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).

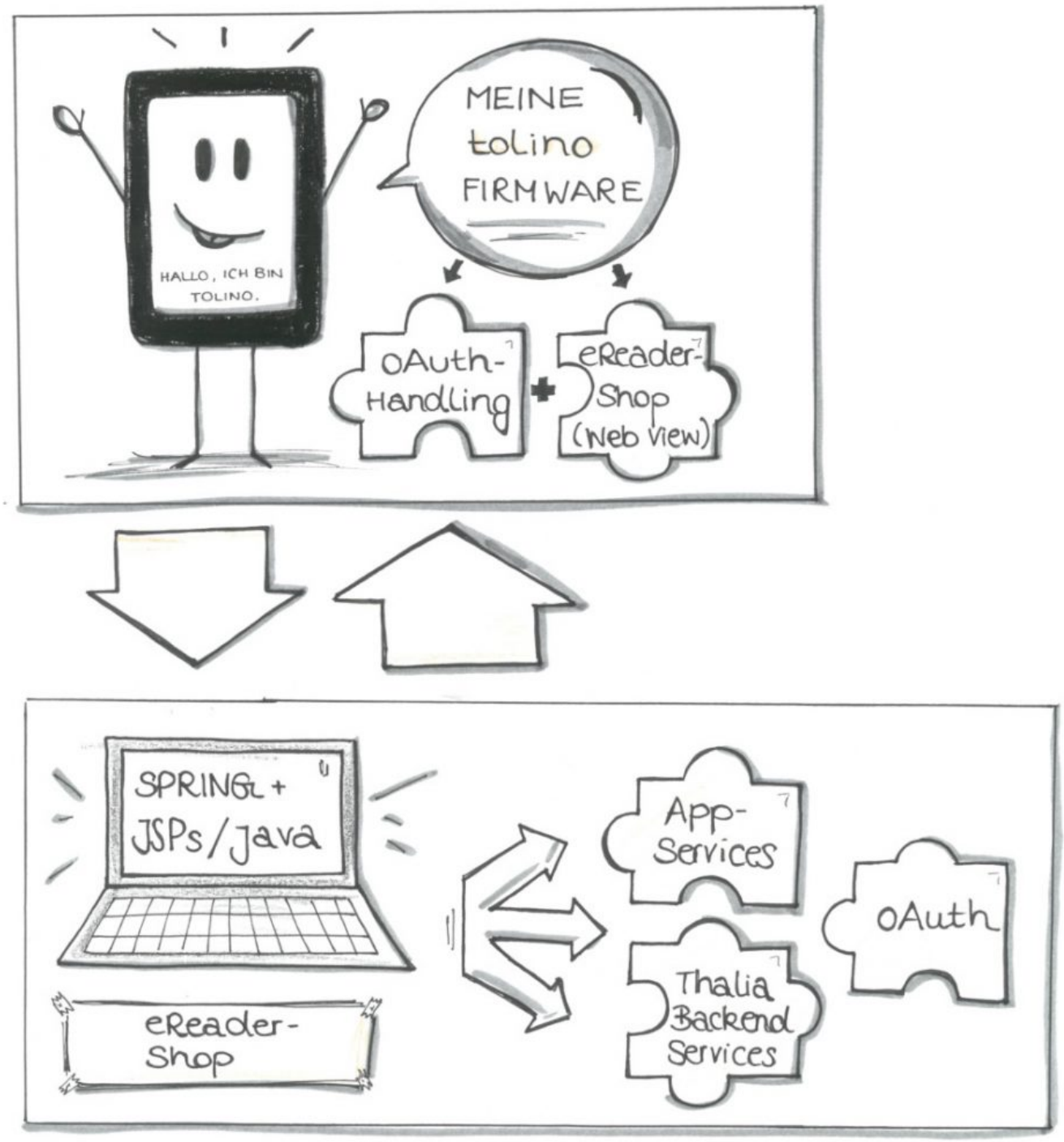


Abbildung 1

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).

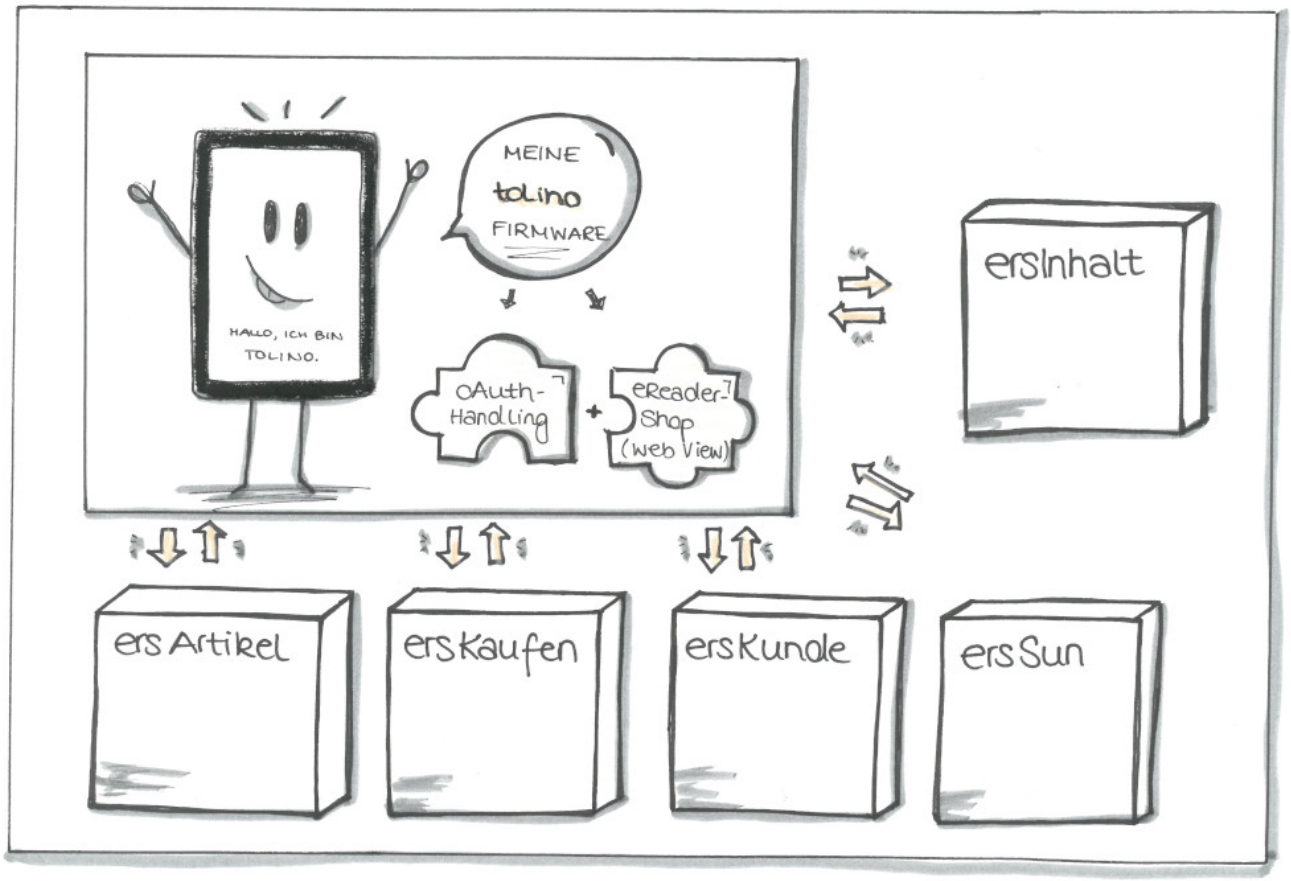


Abbildung 2

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).

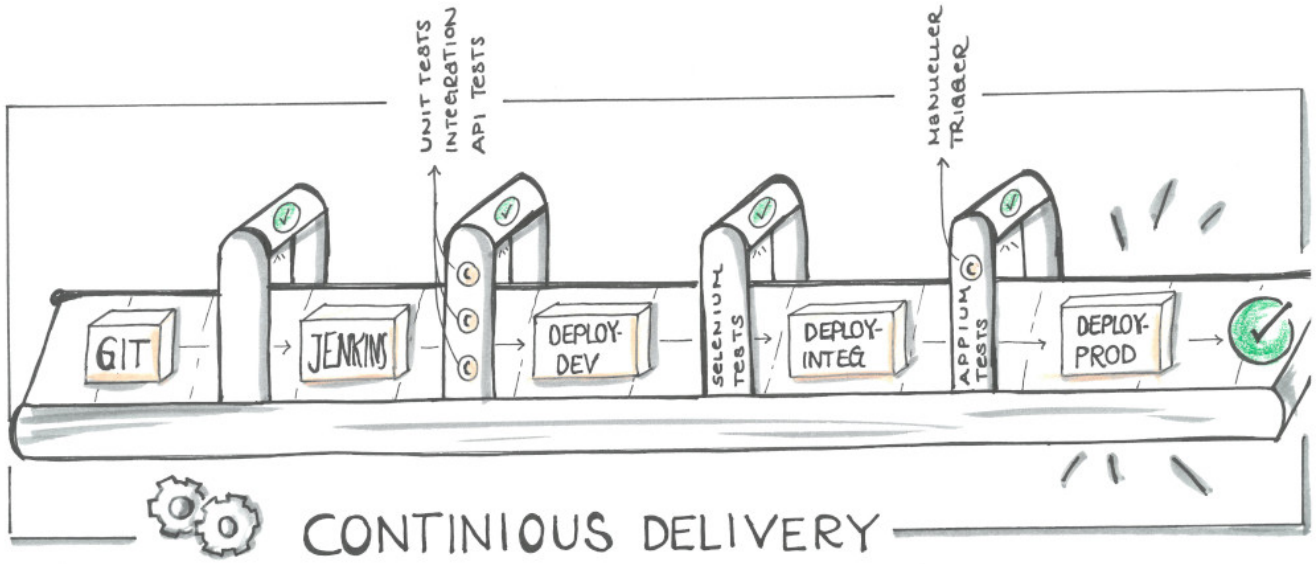


Abbildung 3

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.

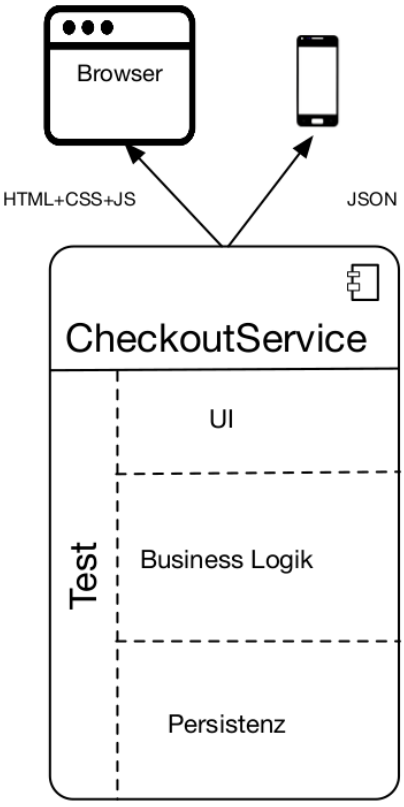
- Der Code unserer bereits livegestellten Services hat eine Testabdeckung von 78% und gibt uns bei der Weiterentwicklung, bei Updates oder bei Livegängen ein sehr gutes Qualitätsgefühl. Zusätzlich decken unsere Selenium- und Appium-Tests für alle neuen Services diverse Sicht- und bereits automatisierte 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!

Self-Contained Systems mit Spring Boot



Exemplarische Struktur eines SCS

„Self-Contained …“ was?

Als Self-Contained System (SCS) wird weithin ein Architekturmuster bezeichnet, welches sich durch folgende charakteristischen Eigenschaften auszeichnet: Ein SCS …

- … hat die Hoheit über alle Daten und Prozesse, fachliche wie technische, die einer klar abgrenzbaren Domäne zuzuordnen sind.
- … wird weiterentwickelt, getestet und betrieben von genau einem Team.
- … kommuniziert mit anderen SCS vorzugsweise asynchron.
- … kann Service-Schnittstellen für z. B. mobile Endgeräte anbieten.
- … bietet eine Benutzeroberfläche (UI), mit darunterliegender Geschäftslogik und einer Persistenzschicht für die Daten seiner Domäne an.
- … basiert so wenig wie möglich auf geteilter Infrastruktur.

Und wozu das Ganze?

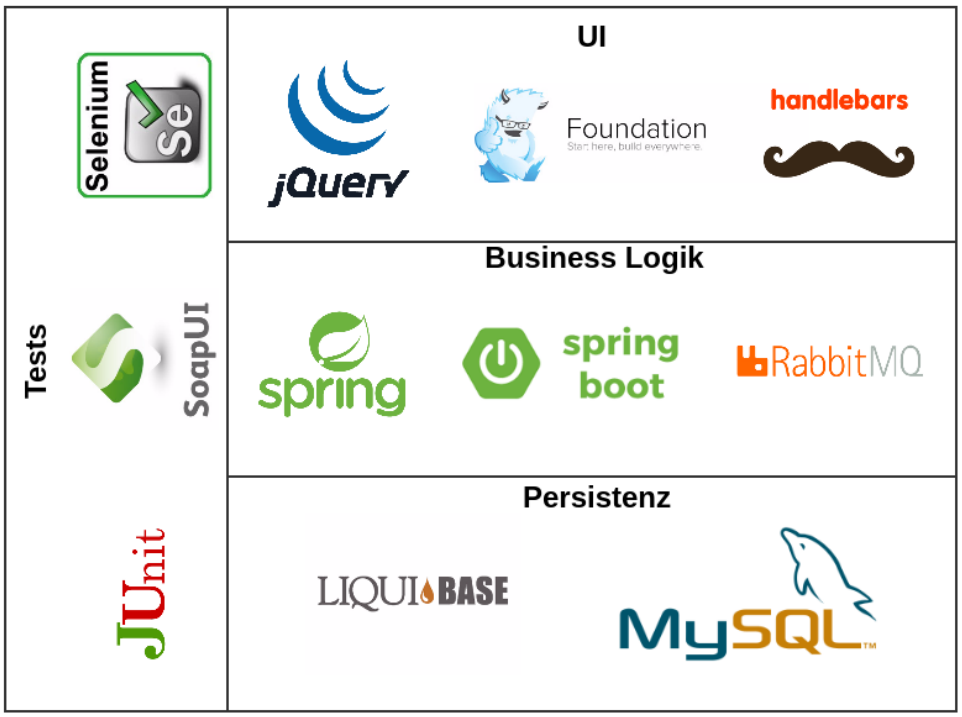
Da der weitere Fokus des Artikels auf der beispielhaften Implementierung des beschrieben Musters liegen soll, möchte ich an dieser Stelle gerne auf eine differenzierte Auseinandersetzung mit den Hintergründen verzichten. Wer sich vorab einen kurzen Überblick über das „Warum?“ verschaffen möchte, dem empfehle ich Why SCS? zu überfliegen. Für ausführlichere Informationen zu den Hintergründen und der Abgrenzung zu monolithischen oder Microservice-Ansätzen lohnt sich ein Blick in die Quellen.

Alternativ ist auch folgender Podcast von InnoQ zum Thema *„Eigenschaften und Umsetzung von SCS“* sehr aufschlussreich (Quelle: https://www.innoq.com/de/podcast/030-scs-teil-1/):

https://www.innoq.com/de/podcast/archive/030-scs-teil-1.mp3

Wie soll das mit Spring Boot funktionieren?

Entsprechend der oben skizzierten Struktur benötigen wir technische Komponenten für die klassischen Schichten UI, Business Logik & Persistenz. Des Weiteren soll der Technologie-Stack Möglichkeiten bereitstellen, jede dieser Komponenten automatisiert testen zu können. Der im Produktteam *„Kunde im Mittelpunkt“* (KIM) eingesetzte Technologie-Stack zur Abbildung dieser Anforderungen setzt sich aus den folgenden Komponenten zusammen, mit Spring Boot als Fundament:



Technologie-Stack im Team KIM

Für das initiale Setup des Spring Boot-Projektes werden im Team KIM beinahe dieselben Dependencies verwendet, wie für eine gewöhnliche Spring MVC-Applikation. Ein gutes Basisprojekt mit den wichtigsten Dependencies hierfür kann in kürzester Zeit über https://start.spring.io/ generiert werden. Alle nötigen Dependencies für ein SCS können der nebenstehenden Abbildung entnommen werden.

Minimal-Set an Dependencies für ein SCS

An dieser Stelle gibt es leider keine Möglichkeit für die Templating-Engine Handlebars alle nötigen Dependencies automatisch mit zu erzeugen, sodass diese manuell ergänzt werden müssen. Des Weiteren verzichte ich hier darauf, das Thema Testing näher zu beleuchten, da hiermit mehrere dedizierte Blog-Artikel gefüllt werden könnten. Schauen wir uns stattdessen einmal genauer an, was es mit den einzelnen Komponenten auf sich hat.

```
{{#if model.generalMessage.message}}
    <div class="{{model.generalMessage.type.value}} callout">
        <p>{{{model.generalMessage.message}}}</p>
        {{#if closable}}
            <button class="close-button" type="button">
                <span>×</span>
            </button>
        {{/if}}
    </div>
{{/if}}
```

Fehler- bzw. Erfolgsmeldung mit Handlebars

Auf oberster Ebene versetzt uns Handlebars dazu in die Lage, dem Benutzer angezeigte Inhalte dynamisch generieren zu lassen. Die Java-Adaption von Handlebars bietet eine einfach Möglichkeit, binnen kürzester Zeit in einer Spring Boot-Applikation derartige Inhalte auszuspielen.

Sind die dynamischen Inhalte, wie z. B. Auftragsdaten, Rechnungen etc. generiert, müssen diese noch in responsiver, benutzerfreundlicher Weise dargestellt werden. Da die Anforderungen an eine endgerätefreundliche Website immer weiter steigen, setzen wir hier auf die Expertise des Foundation-Frameworks, welches bereits umfangreiche Features im Standard enthält, die eine zeitgemäße Visualisierung von Content erlauben. Auf Grund des hohen Bekanntheitsgrades von WebUI-Frameworks a la Twitter-Bootstraps etc. möchte ich hier nicht näher auf Foundation eingehen. Welche andere wichtige Rolle das Frontend spielt, sehen wir noch im Abschnitt zur Integration von SCS in

bestehende Systeme.

```
@Controller
@RequiredArgsConstructor(onConstructor = @__(@Autowired))
@RequestMapping("/site/{mandantId}/login/login.html")
public class LoginController {

    @RequestMapping(
            method = GET,
            produces = TEXT_HTML_VALUE)
    public ModelAndView get(@PathVariable final int mandantId) {
        //@formatter:on
        return new ModelAndView( viewName: "login",
                ImmutableMap.of( k1: "model", createModel()));
    }
```

Controller-Definition mit Spring & Lombok

Eine Schicht unterhalb der UI erreichen wir das Backend über Spring-Controller, die HTML bereitstellen, bzw. -RestController für JSON-Inhalte, wenn es sich bei dem aufrufenden Endgerät um einen mobilen Client handelt, der eine native UI anbietet. Die Controller werden mit Spring-Bordmitteln, wie z. B. Spring Web und Spring Security, abgebildet und stellen die Brücke zur eigentlichen Geschäftslogik dar. Gerät die Geschäftslogik jedoch an ihre Domänengrenzen, wird die Kommunikation mit einem anderen SCS erforderlich.

```
@RabbitListener(bindings = @QueueBinding(
    value = @Queue(
            value = QUEUE,
            durable = "true",
            arguments = @Argument(
                name = "x-dead-letter-exchange",
                value = EXCHANGE_DEADLETTER
            )
    ),
    exchange = @Exchange(
            value = EXCHANGE,
            type = ExchangeTypes.FANOUT,
            ignoreDeclarationExceptions = "true",
            durable = "true"
    )
))
void onKundeUpdate(KundeDto kundeDto) {
```

RabbitListener-Definition mit Deadletter-Exchange

Eine wesentliche Maßgabe hierzu, die sich aus dem SCS-Ansatz ergibt, ist die asynchrone Kommunikation mit anderen SCS, welche wir im Team KIM beispielsweise mit Spring AMQP realisieren. Da Spring Boot das Dogma

„*convention over configuration*“ verfolgt, können wir hier von der RabbitAutoConfiguration des Frameworks profitieren, welche bereits eine Menge häufig benötigter Beans mitbringt, wie ein RabbitTemplate, eine ConnectionFactory, einen AmqpAdmin usw. Benutzername, Passwort und Host für die RabbitMQ-Verbindung müssen so nur noch konfigurativ über die application.yaml gepflegt werden. Auch das Lauschen auf Nachrichten, die von RabbitMQ eingehen, kann komplett deklarativ realisiert werden, sodass sich der Umfang an zu schreibenden Codezeilen auf ein Minimum reduzieren lässt.

Alternativ zum ereignisgetriebenen Datenaustausch können Daten auch im Hintergrund synchronisiert und redundant persistiert werden. Hierfür nutzen wir den Scheduling-Mechanismus von Spring innerhalb derselben Applikation, der in einem eigenen Thread per REST-Call an eine andere Applikation Daten repliziert, die das SCS zwar für Aufgaben innerhalb seiner Domäne braucht, worüber es aber nicht die Hoheit (!) hat. Die Applikation ist so auch dann verfügbar (auf einem potenziell veralteten Datenbestand), wenn das SCS, aus dem es seine Daten bezieht, nicht erreichbar ist.

```
<changeSet id="1" author="dennis.schneider@thalia.de">
    <createTable schemaName="thalia_ecp_login" tableName="benutzer">
        <column autoIncrement="true" name="id" type="BIGINT">
            <constraints nullable="false" primaryKey="true"/>
        </column>
        <column name="kundennummer" type="BIGINT">
            <constraints nullable="false" unique="true"/>
        </column>
        <column name="angelegt_am" type="TIMESTAMP">
            <constraints nullable="false"/>
        </column>
    </createTable>
```

Tabellendefinition mit Liquibase

Um die Daten nun auch noch persistieren zu können, erzeugen und erweitern wir unser Datenbankschema mit Liquibase. Ein entscheidender Vorteil von Liquibase ist, dass es die Anpassungen am Datenbankschema direkt beim Hochfahren der Applikation auf die Datenbank anwendet. Änderungen am Modell können so direkt mit dem Deployment der Applikation automatisch ausgeführt werden, ohne dass ein manueller Eingriff erforderlich ist.

Sind Tabellen und Spalten angelegt, können wir mittels Spring Data JPA auf diese zugreifen. Spring Data übernimmt für uns das gesamte OR-Mapping, nachdem wir die entsprechenden @Entity-, @Table-Annotationen und Co. auf unseren

Objektrepräsentationen des Datenbankschemas ergänzt haben. Zuletzt ist für jede Entity noch ein Repository-Interface zu definieren, welches ein Interface vom Typ Spring Data CrudRepository erweitert. Das Framework sorgt anschließend zur Laufzeit dafür, dass ein SimpleJpaRepository erzeugt wird, mit dem auf die entsprechenden Daten zugegriffen werden kann.

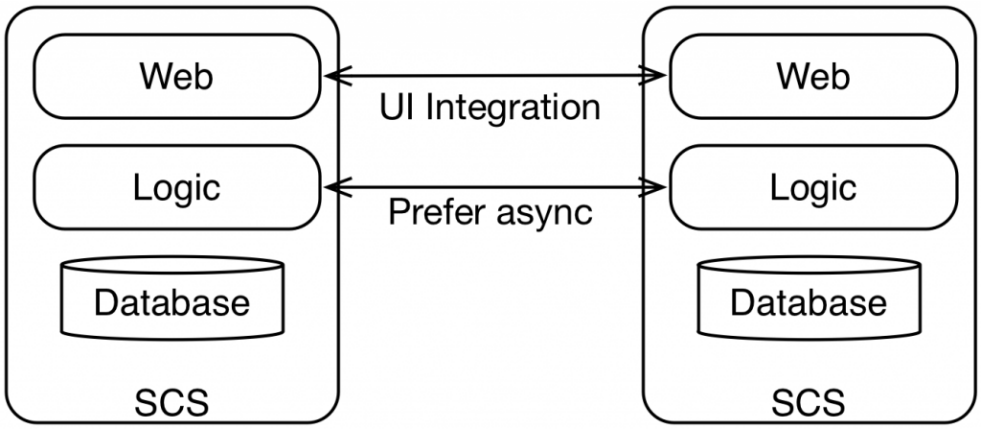
Wie integriert sich ein SCS in bestehende Systeme?

Aus dem vorherigen Abschnitt deutet sich an, dass wir hier über zwei verschiedene Arten von Integration nachdenken müssen:

- Datenintegration
- Prozessintegration

Wie schon beschrieben, erfolgt die Datenintegration vor allem über asynchrone Kommunikation, wie z. B.

- ereignisgesteuerte Mechanismen (RabbitMQ, ActiveMQ, Kafka, …), oder
- regelmäßige bzw. bedarfsgetriebene Abfrage und Persistierung von Daten im Hintergrund (Spring-Scheduler, Quartz, …)



Prozessintegration per UI (Quelle: InnoQ)

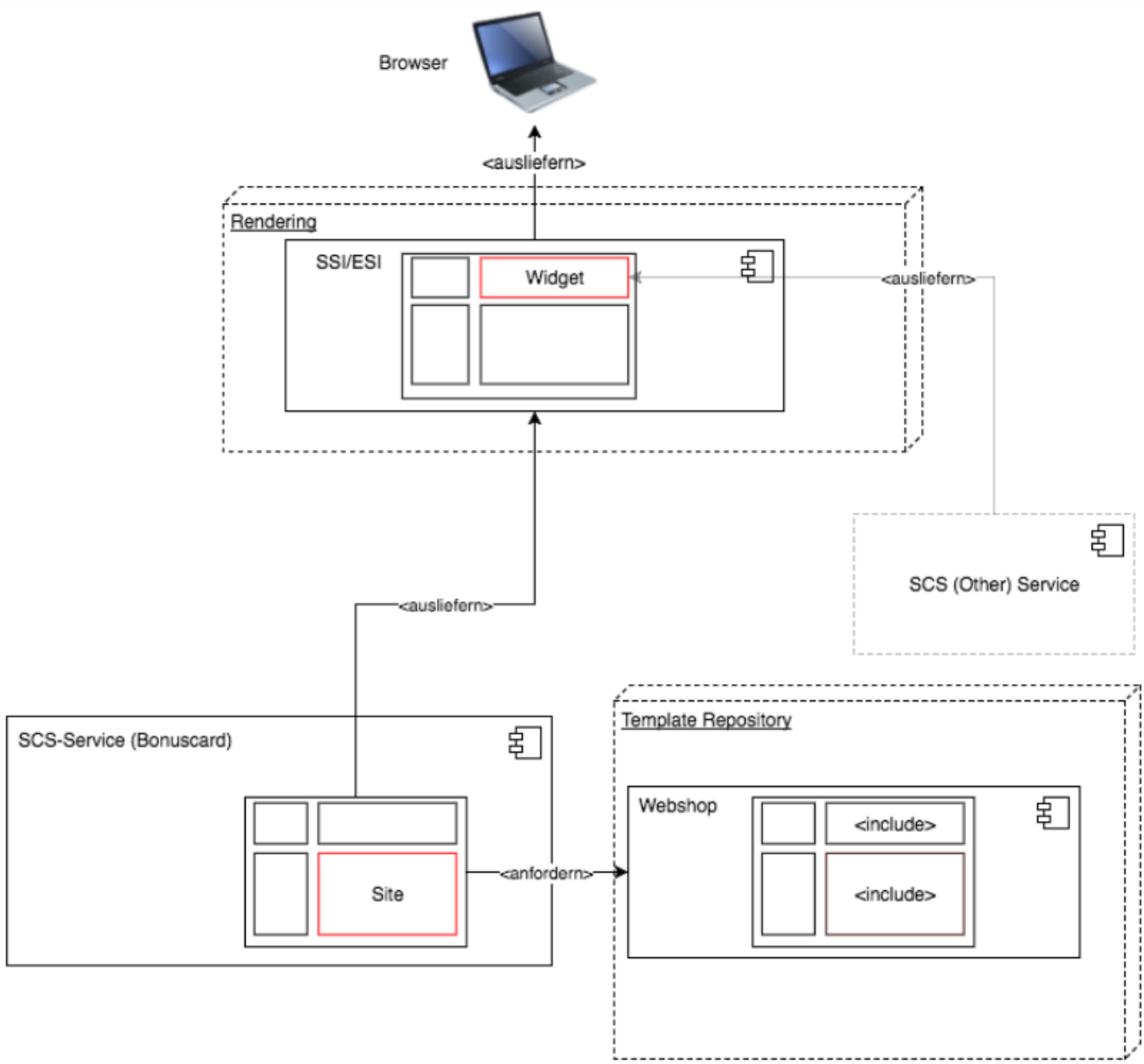
Bezogen auf Prozessintegration rückt die UI wieder in den Mittelpunkt. Da backendseitig vor allem asynchron und vorzugsweise innerhalb des eigenen SCS kommuniziert werden soll, findet die domänenübergreifende Prozessintegration

über das Frontend statt. Aus Implementierungssicht verwenden wir im Team KIM hierfür die beiden folgenden Ansätze:

1. Hyperlinks, als Einstiegspunkt in eine (andere) Domäne
2. Server-side-includes (SSI) & Ajax, zur Orchestrierung von SCS

Unter der Integration über Hyperlinks kann man sich URLs vorstellen, die auf dem SCS der Zieldomäne terminieren. So landet man bei einem Klick auf „Passwort vergessen?“ auf der Anmeldeseite von Thalia.de beispielsweise auf einem unserer SCS, dessen Domäne der Login bzw. die Registrierung ist.

Sollen hingegen auf einer Website die UIs mehrerer SCS angezeigt werden, orchestrieren wir diese mittels SSI oder Ajax. Über beide Verfahren lassen sich Teilbereiche einer Website mit den Inhalten unterschiedlicher SCS befüllen. Die folgende Abbildung veranschaulicht die Integration mehrerer SCS mittels SSI:



Beispielhafte SCS-Integration mit zentralisiertem Template Repository

Was ist sonst noch zu beachten?

Neben der Implementierung des SCS haben sich in diesem Artikel einige Aspekte am Rande angedeutet, die bei der Realisierung derartiger System nicht zu vernachlässigen sind. Fragestellungen, die hierbei ebenfalls betrachtet werden sollten, könnten sein:

- Wie werden SCS gebaut und deployed?
- Wie lassen sich hierbei Unittests / Schnittstellentests / Integrationstests / End-to-end-Tests einbinden?

- Wie wird eine einheitliche User Experience in SCS sichergestellt?
- Welche Bedeutung hat eine SCS-Architektur für das Entwicklungsteam / die Organisation?

Sicherlich hat diese kurze Liste keinen Anspruch auf Vollständigkeit, deutet aber doch an, auf welche Herausforderungen sich eine Organisationseinheit einstellen muss. Vielleicht widme ich mich ja in Folgebeiträgen einer dieser Fragen bzw., wie wir dieser begegnet sind :-).

Quellen

- http://scs-architecture.org/index.html
- https://www.innoq.com/en/articles/2016/11/self-contained-systems-different-microservices/
- https://jaxenter.de/self-contained-systems-45606
- https://www.heise.de/developer/artikel/Self-contained-Systems-ein-Architekturstil-stellt-sich-vor-3038718.html
- https://spring.io/guides

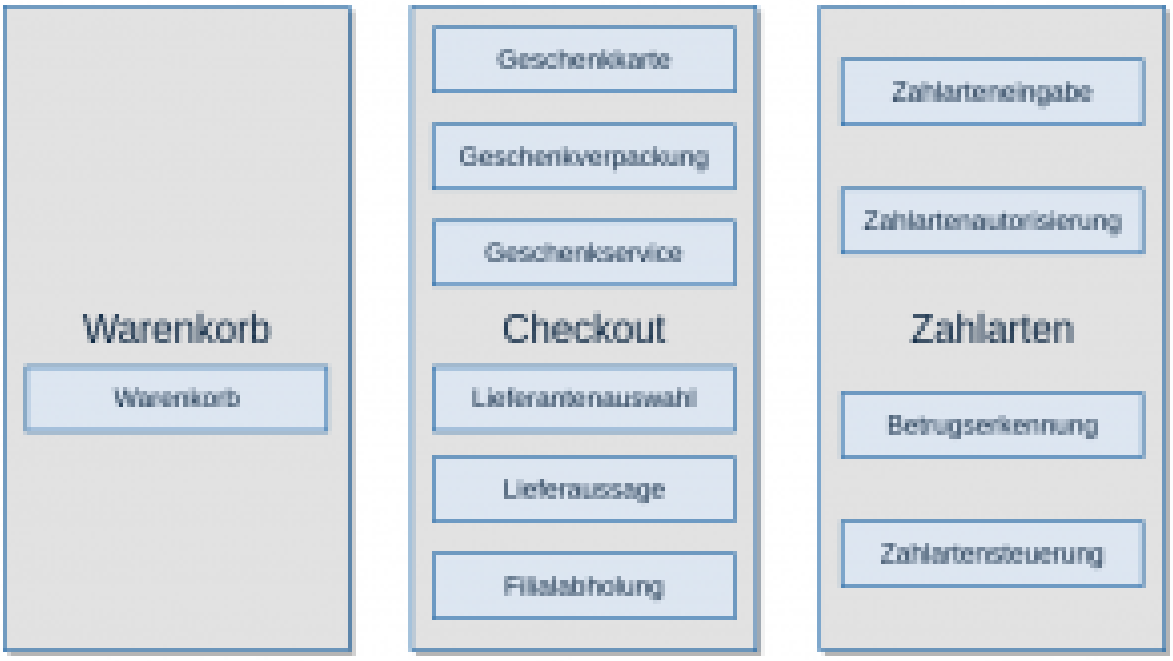
Team Kaufen auf dem Weg zur Omnichannel Plattform – Let's get it started

Als Thalia sind wir dabei, unsere bisherige Systemlandschaft im eCommerce, mit einem Webshop-Monolithen als Kern, auf neue Füße zu stellen. Ziel ist dabei eine zukunftsfähige **Omnichannel Plattform**. Dieser Beitrag beschreibt die Startphase des **Produktteams Kaufen**. Er erläutert, welche Themen zu unserem Produkt gehören und wie wir uns die zugehörige Systemlandschaft vorstellen.

Die Reise beginnt

Die Reise vom Team Kaufen beginnt mit einer Sammlung von Themen, die zu unserem Produkt gehören. Diese drehen sich im Wesentlichen um den Prozess des Kaufens, den Checkout.

Ausgehend von dieser Idee und den verschiedenen Themen rund um den Checkout beginnt der Prozess, die notwendigen Systeme und Services zu definieren. Nach langen Diskussionen entsteht der Plan, unsere Zuständigkeiten auf drei Systeme aufzuteilen. Die folgende Darstellung stellt dabei das Zielbild dar.



Der **Warenkorb** soll im Kern eine Artikelliste abbilden, der nur artikelrelevante Informationen verarbeitet und nach Möglichkeit keine Kundeninformation. Dadurch erhoffen wir uns, die zugehörige Logik einfacher abbilden zu können.

Zudem ergibt sich eine klare Trennung vom **Checkout**, bei dem Kundeninformationen zum Tragen kommen. Der Kunde muss sich somit beim Übergang vom Warenkorb zum Checkout anmelden. Zum System Checkout gehören dann auch Themen wie etwa Thalia Geschenkkarten, Versandoptionen und Geschenkoptionen.

Das Thema **Zahlarten** hat zwar auch Bezug zum Checkout, ist aber so umfangreich, dass wir es als eigenes System abbilden. Dazu gehören das Speichern von Zahlarten zum Kunden, die Steuerung der verfügbaren Zahlarten im Checkout und die Autorisierung von Zahlungen.

Welche Grundsätze liegen den Systemen zugrunde?

Alle drei Säulen sollen eigenständig funktionieren und als Self-Contained System umgesetzt werden. Eine Integration von zwei Säulen, also zwei Systemen, soll somit nicht über synchrone Aufrufe im Backend erfolgen, sondern nach Möglichkeit über das Frontend bzw. durch Datenreplikation.

Es soll **in jedem System einen führenden Service** geben, der die Darstellung im Frontend steuert und auch die Schnittstellen nach außen anbietet. Sollte ein synchroner Aufruf zwischen zwei Systemen im Backend nicht vermeidbar sein, soll die Kommunikation dann über diese Schnittstellen erfolgen. Der führende Service fungiert somit als eine Art Gateway für die eingehende Kommunikation in ein System.

In unserem Zielbild kann **ein System aus 1-n Services** bestehen. Die Integration verschiedener Services innerhalb eines Systems kann dabei sowohl über das Frontend als auch das Backend erfolgen. Schon jetzt haben wir bestimmte Services, die perspektivisch kein Frontend ausliefern werden, und somit reine Backend Services bleiben sollen.

Welche technischen Herausforderungen erwarten wir unterwegs?

Zusätzlich zu den fachlichen Themen erwarten wir zahlreiche technische Herausforderungen während der Neugestaltung der Systeme. Dazu zählen unter anderem:

- Technik Stack
- Konfigurationsmanagement
- Datenreplikation
- Session Management
- Monitoring

Unser Technik Stack setzt sich seit langem und auch weiterhin aus bewährten Frameworks und Tools zusammen, wie etwa Spring Boot, Maven, Handlebars, Tomcat, Jenkins, Git, Gerrit, Puppet, Ansible, Graylog, Grafana, InfluxDB, RabbitMQ etc.

Der Punkt **Konfigurationsmanagement** hat uns dabei sicherlich am meisten beschäftigt. Bisher haben wir für neue Services auf Konfiguration über YAML-Dateien gesetzt. Diese sind gut strukturierbar und mit Spring automatisch in eine

gut nutzbare Objektstruktur zu überführen. Je nach Umfang eines Services stoßen wir hier jedoch an Grenzen, da eine hohe Anzahl an Konfigurationen die Übersichtlichkeit erschwert. Insbesondere in einem mandantenfähigen System, in dem es viele mandantenspezifische Konfigurationen zu verwalten gibt. Eine Lösung hierfür, welche die aktuellen und in Zukunft absehbaren Anforderungen erfüllen kann, planen wir in Zukunft nach und nach umzusetzen.

Datenreplikation ist bei der Verwendung von verteilten Systemen, die wir bei Thalia für die Omnichannel Plattform umsetzen, notwendig, um die ständige synchrone Kommunikation zwischen den Systemen vermeiden zu können. Dabei sind für uns zwei Aspekte die zentralen Diskussionspunkte.

> Zum einen: **Soll jeder einzelne Service seine Replikation haben oder kann ein System für alle dazugehörigen Services die Replikation zentral halten?** Die erste Option unterstützt zwar die lose Kopplung der einzelnen Services, führt unter Umständen aber auch zu einer großen replizierten Datenmenge und zur Implementierung der Replikationslogik an vielen Stellen. Bei der zweiten Option verhält es sich genau anders herum. Zudem bleibt dabei zu klären, wie die einzelnen Services eines Systems sich die Daten teilen.
>
> Zum anderen: **Können wir überall mit Eventual Consistency leben oder brauchen wir irgendwo konsistente Daten über mehrere Service hinweg?** An vielen Stellen können wir in einem verteilten System wie unserem mit Eventual Consistency leben, die wir uns mit der Datenreplikation zwischen den Services einkaufen. Insbesondere beim Checkout kann es jedoch auch notwendig sein, auf konsistenten Daten zu arbeiten. Bspw. sollten während des Kaufprozesses die gleichen Artikelpreise zugrunde gelegt werden. Da wir **nicht mehrere Services mit Zugriff auf ein und dieselbe Datenbank** haben wollen und die **Datenkonsistenz** eine Rolle spielen kann, verfolgen wir aktuell die Idee, dass der führende Service eines Systems die Daten repliziert und den anderen Services in der API übergibt. Ob dieser favorisierte Weg tatsächlich so funktioniert wie gedacht wird der Prozess der Umsetzung der einzelnen Services zeigen müssen.

Session Management war bisher in unseren Legacyanwendungen ein wichtiges Thema. Unsere neuen Services sollen eigentlich **stateless** werden, also auch ohne Session arbeiten können. Aus heutiger Sicht scheint das je nach System

aber nicht immer ein sinnvoller Weg und die Verwendung von einer verteilten In-Memory-Lösung wie Hazelcast auch eine Option zu sein. Wie wir das umsetzen werden, werden wir sehen, sobald die Notwendigkeit dafür besteht.

Ein zentraler Aspekt bei verteilten Systemen ist das **Monitoring** der einzelnen Services. Hierfür nutzen wir sowohl proprietäre Lösungen als auch Tools wie Graylog und Grafana. Hier sind wir technisch ganz gut aufgestellt und dabei, unsere Nutzung der vorhandenen Tools kontinuierlich zu verbessern.

Fazit

Die Reise des Teams Kaufen zu einer zukunftsfähigen Omnichannel Plattform beginnt mit den beschriebenen Rahmenbedingungen und Konzepten. Unterwegs werden wir diese sicher kontinuierlich hinterfragen und bei Bedarf anpassen müssen. Jedoch fühlen wir uns durch die zahlreichen Workshops, in denen wir ausgehend von einer Sammlung der zu verantwortenden Themen und einer Idee der Zielarchitektur diese Grundlage schaffen konnten, sehr gut für den weiteren Weg vorbereitet.