Frontend-Spielereien mit WebAssembly

Was hat ein Omni-Channel Buchhändler mit WebAssembly zu tun? Erstmal nicht viel. Trotzdem beschäftigen wir uns auch mit neuen, interessanten Technologien, welche nicht direkt in unserem täglichen (Entwickler) Leben anwendbar sind. Dieser Blog-Eintrag beschäftigt sich mit der Entstehung und dem aktuellen Stand der WebAssembly Spezifikation.

Die Gegenwart

Wir schreiben das Jahr 2018. Das Internet ist allgegenwärtig. Als Anwender und Entwickler haben wir eine Vielzahl an unterschiedlichsten Technologien für die Entwicklung moderner Web-Anwendungen zur Verfügung. Wir haben den HTML5 Standard. Wir haben CSS. Wir haben IndexedDB oder auch den LocalStore im Browser. Mit diesen Technologien lassen sich schnell und effizient Online- oder auch Offlineanwendungen bauen und verteilen. Seit Anfang 2017 gibt es neu WebAssembly in diesem Universum. Um WebAssembly und dessen Bedeutung genauer zu verstehen, müssen wir jedoch eine kleine Zeitreise in die Vergangenheit unternehmen.

Wie alles begann

Wir schreiben das Jahr 1995. In diesem Jahr wurde ein wichtiger Meilenstein für das gesamte Internet gelegt. Als Ergebnis von Projekt „Mocha“ wurde zusammen mit dem Netscape Navigator Version 2.02 eine Skript-Sprache für dynamische HTML Seiten der breiten Öffentlichkeit zur Verfügung gestellt. Diese Skript-Sprache hatte den Namen Netscape LiveWire. LiveWire wurde später umbenannt in JavaScript. Damit war ein Fundament geschaffen, auf welchem wir noch heute unsere Anwendungen aufbauen. Die Adaption von JavaScript verlief anfangs eher zögerlich. Hintergrund war der laufende Prozess der Sprach-Normierung und Weiterentwicklung gepaart mit den „Browser-Kriegen“ der damaligen Zeit. Es gab noch nicht viele Standards, bzw. wurden die Standards sehr hart erkämpft. Im Jahr 2005 erwähnte James Garret in seinem Aufsatz das erste Mal das Wort „AJAX“. Nun war es möglich, die dynamischen Inhalt im Browser auch mit Backend-Logik zu

koppeln, um noch interaktivere Anwendungen zu bauen. Die Beliebtheit von JavaScript stieg stetig an.

Neue Horizonte und Möglichkeiten

Wir spulen die Zeit nun etwas vor bis in das Jahr 2010. Das Internet ist überall. Auf dem Desktop. Auf unseren Smartphones. Auf embedded Devices. In der Gebäudeautomatisierung. JavaScript ist überall. Diese standardisierte Plattform ist eine ideale Grundlage für weitere Anwendungsfälle. In diesem Jahr betritt ein sehr interessantes Projekt die Bühne: Emscripten. Emscripten ist im Wesentlichen ein C bzw. C++ Compiler. Dieser Compiler erzeugt jedoch keinen Binärcode in Form von x86 Instruktionen, sondern er übersetzt das Programm nach JavaScript. Somit werden völlig neue Deployment-Szenarien für bestehende C++ Anwendungen möglich. Der Early-Adopter von Emscripten war die Entertainment Branche, voran das große Spiele-Studio Epic. Durch den Einsatz von Emscripten war es möglich, die Unreal Game Engine und bestehende Desktop Spiele mit minimalen Aufwand auch im Browser laufen zu lassen. Aus wirtschaftlicher Sicht war das ein gigantischer Sprung. Die Unreal Engine komplett in JavaScript neu zu schreiben hätte sicherlich ein großes Team über mehrere Monate bis Jahre ausgelastet. Durch Emscripten als Compiler wurde dies deutlich optimiert. Das Ergebnis war die Unreal Citadel Demo, welche im Browser lief. Als Laufzeitgeschwindigkeit wurde die halbe Framerate im Vergleich zu der Desktop Variante erreicht. Das Spiel war spielbar, jedoch gab es noch deutlich Luft nach oben.

Es geht noch schneller

Nach intensiver Zusammenarbeit zwischen Mozilla, Epic und dem Emscripten Team wurde im Jahr 2013 asm.js präsentiert. asm.js ist eine Untermenge von JavaScript, welche auf Laufzeiteffizienz ausgelegt ist. Durch die Annotation von Ausdrücken ist es möglich, der JavaScript Runtime Hinweise zu geben, um den Code effizienter auszuführen. Diese Typisierung von JavaScript macht es möglich, Integer und Fließkomma Datentypen genauer zu unterscheiden, und somit gerade die Integer Berechnung deutlich zu verbessern. Diese Unterscheidung ist in JavaScript nicht möglich, da es hier nur Fließkomma Zahlen gibt. Emscripten wurde mit asm.js als Compile-Target erweitert. Das Ergebnis war die Unreal-Engine mit der Cathedral Demo mit annähernd nativer Geschwindigkeit im Browser. Zu diesem Zeitpunkt gab es Browser mit sehr guten JavaScript Just-in-Time Compilern. Durch asm.js wurden noch zusätzliche Hinweise für die Optimierung des Laufzeitverhaltens

gegeben. Insgesamt war das Ergebnis im Sinne der Laufzeit sehr ansehnlich. Jedoch hatte das Ganze auch seine Schattenseite. Die Code-Basen wurden immer größer, der Quellcode immer komplexer. Zusammen mit der Größe und der Komplexität stieg natürlich auch die Download-Zeit für Anwendungen deutlich an. Und nicht nur die Downloadzeit stieg, sondern auch die Parsing-Zeit für JavaScript im Browser. Als Ergebnis ist die Time-To-Interactive für sehr große Anwendungen für spontane Internet-User nur noch bedingt akzeptabel. Die Suche nach einem Werkzeug, welches sowohl das Laufzeitverhalten als auch die Download- und Parsing Zeiten optimiert, begann.

Die Entstehung von WebAssembly

Im Jahr 2015 wurde als Reaktion auf diese Fragestellung die WebAssembly Working Group gegründet. Das Ergebnis dieser Working Group war ein Format, welches unterschiedlichste Anforderungen unter einen Hut bringt. WebAssembly ist Bytecode für das Web. Es handelt sich hier im ein binäres Format für transportable Programme. Dieses Format ist optimiert auf Download-Größe sowie Ausführungsgeschwindigkeit. Interessant an diesem Format ist, dass es in Host-Umgebungen integrierbar ist. Als primäre Host-Umgebung steht natürlich der Browser im Vordergrund. Es ist jedoch auch denkbar, WebAssembly im Backend zu nutzen. Es ist nur eine Host-Umgebung erforderlich, welche den WebAssembly Standard implementiert. Die Browser Hostumgebung setzt auf der bestehenden JavaScript Runtime im Browser auf. Durch diese Integration wird die bestehende Just-in-Time Compiler Mechanik im Browser wiederverwendet. Für die Browserhersteller wurde es so wesentlich einfacher gemacht, WebAssembly zu integrieren.

WebAssembly 1.0 MVP

Anfang 2017 war es dann soweit. WebAssembly wurde in Form eines Minimum-Viable-Product in Version 1.0 veröffentlicht. Bemerkenswert war, dass alle Browser Hersteller im Abstand von wenigen Tagen den WebAssembly Support ausrollten. Was ist nun Teil des MVPs? Schon im initialen Design wurden bewusst Stolpersteine ausgeklammert, um die erste Version des Produktes schlank zu halten. Jedoch wurde gleichzeitig auch dafür gesorgt, dass diese Abgrenzung keine größeren Nachteile verursacht. Beispielhaft beinhaltet die WebAssembly Host Umgebung eine Sandbox, in der das Programm abgeschottet läuft. Diese Sandbox ermöglicht den Zugriff auf einen linearen Speicherbereich. Dieser Speicherbereit

ist jedoch nicht managed. Der WebAssembly Autor muss das Speichermanagement und ggf. auch einen Garbage Collector selber implementieren. Dies ist jedoch kein größerer Nachteil, da WebAssembly ein Modul-System für dynamisches Linken beinhaltet. Es ist damit möglich, einen bestehenden Memory-Manager einfach in das WebAssembly zu linken und darauf aufzubauen. Ebenfalls wurde im MVP bewusst kein DOM Zugriff oder die opaque Datatypes implementiert. Es ist aus der WebAssembly Sandbox nicht direkt möglich, auf das DOM oder Browser APIs zuzugreifen. Auch dies ist auch kein größerer Nachteil, da über das Modul System dieser Zugriff über Wrapper-Types emuliert werden kann. Die Wrapper-Types verursachen zwar gewisse Laufzeit-Nachteile, diese sind jedoch im Vergleich mit der reduzierten Komplexität zu verkraften. Im MVP wurde bewusst auf Unterstützung für Threading und Thread Synchronisierung verzichtet. Dies ist einleuchtend, da die WebAssembly Host Umgebung auf der JavaScript Integration im Browser aufbaut. In JavaScript gibt es keine Threads und keine Thread Synchronisierung. Warum sollte also WebAssembly das in der Version 1.0 beinhalten? Dieser reduzierte Funktionsumfang ist jedoch kein Widerspruch. Durch die bewusste Ausklammerung dieser Funktionalitäten und gleichzeitiger Unterstützung eines Modul-Systems ist eine konsistente, nutzbare und vor allem erweiterbare WebAssembly Version 1.0 entstanden.

Kernkonzepte und Bootstrap

Ok, genug der Theorie. Wie funktioniert nun WebAssembly? WebAssembly selber kommt in zwei verschiedenen Ausprägungen vor, der textuellen und der binären Repräsentation. Die Textform(WAT) ist für uns Menschen gedacht, und soll uns vor allem beim Debuggen unterstützen. Das Textformat ist jedoch nicht besonders effizient für einen Computer lesbar. Deshalb gibt es das Binärformat(WASM), welches nur für den Computer gedacht und auf besonders effiziente Verarbeitung optimiert ist.

Diese beiden Formate sind sehr hardwarenah. Als Entwickler können wir solche Programme schreiben. Das wäre jedoch nicht wirtschaftlich. Der eigentliche Zweck von WebAssembly ist ein anderer. Als Entwickler schreiben wir Programme in einer Hochsprache, wie z.B. C++, Rust, .NET, Java oder auch Kotlin. Ein Compiler übersetzt diese Sprache dann nach WebAssembly. WebAssembly ist also primär ein Compile-Target für eine Hochsprache unserer Wahl. Wir haben somit das Beste aus zwei Welten: hohe Produktivität in einer bekannten Hochsprache und maximale Laufzeitoptimierung durch Einsatz eines optimierten Binärformats.

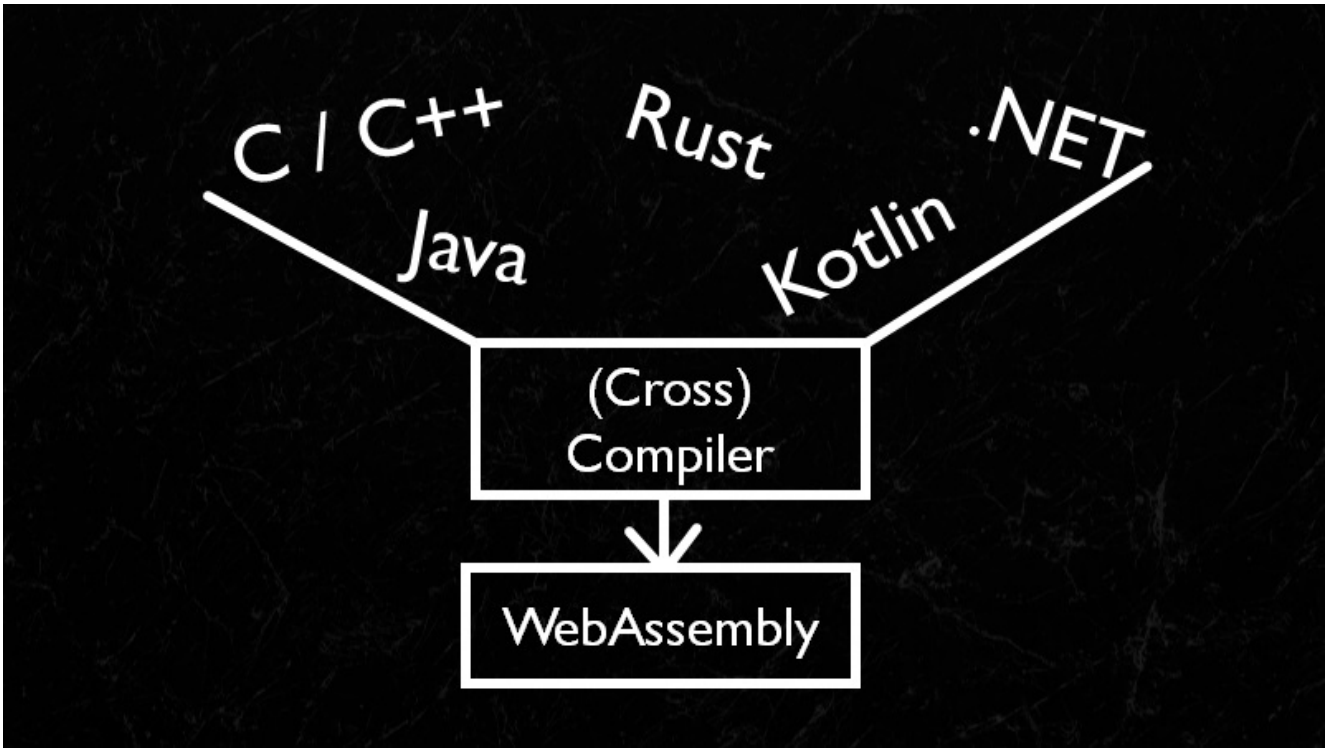


Abbildung 1: WebAssembly ist ein Compile Target

Wie funktioniert nun das WebAssembly Modulsystem? WebAssembly kennt sog. Module und Instanzen. Ein Modul ist eine Art Schablone, aus der mehrere Instanzen erzeugt werden. Jede Instanz bekommt ihre eigene Sandbox und kann darin ausgeführt werden. Das WebAssembly Modul hat eine Liste von Imports und Exports. Ein Export ist eine Funktion, welche von Außerhalb der WebAssembly Instanz aufgerufen werden kann, sprich aus der WebAssembly Host Umgebung. Der naheliegendste Anwendungsfall ist hier der Aufruf der Main Function des Programms. Interessant werden die Imports. Ein Import ist eine Funktion, welche von innerhalb der WebAssembly Instanz aufgerufen werden kann, jedoch nicht in der Instanz definiert ist. Importierte Funktionen werden z.B. für Interaktion mit der WebAssembly Host Umgebung benötigt. Jede Form von I/O wie eine Konsolen-Ausgabe oder eben auch ein DOM Zugriff wird üblicherweise in Form von Imports in das WebAssembly gelinked.

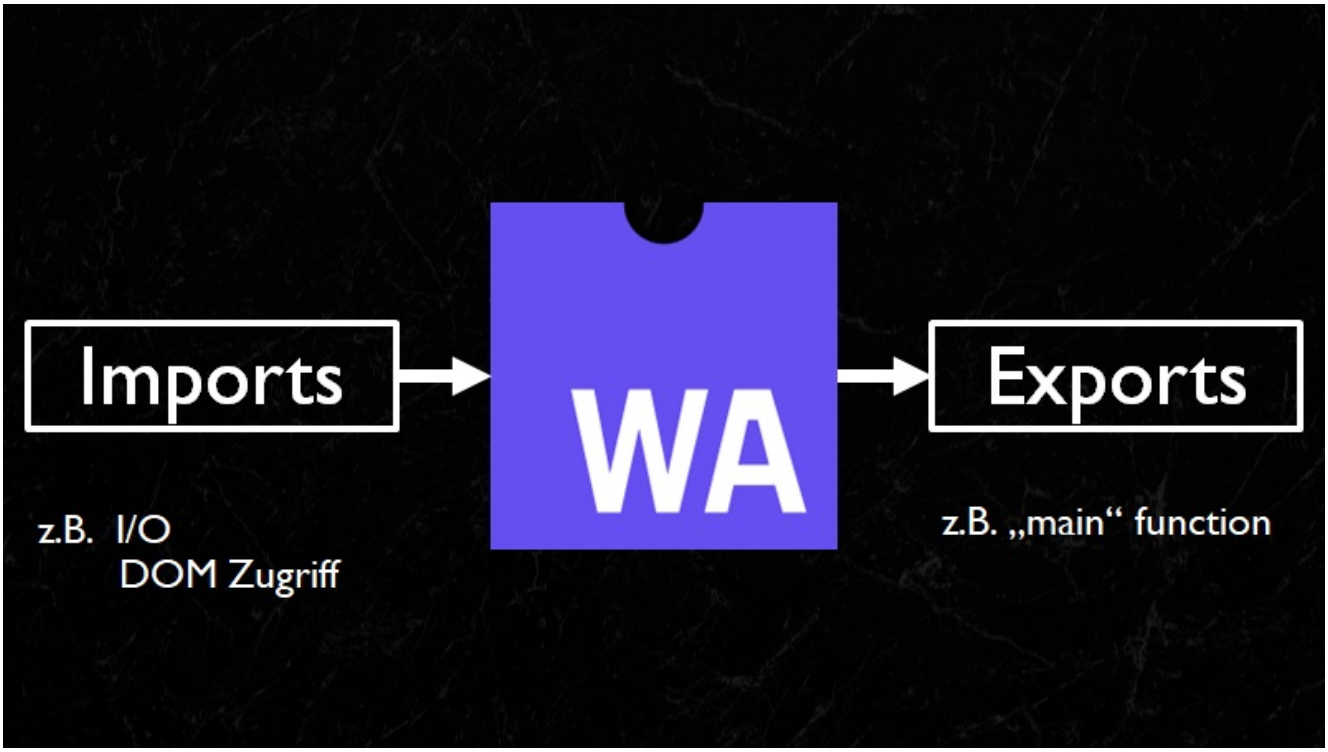


Abbildung 2: Das WebAssembly Modulsystem

Wie funktioniert nun dieses Linken? Um eine lauffähige WebAssembly Instanz zu erhalten, muss eine Bootstrap Sequenz auf dem WebAssembly Host durchlaufen werden. Im Browser gibt es hierfür eine JavaScript API. Im ersten Schritt muss die binäre WebAssembly Repräsentation geladen werden. Dies kann z.B. über einen XMLHttpRequest erfolgen. Wichtig hier ist, dass wir kein JSON oder XML Dokument als Antwort erwarten, sondern einen ArrayBuffer mit den geladenen Binärdaten.

```
var request = new XMLHttpRequest();
request.open('GET', 'bytecoder.wasm');
request.responseType = 'arraybuffer';
request.send();
```

Wenn nun das Promise fulfilled wird, können wir via WebAssembly JavaScript API die Instanziierung starten. An diesem Punkt müssen wir die Imports für das WebAssembly Modulsystem angeben. Ein fehlender Import führt zum Abbruch der Instanziierung.

```
request.onload = function() {
    var bytes = request.response;
    WebAssembly.instantiate(bytes, {
        // Imports
        mymodule: {
```

```
            add: function(a, b) {
                return a + b;
            }
        }
    });
};
```

WebAssembly.instantiate selber liefert ein Promise zurück. Bei der Instanziierung wird die WebAssembly-binäre Repräsentation validiert und kompiliert. Um den Main-Thread im Browser nicht zu blockieren, findet dies asynchron statt. Wenn dieser Vorgang abgeschlossen ist, wird das Promise mit den Referenzen auf das WebAssembly Modul und die WebAssembly Instanz fulfilled.

```
WebAssembly.instantiate(… ).then(function(result) {
    // Zugriff auf Modul und Instanz
    var wasmModule = result.module;
    var runningInstance = result.instance;
    // exportierte Funktion aufrufen
    runningInstance.exports.main();
});
```

Jetzt können wir den Kontrollfluss an die WebAssembly Instanz übergeben, indem wir z.B. eine exportierte Main Function aufrufen.

Lasst uns spielen!

Soweit, so gut. Welche Möglichkeiten habe ich, erste Erfahrungen mit WebAssembly zu machen? Zum Glück gibt es da eine sehr interessante Webseite: WebAssembly Studio. WebAssembly Studio ist JSFiddle für WebAssembly. Über diese IDE kann sehr einfach ein C oder Rust Projekt angelegt und dieses dann nach WebAssembly kompiliert werden. Der JavaScript Code für die Bootstrap Sequenz wird automatisch generiert und kann auch nachträglich editiert werden. Es ist auch möglich, die WebAssembly Text- oder Binärform genauer unter die Lupe zu nehmen und so tiefer in die Funktionsweise abzutauchen.

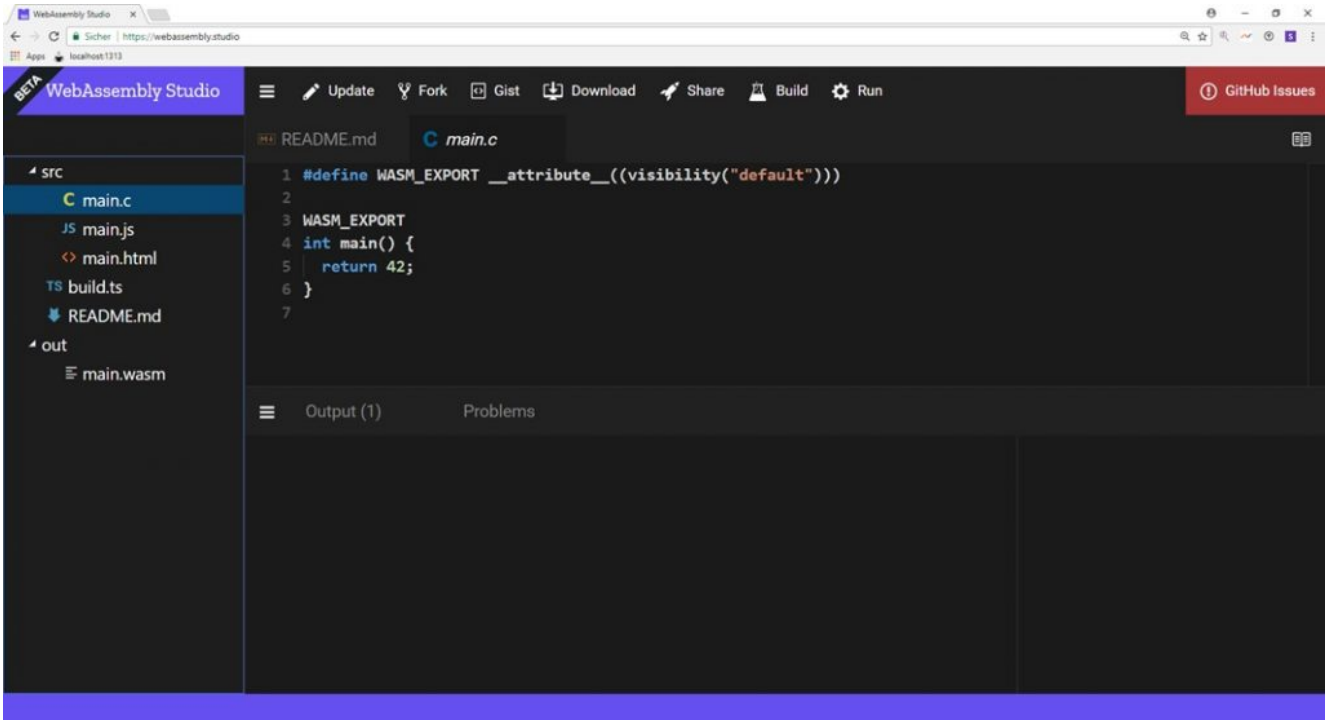


Abbildung 3: WebAssembly Studio

Wer nicht direkt Quellcode schreiben möchte, sondern eher eine lauffähige Anwendung benötigt, kann z.B. das Unity Tanks! Demo spielen. Hier handelt es sich um ein kleines Spiel, in dem Panzer in einer Landschaft bewegt werden können. Die Grafik ist sehr schön über WebGL umgesetzt. Hier zeigt sich sehr gut, wie WebAssembly bereits heute in Lösungen wie der Unity Game Engine integriert ist.

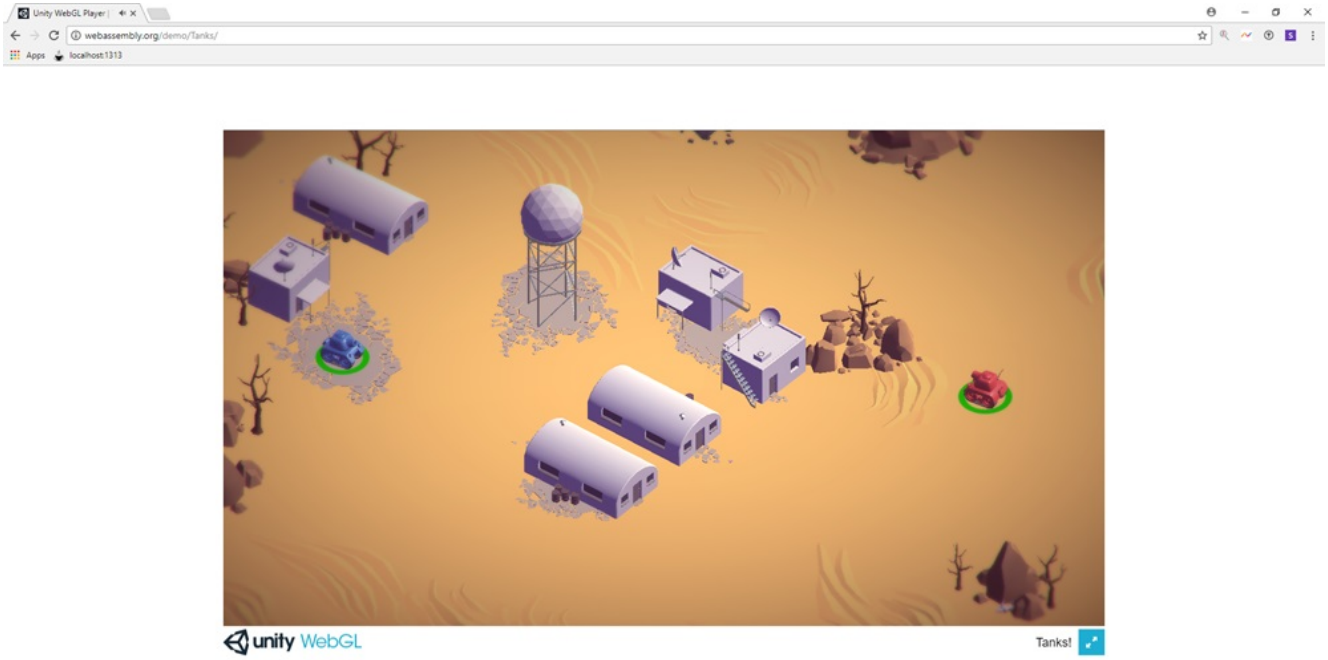


Abbildung 4: Unity Tanks Demo

Was mit der Unity Engine funktioniert, funktioniert auch mit der Unreal Engine. Das Sun Temple Demo zeigt sehr eindrücklich, wie WebAssembly und WebGL zu einem digitalen Kunstwerk verknüpft werden.

Abbildung 5 : Unreal Sun Temple Demo

Andere Anwendungsfälle

Es gibt natürlich auch Anwendungsfälle neben dem Gaming Bereich. Einer davon ist Pocketsphinxs.js. PocketSphinx ist ein System für Spracherkennung. Dieses System wurde via Emscripten nach WebAssembly übersetzt. Damit ist es möglich, Spracherkennung in Applikationen zu nutzen, ohne Remote-Services wie Alexa in eine Anwendung zu integrieren.

WebSight ist ein anderer Anwendungsfall. Hier wurde OpenCV via Emscripten nach WebAssembly übersetzt. Diese Demo greift auf die Webcam zu und sucht in dem Live Video Stream nach Gesichtern. Interessant hier ist, dass gleich drei Varianten der Bildverarbeitung in Form von WebWorker Instanzen parallel laufen. Eine WebAssembly Variante, eine asm.js Variante und eine JavaScript Variante. Hier offenbart sich der mögliche Performance-Gewinn deutlich: die WebAssembly

Variante ist um Faktor 15 schneller als die JavaScript Variante!

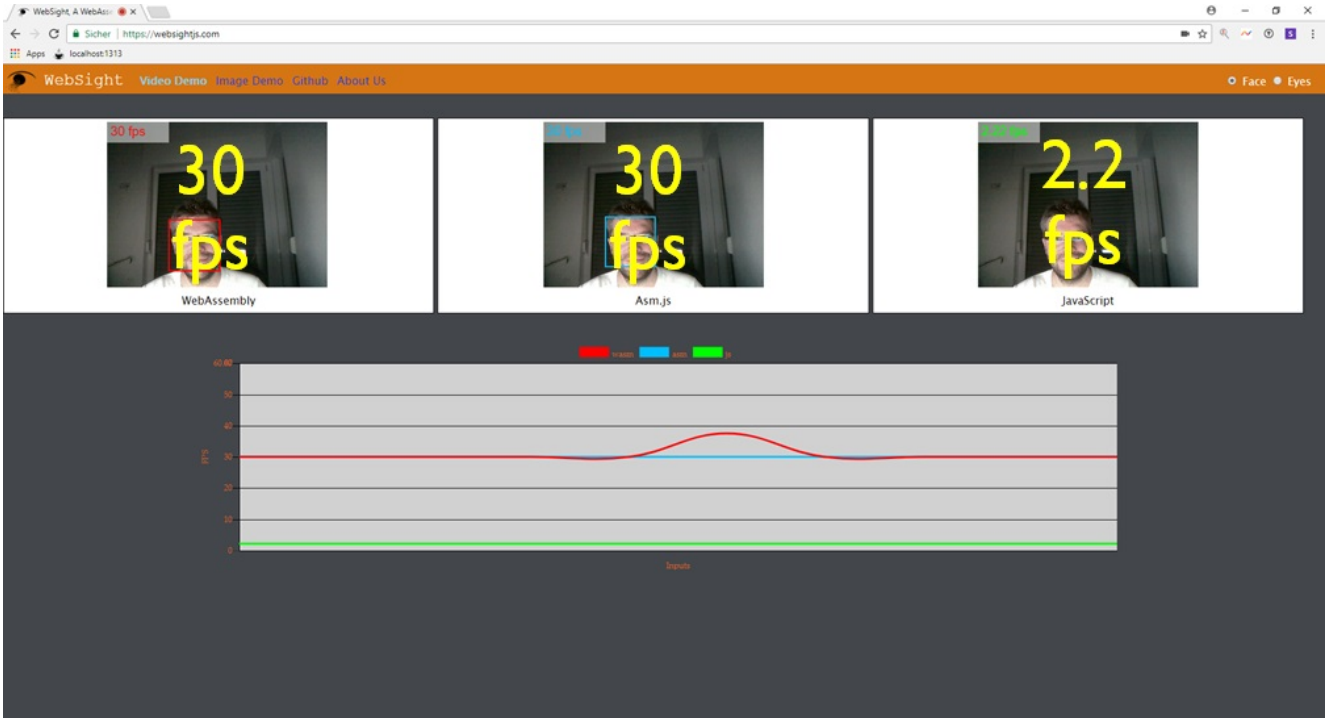


Abbildung 6: WebSight Demo

Missing Parts

Wo Licht ist, ist auch Schatten. Das WebAssembly MVP funktioniert, jedoch zeigen sich in der aktuellen Version besonders die Stellen, an denen bewusst abgespeckt wurde. Aus Sicht Entwickler fällt besonders das noch sehr einfache Tooling rund um die WebAssembly Integration im Browser auf. Es gibt im Moment nur sehr einfache Debugging Möglichkeiten. Die ersten Browser beinhalten Sourcemap-Unterstützung für WebAssembly, diese könnte jedoch noch deutlich optimiert werden. Durch das Fehlen eines Memorymanagers und einer Garbage Collection Integration muss diese Funktionalität via Modul-System importiert werden. In der Konsequenz verursacht dieser Runtime-Code ein unnötig großes WebAssembly Binary. Das Fehlen einer direkten DOM API oder des Zero-cost Exception Handlings im MVP verursacht ebenfalls Gluecode zur Emulation von Hochsprachen mit Exception Handling, welcher eigentlich nicht notwendig wäre. Unterstützung für Threading wäre wünschenswert, jedoch kann im MVP auch ohne native Thread Unterstützung arbeitet werden. Das WebSight Demo zeigt es: Threads können zusammen mit der WebWorker API emuliert werden. Dies ist für die meisten Anwendungsfälle eine gute Grundlage.

Ausblick

Auch wenn das WebAssembly MVP ein paar Ecken und Kanten hat, so ist es doch schon sehr gut einsetzbar. Die Unity Game Engine nutzt bereits heute WebAssembly für die Spieleentwicklung. Unity kann jedoch mehr. Denkbar sind hier alle Formen von besonders interaktiven Produktdemonstrationen, welche von WebAssembly profitieren können. Besonders in Verbindung mit Virtual Reality oder Augmented Reality ergeben sich völlig neue Anwendungsfälle. Eine weitere, interessante Möglichkeit ist die Migration von Legacy Code ins Web. Durch moderne Compiler können wir funktionierende und getestete Software in neuen Umgebungen laufen lassen wie zum Beispiel WebSight OpenCV auf einer Webseite. Die Möglichkeiten sind grenzenlos. Untermauert wird diese Perspektive durch die Übergabe der WebAssembly-Core-Spezification an das W3C. Diese Übergabe ist im Februar 2018 erfolgt. WebAssembly ist somit ein offizieller Standard mit unglaublich viel Potential für die Zukunft. Das Web war schon immer gut für Überraschungen und wird es mit dieser Technologie auch weiterhin bleiben!

Thalia auf der jSpirit Unconference

Dieses Jahr haben wir an der gerade erst gegründeten jSpirit Unconverence teilgenommen. Der Teilnehmerkreis erreicht zwar noch bei weiten nicht die geplanten 100 Personen, setzt sich dafür aber aus sehr guten, erfahrenen Entwicklern zusammen. Zum einen kennen sich einige sich schon länger von diversen Konferenzen in Deutschland, Basel, Moskau, Korea, Japan und USA und hält teilweise selbst mehrere dutzend Vorträge im Jahr oder organisiert Veranstaltungen (z.B. die JavaLand oder den JUG Oberland) und zum anderen wir die Runde ergänzt durch sehr erfahrene Entwickler aus der Region Bayern, aber auch Teilnehmer aus der Schweiz und der USA waren mit dabei.

Für mich war das unconference Format noch neu und daher war ich sehr gespannt, ob eine Konferenz ohne Agenda überhaupt etwas bringt. Aufgrund des zwanglosen Formats kommt man aber sehr schnell in einen direkten Austausch mit den anderen und so zu einem sehr viel intensiveren Erfahrungsgewinn. So

wurde ich dann auch völlig überrumpelt, wie intensiv der Austausch und die Zusammenarbeit von der ersten Minute ab war.

Die Themen wurden morgens vor der ersten Session von allen Teilnehmern eingebracht und per Abstimmung auf die freien Timeslots verteilt. Dadurch waren für alle durchgehend spannende Themen mit dabei. Wie bei Openspaces überlich, konnte man auch jederzeit innerhalb einer Session zur anderen Gruppen wechseln, wenn man zu dem aktuellen Thema nichts mehr beisteuern konnte. Innerhalb der einzelnen Sessions gab es einen sehr guten Austausch der Erfahrungen und best practices der einzelnen Teilnehmer. So war es ein echter Vorteil, dass nicht nur ein Vortragender etwas erzählte sondern ein direkter Austausch stattfand. Dadurch konnten die einzelne Vorschläge schnell bestätigt oder mögliche Alternativen genannt werden.

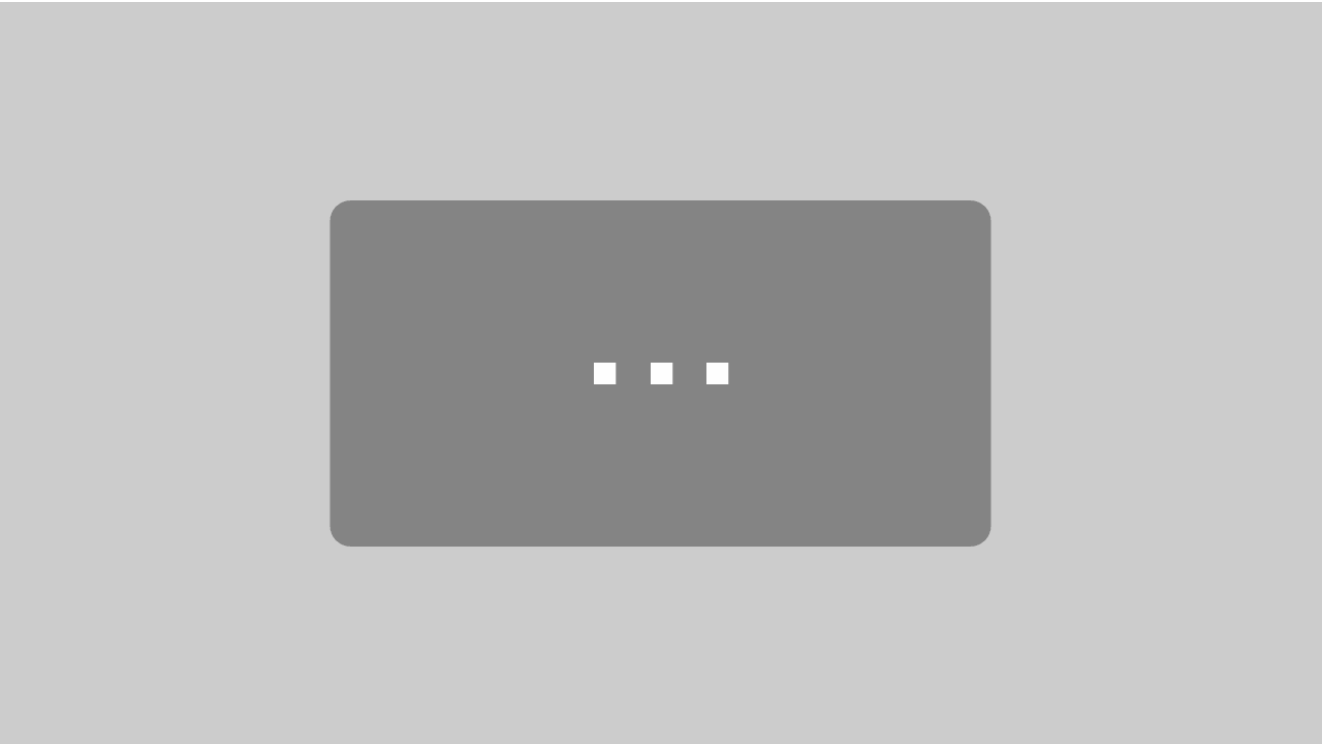
Ein kurzer (unvollständiger) Auszug aus den Themen, die besprochen wurden:

* Build Tools – Maven vs Gradle ?
* Next thing to learn ?! (Go, Rust, Pony, AI ?)
* Functional Patterns with Core Java
* Java After Eigth (9,10,11)
* Java in the Cloud
* BDD – But Test suck
* Dealing with difficult people on work
* Speaker Tips (Abstract, Topics, HowTo)
* Ideas for teaching Kids
* Information flood
* Mob – Programming

Für mich war das inhaltlich eine der besten Konferenzen auf dehnen ich bisher je war.

Ergänzt wurde die Veranstaltung durch ein sehr gutes Rahmenprogramm. So bot sich aufgrund der Lokation natürlich die Gin & Whiskey Verköstigung inkl. Führung durch die Destillerie an, aber auch ein Ausflug zur Alm mit anschließender Rodeltour gehörten mit dazu. Das Programm sorgte dafür, dass der Teilnehmerkreis vollständig bis in die späten Abendstunden zusammenblieb. So war ein fachlicher Austausch auch in diesem geselligen Rahmen weiter gegeben und brachte einige zusätzliche Erkenntnissgewinne.

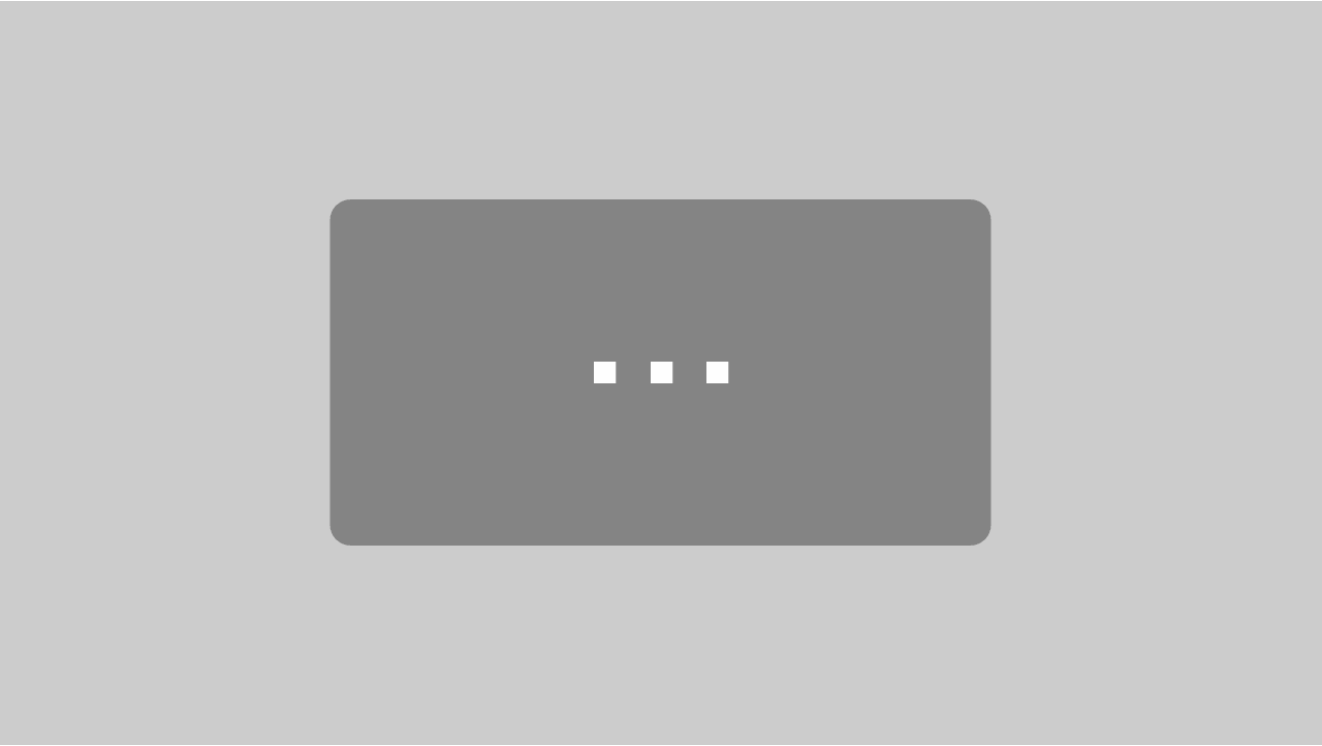
Hier noch ein paar Links auf verschiedene Interviews Live von der jSpirit:



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

Video laden

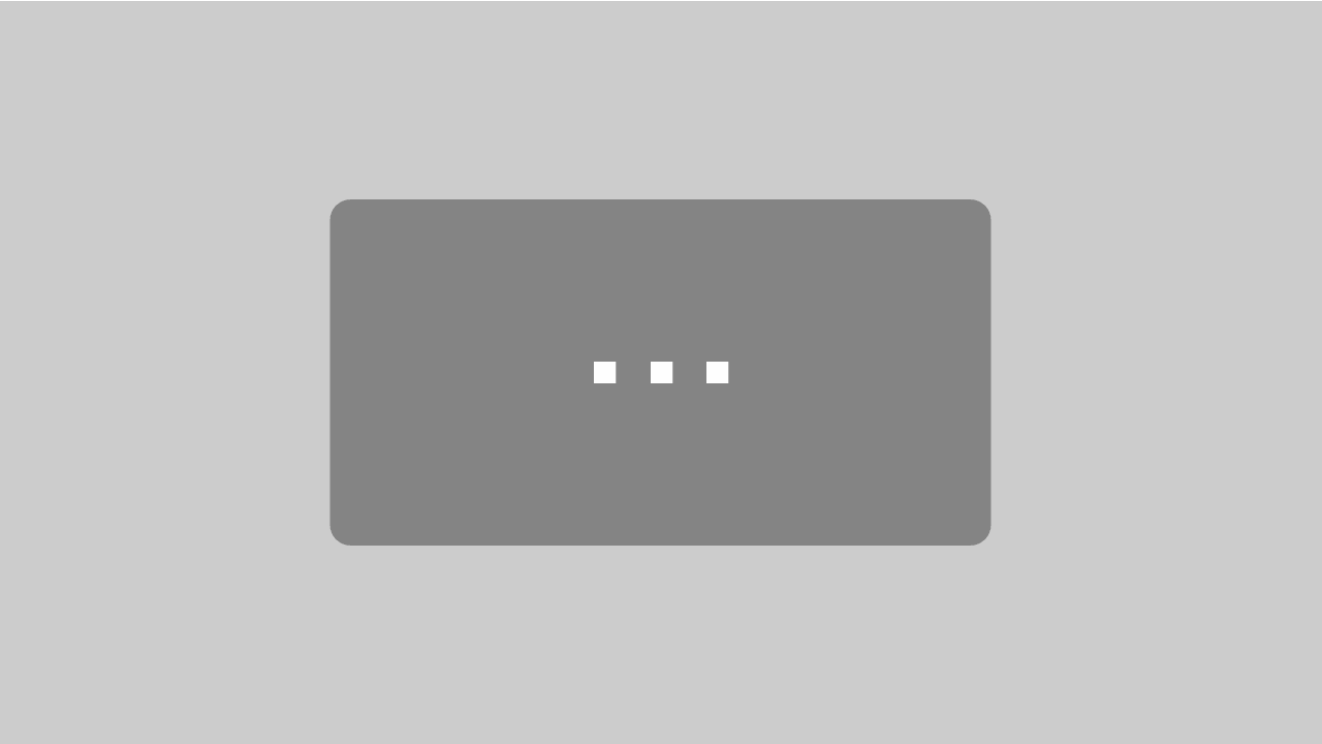
☐ YouTube immer entsperren



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

Video laden

☐ YouTube immer entsperren

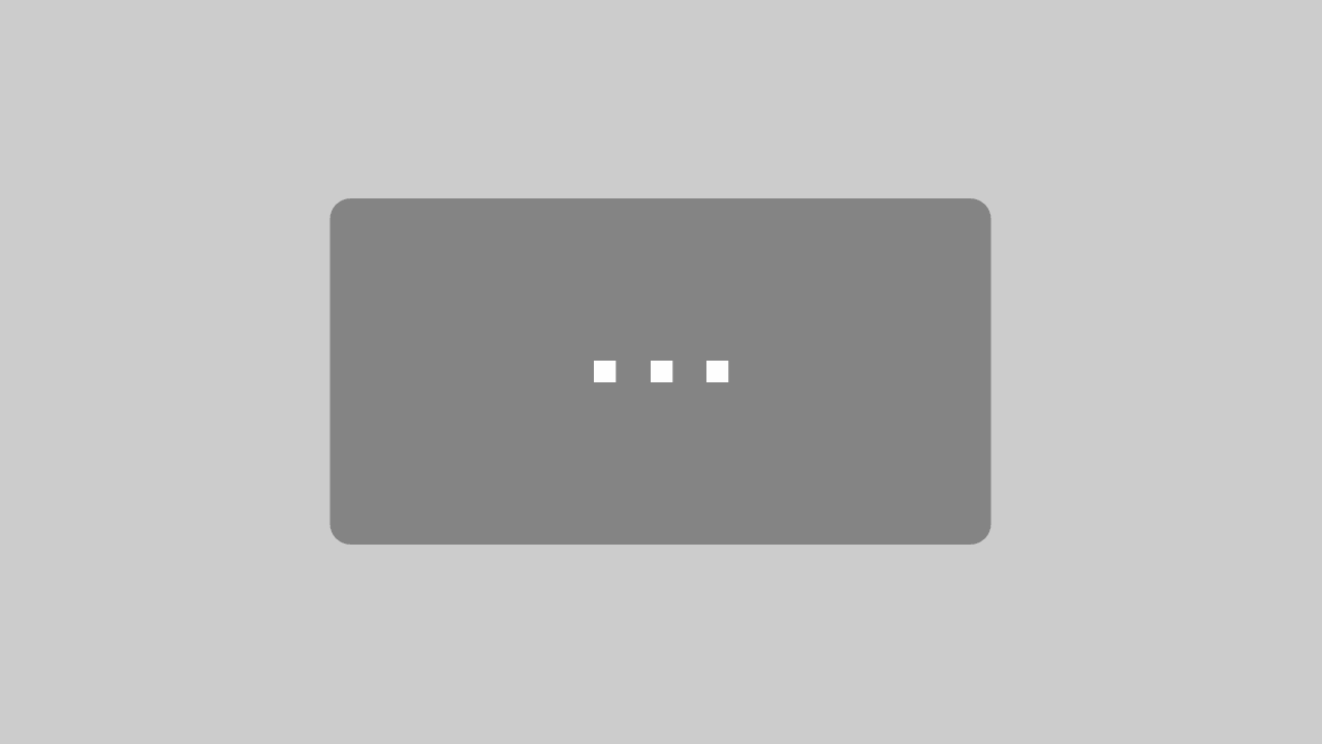


Mit dem Laden des Videos akzeptieren Sie die Datenschutzerklärung von

YouTube.
Mehr erfahren

Video laden

☐ YouTube immer entsperren



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

Video laden

☐ YouTube immer entsperren

jSpirit ist eine neue Java Unconferenz in den Bayrischen Alpen
jspirit.org oder auf Twitter unter #jspiritunconv

Veranstaltungsort: www.lantenhammer.de

Kids Programming – Thalia fördert bereits die Jüngsten!

Nachdem wir in den letzten Jahren einige Sommerpartys und eine riesige Weihnachtsfeier hatten, hat sich unsere Geschäftsleitung entschieden, für Ende 2017 einen „Family Day“ abzuhalten. Und so durften nicht nur die Lebensgefährten sondern auch alle Kinder mal schauen, was Papa / Mama so im Büro machen. Es gab an den einzelnen Standorten ein umfangreiches Rahmenprogramm. Dabei wurde stets versucht, auch einen gewissen Bezug zum Unternehmen zu wahren. So bot sich die Bücherecke mit Vorlesungen für Kinder natürlich an, was vor allem einige der jüngeren Kinder sehr genossen haben. Aber was macht man an einem Software Standort und wie erklärt man die Entwicklung von Programmen seinen Kindern?

Inspiriert von der „Langen Nacht der Wissenschaft“ in Berlin, wurde die Idee des Kids Programming geboren. An einzelnen Plätzen konnten so die Kinder bereits ab 4 Jahren in die Welt des Programmierens eingeführt werden und eine Idee entwickeln, was ein Software Entwickler eigentlich macht.

Die grundlegende Idee

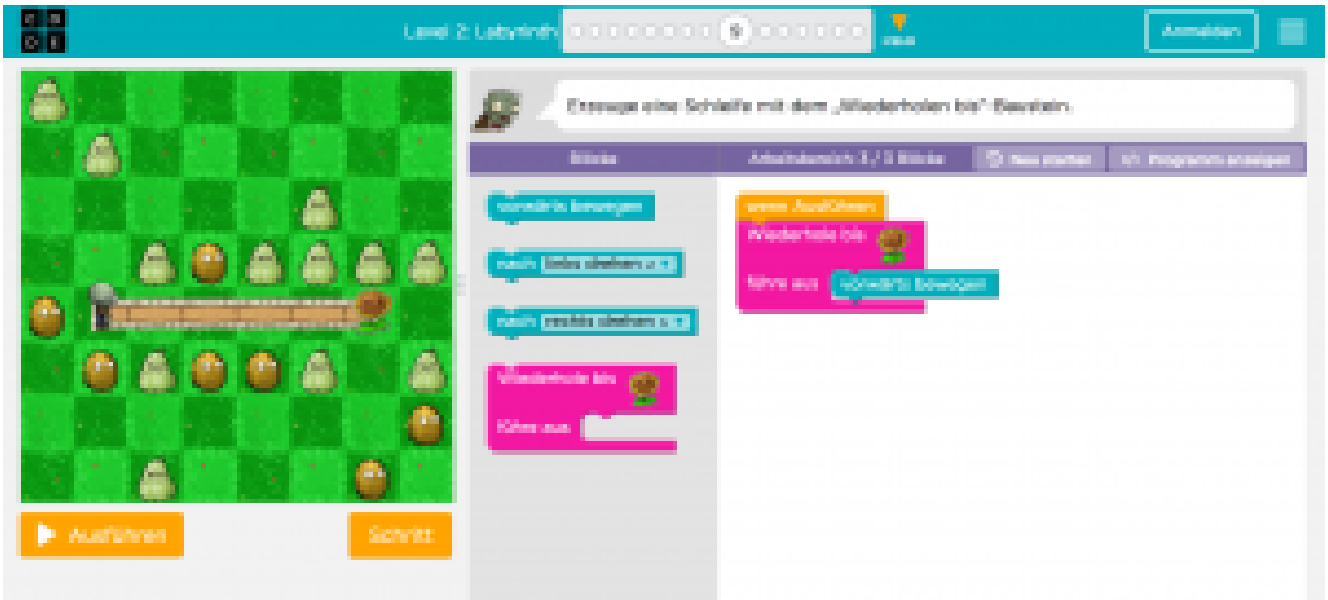
Wir haben dafür mit Hilfe von code.org einige Programme für die Altersgruppe 4-6 und 6-12 rausgesucht und unterschiedliche Themen gewählt. Dadurch

konnten wir später nicht nur Jungen sondern auch einige Mädchen für die Kurse begeistern. Für die fortgeschrittene Programmierung stellten wir darüber hinaus noch Scratch vom MIT zur Verfügung und konnten das Ganze durch einige, speziell für Kinder gestaltete Lehrbücher, unterstützen.

Darüber hinaus wollten wir bei den Kindern aber auch das Interesse für Softwareentwicklung wecken und sie neugierig auf mehr machen. Deshalb haben wir unsere Raspberry Farm ebenfalls zur Scratch Spielwiese umfunktioniert und mit einigen Hardware Projekten ergänzt. Das wurde dann zum echten Blickfang und sorgte für die Aufmerksamkeit von jung und alt. So bekamen wir am Ende mindestens genauso viel interessierte Eltern wie Kinder in den Kurs.

Programmieren mit code.org und Scratch

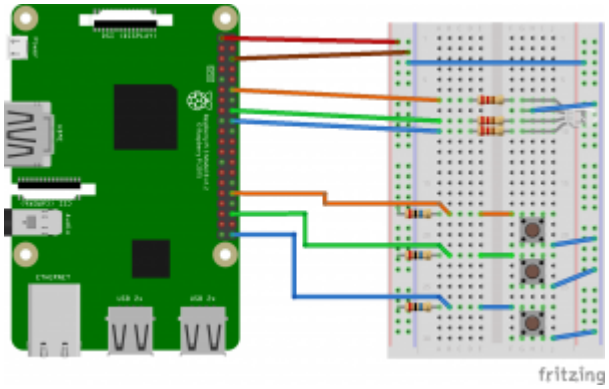
Die die Idee dahinter ist nicht eine echte Programmiersprache zu erlernen, sondern ein Verständnis von strukturierten Abläufen und Funktionsbausteinen zu schaffen. So können die Kinder mit einzelnen Code-Blöcken, im Lego Baukasten Prinzip, ihre Programme zusammenklicken. Dabei geht es vom Erkennen von Wiederholungen / Schleifen bis zum Erstellen eigener Funktionen. Während Scratch eine reine Entwicklungsumgebung bietet, unterstütz code.org das mit stuckierten, auf einander aufbauenden Kursen.



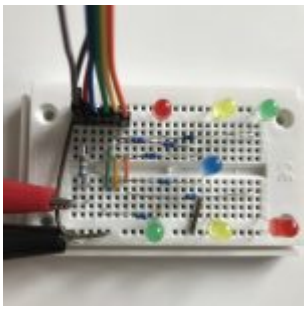
Programmieren von Hardware

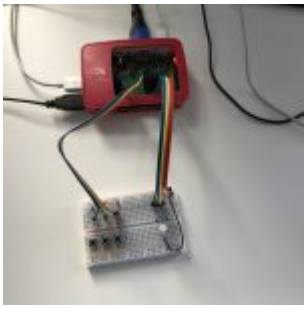
Als letzte Stufe haben wir den Kinder gezeigt, dass mit einfachen Programmen sogar elektronische Schaltungen gebaut und programmiert werden können. Zur Auswahl standen drei Beispiele:

1. Steuerung einer RGB LED mittels dreier Taster,
2. Steuerung einer Fussgängerampel mit dazugehöriger Fahrzeug Ampel und
3. Steuerung eines RGB Würfels, der durch zwei Knetkontakte gesteuert wird.



Die Programmierung erfolgt dabei auch mit Hilfe von Scratch in einer speziellen Version für Raspberry Pi. In dieser Version können die Ein- & Ausgänge als Funktion in Scratch verwendet und gesteuert werden. Inspirirt wurden diese Beispiele durch das Buch „Der kleine Hacker – Programmieren für Einsteiger“. In diesem sind die Schaltungen und Programme sehr schön auch für unerfahrene Einsteiger beschrieben. Bis auf den Raspberry wird auch alles benötigte Zubehör mitgeliefert.





Wer Interesse bekommen hat, das Ganze mal selber auszuprobieren, wird hier fündig:

Christian Immler

Der kleine Hacker: Programmieren für Einsteiger

Mit Scratch schnell und effektiv programmieren lernen

Der kleine Hacker

★★★★★

eBook ab 14,99 €

Taschenbuch 25,99 €

Programmieren ist langweilig und trocken? Nicht mit dem kleinen Hacker! Die grafische Programmiersprache Scratch macht's möglich. Programmieren lernen mit Spaß. Egal, ob du nur die Katze tanzen lassen oder ein richtiges Spiel programmieren willst - Scratch eignet sich sowohl zum Einstieg in die Programmierung als auch für anspruchsvollere Projekte.

Links zu den Seiten: code.org, scratch.mit.edu

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

Spellchecking in der Suche – Von

Andrew Fiala und dem System Wesley

Als Händler mit mehr als 10 Millionen Artikeln, die zumeist auch noch umgangssprachlich benannt sind, muss man einiges an Herausforderungen im Bereich der Suche meistern, um den Kundenwünschen gerecht zu werden. Einer der Aspekte, mit denen man klarkommen muss, sind die vielen kreativen Schreibweisen, mit denen Kunden nach Büchern, Filmen, etc. suchen – was ihnen durch schwungvoll fremdsprachige Titel und Kunstworte auch nicht leicht gemacht wird.

Um den manuellen Aufwand bei der Pflege der Suche möglichst gering zu halten, werden Komponenten benötigt, die automatisch und möglichst intelligent die Suchbegriffe des Kunden interpretieren und das Passende heraussuchen. Dieser Artikel beschäftigt sich mit dem Mechanismen des Spellcheckings, d.h. der Korrektur falsch geschriebener Suchbegriffe.

Der Mix machts

Im Bereich der eCommerce-Suche gibt es keinen „goldenen Hammer“, der magisch alle Probleme, Missstände und Datenlücken behebt. Das gilt auch für die Korrektur von Benutzereingaben. Für den Thalia-Webshop setzen wir Lucidworks Fusion ein, das unter der Haube die bewährte OpenSource-Suchetechnologie Apache Solr verwendet.
Innerhalb von Solr gibt es gleich einen ganzen Satz an Spellchecker-Komponenten, aus denen der geneigte Verwalter sich eine für seinen Anwendungsfall am besten geeignete Lösung „zusammenstöpseln“ kann. Jede dieser Komponenten ist für einen bestimmten Anwendungsfall gedacht, eine Gesamtfunktionalität ergibt sich erst aus der Kombination der einzelnen Bausteine.

Heißt es *Lalaland*, *Lala Land* oder *La La Land*? Damit ein Kunde das nicht wissen muss, gibt es den `WordBreakSolrSpellChecker`, der zusätzliche Worttrennungen hinzufügt oder bestehende entfernt, um den Treffern im Index näher zu kommen.

T.C. Boile wird durch durch `DirectSolrSpellChecker` zum Bestsellerautor *T.C. Boyle*, indem alternative Schreibweisen in einer gewissen Levenshtein-Distanz zum Ausgangsbegriff gebildet werden.

Diese beiden Spellchecker-Komponenten lösen – zusammen mit einem ordentlichen Synonymwörterbuch – nahezu alle falschen Schreibweisen in Ein-Wort-Suchen.

Mehr Wörter, mehr Probleme

Der Kunde hat von einer neuen Serie gehört, von Kritikern gelobt, bei Zuschauern hoch umstritten. Aber wie hieß die noch? Achja: *Tote Mädchen liegen nicht*. Nah dran, aber sofern der Kunde nicht eine mäßige Teenager-Zombie-Serie suchen wollte, müssen wir ihm hier etwas unter die Arme greifen, damit er zu seiner eigentlich gemeinten Serien *Tote Mädchen lügen nicht* kommt.

Hier kommt zunächst das Thema Collations ins Spiel, das Solr veranlasst, einen oder mehrere neue Suchbegriffe aus den „Verbesserungen" der einzelnen Tokens zu generieren. Da die oben erwähnten Komponenten nur auf Token-Ebene arbeiten, führt dies zu eher humoristisch-poetischen Korrekturvorschlägen wie *rote Märchen liehen Nacht* oder *Täte Mädchen liefen noch*, da nur jedes einzelne Token für sich mutiert wird und aus der Vielzahl von Umschreibungen dann neue Begriffe gebildet werden. Das ist langsam, ineffizient und vor allem nicht zielführend: man kann innerhalb der kurzen Zeitspanne, die ein Kunde bereit ist zu warten, nicht beliebig viele Permutationen durchrechnen und bewerten (dazu gleich mehr). Also setzt man dem ganzen ein abruptes Ende und wenn sich die richtige Version des Suchbegriffs nicht zufälligerweise unter den neu gebildeten befindet, hat man nicht nur viel Zeit aufgewendet, sondern steht auch noch mit leeren Händen da und kann dem Kunden nur den äußerst umsatzfeindlichen Satz *„Wir haben leider keine Ergebnisse zu Ihrer Suche gefunden."* anbieten.

Um auch bei Mehrwort-Suchen eine sinnvolle Verbesserung anbieten zu können, nutzen wir zusätzlich noch die `ShingleFilterFactory`, die den Suchbegriff in Wortgruppen zerlegt und diese in den weiteren Schritten als ein Token behandelt, also eine unzertrennliche Sammlung aus Zeichen. Die minimale und maximale Shingle-Länge ist konfigurierbar und wirkt sich darauf aus, wieviele Tokens mindestens oder höchstens in einem Shingle enthalten sein werden.

Nehmen wir für das Beispiel 1 als Minimum und 4 als Maximum, ergeben sich folgende „Shingle-Tokens“:

- Tote
- Mädchen
- liegen
- nicht
- Tote Mädchen
- Mädchen liegen
- liegen nicht
- Tote Mädchen liegen
- Mädchen liegen nicht
- Tote Mädchen liegen nicht

Damit diese Bildung korrekt funktioniert, muss der Suchbegriff `q=...` in einem explizit aufgeführten Parameter `spellcheck.q=...` zusätzlich an den Spellchecker übergeben werden. Nur dann funktioniert die Shingle-Bildung korrekt, da Solr andernfalls zuerst eine Aufteilung an Leerzeichen vornimmt und keine Shingles gebildet werden.

Die so erzeugten Shingles werden dann von den bereits bekannten Spellchecker-Komponenten überprüft und verändert und hier merkt der `DirectSolrSpellChecker` recht schnell, dass sich *Tote Mädchen liegen nicht* mit einer Editierdistanz von 2 in das im Index vorkommende *Tote Mädchen lügen nicht* ändern lässt.

Mit dieser Spellcheck-Konfiguration werden nicht nur sinnvollere Vorschläge bei Mehrwort-Suchen gefunden, sondern auch noch in viel kürzerer Zeit. Gegenüber dem „dummen“ Spellchecking haben wir einen Geschwindigkeitsvorteil (abhängig vom Suchbegriff) zwischen Faktor 5 und 50 gemessen.

Viel hilft viel?

Nur Vorschläge für Suchbegriffkorrekturen erzeugen alleine reicht allerdings nicht, man muss auch noch bewerten können, ob eine Korrektur sinnvoll ist oder nicht.
Out-of-the-box bietet Solr hier die Bewertung von Alternativen durch ihre Anzahl Treffer, die sie erzeugen würde, hätte man danach gesucht. Mit dem Schalter

`spellcheck.onlyMorePopular` teilt man dem System mit, dass man nur Vorschläge sehen möchte, die mehr Treffer als der eigentliche Suchbegriff erzielen.

Was zunächst sinnvoll aussieht – viel hilft viel – erweist sich bei näherer Betrachtung als Mogelpackung: da man nicht im Voraus weiß, ob der Suchbegriff des Kunden überhaupt „falsch“ ist oder nicht, führt man immer ein „brute-force“ Spellchecking durch, d.h. es werden immer Alternativen generiert. Nutzt ein Kunde den exakten Titel eines Artikels oder sucht in einer dünn besetzten Nische, findet er nur wenige (aber richtige) Treffer, der Spellchecker schlägt aber Korrekturen mit mehr Treffern vor. Diese sind allesamt unsinnig, da der Kunde schon gefunden hat, was er sucht. So wird z.B. aus dem Romantitel *Sylter Wellen*, der nur einen Treffer findet, die Alternative *System Wesley*, was etwa 50 Treffer im Bereich der Unix-Administration liefert. Mehr Treffer: ja, höhere Relevanz: nein. Ähnlich ergeht es der Autorin *Andrea Ficala*, die bislang „nur“ ein einziges, aber gefragtes Buch geschrieben hat. Die Umschreibung zu *Andrew Fiala* liefert zwar mehr Treffer, geht aber auch am Thema vorbei und sorgt für Unverständnis beim Kunden.

Bessere Vorschlagsauswahl

Der erste Schritt ist dem Kunden die Korrekturen nur als Vorschläge zu unterbreiten und ihn nur in 0-Treffer-Fällen automatisch zu korrigieren. *„Ihre Suche nach A ergab 1 Treffer. Meinten Sie vielleicht B?“*. So kann der Kunde selber entscheiden kann, ob er die Korrektur annimmt oder nicht. Jetzt wäre es noch toll, wenn dort kein Blödsinn angezeigt werden würde.

Wir nutzen für die Vorschlagsauswahl neben der reinen Trefferzahl ein zusätzliches Kriterium: wie oft Kunden nach einem Suchbegriff suchten. Für die Vervollständigung von Begriffen in der Suchbox (Suggest) speichern wir alle Suchanfragen, die zu Treffern führten (ohne Kundenbezug). Über diese Collection lässt sich ermitteln, wieviele Kundensuchen es nach dem einen oder anderen Suchbegriff gab.

Für die Vorschläge des Spellcheckers und den ursprünglichen Suchbegriff wird ermittelt, wieviele Kunden den jeweiligen Suchbegriff gesucht haben. Ist die Anzahl der Suchen für den Begriff hoch, ist auch die Wahrscheinlichkeit hoch, dass die Korrektur richtig oder zumindest sinnvoll ist. Wir bewerten Anzahl

Suchanfragen deutlich höher als Anzahl Treffer und verhindern darüber die meisten unsinnigen Korrekturvorschläge.

Über diese zusätzliche Zählung lässt sich zudem ein Sonderfall erkennen und behandeln: es gibt Artikel, die schnell ausverkauft sind, z.B. weil sie limitiert oder nicht gut beschaffbar sind. Diese generieren in einem kurzen Zeitraum eine hohe Anzahl an Suchen, liefern aber nach ihrem Ausverkauf keinerlei Treffer mehr. Gibt es zu einem Suchbegriff keine Treffer, ist aber die Anzahl an Suchen hoch (wohlgemerkt, wir speichern nur Suchen, die zum jeweiligen Zeitpunkt auch Treffer hatten), kann daraus gefolgert werden, dass der initiale Suchbegriff mit hoher Wahrscheinlichkeit auch das war, was der Kunde suchen wollte. In diesem Fall ließe sich z.B. die Anzeige eines Korrekturvorschlags unterdrücken und eine *„Es tut uns leid, der von Ihnen gesuchte Artikel ist derzeit nicht im Sortiment.“*-Nachricht ausspielen.

Fazit

Es gibt keine allgemeingültige Lösung für das Thema Spellchecking in der eCommerce-Suche, aber mit einer Kombination aus Solr-Bordmitteln und zusätzlicher, eigener Logik und entsprechenden Verkehrsdaten lässt sich ein passables Grundgerüst aufsetzen, dass viele alltägliche Situationen automatisiert abhandeln kann.

Die Autoren

Jan Marten, Developer im Team *Suche & Beraten*

Hendrik Busch, IT Analyst im Team *Suche & Beraten*

Artikeldaten in Datenbanken: Von einem relationalen Schema zu „Dokumenten“ (2/2)

Teil 2 – Technische Alternativen und deren Vergleich

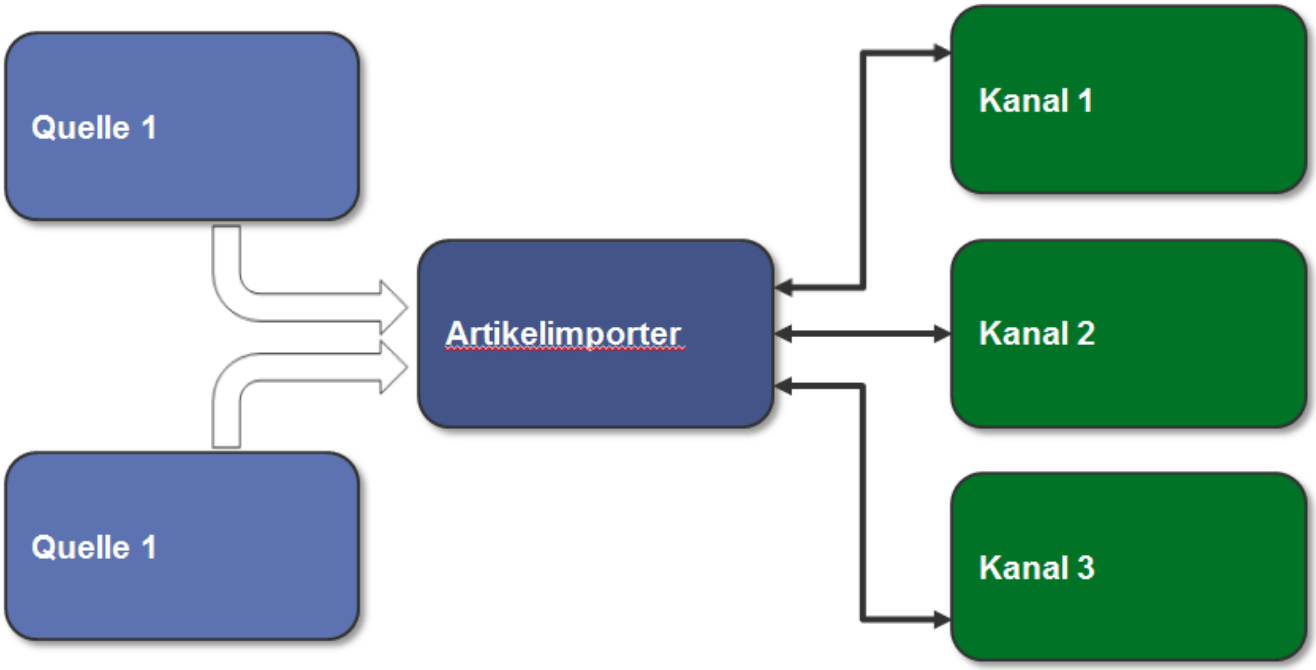
Die im ersten Teil beschriebene Problemstellung („… (wir) führen … die Artikeldaten aus unterschiedlichen Quellsystemen zusammen. Jedes Quellsystem pflegt seine eigenen Update-Frequenzen und –Formen.“) wurde in unserer Systemlandschaft vor einigen Jahren bereits in einer relativ leicht zu beschreibenden aber technisch herausfordernden Anwendung umgesetzt – weiter unten als „Legacy Variante“ beschrieben.

Fachliche Designfehler machten vor kurzem einen Neuentwurf der Anwendung notwendig. Diesen haben wir zum Anlass genommen, u.a. das Thema „Dokumente“ und deren Technologie zu erproben und so auch für andere, zukünftige Anwendungsfälle Erfahrung zu sammeln. Unsere ersten Messungen und unser Vorgehen bei der Auswahl der passenden Technologie soll Inhalt dieses Teils sein.

Die Beispielanwendung / das Szenario

Besagte Anwendung, im weiteren Text der Einfachheit halber „Artikelimporter“ genannt und etwas simplifiziert, erhält aus zwei Quellen Datensätze. Die Kombination aus einem Datensatz mit seinem Zwilling aus der anderen Quelle ergibt einen Artikeldatensatz. Das bedeutet, die Anwendung muss

1. Die Daten aus den beiden Quellen „verheiraten“, d.h. Datensatz-Hälften zu einem vollständigen Satz zusammenstellen (auf Basis eines einheitlichen Schlüssels).
2. Den kompletten Datensatz eines Artikels den Online-Kanälen (z.B. eReader, Webshop, App usw.) zur Verfügung stellen.
3. Neue „Paare“, Änderungen an diesen Datensätzen oder komplette Rückzüge (bei „Verlust“ eines Zwillings) propagieren.



Diese Anwendung haben wir (mit ein paar anderen Konzepten) neu gebaut und daraufhin den Datenstrom aus unserer Produktion abgegriffen und durch die neuen Prototypen geleitet.

Die unten dokumentierten Messungen sind also nicht aus einem schnuckeligen synthetischen Labortest gekommen („wir füllen die … mit einer Millionen … und machen dann … und wiederholen das dann … mal“), sondern auf Basis echter Anforderungen erhoben und mit der bestehenden Legacy Variante verglichen

worden.

Bisheriger Ansatz – Legacy

In der ersten Version haben wir einen generativen Ansatz zur Erstellung des DB Schemas auf Basis der Struktur der empfangenen Nachrichten (aus beiden Quellen) gewählt (HyperJaxb).

So konnten wir gerade in der Anfangszeit, als die Struktur der Datensätze noch volatil war, einiges an Implementierungsarbeit sparen. Die Persistenz wurde mit JPA realisiert. Folgerichtig ist das Datenschema stark normalisiert (jenseits 30 Tabellen für den beschriebenen Anwendungsfall).

Das Schreiben in das Datenbankschema findet generell in mehreren, parallel arbeitenden Threads statt. Dabei verteilen wir die Datensätze per Modulo Funktion auf der Artikelnummer, um die korrekte Reihenfolge der Bearbeitung innerhalb eines einzelnen Artikels zu sichern. Als Framework bot sich hier Spring Integration an.

Einige Nachteile dieses Entwurfs wie Fehleranfälligkeit und Code-Komplexität sollen nicht Teil dieses Textes sein, wohl aber die bislang erreichte Performance:

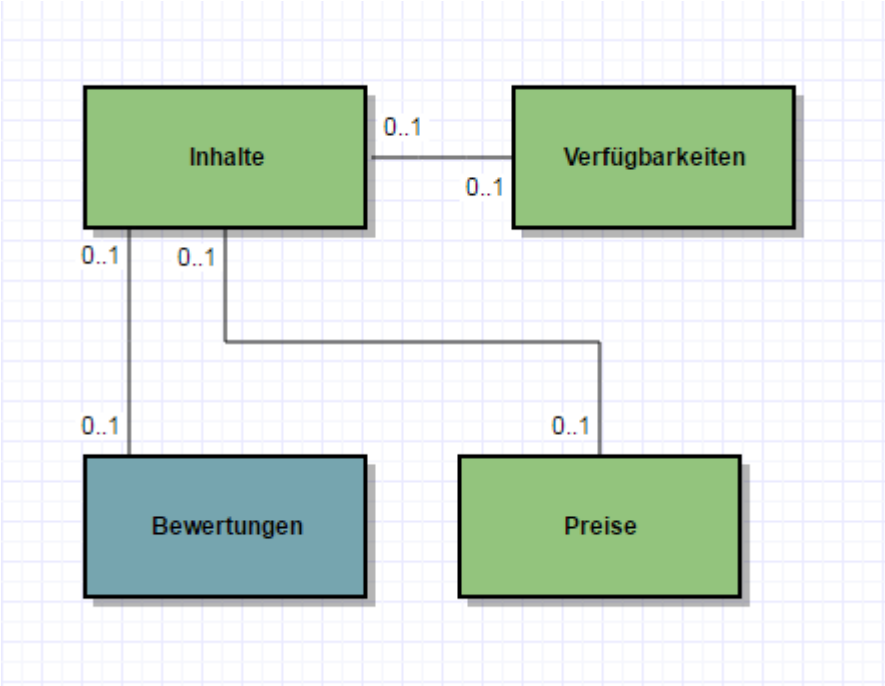
Die PIM Datensätze werden relativ konstant an die Anwendung geliefert und überschreiten im Normalfall nicht die 40k/Stunde. Auf der WWS Seite ist die Anlieferung eher stoßweise, die dann natürlich möglichst schnell übernommen werden müssen. Hier kommen dann auch mal 200k auf einmal.

Neuer Ansatz – PoC

Im ersten Teil des Artikels wurde schon einiges über den Hintergrund und die Designentscheidung bezüglich der Methodik beim Abspeichern berichtet. Hier eine kurze Zusammenfassung:

- Anstatt eine maximale/saubere Normalisierung der Daten anzustreben, wollen wir auf Speicherung von „Dokumenten“ setzen. Gründe dafür sind u.a. die gesteigerte Flexibilität bei Erweiterungen und weniger Performance/Deadlock Themen auf DB Ebene durch radikale Reduktion der Komplexität des Schemas.
- Wir führen ein Dokument pro Datenquelle (in Produktion sind es inzwischen sechs verschiedene Datenquellen) ein, um Konflikte zwischen

den Quellen beim Schreiben zu vermeiden und nicht unnötig viele Daten ändern zu müssen, wenn eine Aktualisierung ansteht.



Hier sei betont, dass in diesem Schaubild Dokumente und nicht etwa Objekte oder Klassen eingezeichnet sind. Das hat Auswirkung auf die Kardinalitäten und die Benennung, z.B. handelt es sich bei „Preise“ um null bis **ein** Dokument, in dem ein oder **mehrere** Preise abgelegt sind.

Die neue Anwendung basiert also auf wenigen Dokumenten statt vieler Klassen. Die DB Statements sind mit Spring-JDBC implementiert.

Auswahl des DBMS / Vergleich der Alternativen

Wir haben im Folgenden das bestehende System mit der alternativen Implementierung auf unterschiedlichen DBMS verglichen.

„Dokumente“ && „JSON“ == NoSQL

Der Umgang mit Dokumenten drängt zur Beschäftigung mit dokumentenbasierten Datenbanken aus dem Topf der „NoSQL“-Lösungen.

Bei dem Schlagwort „NoSQL“ und „Dokumente“ kommt an einer Betrachtung von MongoDB fast nicht vorbei. Abgesehen davon, dass MongoDB von allen NoSQL/Dokumente-Lösungen unbestritten das beste Marketing hat, ist für uns der „Generalisten“-Ansatz interessant („Allzweckdatenbank“).

Wir können und wollen uns keinen Zoo von Technologien halten, in dem jede Speziallösung ein paar Prozente besser ist als ein eventueller Generalist – unter der Voraussetzung, dass sie denn optimal konfiguriert/betrieben wird, wozu bei der Anzahl von Speziallösungen keine vernünftig große Gruppe von Menschen in der Lage ist.

Worauf ich hinaus will: wenn wir einen Spezialisten einführen, dann muss er sehr deutlichen Mehrwert mitbringen, der den Aufwand der Einführung und des Betriebs rechtfertigt. Er sollte kein „one-trick pony“ und auch in zwei Jahren noch von Mitgliedern des Produktteams betreibbar sein. In NoSQL einzusteigen mit einer Lösung, die sich selbst als Generalist vermarktet und daher verspricht, auch die Lösung für zukünftige Probleme zu sein, ist da sehr sexy.

Auf der MongoDB Seite findet man sich auch bestätigt, was die obigen Überlegungen angeht, wird hier doch auch eine „EAV“-Diskussion für Artikeldaten geführt.

Stark, damit haben wir unseren ersten Kandidaten!

„Dokumente“ && „JSON“ != NoSQL

Aber muss es denn was komplett neues sein? Seien wir ehrlich, die Aussicht auf was ganz neues – den „heißen Scheiß“ – macht Spaß. Wir alle wollen ausprobieren, wovon das Internet und die Regenbogenpresse der Entwickler spricht.

Wir müssen die Systeme allerdings auch betreiben – länger als sein Jahr, in Produktion, mit dramatischen Umsatzausfällen bei Ausfall verbunden. Der Kater droht also immer – wir sind da stark sensibilisiert.

Eine kleine Quellenforschung bei den zwei freien DBMS, die in unserem Haus im Einsatz sind, zeigt, dass man auch dort auf JSON/Dokumente aufgesprungen ist. Also haben wir sie in den Vergleich aufgenommen.

Postgres && JSON

Postgres unterstützt den JSON-Datentyp seit der Version 9.2 und erweitert zur Verfügung stehende Funktionen seitdem. Mit Version 9.4 ist JSONB hinzugekommen, der für einige Anwendungsfälle effizienter arbeitet, Indizierung ermöglicht und für unseren Vergleich auch verwendet wurde. Die JSON(B)-

Funktionen sowie die gesamte Notation zum Traversieren der Dokumente sind Postgres-spezifisch. (Eine schnelle Einführung gibt es hier.)

Skeptiker könnten jetzt sagen, dass sei nur ein verkleideter „TEXT“-Datentyp mit ein paar Funktionen drum herum, womit sie natürlich größtenteils recht haben. Im hier beschriebenen Anwendungsfall reicht aber genau dieser Grad der Integration aus, um unsere Anforderungen umzusetzen(!). Solange wir jeweils komplette Dokumente in der DB Tabelle austauschen (was wir aufgrund unseres oben beschrieben Schnitts der Dokumente tun) und nicht Einzelfelder innerhalb der Dokumente editieren, sind wir mit diesem DBMS im Geschäft.

Die Indizierung auf Einzelfelder innerhalb der Dokumente ist möglich.

MySQL && JSON

Siehe da, MySQL unterstützt ebenfalls einen JSON Datentyp ab Version 5.7. Hier gelten gleiche Aussagen wie bei der Postgres-Variante. Wir haben einen neuen Datentyp, der für die Definition einer DB Spalte verwendet werden kann. Gegenüber der Verwendung eines einfachen String-Feldes verspricht die Datenbank eine Validierung auf syntaktische Korrektheit und ein optimiertes Speicherformat für schnellen Lookup von Einzelfeldern.

Die Indizierung auf Einzelfelder innerhalb eines Dokumentes ist nicht möglich – dafür gibt es aber einen kleinen Kniff, der für unseren Anwendungsfall ausreicht.

Performance Vergleich: MongoDB, Postgres, MySQL

Vier verschiedene Implementierungen haben wir gegeneinander antreten lassen.

Das haben auch schon andere vor uns gemacht. Die gesetzten Bedingungen, der jeweilige Anwendungsfall, die Intention der Autoren, die Validität der Ergebnisse bei älteren Versionsständen u.v.m. lässt den Skeptiker des Öfteren zweifeln, zumal die Ergebnisse sich zum Teil widersprechen. (Beispiel für einen eventuellen Interessenskonflikt)

Für uns war es wichtig, den Vergleich innerhalb unserer Umgebung mit unseren Versionsständen und unserem Hintergrundwissen durchzuführen – zumal der Aufwand sich aufgrund der freien Verfügbarkeit der DBMS und der vorhandenen

umfangreichen Dokumentation in Grenzen hält. Niemand sollte allerdings die gemessenen Werte hernehmen und anfangen, einstellige prozentuale Abweichungen zwischen den Alternativen auszurechnen. Hier geht es darum, ein Gefühl für die richtige Entscheidung zu entwickeln – von gewichteten Bewertungsmatrizen sehe ich daher ab.

In der **ersten Ecke** steht der ursprüngliche Artikelimporter: generiertes DB-Schema, normalisiert, JPA als Persistenz-Framework, Abfragen gegen MySQL Datenbank in Version 5.6. Aus Bequemlichkeit steht dieser Anwendung ein größerer Server zur Verfügung als den Alternativen – aber nichts besonders Bemerkenswertes, siehe Tabelle unten.

In der **zweiten Ecke** steht die neue Version des Artikelimporters, bei uns intern auch als „Nitro-Edition" bezeichnet (um mal Erwartungsmanagement zu betreiben…), basierend auf zwei Quellen für Dokumente (WWS und PIM), Spring-JDBC, **Postgres in Version 9.5**.

In der **dritten Ecke** läuft die Nitro-Edition gegen **MySQL 5.7** – mit entsprechenden Syntaxanpassungen.

Ecke Nummer vier, bereit sie alle in den Schatten zu stellen, ist eine Nitro-Edition auf Basis **MongoDB in Version 3.3**. Hier musste natürlich mehr angepasst werden als zwischen den beiden gestandenen RDBMS. Das nicht vorhandene Transaktionshandling muss im Code berücksichtigt werden – total hemdsärmelig darf man hier nicht vorgehen.

	Artikelimporter (MySQL 5.6)	**Nitro Edition (PostgreSQL 9.5)**	**Nitro Edition (MySQL 5.7 (uc))**	**Nitro Edition (MySQL 5.7 (c))**	**Nitro Edition (MongoDB 3.3)**
Anzahl DB Tabellen	30+	2	2	2	1 (Collection)
Dauer WWS Meldung	486	3	7	3	1
Dauer 7 PIM Meldungen	127	26	41	28	23
Lines of Code	Ca. 16k	Ca. 1600	–	–	–
Hardware	8 Kerne, 8 GB RAM	4 Kerne, 4 GB RAM	4 Kerne, 4 GB RAM	4 Kerne, 4 GB RAM	4 Kerne, 4 GB RAM

Der Vergleich MySQL zu Postgres war zunächst verblüffend. Auffällig war vor allem der benötigte Platz auf dem Laufwerk für die gespeicherten Daten. Ein Studium der MySQL Seiten offenbarte, dass die Tabellen tunlichst mit dem

Parameter „row_format=compressed“ erstellt werden sollten und als Voraussetzung dafür der Server mit dem „innodb_file_format“ „Barracuda“ (statt „Antelope“) laufen muss. Das Resultat war eine Reduktion des benötigten Speicherplatzes um ca. 50% (und damit auf Postgres Niveau) und ebenfalls fast eine Halbierung der benötigten Ausführungszeiten (!). Guckst du hier für die offizielle Dokumentation. In der oben abgebildeten Tabelle haben wir zwei MySQL Zeiten aufgenommen – mit (c) und ohne (uc) Komprimierung.

Entscheidung für Technologie

Nach der kleineren Nachbesserung auf MySQL Seite kann man auf Basis der Tabelle folgende objektive Schlüsse ziehen:

- Der neue Entwurf der Anwendung ist dem alten weit überlegen in allen Faktoren. Gute Entscheidung. Gut gelernt.
- Postgres und MySQL tun sich nicht viel im Bereich Ressourcen und Performance. Auf keinen Fall sind Größenordnungen dazwischen.
- MongoDB hat die höchste Geschwindigkeit erzielt.

Schaut man also nur auf die reine Geschwindigkeit, ist MongoDB den beiden RDBMS davongezogen – je nach Betrachtungsweise um ein paar Prozent bis Faktoren. In Relation zur bisherigen Lösung aber sind alle Umsetzungsvarianten schnell – schneller als sie sein müssten, denn PIM und WWS können die Nachrichten gar nicht so schnell generieren wie sie hier abgearbeitet werden (das wäre auch peinlich).

Die größten angepriesenen Stärken von MongoDB sind für uns allerdings nicht relevant:

- Skalierung über Rechenzentren hinweg bis der Geldbeutel aufgibt.
- Ausfallsicherheit bis der Geldbeutel aufgibt.
- Flexibelste Indizierung für „mächtige Abfragen“.
- Textsuche

In unserem konkreten Szenario haben wir von der überlegenen Leistung von MongoDB also nichts, dafür aber ein paar Nachteile.

- Wir kennen uns mit der Benutzung nicht aus (z.B. fehlende Transaktionen).

- Wir kennen uns mit dem Betrieb nicht aus (Früherkennung von Zipperlein, Sizing).
- Wir haben ungeklärte Support-Szenarien (Standzeit, Wer kennt sich aus?).
- Kosten für die Enterprise Lizenz – speziell wenn man vom Einzelserver abrückt.

Ich habe anfangs geschrieben, die Lösung muss „sehr deutlichen Mehrwert mitbringen, der den Aufwand der Einführung und des Betriebs rechtfertigt“. Diesen Mehrwert sehen wir hier nicht.

Die beiden anderen DBMS tun sich in den Messwerten nicht viel. Hier kommen eher pragmatische Argumente:

1. Die Postgres Syntax für Abfragen auf Teildokumenten ist aus meiner Sicht einfacher zu verstehen. Unsere Anwendung lädt allerdings nur ganze Dokumente, so dass diese Funktionalität nicht zum Tragen kommt.
2. Die Indizierung auf Feldern innerhalb von Dokumenten ist in Postgres vorgesehen. In der MySQL Variante muss man über virtuelle Spalten gehen, was zwar funktioniert, aber Applaus gibt es dafür von mir nicht.
3. Für Abwärtskompatibilität benutzen wir Datenbank Views, die dem abfragenden ein normalisiertes Schema vorgaukeln. Dazu müssen wir Arrays innerhalb von Dokumenten in DB Zeilen ausmultiplizieren. Beispielsweise müssen alle Preise aus dem Dokument „Preis“ in jeweils einer Zeile angezeigt werden. Das ist unter Postgres eine schicke Funktion („jsonb_array_elements“), aber unter MySQL muss man ein join mit einer Hilfstabelle machen – geht, aber ebenfalls kein Applaus von mir. Hier verspricht MySQL 8 ein wenig Abhilfe, ich würde bis dahin aber nicht die Luft anhalten wollen.
4. Wir haben hier bei Thalia eine große MySQL Installation zur Auslieferung von Artikeldaten in Betrieb. Das beinhaltet sowohl die Ausfallsicherheit, die Skalierung/Lastverteilung als auch die korrekte Netzwerkzone. Unsere Postgres-Installation ist zwar vom Volumen her größer, steht aber am falschen Ende der Firma

Am Ende der Entscheidungsfindung war der letzte, hoch pragmatische Punkt ausschlaggebend. Hier haben wir auf „time-to-market“ geachtet in dem Wissen, dass wir bei so einfachen Queries die Systeme Postgres/MySQL ohne

nennenswerten Aufwand jederzeit wechseln können.

Also – Überraschung! – wir haben uns für **MySQL** entschieden!

Wir bauen unseren Webshop um

Eine Frage, die wir uns schon lange gestellt haben, war: was wollen wir mit unserem Webshop machen? Dieser ist mittlerweile etwas in die Jahre gekommen. Wir sind zum Entschluss gekommen, er **passt nicht mehr,** und wir wollen ihn ablösen.

Fragen wie

- Wie leicht können Änderungen durchgeführt werden?
- Wie gut passt er zu unserem fachlichen Zielbild?
- Wie gut passt er zu unserer Organisation (Conway's Law)?

sind für uns wichtige Kriterien und haben zu dieser Entscheidung geführt.

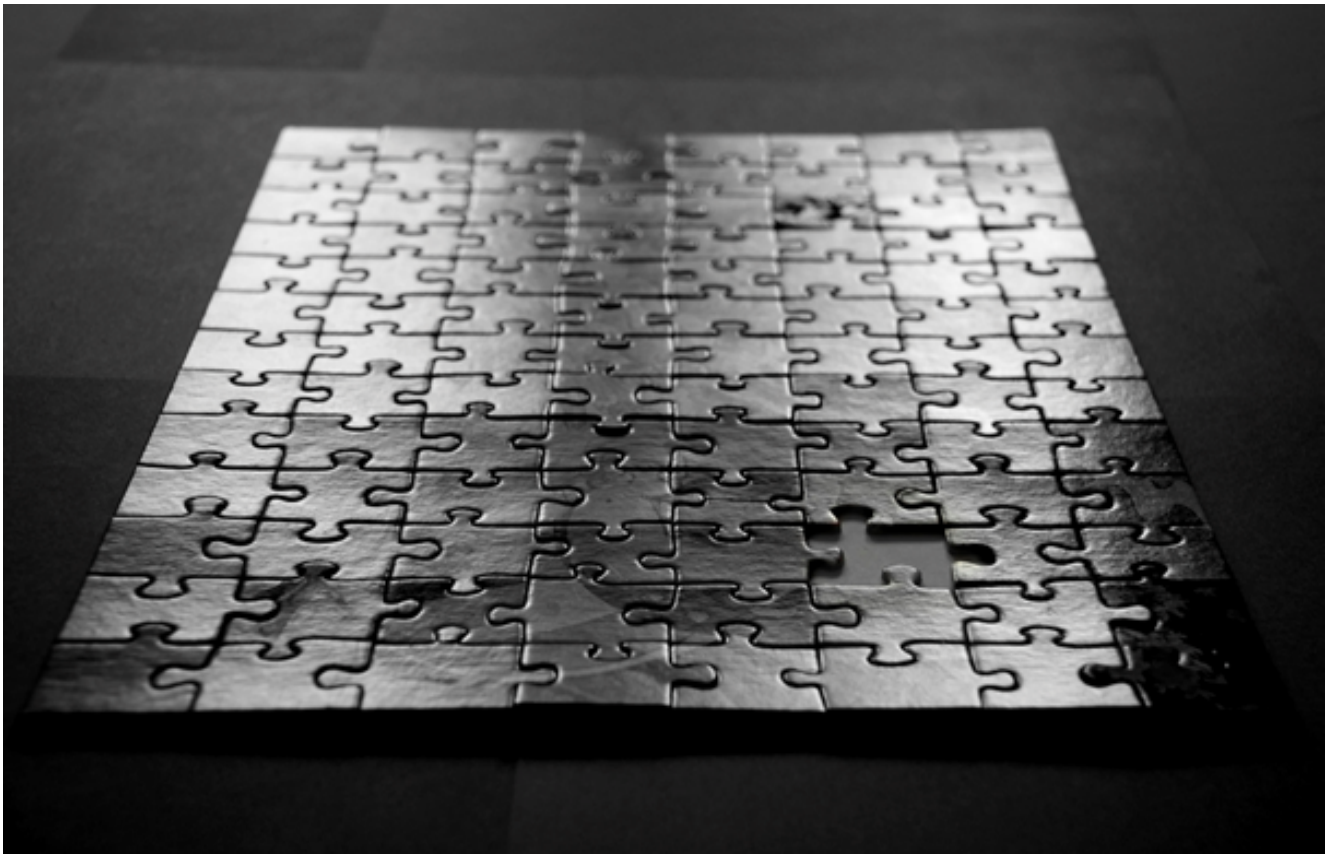
Nur durch was wollen wir unseren Webshop ersetzen?

Wenn wir etwas ablösen wollen, müssen wir auch sagen, wodurch wir das ablösen wollen. Die erste Frage ist: wie groß ist eigentlich das „Greenfield“?

Neben den gewünschten Funktionalitäten gibt es noch eine Reihe weiterer Kriterien zu beachten:

- Wie gut integriert es sich in unsere bestehende IT Landschaft?
  Wir wollen nicht unsere komplette IT Landschaft (ERP, Suche, CRM, Analytics,…) umstellen. Nach dem „best of breed“-Ansatz sind da zum Teil Kauflösungen unterschiedlicher Hersteller zum Teil Eigenentwicklungen dabei.
- Wie integriert es sich mit der bestehenden Organisation & Prozesse?
  Mit z.B. .NET kennt sich hier kaum jemand aus ;-)), wir haben cross-funktionale Produktteams, die das realisieren sollen.
- Wie weit passt es zu unserem geplanten Projektverlauf?
  Wir wollen iterativ vorgehen und parallel weitere Projekte realisieren.

Auswahl des Shopsystems



Tim Geers you make it complete CC BY 2.0

Es gibt ziemlich viele eCommerce Systeme – Open Source, On Premise, SaaS Lösungen etc. Trotzdem haben wir uns gegen eine Standardlösung entschieden und realisieren hauptsächlich mit einer Individuallösung. Wir wollen Systeme haben die sich **uns** anpassen und nicht unsere Organisation, Prozesse, fachliche Features **an Systeme** anpassen. Ein Standardsystem zu nehmen und so **extrem** anzupassen, dass es unseren Wünschen entspricht, halten wir für keine gute Idee.

Zielarchitektur definieren

Welche Architektur ist für uns die richtige? Wir sind von der Organisation und Komplexität so groß, dass wir uns in mehrere Teams aufteilen müssen. Daher müssen wir in der Makroarchitektur einen **Domänenschnitt** finden, der den besten Trade Off zwischen möglichst kleinen Einheiten, möglichst wenigen Schnittstellen, möglichst hoher Kohäsion, möglichst hoher Änderbarkeit, möglichst stabilem Schnitt, möglichst geringer Kosten und noch vieles mehr bildet.

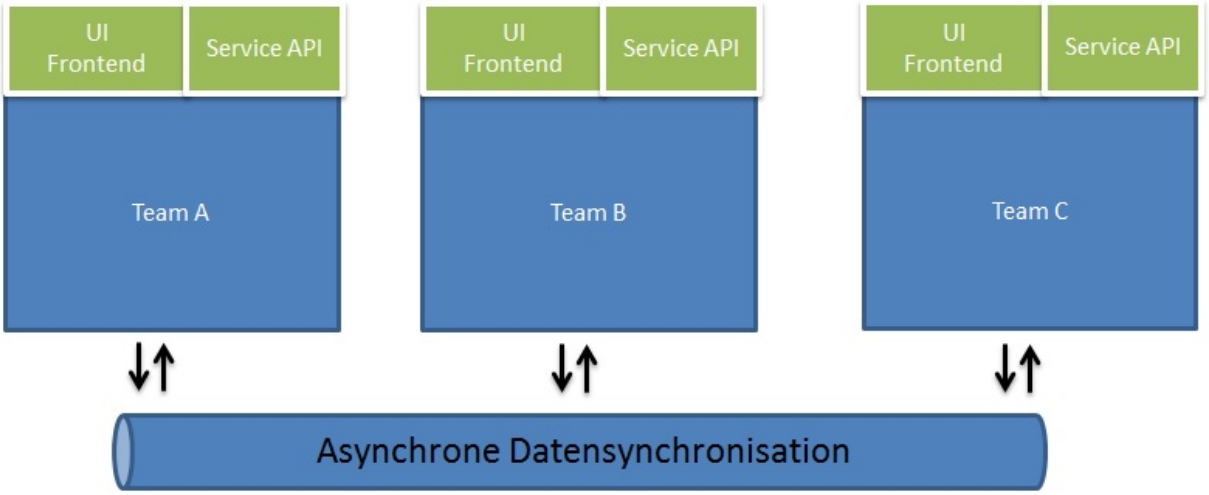
Domänenschnitt

Wir haben uns dafür entschieden, dass eine Kundenfunktionalität möglichst komplett von einem einzigen Team betreut wird. D.h., ein Team ist z.B. für die Artikelsuche zuständig – und das idealerweise von der Visualisierung zum Kunden über die eigentliche Suche bis zur Datenversorgung der Suchmaschine. So sind unsere Produktteams geschnitten und so sollen sie sich auch in unserer technischen Abbildung widerspiegeln.
Da wir nicht nur eine Webseite haben sondern auch Native Apps und spezielle eReader Anwendungen, sind unsere Frontend Technologien sehr vielfältig.

Daher haben wir uns zu einem **Kompromiss** entschieden. Für Webseiten geht die Teamverantwortung bis in die Visualisierung. Für Native Anwendungen greifen unsere Frontend Spezialisten auf Service APIs zu. Er ist also ein Funktionaler Schnitt für Webseiten und eine Schichtenarchitektur für Native Frontends.
Wir orientieren uns für die Webseiten an der SCS Architektur, allerdings haben wir viele Service APIs und stellen zum Teil beide APIs zur Verfügung – eine HTML

Webseite und eine native Suche, die dann im Hintergrund eine Rest-API verwendet. Wir haben uns dafür entschlossen, da Webseiten unser Hauptfokus sind und wir nicht immer einer Service API brauchen. Wie immer gibt es Vor- und Nachteile, aber diese Lösung fühlt sich gut an und hilft uns, unsere Omnichannel Plattform aufzubauen.

Innerhalb der Teamgrenzen ist es jedem Team freigestellt, wie sie ihre Domäne aufteilen. In der Praxis zeigt sich, dass diese in mehreren Microservices aufgeteilt wird – zum Teil auch hier ein Funktionaler Schnitt zum Teil nach Schichten. Wie das in der Praxis aussieht, kann man z.B. bei Team Kaufen sehen.



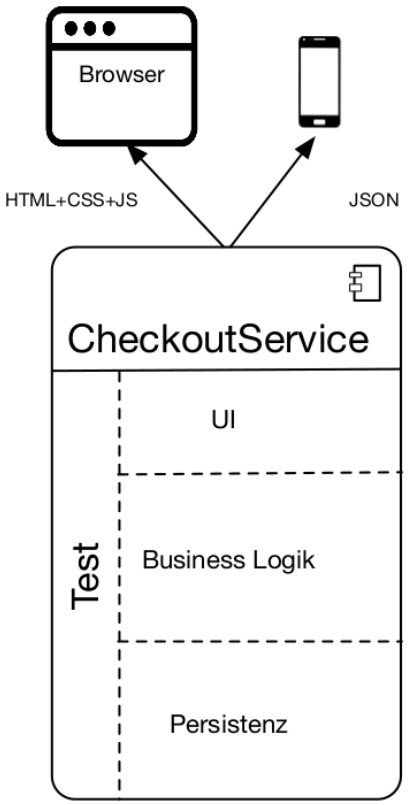
Kommunikation zwischen den SCS Systemen

Für die Kommunikation zwischen den SCS Systemen setzen wir bei synchroner Kommunikation auf REST Schnittstellen und bei asynchroner Kommunikation auf Events. Die Events werden über ein Messaging System verarbeitet und verteilt. Wir wollen möglichst asynchron kommunizieren. Wir klären gerade, wie wir allen Systemen diese Technologien beibringen. Zum Teil müssen wir noch Lösungen finden, da nicht alle bestehende Standardsysteme solche Schnittstellen anbieten.

Zusammenfassung

Wir wollen einen Webshop haben, welcher zu unserer neuen Organisation passt und sich leicht verändern lässt. Da unser jetziger Webshop dies nicht ausreichend leistet, wollen wir in ablösen.
Eine Standardlösung passt nicht, daher haben wir uns entschlossen, eine Individuallösung zu realisieren, die im Kern auf eine SCS Architektur aufsetzt. Für die nativen Frontends werden Service APIs verwendet.

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?

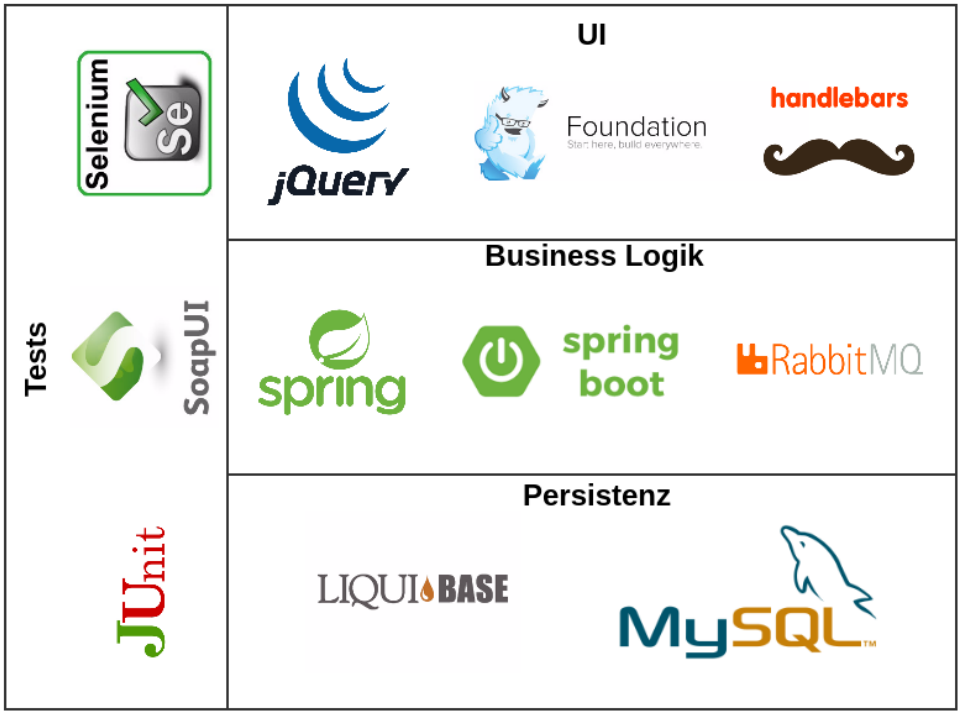
Da der weitere Fokus des Artikels auf der beispielhaften Implementierung des beschrieben Musters liegen soll, möchte ich an dieser Stelle gerne auf eine differenzierte Auseinandersetzung mit den Hintergründen verzichten. Wer sich vorab einen kurzen Überblick über das „Warum?“ verschaffen möchte, dem 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.

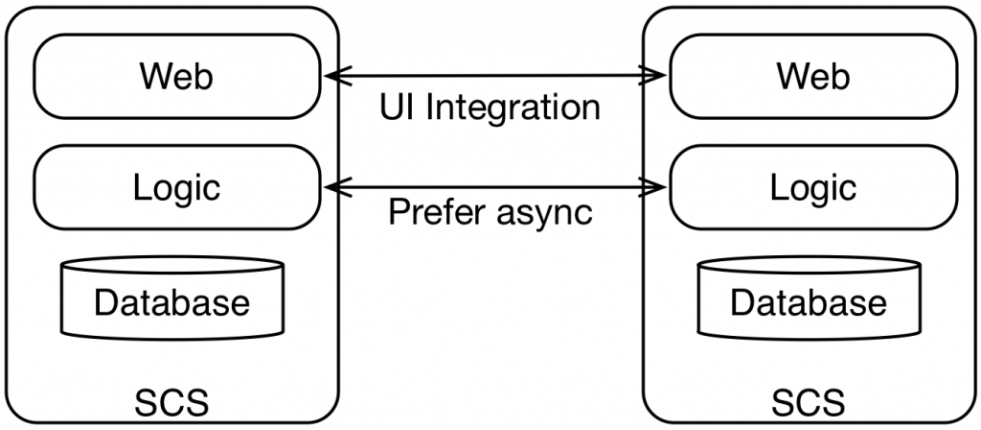
Wie integriert sich ein SCS in bestehende Systeme?

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)

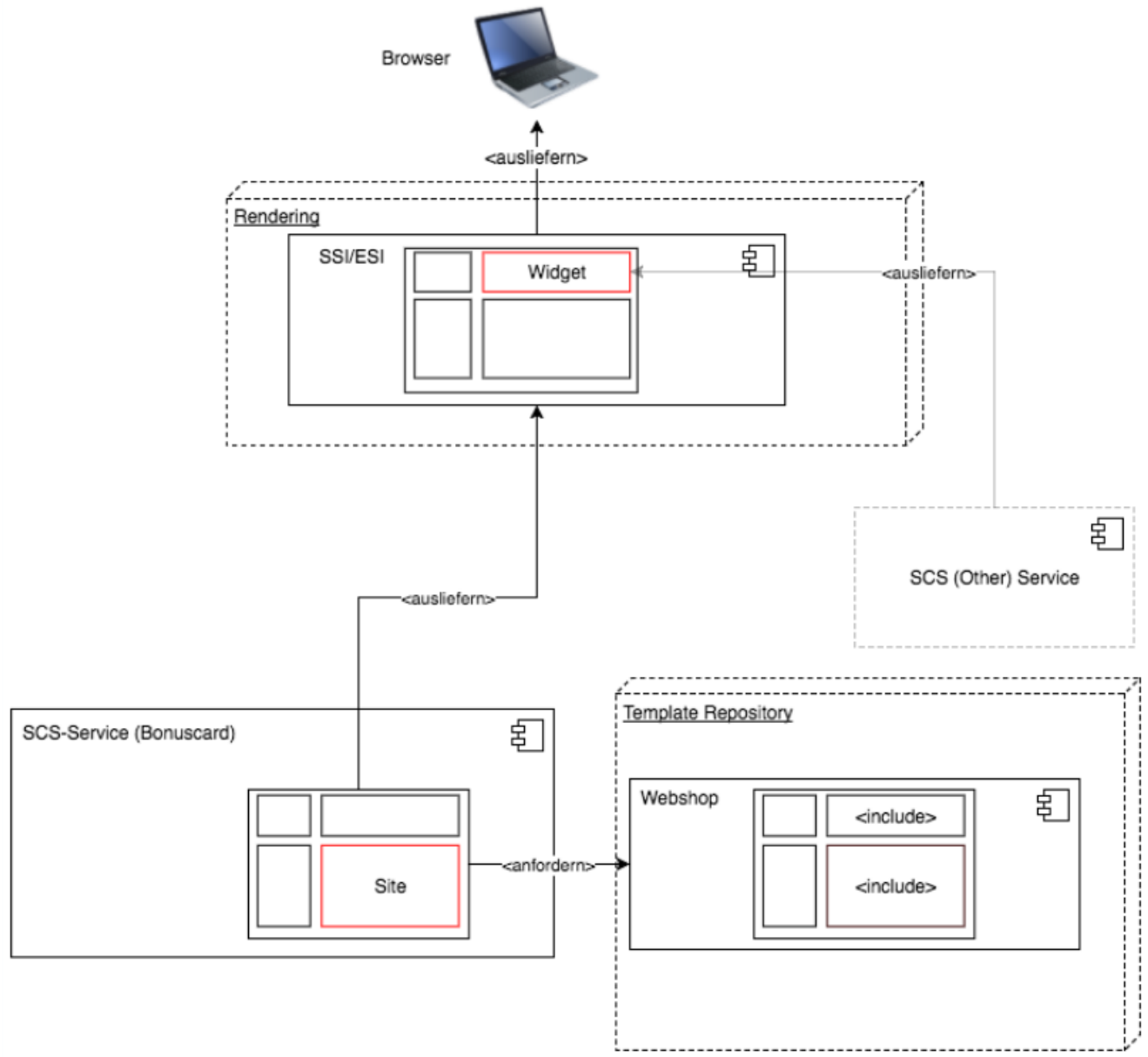
Bezogen auf Prozessintegration rückt die UI wieder in den Mittelpunkt. Da backendseitig vor allem asynchron und vorzugsweise innerhalb des eigenen SCS kommuniziert werden soll, findet die domänenübergreifende Prozessintegration

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

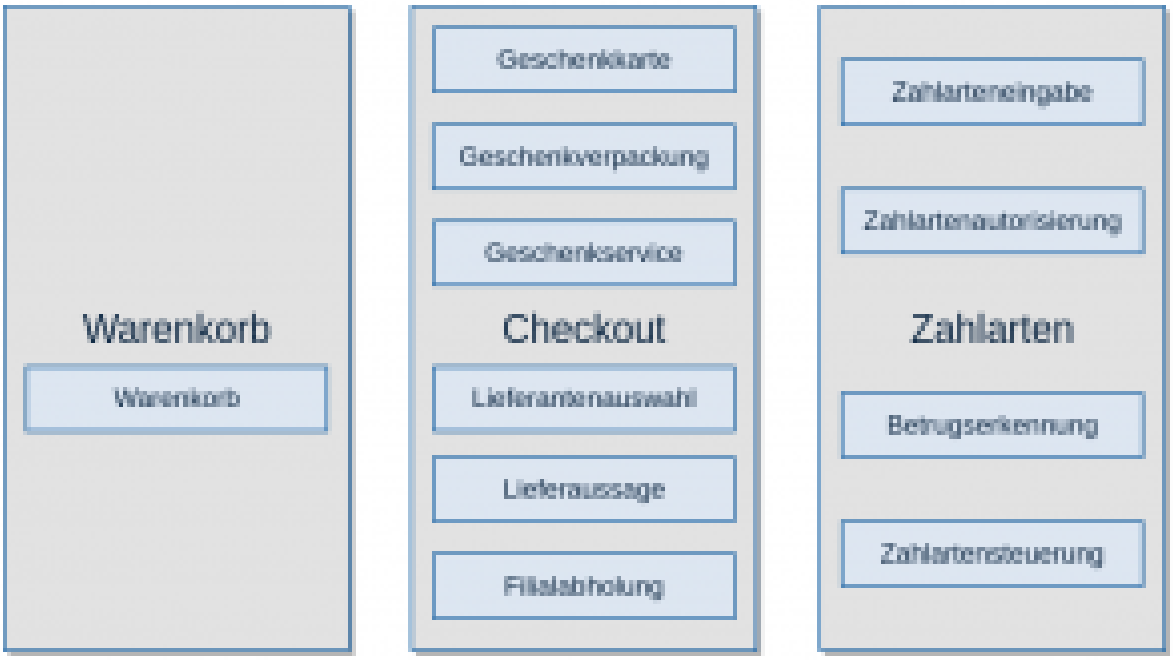
Team Kaufen auf dem Weg zur Omnichannel Plattform – Let’s get it started

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.

Artikeldaten in Datenbanken: Von einem relationalen Schema zu „Dokumenten“ (1/2)

Die Problemstellung

Die geeignete technische Ablage von Artikeldaten beschäftigt uns quasi beruflich. Für Leser, die keinen Bezug zu dieser Domäne haben, hier eine kurze Erläuterung.

Artikel haben (u.a.) Attribute. Die Art und Anzahl von Attributen pro Artikel ist je nach Art (Sortiment) des Artikels (Buch, Haus, Auto, Kaugummi) sehr unterschiedlich.

Artikel haben Attributwerte in diesen Attributen. Nicht jeder Artikel eines Sortiments hat für jedes Attribut einen Wert. Beispielsweise könnte eine Gewichtsangabe fehlen, weil der Hersteller diese nicht zur Verfügung gestellt hat.

Ein Artikel hat mäßig viele Attribute. Beispiel:

Abbildung 1 – die naive Version

Die Realität ist etwas anstrengender als diese Abbildung. Es kann jederzeit neue Attribute in bestehenden Sortimenten oder ganz neue Sortimente geben. Wollte man alle Sortimente in dieselbe DB Tabelle stecken, käme etwas langes und sehr breites heraus – wobei nur ein Bruchteil der Spalten pro Artikel wirklich befüllt wäre. Der Rest bliebe leer.

Wollte man diesen Aspekt komplett flexibilisieren (z.B. in einem universell einsetzbaren PIM System), könnte man die Daten nach dem EAV (entity attribute value) Modell ordnen.

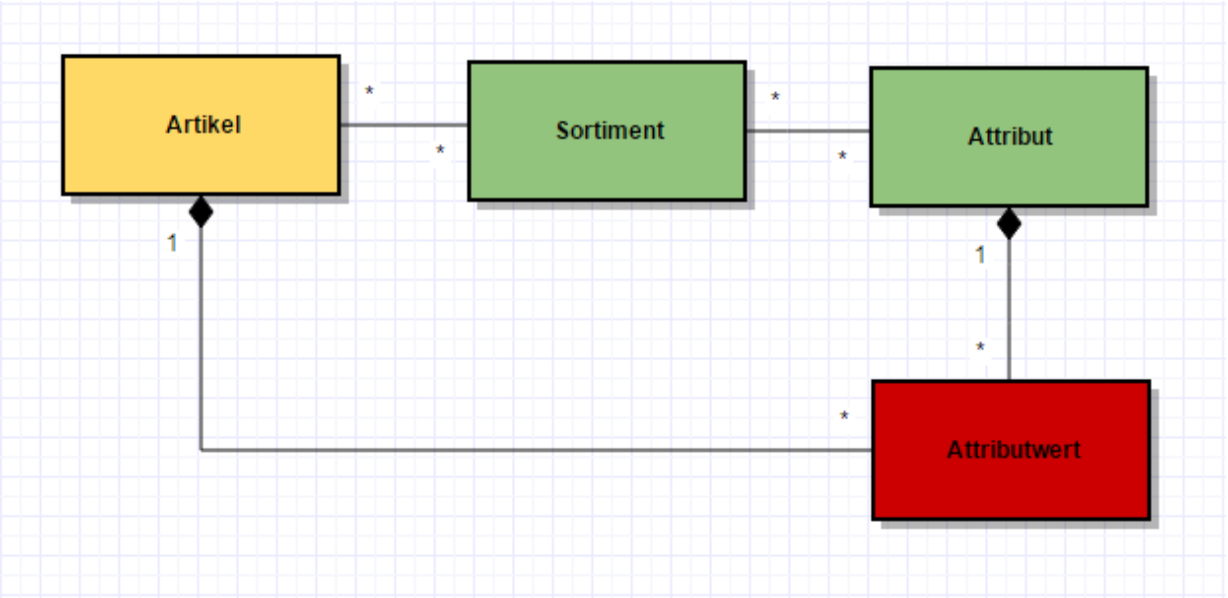


Abbildung 2 – entity, attribute, value (EAV)

Zusätzlich zu einfachen Attributen gibt es noch komplexere Daten, die einen Artikel beschreiben: Preise, Bilder, Autoren, Hersteller, Track-Listen, …

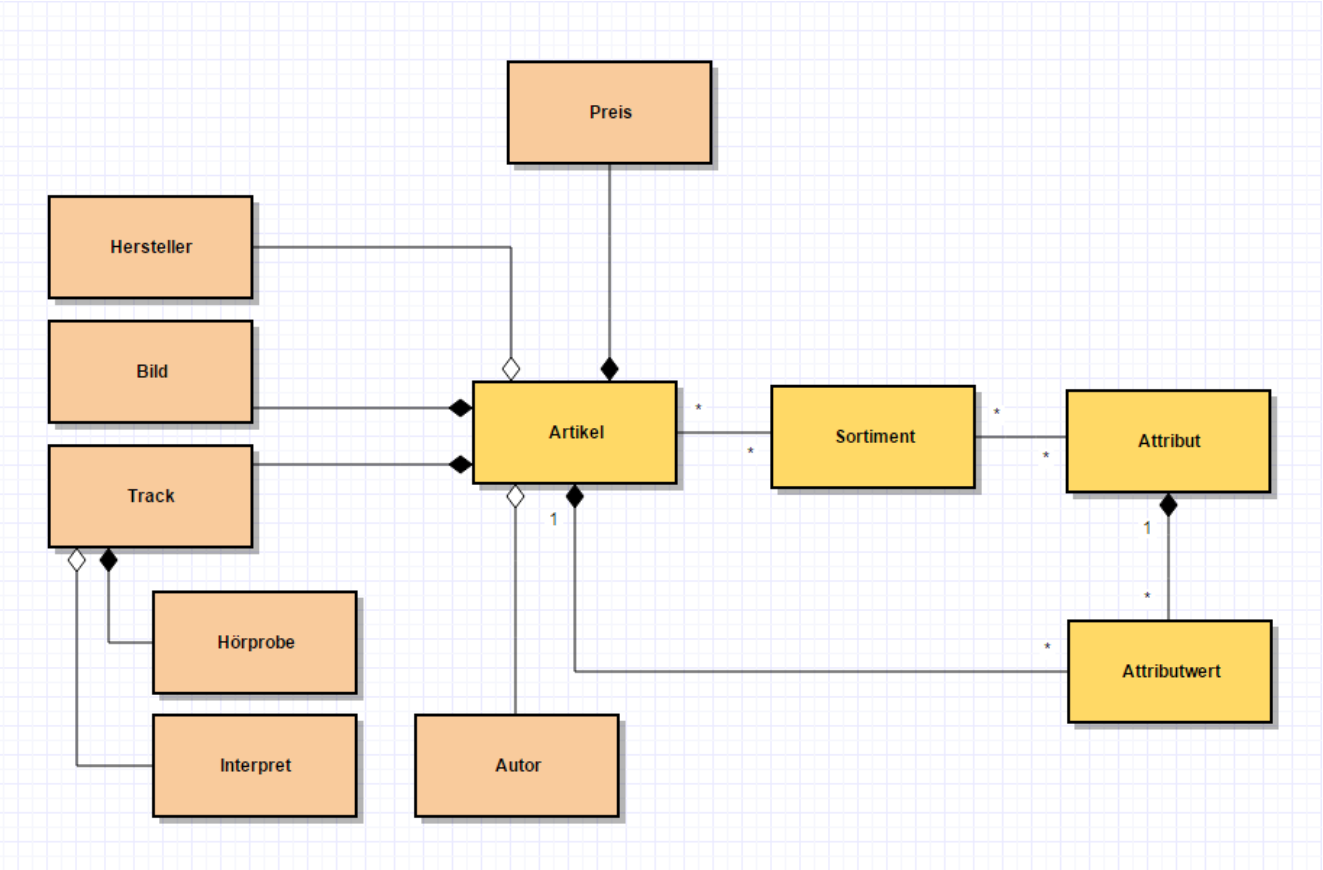


Abbildung 3 – EAV mit totaler Flexibilität

Klasse, total flexibel. Aber die Realität ist hier leider noch nicht am Ende.

Wir haben viele Artikel. Bei hypothetisch angenommenen 50 Millionen Artikel aus 10 Sortimenten mit jeweils 30 Attributen von denen 20 mit Werten belegt sind, ergibt das schon mal **eine Milliarde Zeilen** in der Attributwert-Tabelle. So eine Tabelle (wenige Spalten und viele Zeilen) wird auch „long and skinny“ genannt. Geht – ist aber in der Größe garantiert spaßbefreit.

Auf diese Tabellen müssen hochperformante CRUD-Aktionen möglich sein. Das ist heutzutage gleichzusetzen mit massiv parallelem Schreiben und Lesen. Ersteres ist hinsichtlich von DB Sperren problematisch, zweiteres macht aufgrund verschachtelter „joins“ auch keinen Spaß.

Die Einträge in der „Attributwert“-Tabelle sind relativ generisch, d.h. die Tabelle muss Attributwerte unterschiedlicher Datentypen halten. Das läuft auf String/textbasierte Spalten hinaus, eventuell sogar um Metainformationen bereichert. Die Validität der geschriebenen Daten (z.B. Typsicherheit) kann das DBMS nicht sicherstellen, d.h. dies (und die Konvertierung) muss die Applikation machen. Dabei ist das Abändern von Daten außerhalb der Applikation („mal eben auf der Datenbank“) ein gefährlicher Stunt.

Eine weitere, relativ normale Anforderung ist die Möglichkeit der Auswertung und Analyse von Daten. Beispielsweise soll die Qualität und Quantität der Befüllung des Attributs „Laufzeit“ im Sortiment „Bluray“ für die Artikel geprüft werden, deren Titel mit „A“ beginnen. Das Entwerfen so einer Abfrage ist nicht wild – eine effiziente und effektive Indizierung mit DB-Mitteln schon.

Und was nun?

Ein Lösungsweg

Wie häufig bei neuen Lösungen besteht der Trick darin, an der richtigen Stelle an den Anforderungen zu drehen.

Die Zugriffszeiten müssen sehr gut sein, da kommen wir im eCommerce nicht drum herum. Ebenso muss die Schreibgeschwindigkeit so hoch sein, dass sämtliche anfallenden Aktualisierungen souverän und zeitnah weggearbeitet werden.

Der Aufwand zur Erweiterung oder Kürzung der Attributmengen und Sortimente sollte nahe null gehen. Das ist einfach Tagesgeschäft und muss ohne großen

Eingriff der Softwareentwicklung und Koordinationsaufwände mit Abnehmern der Daten passieren.

Daher sind wir im Bereich der Artikeldaten (PIM) gerade bei dem Schlagwort „schemalos“ hellhörig geworden. Wir haben uns entschlossen, am Grad der Normalisierung des Datenmodells zu experimentieren. Was bedeutet das?

Ich verweise nochmal auf das Diagramm aus Abbildung 3. **Niemand** hat gefordert, dass der „Autor“ eines Artikels normalisiert werden **muss**. Die Anforderung lautet (korrekterweise): Eine Änderung an den Metadaten eines Autors soll auch an jedem Artikel des Autors sichtbar werden. Dem Anforderer selber ist es aber total egal, wie das in einer Persistenzschicht realisiert wird!

Noch einfacher wird es bei Bildern, Tracks und deren Hörprobe. Diese Daten „gehören“ zu einem Artikel (siehe Komposition), d.h. hier wird es keine Änderungen an den Metadaten mit Auswirkung auf viele Artikel geben.

Unser Lösungsweg ist also, alle Informationen, die nicht unbedingt normalisiert werden müssen, in ein „Dokument“ in der Datenbank zu schreiben. Das reduziert die Anzahl der benötigten Tabellen und Referenzen drastisch. Speziell die oben erwähnte Attributwert-Tabelle mit ihren 10^9 Einträgen verschwindet komplett.

Wie viele und welche Dokumente? Wie kommt man zum korrekten Schnitt?

In unserem konkreten Fall führen wir die Artikeldaten aus unterschiedlichen Quellsystemen zusammen. Jedes Quellsystem pflegt seine eigenen Update-Frequenzen und –Formen. Um sich beim parallelen Schreiben auf keinen Fall ins Gehege zu kommen und um im Falle einer Aktualisierung auch nicht ganz viel vermeidbares Zeug schreiben zu müssen, kriegt jede dieser Quellen sein eigenes Dokument. Ausgelesen wird dies – in SQL-Termini gesprochen – über ein „join“ über alle Dokumente eines Artikels, wobei Pflichtteile und optionale Teile unterschieden werden.

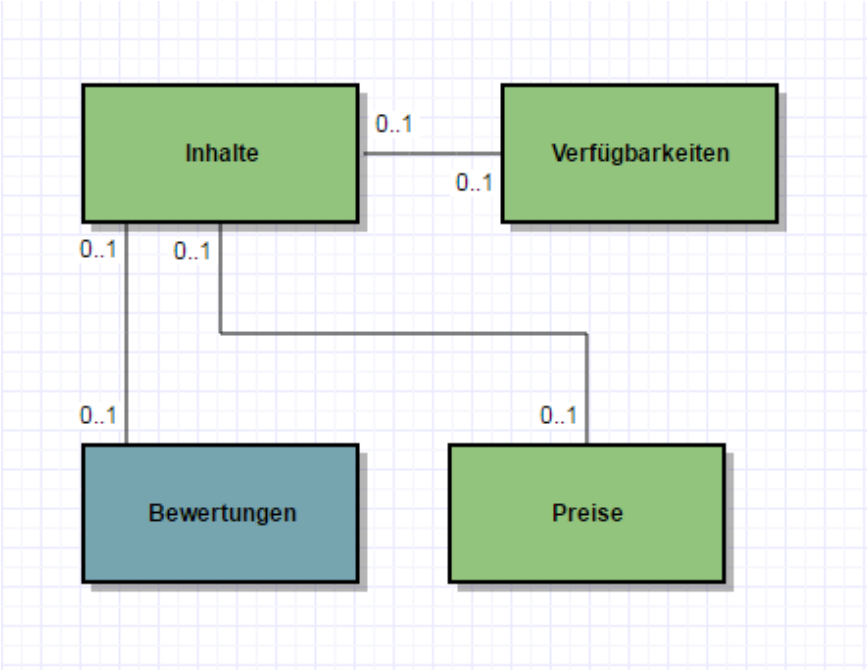


Abbildung 4 – Vereinfachung durch Dokumente

Ein „select“ eines ganzen Artikels wird somit trivial, und unvollständige Artikel gibt es in der Ergebnismenge nicht…

Dokumente! Und nun? NoSQL??

Im eCommerce und speziell im Bereich großer Datenmengen und/oder hoher Abfrageraten existiert eine Menge von Speziallösungen, um die verschiedensten Anforderungen adäquat erfüllen zu können. Ein einziges Werkzeug, um alle Probleme zu beherrschen, wird es – da leg ich mich mal fest – nicht geben.

Speziell in den Bereich der Datenbankmanagementsysteme (DBMS) geschaut, tummeln sich unter dem Begriff des „NoSQL“ sehr viele Spezialisten. Sind also die bisherigen DBMS alle ungeeignet für eCommerce und nur Dinosaurier? Natürlich nicht. Die Spezialisten sind sehr stark in ihrem Bereich, weil sie bewusst auf Features verzichten, die in anderen Anwendungsfällen überlebenswichtig wären (z.B. ACID Transaktionen).

Aber welcher „Spezialist“ passt auf unseren Anwendungsfall und den beschriebenen Lösungsweg?

Das haben wir ausprobiert – wir berichten im nächsten Teil dieses Artikels…