ECLIPSE MICROPROFILE

SEMINARARBEIT

von: Marckmiller, Lukas

Fach: Aktuelle Technologien zur Entwicklung verteilter Java-Anwendungen

Betreuer: Theis, Michael

München, im Juni 2018

INHALT

Microservices 3
drei Microservice-Varianten 3
MicroProfile 4
Java EE vs. MicroProfile 5
How to get started 6
Payara Micro 6
Apache TomEE 6
Wildfly Swarm 6
IBM Open Liberty 6
KumuluzEE 7
Hammock 7
Einfaches Beispiel 7
MicroProfile APIs im Detail 8
MicroProfile 1.0 (Sep, 2016) 9
Context and Dependency Injection 9
JSON-P 10
JAX-RS 10
MicroProfile 1.1 (August 2017) 11
Config 11
MicroProfile 1.2 (September 2017) 12
Health Check 13
Metrics 14
Fault Tolerance 14
JWT Propagation (JWT RBAC) 15
MicroProfile 1.3 (Januar 2018) 16
Open API 17

Open Tracing 17

REST Client 18

MicroProfile zukünftige Releases 19

Erstellen einer einfachen MicroProfile Anwendung 20

Alternativen 20

Fazit 20

Quellen 22

Abbildungsverzeichnis 23

Glossar 23

MICROSERVICES

Um die grundlegende Idee des Eclipse MicroProfile verstehen zu können werde ich zuerst auf Microservices und deren Bedeutung in der heutigen Softwarearchitektur eingehen. In der modernen Softwareentwicklung gehen die Trends von schwergewichtigen, monolithischen Client-Server Architekturen hin zu einer Architektur aus kleinen leichtgewichtigen Services, den sogenannten Microservices. Jeder Service übernimmt dabei einen möglichst kleinen funktionalen Ausschnitt der gesamten Applikation, da jeder Service unabhängig voneinander deployt und ausgeführt wird muss z.B für ein Update nicht die gesamte Applikation für eine Downtime berücksichtigt werden. Auch die Wartbarkeit einzelner Microservices ist ein großer Vorteil gegenüber schwergewichtigen Server Architekturen. Da der Code eines jeden Service in unterschiedlichen Repositories verwaltet werden kann und idealerweise kleinere Teams an einem Service arbeiten. So gliedern sich Microservices auch perfekt in die modernen agilen Softwareentwicklungsmethoden ein. Ein weiterer immer wichtiger werdender Punkt ist das Thema Sicherheit. Da jeder Microservice eine abgeschlossene Einheit (Kommunikation mit anderen Microservices nur z.B über REST Schnittstellen) mit idealerweise eigener Datenbank bildet, können die für jeden Service idealen Sicherheitsvorkehrungen geschaffen werden. Schafft es ein Angreifer dennoch den Service zu kompromittieren und auf dessen Datenbank zuzugreifen, kommt er nur an die Daten des Service selbst und stellt damit keine Gefahr für die übrigen Services und damit die Gesamtheit des Systems da. Da Microservices ausschließlich durch definierte Schnittstellen miteinander kommunizieren, kann die Plattform und die Programmiersprache, in welcher der Service implementiert werden soll, ideal an die Funktion des Service oder den Vorlieben der Entwickler angepasst werden. Stellt man nun fest, dass ein Service unter steigender Last durch mehr Anfragen als geplant zusammenbrechen könnte, dupliziert man diesen Service und bildet ein Cluster mit einem Load Balancer, damit ist man flexibler und skalierbarer als je zu vor.

Netflix ist ein berühmtes Beispiel für ein innovatives Unternehmen welches voll und ganz auf Microservices setzt. Dabei geht man sogar noch einen Schritt weiter, teilweise sparen sich die Entwicklerteams das Beseitigen von schwerwiegenden Fehlern, stattdessen wird der Service kurzerhand neu geschrieben.

(Böttcher, 2017)

DREI MICROSERVICE-VARIANTEN

1. Developer Anarchy von Fred George
    - Jeder im Team entwickelt
    - Verzicht auf Manager etc.
    - Volle Konzentration auf Geschäftsziel
    - Sehr kompakte Services
2. Microservice als REST Schnittstelle (Netflix)
    - Die Microservices implementieren REST-Schnittstellen
    - Funktionalität über REST zur Verfügung gestellt
    - Unabhängige Entwicklung und optimale Skalierung

-

3. Self-contained Systems (SCS)
    - Jedes SCS ist eine eigene Webanwendung
    - Aufteilung eines SCS in einzelne Microservices möglich
    - Fokus auf Integration von Web-Frontend

	Developer Anarchy	Netflix	Self-contained System
Größe eines Microservice	10 bis 100 Zeilen	Von einem Team gut zu bewältigen	1 SCS = 1…n Microservices
Integration	Asynchrone Kommunikation/ Messaging	REST	Bevorzugt Webintegration, ansonsten asynchrone Kommunikation
Programmiersprachen	Beliebig	Beliebig, aber Fokus auf Java	Beliebig

ABBILDUNG 1 - MICROSERVICES-ANSÄTZE IM VERGLEICH

MicroProfile

> *Eclipse MicroProfile is an open-source community specification for Enterprise Java microservices*

(Eclipse, 2018)

Das obenstehende Zitat zeigt, dass es sich bei dem Eclipse MicroProfile nicht um eine Applikation, eine reine API Sammlung oder ein Framework handelt, sondern vielmehr um eine Community aus erfahrenen Java Entwicklern und Firmen, welche die Entwicklung von Microservices mit Java vereinfachen und vorantreiben wollen. Ein Ziel der Community ist es eine Basis Plattformdefinition, welche Java Enterprise für Microservices optimiert bereitzustellen. Diese Basis Plattformdefinition bestand dabei ursprünglich aus den API´s für JAX-RS + CDI + JSON-P. Dabei soll die Community komplett in die Definition involviert werden.

> *Wir versuchen einen portablen und interoperablen Standard für Leute zu schaffen, die daran interessiert sind, Microservices mithilfe von Java EE zu entwickeln.*

(Little, 2016)

ABBILDUNG 2 - ORGANISATIONEN ALS BESTANDTEIL DER COMMUNITY

Java EE vs. MicroProfile

Durch die steigende Beliebtheit der Microservices, stößt man bei der Entwicklung mit Java auf Probleme und Grenzen von JavaEE und wird langsam aber sicher feststellen, dass JavaEE als Plattform für Microservices nicht konzipiert und damit nicht die Ideale Wahl ist. Einige wenige Microservices können zwar parallel innerhalb eines JavaEE Applikation Servers betrieben werden, allerdings kann die Anzahl schnell steigen, ein wichtiger Aspekt, wenn man den Vorteil der Skalierbarkeit von Microservices betrachtet.

Die führenden Applikation Server Hersteller haben genau dies erkannt und bieten in neueren Versionen Leichtgewichte Ihrer Server an, welche sich auf Basiskomponenten beschränken lassen. Leider sind selbst diese Server oftmals zu langsam und unflexibel. Die Idee hierbei ist nur die wirklich für die Lauffähigkeit des Microservice benötigten Komponenten des Applikationsservers mit dem Code des Service selbst zu einem schmalen JAR zu bündeln (Uber JAR Konzept).

Genau dies ist der Ansatz, den die Community mit dem MicroProfile schaffen will. Ein Minimalset an API´s zur Implementierung, Deployment und Runtime eines Microservice so performant und leichtgewichtig wie nur möglich.

Der Fokus des Projektes liegt dabei nicht darauf einen neuen Standard zu schaffen, welcher sich in JavaEE eingliedert oder sogar JavaEE für Microservices ersetzten könnte, sondern vielmehr um gemeinsam mit der Community eine Grundlage für Entwickler und Unternehmen zu schaffen, um qualitativ hochwertige und stabile Microservices zu schreiben. Wichtig ist sich auf Innovationen und Fortschritt zu konzentrieren anstatt die Zeit und Ressourcen aufzuwenden um einen neuen Standard zu integrieren.

How to get started

Wer einen Blick auf die offizielle Homepage (https://microprofile.io) wirft findet neben Foren und Bereichen der Community auch Beispielprojekte mit den jeweiligen GitHub Links. Dabei gibt es für jede Basis API des MicroProfiles ein eigenes Repository mit Beispiel Code.

Anderenfalls kann direkt losgelegt werden, indem man sich seinen persönlichen Favoriten aus einigen MicroProfile Implementierungen aussucht.

Payara Micro

- Open Source
- Nur 57MB groß
- Keine Installation und Konfiguration nötig
- Kein Applikation Server nötig
- Automatisches Clustering
- API für volle Integration in Applikationen

Apache TomEE

- Basiert auf Apache Tomcat
- Schnell und schmal
- Unterschiedliche Versionen verfügbar
- Einfache und schnelle Installation
- Ein-Archiv-Idee
    - EJB´s und Servlet in einem WAR File

Wildfly Swarm

- Red Hat Implementierung
- Single JAR Konzept (Uber JAR)
    - Applikation Server und Anwendung in einer JAR Datei
- Online Maven Projekt Konfigurator
- Gute ausgereifte Dokumentation
- Flexibel durch einfaches weglassen von nicht benötigter Funktionalität
- Unterstützung vieler Drittanbieter Apps
- Integrierbare RedHat Projekte für SSO, Monitoring …

IBM Open Liberty

- Open Source
- Nur 92 MB groß
- 22 Beispiele Projekte, sorgfältig dokumentiert mit Aufwandsschätzung
- Rege Community, regelmäßige Releases

KumuluzEE

- Java Duke's Choice Award Winner
- Noch kein Support für MicroProfile 1.3
- Optimiert für Docker und Kubernetes
- Integrierbar dank eigener KumuluzEE API
- Zusätzliches Single JAR Konzept

Hammock

- Eher unbekanntere Implementierung
- Basiert auf CDI
- Einfache leichtgewichtige Implementierung
- Plugins für Uber Jar
    - Single JAR Konzept

Einfaches Beispiel

Ein neuer Microservice soll nun im Folgenden implementiert werden. Ich habe mich für IBM´s OpenLiberty entschieden, da ich keine bestimmten Anforderungen an die Funktionalität habe und OpenLiberty visuell und inhaltlich gute Beispiele liefert.

Direkt auf der Startseite unter https://openliberty.io/ findet sich eine Anleitung für ein Beispiel Microservice

```
git clone https://github.com/OpenLiberty/sample-getting-started.git

cd sample-getting-started

mvn clean package liberty:run-server


Open browser to: http://localhost:9080
```

Abbildung 3 - Getting Started App

mvn clean package liberty:run-server

Dieser Maven Befehl bereinigt das Verzeichnis und entfernt build Dateien. Außerdem kompiliert, verpackt und verschiebt er den Code wie im POM File spezifiziert. Anschließend werden die Ressourcen, welche für den Liberty Server benötigt werden heruntergeladen und der Server anschließend gestartet und mit meinen Binaries initialisiert, wie ebenfalls im POM File definiert. Nun ist mein Service über den Port 9080 lokal zu erreichen. Dieser Test Service nutzt die in MicroProfile genutzten API´s Health, Config und Metrics, auf die noch genauer eingegangen wird.

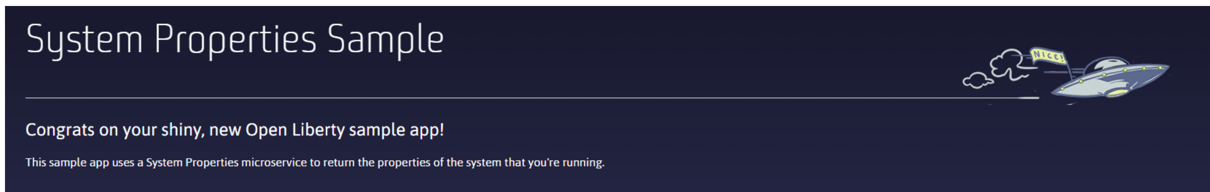

ABBILDUNG 4 - GETTING STARTED MICROSERVICE

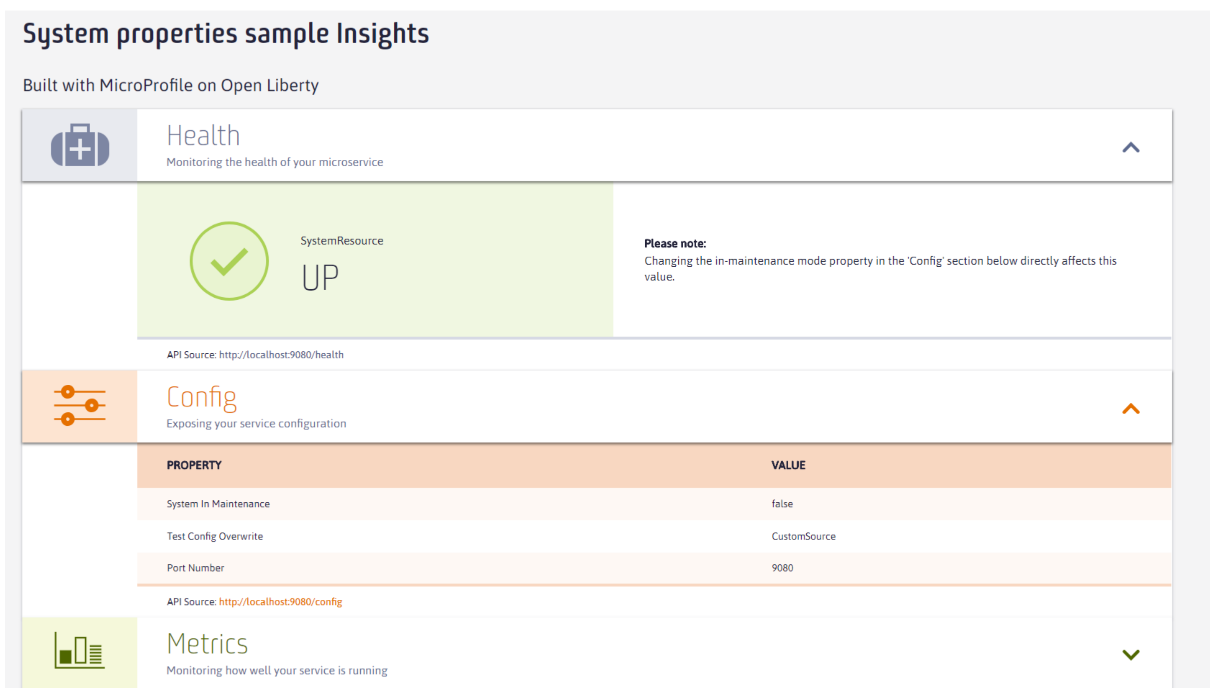

ABBILDUNG 5 - HEALTH, CONFIG, METRICS API (JSON BASIERT)

Nachdem wir nun demonstriert haben, wie schnell ein Java Projekt, welches einen Microservice implementiert, ohne weiteren Konfigurationsaufwand geklont, gebaut und ausgeführt werden kann, werde ich im Folgenden auf die Basis API´s des Microprofiles eingehen.

MicroProfile APIs im Detail

Die Idee für das MicroProfile ist es ein Basis Set an API Spezifikationen zusammenzustellen, welche mindestens benötigt werden, um einen Microservice zu implementieren. Dies soll der Grundbaustein sein, um mit der Community zusammen dieses Basis Set in regelmäßigen Abständen zu erweitern und neue Versionen des Microprofiles zu veröffentlichen.

MICROPROFILE 1.0 (SEP, 2016)

Im folgenden Abschnitt werden die einzelnen Versionen und die beinhalteten API´s inklusive Version aufgelistet. Neue API´s in einer Version werden in einem Quick Overview erklärt.

Bestandteil des ersten Releases waren lediglich 3 Java-Spezifikationen.

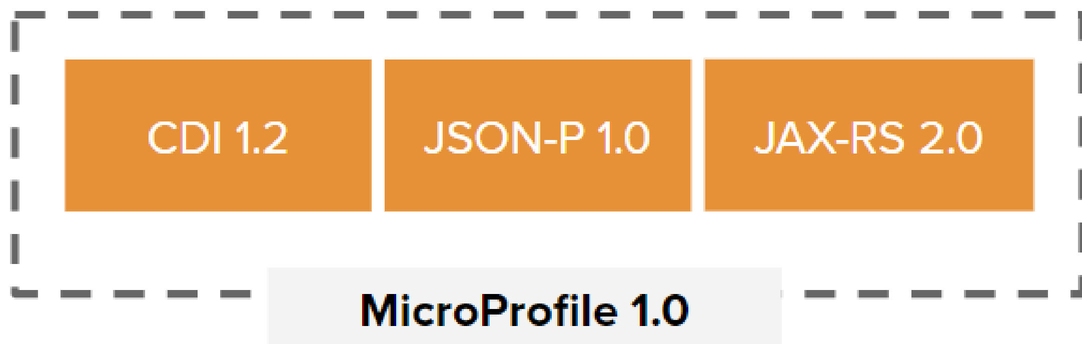

ABBILDUNG 6 - MICROPROFILE 1.0 - ERSTER RELEASE

CONTEXT AND DEPENDENCY INJECTION

Man stellt sich vor, man schreibt eine Klasse (TransactionImpl) für eine Banking App. Diese Klasse stellt jeweils eine Methode für Ein- und Auszahlung bereit. Beide Methoden nutzen ein Objekt (transport), welches für den Geldtransfer zuständig ist. Wie genau die Transfermethode des Objektes funktioniert ist nicht bekannt (es kann z.B. über eine REST Schnittstelle mit dem zentralen Server der jeweiligen Bank kommunizieren). Meine Klasse interessiert es auch nicht wie der Transfer funktioniert, allerdings schaffe ich eine Abhängigkeit von diesem Transport Objekt. Das bedeutet meine Klasse, die sich funktional nicht für das Objekt interessiert muss es erstellen und verwalten. Werden nun Unit Tests für meine neu erstellte Klasse implementiert, schlagen diese fehl sobald das Transport Objekt einen Fehler wirft (z.B. keine aktive Verbindung zum Server).

```
public class TransactionImpl
{
    // Nicht triviale abhängigkeit
    Transport transport;

    public void einzahlung(double betrag)
    {
        //do some stuff
        transport.transfer(betrag);
    }

    public void auszahlung(double betrag)
    {
        //do some stuff
        transport.transfer(0 - betrag);
    }
}
```

Mit der Dependency Injection in CDI soll diese Abhängigkeit nach dem Motto „don´t call us, we´ll call you“ aufgelöst werden. Die Idee ist ein Objekt dieser Transportklasse meiner Klasse zu initiieren und die Abhängigkeit vollständig aufzulösen. Meine Klasse erwartet dann zur Laufzeit dieses Objekt, kümmert sich aber nicht mehr um Erstellung und Verwaltung. Das Objekt wird dafür mit der Annotation **@Inject** markiert. Vereinfacht gesagt sucht CDI dann nach einer passenden Implementierung, erstellt ein Objekt und injiziert dieses meiner Klasse.

CDI ist damit ein wichtiger Bestandteil für die Entwicklung von Microservices um Abhängigkeiten auf eine transaktionale Ebene zu verschieben, Objekte zu erhalten und zwischen Services auszutauschen.

JSON-P

JSON-P bietet verschiedene Möglichkeiten zur Verarbeitung von JSON Nachrichten. Mit JSON-P können JSON basierte Nachrichten erstellt, geparst, verändert und durchsucht werden. JSON-P ist für die Kommunikation zwischen Microservices gedacht.

Beispiel zur Erstellung eines JSON Strings mit JSON-P

```
JsonObject json = Json.createObjectBuilder()
     .add("name", "Falco")
     .add("age", 3)
     .add("biteable", Boolean.FALSE).build();
String result = json.toString();
System.out.println(result);
```

Ausgabe:

```
{
    "name": "Falco",
    "age": 3,
    "bitable": false
}
```

JAX-RS

Mit JAX-RS ist im MicroProfile eine JavaEE API Spezifikation (also eine Sammlung von Interfaces und Annotations) vorhanden mit welcher REST Schnittstellen für einen Web Service und Client definiert werden können. Eine der bekanntesten Referenzimplementierungen ist Jersey, generell liefern Applikationsserver ihre eigene Referenzimplementierung aus, dass bedeutet ein simples Hinzufügen einer Dependency (Abhängig von Applikationsserver) in meiner Maven POM Datei reicht aus.

Die REST Schnittstelle wird dann mit denen von JAX-RS bereitgestellten Annotationen definiert.

```
@GET
@Path("/get/")
@Produces(MediaType.APPLICATION_JSON)
public Response getNotification() {
    //does something}
}
```

Das obenstehende Beispiel implementiert eine GET Methode in einer Ressource, welche ausgeführt wird, wenn der Client (z.B. Webbrowser) ein GET Request an den Server sendet. Die Methode liefert dann ein Ergebnis im JSON Format zurück (MediaType.APPLICATION_JSON). Mit @PATH wird definiert was der URL angehängt werden muss um das Request korrekt zu verarbeiten. Angenommen der Applikationspfad ist /api dann ist die vollständige URL http://SERVER:PORT/api/get.

Eine Alternative zu einer REST basierten Web Anwendung wäre eine auf SOAP basierte. Aufgrund der besseren Skalierung und Performance sowie dem Nutzen von JSON als Nachrichtenformat ist REST ideal für Microservices und damit auch in unserem MicroProfile. Der Vorteil die Implementierungen auf REST zu begrenzen ist, dass dadurch alle Microservices über die gleiche Schnittstelle mit dem Server kommunizieren. Ich brauche also nicht auf meinem Applikationsserver unterschiedliche API´s anzubieten, damit alle meine Microservices (sofern nicht jeder Microservice seinen eigenen Applikationsserver mitliefert siehe Kapitel JavaEE vs. MicroProfile 2. Absatz) mit diesem kommunizieren können.

MicroProfile 1.1 (August 2017)

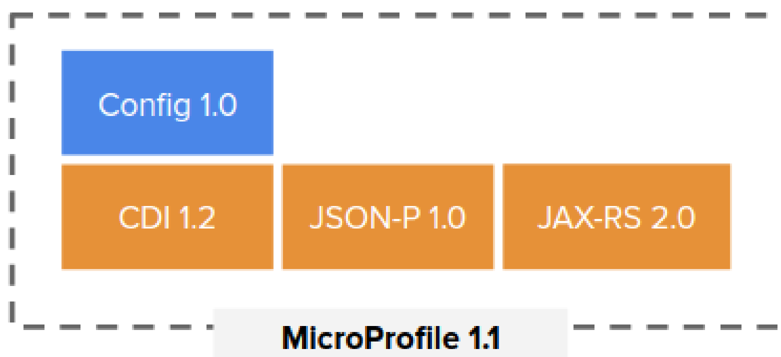

ABBILDUNG 7 - MICROPROFILE 1.1 RELEASE

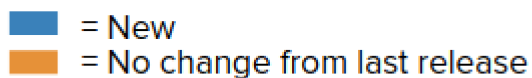

ABBILDUNG 8 - LEGENDE RELEASE FEATURES

Config

Mit dem neuen Jahr kam ein neuer Release und eine weitere in meinen Augen sehr hilfreiche API. Jede Applikation beinhaltet Konstanten, die an Methoden, Konstruktoren, etc. übergeben werden um meine Applikation für die Zielplattform zu konfigurieren. Eine Webapplikation möchte eine E-Mail an den Server Administrator senden sobald ein schwerwiegender Fehler auftritt, die E-Mail-Adresse ist fest im Code hinterlegt. Dies stellt dann ein Problem da, sollte der Server Administrator das Unternehmen verlassen oder meine Webapplikation von unterschiedlichen Unternehmen benutzt werden, welche alle einen eigenen Systemadministrator beschäftigen. Dafür müsste ich also in meinem Code alle Konfigurationen anpassen und das Projekt dann neu deployen. Mit der Config API

können all diese Konstanten in eine eigene Datei ausgelagert werden. Die Konstanten folgen dabei dem Layout des Java Package Pfades.

```
com.acme.myproject.someserver.url = http://some.server/some/endpoint
com.acme.myproject.someserver.port = 9085
com.acme.myproject.someserver.active = true
com.acme.other.stuff.name = Karl
com.acme.myproject.notify.onerror=karl@mycompany,sue@mcompany
some.library.own.config=some value
```

ABBILDUNG 6 - CONFIG DATEI BEISPIEL

Zusätzlich können unterschiedliche Konfigurationsdateien definiert und priorisiert werden oder bestehende Quellen wie System Properties oder Umgebungsvariablen genutzt werden.

Die Konfigurationen können dann über den ConfigProvider geladen werden.

```
Config config = ConfigProvider.getConfig();
String serverUrl = config.getValue("com.acme.myproject.someserver.url",
String.class);
```

Oder man nutzt die bereits kennenglernte CDI API.

```
@Inject
private Config config;
@Inject
@ConfigProperty(name="com.acme.myproject.someserver.url")
private String someUrl;
```

MICROPROFILE 1.2 (SEPTEMBER 2017)

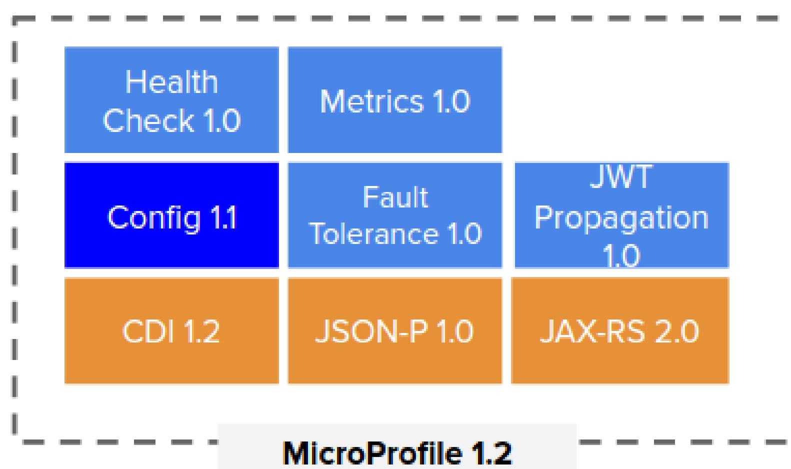

ABBILDUNG 9 – MICROPROFILE RELEASE 1.2

= New
= Updated
= No change from last release

Nur einen Monat nach dem letzten Release wird Version 1.2 mit gleich vier neuen API´s und einer neuen Version der Config API veröffentlicht.

HEALTH CHECK

Mit Health Check kann ein Microservice während der Laufzeit seine eigene Lauffähigkeit prüfen und bei Auftreten von Problemen einen Report an ein definiertes Ziel senden. Dieser Health Knotenpunkt entscheidet dann über weitere Aktionen und startet den Webservice bei Bedarf neu. Dabei können verschiedene Komponenten des Services überprüft werden, wie z.B.

- sind alle benötigten Frameworks und Bibliotheken vorhanden?
- sind System Properties gesetzt, kann eine Verbindung zu einer benötigten Datenbank hergestellt werden?
- können abhängige Services erreicht werden (zum Beispiel kann mein Service den Authentifizierungsserver erreichen, um sein Token zu validieren und sich somit zu authentifizieren)?

Um das Feature zu nutzen, muss das org.eclipse.microprofile.health.HealthCheck Interface implementiert und die Klasse mit @Health annotiert werden. Anschließend wird die Methode call() überschrieben. In der Methode kann dann mein HealthCheck implementiert und das Ergebnis als Objekt vom Typ HealthCheckResponse zurückgegeben werden.

```
public class HealthCheckImpl implements HealthCheck
{
     @Override public HealthCheckResponse call() {
         if(!System.getProperty("com.acme.myproject.someserver.name").equals
                 ("defaultServer")) {
             return
     HealthCheckResponse.named(SystemResource.class.getSimpleName())
                     .withData("default server", "not available")
                     .down().build();
         }
         return HealthCheckResponse.named(SystemResource.class.getSimpleName())
     .withData("default server", "available").up()
     .build();
     }
}
```

Ziele des Health Check Reports sind, dem Administrator genügend Informationen bereitzustellen, dass dieser sich ein Bild über den aktuellen Zustand des Service machen und bei Bedarf schnell handeln kann bzw. das System vordefinierte Schritte einleitet und diese dann für den Administrator ersichtlich macht. Dadurch ist die Wartbarkeit gerade bei vielen kleinen Services für den Administrator optimal und der Ausfall eines Knotens wird direkt erkannt und nicht erst wenn andere Knoten dadurch ausfallen und die Stabilität des ganzen Systems gefährdet wird. Außerdem können Schwachstellen in der Gesamtheit des Systems somit leicht identifiziert und beseitigt werden.

Metrics

Während die HealthCheck API darauf ausgelegt ist möglichst schnell zu ermitteln, ob ein Service überhaupt noch läuft, ist die Metrics API dafür gedacht, relevante Informationen über den Ressourcenverbrauch des Service zu sammeln (z.B. CPU-Last, Laufwerkauslastung, Netzwerkauslastung, Traffic zwischen Services, Verarbeitung und Reaktionszeiten). Diese Informationen sind besonders wichtig, um meine Microservices optimal in meine Hardwareumgebung einzubinden. Durch regelmäßiges Auswerten dieser Daten kann ermittelt werden, ob mein Service mehr Ressourcen benötigt oder in einem Cluster dupliziert werden muss um eine eventuell gestiegene Anzahl von Anfragen noch bearbeiten zu können. Mithilfe der Metrics API kann eine solide und zentralisierte Monitoring Plattform für meine Microservices implementiert werden.

```
@Timed(name = "PropertiesRequestTime", absolute = true,    description = "Time
needed to get the properties of a system from the given hostname")
public Properties get(String hostname) {
    Properties properties = SystemClient.getPropertiesForHostname(hostname);
    return properties;
}
```

Mit der Annotation @Timed kann geprüft werden, wie häufig eine Methode aufgerufen wird und wie viel Zeit es Bedarf, um die Methode auszuführen. Um nur die Anzahl an Aufrufen einer Methode zu bestimmen, muss diese mit der Annotation @Counted versehen werden. Mit @Gauge kann eine Methode ausgeführt und deren Rückgabewert ermittelt werden. Dies ist hilfreich um z.B. die Einträge einer Liste zu zählen, dafür muss einfach @Gauge als Annotation vor einen Getter für die Listengröße geschrieben werden. Jetzt könnte man sich fragen wieso man diesen Getter nicht direkt selbst aufruft. Die Implementation der Metrics API übernimmt dies für mich, deshalb benötige ich keine Referenz auf diese Klasse und die entsprechende Liste. Da die Metrics Annotationen zur Laufzeit ausgewertet werden, kann meine zu analysierende Methode auch als private Methode deklariert werden. Eine Methode muss also nicht extra public gemacht werden, damit sie von der Monitoring Plattform aufgerufen werden kann.

Nutzt man beispielsweise die OpenLiberty Implementierung der Metrics API, reicht es, die Annotation anzugeben. Die Ergebnisse werden dann an von OpenLiberty fest definierte Endpoints geschrieben und können dort abgefragt oder visualisiert werden.

Fault Tolerance

Um zu vermeiden, dass es überhaupt zu einem Absturz meines Service und einem Neustart kommt, sollte meine Applikation "fehlertolerant" entwickelt werden. Das bedeutet, eine Applikation ist besonders stabil, wenn sie sich aus etwaigen Problemen selbst herausmanövrieren kann anstatt direkt mit einer Fehlermeldung abzustürzen. Stattdessen stellt die Fault Tolerance ein Konzept auf, wie eine Applikation zur Laufzeit mit Fehlern und Problemen umgehen sollte. Dabei sollten folgende Aspekte fokussiert werden:

- Timeouts: Definieren von TimeOuts
- RetryPolice: Definition von Kriterien für eine erneute Ausführung

- Fallback: Eine alternative Lösung für eine fehlgeschlagene Ausführung
- Bulkhead: Aufteilung der Anwendung in einzelne isolierte Teile, um einen Gesamtausfall zu vermeiden
- CircuitBreaker: Abbruch der Ausführung bei unbestimmter Wartezeit oder Ausfall des Kommunikationspartners.

Mit der Fault Tolerance im MicroProfile sollen bestehende Implementation vereinheitlicht werden um einen Standard für Fehlertolerante Microservices zu schaffen. Die Fault Tolerance soll, wenn möglich von der Umgebung des Service (z.B. dem Host Server) garantiert werden.

Ein abstraktes Beispiel der OpenLiberty Implementierung im folgenden Absatz zeigt, wie die einzelnen API´s zusammenspielen und auf einen Ausfall des Systems reagieren.

Einer unserer Services wird nun in den Wartungszustand versetzt, dafür wird ein Konfigurationsparameter in unserem Config File gesetzt (Kapitel Config API). Ein Service, welcher mit diesem nun kommunizieren will, startet eine Anfrage, diese schlägt allerdings fehl. Da die Annotation @Fallback(fallbackMethod = "fallbackMethodName") auf dieser Methode gesetzt ist wird nun die alternative Methode (fallbackMethodName) in der aktuellen Klasse gesucht und ausgeführt.

JWT Propagation (JWT RBAC)

Eines der Felder der Informatik, welches in den letzten Jahren enorm an Bedeutung gewonnen hat aber immer noch oft als nicht notwendig erachtet wird, ist die IT-Security. Viele Microservices bedeutet auch viele Angriffspunkte, deshalb ist es umso wichtiger seine Schnittstellen nach außen besonders zu schützen.

Genau hier setzt die JWT RBAC (role based access control = Rollenbasierte Zugriffskontrolle) an und hält sich dabei stark an den RESTful Architekturstil. Dabei ist ein Service zustandlos, kann also keine Informationen über Authentifizierung und Authentisierung speichern. Ich müsste mich somit in jedem Request neu authentifizieren. Um diesen Overhead zu sparen, werden sogenannte Security Tokens eingesetzt. Dieses Token wird ausgestellt sobald sich eine Person erfolgreich authentifiziert hat und kann anschließend in jedem Request mitgeschickt werden.

Diese Security Tokens enthalten Informationen über die identifizierte Person sowie weitere Parameter wie z.B. bestimmte Rechte oder eine Gültigkeitsdauer. Diese Informationen werden zusammengefasst verschlüsselt und signiert. Somit habe ich eine nicht mehr lesbare Zeichenkette, die ich im Header meines Request verschicken kann. Damit ein anderer Service dieses Token auf Gültigkeit prüfen (validieren) oder Information über die Person aus dem Token extrahieren kann, sollte das Format einheitlich sein. Ansonsten muss ein zusätzlicher Authentifikationsservice implementiert werden, der diese Aufgaben übernimmt und die Resultate zurückschickt. Bei Bedarf können dann weitere Authentisierungsschritte eingeleitet werden.

Die heutzutage am häufigsten genutzten Technologien um einen Authentifizierungsprozess in einem RESTful Microservice zu realisieren sind OAuth2, OpenID Connect und JSON Web Token (JWT) Standards.

Die Annotation

```
@RolesAllowed({ "admin", "user" })
```

erlaubt nur authentifizierten Nutzern mit den Rollen „admin“ oder „user“ den Zugriff auf meinen Service.

```
@GET
@RolesAllowed({ "admin", "user" })
@Produces(MediaType.APPLICATION_JSON)
public Properties getProperties() {
    return System.getProperties();
}
```

Das Token wird im Hintergrund injected und verifiziert.

MicroProfile 1.3 (Januar 2018)

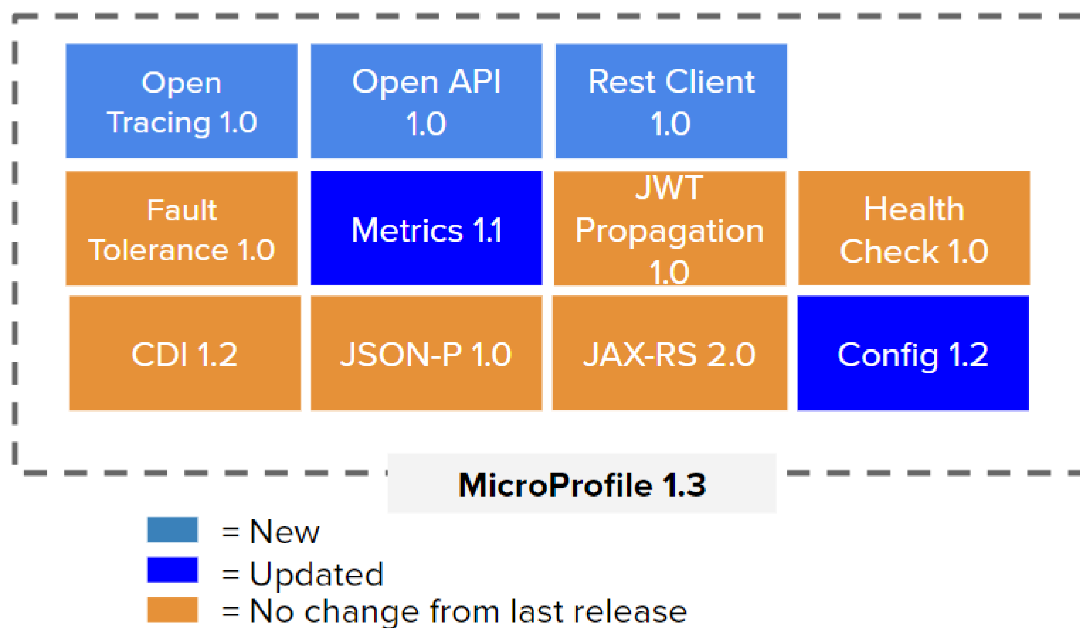

Abbildung 10 - MicroProfile Release 1.3

Mit dem ersten Release im Jahr 2018 werden drei neue Technologien dem MicroProfile hinzugefügt.

Open API

In unseren Szenarien sind wir immer davon ausgegangen, dass wir alle Services selbst implementiert haben und daher alle REST Schnittstellen genau kennen, um mit einem anderen Service zu interagieren. Außerhalb unseres erdachten Szenarios sieht es allerdings ganz anders aus: Unterschiedliche Entwicklerteams, verschiedene Technologien und Sprachen für Microservices erschweren es den Entwicklern sich schnell in diesem Gewirr zurecht zu finden vor allem da saubere, vollständige und einheitliche Dokumentationen nicht immer vorhanden sind.

Mit der Open API Spezifikation sollen Entwickler in der Lage sein für ihre RESTful Services einheitliche OpenAPI v3 Dokumentationen zu erstellen.

Eine Sammlung von Interfaces und Annotationen unterstützt den Entwickler relevante Informationen im Code zu verlinken. Anschließend kann daraus eine API Dokumentation erstellt werden, welche laut Spezifikation unter der Root URL des Service zur Verfügung gestellt werden soll (z.B. http://myHost:myPort/openapi). Ein Beispiel für eine Methode aus einer Bücherei API soll zeigen wie OpenAPI im Quellcode meines Service genutzt wird.

```
@GET
@Path("/findByBookStatus")
@Operation(summary = "Finds Books by status",description = "Multiple status values can be provided with comma separated strings")
public Response findBooksByStatus(...) { ... }
```

Output in YAML (Ausgabe kann alternativ auch in JSON sein):

```
/pet/findByStatus:
get:
  summary: Finds Books by status
  description: Multiple status values can be provided with comma separated strings
  operationId: findBooksByStatus
```

Neben API Dokumentationen kann dieser Output auch als Input für Codegeneratoren genutzt werden um daraus RESTful Services und Clients in anderen Programmiersprachen zu generieren. Dabei wird vorab mit einem geeigneten Editor (z.B. Swagger) eine OpenAPI Spezifikation erstellt und daraus dann Quellcode für meinen Service End Point generiert. Auch das automatisierte Erstellen von Testcode auf Basis des Outputs der OpenAPI Spezifikation ist denkbar, schließlich enthält es alle relevanten Informationen um als Tester Standardtestfälle zu identifizieren.

Open Tracing

In einem verteilten System, das aus einem verstrickten Netz von Microservices besteht, ist es nicht einfach zu erkennen wie einzelne Services untereinander verstrickt sind. So kann ein Service ein Request an einen weiteren Service stellen, dieser wiederrum benötigt für die Abarbeitung ein Request an einen dritten Service. Mit der Open Tracing API sollen diese undurchsichtigen Aufrufketten identifiziert werden.

Vorhandene Tracing Systeme haben den Nachteil, dass jeder Hersteller seine eigenen Verfahren entwickelt hat um Services zu tracen. Das bedeutet, ich muss als Entwickler meinen Service auf eines dieser Tracing Systeme anpassen. Mit Open API soll die Anwendung unabhängig von meinem

eingesetzten Tracing System sein. Lediglich das Tracing System muss mit den ankommenden Daten umgehen können.

Mögliche Implementierungen sind Zipkin oder Jaeger.

Es gibt zwei Arten der Integration von Open Tracing in einen Microservice.

1. Durch Konfiguration von außen
    - dabei bei muss für jeden Service auf einem Applikationsserver eine Implementierung von io.opentracing.Tracer zur Verfügung gestellt werden damit die Einstellungen unabhängig vom eingesetzten Tracing System bleiben
2. Programmatisch durch Verwendung von @Traced

```
@Traced(value = true, operationName = "InventoryManager.list")
public InventoryList list() {
    return invList;
}
```

Nach einem Aufruf von list() sollte auf meinem Tracing System ein neuer Eintrag mit „InventoryManager.list“ zu finden sein der meine Aufrufkette (Span) anzeigt. In unserem Fall besitzt die Kette nur einen Eintrag.

REST CLIENT

Wenn wir von REST mit JAX-RS gesprochen haben, haben wir hauptsächlich davon geredet, wie RESTful Service Anfragen entgegennimmt und entsprechende Antworten liefert. Zwar ist es möglich einen REST Client mit JAX-RS zu entwickeln, allerdings stößt man dabei sehr schnell an Grenzen vor allem beim Thema Typsicherheit.

REST CLIENT MIT JAX-RS

Ausgang ist folgendes Interface

```
@Path("/customers")
@Produces("application/json")
public interface CustomerService {
  @GET
  @Path("/{customerId}")
  public Customer getCustomer(@PathParam("customerId") String customerId);
                                                        …
}
```

Der entsprechende REST Client mit JAX-RS würde wie folgt auf die Ressource zugreifen

```
private static final String CUSTOMER_URI =
"http://localhost:9080/api/customers";
private Client client = ClientBuilder.newClient();

public Customer getCustomer(String customerId) {
    return client.request(MediaType.APPLICATION_JSON)
            .target(CUSTOMER_URI)
            .path(customerId)
            .get(Customer.class);
```

```
}
```

Hier muss der Entwickler selbst den Media Type (JSON) und den Ergebnistyp (Customer) angeben. Auch ein Tippfehler in der URL kann zu einem unnötigen Problem werden. Eine Verbesserung bietet hier die MicroProfile REST Client API.

REST Client mit MicroProfile Ansatz

Hierbei wird ein Objekt mit Hilfe des RestClientBuilders von meinem Service End Point Interface (Customer Service) erzeugt. Anschließend kann ich alle Methoden des Interfaces nutzen. Das Parsen des Media Typs wird hierbei vom Client automatisch übernommen. Mein Ergebnistyp ist damit Typsicher, da ich direkt das End Point Interface im Client nutze.

```
private CustomerService customerService = RestClientBuilder.newBuilder()
        .baseUrl(CUSTOMER_URI)
        .build(CustomerService.class);

public Customer getCustomer(String customerId) {
    return customerService.getCustomer(customerId);
}
```

MicroProfile zukünftige Releases

Für den Release 1.4 im zweiten Quartal 2018 sind Updates für bestehende Spezifikationen, sowie die Ausarbeitung bereits vorhandener Dokumentationen geplant.

Für die Version 2.0 soll eine API für JSON Binding hinzugefügt werden. Damit ist es möglich Java Objekte direkt in Ihre JSON Repräsentation zu konvertieren und umgekehrt.

```
public class Dog {
    public String name;
    public int age;
    public boolean bitable;
}
Dog dog = …;
dog. …
Jsonb jsonb = JsonbBuilder.create();
String result = jsonb.toJson(dog);
```

Ausgabe:

```
{
    "name":"Falco",
    "age":4,
    "bitable":false
}
```

Erstellen einer einfachen MicroProfile Anwendung

Wird in der Präsentation im Zuge eines Live Programmings gezeigt. Der Quellcode kann aus diesem Repository geklont werden.

https://github.com/openliberty/guide-microprofile-intro.git

Alternativen

Das Spring Framework ist die populärste Alternative zum Entwickeln von RESTful Microservices. Der größte Unterschied ist, dass Spring überwiegend auf eigene Konzepte, Ideen und API´s setzt. Stattdessen versucht Eclipse mit seinem MicroProfile so weit wie möglich auf JavaEE aufzusetzen und vorhandene Konzepte zu übernehmen. MicroProfile setzt bei seinen Ideen und Konzepten außerdem voll auf seine Community und die involvierten Unternehmen.

Fazit

Anstatt neue Wege zu gehen wird hier auf das geballte Wissen der erfahrenen JavaEE Entwickler gesetzt, um Java bei der Entwicklung von Microservices für die Zukunft fit zu machen. Dabei wird versucht keine Alternative für JavaEE zu schaffen, sondern vorhandenen Technologien aus JavaEE für Microservices zu optimieren. Die Community ist dabei breit gefächert und verfügt über Wissen zu nahezu allen Technologien im Microservice Umfeld. Dadurch wird eine gewisse Herstellerunabhängigkeit erreicht, die zu Portabilität für entwickelte Microservices führt. Das Hauptziel ist dabei MicroProfile zusammen mit der Community weiter voranzutreiben und auf zukünftige Trends schnell zu reagieren.

Vor Beginn dieser Arbeit hatte ich wenig Erfahrung mit der Entwicklung von Microservices und habe von MicroProfile noch nie etwas gehört. Ich befürchtete, von neuen Technologien förmlich überrollt zu werden und keinen richtigen Einstieg zu finden. Dies hat sich nicht bestätigt. Die überschaubare Anzahl von API´s und die sehr sorgfältige Dokumentation der Spezifikationen sowie die Verlinkung verschiedenster Beispielprojekte macht den Einstieg mehr als leicht.

Hat man sich etwas mit dem Thema beschäftigt, kann man sich konkrete Implementationen für API Spezifikationen ansehen. Dadurch bin ich auf IBM´s OpenLiberty gestoßen welche zum momentanen Stand 22 Guids anbietet, die bis ins kleinste Detail beschrieben werden. Dabei befinden sich auch Musterlösungen in den jeweiligen Repositories. Auch die Payara Micro Entwickler liefern eine Plattform mit Tutorials rund um das MicroProfile. Auf der Seite von Wildfly Swarm kann ein Microservice Projekt bereits online vorkonfiguriert und heruntergeladen werden. Hat man sich etwas in die Thematik eingearbeitet und die nötigen Dokumentationen zur Hand, kann man ohne großen Aufwand einen hochwertigen, stabilen und sicheren Microservice implementieren. Dieser lässt sich auch in einer Host Plattform mit in anderen Sprachen geschriebenen Microservices leicht eingliedern. Wer sich etwas intensiver mit den zukünftig geplanten Technologien im MicroProfile auseinander setzten will, kann der offiziellen Google Group beitreten. Wer sich den Google Kalender ansieht findet Termine für regelmäßige „Hangouts“ ein Video Meeting in welchem zu einem Themengebiet (z.B. JWT)

referiert wird. Teilnehmer der Community sind neben den bereits erwähnten Unternehmen, CEO´s, selbstständige Java Entwickler, Professoren, Studenten aus der ganzen Welt. Ich bin mir sicher, dass Oracle hier sehr bald auf die Community zugehen wird und aus dem MicroProfile ein offizielles JavaEE Profil analog zu Full und Web wird.

QUELLEN

https://www.torsten-horn.de/techdocs/jee-rest.htm#Vergleich-REST-SOAP

https://docs.oracle.com/javaee/6/tutorial/doc/gijqy.html

https://jaxenter.de/microprofile-1-3-openapi-opentracing-type-safe-rest-client-68061

https://microprofile.io/

https://projects.eclipse.org/projects/technology.microprofile

https://wiki.eclipse.org/MicroProfile/Implementation

http://wildfly-swarm.io/

https://jaxenter.de/java-ee-microprofile-42877

https://www.heise.de/developer/meldung/Microservices-Eclipse-MicroProfile-1-1-baut-auf-Java-8-3793972.html

LITERATURVERZEICHNIS

Böttcher, P. D. (05 2017). Vorlesung Microservices.

Eclipse. (05 2018). *https://microprofile.io/.* Von https://microprofile.io/: https://docs.google.com/presentation/d/1BYfVqnBIffh-QDIrPyromwc9YSwIbsawGUECSsrSQB0/edit#slide=id.p abgerufen

Little, M. (29. Juli 2016). Interview mit Mark Little - Wie MicroProfile Java EE und Microservices zusammenbringen will. (G. Motroc, Interviewer)

ABBILDUNGSVERZEICHNIS

Abbildung 1 - Microservices-Ansätze im Vergleich 4

Abbildung 2 - Organisationen als Bestandteil der Community 5

Abbildung 3 - Getting Started APP 7

Abbildung 4 - Getting Started Microservice 8

Abbildung 5 - Health, Config, Metrics API (JSON basiert) 8

Abbildung 6 - MicroProfile 1.0 - erster Release 9

Abbildung 7 - Microprofile 1.1 Release 11

Abbildung 8 - Legende release features 11

Abbildung 9 – Microprofile Release 1.2 12

Abbildung 10 - MicroProfile Release 1.3 16

GLOSSAR

JWT	Jason Web Token
Authentisierung	Nachweis welcher die Identität einer Person bestätigen soll.
Authentifizierung	Prüfung der Authentisierung
REST	Programmierkonzept für die Implementierung von Webservices
OAuth	Standardisiertes Protokoll für API Authentifizierung
API	Programmierschnittstelle
JSON	Textbasiertes Format zum Austausch von Daten
JAR	Java Archiv ähnlich wie ZIP
Repository	Verzeichnis zum Speichern und Verwalten von Quellcode
POM	Konfigurationsdatei für Maven