Hochschule Esslingen
University of Applied Sciences

Studienprojekt

Service Discovery und Konfiguration mit SmallRye

Parallele und verteilte Systeme

Wintersemester 2022/2023

Nico Linder (766043)

Prüfer: Matthias Häussler, Prof. Dr.-Ing. Reinhard Schmidt

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

1 Einführung

1.1 Ziel des Projekts

Ziel dieses Studienprojektes ist zu demonstrieren, wie mithilfe des SmallRye-Projekts Service-Discovery, Client-Side Load-Balancing und Service-Konfiguration für Java-Anwendungen implementiert werden können. Hierzu soll eine verteilte Anwendung entwickelt werden, in der mehrere Instanzen eines Backend-Microservice verfügbar sind, welche alle dieselbe Funktionalität besitzen. Ein Client-Microservice soll SmallRye Stork nutzen, um alle verfügbaren Instanzen des Backend-Services zu entdecken und API- Requests an die am besten geeignete Instanz zu senden. Weiterhin soll SmallRye Config genutzt werden, um die Konfigurationen aller Instanzen des Backend-Services simultan aktualisieren zu können. Alle Microservices sollen mit dem Quarkus-Framework implementiert werden.

1.2 Technologien

1.2.1 Quarkus

Quarkus ist ein Full-Stack Java-Framework, das als Open-Source-Projekt entwickelt wird. Traditionelle Java-Anwendungen sind oft monolithisch konzipiert, haben lange Startzeiten und hohen Speicherbedarf, was für verteilte, Cloud-native Anwendungen unvorteilhaft ist. [1] Mit Quarkus können Java-Anwendungen entwickelt werden, welche für Kubernetes-basierte Containerplattformen optimiert sind und sich besonders durch schnelle Startzeiten und geringen Speicherplatzverbrauch auszeichnen. So können beispielsweise schnell zusätzliche Container gestartet werden, um

Lastspitzen oder Ausfälle abzufangen. Weitere Features von Quarkus sind Live Coding, einheitliche Konfiguration aus einer zentralen Quelle sowie die Möglichkeit, native Images zu erzeugen, die keine JVM benötigen und Startzeit sowie Speicherbedarf der Anwendung noch weiter senken. [2]

1.2.2 SmallRye Stork

SmallRye Stork ist ein Framework für Service Discovery und Client-Side Load-Balancing. In verteilten Systemen existieren oft mehrere Instanzen eines Services, um die Last zu verteilen oder die Ausfallsicherheit durch Redundanz zu erhöhen. Die Liste dieser Instanzen ist dynamisch, es können Instanzen dazukommen oder wegfallen. Stork entdeckt alle verfügbaren Instanzen (Service Discovery) und wählt die aus, die am besten geeignet ist, eine Anfrage zu bearbeiten (Service Selection). Für Service Discovery werden verschiedene Mechanismen wie Consul, Eureka oder Kubernetes-Integration unterstützt. Zur Lastverteilung werden fertige Lösungen für gängige Schema wie Round Robin, Least Requests oder Least Response Time bereitgestellt, es können aber auch eigene Load-Balancer implementiert werden. [3] [4]

1.2.3 SmallRye Config

SmallRye Config ist eine Implementierung der Eclipse MicroProfile Config-Spezifikation, mit der Anwendungen und Container konfiguriert werden können. Konfigurationseigenschaften können dazu von verschiedenen Quellen gelesen werden. Jeder Quelle wird eine Priorität zugewiesen, sind mehrere Quellen verfügbar, wird die Konfiguration aus der Quelle mit der höchsten Priorität angewendet. Neben den vordefinierten Konfigurationsquellen können auch eigene implementiert werden. [5]

1.2.4 Consul

Consul ist eine Service, mit dem Anwendungen auf verteilter Infrastruktur verbunden und konfiguriert werden können. Consul kann als Service Discovery-Backend für SmallRye Stork, sowie als Configuration Provider für Quarkus-Anwendungen genutzt werden. [6]

2 Beschreibung der Anwendung

Die Anwendung besteht aus zwei Microservices: Einem Backend-Service, der auf eine PostgreSQL-Datenbank zugreifen und Operationen auf den Daten ausführen kann und einem Client, der Daten vom Backend-Service anfordern kann. Es können mehrere Instanzen des Backend-Service gestartet werden, welche sich bei einem Consul-Server registrieren müssen. Dieser stellt auch die Konfiguration für die Backend-Instanzen bereit. Alle Komponenten werden als Docker-Container realisiert.

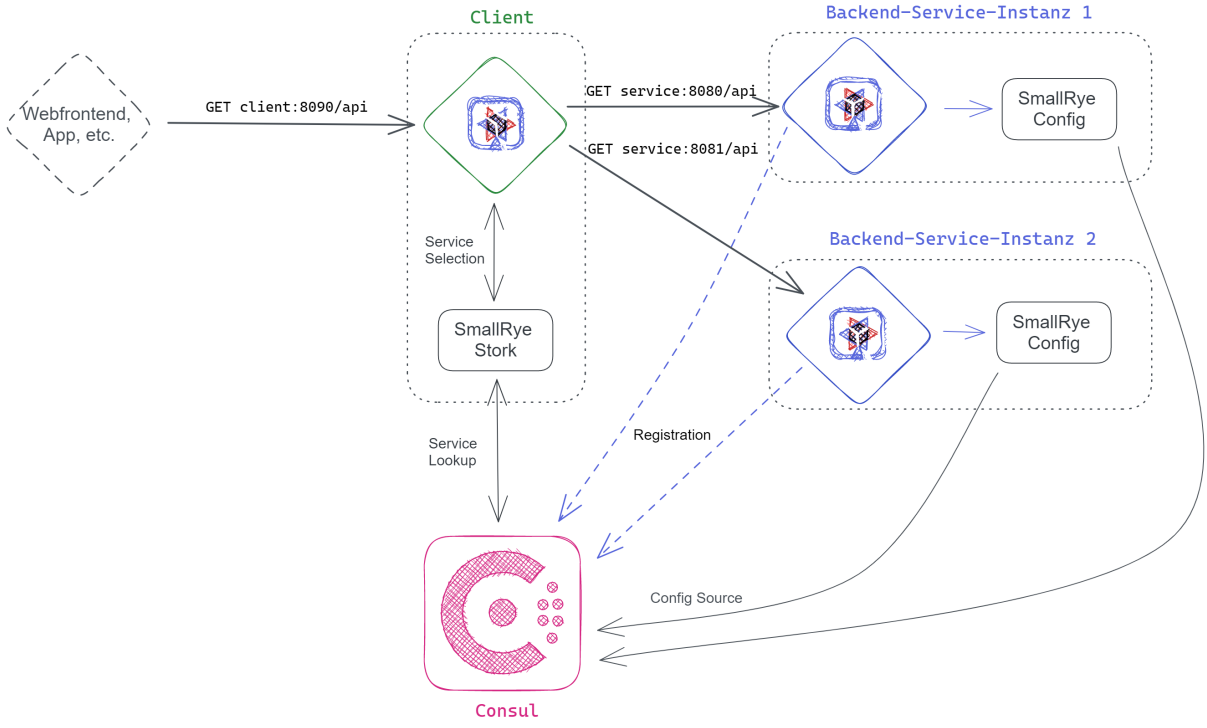

Abb. 2.1: Architekturdiagramm der Anwendung

2.1 Backend-Service

Der Backend-Microservice ist mit Quarkus implementiert und soll mit SmallRye Config konfiguriert werden können. Vorteilhaft ist hier, dass das Quarkus-Framework bereits SmallRye Config verwendet, um Konfigurationswerte z.B. aus einer Properties-Datei zu lesen und per Annotation im Code bereitzustellen.[7] Da wir allerdings nicht jede Instanz des Backend-Service einzeln konfigurieren möchten, sondern alle Instanzen ihre Konfiguration aus einer einheitlichen Quelle beziehen sollen, muss noch ein externer Config-Provider angegeben werden. Hierfür kann Consul verwendet werden.

Der Service definiert einen GET-Endpunkt, mit dem der aktuell verfügbare Warenbestand aus einer Produkt-Datenbank abgefragt werden kann. Mit der Annotation *@Path* wird der Endpunkt */products* definiert, die *@GET*-Annotation bestimmt, welche Methode aufgerufen wird, um HTTP-GET-Requests an diesen Endpunkt zu bearbeiten. Neben dem Abrufen aus der Datenbank wird auch noch ein eventueller Rabatt auf die Produktpreise angewandt. Dazu wird die Variable *discountPercent* verwendet, deren Wert mit der Annotation *@ConfigProperty* aus der Konfiguration des Microservice abgerufen wird. Listing 2.1 zeigt den Aufbau des Endpunkts.

Damit die Instanzen des Backend-Service durch den Client entdeckt werden können, müssen sich diese nach dem Starten bei Consul registrieren, siehe Listing 2.2. Dazu müssen unter anderem eine Instanz-ID sowie Adresse und Port, unter dem die Instanz erreichbar ist, angegeben werden.

List. 2.1: Definition des Endpunkts und Abrufen der Konfiguration

```
@Path("/products")
public class ProductResource {

    @Inject
    ProductDao productDao;

    @ConfigProperty(name = "discount.percent", defaultValue = "0")
    String discountPercent;

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<Product> getAvailableProducts() {
        var allProducts = productDao.getAllProducts();

        // Sort out products that are currently not in Stock
        List<Product> productsInStock =
            allProducts.stream().filter(product -> product.getStock() >
            0).toList();

        // Apply the configured discount
        double modifier = (100 - Integer.parseInt(discountPercent)) /
            100.0;
        productsInStock.forEach(product ->
            product.setPrice(product.getPrice() * modifier));
        return productsInStock;
    }
}
```

List. 2.2: Registrierung der Instanz bei Consul

```
ImmutableRegistration registration = ImmutableRegistration.builder()
        .id(instanceId)
        .name(appName)
        .address(serviceAddress)
        .port(Integer.parseInt(servicePort))
        .putMeta("version", appVersion)
        .build();
consulClient.agentClient().register(registration);
```

2.2 Client-Service

Der Client ist ebenfalls ein Quarkus-Microservice und kann mittels HTTP-Request Daten vom Backend-Service abfragen. Allerdings hat der Client keine Kenntnis davon, wie viele Instanzen des Backend-Service gerade verfügbar sind und unter welche Adresse diese erreichbar sind. Deshalb wird SmallRye Stork verwendet, um die bei Consul registrierten Service-Instanzen abzurufen und eine Instanz auszuwählen. Dazu muss in der Konfiguration des Client-Service angegeben werden, dass Service Discovery mit Consul durchgeführt werden soll, sowie die Adresse des Consul-Containers und ein Load-Balancing-Verfahren, mit dem die am besten geeignete Instanz ausgewählt wird (Listing 2.3). In diesem Projekt wird das Round Robin-Verfahren genutzt, die Instanzen werden also gleichmäßig der Reihe nach aufgerufen. Im REST-Client kann dann für HTTP-Requests statt der URI einer spezifischen Instanz einfach *stork://oder-service/products* verwendet werden, was Stork dann automatisch zur passenden Instanz auflöst, siehe Listing 2.4.

List. 2.3: Konfiguration des Client

```
quarkus.stork.order-service.service-discovery.type=consul
quarkus.stork.order-service.service-discovery.consul-host=localhost
quarkus.stork.order-service.service-discovery.consul-port=8500
quarkus.stork.order-service.load-balancer.type=round-robin
```

List. 2.4: HTTP-Client mit Stork

```
@RegisterRestClient(baseUri = "stork://order-service/products")
public interface ProductClient {

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    String get();
}
```

2.3 Consul

Consul wird in diesem Projekt sowohl für Service Discovery als auch für das Konfigurationsmanagement benutzt.

Für Service Discovery erfüllt Consul die Rolle der Service Registry: Alle verfügbaren Instanzen des Backend-Service registrieren sich nach dem Start bei Consul. Der Client-Service, der Service Discovery mit Stork durchführt, fragt bei Consul die verfügbaren Instanzen ab und wählt eine davon aus, an die er seine Anfrage schickt.

Für die einheitliche Konfiguration der Backend-Instanzen bietet Consul die Möglichkeit, Key/Value-Paare zu speichern, welche dann von den Services abgerufen werden können. In diesem Projekt wurden die Paare der Einfachheit halber im Webinterface von Consul hinzugefügt, dies kann aber auch über API-Aufrufe, z.B. von einem anderen Service gemacht werden. Eine Einschränkung besteht darin, dass die Key/Value-Paare beim Start der Service-Instanzen bereits verfügbar sein müssen, nachträgliche Änderungen werden von laufenden Instanzen nicht erfasst.

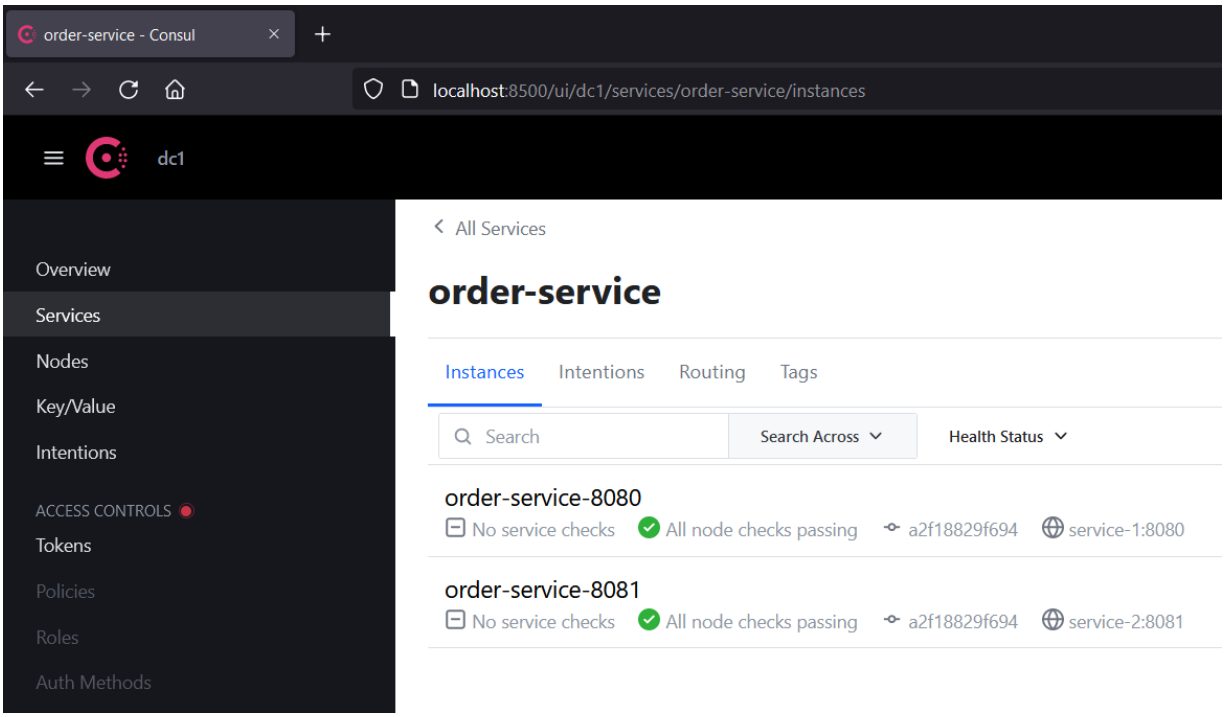

Abb. 2.2: Backend-Instanzen in Consul

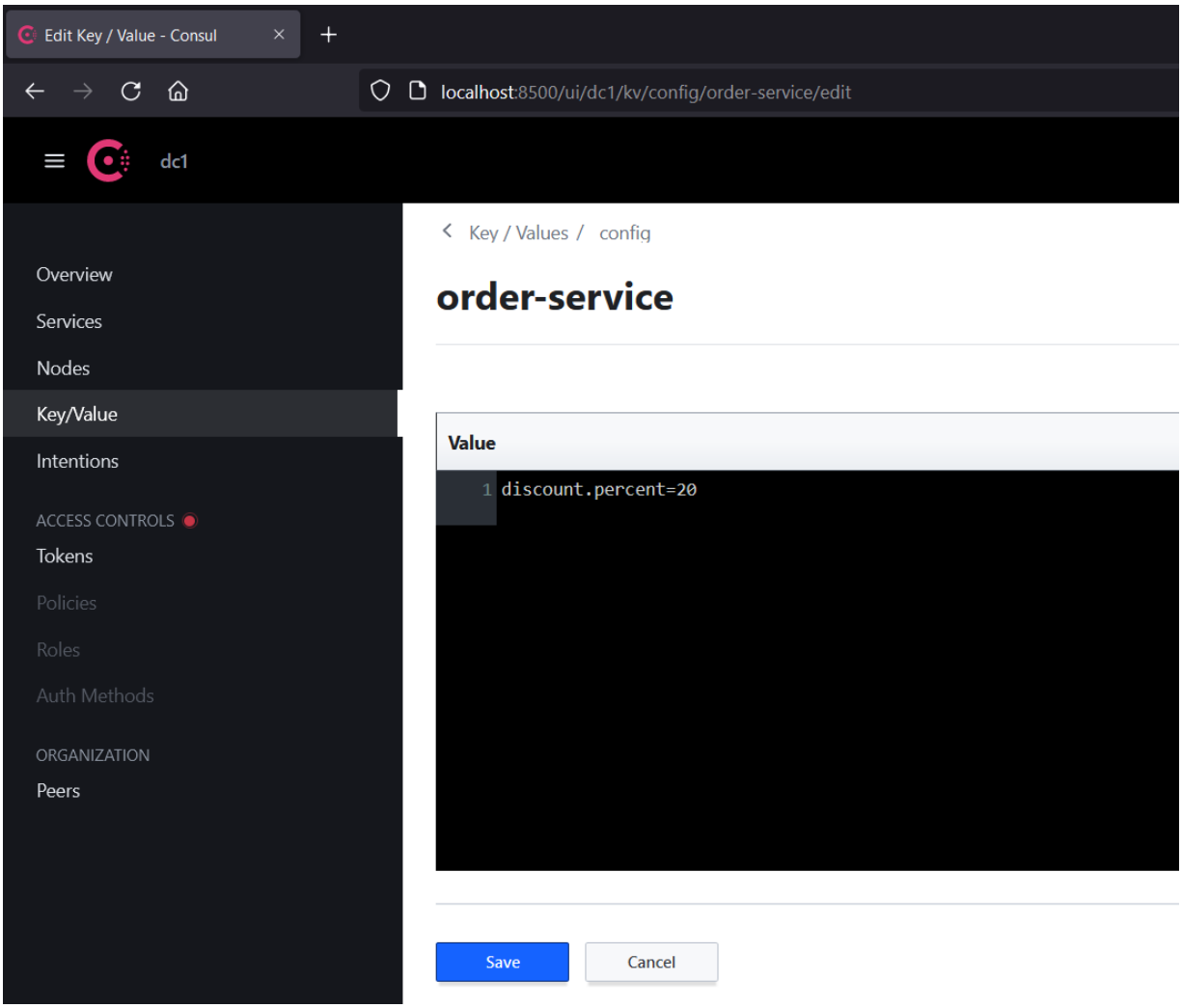

Abb. 2.3: Key/Value-Paare in Consul

3 Deployment und Demonstration

3.1 Docker

Alle Microservices sowie der Consul-Server laufen in einer Docker-Umgebung. Die Dockerfiles zum Erzeugen der Images der Client- und Backend-Services werden von Quarkus bereits automatisch erstellt. Der Einfachheit halber befinden sich in diesem Projekt alle Container im gleichen Docker-Stack, in der Realität muss das jedoch nicht der Fall sein, insbesondere die Backend-Service-Instanzen können sich auch in unterschiedlichen Umgebungen befinden.

Alle Container werden über ein gemeinsames Compose-File gestartet. Hier müssen einige Konfigurationen gemacht werden, damit Service Discovery und Konfigurationsmanagement funktionieren.

3.1.1 Backend-Service

Alle Instanzen des Backend-Service, die gestartet werden sollen, müssen im Compose-File definiert werden. Das ist nötig, weil jede Instanz eine einzigartige Adresse und einen HTTP-Port zugewiesen bekommt, siehe Zeilen 6 und 10 in Listing 3.1. Für mehr Flexibilität könnten diese Werte beim Erstellen neuer Instanzen automatisch zugewiesen werden, es muss jedoch sichergestellt sein, dass diese nach dem Starten der Instanz für die Registrierung bei Consul verfügbar sind. Quarkus bietet Beispielsweise die Möglichkeit, einen zufälligen Port zu nutzen, dieser ist jedoch nicht sofort beim Start der Anwendung bekannt und die Instanz konnte sich deshalb nicht bei Consul registrieren.

List. 3.1: Backend-Service in docker-compose.yaml

```
1  service-1:
2    container_name: order_service_1
3    image: quarkus/service-jvm
4    restart: unless-stopped
5    environment:
6      QUARKUS_HTTP_PORT: 8080
7      QUARKUS_DATASOURCE_JDBC_URL:
           jdbc:postgresql://db:5432/stupro_store
8      CONSUL_HOST: consul
9      QUARKUS_CONSUL_CONFIG_AGENT_HOST_PORT: consul:8500
10     SERVICE_ADDRESS: service-1
11   depends_on:
12     - consul
13     - db
```

3.1.2 Client-Service

In Zeile 8 wird angegeben, dass Consul für Service Discovery unter dem Hostnamen *consul* erreichbar ist, dies bezieht sich auf den Consul-Container, der im gleichen Netzwerk läuft.

List. 3.2: Client-Service in docker-compose.yaml

```
1  client:
2    container_name: client
3    image: quarkus/client-jvm
4    restart: unless-stopped
5    ports:
6      - '8090:8090'
7    environment:
8      - QUARKUS_STORK_ORDER_SERVICE_SERVICE_DISCOVERY_CONSUL_HOST=consul
9    depends_on:
10     - consul
```

3.1.3 Consul

Für den Consul-Server wird konfiguriert, welche Ports sichtbar sein müssen, um Service Discovery zu nutzen und auf das Webinterface zuzugreifen. Außerdem werden für den Consul-Agent Parameter angegeben: Consul soll im Dev-Modus gestartet werden und eine Web-UI bereitstellen.

List. 3.3: Consul in docker-compose.yaml

```
consul:
  container_name: consul
  image: consul
  restart: unless-stopped
  ports:
    - '8500:8500'
    - '8501:8501'
    - '8600:8600'
  command:
    - agent
    - -dev
    - -ui
    - -client=0.0.0.0
    - -bind=0.0.0.0
    - '--https-port=8501'
```

3.2 Demo

Wenn die Anwendung über das Compose-File gestartet wurde, kann das Webinterface des Consul-Servers verwendet werden, um zu überprüfen, ob die Backend-Service-Instanzen korrekt registriert wurden und von Stork entdeckt werden können. Hierzu kann in einem Browser *http://localhost:8500* aufgerufen werden. Im Menüpunkt *Services* sollte ein Service mit dem Namen *order-service* mit zwei laufenden Instanzen zu sehen sein.

Um Service Discovery zu testen, kann dann ein GET-Request an *http://localhost:8090/products* geschickt werden, zum Beispiel mit Postman oder cURL. Der Client wird die Anfrage durch Stork an eine der beiden Backend-Instanzen weiterleiten. Dies kann beobachtet werden, indem man den Request mehrmals abschickt und den Log-Output des Client-Containers beobachtet. Dieser gibt an, an welche Instanz Stork die Anfrage gesendet hat. Als Antwort erhält man eine Liste von verfügbaren Produkten und deren Preisen.

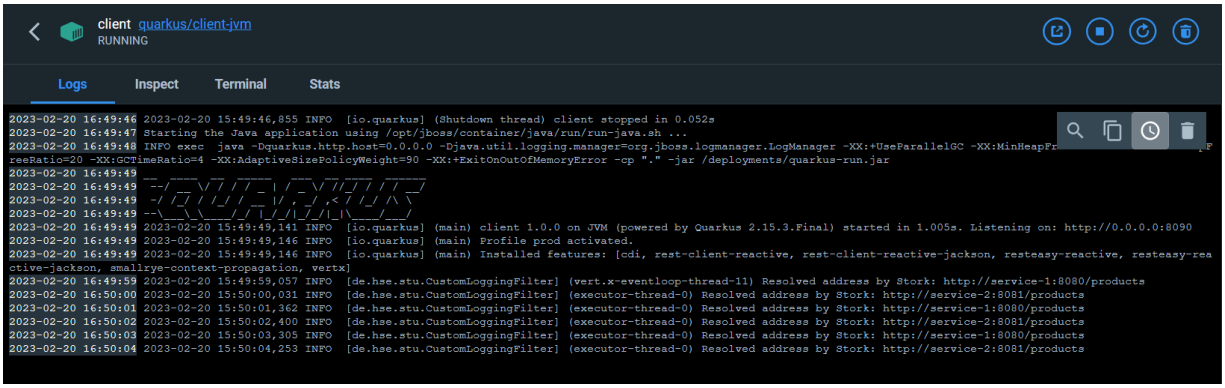

Abb. 3.1: Log-Output des Client-Containers in Docker Desktop

Zum Testen des Konfigurationsmanagements kann in der Web-UI von Consul im Menü *Key/Value* ein neues Paar mit dem Key *config/order-service* und dem Value *discount.percent=20* erstellt werden. Nach einem Neustart der Backend-Services nutzen sie die aktualisierte Konfiguration und bei weiteren GET-Requests an *http://localhost:8090/products* wird nun der eben konfigurierte Rabatt auf die Produktpreise angewandt.

4 Fazit und Ausblick

Mit SmallRye Stork und Consul war es mit relativ wenig Aufwand möglich, Service Discovery zu implementieren. Die größte Herausforderung lag hier darin, dass die einzelnen Instanzen ihre ID, mit der sie sich bei Consul registrieren, sowie den Port und die Adresse, unter der sie erreichbar sind, schon direkt nach dem Starten kennen müssen. Für dieses Projekt wurden diese Werte im Docker Compose-File statisch für jede Instanz festgelegt. Für eine reale Anwendung sollte dies dynamisch beim Start einer Instanz geschehen, damit Instanzen flexibel erstellt und entfernt werden können.

Das Konfigurieren von Services mit SmallRye Config war grundsätzlich ebenfalls leicht möglich, da Quarkus standardmäßig Config nutzt, um Konfigurationen beispielsweise aus Properties-Dateien zu lesen und im Code per Annotation verfügbar zu machen. Um mehrere Instanzen eines Service aus derselben Quelle konfigurieren zu können, benötigt Config noch einen zusätzlichen externen Config Provider. Da hierfür ebenfalls Consul verwendet werden konnte, war dies nur ein geringer Zusatzaufwand. Nachteil dieser Lösung ist es, dass die Konfiguration nicht zur Laufzeit der Services aktualisiert werden kann, die Services müssen neu gestartet werden, um Änderungen zu erfassen. Dies kann zwar auch erwünscht sein, sollte aber beachtet werden, da es bei einer Änderung der Konfiguration zur Laufzeit der Anwendung dazu kommen kann, dass nachträglich gestartete Services die aktualisierte Konfiguration verwenden, während ältere Instanzen die ursprüngliche Konfiguration verwenden. Ist ein "Live-Update"der Konfigurationswerte erwünscht, könnte dies möglicherweise durch Implementierung einer eigenen Config-Source realisiert werden, was von SmallRye Config unterstützt wird.

Literatur

[1] *What is Quarkus?* https://quarkus.io/about/.

[2] *Quarkus - einfach erklärt.* https://blog.doubleslash.de/quarkus-einfach-erklaert/.

[3] *Smallrye Stork.* https://smallrye.io/smallrye-stork.

[4] *SmallRye Stork meets Quarkus.* https://quarkus.io/blog/smallrye-stork-intro/.

[5] *SmallRye Config.* https://smallrye.io/smallrye-config/Main/.

[6] *Consul Service Discovery.* http://smallrye.io/smallrye-stork/1.4.1/service-discovery/consul/.

[7] *Quarkus: Configuring your Application.* https://quarkus.io/guides/config.