Sonderdruck aus OBJEKTspektrum 05/2017

Vom Monolithen zu Microservices

Reducio!

Cloud, Microservices und Container sind Hype-Themen im Bereich der IT-Architektur. Artikel zu Vor- und Nachteilen und Beispiele für kleine „Grüne Wiese“-Projekte gibt es zuhauf. Aber wie stellt sich die Transformation einer großen und lange gewachsenen Systemlandschaft in der Realität dar? Der Artikel fasst Erfahrungen, die bei der Modernisierung eines Portalsystems eines Industrieunternehmens gewonnen wurden, zusammen.

Im Folgenden geben wir einen Einblick in unsere Zusammenarbeit mit einem bekannten deutschen Industrieunternehmen im Bereich der Digitalisierung und der bisher erzielten Ergebnisse. Dabei zeigen wir, wie wir die Architektur der Systemlandschaft in mehreren Phasen verbessert haben:

- Zunächst wurde das vorhandene Kundenportal um ein weiteres System ergänzt, in dem wir *neue Funktionen modular realisiert* haben.
- In der zweiten Phase haben wir die *Zerlegung des alten Portals* in Angriff genommen und die ersten Microservices etabliert.
- Inzwischen sind *Microservices ein bewährtes Architekturprinzip* und bilden die Basis für die Optimierung der Systemlandschaft.

Was tun gegen einen Monolithen? Einen zweiten bauen, oder?

Unser Kunde betreibt seit einigen Jahren ein Portal für Endkunden und stellt mobile Apps bereit, die auf eine gemeinsame Benutzerbasis zugreifen. Gestartet als allgemeines Kundenportal fokussierte sich dessen Entwicklung immer mehr auf produktspezifische Features, sodass Produktanläufe bald die Taktung bestimmten; in Konsequenz wurden große Kundengruppen vernachlässigt.
2014 haben wir deshalb mit den verantwortlichen Fachbereichen begonnen, ein separates Portalsystem aufzubauen mit dem Schwerpunkt „After Sales“, welches sich neben neuen Funktionen vor allem auszeichnen sollte durch:

- gute Performanz,
- hohe Robustheit und Verfügbarkeit,
- einfache Erweiterbarkeit,
- Anpassbarkeit an die Bedürfnisse weltweiter Handelspartner.

Als Plattform wurde aus Konzernräson eine etablierte kommerzielle Portalplatt-

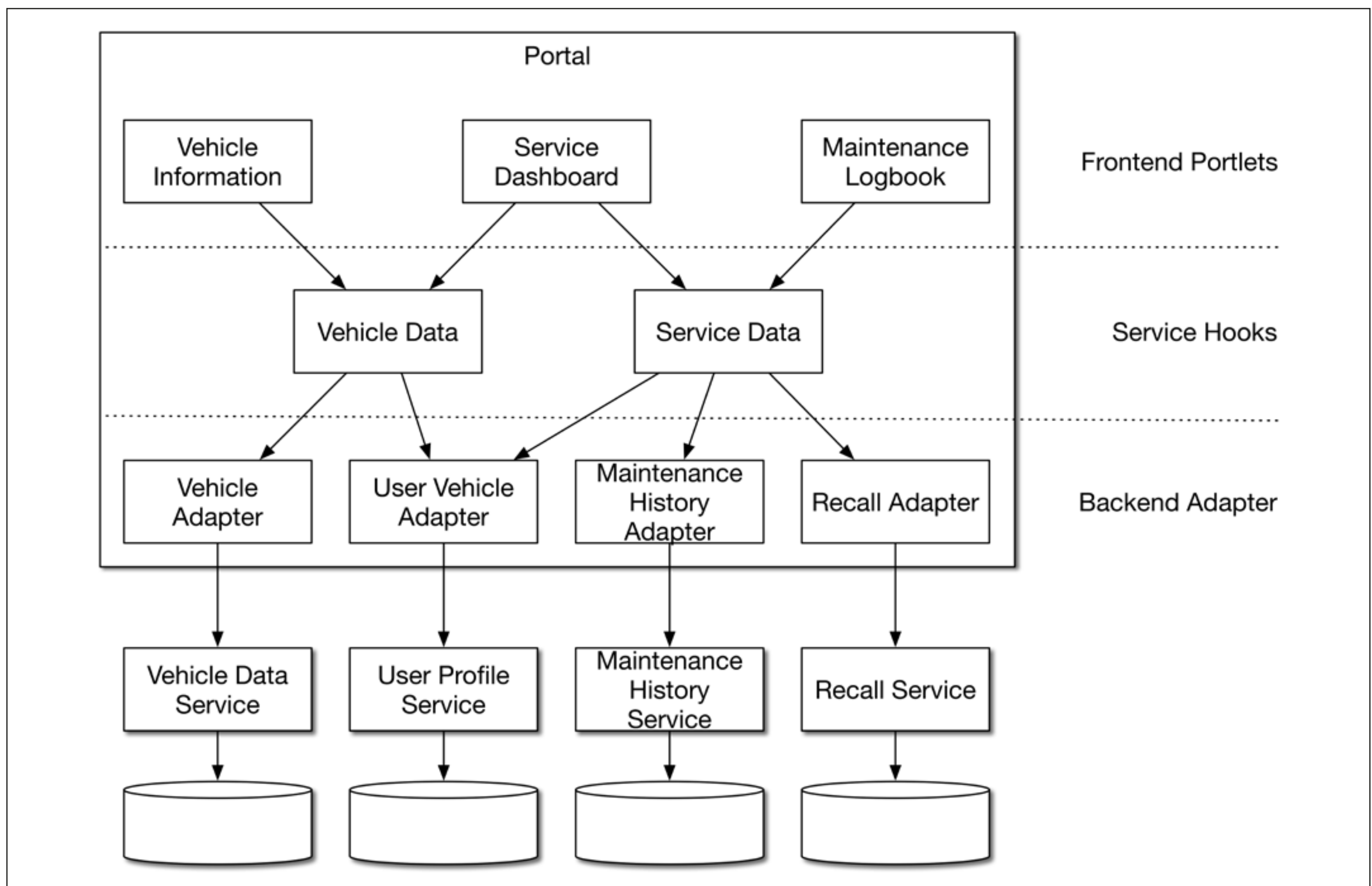

Abb. 1: Architektur aus Frontend-Portlets, Service-Hooks und Backend-Adaptern

form gewählt, die den Portlet-Standard [JSR286] implementiert. Dieser Standard verführt allerdings leicht zum Vernachlässigen jeglicher Schichtentrennung – auf der Website des Herstellers und in einschlägigen Foren werden teilweise Portlets propagiert, die Darstellung, Geschäftslogik und Backend-Anbindung ohne Modularisierung enthalten.

Modularer Aufbau des Portals

Abbildung 1 zeigt die von uns gewählte Architektur aus Frontend-Portlets, Service-Hooks und Backend-Adaptern:

- *Frontend-Portlets* sind für die Darstellung von Informationen im Portal verantwortlich. Sie enthalten lediglich triviale Logik, verkürzen beispielsweise Texte zur Darstellung.
- *Service-Hooks* enthalten die Geschäftslogik und kümmern sich um Aufbereitung und Transformation von Daten. Sie sind fachlich geschnitten und nutzen gegebenenfalls mehrere Backend-Adapter.
- *Backend-Adapter* binden Backends per SOAP oder ReST (JSON) an, bereiten die Daten auf und stellen diese strukturiert den höheren Schichten zur Verfügung.

Dieser Ansatz ermöglichte es, in einem verteilten Team effizient zu entwickeln und beispielsweise Design-Änderungen fast ohne Seiteneffekte umzusetzen.
Doch mit dem Essen kommt der Appetit.

Wachstum

Angespornt durch den erfolgreichen Live-Gang des neuen Portals und die hohe Qualität bei schnellen Fortschritten in der Entwicklung wuchsen die Wünsche der Fachbereiche an die anzubietenden Funktionen. Diese Funktionen erforderten zunehmend die gleichen Produktdaten und es zeichnete sich ab, dass wir bald damit beschäftigt sein würden, Code-Duplikate in unseren Service-Portlets zu pflegen.
Gleichzeitig begrenzte die Portalplattform durch hohe Turnaround-Zeiten für Coding und Testing unsere Effizienz; die fehlende Modularisierung des Plattform-Codes machte es erforderlich, sehr große Code-Fragmente zu erweitern, die entsprechend unübersichtlich wurden.
Die Herausforderung gingen wir an, indem wir eine neue Schicht einzogen. Implementiert wurden die Komponenten in einer eigenen Code-Bibliothek, der „Commons Library“, die unabhängig von der Portalplattform funktionieren sollte. Dazu war es notwendig, bisher genutzte Plattformfunktionen für Protokollierung, Konfiguration und Caching zu abstrahieren und die konkrete Implementierung injizieren zu lassen.
Ein wichtiges Paradigma beim Design der Commons Library war die Aufteilung in fachliche Domänen wie „Kunde“, „Produkt“, „Service“ und „Markt“. Dieses Paradigma – bekannt als „Domain-Driven Design“ [Eva03] – hat alle folgenden technischen Fortschritte überstanden und ist nicht nur für uns eine echte Best Practice.
Durch diese Maßnahmen blieb das Entwicklungstempo bei einem wachsenden Umfang annähernd konstant. Ein wesentliches Problem blieb jedoch ungelöst: Unser Portal war zwar robust und performant, doch lagen Anmeldung und Startseite im alten Portal, weshalb das Benutzererlebnis in Summe schlecht war. Login-Zeiten von 20 bis 60 Sekunden und schlechte Verfügbarkeit sind inakzeptabel.

Den Moloch schneiden – und zwar vertikal!

Zusammen mit dem führenden Fachbereich setzten wir die Verbesserung der Anmeldung auf unsere Agenda. Der verlockende Ansatz wäre gewesen, den Login-Prozess in unser neues Portal zu verlagern, doch konnten wir unseren Kunden von den damit einhergehenden Nachteilen überzeugen:

- *Abhängigkeit* des übergreifenden Logins von einem fachlich geprägten Portal, welches keinerlei Fokus auf die Weiterentwicklung übergreifender Funktionen legt,
- potenzielle *Sicherheitsprobleme* durch Realisierung kritischer Funktionalität auf einer sehr mächtigen, teilweise intransparenten und in der Vergangenheit anfälligen Plattform,
- *mäßige Performanz* durch Aufsetzen auf eine Plattform, deren Funktionalität zwar nicht genutzt wird, deren Mächtigkeit aber zu Geschwindigkeitseinbußen führt.

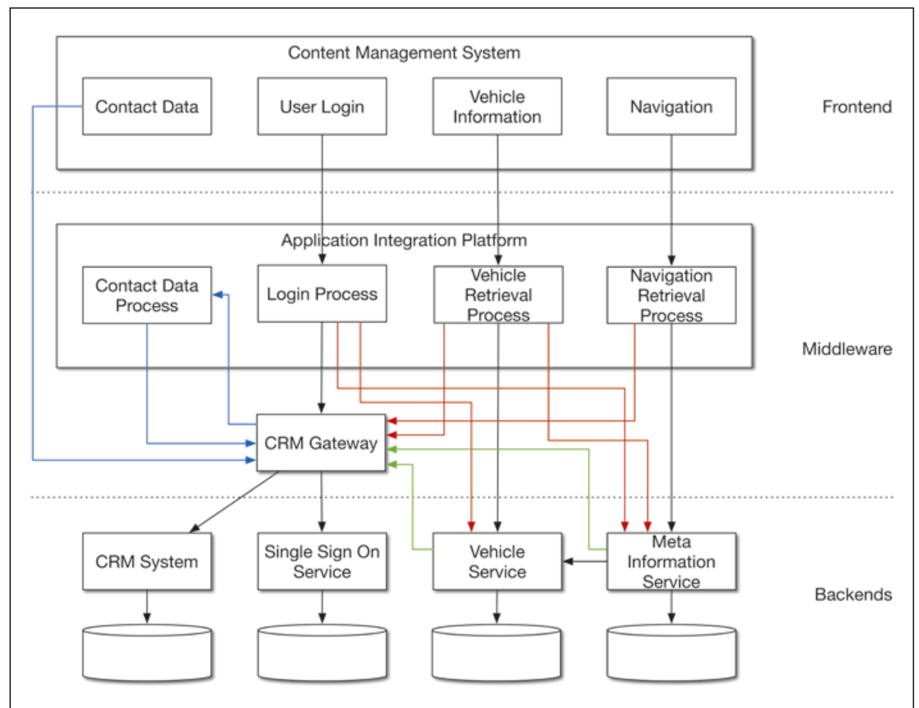

Abb. 2: Der Moloch – über Jahre gewachsene Ausgangsarchitektur

Der Moloch

Vorgefunden haben wir die über Jahre gewachsene Architektur, deren Charakteristika in **Abbildung 2** dargestellt sind. Die Abbildung ist stark vereinfacht und abstrahiert, die tatsächliche Landschaft ist wesentlich größer, verschachtelter und natürlich vertraulich. Ursprünglich vielleicht zugrunde liegende Architekturkonzepte lassen sich heute nicht mehr erkennen, deshalb lässt sich die Frage „Warum ist XY so gebaut worden?“ nicht beantworten.
In der Abbildung fallen folgende Schwächen auf:

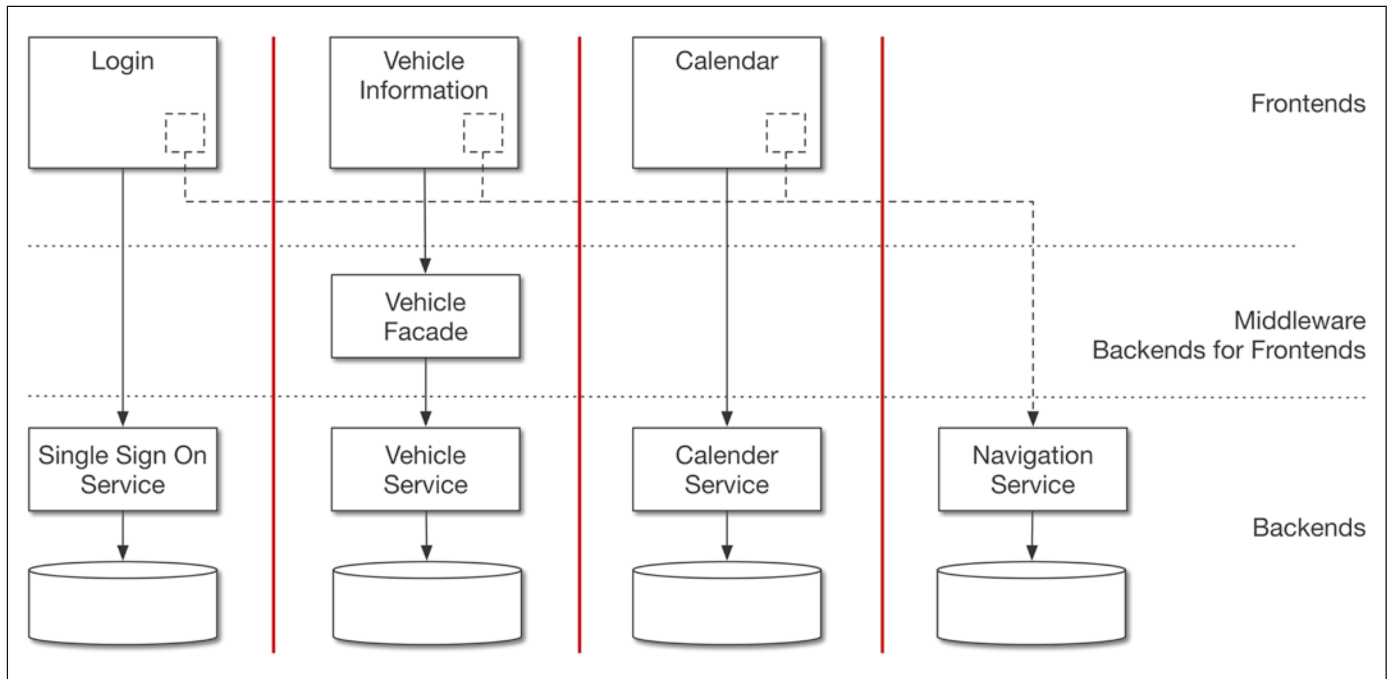

Abb. 3: Unterschied zu horizontalen Schnitten

- Fachliche Funktionen und Systeme sind unterschiedlich geschnitten und Daten aus mehreren Domänen werden unnötig vermengt (rot dargestellt).
- Es gibt keine klare Schichtentrennung, beispielsweise wird ein zunächst als CRM Gateway konzipiertes System von ganz unterschiedlichen Systemen angebunden (grün dargestellt).
- Zwischen einigen Systemen bestehen zyklische Abhängigkeiten (blau dargestellt).
- Kommunikation ist aufgrund der vielen Abhängigkeiten kaum nachvollziehbar.
- Das CRM Gateway bildet einen Single-Point-of-Failure.

Nicht sofort erkennbar ist die Problematik, dass die Geschäftslogik vorwiegend in der Integrationsplattform implementiert wurde. Während technische Integrationslogik dort sinnvoll angesiedelt ist, schränkt Geschäftslogik im Enterprise Service Bus (ESB) die Skalierbarkeit ein, mindert die Flexibilität und verursacht Governance-Probleme (siehe z. B. [Fow08]).

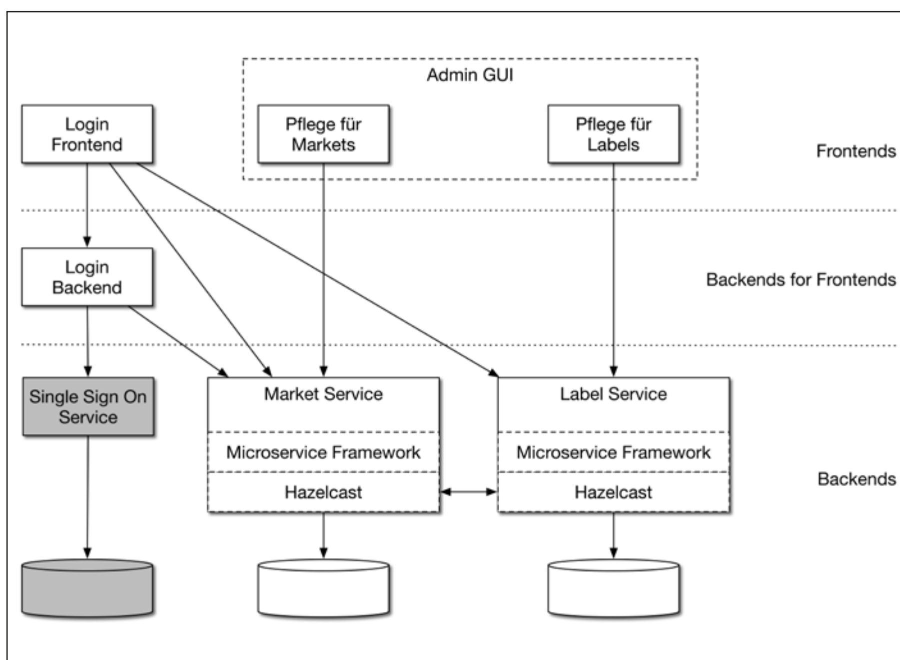

Abb. 4: Wesentliche Bestandteile der Software

Nicht dargestellt ist ein weiteres Muster, nämlich dass Nutzerdaten und Zuordnungen von Produkten zu Nutzern mehrfach redundant gespeichert werden und sowohl die Middleware als auch die redundanten Systeme Synchronisationslogik enthalten.

Ein neues Paradigma ...

Für die neue Login-Anwendung konnten wir ein neues Architekturprinzip durchsetzen: Anstelle einer opportunistischen Zuordnung von Funktionen zu Systemen tritt ein vertikaler Schnitt, der fachlich zusammengehörige Funktionen auf Systeme innerhalb einer Domäne abbildet.
Die neue Anwendung sollte eine klar abgegrenzte Funktionalität bieten:

- Bestehende Benutzer können sich anmelden.
- Bestehende Benutzer können ihr Kennwort zurücksetzen.
- Neue Benutzer können sich registrieren.
- Neue Benutzer können ihren Account aktivieren.
- Angemeldete Benutzer erhalten eine von Land und Sprache abhängige Startseite.

Hinzu kamen natürlich nicht-funktionale Anforderungen vor allem an die Verfügbarkeit und Performanz.
Aufgrund des klaren Funktionsumfangs schied eine Portalplattform oder ein CMS (Content-Management-System) aus und wir entwarfen eine Webanwendung auf

Basis von Spring Boot [SPRI] und AngularJS [ANGU].
Den Unterschied zu horizontalen Schnitten zeigt **Abbildung 3**. Jede Vertikale ist im Idealfall eigenständig und bringt Frontend, Services, Admin-Oberfläche und Persistenz mit. Einheitlichkeit im Frontend wird durch gemeinsame UI/UX-Komponenten realisiert.

... führt zu Microservices

Unsere Anwendung für den Login stellt Daten aus zwei Domänen bereit:

- Domäne *Kunde:* Nutzerdaten, zugeordnete Produkte und individualisierte Navigationsinformationen.
- Domäne *Portal:* Gemeinsame Basisfunktionen wie verfügbare Land-Sprach-Kombinationen und Übersetzungen.

Informationen zu verfügbaren Ländern und Sprachen sind genau wie Übersetzungen nicht spezifisch für den Login. Deshalb ist es sinnvoll, diese eigenständig zu implementieren. Mit Nutzerdaten verhält es sich zwar ähnlich, doch ist es in der heutigen Architektur noch nicht möglich, diese geschützt übergreifend bereitzustellen. Wir arbeiten gerade an der Transformation der Systemlandschaft, welche das ermöglichen wird.
Zu den Funktionen für Endkunden kommen noch Funktionen zur Pflege von Ländern, Sprachen und Übersetzungen. Dazu gibt es ein Pflege-GUI, das Redakteuren nach Authentisierung zur Verfügung steht.
Die wesentlichen Bestandteile der Software zeigt **Abbildung 4**. Es sind dies:

- *Login-Backend:* Stellt Konfiguration und Nutzerdaten bereit und führt die Anmeldung des Nutzers am Single-Sign-on-Backend durch; basiert auf Spring Boot.
- *Login-Frontend:* Stellt Anmeldung und Startseite bereit, bindet Formulare ein; basiert auf AngularJS.
- *Label-Service:* Microservice, der Übersetzungen bereitstellt; basiert auf Spring Boot und unserem Microservice-Framework.
- *Market-Service:* Microservice, der Informationen zu unterstützten Ländern und Sprachen bereitstellt; basiert auf Spring Boot und unserem Microservice-Framework.
- *Microservice-Framework:* Bündelt querschnittliche Funktionen zu Monitoring, Logging, Security, Caching u. a.
- *Hazelcast:* In-Memory-Grid, realisiert einen verteilten, hochperformanten Cache über die auf einem Cluster laufenden Services.
- *Admin-GUI:* Erlaubt den Redakteuren die Pflege der Daten in Market- und Label-Service; nutzt AngularJS und Google Material Design.

Dem *Microservice-Framework* kommt eine wichtige Bedeutung zu, weil uns dieses erlaubt, die einzelnen Services schnell und kostengünstig zu entwickeln. Auch wenn das Microservice-Paradigma vorsieht, dass jedes Team seinen Technologie-Stack selbst wählt, steigert eine gemeinsame Basis die Effizienz. Wichtig ist dabei, dass das Framework keine wechselseitigen Abhängigkeiten zwischen den Services schafft und man die Code-Basis eines „verteilten Monolithen“ erhält. Deshalb streben wir auch nicht an, das Microservice-Framework über mehrere Projekte hinweg einzusetzen und übergreifend zu pflegen.

Mehrwert

Die Architektur des Login mit einer Single-Page-Applikation auf Basis von ReST-Microservices schlug Wellen. Vor allem wurden Bedenken laut, dass das heutige Data Center für eine solche Architektur nicht vorbereitet sei. Diese Bedenken sind zwar immer noch berechtigt, doch konnten wir zuerst den Fachbereich und inzwischen auch den Applikationsbetrieb davon überzeugen, dass das Festhalten an traditionellen Architekturen aus Rücksicht auf manuelle Betriebsprozesse zu Stillstand führt. Und obwohl unsere Lösung für den Betrieb in einer Cloud-Infrastruktur prädestiniert ist, gelingen Deployments im Data Center des Kunden reibungslos und die Anwendung läuft stabil.
Im Zuge des Live-Gangs unserer Login-Anwendungen haben wir in enger Zusammenarbeit mit weiteren Entwicklungspartnern und Agenturen über alle Portale ein einheitliches und verbessertes Design-Konzept umgesetzt. Der Endnutzer profitiert nun von folgenden Verbesserungen:

- deutlich *höhere Verfügbarkeit*,
- *10-mal kürzere Anmeldezeit* gegenüber der bereits optimierten Anmeldung im Altportal,
- *Anmelde- und Startseite responsiv* und komfortabel auf Smartphones nutzbar,
- *durchgängige Navigation* über alle Frontends.

Eine echte Microservice-Architektur ...

Das im Login erprobte Konzept der Microservices bildet seit Monaten die Basis für eine Erweiterung der bisherigen Systemlandschaft. Weitere querschnittliche Funktionen wurden in Microservices ausgelagert und neue Features als solche erstellt. Passend zu den Backend-Services wurden auch die Web-Frontends in „Micro-Frontends“ geschnitten. Gemeinsame UX-Komponenten bilden die Basis für eine erweiterbare, modulare Architektur im Frontend.
Ein weiterer Treiber für die Umgestaltung ist der Wunsch, in unserem Portal vorhandene Informationen aus dem Bereich „Service/After Sales“ in der mobilen App verfügbar zu machen. Dazu ist es notwendig:

- Informationen per *Schnittstellen bereitzustellen,* anstatt diese nur im Frontend des Portals auszugeben,
- Schnittstellen über ein *gemeinsames Gateway* verfügbar zu machen.

Die Bereitstellung der Informationen erfolgt als ReST-Ressourcen durch Microservices, welche durch die früher eingeführten fachlichen Domänen strukturiert werden. Unser Microservice-Framework bildet die zuverlässige Basis und wird entlang der neuen Anforderungen stetig weiterentwickelt und optimiert.
Dank der frühen Modularisierung des Portals mithilfe der „Commons Library“ können wir die Geschäftslogik leicht aus dem Portal in die Microservices überführen und müssen in den Frontends lediglich die Adapter anpassen.
Durch die Bereitstellung sowohl unserer ReST-Ressourcen als auch der über den vorhandenen ESB verfügbaren Schnittstellen über ein gemeinsames Gateway werden weitere Verbesserungen möglich:

- Apps und Web-Frontend nutzen die gleiche Geschäftslogik.
- Web-Frontends können *direkt auf Daten zugreifen*, statt diese über ein CMS oder Portal schleusen zu müssen; damit können diese „Durchlauferhitzer“ entfallen.
- Apps und Web-Frontend können *einheitlich auf Daten zugreifen.*

Die resultierende Ziel-Architektur fokussiert also nicht mehr ein Portal oder CMS, sondern stellt fachliche Daten und Funktionen in den Mittelpunkt. Diese können von unterschiedlichsten Frontends genutzt werden.

... ist Cloud-ready!

In den letzten Monaten wurden zunehmend nicht-funktionale Anforderungen an die Systemlandschaft gestellt:

- *Dezentralisierung*, also die Verteilung von Funktionen in Länder und Regionen,
- *bessere Skalierbarkeit*, um die Nutzerbasis vergrößern zu können,
- *höhere Verfügbarkeit* der Systeme,
- *kürzere Release-Zyklen* für neue oder verbesserte Funktionen.

Diese Anforderungen bedeuten unter anderem, dass Systeme einfacher und schneller operativ gebracht werden. Dies wiederum setzt voraus, dass der Automatisierungsgrad steigt. Hochautomatisierte, skalierbare Infrastruktur wird meist mit Cloud assoziiert und so verwundert es nicht, dass Technologien und Methoden Einzug halten, die typisch sind für eine Cloud-basierte Infrastruktur.
Hier spielt unsere Microservice-Architektur weitere Stärken aus, weil diese bereits Cloud-ready sind und sich einfach auch bei etablierten Providern betreiben lassen.

Fazit

Moderne Architekturkonzepte wie Microservices, Domain-Driven Design und Cloud sind praktikable Instrumente zur Gestaltung wartbarer Systeme. Eine schrittweise Transformation gewachsener Systemgeflechte ist möglich und kann in jedem Schritt nachweisbaren Kundennutzen schaffen.
Das gezeigte Beispiel ist real und erfolgreich – auch wenn wir in der Darstellung den Verlauf geglättet und kleinere Um- und Irrwege weggelassen haben.
Nicht eingegangen sind wir auf die Begeisterung und Motivation, die das komplette Team aus dem Umbau schöpft. Anstelle in der „Regressionshölle“ zu versuchen, die bestehende Funktionalität zu beherrschen und mit neuen Funktionen zu kämpfen, konzentrieren wir uns (wieder) darauf, Kundennutzen zu schaffen!

Neue Herausforderungen

Mit neuen Architekturen und Technologien wachsen die Möglichkeiten und lässt sich das Entwicklungstempo steigern. In unserem Projekt wurden dadurch andere Bereiche zu offensichtlichen Flaschenhälsen wie etwa Freigaben und Deployments im Data Center. Um diese Prozesse im Sinne des DevOps-Ansatzes zu verbessern, haben wir gemeinsam mit den Betriebsverantwortlichen ein Teilprojekt aufgesetzt. Ziele sind beispielsweise robustere Konfiguration, schnellere Deployments, nützlichere Dokumentation und übergreifendes Monitoring. Die ersten Ergebnisse sind vielversprechend, aber es gibt noch weitere Verbesserungspotenziale.
Auch für die Architektur der Systemlandschaft stehen weitere Ausbaustufen an:

- Domänen-übergreifende *Backends weiter zerlegen*.
- *Kommunikation asynchron* gestalten, wo möglich.
- *Message Broker* einführen zur Entflechtung der Backends.
- *Einheitliches, fachlich geschnittenes ReST-API* aufbauen, das durch Dritte genutzt werden kann.

Wichtig ist dabei insbesondere, dass die verteilte Natur der Landschaft erhalten bleibt und kleine wartbare Systeme nicht die Keimzelle für die nächste Generation von Moloch bilden – getreu dem Motto „Das System ist klein und beherrschbar, darum bauen wir dort alle neuen Funktionen ein“. Eine zentrale Rolle wird der Message Broker spielen, der es ermöglicht, dass Microservices auf Basis von Events Informationen austauschen, ohne das Paradigma zu zerstören durch enge Kopplung.

Wider den Opportunismus

Neben der Beherrschung von Technologien, Methoden und Vorgehensweisen ist es allerdings notwendig, diese auch in die Tat umsetzen zu wollen und zu können. Flipcharts, Whiteboards und Präsentationsfolien voll von „Man müsste“-Architekturen sind verschwendete Energie, wenn keine Ergebnisse in Form von Software erzielt werden!
Der Weg zu besserer Software kann jedoch recht steinig sein: Über „opportunistische Architekturen“, in denen Funktionen dort realisiert wurden, wo es gerade am einfachsten war, lässt sich leicht lächeln. Häufig sind sie aber den Schwächen der Organisation geschuldet und wurden von Menschen geschaffen, die trotz organisatorischer Hürden Ergebnisse erzielen wollten. Gerade in streng hierarchischen und prozessorientierten Organisationen findet man diese Muster häufig: Lieber eine Funktion bewusst falsch verorten, als monatelang nur Formulare auszufüllen und Dokumente zu erstellen. Lieber an einer schlechten Entscheidung festhalten, als die erteilte Freigabe neu zu beantragen.
Etablierte Muster zu durchbrechen, kostet Energie und manchmal auch die Sympathie bewahrender Kräfte. Deshalb braucht es Mut, Hartnäckigkeit und Durchsetzungsvermögen sowohl bei uns Technologiepartnern als auch beim Auftraggeber. Und weil keine Architektur die ultimative Lösung für alle Zeiten darstellt, müssen wir uns als Gegengewicht immer wieder selbst hinterfragen und die Architektur lebendig halten. ||

Literatur & Links

[ANGU] https://angularjs.org/

[Eva03] E. Evans, Domain-Driven Design, Takling Complexity in the Heart of Software, Addison-Wesley, 2003 (englisch, http://dddcommunity.org/)

[Fow08] B. M. Fowler & J. Webber, Keynote: Does My Bus Look Big In This?, QCon, 2008, siehe: https://de.slideshare.net/deimos/jim-webber-martin-fowler-does-my-bus-look-big-in-this

[JSR286] Java Specification Request 286: Portlet Specification 2.0, siehe: https://jcp.org/en/jsr/detail?id=286

[SPRI] https://projects.spring.io/spring-boot

Der Autor

Norman Seibert
(norman.seibert@iteratec.de)

begleitet Kunden der Automobilindustrie bei anspruchsvollen Digitalisierungs- und Technologievorhaben als Ratgeber. Sein Schwerpunkt liegt auf dem Zusammenspiel moderner Architekturen und agiler Vorgehensweisen, um die Performanz von Teams zu steigern.