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

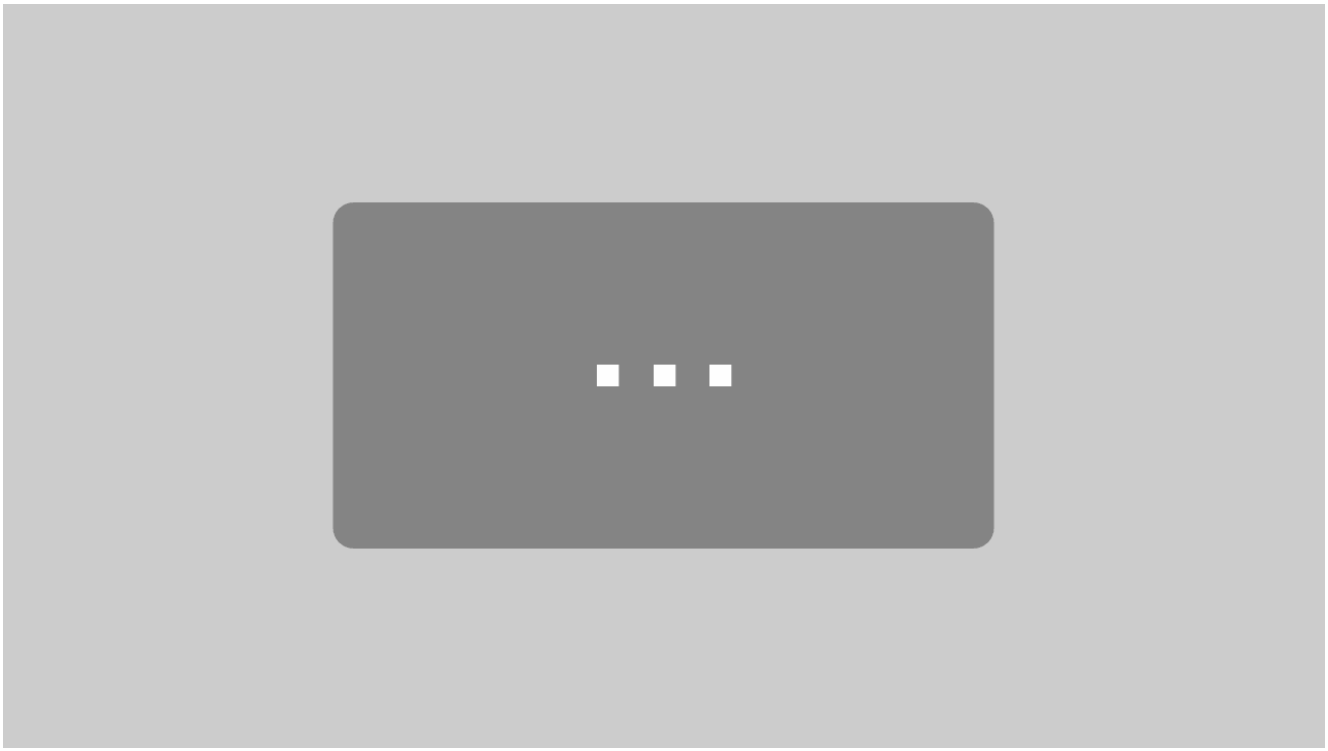
---

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

Thalia auf der microXchg 2018

Reconstructing thalia.de with self-contained systems

Zusätzlich zu diesem Blog wollen wir unsere Erfahrungen auch auf anderem Wege teilen. So werden Benjamin und Dennis auf der microXchg 2018 einen Vortrag halten. Die microXchg findet am 22./23.03. in Berlin statt.

Der Vortrag mit dem Titel *Reconstructing thalia.de with self-contained systems* soll sowohl technische als auch organisatorische Erfahrungen thematisieren, die wir beim Aufbau der Omnichannel Plattform bei Thalia gemacht haben.

Benjamin Leenen

Dennis Schneider

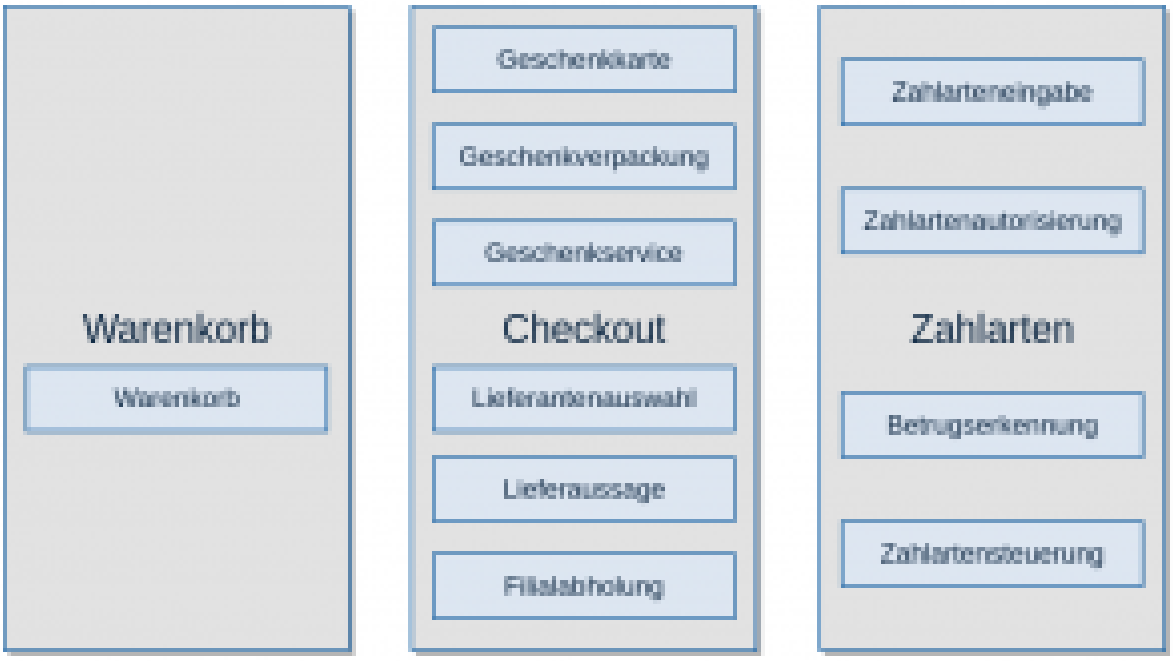
Team Kaufen auf dem Weg zur Omnichannel Plattform – Let’s get it started

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.