Event-driven & Process-oriented Architectures – Assignment 1

Karim Ibrahim, Jonas Länzlinger, Kai Schultz
21. April 2024

Table of Contents

Project Domain **3**

Architecture **3**

Concepts **4**

Kafka (Lecture 1) 4

Kafka with Spring (Lecture 2) 5

Event-carried state transfer pattern (Lecture 2) 5

BPNM with Spring and Camunda (Lecture 3) 5

Events, Commands and additional Process Elements (Lecture 4) 6

Orchestration vs Choreography (Lecture 4) 7

Camunda 8 vs Camunda 7 (Lecture 5) 8

At-least once (Lecture 5) 9

Stateful Resilience Patterns (Lecture 6) 10

Work distribution **12**

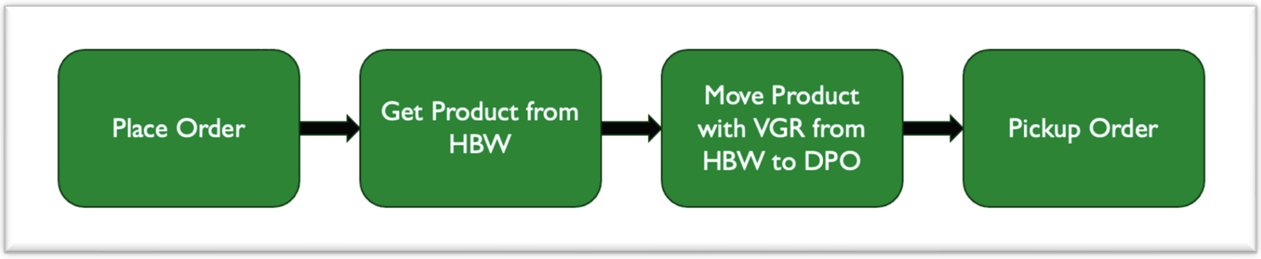
Reflections & Lessons learned **12**

Project Domain

For our project implementation we have decided to work with the “Fischertechnik Industry 9.0 smart factory”. This factory simulation contains multiple stations that can interact with each other. To keep the scope of the project small, we decided to model a straightforward user story, which utilizes the following three stations:

- HBW – High-bay Warehouse
- VGR – Vacuum Gripper
- DPO – Delivery & Pickup

The modelled workflow follows this coarse-grained base structure:



Architecture

The following technologies were used for the project implementation:

- Java Spring Boot: For creating the individual microservices
- Camunda 8: Cloud version of Camunda as workflow-management system, modelled in “Business Process Model and Notation” (BPMN)
- Kafka: For the distributed event streaming platform
- MQTT: Producing and Consuming messages to/from the (simulated) factory
- Docker: Easy local deployment of the system (Camunda engine excluded)
- Maven: Dependency management
- (jQuery: Some minor parts of the front-end)

Project Repository

https://github.com/jonaslanzlinger/EDPO-Group1-Project

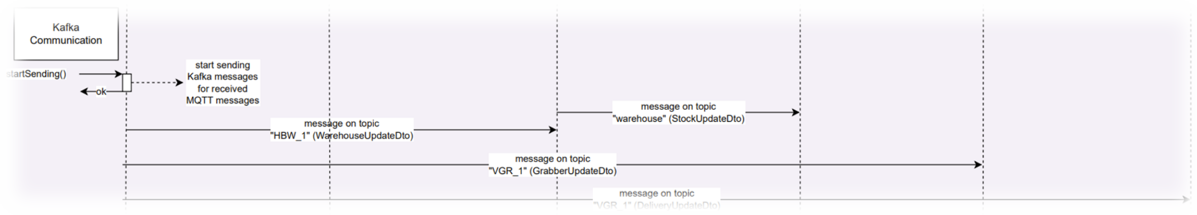
Concepts

Kafka (Lecture 1)

In the first instance, we have decided to use Kafka as the messaging technology in our event-based system architecture. For the exercise "Exercise 1: Kafka – Getting Started", we have investigated the following concepts:

- Impact of Load and Batch size on Latency (GitHub – Experiment 1)
- The risk of data loss due to dropped messages (GitHub – Experiment 2)
- Zookeeper outage (GitHub – Experiment 3)
- Risk of data loss due to consumer lag (GitHub – Experiment 4)
- Risk of data loss due to offset misconfiguration (GitHub – Experiment 5)
- Multiple brokers (GitHub – Experiment 6)
- Consumers and consumer groups (GitHub – Experiment 7)
- Kafka behavior with multiple topics and partitions (GitHub – Experiment 8)

In regards of our software project, we are using Kafka in multiple ways. First, we use Kafka in the FactoryListener service. In this service we consume the MQTT message stream coming from the FactorySimulator service, create a new Kafka topic for each individual factory station, convert the MQTT messages into Kafka messages, and dispatch them to the respective factory station topic. The factory station microservices will then consume their individual topic. By doing this, each factory station only receives messages that are relevant to them, reducing unnecessary overhead. If we decide in the future to consume multiple topics in one station service, this would be easily possible. Note that because in each consumer group there can only be one leader, we have defined each factory station service as its own consumer group.


Kafka Communication
startSending()
ok
start sending Kafka messages for received MQTT messages
message on topic "warehouse" (StockUpdateDto)
message on topic "HBW_1" (WarehouseUpdateDto)
message on topic "VGR_1" (GrabberUpdateDto)
message on topic "VGR_1" (DeliveryUpdateDto)

Secondly, we use Kafka for communication between individual factory station services. For example the Warehouse service emits Kafka events on the topic "warehouse" about its current stock status. The Order service is listening on that topic. Another example is the Monitoring service, that is listening on Kafka messages on the topic "monitoring" to create a dashboard of the current status of all the ongoing orders in the system. All factory station services are publishing Kafka events on the topic "monitoring" in each Camunda service task implementation to inform about the current status of the orders.

We think that investigating more on processing latency issues, impact of the total load, and risk of data loss due to consumer lag will be crucial for the second part of the course, when we are focusing on stream processing where we will be confronted with an increased amount of data.

Kafka with Spring (Lecture 2)

For experimentation of using Kafka together with Spring, we have extended the Flowing Retail example project to fulfill the requirements for “Exercise 2: Kafka with Spring”. To read up on our results from the exercise, visit Exercise 2 where we have documented the topics email notification, event-carried state transfer, and error scenarios.

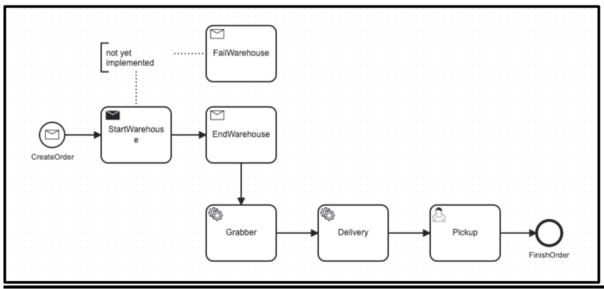
Regarding our software project implementation, we have decided to utilize event notification and event-carried state transfer in our system (see ADR-6), defined specific service granularities / bounded contexts (see ADR-4), a hybrid communication style (see ADR-7), and various error and change scenarios (see sequence diagram or camunda workflow).

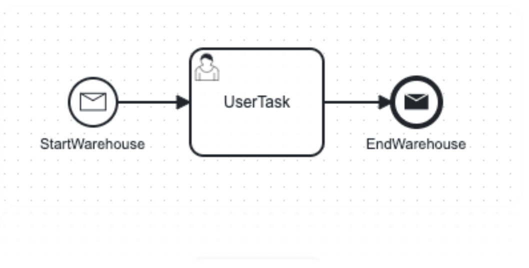
Event-carried state transfer pattern (Lecture 2)

In our software project, we have implemented the Warehouse service and the Order service such that the Order service is aware of the current stock status of the Warehouse. Therefore, in the Warehouse service we have created the StockUpdateProducer to emit Kafka messages on the topic “warehouse”. The Order service is listening (MessageConsumer) on messages on the “warehouse” topic, and subsequently updating its own internal representation of the current stock in the warehouse.

BPNM with Spring and Camunda (Lecture 3)

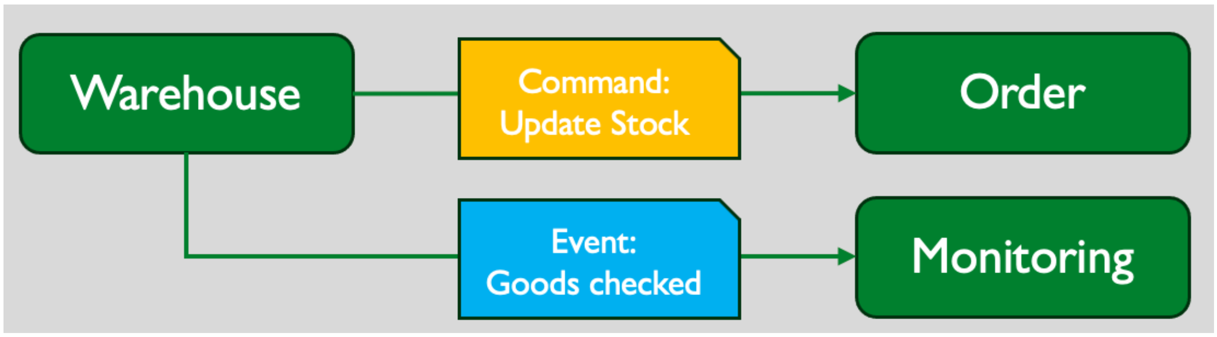
We decided to model our core processes in Camunda. The process from ordering a good, checking the warehouse, grabbing a good from it and moving it trough the factory is primarily sequential, but involves multiple services such as the order-service, the warehouse-service, the grabber-service, and the delivery-service. Modelling this processes in Camunda helped us to give us a visual overview of all the involved steps and helped us to gain a shared understanding. In the beginning we opted for a dedicated Workflow-engine per service by using Camunda 7. Later we decided to go for a centralized approach with Camunda 8 which will be explained later on. The images below show our initial process modelling.



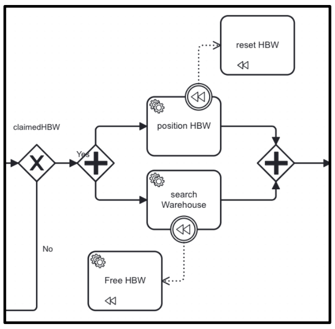


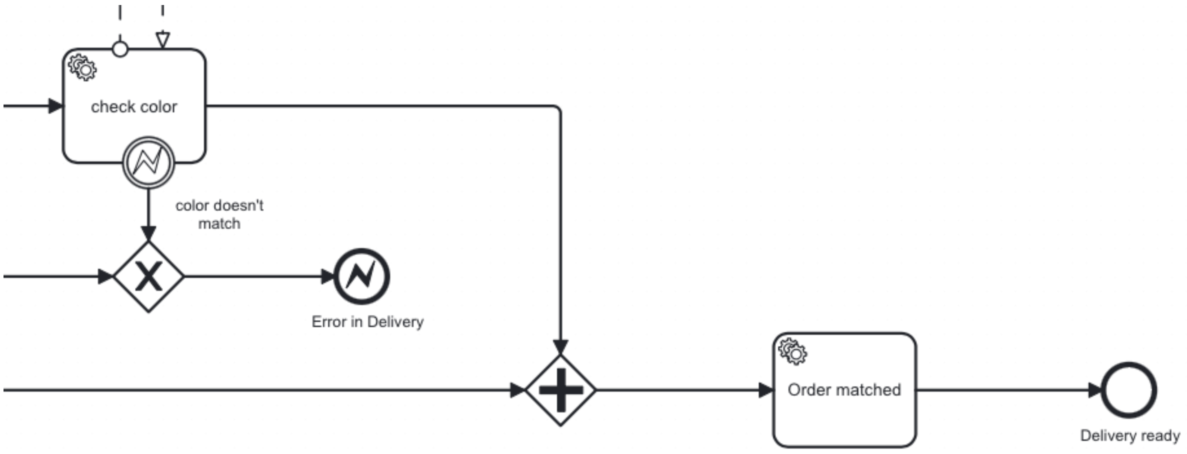
Events, Commands and additional Process Elements (Lecture 4)

Our implementation is mostly based on events that indicate when something happened during the whole process from order to delivery. We use the events to track what methods have been invoked by which service. Commands on the other side are used where it makes sense to have them. For example when a good from the warehouse is removed by the warehouse-service, an event is published that is then consumed by order-service to update the current stock. Another example is the monitoring-service, that gets triggered by Camundas "send task" feature to prepare the service for monitoring. In order to further fine tune process behavior we added process elements such as timers, subprocesses and messages to our BPNM Diagram. For example when starting the monitoring-service, we define a timeout of 20s for the task to complete. Furthermore, we also added retries in cases the monitoring-service, is unresponsive at application start. The image below shows the usage of Commands/Events as an example.



The following screenshots show parts of the bpnm process where we have added additional process elements:





Orchestration vs Choreography (Lecture 4)

Considering the topic of Orchestration vs Choregraphy we have parts of the application like the core BPNM processes that are orchestrated trough Camunda, where on the other side we have choreographed services using Kafka as the Message Broker. For example the monitoring-service is part of an orchestration, as its get triggered by Camunda, but is also part of a choreography where it listens on the topic of monitoring where the other services publish messages to. In the future we planned to further analyze where we can introduce more choreography, but we only want to introduce it if its meaningful and feasible.

Camunda 8 vs Camunda 7 (Lecture 5)

One of the main topics of week 5 was the introduction of the newer Camunda model 8. With this change the business process modelling software is now available as a standalone SaaS solution instead of being integrated in the software as a dependency. This means that we do not have to serve our own business process engine and can rely on cloud technologies and the infrastructure provided by Camunda. Additionally, with Camunda 8 comes the Zeebe client. The Zeebe client is the bridge between our Java code and the Camunda cloud. Instead of writing our own code that correlates messages to workflows, it is all abstracted away behind the Zeebe API. This greatly reduces the amount of boilerplate code and reduces the probability for mistakes since the uniform interface is already provided. There are some drawbacks as well, since Camunda 8 is much more novel than Camunda 7, it does not have the same maturity yet. There are still some processes that are not fully implemented yet in the newer version, and the developer ecosystem still needs to grow until it can compare to the Camunda 7 ecosystem.

We decided to move on with Camunda 8, since even with these trade-offs, the developer experience is much more pleasant. Moving the existing bpmn model was not hard, since we could just import it. The cloud environment additionally allowed us to work on the model at the same time. With this change we were also able to shrink our pom.xml file, since for the business modelling, we only needed this dependency (see any pom.xml file):

```
<dependency>
  <groupId>io.camunda</groupId>
  <artifactId>spring-zeebe-starter</artifactId>
  <version>8.4.0</version>
</dependency>
```

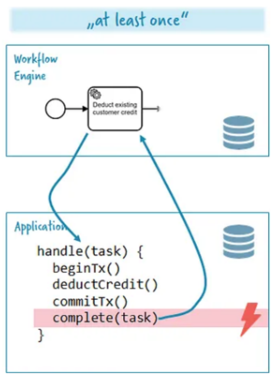
For the application properties we only needed to define the cloud adresses, which are pre-generated by Camunda when you want to register a new client (see any application.yml file).

```
zeebe.client:
  cloud:
    region: syd-1
    clusterId: “******************”
    clientId: "**********************”
    clientSecret: “**********************”
```

We decided to write a wrapper class around the most common camunda messages to increase code reuse, see for example CamundaService.java. This class implements the ZeebeClient, that gives us all the interfaces that we need to communicate with the Business Process Engine. The workers that actually execute the tasks are annotated with "@JobWorker", see for example WarehouseProcessingService.java. Zeebe allows us to simply annotate the method that we want to be called for a specific task with the corresponding type. Additionally, the methods can specify in the parameters typed process variables. This immensely decreases the amount of parsing / deserialization code that we need to write, since it has already been done for us.

At-least once (Lecture 5)

When implementing the job workers once decision that needs to be made is when do you want to send the task completion message back to the engine. Is it done before or after the business logic. Most of our executed business logic is idempotent, so in most cases you will see the completion message being called last in the method.

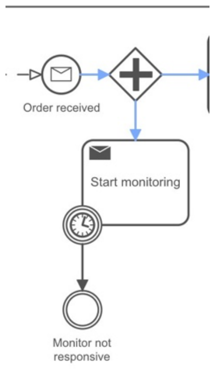


```
handle(task) {
  beginTx()
  deductCredit()
  commitTx()
  complete(task)
}
```

Stateful Resilience Patterns (Lecture 6)

The topic of week 6 was centered around patterns that help make the process more resiliant to possible errors or mistakes. We have implemented a couple of different patterns at places where their utility would be the greatest.

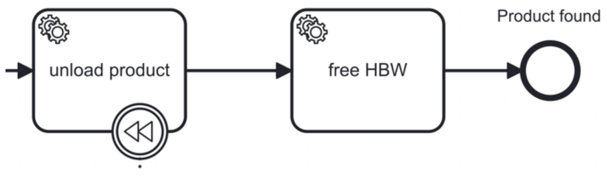
Stateful Retry – Monitoring Service



Stateful retry is implemented in the bpmn model and handled through the business process engine. The amount of retries and the intervals between them are also specified through the bpmn model itself. With Camunda 8, the standard implementation of a normal service task is always executed with stateful retries of 3 tries. To utilize this pattern and react only when there are no more tries left, we implemented simple logic in the model, that will just display to us that the monitoring service is not responsive.

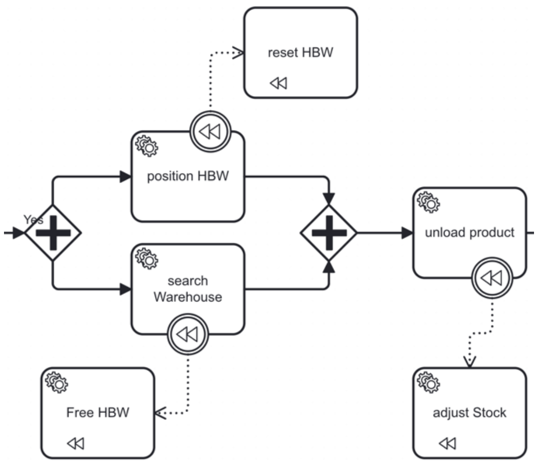
Outbox Pattern – Warehouse Service

The outbox pattern is utilized in the Warehouse Service. Since here we are working with physical entities, it is important that only one process tries to modify the factory at a time. To ensure that the factory is only freed once the business logic has succeeded, we can utilize the outbox pattern. We seperate the sending of the free HBW event from the unload Product business logic. Now if the free HBW task is called, we will know that the unload product task must have succeeded. This greatly strengthens the resilience of the application, since it guarantees, that the different processes do not infere with each other.



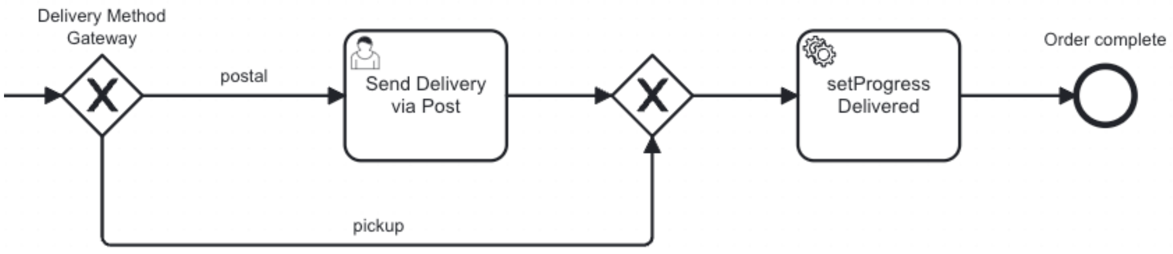
Saga Pattern – Warehouse Service

The Saga Pattern is a useful pattern, when there are complex workflows, where if an error occurs, we cannot simply stop the workflow or adjust the future path. Instead, we must undo the tasks that have already been executed up to that point.



This is exactly the Saga pattern. We implemented this pattern in the Warehouse Service as well. As you can see to the right, there are three tasks” search Warehouse”,” position HBW” and” unload product” that all need to be undone, if there is an error. In the above image, you see the subprocess, that is started if an error occurs anywhere in the warehouse process. This subprocess then goes through all tasks that have been executed up to the error and executes the corresponding undo tasks that relate to the dotted lines. Currently in Camunda 8 this is not fully implemented yet. Which is why you see that we instead execute the task directly in the subprocess. It emulates the wanted functionality well enough. Also, once it is implemented in Camunda 8, we can simply remove this from the subprocess, and it will still work fine.

Human Intervention – Delivery service



We utilized User Tasks at multiple points in our process. For example, if the customer has selected “postal” as the preferred delivery method, a factory worker needs to prepare and send the delivery via post. This is implemented as a User Task, as the intervention of a human is needed because the factory can not take care of that problem itself.

Work distribution

The work has been equally distributed among all team members. Most of the things have been done in pair-programming.

Reflections & Lessons learned

Refer to Slide 19 of the intermediate_presentation.pdf