Ausgabe 04/2019

Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler

D: 4,90 EUR | CH: 9,80 CHF | A: 5,60 EUR | Benelux: 5,80 EUR

4 191978 304903 04

Java aktuell

iJUG Verbund
www.ijug.eu

Java 12
Die neuen Features im Überblick

Jakarta EE
Neuigkeiten und Ergebnisse der Verhandlungen

Web-APIs
Diese Programmierschnittstellen erleichtern Ihnen die Arbeit

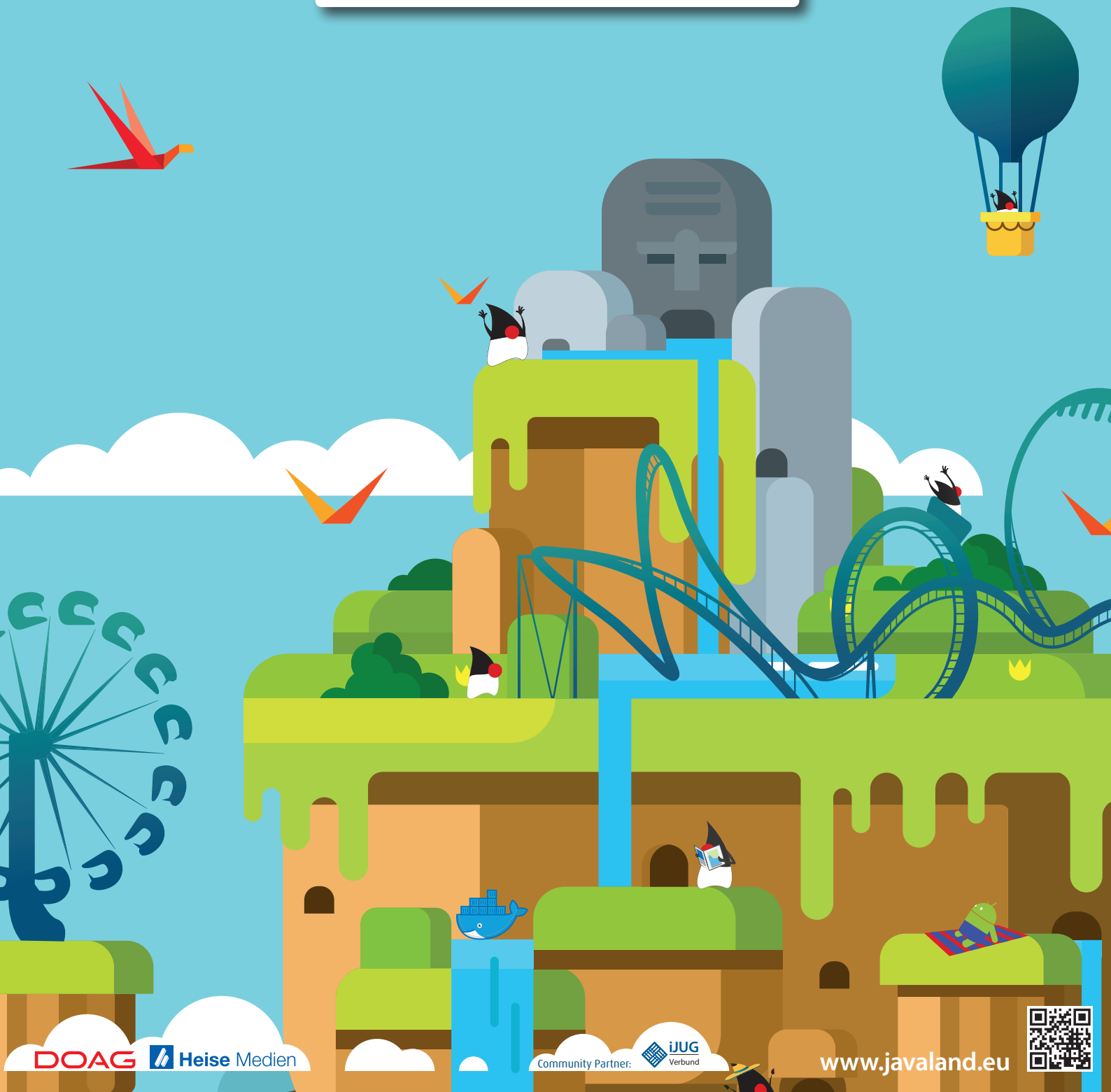
Java im Wandel

JavaLand

Save the Date

16. - 19. März 2020

in Brühl bei Köln



DOAG Heise Medien

Community Partner: iJUG Verbund

www.javaland.eu

Hinter den Kulissen von SCS-Architekturen

Mirko Sertic, Thalia Bücher

Dieser Artikel schaut hinter die Kulissen einer großen SCS-Systemlandschaft. Am Beispiel eines bestehenden eCommerce-Systems wird gezeigt, wie die monolithische Struktur in eine auf dem Self-Contained-Systems-Ansatz basierende Systemlandschaft überführt wurde. Besonderes Augenmerk wird dabei auf die fiesen kleinen Details gelegt. Details, die im SCS-Manifest nicht erläutert werden, sich jedoch als Konsequenz dieser Architektur manifestieren. Die fiesen Details eben, die ein Migrationsprojekt durchaus erfolgreich verhindern können.

Worum geht es hier also genau? Es geht um ein großes Unternehmen aus Deutschland, das durch Omni-Channel-Buchhandel Bekanntheit erlangt hat. Dieses Unternehmen wird im Jahr 2019 100 Jahre alt. Nicht ganz so alt ist die Code-Basis des eCommerce-Systems. Dieses System hatte eine monolithische Struktur. Gerade diese Struktur machte es immer schwerer, mehr Features in kürzerer Zeit für die Kunden zu implementieren. Dieser Artikel beschäftigt sich mit der Migration dieser Systemlandschaft in eine neue Struktur, die auf dem Self-Contained-Systems-Ansatz basiert. In Form einzelner Lektionen möchte ich auf Details eingehen, die für einen erfolgreichen Einsatz des SCS-Ansatzes berücksichtigt werden müssen. Ich arbeite bei der Thalia Bücher GmbH. Bevor ich jedoch in der Verlegenheit komme, Interna auszuplaudern und meinen Job zu verlieren, habe ich deshalb einen Disclaimer. Bei dem hier gezeigten Migrationsprojekt handelt es sich natürlich nicht um die eCommerce-Landschaft der Thalia Bücher GmbH. Es geht hier um unseren schärfsten Konkurrenten, die „Welt, bleib wach!" Online Bücher GmbH *(siehe Abbildung 1)*!

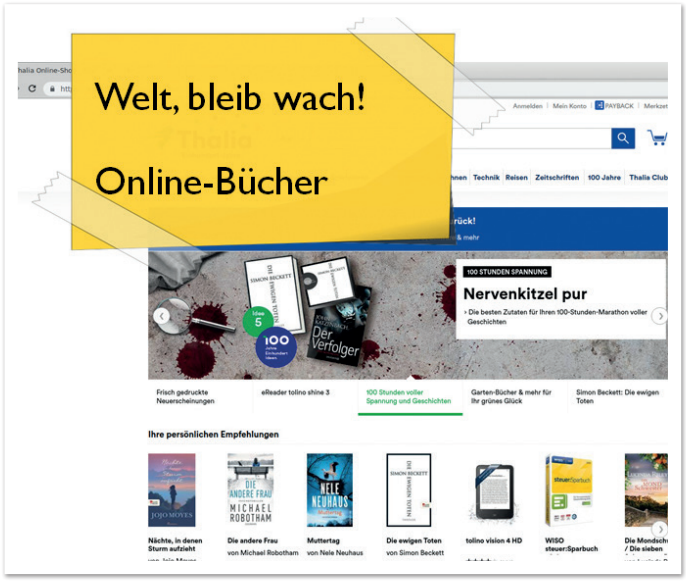


Abbildung 1: Internetauftritt „Welt, bleib wach!" (Quelle: Mirko Sertic)

Lektion #1: Die Organisation

Die Organisation ist das erste, fiese Detail. Der Self-Contained-Systems-Ansatz soll Organisationen skalieren. Diese Skalierung hat im Wesentlichen das Ziel, mehr Funktionalität in weniger Zeit umzusetzen und insgesamt (IT-)Projekte erfolgreicher und wirtschaftlicher zu machen. Im ersten Schritt muss also die Organisation angepasst werden. Ohne diesen Schritt wird der SCS-Ansatz nicht den gewünschten Effekt haben. Bei der Skalierung müssen zwei Fragen gestellt werden: „Does it scale up?" und „Does it scale down?" Hinter diesen Fragen verbirgt sich die Wahrheit, dass nicht jede Organisation die gleichen Anforderungen hat wie beispielsweise Netflix, Spotify oder auch Google. Die „Welt, bleib wach!" GmbH hat schon sehr große Lastanforderungen an den Online-Auftritt, ist aber eben doch deutlich kleiner als Netflix. Diese Organisation muss also nicht die gleichen technischen und fachlichen Probleme lösen wie Netflix. Die Kunst ist also, eine Organisation zu finden, die konkrete Probleme löst. Als Ansatz für die neue Organisation wurde ein Schnitt auf Basis von Produkten gewählt. Mit Produkt ist in diesem Sinne eine Sammlung von Funktionalität aus Sicht des Kunden gemeint. Für diese Produkte wurden Entwicklungsteams definiert, die vom Anfang bis zum Ende für den kompletten Produktlebenszyklus verantwortlich sind. Wichtig bei diesem Schnitt war, keine Überlappungen zwischen den Produkten zu haben. Es geht also darum, klare Schnittstellen zwischen den Produkten und somit auch zwischen den Entwicklungsteams zu schaffen. Beispielhaft für die „Welt, bleib wach!" GmbH sieht der Produktschnitt dann so aus, wie in *Abbildung 2* zu sehen.

Der Screenshot aus *Abbildung 2* zeigt die Produkte aus Sicht eines Besuchers. Wir sehen das Suchformular, das zu einem anderen Produkt gehört als zum Beispiel das Anmelde-Formular, die redaktionellen Inhalte oder auch der Empfehlungsbereich. Die klare Trennung zwischen den Produkten sorgt dafür, dass etwa neue Features

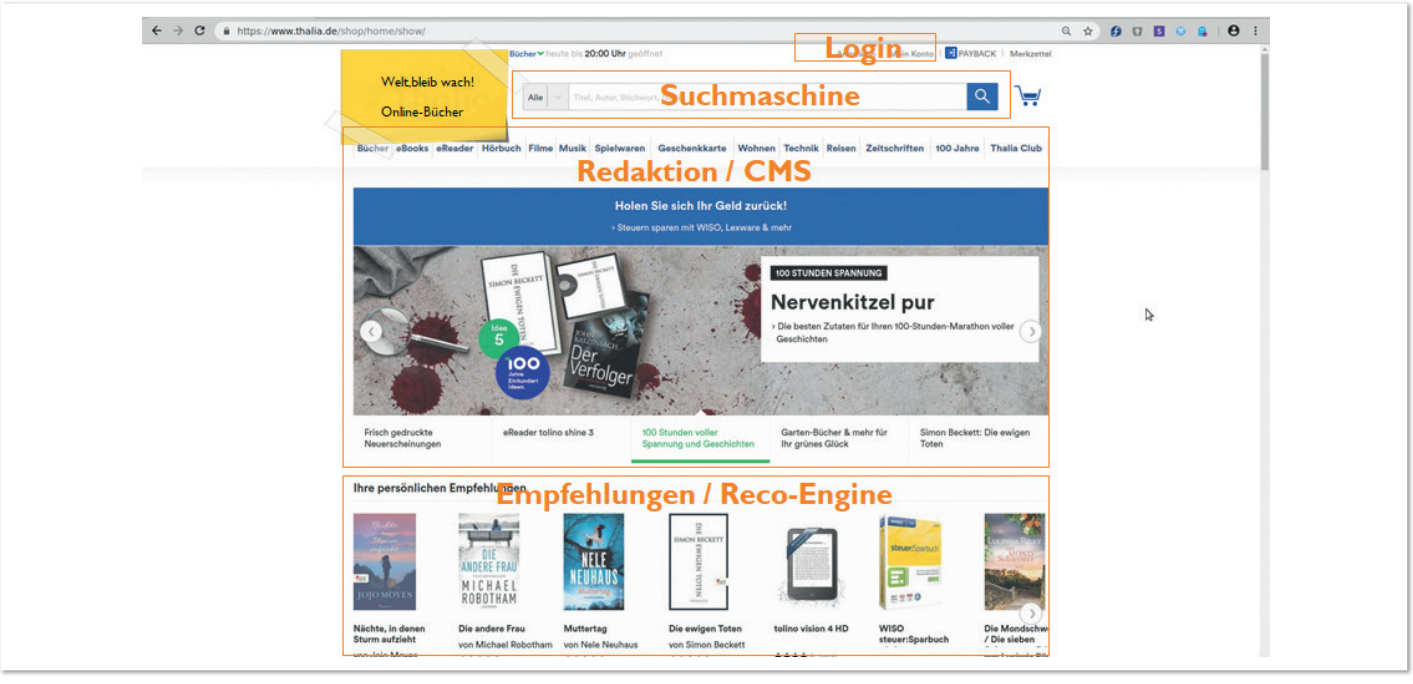


Abbildung 2: Produktschnitt (Quelle: Mirko Sertic)

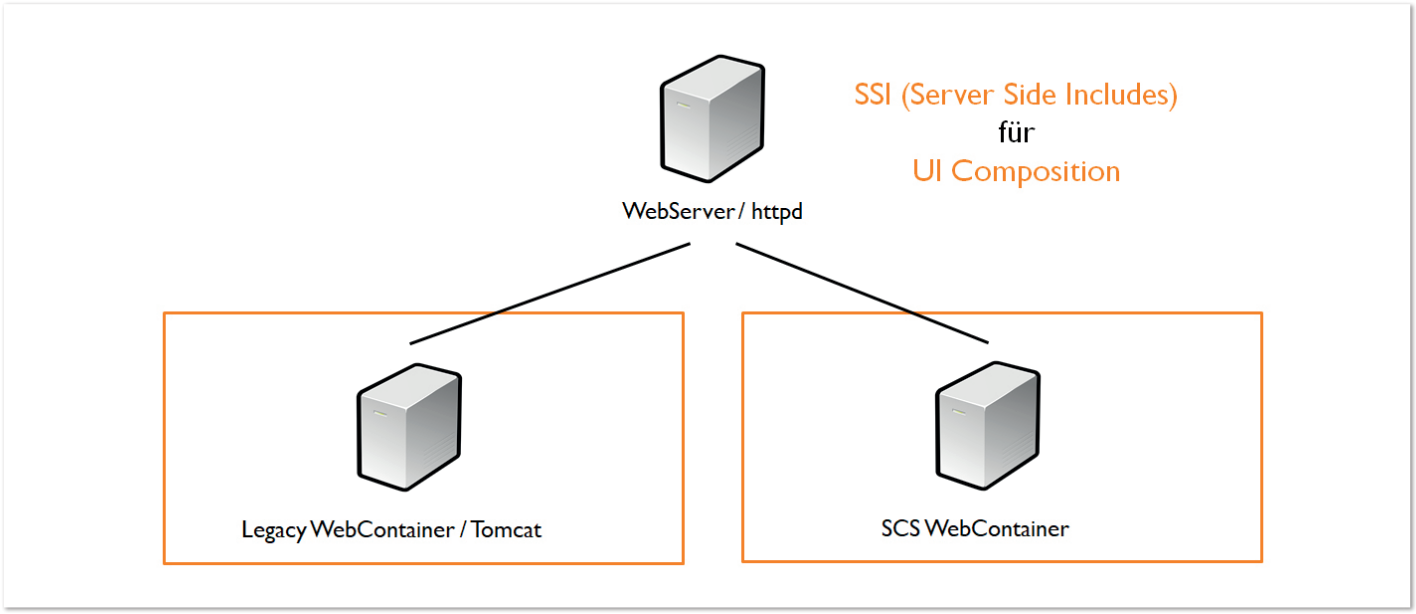


Abbildung 3: Webserver mit SSI (Quelle: Mirko Sertic)

für redaktionelle Inhalte nicht das Login beeinträchtigen können. Es gibt einen klaren Verantwortlichen für die Umsetzung dieses neuen Features pro Produkt. Produkte können unabhängig voneinander entwickelt werden. Dies ist der größte Vorteil dieser Organisationsform. Ohne diesen Schritt wird aus der Migration nur ein technisches Reengineering.

Lektion #2: Der Migrationsweg

Eine wichtige Lektion ist der Migrationsweg auf die neue Systemlandschaft. In der Theorie werden pro Produkt ein eigenes System gebaut und die Inhalte der Systeme beziehungsweise die Funktionalitäten in ein gemeinsames Frontend integriert. Aber wie soll das nun im Detail funktionieren? Ein genauer Blick auf die Inhalte kann helfen, das Problem zu lösen. Der Webshop der „Welt, bleib wach!" GmbH besteht bei genauerer Betrachtung aus zwei unterschiedlichen Inhaltstypen. Es gibt Inhalte wie das Suchfeld, die Anzeige des Anmeldestatus oder auch Slider mit Produktempfehlungen. Diese Inhalte sind in sich abgeschlossen, tauchen allerdings nie allein auf. Sie werden in einen größeren Kontext eingebunden. Dieser größere Kontext ist der zweite Inhaltstyp. Es sind eigenständige Seiten wie z.B. die Startseite, ein Suchergebnis oder auch der Inhalt des Warenkorbs beim Checkout. Für diese Seiten gibt es ein System, das führend für den Inhalt ist. Zusätzlich können jedoch auch noch andere Inhalte, wie eben der Anmeldestatus oder auch Produktempfehlungen aus anderen Systemen, eingebunden werden.

Mit welcher Technologie kann nun diese Frontend-Integration umgesetzt werden? Ein Blick auf die aktuelle Systemlandschaft zeigt, dass bereits flächendeckend HTTPD als Webserver eingesetzt wird. Dieser Webserver bringt eine interessante Technologie für die UI-Komposition mit: „Server Side Includes", kurz SSI *(siehe Abbildung 3)*.

Mit SSI-Direktiven kann der Webserver angewiesen werden, Inhalte aus anderen Systemen in ein Dokument zu integrieren. Im ersten Schritt können nun Inhalte vom ersten Typ, zum Beispiel die Slider, von einem neuen System/Produkt ausgeliefert werden. Der bisherige Monolith liefert die Slider nicht mehr aus, sondern nur noch die SSI-Direktiven, die dann vom Webserver aufgelöst werden. Durch diese Migration wird die Funktionalität in das neu zuständige System verschoben, der alte Monolith wird immer weiter ausgehöhlt und entschlackt. Bereits jetzt können die ersten Früchte der Migration geerntet werden; Slider können unabhängig als Produkt weiterentwickelt werden.

Im nächsten Schritt werden komplette Seiten in die neuen Systeme verschoben, die Inhalte vom Typ 2 werden migriert. Auf Webserver-Ebene kann dann via URL-Rewriting entschieden werden, ob eine einzelne Seite vom Monolithen oder von einem anderen SCS ausgeliefert wird. Grundvoraussetzung dafür ist natürlich, dass jede Seite eine eindeutige URL hat.

Im Normalfall bewegt sich ein Kunde in einer Session durch den Webshop. Mit dieser Session sind Zustandsinformationen verknüpft, wie etwa der Anmeldestatus oder der Inhalt des Warenkorbs. In der monolithischen Struktur gab es nur eine Session, die eventuell via Sticky-Session noch an eine bestimmte Tomcat-Instanz gebunden war. In der neuen Welt gibt es keine zentrale Session mehr, und es gibt auch keine zentrale Datenspeicherung für den Inhalt der Session. Der Anmeldestatus wird vom Login-SCS gespeichert, der Warenkorb vom Warenkorb-SCS usw. Jedes beteiligte System braucht nur noch die ID der Session zu kennen. Die Frage ist nun, wer diese ID vergibt. Die Session-ID wird als Cookie vom vorgelagerten Webserver vergeben. Die beteiligten Systeme inklusive Monolithen verwenden nur diese ID. Das dafür benötigte Federated Session Management kann sehr einfach durch einen Java-Servlet-Filter oder auch durch Frameworks wie Spring Session umgesetzt werden. Für die Migration haben wir nun eine grobe Methode – wie sehen nun die Details der Inhaltstypen aus?

Lektion #3: Frontend-Architektur

Frontends sind aus Sicht eines Backend-Entwicklers der absolute Horror. Es gibt laufend neue Frameworks, tolle, neue Build-Systeme und überhaupt immer neue Erfindungen, die alles versprechen, jedoch leider in vielen Fällen nichts halten. Hier sollte das Mantra „Resist the Hype!" gelten. Im Zweifel zählen die Produktivität eines Teams und die Time-to-Market eines Produktes mehr als die

Evaluation eines neuen Frameworks oder einer Methode. Dies soll natürlich nicht bedeuten, dass Stagnation in der technischen Basisstruktur Einzug hält. Aber gerade in einem Umfeld des stetigen Wandels ist es sinnvoll, Trends sehr genau zu beobachten, bevor auf einen fahrenden Zug aufgesprungen wird. Bei der Frontend-Architektur stellt sich zuerst die Frage, wie viele Frontends es denn überhaupt gibt.

Bei dem Migrationsprojekt der „Welt, bleib wach!" GmbH zeigte sich, dass es mehrere Frontends *(siehe Abbildung 4)* für ein SCS gibt. Redaktionelle Inhalte gibt es zum Beispiel im Online-Shop, auf den eReadern und auch in einer Android- und iOS-App. Aus historischen Gründen basiert der Online-Shop auf dem Foundation Framework, die Darstellung auf dem E-Reader basiert jedoch auf Bootstrap. Die Darstellung in den Apps ist nativ gelöst. Das SCS-Manifest schreibt vor, dass ein SCS eine autonome Webapplikation ist und die komplette Logik zur Darstellung der Inhalte mitbringt. Damit nun die beschriebenen Geräte unterstützt werden können, müssen unterschiedliche HTML-Templates für unterschiedliche Geräte definiert werden. Bei der Entwicklung von neuen Features müssen unterschiedliche HTML-Frontend-Frameworks berücksichtigt werden. Im ersten Schritt bedeutet dies natürlich einen Mehraufwand für die Produktteams. In einem Folgeschritt können die verwendeten Frameworks weiter vereinheitlicht werden. Die Motivation hier sollte natürlich die Produktivität und Time-to-Market für neue Features sein und nicht einfach die „Coolness" eines anderen Frameworks.

Am Beispiel der Apps zeigt sich ein problematischer Punkt im SCS-Ansatz: Er funktioniert nur mit Webapplikationen. Das steht natürlich auch so im Manifest, ist allerdings fern der Realität. Native Apps gibt es und sie haben ihre Rechtfertigung. Natürlich könnte die App auch durch eine WebView/Progressive Webapplication ersetzt werden, die Diskussion über Für und Wider könnte Seiten füllen. Zusammengefasst kann ich hier nur sagen, dass native Apps in Verbindung mit der SCS-Architektur problematisch sind.

Die organisatorische Trennung der Produkte soll Schnittstellen zwischen den Entwicklungsteams reduzieren und klarstellen. Damit jedoch die User-Experience nicht leidet, wenn mehrere Systeme auf der Ebene „Benutzerschnittstelle" aggregiert werden, brauchen wir ein Design-System. Dieses soll einen Leitfaden für die Implementierung der Benutzerschnittstelle bieten und als Grundlage für die Diskussion von Entwürfen dienen. Es ist also ein Kommunikationsmedium und eine Kommunikationshilfe.

Bei der technischen Implementierung von HTML-Frontends und der Aggregation sind einige Grundlagen zu beachten. Ein CSS-Namespacing auf der Ebene „Produkt" verhindert, dass CSS-Styles sich gegenseitig beeinflussen. Das Gleiche gilt für den Einsatz eines JavaScript-Modulsystems, um Seiteneffekte zu verhindern. Assets wie etwa CSS- und JS-Dateien müssen versioniert werden. Diese Versionierung ist die Grundlage für einen sinnvollen Einsatz von Content-Delivery-Networks und Caching auf den Ebenen „CDN" und „Browser". Das Caching soll natürlich die Ladezeit aus Sicht des Kunden optimieren. Eine hohe Cache-Hit-Rate im CDN kann jedoch auch helfen, Infrastrukturkosten zu sparen, da letztendlich weniger Server im Backend für die Bewältigung der gleichen Last benötigt werden.

Lektion #4: Frontend-Performance

Der Einsatz von SSI hat keinen guten Ruf. Je nach Verwendung von Apache HTTPD oder nginx werden die SSI-Direktiven sequenziell oder parallel abgearbeitet. Durch den exzessiven Einsatz von SSI-Direktiven kann eine Seite auch verlangsamt werden. Dabei ist SSI gar nicht das Problem, sondern die Tatsache, dass jede SSI-Direktive einen synchronen HTTP-Aufruf eines anderen Systems darstellt. Gerade dieses System muss den Inhalt erstmal aufbereiten, indem zum Beispiel Datenbanken befragt und Template-Engines aufgerufen werden, um HTML oder andere Inhalte zu generieren.

Da wir ja schon einen klaren Zuständigen für dieses System definiert haben, haben wir auch einen klaren Zuständigen, wenn es um das Thema Performance geht. Wenn die Seitenauslieferungszeit an den Kunden eine eindeutige Obergrenze hat, ergibt es Sinn, im Vorfeld schon Vorgaben in Form von Performance-Anforderungen an das zuständige Team zu stellen und diese Vorgaben auch durch ein entsprechendes technisches Monitoring zu überprüfen.

Die einfachste Form der Optimierung hier ist, Inhalte erst dann wirklich aufzubereiten, wenn sie tatsächlich sichtbar und für das Zielgerät relevant sind. Das Stichwort hier ist „Progressive Enhancement". Ein zuliefernder System kann beispielsweise, statt kompletten Inhalten, einen Platzhalter mit einem JavaScript ausliefern. Dieses JavaScript würde den eigentlichen Inhalt erst kurz, bevor er sichtbar wird, nachladen.

Abbildung 4: Frontends (Quelle: Mirko Sertic)

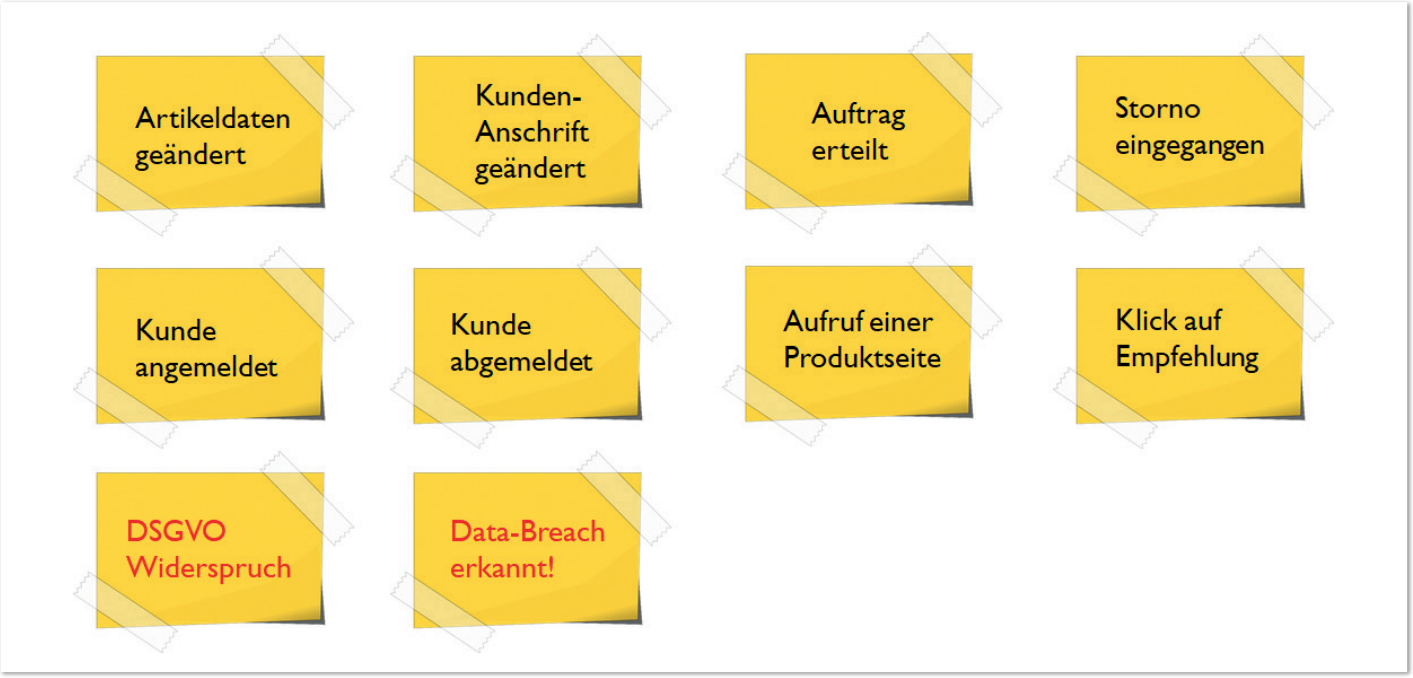


Abbildung 5: Event Storming (Quelle: Mirko Sertic)

Dynamische Inhalte haben natürlich auch ihre Schattenseite. Above-the-Fold-Inhalte nachzuladen, kann problematisch werden, wenn dadurch ein kompletter Content-Reflow der ganzen Seite gestartet wird. Auch ein Flash-of-Unstyled-Content kann in Verbindung mit Lazy-Loading das Seitenlayout vorübergehend zerstören. Auf diese Effekte muss daher unbedingt beim Einsatz von Progressive Enhancement geachtet werden. Auch das Zusammenspiel von Progressive Enhancement und Search Engine Optimization (SEO) hat seine ganz speziellen Nuancen. Allein über diesen Themenkomplex können ganze Artikel geschrieben werden. Zusammengefasst daher folgendes: Wenn SEO wichtig und geschäftskritisch ist, muss hier schon frühzeitig darauf geachtet werden.

Lektion #5: Systemintegration

Systemintegration kann in Verbindung mit dem SCS-Ansatz eine hart erlernte Lektion werden. Mit Systemintegration ist die Kommunikation einzelner SCS untereinander sowie die Kommunikation mit Drittanbietern gemeint. Fehler bei dieser Integration können die ganzen Vorteile der SCS-Architektur negieren. Das SCS-Manifest schreibt vor, dass Kommunikation soweit wie möglich asynchron erfolgen soll. Zusätzlich schreibt das Manifest vor, dass geteilte Infrastruktur minimiert werden soll. Was bedeutet das nun für die technische Umsetzung? Asynchron bedeutet, dass keine synchrone HTTP-Kommunikation zwischen Systemen und Komponenten erfolgen soll. Keine geteilte Infrastruktur bedeutet, dass es keine zentrale Datenbank mit allen Stammdaten für Kunden oder Produktinformationen gibt. Die Idee hinter diesen bewussten Redundanzen ist einfach: Zu viele synchrone Schnittstellen erhöhen das Risiko, letztendlich doch einen verteilten Monolithen zu bauen. Die Nicht-Verfügbarkeit eines Systems kann nicht die Verfügbarkeit anderer Systeme beeinträchtigen. Diese Grundsätze haben weitreichende Konsequenzen. Da es keinen zentralen Datentopf mit Stammdaten gibt, müssen diese in alle abhängigen Systeme repliziert werden. Diese Replikation kann beispielsweise via Event Sourcing erfolgen. Da in der Konsequenz auch sensible Daten wie Kundeninformationen oder Finanztransaktionen in unterschiedlichen Systemen verteilt werden, darf auf keinen Fall das Thema Datenschutz zu kurz kommen.

Als Grundlage für das Event Sourcing benötigen wir eine Spezifikation für die benötigten fachlichen Events. Diese Spezifikation kann über ein Event Storming gefunden werden, eine Teildisziplin des Domain-Driven Design. Diese Methode hilft ebenfalls dabei, Ursache-Wirkungs-Ketten zu erkennen und besser zu verstehen *(siehe Abbildung 5)*.

In der ersten Zeile finden wir recht offensichtliche Events wie „Artikeldaten geändert", „Kundenanschrift geändert" oder „Auftrag erteilt". Diese Events können Fachlogik starten und steuern. Sie können aber auch genutzt werden, um Daten zwischen Systemen zu synchronisieren. Ein „Artikeldaten geändert"-Event kann zum Beispiel genutzt werden, um die lokale Kopie von Artikeldaten in einem SCS zu aktualisieren.

Die zweite Zeile beinhaltet weniger offensichtliche Events, wie „Kunde angemeldet", „Kunde abgemeldet" oder auch „Klick auf eine Produktempfehlung". Diese Events werden beispielsweise für ein Recommender-System und die dahinter liegenden Machine-Learning-Algorithmen benötigt, können jedoch auch für den Aufbau eines Data-Warehouse oder für Predictive Analytics genutzt werden.

Die dritte Zeile hat es in sich. Wie schon erwähnt, gelten besondere Anforderungen an das Thema Datensicherheit. Die Ereignisse „DSGVO-Widerspruch" müssen etwa das Löschen oder die Anonymisierung von persönlichen Daten in allen Systemen starten. Im Umkehrschluss muss das „Data Breach erkannt!"-Event alle Kennwörter oder Tokens von betroffenen Benutzern in allen Systemen invalidieren oder auch ein Report über die betroffenen Daten an eine zentrale Instanz generieren.

Das SCS-Manifest schreibt vor, dass zentrale Infrastruktur minimiert werden soll. Als Konsequenz der asynchronen Architektur

benötigen wir dennoch eine Middleware für den Datenaustausch. Hier empfiehlt sich der Einsatz einer Pull-Architektur, da dies die Komplexität auf der Ebene „Message-Broker" reduziert und die Implementierung von Back-Pressure erleichtert. Apache Kafka kommt als Lösung infrage, da wir auf diesem Wege nicht nur ein Publish-Subscribe Messaging bekommen, sondern auch ein verteiltes Transaktionslog. Dieses Log zahlt auf eine wichtige Qualitätsmetrik der verteilten Architektur ein: Mean-Time-to-Repair/Recover. Über das Kafka-Transaktionslog ist eine Zeitreise in die Vergangenheit möglich, um so effizienter im Fehlerfall einen konsistenten Systemzustand herzustellen.

Finale Lektion: Lessons Learned

Ich möchte nun zur letzten Lektion kommen: der Zusammenfassung. Was hat die „Welt, bleib wach!" GmbH bei der Migration auf die SCS-Architektur gelernt, was ist wichtig, was funktioniert, was würde im nächsten Projekt anders gemacht werden?

Im Wesentlichen waren die wichtigsten Punkte aus dem Migrationsvorhaben die Ergebnisse aus Lektion #1 und Lektion #7. Ohne organisatorische Anpassungen kann die SCS-Architektur ihre größten Hebel nicht entfalten. Die organisatorischen Anpassungen bedingen auch eine Anpassung der technischen Schnittstellen. Durch Fehler bei der technischen Systemintegration können alle organisatorischen Anpassungen wieder zunichte gemacht werden. Zu viele synchrone Schnittstellen erhöhen das Risiko, letztendlich doch nur einen verteilten Monolithen zu bauen. Eine Landschaft aus SCSs ist eine verteilte Landschaft. Verteilte Systeme bedeuten auch potenziell verteilte Probleme. Die Fehleranalyse und Wartung können deutlich komplexer werden als bei einem Single-Point-of-Failure-Monolithen. Infrastrukturkosten müssen im Auge behalten werden. Der Monolith wird in Systeme zerlegt, die wiederum aus einzelnen Services bestehen können. Jeder einzelne Service braucht Rechenzeit, Speicher usw. In der Summe werden nach der Migration deutlich mehr Ressourcen benötigt als für den Monolithen. Das ist nicht verwunderlich, da beispielsweise für jeden Service das Spring Framework in eine Java VM geladen wird, und genau diese Redundanzen kosten Speicher und damit auch Geld. Die verteilte Infrastruktur bringt für das Thema Datensicherheit ganz neue Anforderungen und Probleme mit sich. Es lohnt sich, diese Aspekte schon in einer frühen Projektphase zu adressieren.

Domain-Driven Design ist ein Evergreen. Die fachliche Denkweise hinter DDD ist eine sehr gute Hilfe, um technische und fachliche Strukturen eng aneinander zu binden und so Kommunikations- und Verständnisprobleme zwischen Stakeholdern und Projektteams zu reduzieren und alle Beteiligten an einem Strang ziehen zu lassen. Event Storming ist ein gutes Hilfsmittel, um in einer komplexen Welt Ursache-Wirkungs-Ketten zu identifizieren und zu verstehen. Der daraus entstehende, reaktive und asynchrone Ansatz bringt eine ganze Fülle an Nachrichten und damit verbunden Nachrichtentypen hervor. Diese Nachrichten sind Teil des API und sollten mit entsprechender Sorgfalt spezifiziert werden. Bei der Implementierung der verteilten Welt muss viel Wert auf Resilience gelegt werden. Ein einzelnes, nicht verfügbares System darf nicht die komplette Systemlandschaft beeinträchtigen. Für die Nachrichtenverarbeitung empfiehlt sich Pull-Messaging in Verbindung mit Back-Pressure, wie es etwa mit Apache Kafka sehr einfach zu implementieren ist. Die Konsequenz der verteilten Architektur ist die systemübergreifende Eventual Consistency, eine Erfahrung, die für Anwender eines monolithischen Systems neu sein dürfte. Mit der wichtigste Qualitätsfaktor dieser verteilten Systemarchitektur ist die „Mean Time to Recover/Repair". Ein verteiltes System lässt sich nicht so einfach komplett neu starten wie ein Monolith. Für ein Desaster-Recovery kann es notwendig werden, Stammdaten komplett neu zwischen Systemen zu synchronisieren. Die Laufzeit ist abhängig vom Datenvolumen und kann durchaus Stunden bis Tage dauern.

Trotz aller Fallstricke kann auf jeden Fall gesagt werden, dass die SCS-Architektur ein guter Ansatz für die Produktentwicklung ist. Mit dem verteilten Ansatz hinter der SCS-Architektur können Organisationen gut skaliert werden. Die SCS-Architektur lenkt den Fokus auf das System/das Produkt und auf dessen Schnittstellen. Auch lässt sich sagen, dass die SCS-Architektur mit Brown-Field-Migrationsprojekten funktioniert, auch wenn an dieser Stelle das Risikomanagement noch intensiver erfolgen sollte als bei gewöhnlichen IT-Projekten.

Die Erfahrungen mit dem SCS-Ansatz sind durchaus positiv. Wichtig zu betonen ist der agile Grundsatz: Es ist besser, auf Veränderungen zu reagieren, als einen strikten Plan zu verfolgen. Ein Migrationsprojekt kann nicht von Anfang bis zum Ende komplett durchgeplant werden. Es werden immer Probleme auftauchen, und auf diese Probleme muss sachgemäß und zeitnah reagiert werden. Das SCS-Manifest ist in diesem Kontext als Leitplanke zu verstehen. Leitplanken bieten klare Abgrenzungen, sollten jedoch auch verschoben oder komplett abgebaut werden, etwa wenn eine breitere Straße benötigt wird oder gerade ein Schwertransporter den Weg passiert. Das soll natürlich keine Ausrede für Chaos sein, sondern mehr eine Aufforderung, allzu dogmatische Vorgehensweisen mit einem kritischen Auge zu betrachten.

In diesem Sinne möchte ich mich für die Aufmerksamkeit bedanken. Für Rückfragen, Anmerkungen und Kritik stehe ich natürlich gerne zur Verfügung. Vielen Dank!

Mirko Sertic
mirko@mirkosertic.de

Mirko ist Software Craftsman im Web/eCommerce-Umfeld. In Funktionen als Software-Entwickler, Architekt und Consultant in Projekten in Deutschland und der Schweiz sammelte er Erfahrungen mit einer Vielzahl von Frameworks, Technologien und Methoden. Heute arbeitet er als IT-Analyst bei der Thalia Bücher GmbH in Münster mit Schwerpunkt auf Java, eCommerce sowie Such- und Empfehlungs-Technologien. Seine Freizeit verbringt er mit seiner Familie und hin- und wieder mit Open-Source-Projekten.

Jetzt Anmelden!

Sei dabei

Go DevOps 2019

2.+3. Sept. 2019 in Berlin

DOAG

godevops.doag.org

esentri

IT'S IN ALL OF US *TO CREATE*

Jetzt bewerben unter
career@esentri.com

#inallofus
www.esentri.com