mehr zum thema:
http://dev.otto.de

TEILE UND HERRSCHE: KLEINE SYSTEME FÜR GROSSE ARCHITEKTUREN

Ein System = ein Team, wenige Richtlinien und viel Freiheit, Eigenverantwortung statt Governance, schlanke Integration über den Web-Standard REST, das Teilen eines großen Problems in viele kleine Probleme sowie permanente Veränderung, statt Angst vor dem großen Release. In diesem Erfahrungsbericht lesen Sie, wie wir mit kleinen, lose gekoppelten Systemen eine leichtgewichtige Architektur für otto.de geschaffen haben.

Anfang 2011 hat sich OTTO zum kompletten Neubau seiner E-Commerce-Plattform otto.de entschieden – mit weitreichenden Zielen in Bezug auf das zu unterstützende Datenvolumen, die Dynamik der Daten sowie die Personalisierung und Individualisierung des Shops. Basierend auf den Erfahrungen mit dem bestehenden Shop-System, sollte die neue Software nicht mehr auf Standardsoftware, sondern auf einer Eigenentwicklung und einer komplett neuen Systemarchitektur beruhen. Parallel dazu sollte auch der Entwicklungsprozess auf ein agiles Vorgehen umgestellt werden. Das Ziel: Technologie als Kernkompetenz bei OTTO etablieren, um am wachsenden E-Commerce-Markt schnell agieren zu können.

Auf der Suche nach einer geeigneten Architektur für die neue E-Commerce-Plattform haben wir somit nicht nur die funktionalen und nicht-funktionalen Anforderungen bewertet, sondern auch die organisatorischen. Wie können wir Geschwindigkeit im Entwicklungsprozess umsetzen, ohne große technische Schuld aufzubauen? Wie stellen wir sicher, dass die Software wachsen kann, ohne dass wir die Kontrolle verlieren? Wie können Teamstrukturen wachsen, ohne die Abstimmungsaufwände zu erhöhen?

Die Antworten auf diese prozessualen und organisatorischen Fragen haben wir in der agilen Softwareentwicklung nach Scrum und den Prinzipien von *Continuous Delivery* gefunden. Die Suche nach der dazu passenden Architektur stand allerdings in keinem Lehrbuch.

Wie sieht also eine Software aus, die in allen relevanten Dimensionen skaliert? Gemeint ist hier Skalierung im technischen Sinne, d. h. zur Erfüllung der Last- und Performance-Anforderungen an otto.de, dem Sortimentsvolumen und steigenden Kundenzahlen, vor allem aber auch Skalierung im organisatorischen Sinne, d. h. wie Menschen und Teams organisiert werden können, damit sie effektiv und effizient zusammen arbeiten.

Wir entwickeln an einer Software, die ständiger Veränderung unterworfen ist und die ständig wächst. Von mehreren 100 zu mehreren 1.000 Zeilen Quellcode, sind wir nach eineinhalb Jahren Entwicklung mittlerweile schon bei mehreren 100.000 Zeilen Quellcode angelangt. Angefangen mit einem Kernteam von wenigen Entwicklern, sind wir inzwischen ein ca. 100-köpfiges, interdisziplinäres Team. Gleichzeitig steigt der Druck, immer mehr Anforderungen umzusetzen. Wartbarkeit und Änderungsfähigkeit der Plattform sind somit für die Zukunft entscheidend.

Gewichtsklassen

Mit einer leichtgewichtigen Architektur wollen wir otto.de für die Zukunft rüsten. Doch wie konzipiert man ein komplexes Softwaresystem, das auch mittel- und langfristig eine Chance hat, leichtgewichtig zu bleiben? Und was ist eigentlich eine leichtgewichtige Architektur?

Bei der Definition einer leichtgewichtigen Architektur soll uns die Vorstellung helfen, was wir unter einer *schwergewichtigen Architektur* verstehen:

- Groß
- Monolithisch
- Wenig modularisiert
- Eng gekoppelte Teilsysteme
- Einsatz von Applikationsservern
- Message Queues
- In horizontalen Schichten geschnittene Systeme
- Detaillierte Architekturvorgaben
- Governance mit entsprechenden Freigabe- und Genehmigungsprozessen
- Seltene, große Releases

Unter einer *leichtgewichtigen Architektur* verstehen wir folglich das Gegenteil:

- Kleine Systeme
- Modular aufgebaut und lose gekoppelt

die autoren

Stephan Kraus
(stephan.kraus@otto.de)
hat über zehn Jahre Erfahrung in der Entwicklung und dem Betrieb von großen Web-Anwendungen und E-Commerce-Sites. Er ist Teamleiter Softwareentwicklung im Bereich E-Commerce bei OTTO in Hamburg.

Guido Steinacker
(guido.steinacker@otto.de)
entwickelte bei OTTO im Rahmen eines Prototypen die Grundlagen für die neue Shop-Architektur und ist mittlerweile als Senior-Software-Architekt im Bereich E-Commerce für die Entwicklung des Recommendation-Systems verantwortlich.

Oliver Wegner
(oliver.wegner@otto.de)
ist Abteilungsleiter für Architektur und Qualitätssicherung im eCommerce-Bereich von OTTO. Seit zwei Jahren leitet er als technischer Produktmanager die Neuausrichtung der otto.de Plattform.

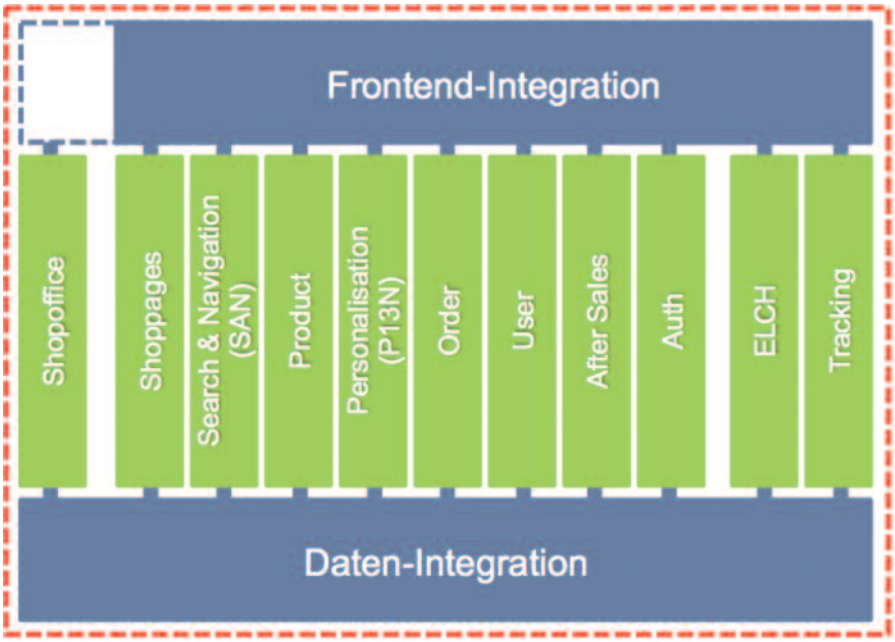

Abb. 1: Systemschnitt anhand von Fachdomänen.

- Wenige Frameworks und Abhängigkeiten
- Leicht testbar
- Nutzung von Open-Source-Komponenten
- Schnelle Release-Zyklen

Mit den genannten Rahmenbedingungen im Hinterkopf und der Vorstellung einer leichtgewichtigen Architektur gehen wir in die konkrete Umsetzung: Auf welchen Architekturprinzipien und Leitlinien basiert das neue otto.de, warum haben wir uns für diese Architektur entschieden und was sind die Erfahrungen?

Conway's Law

Auch bei einer Neuentwicklung entstehen häufig monolithische Architekturen mit einer einzigen Code-Basis und wenigen Artefakten. Die Modularisierung erfolgt eher innerhalb einer Anwendung. Nur selten wird ein einmal entstandenes System in eigenständige Teilsysteme zerlegt.

Interessanterweise entstehen diese grobgranularen Strukturen vor allem dann, wenn nur ein oder zwei Teams an der Entwicklung beteiligt sind, während mehrere unabhängige Teams dazu neigen, separate Anwendungen zu entwickeln. Diese Neigung von Organisationsstrukturen, sich in der Softwarearchitektur abzubilden, wird auch als „Conway's Law" (vgl. [Con68]) bezeichnet. Neuentwicklungen starten häufig mit einem oder wenigen Teams und führen Conway zufolge leicht zu besagter grob-granularer Struktur.

Während es anfangs noch vollkommen angemessen ist, mit einem einzelnen System und einer Code-Basis zu starten, ändert sich das im Laufe der Zeit. Das System wächst sehr schnell, sodass es für einzelne Entwickler immer schwieriger wird, einen Überblick über das gesamte System zu behalten. Häufig wächst parallel dazu auch das Team.

Mit dem wachsenden System ändern sich auch die Ziele der Architektur: Geht es am Anfang um eine einfache und schnelle Entwicklung, werden bei zunehmender Größe des Systems Modularität und lose Kopplung wichtiger. Achtet man jedoch nicht sehr frühzeitig darauf, entsteht genau das, was man vermeiden wollte: eine große, schwer wartbare, monolithische Architektur.

Wir haben daher frühzeitig das Gesamtsystem in kleinere Teilsysteme geschnitten, die jeweils von einem Team entwickelt werden. Den technischen Schnitt dieser Teilsysteme haben wir anhand unserer spezifischen Fachdomäne vorgenommen.

Vertikale

Im einfachsten Fall lässt sich ein Online-Shop wie otto.de beispielsweise in zwei Anwendungen zerlegen: den Shop und eine Backoffice-Anwendung, über die der Shop gesteuert wird. Folgt man der inhärenten Logik eines Online-Shops (Produkt suchen, auswählen, bewerten, bestellen usw.), lassen sich jedoch noch weitere Kandidaten für unabhängige Teilsysteme identifizieren (siehe Abbildung 1). Diese Teilsysteme nennen wir *Vertikale*. Sie sind durch folgende Eigenschaften gekennzeichnet:

- Separat deploybare Einheit
- Eigenes Frontend
- Eigene Datenhaltung
- Klar definierte Zuständigkeit für Daten und Features
- Eigene Code-Basis, d. h. kein geteilter Code zwischen den Vertikalen
- Kein geteilter Zustand zwischen den Vertikalen
- Kommunikation über definierte *REST*-Schnittstellen (*REpresentational State Transfer*)

Aus diesen Eigenschaften lässt sich implizit schließen, warum wir uns für den Begriff „Vertikale" entschieden haben: Im Gegensatz zu einer horizontalen Architektur, in der ein Gesamtsystem nach technologischen Aspekten, wie z. B. Datenhaltung, Anwendungslogik und Präsentation, geschnitten ist, ist eine vertikale Architektur nach fachlichen Aspekten geschnitten und beinhaltet sämtliche horizontale Schichten.

Eine Vertikale lässt sich dabei von anderen Vertikalen weitgehend entkoppeln (*Loose Coupling*) und ist für eine begrenzte, klar definierte Menge von Aufgaben zuständig (*High Cohesion*). Beides sind Begriffe, die aus der objektorientierten ▸

Abb. 2: Beispiel für eine Web-Shop-Seite mit den jeweils ausgelieferten Seitenfragmenten.

Entwicklung bekannt sind, aber bei der Definition von Systemgrenzen häufig aus den Augen verloren werden.

Makro- und Mikro-Architektur

Das Prinzip „Divide & Conquer" kennt vermutlich jeder Softwareentwickler: Teile *ein großes* Problem in *viele kleine* Probleme, die du beherrschen kannst. Mit den Vertikalen wenden wir dieses Prinzip auf die gesamte Architektur an.

Im Ergebnis erhalten wir kleine, lose gekoppelte Systeme, die weitgehend unabhängig voneinander lauffähig sind. Durch die überschaubare Größe der Vertikalen können wir auf komplexe Frameworks vollständig verzichten. Jeder Entwickler im Team versteht sein System und neue Entwickler können sich schnell einarbeiten und produktiv werden.

Die Autonomie der Vertikalen erhöht auch die Testbarkeit und gestattet eine kontinuierliche Livestellung von Features (fast) ohne übergreifende Release-Prozesse.

Diese Vorteile entstehen selbstredend nicht allein dadurch, dass ein großes System in mehrere kleine zerteilt wird. Hierfür werden übergreifende Architektur- und Entwicklungsrichtlinien benötigt, damit alle Teile zueinander passen. Wir unterscheiden dabei zwischen Makro- und Mikro-Architektur. Die Kernidee dahinter: Mit wenigen (bei uns insgesamt vier) Richtlinien zur Makro-Architektur wird das Zusammenspiel des Gesamtsystems festgelegt. Die Mikro-Architektur definiert den inneren Aufbau einer Vertikalen und wird von jedem Team selbst festgelegt.

Die *Makro-Architektur* ist in unserem Fall sehr stark fachlich getrieben. Sie beschreibt auf technischer Ebene, wie mehrere autonome fachliche Systeme (Vertikalen) miteinander interagieren. Auf dieser Ebene muss bzw. darf nicht bekannt sein, wie die interne Architektur eines fachlichen Systems aussieht oder wie sie implementiert wurde. Unsere Regeln zur Makro-Architektur lauten:

- *Vertikaler Systemschnitt:* Das Gesamtsystem besteht aus einzelnen Teilsystemen, die wir „Vertikale" nennen (siehe oben).
- *Klare Verantwortlichkeit für Daten und Datenversorgungsprozesse:* Es gibt jeweils nur ein System, das ein Datenobjekt führend hält. Andere Systeme greifen über die REST-Schnittstellen auf diese Daten zu und halten sie bei Bedarf redundant, um synchrone Zugriffe auf die führende Vertikale zu vermeiden.
- *REST-Architektur:* Die Kommunikation zwischen der Vertikalen erfolgt über REST-Schnittstellen. Auch Aufrufe aus einem Browser sind als Aufrufe eines REST-Services zu verstehen, die eine HTML-Repräsentation einer Ressource anfordern.
- *„Shared Nothing" als Architekturprinzip:* Die Vertikalen halten keinen gemeinsamen Zustand und teilen sich untereinander auch keine Daten in gemeinsamen Datenbanken.

Die *Mikro-Architektur* beschreibt, wie eine Vertikale intern aufgebaut ist. Auf dieser Ebene ist beispielsweise relevant, welche Programmiersprachen und Bibliotheken verwendet werden und welches Datenbanksystem eingesetzt wird. Auf der Mikro-Ebene reagiert das System flexibel auf technischen Wandel und veränderte Businessanforderungen.

In der Mikro-Architektur hat jedes Team maximale Freiheitsgrade. Der Technologie-Stack (z. B. Spring MVC, Play oder Jersey) kann von jedem Team selbst gewählt werden. In den Teams liegt auch die Betriebsverantwortung (Livestellung, Wartung, Fehlerbehebung) für die jeweilige Vertikale, wodurch der Anreiz erhöht wird, auf eine entsprechende Testbarkeit, Testabdeckung und Code-Qualität zu achten.

Wie aus den einzelnen Vertikalen und den Regelungen der Makro- und Mikro-Architektur eine Anwendung entsteht, wollen wir im Folgenden anhand einiger konkreter technischer Lösungen vorstellen.

Page Assembly

Betrachten wir eine typische Seite aus einem Shopsystem, finden wir Elemente wie zum Beispiel eine Suche, den Warenkorb, eine Navigation, Produktinformationen oder Produktempfehlungen. Eine solche Seite setzt sich aus einzelnen Fragmenten zusammen, die von unterschiedlichen Vertikalen ausgeliefert und dann zu einer Seite zusammengesetzt werden (*Page Assembly*):

- Das *Search+Navigation-System* ist für die Navigationsstruktur und die Suche verantwortlich.
- Der Mini-Warenkorb wird vom *Order-System* ausgeliefert.
- Produktinformationen kommen aus dem *Product-System*.
- Empfehlungen zu dem Produkt kommen aus dem *Recommendation-System*.

Jede Seite des Systems liegt in der Zuständigkeit einer Vertikalen und wird von dieser aggregiert und ausgeliefert. Die zuständige Vertikale liefert jeweils den Seitenrahmen inklusive des wesentlichen Inhalts, und bindet den Inhalt anderer Vertikalen in die Seite ein (**siehe Abbildung 2**). Nehmen wir als Beispiel die Einbindung des Mini-Warenkorbs (**in Abbildung 2 oben rechts**) in die verschiedenen Seiten. Die Einbindung der Inhalte kann dabei client- oder serverseitig erfolgen.

Unobtrusive JavaScript

Der Inhalt des Warenkorbes (Anzahl der Artikel und aktueller Wert) wird vom Order-System über eine REST-Ressource zur Verfügung gestellt. Bei der clientseitigen Integration verwenden wir *unobtrusive* JavaScript.

Das die Seite auslieferndе System fügt an der passenden Stelle einen HTML-Link mit einem `class`-Attribut und einer `href` auf die Warenkorb-Ressource in die Seite ein. Über JavaScript werden die Daten von diesem Link geladen und im Browser in die Seite eingebunden (**siehe Listing 1**).

Das Skript zum Auflösen des Links und zum Integrieren der Daten in die Seite wird in diesem Fall von dem Order-System verantwortet. Das System, das die Seite ausliefert, ist damit bis auf eine Vereinbarung über die Integration des Warenkorbs vom Warenkorb-System entkoppelt. Dieses Paradigma ist auch als *Resource-Oriented Client Architecture (ROCA)* bekannt (vgl. [ROC]).

```
<a href=„/minibasket/42“ class=„basket“>Warenkorb</a>
+
$(„a.basket“).ready(function() {
    loadAndDisplay(this.href);
});
```

Listing 1: *Beispiel für den Einsatz von Unobtrusive JavaScript.*

```
<esi:include
    src=„http://example.com/minibasket/42“
onerror=„continue“ />
```

Listing 2: Beispiel für einen ESI-Include.

Edg Side Includes
Eine andere Variante ist die Einbindung des Fragments über *Edge Side Includes (ESI)* (vgl. [W3C01]) bzw. *Server Side Includes* (vgl. [Apa13]): Dabei liefert eine Vertikale zunächst den Seitenrahmen aus und fügt für den Warenkorb ein ESI in die Seite ein (siehe Listing 2).

Ein vorgeschalteter Varnish-Server (vgl. [Var]) (oder ein WebServer bei Verwendung von SSI) parsed die Seite und fügt anstelle der Includes das HTML-Schnipsel ein, das beim Auflösen der URL zurückgegeben wird. An den Browser wird dann die vollständige HTML-Seite einschließlich des Warenkorb-Fragments geliefert.

Natürlich gibt es auch noch verschiedene Varianten einer serverseitigen Integration, die auf Ad-hoc-Anfragen à la SOAP oder RPC zwischen Vertikalen basieren: Diese Art der Integration hat jedoch viele Nachteile, inklusive einer engen Kopplung und eingeschränkten Testbarkeit, weswegen wir hier nicht weiter darauf eingehen wollen.

Asset Server

Obwohl – je nach Seite – unterschiedliche Vertikalen für den Aufbau zuständig sind, sollen sich diese verschiedenen Seiten selbstverständlich einheitlich präsentieren. Damit die verschiedenen Teilsysteme sich dieselben *Cascading Style Sheets (CSS)*, Images und JavaScript-Bibliotheken teilen können, haben wir uns für die zentrale Auslieferung solcher geteilten Ressourcen über einen *Asset Server* entschieden. Hier ist auch die Minifizierung von CSS und JavaScript verortet.

Die von diesem Server verwalteten *Assets* bilden die einzigen Code-Bestandteile, die zwischen den verschiedenen Systemen geteilt werden – doch dazu später mehr.

Aus technischer Sicht ist der *Asset Server* nichts anderes als ein Web-Server zur Auslieferung statischer Inhalte. In unserem Fall verwenden wir dafür „NGiNX“ (vgl. [NGI]).

Datenhaltung und Datenversorgung

Da wir während der Seitenauslieferung auf Ad-hoc-Anfragen zwischen den Anwendungen verzichten wollen, stellt sich natürlich die Frage, wie die einzelnen Systeme mit Daten versorgt werden können. Immerhin handelt es sich um einen Online-Shop und auf die eine oder andere Weise werden viele Komponenten beispielsweise etwas mit Produktdaten zu tun haben. Wir haben uns daher auf einige wenige Regeln geeinigt:

Definierte Verantwortung für Daten
Für alle Daten ist genau eine Vertikale führend verantwortlich. Nur der *Master* definiert den aktuellen Stand. Benötigt eine Vertikale Daten, für die eine andere führend ist, werden die benötigten Informationen redundant gespeichert. Jedes System hält sich die von ihm benötigten Informationen in einer für das System optimalen Struktur.

Insbesondere teilen sich die Systeme kein gemeinsames Datenbankschema (oder auch nur die Datenbank), da das wieder den Abstimmungsbedarf zwischen den Teams erhöhen würde. Ein Team wäre dann nicht mehr in der Lage, Änderungen durchzuführen, da man nicht ausschließen könnte, ein anderes Team in seiner Arbeit zu behindern.

Asynchrone Datenversorgung
Die Datenversorgung erfolgt asynchron im Hintergrund über die REST-Schnittstellen des jeweiligen Masters. Synchrone Zugriffe auf Informationen, also Zugriffe während einer Benutzeranfrage, versuchen wir zu vermeiden, um die Vertikalen weitestgehend voneinander zu entkoppeln.

Pull versus Push
Datenänderungen werden nicht aktiv verteilt (*Push*), sondern zyklisch über Feeds konsumiert (*Pull*). Dabei vermeiden wir Full-Importe und bevorzugen Delta-Importe.

Dass wir Datenänderungen vorzugsweise per Pull konsumieren, hat sich für eine stabile Datenversorgung als besonders vorteilhaft erwiesen. Überlegen wir einmal, was beispielsweise das Product-System tun müsste, um Änderungen der Produktdaten per Push zu verteilen:

- Für jede Änderung eines Produkts (z. B. Preis oder Verfügbarkeit) müsste das System die Änderung aktiv an alle Empfänger verteilen. Die Empfänger müssten dem System also bekannt sein.
- Damit die Zustellung garantiert werden kann (Datenänderungen dürfen ja nicht verloren gehen), müsste in irgendeiner Form ein Handshake erfolgen.
- Für Fehlersituationen (Ausfall eines Empfängers, Fehlermeldungen, Netzwerkprobleme usw.) muss der Sender in der Lage sein, nicht zustellbare Nachrichten später erneut zu versenden.
- Das empfangende System müsste jederzeit in der Lage sein, die Änderungen zu verarbeiten. Das ist nicht immer leicht zu realisieren: Laufen beispielsweise gerade Aggregationen auf Produktdaten, sind parallele Änderungen auf dem Datenbestand unpraktisch. Verwendet man stattdessen einen Pull-Mechanismus, kann das empfangende System selbst entscheiden, wann es Datenänderungen verarbeiten kann.

In der von uns gewählten Variante, stellen die Vertikalen *Atom Pub* (vgl. [Gre07]) *Feeds* mit den Datenänderungen zur Verfügung. Die Interessenten können diesen Feed gelegentlich abfragen und die Events ab dem Zeitpunkt der letzten bekannten Aktualisierung abholen, aufbereiten und persistieren.

Damit können die Interessenten auch leicht mit Fehlersituationen umgehen: Sollte während der Aktualisierung ein Problem auftreten, kann das System ab der letzten erfolgreichen Änderung wieder neu aufsetzen.

Auf Full-Importe kann man trotzdem nicht vollständig verzichten. Wir verwenden sie aber primär für die initiale Befüllung und für einen Abgleich zwischen dem Stand zweier Systeme bei Bedarf.

Eventually Consistent
Dass die einzelnen Vertikalen auf diese Weise nicht zu jedem Zeitpunkt über konsistente Informationen verfügen, muss man

in einem derart verteilten System akzeptieren. Eine strikte Konsistenz würde unweigerlich die Skalierbarkeit des Systems beeinträchtigen und auch wieder zu einer engeren Kopplung der Vertikalen führen.

In unserem Fall ist es jedoch ausreichend, wenn die verschiedenen Informationen regelmäßig wieder einen konsistenten Stand erreichen, also *eventually consistent* sind. Um die Zeitfenster, in denen Inkonsistenzen auftreten, gering zu halten, versuchen wir den Preis für die Bereitstellung eines Feeds möglichst niedrig zu halten. Auf diese Weise können die Feeds häufig abgerufen werden.

Ist das nicht möglich, bietet sich eine Kombination aus Push und Pull an: Stehen neue Änderungen an, sendet der Master ein Event an registrierte Empfänger, das nur als Aufforderung zum Pullen dient. Die Empfänger fragen daraufhin, wie oben beschrieben, den Feed ab. Damit ein solches Event auch mal verloren gehen darf, fragen die Empfänger trotzdem noch gelegentlich den Feed an.

Auf diese Weise lässt sich das Fenster für abweichende Datenstände sehr kurz halten, ohne eine enge Kopplung zwischen den Systemen einzugehen.

Shared Code

Einem guten Entwickler nahezulegen, seinen Code zu modularisieren oder dem Prinzip „Divide & Conquer" zu folgen, ist die eine Sache. Ihm aber zu erklären, dass die verschiedenen Teilsysteme tatsächlich eigenständige Anwendungen sind, die sich nichts teilen und die insbesondere keinen gemeinsamen Code haben sollten, eine ganz andere.

Sobald sich zwei Systeme Code teilen, werden sich die zuständigen Teams über Änderungen an diesem Code abstimmen müssen. Sie werden sich darüber einigen müssen, welche Abhängigkeiten zu 3rd-Party-Bibliotheken erlaubt sind, wann man ein Update vornehmen darf, welche Entwicklungsrichtlinien zu beachten sind, wie viel und in welcher Form zu dokumentieren und zu testen ist – und dergleichen mehr.

Wir verfolgen daher folgenden Ansatz:

- Es gibt prinzipiell keinen *Shared Code*, sondern nur 3rd-Party-Bibliotheken.
- Sobald zwei oder mehr Systeme sich Quellcode teilen wollen, extrahieren wir den Code in ein Open-Source-Projekt, veröffentlichen es auf Github (vgl. [Git]) und in Maven Central und betrachten es als normale 3rd-Party-Bibliothek.

Damit ist es die freie Entscheidung eines Teams, die Bibliothek zu verwenden – oder eben nicht. Wir erhalten die Unabhängigkeit zwischen den Teams und eine lose Kopplung der Systeme.

Shared Nothing

Während *Shared Code* die Skalierbarkeit der Teams beeinträchtigt, ist jede Form von *Shared State* ein begrenzender Faktor für die Skalierbarkeit der Systeme: Das Problem ist dabei nicht, dass eine Information in zwei Systemen gespeichert ist, sondern dass der Zustand sich ändern könnte und die Systeme sich dann über diese Zustandsänderung synchronisieren müssen.

Bei einigen wenigen Systemen ist das kein Problem. Viele Applikationsserver halten beispielsweise HTTP-Sessions im Hauptspeicher und verteilen die Daten dann über das Netzwerk. Kleine Cluster lassen sich auf diese Weise einfach und effizient betreiben.

In großen Clustern funktioniert diese Lösung jedoch nicht mehr zuverlässig. Wir haben uns daher für den *Shared-Nothing*-Ansatz entschieden: Eine Vertikale hält keinerlei *Shared State*: keine Session-Daten und auch keine lokalen Caches. Stattdessen werden solche Daten im Browser, in HTTP-Caches (Browser, Content Delivery Network (CDNs), Reverse Proxys), in verteilten Caches (z. B. Memcached Server) oder Datenbanken gehalten. Alle Systeme haben damit dieselbe Sicht auf Informationen und bleiben agnostisch, was die Existenz anderer Systeme angeht. Auf diese Weise erreicht man zwar nicht die maximale *Single-Node*-Performance, dafür aber eine ausgezeichnete Skalierbarkeit.

Damit man diese Skalierbarkeit nicht eine Ebene tiefer, etwa in der Datenbank, wieder verliert, muss man auch hier nach passenden Lösungen suchen. Das ist einer der Gründe dafür, warum wir uns gegen den Einsatz von relationalen Datenbanken und für die „MongoDB" entschieden haben: Ein klassisches *Relationales Datenbanksystem (RDBMS)* lässt sich einfach nicht horizontal skalieren.

Fazit

Indem wir *Divide & Conquer* nicht nur innerhalb einer Anwendung, sondern als Prinzip auf die gesamte Architektur und Organisation angewendet haben, haben wir es geschafft, ein wirklich komplexes Stück Software in Form von vielen kleinen Systemen zu entwickeln. Für die Herausforderungen der technischen Integration haben wir auf Basis einer lose gekoppelten REST-Architektur gute Lösungen gefunden. Der große Vorteil dieses Prinzips zeigt sich jedoch in der schlanken Organisationsstruktur, in der interdisziplinäre Teams hochmotiviert und eigenverantwortlich Lösungen für ihre spezifischen Anforderungen entwickeln. Während der Projektlaufzeit konnten wir nachweisen, dass diese Organisation einerseits skaliert und andererseits sehr gut steuerbar ist.

Es zeigt sich, dass Conway recht hatte – und das man Conways Gesetz auch gezielt einsetzen kann, um leichtgewichtige Architekturen zu entwickeln und Organisationen daran auszurichten. ■

Literatur & Links

[Apa13] The Apache Software Foundation, Introduction to Server Side Includes, 2013, siehe: www.httpd.apache.org/docs/2.4/howto/ssi.html

[Con68] M.E. Conway, How Do Committees Invent?, 1968, siehe: www.melconway.com/Home/Conways_Law.html

[Git] Otto.de auf Github, siehe: www.github.com/otto-de

[Gre07] J. Gregorio, B. de hOra (Hrsg.), The Atom Publishing Protocol, 2007, siehe: www.tools.ietf.org/html/rfc5023

[NGI] NGINX, siehe: www.nginx.com

[ROC] ROCA – Resource-oriented Client Architecture, www.roca-style.org

[Var] Varnish Software, Varnish Cache, siehe: www.varnish-cache.org

[W3C01] W3C, ESI Language Specification 1.0, W3C Note 04 August 2001, siehe: www.w3.org/TR/esi-lang