Rolf Scheuch
(rolf.scheuch@opitz-consulting.com)
ist Geschäftsführender Gesellschafter und Management-Coach für IT- und Digitalisierungsstrategie. Motto: „Driving the digital transformation and establishing governance for a bimodal IT." Seine Schwerpunkte sind: Bimodale-IT und Lösungsarchitekturen der Digitalisierung.

Torsten Winterberg
(torsten.winterberg@opitz-consulting.com)
ist Digital Evangelist, Berater und Coach für „alles Neue".
Motto: „Skill, Impact, Leadership: Menschen machen den Unterschied. Mit Mut und Veränderung in die digitale Welt". Schwerpunkte: Digitale Geschäftsmodelle, Lösungsarchitekturen, Beratung.

Context-Aware Frontend Architecture (CAFA) als Basis einer digitalen Plattform

Der Druck des digitalen Wandels und in der Folge die rascher werdenden Zyklen der Anpassung und Veränderung der IT-Landschaft machen ein „Design for Change" als übergeordnetes Architekturprinzip der Systemarchitektur notwendig. Die aktuellen Herausforderungen bestehen in der konsequenten Trennung von Front- und Backend, einer klaren Delegation von Aufgaben für unabhängige Release-Zyklen und einer offenen, aber gesicherten Applikationsplattform. Um die steigende Komplexität durch Zerlegung zu beherrschen, müssen diese Herausforderungen als Leitlinien und Prinzipien über alle Architekturen der Digitalisierung in die neue Systemarchitektur einfließen. Der Artikel zeigt, wie ein Blueprint für eine Context-Aware Frontend Architecture (CAFA) als Grundlage für eine digitale Plattform aussehen kann und reißt grundlegende Konzepte für Enterprise App Store, API-Management, Backend for Frontend (BFF) und Microservices-Architekturen an.

Design for Change

Die Digitalisierung und in der Folge die rascheren Zyklen der Anpassung und Veränderung der IT-Landschaft machen ein **Design for Change** als übergeordnetes Architekturprinzip der Systemarchitektur notwendig. Die im Folgenden genannten Herausforderungen sollen als Leitlinien und Prinzipien über alle Architekturen der Digitalisierung in neue Systemarchitekturen einfließen, um die steigende Komplexität durch Zerlegung zu beherrschen:

- Technologien für aktuelle Benutzerschnittstellen und die Philosophie der Mensch-Maschine-Interaktion sind aktuell im Umbruch. Insbesondere wird die textuelle Interaktion um Gesten- und Sprachsteuerung ergänzt, Augmented und Virtual Reality (AR/VR) beginnen ihren Weg in die Business-Welt zu finden. Eine klare Prognose der schnellen Entwicklung ist nicht möglich. Eine sinnvolle Lösungsstrategie ist daher die **konsequente Trennung von Front- und Backend**, um schnell auf neue Möglichkeiten reagieren zu können, um nicht zusehen zu müssen, wie andere die neuen Möglichkeiten nutzen, während man selbst in einem starren Technologiekorsett feststeckt.
- Mit dem Internet of Everything, Big Data und Cloud-Computing steigt die Komplexität der Systeme, was eine klare Delegation von Aufgaben für **unabhängige Release-Zyklen** unabdingbar macht.
- Die traditionellen Wertschöpfungsketten ändern sich zu einem „Eco-System of value" und erfordern eine offene, aber gesicherte **Applikationsplattform**.

Betrachten wir diese drei Herausforderungen etwas näher.

Konsequente Trennung von Front- und Backend

Vor wenigen Jahren war die Oberflächenwelt noch überschaubar. Es gab Desktop-Anwendungen, webbasierte Interfaces und native Oberflächen für spezielle Devices (wie auch Smartphones). Nun aber verschwimmen diese Grenzen durch den Industriestandard HTML5 sowie neue Trends, wie etwa den Universal-Apps-Ansatz bei Microsoft. Gleichzeitig werden im Zuge einer besseren Customer Experience native Benutzerschnittstellen im mobilen Umfeld wieder wichtiger und ergänzen das universelle Web-Interface. Hinzu kommt die Entwicklung, dass die traditionelle explizite Bedienung von Oberflächen mittels Maus, Tastatur und Touchscreens sich hin zu einer eher impliziten Bedienung durch Gesten, Sprache, Augen- und Körperbewegung verändert. Dies geht einher mit den momentan vielleicht noch futuristischen Trends der 3D-Darstellung, um eine

möglichst realitätsnahe Objektdarstellung zu erhalten.

Zudem wird die klassische Kommunikation eines Clients mit genau einem Server-Backend heute immer öfter durch die Kommunikation mit mehreren unabhängigen, auch externen, Service-Providern ersetzt. Clients können durchaus auch Programmierschnittstellen (API) von unterschiedlichen Backend-Systemen, gegebenenfalls sogar von unterschiedlichen Unternehmen (z. B. Partner- oder Dienstleistungsunternehmen) nutzen. Treiber dieser Entwicklung sind die zunehmend eingesetzten Software-as-a-Service-(SaaS-)Lösungen, die zunehmende Implementierung eigener Cloud-Lösungen wie auch die Anbindung einer Vielzahl an heterogenen Endgeräten beim Internet der Dinge über entsprechende Cloud-Plattformen.

Zukünftige Oberflächen verändern sich unabhängig und schneller als die Geschäftslogik im Backend.

Flexibilität durch unabhängige Release-Zyklen

Die steigende Komplexität und die hohe Rate an Veränderung ist nicht ausschließlich ein Trend bei den Client-Systemen, sondern trifft genauso die Backend-Systeme. Der Monolith im Backend, der die erwähnte *strukturelle Zukunftsunfähigkeit* verursacht, wird zunehmend als Architekturentwurf in Frage gestellt. Der aktuell diskutierte **Microservices-Architekturansatz** unterteilt komplette Systeme in kleinere, anhand der Geschäftslogik abgegrenzte Services entsprechend ihren Business Capabilities. Ziel ist es, die Weiter- und gegebenenfalls auch Neuentwicklung voneinander zu entkoppeln, die Abhängigkeiten zu reduzieren und so das Gesamtsystem flexibler zu halten. Dies senkt letztlich nicht die inhärente Komplexität der gesamten Applikationslandschaft, allerdings wird durch die Entkopplung die Komplexität der einzelnen Komponenten reduziert. Gleichzeitig gilt: So unabhängig wie nötig, so einfach wie möglich. Das heißt auch, dass man alternative Architekturentwürfe (etwa Self-contained Systems) mit in die Überlegungen einbeziehen und die Trade-offs bewusst abwägen muss.

Um die Flexibilität und Geschwindigkeit der Produktivsetzung zu erhöhen, beginnen sich die Bereiche Entwicklung (Dev) und Applikationsbetrieb (OPs) einander anzunähern und über die DevOps-Bewegung werden durch (gemeinsam implementierte) automatisierte Prozesse die Qualität der Software (Testautomatisierung) sowie die Geschwindigkeit der Entwicklung und der Auslieferung (Build-Automatisierung) gesteigert. Die halb- oder sogar zweijährigen Release-Zyklen einer neuen Version eines Gesamtsystems sind nicht mehr opportun. Permanente Release-Fähigkeit und insbesondere auch Updates einzelner, kleiner Module sind die Zukunft in einer auf *Design for Change* ausgelegten Systemarchitektur.

Zukünftige Architekturen fördern unabhängige Innovations- und Release-Zyklen.

Applikationsplattform für ein „Eco-System of Value“

Traditionelle Wertschöpfungsketten verändern sich. Mehr und mehr Chancen ergeben sich aus der Einbindung von Geschäftspartnern – sei es um Geschäftsprozesse zu optimieren oder um die eigene Fertigungstiefe an die Marktgegebenheiten anzupassen. Hierfür muss die IT-Plattform eines Unternehmens in der Lage sein, Geschäftspartner bidirektional einzubinden. Zum einen müssen Dienste Dritter zur Reduktion der Fertigungstiefe bzw. zur Prozessoptimierung integriert werden. Zum anderen können Unternehmen Geschäftspartnern mittels entsprechender Schnittstellen einen gesicherten Zugang zu den eigenen Business Capabilities anbieten. Die Analysten von

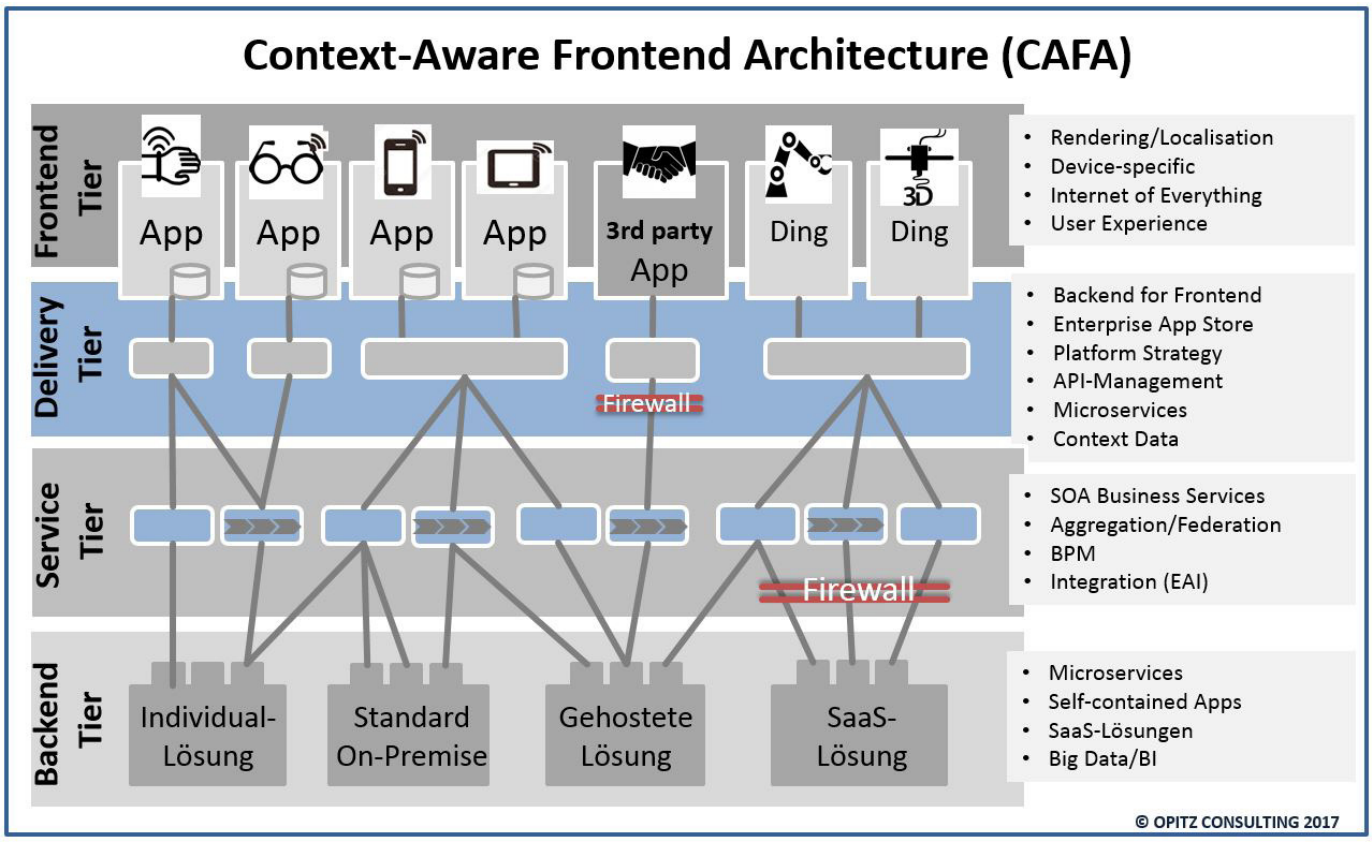

Abb. 1: Flexibilität und Omni-Channel-Strategie mittels Delivery Tier

Forrester sprechen von einem „Eco-System of Value“, das sich permanent an die Kunden- und Marktgegebenheiten anpassen lässt, Kostensenkungspotenziale erschließt und durch die Dienste von Dritten auch die Kundenbindung erhöht.

Es darf kein lähmender Applikationsstau entstehen. Neue Ansätze müssen unkompliziert und transparent implementierbar sein. Die Applikationslandschaft muss die Fähigkeit besitzen, die geschilderten Anforderungen schnell und unkompliziert zu erfüllen.

Die zukünftige eigene Systemlandschaft wird zu einer Plattform für ein „Eco-System of Value“.

Lösungsvorschlag Context-Aware Frontend Architecture (CAFA)

Die im Folgenden präsentierte Referenzarchitektur bezeichnen wir als **Context-Aware Frontend Architecture** (**CAFA**) (siehe auch Abbildung 1). CAFA erweitert die bislang etablierten drei Deployment-Schichten der Architektur um die sogenannte **Delivery-Schicht**. Hierdurch entsteht mit der Deployment-Architektur eine vierschichtige Architektur mit einer eigenen, ausgestaltbaren physikalischen Schicht für die mobile und digitale Welt.

Dieses Konzept der Architektur in 4 Schichten wurde 2015 von Forrester in seinen Analystenberichten aufgegriffen und wir haben es zu einem Blueprint der **Context-Aware-Frontend-Applikationsarchitektur** weiterentwickelt, das nun Basis unserer Beratungsleistung bei der Konzeption einer Zielarchitektur für zukunftsweisende Applikationslandschaften ist.

Das virtuelle Portal: Das Enterprise-App-Store-Konzept

Wie lassen sich die unterschiedlichen Applikationsarten zu einer zusammenhängenden Applikationswelt verbinden? Dazu stellen wir das **Konzept eines Context-Aware Enterprise App Store** vor.

Einem Outside-in-Ansatz folgend, betrachten wir die Erwartungen der Anwender an eine moderne und vor allem kontextabhängige Applikationswelt. Wie bereits ausgeführt, gehen wir davon aus, dass sich im Rahmen der Mensch-Maschine-Interaktion in den nächsten Jahren viel verändern wird – und rückblickend gesehen auch schon verändert hat. Wir greifen heute bereits selbstverständlich mittels Smartphone, Tablet, Laptop oder Smartwatch auf unseren

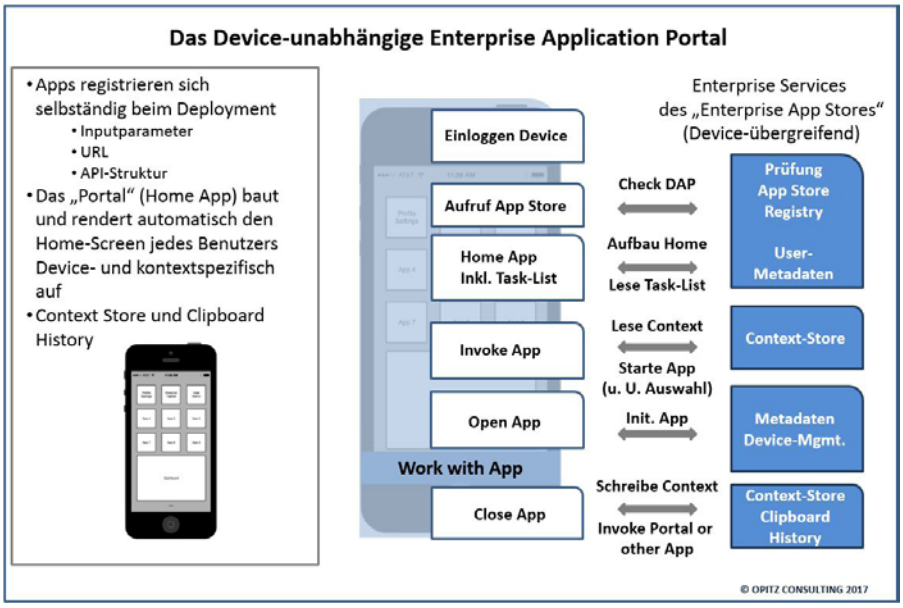

Abb. 2: Interaktion von Device und Enterprise App Store

Terminkalender zu, um nur ein Beispiel zu nennen. Mit Siri und Amazon Alexa, Cortana und weiteren virtuellen Assistenten hält nun die Sprache als weiterer Kommunikationskanal Einzug.

Wir brauchen also eine Omni-Channel-fähige Plattform für einheitliche IT-Services, die auch bei der Gestaltung der Oberflächen beachtet werden muss, denn die Nutzung von Smartphones und Tablets zum Surfen im Internet übersteigt mittlerweile die von Desktop-PCs und Laptops. Ähnlich schnell werden weitere „Zukunftstechnologien“ adaptiert werden.

Sowohl die Nutzung kleinerer Displays als auch die Stärkung des Gedankens der Software-Ergonomie führt dazu, dass die klassischen Expertensysteme immer mehr bedarfsgerechten Oberflächen weichen. Die Verwendung komplexer Personalisierungsstrategien, um Expertenoberflächen in aufgabengerechte Monolithen zu verwandeln, verliert an Bedeutung. Aber nicht nur die Oberflächen auf einem Gerät werden bedarfsgerechter, sondern es wird gleichzeitig auch das am besten geeignete Gerät für eine Aufgabe gewählt.

Diese vielen verschiedenen kleinteiligen Oberflächen müssen aber wieder integriert werden. Nutzer interessiert es nicht, wie eine Anwendung aufgeteilt ist. Im Gegenteil: Der Anwender erwartet sogar, dass er entsprechend seinem aktuellen Kontext gar nicht merkt, dass unterschiedliche Anwendungen verwendet werden. Und wenn der Kontext Wahlmöglichkeiten bietet, dann möchte man als Anwender natürlich nur unter den Applikationen wählen können, die im momentanen Kontext sinnvoll sind. Was ist aber jetzt ein Kontext? Ein Kontext kann sich aus sehr vielen Faktoren ergeben: dem Ort, an dem ich mich befinde (Location Awareness), der Anwendung, mit der ich zuletzt auf einem anderen Gerät gearbeitet habe (fachlicher Kontext), der Uhrzeit (zeitlicher Kontext), dem Vorwissen über Nutzungsgewohnheiten.

Während man in klassischen Desktop-Webanwendungen eine Session hat, die den Zustand hinsichtlich einer Anwendung beschreibt, braucht man nun einen Kontext, der geräte- und anwendungsübergreifend ist und den Anwender bei der Auswahl weiterer sinnvoller Schritte unterstützt. Auch dies erfordert wieder ein Stück „künstliche Intelligenz“. In einem ersten Schritt kann diese Funktion von einem kontextsensitiven Enterprise App Store übernommen werden, weswegen wir auch von einem Context-Aware App Store sprechen. Abbildung 2 beschreibt die Interaktion der Devices mit einem sogenannten Enterprise App Store, den wir im Folgenden näher erläutern.

Analog den App-Ansätzen bei Smartphones, sei es im Apple-Kontext oder auf der Android-Plattform, schlagen wir die Implementierung eines Enterprise-App-Store-Konzepts vor. Jede Applikation, unabhängig von ihrer Implementierung, registriert sich in einer zentralen Registry. Diese Registry ermöglicht die Authentifizierung, auch über die individuellen Mechanismen der eingesetzten Devices (Fingerabdruck, Retinascan usw.), und führt Buch über die erlaubten Funktionalitäten der Benutzergruppen und Benutzer (Autorisierung). Mithilfe der physikalischen Parameter der Devices kann die Registry auch die erlaubten Devices für eine Anwendung erkennen.

Jede App hat zudem erlaubte Input-Parameter, URL- sowie API-Strukturen (auch die REST-Strukturen für die fachliche Logik), die über ein Device-Management im Rahmen des zentralen App Stores verwaltet werden. Mit diesem Ansatz ist es dem virtuellen Portal möglich, einen Home-Screen pro Device zu erstellen, der auf die individuellen Parameter des Benutzers eingeht.

Da die App-Landschaft, im Gegensatz zu den umfassenden Expertensystemen auf dem Desktop, aus einer großen Anzahl von aufgabenspezifischen, disjunkten, kleinen Lösungen besteht, muss es möglich sein, aus einer App eine andere App aufzurufen und hierbei den Kontext zu behalten. Die Übergabe eines Kontextes erfolgt in einer ähnlichen Technologie, die z. B. Android selber verwendet. Jede App schreibt schemafrei den Kontext beim Verlassen in eine Clipboard History, die wiederum von der aufgerufenen App genutzt wird, um sich selbst zu positionieren.

Dieses App-Store-Konzept sichert, trotz aller Vielfalt der Oberflächen-Technologien, eine einheitliche Sicht durch ein virtuelles Portal. Dieses manifestiert sich als Home-Anwendung auf den unterschiedlichen Devices mit einer Beachtung der Parameter des Device-Managements im App Store. Als Folge werden etwa JavaFX-basierende Applikationen auf dem Home-Screen bei Tablets usw. nicht angeboten und umgekehrt werden auf dem Desktop keine Apps angeboten, deren Anwendung auf den Sensoren der spezifischen Devices beruht. Neue Applikationen, die die Sensorik der Devices nutzen, passen ebenfalls in das Schema, solange eine Registrierung in dem betreffenden App Store möglich ist. Dies sollte aus unserer aktuellen Sicht keine Einschränkung bei der Einbindung zukünftiger Technologien sein.

Nachteil bei der Umsetzung des Enterprise-Store-Konzepts ist der starke Anstieg der Anzahl an kleinen Apps, wodurch Benutzer schnell den Überblick verlieren können. Da User Experience ein extrem starker Treiber für Digitalisierungslösungen darstellt, benötigen wir noch einen weiteren Schritt zur angestrebten Lösung: Wir erweitern den Austausch von Kontext zwischen den Apps um die Nutzung externer Kontexte aus der Welt der Dinge. Beispiel: Ein Monteur in einer Fabrikhalle, der sich mit einem Tablet in der Nähe von Maschine A aufhält, bekommt auch nur die Apps angezeigt, die im Kontext der Maschine A sinnvoll zu nutzen sind. Status-Displays passen sich spezifisch an, startbare Prozesse ändern sich, Aufgabenlisten filtern sich neu. Kommt der Monteur in den Kontext von Maschine B, erfährt er eine völlig andere Experience auf der UI, basierend auf den Kontextinformationen. Die App-Vielfalt wird sinnvoll beherrschbar, trägt zu besseren User-Erfahrungen bei und erlaubt zudem die gewünschte Flexibilität durch Erstellung vieler kleiner, passgenauer Frontends mit innovativen Architekturen, die durch Backend for Frontends (BFFs) und APIs von den stabilen Backends getrennt sind. Diese beiden Techniken werden in der Folge noch ausführlicher vorgestellt.

Delivery Tier

Die Schwachstelle beim Deployment der etablierten logischen 3-Schichten-Architekturen liegt in der mangelhaften und zu schwerfälligen Unterstützung der Möglichkeiten unterschiedlicher mobiler Devices und deren eingebauter Sensorik sowie einer fehlenden Erweiterung auf die Anforderungen des Internets der Dinge (IoT) bei der Interaktion mit *Dingen*. Diese etablierte 3-Schichten-Architektur stellt eine Plattform für Business Services zur Verfügung und abstrahiert somit die Backend-Dienste von einer spezifischen Oberfläche. Eigentlich die richtige Denkweise! Diese Services sind oft als Web-Services auf einer SOA-artigen (Service-Oriented Architecture) Infrastruktur implementiert und nutzen einen Enterprise Service Bus, der auf einer eigenen *Service Tier* implementiert ist.

Betrachtet man nun die Anforderungen aus der Sicht der Frontends, d. h. die User Experience als den entscheidenden Faktor, verändert sich die Einschätzung. Die unterschiedlichen Clients benötigen Daten mit unterschiedlichen Aggregationsstufen, verbinden Daten aus unterschiedlichen, auch externen Services zu neuen Informationsobjekten, nutzen Public APIs oder Device-spezifische Möglichkeiten, die den bestehenden Business Services nicht bekannt sind. Bleibt man dem Konzept der 3-Schichten-Architektur treu, müssen die Business Services zeitnah auf die spezifischen Belange der Devices verändert werden bzw. benötigte Funktionalitäten über neue zusammengesetzte *Composite Services* aus bestehenden Services abgebildet werden. Dies wird in der Folge das Paradigma der SOA ein Stück weit aushebeln, da Business Services sich in Richtung von Client-spezifischen Schnittstellen entwickeln bzw. neue Client-spezifische Services geschaffen werden. Dies schränkt somit den Wiederverwendbarkeitsgrad solcher Services ein – siehe dazu auch unser Modell der Service-Kategorien. Der dadurch vorhandene stetige Druck, bestehende Services anpassen bzw. neue implementieren zu müssen, bremst diese Innovationen auf Ebene der Oberflächen aus.

Aktuell wird diese Herausforderung auf der Frontend-Seite häufig umgangen, indem die fehlende Logik *einfach zusätzlich* in den Frontends implementiert wird. Dies führt wiederum zu komplexen und auch aufwendigen Lösungen für *einfache* Frontends und oft zu Widersprüchen, die durch die redundante Implementierung von Teilen der Geschäftslogik entstehen.

Aus architektonischer Sicht benötigen wir deshalb eine eigene physikalische Schicht – die *Delivery Tier*. Die Oberflächen und Funktionalitäten können durch die Entwicklung auf das gewünschte Device optimal abgestimmt werden, ohne die unterlagerten stabilen Business Services permanent zu verändern. Mit dem Anstieg der Anzahl und der Möglichkeiten unterschiedlicher Devices, der rasanten Entwicklung von HTML5 bzw. den Entwicklungsumgebungen für die Devices sind die Innovationszyklen deutlich höher als bei der Veränderung von Business Services für die Backend-Prozesse. In einer Welt, die komplett aus Microservices oder Self-contained Systems besteht, mag ein anderer Blick auf diese Prozesse vorherrschen. Aber wo finden wir schon reine „Grüne-Wiese-Szenarien“ vor?

Die Delivery-Schicht verträgt sich zudem sehr gut mit dem Konzept des API-Managements, wodurch Skalierung und Security von (mobilen) Lösungen ermöglicht werden. Gerade in der mobilen Welt ist die Bandbreite noch immer ein Hemmschuh: Schwergewichtige XML-basierte Web-Services sind in diesem Zusammenhang nicht das Mittel der Wahl. Hier werden leichtgewichtige Services mit simplen Protokollen und intuitiven APIs benötigt. Aktuell nutzen mobile Lösungen oft RESTful-JSON-Dienste (REST), die auf HTTP(S)-Protokollen basieren. Blickt man in die IoT-Zukunft, werden sich vielleicht weitere Protokolle, etwa CoAP oder MQTT, durchsetzen. Darüber hinaus besteht die Chance, einen Teil der Delivery Tier in eine „entmilitarisierte Zone“ (DMZ) auszulagern, um die eigenen Backend-Prozesse externen Dritten zur Verfügung zu stellen. Hierdurch wird die eigene IT-Welt zu einer Applikationsplattform für Geschäftspartner.

API-Management

Wegen der entscheidenden Bedeutung für ein „Eco-System of Value“ möchten wir das Thema API-Management an dieser Stelle gesondert hervorheben. APIs sind das Mittel, um die eigene Geschäftslogik

internen Entwicklerteams oder auch Dritten zur Verfügung zu stellen und somit Innovation, Flexibilität und Geschwindigkeit hinsichtlich neuer Implementierungen zu erreichen. Einige bekannte Beispiele für öffentliche APIs sind die Schnittstellen von Twitter, Facebook, Apple oder auch Google. Dritte können durch Nutzung dieser gekapselten Basisdienste eigene, höherwertige Lösungen für ihre Kunden anbieten. Wir kennen jedoch auch aus der Old Economy bereits APIs oder Interfaces: Banken, Kreditkarteninstitute, Bonuskartensysteme und Versicherungen haben seit Jahren bereits Schnittstellen bereitgestellt, jedoch nur auf einem abgesicherten Netzwerk und nur für spezielle Partner. Und letztlich sind APIs ein wesentlicher Teil der geschilderten Omni-Channel Customer Experience, die sich an den Bedürfnissen der Kunden orientiert und sich somit auch permanent verändert.

Das Denkmuster der heutigen API-Welt sieht die eigene Systemwelt eher als eine Plattform im Sinne einer API-Ecomony inkl. Monetarisierung der APIs: als eigene Schicht zur Verwendung durch beliebige Consumer. Kurzum: Unternehmen öffnen sich zusehends für die Außenwelt und exponieren die eigenen Business Capabilities in Form von APIs, wodurch sich neue Geschäftsmodelle ergeben bzw. sich das komplette Kerngeschäft verändert. Unternehmen wie beispielsweise Google, Expedia oder Salesforce generieren heute einen Großteil ihrer Umsätze mit dem Anbieten von APIs für Partner und Kunden. In der Folge ist API-Management essenziell, auch weil folgende Aspekte damit umsetzbar sind:

- Security (Authentifizierung und Autorisierung, Throttling, Thread Protection, Encryption usw.);
- API-Betrieb (Aufzeichnung des Nutzungsverhaltens, Identifikation von Missbrauch/Fehlnutzung, Monetarisierung);
- API-Lifecycle-Management (E2E-Lifecycle-Management);
- Stakeholder-Management (Management- und Entwicklerportale, API-Kataloge und Dokumentation).

In Diskussionen wird häufig aufgezeigt, dass API-Management nur ein neuer Name für SOA ist. Dies ist definitiv nicht der Fall und zeigt aus unserer Sicht eine Fehlinterpretation des SOA-Architekturparadigmas auf: APIs sollen einfach und transparent für externe Consumer nutzbar sein und können potenziell leicht erweitert werden, um neuen Anforderungen gerecht zu werden. Um die Belange neuer Devices abzudecken, können auch schnell neue APIs erstellt werden. Die Schwerpunkte verlagern sich im Gegensatz zu einer klassischen SOA.

Die Services in klassischen serviceorientierten Architekturen waren auf interne Nutzer ausgerichtet. Ein wichtiger Aspekt dabei war die Effizienz: keine doppelten Arbeiten bzw. Funktionalitäten. Daher spielt in diesem Kontext der Aspekt der Komposition und Wiederverwendbarkeit eine größere Rolle. Aspekte wie Vorwärts- und Rückwärtskompatibilität von APIs wurden dagegen in der Praxis häufig nur unzureichend berücksichtigt, da dies zusätzlichen Aufwand bedeutet und man die Illusion hatte, dass man die internen Nutzer so leichter zu einem Upgrade der Schnittstelle „zwingen“ könnte. Bei externen Kunden oder Geschäftspartnern ist dies allerdings nicht so einfach möglich, da sich inkompatible Änderungen und der Zwang zu Anpassungen auf der Konsumentenseite negativ auf die Geschäftsbeziehung auswirken können.

In Abbildung 3 versuchen wir die unterschiedlichen inhaltlichen Ausrichtungen von API-Management und SOA zu verdeutlichen.

Typische SOA-Projekte lösen überwiegend Integrationsprobleme auf Ebene der Systems of Records. Hier gilt es vor allem, verschiedene Enterprise Information Systems (EIS) wie beispielsweise ein SAP System mit anderen Anwendungssystemen auf standardbasierendem Weg miteinander zu verbinden. Konsequent angewendet führt dies zu einer lose gekoppelten Systemlandschaft auf Ebene der Systems of Records. Wie in Abbildung 3 dargestellt, wird in diesem Zusammenhang auch von der **horizontalen Integration** gesprochen. Dabei zeichnen sich Integrationen dadurch aus, dass es sich zumeist um System-zu-System-Integrationen ohne menschliche Interaktionen handelt. Daher sind asynchrone Interaktionsmuster, wie Publish/Subscribe, Daten-Replikation oder Dateitransfers, eher die Regel als die Ausnahme.

Der asynchrone Aspekt ist sinnvoll zur zeitlichen Entkopplung der Systeme. Allerdings wurde diese meistens auf Basis von Batch-Verarbeitung bzw. File Exchange realisiert, also sehr schwergewichtig, und selten auf Basis von Messaging-Systemen. SOA-Services werden in diesem Zusammenhang auch dazu benutzt, um in den EIS vorhandene Funktionalitäten auf standardbasierenden Wegen im Unternehmen verfügbar zu machen. Das Ergebnis sind dann technische über XML-Schnittstellen exponierte **Integration APIs**.

Demgegenüber stellt ein API-Management leichtgewichtige, intuitive APIs zur Verfügung, um Innovation voranzutreiben, für Differenzierung vom Wettbewerb zu sorgen und neue digitale Geschäftsmodelle zu unterstützen. Beim API-Management steht demnach die Bereitstellung von **Business APIs** im Fokus, die leicht verständlich und somit von der breiten Masse nutzbar sein müssen. Da diese Business APIs in der Regel

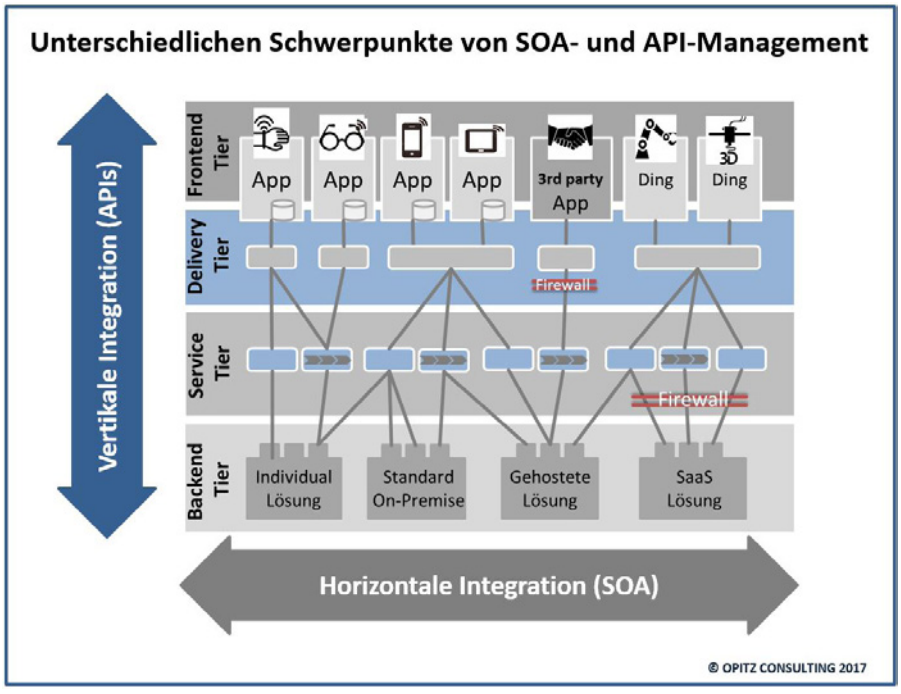

Abb. 3: Schwerpunkte von SOA und API-Management

in Richtung der Endbenutzer operieren und Backend-Services verwenden, u. a. auch auf Ebene der Systems of Record, spricht man hier, wie in Abbildung 3 dargestellt, von einer **vertikalen Integration**.

Für die über das API-Management bereitgestellten APIs sind schnelle Antwortzeiten in Echtzeit ein Muss, da diese in Richtung der Endbenutzer operieren. Anders als bei den SOA-Services sind synchrone Schnittstellen hier die Regel. Asynchrone Interaktionen, wie z. B. Push Notifications in Richtung von Mobile Apps, sind hier in der Richtung Backend-Service zu App zu finden.

Da Business APIs in der Regel in Richtung der Endbenutzer wirken und somit die Charakteristika einer solchen API sowie das Design einen direkten Einfluss auf die User Experience haben, ist es für die Entwicklung an dieser Stelle aus unserer Sicht unerlässlich, einen **API-First-Ansatz** zu verfolgen. Hierbei geht es darum, bereits im Vorfeld der App- bzw. Backend-Service-Entwicklung eine stabile API-Definition festzulegen, die den Anforderungen moderner APIs, was beispielsweise die Intuitivität, klare Strukturen oder Device-Unabhängigkeit angeht, gerecht wird.

API-Management und SOA sind also demzufolge keineswegs gleichzusetzen, haben doch beide Konzepte unterschiedliche Ausrichtungen. Bezogen auf die *Design for Change*-Systemarchitektur kommt man bei genauerer Betrachtung der Konzepte zu dem Schluss, dass beide wichtige Bausteine auf dem Weg zum „digitalen Morgen" sind.

Entkopplung vom Backend: Backends for Frontends

Wie bereits mehrfach angesprochen, werden wir in den nächsten fünf bis sieben Jahren eine grundlegende Veränderung der Mensch-Maschine-Interaktion erleben. Mobile Geräte haben unterschiedlichste Funktionalitäten und Bandbreiten. Der Versuch, diese unterschiedlichen Bedürfnisse über ein allgemeines Frontend-API zu befriedigen, bringt diverse Herausforderungen mit sich.

Ein allgemeiner Frontend-API-Layer übernimmt zwar durchaus verschiedene Verantwortlichkeiten für die unterschiedlichen Devices. Das bedeutet allerdings eine Menge Arbeit. Häufig wird daher ein eigenes Team dafür aufgestellt. Das wiederum widerspricht der Prämisse, dass Teams ihre Applikation möglichst unabhängig weiterentwickeln können sollen.

Statt einer allgemeinen Middleware bauen Unternehmen in solchen Fällen eine separate API pro Frontend auf und schaffen somit ein „Backend for Frontend" (BFF). Ein BFF ist also eng mit einem entsprechenden Frontend gekoppelt, daher sollten Entwicklung und Wartung auch vom gleichen Team übernommen werden. Dies deutet auch darauf hin, wie viele BFFs es geben soll: Wenn die iOS- und die Android-Anwendung von verschiedenen Teams entwickelt werden, sollten es auch unterschiedliche BFFs sein. Wenn diese dagegen im Wesentlichen von einem Mobile-Team entwickelt werden, bleibt es eher bei einem BFF. Das Vermeiden von Reibungsverlusten bestimmt die Entscheidung: Mehr Teams auf einem BFF bedeutet mehr Abstimmungsaufwand zwischen den Teams, was sich gerade im Frontend-Bereich negativ auf die Entwicklungsgeschwindigkeit auswirken kann.

Ein entsprechendes BFF übernimmt auch die Integration verschiedener Microservices, sollte aber keine Business-Logik enthalten. Wenn etwa zum Aufbau einer Seite im Client unterschiedliche Microservices abgefragt werden müssen, macht der Client nur eine Anfrage an das BFF, das BFF fragt daraufhin die einzelnen Microservices ab und gibt eine konsolidierte Antwort zurück. Dies reduziert etwa die Latenzzeiten über das WAN für die unterschiedlichen Microservices-Aufrufe. Während bei den bisher klassisch genutzten Architektur-Patterns die Integration zu einem Teil auf der Oberflächenebene stattfand, verlagert sich diese jetzt ins BFF.

Um klar voneinander getrennte Verantwortlichkeiten und somit auch Lebenszyklen gewährleisten zu können, empfiehlt sich daher die Aufteilung der spezifischen Frontend-Logik. Die Entkopplung der Frontends von den Backend-Services ermöglicht es zudem, eine Applikationsstrategie der zwei Geschwindigkeiten zu verfolgen. Sich permanent verändernde, outside-in-getriebene und bedarfsgerechte Oberflächen beruhen auf sicheren, robusten Business Services. Wenn dann ein neues Business-Feature im Backend realisiert wird, sind zwar mehr Teams zu koordinieren. Aufgrund der Heterogenität der Client-Landschaft erwarten wir aber eine deutlich höhere Veränderungsgeschwindigkeit im Frontend als im Backend.

Zusammenspiel der Komponenten der CAFA

Ein Blueprint einer Architektur ist immer eine Vereinfachung der Gegebenheiten. Abbildung 1 legt eine klare Schichtung nahe, aber in der Realität ist dies nicht so streng. Insbesondere kann es Microservices auf verschiedenen Ebenen geben, da diese ein Prinzip der Microservices-Architekturen sind. Services und Backend Tier stehen somit häufig eher nebeneinander und man kann eher von einer Microservices- und einer Legacy-Domain als einer Unterteilung in der Backend Tier sprechen.

Abbildung 4 zeigt, dass Business-Logik entweder als Microservices oder in klassischen Backend-Applikationen implementiert

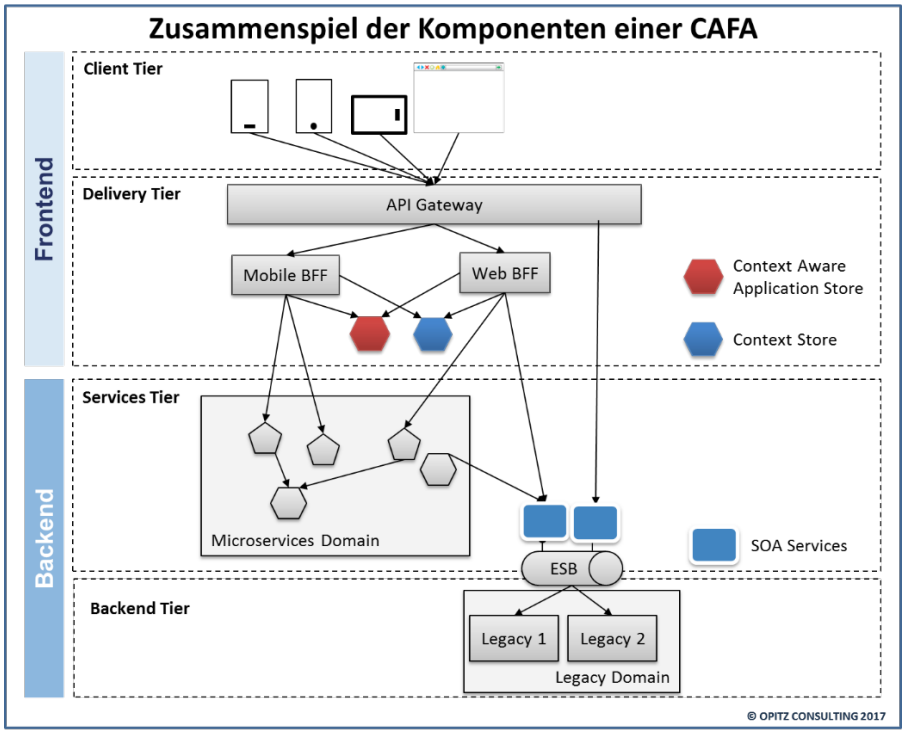

Abb. 3: Schwerpunkte von SOA und API-Management

sein kann. Gleichzeitig befinden sich aber die eher Client-orientierten CAFA-Komponenten in der Delivery Tier. Hier finden sich auch die Context-Store-Konzepte, dargestellt als Hexagons, die es den Clients, bzw. den BFFs ermöglichen, alle Arten von Kontexten zu teilen.

Die Rolle von Microservices in Context-Aware Frontend Architectures

Microservices oder, ähnlich gelagert, auch „Self-contained Systems", sind ein neuer Trend in der IT, womit wir wieder ein neues Buzzword hätten. Leider gibt es zu unserem jetzigen Kenntnisstand noch keine belastbare Definition und somit fehlen auch klare Vorgaben für Kriterien. Gleichwohl ist die Veränderung des Denkmusters *Monolith* zu einer Ansammlung von eigenständigen Subsystemen eine sinnvolle Sichtweise, um flexible und wartbare Lösungen als Alternative für das Altsystem zu schaffen.

Wir verstehen in Anlehnung an J. Lewis und M. Fowler [LaF14] unter einem Monolithen *„eine Enterprise-Applikation, die eine eigenständige, logisch ausführbare Einheit darstellt und somit als Ganzes deployed werden muss"*. Sämtliche notwendige Geschäftslogik liegt in der Applikation vor, inklusive einer Wiederverwendung definierter Dienste. Dies sind die typischen Denkmuster der integrierten Gesamtsysteme mit den geschilderten Schwachstellen im Zeitablauf.

Wir möchten für eine zukunftssichere Applikationslandschaft jedoch Flexibilität erreichen, eine rasche Reaktionsfähigkeit auf veränderte Anforderungen ermöglichen und eine kostengünstige Wartung sicherstellen. Dies sind Ziele, die ein monolithischer Ansatz nicht bedienen kann.

Bei den ersten Unternehmen, die Microservices einsetzen, haben sich folgende Kriterien etabliert:

- Ein Microservice (Synonym: Subsystem) ist eine **selbstständig auslieferbare (deployable) Einheit**, die in der Regel unabhängig von den Release-Zyklen anderer Applikationen ist.
- Der Microservice ist in Komplexität, Größe und Funktionalität beherrschbar. Amazon spricht hier von der „**Two Pizza Rule**" und meint Projektteams in der Größe von 8–12 Personen.
- Die Applikation ist als Produkt und nicht als Projekt zu sehen. Amazon hat dazu die Doktrin „**You build it, you run it**" aufgestellt und hebt damit die Schranken von Entwicklung, Wartung und Betrieb auf.
- Das für den Microservice verantwortliche Team besitzt selbst die **notwendige Domänenexpertise**, um im Sinne eines Produktmanagements die Weiterentwicklung über Budget und Portfolio zu steuern.

Diese Kriterien reichen aus unserer Sicht bereits aus, um einige entscheidende Schlussfolgerungen zu ziehen.

Die Reduzierung von Abhängigkeiten der Subsysteme untereinander, um eigenständige auslieferbare Einheiten zu ermöglichen, ist notwendig, auch falls dies redundante Funktionalität und redundante Datenhaltung mit sich bringt. In der Folge priorisiert die IT-Strategie eine harmonisierte IT-Landschaft nicht mehr so hoch, sondern legt den Fokus auf eine durchgehende Wertschöpfung durch die IT. Dies legt wiederum nahe, dass unterschiedliche Subsysteme unterschiedliche Softwareplattformen besitzen können.

Ein weiterer Aspekt ist, dass das Deployment der Subsysteme (Microservices) meist unabhängig voneinander erfolgen kann. Dies impliziert aus der technischen Sicht einen weitestgehenden Verzicht auf die synchrone Anbindung der Subsysteme untereinander, um Abhängigkeiten bei der Auslieferung und zur Laufzeit zu vermeiden. Endlich erhalten Unternehmen die Möglichkeit, voneinander unabhängige Release-Zyklen zu implementieren, somit eine höhere Flexibilität und eine kürzere Time-to-Market bei neuen Funktionalitäten zu erreichen und damit einen Mehrwert für den Fachbereich durch IT zu generieren.

Ein Microservice soll nur eine Business Capability unterstützen. Deshalb gibt es gemäß dem *Single Responsibility Principle* nur einen Grund, einen Microservice zu ändern, nämlich nur dann, wenn sich relevante Änderungen in Bezug auf die entsprechende Business Capability ergeben. Trotz der geschilderten *Two Pizza Rule* geht es bei einem Microservice nicht um Größe, sondern, wie erläutert, um die zusammenhängende und gekapselte Business Capability. Als Folge dieser Sichtweise benötigt ein Microservice, z. B. nach James Lewis, nicht notwendigerweise ein GUI. Somit wären Microservices ein idealer Kandidat für unsere Backend-Dienste in der CAFA mit einer entsprechenden Business-Service-Sicht.

Die Sichtweise **You build it, you run it**, in Kombination mit der beherrschbaren Größe des Microservice („**Two Pizza Rule**") und der Auffassung, einen Microservice als Produkt zu sehen, bedingen die Notwendigkeit eines Produktmanagements anstelle einer eher singulären Projektsicht. Hintergrund dieser Überlegung ist das Versagen der klassischen Application-Lifecycle-Management-Ansätze in Bezug auf einen zielgerichteten Beitrag zur Wertschöpfung der Anwendung – gerade im Zeitablauf.

Die Erfolgsmessung der klassischen Wartung ist projektorientiert: Zeit, Budget und Anforderungen sind fix. Hier steht die Effizienz der Abwicklung und nicht die Effektivität der Lösung für die Fachbereiche im Vordergrund. Die eher Handover- und Quality-Gate-orientierten Entwicklungs-prozesse sind zu langsam und es fehlt die gemeinsame Verantwortung für das Produkt.

Hier können wir von agilen Ansätzen mit der vertikalen Skalierung von Aufgaben der Teams lernen. Zum Beispiel indem wir Domänenexpertise und Betriebs-Know-how in die Teams hineinziehen, sodass Anforderungen, Test, Qualität sowie Deployment in der Gruppe gelöst werden können. Ob dies notwendigerweise zu agilen Vorgehensweisen führen muss, bleibt zu diskutieren, ist aber aus unserer Sicht durchaus wahrscheinlich.

Wie so oft, bringt ein neuer Ansatz Verbesserungen an der einen Stelle, aber auch Nachteile an einer anderen, oft vorher beherrschten Stelle – so auch bei Microservices: Die **Abhängigkeiten** sind durchaus als Herausforderungen zu sehen.

Obwohl wir die Microservices-Architektur bislang sehr positiv beschrieben haben, gibt es dennoch eine Vielzahl an Hausaufgaben, die hier zu erledigen sind. Aus unserer Sicht steht die geforderte inhaltliche und technische Trennung der Subsysteme im Mittelpunkt einer übergreifenden Governance, die die Aktivitäten der einzelnen Produktentwicklungsteams nicht durch „Bürokratie" lähmen darf. Dabei geht es im Einzelnen um drei wesentliche Herausforderungen:

- Abhängigkeiten bei der Integrationsleistung minimieren und steuern;
- Abhängigkeiten bei zentralen Datenobjekten minimieren und steuern;
- „You build it, you run it"-Doktrin ermöglichen.

Das begleitende Veränderungsmanagement sollte dazu beitragen, dass die Teammitglieder ihre neue ganzheitliche Rolle verstehen, leben und auch wollen. Der isolierte Spezialist muss stärker als bisher auf die fachlichen Anforderungen eingehen, im Team mit engen

Abstimmungszyklen arbeiten und mit dem Betrieb seine Ansätze abstimmen. Dies ist eine deutliche Veränderung des Rollenbilds und eher vergleichbar mit den bekannten Herausforderungen bei der Bildung schlagkräftiger agiler Teams.

Vielleicht wird dieser Sachverhalt noch griffiger, wenn man Microservices mit ihrem dargestellten Produktmanagement als eine Inhouse-SaaS-Lösung betrachtet.

Fazit

Das **Design for Change** ist das entscheidende Architekturprinzip. Design for Change als übergeordnetes Prinzip der Applikationsarchitektur lässt sich realisieren durch die konsequente Trennung von Backend und Frontend, die Flexibilität durch unabhängige Release-Zyklen und die Notwendigkeit einer Applikationsplattform für ein Eco-System of Value.

Zwei wesentliche Denkanstöße prägen die Vorgehensweise hinsichtlich der Applikationsarchitektur als Fundament einer digitalen Plattform:

- die steigende Bedeutung von **kontextsensitiven Applikationen** bei den outside-in-getriebenen Omni-Channel-Apps und die Verbindung dieser meist kleinen, bedarfsgerechten Apps in einem virtuellen **Enterprise App Store**;
- die Nutzung einer weiteren Schicht, der **Delivery Tier**, mit dem Konzept des **Backend for Frontend** (BFF), um die App-Welt effizient zu gestalten.

Ferner haben wir die Bedeutung der Ansätze des API-Managements und der Nutzung von Microservices-Architekturen als zentrale Bausteine erläutert. Was bleibt ist nun der Aufruf: Starten Sie schon jetzt, da der Umbau der Applikationslandschaft ein mehrjähriges Transformationsprogramm darstellt. ■

Referenz

[LaF14] James Lewis, Martin Fowler: Microservices - a definition of this new architectural term; https://martinfowler.com/articles/microservices.html