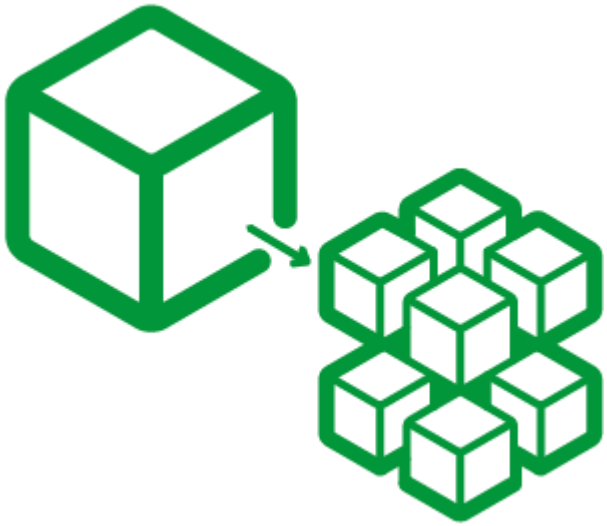
EAI: Implementation ShopStantly

Event-Processing

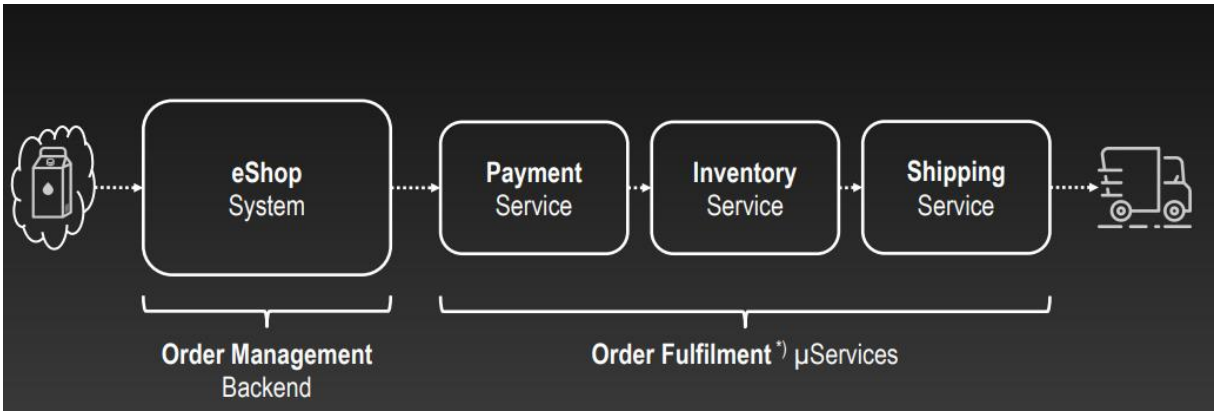


EAI, 2018
Autoren: Lukas Weber, Lukas Gehrig, Adrian Mathys, Murat Kelleci
Dozent: Prof. Dr. Andreas Martin
Ort: Olten
Datum: 10.12.2018

Inhalt

1. Auftrag:

Über einen eShop können Artikel gekauft werden via virtuellem Assistent. Diese werden nach der Eingabe bezahlt, werden aus dem Inventar geholt und dem Kunden versendet:



- Der virtuelle Assistent ist das einzige User Interface das gebraucht wird
- Der virtuelle Assistent holt sich die Informationen, welcher er braucht um die Vorlieben des Kunden zu kennen und eine Bestellung abzugeben
- Die Bestellung vom virtuellen Assistenten ist der Trigger für den Beginn für das Order Fulfillment

1. Use Case

Lukas ist ein Kunde, welcher immer wieder bei ShopStantly einkauft. Wenn Lukas merkt, dass er etwas benötigt, was nicht im Haushalt vorhanden ist, versucht er es über den Google Assistent ShopStantly zu bestellen. In unserem Use Case geht es dabei um das bestellen von Schampoo.

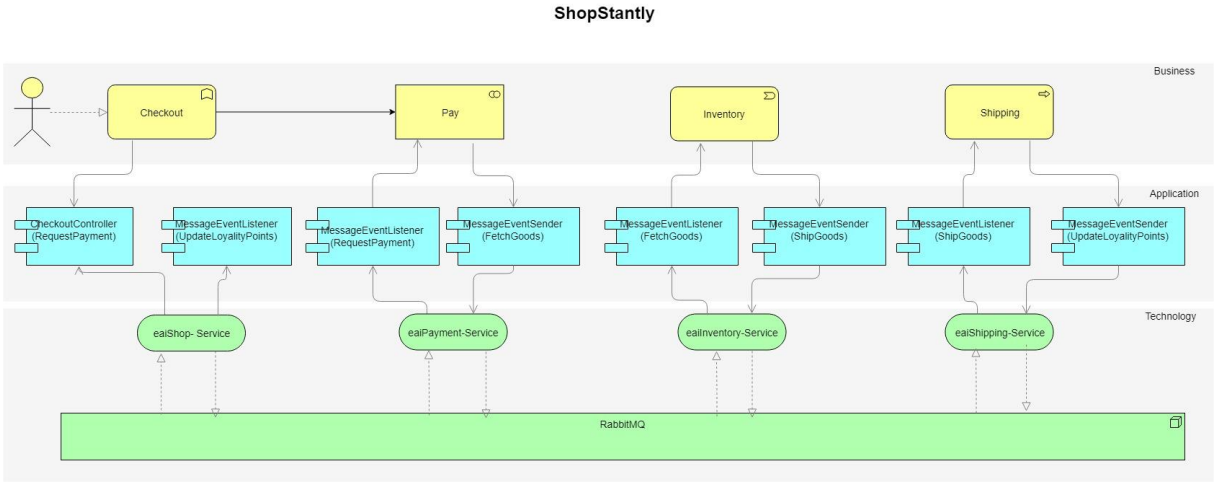
Da Lukas sehr leicht von ShopStantly zu überzeugen ist, bestellt er gleich mehrere Shampoos miteinander.

https://app.botsociety.io/s/5ba5d21153567c726aee7e10?p=9d5bbfc6e726bad51ba361fd765ed1fe4e1e924b

Sobald die ShopStantly den Warenkorb gefüllt hat und die Konversation beendet ist, wird er Prozess der sich um das Payment, Inventory und Shipping sorgt, ausgelöst.

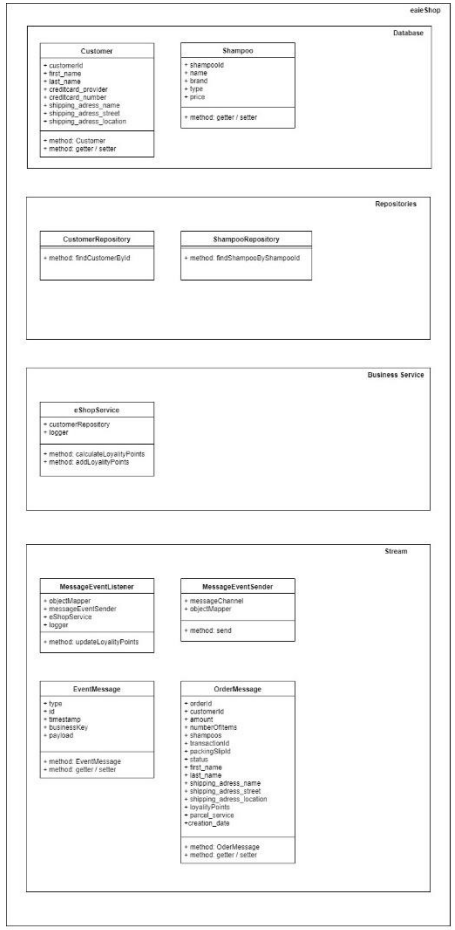
2. Übersicht

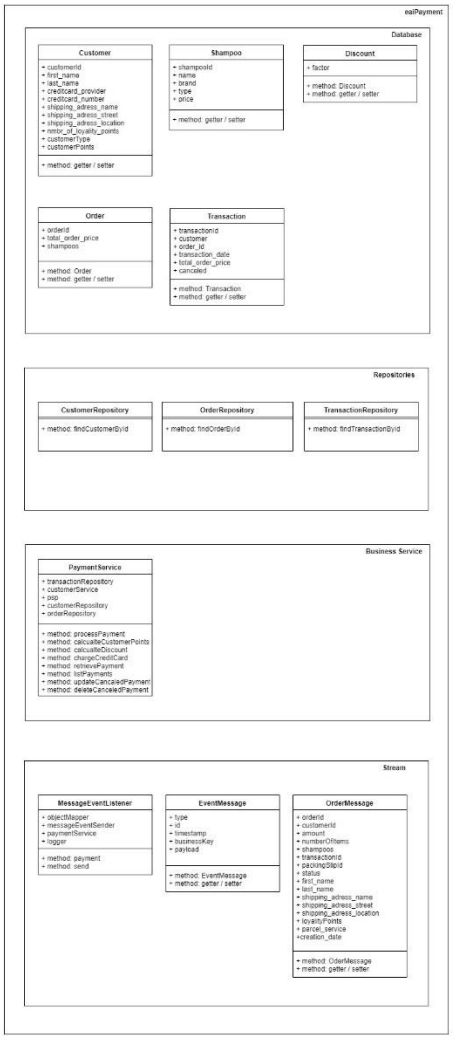
ShopStantly besteht aus 4 Microservices, welche zusammen über den Message-Broker RabbitMQ kommunizieren. Der Anstoss des Prozesses wird mit einem REST-Call dynamicDomain/checkout angestossen.

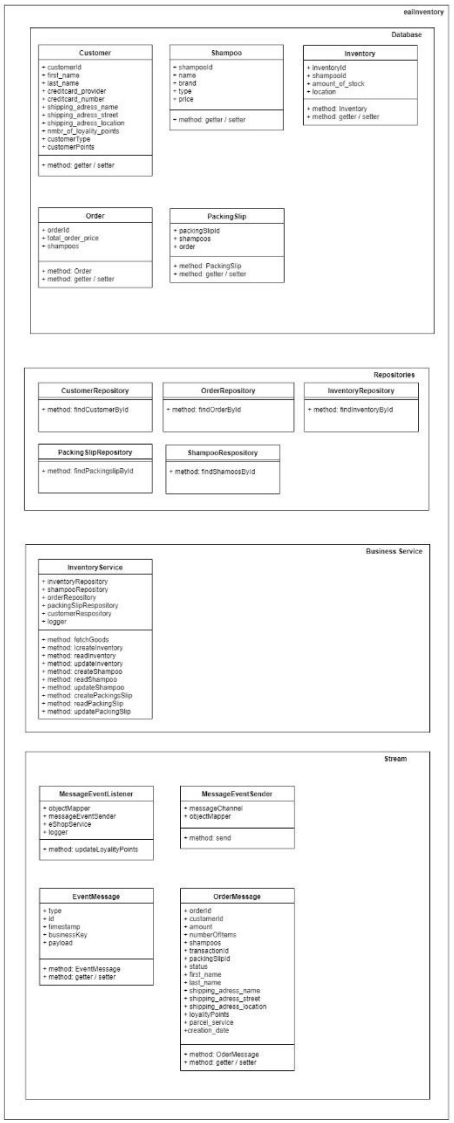


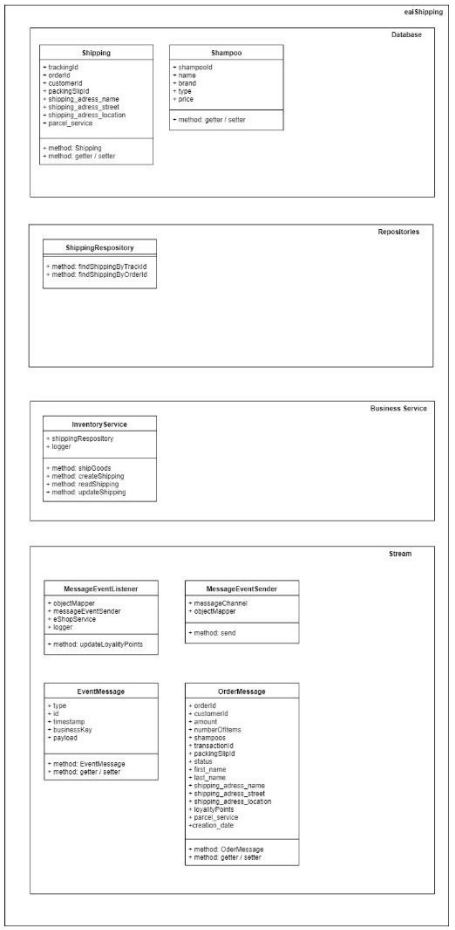
Jeder Service ist bei RabbitMQ angemeldet und hört auf ein definiertes Stichwort (Queue Name), sodass der Service Daten senden und die für ihn gedachten Daten entgegennehmen kann. Sobald ein Service eine Anfrage durchgeführt hat, sendet dieser eine neue Mitteilung über den Message-Broker an den nächsten Service, bis der ganze Prozess mit dem Shipping und UpdateLoyaltyPoints beendet wird.

3. Methoden in den Services









4. Code Fragmente

In diesem Teil werden Code-Fragmente und deren Funktionalität in Textform aufgezeigt, welche für den erfassten Use Case wichtig sind.

2. Order Message

Die Mitteilung, welche erstellt und über den Message-Broker geschickt wird um mit den anderen Services zu interagieren beinhaltet sämtliche notwendigen Informationen, um den Prozess komplett durchführen zu können. Angereichert werden die Informationen jeweils vom entsprechenden Service, welcher die Metadaten führt. Der Preis wird in der OrderMessage berechnet und mit der Variable «amount» mitgeliefert. Dieser berechnet sich aus der Addition aller gekauften Produkte.

```
public class OrderMessage {
    private String orderId;
    private String customerId;
    private Double amount;
    private Integer numberOfItems;
    private List<Shampoo> shampoos;
    private String transactionId;
    private String trackingId;
    private String packingSlipId;
    private String status;
    private String first_name;
    private String last_name;
    private String shipping_address_name;
    private String shipping_address_street;
    private String shipping_address_location;
    private String loyalityPoints;
    private String parcel_service;
    private Date creation_date;

    public OrderMessage() {
    }

    public OrderMessage(String orderId, String customerId, Double amount, Integer numberOfItems, List<Shampoo> shampoos, String
first_name, String last_name, String shipping_address_name, String shipping_address_street, String shipping_address_location, String
parcel_service, String status, String string) {
        this.orderId = orderId;
        this.customerId = customerId;
        if (amount == 0.00) {
                    double price = 0.00;
                    for (Shampoo shampoo : shampoos) {
                                price += shampoo.getPrice();
                    }
                    this.amount = price;
        } else {
           this.amount = amount;
        }
        this.numberOfItems = numberOfItems;
        this.shampoos = shampoos;
           this.first_name = first_name;
           this.last_name = last_name;
           this.shipping_address_name = shipping_address_name;
           this.shipping_address_street = shipping_address_street;
           this.shipping_address_location = shipping_address_location;
           this.parcel_service = parcel_service;
        this.status = status;
        this.creation_date = new Timestamp(System.currentTimeMillis());
```

3. eShop

1. Initial Message

Wie im Use Case aufgezeigt hat Lukas mehrere Shampoos bestellt. Die Adresse ist bereits bekannt aufgrund von früheren Käufen. Diese können vom System nur noch abgerufen werden. Hier werden Fake-Daten verwendet, die «hard» erfasst wurden. Normalerweise würden die Adressdaten aus einer Datenbank abgefragt werden.

```
    @GetMapping(path = "/checkout", produces = "text/plain")
    public ResponseEntity<String> getCheckout(){
           List<Shampoo> shampoos = new ArrayList<>();

           // Generate some Shampoos
           shampoos.add(new Shampoo(Integer.parseInt(UUID.randomUUID().toString()), "Deep Space", "Axe", "AllinOneXL", 8.30));
           shampoos.add(new Shampoo(Integer.parseInt(UUID.randomUUID().toString()), "Africa", "Axe", "AllinOneXL", 9.30));
           shampoos.add(new Shampoo(Integer.parseInt(UUID.randomUUID().toString()), "Adrenaline", "Axe", "AllinOneXL", 5.30));
           shampoos.add(new Shampoo(Integer.parseInt(UUID.randomUUID().toString()), "Excite", "Axe", "AllinOneXL", 10.30));

        OrderMessage orderMessage = new OrderMessage(
                    UUID.randomUUID().toString(), // OrderId (New)
                    "1", // CustomerId
                    0.00, // Amount - Double // Wenn nichts mitgegeben wird das berechnet von den Shampoos
                    3, //NumersOfItem - Integer
                    shampoos, // order
                    "checkout", //Status
                    "Lukas", // First Name
                    "Gehrig", // Last Name
                    "Lukas Gehrig", // shipping_address_name
                    "Musterstrasse 1", // shipping_address_street
                    "8000 Zürich", // shipping_address_location
                    "", // loyalityPoints
                    "DHL" // parcel_service
                    );
        eventSender.send(new EventMessage<>("RequestPayment", orderMessage));
        return ResponseEntity.ok(orderMessage.toString());
    }
}
```

2. updateLoyalityPoints

Wenn der Prozess erfolgreich durchgelaufen ist, werden am Schluss die loyalityPoints für den Kunden nachgeführt. Da der Kunde mit seinem Kundenstamm auf dem eShop geführt wird, muss an diesen Service eine Mitteilung mit den loyaltyPoints gesendet werden. Hier ist ersichtlich wie die Daten aus der Message extrahiert, und im eShopService aktualisiert werden.

Zuerst werden die neuen loyalityPoints berechnet und noch nicht effektiv gesetzt. Dies hat den Vorteil, dass in einer allfälligen Erweiterung des Programmes die Berechnung der Punkte schon abgegrenzt wurde und dies nicht mehr umgeschrieben werden muss.

```
    public Customer caluclateLoyalityPoints(Integer customerId, Double amount) throws Exception {

           logger.info("caluculateLoyalityPoints() for a customer with Id:  " + customerId + " and last amount " + amount);

           //find the customer
        Customer customer = customerRepository.findById(customerId).orElse(null);

        // Calculate
        Double calcAmout = 100 / amount;
        Integer newLoyalityPoints = (int) Math.round(calcAmout);

        logger.info("LoyalityPoints from the last amount are calculated: " + newLoyalityPoints);

        // make a new Customer temporary for the calculated Points
        Customer calcCustomer = new Customer(
                    customer.getCustomerId(),
                    newLoyalityPoints
                    );

        return calcCustomer;
    }
```

Sobald die Berechnung durchgeführt wurde, können die Punkte dann auch persistent geschrieben werden:

```
public Customer addLoyalityPoints(Customer calcCustomer) throws Exception {

      //find the customer
    Customer customer = customerRepository.findById(calcCustomer.getCustomerId()).orElse(null);

      // Add
    customer.setNmbr_of_loyalty_points(customer.getNmbr_of_loyalty_points() + calcCustomer.getNmbr_of_loyalty_points());
    logger.info("New LoyalityPoints are: " + customer.getNmbr_of_loyalty_points());

    return customer;
}
```

4. Payment

1. Payment & processPayment

Ist die Bestellung abgeschlossen, soll die bestellte Ware selbstredend auch bezahlt werden. Dazu hört der Service eaiPayment beim Message-Broker auf das Stichwort «RequestPayment». Sobald eine Message in der Queue ist, welche dieses Stichwort im Header beinhaltet, nimmt sich der MessageEventListener (Class) des Payment die Message und führt das Payment des Kunden über den paymentService aus. Danach wird der Status der orderMessage angepasst sowie die um die neu erstellte Transaction (id) erweitert und an den nächsten Service über den Message-Broker mit dem Stichwort «FetchGoods» gesendet.

```
@StreamListener(target = Sink.INPUT, condition = "headers['type']=='RequestPayment'")
@Transactional
public void payment(@Payload EventMessage<OrderMessage> eventMessage) throws Exception {
    OrderMessage orderMessage = eventMessage.getPayload();
    logger.info("Payload received: " + orderMessage.toString());
    Transaction transaction = paymentService.processPayment(Integer.parseInt(orderMessage.getCustomerId()),
Integer.parseInt(orderMessage.getOrderId()), orderMessage.getAmount());
    orderMessage.setTransactionId(String.valueOf(transaction.getTransactionId()));
    orderMessage.setStatus("PaymentReceived");
    send(new EventMessage<>("FetchGoods", orderMessage));
}
```

```
public Transaction processPayment(Integer customerId, Integer order_id, Double total_order_price) throws Exception {

    Customer customer = customerService.retrieveCustomerById(customerId); // Get Customer
    Order order = orderRepository.findOrderByOrderId(order_id); // Get Order

    Transaction transaction = new Transaction(customer, order_id, total_order_price); //Create new Transaction
    calculateCustomerPoints(customerId, total_order_price); // Calculating is this a new VIP-Class
    Discount discount = calculateDiscount(customerId, total_order_price, order.getShampoos().size()); // Create new Discount

    transaction.setTotal_order_price(total_order_price - (total_order_price * discount.getFactor())); // Price minus the discount

    if (total_order_price > 0) {
        try {
            transaction.setTransactionId(chargeCreditCard(customer, total_order_price, transaction).getTransactionId());
            transaction.setCanceled(false);
        } catch (Exception e) {
            transaction.setCanceled(true);
            transactionRepository.save(transaction);
            throw new Exception("Credit card transaction failed due to " + e.getMessage() + ".");
        }
    }
    transactionRepository.save(transaction);
    return transaction;
}
```

2. caluclateCustomerPoints & calculateDiscount

Abhängig von der Anzahl loyaltyPoints eines Kunden ist es möglich, dass die Bezahlung günstiger ausfällt. Zuerst wird dem Kunden die entsprechende Stufe VIP1-4 zugewiesen, damit der Rabatt zugeteilt werden kann. Danach wurde bevor der Betrag belastet wird eine Berechnungs-Methode entworfen wo die entsprechende Kalkulation basierend auf der VIP-Stufe vornimmt.

```
private void calculateCustomerPoints(Integer customerId, Double total_order_price) {
    Customer customer = customerService.retrieveCustomerById(customerId);
    Double pointsTemp = customer.getCustomerPoints() + total_order_price;

    if (pointsTemp >= 10000) {
        customer.setCustomerType("VIP1");
    }
    if (pointsTemp >= 20000) {
        customer.setCustomerType("VIP2");
    }
    if (pointsTemp >= 30000) {
        customer.setCustomerType("VIP3");
    }
    if (pointsTemp >= 40000) {
        customer.setCustomerType("VIP4");
    }
    customer.setCustomerPoints(pointsTemp.intValue());

}
```

```
public Discount calculateDiscount(Integer customerId, Double total_order_price, Integer numberOfItems) {
    Customer customer = customerService.retrieveCustomerById(customerId);
    if (total_order_price >= 1000 && customer.getCustomerType().equals("VIP1")) {
        if (numberOfItems >= 10) {
            return new Discount(0.01);
        } else {
            return new Discount(0.005);
        }
    }
    if (total_order_price >= 1000 && customer.getCustomerType().equals("VIP2")) {
        if (numberOfItems >= 10) {
            return new Discount(0.02);
        } else {
            return new Discount(0.01);
        }
    }
    if (total_order_price >= 1000 && customer.getCustomerType().equals("VIP3")) {
        if (numberOfItems >= 10) {
            return new Discount(0.03);
        } else {
            return new Discount(0.015);
        }
    }
    if (total_order_price >= 1000 && customer.getCustomerType().equals("VIP4")) {
        if (numberOfItems >= 10) {
            return new Discount(0.04);
        } else {
            return new Discount(0.02);
        }
    }

    if (total_order_price >= 500 && customer.getCustomerType().equals("NORMAL")) {
        return new Discount(0.05);
    }

    if (customer.getCustomerType().equals("STUFF")) {
        return new Discount(0.20);
    }

    return new Discount(0.00);
}
```

Betreffend des Customers in diesem MicroService ist besonders zu beachten, dass hier die loyalityPoints entscheidend sind, da nur so die VIP-Klasse zugewiesen werden kann und natürlich auch nur so der effektive Rabatt berechnet werden kann. Dazu gibt der Customer-Service diese Informationen auch mit:

```
public Customer retrieveCustomerById(Integer customerId) {
    return new Customer(1, "Lukas", "Gehrig", "Lukas Gehrig", "Musterstrasse 1", "8000 Zürich", "VIP1", 11000);
}
```

5. Inventory

1. fetchGoods

Sobald der Service im Inventory beim Message-Broker seine Message über das Stichwort «FetchGoods» erhält, wird die bestellte Ware aus dem Lager abgezogen. Daraus wird direkt ein Lieferschein erstellt, welcher der Message beigefügt wird. Da nicht wirklich ein Lager besteht, sind die Daten hierbei gefaked. Normalerweise würde hier die Aktualisierung einer Datenbank erfolgen, sobald die Waren das Lager verlassen hätte. Die aktualisierte OrderMessage wird an den nächsten Service über den Message-Broker mit dem Stichwort «ShipGoods» gesendet.

```
    @StreamListener(target = Sink.INPUT,
            condition="(headers['type']?:'')=='FetchGoods'")
    @Transactional
    public void payment(@Payload EventMessage<OrderMessage> eventMessage) throws Exception {
        OrderMessage orderMessage = eventMessage.getPayload();
        logger.info("Payload received: "+orderMessage.toString());
        List<Shampoo> orderItems = objectMapper.convertValue(orderMessage.getItems(), new TypeReference<List<Shampoo>>() {});
        PackingSlip packingSlip = inventoryService.fetchGoods(Long.valueOf(orderMessage.getCustomerId()), orderMessage.getOrderId(),
orderItems);
        orderMessage.setPackingSlipId(packingSlip.getPackingSlipId().toString());
        orderMessage.setStatus("GoodsFetched");
        messageEventSender.send(new EventMessage<>("ShipGoods", orderMessage));
    }
```

```
    public PackingSlip fetchGoods(Long customerId, String orderId, List<Shampoo> shampoos) throws Exception {
        logger.info("fetchGoods() with customerId " + customerId + " and orderId " + orderId + " called and going to pick "+
shampoos.size() +" items in the inventory");
        logger.info("In the basket are this items: ");
        for (Shampoo shampoo : shampoos) {
           logger.info("Name: " + shampoo.getName(), "Brand: " + shampoo.getBrand() + " Type: " + shampoo.getType());
        }
        for(long seconds = shampoos.size(); seconds > 0; seconds--) {
            logger.info(seconds + " items remaining");
            Thread.sleep(1000);
        }
        // ...
        PackingSlip packingSlip = new PackingSlip();
        logger.info("Packing slip generated with packing slip id: " + packingSlip.getPackingSlipId());
        return packingSlip;
    }
```

6. Shipping

1. shipGoods

Sobald der Service im Shipping beim Message-Broker das Stichwort «ShipGoods» erkennt, wird der Versand der Ware ausgelöst. Der shippingService ist nicht real an einen Provider angebunden. Hier wird lediglich der Versand simuliert. In einem Realbeispiel könnte hier z.B. eine Anbindung an einem Versand-Provider per REST-Call ausgelöst werden. Am Schluss werden die loyalityPoints noch aktualisiert. Dafür wird über den Message-Broker eine Message an den eShop mit dem Stichwort «UpdateLoyalityPoints» gesendet.

```
    @StreamListener(target = Sink.INPUT,
            condition="(headers['type']?:'')=='ShipGoods'")
    @Transactional
    public void payment(@Payload EventMessage<OrderMessage> eventMessage) throws Exception {
        OrderMessage orderMessage = eventMessage.getPayload();
        logger.info("Payload received: "+ orderMessage.toString());
        Shipping shipping = shippingService.shipGoods(
                    Integer.parseInt(UUID.randomUUID().toString()),  // trackId -> new in this Service
                    Integer.parseInt(orderMessage.getOrderId()),  // orderId -> given from eaieShop
                    Integer.parseInt(orderMessage.getCustomerId()),  // customerId -> given from eaieShop
                    Integer.parseInt(orderMessage.getPackingSlipId()), //packingSlipId -> given from eaiInvetory
                    orderMessage.getParcel_service(), // ParacelService -> given from eaieShop
                    orderMessage.getShipping_address_name(), // ShippingAdress the Name from the customer -> given from eaieShop
                    orderMessage.getShipping_address_street(), // ShippingAdress the Street from the customer -> given from eaieShop
                    orderMessage.getShipping_address_location()); // ShippingAdress the PLZ and destination from the customer -> given
from eaieShop
        orderMessage.setTrackingId(shipping.getTrackingId().toString());
        orderMessage.setStatus("GoodsShipped");
        logger.info(orderMessage.toString());
        messageEventSender.send(new EventMessage<>("UpdateLoyalityPoints", orderMessage));
    }
```

```
    public Shipping shipGoods(Integer trackingId, Integer orderId, Integer customerId , Integer packingSlipId, String parcelService,
String shipping_address_name, String shipping_address_street, String shipping_address_location) throws Exception {
        logger.info("shipGoods() with orderId " + orderId + " and customer_id " + customerId + " and packingSlip_id " + packingSlipId + "
called and hand over the parcel to the delivery service " + parcelService + "to adress " + shipping_address_name + " " +
shipping_address_street + " " + shipping_address_location);
        for(long seconds = 5; seconds > 0; seconds--) {
            logger.info("Delivery service ready in " + seconds + " seconds");
            Thread.sLeep(1000);
        }
        // ...
        Shipping shipping = new Shipping(trackingId, orderId, customerId, packingSlipId, parcelService, shipping_address_name,
shipping_address_street, shipping_address_location);
        logger.info("Packet transferred to delivery service and tracking number "+ shipping.getOrderId() + " received");
        return shipping;
    }
```

7. Erweiterung des Projektes

Es wird ein Use Case für den Kauf von Shampoos in diesem Projekt abgebildet und auch durchgeführt. Für diesen Use Case wird davon ausgegangen, dass Case-spezifische Daten zur Verfügung stehen. Damit das Projekt auf weitere Use Cases erweitert werden kann, wurden in den MircoServices bereits Vorkehrungen getroffen um diese möglichst schnell auch erweitern zu können.

5. Inventory

So wird im Inventory Service nicht nur die Methode «fetchGoods» geführt, sondern auch die Methoden um neue Inventare, Shampoos oder PackingSlips zu eröffnen, diese zu lesen oder auch anzupassen:

```
public Inventory createInventory(Integer inventory_id, Integer shampoo_id, Integer amount_of_stock, String location) {
          Inventory inventory = new Inventory(inventory_id, shampoo_id, amount_of_stock, location);
          return inventoryRepository.save(inventory);
}

public Inventory readInventoryById(String inventoryId) {
          return inventoryRepository.findById(Integer.parseInt(inventoryId)).orElse(null);
}

public Inventory updateInventory(String inventory_id, Integer shampoo_id, Integer amount_of_stock, String location) {
          Inventory inventory = new Inventory(Integer.parseInt(inventory_id), shampoo_id, amount_of_stock, location);
          inventory.setInventoryId(Integer.parseInt(inventory_id));
          return inventoryRepository.save(inventory);
}

public Shampoo createShampoo(Integer shampoo_id, String name, String brand, String type, Double price) {
          Shampoo shampoo = new Shampoo(shampoo_id, name, brand, type, price);
          return shampooRespository.save(shampoo);
}

public List<Shampoo> readShampooById(String shampooId) {
          return shampooRespository.findShampoosByShampooId(Integer.parseInt(shampooId));
}

public Shampoo updateShampoo(String shampoo_id, String name, String brand, String type, Double price) {
          Shampoo shampoo = new Shampoo(Integer.parseInt(shampoo_id), name , brand, type, price);
          shampoo.setShampooId(Integer.parseInt(shampoo_id));
          return shampooRespository.save(shampoo);
}


public PackingSlip createPackingSlip(Integer packing_slip_id) {
          PackingSlip packingSlip = new PackingSlip(packing_slip_id);
          return packingSlipRepository.save(packingSlip);
}

public PackingSlip readPackingSlipById(String packingSlipId) {
          return packingSlipRepository.findPackingSlipByPackingSlipId(Integer.parseInt(packingSlipId));
}

public PackingSlip updatePackingSlip(String packing_slip_id) {
          PackingSlip packingSlip = new PackingSlip(Integer.parseInt(packing_slip_id));
          packingSlip.setPackingSlipId(Integer.parseInt(packing_slip_id));
          return packingSlipRepository.save(packingSlip);
}
```

6. Shipping

Auch im Shipping Service sind Methoden vorbereitet, um mögliche Versendungen Hand zu haben:

```
	public Shipping createShipping(Integer tracking_id, Integer orderId, Integer customer_id, Integer packingSlip_id, String
shipping_address_name, String shipping_address_street, String shipping_address_location, String parcel_service) {
		Shipping shipping = new Shipping(tracking_id, orderId, customer_id, packingSlip_id, shipping_address_name,
shipping_address_street, shipping_address_location, parcel_service);
		return shippingRepository.save(shipping);
	}

	public Shipping readShippingById(String orderId) {
		return shippingRepository.findById(Integer.parseInt(orderId)).orElse(null);
	}

	public Shipping updateShipping(Integer tracking_id, Integer orderId, Integer customer_id, Integer packingSlip_id, String
shipping_address_name, String shipping_address_street, String shipping_address_location, String parcel_service) {
		Shipping shipping = new Shipping(tracking_id, orderId, customer_id, packingSlip_id, shipping_address_name,
shipping_address_street, shipping_address_location, parcel_service);
		shipping.setOrderId(orderId);
		return shippingRepository.save(shipping);
	}
```