Microprofile

Optimizing Enterprise Java for a microservices architecture

Björn Beha und Suhay Sevinc

Zusammenfassung—Dieser Artikel beschäftigt sich mit dem Einsatz von Java-EE in Microservices-Architekturen. Dabei bietet sie alle Features an, welche zur Implementierung von Microservices benötigt werden. Aus Sicht der angebotenen Funktionalitäten stellt Java-EE keine Hürde dar. Jedoch ist die Automatisierung des Build- und Deployment-Vorgangs ein langfristiges Problem. Dies ist auf den erzeugten Overhead zurückzuführen. Aufgrund dessen stellt sich der Einsatz von Continuous Delivery, das eine essentielle Anforderung an Microservices-Architekturen darstellt, als schwierig heraus. Als Lösung für dieses Problem wird Microprofile in Betracht gezogen, welches eine Umsetzung von Microservices auf Basis von leichtgewichtigen Java-EE-Funktionalitäten ermöglichen soll. Das Framework bietet bereits einige Funktionen wie beispielsweise HealthCheck an. Sie befindet sich noch im Anfangsstadium und deswegen sind nicht alle benötigten Features implementiert.

Index Terms—Microprofile, Java-EE, Microservices , API, Version 1.3

1 Einführung

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 [1]. 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 [2]. Dies widerspricht allerdings den Anforderungen an Microservices, welche durch ihren technologischen Ansatz sogar ein völliges Umstrukturieren der Architektur sowie Organisation nach sich ziehen können [1]. 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 [3]. 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.
Die Arbeit von Ueda et all beschäftigt sich unter anderem mit der Erstellung eines Benchmarks für ein Microservice basierend auf Java. Der Benchmark zeigt, dass ein Overhead zu beobachten ist. Aus dem Benchmark geht hervor, dass die Performance gedrosselt wird [4]. Dieses Papier stützt sich auf die Bücher Microservices [1] und Continuous Delivery [5] 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 [2]. Microservices ergeben sich unter Berücksichtigung folgenden Eigenschaften, welche zudem Indikatoren für die Größe darstellen ¸

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

Soll ein Microservice mit Java EE implementiert werden, lassen sich diese Indikatoren ebenfalls heranziehen. 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. Microservices sollen eine geschlossene Fachlichkeit abbilden (Bounded Context), um sich eindeutig von anderen Services abzugrenzen. Durch diesen fachlich orientierten Aufbau,

lässt sich der Dienst unabhängig von anderen Teams implementieren, testen 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 [1]. Da sie Teil eines verteilten Systems sind, bedienen sich über die verteilte Kommunikation einer anderen Fachlichkeit oder stellen anderen Services die eigene zur Verfügung.
Diesen Anforderungen genügt Java EE bereits. Microservices lassen sich mit den vorhandenen APIs problemlos erstellen. Über JAX-RS (Java API for RESTful Web Services) lässt sich die Kommunikation der Dienste realisieren. Über Bean Validation (Java EE-Standard für eine schichtenübergreifende Validierung) können Requests auf ihre Korrektheit überprüft werden. Die Nutzlast kann beispielsweise über JSON-P entsprechend umgewandelt werden, um die Übermittlung zu ermöglichen. Die fachliche Logik innerhalb des Dienstes sowie die Datenhaltung lässt sich über CDI (Context and Dependency Injection) sowie JPA (Java Persistence API) realisieren [6]. Selbst NoSQL-Datenbanken können über entsprechende Bibliotheken an das System angebunden werden. Auch eine eigene Benutzeroberfläche lassen sich in solche Anwendungen integrieren, 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 [2].
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 [1]. 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 bzw. dem reinen Entwickeln, es geht um die potenzielle Automatisierung dieser Entwicklung. Auch zur Laufzeit zeigen sich Probleme. Das Vorgehen für Build und Deployment, welches in Java EE-Anwendungen vorgesehen ist, geht über das Zusammenpacken der Komponenten, die dann in einem Application Server deployt werden (siehe Abbildung .1). Die notwendige Infrastruktur wird von diesem Server bereitgestellt [6]. 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 [2].
Leichtgewichtige Server, die nur Bestandteile mit sich bringen, welche für den Service benötigt werden, würden Abhilfe schaffen [2]. Denkbar wäre hier als erster Ansatz das separate Packen der Anwendungen, welche anschließend auf dem Server deployt werden. Die durch den Server können sich die Dienste die zur Verfügung gestellte Infrastruktur teilen. Microservices sollen allerdings alle in einem eigenen Prozess unabhängig voneinander laufen. Diese Unabhängigkeit kann durch den eben genannten Ansatz nicht realisiert werden. Das Deployment eines Service würde das Laufzeitverhalten der anderen Dienste beeinflussen. Fällt der Server aus oder weist ein Fehlverhalten durch einen defekten Service auf, können die anderen Dienste ebenfalls nicht mehr erreicht werden [6].

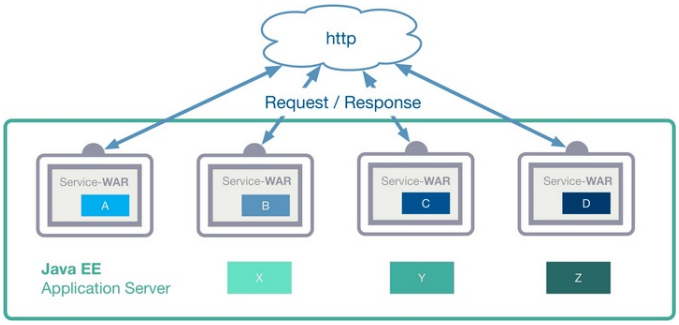

Abbildung 1. Deployment in Application Server [6]

4 Die Optimierung von Java EE

Um Dienste auf Basis von Java EE bezüglich der Unabhängigkeit zu optimieren, wäre es denkbar jede Komponente in einen eigenen Server zu deployen. Jeder Dienst hat somit seine eigene Infrastruktur. Dies wird in Abbildung (2) skizziert. Wie bereits erwähnt handelt es sich bei Microservice-basierten Systemen oft um etliche Service-Instanzen und deployments. Durch die angestrebte Automatisierung über die Delivery-Pipeline wären operative Aufwände wie Deployment, Konfiguration etc. zu stemmen. Automatisierung kann beispielsweise durch das Einbetten in Docker-Images erreicht werden. Allerdings ergeben sich auch hier Probleme zur Laufzeit. Der Ressourcenverbrauch der Dienste treibt die Last auf den Servern entsprechend in die Höhe [6].
Führende Application-Server-Hersteller (WildFly Swarm, TomEE Shades, Payara Micro etc.) sind bereits daran entsprechende Lösungen zu entwickeln [7]. Dabei sollen nur

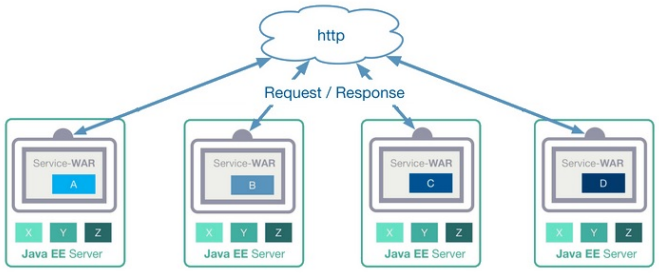

Abbildung 2. Dienste mit eigener Infrastruktur [6]

die benötigten Bestandteile und Funktionalitäten des Servers in den Service eingebunden werden [2]. Eine Kombination aus Server-Komponenten und Service-Geschäftslogik lassen sich daraufhin als ausführbares Programm packen. Dies wird in Abbildung 3 veranschaulicht. Ein fachlich gekapselter Service bringt so seine eigene Laufzeitumgebung mit. Diese Umgebung lässt sich dabei entsprechend auf den Service anpassen, wodurch der Overhead hier auch drastisch reduziert werden kann. Sobald dieser Service deployt wird, kann er gestartet und ausgeführt werden. Er ist autark und läuft als eigener Prozess [2]. Trotz, dass mit diesem Ansatz scheinbar alle Aspekte umgesetzt wurden, bleibt jedoch das fortwährende Einbinden der Server-Komponenten, was den Build-Prozess verlangsamt [6]. Zu-

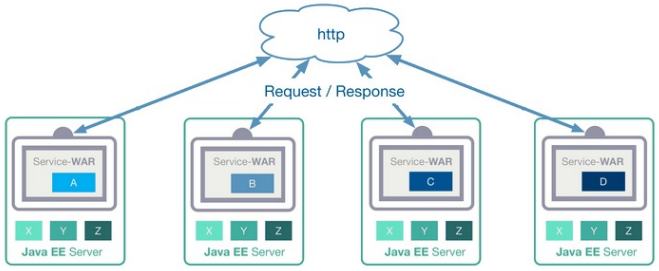

Abbildung 3. Dienste mit eigener Infrastruktur [6]

dem ist dieser Ansatz noch zu grobgranular. Er stellt daher weniger einen Microservice dar, es handelt sich eher um ein Self-contained System (SCS) [2]. 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 [8]. Sollen noch unabhängigere, kleinere Komponenten entwickelt werden, die auch mit Continuous Delivery arbeiten, muss noch ein Schritt weitergegangen werden.

5 MICROPROFILE

Wie aus den obigen Kapiteln hervorging, erfolgen Optimierungen bisher durch triviale Ansätze. Ist der Anwendungsserver zu groß, werden nur die für den Microservice zwingend benötigten Komponenten mit eingebunden. Allerdings weisen Microservices, die auf diese Basis aufbauen noch immer Schwachstellen auf.

MicroProfile verspricht hier eine Optimierung von Enterprise Java für Microservice-Architekturen. Hinter dem Namen MicroProfile stecken einige große Unternehmen, die gemeinsam mit der Community an dieser Lösung arbeiten, um den aktuellen Marktbedürfnissen entgegenzukommen. Es wird versucht einige vorhandene Tools zu nutzen und diese mit neuen Werkzeugen zu verbinden. Dadurch soll eine Basisplattform geschaffen werden, die dem Microservice-Ansatz genügt. Die initiale Version dieser Lösung stellte dabei einen minimalen Satz an APIs zur Verfügung. Aktuell werden die Funktionen angeboten, welche in Abbildung 4 aufzeigt werden.

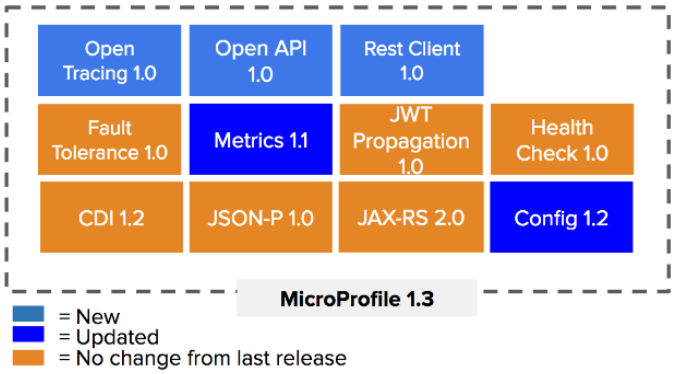

Abbildung 4. Microprofile-1.3-Features [3]

5.1 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 getOptionalValue().orelse() in Betracht gezogen, sodass im Zweifelsfall eine andere Konfiguration angezogen werden kann.

5.1.1 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.

5.1.2 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.

5.2 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 [9]. 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) [10].

5.3 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 [11].

5.4 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 [10].

5.5 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 [12].

5.6 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 [13].

6 Fazit und Ausblick

Wie aus diesem Artikel hervorgeht, bietet Java EE bereits alles, was für die Entwicklung von Microservices benötigt wird. Zumindest, was den technologischen Aspekt angeht. Es können auch qualitativ hochwertige, fachlich orientierte Microservices mit Java EE implementiert werden. Allerdings ist die reine Entwicklung nicht das Problem. Die Automatisierung sämtlicher Phasen, welche bei Microservices in Verbindung mit Continuous Delivery verfolgt wird, kann mit Java EE nicht ausreichend umgesetzt werden. Aufgrund des schwergewichtigen Application Server, weist die Runtime schwächen auf. Es ist schlicht nicht möglich, zigtausend Instanzen kontinuierlich neu zu deployen. Erst wenn der Server nur auf die nötigsten Komponenten reduziert wird, wird eine entsprechende Automatisierung mit der gewünschten Effizienz realistisch. Zwar erreicht man durch diesen Ansatz eher Self-contained Systems, die zwar einige Parallelen zu

dem Microservice-Ansatz aufweisen, allerdings noch zu groß und grobgranular sind. Viele Anbieter wie auch die Gründer von MicroProfile arbeiten daher an Lösungen, mit denen Java EE auf die für den Service benötigten Bestandteile reduziert wird. Somit können auch schlanke und schnell deploybare Microservices auf Grundlage von Enterprise Java konzipiert und implementiert werden. Somit können Teams mit umfangreicher Erfahrung weiter Anwendungen auf Basis von Java EE entwickeln.

Die Schöpfer von MicroProfile wollen hier einen neuen Standard etablieren. Ihre erste Veröffentlichung (MicroProfile 1.0) umfasste daher lediglich JAX-RS 2.0, CDI 1.2 und JSON-P 1.0. Für die Entwicklung der Spezifikationen wird ein Open Source-Ansatz verfolgt. Man arbeitet öffentlich als Gemeinschaft unter Einbezug der Community. Auf der offiziellen Webseite von www.microprofile.io werden Interessierte dazu aufgefordert über verschiedene Diskussionen in der MicroProfile Google Group, bei der Optimierung von Enterprise Java für Microservices zu helfen. Durch diesen Ansatz wird die Entwicklung beschleunigt und ist eine Spezifikation ausreichend ausgereift, wird das Ergebnis zur Standardisierung in Betracht gezogen.

Die Zukunft von MicroProfile bleibt höchst spannend. Aktuell wird das Framework weiter ausgebaut, sodass die Diskrepanz zwischen Java-EE und den verschiedenen Microservice-Frameworks weiter verringert wird. Vor allem da Microservices gerade eine besondere Aufmerksamkeit genießen, steigt die Notwendigkeit für MicroProfile weiter an [14]. Mit dieser Initiative kann Java EE auch ernsthaft im Microservice-Umfeld etabliert werden. Das nächste Update MicroProfile 2.0 wird bereits März 2018 veröffentlicht. Die neuen Features sind in Abbildung 5 aufgeführt.

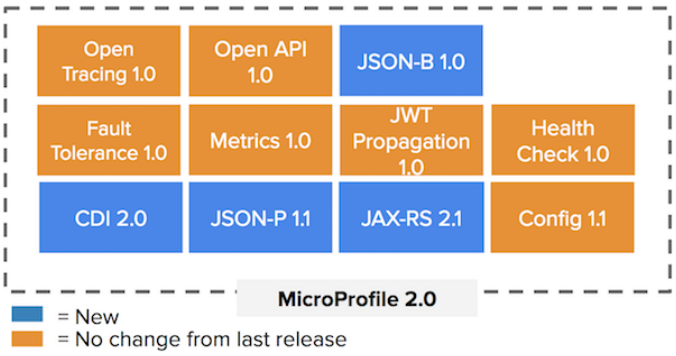

Abbildung 5. Microprofile-2.0-Features [3]

LITERATUR

[1] Eberhard Wolff, *Microservices*. dpunkt, 2015.
[2] jaxenter, "Java ee trifft microservices: Elefant im porzellanladen?" 2016. [Online]. Available: https://jaxenter.de/java-ee-trifft-microservices-elefant-im-porzellanladen-38432
[3] Microprofile, 2017. [Online]. Available: https://microprofile.io/blog/2017/10/eclipse-microprofile-1.2-available
[4] uht, *Proceedings of the 2016 IEEE International Symposium on Workload Characterization: September 25-27, 2016, Providence, RI, USA*. Piscataway, NJ: IEEE, 2016.
[5] Eberhard Wolff, *Continuous Delivery, 2nd Edition*. dpunkt, 2016.
[6] Lars Röwekamp, "Microservices mit java ee: Alptraum oder dreamteam?" 2017. [Online]. Available: https://www.informatik-aktuell.de/entwicklung/programmiersprachen/microservices-mit-java-ee-alptraum-oder-dreamteam.html
[7] ——, "Java ee in a box: Microprofile – ein neuer community-driven standard," 2016. [Online]. Available: https://jaxenter.de/java-ee-microprofile-42877
[8] self-contained services, 17.12.2017. [Online]. Available: http://scs-architecture.org/
[9] 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
[10] ——, 2017. [Online]. Available: https://www.heise.de/developer/meldung/Java-MicroProfile-1-2-hat-vier-neue-APIs-an-Bord-3849320.html
[11] ibm, 2017. [Online]. Available: https://developer.ibm.com/wasdev/blog/2017/09/01/microprofile-configuration-api-beta-liberty/
[12] Lars Röwekamp, 2017. [Online]. Available: https://www.heise.de/developer/meldung/Java-MicroProfile-1-2-hat-vier-neue-APIs-an-Bord-3849320.html
[13] 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
[14] 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