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).

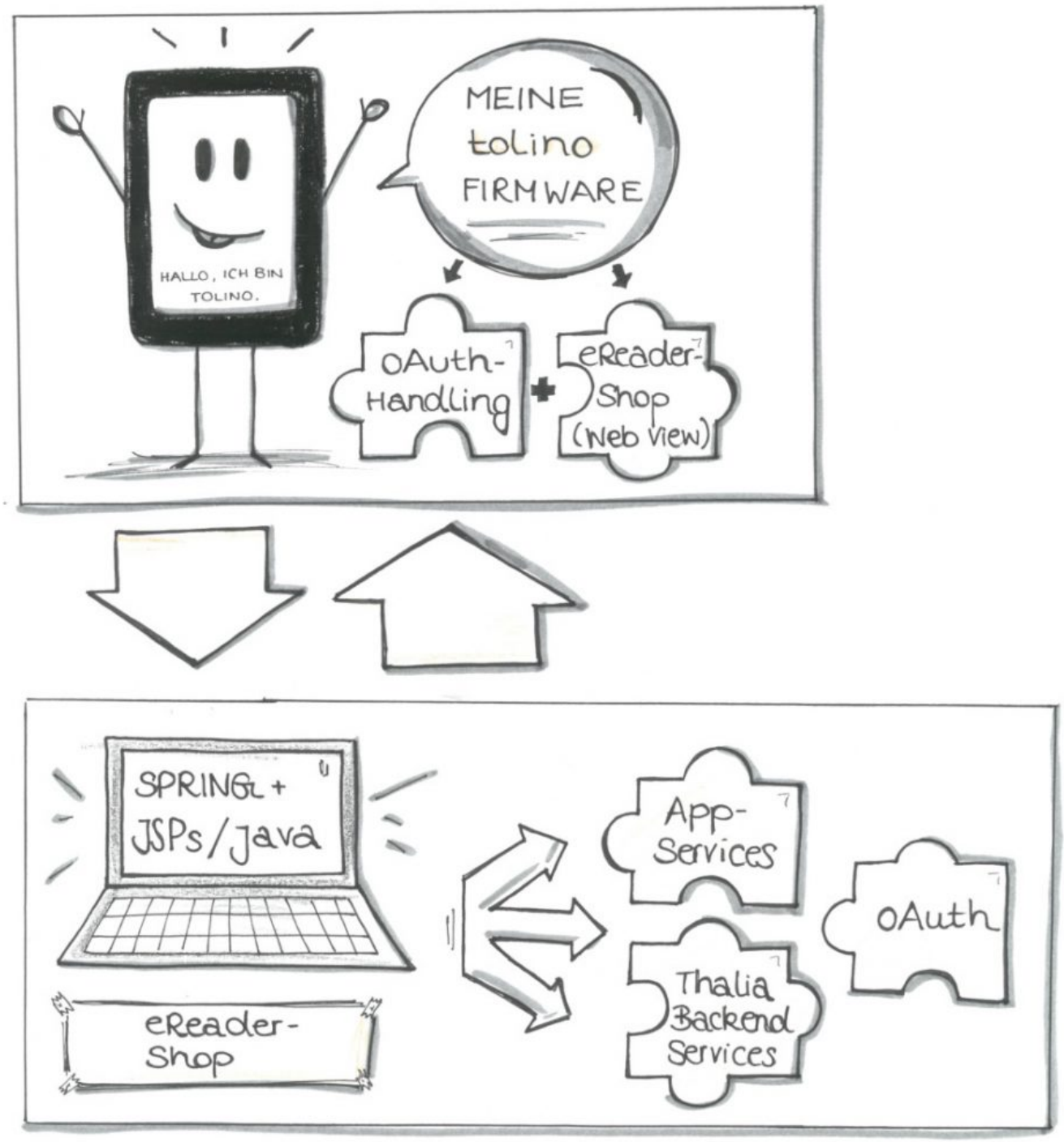

Abbildung 1

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).

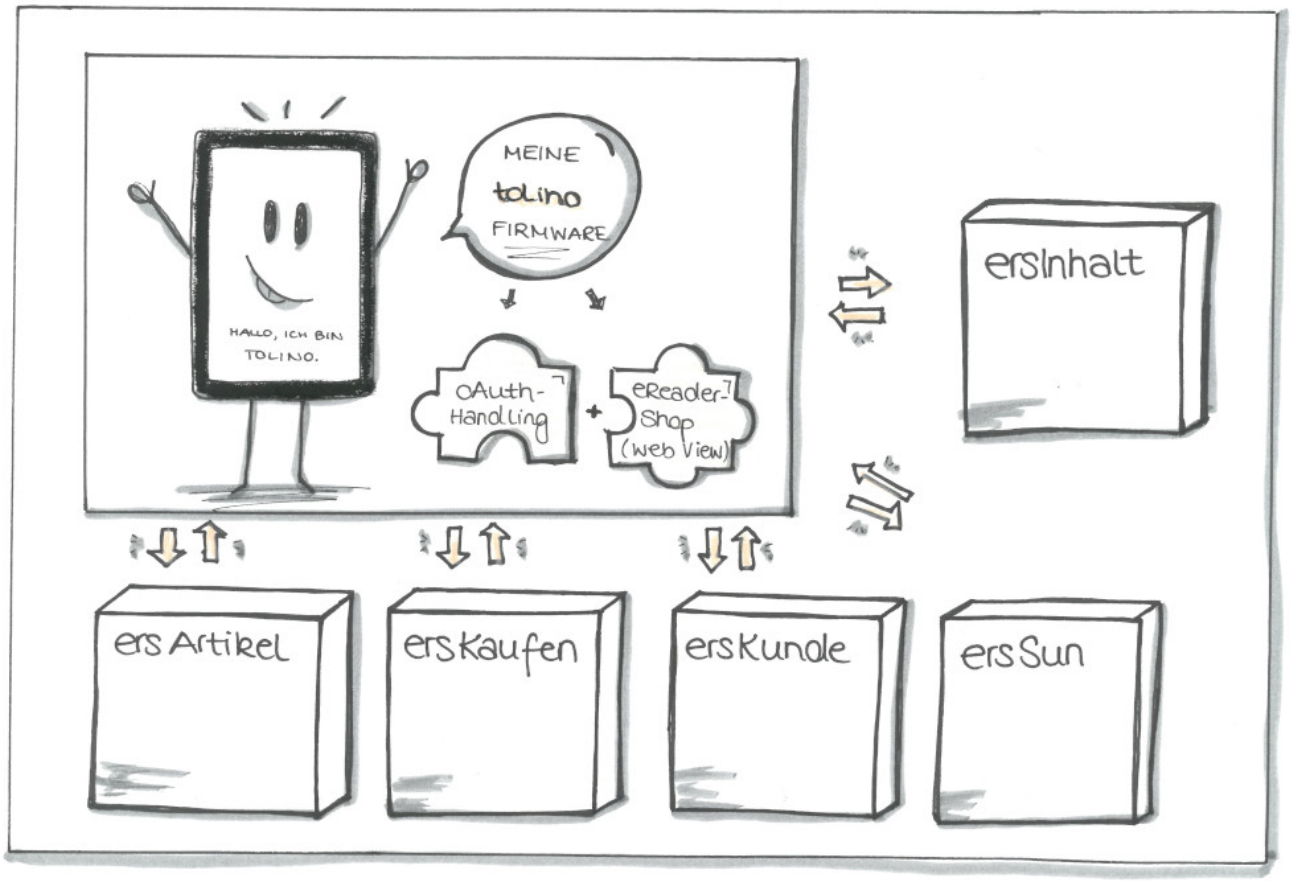

Abbildung 2

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).

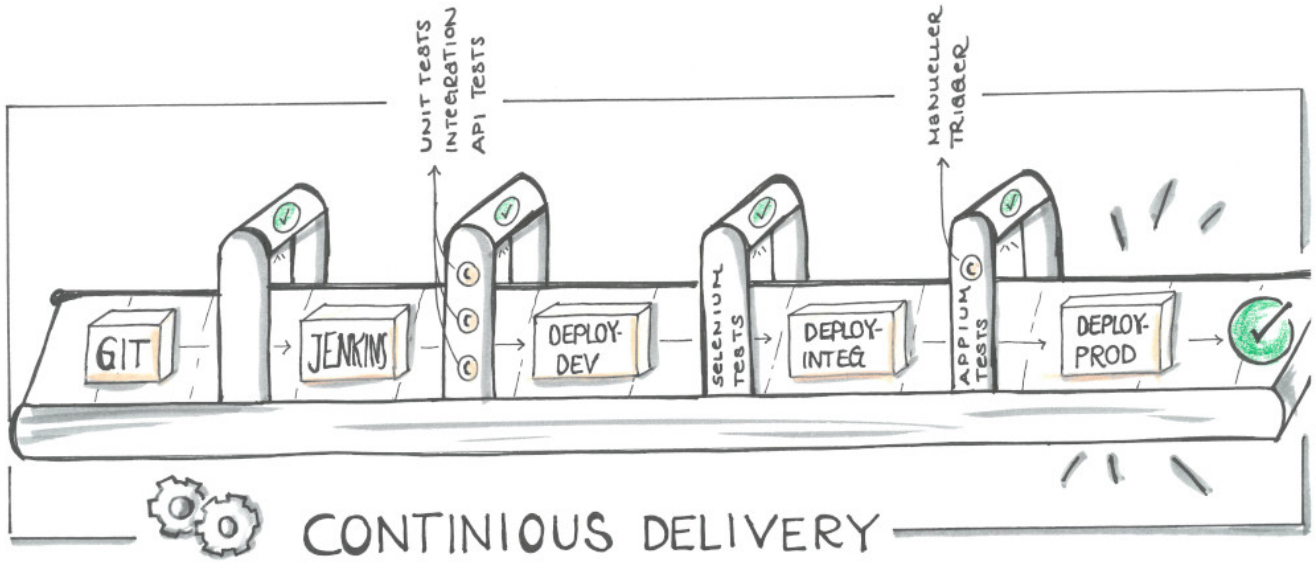

Abbildung 3

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!