Effiziente Delivery mit APIs, Microservices und DevOps

Sven Bernhardt, OPITZ CONSULTING Deutschland GmbH

Traditionelle IT-Systemlandschaften bestehen oft aus monolithischen Applikationen, die häufig langwierigen, formalisierten Release-Zyklen unterliegen. Aus Sicht der Gesamtstabilität ist das auch sinnvoll, da monolithische Applikationen in der Regel eine Sammlung von eng miteinander verzahnten Business Capabilities abbilden – fachlich bedingte Änderungen sind so allerdings nur innerhalb der Release-Zyklen möglich – und somit wenig agil sind. Konträr dazu steht heute die zentrale Herausforderung der IT, Agilität im Unternehmen sicherzustellen. Änderungen an bestehenden Komponenten sollen schnell durchgeführt und bereitgestellt werden, ohne dass dies Auswirkungen auf bestehende Systeme und Services hat. Definitiv eine Herausforderung! Mit der richtigen Architektur-Idee und geeigneten Werkzeugen jedoch keine unlösbare …

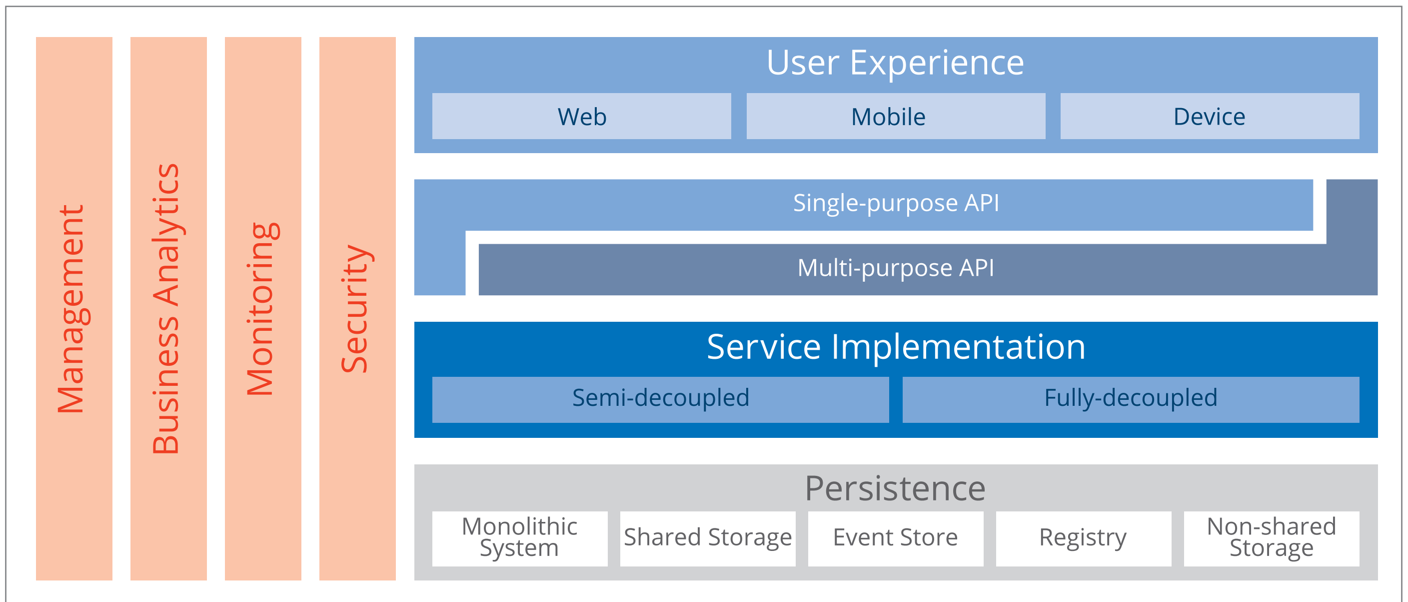

Abbildung 1: OMESA-Referenz-Architektur

Moderne Ansätze wie Microservices-Architekturen *[1]* setzen auf die strikte Trennung unterschiedlicher Business Capabilities, die unabhängig voneinander implementiert, bereitgestellt und betrieben werden können. Ist eine Kommunikation zwischen den Services erforderlich, erfolgt diese in der Regel eventbasiert über einen Event-Hub. Einzelne Microservices sind somit vollständig voneinander entkoppelt. Änderungen, bedingt durch neue oder geänderte Anforderungen, können also ohne Beeinträchtigung bereits vorhandener Funktionalitäten erfolgen; soweit die Theorie.

In der Praxis geht es allerdings nicht ohne ein Umdenken in der Organisation. Betrieb und Entwicklung müssen enger zusammenrücken. Die konsequente Umsetzung eines DevOps-Ansatzes ist unabdingbar für den Erfolg von Microservices, um Vorteile wie die Steigerung von Agilität und Effizienz realisieren zu können. Die Einführung eines DevOps-Ansatzes bedingt neue Denk- und Arbeitsweisen innerhalb der IT-Organisation. Dazu zählen unter anderem:

- Das Aufbrechen getrennter Entwicklungs- und Betriebsbereiche
- Die Formung neuer, heterogener Teams
- Die Automatisierung großer Teile des Entwicklungsprozesses

Das neue Mantra, das die neu geformten Teams in diesem Zusammenhang verinnerlichen müssen, lautet: „You build it, you run it!" Ein Prozess, der nicht von heute auf morgen abgeschlossen ist. Die erfolgreiche Einführung von Microservices ist also mit nicht zu unterschätzenden organisatorischen Herausforderungen verbunden, insbesondere für gewachsene IT-Organisationen. Aber auch technologisch beziehungsweise architektonisch ergeben sich diverse Herausforderungen – denn in den seltensten Fällen startet ein Unternehmen auf der grünen Wiese.

Referenz-Architektur für flexible Anwendungs-Architekturen

Das Projekt der Open Modern Enterprise Software Architecture (OMESA) *[2]* beschäftigt sich mit verschiedenen Fragestellungen, zum Beispiel damit, wie bewährte architektonische Grundprinzipien und Architekturmuster in modernen Software-Architekturen zu verankern sind. Ziel ist es, alte und neue Welt sinnvoll miteinander zu kombinieren. Anstelle kompletter Restrukturierung und Refaktorisierung heißt die Devise „sinnvolle Koexistenz". Ein solches Vorgehen ist gerade in Bezug auf langjährig gewachsene IT-Systemlandschaften sinnvoll.

Eine der zentralen Botschaften von OMESA lautet: „Microservices are no silver bullets!" OMESA definiert zu diesem Zweck eine Referenz-Architektur sowie ein mehrstufiges Capability Model *(siehe Abbildung 1)*. Sie zeigt die zentralen Ebenen, wobei Microservices im Bereich der „Service Implementation" *[3]* einzuordnen sind, als sogenannte „Fully-decoupled Services". Auf der Ebene der „Persistence" befinden sich die bestehenden und zumeist monolithischen IT-Systeme, die über „Semi-decoupled Services" in die Gesamt-Architektur eingebunden sind.

Oberhalb der Ebene „Service Implementation" ist die API-Ebene verortet, die sich wiederum in die Bereiche „Single-Purpose API" und „Multi-Purpose API" aufteilt. Auf der obersten Ebene, der „Delivery Experience", geht es schließlich um die Interaktion mit der Außenwelt; in vielen Fällen handelt es sich hierbei um eine Mensch-Maschine-Interaktion. Dabei geht es vor allem um Themen wie moderne Client-Applikationen oder die Möglichkeit, über verschiedene Kanäle wie Chatbots mit den Services eines Unternehmens interagieren zu können.

APIs verbinden Microservices und UIs

In der OMESA-Referenz-Architektur ist die API-Ebene ein grundlegender Baustein moderner Software-Architekturen. Aber warum sind APIs so essenziell? Um möglichst unabhängig voneinander zu bleiben, interagieren Microservices untereinander hauptsächlich asynchron beziehungsweise eventbasiert. Für die Kommunikation mit der Außenwelt, bei-

spielsweise über Benutzeroberflächen, ist dieser Kommunikationsstil allerdings im Sinne einer guten User Experience (UX) nicht geeignet. Ein synchrones Kommunikationsverhalten, das sich durch zeitnahe Reaktionen auszeichnet, fühlt sich für menschliche Akteure natürlicher an und sollte deshalb auch hier das bevorzugte Kommunikationsmuster sein. Das bedeutet, dass Microservices, deren Funktionalitäten extern verfügbar gemacht werden, ein synchrones Service-Interface, also ein API, bereitstellen müssen.

Der in OMESA propagierte, zweischichtige API-Ansatz, der Single- und Multi-Purpose-APIs unterscheidet, macht die Gesamt-Architektur flexibler und agiler *[4]*. Multi-Purpose-APIs sind allgemeiner, bieten einen breiteren Funktionsumfang und sind somit potenziell wiederverwendbar; Single-Purpose-APIs hingegen können auf den Multi-Purpose-APIs aufbauen und bilden dabei UI- oder Device-spezifische Logik ab. Single-Purpose-APIs stellen im Grunde eine Implementierungsvariante des „Backends For Frontends"-Pattern (BFF) *[5]* dar.

Die API-Ebene dient also vor allem dazu, die Service-Implementierung von der Benutzeroberfläche zu abstrahieren und die von den Services bereitgestellten Funktionalitäten sicher nach außen zu exponieren. „Sicher" bedeutet hier, dass die API-Ebene übergreifende Aspekte wie grundlegende Sicherheitsmechanismen (wie Authentifizierung und Autorisierung), Origin-Controls (Cross-Origin Sharing Resources, CORS *[6]*) oder Threat-Protection-Maßnahmen (wie Rate Limits) zentral und konsistent definiert werden, ohne diese explizit in jedem Service ausimplementieren zu müssen. Das hat den Vorteil, dass sich Backend-Entwickler voll und ganz auf die Implementierung der Geschäftslogik konzentrieren können.

Zwischenfazit der wichtigsten Fakten

Zusammenfassend kann bislang festgehalten werden, dass die Umsetzung moderner Architekturen auf der Basis von Microservices sowohl organisatorische als auch architektonische sowie technische Herausforderungen mit sich bringt. Zudem ist die Etablierung von DevOps essenziell für den Erfolg von Microservices. APIs sind in diesem Zusammenhang wichtig, um Service-Funktionalitäten nach außen anbieten zu können und so Mehrwerte wie die Etablierung neuer digitaler Economies zu generieren.

Im zweiten Teil dieses Artikels sollen die angesprochenen Aspekte kurz anhand eines praktischen Beispielprojekts näher beleuchtet werden. Es handelt sich dabei um eine gemeinschaftliche Entwicklung der Oracle-ACEs Lonneke Dikmans, Lucas Jellema, Luis Weir, Guido Schmutz, José Rodrigues und Sven Bernhardt, die sich in Vorbereitung auf einen Vortrag für das jährlich stattfindende PaaS-Forum im März 2018 als Team zusammenfanden, um einen Showcase auf Basis der Oracle Cloud zu entwickeln. Die Ideen der OMESA-Referenz-Architektur dienten dem Team dabei als architektonische Grundlage für die Implementierung.

Beispielszenario: eine Webshop-Lösung

Als Beispielszenario diente dem Team ein fiktiver Webshop, der auf einer Microservices-Architektur basiert. Jedes

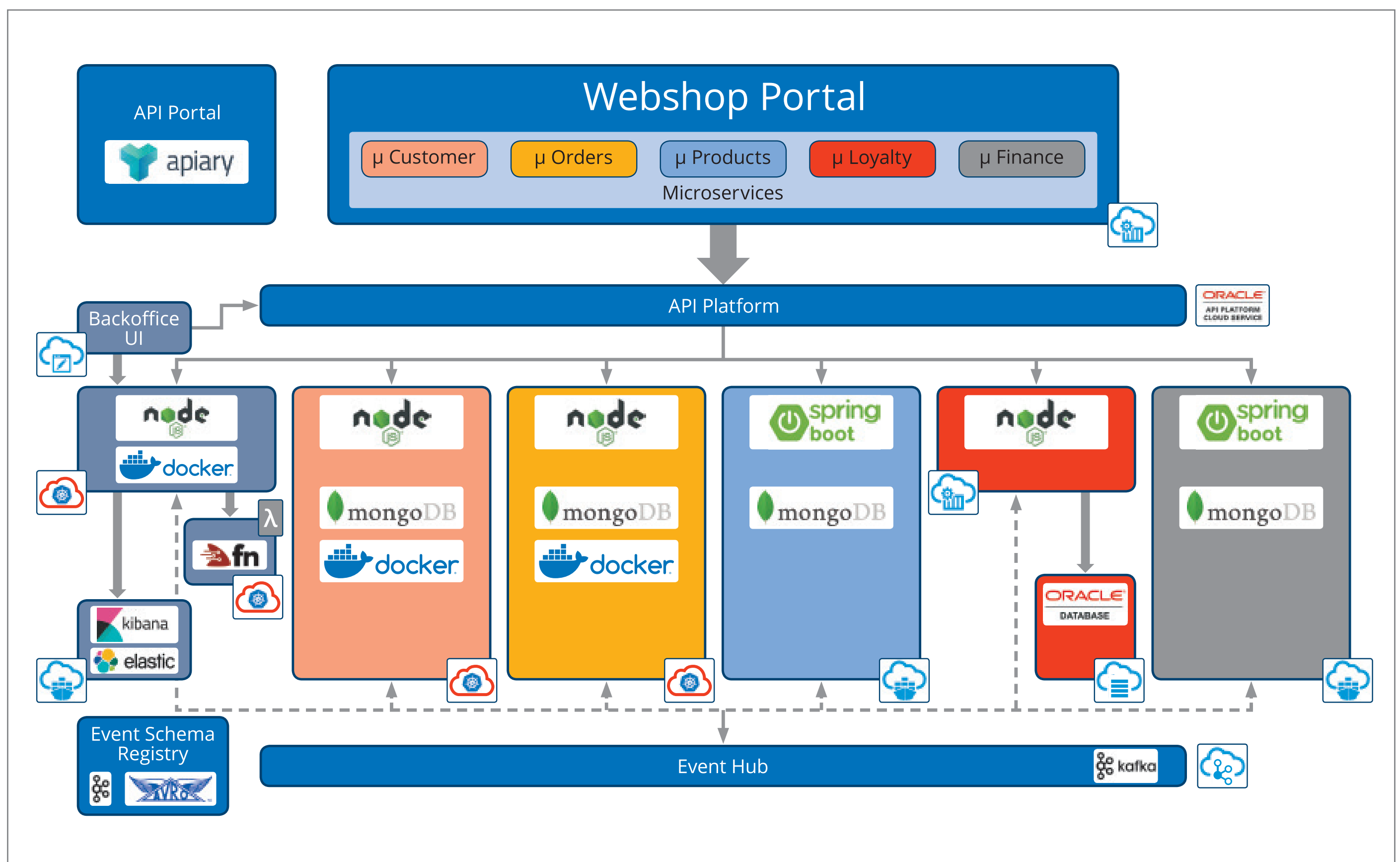

Abbildung 2: Gesamt-Architektur der Webshop-Lösung

Teammitglied war für die Umsetzung einer bestimmten Business Capability, also für jeweils einen Microservice, zuständig. *Abbildung 2* zeigt die Gesamt-Architektur sowie die verwendeten Technologien im Überblick.

Wie der *Abbildung 2* entnommen werden kann, sind bei der Umsetzung viele verschiedene Technologien zum Einsatz gekommen, um den Webshop mit seinen sechs Microservices „Logistics", „Customers", „Orders", „Products", „Loyalty" und „Finance" umzusetzen. Als Laufzeit-Umgebung für die Microservices dienten verschiedene Cloud Services: Oracle Application Container Cloud Service, Oracle Container Cloud Classic und Oracle Container Engine for Kubernetes.

Die Interaktion zwischen den Microservices erfolgt eventbasiert über den Oracle Event Hub Cloud Service. Um die Verbindlichkeit der Event-Definitionen sicherzustellen und die Abstimmungen zwischen den Teams zu vereinfachen, kommt eine Avro Event Schema Registry zum Einsatz *[7]*.

Da die Microservices von UIs verwendet werden sollen, müssen sie REST-APIs anbieten, die über den Oracle API Platform Cloud Service (APIP CS) nach außen exponiert werden. Der Einfachheit halber und da im ersten Schritt nur eine Web UI bedient werden musste, wird auf API-Ebene nicht zwischen Multi- und Single-Purpose-APIs unterschieden.

Beim Webshop-Portal, das auf Oracle JET basiert, handelt es sich um keine klassische, monolithische Web-Applikation. Vielmehr stellt es nur den Rahmen zur Verfügung, der die übrigen Microservice-spezifischen UIs einbindet, was maximale Flexibilität bei Änderungen bedeutet. Wenn beispielsweise aufgrund technischer Anpassungen ein Deployment des Microservice „Finance" notwendig wird, kann dieses jederzeit durchgeführt werden, ohne die übrigen Services zu beeinträchtigen. Benutzer können also weiterhin den Produktkatalog durchsuchen oder Bestellungen durchführen; nur im „Finance"-Bereich kommt es kurzfristig zu Einschränkungen. Die Gesamt-Architektur ist also sehr flexibel aufgebaut. Sie besteht aus Einzelkomponenten, die auf horizontaler Ebene voneinander unabhängig sind. Auf diese Weise kann sehr agil auf sich ändernde Fachanforderungen reagiert werden.

„API first"-Entwicklung

Intuitive APIs sind kritische Erfolgsfaktoren moderner Software-Architekturen. Sie sollten einfach zu bedienen, schwer zu missbrauchen, benutzer- sowie wartungsfreundlich und konsistent definiert sein. Kurzum: Gutes API-Design ist wichtig für die Akzeptanz der Anwender und damit äußerst relevant für die Nutzung eines API. Wie bei der UX für UIs muss man sich also auch hier genauer ansehen, wie das API vom Konsumenten verwendet wird. Deshalb starten wir die Entwicklung nicht etwa mit der Implementierung der Backend-Logik, sondern mit dem Design des API.

Ein „API first"-Design-Ansatz ist für die Flexibilität und Agilität während des gesamten API-Lebenszyklus unerlässlich. Darüber hinaus ermöglicht „API first" die kollaborative Zusammenarbeit verschiedener Stakeholder an einer API-Definition und entkoppelt so API-Implementierung, UI- und Backend-Service-Entwicklung. Dieser Zusammenhang wird in *Abbildung 3* gezeigt.

Für die Microservices des Webshops wurde also zunächst das API beschrieben. Das Team setzte dafür die Oracle Apiary Plattform ein, über die das zugehörige API wahlweise mit Swagger oder API Blueprint beschrieben werden kann. Der Vorteil von Apiary ist, dass das API nach Fertigstellung der Beschreibung durch die Plattform direkt in einer Mock-

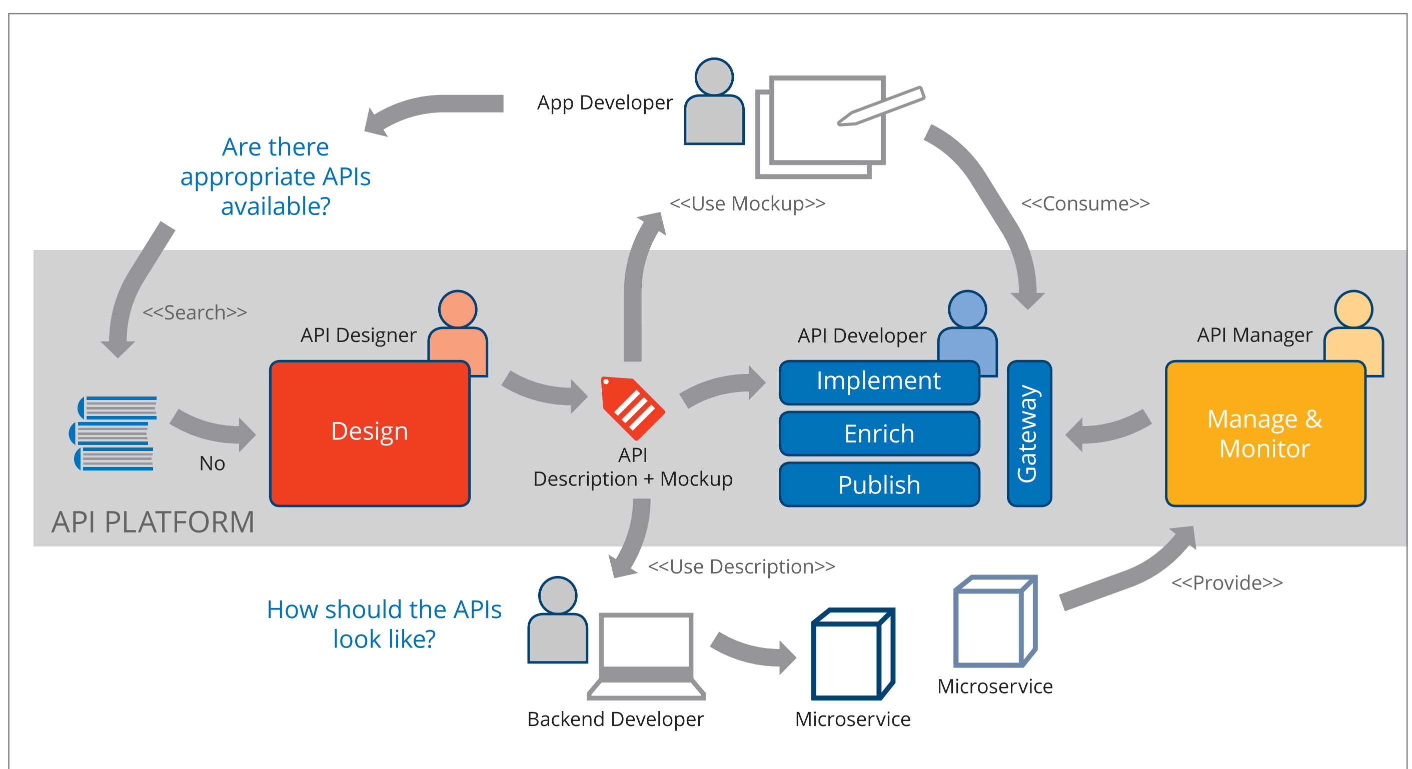

Abbildung 3: „API first"-Design-Ansatz

Variante zur Verfügung gestellt wird und verwendet werden kann.

Wie in *Abbildung 3* dargestellt, können nun App Developer, API Developer und Backend Developer direkt und völlig unabhängig voneinander mit der Implementierung der Web-UI, des API und der Backend-Service-Implementierung starten. Da die API-Beschreibung so allen Stakeholdern bereits zu einem sehr frühen Zeitpunkt zur Verfügung steht, sind Änderungen am API einfach und ohne großen Aufwand möglich. Dank der kurzen Feedbackzyklen kommt man schnell zu einer guten API-Usability.

Test-Automatisierung

Das Thema „Test-Automatisierung" spielt im Kontext von DevOps eine wichtige Rolle. Um Änderungen möglichst schnell produktiv zur Verfügung stellen zu können, ist eine hohe Test-Abdeckung notwendig. Bei der Entwicklung der Backend-Logik der Microservices schrieb das Team Unit-Tests, die dann zum Build-Zeitpunkt mithilfe entsprechender Build-Management-Tools wie Apache Maven automatisiert ausgeführt wurden. In diesem Zusammenhang stellte sich allerdings die Frage, ob und wie es möglich sein würde, Tests für API-Definitionen zu automatisieren. Denn schließlich soll ja auch sichergestellt werden, dass sich ein Backend Service konform zur API-Definition verhält.

Genau solche Tests lassen sich mit Dredd *[8]* automatisieren, einem HTTP-API-Testing-Framework, das Tests gegen ein API ausführen kann. Die lokale Installation des Frameworks erfolgt über NPM. Im Anschluss daran können per „dredd init"-Kommando eine „dredd.yml" Datei erzeugt und darin die Test-Details für das entsprechende API definiert werden, etwa der HTTP-Endpunkt des zu testenden API. Output ist dann ein entsprechender Test-Report auf der Kommandozeile. Darüber hinaus können die Test-Ergebnisse auch in der Apiary Console eingesehen werden.

Build- und Delivery-Automatisierung

Als Build- und Delivery-Plattform kam bei der Webshop-Lösung Oracle Wercker *[9]* zum Einsatz, eine Automatisierungsplattform für den Build und die Bereitstellung von Microservice- und Container-basierten Applikationen. Um Wercker für den Build und das Deployment einer Container-basierten Applikation nutzen zu können, muss die Plattform mit dem GitHub-Repository der zu bauenden Applikation verbunden sein. Im Anschluss daran kann der Build und Delivery Workflow über eine Sequenz sogenannter „Pipelines" definiert werden.

Was innerhalb der einzelnen Pipelines passiert, wird in einer YAML-Datei definiert, der „wercker.yml", die im Git-Repository abzulegen ist. Es gibt einen Workflow, der zunächst den Container baut, diesen dann in einer Registry registriert und ihn im Anschluss einrichtet. Darüber hinaus können noch weitere Pipelines in den Workflow aufgenommen werden, zum Beispiel um die Dredd-Tests auszuführen, bevor der Service ausgerollt wird.

Fazit

Wie der Artikel zeigt, sind die Herausforderungen bei der Entwicklung von Microservices-basierten Applikationen mannigfaltig. Eine zentrale Rolle spielen APIs, um die externe Kommunikation abzubilden, sowie ein konsistenter DevOps-Ansatz. Die Vorgehensweise bei der Entwicklung ist ein entscheidender Faktor; der „API first"-Ansatz hilft dabei, die Entwicklung möglichst effizient zu gestalten.

Anhand einer beispielhaften Webshop-Entwicklung wurde skizziert, worauf es bei der Umsetzung moderner Applikationen in Bezug auf Microservices ankommt. Da die Teammitglieder über fünf verschiedene Länder verteilt waren, konnten wir zudem einen ersten Eindruck bezüglich der organisatorischen Aspekte bekommen, die mit einer Microservices-Implementierung einhergehen. Die Team-Kommunikation lief teilweise über Slack; im Laufe des Projekts, das sich über zwei Monate erstreckte, wurden mehr als dreitausend Nachrichten ausgetauscht! Ein Indikator dafür, wie groß der Abstimmungsbedarf selbst in einem solch kleinen Use Case sein kann.

Weitere Herausforderungen technischer Natur werden uns zukünftig noch weiter beschäftigen. Eine Frage, die intensiv diskutiert wurde, war zum Beispiel: „Wie gestalte ich eine Choreographie für die unterschiedlichen Microservices, um einen Bestellprozess abbilden zu können?" Gedanken hierzu finden sich unter *[10]*.

Quellen

[1] James Lewis, Martin Fowler: „Microservices – A Definition of this New Architectural Term", 2014: *https://www.martinfowler.com/articles/microservices.html*

[2] OMESA Group: „Open Modern Software Architecture Project", OMESA Website, 2017: *http://omesa.io*

[3] OMESA Group: „Capabilities Service Implementation", OMESA Website, 2017: *http://omesa.io/serviceimplementation*

[4] OMESA Group: „Capabilities API", OMESA Website, 2017: *http://omesa.io/apilayer*

[5] Sam Newman: „Pattern: Backends For Frontends", Blog des Autors, 11/2015: *https://samnewman.io/patterns/architectural/bff*

[6] Anne van Kesteren: „Cross-Origin Resource Sharing", W3C Recommendation, 1/2014: *https://www.w3.org/TR/cors*

[7] Apache Software Foundation: „Welcome to Apache Avro!", Projektdokumentation, 2012: *https://avro.apache.org*

[8] „Dredd – HTTP API Testing Framework", Projektdokumentation 5/18, *http://dredd.readthedocs.io/en/latest*

[9] Oracle + Wercker: „Increase developer velocity …", Oracle 2018: *http://www.wercker.com*

[10] Luis Weir: „Is BPM Dead, Long Live Microservices?", Blog des Autors, 2/2018: *http://www.soa4u.co.uk/2018/02/is-bpm-dead-long-live-microservices.html*

Sven Bernhardt
sven.bernhardt@opitz-consulting.com