Wie, ich soll jetzt was in JavaScript schreiben?

Vor Einführung der Produktorganisation Ende 2016 gab es ein dediziertes Team, dass sich mit Frontendtechnologien wie JavaScript und Cascading Style Sheets (CSS) beschäftigt hat. Die einzelnen Entwicklungsteams fokussierten sich auf die Backendentwicklung in Java. Seit Einführung der Produktteams gibt es kein zentrales Team mehr, das die Frontendentwicklung für alle übernimmt. Stattdessen ist jedes Produktteam selbst dafür verantwortlich, ganz im Sinne von Self-Contained Systems (SCS) bzw. Micro Frontends.

Im Team Kaufen haben wir vor eineinhalb Jahre mit relativ einfachen Mitteln angefangen, JavaScript und CSS zu schreiben, und seitdem unser Vorgehen und Tooling stetig weiterentwickelt.

Die ersten Gehversuche

Das erste eigene Frontend haben wir im Team Kaufen Ende 2017 angefangen zu entwickeln. Der einfachste Weg zur Auslieferung von JavaScript und CSS war die Integration von Inline-Skript in unseren HTML-Code. Die Skripte landeten dadurch eins-zu-eins so im Browser, wie wir sie geschrieben hatten.

Durch dieses Vorgehen benötigten wir kein besonderes Tooling, sondern konnten schnell in die Entwicklung der benötigten Funktionalitäten einsteigen. Gleichzeitig ergeben sich bei dieser Vorgehensweise auch verschiedene Nachteile:

- Aufgeblähter Code im Browser
- Codekommentare im Browser
- kein Caching des Codes
- keine Tests des JavaScripts
- globale Variablen in JavaScript
- kein Sicherstellen, dass JavaScript in allen gängigen Browsern ausführbar ist

Um diese Probleme zumindest teilweise zu adressieren, haben wir nach einer

Weile begonnen, den CSS-Code in eigene Dateien auszulagern und durch Maven-Plugins zu verkleinern. So konnten wir die CSS-Dateien dann über Link-Tags im HTML einbinden und vom Caching durch den Browser profitieren. Die Verarbeitung über Maven fügte sich sehr gut in unseren bestehenden Buildprozess. Uns wurde allerdings schnell klar, dass die Verarbeitung über Maven nicht zukunftsfähig ist. Da wir nach und nach insbesondere immer mehr JavaScript geschrieben haben, erkannten wir, dass wir uns mit den Werkzeugen aus dem Node Package Manager (NPM)-Kosmos beschäftigen müssen, um uns hier zukunftsfähig aufstellen zu können.

Auf in neue Gefilde

Im ersten Schritt wollten wir die folgenden Punkte verbessern:

- Kein Inline-Skipt mehr. Stattdessen sollen JavaScript- und CSS in eigenen Dateien entwickelt und auch als solche ausgeliefert werden.
- Verarbeitung von JavaScript und CSS, wie Minify und Uglify, damit der Code kleiner wird, keine Kommentare mehr im Browser sichtbar sind und der ausgelieferte Code schlechter lesbar ist.
- Sicherstellen, dass JavaScript in allen gängigen Browsern ausführbar ist, durch Linting, dass prüft, ob JavaScript in Version ECMA Script 5 (ES5) geschrieben ist.
- JavaScript-Abhängigkeiten sollen über NPM bezogen werden können, anstatt über WebJars, die wir bisher über Maven gezogen hatten.

Resultat der Verbesserung unseres Buildprozesses war der Einsatz von

- Yarn als Package Manager und Alternative zu NPM, der aber dennoch die NPM Repositories nutzen kann
- Gulp zur Automatisierung der verschiedenen notwendigen Buildschritte inkl. zahlreicher Plugins für Gulp zur Verarbeitung des JavaScript- und CSS-Codes
- Linting mit ESLint
- Bezug externer JavaScript-Abhängigkeiten über NPM

Unsere gesetzten Ziele konnten wir so erreichen. Trotzdem hatten wir noch ein paar wesentliche Aspekte zu verbessern.

- Zur besseren Wartbarkeit des JavaScript-Codes hatten wir mehrere JavaScript-Dateien, die dann auch einzeln ausgeliefert wurden. Dadurch gab es Abhängigkeiten zwischen den Dateien, die beim Einbinden in den HTML-Code zu berücksichtigen waren.
- Die JavaScript-Dateien waren relativ groß. Dadurch war der Code schwer lesbar und Refactorings waren sehr aufwändig und fehleranfällig.
- Der JavaScript-Code wurde nach wie vor durch globale Variablen angesteuert, bspw. durch Click-Events im HTML-Code.
- Fehler im JavaScript-Code fielen erst relativ spät in unserer Build & Deployment Pipeline auf, da wir mit der vorhandenen Struktur keine Unittests schreiben konnten.

Modularisierung des JavaScript-Codes

Zur Adressierung der genannten Schwachstellen wollten wir unseren JavaScript-Code in erster Linie in kleinere Module aufteilen, die dann durch den Buildprozess in einer Datei gebündelt werden.

Zunächst haben wir analysiert, ob sich für unsere Zwecke der Einsatz eines der gängigen JavaScript-Frameworks wie React, Angular oder Vue.js eignet. Entschieden haben wir uns allerdings für einen Model-View-Controller (MVC)-Ansatzes mit Vanilla JavaScript. Dieser Ansatz ist bspw. leichtgewichtiger und besser mit unserem Einsatz der Template-Engine Handlebars, die wir für die HTML-Generierung aktuell einsetzen, zu vereinbaren.

Zur konsequenten Umsetzung eines MVC-Patterns haben wir unseren JavaScript-Code in viele kleine JavaScript-Dateien unterteilt, je eine für Model, View und Controller. Das Bündeln der einzelnen kleinen Dateien zu einer großen Datei, die dann auch über ein Script-Tag im HTML eingebunden wird, erfolgt im Buildprozess mit Webpack. Damit Webpack die einzelnen Dateien zu einer zusammenfügen kann, definieren die einzelnen Dateien Node.js-Module. Das Resultat dieses Buildprozesses ist eine anonyme JavaScript-Datei, die keine globalen Variablen definiert, sich selbst auf HTML DOM Events, wie etwa Click-Events, registriert und keine Abhängigkeiten zu anderen Dateien hat.

Im ersten Schritt wollten wir auf ECMA-Script 6 (ES6)-Funktionalitäten verzichten und uns stattdessen strikt an die JavaScript ES5-Version halten. So

mussten wir in unseren Buildprozess nicht auch direkt eine Konvertierung von ES6 zu ES5 einbauen, sondern können diesen möglichst simpel halten.

Durch die kleinen JavaScript-Module waren wir auch in der Lage, einfache Unittests für unseren Code zu schreiben. Nach einer kurzen Sondierung der zahlreichen Frameworks haben wir uns für die Nutzung von Jest entschieden. Durch diese Tests konnten wir unsere Codequalität maßgeblich steigern.

Kommt da noch mehr?

Der aktuelle Aufbau unseres JavaScript-Codes funktioniert so für uns weitestgehend schon sehr gut. Trotzdem gibt es Punkte, die wir in der Zukunft erneut evaluieren und verbessern werden:

- Einsatz von ES6 statt ES5: ES6 bietet einige Möglichkeiten, den Code lesbarer zu gestalten. Zudem sind wir von der Strukturierung über Node.js-Module nicht überzeugt und würden stattdessen lieber auf ES6-Module bzw. Klassen setzen.
- Ausführbarkeit in allen gängigen Browsern sicherstellen durch Browser-spezifische Tests oder Polyfills, bspw. durch den Einsatz von Babel
- Schnelleres Feedback und bessere Qualität durch Browser-spezifische Tests von zusammenhängenden JavaScript-Module als Erweiterung zu den Unittests mit Jest

---

Die innovative Idee der Voice App „Thalia Buchtipps“ kommt an!

Seit ein paar Monaten ist der Thalia Alex Skill erst online und erfreut sich bereits einer kontinuierlichen Nutzung. Vor allem die abwechslungsreiche und neckische Art kommt bei den Kunden gut an und konnte nun auch die Jury des „Deutsche Exzellenz-Preis“ überzeugen.

Another One Bites the Dust – Wie ein Monolith kontrolliert gesprengt wird… Teil I

Neben unserem Webshop, der auf euren Smartphones, Tablets, PCs oder Laptops läuft, gibt es bei Thalia noch ein weiteres interessantes Produkt, welches direkt an eine separate Shop-Plattform geknüpft ist: den Thalia eReader-Shop, der auf unseren tolino-eReadern läuft.
Bereits seit 2010 mit dem OYO, gibt es bei Thalia die Möglichkeit auf eReadern nicht nur zu lesen, sondern auch zu shoppen und sich von neuen eBooks inspirieren zu lassen.

Doch was steckt genau dahinter?

Wir möchten hier ein kleines Experiment wagen und euch auf eine mehrteilige Reise mitnehmen, die davon erzählt, wie wir es geschafft haben, ein ursprünglich außer Haus entwickeltes, monolithisches Shopsystem in einen modernen, Microservice/Microfrontend-basierten Shop zu überführen.
Auch die Herausforderung, die InHouse-Entwicklung durch ein schlagkräftiges Scrum-Team zu betreiben, und welche Fallstricke wir dadurch meistern mussten, sollen Bestandteile eines weiteren Teils dieser Reise werden.
Welche Rolle für uns in diesem Rahmen eine stimmige Produktvision und das ‚eReader-First'-Denken bzw. der MVP-Gedanke hat, soll in einem dritten Teil vorgestellt werden.

Von Jenga-Türmen, Schrödinger – Deployments und der Reise ins große Ungewisse

Seien wir mal ehrlich, für die meisten von uns sind heutzutage der Begriff ‚Microservices' und die damit zusammenhängenden Design-Patterns doch fast

schon ein alter Hut. Wenn wir neue Systeme bauen, versuchen wir natürlich einen klaren, technisch sauberen Schnitt und Verantwortlichkeiten nach denselben Kriterien zu schneiden.
Auf der grünen Wiese ist dies auch meist eine sehr angenehme Herausforderung. Was aber, wenn man nicht auf der grünen Wiese starten möchte oder kann?

2016 wurde bei Thalia die Entscheidung getroffen, die Entwicklung des zuvor extern entwickelten eReader-Shops ins Haus zu holen und hierfür ein kleines, anfangs noch vierköpfiges Scrum-Team einzusetzen.

Den eReader-Shop selber muss man sich hierbei als klassisches, über die Jahre gewachsenes, monolithisches System vorstellen.
Auf unseren E-Ink-basierten eReadern läuft ein von diesem System ausgelieferter Shop.
Der eReader erfüllt hierbei zwei Funktionen:
Zum Einen bietet er eine in einer Activity gekapselte, separate Shop-Webview, in der das HTML unseres Shops dargestellt werden kann.
Zum Anderen läuft diese Activity auf einer Android AOSP-Firmware, die zusätzlich eReader-spezifische Callbacks verarbeiten kann. Zum Beispiel ein/e eBook/Leseprobe auf das Gerät laden oder unseren oAuth-basierten Securityprozess verwalten.

Die meiste Logik wird dabei von weiteren Thalia-Backend Systemen verarbeitet. Auftragsanlage, Suchen etc. werden von dedizierten Produktteams verwaltet, die via JSON-Schnittstellen vom eReader-Shop-Backend aufgerufen wurden. Das Backend des eReader-Shops war also im Wesentlichen zuständig für das HTML und die eReader-spezifischen Callbacks sowie spezielle, gerätespezifische Logik (siehe Abbildung 1).

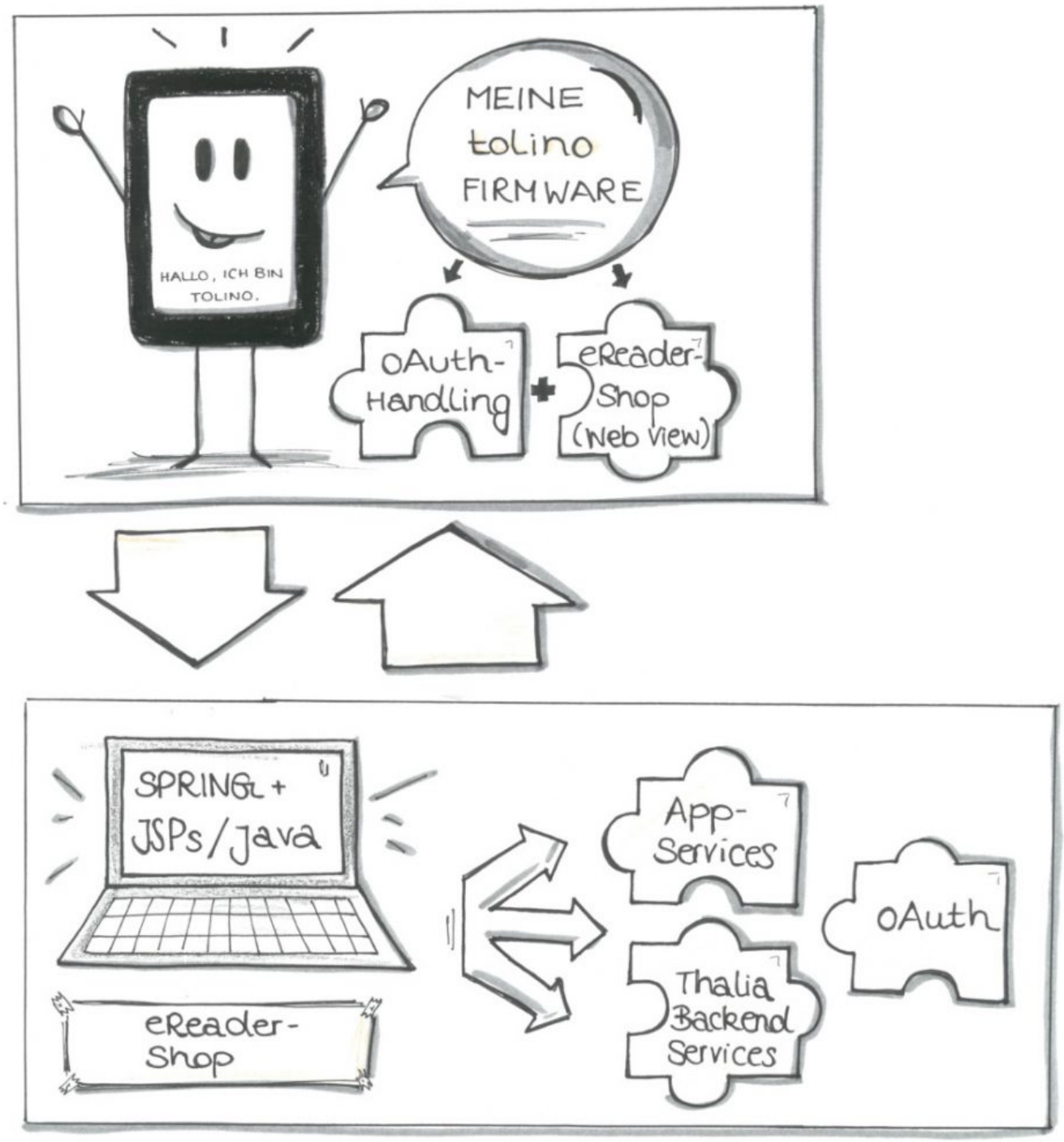


Abbildung 1

Auch wenn ich kein Freund von Metriken wie ‚Lines of Code' bin, wenn es darum geht, die Komplexität eines Systems zu beschreiben mag es hier helfen, eine ungefähre Vorstellung des Monolithen zu bekommen:

Der ursprüngliche, von uns übernommene Monolith hatte etwa 72000 sloc (source lines of code) bestehend aus JAVA Code (542 Klassen), JSPs (150 Klassen), CSS (31 Klassen) und JS (27 Klassen) bei einer Testabdeckung von etwa 17%.

Also ein wackeliges, großes, eventuell auch wenig wartbares Konstrukt! Klingt ein wenig nach Jenga-Turm, oder?
Der eReader-Shop in seiner ursprünglichen Form war zwar kein klassischer, mächtiger Monolith mit Millionen Zeilen an Code, dennoch reichte seine Größe und die fehlende Testabdeckung aus, einen maßgeblichen Unsicherheitsfaktor bei der Weiterentwicklung mit einzubringen. So entstand das Bild eines Jenga-Turms in unseren Köpfen, bei dem man gar nicht so genau weiß, was als nächstes passiert, wenn man nur eine Kleinigkeit am Gesamtkonstrukt verändert.

Wir standen nicht unbedingt vor einem Scherbenhaufen, dennoch störte uns damals die fehlende Transparenz des Produkts. Man kann sagen, dass der Ist-Zustand nicht mehr zu unserer Vorstellung des Soll-Zustands oder zu unserer Produktvision passte.
Eines unserer ersten Live-Deployments verdeutlichte zusätzlich, dass sich diese Intransparenz nicht nur auf die Software bezog:

Wir deployten den Shop, nachdem er seine Tests und die zwei Stages (dev, integ) vor unserer Liveumgebung (prod) durchwandert hatte. Die Deployment-Pipeline im Jenkins war grün und wir zufrieden.
Komischerweise fehlte vom Feature, welches wir nun beim Live-Test erwarteten, jede Spur. Beim genaueren Überprüfen kam heraus, dass es beim Deployment zu einem Fehler kam. Die neue Version wurde nie ausgerollt, unsere Deployjobs allerdings grün – ein optisches Feedback für den Erfolg.

In einem Workshop beschlossen wir daraufhin, dass wir auf eine wesentlich wartbarere, klarer geschnittene Architektur auf Basis von Microfrontends setzen wollten. Zusätzlich wollten wir natürlich die Code Coverage sowie die gesamte Systemtransparenz durch stärkeres Monitoring/Tracking erhöhen.

Eine Frage blieb dabei: Konnte das alles funktionieren? Der tolino und seine Technologie waren für uns Entwickler damals noch eine Blackbox, die Herstellung gerade von der Deutschen Telekom auf die Firma Rakuten kobo gewechselt, die Kommunikation mit Rakuten kobo gerade erst im Aufbau.
Wir waren uns dennoch sicher: die Herausforderung müssen wir angehen.

Aus Groß mach klein

Der eReader-Shop hat im Grunde 5 Verantwortungsbereiche:

- Artikel (ersArtikel): Die Darstellung von Artikeldetailseiten und die Verwaltung von Artikelinformationen.
- Suchen und Navigieren (ersSuN): Die Darstellung von Navigations- und Suchergebnislisten
- Kunde (ersKunde): Die Darstellung des Kontobereichs/kundenspezifischer Informationen
- Kaufen (ersKaufen): Die Darstellung des Warenkorbs/Auftragserstellung über den eReader
- Redaktionelle Inhalte (ersInhalt): Die Darstellung von redaktionellen Inhalten

Auf Basis dieser Domänen wählten wir ebenfalls unseren Schnitt der späteren Microservices.
Dabei sollte jeder Microservice komplett in der Lage sein, unabhängig von den restlichen Services zu arbeiten, deployt zu werden und messbar zu sein.
Beim Surfen durch den Shop sollte der Kunde nun von Service zu Service ’springen‘, je nachdem, welches HTML er aus welcher Domäne gerade anfordert (siehe Abbildung 2).

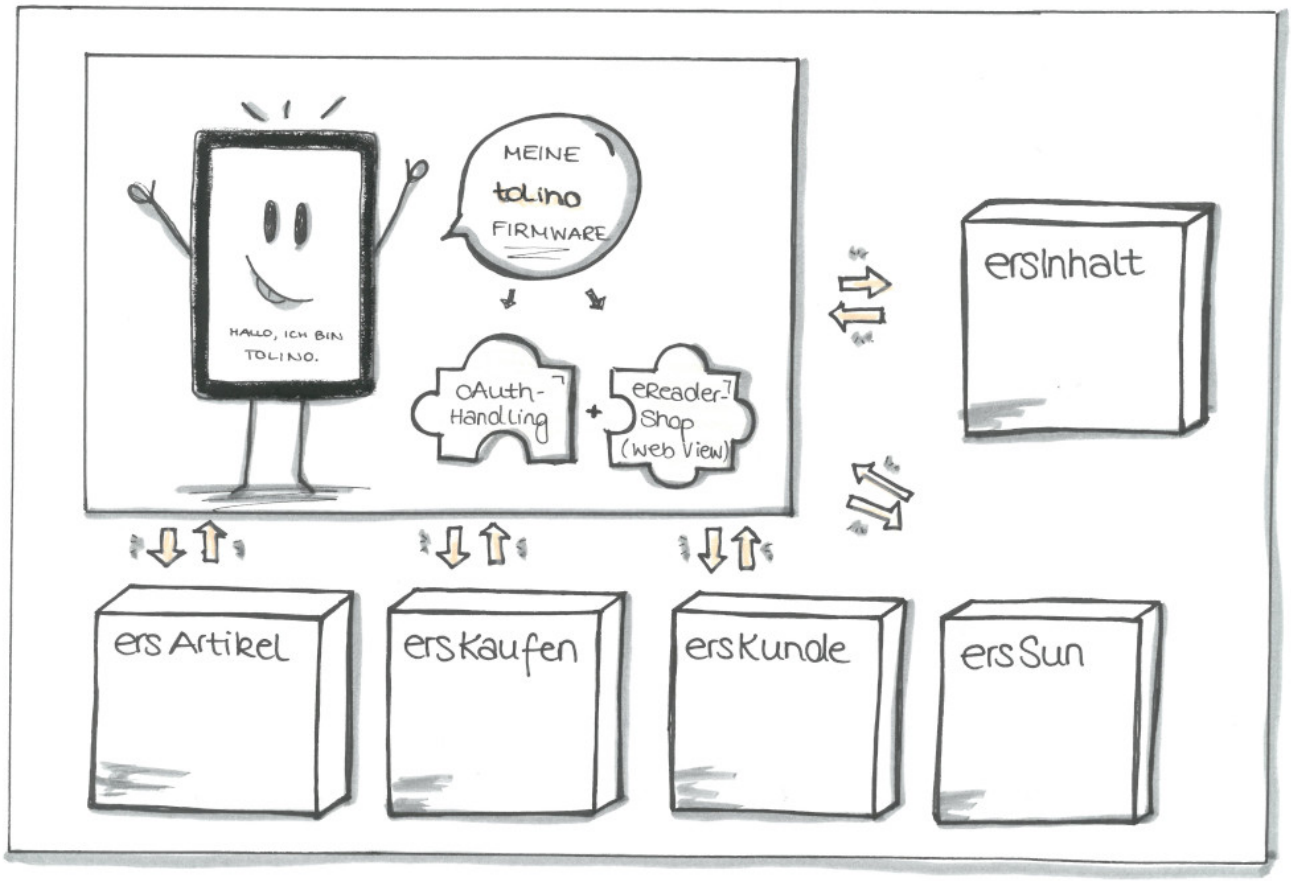


Abbildung 2

Die Zutatenliste

Architektur und Design

Seit April 2017 haben wir – immer den MVP (Minimum Viable Product) Gedanken im Hinterkopf – genau diesen Umbau vorgenommen.

Die Basis unserer Applikationen bildet Spring Boot (2) zusammen mit der Templating Engine Thymeleaf und dem Bootstrap Framework.
Die Microfrontends laufen auf Apache Tomcats und kommunizieren in den meisten Fällen via REST (zu beiden Punkten später mehr).

Da wir nicht auf der grünen Wiese anfangen konnten, haben wir nach und nach einen Bereich nach dem Anderen im eReader-Shop deaktiviert und sein Äquivalent als Microservice in Betrieb genommen (Brownfield Ansatz).
Der eReader-Shop-Monolith fungiert bei einer solchen Ablösung meist nur noch als Proxy.
Spricht man zum Beispiel die Startseite des eReader-Shops an, so landet man

initial auf den Tomcats des Monolithen, dessen Spring Controller daraufhin auf den Microfrontend-Service ersSuN weiterleitet.
Dies war nötig, da der Monolith eine interne Session aufbaut, die weiterhin von alten Funktionen benutzt wurde, die wir noch nicht in die neue Microservice-Welt überführt hatten (wir entschieden uns im April auf Basis der Testabdeckung, dass wir so wenig wie möglich in der Altsoftware verändern wollten – dies betraf auch das beschriebene Session-Handling).

Technisch gesehen koppeln wir unsere Microservices also via Direktlinkverlinkung, so dass der Kunde terminierend auf einem Webserver über den jeweiligen Inhalt von Microservice zu Microservice springt.
Mittlerweile haben wir ein weiteres, ebenfalls sehr sinnvolles Konzept eingesetzt: Das Konzept der Transklusion, in unserem Fall via SSIs (Server Side Includes). Hierbei werden zum Beispiel auf der eReader-Shop-Startseite Inhalte nicht mehr von unserem Microservice mittels Templates erstellt, sondern vom entsprechenden, verantwortlichen Team (OmniChannel Service Team) über unseren Webserver direkt ins HTML gerendert.
Die Verantwortung eines solchen HTML-Snippets liegt somit beim OCS-Team und nicht mehr beim eReader-Shop Team.
Wir haben mit diesem Konzept sehr gute Erfahrungen gemacht; dabei müssen allerdings bestimmte Rahmenbedingungen technisch, organisatorisch und qualitativ eingehalten werden.

Ein weiterer Faktor, der sich sehr verbessert hat, ist die Einbringung des Konzepts „Responsive Webdesign“, welches wir maßgeblich mit dem Bootstrap-Framework umgesetzt haben.
Hierdurch müssen wir nicht ein bestimmtes Styleset pro Gerät halten, sondern lassen dies dynamisch von Bootstrap verwalten und halten uns an ein Grid-Layout beim Erstellen unserer Seiten.
Im Monolithen mussten wir im Vergleich hierzu für jedes neue Gerät eine Sonderlocke und eigenes CSS einbringen – eigentlich eine total überflüssige und leider auch fehleranfällige Aufgabe.

Continuous Delivery & Qualitätsansprüche

Nun, wo wir ein solches Microfrontend Design und damit unabhängige Services etabliert haben, sind wir auch in der Lage, diese regelmäßig für unsere Kunden auszurollen.

Wir benutzen Deployment-Pipelines über insgesamt 3 Umgebungen (dev, integ, prod) mit Quality Gates auf jeder Ebene (siehe Abbildung 3).

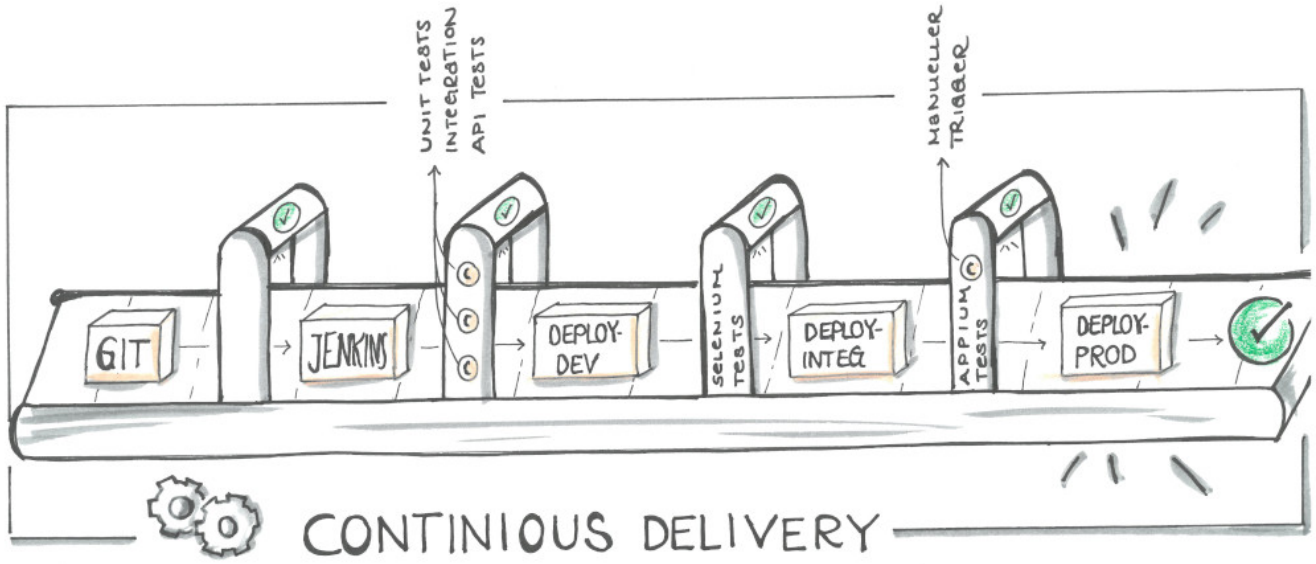


Abbildung 3

Technisch haben wir hierbei TestNG/JUnit Tests im Einsatz. Wir benutzen Mockito/MockMVC Tests um Integrationstest auf SpringBoot-Ebene durchzuführen. Beides machen wir zur Build-Zeit, also noch vor jedem Deployment.

Zusätzlich setzen wir einen SonarQube-Server ein, der uns hilft, die Testabdeckung zu prüfen, Code Smells zu finden und Checkstyle-Kriterien einzuhalten.

Ist alles in Ordnung, erfolgt das Deployment auf unsere erste Staging-Umgebung (dev). Wenn die Tomcats danach erfolgreich mit der neuen Version hochgefahren sind, folgen Selenium Tests (End-To-End Tests), die mittels PhantomJS (Headless) ausgeführt werden.

Falls auch hier alles in Ordnung ist, wird auf die nächste Umgebung deployt und es folgen nach denselben Kriterien weitere Selenium Tests, die nun allerdings schon in unserem Device Lab laufen.

Das Device Lab besteht aus derzeit 7 Geräten, die mittels eines Appium-Servers remote angesprochen werden können. Dafür können wir dasselbe Framework wie bei unseren dev-Tests benutzen (was sehr angenehm ist).

Man kann sich unsere Appium Tests so vorstellen, als würde ein Tester Regressionstests auf den Geräten selbst ausführen, nur dass dieser Tester hier der Appium-Server ist und alles vollautomatisiert erfolgt.

Sind auch diese letzten Tests erfolgreich, gibt es einen letzten, manuellen Trigger, um das getestete Artefakt auf unsere Prod-Umgebung zu deployen.

Ein so ausgerolltes Feature ist meist zusätzlich mit einem Featureschalter versehen, um zu verhindern, dass es unkontrolliert aktiviert wird.
In der Continuous Delivery ‚Welt' hat Feature Toggling unterschiedlichste Vorteile, die in diesem Blogbeitrag allerdings nicht behandelt werden können. An dieser Stelle reicht der Aspekt, dass wir die Kontrolle darüber haben möchten, wann ein Feature live – vor Kunde – aktiviert wird.

Fazit

Für uns bei Thalia und besonders im eReader-Shop-Team ist das hier beschriebene Experiment extrem erfolgreich gewesen. Wir sind innerhalb der letzten zwei Jahre zu einem schlagkräftigen Team zusammengewachsen.
Wir sind nicht nur schneller und effektiver im Ausrollen unserer Features geworden, sondern können dies nun auch noch mit einer wesentlich höheren Qualität, größerer Transparenz und starker Unabhängigkeit der einzelnen Services tun.

Einige Beispiele:

- Vor unseren Microfrontends haben wir unseren Monolithen etwa einmal im Sprint auf unsere Produktivumgebung deployt; eher seltener.
  Da wir keine konkreten, alten Daten mehr haben, hier einmal die Anzahl der Deployments in der Zeit von Oktober bis Dezember: Wir haben den eReader-Shop in dieser Zeit 5 Mal deployt.
  Unsere neuen Services wurden insgesamt 49 Mal auf die Produktiv-Umgebung deployt – und wir streben noch weitaus mehr Deployments an (kleineres Delta der Stände = weniger Komplexität, die live gebracht werden muss = höhere Entspanntheit).
  Ein Ziel, an dem wir arbeiten, ist es, jeden unserer Services mehrmals die Woche, vll täglich zu deployen. Somit würden wir ein minimales Delta an Änderungen zwischen integ und prod halten und jederzeit fähig sein, einzelne Bugfixes/Hotfixes oder Konfigurationen auf prod zu deployen.

- Der Einsatz von Featureschaltern verschlankt den Planungsaspekt von Livegängen. Auch wenn wir bei Thalia in der Softwareentwicklung einen

großen Fokus auf die Unabhängigkeit der Teams zueinander anstreben, kann es vorkommen, dass ein Feature in Verbindung mit einem anderen Schalter zum Beispiel im Thalia-Backend aktiviert werden muss.
Früher mussten wir hierfür ein zeitlich gezieltes Deployment vornehmen und hatten einen prozessualen und kommunikativen Zusatzaufwand.
Unter Einsatz der Featureschalter entfällt die Auslagerung des Aktivierens auf das Deployment und wir haben die Möglichkeit, zur Laufzeit ein Feature zu aktivieren.
Außerdem entkoppeln diese Schalter den Livegangsaspekt (Sichtbarkeit für den Kunden) vom Kleinhalten des Deltas der Änderungen zur prod-Umgebung. Über diesen Aspekt mag man gewiss unterschiedliche Ansichten haben, für uns hat es sich aber als sehr positiv herausgestellt, jederzeit deployen zu können.

- Der Code unserer bereits livegestellten Services hat eine Testabdeckung von 78% und gibt uns bei der Weiterentwicklung, bei Updates oder bei Livegängen ein sehr gutes Qualitätsgefühl. Zusätzlich decken unsere Selenium- und Appium-Tests für alle neuen Services diverse Sicht- und bereits automatisierte Regressionstests ab.
  Durch den Grad der Testautomatisierung können wir unseren Fokus beim manuellen Testen von klassischen Regressionstests auf gezieltere Edge-Case-Tests, sowie dedizierte Feature-Tests und Exploration-Tests lenken.
  Dies erhöht unsere Geschwindigkeit bei der Feature-Weiterentwicklung maßgeblich und bringt sehr viel mehr Sicherheit in die Softwareentwicklung des Teams.

- Unsere Deploymentdauer bis zur prod-Umgebung hat sich insgesamt ein wenig verlängert.
  Wir haben unsere Kette grundsätzlich beibehalten und deployen auf zwei Umgebungen bevor wir danach auf prod deployen.
  Was neu hinzugekommen ist, sind die automatisierten Schritte zwischen unseren Deployments: Wir prüfen nach jedem Deployment, ob die Tomcats einen ‚health'-Status zurück geben und die erwartete Version deployt wurde, und testen das Ganze mit den oben beschriebenen Selenium- und Appium-Tests.
  Weiterhin wird bei jedem Deployment auf die Umgebung direkt vor prod die Anzahl der Changes – also unser Delta zwischen integ und prod

ausgegeben.
Diese Zwischenschritte führen dazu, dass wir mehr Gewissheit darüber haben, was live gehen wird, und bereits die Testabdeckung kennen.
Treffen wir die Entscheidung, mit einem getesteten Feature live gehen zu wollen, geschieht dies in den meisten Fällen in etwa 40 Minuten.
Früher hat dieser Vorgang noch mehrere Stunden gedauert, da natürlich alle nun automatisierten Testschritte noch einmal in Kurzform manuell vorgenommen werden mussten, was letzten Endes die Gesamt-Deploymentdauer hinausgezögert hat.

Hier endet die erste Etappe unserer Reise, und wir möchten uns schon einmal bei euch bedanken, so geduldig mitgelesen zu haben!

XPdays: DevOps ist keine Rolle, oder?

Am 8.11.2018 haben wir auf den XPdays unsere Erfahrungen mit der Einführung einer Produkt-Organisation aus Sicht von IT-Operations geteilt.

Was passiert eigentlich mit einem klassischen IT-Betrieb, wenn ein Unternehmen sich für eine Produktorganisation entscheidet? Eines ist klar: Alles verändert sich!

In diesem Beitrag wollen wir euch auf unsere spannende Reise vom IT-Betrieb hin zu Platform Engineering und DevOps mitnehmen. Woher kommen wir? Wie haben wir uns technologisch, kulturell, prozessual und als Team verändert um optimal mit und in agilen Produkteams zusammenzuarbeiten? Wie unterstützen wir unsere Entwicklungsteams als zentrales Plattform Team aber auch als Mitglied eines crossfunktionalen Produktteam? Was passiert mit unserem Mindset „Never change a running system“ in einer Welt mit 1.000+ Deployments pro Monat? Wie haben wir uns organisiert und wie versuchen wir täglich aufs Neue das Beste aus einer agilen Produktentwicklung herauszuholen? Wie verstehen wir DevOps? Ist das, was früher mal der IT-Betrieb war, auch zu einem crossfunktionalen Produktteam geworden?

https://www.xpdays.de/2018/sessions/150-devops-ist-keine-rolle-oder.html

Natürlich ist es nicht möglich alle Erfahrungen in einen kurzen Betrag zu packen. Uns hat es aber viel Spaß gemacht unsere Erfahrungen zu teilen und haben uns sehr über die vielen Rückfragen und Gespräche zum Thema gefreut. Sprecht uns auch gerne weiterhin an, wenn ihr euch für das Thema interessiert.

Christoph Drosten

c.drosten@thalia.de
https://www.xing.com/profile/Christoph_Drosten

Matthias Wellmeyer

m.wellmeyer@thalia.de
https://www.xing.com/profile/Matthias_Wellmeyer

Hier die Folien zum Beitrag:

Unseren gesamten Reisebericht findet ihr hier:

Vom IT-Betrieb zu Platform Engineering. Ein Reisebericht (1/3)

Vom IT-Betrieb zu Platform Engineering. Ein Reisebericht (2/3)

Vom IT-Betrieb zu Platform Engineering. Ein Reisebericht (3/3)

Akzeptanztests bei Thalia

Im Team „Kaufen“ und im Team „Kunde im Mittelpunkt“ setzen wir seit über einem Jahr automatisierte Akzeptanztests zur Sicherung unserer Qualität ein. Beide Teams sind begeistert von diesem Vorgehen. Höchste Zeit für einen Beitrag in unserem Techblog

Was ist ein Akzeptanztest?

Ein Akzeptanztest ist ein funktionaler Test, der das Benutzerverhalten beschreibt, um User Story und Akzeptanzkriterien zu überprüfen.
Dabei steht die Sicht des Benutzers im Vordergrund. Es ist wichtig nicht irgendwas zu testen, sondern genau das, was wir mit einer User Story umsetzen wollen. Durch die Akzeptanztests wird sichergestellt, dass die Software aus Sicht des Benutzers wie gewünscht funktioniert.

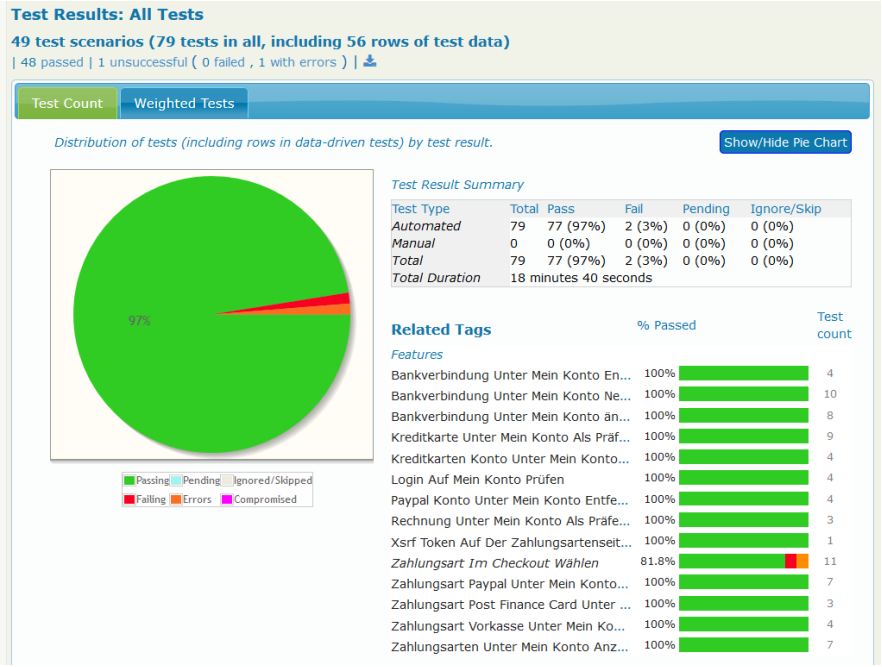
Ein Akzeptanztest besteht aus drei Teilen: Vorbedingungen (Angenommen), Aktionen die durchgeführt werden (Wenn) und erwartete Ergebnisse (Dann). Diese sogenannten Szenarien werden später zu automatisierten Tests auf verschiedenen Ebenen der Testpyramide. Zur Definition der Szenarien verwenden wir in unseren Teams das Framework Cucumber.

Beispiel eines einfachen Szenarios:

```
Szenario: Anzeige der Zahlungsarten unter mein Konto mit Rechnung als präferierte Zahlungsart
  Angenommen es existiert ein angemeldeter Kunde bei Thalia.de
   Angenommen der Kunde hat die präferierte Zahlungsart Rechnung
  Wenn dieser Kunde die Seite Mein Konto Zahlungsarten aufruft
  Dann ist das Akkordeon an der Stelle Rechnung aufgeklappt
```

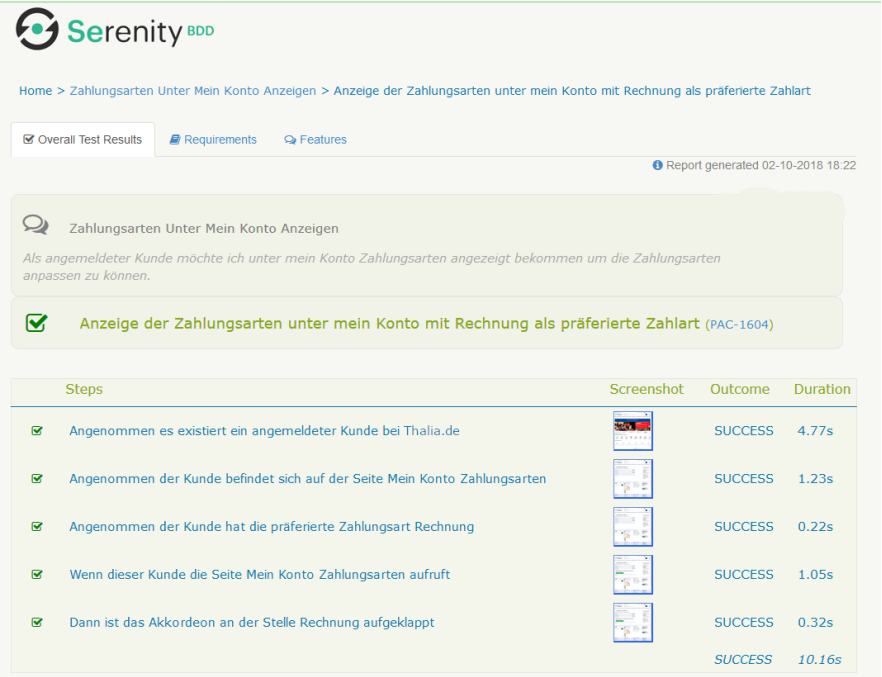
Testreport mittels Serenity

Bei jedem Testdurchlauf wird automatisch mittels Serenity ein Bericht mit den Testergebnissen erstellt.



Serenity Report – Übersicht

Zu jedem Szenario können die einzelnen Testschritte inklusive Screenshots betrachtet werden. Im Fehlerfall sieht man somit direkt, welcher Testschritt fehlgeschlagen ist.



Serenity Report – Übersicht

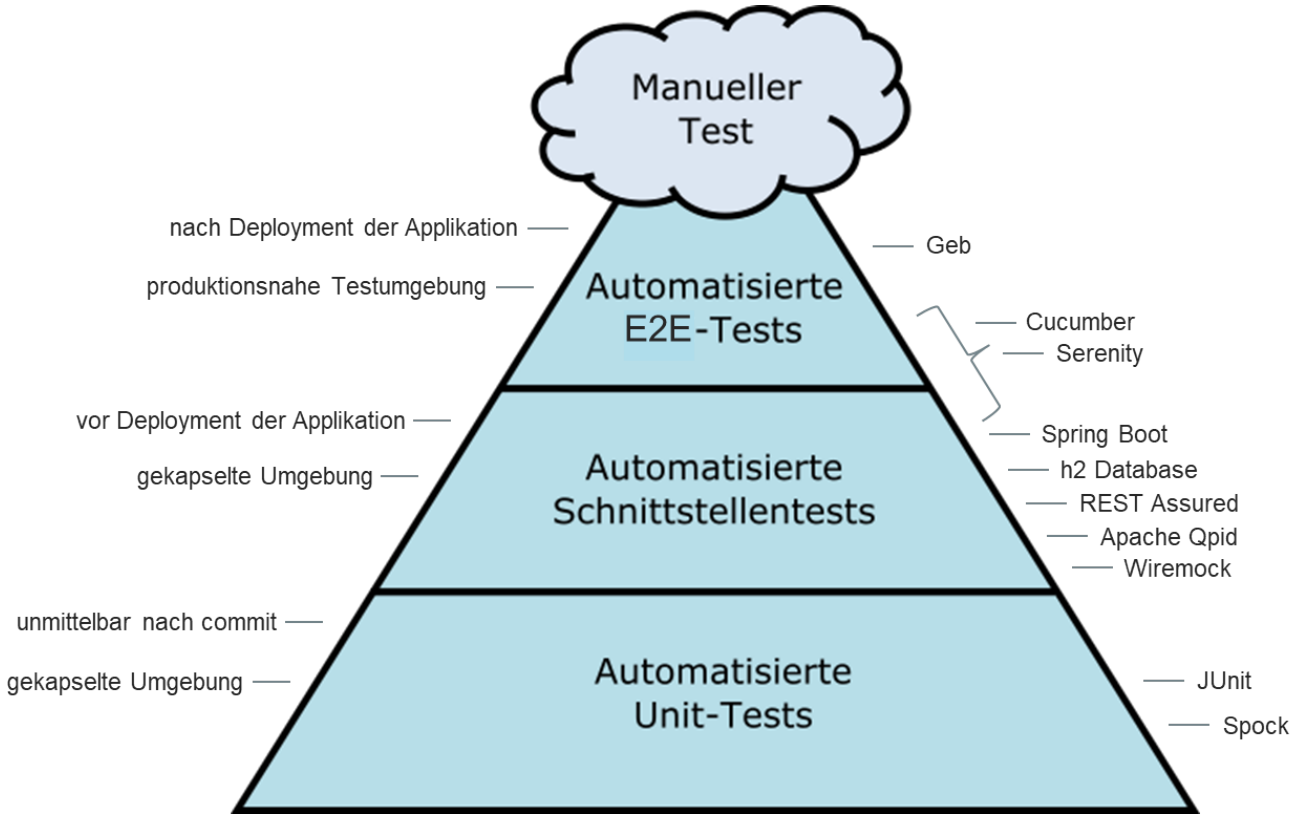
Zielbild Testpyramide

Unser Zielbild bei der Qualitätssicherung im Team „Kaufen“ orientiert sich an der

klassischen Testpyramide. Wir wollen unsere Fehler natürlich möglichst früh und möglichst vor dem End-to-End-Test finden. Da wir allerdings gerade dabei sind unser Frontend abzulösen und unseren Checkout neu zu gestalten, schreiben wir im Moment relativ viele End-to-End-Tests.
Um dem entgegen zu wirken, begeben wir uns als nächstes auf die Suche nach einem geeigneten JavaScript-Test-Framework. Wir erhoffen uns dadurch mögliche Fehler im Frontend bereits auf Unit-Test-Ebene aufzudecken und ggf. sogar einzelne End-to-End-Tests ablösen zu können.

Welche Frameworks verwenden wir und wann werden die Tests ausgeführt?

Die gleichen Szenarien nutzen wir sowohl für die automatisierten End-to-End-Tests, als auch für die Schnittstellentests.



Die automatisierten End-to-End-Tests erfolgen nach dem Deployment in einer produktionsnahen Testumgebung. Dabei verwenden wir das Framework Geb, welches den Selenium WebDriver nutzt und der Browser-Automatisierung dient. Die Tests werden hierbei in der Programmiersprache groovy geschrieben.

Durch die Schnittstellentests soll die jeweilige Applikation inklusive Datenbank- und RabbitMQ-Prüfungen (wo notwendig) zur Laufzeit getestet werden. Um möglichst unabhängig zu sein, erfolgt der Test in einer gekapselten Umgebung. Dazu wird die Applikation mittels h2 Database, Apache Qpid und Spring Boot gestartet, so dass die Schnittstellentests zwar zur Laufzeit , aber noch vor dem Deployment ausgeführt werden können. Fremdsysteme werden dabei mit Wiremock simuliert. REST Assured wird zur Durchführung von HTTP Requests und Validierung der HTTP Response verwendet.

Welche Vorteile bringen uns die Akzeptanztests?

Die Akzeptanztests helfen uns offene Punkte und Ungereimtheiten aufzudecken und mit dem Product Owner zu besprechen.

Aktuell werden im Team „Kaufen“ die Szenarien meistens durch eine Person aus dem Dev-Team kurz vor oder zeitgleich zur Implementierung geschrieben. Dies kann, muss aber nicht, die QAlerin des Teams sein. Wie beim Code-Review erfolgt auch hier ein Review durch eine zweite Person des Dev-Teams.

Noch besser wäre es die Szenarien bereits vor dem Start der Software-Implementierung zu schreiben. Optimalerweise sogar in der „3 Amigos“ Konstellation, welche aus Product Owner, QAlerin/Analyst und Entwickler besteht, um die Anforderung aus unterschiedlichen Blickwinkeln zu betrachten. Dadurch können Ungereimtheiten und Komplexitätstreiber bereits während der Anforderungsanalyse aufgedeckt und besprochen werden. Bisher fehlte uns im Team „Kaufen“ leider die Zeit / der Vorlauf diese Variante auszuprobieren. Wir sind aber zuversichtlich, dass es dazu eine Gelegenheit geben wird

Im Team „Kunde im Mittelpunkt“ wird bei der Erstellung der Szenarien bereits mit der Product Ownerin und der Fachseite zusammen gearbeitet. Dabei wurden bisher sehr gute Erfahrungen gemacht, da anschließend alle relevanten Personen den gleichen Wissenstand auf einer formalen Ebene haben. Siehe auch hybride Testkonzeption.

Es gibt weitere Vorteile, die uns als Dev-Team zugutekommen.
Wir haben es geschafft die Teamsicht auf unsere Tests zu verbessern. Alle im Dev-Team schreiben Szenarien und implementieren die daraus resultierenden automatisierten Tests. Die Akzeptanztests sind Teil des Repositories der jeweiligen Applikation, werden in der Entwicklungsumgebung geschrieben und

bei Bedarf lokal ausgeführt. Durch die Verknüpfung zwischen Fachlichkeit und automatisierten Tests in Form der Szenarien, sind die Auswirkungen bei fehlgeschlagenen Testfällen außerdem direkt sichtbar und helfen uns so bei GoLive-Entscheidungen.

BDD + DDD = eine hybride Testkonzeption

Das agile Produktteam „KiM“, verwendet in Bezug auf die Qualitätssicherung seiner Produkte zwei wesentliche Ansätze, die wir im Rahmen dieses Blogs kurz skizzieren wollen. Dabei geben wir nicht nur einen konzeptionellen Einblick, sondern stellen auch aus unserer Sicht die Vorteile für die Anwendung eines hybriden Ansatzes heraus.

Behaviour Driven Development

Beim Behaviour Driven Development steht das erwünschte Verhalten der Software aus Sicht des Kunden im Vordergrund. In der Anforderungsanalyse werden die Ziele, Aufgaben und Ergebnisse der Software in Textform als Szenarien festgehalten. Die Szenarien werden unter Verwendung des „Given-When-Then“ – Konzepts erstellt (siehe auch: Akzeptanztests bei Thalia) und dienen als Basis für die Entwicklung automatisierter Tests. Somit wird bei den automatisierten Tests stark die Perspektive des Kunden eingenommen.

Dieser Ansatz bietet nicht nur den Vorteil der einfachen Lesbarkeit, sondern auch die Möglichkeit alle beteiligten Akteure (wie bspw. den Product Owner, die Fachseite, etc.) während der Testfallerstellung mit einzubeziehen und dadurch ein gemeinsames Verständnis vom System bzw. der Software aufzubauen.
Ein mögliches Beispiel für ein Szenario:

```
Szenario: Benutzer meldet sich über die Mobile-App an
Angenommen "Max Mustermann" benutzt die Mobile-App
```

```
Wenn "Max Mustermann" sich während des Bestellvorgangs erfolgreich anmeldet
Dann soll die Checkout-Seite angezeigt werden
Und "Max Mustermann" ist in der Mobile-App angemeldet
```

Ein weiteres Element, welches beim BDD-Konzept das gemeinsame Verständnis stärkt, ist der Einsatz einer „ubiquitous language“.
Die „ubiquitous language“ dient als gemeinsam definiertes und genutztes Vokabular, wodurch Fehlinterpretationen bei der Verwendung von Begriffen minimiert werden. Beispielsweise definieren Fachbereiche wie das CRM oder die Softwareentwicklung Begriffe wie „Kunde“ oder „Auftrag“ unterschiedlich. Der Kunde könnte zum einen als Kunde interpretiert werden, sobald er sich im Webshop eingeloggt hat oder zum anderen wenn er mindestens einen Auftrag durchgeführt hat. Dieses heterogene Verständnis kann zu Problemen bei der Testfallerstellung führen, beispielsweise durch fehlende oder kontroverse Testfälle.

Die eingenommene Perspektive beim BDD-Ansatz offenbart allerdings auch einige Schwächen, die im folgenden beschrieben werden und weshalb sich das Team KiM zu einem Einsatz eines weiteren Konzepts entschieden hat: Dem Domain-Driven Design.

Domain-Driven-Design in der Praxis

Um den eingenommenen Blickwinkel des Endkunden durch den BDD Ansatz im Softwareentwicklungsprozess zu erweitern und mögliche Grenzen zwischen IT und Business aufzuweichen, kombiniert das Team KiM während der Anforderungsanalyse den Behaviour-Driven-Development mit dem Domain-Driven-Design (DDD) Ansatz. Ausgehend vom Business eines Unternehmens, bis hin zu einer fachlichen Domäne (wie bspw. Vertrieb, Buchhaltung, CRM, etc.) bietet das strategic Design des DDD Ansätze zur Strukturierung und Aufteilung der Domäne. Das strategic Design, als einer- von vier Bestandteilen des DDD-Konzeptes, steht bei Team KiM im besonderen Fokus, da sich hier sehr schnell ein Mehrwert generieren lässt. Aufbauend auf der ubiquitous language bietet das strategic Design u.a. folgende Techniken bzw. Konzepte:

- Event Storming
- bounded Context
- Context Map

Event Storming

Nach der Bestimmung der Domäne können anhand des Event Storming die jeweilig relevanten Events innerhalb einer Domäne identifiziert und zugeordnet werden.
Am Beispiel eines sog. „Newsletter“-Projektes“ hat das Team KiM in enger Zusammenarbeit mit Product-Owner und dem Fachbereich die Events in der Domäne ermittelt und das Ergebnis festgehalten:

Event Storming – Newsletter

bounded Context

Die im Event Storming ermittelten Events lassen sich in bounded Contexts (sog. eigenständige Modelle) clustern. Folgende bounded Contexts wurden festgehalten:

- Kontaktverwaltung
- Registrierung
- Newsletter (An/-Abmeldung)
- Gewinnspiel
- Präferenzen

Der bounded Context (s. Bild unten) bietet leider noch keinen brauchbaren Systemüberblick unter Berücksichtigung des Kommunikations-, Daten- und Modellflusses. Diesen Part übernimmt die Context map, die den Zusammenhang zwischen Model und dem Kontext skizziert.

Context Map

Die Context Map bietet einen guten Ausgangspunkt für die nachfolgende Softwareentwicklung und die Testfallerstellung. Am Beispiel des Newsletter-Projektes hat das Team KiM folgende context map erstellt:

Context Map – Newsletter

Die Domain Events aus dem Event-Storming werden den einzelnen bounded contexts (blaue Post-its) zugeordnet. Darüber hinaus ist der Daten- und Modellfluss anhand von Pfeilen (rot + schwarz) skizziert.
Die Abkürzungen CF, OHS, SUP und CUS stellen sogenannte context map patterns dar, mit der Bedeutung:

- CF = Conformist (Das Downstream Team übernimmt das Modell des Upstream Teams)
- OHS = Open / Host Service (der Host bietet einen vordefinierten Satz an Services an)
- SuP = Supplier (Supplier-Customer i.S.v. Kunde-Lieferantenbeziehung)
- CuS = Customer (Supplier-Customer i.S.v. Kunde-Lieferantenbeziehung)

Die erstellte Context Map bündelt nicht nur die Ergebnisse der hier vorgestellten Techniken, sondern unterstützt zugleich die Testfallentwicklung. Hierbei hilft nicht nur die Visualisierung des Systems, sondern auch die Gruppierung der Events und die Modellzuordnung. Dadurch lassen sich präzisere bzw. eindeutig definierte Testfälle erstellen.

Vorteile

Den hier skizzierten hybriden Ansatz hat das Team KiM bereits erfolgreich in den Softwareentwicklungsprozess integriert und aufgrund folgender Vorteile zu schätzen gelernt:

- Berücksichtigung mehrerer Perspektiven (Business und Endkunde)
- Integration der Testfallentwicklung
- Beteiligung aller relevanten Akteure am Entwicklungsprozess
- Reduzierung von Fehlinterpretationen bei Sachverhalten/Begrifflichkeiten (ubiquitous langauge)
- Reduzierung des Testaufwandes durch Testautomatisierung

Abschließend zu diesem Blog stellen sich dem Team die Fragen:

1. Wie ist eure Meinung zu dem o.g. Vorgehen ?
2. Welche Erfahrungen habt ihr mit dem BDD- bzw dem DDD-Ansatz gemacht?

<#0100/> <hackathon@thalia/> in Berlin

Ende Mai fand der 2. Hackathon in Berlin statt. Zum Thema waren diesmal alle neuen Technologien gesetz.

<#0101/> <hackathon@thalia/> in

Münster

<#0101/> <hackathon@thalia.de/> „Acceleration Technologies“

… fand am 8.6. in Münster statt. Was beim letzten Mal gut funktioniert hat, wollten wir beibehalten, weshalb wir uns wieder fernab des Tagesgeschäfts in einem anderen Gebäude eingerichtet haben. Kulinarisch/mit Erfrischungen hochgerüstet und nach diversem Dosen-Gepatche einsatzbereit haben wir dieses Mal unter dem Motto „Acceleration Technologies“ zu spannenden Projektvorschlägen aufgerufen. Drei Projekte haben sich daraufhin formiert und konnten passend 16 Uhr Ergebnisse vorzeigen – wenn auch anders als antizipiert…

Projekt „OpenStack Explorer“

Ich weiß noch nicht ganz, ob Christoph mit dem Logo zum Projekt schon absichtlich den Teaser zum nächsten Mal rausgeworfen hat, aber was das Selbstmarketing des Projektes angeht, ist das schon weit vorne.
Nach dem Pitch haben sich 6 Personen (stark betriebslastig) für dieses Thema zusammengeschlossen.

Worum ging es?

„Ich möchte OpenStack und seine Möglichkeiten kennen lernen. Ich verspreche mir davon eine Menge an coolen Features die unser Leben erleichtern können.“

- Was kann OpenStack?
- Können wir mit OpenStack schneller werden?
- Passt OpenStack zu unseren Prozessen und Infrastruktur?

Ergebnis

OpenStack verspricht die „Cloud für zuhause“, quasi „infrastructure as a service“. Das Team ist sowohl auf virtuellen Maschinen wie auch auf Hardware in das Thema gestartet. Es wurde eine Anleitung für eine all-in-one-cloud auf einer Maschine befolgt, die zur Konsequenz hatte, dass Puppet 20 Minuten lang Zeug nachinstalliert hat und etwa 1300 Variablen zu setzen waren. Da war dann alles dabei, Firewall, Netzwerk, alles.

Das Team konnte Maschinen hochfahren mit block storage, image service … 6-7 Kernelemente, die GUI dazu. Als Fazit kamen die OpenStack Explorers zu dem sehr klaren Ergebnis, dass diese Technologie eher was für große Internet Provider oder die NASA ist, aber nicht passend für uns. Für Thalia Anwendungsbereiche ist OpenStack unnötig komplex.

Projekt „GitLab Explorers“

Worum ging es?

In diesem Projekt sollte GitLab CI evaluiert werden. 2 Personen aus dem Entwicklungsbereich überprüften Merging, Builds, das Zusammenspiel mit Jenkins – und vor allem: Was unterscheidet diese Lösung von unserer bisher im Einsatz befindlichen?

Ergebnis

Die Ergebnisse wurden anhand des Beispiels „checkout service“ präsentiert. Das Fazit der beiden ist klar pro GitLab CI!
Das Ganze funktioniert komplett „on-premises“, jedes Team kann sich aussuchen, ob es das mit oder ohne Jenkins einsetzen will. „Geht alles!“

Projekt „Our Vault“

Worum ging es?

Drei Teilnehmer hatten sich dazu entschieden, den Hashicorp Vault auf seine Eignung für Thalia-eigene Anwendungsfälle im Vergleich zur heutigen Lösung zu testen. Kurz gesagt geht es bei diesem Werkzeug um die „sichere“ Ablage von Passwörtern für Anwendungen.

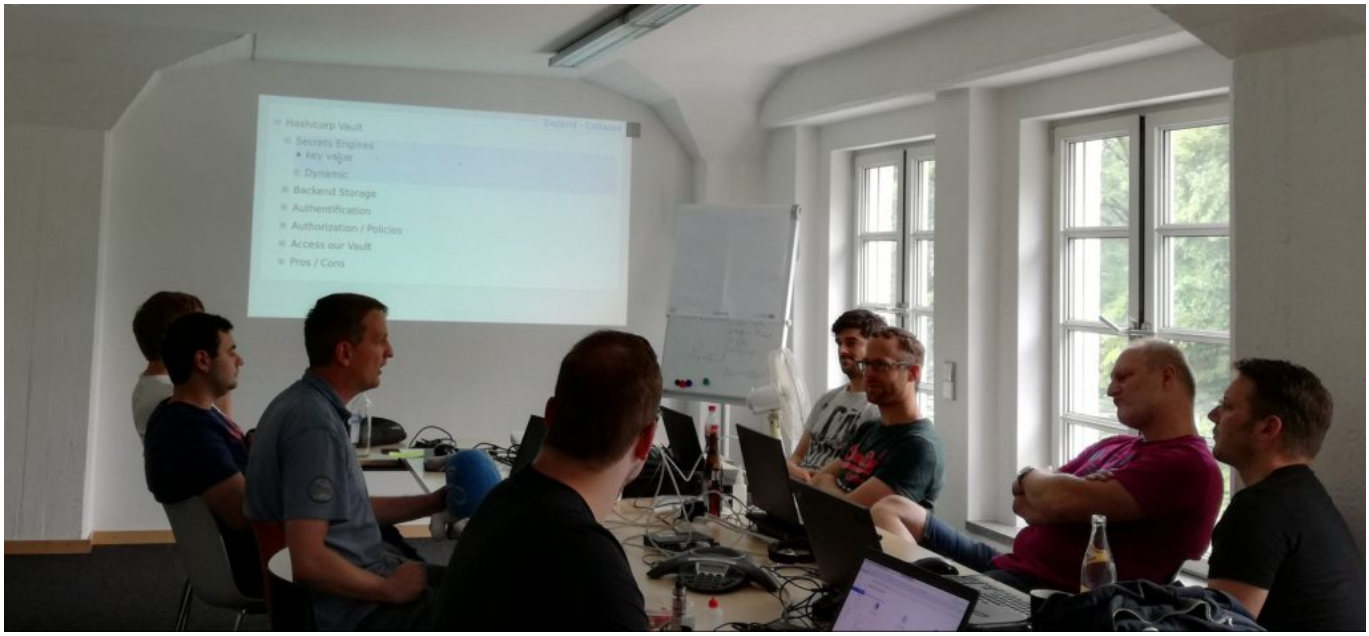
Ergebnisse

Das Team hatte sich dazu entschlossen, zweigleisig zu fahren. Auf der einen Seite wurden gängige Anwendungsfälle aus unserem Dunstkreis gegen das Produkt gehalten und umgekehrt der Featurekatalog des Vault analysiert. Auf der anderen Seite wurde eine Einbindung in unsere Infrastruktur durch zwei Entwickler umgesetzt.

Hierbei zeigte sich, dass Vault weit mehr als nur einige key/value Paare hält. Die Anzahl der Backend Storage Möglichkeiten ist hoch, diverse

Authentifizierungsmöglichkeiten sind gegeben. Der Zugriff kann über Kommandozeile oder http/REST erfolgen, eine UI steht zur Verfügung. Die Verwendung von Spring Cloud Vault ermöglichte den Entwicklern das Ansteuern des Vaults mit Spring-Mitteln.

Als Fazit darf gesagt werden: da geht viel, es bringt mehr Sicherheit, dadurch dass man mehr Keys benutzt, die Integration ist sehr einfach (Kubernetes, AWS). Aber: der erzeugte „single point of failure“ ist ein Problem, das Team sah den Nutzen nicht hoch genug, um hier noch weitere Schritte Richtung Vault zu unternehmen.



Fazit

Auch dieses Mal können wir wieder Erkenntnisse aus den Projektgruppen rausziehen – diesmal auch sowas wie „ungeeignet für uns“.

Ich möchte an dieser Stelle exemplarisch für die teilweise fantastische Kühlkette mit folgendem Stimmungsbild schließen. Bis zum nächsten Mal!

Chaos Monkey Tag – Geplanter Wahnsinn

- Ein Kollege rennt über den Flur
- Telefone klingeln
- Hektische / Laute Gespräche
- Es stehen mehrere Kollegen aus dem IT-Betrieb bei einem Entwickler
- …

Alles keine guten Zeichen. Irgendwas ist passiert.

- Ist der Shop noch da?
- Oje…
- Ist da wohl schon wer dran?
- Haben wir gerade deployt?
- Was zeigt denn unser Monitoring?
- Wo stehen nochmal die Logs?
- …

Klingt nach eher hektischen Reaktionen, die darauf hinweisen, dass das Handwerkzeug noch nicht 100% sitzt.

Die gute Nachricht: Auch solche Verhalten können trainiert werden! Notfälle müssen nicht erst abgewartet werden!

Bei Thalia arbeiten wir auf allen Ebenen an der Verbesserung. Dazu gehört auch der Umgang mit Störungen in unseren Produkten.
Warum? Weil wir besser werden wollen!

Inspect & Adapt in Incident-Fällen

Die Grundlage für unsere hier vorgestellte Maßnahme stammt aus einer „Retrospektive“. Dieses Format kennt ihr vielleicht bereits aus „Scrum“. Wir nutzen es bei Thalia in vielen Situationen. In den vergangenen Monaten hatten wir an der ein oder anderen Stelle Probleme bei der Bereitstellung unserer Produkte im Online-Shop. Anschließend kristallisierte sich meist ein Kreis von Hauptbetroffenen heraus, mit dem wir gemeinsam reflektiert haben, wie die Zusammenarbeit funktioniert hat.

Hierbei trat immer wieder die Wahrnehmung in den Vordergrund, dass unterschiedlich mit Störungen umgegangen wird. Die schlimmsten Anzeichen dafür waren:

- Hektik
- Ignoranz
- keine Kommunikation
- Verzweiflung

Wir beschlossen als Maßnahme, solche Situationen gezielt herbeizurufen, um den

Umgang zu üben. Dabei sollte auch herauskommen, wo unsere Teams überall noch Luft zur Verbesserung haben.

Chaos-Monkey-Tag

Offiziell stand dieser Tag ausgeschrieben als „Notfallübungstag“, doch intern wurde schnell der Begriff „**Chaos-Monkey-Tag**“ ausgerufen. „Chaos Monkey“ ist ein Tool, das als einzige Aufgabe hat, die IT-Infrastruktur von Unternehmen zu penetrieren. Es wird dort eingesetzt, wo Schwachstellen aufgedeckt werden müssen, um die Produkte nachhaltiger zu entwickeln. In unserem Falle haben wir hier nicht die Software installiert, sondern zunächst die Arbeit manuell gemacht. Dafür haben wir Mitarbeiter gesucht, die Zugriff auf alle Systeme haben und entsprechende Kreativität mitbringen diese zu ärgern: „Die Monkeys“! Diese waren schnell gefunden und noch schneller die Maßnahmen, die zu einem „Notfall“ führen.

Das Ergebnis tracken

Für den ersten Versuch haben wir uns geeinigt die Reaktionen in den Teams zu messen. Um sicherzustellen, dass die Behebung nicht wieder von „den üblichen Verdächtigen“ getan wird, also denen, die sich sowieso immer kümmern, haben wir diese im Vorfeld gefragt, ob sie das „Tracking“ übernehmen. Auf der einen Seite wissen diese Person am besten worauf man achten muss und auf der anderen Seite konnten wir so besser sehen, was die anderen machen. Die Tracker haben an diesem Tag für ihr Team die Messung vorgenommen. Selbst sollen sie sich bei der Lösung aktiv nicht beteiligen, aber zur Lösungsfindung natürlich bereit stehen. In erster Instanz war uns wichtig die Reaktionsgeschwindigkeit festzuhalten: Wann und wie erscheint das Problem an der Oberfläche?

- Springt das Monitoring an?
- Steht ein Kollege im Büro?
- Teilt am Ende sogar der Tracker selbst mit, dass das Team aufwachen sollte?
- Ist das Team ansprechbar? Oder treiben sich alle in einem dieser wichtigen Refinements oder Retrospektiven herum?

Was waren unsere Störungen

Der Kreativität sind keine Grenzen gesetzt. Nach einem kurzen Brainstorming haben wir uns für die Vergleichbarkeit auf folgende Maßnahmen geeinigt:

1. Datenbank-Instanzen in kurzen Abständen alle auf „Read-Only“ konfigurieren, bis keine mehr zur Verfügung steht.
2. Herunterfahren von Application-Server Instanzen, bis keine mehr zur Verfügung steht.
3. Fehlkonfigurationen der Webserver aktivieren.

Was nehmen wir mit

Die Notfälle wurden alle behoben, aber in Summe haben wir festgestellt, dass noch Luft nach oben ist. Im Vergleich waren die Reaktionszeiten der Teams sehr unterschiedlich und teilweise zu langsam.

Spätestens in den folgenden Sprint-Retrospektiven der Teams wurden Maßnahmen verabschiedet, um das Monitoring oder sogar die Produkte selbst zu verbessern.

Und als Organisator:

Auch wenn der Tag natürlich wieder vom Zeitpunkt „völlig falsch“ gewählt war wurden die Maßnahmen sehr begrüßt und einem Folgetermin – auch in Serie – wollte keiner im Wege stehen.

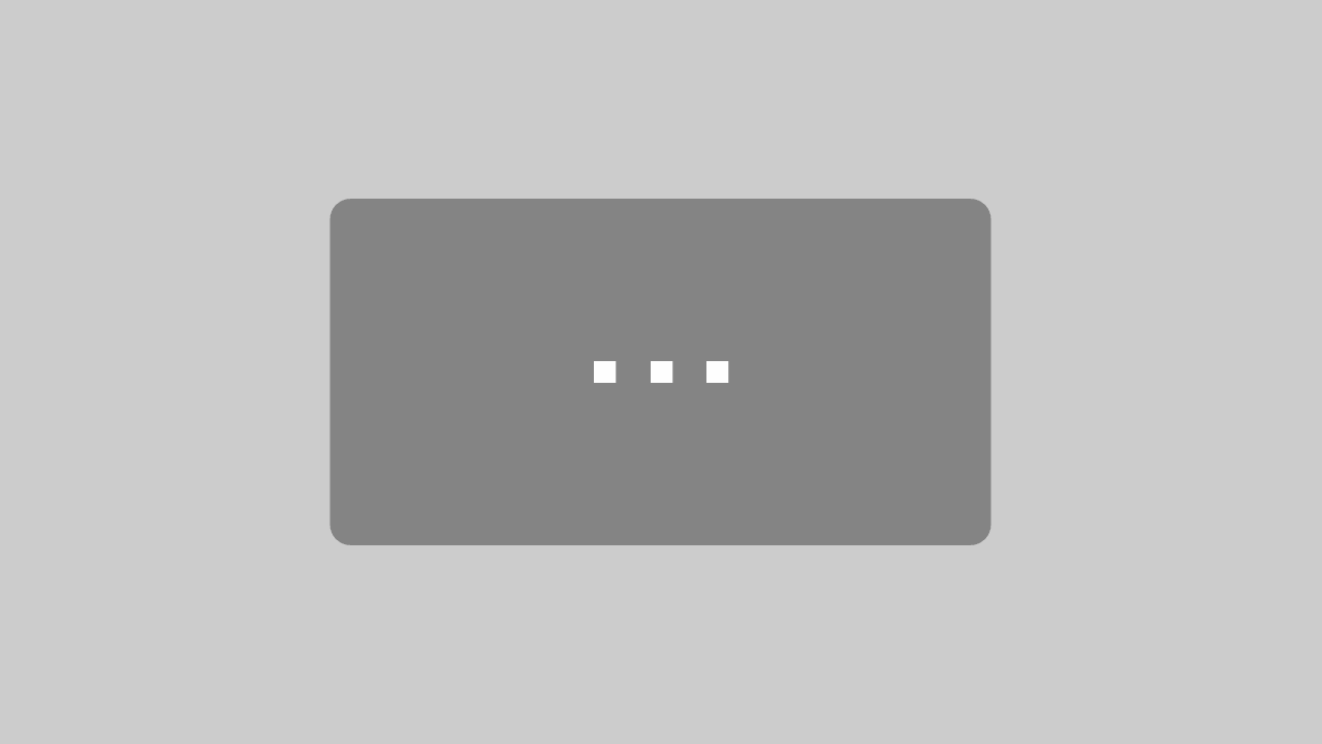
---

microXchg: Reconstructing thalia.de with self-contained

systems

Am 22. und 23.03.2018 waren wir in Berlin auf der microXchg 2018. Neben dem Besuch verschiedener Vorträge und dem Austausch mit den anderen Teilnehmern hatten wir zudem die Möglichkeit, selbst den Vortrag *Reconstructing thalia.de with self-contained systems* beizusteuern.

Hier der Beitrag auf YouTube



Mit dem Laden des Videos akzeptieren Sie die Datenschutzerklärung von YouTube.
Mehr erfahren

Video laden

☐ YouTube immer entsperren

Hier die Folien zum Vortrag:

Benjamin Leenen

Dennis Schneider