



**Hochschule  
Augsburg** University of  
Applied Sciences

**Fakultät für  
Informatik**

## **Bachelorarbeit**

Bachelorstudiengang  
Wirtschaftsinformatik

**Dominic Schmitz**  
**Cypress, das neue Selenium?**

Prüfer: Prof. Dr. Juergen Scholz  
Abgabe der Arbeit: 20.04.2018

Name:  
Dominic Schmitz

Matrikelnummer:  
944833

Anschrift:  
Wilhelm-Hauff Straße  
22, 86161 Augsburg

Email:  
Dominic.Schmitz@HS-  
Augsburg.de

Hochschule für angewandte  
Wissenschaften Augsburg  
University of Applied Sciences

An der Hochschule 1  
D-86161 Augsburg

Telefon +49 821 55 86-0  
Fax +49 821 55 86-3222  
[www.hs-augsburg.de](http://www.hs-augsburg.de)  
[info@hs-augsburg.de](mailto:info@hs-augsburg.de)

# Erklärung zur Abschlussarbeit

---

Hiermit versichere ich, die eingereichte Abschlussarbeit selbständig verfasst und keine andere als die von mir angegebenen Quellen und Hilfsmittel benutzt zu haben. Wörtlich oder inhaltlich verwendete Quellen wurden entsprechend den anerkannten Regeln wissenschaftlichen Arbeitens zitiert. Ich erkläre weiterhin, dass die vorliegende Arbeit noch nicht anderweitig als Abschlussarbeit eingereicht wurde.

Das Merkblatt zum Täuschungsverbot im Prüfungsverfahren der Hochschule Augsburg habe ich gelesen und zur Kenntnis genommen. Ich versichere, dass die von mir abgegebene Arbeit keinerlei Plagiate, Texte oder Bilder umfasst, die durch von mir beauftragte Dritte erstellt wurden.

---

Ort, Datum

---

Unterschrift des/der Studierenden

# Danksagung

---

An dieser Stelle möchte ich mich bei meiner Betreuerin Jana Knopp für das Korrekturlesen und der ständigen Unterstützung bedanken.

Ebenso gilt mein Dank Herrn Prof. Dr. Jürgen Scholz für die vielen nützlichen Tipps und Anmerkungen, die einen wesentlichen Teil zu dieser Bachelorarbeit beigetragen haben.

Weiterhin danke ich noch dem Unternehmen glomex GmbH für die Bereitstellung der benötigten Hard- und Software.

# Kurzfassung

---

Die meisten Menschen, die sich schon einmal mit der Testautomatisierung von Weboberflächen beschäftigt haben, haben bereits von Selenium gehört. Seit dem Jahre 2015 scheint es eine Alternative namens Cypress zu geben. Das Unternehmen glomex GmbH versucht herauszufinden, ob diese Alternative, das momentan eingesetzte System ablösen kann. Momentan wird ein Verbund aus Selenium, Pytest und Python zur Testautomatisierung eingesetzt. Um beide Systeme miteinander zu vergleichen, wurden die für das Unternehmen wichtigsten Kriterien Installation, Abhängigkeiten, Ausführung während der Testerstellung, Kompatibilität und iFrame Support aufgestellt. Jedes System wurde auf die genannten Kriterien hin geprüft und mit einer Punktzahl zwischen 1 (ungeeignet) bis 3 (gut geeignet) versehen. Abschließend wurde eine Gesamtpunktzahl je System ermittelt, um herauszufinden, welches sich besser eignet. Dabei hat sich herausgestellt, dass sich Selenium zum momentanen Zeitpunkt (Stand 07.03.2018) besser für den Einsatz bei der glomex GmbH eignet. Während dem Vergleich wurde deutlich, dass Cypress besonders bei den zwei wichtigsten Kriterien, Kompatibilität und dem iFrame Support nachholen muss. Cypress arbeitet bereits an diesen Punkten und könnte so zukünftig zu einem sehr guten Alternativsystem werden.

# Inhaltsverzeichnis

---

|   |             |
|---|-------------|
| <b>Abbildungsverzeichnis</b>                      | <b>VI</b>   |
| <b>Tabellenverzeichnis</b>                        | <b>VII</b>  |
| <b>Abkürzungsverzeichnis</b>                      | <b>VIII</b> |
| <b>Glossar</b>                                    | <b>IX</b>   |
| <b>1 Einleitung</b>                               | <b>1</b>    |
| 1.1 Motivation . . . . .                          | 1           |
| 1.2 Zielsetzung . . . . .                         | 1           |
| 1.3 Das Unternehmen glomex GmbH . . . . .         | 2           |
| <b>2 Stand der Technik</b>                        | <b>3</b>    |
| 2.1 Bedeutung-Qualitätsmanagement . . . . .       | 3           |
| 2.1.1 Qualitätsmanagementsysteme . . . . .        | 3           |
| 2.1.2 Qualität . . . . .                          | 3           |
| 2.2 QA-Abteilung . . . . .                        | 4           |
| 2.3 Testpyramide . . . . .                        | 4           |
| 2.4 Test-Arten . . . . .                          | 5           |
| 2.4.1 Unittests . . . . .                         | 5           |
| 2.4.2 Integrationstests . . . . .                 | 5           |
| 2.4.3 Oberflächentests (GUI-Tests) . . . . .      | 5           |
| 2.5 Selenium . . . . .                            | 6           |
| 2.5.1 Behavior Driven Development (BDD) . . . . . | 7           |
| 2.5.2 Pytest . . . . .                            | 8           |
| 2.6 Cypress . . . . .                             | 11          |
| <b>3 Installation / Konfiguration</b>             | <b>12</b>   |
| 3.1 Umgebung . . . . .                            | 12          |
| 3.2 Cypress Test Runner . . . . .                 | 13          |
| 3.2.1 Installation unter Mac OS . . . . .         | 13          |
| 3.2.2 Konfiguration . . . . .                     | 14          |
| 3.2.3 Update . . . . .                            | 14          |
| 3.3 Visual Studio Code . . . . .                  | 15          |
| <b>4 Cypress Einführung</b>                       | <b>16</b>   |
| 4.1 Projektstruktur . . . . .                     | 16          |

|          |   |           |
|----------|---|-----------|
| 4.2      | Blockstrukturen . . . . .               | 18        |
| 4.3      | Wichtige Kommandos . . . . .            | 21        |
| <b>5</b> | <b>Automatisierung der Ausführung</b>   | <b>24</b> |
| 5.1      | Testimplementierung in Travis . . . . . | 24        |
| 5.1.1    | Travis . . . . .                        | 24        |
| 5.1.2    | Git / Github . . . . .                  | 24        |
| 5.1.3    | Testimplementierung . . . . .           | 25        |
| 5.2      | Cypress Dashboard . . . . .             | 28        |
| <b>6</b> | <b>Vergleich</b>                        | <b>31</b> |
| 6.1      | Momentanes System . . . . .             | 31        |
| 6.2      | Konzept Vergleich . . . . .             | 32        |
| 6.2.1    | Auswertung . . . . .                    | 41        |
| <b>7</b> | <b>Fazit</b>                            | <b>43</b> |
| 7.1      | Reflexion . . . . .                     | 43        |
| 7.2      | Ausblick . . . . .                      | 44        |
|          | <b>Literatur</b>                        | <b>1</b>  |

# Abbildungsverzeichnis

---

|     |   |    |
|-----|---|----|
| 1.1 | glomex Webseite (Quelle: [ <b>glomex</b> ]) . . . . .                                 | 2  |
| 2.1 | Testpyramide nach (Quelle: [ <b>frueheTesterfaengtdenWurm</b> ]) . . . . .            | 4  |
| 2.2 | End-to-End (Beispiel Registrierungsprozess) . . . . .                                 | 6  |
| 2.3 | Pytest, Vergleich (Assert) ergibt True . . . . .                                      | 10 |
| 2.4 | Pytest, Vergleich (Assert) ergibt False . . . . .                                     | 10 |
| 2.5 | Cypress Test Runner während einer Testausführung . . . . .                            | 11 |
| 3.1 | Cypress Installationsprozess (Mac OS) . . . . .                                       | 13 |
| 3.2 | Cypress Update . . . . .  | 15 |
| 4.1 | Cypress Projektstruktur . . . . .   | 16 |
| 4.2 | Cypress Test Struktur . . . . .   | 19 |
| 5.1 | Travis Testausführung der letzten sechs Tage (Cypress Tests) . . . . .                | 27 |
| 5.2 | Travis Log nach Abschluss eines Cypress Tests . . . . .                               | 27 |
| 5.3 | Dashboard Login (Quelle: [ <b>cypressDashboardLogin</b> ]) . . . . .                  | 28 |
| 5.4 | Project ID (aus Sicherheitsgründen hier nur fiktiv) . . . . .                         | 28 |
| 5.5 | Testausführungen im Projekt cypress_glomex . . . . .                                  | 29 |
| 5.6 | Testdetails . . . . .   | 29 |
| 6.1 | Test Umgebung . . . . .   | 31 |
| 6.2 | Cypress Test Runner während der Ausführung des Tests „test_player.js“ . . . . .       | 35 |
| 6.3 | Testausführung des Tests „test_player.js“ im Browser Google Chrome . . . . .          | 35 |
| 6.4 | erster_test.js, Testausführung im Browser mithilfe des Cypress Test Runners . . . . . | 39 |
| 6.5 | Unterstützte Browser im Cypress Test Runner (Stand 27.02.2018 V2.0.4) . . . . .       | 40 |

# Tabellenverzeichnis

---

6.1 Framework Auswertung anhand Kriterien . . . . . 42



# Abkürzungsverzeichnis

---

**BDD** Behavior Driven Development

**GUI** Graphical User Interface

**HTML** Hypertext Markup Language

**IDE** Integrated Development Environment

**QA** Quality Assurance

**QM** Qualitätsmanagement



# 1 Einleitung

---

## 1.1 Motivation

Je umfangreicher und komplexer Software-Systeme sind, desto fehleranfälliger werden diese. Jedes softwareproduzierende Unternehmen bemüht sich daher, eine entsprechend hohe Qualität zu gewährleisten, damit der Kunde letztendlich kein fehlerhaftes Produkt erhält. Aus diesem Grund ist es essenziell, Software auf ihre Funktionalität hin zu testen.

Die vielen Abhängigkeiten innerhalb moderner Software-Systeme machen es notwendig nach jeder Änderung am Quellcode, die betroffenen Bereiche erneut zu testen. Das wiederum führt zu einem enormen Aufwand, da häufig das komplette Produkt wiederholt getestet werden muss [**softwarequalitaet**]. Durch die Automatisierung von Tests versprechen sich viele Unternehmen eine Zeitersparnis des Aufwandes. Daher wird meistens eine Kombination aus bestimmten Frameworks verwendet, die in Verbindung ganze Webanwendungen testen können und in der Lage sind, alle Funktionalitäten und Abhängigkeiten einer Software zu testen.

## 1.2 Zielsetzung

Ziel dieser Arbeit ist die Evaluation eines neuen Frameworks namens „Cypress“ (auch Cypress.io genannt), um den Testaufwand bei der glomex GmbH zu verringern. Mithilfe von Cypress ist es möglich Oberflächen von Webanwendungen (graphisch) im Browser zu testen. Bisher wurde hierfür die Programmiersprache Python in Verbindung mit Selenium und Pytest verwendet. Da alle Produkte der glomex beim Kunden zum Einsatz kommen, sind während der Produktentwicklung andere Browserplattformen zu behandeln als im Unternehmen selbst. Es sollen alle wichtigen Unterschiede und eventuelle positive sowie negative Aspekte von Cypress gegenüber dem momentanen System herausgearbeitet und aufgezeigt werden. Des Weiteren wird ermittelt, ob und wie sich die neuen Tests automatisiert ausführen lassen und inwiefern Cypress Lösungen in diesem Bereich bereitstellt.

## 1.3 Das Unternehmen glomex GmbH

Das Unternehmen glomex GmbH, im Weiteren glomex genannt, ist eine Tochter der ProSiebenSat.1 Media SE mit Hauptsitz in der Landsberger Str. 110 in München. Die glomex ist ein technischer Dienstleister zur Onlineverbreitung von Premium Videos. Das Unternehmen ermöglicht es Anbietern und Verbreitern von Videos, Inhalte zu syndizieren. In Form einer cloudbasierten Transaktionsplattform wird dazu die technische Infrastruktur zur Verfügung gestellt. Auf dieser Plattform, ist es möglich Videos zur Verfügung zu stellen, die wiederum von Verbreitern (Publishern) auf ihren Websites oder Apps eingebunden werden können. Um dies zu ermöglichen, bietet die glomex außerdem eine Video-Delivery-Infrastruktur zur Verbreitung von Videos, mit deren Hilfe der weltweite Content distribuiert und monetarisiert wird. Durch die signifikante Steigerung der Reichweite werden neue Umsätze erschlossen und gleichzeitig die Betriebskosten für Videoauftritte und Verbreitung reduziert [glomexAbout].

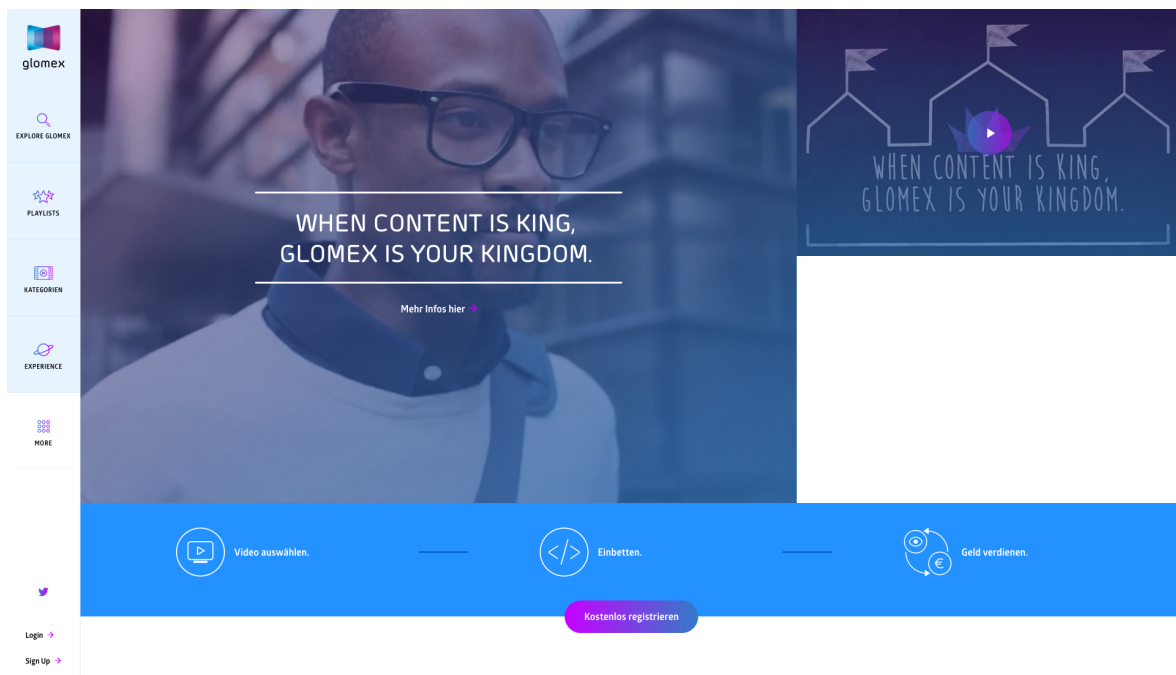


Abbildung 1.1: glomex Webseite (Quelle: [glomex])

## 2 Stand der Technik

---

Im Folgenden wird beschrieben was Qualitätsmanagement ist, welche die momentan wichtigsten Testarten sind und von welchen Frameworks Selenium profitiert. Zusätzlich werden weitere Blickwinkel zu diesem Themengebiet reflektiert und ein Überblick über den bisherigen Stand der Forschung gegeben.

### 2.1 Bedeutung-Qualitätsmanagement

#### Definition Qualitätsmanagement nach DIN-EN-ISO-9000:2000

Aufeinander abgestimmte Tätigkeiten zum Leiten und Lenken einer Organisation bezüglich Qualität. [schwarze2003kundenorientiertes]

Somit beschreibt der Begriff Qualitätsmanagement alle Maßnahmen und Tätigkeiten, die notwendig sind, um Qualität zu erzeugen [kochendorfer2004bau]. Qualitätsmanagement findet unternehmensweit statt und erfordert daher eine sehr gute Koordination. [reinhart1996unternehmen]

#### 2.1.1 Qualitätsmanagementsysteme

Damit Qualitätsmanagement überhaupt umgesetzt werden kann, wird ein sogenanntes Qualitätsmanagementsystem benötigt. Es enthält die Aufbau- und Ablauforganisation, um alle Qualitätsmanagementaufgaben durchzuführen und die folgenden Fragen zu klären [kochendorfer2004bau]

- **Was?** - Um welche qualitätsbezogenen Aufgaben handelt es sich?
- **Wer?** - Welche Personen nehmen qualitätsbezogene Aufgaben wahr?
- **Wann?** - Wie gestaltet sich der Ablauf qualitätsbezogener Aufgaben?
- **Wie?** - Was für Verfahren werden angewendet?

Mit der Norm DIN-EN-ISO-9000 wurde ein einheitlicher Standard zum Vergleich und Aufbau verschiedener Qualitätsmanagementsysteme erstellt [kochendorfer2004bau].

#### 2.1.2 Qualität

Der Begriff „Qualität“ sagt aus, dass jeder Kunde, Wünsche aufgrund der von ihm festgelegten Ansprüche besitzt, die gegenüber dem Hersteller gerne als Qualitätsanforderungen oder in Form von Erwartungen geäußert werden. Durch diese Anforderungen und Erwartungen

bildet sich wiederum eine Vorstellung des Produktpreises. Was für eine Qualität ein Produkt besitzt, ergibt sich schließlich erst durch den Vergleich zwischen der Beschaffenheit und den Produkthanforderungen des Kunden [**qualitaetsmanagement**].

## 2.2 QA-Abteilung

Eine Quality Assurance (QA)-Abteilung beschäftigt sich mit der Sicherstellung vorher definierter Qualitätsanforderungen. Großteils geht es darum, die technische Zuverlässigkeit von Prozessen und Produkten eines Unternehmens zu gewährleisten. Diese Gewährleistung führt zu einer Risikominimierung für das Unternehmen und den Kunden. Ein weiterer Aspekt ist die Kostenreduktion durch eine geringere Haftung auf Schadensersatz gegen den Hersteller. QA-Abteilungen müssen daher ständig die Qualität der Produkte und Dienstleistungen im Auge behalten und deren Zuverlässigkeit sicherstellen [**qa\_buch**].

## 2.3 Testpyramide

### Test-Pyramide

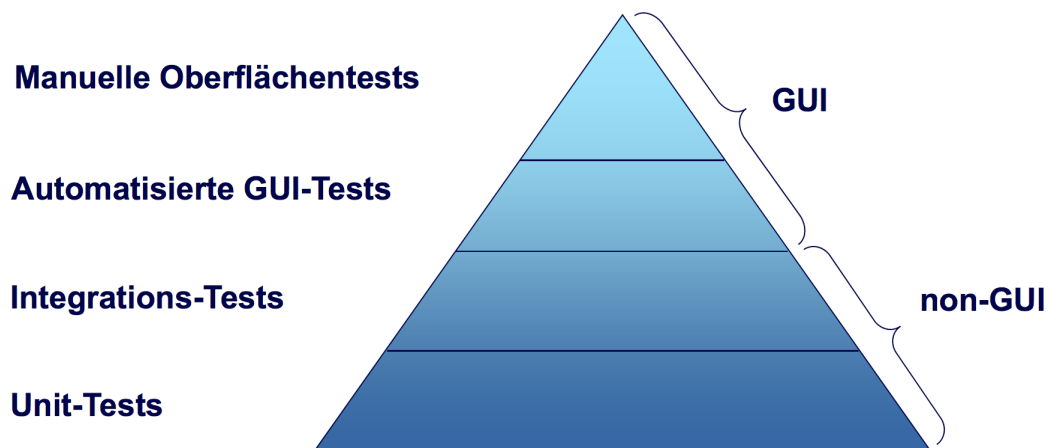


Abbildung 2.1: Testpyramide nach (Quelle: [**frueheTesterfaengtdenWurm**])

Zur besseren Einordnung bestimmter Tests wurde von Mike Cohn das Konzept der Test-Pyramide entwickelt. Die Pyramide soll die richtige Verteilung der unterschiedlichen Testarten verdeutlichen. Idealerweise gibt es im Fundament einen sehr hohen Anteil an einfach zu wartenden Unittests (siehe Kapitel 2.4.1). Integrations-Tests (siehe Kapitel 2.4.2) haben in der Regel längere Ausführzeiten und sind aufwendiger zu pflegen. Durch diesen Umstand werden Integrations-Tests meist nur zur zielgerichteten Prüfung von kritischen Schnittstellen

eingesetzt. Manuelle und automatisierte Oberflächentests (GUI-Tests) bilden die Spitze. Diese Tests eignen sich dazu, in der Gesamtheit die Funktionalität der Software zu gewährleisten. Oberflächentests eignen sich nicht dazu, alle möglichen Zweige innerhalb des Quellcodes zu überprüfen, daher sollte ihre Anzahl auch minimal gehalten werden [testpyramide].

## 2.4 Test-Arten

Im Kapitel 2.3 Testpyramide wurde beschrieben welche Testarten sich innerhalb der Testpyramide befinden und welchen Anteil diese ausmachen. Anhand der Testpyramide gegliedert, werden in diesem Kapitel einige Testarten genauer beleuchtet.

### 2.4.1 Unittests

Unittests, auch Modultests genannt, werden häufig seitens der Entwickler durchgeführt. Diese Art von Tests testen sämtliche Teile (Units) einer Software auf Funktionalität. Dabei wird als erstes der Ausgangszustand (Soll-Zustand) definiert. Hierbei werden Werte definiert, die die Funktionalität einer Unit gewährleisten. Anschließend wird bei der Ausführung, der resultierende Wert (Ist-Zustand) mit dem vorher definierten verglichen. Stimmen diese nicht überein, schlägt der Test fehl [testarten].

### 2.4.2 Integrationstests

Im Vergleich zu den Unittests, bei denen nur einzelne Module getestet werden, wird bei einem Integrationstest das Zusammenspiel der Module getestet. Die Voraussetzung dafür ist, dass geeignete Testdaten und Schnittstellen technisch zur Verfügung stehen. Integrationstests werden mit Testdaten (auch synthetische Daten genannt) durchgeführt. Wie auch bei anderen Testarten wird eine Liste mit einzelnen Schritten, die jeweils eine Beschreibung und ein erwartetes Ergebnis definieren, im Vorhinein erstellt und abgearbeitet [testarten].

### 2.4.3 Oberflächentests (GUI-Tests)

Unter dem Begriff Oberflächentests (GUI-Tests) werden alle Tests verstanden, bei denen der Tester versucht mit dem Produkt über die grafische Oberfläche (GUI) zu interagieren. Bei diesen Tests gibt es verschiedene Vorgehensweisen, die in diesem Kapitel beschrieben werden.

#### Explorativer Test

Exploratives Testing deckt Fehler in einer Software durch die Intuition des ausführenden Testers auf. Es werden keine detaillierten Testfälle im Vorhinein definiert. Dem Tester wird

nur eine grobe Vorgabe in Form einer Checkliste oder der Nennung eines Testgebietes, wie etwa „Teste eine Stunde das User Interface“, vorgegeben. Durch diese Vorgabe wird gewährleistet, dass der Tester zielgerichtet vorgeht. Auf welche Art und Weise der Tester dabei vorgeht, bleibt ihm vollkommen selbst überlassen [**winter2013testverfahren**].

## Akzeptanztest / Abnahmetest

Ein Akzeptanztest, auch Abnahmetest genannt, überprüft ob ein System alle vom Auftraggeber geforderten Anforderungen erfüllt. Diese Art von Test stellt sicher, dass der Anwender im späteren Betrieb mit der Anwendung arbeiten kann. Damit der Test realitätsnah ist, muss dieser auf einer möglichst realen Plattform mit entsprechenden Daten und Berechtigungen erfolgen [**testarten**].

## End-to-End Test

Komplexe Software enthält bestimmte Prozesse, die im Rahmen eines End-to-End Tests durchlaufen werden können. Diese Prozesse beziehen meistens mehrere Komponenten eines Systemverbunds mit ein. Ein End-to-End Test überprüft die Funktionstüchtigkeit eines bestimmten Prozesses. Als Beispiel lässt sich der Registrierungsprozess eines Forums nehmen (siehe Abbildung 2.2) [**testarten**].



Abbildung 2.2: End-to-End (Beispiel Registrierungsprozess)

## 2.5 Selenium

Wie bereits im Kapitel 2.3 Testpyramide veranschaulicht, lässt sich sehr gut erkennen, dass nur ein sehr geringer Teil aus Oberflächentests besteht. Um diese Tests zu automatisieren wurde im Jahre 2004 ein Framework namens Selenium entwickelt, mit dessen Hilfe es möglich ist auf Webelemente zuzugreifen und mit ihnen zu interagieren. Selenium ermöglicht es, Tests auf Basis der Skriptsprache JavaScript, durchzuführen und unterstützt dabei mehrere Browser. Im Laufe der Entwicklung wurde Selenium immer weiter ausgebaut und unterstützt mittlerweile Programmiersprachen wie etwa Java, C#, Ruby, Python und JavaScript. Mithilfe dieser Sprachen ist es möglich das Framework zu verwenden und somit auf Webelemente zuzugreifen [**selenium**].



## 2.5.1 Behavior Driven Development (BDD)

Oft fällt es QA-Mitarbeitern und Abteilungen schwer einen guten Startpunkt in Bezug auf das Testen zu finden und genau zu definieren was alles getestet werden soll und was nicht. Ein Resultat aus dieser Erkenntnis ist, dass die Sprache in der Tests definiert werden eine wichtige Rolle spielen. Eine Technik der agilen Softwareentwicklung ist das sogenannte Behavior Driven Development (BDD). Inspiriert durch [evans2004domain] nutzt BDD natürliche Sprache, um Tests zu beschreiben. Mithilfe der natürlichen Sprache wird gewährleistet, dass sowohl Stakeholder und Entwickler verstehen, um was es in diesem Test geht [soeken2012assisted].

Da Selenium alleine nur eine Schnittstelle zwischen Webanwendungen und Tests darstellt, reicht Selenium nicht aus, um automatisierte Tests in natürlicher Sprache zu definieren. Aus diesem Grund gibt es die Beschreibungssprache „Gherkin“. Gherkin ist eine zeilenorientierte Sprache, die ähnlich wie Python Einzüge zur Orientierung nutzt. Der Parser unterscheidet hierbei die Schlüsselwörter „Feature“, „Scenario“, „Given“, „When“, „Then“, „And“ und „But“. Je nach Framework kann die Sprache angepasst werden [gherkin]. Wenn beispielsweise Deutsch gewählt wird, werden die Schlüsselwörter wie folgt angepasst:

- Given - Angenommen / Gegeben sei / Gegeben seien
- When - Wenn
- Then - Dann
- And - Und
- But - Aber

Auf Basis von Gherkin gibt es Frameworks, die aufgrund der textuellen Spezifikation, mithilfe einer bestimmten Programmiersprache Tests automatisieren. Einige Beispiele sind:

- Cucumber [cucumber]
- Lettuce [lettuce]
- Behave [behave]

```
# language: de
```

```
Funktionalität: Ein einfacher Test zur demonstration
Dieser Test dient der Demonstration des Lesers,
```

```

5  damit dieser einen Eindruck davon erhält wie ein
   Cucumber Test aussieht.

   Scenario: Demonstration
       Angenommen eine Tomate wiegt 100 Gramm
10  Wenn ich diese Tomate halbiere
       Dann gibt es 2 Stücke mit je 50 Gramm

```

Codebeispiel 2.5.1: Beispieltest Cucumber

Jede Zeile innerhalb eines Szenarios führt eine Methode aus, die ein entsprechendes Ergebnis zurück gibt. Die ausführbaren Methoden werden mithilfe einer Programmiersprache wie etwa Python, Ruby oder Java erstellt. Anzumerken ist hierbei, dass das verwendete BDD-Framework die Programmiersprache unterstützen muss. Behave unterstützt beispielsweise nur Python [**behave**].

Innerhalb dieser Methoden kann Selenium verwendet werden, um auf Webelemente zuzugreifen. So können beispielsweise mithilfe von Selenium, Werte aus Eingabefeldern oder Webelementen abgefragt werden. Diese Werte können entsprechend gegen geprüft werden. Entspricht ein Ergebnis nicht dem Erwarteten, wird das entsprechende Szenario mit einem Fehler abgebrochen [**behave**].

## 2.5.2 Pytest

Doch nicht alle Frameworks nutzen Gherkin als Beschreibungssprache. Ein Beispiel hierfür ist Pytest. Pytest lässt sich nicht wie Gherkin lesen, sondern erhält über sogenannte „Fixtures“ Objekte oder Werte die relevant für den jeweiligen Test sind. Anschließend werden Methoden Zeile für Zeile ausgeführt und daraus generierte Werte mithilfe von „Assertions“ geprüft [**pytest**]. Eine Assertion (englisch für Behauptung) ist die Aussage eines Zustandes innerhalb eines Programms.

```
import pytest

class WebKonfiguration:

5   @pytest.fixture
   def port(self):
       return 80
```

Codebeispiel 2.5.2: fixture\_beispiel.py, enthält die Beispielfixture „port“

```
from fixture_beispiel import WebKonfiguration

class Test(WebKonfiguration):

5   def test_proof_port(self, port):
       assert port == 80
```

Codebeispiel 2.5.3: test\_beispiel.py, enthält den Test „test\_proof\_port“, assert True

In der Datei „fixture\_beispiel.py“ (siehe Codebeispiel 2.5.2) wird mithilfe des Befehls „class“ eine Klasse gebildet, um eine Struktur zu schaffen (siehe Zeile 3). Anschließend wird eine Fixture mithilfe des „@pytest.fixture“ Befehls erzeugt. Diese Fixture mit dem Namen „port“, generiert einen Wert, der zurückgegeben wird (Zeile 7).

Die Datei „test\_beispiel.py“ enthält die eigentlichen Tests (siehe Codebeispiel 2.5.3). Über den Befehl „from fixture\_beispiel import WebKonfiguration“ werden alle Fixtures der Klasse „WebKonfiguration“ aus der Datei „fixture\_beispiel.py“ importiert und somit nutzbar gemacht. Die soeben importierte Webkonfiguration wird an die Klasse „Test“ übergeben, damit sie von allen hierarchisch folgenden Tests genutzt werden kann.

Der Test „test\_proof\_port“ (siehe Codebeispiel 2.5.3) erhält über den Übergabeparameter „port“ den Rückgabewert der Fixture „port“ (Wert 80, siehe Codebeispiel 2.5.2 Zeile 7). Innerhalb des Tests wird mithilfe des „assert“-Befehls überprüft, ob der erwartete Wert dem aus der Fixture „port()“ gleicht.

Ergibt dieser Vergleich True, wird während der Testausführung folgende Ausgabe generiert:

```
===== 1 passed in 0.01 seconds =====
[c700693:vvs-automation sch0053d$ pytest test_beispiel.py
===== test session starts =====
platform darwin -- Python 2.7.10, pytest-3.1.0, py-1.4.31, pluggy-0.4.0
rootdir: /Users/sch0053d/PycharmProjects/vvs-automation, inifile:
collected 1 items

test_beispiel.py F
```

Abbildung 2.3: Pytest, Vergleich (Assert) ergibt True

Würde hingegen (siehe Codebeispiel 2.5.4) in Zeile sechs gegenteilig geprüft werden, ergibt sich folgendes:

```
from fixture_beispiel import WebKonfiguration

class Test(WebKonfiguration):

5  def test_proof_port(self, port):
    assert port != 80
```

Codebeispiel 2.5.4: test\_beispiel.py, enthält den Test „test\_proof\_port“, assert False

```
===== FAILURES =====
test_proof_port

port = 80

def test_proof_port(port):
> assert port != 80
E assert 80 != 80

test_beispiel.py:8: AssertionError
===== 1 failed in 0.03 seconds =====
```

Abbildung 2.4: Pytest, Vergleich (Assert) ergibt False

Aus dem Kapitel 2.5.1 geht hervor, dass alle erwähnten Frameworks Selenium benötigen um auf Webelemente zuzugreifen (Siehe [cucumber], [lettuce], [behave], [pytest]). QA-Mitarbeiter haben so zwar eine Auswahl verschiedener Programmiersprachen und Frameworks, sind allerdings bei der Erstellung der Oberflächentests für Webanwendungen auf Selenium angewiesen.

## 2.6 Cypress

Cypress, auch Cypress.io genannt, versucht ein Gesamtpaket zur Erstellung von Oberflächentests für Webanwendungen anzubieten. Anders als alle Frameworks die in Kapitel 2.5.1 beschrieben wurden, ist Cypress nicht auf Selenium angewiesen und verspricht vor allem das Erstellen, Schreiben, Ausführen und Debuggen von Tests zu vereinfachen. Da es ebenfalls ein Funktions- und Akzeptanztesttool für Webanwendungen ist, wird es häufig mit Selenium verglichen [cypress].

Standardmäßig bringt Cypress den sogenannten „Cypress Test Runner“ mit. Dieser listet alle verfügbaren Tests auf, zeigt Testdurchläufe an und ist in der Lage ein Cypress Projekt über die Einstellungen individuell anzupassen. Zusätzlich ist es möglich die aufgelisteten Tests in einem ausgewählten Browser laufen zu lassen. Während der Ausführung, wird im Browser in einer Spalte der Ablauf in Echtzeit übersichtlich dargestellt [cypress].

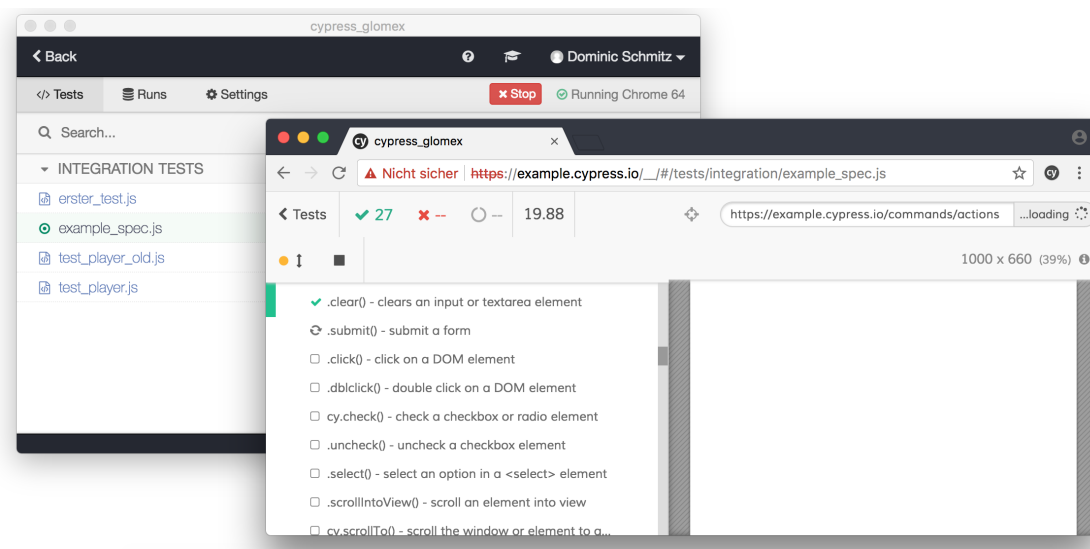


Abbildung 2.5: Cypress Test Runner während einer Testausführung

## 3 Installation / Konfiguration

---

Damit beide Frameworks miteinander verglichen werden können, muss zunächst Cypress installiert werden. Die bestehende Selenium-Pytest Infrastruktur existiert bereits im Unternehmen und muss nur mithilfe des Versionsverwaltungstools Git, heruntergeladen werden.

In diesem Kapitel wird beschrieben, wie die Installation und Konfiguration von Cypress stattfindet. Zusätzlich werden die Voraussetzungen zur Installation von Cypress erläutert und welches System zur Testautomatisierung im Rahmen dieser Arbeit verwendet wird.

### 3.1 Umgebung

Das Unternehmen glomex benötigt ein System, dass alle Anforderungen unterstützt. Die Umgebung muss von Cypress.io sowie von Selenium unterstützt werden.

Cypress wird derzeit (Stand 13.02.2018) auf folgenden Plattformen supportet [**cypress**]:

- Mac OS 10.9+ (Mavericks+), only 64bit binaries are provided for macOS.
- Linux Ubuntu 12.04+, Fedora 21, Debian 8.
- Windows 7+, only 32bit binaries are provided for Windows.

Selenium setzt nur einen installierten Browser und eine der folgenden Programmiersprachen voraus [**selenium**]:

- Java
- C#
- Ruby
- Python
- Javascript (Node)

Da unternehmensweit bei der glomex Mac OS genutzt wird und beide Frameworks, sowohl Cypress als auch Selenium unterstützt werden, wurde entschieden Mac OS zu verwenden.

## 3.2 Cypress Test Runner

### 3.2.1 Installation unter Mac OS



Abbildung 3.1: Cypress Installationsprozess (Mac OS)

Bei der Verwendung von Mac OS muss erst das Programm „Node.js“ installiert werden, da dieses den Paketmanager Npm enthält (siehe Abbildung 3.1). Npm ist ein Paketmanager für JavaScript und enthält tausende frei zum Download verfügbare Pakete [**npm**]. Für das Betriebssystem Mac OS wird empfohlen, Cypress mithilfe von Npm zu installieren, da Npm in der Lage ist, Cypress zu aktualisieren und zu installieren [**cypress**].

Nach der Installation steht der Befehl „npm“ in einer eingabebasierenden Benutzerschnittstelle der sogenannten Bash zur Verfügung. Mithilfe des Befehls „npm install cypress“ kann nun die Installation des Cypress Test Runners gestartet werden (siehe Abbildung 3.1).

### 3.2.2 Konfiguration

Nach Abschluss der Installation ist der sogenannte „Cypress Test Runner“ installiert. Dieser enthält alle wichtigen Tools und Einstellungen um Tests zu starten und Ergebnisse auszuwerten.

Es gibt zwei Wege den Cypress Test Runner zu starten:

- **Mithilfe des Bashbefehls „node\_modules/.bin/cypress open“ (siehe Abbildung 3.1)**  
Dabei ist zu beachten, dass der Befehl nur funktioniert, wenn der Anwender sich im Cypress Projektverzeichnis befindet. Zusätzlich ist der Anwender mit dieser Methode nicht in der Lage das Projektverzeichnis auszuwählen.
- **Direkt über Programme**  
Bei dieser Methode kann der Anwender selber ein vorhandenes Projektverzeichnis auswählen oder ein neues erstellen.

### 3.2.3 Update

Cypress überprüft vor jedem Start ob eine neue Version verfügbar ist. Sobald eine neue Version zur Verfügung steht, erscheint ein Update-Button im unteren Bereich des Cypress Test Runners. Über diesen Button wird einem explizit angezeigt, welche Schritte der Anwender zu gehen hat, damit Cypress aktualisiert wird.





Abbildung 3.2: Cypress Update

Nachdem das Update durchgeführt wurde, kann die neue Version des Cypress Test Runners, wie in Kapitel 3.2.2 beschrieben, gestartet werden.

### 3.3 Visual Studio Code

Sobald der Cypress Test Runner installiert wurde, wird noch zusätzlich ein Editor zum Erstellen und Editieren der Tests in Form von JavaScript (.js) Dateien benötigt. Im Rahmen dieser Arbeit wurde entschieden „Visual Studio Code“ zu verwenden, da dieser Editor alle gängigen Plattformen unterstützt und ein Syntax-Highlighting für JavaScript beinhaltet. Im Vergleich zur Installation des Cypress Test Runners, muss nur eine Installationsdatei heruntergeladen und installiert werden [**visualstudio**].

# 4 Cypress Einführung

---

Um mit Cypress effektiv Tests zu erstellen, muss erst verstanden werden, wie ein Cypress Projekt grundsätzlich aufgebaut ist. Die in diesem Kapitel beschriebene Struktur wird von Cypress empfohlen. Es ist möglich diese Struktur anzupassen und zu verändern. Im Rahmen dieser Arbeit wird darauf allerdings verzichtet. Zusätzlich werden in diesem Abschnitt die wichtigsten Kommandos erläutert.

## 4.1 Projektstruktur

Die von Cypress empfohlene Struktur wird bereits beim ersten Anlegen eines Projektes erstellt. Änderungen können mithilfe des Cypress Test Runners über die Einstellungen konfiguriert werden.

Beim ersten Start des Cypress Test Runners kann ein Projektordner ausgewählt werden. Wird ein Ordner gewählt, wird geprüft, ob bereits eine cypress.json Datei vorhanden ist. Diese Datei enthält die Project ID mithilfe dieser das Projekt einer Organisation zugeordnet werden kann (siehe 5.2). Ist diese Datei nicht enthalten, erstellt Cypress ein neues Projekt mit der folgenden Struktur:

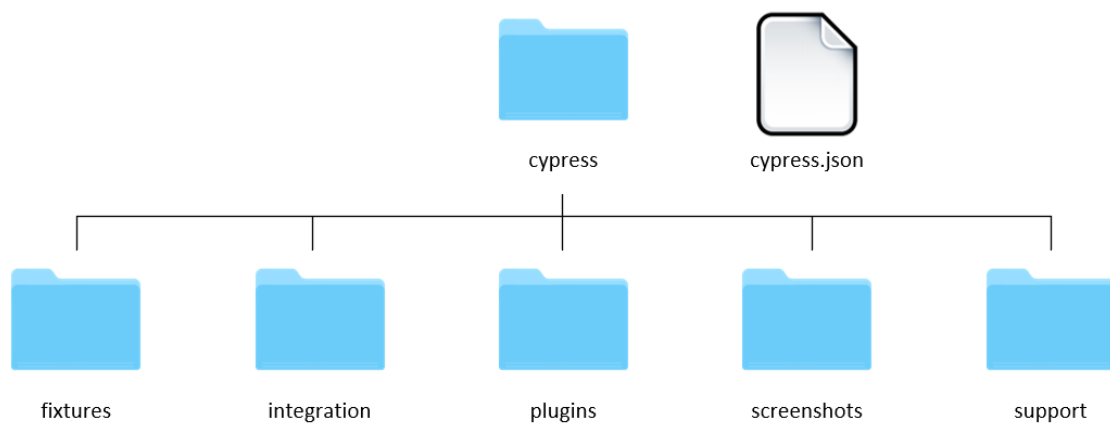


Abbildung 4.1: Cypress Projektstruktur

Neben einer cypress.json Datei wird der Ordner cypress erstellt, der die Ordner „fixtures“, „integration“, „plugins“, „screenshots“ und „support“ enthält (siehe Abbildung 4.1). Standardmäßig enthält jeder Ordner Beispieldateien.

- **fixtures**

Der Ordner fixtures enthält json Dateien. Das Format json ist ein schlankes Datenaustauschformat, dass für Menschen einfach zu lesen und zu schreiben ist [json]. Cypress verwendet diese Dateien dazu Werte zu speichern, die wiederholt für Tests benötigt werden. Jeder Test kann eine oder mehrere fixtures implementieren, um gespeicherte Werte wiederholt abzurufen.

```
{
  "name": "Using fixtures to represent data",
  "email": "hello@cypress.io"
}
```

Codebeispiel 4.1.1: Fixture json Beispiel (Quelle: [cypress])

- **integration**

Eine der wichtigsten Funktionen übernimmt der Ordner „integration“. In diesem Ordner werden alle Tests in Form von JavaScript Dateien (.js) gespeichert. Cypress basiert auf der Skriptsprache JavaScript und verwendet diese zur Erstellung von Tests [cypress].

- **plugins**

Plugins erlauben dem Benutzer das grundlegende Verhalten von Cypress zu verändern oder zu erweitern. Sie sind eine Nahtstelle, um eigenen benutzerdefinierten Code zu schreiben und diesen während bestimmter Phasen eines Tests auszuführen [cypress].

- **screenshots**

Um zu gewährleisten, dass bestimmte Zustände erreicht wurden, können zur Sicherheit Screenshots während des Testablaufs erstellt werden. Ein Screenshot ist eine Momentaufnahme, die in der Regel als Bild gespeichert wird. Dieses Bild wird in dem Ordner „screenshots“ gespeichert und kann entsprechend nach dem Testablauf aufgerufen werden [cypress].

- **support**

Innerhalb des Ordners „support“, befindet sich die Datei index.js. Diese Datei bestimmt, welcher Code automatisch vor jeder Testausführung geladen wird. So können beispielsweise bestimmte Kommandos erstellt, in einer Datei gespeichert und mithilfe der index.js vor jeder Testausführung automatisch geladen werden. So kann ein selbst definiertes Kommando für jeden Test bereitgestellt werden [cypress].

Im Rahmen dieser Arbeit ist seitens der glomex gewünscht, die in Abbildung 4.1 verwendete Struktur beizubehalten. Alle Cypress Tests, die in dieser Arbeit erstellt und ausgeführt werden, verwenden die beschriebene Struktur.

## 4.2 Blockstrukturen

Um einen Test zu erstellen, wird eine neue JavaScript Datei (.js) mithilfe Visual Studio Code (siehe Kapitel 3.3) innerhalb des „integration“ Ordners (siehe Abbildung 4.1) erstellt. Jeder Cypress Test besitzt eine definierte Struktur. Diese enthält grundlegend einen „describe Block“, der beschreibt, um was für einen Test es sich handelt. Innerhalb des „describe Blocks“ können nun folgende Konstrukte verwendet werden:

- **it**

Ein sogenannter „it Block“ repräsentiert einen Test. In ihm befinden sich alle Kommandos die zur Abarbeitung des Tests benötigt werden [cypress].

```
describe('describe Block', function() {  
  it('Demonstriert einen it Block', function() {  
    // Hier Kommandos  
  })  
5  })
```

Codebeispiel 4.2.1: it Block

- **context**

„context Blöcke“ enthalten Tests und dienen der Übersichtlichkeit. Aus diesem Grund können „context Blöcke“ „it Blöcke“ und weitere Konstrukte enthalten. Die Aufteilung in „context Blöcke“ hat den Vorteil, dass mithilfe eines „describe Blocks“ ein Gebiet definiert werden kann, in dem mehrere „Context Blöcke“ abgearbeitet werden [cypress].

Ein Beispiel wäre ein Test eines glomex Video Players. So kann als „describe Block“ „Player Test“ definiert werden. Innerhalb dieses Blocks können nun verschiedene Videoplayer mithilfe der „context Blöcke“ getestet werden.

```

describe('Player Test', function() {

  context('Flash Player', function() {
    it('Pause Test', function() {
5      // Kommandos um Pause im Flash Player zu testen
    })
  })

  context('HTML Player', function() {
10    it('Pause Test', function() {
      // Kommandos um Pause im HTML Player zu testen
    })
  })

15 })

```

Codebeispiel 4.2.2: context Block

- ▼ Player Test
  - ▼ Flash Player
    - ✓ Pause Test
  - ▼ HTML Player
    - ✓ Pause Test

Abbildung 4.2: Cypress Test Struktur

- **before und beforeEach**

Um Code vor einem oder mehreren Tests auszuführen, werden sogenannte „before“ oder „beforeEach“ Blöcke verwendet. Der Unterschied zwischen diesen beiden Blöcken ist, dass der „before Block“ nur einmalig vor dem nächst folgenden „it Block“ ausgeführt wird. Hingegen wird der „beforeEach Block“ vor jedem „it Block“ der hierarchisch folgt erneut abgearbeitet. Beide Blöcke können in jeder Hierarchieebene verwendet werden.

```
describe('Player Test', function() {  
  
    before(function() {  
        /*  
5      Wird nur vor "Flash Player – Pause Test"  
        ausgeführt, da es sich um den nächst  
        hierarchischen Test handelt.  
        */  
    })  
10  
  
    beforeEach(function() {  
        /*  
        Wird vor jedem hierarchisch folgenden Test  
        ausgeführt. In diesem Fall vor:  
15  
        – Flash Player – Pause Test  
        – HTML Player – Pause Test  
        */  
    })  
20  
  
    context('Flash Player', function() {  
        it('Pause Test', function() {  
            // Kommandos um Pause im Flash Player zu testen  
        })  
25    })  
  
    context('HTML Player', function() {  
        it('Pause Test', function() {  
            // Kommandos um Pause im HTML Player zu testen  
30        })  
    })  
    })  
})
```

Codebeispiel 4.2.3: before und beforeEach BBlock

## 4.3 Wichtige Kommandos

Tests arbeiten eine Folge von Kommandos sequentiell ab. Cypress stellt eine Reihe dieser Kommandos zur Verfügung, von denen jedes eine bestimmte Funktionalität übernimmt. In diesem Abschnitt werden nur Kommandos mit Relevanz für diese Arbeit erläutert. Unter [cypress] kann eine vollständige Kommandoliste eingesehen werden.

- **cy.visit(URL)**

Mithilfe des „visit“ Befehls wird eine Webseite aufgerufen.

**Beispiel:**

```
cy.visit('www.google.de')
```

Codebeispiel 4.3.1: Ruft die Webseite „www.google.de“ auf

- **cy.wait(Zeit)**

Der Befehl „wait“ pausiert einen Test für eine bestimmte Zeit. Die Zeit wird in Millisekunden angegeben.

**Beispiel:**

```
cy.wait(1000)
```

Codebeispiel 4.3.2: Wartet eine Sekunde bis der Test fortgeführt wird

- **cy.get(Locator)**

Um auf ein Webelement zuzugreifen gibt es den „get“ Befehl. Dieser gibt ein oder mehrere Webelemente einer Webseite zurück.

**Beispiel:**

```
cy.get('button[data-qa=playbackButton]')
```

Codebeispiel 4.3.3: Gibt einen Playback Button zurück

- **.click()**

Es gibt Befehle, die nur mithilfe weiterer Funktionen funktionieren. Der Befehl `click()` klickt ein Webelement. Da das Webelement erst gefunden werden muss, um geklickt werden zu können, wird der Befehl `get` (siehe Codebeispiel 4.3.3) vorausgesetzt.

**Beispiel:**

```
cy.get(button[data-qa=playbackButton]).click()
```

Codebeispiel 4.3.4: Klickt den Playback Button

- **.should(Bedingung)**

Zusätzlich zum „`get`“ Befehl können weitere Bedingungen mithilfe des „`should`“ Befehls angegeben werden, um ein Webelement genau zu spezifizieren. Trifft diese Bedingung nicht innerhalb einer bestimmten Zeit zu, wird der Test mit einem Fehler beendet (siehe Assertion).

**Beispiel:**

```
cy.get(button[data-qa=playbackButton])  
  .should('be.visible')
```

Codebeispiel 4.3.5: Sucht nach einem „sichtbaren“ Playback Button

- **.find(Locator)**

Das Kommando „`find`“, findet ein Webelement innerhalb eines Webelements.

**Beispiel:**

```
cy.get('.article').find('footer')
```

Codebeispiel 4.3.6: Finde „`footer`“ innerhalb „`article`“

- **cy.screenshot(Name)**

Um einen Screenshot zu erstellen, muss der Befehl „`screenshot`“ ausgeführt werden. Als Parameter muss ein Name übergeben werden, unter dem dieser gespeichert wird. Der erstellte Screenshot wird in dem Ordner „`screenshots`“ (siehe Abbildung 4.1) als Bilddatei abgespeichert.

**Beispiel:**

```
cy.screenshot('Bildname')
```

Codebeispiel 4.3.7: Erstellt ein Bild mit dem Namen „`Bildname`“ im Ordner „`screenshots`“



- **cy.log(Meldung)**

Jeder Test gibt während der Ausführung einen Eventlog aus. Mithilfe des „log“ Befehls kann eine Meldung erzeugt und zur Laufzeit im Eventlog ausgegeben werden.

**Beispiel:**

```
cy.log(Dies ist eine Meldung)
```

Codebeispiel 4.3.8: Schreibt „Dies ist eine Meldung“ in den Log

- **cy.reload()**

Der Befehl „reload“ lädt die Webseite neu, auf der sich der Test Momentan befindet.

**Beispiel:**

```
cy.reload()
```

Codebeispiel 4.3.9: Lädt die Seite neu

- **cy.scrollTo(Position)**

Mithilfe des „scrollTo“ Kommandos scrollt der Test an eine angegebene Position. Es können xy Koordinaten als Position verwendet werden oder direkte Befehle wie etwa top, right, left oder bottom.

**Beispiel:**

```
cy.scrollTo('top')
```

Codebeispiel 4.3.10: Scrollt bis an den Anfang (top) der Webseite

- **cy.wrap(subject)**

Der Befehl „wrap“, zu Deutsch „wickeln“ oder auch „verpacken“ legt ein zu nutzendes Objekt fest. Mithilfe von invoke kann mit dem Objekt interagiert werden.

**Beispiel:**

```
cy.wrap({ name: "Jane Lane" })  
  .invoke('name').should('eq', 'Jane Lane') // true
```

Codebeispiel 4.3.11: Holt den Namen „Jane Lane“

# 5 Automatisierung der Ausführung

---

## 5.1 Testimplementierung in Travis

In diesem Kapitel wird erläutert, was Travis ist und inwiefern damit Cypress Tests automatisiert werden können. Zusätzlich wird näher auf das Cypress Dashboard eingegangen.

### 5.1.1 Travis

Mit Travis ist es möglich Software Tests automatisiert auszuführen. Travis kann mit einem öffentlichen (kostenfrei) oder einem privaten (Kostenpflichtig) Github Repository verknüpft werden. Die Website hat dafür zwei unterschiedliche Domains bereitgestellt. `travis-ci.org` für öffentliche Projekte und `travis-ci.com` für private Projekte. Jedes Mal wenn eine Änderung des Codes auf dem verknüpften Github-Repository stattfindet, lässt sich mithilfe von Konfigurationsdateien bestimmen, welche Anweisungen erfolgen. Diese Anweisungen müssen nicht zwingend das Ausführen von Tests enthalten, es können auch Anweisungen zum Versenden von Nachrichten oder Erstellen von Berichten sein [[travis](#)].

### 5.1.2 Git / Github

- **Git**

Git ist ein System zur verteilten Versionsverwaltung mehrerer Dateien. Es wurde unter den Aspekten Geschwindigkeit, einfaches Design, guter Unterstützung von nicht-linearer Entwicklung, vollständiger Verteilung und der Fähigkeit große Projekte effektiv zu verwalten entwickelt. Das Prinzip besteht darin, den Zustand einer oder mehrerer Dateien zu sichern und diese sogenannten Snapshots, unter einer Referenz abzuspeichern. Bei diesem Vorgang werden unveränderte Dateien nicht erfasst, es wird lediglich eine Verknüpfung zu der Datei erstellt. Durch diese Vorgehensweise wird die Effizienz und Geschwindigkeit besonders bei großen Projekten enorm gesteigert. Ein weiterer großer Vorteil ist die Möglichkeit einen Branch zu erstellen. Ein Branch in Git ist ein simpler Zeiger, der auf einen Codestand verweist. Er ermöglicht es, an mehreren Codeständen zur selben Zeit zu entwickeln und diese später zusammenzuführen [[gitbook](#)].

- **Github**

Github ist eine im Jahre 2007 in San Francisco gegründete Website, mit der es möglich ist Entwicklungsprojekte anzulegen und zu verwalten. In sogenannten Repositorys werden Daten und Codestände verschiedener Projekte gespeichert und können mithilfe der Versionsverwaltung Git abgerufen und verändert werden [[github](#)].

### 5.1.3 Testimplementierung

Um Travis nutzen zu können, müssen im ersten Schritt folgende Voraussetzungen erfüllt werden [**travisStart**]:

- GitHub login
- Zu automatisierendes Projekt, bereitgestellt auf einem GitHub-Repository
- Lauffähiger Code im Projekt
- Lauffähiger Stand oder Testscript

Nachdem alle Voraussetzungen erfüllt sind, kann nun der Travis Login stattfinden. Hierbei ist zu beachten, dass die Zugriffsvereinbarungen akzeptiert werden, da Travis sonst keinen Zugriff auf das zu automatisierende Projekt erhält. Anschließend kann direkt über die eigenen Profileinstellungen das GitHub-Repository mit dem lauffähigen Code ausgewählt werden. Über diese Einstellung kann auch konfiguriert werden, wann Tests automatisiert ausgeführt werden sollen. Im Rahmen dieser Arbeit wurde eine Testausführung, einmal am Tag und immer wenn der Master Branch des Github-Repositorys verändert wurde, eingestellt.

Im nächsten Schritt muss eine Datei mit dem Namen „.travis.yml“ im Hauptverzeichnis des GitHub-Repositorys erstellt werden. Mithilfe dieser Datei erhält Travis Informationen darüber, was während einer Testausführung geschehen soll.

```
language: node_js

node_js:
  - 6

5 cache:
  directories:
    - ~/.npm
    - node_modules

10 install:
  - npm install cypress --save-dev

before_script:
15  - npm start -- --silent &
```

**script:**

```
– cypress run --record --key 12345678-90ab-cdef-1234-567890  
  abcdef --spec cypress/integration/test_player.js
```

## Codebeispiel 5.1.1: Travis Konfigurationsdatei (.travis.yml)

Die Hauptparameter des Codebeispiels 5.1.1 wurden der Webseite [**cypressTravisYml**] entnommen. Wichtig hierbei ist vor allem die Angabe des Tests in Zeile 18. Dabei übernehmen die folgenden Parameter, bestimmte Aufgaben.

- **–record**

Mithilfe von `–record`, ist der Test in der Lage ein Video oder Screenshots des Testablaufs zu erstellen. Diese können über das Cypress Dashboard angesehen oder heruntergeladen werden (siehe Kapitel 5.2).


- **–key**

Um Videos und Screenshots einem Cypress Account zuzuordnen, gibt es den `–key` Parameter. Ein Key kann über das Cypress Dashboard generiert und muss bei der Testausführung mit angegeben werden.

- **–spec**

Durch die Angabe von `–spec`, wird genau spezifiziert, welcher Test ausgeführt werden soll. Wird der Parameter nicht angegeben, werden alle Tests die sich in dem Ordner `integration` befinden, chronologisch abgearbeitet.

Alle bisher ausgeführten Tests können auch über die Weboberfläche händisch neu gestartet werden. Zusätzlich gibt es einen Log, der den genauen Ablauf eines jeden Tests genau dokumentiert (siehe Abbildung 5.2).

glomex / cypress\_glomex 

Current Branches Build History Pull Requests More options

|          |   |                         |                                   |
|----------|---|-------------------------|-----------------------------------|
| ✓ master | travis.yml optimized  | ✓ #98 passed<br>10e8c66 | 5 min 15 sec<br>43 minutes ago    |
| ✓ master | travis.yml optimized  | ✓ #97 passed<br>10e8c66 | 6 min 22 sec<br>about an hour ago |
| ✓ master | travis.yml changed  | ✓ #96 passed<br>2f01d22 | 5 min 20 sec<br>about an hour ago |
| ✗ master | travis.yml changed  | ✗ #95 failed<br>e6627de | 1 min 59 sec<br>3 minutes ago     |
| ✓ master | CRON Tests fixed and implemented a new Command named "iframe" | ✓ #94 passed<br>b804918 | 4 min 32 sec<br>a day ago         |
| ✓ master | CRON Tests fixed and implemented a new Command named "iframe" | ✓ #93 passed<br>b804918 | 4 min 48 sec<br>2 days ago        |
| ✓ master | CRON Tests fixed and implemented a new Command named "iframe" | ✓ #92 passed<br>b804918 | 4 min 44 sec<br>3 days ago        |
| ✓ master | CRON Tests fixed and implemented a new Command named "iframe" | ✓ #91 passed<br>b804918 | 5 min 56 sec<br>4 days ago        |
| ✓ master | CRON Tests fixed and implemented a new Command named "iframe" | ✓ #90 passed<br>b804918 | 4 min 49 sec<br>5 days ago        |
| ✓ master | Tests fixed and implemented a new Command named "iframe"      | ✓ #89 passed<br>b804918 | 5 min 28 sec<br>6 days ago        |

Abbildung 5.1: Travis Testausführung der letzten sechs Tage (Cypress Tests)

```

(Tests Finished)

- Tests:      3
- Passes:     3
- Failures:   0
- Pending:    0
- Duration:   1 minute, 52 seconds
- Screenshots: 6
- Video Recorded: true
- Cypress Version: 2.1.0

(Screenshots)

- /home/travis/build/glomex/cypress_glomex/cypress/screenshots/Preroll.png (1280x720)
- /home/travis/build/glomex/cypress_glomex/cypress/screenshots/Paused.png (1280x720)
- /home/travis/build/glomex/cypress_glomex/cypress/screenshots/Midroll.png (1280x720)
- /home/travis/build/glomex/cypress_glomex/cypress/screenshots/Postroll.png (1280x720)
- /home/travis/build/glomex/cypress_glomex/cypress/screenshots/Muted.png (1280x720)
- /home/travis/build/glomex/cypress_glomex/cypress/screenshots/Unmuted.png (1280x720)

(Video)

- Started processing: Compressing to 32 CRF
- Finished processing: /home/travis/build/glomex/cypress_glomex/cypress/videos/u0io8.mp4 (40 seconds)

(Uploading Assets)

- Done Uploading (1/7) /home/travis/build/glomex/cypress_glomex/cypress/screenshots/Unmuted.png
- Done Uploading (2/7) /home/travis/build/glomex/cypress_glomex/cypress/screenshots/Midroll.png
- Done Uploading (3/7) /home/travis/build/glomex/cypress_glomex/cypress/videos/u0io8.mp4
- Done Uploading (4/7) /home/travis/build/glomex/cypress_glomex/cypress/screenshots/Preroll.png
- Done Uploading (5/7) /home/travis/build/glomex/cypress_glomex/cypress/screenshots/Muted.png
- Done Uploading (6/7) /home/travis/build/glomex/cypress_glomex/cypress/screenshots/Postroll.png
- Done Uploading (7/7) /home/travis/build/glomex/cypress_glomex/cypress/screenshots/Paused.png

(All Done)

The command "cypress run --record --key [redacted] --spec cypress/integration/test_player.js" exited with 0.
store build cache

Done. Your build exited with 0.

```

Abbildung 5.2: Travis Log nach Abschluss eines Cypress Tests

## 5.2 Cypress Dashboard

Cypress bietet ein eigenes Dashboard zur genaueren Analyse automatisierter Tests unter [cypressDashboard] an. Wie auch bei Travis ist für einen Login, ein GitHub Account Voraussetzung.

### Dashboard Login



Abbildung 5.3: Dashboard Login (Quelle: [cypressDashboardLogin])

Damit das Dashboard auch Daten zur Auswertung erhält, muss es erst verknüpft werden. Diese Verknüpfung geschieht durch die Erstellung einer sogenannten „Project ID“. Um diese zu generieren, muss ein Projekt erstellt und einer Organisation zugeordnet werden. Als Organisation wird die glomex angegeben. Der Projektname lässt sich frei wählen und lautet in diesem Fall „cypress\_glomex“. Nachdem das Projekt erstellt ist, kann über die Einstellungen die sechs stellige Project ID ausgelesen werden.

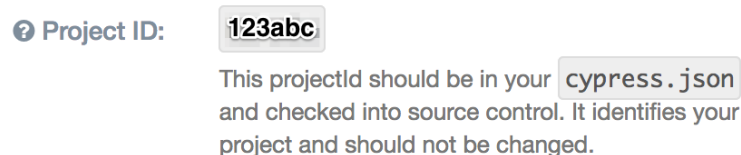


Abbildung 5.4: Project ID (aus Sicherheitsgründen hier nur fiktiv)

Cypress gibt an, die Project ID in die „cypress.json“ (siehe Abbildung 4.1) Datei einzutragen und nicht wieder zu verändern (siehe Abbildung 5.4). Die Datei muss ebenso in das aktuelle Repository hochgeladen werden. Anhand der Project ID verknüpft Cypress während jeder der Testausführung über Travis, den ausgeführten Test, mit dem im Dashboard erstellten Projekt. So wird nach jeder Testausführung automatisch jeder Test im Cypress Dashboard angezeigt.

| NO. | DETAILS   | RAN         | DURATION | ENVIRONMENT    | CI         | PASSING | FAILING |
|-----|---|-------------|----------|----------------|------------|---------|---------|
| #42 | master /<br>/home/travis/build/glomex/cypress_glomex/cypress/integration/test_player.js<br>travis.yml optimized                                     | 2 hours ago | 01:52    | Ubuntu - 14.04 | Travis #98 | 3       | 0       |
| #41 | master /<br>/home/travis/build/glomex/cypress_glomex/cypress/integration/test_player.js<br>travis.yml optimized                                     | 2 hours ago | 01:52    | Ubuntu - 14.04 | Travis #97 | 3       | 0       |
| #40 | master /<br>/home/travis/build/glomex/cypress_glomex/cypress/integration/test_player.js<br>travis.yml changed                                       | 2 hours ago | 01:53    | Ubuntu - 14.04 | Travis #96 | 3       | 0       |
| #39 | master /<br>/home/travis/build/glomex/cypress_glomex/cypress/integration/test_player.js<br>Tests fixed and implemented a new Command named "iframe" | a day ago   | 01:46    | Ubuntu - 14.04 | Travis #94 | 3       | 0       |
| #38 | master /<br>/home/travis/build/glomex/cypress_glomex/cypress/integration/test_player.js<br>Tests fixed and implemented a new Command named "iframe" | 2 days ago  | 01:44    | Ubuntu - 14.04 | Travis #93 | 3       | 0       |

Abbildung 5.5: Testausführungen im Projekt cypress\_glomex

**#42**

Passed 0 3 0

|                              |                                 |   |
|------------------------------|---------------------------------|---|
| Branch: master               | Started: Mar 5, 2018 at 02:04pm | CI: Travis #98  |
| Author: Dominic Schmitz      | Ended: Mar 5, 2018 at 02:06pm   | OS: Linux Ubuntu - 14.04  |
| Com... 10e8c66               | Durati... 01:52                 | Browser: Electron 59.0.3071.115   |
| Mess... travis.yml optimized |                                 | Spec: /home/travis/build/glomex/cypress_glomex/cypress/integration/test_player.js |
|                              |                                 | Cypress: v2.1.0   |

Output Failures Videos (1) Screenshots (6)

Abbildung 5.6: Testdetails

Um mehr Daten zur Ausführung eines Tests zu erhalten, kann der jeweilige Test angeklickt werden. Innerhalb des Tests werden genauere Daten wie etwa der auszuführende Branch, das Start- und Enddatum, die Länge der Testausführung, das Betriebssystem oder der Browser angezeigt (siehe Abbildung 5.6). Zusätzlich können die folgenden Reiter angeklickt werden:

- **Output**

In diesem Reiter wird die Ausgabe des Logs angezeigt (siehe Anhang 1).

- **Failures**

Sobald ein Fehler aufgetreten ist, wird er unter dem Reiter „Failures“ angezeigt (siehe Anhang 2).

- **Videos**

Wie in Zeile 18 im Codebeispiel 5.1.1 angegeben, wird ein Video des kompletten Testablaufs erstellt. Unter dem Reiter „Videos“ kann dieses Video angesehen werden (siehe Anhang 3).

- **Screenshots**

Wenn im Code mithilfe des `cy.screenshot()` Befehls ein Screenshot erstellt wurde, wird dieser im Reiter „Screenshots“ angezeigt (siehe Anhang 4).

Zusammenfassend zeigt sich, dass Cypress auch mit dem Dashboard, Fokus auf Nachvollziehbarkeit und Einfachheit legt. Es ist Cypress gelungen eine perfekte Ergänzung, zu Travis und dem Cypress Test Runner zu schaffen. Durch die Videoaufnahme und Sicherung der Screenshots kann jederzeit nachvollzogen werden, wo das Problem während der Ausführung aufgetreten ist. Somit können auch Probleme ohne jegliche Programmierkenntnisse ermittelt werden, wenn beispielsweise die Webseite nicht erreichbar ist oder das Video nicht abspielt. Das Dashboard ist momentan noch kostenfrei (Stand 05.03.2018). Eine kostenpflichtige Version für private Repositories ist in Planung.



# 6 Vergleich

---

Dieses Kapitel soll die wesentlichen Unterschiede des momentan bei der glomex verwendeten Systems Selenium in Verbindung mit Pytest und Python gegenüber Cypress ermitteln. Anzumerken ist hierbei, dass ausschließlich Webplayer Tests für diese Arbeit verwendet werden. Anschließend sollen diese Unterschiede übersichtlich und nachvollziehbar dargestellt werden.

## 6.1 Momentanes System

Wie bereits beschrieben, besteht das momentan eingesetzte System aus einer Mischung zwischen Selenium, Python und Pytest. Hierbei übernehmen die folgenden Komponenten die folgenden Aufgaben:

- **Pytest (siehe Kapitel 2.5.2)**

Mithilfe von Pytest werden die Tests nachvollziehbar geschrieben.

- **Selenium (siehe Kapitel 2.5)**

Selenium ermöglicht es auf Webelemente zuzugreifen und mit ihnen zu interagieren.

- **Python**

Python ist eine Programmiersprache und schafft somit die Logik des Tests (Klassen, Methoden, Schleifen etc.).

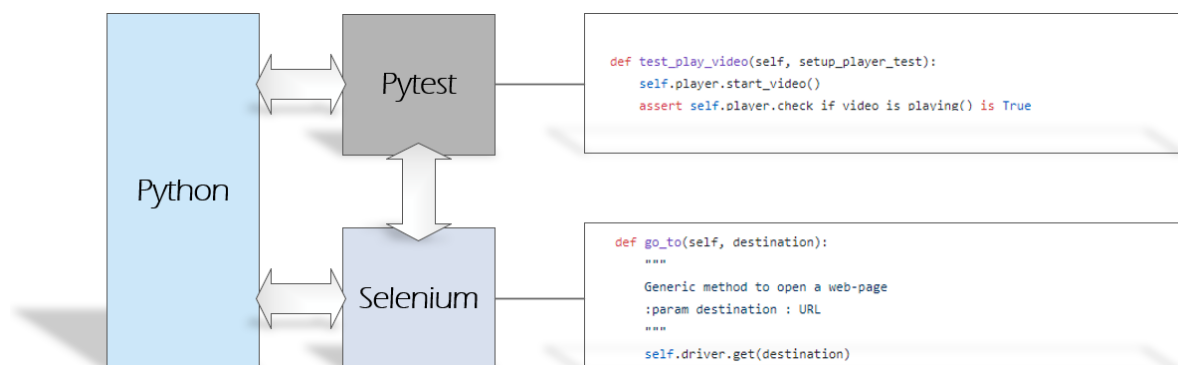


Abbildung 6.1: Test Umgebung

## 6.2 Konzept Vergleich

Beide Frameworks (Selenium und Cypress) verfolgen ein Konzept um Webanwendungen zu testen. Selenium interagiert mithilfe eines Treibers direkt mit dem Browser. Das setzt voraus, dass der jeweilige Treiber des Browsers installiert und konfiguriert ist. Es gibt eine Selenium IDE, mit deren Hilfe es möglich ist, einzelne Schritte zu automatisieren. Diese ist allerdings nur als Plugin für den Browser Mozilla Firefox erhältlich und eher für die Erstellung eines einfachen Skripts oder zur Unterstützung des explorativen Testens (siehe Kapitel 2.4.3) gedacht. Um allerdings lokale Tests zu schreiben, die mithilfe eines Servers automatisiert ausgeführt werden können, wird Selenium für eine bestimmte Programmiersprache benötigt [seleniumIDE].

Einen anderen Ansatz versucht Cypress zu realisieren. Cypress Tests werden zwar lokal mithilfe des Cypress Test Runners (siehe Kapitel 3.2) gestartet, aber direkt im Browser ausgeführt, was dazu führt, dass auch während der Testausführung auf die Funktionen des Browsers zugegriffen werden kann. Ein weiterer Vorteil ist der geringe Konfigurationsaufwand, da nicht erst ein Treiber installiert und konfiguriert werden muss [cypress].

Um herauszufinden, welches Konzept sich besser für den Einsatz der glomex eignet, wurden verschiedene Kriterien aufgestellt. Die Kriterien wurden aufgrund von Erfahrungen eines sechsmonatigen Praktikums innerhalb der QA-Abteilung und der Befragung von QA-Mitarbeitern ermittelt.

- **Installation**

Um Tests lokal zu entwickeln und auszuführen, wird eine Installation und Konfiguration des jeweiligen Frameworks benötigt. Besonders in Unternehmen oder Projekten, bei denen die Fluktuation sehr hoch ist, ist eine einfache, kurze und nachvollziehbare Installation von Relevanz.

- **Abhängigkeiten**

Jeder Test, der mithilfe eines der Frameworks entwickelt wird, muss ausführbar sein. Die Frage bei diesem Kriterium ist, in wie fern das eingesetzte Framework abhängig von weiteren Frameworks, Programmiersprachen und Tools ist, bis sich dieses ausführen lässt. Desto mehr Software vorhanden sein muss, desto höher ist die Komplexität. Das kann dazu führen, dass die Ursachen entstandener Fehler aufgrund des Zusammenspiels mehrerer Komponenten nur schwer zu identifizieren sind.

- **Ausführung während der Testerstellung**

Während der Testerstellung wird ein Test mehrfach wiederholt. Hier stellt sich die Frage, ob das verwendete Framework in Echtzeit Änderungen im Code erkennt und anzeigt, oder bei jeder Änderung der Test nur über Umwege ausgeführt und überprüft werden kann.

- **Kompatibilität**

Eines der wichtigsten Kriterien ist die Kompatibilität, da Unternehmen wie die glomex darauf angewiesen sind, auf so vielen Plattformen wie Möglich zu testen. Nur so kann gewährleistet werden, dass Software weitestgehend fehlerfrei beim Anwender ausgeliefert wird. Im Rahmen dieser Arbeit spielt dabei das sogenannte Cross-Browser-Testing eine Rolle. Cross-Browser-Testing testet eingebetteten Inhalt wie etwa einen Webplayer auf mehreren Browsern, um zu gewährleisten, dass dieser sich unabhängig vom Browsertyp identisch verhält [**crossBrowser**].

- **iFrame Support**

Ein weiterer Punkt ist der Support von sogenannten „iFrames“, auch „Inlineframes“ genannt. Ein iFrame dient der Strukturierung von Webseiten und stellt weitere Webinhalte als eigenständige Dokumente innerhalb eines definierten Bereichs eines Browsers dar [**iFrame**]. Da alle Produkte der glomex iFrames zur Darstellung benutzen, ist der Support des Frameworks entsprechend wichtig.

Jedes der beiden Frameworks wird nun auf das jeweilige Kriterium geprüft und verglichen. Dabei erhalten beide Frameworks jeweils eine Punktzahl zwischen 1 (schlecht) bis 3 (gut). Mithilfe dieser vergebenen Punkte wird bei der Auswertung eine Gesamtpunktzahl ermittelt. Mit diesen ist es möglich, dass für die glomex geeignetere Framework zu ermitteln.

- **Installation**

Wie bereits in Kapitel 3 beschrieben ist es möglich Cypress auf allen gängigen Betriebssystemen wie Linux, Mac OS und Windows zu installieren. Cypress bringt mit der Installation alles mit, um Tests zu erstellen und auszuführen. Zusätzlich benötigt die Installation kaum Vorwissen und ist auf der Seite [**cypress**] nachvollziehbar erläutert. Die Installation des Cypress Test Runners ist mit wenig Aufwand nach kurzer Zeit abgeschlossen. Ein weiterer positiver Aspekt ist, Cypress warnt sobald eine neuere Version verfügbar ist und lässt sich mit nur wenig Aufwand updaten (siehe Kapitel 3.2.3).

Selenium unterstützt wie auch Cypress alle gängigen Betriebssysteme, beinhaltet allerdings nicht alle benötigten Komponenten um Tests zu entwickeln. Um Selenium zu nutzen, muss vorher festgelegt werden, in welcher Programmiersprache die Tests später entwickelt werden. Ist dies geschehen, muss die benötigte Programmiersprache, das dazu passende Selenium Paket und der Treiber installiert werden. Anschließend muss der Treiber konfiguriert werden [**seleniumInstallation**].

Der Vergleich zeigt deutlich, dass Cypress großen Wert auf eine einheitliche, einfache und komplette Installation legt und erhält daher 3 Punkte. Bei der Installation von Selenium gestaltet sich die Installation im Vergleich deutlich aufwendiger, da dieser die Installation und Konfiguration mehrerer Komponenten zugrunde liegt, was nur zu einer Bewertung von einem Punkt geführt hat.

- Cypress **3 Punkte**
- Selenium **1 Punkt**

- **Abhängigkeiten**

Während der Installation wird erläutert, dass Cypress bereits alles an Software mitbringt, um Tests auszuführen. Somit werden alle Abhängigkeiten durch die Installation beseitigt. Um allerdings Tests komfortabel zu erstellen, wird eine entsprechende IDE wie etwa Visual Studio Code angeraten (siehe Kapitel 3.3).

Mindestens eine Programmiersprache wird dagegen von Selenium benötigt. In der Realität ist das allerdings nicht die einzige Abhängigkeit. Selenium alleine führt nur Befehle aus, die an den Browser geschickt werden, um den Browser zu steuern [**selenium**]. Aus diesem Grund muss ein entsprechender Treiber für jeden Browser, der genutzt werden will, vorhanden und konfiguriert sein. Zusätzlich werden in der Praxis häufig zusätzlich BDD-Frameworks verwendet (siehe Kapitel 2.5.1), um die Nachvollziehbarkeit eines Tests zu verbessern.

Bezüglich der Abhängigkeiten hat Cypress einige Vorteile gegenüber Selenium, da die Hersteller darauf geachtet haben alles an Software das zur Ausführung von Tests benötigt wird, direkt mit zu installieren. Da Cypress so gesehen keine Abhängigkeiten aufweist, erhält das Framework 3 Punkte. Selenium hingegen erhält nur einen Punkt, da alleine ohne Konfiguration und Installation weiterer Programmiersprachen und Treiber keine Tests entwickelt werden können.

- Cypress **3 Punkte**
- Selenium **1 Punkt**

- **Ausführung während der Testerstellung**

Während der Testerstellung ermöglicht der Cypress Test Runner einen parallelen Testablauf im Browser. So wird nach jeder Änderung am Quellcode direkt der Test erneut ausgeführt und der Entwickler erhält Feedback, ob die Änderung die gewünschte Wirkung erzielt hat. Da zusätzlich der Cypress Test Runner im Browser läuft, hat der Entwickler Zugriff auf alle Entwicklerfunktionen die Seitens des Browser zur Verfügung gestellt werden.

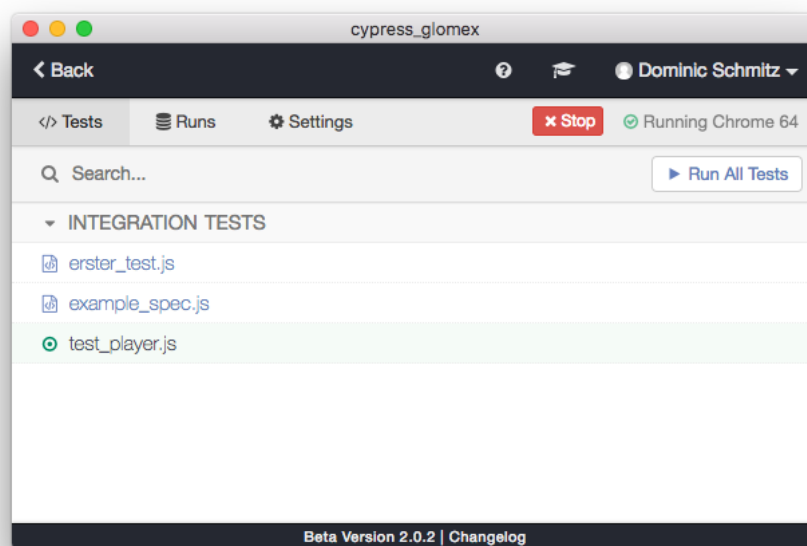


Abbildung 6.2: Cypress Test Runner während der Ausführung des Tests „test\_player.js“

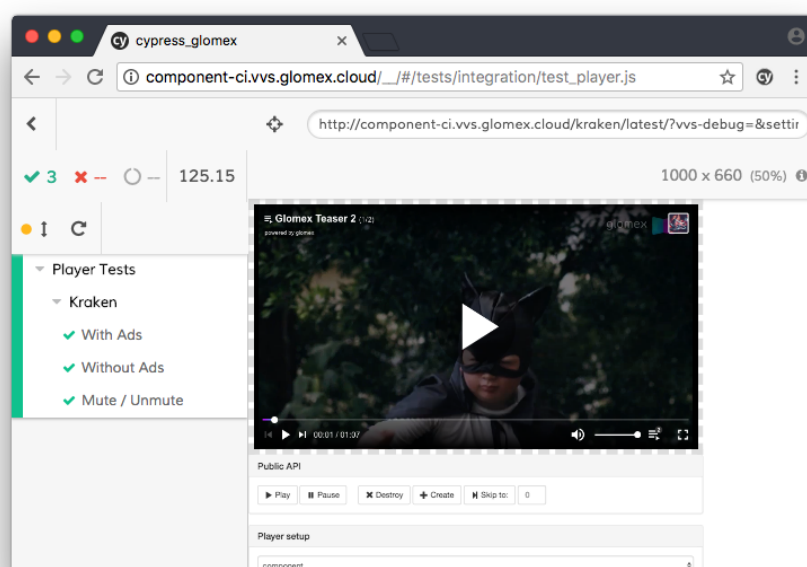


Abbildung 6.3: Testausführung des Tests „test\_player.js“ im Browser Google Chrome

Aus dem Grund, das Selenium alleine nicht ausgeführt werden kann, wird das momentane System als Referenz verwendet (siehe Kapitel 6.1). Bei dem momentanen System wird Selenium mithilfe von Pytest in Verbindung mit der Programmiersprache Python ausgeführt. Pytest erkennt keine Änderungen im Code, was dazu führt, dass jeder Test manuell nach jeder Änderung neu gestartet werden muss. Das kann dazu führen, dass der Entwickler Fehler übersieht, da ihm diese nicht direkt angezeigt werden. Da Selenium nur Befehle an den Browser schickt, die anschließend ausgeführt werden, muss weiterhin eine zusätzliche Browserinstanz geöffnet bleiben, um über die Entwicklerkonsole neue Webelemente zu identifizieren.

Aufgrund der Komfortabilität, Lauffähigkeit im Browser und aktiven Testausführung während der Entwicklung erhält Cypress drei Punkte. Da im Rahmen dieser Arbeit nur das momentane System betrachtet wird und es sehr viele verschiedene Kombinationen der Frameworks gibt, ist nicht ausgeschlossen, dass eine Frameworkkombination (inklusive Selenium) mit einer hohen Komfortabilität und aktiven Testausführung existiert. Selenium allerdings kann nicht im Browser ausgeführt werden, was zu einer Punktzahl von zwei Punkten führt.

- Cypress **3 Punkte**
- Selenium **2 Punkte**

#### • iFrame Support

Seitens Cypress gibt es keinen vollen iFrame-Support (stand 27.02.2018). Cypress arbeitet bereits an dem Problem. Intern trägt es die Problemnummer 136 (englisch Issue #136). In der Offiziellen Dokumentation (zu finden unter [[cypressIssue136](#)]) wird folgendes geschrieben:

##### Zitat aus [[cypressIssue136](#)]

You cannot target elements or interact with anything in an iframe - regardless of it being a same domain or cross domain iframe.

This is actively being worked on in Cypress and you'll first see support for same domain iframes, followed by cross domain (they are much harder to do).

##### **Workaround:**

Sit tight, comment on the issue so we know you care about this support, and be patient.

Da die Produkte der glomex iFrames benutzen, muss eine Lösung für dieses Problem gefunden werden. Als Workaround wird empfohlen sich direkt über den Github Thread [cypressGithubIssue136] an die Entwickler zu wenden. Innerhalb des Threads werden bereits einige Lösungsvorschläge bereitgestellt, daher ist eine Kontaktaufnahme nicht nötig.

Der in dieser Arbeit verwendete Workaround sieht vor, einen neuen Befehl zu erstellen (siehe 4.1, Support). Der erstellte Befehl soll auf ein übergebenes iFrame zugreifen und es als eine Referenz zurück geben, die dann innerhalb eines Alias gespeichert wird. Ein Alias beschreibt in dieser Arbeit, einen Namen für eine Referenz auf ein Objekt.

```

Cypress.Commands.add('iframe',
  (locator, element, wait=10000) => {

    return cy.get('iframe', { timeout: wait })
5      .should('have.class', locator)
      .should(($iframe) => {
        expect($iframe.contents().find(element)).to.exist
      }).then(($iframe) => {
10        return cy.wrap($iframe.contents().find("body"))
      });
  })

```

Codebeispiel 6.2.1: Eigener Befehl „iframe“

Wie in Zeile zwei im Codebeispiel 6.2.1 dargestellt, werden die folgenden drei Parameter an den Befehl „iframe“ übergeben:

– **locator**

Der Locator der das iFrame Element eindeutig adressiert. Ein Beispiel hierfür könnte der Klassenname des iFrames sein.

– **element**

Ein Element innerhalb des iFrames, um zu gewährleisten, dass auf Elemente innerhalb zugegriffen werden kann.

– **wait**

Die Zeit, wie lange gewartet wird, bis das iFrame Element gefunden werden muss. Standard hier sind 10000 Millisekunden, was 10 Sekunden entspricht.

Weiterhin ab Zeile vier im Codebeispiel 6.2.1 ist zu sehen, dass mithilfe des „get“-Befehls auf das iFrame zugegriffen wird. Über „should“ wird zusätzlich geprüft, ob es sich hierbei um das übergebene iFrame handelt. Seit Version 0.20.0 kann mit iFrames über den „wrap“-Befehl (siehe Codebeispiel 4.3.11) interagiert werden [[cypressChangelog](#)]. Vollständiger Zugriff, damit Cypress innerhalb eines iFrames ausgeführt werden kann, ist noch nicht möglich, wird aber in Aussicht gestellt (Stand 27.02.2018). Der eigentliche Workaround findet in der Zeile sieben (siehe Codebeispiel 6.2.1) statt. Hier wird mit dem wrap-Befehl auf den Content (body) des iFrames zugegriffen und anschließend zurück gegeben.

```

describe('Player Test', function() {

  beforeEach(function() {
    // Fixtures aus anderen Dateien laden
5    cy.fixture('kraken/locators.json').as('locator')
    cy.fixture('kraken/web.json').as('web')
    cy.fixture('kraken/settings.json').as('setting')
  })

10  context('Kraken', function() {

    it('With Ads', function() {

      // Aufruf der Kraken-Webseite
15    cy.visit(this.web.krakenUrl)

      // iFrame mit dem Alias "player" (siehe .as(player))
    cy.iframe(this.locator.playerIframe, "#player")
      .as('player')

20    // Rufe den Alias mithilfe von ".get" auf,
      finde das Webelement und klicke auf Play
    cy.get("@player").find(this.locator.playButton)
      .click()

25  })
  })
})

```

Codebeispiel 6.2.2: erster\_test.js, Quellcode



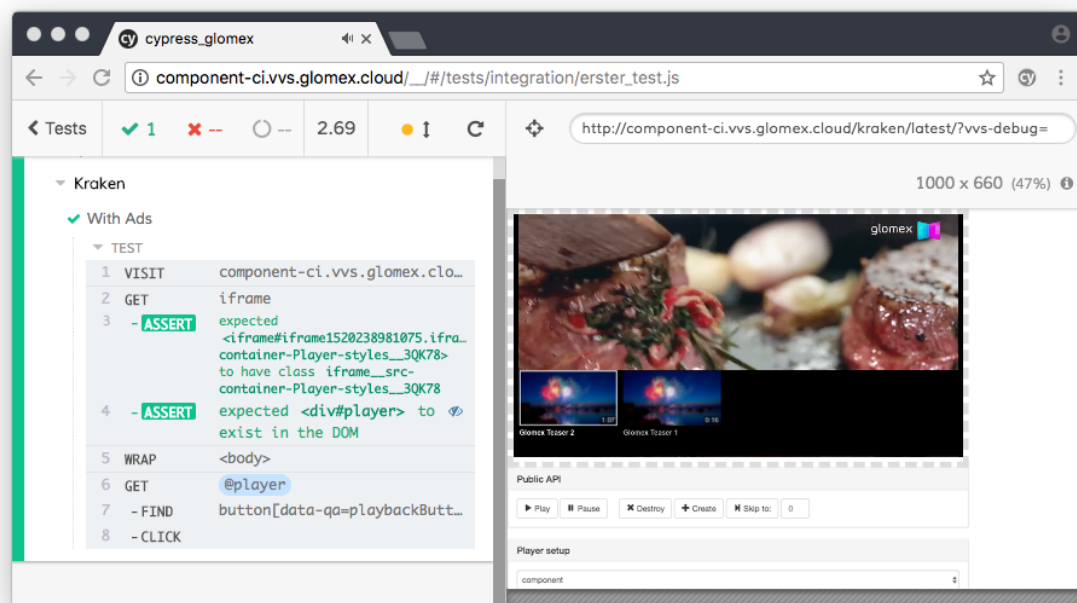


Abbildung 6.4: erster\_test.js, Testausführung im Browser mithilfe des Cypress Test Runners

Codebeispiel 6.2.2 zeigt, wie der im Codebeispiel 6.2.1 erstellte Befehl angewendet wird. Erst wird mithilfe des „visit“-Befehls in Zeile 20 auf die Webseite zugegriffen. In Zeile 23 wird der Befehl „iframe“ aufgerufen. Während des Aufrufs wird der Locator (`this.locator.playerIframe`) und ein Webelement, das sich innerhalb des iFrames befindet (`#player`) an den Befehl übergeben. Dabei ist anzumerken, dass innerhalb eines css Selektors eine Raute auf eine id hinweist. So bedeutet in unserem Beispiel `#player`, das der Test nach dem Webelement mit der id „player“ auf der aufgerufenen Webseite suchen soll. Anschließend wird mit dem Befehl „as“ ein Alias erstellt. Dieser Alias kann nun weiterhin verwendet werden und repräsentiert das komplette iFrame des Players. Zu erwähnen ist, dass der Alias bis zum ende des it-Blocks bestehen bleibt.

In Zeile 27 des Codebeispiels 6.2.2 ist sehr gut zu erkennen, dass mithilfe des „@“ Zeichens der Alias, an den `get`-Befehl übergeben wird. Folgend kann durch den `find`-Befehl ein Webelement innerhalb des iFrames gesucht und geklickt (`.click()`) werden.

Da es einen Workaround für Cypress gibt mit dessen Hilfe es möglich ist auf iFrames zuzugreifen, erhält Cypress für dieses Kriterium 2 Punkte. Selenium dagegen unterstützt iFrames vollständig. Der Befehl „`switchTo()`“ erlaubt es dem Selenium Treiber, direkt das iFrame aufzurufen und mit ihm zu interagieren. Aus diesem Grund erhält Selenium 3 Punkte.

- Cypress **2 Punkte**
- Selenium **3 Punkte**

- **Kompatibilität**

Das beschriebene Cross-Browser-Testing ist mit Cypress momentan nicht möglich (siehe Offiziellen Post [[cypressGithubIssue310](#)]), da Cypress nur den Browser „Google Chrome“ (Chrome, Chromium und Canary) unterstützt. Die Entwickler stellen aber die Verwendung weiterer Browser in Aussicht (siehe [[cypressGithubIssue310](#)]) für zukünftige Versionen. Zum momentanen Zeitpunkt (Stand 27.02.2018) ist dies allerdings nicht möglich.

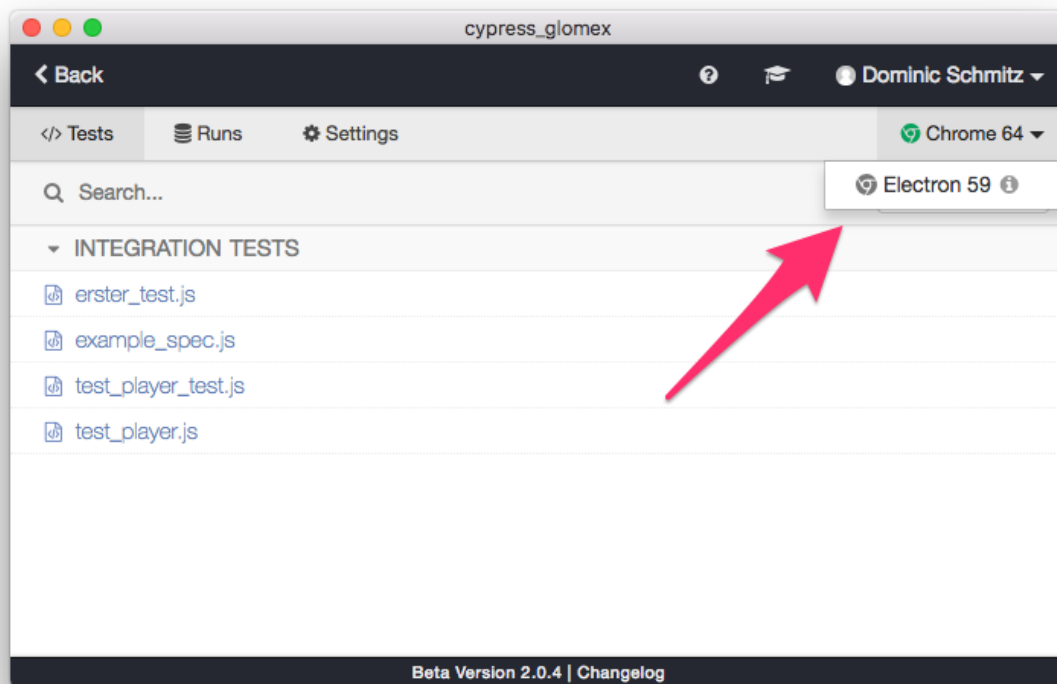


Abbildung 6.5: Unterstützte Browser im Cypress Test Runner (Stand 27.02.2018 V2.0.4)

Offiziell unterstützt dagegen Selenium alle gängigen Browser wie „Firefox“, „Microsoft Internet Explorer“, „Safari“, „Opera“ und „Google Chrome“.

Um ein Webelement anzusprechen, werden bei beiden Frameworks sogenannte „Locators“ verwendet. Ein Locator beschreibt ein HTML Element, beispielsweise mithilfe einer id, eines Klassennamens, weiterer Attribute oder eines Xpaths um dem Framework exakt zu beschreiben, welches Webelement einer Webseite verwendet werden soll. Cypress unterstützt derzeit (Stand 22.02.2018) nur css Selektoren um ein Webelement zu beschreiben [[cypress](#)].

Verschiedene Arten unterstützt dagegen Selenium. Laut [seleniumLocators] ist es möglich über folgende Arten ein Webelement zu erreichen:

- id
- Name
- Linktext
- Partial Linktext
- Tag Name
- Class Name
- Css
- Xpath

Die genaue Beschreibung jeder Möglichkeit kann auf [seleniumLocators] nachgelesen werden.

Das Cypress noch relativ neu im Vergleich zu Selenium ist, wird deutlich im Vergleich der Kompatibilität. Besonders da Cross-Browser-Testing seitens Cypress aktuell (Stand 22.02.2018) nicht supported wird, ist es schwer für Unternehmen ihre Anwendungen plattformübergreifend zu Testen. Das ist besonders problematisch, da diese nicht garantieren können das die Anwendung fehlerfrei beim Anwender funktioniert. Aufgrund dessen erhält Cypress im Vergleich nur einen Punkt. Selenium dagegen supported alle gängigen Browser und Locator-Arten, was zu einem Ergebnis von drei Punkten führt.

- Cypress **1 Punkt**
- Selenium **3 Punkte**

## 6.2.1 Auswertung

Nachdem alle Kriterien aufgestellt und analysiert wurden, muss nun ausgewertet werden, welches der beiden Frameworks sich besser für den Einsatz bei der glomex eignet. Um dies zu ermitteln, wird eine Tabelle mit allen Punkten je Kriterium dargestellt. Die Kriterien Kompatibilität und iFrame Support werden jeweils doppelt gewichtet, da alle Produkte der glomex auf diversen Browsern getestet werden und iFrames benutzen.

| Kriterium                             | Gewichtung     | Selenium         | Cypress          |
|---------------------------------------|----------------|------------------|------------------|
| Installation                          | einfach        | 1                | 3                |
| Abhängigkeiten                        | einfach        | 1                | 3                |
| Ausführung während der Testerstellung | einfach        | 2                | 3                |
| <b>Kompatibilität</b>                 | <b>doppelt</b> | <b>6</b>         | <b>4</b>         |
| <b>iFrame Support</b>                 | <b>doppelt</b> | <b>6</b>         | <b>2</b>         |
| Gesamt                                |                | <u><b>16</b></u> | <u><b>15</b></u> |

Tabelle 6.1: Framework Auswertung anhand Kriterien

Wie in Tabelle 6.1 dargestellt, schneidet Selenium in den Punkten Installation, Abhängigkeiten und Ausführung während der Testerstellung schlecht bis mittelmäßig ab. Das liegt vor allem daran, dass die Installation von Selenium mit diversem Konfigurationsaufwand und der Installation weiterer Frameworks und Programmiersprachen verbunden ist. Ohne eine weitere Programmiersprache kann Selenium nur als Plugin für Mozilla Firefox (siehe Selenium IDE [[seleniumIDE](#)]) genutzt werden. iFrames unterstützt Selenium nativ mithilfe des „switchTo()“ Befehls. Mit ihm ist es möglich direkt den Fokus des Treibers auf das iFrame zu legen. Selenium funktioniert mit allen gängigen Browsern, was sehr von Vorteil ist, da alle Produkte der glomex browserunabhängig funktionieren sollen.

Cypress wird ständig weiterentwickelt und ist im Gegensatz zu Selenium noch relativ frisch auf dem Markt (Stand 01.03.2018). Die Entwickler legen großen Wert auf Komfortabilität und Übersichtlichkeit, was sich in der Struktur und Einfachheit der Installation widerspiegelt. Entwickler können während der Erstellung und Bearbeitung der Tests genau sehen und nachvollziehen, ob ein Test funktioniert oder nicht, da Cypress nach jeder Änderung den jeweiligen Test erneut ausführt. Der iFrame Support wurde bereits teilweise umgesetzt (siehe Codebeispiel 6.2.1). Leider fehlt noch eine native Lösung im Bezug auf iFrames, um es Cypress direkt zu ermöglichen in das iFrame zu wechseln und nicht nur mit ihm zu interagieren [[cypressGithubIssue136](#)]. Ein wichtiger Punkt ist zusätzlich die Kompatibilität. Stand 27.02.2018 Version 2.0.4 ist es nicht möglich einen anderen Browser als Google Chrome für Tests zu verwenden [[cypressGithubIssue310](#)]. Es wurde bereits bestätigt das Cross-Browser-Testing zukünftig implementiert wird.

Aufgrund der beschriebenen Defizite seitens Cypress ist (Stand 01.03.2018) das momentane System mit 16 (Selenium) gegenüber 15 (Cypress) Punkten besser für den Einsatz bei der glomex geeignet (siehe Tabelle 6.1). Werden in zukünftigen Versionen allerdings die Mängel in Richtung Kompatibilität und iFrame Support behoben, wird Cypress zur deutlich besseren und komfortableren Wahl.

# 7 Fazit

---

## 7.1 Reflexion

Um Webanwendungen grafisch im Browser zu testen, führte bisher kaum ein Weg an Selenium vorbei. Da das Framework in Verbindung mit bekannten Programmiersprachen wie Java, C#, Ruby, Python und JavaScript arbeitet, lässt es sich nahtlos in viele Unternehmensumgebungen einfügen. Wer jedoch bisher eine richtige Alternative gesucht hat, hat diese vergeblich gesucht.

Cypress verspricht ein Gesamtpaket, zur Erstellung von Oberflächentests um Webanwendungen graphisch im Browser zu testen. Dabei macht Cypress vieles richtig. Selenium interagiert mit dem Browser durch das senden und empfangen von Daten über einen Webtreiber. Cypress verfolgt dagegen ein anderes Konzept, indem es direkt im Browser als Anwendung ausgeführt wird. Das bietet den Vorteil das Testentwickler während der Testausführung, Zugriff auf alle Entwicklerfunktionen des Browsers haben. So kann an Stellen die Probleme verursachen, direkt eine Fehlerbehebung stattfinden.

Zusätzlich verspricht Cypress, das Erstellen, Schreiben, Ausführen und Debuggen von Tests zu vereinfachen. In dieser Arbeit hat sich diese Behauptung als wahr herausgestellt, denn besonders in den Punkten Installation, Abhängigkeiten und der Ausführung während der Testerstellung, kann Cypress punkten. Die Installation ist unaufwendig und bringt bereits alles mit, um erfolgreich Tests zu erstellen, schreiben, ausführen und debuggen zu können. Zusätzlich wird nur ein Editor zum Schreiben des Quellcodes benötigt. Selenium hingegen ist schon während der Installation sehr aufwendig, da erst mehrere Komponenten installiert und konfiguriert werden müssen. Ohne diese Komponenten wie etwa der Programmiersprache, ist es mit Selenium nicht möglich Tests zu entwickeln. Da Selenium nur Daten an den Browser schickt, lassen sich während der Testausführung nicht die Entwicklerfunktionen des Browsers aufrufen, um aktiv Fehler zu identifizieren.

Doch Selenium hat auch Vorteile. Es unterstützt iFrames nativ, was besonders wichtig für Unternehmen wie die glomex ist, da deren Produkte ausschließlich iFrames nutzen. Cypress hingegen stellt zwar einen Workaround bereit, unterstützt aber nicht nativ iFrames. Das führt dazu, das Entwickler Lösungen wie etwa das Erstellen eines neuen Kommandos, finden müssen um überhaupt mit iFrames zu interagieren. Ein weiterer sehr wichtiger Aspekt spielt das Cross-Browser-Testing. Selenium unterstützt bereits alle gängigen Browser wie etwa Firefox, Microsoft Internet Explorer, Safari, Opera und Google Chrome. Cypress dagegen supportet zum momentanen Zeitpunkt (07.03.2018) nur Google Chrome.

In Anbetracht der genannten Aspekte hat sich herausgestellt, dass Selenium momentan immer noch die bessere Wahl für Unternehmen wie die glomex sind. Das liegt vor allem daran, welchen Wert, welches Unternehmen auf bestimmte Kriterien legt. Für die glomex ist es von Bedeutung, dass das Framework iFrames unterstützt und eine Cross-Browser-Funktionalität besitzt, da alle Produkte im Unternehmen iFrames verwenden und in allen gängigen Browsern gleich dargestellt werden sollen.

## 7.2 Ausblick

Innerhalb der letzten vier Monate (Stand 07.03.2018) ist Cypress von Version 1.1.2 auf Version 2.1.0 gestiegen. Das zeigt, dass das Framework stetig weiterentwickelt wird. Probleme, Bugs und Feature Anfragen können direkt über Github an die Entwickler gemeldet werden. So entwickelt sich Cypress laufend weiter. Das Problem mit der fehlenden iFrame Unterstützung wurde bereits gemeldet und soll in einer späteren Version behoben werden. Die Entwickler haben zudem bereits bestätigt, dass die fehlende Cross-Browser-Funktionalität in einer zukünftigen Version implementiert wird. Zusätzlich wird sogar eine Kompatibilität mit Mobilgeräten angestrebt.

Für das Unternehmen glomex könnte so in Zukunft Cypress eine echte Alternative zu Selenium werden. Denn durch die stetige Weiterentwicklung und die Lösung mithilfe von Travis, Tests zu automatisieren und eine detailliertere Ablaufdarstellung über das Cypress Dashboard anzubieten, bietet Cypress eine Komplettlösung für Unternehmen. So könnte nicht nur für die glomex, Cypress zukünftig eine Alternative sein, sondern auch für Unternehmen, die nach einer entwicklerfreundlichen und übersichtlichen Alternative zu Selenium suchen.

# Allgemeine Ergänzungen

Output

Failures

Videos (1)

Screenshots (6)

/home/travis/build/glomex/cypress\_glomex/cypress/integration/test\_player.js

Electron 59.0.3071.115Linux Ubuntu - 14.04

Started video recording: /home/travis/build/glomex/cypress\_glomex/cypress/videos/uoi08.mp4

(Tests Starting)

Player Tests

Kraken

✓ With Ads (91610ms)

✓ Without Ads (10477ms)

✓ Mute / Unmute (6520ms)

3 passing (2m)

(Tests Finished)

- Tests: 3

- Passes: 3

- Failures: 0

- Pending: 0

- Duration: 1 minute, 52 seconds

- Screenshots: 6

- Video Recorded: true

- Cypress Version: 2.1.0

(Screenshots)

- /home/travis/build/glomex/cypress\_glomex/cypress/screenshots/Preroll.png (1280x720)

- /home/travis/build/glomex/cypress\_glomex/cypress/screenshots/Paused.png (1280x720)

- /home/travis/build/glomex/cypress\_glomex/cypress/screenshots/Midroll.png (1280x720)

- /home/travis/build/glomex/cypress\_glomex/cypress/screenshots/Postroll.png (1280x720)

- /home/travis/build/glomex/cypress\_glomex/cypress/screenshots/Muted.png (1280x720)

- /home/travis/build/glomex/cypress\_glomex/cypress/screenshots/Unmuted.png (1280x720)

(Video)

- Started processing: Compressing to 32 CRF

- Finished processing: /home/travis/build/glomex/cypress\_glomex/cypress/videos/uoi08.mp4 (40 seconds)

(Uploading Assets)

- Done Uploading (1/7) /home/travis/build/glomex/cypress\_glomex/cypress/screenshots/Unmuted.png

- Done Uploading (2/7) /home/travis/build/glomex/cypress\_glomex/cypress/screenshots/Midroll.png

- Done Uploading (3/7) /home/travis/build/glomex/cypress\_glomex/cypress/videos/uoi08.mp4

- Done Uploading (4/7) /home/travis/build/glomex/cypress\_glomex/cypress/screenshots/Preroll.png

- Done Uploading (5/7) /home/travis/build/glomex/cypress\_glomex/cypress/screenshots/Muted.png

- Done Uploading (6/7) /home/travis/build/glomex/cypress\_glomex/cypress/screenshots/Postroll.png

- Done Uploading (7/7) /home/travis/build/glomex/cypress\_glomex/cypress/screenshots/Paused.png

(All Done)

Abbildung 1: Anhang Testdetail Output

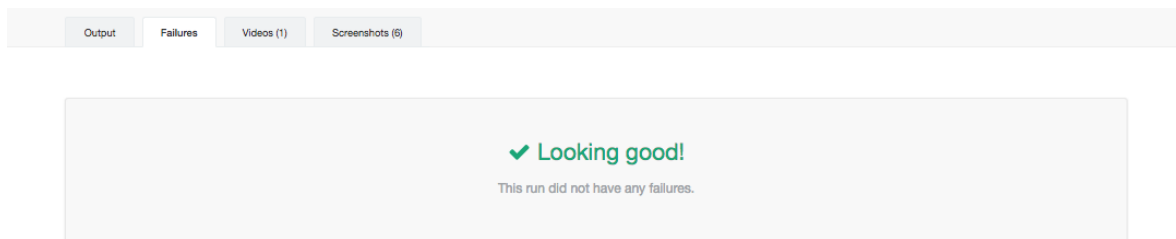


Abbildung 2: Anhang Testdetail Failures

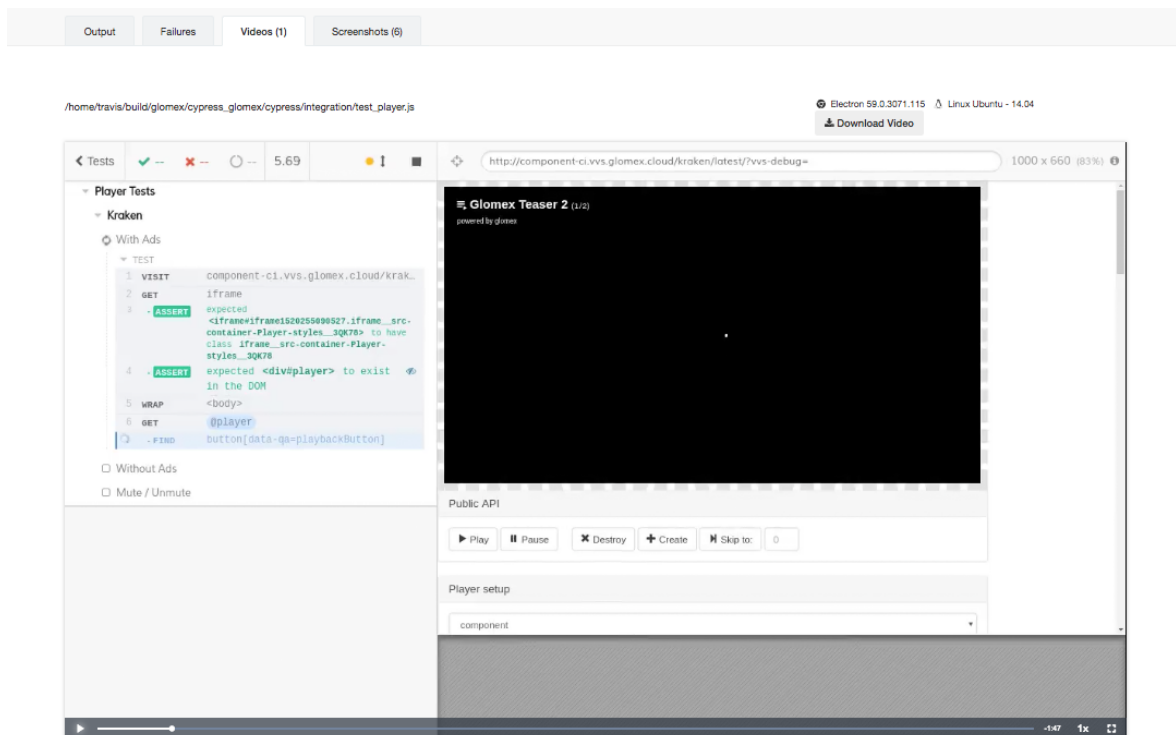


Abbildung 3: Anhang Testdetail Videos

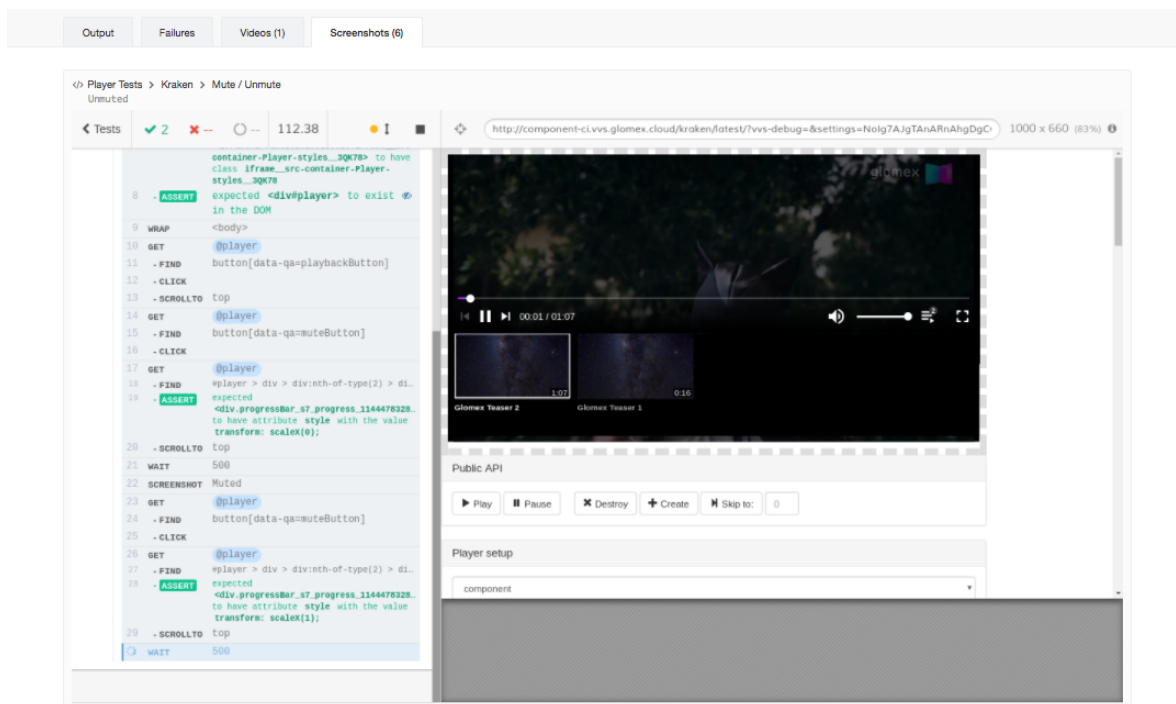


Abbildung 4: Anhang Testdetail Screenshots



```

describe('Player Tests', function() {

  /**
   * Get all the pre-stuff before every Test
5  */
  beforeEach(function(){
    cy.fixture('kraken/locators.json').as('locator')
    cy.fixture('kraken/web.json').as('web')
    cy.fixture('kraken/settings.json').as('setting')
10  })

  context('Kraken', function(){

15    /**
     * Krakentest with Ads
     */
    it('With Ads', function(){

20      const loc = this.locator
      const time = 10000

      // Visit the Website
      cy.visit(this.web.krakenUrl)

25      // find the iFrame
      cy.iframe(loc.playerIframe, "#player")
        .as('player')

      // play the Video
30      cy.get("@player")
        .find(loc.playButton, {timeout: time})
        .click()

      // scroll to top
35      cy.scrollTo('top')

      // is the Ad displaying?

```

```

40    cy.get("@player").find(loc.adDisclaimer, {timeout: time})
    cy.wait(2000).screenshot('Preroll')

    // is the Video resuming after ad?
    cy.get("@player").find(loc.pauseButton, {timeout: time})

45    // Test if the Video is playing
    cy.get("@player").find(loc.videoTimePassed)
        .contains("00:05", {timeout: 10000})

    // Test if pause is working
50    cy.get("@player").find(loc.pauseButton).click()
    cy.get("@player").find(loc.playButtonBig)
        .should('be.visible')
    cy.scrollTo('top')
    cy.wait(1000).screenshot('Paused')
55    cy.get("@player").find(loc.pauseButton).click()
    cy.scrollTo('top')

    // Test if Midroll is playing
    cy.get("@player")
60        .find(loc.adDisclaimer, {timeout: 50000})
    cy.wait(2000).screenshot('Midroll')
    cy.get("@player")
        .find(loc.pauseButton, {timeout: time})

65    // Test if Preroll is playing
    cy.get("@player")
        .find(loc.adDisclaimer, {timeout: 50000})
    cy.wait(2000).screenshot('Postroll')
    })

70    /**
    * Krakentest without Ads
    */
    it('Without Ads', function(){
75
        const loc = this.locator
        const time = 10000

```

```

80      // Visit the Website and deactivate Hornet
      cy.visit(this.web.krakenUrl)
      cy.de_activate_hornet()

      // Find the iFrame
      cy.iframe(loc.playerIframe, "#player").as('player')
85
      // Play the Video
      cy.get("@player").find(loc.playButton).click()

      // Scroll to Top
90      cy.scrollTo('top')

      // Test if the Video is playing
      cy.get("@player").find(loc.videoTimePassed)
        .contains("00:05", {timeout: time})
95
      // Test if pause is working
      cy.get("@player").find(loc.pauseButton).click()
      cy.get("@player").find(loc.playButtonBig)
        .should('be.visible')
100      cy.scrollTo('top') // scroll to top
    })

    /**
    * Krakentest without Ads and Muted / Unmuted
105    */
    it('Mute / Unmute', function() {

      const loc = this.locator
      const time = 5000
110

      // Visit the Website and deactivate Hornet
      cy.visit(this.web.krakenUrl)
      cy.de_activate_hornet()

      // Find the iFrame
115      cy.iframe(loc.playerIframe, "#player").as('player')

```

```

120 // Play the Video
    cy.get("@player").find(loc.playButton).click()

    // Scroll to Top
    cy.scrollTo('top')

    // Test if the Mute Button works
125 cy.get("@player").find(loc.volumeButton).click()
    cy.get("@player").find(loc.volumeBar)
        .should('have.attr', 'style', 'transform: scaleX(0);')
    cy.scrollTo('top') // scroll to top
    cy.wait(500).screenshot('Muted')
130

    // Test if the Unmute Button works
    cy.get("@player").find(loc.volumeButton).click()
    cy.get("@player").find(loc.volumeBar)
        .should('have.attr', 'style', 'transform: scaleX(1);')
135 cy.scrollTo('top') // scroll to top
    cy.wait(500).screenshot('Unmuted')
  })
})
})

```

Codebeispiel .1: test\_player.js, läuft derzeit (07.03.2018) aktiv auf dem Travis

```

{
  "playerIframe": "iframe__src-container-Player-styles__3QK78",
  "playButton": "button[data-qa=playbackButton]",
  "pauseButton": "button[data-qa=playbackButton]",
5  "playlistButton": "button[data-qa=playlistButton]",
  "pauseButton": "button[data-qa=playbackButton]",
  "fullscreenButton": "button[data-qa=fullscreenButton]",
  "volumeButton": "button[data-qa=muteButton]",
  "adDisclaimer": "#player > div > div:nth-of-type(2)
10   > div:nth-of-type(2) > div > div > div > div
   > div:nth-of-type(5) > div > div:nth-of-type(1)
   > div:nth-of-type(1) > div > div > strong",
  "videoTimePassed": "#player > div > div:nth-of-type(2)
   > div:nth-of-type(2) > div > div > div > div
15   > div:nth-of-type(5) > div > div:nth-of-type(2)
   > div:nth-of-type(1) > div > span",
  "playButtonBig": "#player > div > div:nth-of-type(2)
   > div:nth-of-type(2) > div > div > div > div
   > div:nth-of-type(4) > div:nth-of-type(2) > div
20   > div > button"
  "volumeBar": "#player > div > div:nth-of-type(2)
   > div:nth-of-type(2) > div > div > div > div
   > div:nth-of-type(5) > div > div:nth-of-type(2)
   > div:nth-of-type(3) > div:nth-of-type(1) > div
25   > div > div:nth-of-type(1)"
}

```

Codebeispiel .2: fixtures/kraken/locators.json

```

{
  "delay": 20000
}

```

Codebeispiel .3: fixtures/kraken/settings.json

```
{  
  "krakenUrl": "LINK ZUR KRAKEN TESTPAGE"  
}
```

Codebeispiel .4: fixtures/kraken/web.json

```
Cypress.Commands.add('de_activate_hornet', () => {  
  
  cy.get('#hornet-input').click()  
  cy.scrollTo('top')  
5  cy.reload()  
  
  })  
  
Cypress.Commands.add('iframe',  
10 (locator, element, wait=10000) => {  
  
  return cy.get('iframe', { timeout: wait })  
    .should('have.class', locator)  
    .should($iframe) => {  
15  expect($iframe.contents().find(element)).to.exist  
    }.then($iframe) => {  
      return cy.wrap($iframe.contents().find("body"))  
    };  
20  })  
  })
```

Codebeispiel .5: support/commands.js