Hochschule **Rosenheim**
University of Applied Sciences

Fakultät für Informatik

Studiengang Wirtschaftsinformatik

Vergleich von JavaScript-E2E-Test Werkzeugen für Single-Page-Anwendungen am Bespiel einer Angular-Anwendung

Seminararbeit

von

Dominik Ampletzer

Datum der Abgabe: 24.06.2019
Erstprüfer: Herr Reimer
Zweitprüfer: Prof. Beneken

ERKLÄRUNG

Ich versichere, dass ich diese Arbeit selbständig angefertigt, nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Rosenheim den, 24.06.2019

Dominik Ampletzer

Abstract

Diese Seminararbeit beschäftigt sich mit dem Vergleich dreier E2E-Test-Werkzeuge, welche auf JavaScript basieren und für den Einsatz mit Web-Anwendungen entwickelt wurden. Bei diesen Werkzeugen handelt es sich um Protractor, Nightwatch.js und WebdriverIO. Diese wurden einem Kurzcheck unterzogen, welche u.a. Lizenz, Reifegrad, Support, Dokumentation, Projekt-Aktivität und Bekanntheitsgrad beinhaltet. Auf einem Kurzcheck basierend, wurden dann die zwei besten Werkzeuge an einer Angular-Anwendung mit verschidenen Anwendungsfällen getestet. Dadurch sollten Integrationsfähigkeit, Aufwand und Leistungs- bzw. Fähigkeitsumfang der Werkzeuge getestet werden. Aus dem Kurzcheck und der Testimplementierung sollten ersichtlich werden, welches Werkzeug im Spezialfall von Angular-Anwendungen, aber auch im Allgemeinen das bessere ist und demnach verwendet werden soll.

Es wurde ersichtlich, dass Protractor im Spezialfall von Angular-Anwendungen wie auch im Allgemeinen das bessere Werkzeug ist. Demnach kann Protractor vor WebdriverIO und WebdriverIO vor Nightwatch.js empfohlen werden.

Schlagworte

E2E-Werkzeuge, Angular, Protractor, Nightwatch.js, WebdriverIO

Inhaltsverzeichnis

Abbildungsverzeichnis

Glossar / Abkürzungsverzeichnis

UI	User-Interface – Benutzeroberflächen
HTTP	Hypertext Transfer Protocol – Datenübertragungsprotokoll welches u.a. dazu genutzt wird, um Web-Seiten / Web-Anwendungen zu transportieren
E2E	End-To-End – End zu End, dahinter versteckt sich der komplette Ablauf in einem System durch alle Schichten. Von der Oberfläche über den Server zur Datenbank und zurück
Shadowing	Überschatten von Variablennamen durch mehrmaliges Verwenden innerhalb eines Funktionsbereiches
GPL	Gnu Public Licence – weit verbreitete offene Lizenz im Open-Source-Bereich
MIT-LIZENZ	Massachusetts Institute of Technology Lizenz - eine Lizenz, welche vom MIT stammt und sehr freie Nutzung erlaubt

1 Einleitung

In dieser Arbeit geht es um die Analyse und den Vergleich von drei End-to-End-Test-Werkzeugen (E2E-Test). Die Werkzeuge simulieren den Anwender und überprüfen die Reaktionen bzw. das Ergebnis der Interaktion mit einer Web-Anwendungen. Die Werkzeuge sind für das Testen von Web-Anwendungen und Web-Seiten geeignet. Aus diesem Grund wird als testfallgebende Anwendung eine Single-Page-Anwendung verwendet, welche mit dem Framework Angular entwickelt wurde.

1.1 Problemstellung

Im Web-Umfeld gibt es sehr viele Test-Werkzeuge und dies ist für E2E-Tests ähnlich. Deshalb stellt sich die Frage, welches Werkzeug eignet sich am besten. Da viele Web-Anwendungen als Single-Page-Anwendung entwickelt werden, geht es speziell um die Frage, welches E2E-Werkzeug für Single-Page-Anwendungen am besten geeignet ist.

1.2 Vorgehen & Ziel der Arbeit

Diese Arbeit gibt einen kurzen Überblick über Protractor, Nightwatch.js und WebdriverIO. Mittels einer Kurzüberprüfung werden die Werkzeuge analysiert und das schlechterste Werkzeug verworfen, um die beiden besseren Werkzeuge anhand einer Beispielsanwendung, welche mit dem Framework Angular entwickelt wurde, zu testen. Hierfür wurden sowohl die Implementierung als auch verschiedene Testfälle betrachtet.

Ziel der Arbeit ist es unter Berücksichtigung der Kurzüberprüfung und des Werkzeug-Tests, die Aussage treffen zu können, welches E2E-Test-Werkzeug für Angular-Anwendungen im Speziellen und Single-Page-Anwendungen im Allgemeinen zu verwenden ist.

1.3 Rahmenbedingungen

Die Werkzeuge, welche in dieser Arbeit betrachtet werden und ausgewählt wurden, sind folgenden Rahmenbedingungen unterworfen:

- Programmiersprache JavaScript und/oder TypeScript
- Außerhalb der Beta-Phase
- Node.js basierend
- WebDriver API basierend
- Interessant für den Autor ☺

2 E2E-Test-Werkzeuge

Einführend ist zu den Test-Werkzeugen zu erwähnen, dass die Begriffe UI-Test-Werkzeuge und E2E-Test-Werkzeuge als Synonym betrachtet werden.

In Abbildung 1: Vereinfachter Ablauf ist zu sehen, wie die Interaktion mit einer Web-Anwendung vonstatten geht. Der Anwender interagiert mit der Benutzeroberfläche (UI), welche ein Teil der Web-Anwendung ist. Die Web-Anwendung stellt Anfragen bei einem Server, welcher antwortet. Diese Antworten werden in welcher Form auch immer auf der UI angezeigt, was der Anwender wiederum wahrnimmt.

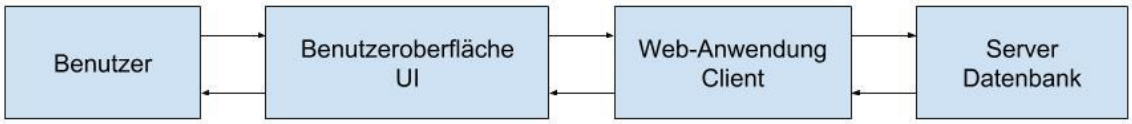

Abbildung 1: Vereinfachter Ablauf

Bei einem E2E-Test-Werkzeug wird der Benutzer durch das Werkzeug ersetzt, was zur Folge hat, dass die Interaktion mit der UI durch das Werkzeug passiert. Dadurch kann geprüft werden, ob die UI so reagiert wie gewünscht und ob der Ablauf durch das komplette System und zurück so funktioniert wie erwartet. Außerdem kann eine Automatisierung durch das E2E-Test-Werkzeug geschehen, siehe Abbildung 2: Ablauf mit E2E-Test-Werkzeug.

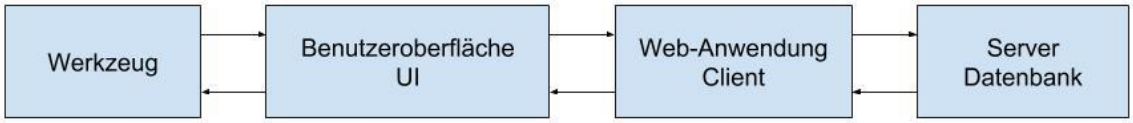

Abbildung 2: Ablauf mit E2E-Test-Werkzeug

2.1 Kurzüberprüfung / Schnelltest

Für die Kurzüberprüfung wurde ein Katalog mit folgenden Kriterien zusammengestellt:

- Lizenz: Art der Lizenz, eingestuft nach MIT/Apache – GPL – proprietär
- Reifegrad: Existenzzeit und stabile Veröffentlichung, Versionsnummer
- Support: Öffentliche Unterstützung und professionelle Unterstützung
- Dokumentation: Qualität der Dokumentation, Tutorials, Starthilfen, Alter der Dokumentation
- Qualität des Projekts: Testabdeckung / Bugs durch statische Code-Analyse
- Aktivitäten: Stetigkeit von Commits, letzte Veröffentlichung, Anzahl aktiver Entwickler
- Bekanntheitsgrad: öffentliches Interesse am Werkzeug anhand von Trendanalysen, Foren Aktivitäten
- Unterstützer des Projekts: Firma oder Personen die hinter dem Werkzeug stehen, Rückschluss auf Langlebigkeit

Diese Kriterien werden als schwerwiegend betrachtet und können dazu führen, dass bei sehr schlechter Beurteilung eines Kriteriums das Werkzeuge nicht zu empfehlen ist. So ist Beispielsweise ein Werkzeug, welches keine Dokumentation hat, oder seit Jahren nicht mehr gepflegt wird, nicht für produktiven Einsatz zu empfehlen.

2.2 Protractor[1]

Protractor ist ein Werkzeug welches 2013 in der Version 0.2 durch das JavaScript-Framework AngularJS zur Verfügung steht. Es wurde für AngularJS[2] entwickelt, kann mittlerweile jedoch

[1] https://www.protractortest.org
[2] AngularJS und Angular unterscheiden sich stark voneinander und sind somit nicht das gleiche Framework

auch für andere Web-Anwendungen verwendet werden. Dennoch liegt der Schwerpunkt von Protractor in der Verwendung mit Angular was der Nachfolger von AngularJS ist. Hinter dem JavaScript-Framework Angular steht wiederum Google.

Protractor wurde demnach für das Angular Ökosystem entwickelt, kann aber ohne weiteres mit anderen Frameworks wie React verwendet werden. Hierfür gibt es eine *waitForAngular*-Funktion, welche deaktiviert werden kann. Auch das meist in Angular verwendete Test-Framework Jasmine kann durch andere Test-Frameworks wie Mocha oder Cucumber ersetzt werden.

Neben JavaScript können Protractor Tests mit TypeScript geschrieben werden, was besonders in Angular- Anwendungen, den Vorteil mit sich bringt, sich nicht an andere Syntax gewöhnen zu müssen.

2.2.1 Lizenz

MIT

2.2.2 Reifegrad

Der Reifegrad ist als hoch anzusehen, da das Werkzeug seit 2013 existiert und mittlerweile bei der Version 6.0.0 angekommen ist. Des Weiteren haben seit 2013 ca. 100 Veröffentlichungen stattgefunden. Aktuell gibt es 13 Bugs in der offiziellen Issue-Liste und die aktuelle Version ist stabil.

2.2.3 Support

Leider kein professioneller Support. Der Community-Support[3] hingegen ist mit StackOverflow (ca. 10.000 Fragen), Gitter (ca. 1.250 Nutzer) und Angular-Discussion-Group (ca. 19.000 Themen) sehr gut.

2.2.4 Dokumentation

Auf der Protractor Web-Seite findet sich neben einem Tutorial und einem Schnelleinstieg einige Beschreibungen mit Beispielen für die verschiedenen Bereiche von Protractor. Die Dokumentation ist leicht veraltet, jedoch sehr ausführlich und besonders in den unterschiedlichen Bereichen und der Api-Referenz sehr übersichtlich und hilfreich. Die Dokumentation ist in Version 5.4.1, aktuell ist jedoch Version 6 welche 2 Veröffentlichungen nach Version 5.4.1 ist.

2.2.5 Qualität des Projekts

Die interne Struktur von Protractor ist sehr übersichtlich und klar. Zusätzlich zum eigentlichen Werkzeug sind im Repository noch Beispiele und die Website enthalten. Durch statische Code-Analyse konnte auf den ca. 14.000 Zeilen Code 6 Bugs gefunden werden, welche bis auf einen potentiell falschen Rückgabewert vernachlässigt werden können. Negativ sind jedoch 102 „unschöne“ Stellen zu bewerten, bei denen es sich um viel Shadowing und mangelndes Aufräumen des Codes handelt. Jedoch ist alles noch im vertretbaren Rahmen.

[3] Stand 20.04.19

2.2.6 Aktivitäten, Bekanntheit und Unterstützung

Die letzte Veröffentlichung war im ersten Quartal 2019. Seitdem ist die Commit Rate relativ gering, was jedoch auch auf den Reifegrad des Werkzeuges zurückzuführen ist.

Der Bekanntheitsgrad von Protractor kann als hoch betrachtet werden. Dies ruht daher, dass Protractor standardmäßig in Angular-Anwendungen integriert ist. Weiter ist die Bekanntheit und Verbreitung an ca. 500 Beobachtungen, ca. 8.000 Sternen und ca. 2.000 Forks des Git-Repositories, sowie den ca. 3.500 Followern auf Twitter zu sehen.

Die Hauptunterstützer von Protractor sind Google sowie mehr als 250 beitragende Entwickler. Durch die Unterstützung von Google kann prognostiziert werden, dass Protractor noch längere Zeit existieren wird und entsprechend weiterentwickelt wird.

2.3 Nightwatch.js

Nightwatch.js ist ebenfalls ein E2E-Test-Werkzeug. Es stammt vom norwegischen Entwickler PineView Software. Neben der Test-Automatisierung legt Nightwatch.js Wert auf die Erweiterbarkeit und Konfigurierbarkeit für eigene Projekte. Außerdem können mit Nightwatch.js Unit-Tests geschrieben werden und mit Mochas Export Schnittstelle verbunden werden.

2.3.1 Lizenz

MIT

2.3.2 Reifegrad

Dies ist eher als mittel bis gering anzusehen. Das Werkzeug ist ab 2014 in der Version 0.2.8 verfügbar. Seitdem gab es über 150 Veröffentlichungen. Diese stammten allerdings immer von einer Person. Version 1 wurde im April 2018 veröffentlich und befindet sich somit ca. 1 Jahr außerhalb der Beta-Phase. Dennoch ist die aktuelle Version stabil und beinhaltet in der Issue-Liste 7 Bugs.

2.3.3 Support

Professionellen Support gibt es indirekt durch Pine View Software. Das Unternehmen bietet Consulting und Trainings für IT-Projekte an. Einen klassischen Maintenance-Vertrag gibt es jedoch nicht.

Nightwatch.js wird in erster Linie durch die Community supported[4]. Es gibt eine Google-Gruppe, welche ca. 1.500 Themen umfasst, die zeitnah beantwortet werden und StackOverflow mit ca. 1.000 Fragen, welche jedoch zu großen Teilen nicht beantwortet wurden.

2.3.4 Dokumentation

Die Dokumentation macht einen sehr guten Eindruck. Es finden sich Schnelleinstiege und Hilfestellungen wie das Werkzeug zu verwenden ist. Die Dokumentation ist auf den neuesten Stand. So entspricht diese zum Stand 22.04.19 bereits der Version 1.1.2, welche es zu diesem Zeitpunkt nur als Vor-Veröffentlichung gibt. Api-Referenzen sind ebenfalls sehr ausführlich und leicht verständlich.

[4] Stand 20.04.19

2.3.5 Qualität des Projekts

Der Nightwatch.js-Quellcode ist überschaubar und klar strukturiert. Die Testabdeckung ist mit 85% auf einem guten Niveau. Statische Code-Analyse deckt vier Bugs auf, welche sich jedoch nur auf zwei unnötige Strings und zweimal auf einen Objektvergleich mit einem Objekt, welches mit sich selbst verglichen wird beziehen. Diese Fehler sind jedoch nicht gravierend. Was außerdem positiv zu werten ist, dass bei den 27.000 Zeilen Code des Projekts nur 33 „unschöne“ stellen zu finden waren.

2.3.6 Aktivitäten, Bekanntheit und Unterstützung

Die Veröffentlichungs-Häufigkeit ist sehr hoch. So gab es in Q4 2018 und Q1 2019 zusammen 8 Veröffentlichungen. Passend dazu ist auch die Commit-Rate sehr hoch. Unschön ist hingegen, dass das Projekt zwar 74 beteiligte Entwickler besitzt jedoch nur einen, der ernsthaft entwickelt. Der Rest der beteiligten Entwickler machte in der Vergangenheit nur sehr kleine Änderungen.

Eine große Bekanntheit ist bei Nightwatch.js nicht festzustellen. So ist der erste Eindruck durch das Git-Repository mit 282 Beobachtungen, ca. 9.000 Sternen und 883 Forks sowie ca. 3.000 Twitter-Followern sehr positiv. In Berücksichtigung des Community-Supports auf StackOverflow und der Google-Trend-Analyse wie in Abbildung 4: Trendanalyse zu sehen, kann der Bekanntheitsgrad auf einen mittleren Bereich eingeschränkt werden.

Der Hauptunterstützer Pine View Software aus Oslo ist ebenfalls kaum bekannt, was zu einer Unsicherheit in der Lebensdauer von Nightwatch.js führt.

2.4 WebdriverIO

WebdriverIO ist eine weitere WebDriver API Implementierung, welche bereits im Namen verrät, dass es sich um eine solche handelt. WebdriverIO kann sowohl mit TypeScript als auch mit JavaScript verwendet werden und unterstützt die großen Test-Frameworks Jasmine, Cucumber und Mocha.

2.4.1 Lizenz

MIT

2.4.2 Reifegrad

Der Reifegrad von WebdriverIO ist als hoch anzusehen. Es wurde die Version 1 bereits 2013 veröffentlicht. Seitdem gab es ca. 200 Veröffentlichungen, was einen sehr kleinen Release-Zyklus kennzeichnet. Die aktuelle Version 5.7.16 ist eine stabile Version, die 8 Bugs in der Issue-Liste aufweist.

2.4.3 Support

Community-Support[5] ist in erster Linie durch einen aktiven Gitter-Chat mit ca. 4.000 Nutzer gegeben. Zusätzlich gibt es Support durch StackOverflow (ca. 800 Fragen).

2.4.4 Dokumentation

Die Dokumentation macht einen sehr guten Eindruck. Es finden sich Schnelleinstiege, Code-Beispiele und Hilfestellungen wie das Werkzeug zu verwenden ist. Zusätzlich werden noch

[5] Stand 20.04.19

Online-Kurse angeboten. Diese sind jedoch kostenpflichtig. Die Dokumentation bedient die aktuelle Version 5, jedoch kann die exakte Version nicht ermittelt werden.

2.4.5 Qualität des Projekts

Durch statische Code-Analyse wurden drei Bugs aufgeworfen, zwei hiervon können potenziell zu ernsthaften Fehlern führen. So gibt es eine *if*-Bedingung welche immer *true* ist und fehlende *null*-Überprüfungen.

Die restlichen 22.000 Zeilen Code sind mit 61 „unschönen“ Stellen sehr sauber und mit 98% Testabdeckung auch gut getestet.

Der Quellcode ist in sehr viele kleine Pakete unterteilt, nach kurzer Zeit sehr klar, sauber strukturiert und gleich aufgebaut.

2.4.6 Aktivitäten, Bekanntheitsgrad und Unterstützung

Die Veröffentlichungs-Häufigkeit ist sehr hoch. Mit mehreren Veröffentlichungen teilweise innerhalb eines Monats. Die Commit-Rate ist dazu passen ebenfalls hoch.

Die Bekanntheit von WebdriverIO kann als mittel bis hoch betrachtet werden. So kann dem Git-Repository entnommen werden, dass es 215 Beobachtungen, 4.800 Sterne und 1.370 Forks gibt.

WebdriverIO wird durch 116 beteiligte Entwickler unterstützt, wobei vier von diesen als Hauptentwickler bezeichnet werden können. Hinter WebdriverIO steht die JS.Foundation als starker Unterstützer, was auf eine längere zeitliche Unterstützung und dadurch lange Lebenserwartung schließen lässt.

2.5 Zusammenfassung/ Kurzübersicht des Schnelltests

KRITERIUM	PROTRACTOR	NIGHTWATCH.JS	WEBDRIVER.IO
LIZENZ	MIT	MIT	MIT
REIFEGRAD	Hoch	Mittel bis niedrig	Mittel bis hoch
STABILE-VERSION-VERÖFFENTLICHT	Ja	Ja	Ja
SUPPORT	Gut durch Community	Gemäßigt durch Community; Unsicher durch Pine View Software (kostenpflichtig)	Gemäßigt bis gut durch Community. Zusätzliche Tutorials gegen Gebühr
DOKUMENTATION	Gut	Gut	Sehr gut
QUALITÄT DES PROJEKTS	Gut	Gut	Gut

AKTIVITÄTEN	Ja, nach Veröffentlichung ruhig	Sehr viel Aktivität	Viel Aktivität
BEKANNTHEIT	Hoch	Gering bis mittel	Mittel bis hoch
UNTERSTÜTZUNG	Google	PineView Software	JS.Foundation

Abbildung 3: Kurzübersicht des Schnelltests

Zur Bekanntheit sollte noch Abbildung 4: Trendanalyse betrachtet werden. Dort ist zu sehen, wie die drei Werkzeuge im Google-Trend zueinander abschneiden. Als Kritik an der Grafik ist anzumerken, dass die Begriffe Protractor, Nightwatch und Webdriver nicht nur für die Werkezeuge benutzt werden. Somit ist die Abbildung nur als Indikator zu betrachten. Zusätzlich normiert sich die Trendanalyse immer auf 100 und zeigt keine aussagekräftige Skala. Weiterer Kritikpunkt an diesem Kapitel ist die Tatsache, dass Werte wie Veröffentlichungen, StackOverflow-Themen und beteiligte Entwickler-Zahlen nur Momentaufnahmen darstellen und leider keine Schnittstellen angeboten werden, um diese Zahlen für wissenschaftliche Zwecke zu erheben.

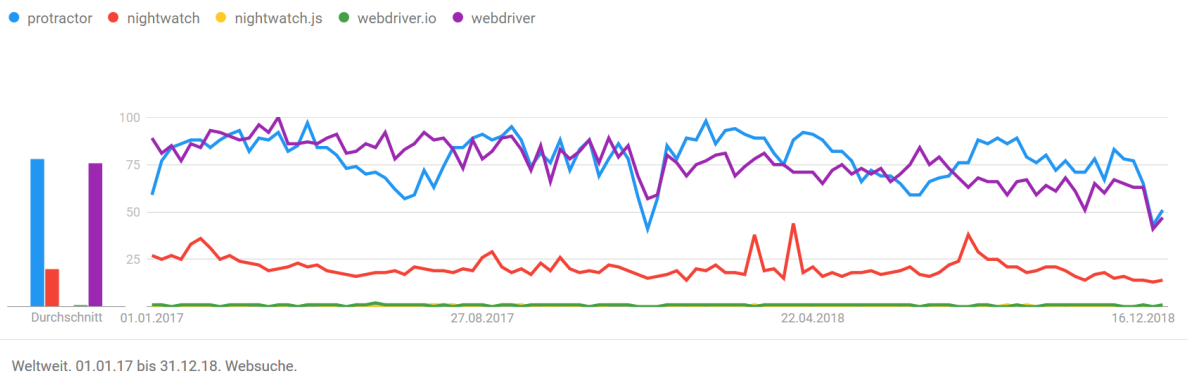

Abbildung 4: Trendanalyse

3 Vergleich von Protractor und Webdirver.IO

Nach der Schnellüberprüfung werden für den detaillierten Vergleich nur noch die zwei Werkzeuge Protractor und WebdriverIO betrachtet. Dies liegt daran, dass Nightwatch.js aufgrund von schlechten Support-Möglichkeiten, geringem Reifegrad und nur einem Hauptentwickler einen negativen Eindruck hinterlässt. Die Unterstützung von Pine View Software im Gegensatz zu Google und JS.Foundation ist ebenfalls als eher klein/gering anzusehen, was sich außerdem schlecht auf die Lebenserwartung bzw. Langlebigkeit des Werkzeugs auswirken könnte.

Verglichen wird in diesem Kapitel der Funktionsumfang in Form von gelösten Anwendungsfällen, der Aufwand für Integration und das Schreiben von Tests sowie die Einfachheit in Form von Code-Zeilen (Lines-of-Code = LoC) für die Tests und auch deren Lesbarkeit.

3.1 Implementierungsgrundlage und Anwendungsfälle

Als Implementierungsgrundlage der Werkzeuge dient eine einfache Angular-Anwendung. Diese besteht neben Boilerplate-Code, aus dem Testframework Jasmine, wenigen Komponenten sowie wenigen Modulen um Grundfunktionalität abbilden zu können. Es wurde bei der Anwendung kein Wert auf Design oder Feature-Umfang gelegt. Vielmehr sollen Anwendungsfälle abgedeckt werden, wie sie in Produktivsystemen gewöhnlich vorkommen.

Anwendungsfälle:

1. Navigation in einer Anwendung mit Nutzer Eingaben und Ergebnissen
2. Navigation auf asynchron geladenes Modul mit dynamisch geladenen Inhalten. Während des Ladens werden Animationen gezeigt.
3. Öffnen eines Dialogs mit dynamisch geladenen Inhalten. Diese öffnen weiteren Dialog mit identischen Element-Ids.
4. Neuen Browser-Tab öffnen, in diesen Eingaben vornehmen und auf den ursprünglichen Tab zurück wechseln.
5. Daten-Upload (kein Drag & Drop) mittels HTML-File-Input
6. Nutzung eines i-Frame (Webseite der Technischen Hochschule Rosenheim)

3.2 Protractor-Details

Protractor kann auch in nicht Angular-Anwendungen verwendet werden. Dafür muss das Warten auf Angular-Ereignisse mittels *browser.waitForAngularEnabled(false)* deaktiviert werden.

3.2.1 Browserunterstützung

Browserunterstützung findet für alle „großen“ Browser statt. Somit werden Chrome, Firefox und Safari Out-Of-The-Box unterstützt. Für Microsoft Edge und Internet Explorer muss der Microsoft Web Driver installiert werden.

Des Weiteren wird von PhantomJS / GhostDriver abgeraten, da diese im Zusammenspiel mit Protractor Fehler verursachen können und sich zu sehr von echten Browsern unterscheiden. Dies bedeutet allerdings nicht, dass man auf oberflächenlose Browser verzichten muss. Einige Browser wie Chrome und Firefox bieten einen oberflächenlosen Modus, wodurch Werkzeuge wie PhantomJS abgelöst werden können. Zusätzlich hat der oberflächenlose Modus den Vorteil, dass der Ziel-Browser in einer CI/CD-Pipeline auf einfache Weise getestet werden kann.

Neben klassischen Web-Browsern werden mobile Browser durch Appium unterstützt.

3.2.2 Page-Object-Pattern

Protractor ermöglicht die einfache Nutzung das Page-Object-Patterns. Hierbei wird eine Objekt-Repräsentation der zu testenden Seite erzeugt. Dadurch kann auf die Elemente der Seite, wie aus der objektorientierten Programmierung bekannt und auf Attribute, zugegriffen werden. Hierfür wird den Objekten wie in Abbildung 5: Protractor Page-Object-Pattern zu sehen dem Attribut ein Element-Finder zugewiesen, welcher bei Zugriff das Element auf der Seite sucht bzw. findet. Des Weiteren können komplexere Funktionen hinterlegt werden, um z.B. Elemente zu finden, welche nicht eindeutig identifiziert werden können oder nur den Text eines Elements statt das Element selbst zurückzuliefern.

Das Page-Object-Pattern dient außerdem dazu bei Seitenänderungen einfacher die Tests anzupassen. So wird beim Austausch eines Elements nach Möglichkeit nicht der Test angepasst, sondern der Element-Finder.

```
export class SomePage  extends AppPage {
  public dialogOpenButton: ElementFinder=  element(by.id('openDialogButton'));
  public iFrameButton: ElementFinder = element(by.partialButtonText('iFr'));
  public intranetButton: ElementFinder = element(by.partialLinkText('ntrane'));
  public iFrame: ElementFinder = element(by.id('thRosenheim'));
  public subTitleFromIntranet: ElementFinder =
element(by.xpath('//*[@id="c55958"]/div/h2'));

  getRow(index?: number): ElementFinder {
    return element(by.id('customItem' + index || '0'));
  }

  getSecondSpinner(): ElementFinder {
    return element.all(by.id('spinner')).get(1);
  }

  getSecondListRowText(index: number): promise.Promise<string> {
    return element.all(by.id('customItem' + index)).get(1).getText();
  }
}
```

Abbildung 5: Protractor Page-Object-Pattern

3.2.3 Zugriffsmöglichkeiten

Wie in Abbildung 5: Protractor Page-Object-Pattern zu sehen bedient sich das Page-Object der *element*-Funktion um Elementfinder zu erzeugen. Diese Elementfinder können nach unterschiedlichen Attributen der Elemente suchen. Hierfür bietet Protractor viele Lokalisatoren/Selektoren (Locators/Selectors) an. Dazu gehören die durch die Webdriver-Api geerbten Lokalisatoren/Selektoren:

- className
- css
- id
- linkText
- js
- name
- tagName
- partialLinkText
- xpath

Zusätzlich gibt es welche die, teilweise nur auf Angular bezogen sind, z.B.:

- binding
- exactBinding
- model
- buttonText
- partialButtonText
- repeater

- exactRepeater
- cssConainigText
- options
- deepCss

Diese Lokalisatoren werden mittels der *by*-Klasse erzeugt und ermöglichen es so, sehr leserlich den gewünschten Lokalisator der *element*-Funktion zu übergeben.

3.2.4 Bedingungen – Warten auf Ereignisse

Als grundlegendes Konzept, um mit E2E-Tests beginnen zu können müssen noch die Bedingungen bzw. das Warten auf Ereignisse verstanden werden. Auf Grund von Wartezeiten, z.B. durch Netzwerkkommunikation oder Rendern (Visualisieren) müssen Tests gelegentlich warten. Dies kann naiv durch die *sleep/wait*-Funktion mit einer Zeitangabe des Browsers gemacht werden oder durch das Warten auf Bedingungen. Für das Erstellen und Warten auf Bedingungen bietet Protractor sehr viele Möglichkeiten einige davon sind in Abbildung 6: Protractor - Ausschnitt 1 - Logische Operatoren zu sehen. Die gesamte Liste umfasst:

- alertIsPresent
- elementToBeClickable
- textToBePresentInElement
- textToBePresentInElementValue
- titleContains
- titleIs
- urlContains
- urlIs
- presenceOf
- stalenessOf
- visibilityOf
- invisibilityOf
- elementToBeSelected

Hierdurch können bestimmte Zustände sehr granular abgepasst werden und auf diese reagiert werden. So ist der Unterschied für viele Anwendungen zwischen elementToBeClickable, visibilityOf und presenceOf nicht wirklich gegeben. Sollte der Fall jedoch auftreten, dass unterschieden werden muss, bietet Protractor diese Möglichkeit.

Zusätzlich können diese Bedingungen noch mit den logischen Operatoren NOT, AND und OR in Form von Funktionen verknüpft werden. Dies ist ebenfalls in Abbildung 6: Protractor - Ausschnitt 1 - Logische Operatoren unter Alternative 1 zu sehen. Für die Leserlichkeit wird hier jedoch Alternative 2 empfohlen.

```
it('Should wait for Data', () => {
  page.navigateTo();
  page.lazyModuleButton.click();

  // Alternative 1
  browser.wait(conditions.and(
    conditions.visibilityOf(page.title),
    conditions.invisibilityOf(page.spinner),
    conditions.visibilityOf(page.list))
    , 5000);

  // Alternative 2
  browser.wait(conditions.visibilityOf(page.title), 1000);
  browser.wait(conditions.invisibilityOf(page.spinner), 5000);
  browser.wait(conditions.visibilityOf(page.list), 1000);

  expect(page.getRowText(0))
                .toBe('Item Nr 1ein total nutzloses Objekt42');
  expect(page.getRowText(1))
                .toBe('Item Nr 2ein total nutzloses Objekt3');
  expect(page.getRowText(2))
                .toBe('Item Nr 3ein total nutzloses Teil17');
});
```

Abbildung 6: Protractor - Ausschnitt 1 - Logische Operatoren

3.2.5 Ergebnisse der Implementierung

Bei der Verwendung mit Angular ist darauf hinzuweisen, dass Protractor komplett mitgeliefert wird, wodurch sich alle Konfigurationen und Einstellungen als sehr einfach und schnell erweisen. Dennoch ist es möglich, eigene Strukturen zu konfigurieren. Leider stößt man bei neuen Angular-Projekten, welche mit der Angular-CLI in Version 7.1.4 erzeugt werden, auf ein Problem. Durch Angular-CLI kann Protractor nicht gestartet werden, wenn die Bibliothek „@angular-devkit/build-angular" nicht aktualisiert wird. Sollte Protractor nicht durch die Angular-CLI gestartet werden, fällt dieser Umstand nicht auf.

Wie erwähnt, kann in einem Angular-Projekt Protractor über die Angular-CLI gestartet werden, was sich für Build-Prozesse sehr gut eignet. Direktes starten ist ebenfalls möglich, was sich gerade während der Entwicklung durch das schnelle Starten von Tests ohne Build-Prozess als vorteilhaft erweisen kann.

Die Konfigurations-Datei von Protractor beträgt ca. 30 Zeilen und ist auf den ersten Blick übersichtlich und leicht verständlich. In ihr werden in einfachen Fällen neben Testdateien, Timeouts, Test-Framework, Browser und zu testender URL nichts weiter beschrieben. Was alles durch die Konfigurations-Datei eingestellt werden kann, findet sich in der Dokumentation sehr gut beschrieben.

Der Testaufbau ist abhängig vom Test-Framework. In Abbildung 7: Jasmine-Testaufbau ist der Aufbau zu sehen, wie er durch Jasmine vorgeschrieben ist. So wird der Test mit der Beschreibung der Test-Datei begonnen, gefolgt von den variablen Deklarationen. Es gibt in jedem Test eine *beforeEach*-Funktion, welche wie der Name sagt vor jedem Test ausgeführt

wird und dazu genutzt wird, neue Objekte zu erzeugen und unabhängige Tests zu ermöglichen. Nach jedem Test wird eine *afterEach*-Funktion ausgeführt, welche die Tests bereinigen kann. Die Tests selbst werden entsprechend mit *it*, gefolgt von der „Should ..“ und der Test-Beschreibung eingeleitet.

```
describe('Suite Description', () => {
  let page: SomePageObject;
  const conditions: ProtractorExpectedConditions =
protractor.ExpectedConditions;

  beforeEach(() => {
    page = new SomePageObject();
  });

  afterEach(() => {
  ...
  });

  it('Should ...', () => {
  ...
  });
}
```

Abbildung 7: Jasmine-Testaufbau

Betrachtet man die ausprogrammierten Anwendungsfälle, so stellt man wie in Abbildung 8: Protractor - Ausschnitt 2 - Anwendungsfall 1 fest, dass der Code durch das Page-Object-Pattern fast schon für Nicht-Informatiker leserlich ist. Die kompletten Anwendungsfälle sind unter https://github.com/dominikampletzer/web-application zu finden.

```
it('Should insert Data into Form', () => {
  page.navigateTo();
  page.formButton.click();

  page.formName.sendKeys('Ampletzer');
  page.formFirstName.sendKeys('Dominik');

  page.formSex.click();
  const sexOption = element(by.id('sexOption1'));
  browser.wait(conditions.elementToBeClickable(sexOption), 1000);
  sexOption.click();

  page.saveButton.click();

  browser.wait(conditions.visibilityOf(page.sentName), 5000);
  expect(page.sentName.getText()).toBe('Ampletzer');
  expect(page.sentFirstName.getText()).toBe('Dominik');
  expect(page.sentSex.getText()).toBe('Männlich');
});
```

Abbildung 8: Protractor - Ausschnitt 2 - Anwendungsfall 1

3.2.6 Weitere Erkenntnisse

Es gab bei keinem der sechs Anwendungsfälle Schwierigkeiten, die nicht gelöst werden konnten.

Erwähnenswerte Erkenntnisse:

- Warten auf asynchrone Module und dynamsiches Laden von Inhalten funktioniert
- Elemente mit gleicher ID übereinanderliegend können korrekt gefunden und betätigt werden
- Tab-Wechsel funktioniert sehr gut; es muss berücksichtigt werden, dass der Kontext gewechselt werden muss
- Upload von Dateien kann stattfinden, allerdings nur mittels Pfadübergabe als String und nicht durch Betriebssystem eigenes Fenster oder Drag and Drop
- iFrame kann genutzt werden, es muss allerdings auf Kontext geachtet werden und Warten auf Angular muss deaktiviert werden (siehe *Kapitel Protractor-Details)*

Integration, Konfigurierung und Implementierung der Tests haben ca. 6 Stunden in Anspruch genommen.

3.3 WebdriverIO Details

3.3.1 Browserunterstützung

WebdriverIO unterstützt die Browser Firefox, Chrome, Opera, Safari, Microsoft Internet Explorer und Edge sowie mobile Browser durch Appium. In dieser Hinsicht unterscheidet sich WebdriverIO nicht von Protractor. Allerdings findet man in der WebdriverIO-Dokumentation nur indirekt, welche Browser unterstützt werden. So gibt es keinen Bereich „Unterstützte Browser“. Die Browserunterstützung muss aus den Optionen der Konfiguration und diversen Blog-Beiträgen erarbeitet werden.

3.3.2 Page-Object-Pattern

WebdriverIO setzt seit Version 5 ebenfalls auf das Page-Object-Pattern. Die Integration ist ebenfalls sehr einfach. WebdriverIO gibt in den Beispielen des Pattern den Zugriff auf die Elementfinder mittels der in Abbildung 9: WebdriverIO - Page-Object-Pattern zu sehenden *get*-Notation an, was jedoch als Äquivalent zu der Attributsinitialisierung im Falle des Protractor-Beispiels zu betrachten ist. Dies scheint allerdings der einzige Unterschied in der Implementierung des Page-Object-Pattern zu sein.

```
const AppPage = require('../app.po');

class SomePage extends AppPage {
  get iFrameButton() {
    return $("#iFrame");
  }

  get intranetButton() {
    return $('*=ntrane');
  }

  get iFrame() {
    return $('#thRosenheim');
  }

  get subTitleFromIntranet() {
    return $('//*[@id="c55958"]/div/h2');
  }

  getSecondTitle() {
    return $$('#dialogTitle')[1];
  }

}

module.exports = UseCase6Po;
```

Abbildung 9: WebdriverIO - Page-Object-Pattern

3.3.3 Zugriffsmöglichkeiten

Elementfinder werden wie in Abbildung 9: WebdriverIO - Page-Object-Pattern zu sehen, mittels der *$*-Funktion bzw. für mehrere erwartete Elemente mit der *$$*-Funktion erzeugt. Um zu entscheiden, welches Attribut überprüft werden soll, um ein Element zu finden, übergibt WebdriverIO der *$*-Funktion ebenfalls einen Parameter. Bei diesem Parameter handelt sich um einen Lokalisator, welcher mittels Kurzschreibweise, meist ein Sonderzeichen erzeugt. Diese sind:

- CSS-Query = $(„h2“)
- Link-Text = $(„=someString“)
- Partial-Link-Text = $(„*=someString“)
- CSS-Klasse = $(„.someString“)
- Id = $(„#someString“)
- Tag = $(„<my-tag/>“)
- xPath = $(„x-PATH as String“)

Die Lokalisatoren können kombiniert werden, um Elemente zu finden, welche mehrere Kriterien erfüllen sollen, z.B. CSS-Query mit Text = $(„h1=SomeText“).

Zusätzlich gibt es noch die Möglichkeit, für Mobile-Anwendungen spezielle Elementfinder zu verwenden, sowie die Möglichkeit mittels JavaScript-Funktionen Elemente zu finden.

3.3.4 Bedingungen – Warten auf Ereignisse

WebdriverIO benötigt ebenfalls Bedingungen bzw. muss auf Ereignisse warten. Diese Bedingungen sind in die Elemente bzw. in die Elementfinder integriert, was es sehr einfach macht, diese zu verwenden. So sieht man in Abbildung 10: WebdriverIO - Ausschnitt, dass die *waitForDisplayed*-Funktion direkt aus dem Elementfinder aufgerufen werden kann. WebdriverIO bietet neben der naiven *wait/sleep*-Variante mit einer Zeitangabe vier weitere Möglichkeiten:

- waitForDisplayed
- waitForEnabled
- waitForExist
- waitUntil

Die meist kurzen Schreibweisen von WebdriverIO kommen auch in den Bedingungen zu tragen, so wurde z.B. auf eine *waitForInvisible* verzichtet. Stattdessen wurde die *waitForDisplayed* mit einem Reverse-Parameter versehen, welcher für die Unsichtbarkeit des Elements verwendet wird.

Die vier Möglichkeiten sind bis auf *waitUntil* absolut klar. *WaitUntil* nimmt als Parameter eine benutzerdefinierte Funktion und wartet bis die Bedingung dieser erfüllt ist.

```
it('Should insert Data into Form', () => {
  page.navigateTo();
  page.formButton.click();

  page.formName.setValue('Ampletzer');
  page.formFirstName.setValue('Dominik');

  page.formSex.click();
  let sexOption = $('#sexOption1');
  sexOption.waitForDisplayed(1000);
  sexOption.click();

  page.saveButton.click();
  page.sentName.waitForDisplayed(5000);
  assert.equal(page.sentName.getText(), 'Ampletzer');
  assert.equal(page.sentFirstName.getText(), 'Dominik');
  assert.equal(page.sentSex.getText(), 'Männlich');
});
```

Abbildung 10: WebdriverIO - Ausschnitt

3.3.5 Ergebnisse der Implementierung

Einführend ist zu erwähnen, dass das Aufsetzen von WebdriverIO sich mühsam gestaltet, selbst wenn man sich an das bereitgestellte Tutorial hält. In diesem wird der Einfachheit halber auf

Dienste, welche durch WebdriverIO mit installiert werden können, verzichtet. Leider ist gerade der Silenium-Dienst unverzichtbar, um die Tests ohne größeren Aufwand auf einem Gerät durchführen zu können.

Die Konfigurationsdatei von WebdriverIO ist sehr übersichtlich und ebenfalls leicht verständlich. Was Anfangs leicht unübersichtlich wirkt, allerdings nach kurzem sehr hilfreich ist, ist die Code-Dokumentation, welche sich in der Konfigurationsdatei befindet. So ist die Datei ohne Kommentare nur ca. 30 Zeilen lang, mit Kommentaren jedoch ca. 250, was es einfach macht, herauszufinden, welche Einstellungsmöglichkeiten gemacht werden können, ohne in einer Dokumentation nachschlagen zu müssen.

Der Testaufbau selbst ist wie bereits im Kapitel 3.2.5 beschrieben, durch Jasmine bestimmt und wird an dieser Stelle nicht wiederholt.

Das Betrachten der Abbildung 10: WebdriverIO - Ausschnitt zeigt wie klar leserlich mit WebdriverIO und dem Page-Object-Pattern entwickelt werden kann. Allerdings können die sehr kurz gehaltenen *$*-Funktion und die Kurzschreibweise für die Lokalisatoren zu Unklarheit führen.

Die Test selbst sind in der Anwendung unter https://github.com/dominikampletzer/web-application zu finden.

3.3.6 Weitere Erkenntnisse

- Warten auf asynchrone Module und dynamisches Laden von Inhalten funktioniert
- Elemente mit gleicher ID übereinanderliegend können korrekt gefunden und betätigt werden
- Upload von Dateien kann stattfinden, allerdings nur mittels Pfadübergabe als String und nicht durch Betriebssystem eigenes Fenster oder Drag and Drop
- iFrame kann genutzt werden, es muss allerdings auf Kontext geachtet werden

Anwendungsfall 4 konnte nicht gelöst werden. Dieser umfasst folgende Schritte:

- einen neuen Tab öffnen
- in diesem ein Formular ausfüllen und absenden
- den Tab schließen
- zurückkehren auf den ursprünglichen Tab

Tests brechen immer wieder ab, obwohl die *switchWindow*-Funktion, welche den Tab-Wechsel laut Dokumentation ermöglicht, verwendet wird. Wechselt man auf die switchToWindow-Funktion aus Version 4, funktioniert der Tab-Wechsel, allerdings bricht der Test beim Schließen und Zurückkehren des Tabs ab. Dieses Problem konnte nach 1,5 Stunden nicht behoben werden und wird auf Grund dessen als nicht gelöst gewertet. Für den Tab-Wechsel, gibt es Beispiele in der WebdriverIO-Dokumentation, welche funktionieren. Somit kann nicht generell gesagt werden, ob es am Anwendungsfall, Unwissenheit des Entwicklers oder dem Framework lag.

Als persönliche Anmerkung muss noch erwähnt werden, dass das Konfigurieren und nur das Adaptieren der Tests (diese wurden bereits durch die Protractor-Tests durchdacht und

entwickelt; es mussten fast nur die Selektoren und Bedingungen angepasst werden) sehr viel Zeit in Anspruch genommen hat. So wurden ca. 8-9 Stunden benötigt, ohne dass alle Anwendungsfälle gelöst werden konnten.

3.4 Direkter Vergleich

Als Vergleichskriterien für den direkten Vergleich werden

- **Benötigte Codemenge** (Lines-of-Code = LoC), um die Komplexität des Codes darzustellen
- **Benötigte Zeit** für die Implementierung, um die Komplexität und „Schwierigkeit" der Implementierung auszudrücken
- **Anzahl der gelösten Anwendungsfälle** ebenfalls ein Kriterium für „Schwierigkeit" der Implementierung und um den Funktionsumfang darzustellen

verwendet.

	PROTRACTOR	**WEBDRIVERIO**
LOC	458	494
BENÖTIGTE ZEIT	5-6h	8-9h
GELÖSTE ANWENDUNGSFÄLLE	6	5
ANMERKUNGEN	Für Nicht-Angular-Anwendungen muss *waitForAngular* deaktiviert werden	Wechseln des Fensters/Tabs konnte mit akzeptablen Aufwand nicht erreicht werden

Abbildung 11: Protractor vs. WebdriverIO

Der LoC-Uunterschied von ca. 10% kommt in erster Linie von der unterschiedlichen Page-Object-Pattern-Implementierung. Betrachtet man die Tests selbst, stellt man kaum einen Unterschied in der Länge fest.

Der gewaltige Zeitunterschied ist zu großen Teilen der komplexen kürzeren Schreibweise für Elementfinder und Lokalisatoren geschuldet, sowie den anfänglichen Schwierigkeiten bei der Konfiguration von WebdriverIO.

4 Fazit

Aus dieser Arbeit ergibt sich, dass Nightwatch.js das am wenigsten zu empfehlende Werkzeug ist. Dies liegt daran, dass Nightwatch.js nur einen Haupt-Entwickler besitzt, sowie daran, dass es den schlechtesten Reifegrad, Community-Support und die ungewisseste Untersützung durch Pine View Software aufweist.

WebdriverIO und Protractor ermöglichen beide auf einfache Weise das Nutzen des Page-Object-Pattern, welches die Leserlichkeit und die Wartung erleichtert. Beide Werkzeuge

ermöglichen das Testen mittels JavaScript als auch mit TypeScript. Außerdem nutzen beide Werkzeuge „Bedingungen“ um auf Zustände des Browsers zu warten. Hierdurch kann Asynchronität und dynamisches Laden sehr gut abgebildet werden.

WebdriverIO ist ein sehr gutes Werkzeug, welches kleinere Schwächen aufweist. So gab es in der Testimplementierung das Problem, dass der Anwendungfall, welcher einen Tab-Wechsel einschließt, nicht lauffähig gemacht werden konnte, obwohl dies beim WebdriverIO-Testbeispiel funktioniert. WebdriverIO bietet außerdem weniger Möglichkeiten Elemente zu finden als Protractor, allerdings können Lokalisatoren einfach kombiniert werden. Was nicht unbedingt als Vorteil bezeichnet werden kann, ist die durchgehend spürbar knappe Schreibweise. So ist diese natürlich kurz und kompakt, dies geht aber auf Kosten der Lesbarkeit, da ganz explizites Wissen benötigt wird, welche Funktionen und Elementfinder benutzt werden sollen, oder welche Bedingung negiert werden muss, um eine andere Bedingung zu erhalten.

Abschließend kann gesagt werden, dass für Angular-Anwendungen im Speziellen als auch im Allgemeinen Protractor empfohlen werden kann. Dieses Werkzeug bietet einen hohen Reifegrad, starke Unterstützung durch die Community sowie durch Google. Es konnte im Test sowohl durch die geringe Einarbeitung als auch durch das Lösen aller Anwendungsfälle überzeugen. Als großer Vorteil wird sowohl die große Anzahl an Bedingungen und Lokalisatoren betrachtet, wie auch die gute Lesbarkeit.

Literaturverzeichnis

Gitter. (2019). *Gitter - Homepage.* www.gitter.im - gefunden 22.04.2019.

Google. (2019). *Google Gruppen - Homepage.* https://groups.google.com - gefunden 20.04.2019.

Nightwatch.js. (2019). *Nightwatch.js - Homepage.* www.nightwatchjs.org - gefunden 21.04.2019.

Protractor. (2019). *Protractor - Homepage.* www.protractortest.org - gefunden 20.04.2019.

StackOverflow. (2019). *Stack Overflow - Homepage.* https://stackoverflow.com - gefunden 20.04.2019.

Webdriver.IO. (2019). *WebdriverIO - Homepage.* www.webdriver.io - gefunden 20.04.2019.