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

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

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

Projekt „OpenStack Explorer“

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

Worum ging es?

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

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

Ergebnis

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

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

Projekt „GitLab Explorers“

Worum ging es?

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

Ergebnis

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

Projekt „Our Vault“

Worum ging es?

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

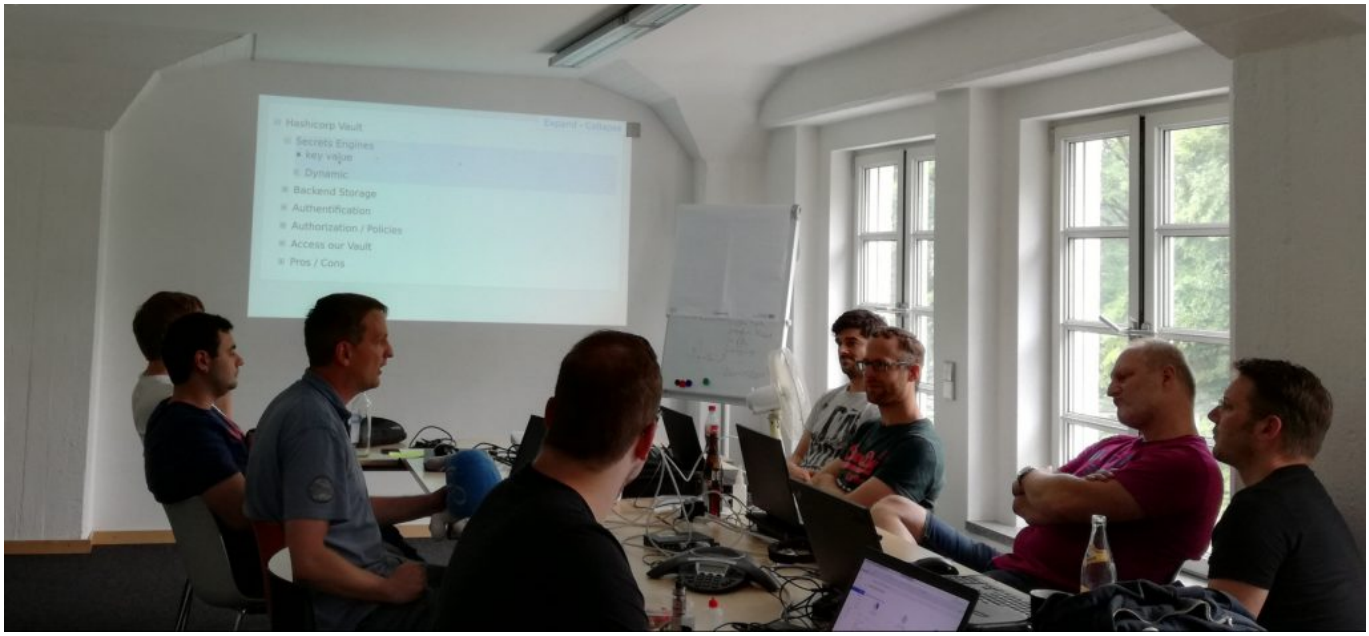
Ergebnisse

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

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

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

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



Fazit

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

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

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

Teil 2 – Technische Alternativen und deren Vergleich

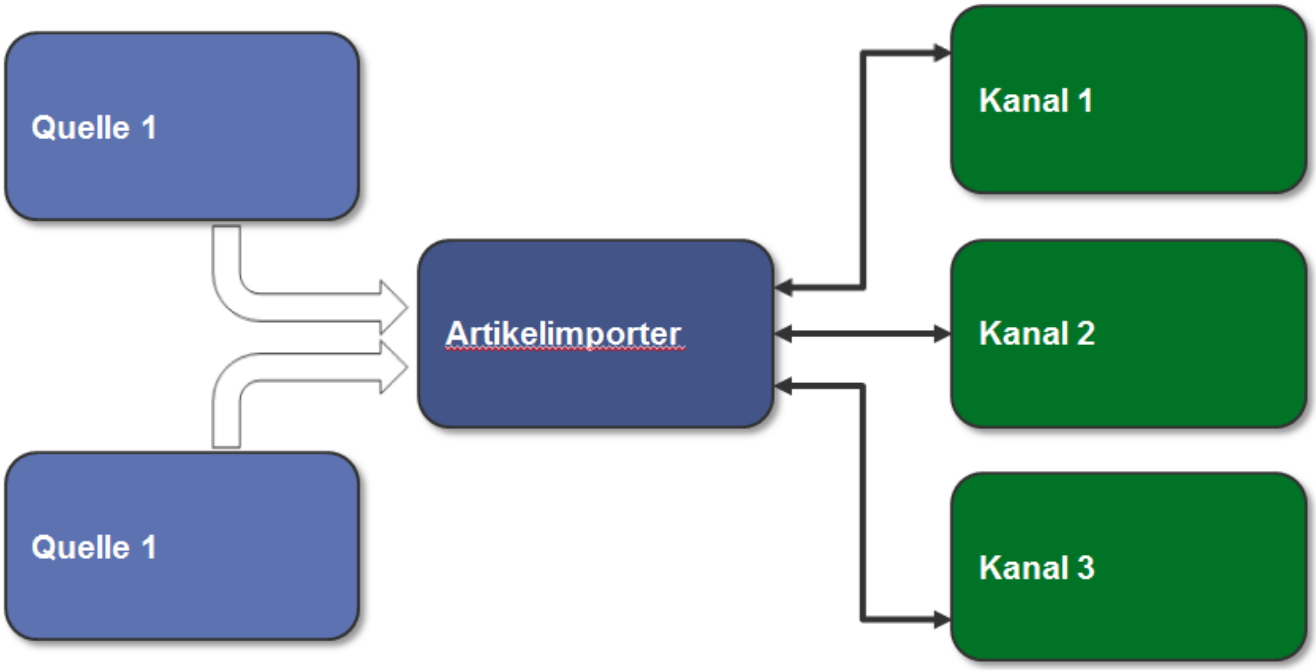
Die im ersten Teil beschriebene Problemstellung („… (wir) führen … die Artikeldaten aus unterschiedlichen Quellsystemen zusammen. Jedes Quellsystem pflegt seine eigenen Update-Frequenzen und –Formen.“) wurde in unserer Systemlandschaft vor einigen Jahren bereits in einer relativ leicht zu beschreibenden 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:

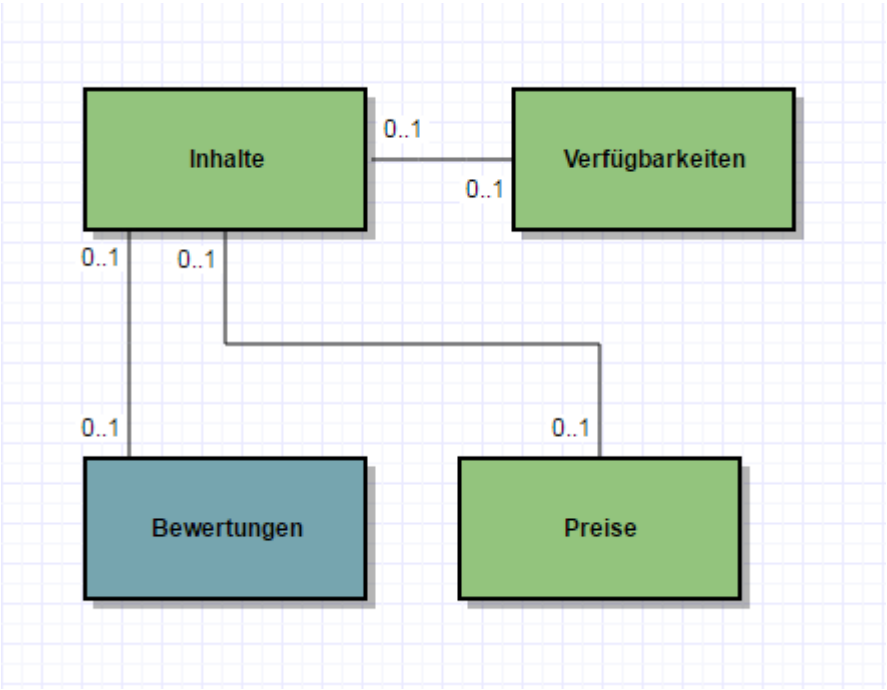
Die PIM Datensätze werden relativ konstant an die Anwendung geliefert und überschreiten im Normalfall nicht die 40k/Stunde. Auf der WWS Seite ist die Anlieferung eher stoßweise, die dann natürlich möglichst schnell übernommen werden müssen. Hier kommen dann auch mal 200k auf einmal.

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!

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.

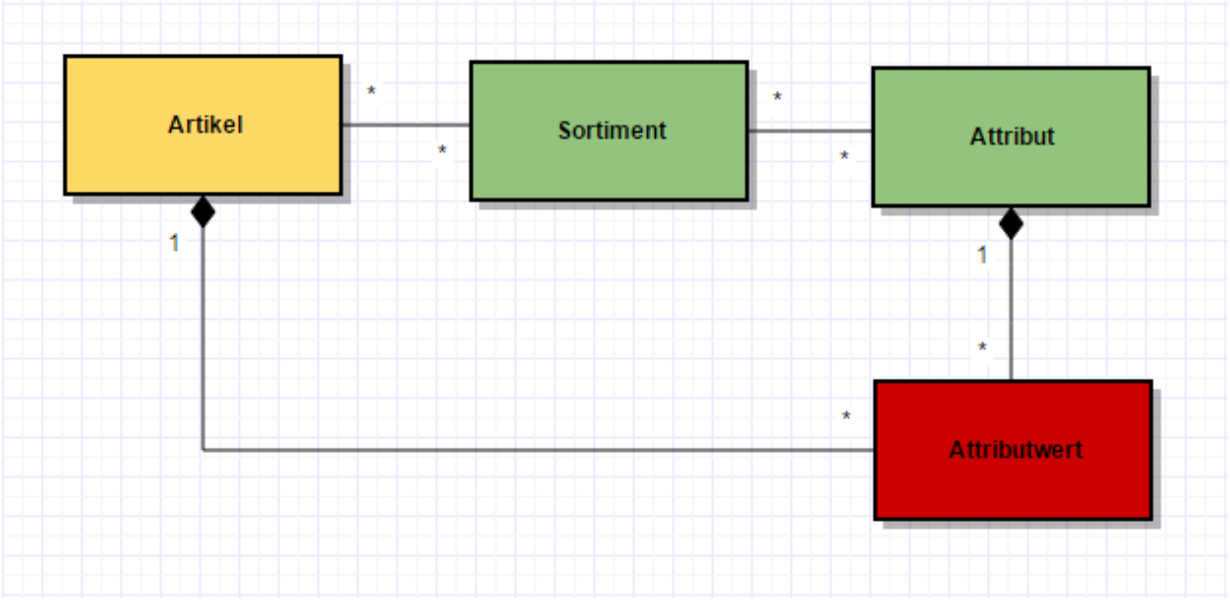


Abbildung 2 – entity, attribute, value (EAV)

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

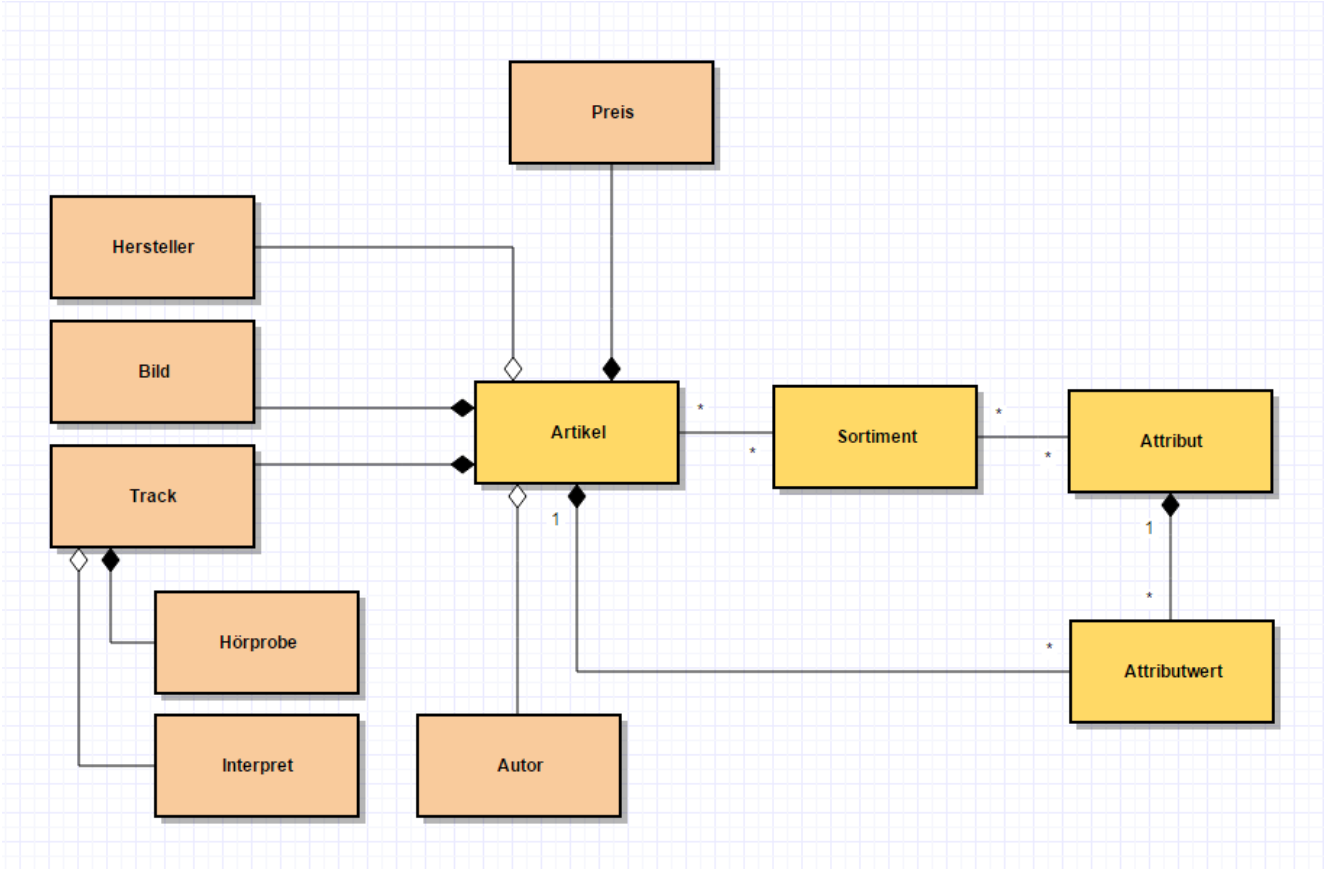


Abbildung 3 – EAV mit totaler Flexibilität

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.

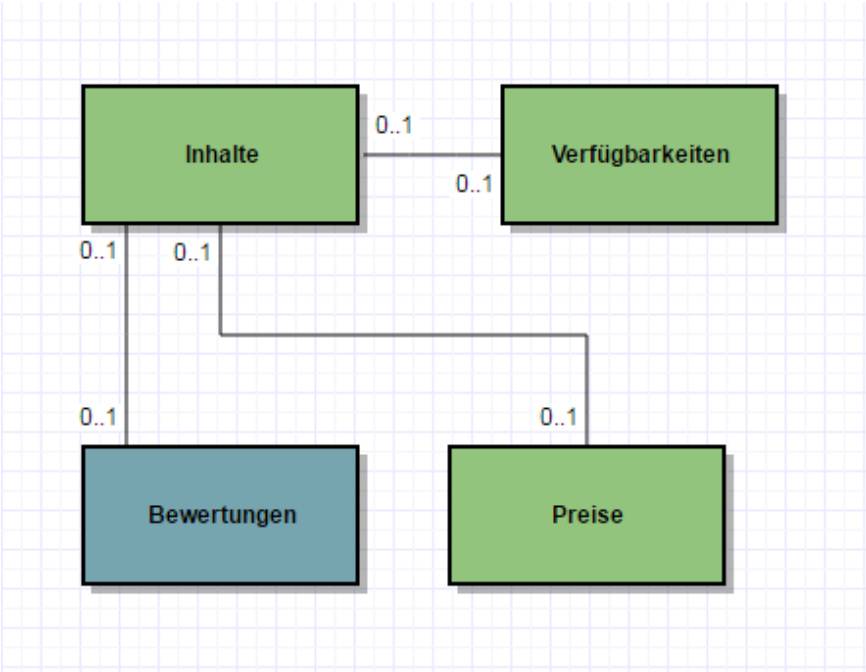


Abbildung 4 – Vereinfachung durch Dokumente

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).

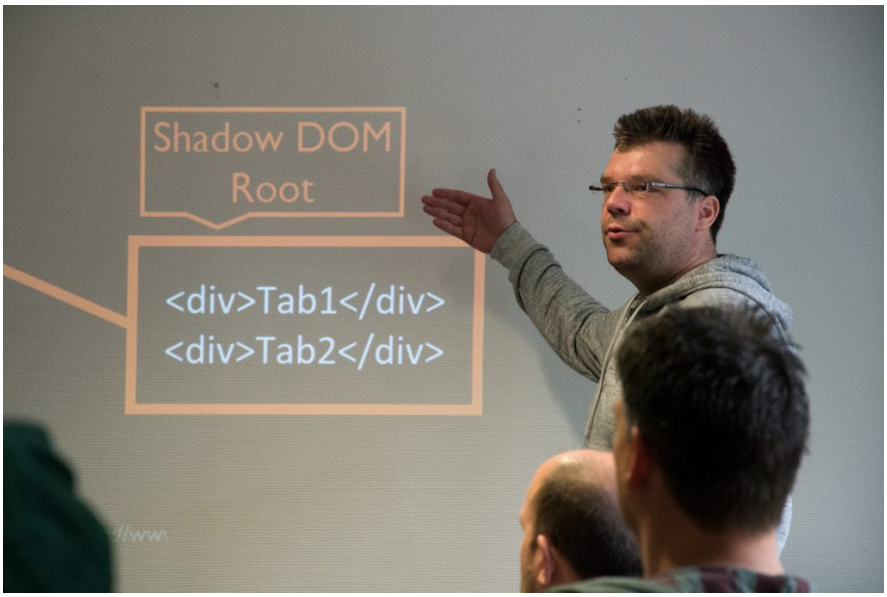
Aber welcher „Spezialist“ passt auf unseren Anwendungsfall und den beschriebenen Lösungsweg?

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

<#0001/> <hackathon@thalia/> in Münster

Am 12.05.2017 fand unser Hackathon am Standort Münster statt. Auch unsere Berliner Kollegen waren mit vertreten.

Wir haben uns morgens weit entfernt vom Tagesgeschäft getroffen. Für Verpflegung war (reichlich) gesorgt, Kabel waren gelegt, Technik war eingerichtet, die Agenda stand – wir konnten ohne Verzögerung sofort „offkicken“!



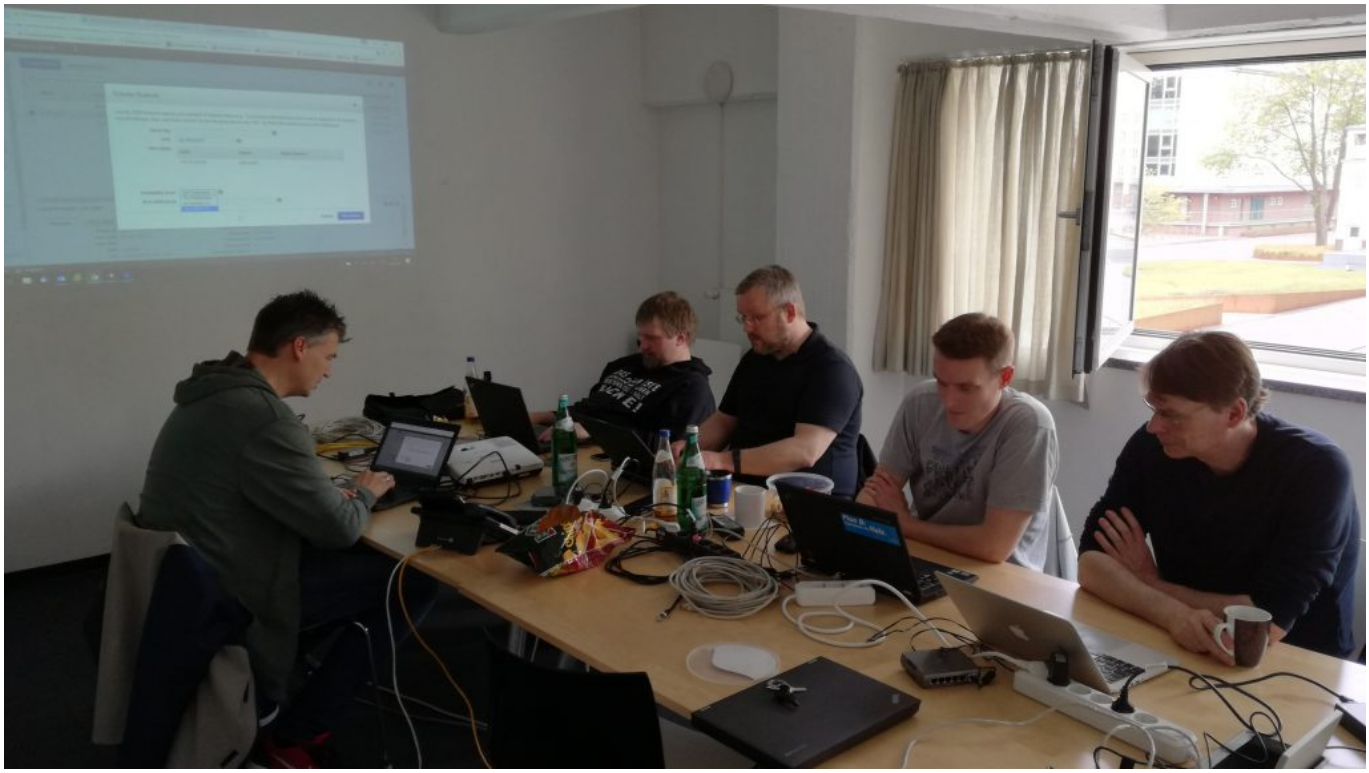
Lightning Talk „HTML 5 WebComponents“

17 Aktivposten, ein Organisator und ein paar Schaulustige (dazu gehöre dann wohl ich) konnten sich zunächst zwei informative „Lightning Talks“ zu den Themen „HTML 5 WebComponents“ und „WebGL“ anhören. Wer nicht teilnehmen konnte und sich trotzdem für eine grundlegende Einführung interessiert (Ursprung, Intention, Potential, Verbreitung), den Verweise ich gerne auf die beiden Referenten.

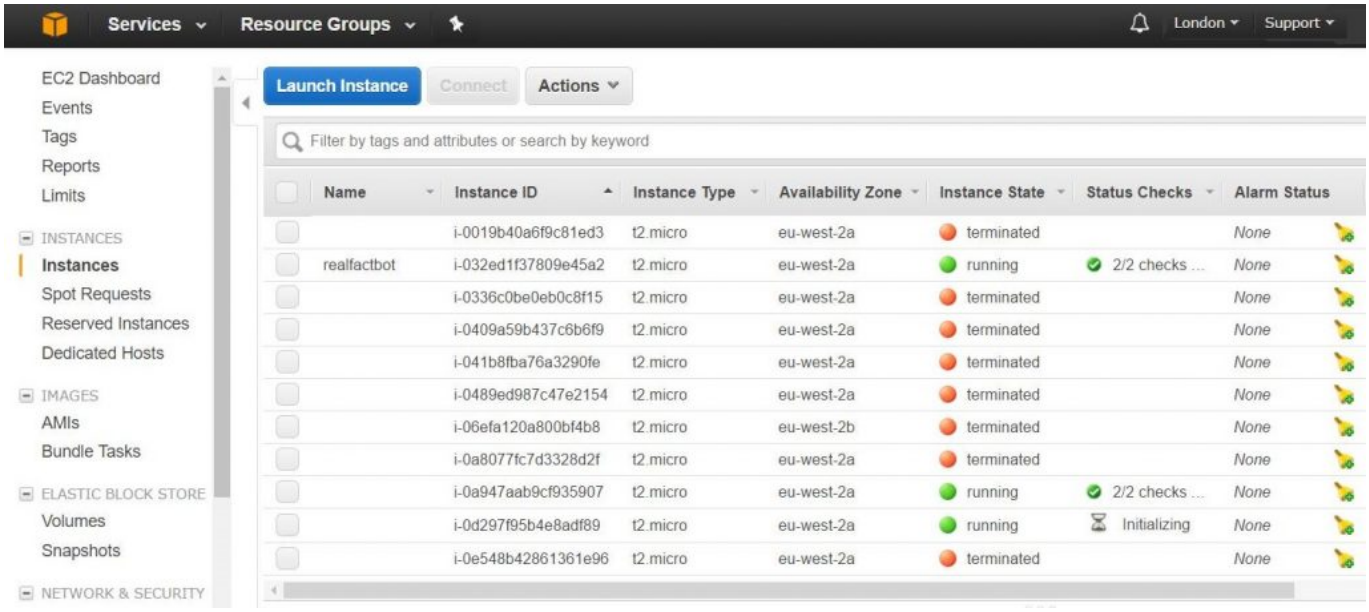
Danach wurden vier Projektvorschläge vorgestellt, von denen drei basisdemokratisch ausgesucht und in Folge in den gut besetzten Teams (5/6/6) bearbeitet wurden. Die Agenda sah eine Präsentation der Ergebnisse zu 16:30 Uhr vor, was alle Teams einhalten konnten.

„AWSome Scaling“ mit „Alternative Fact Provider“

Das Projekt hatte seinen Schwerpunkt in betrieblichen Aspekten. Es wurde eine Public Cloud im Hinblick auf Infrastruktur, RZ-Anbindung, Sicherheit, Automation, Deployment sowie automatische Skalierung evaluiert. Für einen reibungslosen Ablauf wurde das Teamgefüge zunächst mit Currywurst Chips geschmiert. Dann wurde von Ops Experten eine Infrastruktur aufgebaut und mit ersten Systemen bestückt, die möglichst nah an einem produktiven Einsatzszenario angelehnt war.



Fazit nach der ersten Stunde: „Wir haben Frankfurt gelöscht.“ Das war unsere Chance, ungeplant auch den Support des Cloud Anbieters zu testen. Nach einer steilen Lernkurve stand dann die Infrastruktur und wartete hungrig auf eine ganz besondere Anwendung.

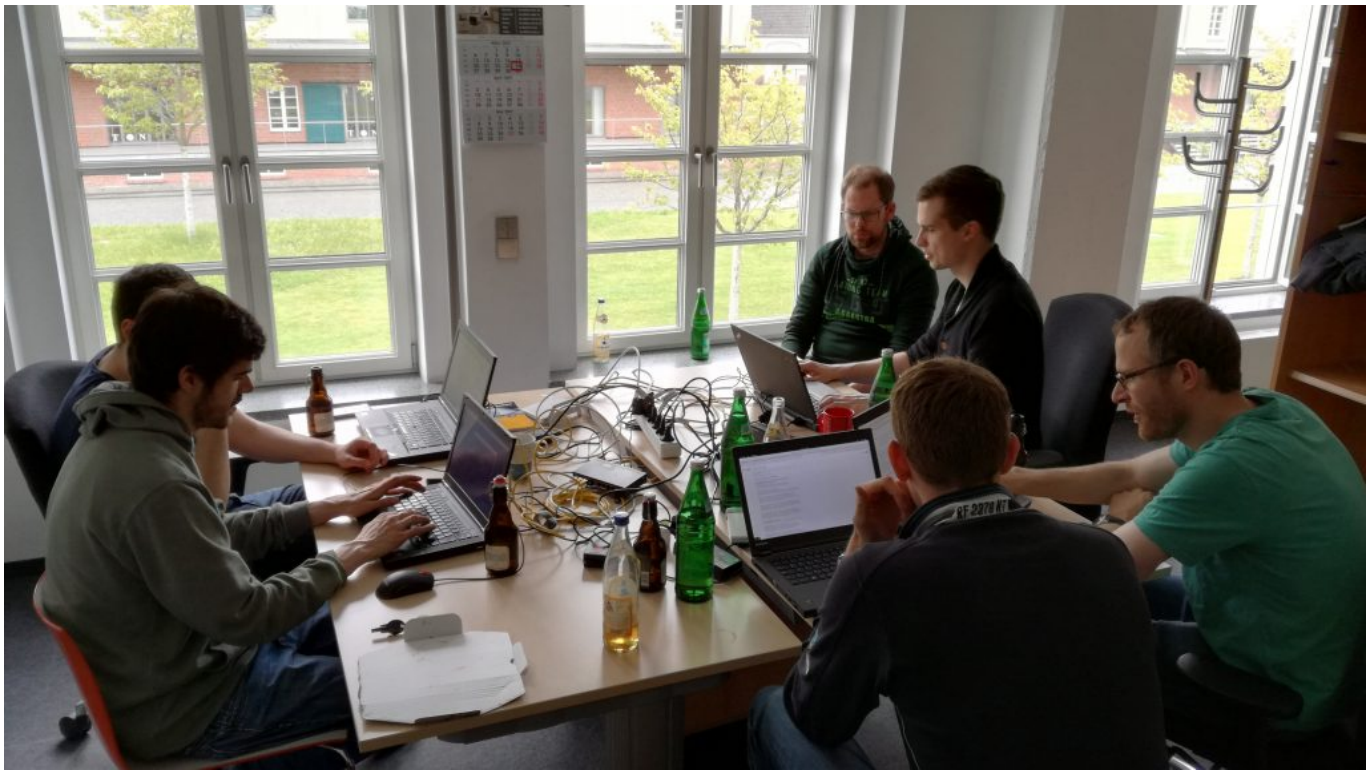


Um das angestrebte Ziel zu komplettieren, steuerten die Dev Experten die hochaktuelle Zeitgeist-Anwendung „Alternative Fact Provider“ bei. Die Anwendung wurde dabei in die Cloud deployed und kommunizierte mit einer Datenbank im Thalia-Rechenzentrum. Aspekte wie das automatisierte Ausrollen neuer Softwareversionen inklusive einer automatischen, bedarfsorientierten Skalierung der Rechenkapazität, Kommunikation mit unserer eigenen Infrastruktur und Bewerten der dabei (natürlich) auftretenden Latenzen wurden betrachtet.

Dass das Thema Cloud nicht so trivial zu handhaben ist, wie es im Hochglanz Marketing dargestellt ist, wussten wir vorher. Als eines von vielen Werkzeugen in unserem Portfolio ist „Public Cloud“ aber definitiv eine spannende Ergänzung.

Spring 5 – Functional Reactive Programming

Am Beispiel einer Chat-Funktion im Browser, mit der der Kundensupport die Kunden im Webshop unterstützen kann, hat das zweite Team neue Funktionalitäten aus dem stark ersehnten Spring 5 angetestet.



„Functional Reactive Programming“ in Spring 5 ändert maßgeblich die Art und Weise, wie Client und Server miteinander kommunizieren. Während bei gewohnter REST-Kommunikation der Client die gesamte Response erwartet, eröffnet die Reactor-Adaption von Spring 5 ein sukzessives Verarbeiten von Teil-Responses. Im Rahmen des Hackathons nutzte das zweite Team die starke Spring-Abstraktion der Websocket-Implementierung von Reactor. Dabei wurde eine „ereignisgetriebene“ Anwendung entworfen, die auch bei hoher Last mit wenigen Threads auskommt, und gleichzeitig durch die anhaltend geöffnete TCP-Verbindung per Websocket nicht (wie bei alternativen Technologien) die Nutzung der Anwendung blockiert.

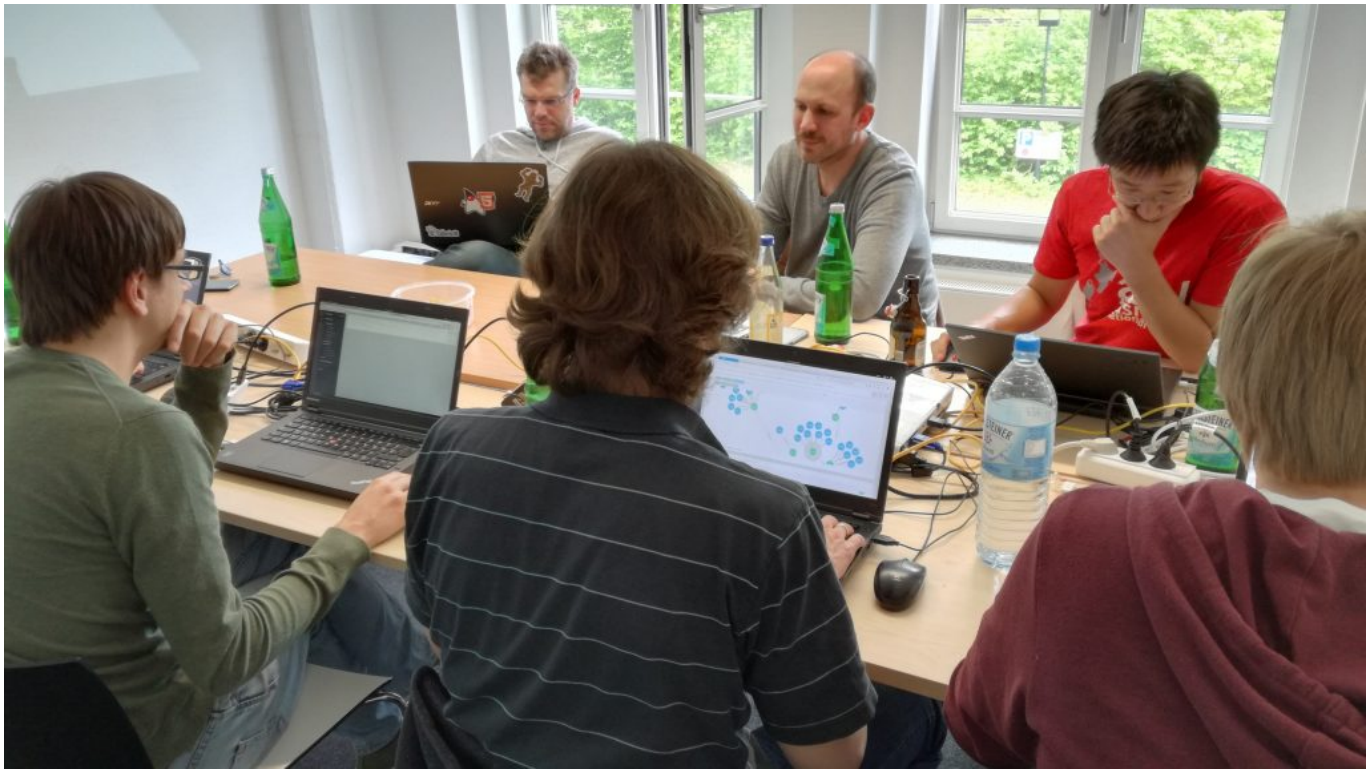
Fazit: Die Technologie ist sehr grundlegend, die Anwendungsfälle dafür zahlreich. Spring 5 wird freudig erwartet. Das würde uns erlauben, „neu über Sachen nachzudenken“.

Auswertung von Artikelbewertungen mit Hilfe einer Graphendatenbank

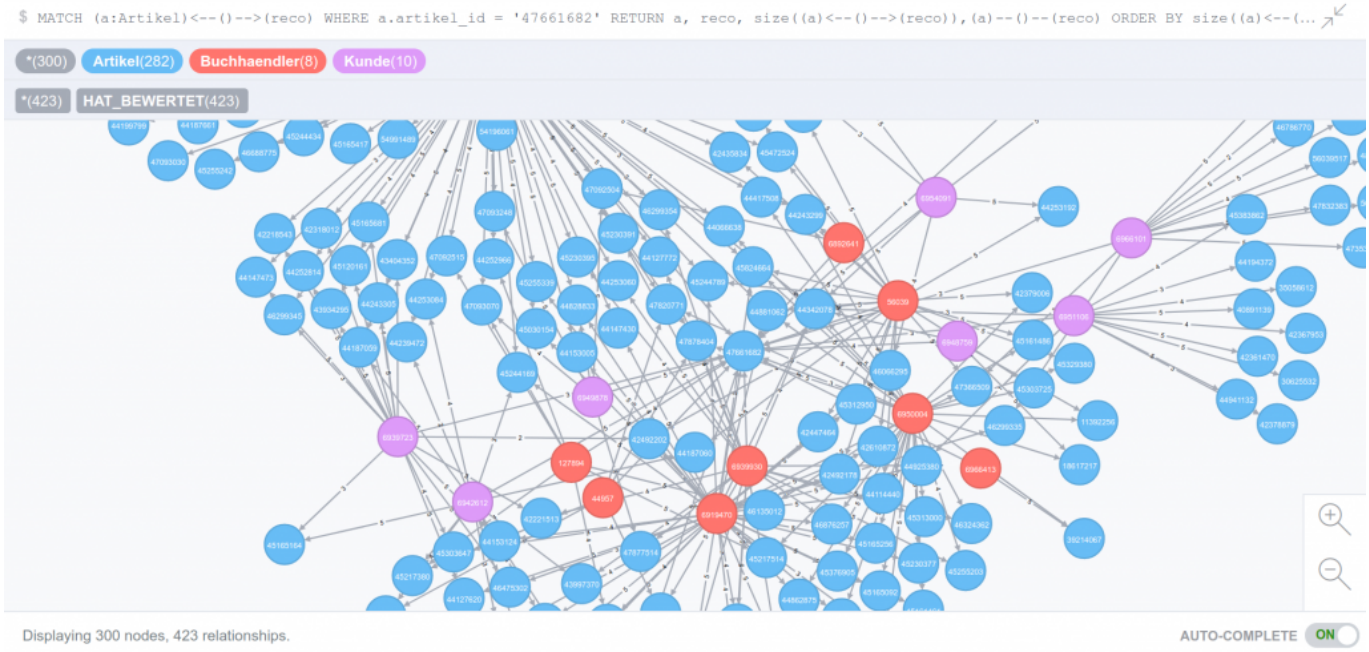
Auch das dritte Projekt hat zwei Themen (technologisch und fachlich) miteinander kombiniert. Neben dem Aufbau von technischen Erfahrungen im Bereich der Graphendatenbanken, in diesem Fall „Neo4j“, sollte die Frage beantwortet werden, wie die zahlreichen Artikelbewertungen sowohl unserer Kunden als auch unserer Buchhändler miteinander in Beziehungen stehen und inwiefern man

daraus gute Empfehlungen ableiten kann.

Weiterhin wurde eine exemplarische Spring Anwendung hochgezogen, um eine Anbindung an diese Datenbank zu prüfen.



Fazit: Das Team war beeindruckt, wieviel in so kurzer Zeit machbar ist. Die Fachobjekte „Kunde“, „Artikel“ und „Buchhändler“ wurden in diversen Darstellungen korreliert, die Ergebnisse sind plausibel. Das Reindenken in die proprietäre Abfragesprache war allerdings nicht ohne Tücken (aka lange Laufzeit). Ideen zu weiteren verwendbaren Datenquellen existieren. Hier lohnt es sich, weiter zu forschen!



Abschluss

Auf die Auswertung eines „Applaus Voting“ für das beste Projekt und damit einhergehendes Schulterklopfen und Preise wurde einhellig verzichtet. Die Erfahrungen allerdings haben die Teilnehmer gerne mitgenommen.

Das verwendete GitLab bleibt für Interessenten erstmal bestehen, die Links findet ihr im Wiki.

Als Verbesserungsvorschlag wurde das Thema „T-Shirts“ genannt, ich selber möchte noch „griffige Team Namen“ einwerfen – ich will aber natürlich nicht an einer eventuellen Eskalation bei der Namensfindung schuld gewesen sein. (Zwei Projekte hatten immerhin schon Namen mitgebracht.)

Die vorgeschlagenen und bearbeiteten Themen waren alle technologisch hochinteressant und gleichzeitig nah „am Business“, so dass sich ihre Bearbeitung für uns doppelt gelohnt hat. Ich freue mich schon, die ersten Ergebnisse in der Produktion bewundern zu können.

Müssen wir wiederholen!

Jonas de Buhr

Martin Ernst