Untersuchung nicht berücksichtigt.

4

KONZEPT

In diesem Abschnitt wird die Anwendung vorgestellt, die sowohl mit Flutter als auch mit React Native entwickelt wurde. Zunächst wird ein umfassender Überblick über die Anwendung gegeben, einschließlich ihrer Funktionalität und ihrer wichtigsten Merkmale. Dies ermöglicht eine fundierte Betrachtung der verschiedenen Aspekte, die für die Durchführung von Tests relevant sind. Des Weiteren werden die Implementierungen in den jeweiligen Frameworks kurz erläutert, wobei auf spezifische Unterschiede und Besonderheiten eingegangen wird.

Das Ziel dieses Abschnitts besteht darin, ein umfassendes Verständnis der Funktionalität der Anwendung zu vermitteln und eine solide Grundlage für die Durchführung von End-to-End-Tests zu schaffen.

Im Anschluss daran wird der Kriterienkatalog erstellt, der auf relevanten Literatur und den spezifischen Anforderungen der Anwendung basiert. Dieser Katalog dient als Leitfaden für die Bewertung der Testfähigkeit der Frameworks und enthält Kriterien, die die wichtigsten Aspekte der Anwendung und ihrer Funktionalität abdecken. Aus diesem Kriterienkatalog wird zusätzlich eine gewichtete Bewertungsmatrix erstellt, die für die Evaluation der Ergebnisse relevant ist.

Darüber hinaus werden Use Cases und deren Testszenarios definiert, um sicherzustellen, dass die Testabdeckung alle wesentlichen Funktionen und Interaktionen der Anwendung umfasst. Dies ermöglicht eine systematische Durchführung von Tests, die sicherstellt, dass alle relevanten Aspekte der Anwendung auf ihre Funktionalität und Leistungsfähigkeit überprüft werden.

4.1 ANWENDUNG

4.1.1 *Grundidee*

Die exemplarische Anwendung soll die Fähigkeit besitzen, einen begrenzten Nutzerfluss zu orchestrieren. Eine einfache To-do-Anwendung bietet eine geeignete Grundlage für dieses Vorhaben.

Das Erstellen und Abhaken von Aufgaben stellt dabei einen zentralen Nutzerfluss dar, der sowohl effizient als auch intuitiv gestaltet sein sollte. Um die Funktionalität der Anwendung zu erweitern und anspruchsvollere Anforderungen zu testen, erscheint es sinnvoll, die Benutzeroberfläche um komplexe Eingabemöglichkeiten zu ergänzen.

Ein Canvas, der dem Benutzer die Möglichkeit bietet, handschriftliche Notizen anzufertigen oder frei zu zeichnen, stellt sich als ideale Lösung für diese

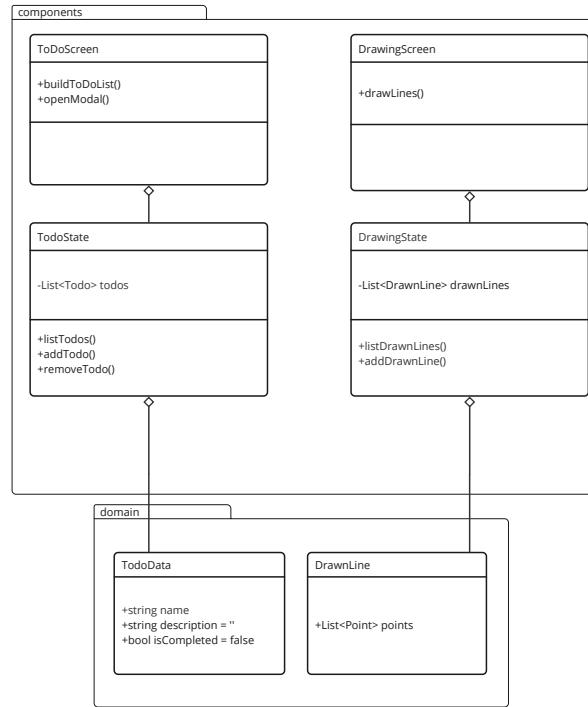


Abbildung 4.1: UML-Diagramm – Anwendung

Anforderung dar. Zur Verdeutlichung der Domäne und der Abhängigkeiten wurde ein UML-Diagramm erstellt (siehe Abbildung 4.1).

4.1.2 Design

Die Anwendung soll das Designsystem *Material 3* als Grundlage verwenden. Diese Implementierung sieht vor, die Anwendung in zwei separate Ansichten zu unterteilen, zwischen denen der Nutzer mithilfe einer Navigation-Bar navigieren kann.

Der Homescreen soll eine Liste von To-dos anzeigen. Diese werden mithilfe der *ListTile*-Komponente von Material 3 in einer Listenansicht präsentiert, wobei jede Aufgabe eine Checkbox zum An- oder Abhaken besitzt. Um eine Aufgabe zu löschen, soll der Benutzer sie nach links wischen können.

Ein weiterer bedeutender Aspekt ist das Hinzufügen neuer Aufgaben. Hierfür soll eine Schaltfläche am unteren rechten Bildschirmrand platziert werden, die es dem Benutzer ermöglicht, nötige Informationen für die zu erstellende Aufgabe mithilfe eines Dialogs zur Liste hinzuzufügen (siehe Abbildung 4.2a).

Im zweiten Ansichtsmodus, dem *Zeichen-Screen*, soll es dem Benutzer möglich sein, mit dem Finger zu zeichnen oder zu schreiben (siehe Abbildung 4.2b).

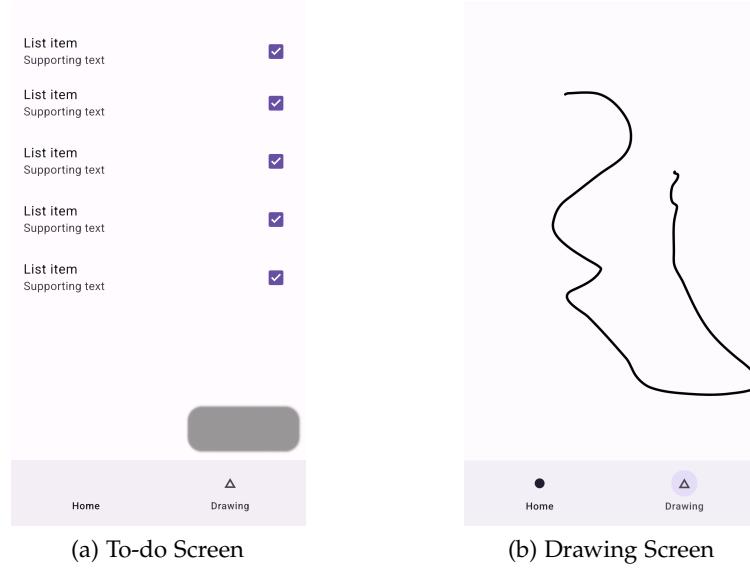


Abbildung 4.2: Wireframes für die zu verwendete Anwendung

4.1.3 Implementierung in Flutter

Flutter wird standardmäßig mit dem Designsystem Material 3 ausgeliefert, welches vordefinierte Widgets bereitstellt, die für die Implementierung nützlich sind.

In dem vorliegenden Flutter-Projekt wird eine Startseite als Grundlage für die Navigation zwischen verschiedenen Screens verwendet. In der `build`-Methode der Klasse `MyApp` wird das `MaterialApp`-Widget als Wrapper verwendet, um sicherzustellen, dass die darin enthaltenen Widgets einen konsistenten Stil nach dem Material Designsystem aufweisen. Die Klasse `MyHomePage` wird als Startscreen innerhalb der `MyApp`-Klasse verwendet. Diese Klasse erweitert `StatefulWidget`, wodurch sie den UI-Zustand verwalten kann, was entscheidend für die Navigation ist.

Um den UI-Zustand zu verwalten, wird eine separate Klasse benötigt, die den Status des Widgets definiert. In diesem Fall initialisiert die Klasse `_MyHomePageState` die Variable `currentIndex` mit dem Wert 0. In der `build`-Methode dieser Klasse wird der Widget-Baum weiter ausgebaut. In diesem Fall wird ein `IndexedStack` verwendet, um zwischen verschiedenen Widgets zu navigieren.

Die `bottomNavigationBar` ermöglicht es, den Zustand des aktuellen Widgets zu manipulieren, indem der `currentIndex` auf einen bestimmten Wert gesetzt wird. Da in diesem Fall nur zwischen zwei Widgets navigiert wird, gibt es nur zwei mögliche Werte für `currentIndex`: 0 und 1. Wenn sich `currentIndex` ändert, rendert `IndexedStack` das Widget mit dem entsprechenden Index.

4.1.3.1 To-do-Screen

Der To-do-Screen ist ebenfalls ein `StatefulWidget`, das einen anfänglichen Zustand initialisiert. In diesem Kontext werden die To-do-Elemente bei der Widget-Initialisierung geladen. Dies geschieht durch das Abrufen einer vordefinierten Menge von To-do-Daten, die in einer JSON-Datei gespeichert sind. Anschließend werden diese geladenen To-do-Elemente der Liste von To-dos im Widget-Status hinzugefügt.

4.1.3.2 Zeichen-Screen

Der Zeichen-Screen präsentiert einen Canvas, der die gesamte Bildschirmfläche einnimmt. Für das Zeichnen auf dem Canvas in Flutter wird eine Klasse benötigt, die den `CustomPainter` erweitert. Dieser ist verantwortlich für das Zeichnen auf dem Canvas. Der `CustomPainter` verfügt über zwei zu implementierende Funktionen:

`paint`: Diese Methode erhält den Canvas und die Größe des Canvas als Parameter und wird aufgerufen, wenn Zeichenaktionen auf dem Canvas stattfinden. Um den Bereich auf dem Screen zu identifizieren, auf dem gezeichnet wird, wird der `GestureDetector` verwendet. Dieser erfasst die Bildschirmkoordinaten basierend auf Benutzerinteraktionen, welche dann verwendet werden, um dem `CustomPainter` mitzuteilen, wo auf dem Canvas Linien gezeichnet werden sollen.

`shouldRepaint`: Diese Methode gibt einen Boolean-Wert zurück, der signalisiert, ob nach einem Widget-Update die `paint`-Methode erneut ausgeführt werden soll.

4.1.4 Implementierung in React Native

Das im Rahmen dieser Arbeit entwickelte React-Native-Projekt wurde unter Verwendung des Frameworks Expo realisiert. Expo bietet unter anderem ein dateibasiertes Routing sowie eine Standardbibliothek nativer Module und wird offiziell von React Native empfohlen [Read]. Zur Implementierung des Material 3 Designsystems in React Native wurde das separate Node-Package `React Native Paper` verwendet, welches vorgefertigte Komponenten gemäß dem Material 3 Designsystem zur Verfügung stellt, die in React Native integriert werden können [Real].

Des Weiteren nutzt das Projekt ebenfalls eine Navigation, die `BottomNavigation` von React Native Paper, welche die beiden Komponenten für den To-do-Screen und den Zeichen-Screen enthält. Die Änderung des aktuellen Screens wird durch einen lokalen Zustand erfasst und erfolgt bei einem Klick auf den entsprechenden Punkt der Navigationsleiste.

4.1.4.1 To-do-Screen

Der To-do-Screen hält selbst keinen eigenen Zustand, sondern greift auf einen globalen Zustand zu, der die To-do-Daten initialisiert und der Kom-

ponente zurückgibt. Dies geschieht durch das Abrufen einer vordefinierten Menge von To-do-Daten, die in einer JSON-Datei gespeichert sind. Anschließend baut die Komponente eine Liste von Aufgaben auf und gibt diese aus.

4.1.4.2 Zeichen-Screen

Der Zeichen-Screen präsentiert einen Canvas, der die gesamte Bildschirmfläche einnimmt. Für das Zeichnen auf dem Canvas wird ein `GestureHandler` benötigt, der Nutzereingaben aufnimmt und ermittelt, an welcher Stelle diese Nutzereingabe getätigt wurde. Für diesen Handler wird ein separates Package, `react-native-gesture-handler`, verwendet [Gite]. Die Koordinaten der Eingaben werden im Zustand der Komponente gespeichert und als Pfad auf dem Canvas gezeichnet.

4.1.5 Gemeinsamkeiten und Unterschiede

Sowohl in Flutter als auch in React Native wird das Material 3 Designsystem verwendet, jedoch unterscheiden sich die Implementierungen in einigen Aspekten. Flutter nutzt das Designsystem direkt, während React Native das Node-Package `React Native Paper` integriert, um Material 3 Komponenten zu verwenden. Beide Projekte implementieren eine Startseite zur Navigation zwischen verschiedenen Screens, wobei Flutter `MaterialApp` und `bottomNavigationBar` und React Native `BottomNavigation` von React Native Paper nutzt.

Der To-do-Screen in beiden Projekten lädt im Kern die gleichen To-do-Daten aus einer JSON-Datei und verwendet diese, um eine Liste von To-dos zu erstellen. Während Flutter dafür `StatefulWidget` nutzt und den Zustand innerhalb des Widgets verwaltet, greift React Native dabei auf einen globalen Zustand zurück mithilfe von `Redux`. Der Zeichen-Screen in beiden Implementierungen bietet einen Canvas, der die gesamte Bildschirmfläche einnimmt und zeichnen ermöglicht. In Flutter wird hierfür `CustomPainter` und `GestureDetector` verwendet, während in React Native `react-native-gesture-handler` zum Einsatz kommt, um Nutzerinteraktionen zu erfassen und die Zeichenlogik umzusetzen.

Flutter verwendet eine spezifische Widget-Hierarchie mit `IndexedStack` und `StatefulWidget`, während React Native auf Komponenten wie `BottomNavigation` und `GestureHandler` setzt. Flutter bietet eine umfassende, integrierte Lösung mit eigenen Widgets und Klassen, während React Native jedoch zusätzliche Bibliotheken wie `React Native Paper` und `react-native-gesture-handler` benötigt. Trotz ähnlicher Ziele und Funktionen zeigen sich in der Umsetzung deutliche Unterschiede in den Ansätzen und Technologien der beiden Frameworks. Beide Anwendungen sind in einem öffentlich zugänglichen Git-Repository einsehbar¹.

¹ siehe <https://github.com/Ceddy610/thesis-end-to-end-test>

4.2 KRITERIENKATALOG

Um die verschiedenen Test-Frameworks bewerten zu können, soll in diesem Abschnitt verschiedene Kriterien für Tests dargestellt, evaluiert und für die Bewertung festgelegt werden.

4.2.1 Kriterien

Bestand dieser Untersuchung ist es, ein geeignetes End-to-End-Test-Framework zu identifizieren, das unter einer konstanten Testinfrastruktur sowohl für in Flutter als auch in React Native geschriebene Anwendungen anwendbar ist. Der Schwerpunkt liegt dabei auf der Fähigkeit des Frameworks, die Funktionalität bei gleichbleibenden Testfällen in beiden Anwendungen sicherzustellen.

Anhand der relevanten Literatur und den spezifischen Anforderungen dieser Untersuchung lassen sich die Evaluierungskriterien weiter spezifizieren. Diese Kriterien dienen als Grundlage für die Bewertung der verschiedenen Test-Frameworks, welche anhand dieser Kriterien systematisch untersucht und bewertet werden.

- **Verifizierung der Funktionalität:**
Wie bereits im Kapitel der relevanten Arbeit erläutert, ist eines der Hauptziele für GUI bzw. End-to-End-Tests die Verifizierung der Funktionalität. Hier liegt der Fokus, bei einem gleichbleibenden Test, die Funktionalität in der Anwendungen zu gewährleisten.
- **Gemeinsame Testfälle:**
Möglichkeit, dieselben Testskripte für beide Plattformen zu verwenden, ohne umfangreiche Anpassungen an diesen vorzunehmen. Dies ist wichtig, um bestimmen zu können, ob eine Migration ohne das Ändern der Tests möglich ist.
- **Cross-Plattform Unterstützung:**
Das Framework sollte verschiedene Browser und Plattformen unterstützen, um sicherzustellen, dass die Tests auf allen Zielumgebungen laufen. Unabhängig davon, welches Framework zur Erstellung der Anwendung verwendet wurde.
- **Unterstützung von UI-Elementen beider Plattformen:**
Das Framework sollte in der Lage sein, UI-Elemente in beiden Plattformen (Flutter und React Native) auf die gleiche Weise zu identifizieren und anzusprechen.
- **Benutzerfreundlichkeit und Dokumentation:**
Gute Dokumentation und Beispiele helfen beim schnellen Einstieg und erleichtern die Implementierung und Anpassung von Tests.
- **Integration mit CI/CD Pipelines:**
Das Framework sollte sich nahtlos in Continuous Integration ([CI](#)) und

Kriterium	Gewichtung (%)
Verifizierung der Funktionalität	25
Gemeinsame Testfälle	25
Unterstützung von UI-Elementen	20
Cross-Plattform Unterstützung	10
Benutzerfreundlichkeit und Dokumentation	10
Integration mit CI/CD Pipelines	10
Gesamtbewertung	100

Tabelle 4.1: Gewichtung der Kriterien

Continuous Deployment ([CD](#)) Pipelines integrieren lassen, um automatische Tests bei jedem Build auszuführen.

4.2.2 *Gewichtung*

Um sicherzustellen, dass das ausgewählte Framework die Anforderungen an die Testinfrastruktur erfüllt und die Funktionalität beider Plattformen zuverlässig überprüft, wurden verschiedene Kriterien identifiziert und gewichtet. Die Gewichtung basiert auf den Präferenzen des Autors im Kontext der Schwerpunkte dieser Untersuchung und wird in der Diskussion nochmals aufgegriffen.

Die zwei wichtigsten Kriterien, die Verifizierung der Funktionalität und die Möglichkeit, gemeinsame Testfälle für beide Plattformen zu verwenden, erhalten jeweils eine Gewichtung von 25%. Diese hohe Gewichtung betont die zentrale Rolle der Sicherstellung der korrekten Funktionalität der Anwendungen sowie der Wiederverwendbarkeit von Testskripten, um den Aufwand für die Testentwicklung zu minimieren. Zudem sind diese Hauptbestandteile der Untersuchungen dieser Arbeit. Ein Aspekt von erhöhter Relevanz ist die Unterstützung von UI-Elementen. Diesem Aspekt wird eine Gewichtung von 20% zugewiesen, da die Unterstützung von UI-Elementen die Wiederverwendbarkeit der Tests erheblich fördert.

Die restlichen Kriterien – Cross-Plattform-Unterstützung, Benutzerfreundlichkeit und Dokumentation sowie Integration mit CI/CD Pipelines – erhalten jeweils eine Gewichtung von 10%. Diese Kriterien sind ebenfalls entscheidend für die Testinfrastruktur, jedoch von etwas geringerer Priorität im Vergleich zu den Hauptkriterien und nicht Hauptbestandteil dieser Arbeit. Durch diese gewichtete Bewertungsmatrix kann mit den Ergebnissen dieser Arbeit eine fundierte Entscheidung darüber getroffen werden, welches Test-Framework die Anforderungen am besten erfüllt (siehe [Tabelle 4.1](#)). Jedoch basiert die Gewichtung dabei auf den Präferenzen des Autors im Kontext der Schwerpunkte dieser Untersuchung. Dies wird nochmals in der Diskussion aufgegriffen.

4.3 TESTSzenARIEN

Um die Anwendung adäquat evaluieren zu können, ist es von Vorteil, spezifische Anwendungsfälle (Use Cases) der Anwendung zu formulieren. Wie in der Studie von [MB23] beschrieben, können diese Anwendungsfälle genutzt werden, um daraus sinnvolle und für die Anwendung relevante Testszenarien abzuleiten.

4.3.1 *Use Cases*

Im Verlauf dieses Kapitels wurde die Funktionsweise der exemplarischen Anwendung detailliert erläutert. Diese Anwendung umfasst sowohl einen To-do-Screen zur Verwaltung von Aufgaben als auch einen Zeichen-Screen, der zum Zeichnen oder Schreiben genutzt werden kann. Daraus ergeben sich folgende Use Cases:

1. **Bereits vorhandene Aufgabe abhaken.** Als Nutzer möchte ich bereits erstellte Aufgaben abzuhaken, damit diese als erledigt markiert sind.
2. **Bereits vorhandene Aufgabe löschen.** Als Nutzer möchte ich bereits erstellte Aufgaben löschen, damit diese nicht mehr angezeigt werden.
3. **Neue Aufgabe erstellen.** Als Nutzer möchte ich neue Aufgaben anlegen können, damit diese in der Übersicht der Aufgaben auftaucht.
4. **Neue Zeichnung anfertigen.** Als Nutzer möchte ich auf dem Zeichen-Screen neue Zeichnungen anlegen, damit ich meine kreativen Ideen und Notizen festhalten und visualisieren kann.

4.3.2 *Use Cases zu Testszenarien*

Um Testszenarien abbilden zu können, wird die Methode der oben genannten Arbeit verwendet. Diese Methode sieht als ersten Schritt die Identifizierung des Basis-Flows vor, auch bekannt als Happy Path, welcher den gewünschten Ausgang des Use Cases darstellt. Dabei wird angemerkt, dass in diesem Schritt auch die Aufteilung in verschiedene separate Aktionen erfolgt, die den Basis-Flow vollständig machen. Darüber hinaus wird die Identifizierung alternativer Flows erwähnt, die in dem Rahmen dieser Arbeit jedoch nicht weiter betrachtet wird.

Sobald der Basis-Flow der User Story identifiziert und in einzelne Schritte aufgeteilt wurde, wird der gewünschte Ausgang, den die Nutzerinnen und Nutzer der Anwendung erreichen können, festgelegt. Auch hier wird sich ausschließlich auf den Happy Path fokussiert.

Im letzten Schritt werden die Testszenarien anhand der Aktionen der Nutzenden, des Nutzerinputs, hier in diesem Falle Nutzereingaben, wie Texte oder Daten, und des erwarteten Ausgangs definiert. Um den gesamten Prozess exemplarisch zu veranschaulichen, werden im Anschluss für die definierten Use Cases spezifische Testszenarien entwickelt.

Das beschriebene Vorgehen wird nun angewendet, um die bereits definierten Use Cases in Testszenarien zu überführen.

1. **Bereits vorhandene Aufgabe abhaken.** Um eine bereits vorhandene Aufgabe abzuhaken, muss in der Aufgabenübersicht die jeweilige Aufgabe mit einem angeklickt werden. Daraufhin sollte die Anwendung diese Aufgabe als erledigt markieren.
 - a) Aktionen: Aufgabe anklicken.
 - b) Input: Kein Input.
 - c) Erwarteter Ausgang: Aufgabe wird als erledigt markiert.
2. **Bereits vorhandene Aufgabe löschen.** Um eine bereits vorhandene Aufgabe zu löschen, muss in der Aufgabenübersicht die jeweilige Aufgabe nach links gewischt werden. Daraufhin sollte die Anwendung diese Aufgabe löschen und nicht mehr anzeigen.
 - a) Aktionen: Aufgabe nach links wischen.
 - b) Input: Kein Input.
 - c) Erwarteter Ausgang: Aufgabe wird nicht mehr angezeigt.
3. **Neue Aufgabe erstellen.** Um eine neue Aufgabe zu erstellen, muss in der Aufgabenübersicht auf der Hinzufügen-Knopf betätigt werden. Daraufhin sollte sich ein Eingabefenster öffnen, welcher den Namen und eine optionale Beschreibung der Aufgabe durch den Nutzer erwartet. Anschließend, nach dem Klick auf den Speichern-Knopf, sollte die Anwendung das Eingabefenster schließen und die neu erstellte Aufgabe in der Übersicht anzeigen.
 - a) Aktionen: Hinzufügen Button klicken, Aufgabe erstellen.
 - b) Input: name: Task 1, description: Description 1
 - c) Erwarteter Ausgang: Die neu erstellte Aufgabe wird in Übersicht angezeigt.
4. **Neue Zeichnung anfertigen.** Um eine neue Zeichnung oder Notiz anzufertigen, muss der Nutzer den Zeichen-Screen mittels der Navigation öffnen und auf diesem freihändig eine Zeichnung oder Notiz anlegen. Daraufhin sollte die Anwendung diese Zeichnung oder Notiz genauso so repräsentieren, wie der Nutzer diese angelegt hat.
 - a) Aktionen: Zeichen-Screen öffnen, Zeichnung anlegen.
 - b) Input: Kein Input.
 - c) Erwarteter Ausgang: Zeichnung wird genauso angezeigt, wie die Nutzereingabe.

Diese definierten Testszenarien werden zur Übersicht nochmals in der [Tabelle 4.2](#) festgehalten.

Use Case	Aktionen	Inputs	Ausgang
Aufgabe erledigen	Aufgabe anklicken	-	Aufgabe wird in der Übersicht als erledigt markiert
Neue Aufgabe wird erstellt	Hinzufügen Button klicken, Aufgabe erstellen	name: Some-Todo description: Some-Description	Neue Aufgabe wird in der Übersicht angezeigt.
Aufgabe löschen	Aufgabe nach links wischen	-	Aufgabe wird nicht mehr in der Übersicht angezeigt
Neue Zeichnung anlegen	Zeichen-Screen öffnen, Zeichnung anlegen	-	Zeichnung wird genauso angezeigt, wie die Nutzereingabe

Tabelle 4.2: Use Case zu Testszenario

IMPLEMENTIERUNG

In diesem Kapitel werden die Testfälle unter Verwendung der Test-Frameworks Maestro und Appium entwickelt und auf den entsprechenden Anwendungen ausgeführt. Zunächst werden die notwendigen Rahmenbedingungen für die Schaffung einer reproduzierbaren Testumgebung detailliert beschrieben. Darauf aufbauend wird für jedes der genannten Test-Frameworks umfassend auf die Schritte der Installation, Konfiguration und Implementierung der End-to-End-Tests eingegangen. Schließlich wird die Durchführung dieser Tests in der definierten Testumgebung erläutert. Die in diesem Kapitel implementierten Tests sind in einem öffentlichen Git-Repository einsehbar¹.

5.1 RAHMENBEDINGUNGEN

Um eine reproduzierbare Testumgebung zu schaffen, wurde ein Android-Emulator des Geräts Google Pixel 3a mit Android-Version 15 verwendet. Ein solcher Emulator ermöglicht die lokale Ausführung auf einem Entwicklungsrechner und wird typischerweise über die integrierte Entwicklungsumgebung (IDE) Android Studio eingerichtet. Android Studio initialisiert beim ersten Öffnen alle erforderlichen Software Development Kits (SDKs) für die Android-Entwicklung und ermöglicht die Nutzung des Emulators.

In dieser Untersuchung wurde der Android-Emulator als Testumgebung gewählt, da Maestro keine Webanwendungen im Browser auf anderen Geräten als mobilen Endgeräten unterstützt [Mobe] und für die Nutzung von iOS-Emulatoren ein Mac erforderlich ist.

Nachdem Android Studio eingerichtet ist, kann der Emulator über die Navigationsleiste unter Tools → Device Manager konfiguriert werden. Durch die Option Create Device öffnet sich ein Dialogfenster, das es erlaubt, spezifische Einstellungen für den Emulator festzulegen. Dazu gehören die Auswahl des zu emulierenden Geräts ([Abbildung a.2a](#)), die Android-Version ([Abbildung a.2b](#)) sowie zusätzliche Konfigurationen wie Auflösung und Hardware-Optionen ([Abbildung a.2c](#)). Nach Abschluss der Konfiguration wird der Emulator durch Klicken auf Finish erstellt.

In der Liste der Android Virtual Devices (AVD) wird der neu erstellte Emulator angezeigt und kann durch Klicken auf das Play-Symbol gestartet werden ([Abbildung a.2d](#)) [[Anda](#)].

¹ siehe <https://github.com/Ceddy610/thesis-end-to-end-test>

5.2 MAESTRO

5.2.1 Installation

Um Maestro zu installieren ist im Terminal/Konsole ein curl Aufruf mit folgender Eingabe nötig:

```
curl -Ls "https://get.maestro.mobile.dev" | bash
```

Ist dieser Befehl ausgeführt wird eine ausführbare Datei in das .bin des Home-Verzeichnisses hinterlegt. Danach kann bereits die installierte Anwendung auf Emulator gestartet und Maestro erkennt automatisch den jeweiligen Android Emulator [Mobb].

5.2.2 Konfiguration

Für die Nutzung von Maestro muss die entsprechende Anwendung auf dem Emulator installiert sein. Maestro ermöglicht das Schreiben von Tests in so genannten Flows, die im Dateiformat .yaml verfasst werden. Um die korrekte Anwendung zu identifizieren, benötigt Maestro die App-ID, die während des Buildprozesses der jeweiligen Cross-Platform-Frameworks festgelegt wird.

Dies wird wie folgt dargestellt:

```
appId: your.app.id
---
- launchApp
```

Die Variable `your.app.id` repräsentiert in diesem Kontext die App-ID der zu testenden Anwendung. Zur Veranschaulichung dieses Konzepts innerhalb dieser Arbeit betrachten wir das Beispiel der in dieser Arbeit erstellten Flutter-Anwendung mit der App-ID `com.example.todoappflutter`. In der entsprechenden .yaml-Datei müssen folgende Anweisungen angegeben werden:

```
appId: com.example.todoappflutter
---
- launchApp
```

Der folgende Befehl wird verwendet, um die Verbindung des Maestro-Drivers mit dem Emulator herzustellen und die Anwendung auf dem Gerät zu starten.

```
maestro test example.yaml
```

5.2.3 Implementierung der End-to-End-Tests

Nachdem nun erläutert wurde, wie Maestro eingerichtet wird, wird im Folgenden die Erstellung eines Tests in diesem Framework betrachtet. Auf das

im vorherigen Kapitel definierte erste Testszenario wird zurückgegriffen. Zur Erinnerung: Die Anwendung wird zu Beginn mit vordefinierten Testdaten geladen und entsprechend gerendert. Der Flow, basierend auf diesem ersten Testszenario, gestaltet sich wie folgt:

```
appId: com.example.todoappflutter
---
- launchApp
- tapOn:
  label: "Check first todo"
  text: "Some-todo-1.*"
- assertVisible:
  label: "Is first todo checked?"
  text: "Some-todo-1.*"
  checked: true
```

Der Flow wird Schritt für Schritt von oben nach unten durchgegangen. Zunächst wird die Anwendung mittels der App-ID definiert. Diese wird mit dem ersten Befehl gestartet und wartet, bis die App initialisiert ist. Anschließend klickt der Driver, mittels der `tapOn` Anweisung, auf die erste Aufgabe mit dem Namen *Some-todo-1*, um diese als erledigt zu markieren. Danach überprüft der Driver, ob die Aufgabe tatsächlich als erledigt markiert wurde.

Es wird nun betrachtet, wie die übrigen Testszenarien in Maestro implementiert werden können.

1. **Aufgabe löschen.** Wir benötigen hier das nach links wischen innerhalb der Anwendung. Maestro bietet hier den Befehl `swipe` der mit Start- und Endkoordinaten arbeitet. Woher wissen wir jedoch, auf welcher Höhe das sich zu löschen Element befindet? Maestro kann mittels `Selectors` ermitteln, auf welcher Höhe sich das Element befindet. Dies könnte folgendermaßen dargestellt werden:

```
appId: com.example.todoappflutter
---
- launchApp
- swipe:
  from:
    text: "Some-todo-1.*"
  direction: LEFT
```

Dieser Befehl mit den gegebenen Instruktionen wischt das Element von der Mitte des Elements nach links. Danach sollte das Element nicht mehr sichtbar sein. Dazu bietet Maestro ebenfalls einen passenden Befehl `assertNotVisible`. Fügt man nun alles zusammen, schaut der Flow folgendermaßen aus:

```
appId: com.example.todoappflutter
---
- launchApp
- swipe:
  from:
    text: "Some-todo-1.*"
  direction: LEFT
- assertNotVisible:
  text: "Some-todo-1.*"
```

2. **Neue Aufgabe erstellen.** In diesem Falle muss der Maestro-Driver den Button zum Erstellen von einer neuen Aufgabe finden und anklicken. Wird dem Befehl tapOn einen Text übergeben, wie in diesem Fall für den Text des Buttons, sucht der Driver genau nach diesem Element und klickt darauf. Daraufhin sollte sich ein Eingabefenster öffnen, das die Eingabe von Aufgabennamen und Beschreibung erfordert. In der exemplarischen Anwendung sind dies die Felder *Task* und *Description*. Um Nutzereingaben zu simulieren, bietet sich der Befehl inputText an, der nach dem Klick auf die Eingabefelder die entsprechenden Daten eingibt. Nach dem Klick auf den Add Knopf sollte die neu erstellte Aufgabe angezeigt werden. Dieser Ablauf ist in Maestro wie folgt gestaltet:

```
appId: com.example.todoappflutter
---
- tapOn: "Add a task"
- tapOn: "Task"
- inputText: "Task 1"
- tapOn: "Description"
- inputText: "Description 1"
- tapOn: "Add"
- assertVisible: "Task 1.*"
```

Das letzte Testszenario, also das **Anlegen einer neuen Zeichnung**, gestaltet sich hingegen etwas schwieriger. Hier benötigen wir einen Visual Regression Test, der mit der aktuellen Version von Maestro in der Standardkonfiguration noch nicht möglich ist [Gita]. An diesem Punkt musste eine alternative Lösung gefunden werden. Maestro bietet die Möglichkeit, einen Screenshot des aktuellen Zustands der Anwendung mittels des Befehls takeScreenshot zu erstellen und außerdem eine externe JavaScript-Datei mit dem Befehl runScript auszuführen. Diese Funktionen können wir in diesem Abschnitt der Arbeit nutzen.

Wir können in der JavaScript-Datei http Anfragen senden an einen Server. Dies können wir nutzen, um die Funktionalität zu erweitern. Eine solche JavaScript-Datei für Maestro könnte in diesem Kontext wie folgt aussehen:

```

const response = http.post('http://127.0.0.1:4567', {
  headers: {
    'Content-Type': 'application/json',
  }
});

output.example = json(response.body);

```

In diesem Beispiel wird auf einem lokal bereitgestellten Server eine Anfrage gesendet und die zurückgelieferte Antwort auf die Variable `output` geschrieben. In dem dazugehörigen Flow kann Maestro den in der JavaScript-Datei definierten `output` auslesen und für die Ausgabe nutzen. Ein solcher Flow kann wie folgt aussehen:

```

- runScript:
  file: scripts/exec.js
- evalScript: ${console.log(output.example)}
- assertTrue:
  condition: ${output.example === 'some-expectation'}

```

Wie zu sehen, ruft Maestro die JavaScript-Datei `exec.js` auf. Diese schreibt, wie bereits im obigen Beispiel dargelegt, den `output` mit dem `example` mit der Antwort des Servers. Diesen geben wir in dem Schritt `evalScript` nochmals in der Konsole aus und im letzten Schritt verifizieren wir den Wert mit einem erwartetem Wert.

Genau dieser Ansatz wird genutzt, um Maestro um ein Visual Regression Tool zu erweitern. Zu Beginn muss klar definiert werden, welche Anforderungen für den Bau eines solchen Tools erfüllt sein müssen. In diesem Fall muss es möglich sein, zwei Screenshots der Anwendung visuell miteinander zu vergleichen. Daher verwenden wir einen `Node.js`-Server mit `Express.js`, der einen entsprechenden Endpunkt bereitstellt. Wenn dieser Endpunkt auf dem Server aufgerufen wird, werden zwei Screenshots der Anwendung miteinander verglichen: einerseits die Baseline, also das erwartete Ergebnis, und andererseits das tatsächliche Ergebnis. Der Vergleich der einzelnen Screenshots erfolgt mit dem Node-Package `ResembleJS`.

Durch die Recherche verschiedener End-to-End-Test-Frameworks wurde dieses Package identifiziert. Dabei zeigte sich, dass Appium in Verbindung mit WebdriverIO genau dieses Package für visuelle Tests verwendet [[Webf](#)].

`ResembleJS` lädt dabei diese zwei Bilder, in diesem Fall die Screenshots, und konvertiert diese in ein Format, das für die Analyse geeignet ist, typischerweise in Form von Pixelarrays. Diese Arrays enthalten Informationen über die Farbe und Transparenz jedes Pixels im Bild. Anschließend werden die Pixel der beiden Bilder verglichen, indem sie die Unterschiede in den RGB-Werten und der Alpha-Transparenz berechnet. Dabei können Toleranzwerte einstellen, um zu bestimmen, wie empfindlich die Unterschiede betrachtet werden sollen. Höhere Toleranzwerte bedeuten, dass nur signifikante Unterschiede gemeldet werden, während niedrigere Werte auch geringfügige Un-

terschiede erkennen lassen. Abschließend aus dieser Analyse ein Differenz-Bild erstellt werden, dass die Unterschiede der beiden Bilder markiert, wie auch eine prozentuale Differenz der Pixel zurückgeben werden (siehe Abbildung 5.1) [Gitd; Rsm].

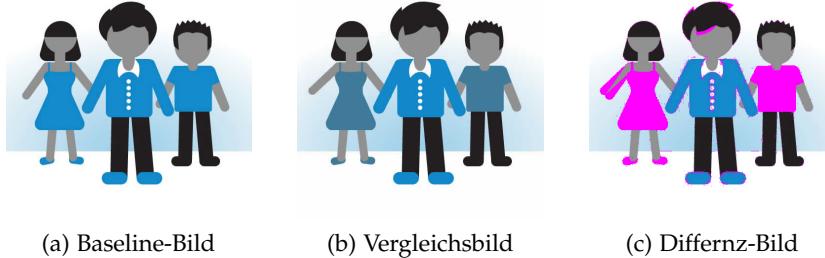


Abbildung 5.1: Bildervergleich mittels ResembleJS [Rsm]

Wird dieses Package nun in einem Server implementiert, könnte dies wie folgt aussehen:

```
app.post("/", (req, res) => {
  const pictureBaseline = req.body.baseline;
  const pictureCurrent = req.body.current;

  resemble
    .default(`../${pictureBaseline}`)
    .compareTo(`../${pictureCurrent}`)
    .onComplete((data) => {
      res.status(200).send({
        diff: data.misMatchPercentage,
      });
    });
});
```

Hier werden dem Endpunkt im Anfrage-Body zwei Dateinamen übergeben. Diese beiden Bilder werden von ResembleJS geladen und Pixelweise verglichen. Der Endpunkt gibt die ermittelte prozentuale Differenz der beiden Bilder zurück. Integriert man diesen Prozess in einen Flow, könnte dieser wie folgt aussehen:

```
- takeScreenshot: actual
- runScript:
  file: scripts/image-compare.js
  env:
    baseline: 'expected.png'
    current: 'actual.png'
- evalScript: ${console.log(output.diff * 100)}
- assertTrue:
  condition: ${output.diff * 100 <= 30}
```

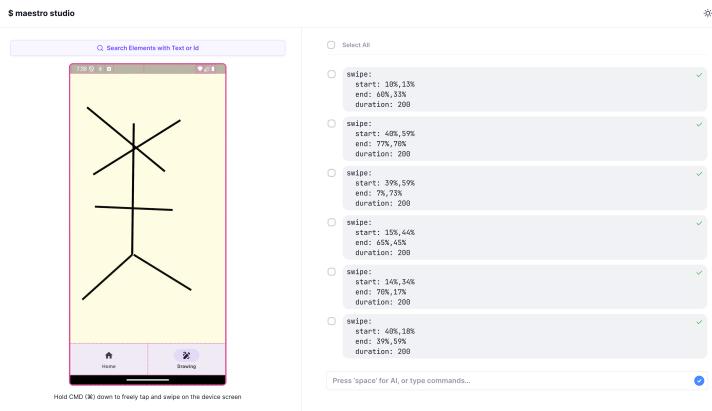


Abbildung 5.2: Maestro Studio – Zeichnen einer Figur

In diesem Prozess erstellt der Maestro-Driver einen Screenshot des aktuellen Zustands der Anwendung und speichert diesen als Datei. Anschließend wird eine JavaScript-Datei ausgeführt, die sowohl diesen Screenshot als auch den Baseline-Screenshot, der das erwartete Ergebnis repräsentiert, per HTTP-Anfrage an den Server sendet. Der Server berechnet die prozentuale Abweichung zwischen den beiden Screenshots und übermittelt diese und die JavaScript-Datei schreibt diese in den output, welchen wir in diesem Prozess verifizieren können.

Selbstverständlich muss für diesen Prozess auch eine Baseline erstellt werden, wofür das Maestro Studio Unterstützung bietet. Mithilfe des Maestro Studios kann mit der Anwendung auf dem Emulator interagiert werden. Die Interaktionen werden vom Maestro Studio aufgezeichnet und als Flow gespeichert [Mobic].

Wie in Abbildung 5.2 zu sehen, wurde mithilfe des Maestro Studios in dem Zeichen-Screen eine Zeichnung angefertigt und als Screenshot gespeichert als Baseline für den visuellen Vergleich des Testes. Während der manuellen Eingabe zeichnete Maestro Studio die jeweiligen Schritte auf, die wiederum für diesen Test verwendet werden können.

Werden diese Schritte in den Flow integriert, ergibt sich folgende Datei:

```
- tapOn:
  label: "Go to drawing page"
  text: "Drawing.*"
- swipe:
  start: "10%,13%"
  end: "60%,33%"
  duration: 200
- swipe:
  start: "40%,59%"
  end: "77%,70%"
  duration: 200
- swipe:
```

```

    start: "39%, 59%"
    end: "7%, 73%"
    duration: 200
- swipe:
    start: "15%, 44%"
    end: "65%, 45%"
    duration: 200
- swipe:
    start: "14%, 34%"
    end: "70%, 17%"
    duration: 200
- swipe:
    start: "40%, 18%"
    end: "39%, 59%"
    duration: 200
- takeScreenshot: DrawingPageActual
- runScript:
    file: scripts/image-compare.js
    env:
        baseline: "DrawingPageExpected.png"
        current: "DrawingPageActual.png"
- evalScript: ${console.log(output.diff * 100)}
- assertTrue:
    label: "Images are similiar"
    condition: ${output.diff * 100 <= 30}

```

Um beide Anwendungen, sowohl die mit Flutter und React Native kompilierte Anwendung, mit diesen Tests zu prüfen, muss berücksichtigt werden, wie die unterschiedlichen App-IDs gehandhabt werden sollen. Diese unterscheiden sich in beiden Anwendungen und können daher nicht in den Testdateien festgeschrieben werden. Auch hier bietet Maestro eine Lösung. Es ist möglich, dem Maestro CLI Umgebungsvariablen zu übergeben, die anschließend in den Testdateien verwendet werden können.

```
maestro test -e APP_ID=com.example.todoappflutter example.yaml
```

Somit können wir in den Dateien folgende Änderungen vornehmen:

```

appId: ${APP_ID}
---
- launchApp

```

Wenn die Maestro CLI mit der App-ID der Anwendung als Umgebungsvariable aufgerufen wird, öffnet der Maestro-Driver die entsprechende Anwendung, sofern sie auf dem Emulator installiert ist.

5.2.4 Durchführung

Um die Tests für beide Anwendungen durchzuführen, muss der Emulator gestartet und beide Anwendungen installiert sein. Sind beide Anwendungen



Abbildung 5.3: Ausführung der Maestro Tests

installiert, werden die Tests mithilfe eines bash Script gestartet. Dabei wird sowohl der Node.js-Server gestartet als auch die Testdateien ausgeführt. Diese sieht folgendermaßen aus:

```
cd server;
npm run start &
cd ..;
maestro test -e APP_ID=com.example.todoappflutter android-flow.yaml;
maestro test -e APP_ID=com.example.todoappflutter android-flow-drawing.yaml;
maestro test -e APP_ID=com.example.ToDoApp android-flow.yaml;
maestro test -e APP_ID=com.example.ToDoApp android-flow-drawing.yaml;
pkill node;
```

Hier zu sehen, startet das Skript den Node.js-Server und öffnet im nächsten Schritt mithilfe des Befehls `maestro test` und der Variable `APP_ID` die jeweiligen Anwendungen. Diese werden daraufhin in Echtzeit auf dem Endgerät ausgeführt und dabei in der Konsole geloggt (siehe Abbildung 5.3). Zusätzlich wurden Tests durchgeführt, um das Verhalten bei Fehlerfällen zu untersuchen. Hierfür wurde eine Koordinate im Testszenario für den Zeichen-Screen absichtlich geändert, um einen Fehlerfall zu simulieren. Die Ergebnisse dieser Tests sowie weitere relevante Ausgaben der Testfälle sind im Anhang dokumentiert.

5.3 APPiUM

5.3.1 Installation

Um Appium für das Projekt zu installieren, benötigt es mehrere Schritte. Zum einen müssen auf lokalen System bereits das Java Development Kit (JDK), als auch die Android SDK installiert sein. Zudem benötigt es Node.js auf dem System, um npm Packages global zu installieren [AppE; AppH]. Sind diese Voraussetzungen erfüllt, kann mittels dieses Konsolenbefehls Appium global installiert werden:

```
npm install -g appium
```

Rufen wir `appium` in der Konsole auf, wird folgende Meldung in der Konsole zurückgegeben:

```
[Appium] Welcome to Appium v2.0.0
```

Dies startet den Appium Server. Damit Appium überhaupt mit einem Endgerät kommunizieren kann, muss ein Driver installiert. Ein Driver instrumentiert die Interaktionen mit dem jeweiligen Endgerät [AppG]. Da in dieser Arbeit ausschließlich einen Android-Emulator verwendet wird, nutzen wir als Driver den `UiAutomator2`. Dieser kann mit einer installierten Anwendung auf einem Android-Gerät, sei es ein Emulator oder angeschlossenem Gerät, interagieren [AndB; AppF]. Dieser wird folgendermaßen installiert:

```
appium driver install uiautomator2
```

Zusätzlich wird für Appium noch einen Client benötigt, der die Instruktionen an diesen sendet. Zum Verständnis sieht im Groben der Kommunikationsfluss in Appium folgendermaßen aus [AppD]:

- Client sendet eine Anfrage an den Appium Server.
- Appium Server verarbeitet die Anfrage und leitet sie an den Driver weiter.
- Driver führt die entsprechende Aktion auf dem Endgerät aus und sendet das Ergebnis zurück an den Appium Server.
- Appium Server sendet das Ergebnis an den Client zurück.

Für den Client nutzen wir hier in dieser Arbeit `WebdriverIO` [WebD], dies ist ein JavaScript-Client für Appium, der Instruktionen abstrahiert und diese an Appium zur Ausführung von Instruktionen weitergibt. Um diesen Client in dieser Arbeit zu verwenden, sollte zuallererst ein neues Node.js-Projekt angelegt werden. In diesem Projekt wird nun der `WebdriverIO` Client mittels npm installiert.

```
npm install @wdio/cli
```

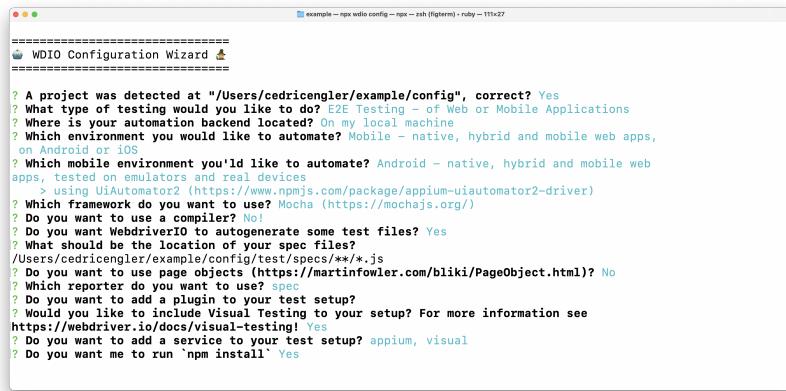


Abbildung 5.4: WebdriverIO – Konfiguration

5.3.2 Konfiguration

Sind die oben genannten Schritte getan, so hilft WebdriverIO mit der Konfiguration des Frameworks. Dazu wird folgender Befehl innerhalb des Projektes ausgeführt:

```
npx wdio config
```

Dieser Befehl geht die Konfiguration des Projektes Schritt für Schritt durch mithilfe von Auswahlmöglichkeiten, welche der Nutzer treffen kann (siehe Abbildung 5.4). Zu Beginn wird abgefragt, für welche Art von Test WebdriverIO verwendet werden soll. Hier stehen verschiedene Auswahlmöglichkeiten zur Wahl, wie beispielsweise Unit-Tests oder End-to-End-Tests. Hier wird End-to-End-Test gewählt.

Zudem wird eine Auswahl getroffen, auf welchen Geräten die Tests ausgeführt werden sollen und welches Betriebssystem diese verwenden. Es besteht die Möglichkeit, zwischen Emulatoren und physischen Geräten zu wählen und spezifische Android- oder iOS-Versionen festzulegen. Anschließend wird nach dem Framework ausgewählt, in dem die Tests geschrieben werden sollen. In dieser Untersuchung wird Mocha verwendet. Mocha ist ein JavaScript-Test-Framework, in welchen die Tests folgendermaßen geschrieben werden [Moc]:

```
describe('Array', function () {
  describe('#indexOf()', function () {
    it('should return -1 when the value is not present', function () {
      assert.equal([1, 2, 3].indexOf(4), -1);
    });
  });
});
```

Die Funktion `describe` definiert den Kontext des zu testenden Elements, während `it` die erwartete Verhaltensweise spezifiziert.

Diese Struktur von Testszenarien ist charakteristisch für das Behavior-Driven

Development (BDD) [Not]. Anschließend wird mit Hilfe von assert überprüft, ob das tatsächliche Ergebnis mit dem erwarteten übereinstimmt. Darüber hinaus werden weitere Konfigurationseinstellungen abgefragt, beispielsweise die Verwendung von TypeScript oder die automatische Generierung von Testdateien. Eine weitere optionale Komponente ist der Einsatz von Visual Regression Tests mittels WebdriverIO. Diese Option wird gezielt ausgewählt, da für einen der definierten Tests ein visueller Vergleich erforderlich ist.

Nach Abschluss der Konfiguration über die CLI wird eine Konfigurationsdatei generiert, die sämtliche Entscheidungen dieses Konfigurationsprozesses dokumentiert. Diese Datei hat folgendes Format:

```
exports.config = {
  runner: 'local',
  port: 4723,
  specs: [
    './test/specs/**/*.js'
  ],
  exclude: [],
  capabilities: [
    {
      platformName: 'Android',
      'appium:deviceName': 'emulator-5554',
      'appium:platformVersion': '12.0',
      'appium:automationName': 'UiAutomator2'
    },
    {
      services: ['appium', 'visual'],
      framework: 'mocha',
      reporters: ['spec'],
      mochaOpts: {
        ui: 'bdd',
        timeout: 60000
      }
    }
};
```

Diese definiert `capabilities`, hier die Konfiguration, auf welchem Gerät die Tests ausgeführt werden sollen. Außerdem in welcher Ordnerstruktur die Tests geschrieben werden, mit dem `specs` Abschnitt [Webb].

5.3.3 Implementierung der End-to-End-Tests

Nach der ausführlichen Erläuterung zur Installation und Konfiguration von Appium wird nun die Erstellung eines Tests innerhalb dieses Frameworks betrachtet. Dabei wird das im vorherigen Kapitel definierte erste Testszenario herangezogen. Der Ablauf dieses ersten Testszenarios gestaltet sich wie folgt:

```

describe('Todo screen', () => {
  it('checks a todo as done', async () => {
    let firstTodo = await driver.$(`~Some-todo-1`);
    await firstTodo.click();

    firstTodo = await driver.$(`~Some-todo-1`);

    const isChecked = await firstTodo.getAttribute('checked');

    expect(isChecked).toBe('true');
  });
});

```

Nun wird der Test schrittweise durchgegangen. Zunächst wird die Aktivität initiiert, wobei in diesem Kontext die zu testende Anwendung gemeint ist. Die Anwendung wird durch die App-ID und die entsprechende Aktivität spezifiziert, welche den Startprozess der Android-Anwendung definiert. Eine Android-Anwendung besteht aus mehreren Activity-Klassen, wobei jede Aktivität eine einzelne, fokussierte Aufgabe repräsentiert, die der Benutzer ausführen kann [Andc].

Die Anwendung, die mit Flutter oder React Native für Android entwickelt wird, enthält die MainActivity-Klasse, die typischerweise die Hauptaktivität darstellt, welche beim Starten der App als erstes angezeigt wird. Aus diesem Grund initiieren wir in diesem Test die MainActivity, um den Testprozess zu beginnen. Im nächsten Schritt sucht der Driver nach dem ersten Element mit der accessibilityId = Some-todo-1. WebdriverIO bietet verschiedene Selektoren für das Finden von Elementen innerhalb Appium's. In diesem Fall wird der Accessibility-ID-Selektor mithilfe des ~-Symbols verwendet. Dieser findet Elemente anhand ihrer Accessibility-ID. Accessibility-IDs sind spezielle IDs, die für die Barrierefreiheit von Anwendungen verwendet werden und oft als eindeutige Identifikatoren für UI-Elemente dienen.

Neben diesem Accessibility-ID-Selektor bietet WebdriverIO auch CSS-Selektoren, die häufig in Webanwendungen verwendet werden, sowie ID-Selektoren, die direkt auf die ID eines Elements zugreifen. Außerdem ist es möglich mittels XPath Elemente basierend auf ihrer Hierarchie und ihren Attributen im UI-Baum zu lokalisieren [Webe].

Um den Accessibility-ID-Selektor nutzen zu können, ist es erforderlich, die genaue Accessibility-ID des jeweiligen Elements sowohl in der Flutter-Anwendung als auch in der React Native-Anwendung zu kennen. In React Native werden alle Textfelder, die in einem View leben, als accessibilityLabel zusammengeführt, welche wiederum mittels der Accessibility-ID gefunden werden kann [Reaa].

Flutter wiederum verwendet Semantics, um Accessibility-IDs in den Elementen zu spezifizieren. Die meisten der von Flutter definierten Widgets definieren bereits Semantics, wie beispielsweise das ListTile-Widget. Diese beinhaltet zum einen title und einen subtitle. Beide sind Text-Widgets,

und damit auch automatisch Semantics, mit deren Text als Label. Wird dann der Konstruktor des ListTile-Widgets aufgerufen, wird mithilfe der MergeSemantics-Methode, die verschiedenen Semantics-Knoten zusammengeführt. Wird nun dieser Konstruktor mit folgenden Werten aufgerufen [Gitg]:

```
ListTile(
    leading: Icon(Icons.access_alarm),
    title: Text('Alarm'),
    subtitle: Text('Wake up at 7:00 AM'),
    trailing: Icon(Icons.more_vert),
);
```

wird der Semantic-Tree in Flutter zusammengeführt und ergibt den folgenden Semantic-Knoten:

```
SemanticsNode:
Role: ListItem
Label: "Alarm Wake up at 7:00 AM"
```

Um für die Tests eine einheitliche Accessibility-ID zu gewährleisten, müssen sowohl in den Anwendungen von Flutter als auch von React Native, Änderungen vorgenommen werden. Damit in der Flutter Anwendung die `description` der Aufgabe nicht als Teil der Accessibility-ID gewertet wird, müssen wir diese explizit ausschließen.

```
CheckboxListTile(
    title: Text(_todoItems[index].task),
    subtitle: ExcludeSemantics(
        excluding: true,
        child: Text(_todoItems[index].description)
    ),
    value: _todoItems[index].isCompleted,
    onChanged: (bool? value) {
        setState(() {
            _todoItems[index].isCompleted = value!;
        });
    },
));
```

Wie in diesem Beispiel zu erkennen, wird explizit `ExcludeSemantics` für den Parameter `subtitle` aufgerufen, um diesen explizit für den Semantic-Knoten auszuschließen. Daraus ergibt sich für dieses Element eine Accessibility-ID nur aus dem Parameter `title`. In React Native hingegen reicht es aus, das `accessibilityLabel` auf den Namen der Aufgabe zu setzen:

```
<List.Item
    accessibilityLabel={title}
    ...more props
/>
```

Sind nun für beide Anwendungen die gleichen Accessibility-IDs gesetzt, kann der obige Test mittels des Selektors auf dieses Element zugreifen und darauf klicken. Danach wird überprüft, ob das Element angehakt wurde.

Nach der detaillierten Erläuterung des ersten Testszenarios wird nun die Untersuchung der weiteren Testszenarien in Appium fortgesetzt.

- Aufgabe löschen.** In diesem Abschnitt wird die Implementierung einer horizontalen Wischgeste innerhalb der Anwendung analysiert. Appium bietet die Möglichkeit, dem Driver mittels eines Skripts spezifische Gesten zu übermitteln. Hierbei wird die `swipeGesture`-Funktion genutzt. Um die exakte Position des zu löschen Elements zu bestimmen, ist es notwendig, die Höhe des betreffenden Elements zu ermitteln. Dies kann durch die Angabe der `elementId` des betreffenden Elements sowie der Richtung und des Prozentsatzes der durchzuführenden Wischgeste als Parameter erreicht werden. Ein Beispiel für diesen Test sieht wie folgt aus:

```
it('removes todo', async () => {
  let firstTodo = await $(`~Some-todo-1`).elementId;

  await driver.executeScript('mobile: swipeGesture', [
    elementId: firstTodo,
    direction: 'left',
    percent: 0.9,
  ]);

  firstTodo = await `~Some-todo-1`;

  expect(firstTodo).not.toBeDisplayed();
});
```

Zu Beginn wird die `elementId` des betreffenden Elements mithilfe des Accessibility-ID-Selektors ermittelt. Anschließend wird die Wischgeste ausgeführt. Danach erfolgt eine Überprüfung, ob das Element nicht mehr sichtbar ist.

- Neue Aufgabe erstellen.** In diesem Szenario muss der Driver den Button zum Hinzufügen einer neuen Aufgabe finden und anklicken. Der `click`-Befehl wird auf das Element mit der Accessibility-ID `Add a task` angewendet. Anschließend öffnet sich ein Eingabefenster, in dem der Name und die Beschreibung der Aufgabe eingegeben werden müssen. Die Eingabefelder sind durch ihre Ressourcen-IDs `task-name` und `task-description` identifizierbar und werden mittels XPath aus dem UI-Baum gesucht. `/*` gibt dabei jeden Tag für Elemente im UI-Baum an, während `[@resource-id="task-name"]` diese Auswahl nochmals nach Elementen mit Attribut `resourceId` filtert [W3s]. Dafür werden in Flutter die jeweiligen Semantics-Identifier und in React Native die `testIDs` entsprechend den zugehörigen Resource-IDs definiert. Der

setValue-Befehl wird verwendet, um die entsprechenden Texteingaben vorzunehmen. Nachdem die Eingabefelder ausgefüllt sind, wird der Bestätigungsbutton, der mit Add als Accessibility-ID gekennzeichnet ist, angeklickt. Schließlich wird überprüft, ob die neu erstellte Aufgabe, die mit der Accessibility-ID Task 1 bezeichnet ist, angezeigt wird. Der gesamte Ablauf wird in Appium wie folgt durchgeführt:

```
it('creates a new todo', async () => {
    const button = await $(`~Add a task`);
    await button.click();

    const input1 = await $('*[@resource-id="task-name"]');
    await input1.click();
    await input1.setValue("Task 1");

    const input2 = await $('*[@resource-id="task-description"]');
    await input2.click();
    await input2.setValue("Description 1");

    const addButton = await $(`~Add`);
    await addButton.click();

    const newTask = await $('~Task 1');

    expect(newTask).toBeDisplayed();
});
```

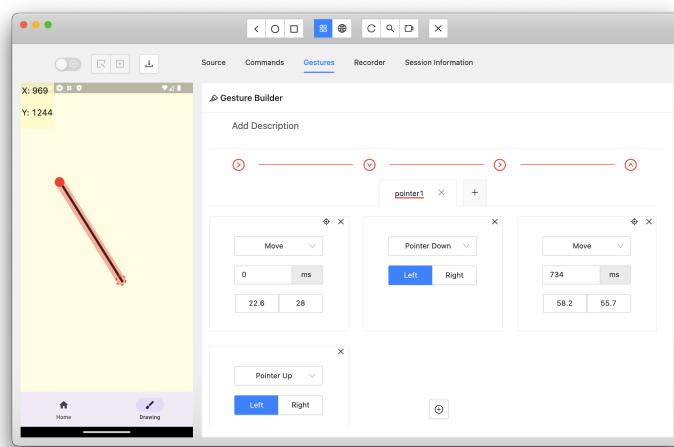


Abbildung 5.5: Appium Inspector

3. **Neue Zeichnung anlegen.** Für das letzte Testszenario wird das Package `visual-testing` von `WebdriverIO` benötigt. Dieses wurde bereits bei

der Konfiguration installiert. Dieses Package liefert eine Erweiterung für `expect` mit. In diesem Fall relevant:

```
await expect(driver).toMatchScreenSnapshot('drawing');
```

Diese Methode vergleicht den aktuellen Stand der UI mit einem bereits gespeicherten Snapshot. Sind keine anderen Konfigurationen vorgenommen worden, so speichert diese Methode bei erstmaligen Aufrufen eine Baseline als Snapshot, welcher dann beim wiederholten Male zum Vergleich des aktuellen Standes der UI genutzt wird. Natürlich muss noch etwas auf den Zeichen-Screen gezeichnet werden. Dazu wurde im ähnlich zu Maestro der Appium Inspector verwendet. Dieser bietet ebenfalls die Möglichkeit, Gesten aufzunehmen und in einen Test zu integrieren (siehe Abbildung 5.5). Integriert man diese Gesten in den Test, so erhält man folgenden Datei:

```
describe('Drawing Screen', () => {
    it('validates the screen', async () => {
        const button = await $(`//*[@resource-id="drawing"]`);
        await button.click();

        await driver.action('pointer', {
            parameters: {
                pointerType: 'touch',
            },
        })
        .move({ origin: 'viewport', x: 210, y: 362 })
        .down()
        .move({ origin: 'viewport', x: 751, y: 908 })
        .up()
        .perform();

        await driver.action('pointer', {
            parameters: {
                pointerType: 'touch',
            },
        })
        .move({ origin: 'viewport', x: 862, y: 423 })
        .down()
        .move({ origin: 'viewport', x: 210, y: 900 })
        .up()
        .perform();
    });
});
```

```

        await driver.action('pointer', {
            parameters: {
                pointerType: 'touch',
            },
        })
            .move({ origin: 'viewport', x: 553, y: 408 })
            .down()
            .move({ origin: 'viewport', x: 481, y: 1442 })
            .up()
            .perform();

        await expect(driver).toMatchScreenSnapshot('drawing', 10, {
            ignoreAntialiasing: true,
            ignoreAlpha: true,
        });
    });
}
)

```

5.3.4 Durchführung

Um die Tests für beide Anwendungen durchzuführen, muss der Emulator gestartet und beide Anwendungen installiert sein. Sobald dieser Schritt abgeschlossen ist, erfolgt die Erstellung von zwei Konfigurationsdateien. Diese beschreiben verschiedene capabilities, also Endgeräte und Anwendungen, auf welche diese Tests laufen sollen. Jene sehen in diesem Fall so aus:

```

capabilities: [
    platformName: 'Android',
    'appium:deviceName': 'Android GoogleAPI Emulator',
    'appium:automationName': 'UiAutomator2',
    'appium:appPackage': 'com.example.todoappflutter',
    'appium:appActivity': '.MainActivity',
]

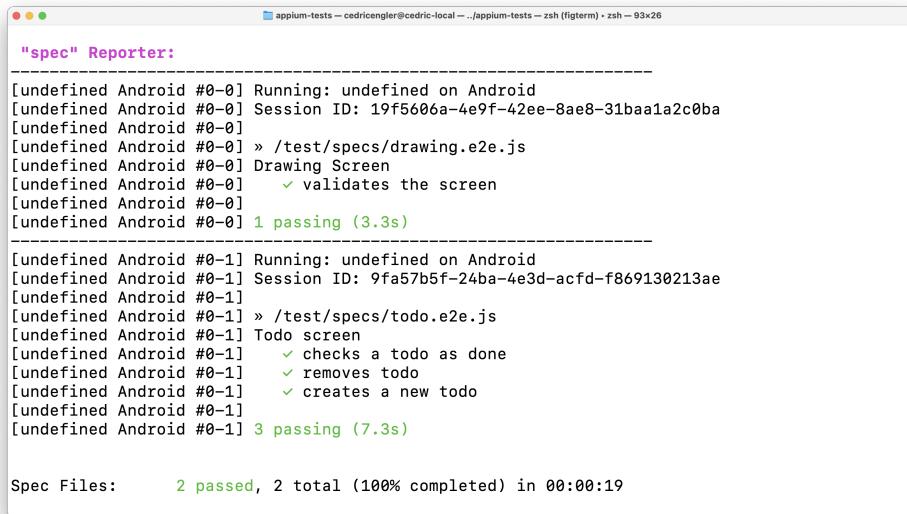
capabilities: [
    platformName: 'Android',
    'appium:deviceName': 'Android GoogleAPI Emulator',
    'appium:automationName': 'UiAutomator2',
    'appium:appPackage': 'com.exmaple.ToDoApp',
    'appium:appActivity': '.MainActivity',
]

```

Der erste Eintrag startet die Tests für die Flutter-Anwendung, während der zweite Eintrag die Tests für die React Native-Anwendung ausführt. Sobald folgender Befehl in der Konsole eingegeben wird, öffnet sich die Anwendung im Emulator und die Tests werden ausgeführt.

```
npx wdio run ./flutter.conf.js && npx wdio run ./reactnative.conf.js &&
```

Die Testdurchführung erfolgt in Echtzeit direkt auf dem Endgerät und wird in der Konsole protokolliert (siehe Abbildung 5.6). Dabei wird detailliert aufgezeigt, welche Tests durchgeführt wurden und ob diese erfolgreich abgeschlossen wurden oder Fehler aufgetreten sind.



```
"spec" Reporter:
-----
[Android #0-0] Running: undefined on Android
[Android #0-0] Session ID: 19f5606a-4e9f-42ee-8ae8-31baa1a2c0ba
[Android #0-0]
[Android #0-0] » /test/specs/drawing.e2e.js
[Android #0-0] Drawing Screen
[Android #0-0]   ✓ validates the screen
[Android #0-0]
[Android #0-0] 1 passing (3.3s)
-----
[Android #0-1] Running: undefined on Android
[Android #0-1] Session ID: 9fa57b5f-24ba-4e3d-acfd-f869130213ae
[Android #0-1]
[Android #0-1] » /test/specs/todo.e2e.js
[Android #0-1] Todo screen
[Android #0-1]   ✓ checks a todo as done
[Android #0-1]   ✓ removes todo
[Android #0-1]   ✓ creates a new todo
[Android #0-1]
[Android #0-1] 3 passing (7.3s)

Spec Files:      2 passed, 2 total (100% completed) in 00:00:19
```

Abbildung 5.6: Ausführung der Appium Tests

6

ERGEBNISSE

6.1 AUSWERTUNG NACH DEN KRITERIEN

Der Hauptbestandteil dieser Arbeit bestand darin, in dem jeweiligen Test-Framework Tests zu schreiben, welche jeweils für die Flutter und React Native geschriebene Anwendungen ausgeführt werden. Dabei lag der Fokus darauf, dass die beschriebenen Testszenarien einmal in einem Test abgebildet werden und in beiden Anwendungen ausgeführt werden.

6.1.1 *Verifizierung der Funktionalität*

Unter der Annahme, dass beide Anwendung die gleiche Funktionalität bieten, konnten beide Test-Frameworks diese verifizieren, ohne große Anpassungen an den eigentlichen Tests vorzunehmen. Dabei wurden auch Fehler in der Implementierung der jeweiligen Anwendung aufgedeckt, wie bspw. das Zeichnen auf dem Zeichen-Screen.

6.1.2 *Gemeinsame Testfälle*

Während in Maestro die Testfälle für beide Anwendungen gleich waren und diese auch zuverlässig funktioniert haben, sind bei Appium einige Schwierigkeiten aufgetreten. Es mussten Anpassungen in den Anwendungen, als auch in den Testdateien gemacht werden. Sei es das Manipulieren der Accessibility-IDs, die den Prozess zwar einfacher gestaltet hat, aber dennoch wieder auf Implementierung der Anwendung hinzukommt, dass es zu beachten gilt. In Maestro hingegen wurde viel mit dem Attribut `text` gearbeitet. Dies hatte den Vorteil, dass die Elemente nur mit dem ausgewiesenen Text gefunden werden können, der in beiden exemplarischen Anwendungen gleich ist. Auch hat dies den Vorteil, dass die innere Implementierung, also die Vergabe von jeglichen IDs, nicht mit der oberflächlichen Anwendungsschicht zusammenhängen.

6.1.3 *Cross-Plattform Unterstützung*

Appium als auch Maestro bieten verschiedene Schnittstellen an, verschiedene Geräte zu testen. Während Maestro vor allem Mobile-First ausgerichtet ist, lässt sich mit der Testinfrastruktur, die mit Appium einhergeht, auch ein Webinterface testen. Zudem ließen sich beide kompilierte Anwendungen sowohl auf Appium als auch in Maestro über den Android Emulator.

6.1.4 Unterstützung von UI-Elementen

Das Suchen von UI-Elementen gestaltet sich in Maestro wesentlich einfacher, als in Appium. In Maestro kann ein Button nach dem Text oder ID gesucht, während hingegen in Appium klar sein muss, auf welchen Gerät getestet wird. Denn die Suche nach Elementen unterscheidet sich je nach Plattform in Appium. In manchen Tests wurde die `resourceId` verwendet, um ein Element zu finden. Diese wird jedoch nur in einer Android-Anwendung vergeben. In einer iOS-Anwendung werden diese als `accessibilityIdentifier` gespeichert. Dies kann in Appium zu Inkonsistenzen oder mehr Aufwand im Schreiben der Testfälle führen.

6.1.5 Benutzerfreundlichkeit und Dokumentation

Da Maestro noch recht neu ist, ist die Dokumentation recht überschaulich, dennoch recht einfach zu verstehen. Zudem ist die Installation und Konfiguration schnell abgeschlossen. Hingegen Appium bietet verschiedene Möglichkeiten an, dieses zu implementieren. Sei es verschiedene Clients, die in verschiedenen Programmiersprachen geschrieben sind oder verschiedene Driver für die jeweiligen Endgeräte. Doch hier beginnt es unübersichtlich zu werden. Appium verweist auf die verschiedenen Clients und deren Dokumentationen, die wiederum auf Appium verweisen [[Weba](#); [Appi](#)].

Das Schreiben der Tests in Maestro erfolgt in einer `yml`-Datei und ist in unterschiedliche Schritte aufgeteilt und übersichtlich. Appium wurde in dieser Untersuchung in Kombination mit WebdriverIO und Mocha verwendet, was in diesem Falle mit JavaScript arbeitet. Hier findet das Behavior-Driven-Testing Einzug mithilfe von `describe` und `it`, die jeweils das Szenario und den erwartenden Ausgang beschreiben. Wie bereits erläutert, hängt die Implementierung der Tests jedoch davon ab, welcher Client und die damit verbundene Programmiersprache letztendlich gewählt wird.

6.1.6 Integration mit CI/CD Pipelines

Appium bietet eine bereits vorgefertigte Datei für einen CI Workflow in Github an [[Gitf](#)]. Diese muss noch für den spezifischen Fall angepasst werden, auch in Hinsicht, ob Appium in Github oder eine andere git-Plattform genutzt wird. Hingegen bietet Maestro einen Cloud-Service an, der bereits alles zur Verfügung stellt [[Mobe](#)].

Dabei kann diese jedoch nur die eingebauten Methoden zum Testen der Anwendungen verwenden. In unserem Beispiel haben wir eine Erweiterung zum visuellen Vergleich der UI hinzugefügt. Da diese Erweiterung einen separaten Server benötigt, kann dieser nicht in der Cloud ausgeführt werden (außer dieser Server ist öffentlich zugänglich). Zudem ist die Cloud mit Kosten verbunden.

Kriterium	Gewichtung (%)	Maestro	Gewichtete Bewertung (Maestro)	Appium	Gewichtete Bewertung (Appium)
Verifizierung der Funktionalität	25	5	1.25	5	1.25
Gemeinsame Testfälle	25	5	1.25	3	0.75
Unterstützung von UI-Elementen	20	5	1.00	3	0.60
Cross-Plattform Unterstützung	10	4	0.40	5	0.50
Benutzerfreundlichkeit und Dokumentation	10	4	0.40	2	0.20
Integration mit CI/CD Pipelines	10	4	0.40	4	0.40
Gesamtbewertung	100		4.70		3.70

Tabelle 6.1: Bewertungsmatrix – Auswertung

6.2 AUSWERTUNG DER BEWERTUNGSMATRIX

Nehmen wir nun unsere bereits definierte Bewertungsmatrix zur Hand. Hier kann pro Kriterium eine maximale Punktzahl von 5 Punkten erreicht werden. Anhand der im vorherigen Abschnitt erläuterten Ergebnissen evaluieren wir diese Punktzahl. Dabei kommen wir zu folgenden Ergebnissen:

- **Verifizierung der Funktionalität:** Beide Test-Frameworks konnten die Funktionalität der Anwendungen erfolgreich sicherstellen. Daher erhielten beide Frameworks die volle Punktzahl.
- **Gemeinsame Testfälle:** In Maestro funktionierten alle Testfälle in beiden Anwendungen ohne Anpassungen. Bei Appium hingegen waren Änderungen in den Anwendungen und den Testfällen notwendig. Daher erhielt Maestro die volle Punktzahl, während Appium aufgrund der erforderlichen Anpassungen Abzüge erhielt.
- **Unterstützung von UI-Elementen:** Maestro ermöglichte eine einheitliche Suche nach UI-Elementen unabhängig von der Plattform, was ihm die volle Punktzahl einbrachte. Appium benötigte hingegen unterschiedliche Anpassungen je nach Plattform, was zu Punktabzügen führte.
- **Cross-Plattform Unterstützung:** Maestro erhielt eine hohe Punktzahl für seine mobile Ausrichtung, während Appium aufgrund seiner Fähigkeit, sowohl mobile als auch Webanwendungen zu testen, die volle Punktzahl erhielt.
- **Benutzerfreundlichkeit und Dokumentation:** Maestro punktete mit einfacher Installation, Konfiguration und übersichtlicher Dokumentation. Appium hingegen war komplexer in der Handhabung und die Dokumentation war weniger klar strukturiert, was zu Abzügen führte.
- **Integration mit CI/CD Pipelines:** Beide Frameworks erhielten hier eine hohe Punktzahl. Maestro bietet eine kostspielige Cloud-Lösung, die

jedoch nicht alle benutzerdefinierten Erweiterungen unterstützt. Appium erforderte einen höheren manuellen Aufwand, bot jedoch umfassende Anpassungsmöglichkeiten.

Die Ergebnisse der Bewertungsmatrix und der detaillierten Analyse der einzelnen Kriterien zeigen (siehe [Tabelle 6.1](#)), dass sowohl Maestro als auch Appium spezifische Stärken und Schwächen aufweisen, jedoch grundsätzlich für den Einsatz in migrationsbezogenen Testumgebungen geeignet sind. Maestro erreichte eine Gesamtbewertung von 4,70, während Appium eine Gesamtbewertung von 3,70 erhielt. In dieser Untersuchung schnitt Maestro insgesamt besser ab, insbesondere aufgrund der Benutzerfreundlichkeit, der einheitlichen Unterstützung von UI-Elementen und der einfachen Handhabung gemeinsamer Testfälle. Appium zeichnete sich durch umfassende Cross-Plattform-Unterstützung und Flexibilität bei der Integration in CI/CD-Pipelines aus, zeigte jedoch Schwächen bei der Benutzerfreundlichkeit und Dokumentation. Diese Ergebnisse legen nahe, dass die Wahl des Test-Frameworks vorzugsweise anhand der spezifischen Anforderungen und Prioritäten des jeweiligen Anwendungsfalls getroffen werden sollte. Anwendungsbereiche, die auf schnelle und unkomplizierte mobile Tests fokussiert sind, könnten von der Nutzung von Maestro profitieren. Demgegenüber könnten komplexere Anwendungsbereiche mit vielfältigen Testanforderungen durch den Einsatz von Appium besser abgedeckt werden.

DISKUSSION

In der vorliegenden Arbeit wurde sich intensiv mit der Gestaltung und Bewertung von End-to-End-Tests für Cross-Plattform-Anwendungen beschäftigt, die von Flutter zu React Native migriert werden sollen. Dabei wurde klar, dass es durchaus möglich ist, ein End-to-End Test-Framework dafür zu verwenden, die Funktionalität der zu migrierenden Anwendungen sicherzustellen. Doch trotz dieser wertvollen Einblicke und Ergebnisse gibt es bestimmte methodische Einschränkungen und Schwächen, die beachtet werden sollten, um die Vollständigkeit und Aussagekraft der Untersuchung besser zu verstehen.

Eine wesentliche Einschränkung der Arbeit liegt in der begrenzten Auswahl der Test-Frameworks. Die Untersuchung konzentrierte sich auf Maestro und Appium. Andere potenziell geeignete Frameworks wurden nicht berücksichtigt. Eine umfassendere Analyse, die eine breitere Palette von Test-Frameworks einbezieht, hätte möglicherweise zu differenzierteren und umfassenderen Ergebnissen geführt.

Des Weiteren fehlen in der Arbeit Langzeitstudien, die die langfristige Wartbarkeit und Skalierbarkeit der Tests berücksichtigen. Die Bewertung der Frameworks basiert hauptsächlich auf kurzfristigen Test-szenarien und berücksichtigt nicht die potenziellen Herausforderungen und Vorteile, die bei der langfristigen Nutzung auftreten könnten. Langzeitstudien und -analysen wären hilfreich, um die Effizienz und Effektivität der Test-Frameworks im Laufe der Zeit besser zu verstehen.

Ein weiterer methodischer Schwachpunkt liegt in der eingeschränkten Testumgebung. Die durchgeführten Tests wurden hauptsächlich auf Android-Emulatoren durchgeführt. Tests auf physischen Geräten und unterschiedlichen Betriebssystemversionen wurden nicht umfassend berücksichtigt. Eine breitere Testumgebung hätte zu robusteren und aussagekräftigeren Ergebnissen führen können, insbesondere in Bezug auf die plattformübergreifende Funktionalität und Leistung.

Zusätzlich wurden in der Arbeit nicht-funktionale Anforderungen wie Leistung, Sicherheit und Energieverbrauch nicht berücksichtigt. Diese Aspekte werden auch als Hauptziele solcher End-to-End Tests gewertet [Sai+21; Kon+18] und sind für andere Entwickler möglicherweise von Bedeutung für die Gesamtbewertung eines Test-Frameworks. Eine umfassendere Analyse, die diese nicht-funktionalen Anforderungen einbezieht, wäre daher wünschenswert gewesen, war jedoch nicht der Fokus dieser Arbeit.

Schließlich ist die Gewichtung der Kriterien im Bewertungsprozess

subjektiv und könnte von anderen Entwicklern unterschiedlich wahrgenommen werden. Die in dieser Arbeit verwendete Methodik zur Gewichtung der Kriterien basierte auf den Präferenzen und Einschätzungen des Autors. Eine objektivere Methodik oder die Einbeziehung mehrerer Expertenmeinungen hätte möglicherweise zu ausgewogeneren und allgemein akzeptierbaren Ergebnissen geführt. Durch die Anwendung der Analytic Hierarchy Process (AHP)-Methode oder anderer Multi-Kriterien-Entscheidungsanalyse (MCDA)-Techniken hätte die Gewichtung objektiviert werden können, um die Präferenzen und Bewertungen mehrerer Experten systematisch zu erfassen und zu quantifizieren [Saa08]. Darüber hinaus hätten empirische Daten, wie Nutzungshäufigkeit und Nutzerzufriedenheit, durch Umfragen, Bewertungen oder Nutzungsanalysen in die Gewichtung oder sogar in die Bewertung einfließen können [Kit+02]. Andere Kriterien zur Bewertung dieser Test-Frameworks hätten potenziell auch zu abweichenden Ergebnissen führen können, wie die Popularität [CM23] oder wirtschaftliche Aspekte wie Kosten, einschließlich Lizenzgebühren und betriebsinterner Aufwendungen für den Aufbau der Testinfrastruktur [AAS21]. Zudem werden Kriterien wie Flexibilität und Erweiterbarkeit betrachtet, welche die Anpassungsfähigkeit eines Test-Frameworks an spezifische Anforderungen und die Unterstützung verschiedener Programmiersprachen und Plattformen einschließen [Ama+17]. Andere Arbeiten heben die Bedeutung der Robustheit gegenüber unvorhersehbaren Ereignissen, wie Netzwerkfehler, sowie die Leistung während der Testausführung hervor [Che16; SS21; CM23]. Obwohl einige dieser Kriterien im Kontext der genannten Arbeiten valide Bewertungsgrundlagen darstellen, eigneten sie sich nicht im spezifischen Kontext dieser Untersuchung, die sich primär auf die Funktionalität gegenüber Anwendungen zweier Cross-Plattform-Frameworks konzentrierte. Bei ausschließlicher Betrachtung der vom Autor festgelegten Kriterien hätte eine alternative Gewichtung dieser Kriterien zu abweichenden Ergebnissen führen können. In einigen Quellen werden häufig die Kriterien der Integration in eine CI/CD-Pipeline oder der Cross-Plattform Unterstützung genannt [Gra; Tho]. Würden beiden Kriterien mehr Gewicht beigemessen, könnten die Ergebnisse knapper ausfallen. Insgesamt würde eine unterschiedliche Gewichtung zwar den Abstand der Bewertungen verringern, jedoch das endgültige Ergebnis nicht maßgeblich beeinflussen. Eine maßgebliche Veränderung des Ergebnisses würde erst durch die Einführung weiterer Kriterien in die Bewertungsmatrix erreicht.

Abschließend war ein wesentlicher einschränkender Faktor dieser Arbeit der begrenzte zeitliche Rahmen von drei Monaten. Diese zeitliche Einschränkung beeinträchtigte die Möglichkeit, umfassende Untersuchungen durchzuführen, die eine tiefere Analyse und Bewertung der Test-Frameworks erlaubt hätten. Der enge Zeitrahmen führte dazu, dass nur eine begrenzte Anzahl von Test-Frameworks berücksichtigt und evaluiert werden konnte und Langzeitstudien sowie Tests auf physischen Geräten nicht durchgeführt wurden. Eine verlängerte Forschungsdauer hätte potenziell detailliertere und robustere Ergebnisse geliefert, da mehr Zeit für die Durchführung umfangreicherer Tests und die Berücksichtigung zusätzlicher Faktoren zur Verfügung gestanden hätte.

8

ZUSAMMENFASSUNG

Zusammenfassend lässt sich sagen, dass die Arbeit wertvolle Erkenntnisse über die Eignung von Maestro und Appium für die End-to-End-Testung von Cross-Plattform-Anwendungen liefert. Die Implementierungen und Testszenarien in beiden Frameworks haben gezeigt, dass beide Tools ihre spezifischen Stärken und Schwächen haben, jedoch für das Szenario der Migration gut geeignet sind. Maestro überzeugt dabei durch seine Benutzerfreundlichkeit, Effizienz und einfache plattformübergreifende Implementierung, während Appium durch seine Vielseitigkeit und umfassende Anpassungsfähigkeit besticht. Gleichzeitig gibt es methodische Einschränkungen und Schwächen, die die Aussagekraft der Ergebnisse beeinträchtigen könnten.

Zukünftige Arbeiten könnten von einer breiteren Methodik und der Berücksichtigung zusätzlicher Faktoren profitieren, um noch robustere und umfassendere Ergebnisse zu erzielen. Insbesondere eine längere Forschungsdauer, die Einbeziehung einer breiteren Palette von Test-Frameworks, Langzeitstudien zur Wartbarkeit und Skalierbarkeit sowie Tests auf physischen Geräten und unterschiedlichen Betriebssystemversionen könnten zu detaillierteren und belastbareren Ergebnissen führen. Dies würde eine fundiertere Basis für die Entscheidung über die Wahl eines End-to-End Test-Frameworks bieten und dazu beitragen, die Effizienz und Qualität der Testprozesse in plattformübergreifenden Entwicklungsprojekten weiter zu verbessern.

Teil II
APPENDIX

a

ANHANG

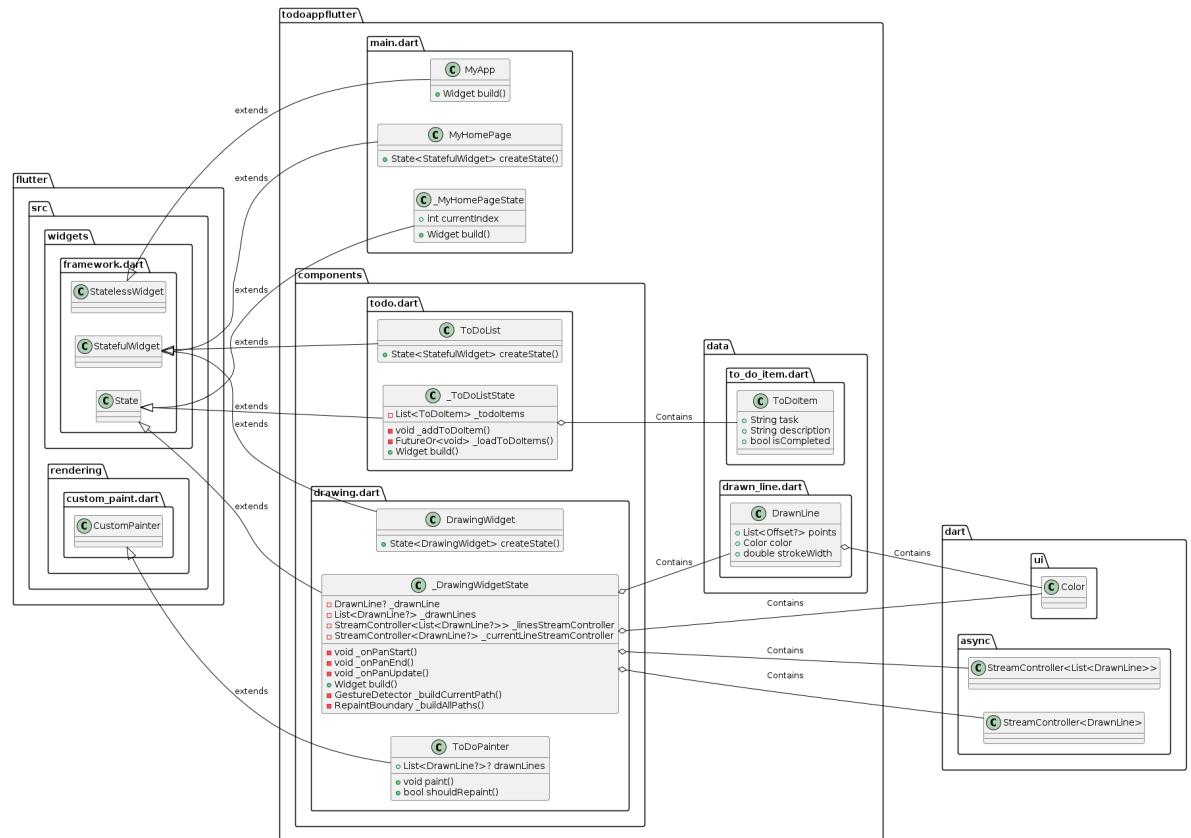
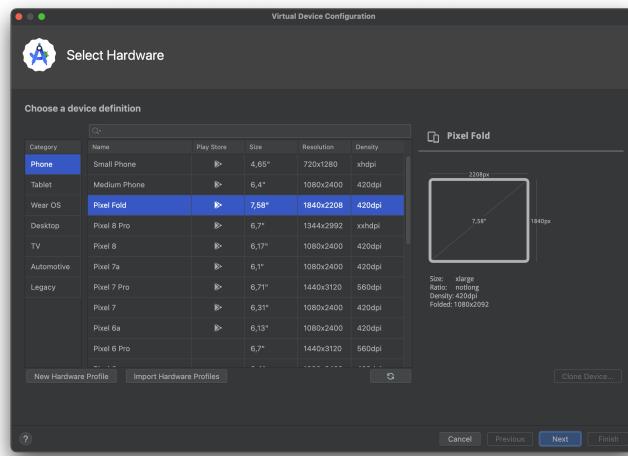
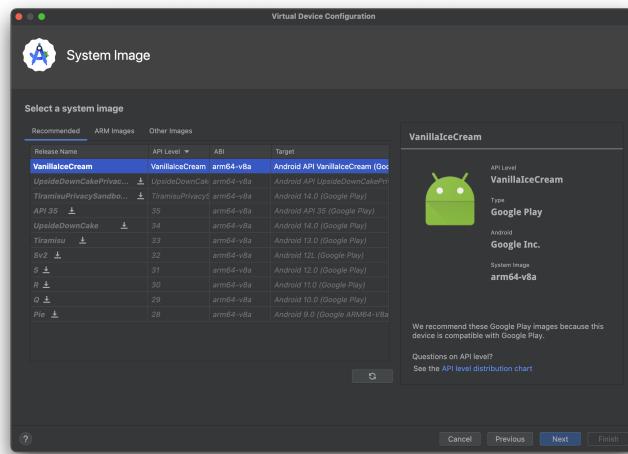


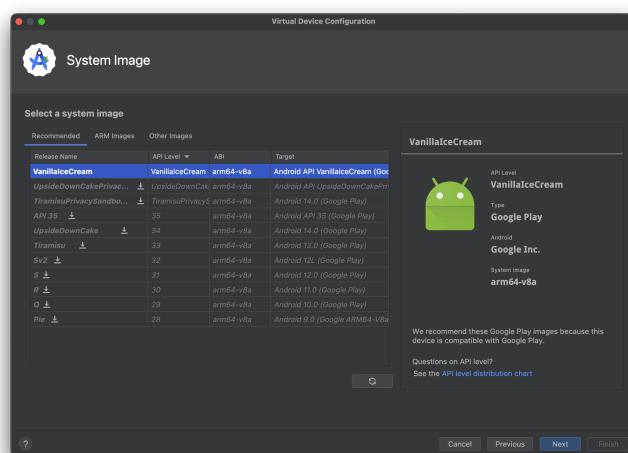
Abbildung a.1: UML-Diagramm der Flutter Anwendung



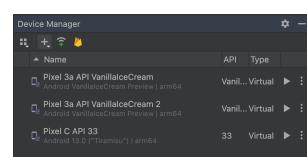
(a) Android Studio – Emulator erstellen – Auswahl des Gerätes



(b) Android Studio – Emulator erstellen – Auswahl der Android-Version



(c) Android Studio – Emulator erstellen – Weitere Konfigurationen



(d) Android Studio – Device Manager

Abbildung a.2: Android Studio – Schritte zur Erstellung eines Emulatoren

```
Running on emulator-5554
> Flow
  ✓ Launch app "com.example.todoappflutter"
  ✓ Go to drawing page
  ✓ Swipe from (10%,13%) to (60%,33%) in 200 ms
  ✓ Swipe from (40%,59%) to (77%,70%) in 200 ms
  ✓ Swipe from (39%,59%) to (78%,73%) in 200 ms
  ✓ Swipe from (15%,44%) to (65%,45%) in 200 ms
  ✓ Swipe from (20%,34%) to (70%,17%) in 200 ms
  ✓ Swipe from (40%,18%) to (39%,59%) in 200 ms
  ✓ Take screenshot DrawingPageActual
  ✓ Compare Images
  ✓ Run ${console.log(output.diff)}
    Log messages:
      0.43
  ✘ Images are similar

Assertion is false: false is true
===== Debug output (logs & screenshots) =====
```

Abbildung a.3: Fehlerfall – Maestro Test für Flutter

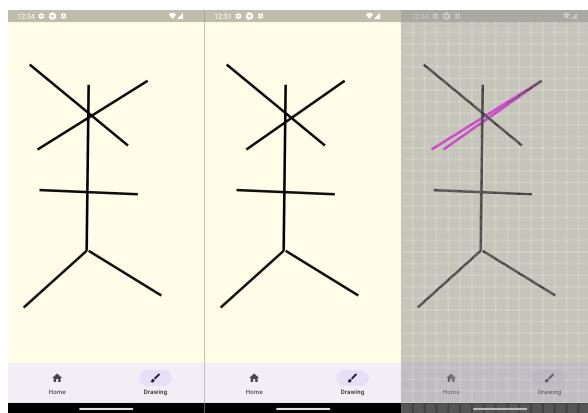


Abbildung a.4: Vergleichsbilder des visuellen Vergleichs – Fehlerfall a.3

LITERATUR

- [AAS21] Asma J. Abdulwareth und Asma A. Al-Shargabi. "Towards a Multi-Criteria Framework for Selecting Software Testing Tools". In: *IEEE Access* PP (2021), S. 1–1. DOI: [10.1109/ACCESS.2021.3128071](https://doi.org/10.1109/ACCESS.2021.3128071).
- [Reaa] *Accessibility · React Native - reactnative.dev*. <https://reactnative.dev/docs/accessibility>. [Aufgerufen 09-07-2024].
- [ATY20] Yu Adachi, Haruto Tanno und Yu Yoshimura. "A Method to Mask Dynamic Content Areas Based on Positional Relationship of Screen Elements for Visual Regression Testing". In: *2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC)*. 2020, S. 1755–1760. DOI: [10.1109/COMPSAC48688.2020.000-1](https://doi.org/10.1109/COMPSAC48688.2020.000-1).
- [Flua] *Add interactivity to your Flutter app - docs.flutter.dev*. <https://docs.flutter.dev/ui/interactivity>. [Aufgerufen 10-07-2024].
- [Ama+17] Domenico Amalfitano, Nicola Amatucci, A. Memon, Porfirio Tramontana und A. R. Fasolino. "A general framework for comparing automatic testing techniques of Android mobile apps". In: *J. Syst. Softw.* 125 (2017), S. 322–343. DOI: [10.1016/j.jss.2016.12.017](https://doi.org/10.1016/j.jss.2016.12.017).
- [Flub] *An introduction to integration testing - docs.flutter.dev*. <https://docs.flutter.dev/cookbook/testing/integration/introduction>. [Aufgerufen 26-04-2024].
- [Appa] *Appium Clients - Appium Documentation - appium.io*. <https://appium.io/docs/en/latest/ecosystem/clients/>. [Aufgerufen 18-07-2024].
- [Appb] *Appium Drivers - Appium Documentation - appium.io*. <https://appium.io/docs/en/latest/ecosystem/drivers/>. [Aufgerufen 10-07-2024].
- [Weba] *Appium-Setup | WebdriverIO - webdriver.io*. <https://webdriver.io/de/docs/appium>. [Aufgerufen 10-07-2024].
- [Appc] *Appium in a Nutshell - Appium Documentation - appium.io*. <https://appium.io/docs/en/latest/intro/>. [Aufgerufen 26-04-2024].
- [Anda] *Apps im Android-Emulator ausführen | Android Studio | Android Developers - developer.android.com*. <https://developer.android.com/studio/run/emulator?hl=de>. [Aufgerufen 03-07-2024].

- [Andb] *Automatisierte Tests mit UI Automator schreiben | Android Developers - developer.android.com.* <https://developer.android.com/training/testing/other-components/ui-automator?hl=de>. [Aufgerufen 05-07-2024].
- [Fluc] *Check app functionality with an integration test - docs.flutter.dev.* <https://docs.flutter.dev/testing/integration-tests>. [Aufgerufen 18-07-2024].
- [Che16] Yury Chernov. *Test-Data Quality as a Success Factor for End-to-End Testing - An Approach to Formalisation and Evaluation.* 2016. DOI: [10.5220/0005971700950101](https://doi.org/10.5220/0005971700950101).
- [Reab] *Communication between native and React Native · React Native - reactnative.dev.* <https://reactnative.dev/docs/communication-ios>. [Aufgerufen 17-07-2024].
- [Webb] *Configuration | WebdriverIO - webdriver.io.* <https://webdriver.io/docs/configuration>. [Aufgerufen 05-07-2024].
- [Reac] *Core Components and Native Components · React Native - reactnative.dev.* <https://reactnative.dev/docs/intro-react-native-components>. [Aufgerufen 10-07-2024].
- [CM23] Gustavo Costa und Breno Miranda. "A Comparative Analysis of Mobile UI Testing Frameworks in Continuous Integration Environments". In: *Proceedings of the 8th Brazilian Symposium on Systematic and Automated Software Testing* (2023). DOI: [10.1145/3624032.3624047](https://doi.org/10.1145/3624032.3624047).
- [Ddi] *DDiT - Forschungsprojekt.* <https://www.lidia-hessen.de/projekte/ddit>. [Aufgerufen 18-07-2024].
- [Andc] *Einführung in Aktivitäten | Android Developers - developer.android.com.* <https://developer.android.com/guide/components/activities/intro-activities?hl=de>. [Aufgerufen 08-07-2024].
- [Wixa] *Environment Setup | Detox - wix.github.io.* <https://wix.github.io/Detox/docs/introduction/environment-setup>. [Aufgerufen 11-07-2024].
- [Flud] *FAQ - docs.flutter.dev.* <https://docs.flutter.dev/resources/faq>. [Aufgerufen 10-07-2024].
- [Gita] *[Feature Request] Visual Regression Testing · Issue #1222 · mobile-dev-inc/maestro - github.com.* <https://github.com/mobile-dev-inc/maestro/issues/1222>. [Aufgerufen 04-07-2024].
- [Flue] *Flutter - Build apps for any screen — flutter.dev.* <https://flutter.dev>. [Aufgerufen 18-07-2024].
- [Moba] *Flutter | Maestro by mobile.dev - maestro.mobile.dev.* <https://maestro.mobile.dev/platform-support/flutter>. [Aufgerufen 10-07-2024].

- [Fluf] *Flutter architectural overview - docs.flutter.dev.* <https://docs.flutter.dev/resources/architectural-overview>. [Aufgerufen 10-07-2024].
- [Rad] *Flutter vs React Native: Which One is the Best Framework? - radixweb.com.* <https://radixweb.com/blog/flutter-vs-react-native>. [Aufgerufen 17-07-2024].
- [Webc] *Frameworks | WebdriverIO - webdriver.io.* <https://webdriver.io/docs/frameworks>. [Aufgerufen 11-07-2024].
- [Read] *Get Started with React Native · React Native — reactnative.dev.* <https://reactnative.dev/docs/environment-setup>. [Aufgerufen 21-07-2024].
- [Wixb] *Getting Started | Detox - wix.github.io.* <https://wix.github.io/Detox/docs/introduction/getting-started>. [Aufgerufen 26-04-2024].
- [Webd] *Getting Started | WebdriverIO - webdriver.io.* <https://webdriver.io/docs/gettingstarted/>. [Aufgerufen 05-07-2024].
- [Gitb] *GitHub - mobile-dev-inc/maestro at v1.0.0 - github.com.* <https://github.com/mobile-dev-inc/maestro/tree/v1.0.0>. [Aufgerufen 26-04-2024].
- [Gitic] *GitHub - mojoaxel/awesome-regression-testing: A curated list of resources around the topic: visual regression testing — github.com.* <https://github.com/mojoaxel/awesome-regression-testing?tab=readme-ov-file>. [Aufgerufen 21-07-2024].
- [Gitd] *GitHub - rsmb1/Resemble.js: Image analysis and comparison - github.com.* <https://github.com/rsmb1/Resemble.js?tab=readme-ov-file>. [Aufgerufen 04-07-2024].
- [Gite] *GitHub - software-mansion/react-native-gesture-handler: Declarative API exposing platform native touch and gesture system to React Native. - github.com.* <https://github.com/software-mansion/react-native-gesture-handler>. [Aufgerufen 10-07-2024].
- [Gra] Joe Gray. *Mastering Android App Quality: 5 Key Testing Frameworks for 2024 - saiyan.jo147th248.* <https://medium.com/@saiyan.jo147th248/mastering-android-app-quality-5-key-testing-frameworks-for-2024-222b7a227999>. [Aufgerufen 19-07-2024].
- [Andd] *Grundlagen zum Testen von Android-Apps | Android Developers - developer.android.com.* <https://developer.android.com/training/testing/fundamentals?hl=de>. [Aufgerufen 12-06-2024].

- [Wixc] *How Detox Works | Detox - wix.github.io.* <https://wix.github.io/Detox/docs/articles/how-detox-works>. [Aufgerufen 11-07-2024].
- [Appd] *How Does Appium Work? - Appium Documentation - appium.io.* <https://appium.io/docs/en/latest/intro/appium/>. [Aufgerufen 05-07-2024].
- [Flug] *Inside Flutter - docs.flutter.dev.* <https://docs.flutter.dev/resources/inside-flutter>. [Aufgerufen 10-07-2024].
- [Appe] *Install Appium - Appium Documentation - appium.io.* <https://appium.io/docs/en/latest/quickstart/install/>. [Aufgerufen 05-07-2024].
- [Appf] *Install the UiAutomator2 Driver - Appium Documentation - appium.io.* <https://appium.io/docs/en/2.0/quickstart/ui-autom2-driver/>. [Aufgerufen 05-07-2024].
- [Mobb] *Installing Maestro | Maestro by mobile.dev - maestro.mobile.dev.* <https://maestro.mobile.dev/getting-started/installing-maestro>. [Aufgerufen 04-07-2024].
- [Appg] *Intro to Appium Drivers - Appium Documentation - appium.io.* <https://appium.io/docs/en/latest/intro/drivers/>. [Aufgerufen 05-07-2024].
- [Reae] *Introducing Hooks.* Aufgerufen: 2024-07-10. URL: <https://reactjs.org/docs/hooks-intro.html>.
- [Reaf] *Introduction to React.* Aufgerufen: 2024-07-10. URL: <https://reactjs.org/docs/getting-started.html>.
- [Tma] *Introduction to quality engineering and testing - tmap.net.* <https://www.tmap.net/page/introduction-quality-engineering-and-testing>. [Aufgerufen 04-06-2024].
- [Reag] *JSX in Depth.* Aufgerufen: 2024-07-10. URL: <https://reactjs.org/docs/jsx-in-depth.html>.
- [Kit+02] Barbara Kitchenham, S.L. Pfleeger, L.M. Pickard, Peter Jones, David Hoaglin, Khaled Emam und Jarrett Rosenberg. "Preliminary Guidelines for Empirical Research in Software Engineering". In: *Software Engineering, IEEE Transactions on* 28 (Sep. 2002), S. 721–734. DOI: [10.1109/TSE.2002.1027796](https://doi.org/10.1109/TSE.2002.1027796).
- [Kon+18] Pingfan Kong, Li Li, Jun Gao, Kui Liu, Tegawendé Bissyandé und Jacques Klein. "Automated Testing of Android Apps: A Systematic Literature Review". In: *IEEE Transactions on Reliability PP* (Sep. 2018), S. 1–22. DOI: [10.1109/TR.2018.2865733](https://doi.org/10.1109/TR.2018.2865733).
- [Fluh] *Layouts in Flutter - docs.flutter.dev.* <https://docs.flutter.dev/ui/layout>. [Aufgerufen 10-07-2024].

- [Lel23] Panagiotis Leloudas. "Software Testing Types and Techniques". In: *Introduction to Software Testing: A Practical Guide to Testing, Design, Automation, and Execution*. Berkeley, CA: Apress, 2023, S. 5–34. ISBN: 978-1-4842-9514-4. DOI: [10.1007/978-1-4842-9514-4_2](https://doi.org/10.1007/978-1-4842-9514-4_2). URL: https://doi.org/10.1007/978-1-4842-9514-4_2.
- [Leo+16] Maurizio Leotta, Diego Clerissi, Filippo Ricca und Paolo Tonella. "Chapter Five - Approaches and Tools for Automated End-to-End Web Testing". In: Hrsg. von Atif Memon. Bd. 101. Advances in Computers. Elsevier, 2016, S. 193–237. DOI: <https://doi.org/10.1016/bs.adcom.2015.11.007>. URL: <https://www.sciencedirect.com/science/article/pii/S0065245815000686>.
- [Mobc] *Maestro Studio | Maestro by mobile.dev - maestro.mobile.dev.* <https://maestro.mobile.dev/getting-started/maestro-studio>. [Aufgerufen 04-07-2024].
- [Moc] *Mocha - the fun, simple, flexible JavaScript test framework - mochajs.org*. <https://mochajs.org>. [Aufgerufen 05-07-2024].
- [MB23] Humaid Mollah und Petra van den Bos. "From User Stories to End-to-end Web Testing". In: *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 2023, S. 140–148. DOI: [10.1109/ICSTW58534.2023.00036](https://doi.org/10.1109/ICSTW58534.2023.00036).
- [Not] Jay's Notebook. *Javascript Testing with Mocha (Everything You Need To Know) - JaysNotebook*. <https://medium.com/@JaysNotebook/learning-mocha-everything-you-need-to-know-74e8200f90ec>. [Aufgerufen 10-07-2024].
- [Reah] *Optimizing Performance*. Aufgerufen: 2024-07-10. URL: <https://reactjs.org/docs/optimizing-performance.html>.
- [O9] G. O'Regan. "Fundamentals of software testing". In: *Concise Guide to Software Testing* (2019), S. 59–78. DOI: [10.1007/978-3-030-28494-7_3](https://doi.org/10.1007/978-3-030-28494-7_3).
- [Wixd] *Preparing for CI | Detox - wix.github.io*. <https://wix.github.io/Detox/docs/introduction/preparing-for-ci>. [Aufgerufen 11-07-2024].
- [Reai] *Quick Start – React — react.dev*. <https://react.dev/learn>. [Aufgerufen 21-07-2024].
- [Reaj] *React Ecosystem*. Aufgerufen: 2024-07-10. URL: <https://reactjs.org/community/ecosystem.html>.
- [Reak] *React Native - A framework for building native apps using React*. Aufgerufen: 2024-07-10. URL: <https://reactnative.dev/>.

- [Mobd] *React Native | Maestro by mobile.dev - maestro.mobile.dev.* <https://maestro.mobile.dev/platform-support/react-native>. [Aufgerufen 10-07-2024].
- [Real] *React Native Paper - reactnativepaper.com.* <https://reactnativepaper.com>. [Aufgerufen 10-07-2024].
- [Red] *Redux - A JS library for predictable and maintainable global state management | Redux - redux.js.org.* <https://redux.js.org>. [Aufgerufen 18-07-2024].
- [Rsm] *Resemble.js : Image analysis - rsmbi.github.io.* <http://rsmbi.github.io/Resemble.js/>. [Aufgerufen 04-07-2024].
- [Saa08] Thomas Saaty. "Decision making with the Analytic Hierarchy Process". In: *Int. J. Services Sciences Int. J. Services Sciences* 1 (Jan. 2008), S. 83–98. doi: [10.1504/IJSSCI.2008.017590](https://doi.org/10.1504/IJSSCI.2008.017590).
- [Sai+21] Kabir S. Said, Liming Nie, Adekunle A. Ajibode und Xueyi Zhou. "GUI Testing for Mobile Applications: Objectives, Approaches and Challenges". In: *Proceedings of the 12th Asia-Pacific Symposium on Internetware*. Internetware '20. Singapore, Singapore: Association for Computing Machinery, 2021, 51–60. ISBN: 9781450388191. doi: [10.1145/3457913.3457931](https://doi.org/10.1145/3457913.3457931). URL: <https://doi.org/10.1145/3457913.3457931>.
- [Webe] *Selektoren | WebdriverIO - webdriver.io.* <https://webdriver.io/de/docs/selectors/>. [Aufgerufen 09-07-2024].
- [SS21] Alla Shtokal und J. Smołka. "Comparative analysis of frameworks used in automated testing on example of TestNG and WebdriverIO". In: *Journal of Computer Sciences Institute* (2021). doi: [10.35784/jcsi.2595](https://doi.org/10.35784/jcsi.2595).
- [Sta] *Stack Overflow Developer Survey 2022 - survey.stackoverflow.co.* <https://survey.stackoverflow.co/2022#section-most-popular-technologies-other-frameworks-and-libraries>. [Aufgerufen 25-04-2024].
- [Apph] *System Requirements - Appium Documentation - appium.io.* <https://appium.io/docs/en/latest/quickstart/requirements/>. [Aufgerufen 05-07-2024].
- [Tea23] Flutter Team. "Impeller: A New Rendering Engine for Flutter". In: *Flutter Documentation* (2023). [Aufgerufen 10-07-2024]. URL: <https://docs.flutter.dev/perf/impeller>.
- [Ream] *Thinking in React.* Aufgerufen: 2024-07-10. URL: <https://reactjs.org/docs/thinking-in-react.html>.

- [Tho] Sarah Thomas. *Best Automation Tools For Mobile App Testing (Pros & Cons)* - [sarah.thoma.456.medium.com/@sarah.thoma.456/best-automation-tools-for-mobile-app-testing-978b348def25](https://medium.com/@sarah.thoma.456/best-automation-tools-for-mobile-app-testing-978b348def25). [Aufgerufen 18-07-2024].
- [Webf] *Visual Testing* | WebdriverIO - [webdriver.io](https://webdriver.io/docs/visual-testing). <https://webdriver.io/docs/visual-testing>. [Aufgerufen 04-07-2024].
- [W3s] W3Schools.com - [w3schools.com](https://www.w3schools.com/xml/xpath_syntax.asp). https://www.w3schools.com/xml/xpath_syntax.asp. [Aufgerufen 09-07-2024].
- [Wan+15] Mian Wan, Yuchen Jin, Ding Li und William G. J. Halfond. "Detecting Display Energy Hotspots in Android Apps". In: *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. 2015, S. 1–10. DOI: [10.1109/ICST.2015.7102585](https://doi.org/10.1109/ICST.2015.7102585).
- [Mobe] *Welcome* | Maestro Cloud Documentation - cloud.mobile.dev. <https://cloud.mobile.dev>. [Aufgerufen 10-07-2024].
- [Mobf] *What is Maestro?* - maestro.mobile.dev. <https://maestro.mobile.dev>. [Aufgerufen 15-01-2024].
- [Appi] *Write a Test (JS)* - Appium Documentation - [appium.io](https://appium.io/docs/en/2.0/quickstart/test-js/). <https://appium.io/docs/en/2.0/quickstart/test-js/>. [Aufgerufen 10-07-2024].
- [YYR13] Shengqian Yang, Dacong Yan und Atanas Rountev. "Testing for poor responsiveness in android applications". In: *2013 1st International Workshop on the Engineering of Mobile-Enabled Systems (MOBS)*. 2013, S. 1–6. DOI: [10.1109/MOBS.2013.6614215](https://doi.org/10.1109/MOBS.2013.6614215).
- [Wixe] *Your First Test* | Detox - [wix.github.io](https://wix.github.io/Detox/docs/introduction/your-first-test). <https://wix.github.io/Detox/docs/introduction/your-first-test>. [Aufgerufen 11-07-2024].
- [Gitf] *appium-uiautomator2-driver/.github/workflows/functional-test.yml at master* · [appium/appium-uiautomator2-driver](https://github.com/appium/appium-uiautomator2-driver) - [github.com](https://github.com/appium/appium-uiautomator2-driver/blob/master/.github/workflows/functional-test.yml). <https://github.com/appium/appium-uiautomator2-driver/blob/master/.github/workflows/functional-test.yml>. [Aufgerufen 10-07-2024].
- [Gitg] *flutter/packages/flutter/lib/src/material/list_tile.dart* — [github.com](https://github.com/flutter/flutter/blob/master/packages/flutter/lib/src/material/list_tile.dart). https://github.com/flutter/flutter/blob/master/packages/flutter/lib/src/material/list_tile.dart. [Aufgerufen 09-07-2024].
- [Pub] *golden_toolkit* | Flutter package — [pub.dev](https://pub.dev/packages/golden_toolkit). https://pub.dev/packages/golden_toolkit. [Aufgerufen 21-07-2024].