



HTBL Imst
AUFBAULEHRGANG FÜR INFORMATIK



Übungszettel – Microservices

Name: Michael Bogensberger

Datum: 15.06.2022

1	Inhalt	
2	Was sind Microservices?	3
3	Was ist ein Message Broker?	3
3.1	Was ist RabbitMQ?	3
4	Inbetriebnahme	5
5	Architekturbeschreibung	5
5.1	API Gateway	5
5.1.1	Spring Cloud Gateway	5
5.2	Netflix Eureka	5
5.3	Service-Discovery	6
5.4	C4-Diagramme	7
5.4.1	System Context	7
5.4.2	Component View	7
5.5	API-Gateway	8
5.5.1	Dependencies	8
5.6	RabbitMQ – Delivery	9
5.7	Payment verifizieren	10
6	Abbildungsverzeichnis	12

2 Was sind Microservices?

Microservices sind ein architekturbezogener und organisatorischer Ansatz in der Softwareentwicklung, bei dem Software aus kleinen unabhängigen Services besteht, die über sorgfältig definierte APIs kommunizieren.

Sie sind ein Architekturkonzept für die Anwendungsentwicklung. Als Architektur-Frameworks sind Microservices verteilt und lose gekoppelt, sodass die Änderungen eines Teams nicht dazu führen können, dass die gesamte Anwendung nicht mehr funktioniert. Der Vorteil bei der Verwendung von Microservices liegt darin, dass Entwicklungsteams schnell neue Anwendungskomponenten bauen können, um sich ändernden geschäftlichen Anforderungen gerecht zu werden.

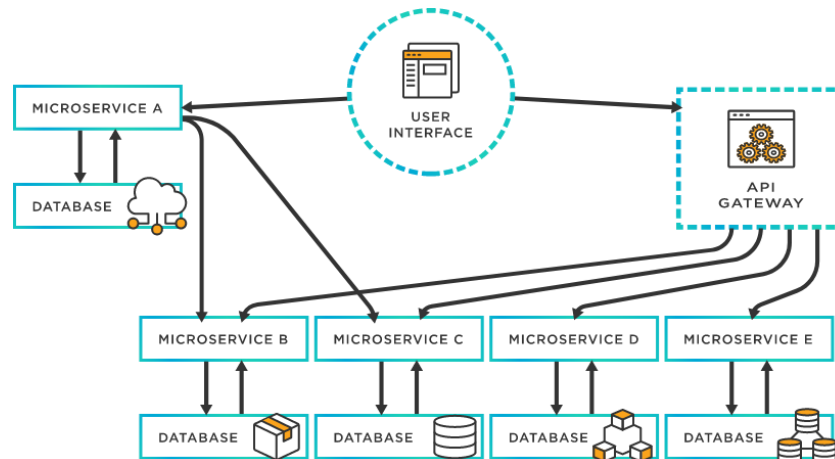


Abbildung 1 Microservices Grafik

3 Was ist ein Message Broker?

In der IT müssen ständig Nachrichten von einem Dienst zum anderen geleitet werden. Das muss auf eine kontrollierte Weise geschehen, sonst blockieren sich Nachrichten gegenseitig, es entsteht ein Stau und Prozesse können nicht optimal ablaufen. Damit Anwendungen problemlos miteinander kommunizieren, ist es sinnvoll, einen Mittelsmann einzuschalten – einen Dienst, der die Verteilung der Nachrichten übernimmt. Diesen nennt man Messaging Broker.

Zunächst hat der Message Broker als Middle Ware die Aufgabe, Nachrichten zu empfangen und an einen oder mehrere Empfänger weiterzuleiten. Nachrichten können Mails aber auch Informationen von Sensoren oder sonstige Daten sein. Dabei passt er das Nachrichtenformat an die jeweiligen Empfängersysteme an. Das hat den Vorteil, dass im Netz ein "Vermittler" etabliert wird, der ankommende Nachrichten zentral verteilt und diese dabei in die Sprache der Empfänger übersetzt.

3.1 Was ist RabbitMQ?

RabbitMQ basiert auf der Idee des Advanced Message Queuing Protocols (AMQP). Der große Vorteil von AMQP: Sender und Empfänger müssen nicht die gleiche Programmiersprache verstehen. Inzwischen hat sich der Messaging Broker etwas von AMQP gelöst und geht dank Plug-ins auch mit anderen Nachrichtenprotokollen wie STOMP oder MQTT um – die Idee bleibt aber die gleiche: Zwischen dem Produzenten und dem Konsumenten einer Nachricht liegt eine Warteschlange. In dieser werden die Messages zwischengelagert. Einen passenden Guide findet man hier: [IONOS-RabbitMQ-Guide](#).

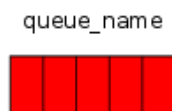
RabbitMQ ist ein Message Broker: Es akzeptiert und leitet Nachrichten weiter. Sie können es sich wie ein Postamt vorstellen: Wenn Sie die Post, die Sie aufgeben möchten, in einen Briefkasten werfen, können Sie sicher sein, dass der Briefbote die Post schließlich Ihrem Empfänger zustellt. In dieser Analogie ist RabbitMQ ein Briefkasten, ein Postamt und ein Briefträger.

Der Hauptunterschied zwischen RabbitMQ und dem Postamt besteht darin, dass es nicht mit Papier umgeht, sondern binäre Datenblöcke – Nachrichten – akzeptiert, speichert und weiterleitet.

Produzieren heißt nichts anderes als versenden. Ein Programm, das Nachrichten sendet, ist ein **Producer**:



Warteschlange (**Queue**) ist der Name für einen Briefkasten, der sich in RabbitMQ befindet. Obwohl Nachrichten durch RabbitMQ und Ihre Anwendungen fließen, können sie nur in einer Warteschlange gespeichert werden. Eine Warteschlange ist nur durch die Speicher- und Festplattenbeschränkungen des Hosts gebunden, sie ist im Wesentlichen ein großer Nachrichtenpuffer. Viele Producer können Nachrichten senden, die an eine Warteschlange gehen, und viele Consumer können versuchen, Daten aus einer Warteschlange zu empfangen. So stellen wir eine Warteschlange dar:



Konsumieren hat eine ähnliche Bedeutung wie Empfangen. Ein **Consumer** ist ein Programm, das hauptsächlich auf den Empfang von Nachrichten wartet:



Im Diagramm unten ist „P“ unser Producer und „C“ unser Consumer. Das Kästchen in der Mitte ist eine Queue.



4 Inbetriebnahme

Möchte man das Projekt in Betrieb nehmen, so muss man die `docker-compose.yml` Datei ausführen. Hier ist nur wichtig darauf zu achten, dass die Ports nicht bereits belegt sind. Ansonsten kann man die Docker Container direkt starten. Nun kann man auch schon die jeweiligen Java Programme starten. In der Startreihenfolge ist die Service-Discovery optimalerweise Nr. 1. Danach folgen dann Orders, Stock und das API Gateway.

5 Architekturbeschreibung

Die Applikation besteht aus folgenden Komponenten:

- Orders Microservice (Spring Boot Applikation nach einer Ports And Adapters-Architektur und mit Implementierung einiger taktischer DDD-Muster)
- Stock Microservice (Spring Boot Applikation ohne explizite Architektur)
- Delivery Microserver(Spring Boot Applikation ohne explizite Architektur)
- API Gateway (Spring Boot Applikation mit Spring Cloud Gateway)
- Service Discovery (Spring Boot Applikation mit Netflix Eureka)
- Vanilla JavaScript Frontend in Form von 3 Seiten (Webshop, Stock, Delivery), die die MS-Infrastruktur (Backend) über das API-Gateway verwenden.

5.1 API Gateway

Das API Gateway sorgt für die Entkoppelung von Clients und Services. Dabei bildet es den alleinigen Kontaktpunkt für eingehenden und ausgehenden Traffic. Ein Microservices-Backend kann viele miteinander interagierende Services beinhalten, wobei diese eine heterogene Struktur hinsichtlich URLs und Protokollen aufweisen können.

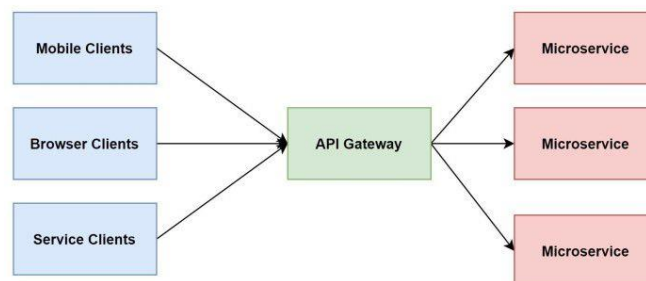


Abbildung 2 API Gateway Grafik

5.1.1 Spring Cloud Gateway

Das Spring Cloud Gateway bietet in Verbindung mit den anderen Spring Projekten wie Security, Service Discovery, Logging und Monitoring eine leistungsfähige Lösung und steht in Konkurrenz zu anderen API Gateway Lösungen. Spring Cloud Gateway bietet den Vorteil, dass es nahtlos integriert werden kann.

5.2 Netflix Eureka

Netflix Eureka ist eine REST-basierte Middleware, die für die Erkennung und den Lastausgleich (load balancing) von Webanwendungen entwickelt wurde.

5.3 Service-Discovery

Stellen wir uns mehrere Microservices vor, die eine mehr oder weniger komplexe Anwendung bilden. Diese werden irgendwie miteinander kommunizieren (z. B. API Rest).

Eine auf Microservices basierende Anwendung wird normalerweise in virtualisierten oder containerisierten Umgebungen ausgeführt. Die Anzahl der Instanzen eines Dienstes und seiner Standorte ändert sich dynamisch. Wir müssen wissen, wo sich diese Instanzen befinden und wie sie heißen, damit Anfragen beim Ziel-Microservice ankommen. Hier kommen Taktiken wie Service Discovery ins Spiel.

Der Service-Discovery hilft uns zu wissen, wo sich jede Instanz befindet. Auf diese Weise fungiert eine Service Discovery-Komponente als Register, in dem die Adressen aller Instanzen verfolgt werden. Die Instanzen haben dynamisch zugewiesene Netzwerkpfade. Folglich muss ein Client, wenn er eine Anforderung an einen Dienst stellen möchte, einen Service-Discovery Mechanismus verwenden.

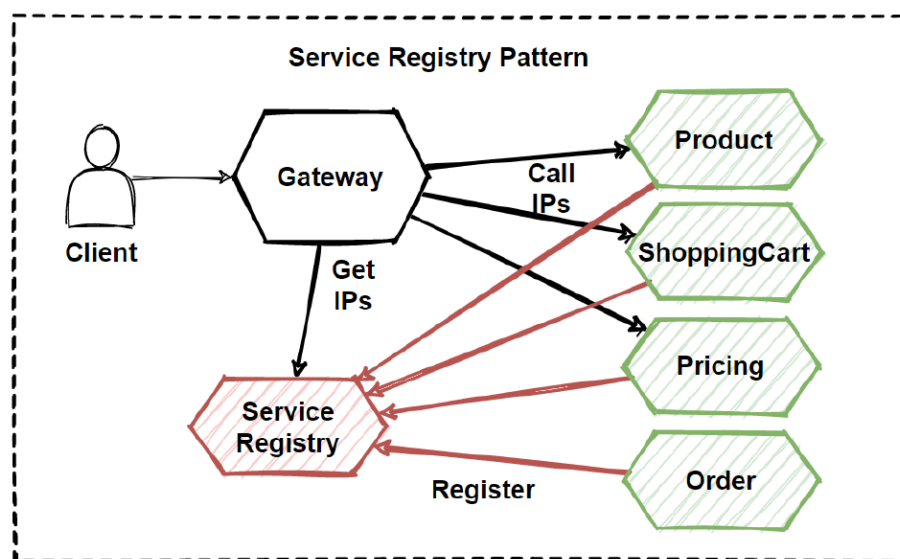


Abbildung 3 Service Discovery Grafik

5.4 C4-Diagramme

In folgendem Abschnitt sind die jeweiligen C4-Diagramme zu sehen. Hier wird immer weiter ins System geblickt.

5.4.1 System Context

In folgendem C4-Diagramm ist der System Context zu sehen. Sprich, hier sieht man das System von ganz außen.

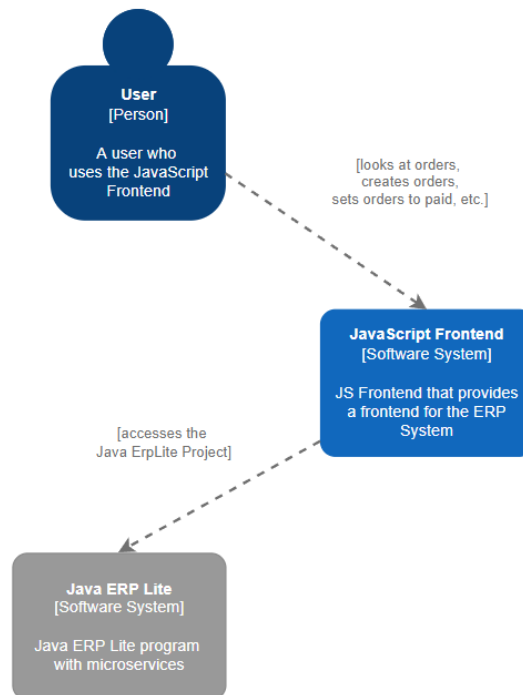


Abbildung 4 C4-Diagramm System Context

5.4.2 Component View

Hier ist das System mit den jeweiligen Microservices zu sehen. Hinweis: Das ursprüngliche C4-Shema wurde etwas verändert, damit sich die Grafik etwas besser darstellen lässt.

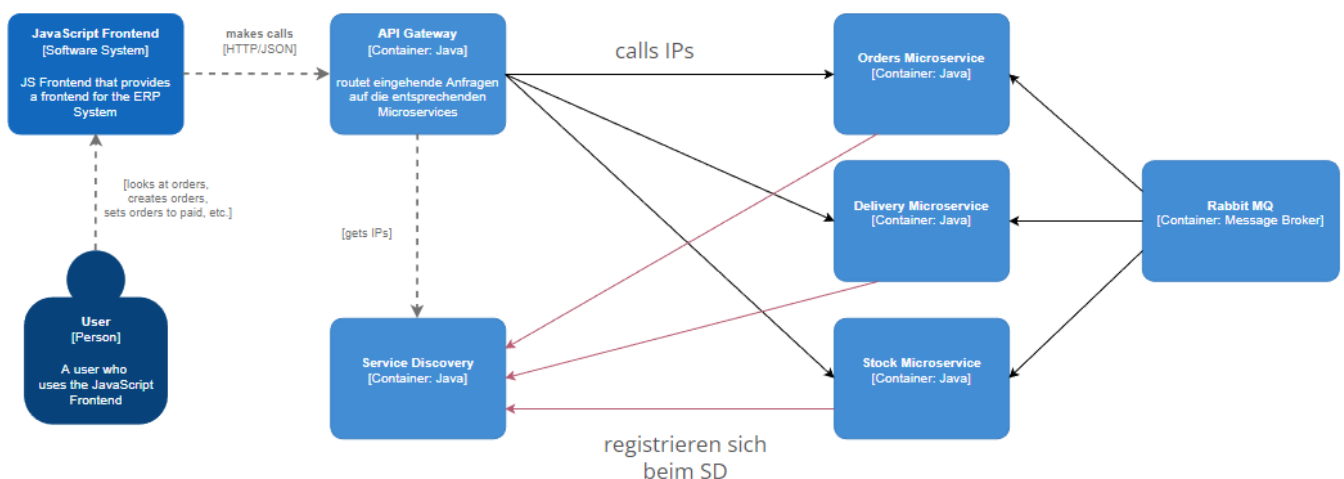


Abbildung 5 C4-Diagramm Component View

5.5 API-Gateway

Sieht man in das ApiGateway Projekt, so sieht man, dass die Config für das Gateway in der application.yml zu finden ist.

In folgender Grafik ist die application.yml zu sehen. Hier wird unter anderem der Port und der Name festgelegt. Des weiteren sehen wir hier, dass ein Service-Discovery verwendet wird.

```
# TODO: API-Gateway für Absicherung der Endpunkte https://github.com/timtebeek/spring-security-samples/blob/main/spring-cloud-gateway-oidc-tokenrelay/README.adoc
server:
  port: 9999
spring:
  application:
    name: apigateway
  cloud:
    gateway:
      discovery:
        locator:
          # Verwendung der Service-Discovery für das Ansprechen der Microservices
          enabled: true
          # Alle in der Service-Discovery registrierten Services in Lower-Case ansprechen (Standardmäßig UPPERCASE)
          lower-case-service-id: true
```

Abbildung 6 API-Gateway Config 1

Weiter unten sehen wir die Konfiguration der Routen. Hier sehen wir auch das zum Beispiel der Load Balancer für die Ermittlung der Routen verwendet wird. Ansonsten finden wir in der application.yml Datei noch Konfigurationen zu CORS sowie zum Actuator.

```
routes:
- id: orders
  # lb bedeutet hier, dass der Load-Balancer für die Ermittlung der eigentlichen Route verwendet wird.
  # Dieser wiederum fragt an der Service-Registry nach dem korrekten Server und Port (siehe discovery-Zweig)
  # Eine Anfrage mit dem Pfad /api/v1/orders/** wird an das Service geschickt, das in der Service-Registry
  # mit dem Namen erpliteorders registriert ist (ServerIP:Port wird ersetzt, Rest wird übernommen).
  uri: lb://erpliteorders
  predicates:
    - Path=/api/v1/orders/**
  filters:
    # Falls ein Response-Location header (wie z.B. beim Erstellen einer
    # Ressource über POST-Request) vom Backend-Service im Response-Header zurückkommt,
    # wird der Server:Port-Teil mit den Serverdaten des API-Gateways überschrieben.
    - RewriteLocationResponseHeader=AS_IN_REQUEST, Location, ,
- id: stock
  uri: lb://erplitestock
  predicates:
    - Path=/stock/**
- id: delivery
  uri: lb://erplitdelivery
  predicates:
    - Path=/delivery/**
```

Abbildung 7 API-Gateway Config 2

5.5.1 Dependencies

- Authentication with the OpenID Provider is handled through `org.springframework.boot:spring-boot-starter-oauth2-client`
- Gateway functionality is offered through `org.springframework.cloud:spring-cloud-starter-gateway`
- Relaying the token to the proxied resource servers comes from `org.springframework.cloud:spring-cloud-security`

5.6 RabbitMQ – Delivery

Hier ist die Anwendung von RabbitMQ im Delivery Service zu sehen. Dafür muss zunächst RabbitMQ in der `application.properties` konfiguriert werden. Hier sehen wir den host, das Passwort, den Usernamen sowie den Port.

```
#RabbitMq
spring.rabbitmq.host=localhost
spring.rabbitmq.password=guest
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
```

Abbildung 8 RabbitMQ application.properties

Um Nachrichten aus der Queue zu lesen, müssen Sie die Annotation `@RabbitListener` verwenden. Diese Anmerkung nimmt einen Queue-namen an. Der folgende Code gibt beispielsweise zuerst aus, dass eine Delivery Event initiiert wurde. Dann wird gecheckt, ob die Delivery bereits existiert. Wenn ja, dann wird einfach returned und geloggt das die Delivery rejected wurde. Ansonsten wird eine neue `orderDelivery` angelegt. Nun wird die Delivery gespeichert und dieses Event ebenfalls geloggt. Nun wird noch das Event gestartet, dass sich die Order in Versand befindet. Zudem wird wieder geloggt.

```
@RabbitListener(queues = "q.initiate_delivery")
public void receive(OrderInitiateDeliveryEvent orderInitiateDeliveryEvent) {
    Logger.getLogger(this.getClass().getName()).log(Level.INFO, "Initiate Delivery event received in delivery MS for order# " +
        orderInitiateDeliveryEvent.orderID() + "!");

    if(orderDeliveryRepository.findByOrderID(orderInitiateDeliveryEvent.orderID()).isPresent()) {
        Logger.getLogger(this.getClass().getName()).log(Level.INFO, "Initiate Delivery event for order# " + orderInitiateDeliveryEvent.orderID()
            + " rejected, order delivery already initiated!");
        return;
    }

    OrderDelivery orderDelivery = new OrderDelivery(
        null,
        orderInitiateDeliveryEvent.orderID(),
        orderInitiateDeliveryEvent.customerID(),
        orderInitiateDeliveryEvent.customerFirstname(),
        orderInitiateDeliveryEvent.customerLastname(),
        orderInitiateDeliveryEvent.customerEmail(),
        orderInitiateDeliveryEvent.customerStreet(),
        orderInitiateDeliveryEvent.customerZipcode(),
        orderInitiateDeliveryEvent.customerCity(),
        orderInitiateDeliveryEvent.customerCountry(),
        false);

    orderDeliveryRepository.save(orderDelivery);
    Logger.getLogger(this.getClass().getName()).log(Level.INFO, "Delivery for order# " + orderInitiateDeliveryEvent.orderID() + " initiated and
        saved in delivery DB!");

    //Immediately publish event that order is in delivery
    this.template.convertAndSend("q.order_in_delivery", new OrderInDeliveryEvent(orderDelivery.getOrderID()));
    Logger.getLogger(this.getClass().getName()).log(Level.INFO, "Order in delivery RabbitMQ event for order# " + orderDelivery.getOrderID() + "
        published!");
}
```

Abbildung 9 Delivery - RabbitMQ

5.7 Payment verifizieren

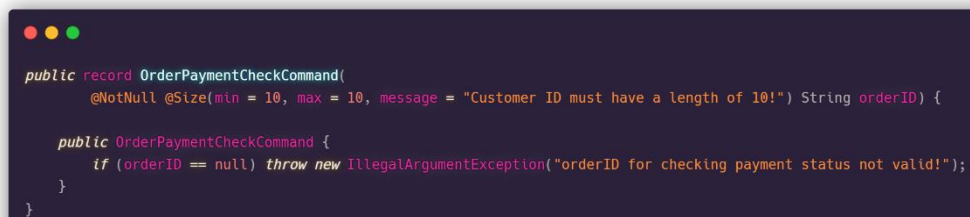
Hier ist der `OrderRestController` zu sehen. Genauer gesagt ist jenes Mapping zu sehen, dass ein Payment auf verifiziert setzt. Hier wird zunächst wieder geloggt, dass payment verifizieren gehandelt wird. Nun wird vom `oderCommandService` die `handle` Methode mit einem neuen `OrderPaymentCheckCommand` mit der jeweiligen `orderid` ausgeführt. Nun wird nur noch eine `accepted` Response zurückgegeben.



```
@PostMapping("/orders/checkpayment/{orderid}")
public ResponseEntity validatePaymentForOrderWithId(@PathVariable String orderid) {
    Logger.getLogger(this.getClass().getName()).log(Level.INFO, "Handling check payment for order api request ...");
    this.orderCommandService.handle(new OrderPaymentCheckCommand(orderid));
    return ResponseEntity.accepted().body("Order payment check executed. Order payment ok!");
}
```

Abbildung 10 checkpayment Mapping

Hier ist der `OrderPaymentCheckCommand` zu sehen. Hier handelt es sich um einen record. Des Weiteren ist hier die Validierung der `orderId` zu sehen (`@Size`, `Exception` usw.).



```
public record OrderPaymentCheckCommand(
    @NotNull @Size(min = 10, max = 10, message = "Customer ID must have a length of 10!") String orderId) {

    public OrderPaymentCheckCommand {
        if (orderId == null) throw new IllegalArgumentException("orderId for checking payment status not valid!");
    }
}
```

Abbildung 11 OderPaymentCheckCommand

Hier sehen wir nun die Implementierung des `OrderCommandServices` (`OrderCommandServiceImpl`). Genauer gesagt sehen wir hier die `handle` Methode für den `OrderPaymentCheckCommand`. Hier wird zunächst einiges geloggt und zum Beispiel gecheckt, ob die ID valide ist. Nun wird die Order herangezogen und der `OrderState` zu verifiziert gesetzt. Nun wird die Order noch geupdated. Außerdem wird hier wieder mit dem Logger geloggt. Wichtig ist hier im Prinzip nur zu wissen, dass hier viel validiert und abgefangen wird und schlussendlich die Order auf verifiziert gesetzt wird.

```
@Override
@Transactional
public void handle(OrderPaymentCheckCommand orderPaymentCheckCommand) throws OrderPaymentCheckFailedException {
    Logger.getLogger(this.getClass().getName()).log(Level.INFO, "Handling order payment check command ...");
    if (orderPaymentCheckCommand == null)
        throw new OrderPaymentCheckFailedException("Empty command for order payment check!");
    if (!OrderID.isValid(orderPaymentCheckCommand.orderID()))
        throw new OrderPaymentCheckFailedException("Order ID for order payment check not valid!");
    Optional<Order> optionalOrderToCheck = this.orderRepository.getByID(new OrderID(orderPaymentCheckCommand.orderID()));
    if (optionalOrderToCheck.isPresent()) {
        Order order = optionalOrderToCheck.get();
        try {
            order.orderStateTransitionTo(OrderState.PAYMENT_VERIFIED);
            this.orderRepository.updateOrderWithNewState(order);
            this.orderOutgoingMessageRelay.publish(new OrderPaymentValidatedEvent(OrderResponseMapper.toResponseFromDomain(order)));
            Logger.getLogger(this.getClass().getName()).log(Level.INFO, "Payment validated event published!");
        } catch (OrderStateChangeNotPossibleException orderStateChangeNotPossibleException) {
            throw new OrderPaymentCheckFailedException("Order payment check not possible. Order in wrong state! " +
                orderStateChangeNotPossibleException.getMessage());
        }
    } else {
        throw new OrderPaymentCheckFailedException("Order with Id " + orderPaymentCheckCommand.orderID() + " not found for payment check!");
    }
}
```

Abbildung 12 handle OrderPaymentCheckCommand

6 Abbildungsverzeichnis

Abbildung 1 Microservices Grafik.....	3
Abbildung 2 API Gateway Grafik.....	5
Abbildung 3 Service Discovery Grafik	6
Abbildung 4 C4-Diagramm System Context	7
Abbildung 5 C4-Diagramm Component View.....	7
Abbildung 6 API-Gateway Config 1.....	8
Abbildung 7 API-Gateway Config 2.....	8
Abbildung 8 RabbitMQ application.properties	9
Abbildung 9 Delivery - RabbitMQ.....	9
Abbildung 10 checkpayment Mapping	10
Abbildung 11 OderPaymentCheckCommand.....	10
Abbildung 12 handle OrderPaymentCheckCommand	11