Titel der Arbeit  
ggf. etwas länger

[1./2. Projektarbeit / Bachelorarbeit / Seminararbeit]

vorgelegt am 30. Januar 2018

Fakultät Wirtschaft

Studiengang Wirtschaftsinformatik

Kurs [Kursbezeichnung]

von

[Vorname Nachname]

|  |  |
| --- | --- |
| Betreuer in der Ausbildungsstätte: | DHBW Stuttgart: |
| [Name des Unternehmens] [Titel, Vorname und Nachname d. Betreuers] [Funktion des Betreuers] [Unterschrift des Betreuers] | [Titel, Vorname und Nachname des wissenschaftl. Betreuers/Prüfers] |

**Inhaltsverzeichnis**

[Abkürzungsverzeichnis (bei Bedarf) IV](#_Toc520458503)

[Abbildungsverzeichnis (bei Bedarf) V](#_Toc520458504)

[Tabellenverzeichnis (bei Bedarf) VI](#_Toc520458505)

[1 Einleitung 1](#_Toc520458506)

[2 Problemstellung 2](#_Toc520458507)

[3 Zentrale Transaktionsbestätigung 3](#_Toc520458508)

[3.1 Definition der zentralen Transaktionsbestätigung 3](#_Toc520458509)

[3.2 Der IST-Zustand der End-to-End Tests in der ZTB 4](#_Toc520458510)

[3.3 Der Soll-Zustand der End-to-End Tests in der ZTB 6](#_Toc520458511)

[4 Tests 8](#_Toc520458512)

[4.1 Motivation fürs Testen 8](#_Toc520458513)

[4.2 Testarten/Stages 8](#_Toc520458514)

[4.3 Art meiner Tests vorgestellt 8](#_Toc520458515)

[5 Verwendeten Tools (rein sachlich Entscheidung bei Lösungsansatz) 9](#_Toc520458516)

[5.1 Cypress.io 9](#_Toc520458517)

[5.2 Svelte 9](#_Toc520458518)

[5.3 Weitere 9](#_Toc520458519)

[6 Lösungsansatz 10](#_Toc520458520)

[6.1 Anforderungen 10](#_Toc520458521)

[6.2 Erwartete Probleme 10](#_Toc520458522)

[6.3 Thesenaufstellung 10](#_Toc520458523)

[7 Durchführung 11](#_Toc520458524)

[7.1 Was hab ich gemacht 11](#_Toc520458525)

[7.2 Was für Probleme hatte ich und wie habe ich sie gelöst 11](#_Toc520458526)

[7.3 Noch offene Anforderungen 11](#_Toc520458527)

[8 Ausblick 12](#_Toc520458528)

[9 Zusammenfassung 13](#_Toc520458529)

[Quellenverzeichnis 16](#_Toc520458530)

[Erklärung 17](#_Toc520458531)

# Abkürzungsverzeichnis (bei Bedarf)

AktG = Aktiengesetz

BFH = Bundesfinanzhof

# Abbildungsverzeichnis (bei Bedarf)

[Abb. 1: ZTB Testansicht 3](file:///C:\AZ_DATEN\ProjektarbeitGit\PA\RichtigeVorlagePa4.docx#_Toc521922689)

[Abb. 2: ZTB mTan Eingabe 3](file:///C:\AZ_DATEN\ProjektarbeitGit\PA\RichtigeVorlagePa4.docx#_Toc521922690)

[Abb. 3: Fachlicher Kontext der ZTB 4](#_Toc521922691)

[Abb. 4: Ist Zustand Test 1 5](#_Toc521922692)

[Abb. 5: Ist Zustand Test 2 5](#_Toc521922693)

[Abb. 6: Ist Zustand Test 3 5](#_Toc521922694)

[Abb. 7: Ist Zustand Test 4 6](#_Toc521922695)

[Abb. 8: Ist Zustand Test 5 6](#_Toc521922696)

# Tabellenverzeichnis (bei Bedarf)

[Tab. 1: Eine Beispieltabelle 3](#_Toc505086758)

1. Einleitung

In der Einleitung sind die Problemstellung und die daraus resultierende Zielsetzung der Arbeit exakt zu formulieren und die Vorgehensweise zu begründen. Notwendige Abgrenzungen sind hier vorzunehmen. Anstelle der Überschrift „Einleitung“ kann auch eine aussagekräftigere Überschrift verwendet werden.

1. Problemstellung

Für den reibungslosen Ablauf einer Transaktion ist es zwingend notwendig, Änderungen zu überprüfen und fehlerhafte Eingaben zu erkennen. Eine effektive Vorgehensweise ist die Automatisierung der zu diesem Zweck erstellten Testroutinen. Der Vorteil von automatisierten Tests ist, dass Entwickler ihren gesamten Code automatisch gegen alle Anwendungsfälle testen können und auftretende Fehler direkt sichtbar werden. Die Analyse von Fehlern wird auch erweitert, da auf den ersten Blick ersichtlich wird, welche Tests fehlschlagen. Nach den abgeschlossenen Änderungen kann überprüft werden, ob diese Tests jetzt fehlerfrei ausgeführt werden. Wenn dieser Ansatz nicht eingesetzt wird, sondern nur eine spärliche Teststruktur vorhanden ist, sind Entwickler gezwungen nach jeder Änderung umfangreiche manuelle Tests durchzuführen, um die Qualität des Codes sicherzustellen. Des Weiteren wird die Bearbeitung von Fehlern, die von außen ohne konkreten Hinweis gemeldet werden, unnötig erschwert. In diesen Fällen muss jeweils eine ausführliche Fehleranalyse durchgeführt werden, um den Fehler zu finden. Dieses manuelle Testen und umfangreiche Analysieren von Fehlern erhöht die Personal- und Betriebskosten. Dadurch wird ein Teil des Referatsbudgets und der Kapazität der Entwickler blockiert, die ansonsten auf andere Themen verwenden werden könnten. Aus diesem Grund sinkt die Produktivität des Referates (eine Bezeichnung für eine Allianz interne Unterabteilung) um ein Vielfaches.

Um diese Kosten zu verringern und die Code-Qualität zu erhöhen, soll in der zentralen Transaktionsbestätigung eine umfangreiche Teststruktur und der Ansatz des automatisierten Testens realisiert werden.

1. Zentrale Transaktionsbestätigung
   1. Definition der zentralen Transaktionsbestätigung

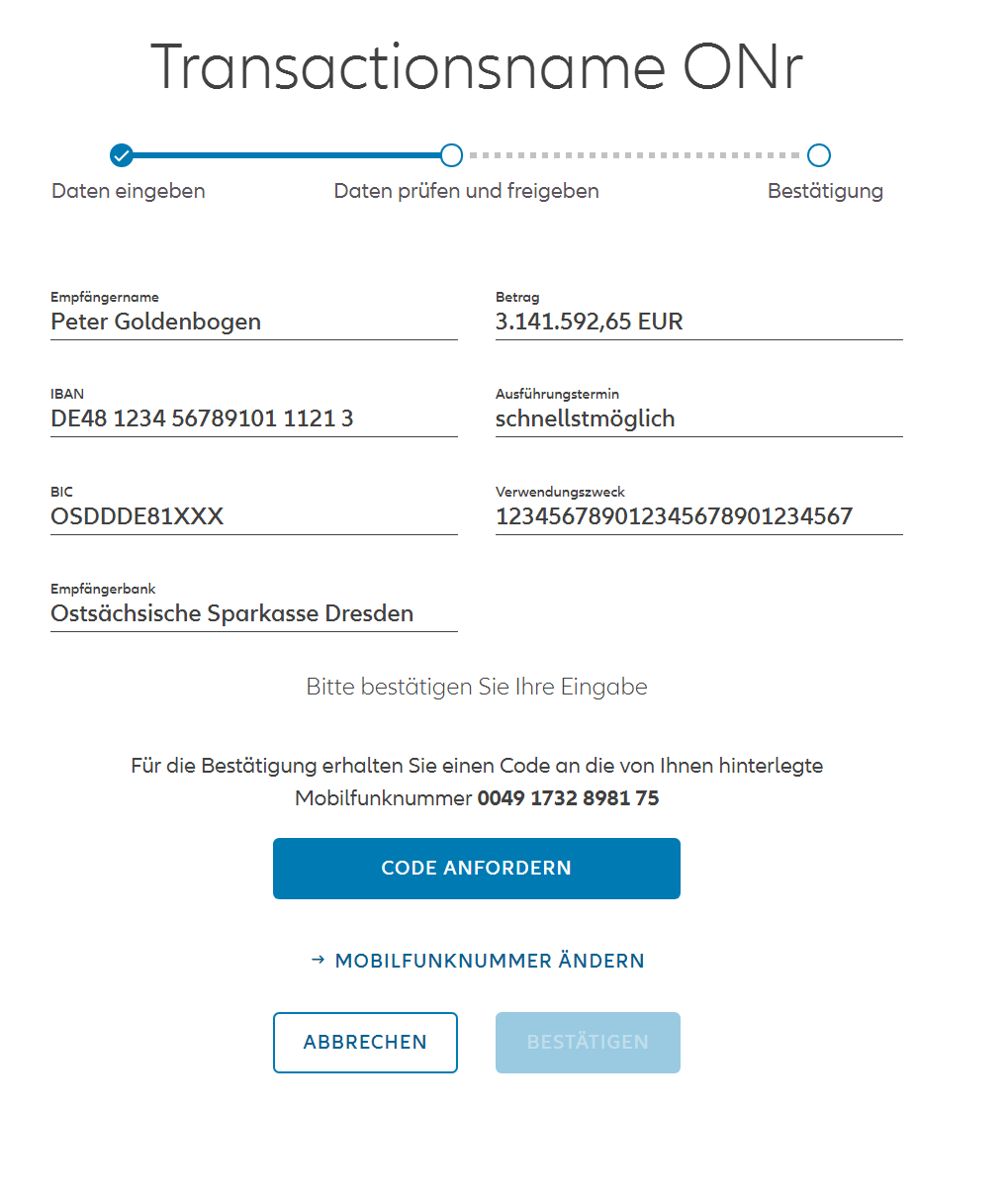
Die Aufgabe der Anwendung „zentrale Transaktionsbestätigung“ auch ZTB genannt, ist die Autorisierung von personenspezifischen Vorgängen durch eine Zwei-Faktor Authentifizierung.

Abb. 1: ZTB Testansicht

Die Anwendung „meine Allianz“, in der Nutzer ihre persönlichen Daten und Verträge einsehen und bearbeiten können, ruft im Falle eines autorisierungspflichtigen Vorgangs die Anwendung „zentrale Transaktionsbestätigung auf. Autorisierungspflichtige Vorgänge sind alle, die sicherheitsrelevante Daten, oder generell Änderungen am Vertrag betreffen.

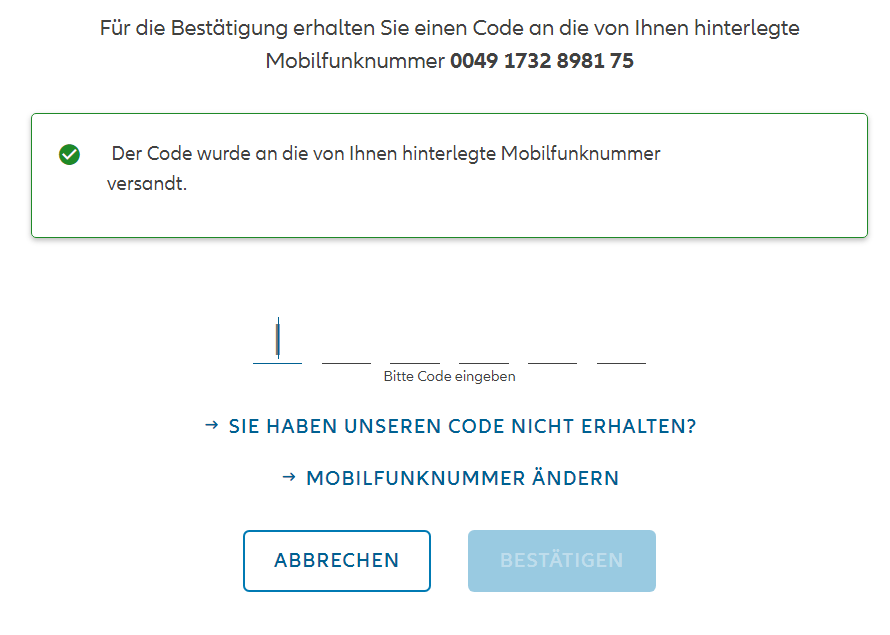
Der zentralen Transaktionsbestätigung, dargestellt in Abb. 1, werden bereits die kundenspezifischen Daten übermittelt und dort angezeigt. Im Beispielfall heißt der Kunde Peter Goldenbogen. Um den Vorgang fortzusetzen, muss der Nutzer einen Code anfordern, der als mTan an seine eingetragene Mobilfunknummer gesendet wird. Diese mTan wird dann in das Inputfeld (Abb. 2) eingetragen und auf Echtheit überprüft. Wenn die eingegebene Tan validiert wurde, wird der Vorgang autorisiert und die Autorisierung wird zurück an „Meine Allianz“ gesendet.

Abb. 2: ZTB mTan Eingabe

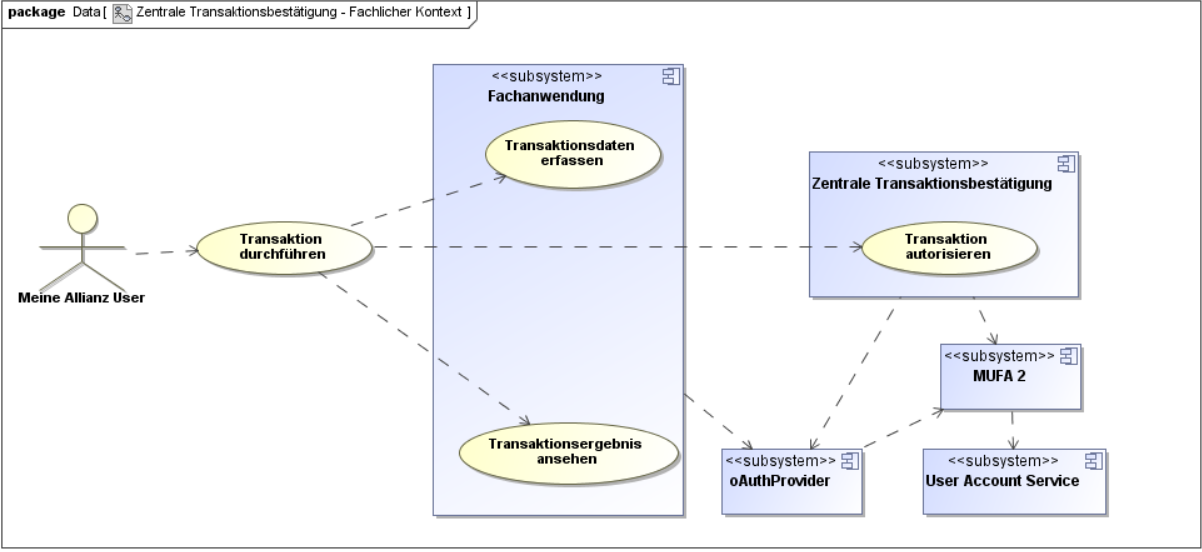


Abb. 3: Fachlicher Kontext der ZTB[[1]](#footnote-1)

Die technischen Hintergründe werden in dem Anwendungsfalldiagramm in Abb. 3 ausführlicher dargestellt. Der „Meine Allianz User“ ist unser Kunde, der eine Transaktion durchführen möchte. Diese Transaktion muss wie bereits beschrieben autorisiert werden. Die Zentrale Transaktionsbestätigung überprüft direkt beim Eingang der Daten, ob die eingetragene Telefonnummer und die Transaktionsnummer valide ist. Wenn dies nicht der Fall ist, wird der Vorgang abgebrochen und der Kunde muss die Daten in der Anwendung „meine Allianz“ nachtragen, ansonsten wird der Vorgang wie beschrieben weiter fortgeführt. Die ZTB generiert die mTan jedoch nicht selbst sondern dies wird durch die Subsyteme Mufa 2 und den oAuthProvider durchgeführt. Nachdem die mTan versandt wurde, hat der User drei Versuche, diese richtig in das Eingabefeld einzugeben.

* 1. Der IST-Zustand der End-to-End Tests in der ZTB

Die zentrale Transaktionsbestätigung verfügt aktuell bisher über sehr rudimentäre Tests, die in dem Framework Mocha geschrieben wurden. Dieses Testframework läuft mit Node.js und wurde durch den Java Script Runner von Angular (Karma) ausgeführt. Es ist bereits in der Anwendung ein Mock-Server implementiert, der das Testen ohne reale Testdaten ermöglichen soll. Zu diesem Zeitpunkt muss der Mock-Server jedoch per Hand umgestellt werden, um Fehlerfälle, die sonst nur durch Fehler in der Kommunikation mit anderen Anwendungen entstehen können, darzustellen.

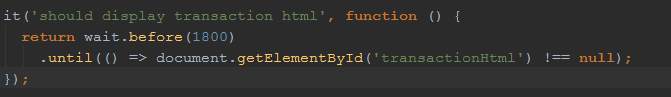


Abb. 4: Ist Zustand Test 1

Der erste Test aus Abb. 4 ist sehr simpel und überprüft nur, ob die richtige HTML-Seite geladen wurde. Jedoch ist dieser Test sehr wichtig, denn wenn tatsächlich eine falsche Seite geladen wurde, schlagen alle anderen Tests automatisch ohne erkennbaren Grund fehl.

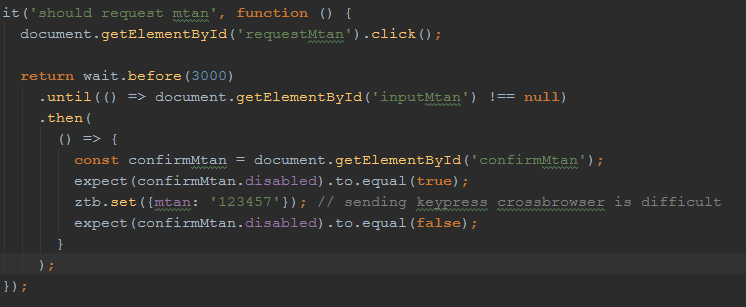


Abb. 5: Ist Zustand Test 2

Der nächste Test (Abb. 5) ist etwas komplexer und überprüft, ob bei Betätigung des Buttons „requestmTan“ die Seite richtig angezeigt wird. Des Weiteren wird die Funktion der Eingabefelder und des „Bestätigen“ Buttons rudimentär getestet. Bei fehlerhafter Eingabe, zum Beispiel nicht vollständig ausgefüllte Felder, könnte es zu unerwarteten Fehlern im Backend kommen, da dort andere Eingaben vorausgesetzt werden.

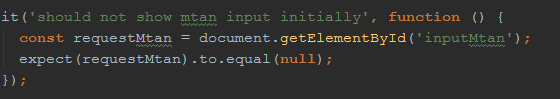


Abb. 6: Ist Zustand Test 3

Der dritte Test (Abb. 6) bezieht sich ebenfalls auf die Eingabefelder. In diesem Fall wird überprüft, ob diese direkt beim Aufrufen der Startseite zu sehen sind oder ob sie tatsächlich wie gewünscht erst nach Betätigen des Buttons „requestmTan“ erscheinen.

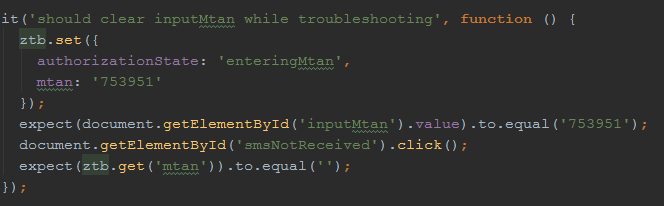


Abb. 7: Ist Zustand Test 4

Der Test in Abb. 7 beschreibt das Troubleshooting. Wenn in das Eingabefeld der mTan Zahlen eingegeben wurden und der User auf den Link „smsNotReceived“ klickt, sollte er eine neue SMS bekommen und das Eingabefeld sollte geleert worden sein.

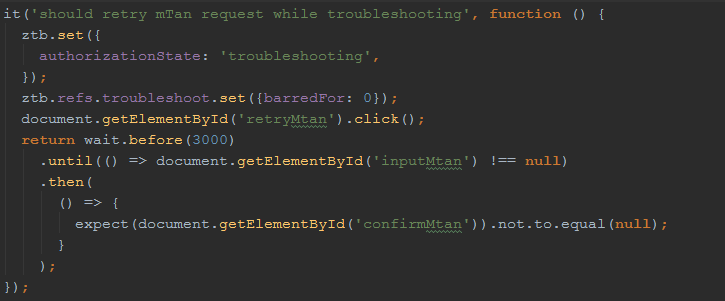


Abb. 8: Ist Zustand Test 5

In diesem Test (Abb. 8) wird überprüft, ob der Button, mit dem eine neue mTan angefordert wird, nach Ablauf des Counters, der zu Testzwecken auf 0 gesetzt wird, wieder aktiv ist.

In den beschriebenen Tests ist zu erkennen, dass teilweise wichtige logische Funktionen der Anwendung bereits getestet wurden. Problematisch ist jedoch, dass immer nur einige Teilaspekte getestet wurden und der größte Teil der Anwendung weiterhin ungetestet bleibt.

* 1. Der Soll-Zustand der End-to-End Tests in der ZTB

Die End-to-End Tests in der zentralen Transaktionsbestätigung sollen alle Anwendungsfälle im Frontend-Bereich umfassen und nicht nur wie bisher einen Bruchteil. Die Testfälle sollen so verlässlich sein, dass die Entwickler davon ausgehen können, dass im Frontend keine Fehler auftreten, wenn alle Tests fehlerfrei durchlaufen werden. Diese Testfälle sollen alle automatisch beim Einchecken des Codes ausgeführt werden, um direkt dem Entwickler als Feedback zu dienen, ob die Anwendung fehlerfrei ist. Die Testfälle sollen alle eindeutig beschrieben sein, um sicherzustellen. dass die Entwickler, wenn ein Test fehlschlägt, direkt erkennen können, in welchem Abschnitt des Codes der Fehler auftritt.

Der Wunsch des Referates ist es, dass diese Tests sofern möglich mit einem neuen Framework namens Cypress.io (siehe Kapitel …) geschrieben werden.

1. Tests
   1. Motivation fürs Testen

„Ein wesentliches Merkmal der fortschreitenden Industrialisierung seit dem Ende des 18. Jahrhunderts ist die Mechanisierung kräfteraubender oder zeitaufwendiger manueller Tätigkeiten in beinahe allen Produktionsprozessen.“[[2]](#footnote-2)

Die Situation des Testens wird durch dieses Zitat von Richard Seidl sehr passend beschrieben, denn ausführliches manuelles Testen nimmt viel Zeit in Anspruch. Auf diese Tests zu verzichten wäre jedoch fatal, da die Funktionalität der Anwendung gewährleistet sein muss. Anwendungen vor dem Produktionsgang zu testen, ist daher unumgänglich und da es in komplexen Bereichen, wie der Softwareentwicklung, leicht zu Fehlern kommt, sollten diese so früh wie möglich entdeckt werden. Um dies zu bewerkstelligen sollte nicht nur vor dem Produktionsgang getestet werden, sondern nach jeder Änderung die ein einzelner Entwickler vornimmt. Dies muss, ohne automatisierte Tests, von den Entwicklern selbst durchgeführt werden und kostet viel Zeit.

Die Komplexität von großen Anwendungen, grade in Unternehmensstrukturen, erschwert ein manuelles Testen, da nun Kenntnisse der gesamten Anwendung von Nöten sind, um die möglichen Auswirkungen einzelner Änderungen zu erkennen. Aus diesem Grund sollten zumindest die Hauptfunktionen durch automatisierte Tests abgedeckt werden, damit direkt erkennbar ist ob diese nach einem Commit noch funktionieren.

In den Bereichen der Agilen Softwareentwicklung ist man diesem Verfahren schon ein paar Schritte voraus. Statt nachträglich Tests für einzelne Funktionen zu schreiben wird zuerst ein Test geschrieben und erst dann die benötigten Funktionen implementiert, die der Test validieren soll. Dieses Konzept des Test-Driven-Developments soll eine höhere Qualität und Übersicht des Codes sicherstellen. Da nur kleine atomare Funktionalitäten durch jeden Test hinzugefügt werden bleibt der Code verständlich und aufgeräumt. Bei dieser Methode wird also jede Funktion der Anwendung getestet und es ist jederzeit ersichtlich sobald irgendeine Funktionalität fehlschlägt. Test-Driven-Development eröffnet den Entwicklerteams auch die Möglichkeit diese Tests als Regressions- Integrationstests (besser beschrieben im nachfolgenden Kapitel) bei jedem Commit durchlaufen zu lassen, da die Tests bereits vorhanden sind.

* Warum
* Nutzen
* CodeQualität
* Veränderung in der Programmiertechnik (Zeitablauf/agil usw)
* TDD
* Fehlersuche
* Zeitaufwand
* Direktes Feedback
* BO Testet
* Aussagekräftige Fehlermeldungen
* Continuous deployment/integration
  1. Testarten/Stages

Es gibt viele verschiedene Testarten. Die wichtigsten die um Test Prozess vorhanden sein sollten sind Unit, Integration und End-to-End Tests, da diese die gesamte Funktionalität der Anwendung sicherstellen. Es gibt aber noch weitere Testarten die nicht vergessen werden dürfen. Es gibt den Penetrationstest, bei dem die Sicherheit der Anwendung beziehungsweise des Systems sichergestellt werden soll, die Regressionstests, die sicherstellt, dass eine Modifikation der Anwendung keine Fehler verursacht hat. Die Regressionstest sind kein Ersatz für normales Testen, da nur überprüft ob die Modifikation Fehler hervorgerufen hat, aber nicht alle Funktionalitäten der ursprünglichen Anwendung. Ein weiterer wichtiger Test ist der Nutzer-Akzeptanztest. Dieser Test soll sicherstellen, dass die Anwendung aus Sicht des Nutzers, wie gewünscht, funktioniert. Häufig ist es schwer für den Entwickler zu beurteilen welche Funktionen der Nutzer wirklich benötigt beziehungsweise mit seiner Anfrage gemeint hat.

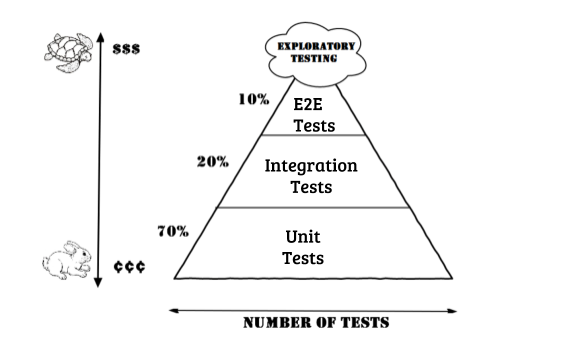
[[3]](#footnote-3)

Abb. Testing Pyramid

Unit Tests werden von Martin Fowler als “Low-Level-Tests beschrieben, sich die nur auf einen kleinen Teil des Systems beziehen und von den Entwicklern selber in ihren normalen Programmen geschrieben werden. Mit Ausnahme eines Test Frameworks.“[[4]](#footnote-4) Diese Art von Tests werden verwendet um kleine Teile des Systems zu überprüfen und sind die schnellsten Tests. Im Normalfall sind, wie auch in Abb. 9 dargestellt, 70% aller Tests Unit Tests.

Die nächste Stufe des Testens sind die Integrations-Tests, jedoch ist der Übergang zwischen dieser Art und den Unit Tests fließend. Viele Integrations-Tests sehen aus wie erweiterte und langsamere Unit Tests.[[5]](#footnote-5) Der Grundsätzliche Sinn der Integrations-Tests besteht darin, das Zusammenspiel individuell entwickelter Softwarekomponenten zu testen und wie der Name impliziert überprüfen diese Tests, ob diese Komponenten in eine größere Umgebung integriert werden können. Dennoch müssen nicht immer breit gefächerte Anwendungen getestet werden, da auch das Zusammenwirken von zwei Softwareteilen bereits ein Integration-Test ist.[[6]](#footnote-6) Ein guter Richtwert für die Anzahl dieser Art von Tests ist 20%, also deutlich weniger als die Unit-Tests. Der Grund für diesen großen Unterschied ist die Tatsache, dass bereits alle Funktionalitäten der einzelnen Bausteine, durch die Unit-Tests, getestet wurden und nur noch die Zusammenarbeitet der Bausteine getestet werden muss.

Die letzte Test Art sind die End-to-End Tests. Diese Tests geben dem Entwickler das größte Vertrauen in seine Anwendung, da diese hier die gesamte Anwendung vom Start bis zum Ende durchgetestet wird. Das bedeutet also, dass in der Gui alle Eingaben getätigt werden können und überprüft wird ob die erwarteten Ergebnisse auftreten. Ein Vorteil dieser Tests ist, dass wenn sie vernünftig geschrieben wurden, die Sicherheit besteht von möglichen Fehlern im Vorfeld zu erfahren und diese beheben zu können. Problematisch ist jedoch das diese Tests regelmäßig gewartet werden müssen und die vergleichsweise sehr hohe Laufzeit.

Diese Testpyramide ist nur wirksam wenn alle drei Testarten umfangreich implementiert wurden, da sonst nicht die gesamte Anwendung getestet wird. Diese Testpyramide bietet sich auch an automatisiert regelmäßig durchlaufen zu lassen. Die anderen beschriebenen Testarten sind eher einmalige beziehungsweise unregelmäßige Tests die durchgeführt werden um einmalig Informationen einzuholen oder Änderungen zu testen.

* 1. Art meiner Tests vorgestellt

1. Verwendeten Tools (rein sachlich Entscheidung bei Lösungsansatz)
   1. Cypress.io

Cypress.io ist ein Open Source all-in-one Test Framework und existent seit 2014, jedoch war es nur in einer Privaten Beta verfügbar. Am 10.10.2017 wurde Cypress.io zu einem Open-Source Produkt.[[7]](#footnote-7) Dieses Produkt unterscheidet sich von den bisherigen Standards wie zum Beispiel Selenium. [[8]](#footnote-8)Während Selenium und die meisten anderen Test Anwendungen außerhalb vom Browser laufen, wird Cypress direkt im Durchlauf der Anwendung ausgeführt. Und wie in Abb. 9 dargestellt benötigt man in Cypress keine weiteren Frameworks usw.

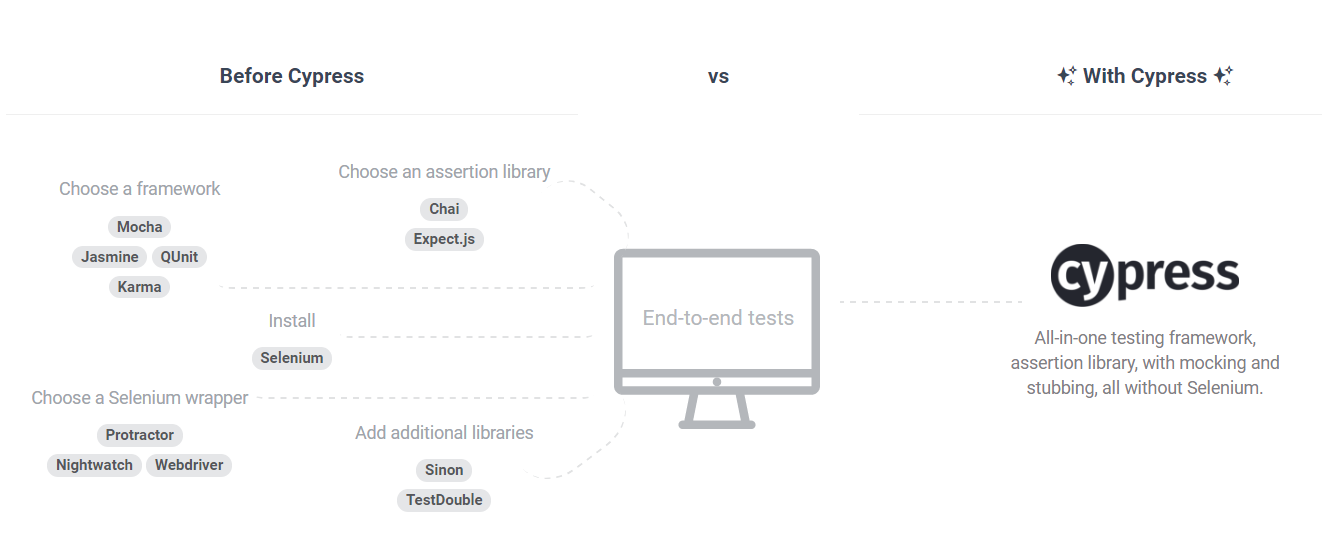


Abb. Selenium vs Cypress[[9]](#footnote-9)

Hinter Cypress.io steht ein Node.js Server der die ganze Zeit mit Cypress.io kommuniziert. Durch diese Architektur hat der Entwickler Zugriff auf beide Teile und kann auf alle Events der Anwendung in Echtzeit reagieren. Des Weiteren ist es möglich „WebTraffic“ während der Durchführung zu ändern und ermöglicht somit genau die Daten die in den Browser rein- bzw. aus dem Browser herausgesendet werden und diese wenn Nötig auch noch zu ändern. Da Cypress.io lokal installiert wird, ist es möglich Bilder, Videos und generell Dateisystem Funktionen auszuführen. Dies und weitere Features, wie zum Beispiel in Abb. 10, machen das neue Testing Framework CypressIO aus. Einige dieser Features werden auch von anderen Frameworks angeboten, jedoch benötigt man dort Erweiterungen/Plugins oder noch andere Tools um dieselben Funktionen zu erreichen. Besonders das automatische Warten ist ein Vorteil der Cypress für unsere Zwecke empfohlen hat. Automatisches Warten, das Warten auf Veränderungen auf der Website, wird von Selenium zum Beispiel nur im Zusammenspiel mit Protractor möglich. Dies ist vorallem bei Asynchronen Websiten von Nöten, da Anfragen teilweise länger dauern können und die neuen Begebenheiten erst getestet werden können sobald sie geladen haben. Weitere nützliche Features ist die komplette Kontrolle über Ein- und ausgehende Daten im Netzwerk und die Zeitreisen Funktion. Diese ist automatisch implementiert und sorgt dafür, dass nach jedem Schritt im Test ein Snapshot gemacht wird. Dies ermöglicht dem Entwickler bei jedem einzelnen Test die Schritte nachzuvollziehen und bei einem fehlgeschlagenen Test genau zu erkennen an welcher Stelle der Test fehlschlägt.

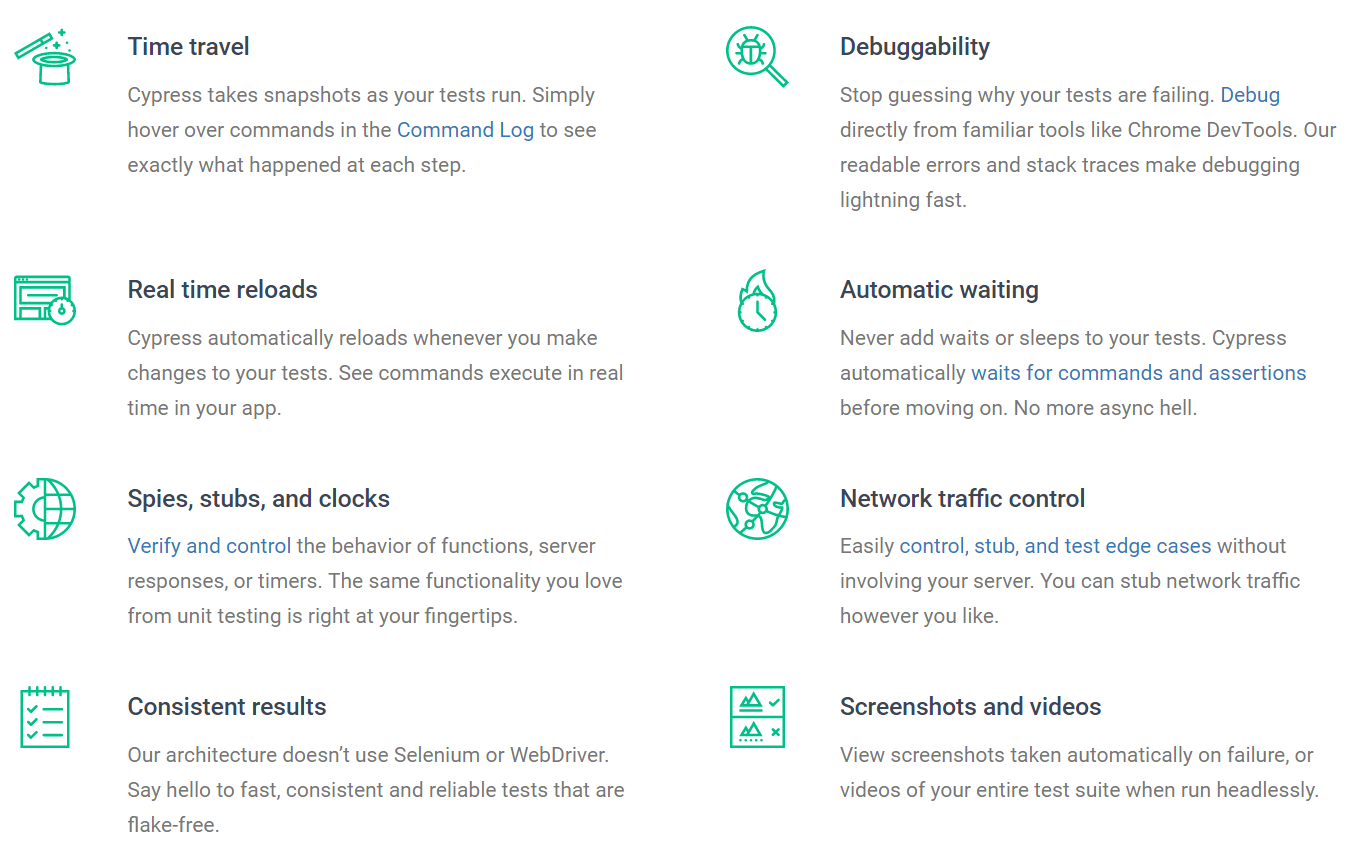


Abb. Features Cypress.io

Die Tests in Cypress werden in reinem Java Script geschrieben und sind somit für die Mehrheit der Entwickler sehr leicht zu verstehen und durch Aussagekräftige Namen, der Funktionen, sehr leserlich. Auffällig ist allerding auch gewesen, dass das Testen mit Cypress in der Unternehmensumgebung nur mit dem Browser Electra möglich war.

* 1. Node.js
  2. Weitere
* Npm
* IntelliJ

1. Lösungsansatz
   1. Anforderungen
2. Testen ob das Tool Cypress im Komerziellen bereich anwendbar ist
3. Tests sollen Aussagekräftig benannt werden
4. Tests sollen bei Fehlschlag aussagekräftig zeigen was fehl schlägt
5. Alle Funktionen sollten abgedeckt werden
6. End2End ?
7. Gegebenenfalls eine implementierung in Jenkins
8. Tests sollten verständlich geschrieben werden
9. Zeitersparnis beim manuellen Testen, da Entwickler sich auf die Funktionalität verlassen können
10. Aussagekräftige Strukturierung der Tests um nicht immer alle Dateien durchlaufen zu lassen
    1. Erwartete Probleme
11. Implementierung in jenkins
12. Reelle Daten testen
13. Einarbeitung in Frontend Arbeitsweisen
14. Probleme mit Cypress (neue Anwendung/wenig Dokumentation)
    1. Thesenaufstellung notwendig ???
15. Durchführung
    1. Was hab ich gemacht
16. Einarbeitung in Anwendung/Cypress/Javascript
17. Analyse für Testbare fälle
18. Seitenaufruf testen
19. Funktionen testen / Elemente einzeln (Inputfeld)
20. Fehlermeldungen testen
21. Syntax teilweise verwirrend ???
    1. Was für Probleme hatte ich und wie habe ich sie gelöst
22. Cypress localhost
23. Cypress Browser (Electra)
24. Elemente mit gleichen Namen
25. Mock Daten (Stubing)
26. Implementierung in jenkins
    1. Noch offene Anforderungen
27. Jenkins
28. End2End
29. Funktionen (TanValidierung)
30. Ausblick
31. Cypress soll hauptsächlich für Unit tests angewendet werden
32. Jenkins ???
33. Zusammenfassung

**Anhang (bei Bedarf)**

[Anhang 1: Beispielüberschrift 5](#_Toc505081892)

[Anhang 2/1: Teil 1 des Anhangs 2 5](#_Toc505081893)

[Anhang 2/2: Teil 2 des Anhangs 2 6](#_Toc505081894)

Anhang 1: Beispielüberschrift

Hier können Sie dann Unterlagen einfügen (Empfehlung: Im Text auf den Anhang verweisen). Die Überschrift des Anhangs bitte über Start – Formatvorlagen – Anhang formatieren; nur dann kann ein korrektes Anhangsinhaltsverzeichnis erstellt werden, das nicht im Inhaltsverzeichnis vorne erscheint.

Anhang 2/1: Teil 1 des Anhangs 2

Bei Anhängen, die nicht auf eine Seite passen, können Sie den Anhang ggf. aufteilen.

Anhang 2/2: Teil 2 des Anhangs 2

Bei Anhängen, die nicht auf eine Seite passen, können Sie den Anhang ggf. aufteilen.

# Quellenverzeichnis

**Literaturverzeichnis**

**Preiß, N. (2007):** Entwurf und Verarbeitung relationaler Datenbanken, München/Wien: Oldenbourg

**Steger, J. (2006):** Kosten- und Leistungsrechnung, 4. Aufl., München/Wien: Oldenbourg

**Stoi, R. (2003):** Management und Controlling von Intangibles, in: Studium & Praxis, 4. Jg., Nr. 1, S. 34-46

**Gesprächsverzeichnis (bei Bedarf)**

**Musterfrau, E. (2018):** Bereichsleiterin Sales, Fantasieunternehmen GmbH, Stuttgart, persönliches Gespräch am 25.01.2018 in Waiblingen

# Erklärung

Ich versichere hiermit, dass ich meine [Projektarbeit / Bachelorarbeit / Seminararbeit] mit dem Thema: [ Thema einfügen ] selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(Ort, Datum) (Unterschrift)

**Brian Mann (2017):** Cypress is now public beta, https://www.cypress.io/blog/2017/10/10/cypress-is-now-public-beta/#, Abruf: 13.08.2018

**D-It-Kuv2-L1 (2018):** It-Konzept Zentrale Transaktionsbestätigung (ZTB) vom 2018, **[Erscheinungsort fehlt!]**,

**Fowler, M. (2014):** UnitTest, https://martinfowler.com/bliki/UnitTest.html, Abruf: 14.08.2018

**Fowler, M. (2018):** Integration Tests, https://martinfowler.com/bliki/IntegrationTest.html, Abruf: 14.08.2018

**o. V. (o. J.):** Testing has been broken for too long, https://www.cypress.io/how-it-works/, Abruf: 14.08.2018

**o. V. (2018):** Key Differences- Architecture, https://docs.cypress.io/guides/overview/key-differences.html#Architecture, Abruf: 15.08.2018

**Raquel Pau (2018):** Testpyramid, - http://bytes.schibsted.com/impact-testing-stop-waiting-tests-not-need-run/testpyramid/, Abruf: 13.08.2018

**Seidl, R./Baumgartner, M./Bucsics, T. (2012):** Basiswissen Testautomatisierung, Konzepte, Methoden und Techniken, Auflage 1, Heidelberg: dpunkt.verlag

**Tarlinder, A. (2017):** Developer testing, Building quality into software, Online-Ausg, Boston: Addison-Wesley

1. D-It-Kuv2-L1 **(Hrsg.) (2018**. [↑](#footnote-ref-1)
2. Seidl/Baumgartner/Bucsics 2012, S. 1. [↑](#footnote-ref-2)
3. Raquel Pau 2018. [↑](#footnote-ref-3)
4. Fowler 2014. [↑](#footnote-ref-4)
5. vgl. Tarlinder 2017, S. 26. [↑](#footnote-ref-5)
6. vgl. Fowler 2018. [↑](#footnote-ref-6)
7. vgl. Brian Mann 2017. [↑](#footnote-ref-7)
8. vgl. o. V. 2018. [↑](#footnote-ref-8)
9. o. V. o. J. [↑](#footnote-ref-9)