Microprofile

"Optimizing Enterprise Java for a microservices architecture"

Suhay Sevinc und Börn Beha

Zusammenfassung—Dieser Artikel beschäftigt sich mit dem Einsatz von Java-EE in Microservices-Umfeld. Dabei bietet Java-EE alle benötigten Features an, sodass Microservices auf Basis von Java implementiert werden können. Es stellt sich heraus, dass Java-EE aus Sicht der angebotenen Funktionalitäten kein Hindernis darstellt. Lediglich die Automatisierung des Build- und Deployment-Vorgangs stellt hier ein langfristiges Problem dar. Dies ist auf den erzeugten Overhead beim Deployment zurückzuführen. Aufgrund dessen stellt sich der Einsatz von Continuous Delivery als schwierig heraus, da dies eine essentielle Anforderung von Microservices darstellt. Als Lösung für dieses Problem wird Microprofile in Betracht gezogen, welches eine Umsetzung von leichtgewichtigen Microservices ermöglichen soll. Das Framwork befindet sich im Anfangsstadium und es sind nicht alle benötigten Features implementiert.

Index Terms—Computer Society, IEEE, IEEEtran, journal, LaTeX, paper, template.

1 Introduction

Der Hype um den Modularisierungsansatz Microservices dauert an. Microservice-basierte Architekturen zeichnen sich durch viele Charakteristika aus. Neben der losen Kopplung und der moderaten Größe der Komponenten fokussiert sich dieser Ansatz vor allem auf eine erhöhte Flexibilität bei der Entwicklung und dem Deployment. Jede Komponente soll als eigene Einheit angesehen werden können, welche selbständig (weiter-) entwickelt und vor allem unabhängig von anderen Komponenten deployt werden kann. Durch Automatisierung soll die Software schnell und sicher über eine Continuous-Delivery-Pipeline in Produktion gebracht werden [5]. Allerdings hat Java EE in Bezug auf Microservices einen schlechten Ruf. Die agile Entwicklung und die kurzen Zyklen, bis ein Release ausgerollt werden kann, passen nicht zu den Spezifikationen des Enterprise-Umfelds. Java EE wird bevorzugt in großen Unternehmensanwendungen eingesetzt, welche oft komplexe Prozesse und unflexible Organisations- und Kommunikationsstrukturen enthalten [1]. Dies widerspricht allerdings den Anforderungen an Microservices, welche durch ihren technologischen Ansatz sogar ein völliges Umstrukturieren der Architektur sowie Organisation nach sich ziehen können [6]. Obwohl Java EE ursprünglich für stark verteilte Anwendungen sowie fachlich orientierte Systeme ausgelegt wurde und es sämtliche Funktionalitäten bietet, welche zur Implementierung solcher Komponenten benötigt werden, gründeten eine Sammlung verschiedener namenhafte Anbieter und Organisationen MicroProfile.
Durch die Definition einer Enterprise Java-Spezifikation, welche Microservice-Muster adressieren soll, wird eine Optimierung von Enterprise Java für Microservice-basierte Architekturen versprochen [2]. Diese wissenschaftliche Arbeit beschäftigt sich mit der Frage, warum überhaupt eine Optimierung gebraucht wird und warum Java EE tatsächlich kein geeignetes Werkzeug für diesen Architekturansatz ist. Dabei wird zuerst untersucht, wie Java EE die priorisierten Eigenschaften von Microservices umsetzt und welche Probleme es dabei gibt. Anhand dieser Probleme wird erläutert, warum MicroProfile ins Leben gerufen wurde und was dieser Ansatz an Enterprise Java verbessert.
Exemplarisch wichtige Arbeiten, welche zu diesem Artikel in Beziehung stehen, gibt es keine. Dieses Papier stützt sich auf die Bücher Microservices [5] und Continuous Delivery [6] von Eberhard Wolff, welche umfangreiche Grundlagen bezüglich Microservices vermitteln. Das hier beschriebene Vorgehen zur Untersuchung der Technologie kann mit den Beiträgen von Lars Röwekamp auf jaxenter.de verglichen werden. In dessen Beitrag wird auf das Problem von Java EE eingegangen.

2 Java EE und Microservices

Aus rein technologischer Sicht bietet Java EE alles, was für eine Microservice-basierte Architektur erforderlich ist. Das wird bereits klar, wenn die verfügbaren APIs betrachtet werden [1]. Microservices ergeben sich unter Berücksichtigung folgenden Eigenschaften, welche zudem Indikatoren für die Größe darstellen [5]. em Teamgröße

- Modularisierung
- Ersetzbarkeit
- Transaktionen und Konsistenz
- Infrastruktur
- Verteilte Kommunikation

Soll nun ein Microservice mit Java EE implementiert werden, könnten diese Indikatoren ebenfalls herangezogen werden. Die Anzahl der Teammitglieder beschränkt die Größe der Komponente. Sie beinhaltet dabei so viel Logik, dass sie von einem Team entwickelt und betrieben werden kann. Der Dienst lässt sich so unabhängig von anderen Teams implementiert und in Produktion bringen. Auch die Komplexität ist handhabbar, wenn der Service als Modul entwickelt wird und lässt sich somit auch ersetzen, sollte dies von Nöten sein. Die Transaktion, welche die ACID-Eigenschaften verfolgt, kann ebenfalls entsprechend implementiert werden. Die Kommunikation über leichtgewichtige Mechanismen ergibt sich durch diesen Ansatz [5]. Da

sie Teil eines verteilten Systems sind, bedienen sich über die verteilte Kommunikation einer anderen Fachlichkeit oder stellen anderen Services die eigene zur Verfügung. Microservices sollen eine geschlossene Fachlichkeit abbilden (Bounded Context), um sich eindeutig von anderen Services abzugrenzen.
Diesen Anforderungen genügt Java EE über CDI (Context and Dependency Injection), JAX-RS (Java API for RESTful Web Services) und der Java Persistence API bereits. Selbst NoSQL-Datenbanken können über entsprechende Bibliotheken an das System angebunden werden. Auch eine eigene Benutzeroberfläche können in solche Anwendungen integriert werden, wie es bei Microservices priorisiert wird. Es ist also nicht der Fall, dass ein Service, welcher über das Java EE Framework entwickelt wurde, automatisch eine aufgeblähte Mehrschichtenarchitektur mit sich bringt, welche entsprechende Methodenaufrufe nur über Dependency Injection Schicht für Schicht weiter delegiert. Durch verwenden entsprechender Features lassen sich sehr effiziente und schlanke Architekturen realisieren [1].
Microservice-basierte Anwendungen sind Applikationen, die sich aus einer Reihe von kleinen Komponenten zusammensetzen. Diese durchlaufen dabei alle ihre eigenen Prozesse. Somit ist ein weiterer, zu berücksichtigender Punkt, die Infrastruktur. Jede Einheit muss unabhängig deployt werden können. Hierfür gibt es die Continuous-Delivery-Pipelne, die durch eine weitgehende Automatisierung dafür sorgt, dass Software bereitgestellt werden kann. Jeder Dienst benötigt somit eine eigene Infrastruktur, um ihn auszuführen. Diese kann auch Datenbanken oder Application Server beinhalten [5]. Hier ergibt sich allerdings ein Problem mit Enterprise Java.

3 Das Problem mit Java EE

Die oben aufgeführten Punkte sind Ansätze für Continuous Delivery. Diese Disziplin erfolgt durch eine weitgehende Automatisierung entsprechender Prozesse. Das eigentliche Problem liegt nicht in der zu verwendenden Technologie, es geht um die potenzielle Automatisierung des Development Lifecycle. Das Vorgehen für Build und Deployment, welches in Java EE-Anwendungen vorgesehen ist, geht über das Zusammenpacken der Komponenten, die dann deployt werden. Eine Skalierung, die eine anfallende Last pro Fachlichkeit ausgleicht, kann somit nur sehr umständlich erreicht werden. Bei Anwendungen, die auf Microservices basieren, geht es teilweise um tausende Serverinstanzen mit zigtausend Deployments pro Jahr (laut Amazon sogar etliche Millionen). Dies mit Java EE zu bewerkstelligen stellt sich dabei als sehr schwer heraus. Dafür sind entsprechende Server nicht schnell genug, denn der durch Java-EE anfallende Overhead an zu unterstützenden APIs und Features ist zu groß bzw. die Anwendungsserver einfach zu schwerfällig [1].
Leichtgewichtige Server, die nur Bestandteile mit sich bringen, welche für den Service benötigt werden, würden Abhilfe schaffen [1]. Führende Application-Server-Hersteller (WildFly Swarm, TomEE etc.) sind bereits daran entsprechende Varianten zu entwickeln [2]. Dabei sollen Bestandteile und Funktionalitäten des Servers in den Service eingebunden werden. Bisher wurde der Service lediglich auf dem Server deployt [1]. Abbildung (...) illustriert dies. Somit

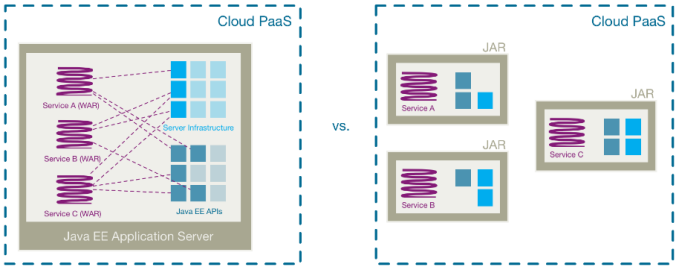

Abbildung 1. Bestandteile und Funktionalitäten im Service [1]

erhält man einen fachlich abgekapselten Service, der zudem seine eigene Laufzeitumgebung mit sich bringt. Diese Umgebung kann dabei entsprechend auf den Service angepasst werden, wodurch der Overhead hier auch drastisch reduziert werden kann. Sobald dieser Service nun deployt wird, kann er gestartet und ausgeführt werden. Er ist autark und läuft als eigener Prozess. Allerdings ist dieser Ansatz noch etwas zu grobgranular. Von einem Self-contained System (SCS) ist hier somit eher die Sprache [1]. Dieser Ansatz teilt sich zwar eine Vielzahl von Konzepten mit Microservices (Isolation, unabhängige Einheiten, fachliche Trennung, Technologiefreiheit, keine zentrale Infrastruktur), besitzt jedoch noch einige Unterschiede zum feingranularen Ansatz der Microservices. Wie eben bereits hervorgegangen ist, weist bereits die Größe ein Unterschied auf. Ein System besitzt normalerweise weniger SCS als Microservices. Ein wichtiger Aspekt ist die Kommunikation zwischen den Komponenten. Microservices können untereinander kommunizieren. SCSs sollten dies idealerweise nicht. Auch bringen Microservices oft ihre eigene Benutzeroberfläche mit sich, während sich SCSs eine gemeinsame teilen. Es wird an dieser Stelle also nicht das gewünschte Problem gelöst. SCSs sind eher für Architekturen größerer Projekte gedacht [3]. Sollen noch unabhängigere, kleinere Komponenten entwickelt werden, die auch mit Continuous Delivery arbeiten, muss noch ein Schritt weitergegangen werden.

4 MicroProfile

Optimierungen erfolgen bisher durch recht triviale Ansätze. Ist der Anwendungsserver zu groß werden lediglich die benötigten Komponenten verwendet, die zwingend für den Microservice gebraucht werden. Diese Ansätze haben allerdings immer noch Schwachstellen, wie bereits aus dem obigen Kapitel hervorgegangen ist. Java EE MicroProfile wurde somit angekündigt, welches bei seiner initialen Version lediglich einen minimalen Satz an APIs zur Verfügung stellt. Somit enthielt die erste Version von MicroProfile JAX-RS für die Verwendung von REST, CDI und JSON-P. Dies reichte für einen Microservice ohne eigene Benutzeroberfläche aus. Bei Bedarf können zusätzliche APIs hinzugefügt werden.

Aktuell gibt es die Version 1.2 von Microprofile, welches die Features in Abbildung 2 aufzeigt.

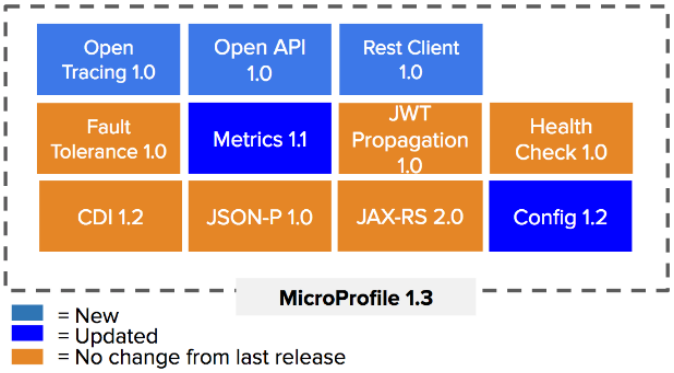

Abbildung 2. Microprofile-1.3-Features [4]

4.1 Config-API

Die Confg-API trennt Anwendungslogik und Konfiguration. Dadurch kann ein Microservice dynamisch auf die Laufzeitumgebung angepasst werden. Bei diesem Ansatz stellt die Konfiguration von heterogenen Quellen, wie beispielsweise Umgebungsvariablen und Datenbanken, eine Herausforderung dar. Dabei können diese Konfigurationen in unterschiedlichen Datenformaten vorliegen, was die Administration zusätzlich erschwert. Bei statischen Konfigurationswerten genügt eine Initiierung beim Start des Prozesses. Hingegen beim dynamischen Konfigurationen sollten administrative Maßnahmen ergriffen werden, welche zur Laufzeit erfolgen müssen, wie zum Beispiel das Prüfen auf Aktualität und Korrektheit der Konfigurationen. Diese Problematik greift Config-API auf. Sie ermöglicht die Zugriffsvereinheitlichung auf unterschiedliche Konfigurationen, die wiederum priorisiert und dadurch gezielte Konfigurationsüberschreibung möglicht wird. Standardwerte werden in einer `microprofile-config.properties` abgelegt .Diese können dann bedarfsgerecht angezogen und in der Umgebung überschrieben werden. Auch das Einbinden von weiteren Quellen ist möglich. Die Zugriffsverwaltung auf die Konfigurationen können auf zwei Wegen geschehen: ConfigProvider und ConfigBuilder. Um diese umsetzen zu können benötigt es davor die Instanziierung der Config-Klasse. Der Builder erlaubt die individuelle Anpassung der Konfiguration und die Instanz wird nicht gecached. Beim ConfigProvider gilt es zu erwähnen, dass beim Aufruf der Methode getConfig(), die zurückgelieferte Konfigurations-Instanz - aus Effizienzgründen - gecached wird. Dies müsste bei paralleler Programmierung beachtet werden. Der Zugriff kann allerdings auch über CDI-Injection-Annotationen geschehen. Sobald ein Konfigurationswert durch den Schlüssel nicht gefunden wird, reagiert das System mit einer NoSuchElement-Exception. Bei CDI-Injection wird Deployment-Exception geworfen. Auch optionale Konfiguration kann durch die Methode getOptional-Value().orelse() in Betracht gezogen, sodass im Zweifelsfall eine andere Konfiguration angezogen werden kann.

4.1.1 Just-In-Time-Konfiguration

Damit ein Microservice bei Konfigurationsänderung nicht neugestartet werden muss, bietet die Config-API ein Mechanismus an, sodass Konfigurationswerte dynamisch zur Laufzeit geladen werden können. Um den aktuellen Konfigurationswert zu erhalten und nicht den Wert zum Zeitpunkt der Injection, sollte, muss ein Provider injiziert werden. Dadurch wird immer der aktuelle Wert angezogen. Just-in-Time steht in diesem Zusammenhang für die Aktualität des Konfigurationswertes. Die Aktualisierung der Werte innerhalb der Quelle und darauf bezogen das Refresh ist dem Autor der ConfigSource-Klasse überlassen.

4.1.2 Converter

Durch den Converter können die Konfigurationswerte, welche ausschließlich aus Strings besteht, in Javatypen konvertiert, sodass auch andere Typen als Strings verwendet werden können. Bereits für einige Javatypen gibt es den Build-in-Converter. Es werden Typen wie beispielsweise Boolean, Integer, Long, Double, URL und LocalDateTime unterstützt.

4.2 HealthCheck-API

Die HealthCheck-API ermöglicht, dass der aktuelle Status eines Services abgefragt werden kann. Dabei fallen Antworten im Form von 'Up' sowie 'Down' an. Falls der Service nicht verfügbar sein sollte, können hier Maßnahmen, wie beispielsweise das Neustarten, zur Erhaltung der Systemstabilität ergriffen werden. Um diese Funktion nutzen zu können, sollte die entsprechende Schnittstelle implementiert werden [5]. Des Weiteren wird durch die Annotation `Health` eine automatische REST-API-Funktion zur Verfügung gestellt. Auf Basis dieser Funktion können Monitoring- und Management-Tools durch Rest-Calls automatisiert überprüfen, ob ein Neustart erforderlich ist. Diese API sieht den Einsatz in Container-basierten Umgebungen, statt manuellen Gebrauch, vor (Machine-To-Machine) [6].

4.3 FaultTolerance-API

Die FaultTolerance-API hat den Zweck, eine verbesserte Systemstabilität zu erreichen. Des Weiteren können Resilience-Patterns wie beispielsweise Fallback, Timeout oder Bulkhead auf Basis von Java-Annotationen umgesetzt werden. Aber auch die Ausprogrammierung dieser Patterns ist möglich. Dadurch erhöht sich die Verfügbarkeit eines Microservices [7].

4.4 Metrics

Gegensatz zur HealthCheck-API dient die Metrics-API dazu, fein granulare Systeminformationen und kritische Systemparameter zur Laufzeit zu überwachen. Diese Überwachung geschieht durch Softwareagenten, wodurch Prognosen über künftige Serviceverhalten erstellt werden kann. Durch Einsatz von Contexts und Dependency Injection kann diese Funktionalität verwendet werden. Auf diese Weise lässt sich zum Beispiel auf eine erhöhte Auslastung mit dem Start neuer Service-Instanzen oder proaktiv auf sich abzeichnende Engpässe reagieren [6].

4.5 Json Web Token (JWT) Propagation

Diese API bietet die Authentifizierung und Autorisierung innerhalb eines Microservices an. Mit entsprechenden Authorization-Header (mit Token) sind beispielsweise Token-Gültigkeit, den Token-Aussteller und Token-Autoritär möglich. Innerhalb eines JAX-RS-Containers können durch Annotationen auf Instanz des JWT sowie Claims zugegriffen werden. Hier lassen sich Java-EE-Funktionalitäten wie zum Beispiel RolesAllowed gemeinsam mit JWT nutzen. Dies ist aufgrund von automatisches Mapping auf den Java EE SecurityContext möglich [8].

4.6 OpenAPI und OpenTracing-API

Durch OpenAPI können OpenAPI-V3-Dokumente aus JAX-RS-Applikationen erstellt und als unabhängige Beschreibungsformat für REST APIs genutzt werden. Diese API hat den Zweck, eine automatisierte Generierung von Microservices-APIs bereitzustellen. Das erlaubt die Verwaltung über API-Management-Werkzeugen. OpenTracing-API wird versucht ein Standard bzgl. verteiltes Tracing zu setzen. Hierdurch können Requests nachverfolgt werden [9].

5 Conclusion

Fazit vorerst nur kopiert!!

Java EE bietet aus rein technologischer Sicht alles, was es zur Entwicklung von Microservices braucht. Trotzdem ist der Enterprise-Java-Standard für viele Microservice-Neueinsteiger nicht unbedingt die nächstliegende Wahl. Neben der eigentlichen Entwicklung ist vor allem die vollständige und effiziente Automatisierung sämtlicher Phasen des Software Development Lifecycles für eine erfolgreiche Einführung von Microservices von Belang. Und genau hier haben Java EE und die zugehörige Runtime klare Schwächen. Der Application Server ist einfach zu schwergewichtig, als dass tausende Instanzen permanent neu deployt werden könnten. Dies gilt nicht nur für die traditionsbehafteten Dinos am Markt, sondern auch für die auf das Web Profile ausgerichteten Leichtgewichte.

Die notwendige Automatisierung ist mit der gewünschten Effizienz nur dann realistisch, wenn die Server noch weiter abspecken. Nach dem Vorbild von Dropwizard und Spring Boot tauchen in den letzten Monaten daher vermehrt Lösungen auf, mit denen sich die eigene Java-EE-Anwendung oder der Java-EE-basierte Microservice mit genau den Bestandteilen bootstrapen lässt, die für den Service benötigt werden. Das Resultat sind schlanke, schnell deploybare Microservices auf Basis von Java EE. Dank guter Integration von im Microservice-Umfeld etablierten Open-Source-Lösungen à la Netflix OSS und Co. kommen dabei auch das Management und Monitoring der Microservices nicht zu kurz – sowohl auf dem eigenen Server als auch in der Cloud.

Es ist also mit Java EE durchaus möglich, neue Features in Form von fachlich orientierten Microservices mit hoher Qualität zu implementieren und schnell an den Markt zu bringen. Time-to-Market und Java EE müssen sich, richtig angegangen, nicht widersprechen, ganz im Gegenteil. Dank Standard kann auf jahrelang aufgebautes Fachwissen zurückgegriffen werden, zumindest dann, wenn die Java-EE-Entwickler bereit sind, den einen oder anderen alten Zopf abzuschneiden und sich auf neue, spannende Welten einzulassen. Denn, wie bereits zu Anfang angedeutet: Nicht Java EE ist das Problem, sondern die Zeit, in der das Framework groß geworden ist.

Ein eigener JSR und eine damit verbundene Überführung des MicroProfile in den Java-EE-Standard ist zwar durchaus eine wichtige Option, steht aber nicht im Fokus der Bemühungen. Stattdessen geht es vielmehr darum, Einigkeit der involvierten Player zu signalisieren und gemeinsam mit der Community einen De-facto-Standard zu generieren. Gemeinsam mit der Community? Genau! Dass man das ganze Thema nicht nur aus Sicht der Application-Server-Hersteller betrachtet, sondern durchaus an der Meinung der Community interessiert ist, zeigt eine entsprechende Umfrage auf der Webseite von www.microprofile.io. Hier wird jeder aufgefordert, die aus seiner Sicht wichtigsten Aspekte eines Microservice (z. B. Start-up Time, Metrics, Disk Space, Circuit Breaker) sowie die für die Implementierung von Microservices sinnvollen Java-EE-APIs zu nennen.

Mit der Initiative microprofile.io und dem zugehörigen MicroProfile ist etwas entstanden, das eine reale Chance darauf hat, Java EE im Umfeld von Microservices zu etablieren. Glaubt man den bisherigen Ankündigungen, haben die Initiatoren es verstanden, das Beste aus Standard und Community-driven in einem Ansatz zu vereinen: Herstellerunabhängigkeit und damit verbunden Portabilität bei gleichzeitig schnellen Reaktionszeiten.

Wichtig zu verstehen ist, dass die Initiatoren von www.microprofile.io nicht den Glauben in Java EE verloren haben. Ganz im Gegenteil, Red Hat hat erst vor Kurzem noch einmal sein weiteres Engagement im Umfeld von Java EE 8 betont. Eine spätere Überführung des MicroProfiles in den Java-EE-Standard ist also durchaus denkbar und gewünscht. Wichtig ist aber auch, dass zukünftig alternative Wege im Enterprise-Java-Umfeld eine wichtige Rolle spielen werden, die sich mit hoher Wahrscheinlichkeit jenseits des Java Community Process in seiner derzeitigen Form bewegen.

Jeder ist aufgefordert, diese alternativen Wege möglichst aktiv und intensiv mitzugestalten. Im konkreten Fall kann man dies schon heute über die MicroProfile Google Group. In diesem Sinne, zum Ende ausnahmsweise einmal nicht Stay tuned, sondern: Participate! Mit Microprofile können Microservices auf Grundlage von Java konzipiert und implementiert werden. Aktuell wird das Framework weiterausgebaut, sodass die Diskrepanz zwischen Java-Ee und den verschiedenen Microservices-Framework-Ansätzen kompensieren. Das nächste Update Microprofile 2.0 kommt bereits im März 2018 mit den Features in Abbildung 3. Mit dem Hype um Microservices wird die Notwendigkeit für Microprofile noch weiter ansteigen [10].

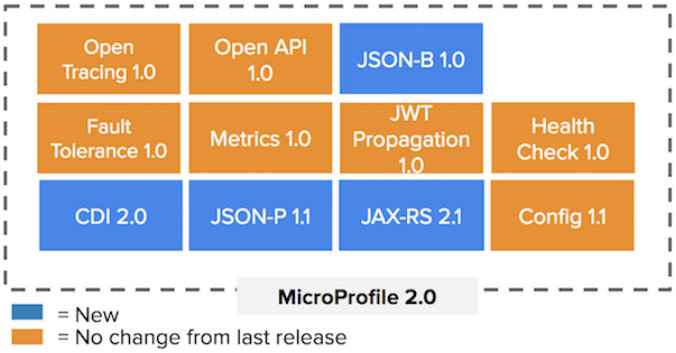

Abbildung 3. Microprofile-2.0-Features [4]

LITERATUR

[1] jaxenter, "Java ee trifft microservices: Elefant im porzellanladen?" 2016. [Online]. Available: https://jaxenter.de/java-ee-trifft-microservices-elefant-im-porzellanladen-38432

[2] Lars Röwekamp, "Java ee in a box: Microprofile – ein neuer community-driven standard," 2016. [Online]. Available: https://jaxenter.de/java-ee-microprofile-42877

[3] self-contained services, 17.12.2017. [Online]. Available: http://scs-architecture.org/

[4] Microprofile, 2017. [Online]. Available: https://microprofile.io/blog/2017/10/eclipse-microprofile-1.2-available

[5] Lars Röwekamp, "Microprofile unter der lupe, teil 1: Config api," 2017. [Online]. Available: https://www.heise.de/developer/artikel/MicroProfile-unter-der-Lupe-Teil-1-Config-API-3928886.html

[6] ——, 2017. [Online]. Available: https://www.heise.de/developer/meldung/Java-MicroProfile-1-2-hat-vier-neue-APIs-an-Bord-3849320.html

[7] ibm, 2017. [Online]. Available: https://developer.ibm.com/wasdev/blog/2017/09/01/microprofile-configuration-api-beta-liberty/

[8] Lars Röwekamp, 2017. [Online]. Available: https://www.heise.de/developer/meldung/Java-MicroProfile-1-2-hat-vier-neue-APIs-an-Bord-3849320.html

[9] Dominik Mohilo, "Microprofile 1.3: Drei neue apis und ein ausblick auf version 2.0," 2018. [Online]. Available: https://jaxenter.de/microprofile-1-3-66024

[10] Lars Röwekamp, "Java: Microprofile 1.3 bringt drei neue apis," 2018. [Online]. Available: https://www.heise.de/developer/meldung/Java-MicroProfile-1-3-bringt-drei-neue-APIs-3932625.html