Richard Attermeyer
(Richard.Attermeyer@opitz-consulting.com)
arbeitet als Senior Solution Architect bei der OPITZ CONSULTING Deutschland GmbH. Er beschäftigt sich seit mehreren Jahren mit der Architektur und Implementierung von Anwendungen im agilen Umfeld. Er fokussiert sich dabei aktuell auf moderne Architekturansätze rund um Microservices, Cloud, DevOps und Continuous Delivery.

Frontend-Architekturen für Microservice-basierte Systeme

Microservices haben sich als Architekturstil für Softwaresysteme etabliert. Häufig steht dabei die Frage im Raum, ob die Benutzerschnittstelle zu einem Microservice gehört oder nicht. Je nach Antwort gibt es unterschiedliche Möglichkeiten, die Frontend-Architektur zu entwerfen. Wie immer hat jedes Pattern seine Architekturtreiber, Zielkonflikte sowie Chancen und Risiken. Der Artikel beschreibt drei häufig anzutreffende Muster und diskutiert, wie sich die aktuelle Entwicklung in der Nutzung von verschiedenen Endgeräten auf die Architektur auswirkt. Dazu stellen wir das Konzept „Context-Aware Frontend Architecture" vor, das diese neuen Herausforderungen adressiert.

Einleitung

Microservices verfolgen den Ansatz, Systeme entlang ihrer fachlichen Grenzen zu schneiden. Jeder Microservice implementiert einen fachlichen Aspekt (frei nach dem Motto: „Mache eine Sache, aber mache sie richtig"). Microservices leben aber nicht in Isolation, sie interagieren miteinander und bilden zusammen ein System. Ein solches System braucht zusätzlich auch Schnittstellen, über die ein Benutzer mit ihm interagieren kann.

Es stellt sich daher die Frage: Welche Frontend-Architektur wählt man für ein Microservice-basiertes System? Dazu schauen wir uns verschiedene Architekturpattern an, ausgehend von einer monolithischen Architektur, über spezifische Benutzeroberflächen für jeden Service, hin zu spezifischen Oberflächen je Fachlichkeit und je Eingabegerät (Desktop, Smartphone, Tablet).

Wir diskutieren, welche Auswirkung die weiter zunehmende Anzahl unterschiedlicher Eingabe- und Anzeigegeräte hat. Diese werden in Zukunft kontextspezifisch eingesetzt. Mehrere Geräte werden dabei zusammen oder nacheinander zur Aufgabenbearbeitung genutzt. Damit das funktioniert, muss die Architektur weiterentwickelt werden. Wir schließen daher mit der Vorstellung einer „Context-Aware Frontend Architecture" Architekturpattern.

UI-Monolith

Das erste Pattern ist eine Microservice-basierte Backend-Architektur mit einem monolithischen Frontend (siehe Abbildung 1). Es scheint vielleicht seltsam, dass eine Organisation das Backend auf Microservices umstellt, das Frontend aber monolithisch lässt. Das ist durchaus eine Abwägungsfrage.

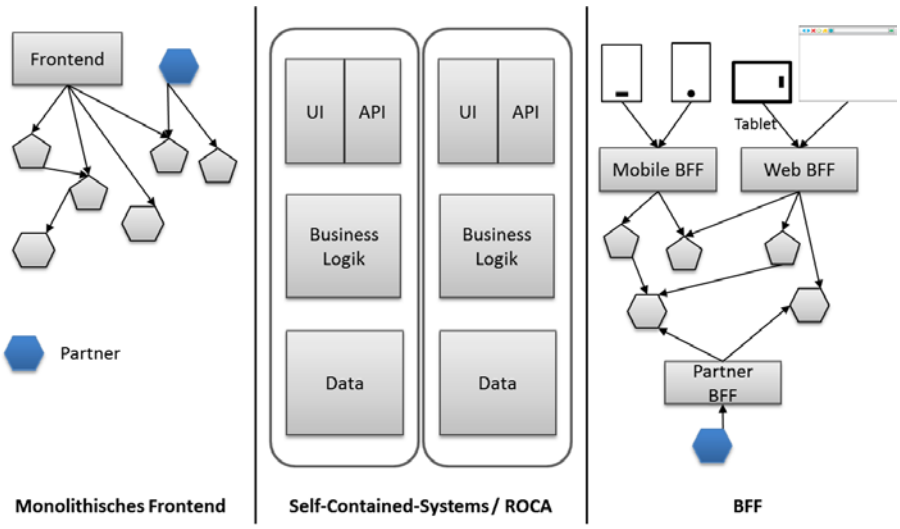

Abb. 1: Frontend-Architekturen

Unter Microservices verstehen wir Services, die sich durch eine abgeschlossene fachliche Logik auszeichnen, die klein genug ist, um von einem Team fachlich und technologisch beherrscht zu werden. Dieses Pattern findet sich bei vielen derzeitigen Neuentwicklungen. Bei diesen Neuentwicklungen wird das Backend vielfach entsprechend der Business Capabilities in unterschiedliche Systeme aufgeteilt.

Auch wenn Entwicklungsabteilungen Microservices in diesem Fall nicht mit allen Konsequenzen umsetzen, müssen sie sich mit vielen neuen Technologien und Methoden auseinandersetzen. Oft geht die Einführung von Microservices mit der Einführung und dem Ausbau der Fähigkeiten im Continuous Delivery einher. Der Weg von Continuous Integration zu Continuous Delivery, die Nutzung von Cloud oder Containerization bringt viele neue Methoden und Technologien mit.

Im Backend-Bereich, der häufig Java-basiert ist, sind die Prozesse und Werkzeuge schon relativ gut verstanden. Anders sieht es im Frontend-Bereich aus. Viele Neuentwicklungen gehen Richtung Single-Page-Applikationen, etwa auf Basis von Angular. Die Java-Spezialisten in den Entwicklungsabteilungen müssen sich demnach mit einer größer werdenden Anzahl an neuen Werkzeugen auseinandersetzen.

Gleichzeitig steckt die Unterstützung für unternehmenstypische Abläufe noch in den Kinderschuhen. So bieten erst die aktuellsten Versionen von Maven Repository eine Unterstützung für das Proxying von NPM-Artefakten. Dazu kommt, dass sich im JavaScript-Umfeld momentan noch viel ändert. Mit der Entwicklung von großen JavaScript-Anwendungen haben nur wenige Unternehmen Erfahrung.

Vor diesem Hintergrund entscheiden sich einige Unternehmen für ein monolithisches Frontend und gegen eine kleinteiligere Frontend-Architektur. Die Entwicklungsorganisation wird dabei sehr unterschiedlich aufgesetzt: Mal kümmert sich ein dediziertes Team um das Frontend, in anderen Fällen arbeiten die einzelnen Microservice-Teams an der gemeinsamen Frontend-Codebasis. Letzteres ist vorzuziehen, wenn sich die Teams dabei nicht gegenseitig behindern, denn je mehr Komponenten der Anwendung unter der Kontrolle eines Teams liegen, desto autonomer kann das Team arbeiten und damit Anforderungen zügiger umsetzen.

Eine typische Konsequenz dieser Architektur ist, dass sich die Single-Page-Applikation wie jeder andere Client direkt an den REST-Schnittstellen der Microservices bedient. Es findet also eine Integration oder Aggregation der Services im UI-Layer statt.

Deshalb ist das Frontend auch relativ stark an das Backend gekoppelt. Änderungen an Struktur und Schnittstellen der Backend-Services müssen Entwickler somit direkt im Frontend umsetzen. Mit den folgenden häufig zu beobachtenden Konsequenzen:

- Der Server gibt Antworten im JSON Format zurück,
- JavaScript spielt eine zentrale Rolle. Ohne JavaScript funktioniert die Anwendung nicht.

Diese Art des Frontends findet sich daher eher bei Anwendungen für Sachbearbeiter, die primär auf dem Desktop laufen. Responsive Design spielt hier eine untergeordnete Rolle (siehe auch **Abbildung 1**).

ROCA und Self-contained Systems

Jedes Self-contained System [SCS] stellt eine autonome Webanwendung dar, die ein Team verantwortet. Neben der Benutzeroberfläche kann ein SCS über eine optionale Service-API verfügen. Jede Benutzeroberfläche eines SCS ist unabhängig von den Oberflächen anderer SCS. Die Integration erfolgt im Wesentlichen über Links. Insbesondere sollte die Oberfläche eines SCS auch dann noch funktionieren, wenn die Oberfläche eines anderen SCS nicht verfügbar ist.

Die Benutzeroberfläche eines SCS orientiert sich an den Prinzipien der Resource-oriented Client Architecture (ROCA) [ROCA]. Deshalb werden in der Regel auch keine unterschiedlichen Oberflächen für verschiedene Geräte angeboten. Stattdessen wird die Webanwendung mittels Responsive Design [RespDes] so flexibel gestaltet, dass sie auf unterschiedlichen Endgeräten und Auflösungen trotzdem eine gute Benutzerfreundlichkeit bietet.

Dieser Stil eignet sich sehr gut, wenn im Team zwar Fähigkeiten zur Entwicklung von klassischen Webanwendungen vorhanden sind, aber wenig Erfahrung mit JavaScript-basierten Benutzeroberflächen. Letztlich ist es eine Frage der Entscheidung: Betrachtet man den Browser eher als Anzeigeplattform oder als Ablaufumgebung für Anwendungen?

„Backends for Frontends“

Die Unterstützung für verschiedene Geräte wird im Allgemeinen immer wichtiger. So befinden sich auch computergestützte Benutzerschnittstellen aktuell im Umbruch. Während die Oberflächenwelt vor wenigen Jahren mit Desktop-Anwendungen, webbasierenden Interfaces und nativen Oberflächen für spezielle Devices wie Smartphones noch überschaubar war, haben sich die Möglichkeiten der Frontend-Technologien heute stark verändert und erweitert und schaffen somit neue Möglichkeiten für die bedarfsgerechte Informationsbereitstellung.

Insbesondere bei den mobilen Geräten kommen neue Klassen hinzu wie Smart Watches oder 3D-Brillen. Gleichzeitig wird die traditionell explizite Bedienung von Oberflächen mittels Maus, Tastatur und Touch-Screens ergänzt durch eine eher implizite Bedienung durch Gesten, Sprache, Augen- und Körperbewegung. Dies geht einher mit den noch etwas futuristischen Trends der 3D-Darstellung, die eine realitätsnahe Objektdarstellung möglich machen sollen.

Die Leistungsfähigkeit von Rechnern, Endgeräten und Devices befeuert diese Entwicklung zusätzlich. Interaktive 3D-Brillen werden bei steigender Leistungsfähigkeit schon immer kleiner. Insgesamt werden wir in den nächsten fünf bis sieben Jahren eine grundlegende Veränderung der Mensch-Maschine-Interaktion erleben [HCI].

Mobile Geräte haben unterschiedliche Funktionalitäten und Bandbreiten. Möchte etwa ein Einzelhändler auf seiner Smartphone-App die Funktionalität zum Scannen von Barcodes anbieten, macht das Smartphone gegebenenfalls ganz andere Aufrufe an das Backend als die Browseranwendung auf dem Laptop.

Der Versuch, diese unterschiedlichen Bedürfnisse über ein allgemeines Frontend-API zu befriedigen, wird aus verschiedenen Gründen Probleme verursachen: 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 ein API Gateway pro Frontend auf, und schaffen somit ein „Backend for Fron-

tend“, oder kurz ein BFF (siehe [BFF1] und [BFF2]). Ein BFF ist also eng mit einem entsprechenden Frontend gekoppelt und daher sollte auch das gleiche Team Wartung und Entwicklung übernehmen.

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. Vermeiden von Reibungsverlusten bestimmt die Entscheidung: Mehr Teams auf einem BFF bedeutet mehr Abstimmungsaufwand zwischen den Teams, was sich gerade im Frontendbereich negativ auf die Entwicklungsgeschwindigkeit auswirken kann.

Ein entsprechendes BFF übernimmt auch die Integration der unterschiedlichen Microservices. Wenn etwa zum Aufbau einer Seite im Client unterschiedliche Microservices abgefragt werden müssen, dann macht der Client nur eine Anfrage zum BFF, das BFF fragt daraufhin die einzelnen Microservices ab und gibt eine konsolidierte Antwort zurück. Dies reduziert etwa die Latenzzeiten übers WAN für die unterschiedlichen Microservices-Aufrufe. Während bei den beiden vorhergehenden Architektur-Pattern die Integration zu einem Teil noch auf der Oberflächen-Ebene stattfand, verlagert sich diese jetzt ins Frontend.

Wenn sich herausstellt, dass die gleiche Logik in unterschiedlichen BFF genutzt wird, kann es ratsam sein, die Kosten für einen neuen Service, der die entsprechende Logik kapselt gegenüber den Kosten abzuwägen, die eine Shared Library oder eine doppelte Logik verursachen. Viele Organisationen entscheiden sich dafür, die Kernfunktionalitäten (siehe Tabelle) zu zentralisieren.

Kernfunktionalitäten	Frontend-spezifische Aufgaben
■ Caching von Inhalten ■ Authentifizierung (oAuth2, SSL, Basic Authentication, SSO), Autorisierung und Auditing ■ API Key Management ■ API Traffic Control (Throtteling und Quotas) ■ Schutz vor Angriffen ■ Logging, Monitoring, Analytics	■ Geräteklassen-spezifische Inhaltsaufbereitung (z.B. Filterung) ■ Geräteklassen-spezifische Formatkonvertierungen (e.g. JSON <-> XML) ■ Geräteklassen-spezifische APIs (Aggregation von feingranularen Backend-APIs)

Tab.: Aufgaben „Backends for Frontends“

Context-Aware Frontend-Architecture

Das reine BFF-Pattern betrachtet zwar jedes Frontend für sich, dennoch werden häufig unterschiedliche Endgeräte benutzt, um ein Ziel zu erreichen: Die Smartwatch zeigt an, dass eine Mail eingegangen ist, die dann aber mit dem Smartphone beantwortet wird. Das zeigt, dass zwei nicht ganz trennscharfe Trends hinsichtlich der Nutzung von Computern in echten Einsatzszenarien immer wichtiger werden: „Ubiquitous Computing“ [UBI] und „Pervasive Computing“.

Ersterer ist ein Trend in der Arbeitswelt, bei dem „alles“ zu einen „Computer“ wird und die Arbeit unterstützt. Smartphones, Tablets wie auch Armbänder oder 3D-Brillen sollen jederzeit bei der Arbeit Verwendung finden und situativ unterstützen. Feste Arbeitsstationen gehören der Vergangenheit an und Rechenleistung ist überall „zu Diensten“. Pervasive Computing beschreibt hingegen den Trend einer vernetzten Welt, in der Sensoren, Gegenstände wie auch menschliche Akteure in einem gemeinsamen Netzwerk agieren. Ein bekannter Treiber dieser Entwicklung ist die deutsche „Industrie 4.0“-Initiative.

Beide Trends befeuern sich gegenseitig. Je eher ich die Dinge vernetze, desto eher kann ich „überall“ elektronische Hilfsmittel nutzen. Und je eher ich „überall“ arbeiten möchte, desto eher sind wir bestrebt, diese Gegenstände zu vernetzen. Beim Pervasive Computing erfolgt die Vernetzung der Umwelt, bei der computer-basierende Systeme mit Sensorik beginnen, die Umwelt wahrzunehmen und diese Informationen mit dem Anwender zu teilen. Das können Informationen aus dem sozialen Netzwerk sein oder die einer Fabrikhalle. In einem sozialen Netzwerk können Benutzer miteinander kommunizieren und teilweise in einem virtuellen Raum interagieren.

Mit dieser Auffassung eines sozialen Netzwerks kann man auch in Zukunft nicht nur die Mitarbeiter in einer Fabrikhalle als Mitglieder eines solchen Netzwerks betrachten, sondern auch Maschinen. Mitarbeiter können angefragt werden, wenn es um Tipps zur Problemlösung geht. Maschinen können ein „Statusupdate“ senden und mehrere Mitarbeiter können in einem virtuellen Raum zusammen an der Lösung einer Maschinenstörung arbeiten. Dabei müssen nicht alle Akteure am gleichen Ort sein. Zusammen ergibt sich ein virtuelles Gesamtbild beim Betrachter, etwa mithilfe von Ansätzen aus dem Bereich der „Augmented Reality“.

Die User Journey eines Fabrikangestellten

Genauso wie ein Kunde verschiedene Berührungspunkte mit einem Unternehmen haben kann, so hat ein Mitarbeiter verschiedene Berührungspunkte mit den IT-Systemen des Unternehmens, in dem er arbeitet. Das ist auch klassisch so. Was aber häufig nicht im Rahmen einer Anforderungsdefinition untersucht wird ist, wie diese Interaktion stattfindet wenn ein Mitarbeiter eine bestimmte Aufgabe erledigen will.

Traditionell werden diese Systeme häufig unabhängig voneinander spezifiziert. Ein übergreifender Ablauf wird häufig nur technisch auf Basis von recht technischen Workflows beschrieben, aber nicht aus Benutzungssicht eines Mitarbeiters.

Die User Journey [UJ] ist ein Verfahren, mit dem unterschiedliche Berührungspunkte aus Nutzersicht beschrieben werden. Das Verfahren ist visuell strukturiert und setzt den Nutzer in den Mittelpunkt (durch Personas und Aspekte des Value Proposition Canvas [VPC]). Es eignet sich daher gut für Workshops und eine Posterdarstellung im Projektbüro. Das in [UJ] dargestellte Verfahren lässt sich auch gut auf die Unternehmens-IT übertragen, wenn man den Kunden durch den Mitarbeiter ersetzt.

Noch eindrucksvoller lässt sich das Verfahren nachvollziehen, wenn man an dieser Stelle einmal nicht den Sachbearbeiter im Büro betrachtet, sondern einen Mitarbeiter in einer Fabrik. Dies haben wir im Rahmen eines Proof of Concept gemacht und daraus eine beispielhafte User Journey erstellt (siehe Abbildung 2), die auch Devices und Applikationsarten mit aufgeführt, die der Nutzer verwendet. Ausgehend von einer Journey sieht man in Ab-

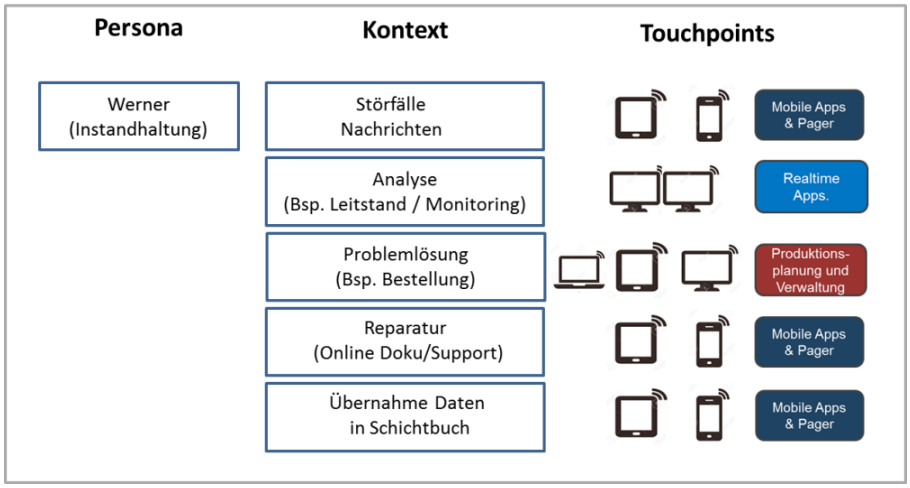

Abb. 2: User Journey des Fabrikangestellten „Werner“

bildung 2, dass ein Mitarbeiter verschiedene „Apps“ verwendet, um sein Ziel – hier die Reparatur einer Maschine – zu erreichen.

Welche App der Mitarbeiter nutzt, hängt davon ab, in welchem Kontext er sich befindet. Der Kontext ergibt sich durch den Ort und die Phase der Problembehebung. Am Anfang steht die Benachrichtigung. Die bekommt er auf sein Handy oder den Pager, dann geht es über die Analyse und die Reparatur bis zur Dokumentation der geleisteten Arbeit.

Context-Awareness

Wie wir an der beispielhaften User Journey aus Abbildung 2 sehen, werden für die Erfüllung einer Aufgabe verschiedene Devices genutzt. Wenn der Benutzer das Device wechselt, will er aber auch mit der Desktopanwendung häufig da weiter machen, wo er auf dem Smartphone oder dem Tablet aufhören musste. Wenn er also gerade an der Warenannahme war und eingehende Waren mittels seiner MDE eingebucht hat und jetzt im Büro weiterarbeiten will, dann sollte das System ihm als mögliche nächste Aktion im Büro den Bericht zum Lagerbestand anbieten.

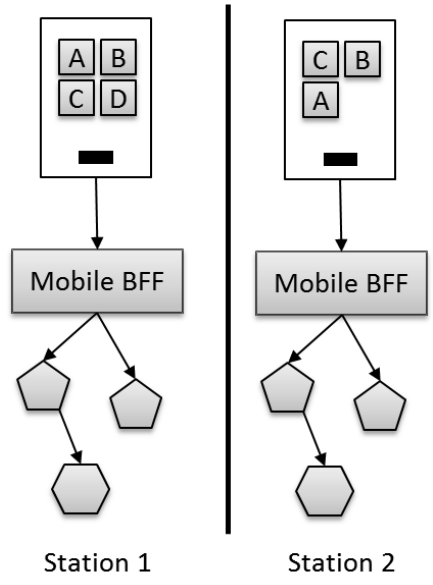

Abb. 3: Kontextabhängige Applikationspräsentation

Aber es könnte genauso gut sein, dass der Anwender ein Smart Device unterschiedlich nutzt, je nachdem, an welchem Ort er sich gerade befindet. Wenn sich der Mitarbeiter am Wareneingang befindet, will er vermutlich eher eingehende Waren ins System buchen. Wenn er sich hingegen zwischen den Regalen befindet, will er eventuell eher eine Inventur vornehmen. Das Device sollte ihm also abhängig vom Kontext „Ort“ die wahrscheinlich präferierten Werkzeuge zur Erledigung seiner Aufgaben anbieten. Es sollte daher nicht nur „die eine“ mobile App geben, sondern abhängig von der Aufgabe unterschiedliche.

Während der Fokus bei den BFFs darauf lag, pro Frontend ein Backend zu erstellen, fächert sich das im Frontend weiter auf. Je nachdem in welchem Kontext sich der Mitarbeiter befindet, werden ihm hier spezifische Anwendungen zur Aufgabenerledigung angeboten.

Ein Kontext kann sich dabei aus verschiedenen Informationen ergeben:

- dem Ort, an dem sich der Mitarbeiter befindet,
- den eingehenden Events auf dem Handy, z. B. ein Alarm für den Ausfall einer Maschine, oder
- den Schritten in einem Geschäftsprozess.

Der Einfluss von Orten auf die Angebote in einer Applikation ist schon unter dem Namen „Location-based Services“ bekannt. Allerdings betrifft das eher die in der Anwendung dargestellte Information und diese ist häufig noch sehr grob. Mit der Nutzung von iBeacons ist es schon möglich, Ortsinformationen in geschlossenen Räumen nutzbar zu machen.

Somit können Applikationen dynamisch ein- und ausgeblendet oder die Reihenfolge geändert werden. Applikationen, die in einem Kontext häufiger genutzt werden, sollten einfacher erreichbar sein. In Abbildung 3 ist dargestellt, wie sich Anordnung und Reihenfolge von Applikationen ändern. Auch diese Funktion haben wir in dem erwähnten Proof of Concept umgesetzt.

Betrachtet man die unterschiedlichen Kontexte, dann ergeben sich unterschiedliche Interaktionsmuster. Häufig findet man in „Sachbearbeiteranwendungen“ noch eine sogenannte Expertenoberfläche. Begründet wird dies häufig damit, dass ein Sachbearbeiter im Laufe seiner Arbeit all diese Informationen benötigen würde. Dabei braucht er diese Informationen selten alle zur gleichen Zeit.

Solche Oberflächen tendieren dann eher in Richtung „monolithische Benutzeroberfläche“. Wenn man die Kontexte klarer trennt, eröffnen sich auch Möglichkeiten hinsichtlich Self-contained Services oder BFF.

Wir schlagen vor, dass sich eine „Context-Aware-Frontend-Architecture“ (CAFA) am BFF-Muster orientiert, aber darüber hinaus noch zwei wesentliche Services bereitstellt (siehe auch Abbildung 4):

- Context-Aware Application Store (CAAS) und
- Context Store (CS).

Der CAAS sorgt dafür, dass Applikationen registriert und Kontexten zugeordnet werden. Dabei kann die Zuordnung von Applikationen zu Kontexten dynamisch vom CAAS gelernt werden („Andere Benutzer in diesem Kontext verwenden am häufigsten …“). Außerdem werden hier Regeln auf Basis von Events hinterlegt (Ortsinformation, Business Process Events, genutztes Device), um einen Kontext zu definieren.

Der CS ist so etwas wie ein applikations- und geräteübergreifender „Session Store“. In ihm ist gespeichert, in welchem

Kontext der Nutzer zuletzt unterwegs war, in welchem Kontext er sich aktuell befindet und welche weiteren Informationen von Interesse sein könnten.

Das kann man sich so vorstellen: Eine Mobile App kombiniert zur Anzeige von Maschinenparametern den Kontext „Maschineninformation auf Tablet“ mit der Zusatzinformation „Stanze“ im CS. Wenn der Mitarbeiter jetzt in den neuen Kontext „Arbeitsplatzrechner Linie 1“ wechselt, dann wird der CAAS die Applikation „Wartungsbericht Stanze“ hoch priorisieren und dem Mitarbeiter als erste Option anbieten.

Die Kernaufgaben (siehe Tabelle) einer BFF sind in einem transparenten und leichtgewichtigen API Gateway gebündelt. Ein solches API Gateway soll in der Regel keine Frontend-spezifische Logik enthalten. Für die Entwicklung ist es wichtig, dass dieses API Gateway auch auf einem Entwicklungsrechner einfach betrieben werden kann, damit das Frontend-Entwicklungsteam Client- und Delivery-Layer-Komponenten weitgehend unabhängig von anderen Teams entwickeln kann.

Dabei definieren wir die einzelnen Layer wie folgt:

- **Client Tier**: Applikationsteil, der auf dem Client läuft, etwa nativ, als Single-Page-App oder als klassische Webanwendung mit Progressive Enhancement mittels JavaScript.
- **Delivery Tier**: Adressiert die spezifischen Anforderungen eines Frontends hinsichtlich Caching oder Aggregation von Backend Business Services. Mehrfachentwicklung von Aggregationscode wird akzeptiert, um eine höhere Unabhängigkeit der Frontend-Teams zu erhalten. Der Delivery Layer gehört auch zum Frontend-Bereich.
- **Service Tier**: Dies ist das klassische Backend. Hier werden Business Services bereitgestellt.

Client und Delivery Tier stellen somit den Bereich der Frontend-Entwicklung dar.

Wer Forresters 4-Tier-Modell [4Tier] kennt, wird feststellen, dass wir die gleichen Begriffe mit einer anderen Bedeutung verwenden. Forrester hat mit dem Modell dazu angeregt, über die Frontend-Architektur nachzudenken und das ist auch gut so. Allerdings sieht das Modell auch noch einen separaten „Aggregation Tier“ vor, obwohl Aggregation an unterschiedlichen

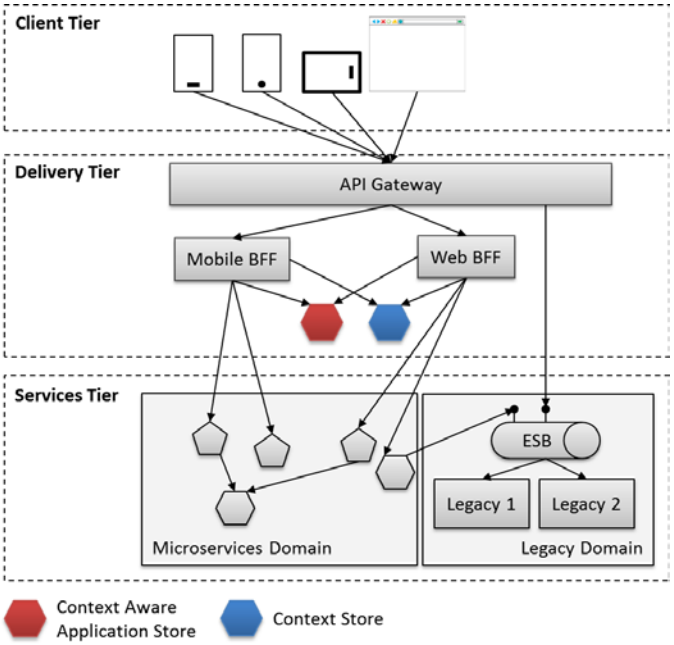

Abb. 4: Context-Aware Frontend Architecture

Stellen in einer modernen Frontendarchitektur stattfindet.

Typische Stellen sind die BFF, einzelne Microservices, ein Enterprise Service Bus oder auch der API Gateway. Komponenten wie das BFF sind daher schlecht einzuordnen: Gehören sie nach diesem Modell zum Client Tier oder zum Aggregation Tier? Weiterhin kann man sich fragen, ob der Delivery Tier wirklich nur für das Caching (etwa in Form eines CDN) zuständig ist. Denn Caching kann ebenfalls an verschiedenen Stellen in der Architektur stattfinden (typisch im API Gateway, BFF oder ESB).

Fazit

Wir haben gesehen, dass es sehr unterschiedliche Möglichkeiten für eine Frontend-Architektur gibt. Insbesondere die weitere Differenzierung der Endgeräte in den kommenden Jahren erfordert eine Architektur, die neue Gerätetypen einfach unterstützen kann und einen multi-modalen Zugriff auf die IT-Systeme nahtlos unterstützt.

Einen Vorschlag, der diese multi-modale Nutzung unterstützt, haben wir mit dem Stil der Context-Aware Frontend Architecture vorgestellt. Die nächsten Jahre werden zeigen, ob sich dieser Stil trägt und welchen Einfluss eine breite kontextabhängige Gerätenutzung auf Frontend-Architekturen haben wird. ■

Referenzen

[UJ] User (Customer) Journey Mapping, http://www.businessmodelcreativity.net/user-customer-journey-mapping/

[SCS] Self-contained-systems, http://scs-architecture.org/

[ROCA] Roca Style, http://roca-style.org/

[RespDes] Responsive Design, http://www.responsive-webdesign.mobi/was-ist-responsive-webdesign/

[HCI] Human-Computer Interaction: Present and Future Trends, https://www.computer.org/web/computingnow/archive/september2014#sthash.Xet4cVMu.dpuf

[BFF1] http://samnewman.io/patterns/architectural/bff/

[BFF2] https://www.thoughtworks.com/insights/blog/bff-soundcloud

[UBI] http://www.computerwoche.de/a/ubiquitous-computing-ist-realitaet,3093580

[4Tier] http://blogs.forrester.com/ted_schadler/13-11-20-mobile_needs_a_four_tier_engagement_platform

[VPC] The Value Proposition Canvas, http://www.businessmodelgeneration.com/canvas/vpc