Event-driven & Process-oriented Architectures – Assignment 2

Karim Ibrahim, Jonas Länzlinger, Kai Schultz

2. June 2024

Table of Contents

Introduction 3

Change Log 3

Project Domain 4

Architecture 5

Concepts 6

- Kafka (Lecture 1) 6
- Kafka with Spring (Lecture 2) 8
- Event-carried state transfer pattern (Lecture 2) 8
- BPNM with Spring and Camunda (Lecture 3) 8
- Events, Commands, and additional Process Elements (Lecture 4) 9
- Orchestration vs Choreography (Lecture 4) 11
- Camunda 8 vs Camunda 7 (Lecture 5) 11
- At-least once (Lecture 5) 12
- Stateful Resilience Patterns (Lecture 6) 13
  - Stateful Retry – monitoring service 13
  - Outbox Pattern – warehouse service 13
  - Saga Pattern – warehouse service 14
  - Human Intervention – delivery service 16

Stream Processing 17

- Streamprocessor Topology 18
- Monitoring Topology 19
- Serializing / Deserializing (Lecture 8) 20
- Timestamp Extractors (Lecture 10) 21
- Stateless Operations / Single-Event Processing (Lecture 8) 22
  - Event Filter (Lecture 8) 22

Event Router (Lecture 8) 23
Content Filter (Lecture 8) 23
Event Translation (Lecture 8) 24
Event Stream Merger (Lecture 8) 24
Stateful Operations / Local State (Lecture 9) 24
Previous Event Filter (Lecture 9) 25
Color Stats (Lecture 9) 26
Time Difference Aggregation (Lectures 9 & 10) 28
Joining of Streams / Session Window (Lectures 9 & 10) 30

Architectural Decisions 32

Reflections & Lessons learned 32
Messaging 32
Kafka 33
Specific learnings more in depth 33
Behavior of different stores in Kafka 33
Plotting of Station Data 35

Work Distribution 36

Introduction

For the report of the second assignment, we decided to extend the report from the the first assignment and extend the existing sections where needed. Specifically, we have answered to the feedback given by the teachers for the first assignment. The changes / modifications are marked in "red" (as suggested by the tutors), or you can have a look at the *Change Log* further down. The report for the second assignment – exercises 7, 8, 9 – can also be found in this document.

In the following we provide you with the necessary link to our GitHub repositories:

- Release Assignment 1: https://github.com/jonaslanzlinger/EDPO-Group1-Project/releases/tag/Assignment-1
- Release Assignment 2: https://github.com/jonaslanzlinger/EDPO-Group1-Project/releases/tag/Assignment-2
- Software Project: https://github.com/jonaslanzlinger/EDPO-Group1-Project
- Exercises 1-4 Experiments: https://github.com/KaiTries/EDPO-Group1

Change Log

This change log contains all relevant modifications on the reports and the software implementation since the hand-in of assignment 1 (also according to the feedback from the tutors after the intermediate presentation):

Report

- Refactored structure of report
- Added Introduction section
- Added Change Log section
- List all microservices in our system under Architecture section
- Kafka (Lecture 1): Updated the current Kafka architecture logic; updated sequence diagram screenshot; added paragraph about Kafka issue considerations
- Events, Commands, and additional Process Elements (Lecture 4): Clearer reasoning about usage of events and commands; additional screenshot of sequence diagram to illustrate our use of events and commands
- Camunda 8 vs Camunda 7 (Lecture 5): Specifically name limitations of Camunda 8
- Stateful Retry – monitoring service: Documented the synchronization error introduced by the missing "AND"

- Outbox Pattern – warehouse service: Documented the changes in the implementation

- Human Intervention – delivery service: Specifically reference the user tasks from the Camunda BPMN and explain use case
- Reflections & Lessons learned: Added learned lessons from assignment 1 in the report
- Added all documentations for assignment 2

Miscellaneous

- Extended sequence diagram according to feedback: Added actor names to each of the entities; Additional program logic for assignment 2
- Updated some of the ADR's according to the current implementation
- Updated README file
- Updated "plotter" jupyter notebook
- Uploaded all relevant documents for assignment 2

Software Project

- Beginning with part two, we translated our implementation such that it works with the real factory (see demo instructions)
- Implementation of new streamprocessor service
- Implementation of stream processing capabilities for monitoring service

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:

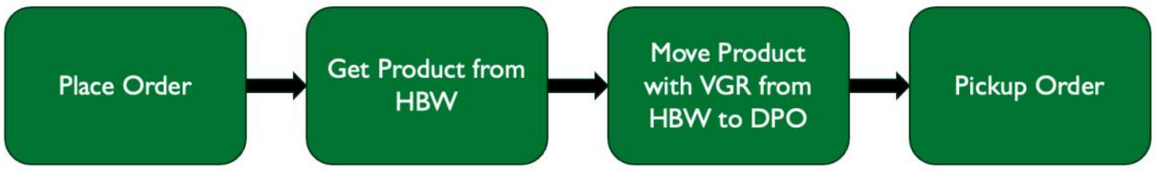


Figure 1: Workflow

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)
- (Ajax: Some minor parts of the front-end)

The following Microservices are deployed:

- order service
- warehouse service
- grabber service
- delivery service
- factorysimulator service
- factorylistener service
- monitoring service
- streamprocessor service

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 or the real factory, and create a new Kafka topic for all factory events ("factory-all"). The streamprocessor service is consuming this Kafka topic and processes the data stream accordingly. In this service, the branching of the topic "factory-all" into the respective factory station topics ("VGR_1-processed", "HBW_1-processed") takes place. 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. Even more, the streamprocessor service only propagates events, that contain changes in their payload from the previous event. This, furthermore, reduces 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. In figure 2, which is an excerpt from our modeled sequence diagram, we show how the Kafka communication is implemented in our service environment.

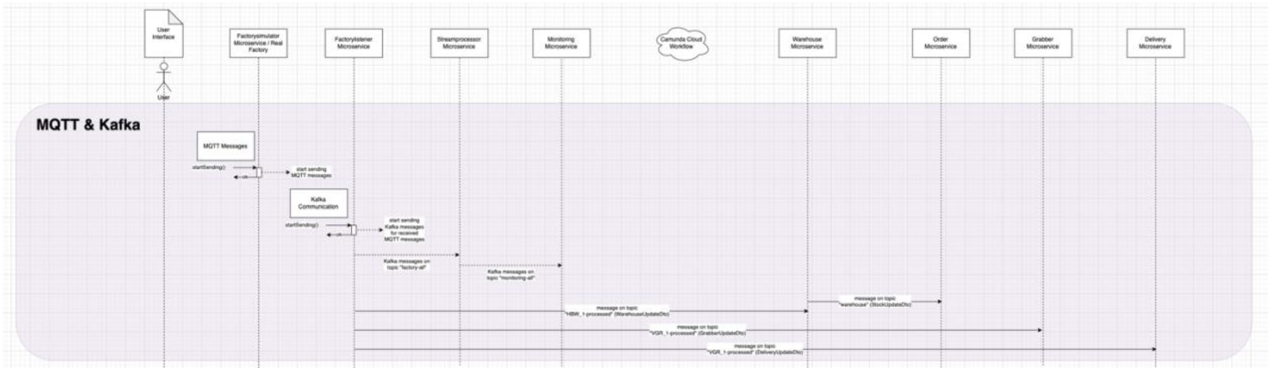

User Interface
User
Factorysimulator Microservice / Real Factory
Factorylistener Microservice
Streamprocessor Microservice
Monitoring Microservice
Camunda Cloud Workflow
Warehouse Microservice
Order Microservice
Grabber Microservice
Delivery Microservice
MQTT & Kafka
MQTT Messages
startSending()
ok
start sending MQTT messages
Kafka Communication
startSending()
ok
start sending Kafka messages for received MQTT messages
Kafka messages on topic "factory-all"
Kafka messages on topic "monitoring-all"
message on topic "HBW_1-processed" (WarehouseUpdateDto)
message on topic "warehouse" (StockUpdateDto)
message on topic "VGR_1-processed" (GrabberUpdateDto)
message on topic "VGR_1-processed" (DeliveryUpdateDto)

Figure 2: Sequence Diagram

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.

During the implementation for the second assignment, we never ran into any of the issues described above. This can have several reasons. Both, the simulation of the factory, and the real factory, are located in the same network as the host of the dockerized microservices, which reduces latency to a minimum. Also, data loss is not a concern at all because the factory stations are continuously emitting events about their complete state (all sensors and metrics). This means that even if an event is lost, the system will be aware of the current state again by the consumption of the following event. For example, events like a broken light barrier are emitted multiple times by the factory, creating redundancy. Our initial intuition to use Kafka as it comes out of the box proved to be sufficiently performant and resilient.

Kafka with Spring (Lecture 2)

For experimentation of using Kafka together with Spring, we have extended the Flowing Retail example to fulfill the requirements. 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 thus updating its own 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 these 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. Figure 3 below shows the initial process modelling.

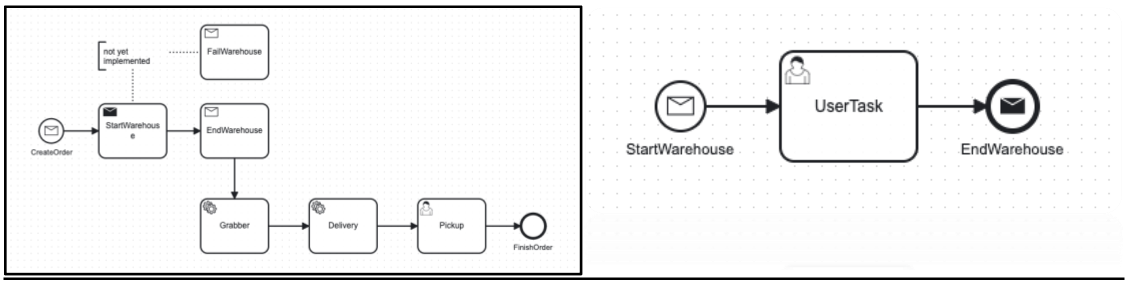


Figure 3: Initial process modelling

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

In general, we employed the distinction between events and commands semantically. Events inform about something that has happened in the system, while commands inform an intent that something should happen in the future of the system. You can infer the semantics from the name of the invoked action. An example for an event would be "Goods checked". An example for a command would be "Update Stock".

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. More specifically, commands are used in places where we have an intent for something to happen in the system. 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. Figure 4 below shows the usage of commands / events as an example.

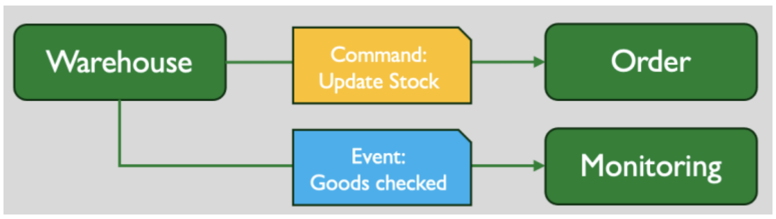


Figure 4: Command & Event Pattern

In figure 5 you can see an excerpt of our sequence diagram, how we used the event- and command semantics. In this example "freeHBW" denotes an intent; "HBWAbailable" denotes the happening of something in the past.

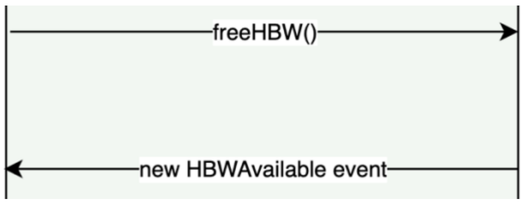


Figure 5: Excerpt of Sequence Diagram

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

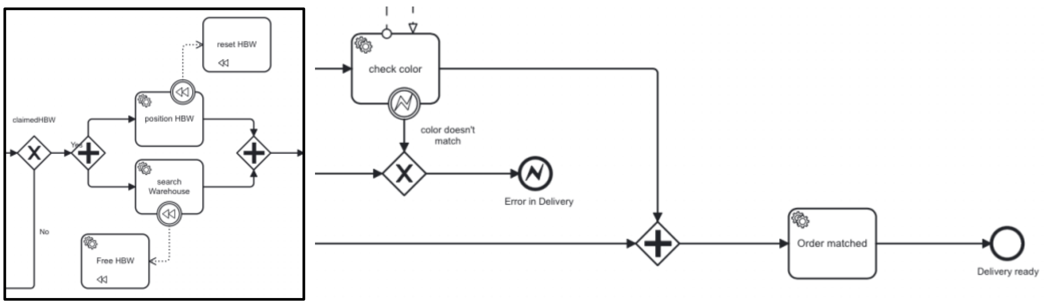


Figure 6: Additional Process Elements

Orchestration vs Choreography (Lecture 4)

Considering the topic of orchestration vs choreography, 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. Specifically, in Camunda 8 we felt that we are limited in the diversity of BPMN elements. For example, in Camunda 8, conditional and signal events are not available. Though we managed to create workarounds with the available toolbox of Camunda 8.

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>
```

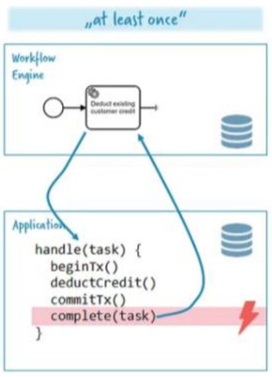
For the application properties we needed to define the cloud adresses, which are pre-generated by Camunda when registering a client (see any application.yml file).

```
zeebe.client:
  cloud:
    region: bru-2
    clusterId: "******************"
    clientId: "***********************"
    clientSecret: "***********************"
```

We decided to write a wrapper class around the most common Camunda messages to increase code reuse, see for example CamundaService. 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. 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 that we need to write; 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. Figure 7 shows the At-least-once concept.



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

Figure 7: At-least-once

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.

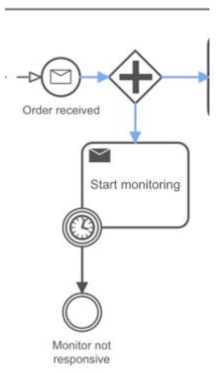


Figure 8: External Task

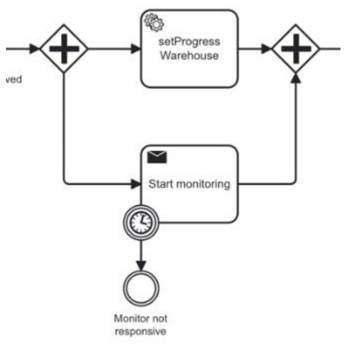


Figure 9: Merging

In the implementation before the break, we have introduced an out-of-sync error by not merging the "AND" gate after the monitoring has been started. This error has now been fixed.

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.

We removed this functionality from the business process in the final version. This is due to how the communication with the actual physical factory works. Since the factory will only return a response to a request, once it has carried out the task, we cannot easily do this with zeebe workers. Our solution is that we simply free the HBW agnostic to the actual Camunda process. We did this by implementing the logic in the Java code directly. Now the HBW gets freed once the asynchronous http response has successfully been received. While the logic of the HBW lock is now not as transparent anymore in the business process, it is much more intuitively built into the code. This was a trade-off that we considered, mainly due to time constraints and ease of implementation constraints.

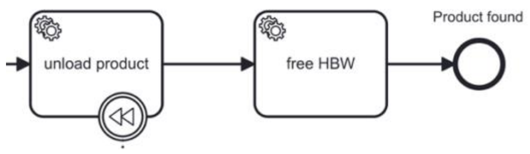


Figure 10: Rollback Mechanism

<u>Saga Pattern – warehouse service</u>

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.

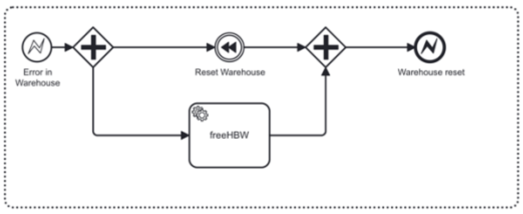


Figure 12: Rollback

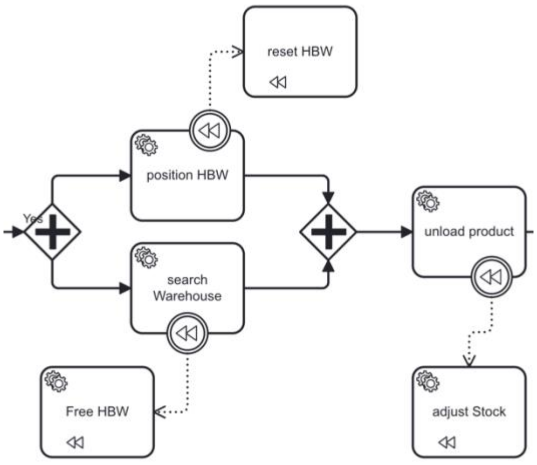


Figure 11: Parallel execution of a workflow

Human Intervention – delivery service

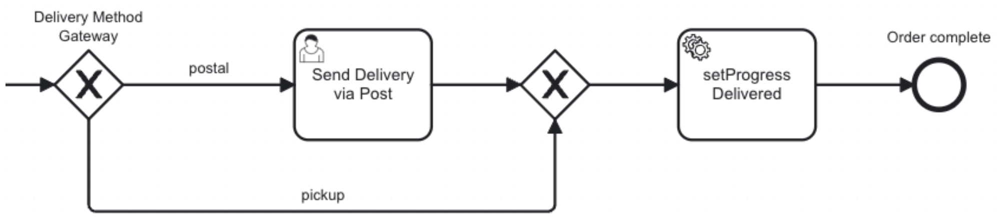


Figure 14: Human Intervention Task

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.

We have renamed the activities that relate to human intervention. The following describes all User Tasks, and what they are intended to do:

- "send postalDelivery": A factory worker should pickup the workpiece and send the order via post office to the customer. Since there is no implementation for this, the worker just needs to tick the checkbox to confirm, that he has sent the postal delivery.
- "remove Product": In the delivery station happened an error (i.e. retrieving the product color took too much time or the retrieved color doesn't match any order). The factory worker needs to remove the product from the factory and confirm it via the user task.
- "reset VGR": In the grabber station happened an error (i.e. transporting the product to the delivery station took too much time). The factory worker needs to remove the product from the factory and confirm it via the user task.
- "reset Warehouse": In the warehouse station happened an error (i.e. no goods are available for this product type). The factory worker needs to remove the product from the factory and confirm it via the user task.

Stream Processing

For the most part of assignment 2 we worked on the stream processing of our application. In the following we describe our stream processing topologies. There are two microservices that include a stream processing topology: (1) streamprocessor and (2) monitoring. In ADR-8 we have documented why we went for a centralized stream processing approach. According to the ADR, this enables us to have a centralized and dedicated service that can easily be scaled if needed. This also entails that the individual factory station services are lightweight, and don't need to process the data stream by themselves. By splitting the streamprocessor service apart from the monitoring service, we have a fitting separation of concerns: The streamprocessor service handles the splitting and the preparation of the data stream, while the monitoring service is only used to create aggregated information and statistics based on the pre-processed data stream.

Streamprocessor Topology

As we can see in the following figure, a specific topology has been carefully designed and implemented for the streamprocessor service, which is responsible for handling the "factory-all" Kafka stream. Initially, it branches the generic stream into two distinct substreams, namely HBW_1 and VGR_1. Moving forward, the next step involves proceeding with the deserialization of both substreams, and subsequently transforming them into dedicated streams, thereby enabling further processing and analysis. We will delve deeper into the details in the next paragraphs.

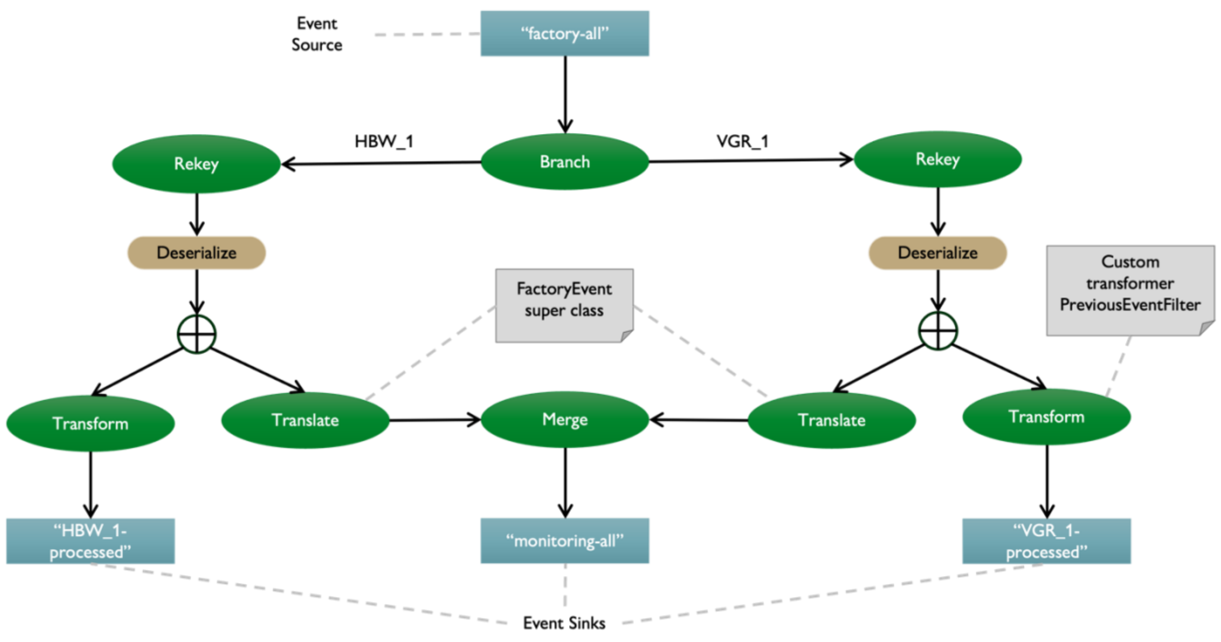


Figure 15: Topology of the Streamprocessor

Monitoring Topology

As we can observe in the following figure, a comprehensive topology has been meticulously designed and developed for the monitoring service, which is fundamentally responsible for overseeing and monitoring the operational activities of the "High Bay Warehouse" and the "Vaccum Gripper Arm" services. Notably, the monitoring service consumes the "monitoring-all" topic, which is produced by the streamprocessor service. Interestingly, when compared to the streamprocessor topology that has been described above, this topology incorporates a significantly greater amount of custom logic, which will be further explained and elaborated upon in the coming sections, providing a more detailed understanding of its underlying mechanisms and functionalities.

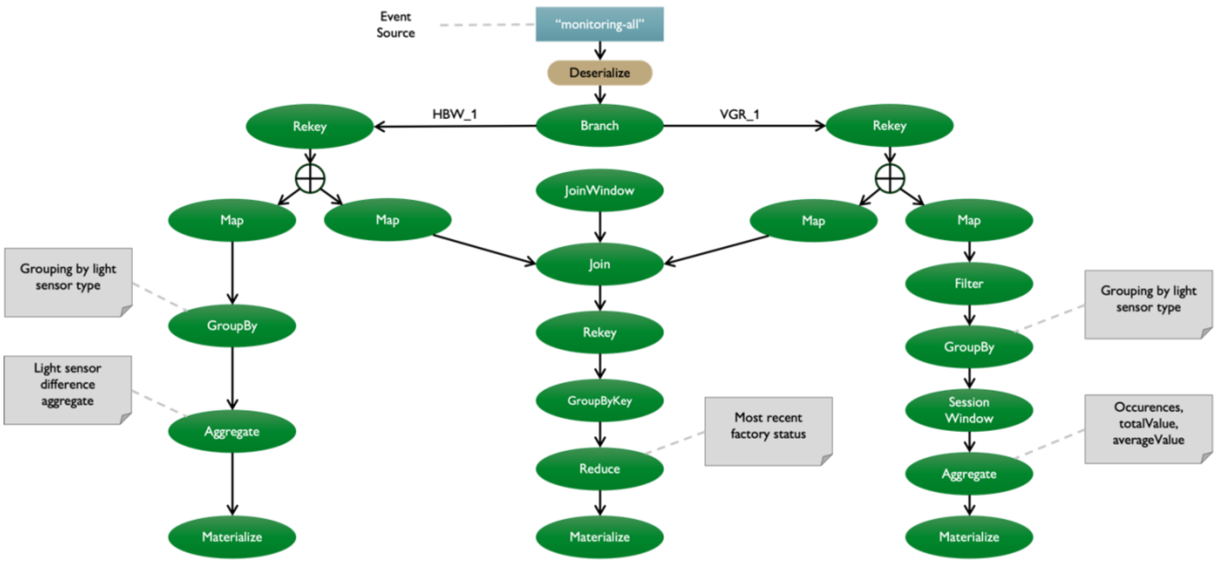
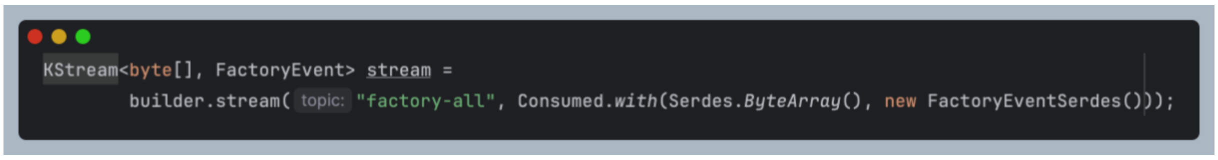


Figure 16: Topology of the Monitoring service

Serializing / Deserializing (Lecture 8)

In order to employ stream processing in our services streamprocessor and monitoring, we first need to consume the raw data stream for the Kafka topic “factory-all” and “monitoring-all” respectively. The data streams need to be deserialized to transform the data into event objects. This enables easier data handling and overall type safety.

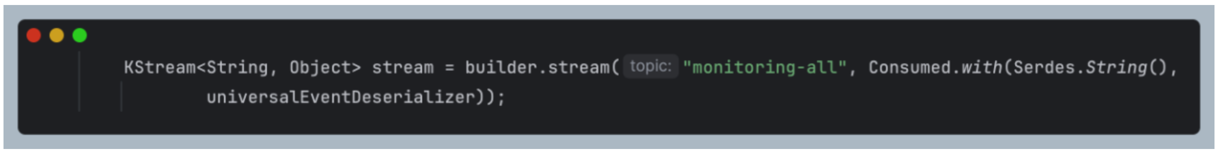
In the streamprocessor service, we consume the “factory-all” Kafka stream with the FactoryEventSerdes, which deserializes the events into FactoryEvent, where the actual data about the factory is stored in a generic “Object” object. This is needed because HBW_1 and VGR_1 events have different attributes in their “data” field. Later, after we split the event stream into “HBW_1”-, and “VGR_1"-events, we create the actual HbwEvent and VgrEvent objects, such that we maintain fully typed streamed data.



```
KStream<byte[], FactoryEvent> stream =
        builder.stream( topic: "factory-all", Consumed.with(Serdes.ByteArray(), new FactoryEventSerdes()));
```

Figure 17: KStream «factory-all»

In the monitoring service, we consume the “monitoring-all” Kafka stream a little differently. Because the incoming events are serialized as HbwEvent and VgrEvent, we can simply deserialize them as HbwEvent and VgrEvent objects again. But because Kafka Streams doesn’t provide a way to specify two different Serdes for one stream, that deserializes the events depending on their type, we have to specify a custom UniversalEventSerdes, that does exactly that. This Serdes deserializes “HBW_1” and “VGR_1” data objects correctly but returns them as “Object” types. Later, after splitting the event stream into “HBW_1”- and “VGR_1”- events again, these objects only need to be casted into HbwEvent and VgrEvent objects to maintain fully typed streamed data again.



```
KStream<String, Object> stream = builder.stream( topic: "monitoring-all", Consumed.with(Serdes.String(),
        universalEventDeserializer));
```

Figure 18: KStream «monitoring-all»

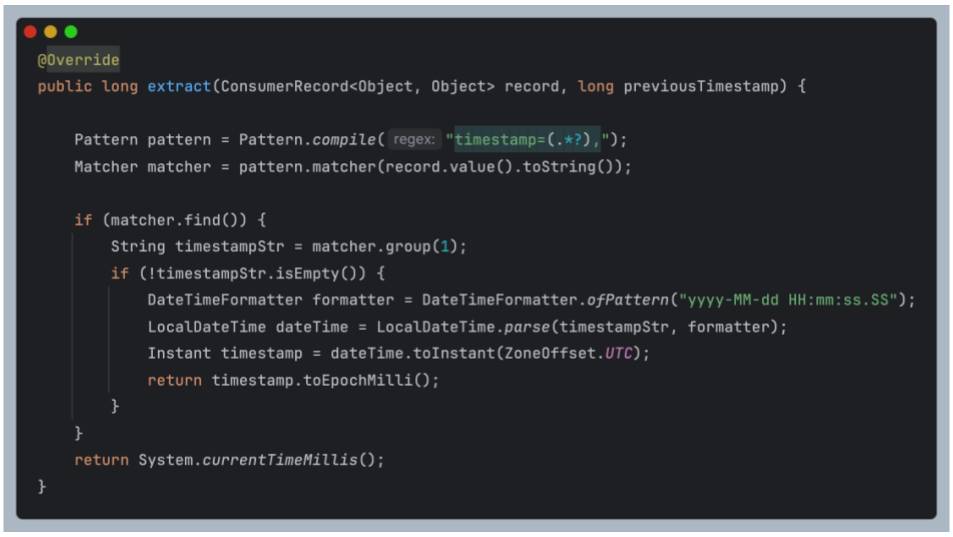
The serialization of the data in both stream processing services is implemented in a typical standard way.

Timestamp Extractors (Lecture 10)

When we talk about serializing and deserializing the streams, we directly want to add our thoughts about how we utilized customized timestamp extractors, because this is also part of the deserialization process in Kafka Streams.

From the available time semantics – “event time”, “ingestion time”, “processing time” –, we decided that the event time is the most useful in our setting. The time when the factory emitted the actual MQTT message.

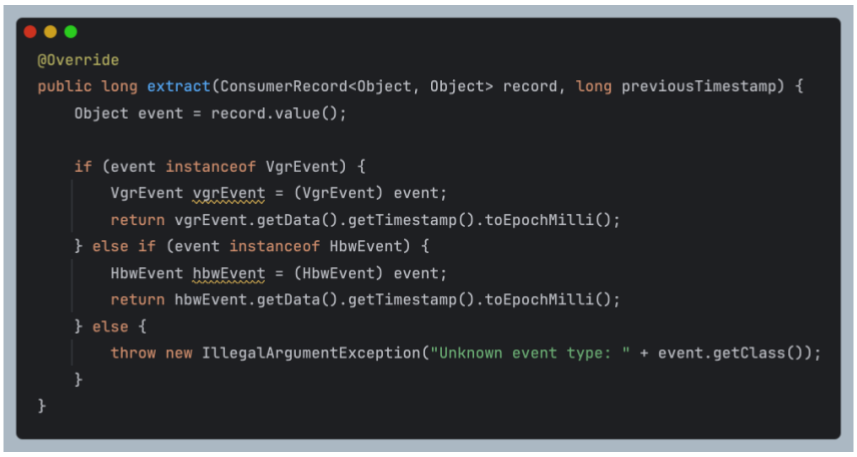
In the streamprocessor the CustomTimestampExtractor overrides the “extract()" method from the "TimestempExtractor” interface. Because we can’t access the typed event data in the streamprocessor yet, we had to extract the timestamp from the event data via regular expressions, and bring it in the correct timestamp format that we need.



```java
@Override
public long extract(ConsumerRecord<Object, Object> record, long previousTimestamp) {

    Pattern pattern = Pattern.compile("timestamp=(.*?),");
    Matcher matcher = pattern.matcher(record.value().toString());

    if (matcher.find()) {
        String timestampStr = matcher.group(1);
        if (!timestampStr.isEmpty()) {
            DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss.SS");
            LocalDateTime dateTime = LocalDateTime.parse(timestampStr, formatter);
            Instant timestamp = dateTime.toInstant(ZoneOffset.UTC);
            return timestamp.toEpochMilli();
        }
    }
    return System.currentTimeMillis();
}
```

Figure 19: CustomTimestampExtractor

In the monitoring service we did it differently. Because here we already have the typed event data, we were able to simply parse the data according to its event type, and then access the timestamp in an OOP manner.



```java
@Override
public long extract(ConsumerRecord<Object, Object> record, long previousTimestamp) {
    Object event = record.value();

    if (event instanceof VgrEvent) {
        VgrEvent vgrEvent = (VgrEvent) event;
        return vgrEvent.getData().getTimestamp().toEpochMilli();
    } else if (event instanceof HbwEvent) {
        HbwEvent hbwEvent = (HbwEvent) event;
        return hbwEvent.getData().getTimestamp().toEpochMilli();
    } else {
        throw new IllegalArgumentException("Unknown event type: " + event.getClass());
    }
}
```

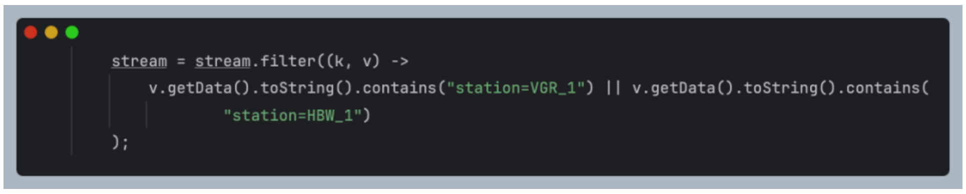
Figure 20: Extraction of Timestamp of distinct events

Stateless Operations / Single-Event Processing (Lecture 8)

In the lectures stream processing, we have been introduced to stateless processing operations. Therefore, we have implemented multiple stateless operations in our two stream processing services, that we will cover in the following sections.

Event Filter (Lecture 8)

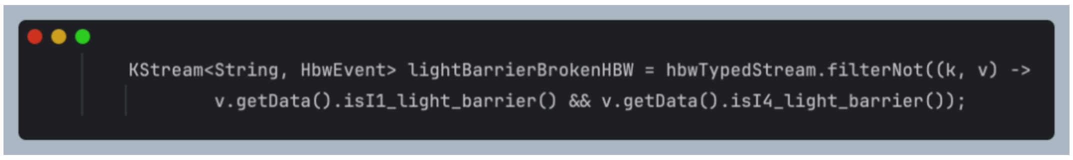
In the streamprocessor we begin with the source stream that in our case is a combined stream containing all events produced by the factory. Because our application is only interest in events for the factory stations HBW_1 and VGR_1, we reduce the amount of data that needs to be processed by using an event filter, see figure 20. The event filter reduces the source stream to events, that contain either VGR_1 or HBW_1 in their "station" attribute. This single-event processing operation makes most sense to perform right at the beginning of the stream processing.



```java
stream = stream.filter((k, v) ->
    v.getData().toString().contains("station=VGR_1") || v.getData().toString().contains(
            "station=HBW_1")
);
```

Figure 21: Filtering based on specific attributes

In the monitoring service, filtering is implemented for the use-case of only stream events with at least one broken light barrier in the HBW stream. This filtered stream will then later be used for determining the time needed until the workpiece is at a certain location in the factory. In figure 21 you can see how we used the ".filterNot()" method to exclude all events that have both light barriers unbroken (true).

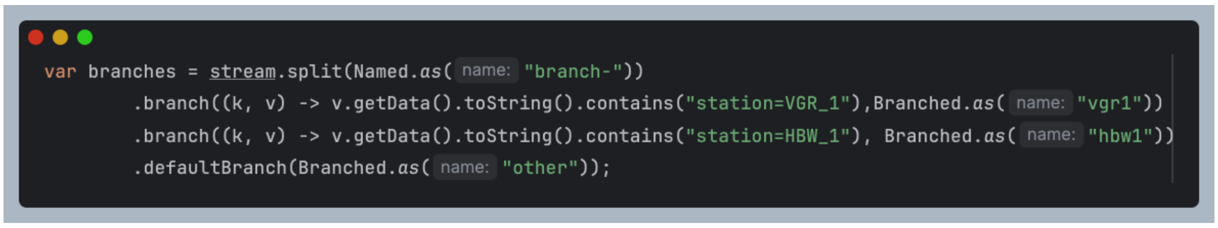


```java
KStream<String, HbwEvent> lightBarrierBrokenHBW = hbwTypedStream.filterNot((k, v) ->
        v.getData().isI1_light_barrier() && v.getData().isI4_light_barrier());
```

Figure 22: Filtering of Events where the light barrier has not been broken

Event Router (Lecture 8)

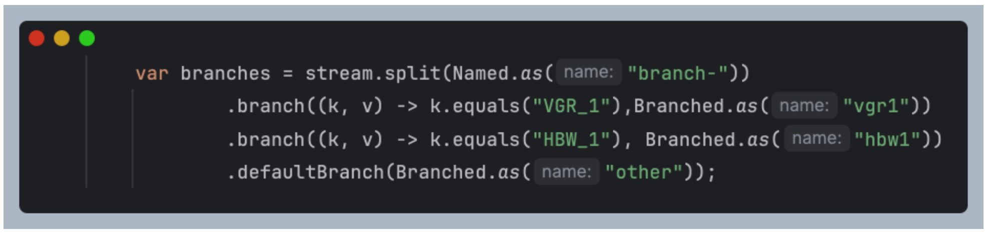
Since we want each type of event (one type of event for each factory station) to be streamed in its own data stream, we employ event routing to the filtered source stream in both the streamprocessor and the monitoring service. Therefore, the source stream (“factory-all” or “monitoring-all” respectively) gets branched into the two dedicated VGR_1 and HGW_1 streams that correspond to the “Vaccum Gripper Arm” and the “High Bay Warehouse”. In the streamprocessor we have to do the branching based on the content of the “station” attribute.



```java
var branches = stream.split(Named.as( name: "branch-"))
        .branch((k, v) -> v.getData().toString().contains("station=VGR_1"),Branched.as( name: "vgr1"))
        .branch((k, v) -> v.getData().toString().contains("station=HBW_1"), Branched.as( name: "hbw1"))
        .defaultBranch(Branched.as( name: "other"));
```

Figure 23: Branching/Routing

In the monitoring service – because this stream is already correctly typed – we can simply branch the stream based on the key of the event.



```java
var branches = stream.split(Named.as( name: "branch-"))
        .branch((k, v) -> k.equals("VGR_1"),Branched.as( name: "vgr1"))
        .branch((k, v) -> k.equals("HBW_1"), Branched.as( name: "hbw1"))
        .defaultBranch(Branched.as( name: "other"));
```

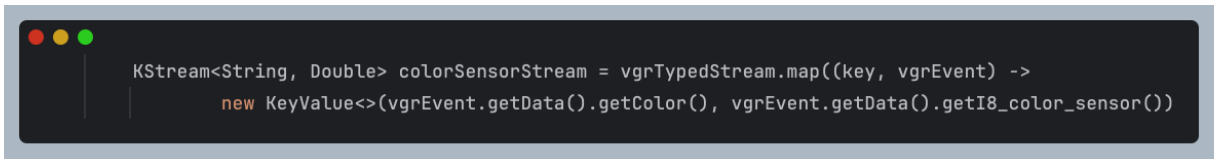
Figure 24: Branching and renaming

Content Filter (Lecture 8)

In general, we are not interested in filtering out specific attributes from our events. This is due to the fact that we want to represent the total state of the factory in the monitoring service, for which we want to have all attributes for the stations HBW and VGR. This is really easy to implement however, by using the “.mapValues()” method. Technically we use this method in the streamprocessor, but we use it to create our typed streams, where we don’t filter out attributes.

Event Translation (Lecture 8)

One instance where we use event translation is in the monitoring service. We formulated the use case that we want to calculate “totalReadings”, “totalColorValues”, and “averageColorValue” metrics for each color that a workpiece can have in the factory. To easily accomplish this, we prepare the typed stream of data by creating a new “colorSensorStream”, where we translate the original VgrEvent to events where the key is the discretized color of the workpiece, and the value is the actual color value that is coming from the color sensor in the VGR station. Note: This is technically also content filtering, but we’d rather categorize this behavior as event translation.

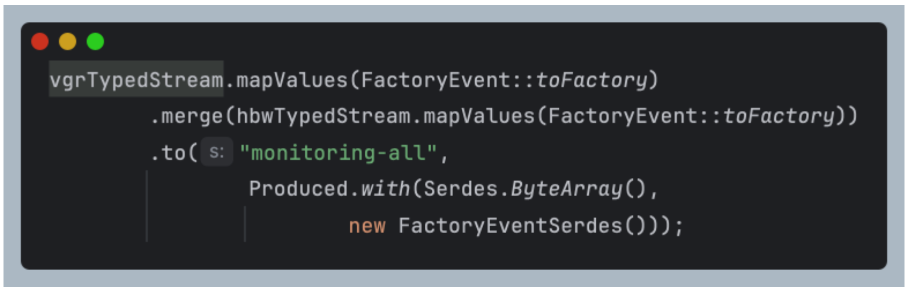


```
KStream<String, Double> colorSensorStream = vgrTypedStream.map((key, vgrEvent) ->
        new KeyValue<>(vgrEvent.getData().getColor(), vgrEvent.getData().getI8_color_sensor())
```

Figure 25: Event Translation

Event Stream Merger (Lecture 8)

In the streamprocessor service, we have split the two event types into HBR_1 and VGR_1. These two typed streams are sent to the sink topics “VGR_1-processed” and “HBW_1-processed”. But the monitoring service is interested in the unprocessed – but already typed – events. Therefore, we need to merge the branched, typed streams together again. To be able to merge those two event types, we need to wrap them into objects of the Wrapper Event class FactoryEvent. Afterwards the merged stream gets sent to the sink topic “monitoring-all” with the respective FactoryEventSerdes as you can see in figure 25 below.



```
vgrTypedStream.mapValues(FactoryEvent::toFactory)
        .merge(hbwTypedStream.mapValues(FactoryEvent::toFactory))
        .to( s: "monitoring-all",
                Produced.with(Serdes.ByteArray(),
                        new FactoryEventSerdes()));
```

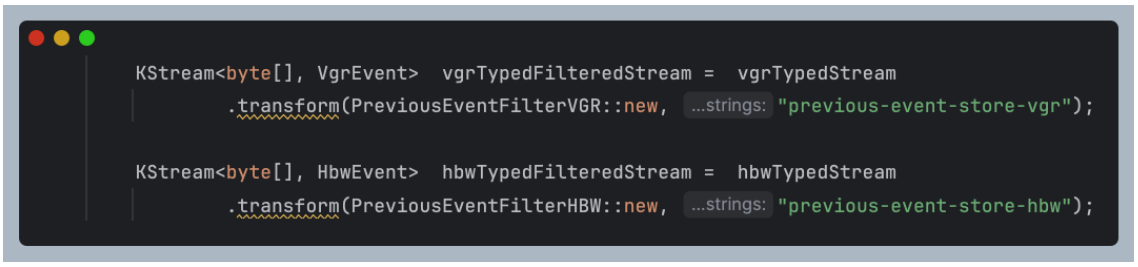
Figure 26: Event Stream merging

Stateful Operations / Local State (Lecture 9)

From lecture 9 on, we investigated the possibilities of how to utilize stateful operations with local state in our stream processing services. This enables us to process our data streams over time, creating aggregates, and employing windowed operations. To link the concepts about stateful operations from the lectures, we think it is best, to explain our key implementations that are making use of stateful operations. These four implementations: PreviousEventFilter, ColorStats, TimeDifferenceAggregation, and "Joining of Streams" are described in the following.

Previous Event Filter (Lecture 9)

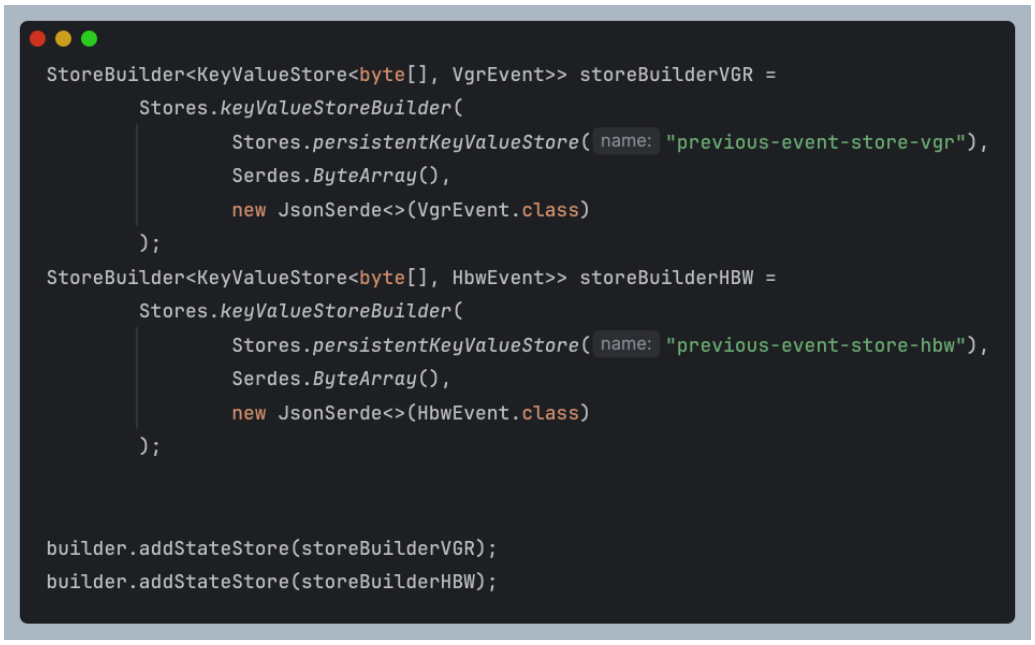
In the streamprocessor service we only want to send events to the respective factory station microservices (warehouse, grabber, delivery) that have differing event data from the previous events. By employing this, we reduce the communication needed between the streamprocessor service and the respective factory station microservices, because only the relevant events will get propagated along. Note: This is stateful event filtering.



```
KStream<byte[], VgrEvent>  vgrTypedFilteredStream =  vgrTypedStream
        .transform(PreviousEventFilterVGR::new,  ...strings: "previous-event-store-vgr");

KStream<byte[], HbwEvent>  hbwTypedFilteredStream =  hbwTypedStream
        .transform(PreviousEventFilterHBW::new,  ...strings: "previous-event-store-hbw");
```

Figure 27: Previous Event Filter

This is done using the "transform()" method, which allows transforming the input stream into an output stream. The method takes two parameters: (1) The first parameter has to be of Type "Transformer" – "PreviousEventFilterXYZ" in our case -, that is responsible for transforming an input event to an output event. (2) The second parameter is the name of the state store, which is used by the transformer to maintain any state required for processing. Those two defined state stores need to be created and added to the "StreamBuilder" first, as you can see in figure 26.



```java
StoreBuilder<KeyValueStore<byte[], VgrEvent>> storeBuilderVGR =
        Stores.keyValueStoreBuilder(
                Stores.persistentKeyValueStore( name: "previous-event-store-vgr"),
                Serdes.ByteArray(),
                new JsonSerde<>(VgrEvent.class)
        );
StoreBuilder<KeyValueStore<byte[], HbwEvent>> storeBuilderHBW =
        Stores.keyValueStoreBuilder(
                Stores.persistentKeyValueStore( name: "previous-event-store-hbw"),
                Serdes.ByteArray(),
                new JsonSerde<>(HbwEvent.class)
        );


builder.addStateStore(storeBuilderVGR);
builder.addStateStore(storeBuilderHBW);
```

Figure 28: Key Stores

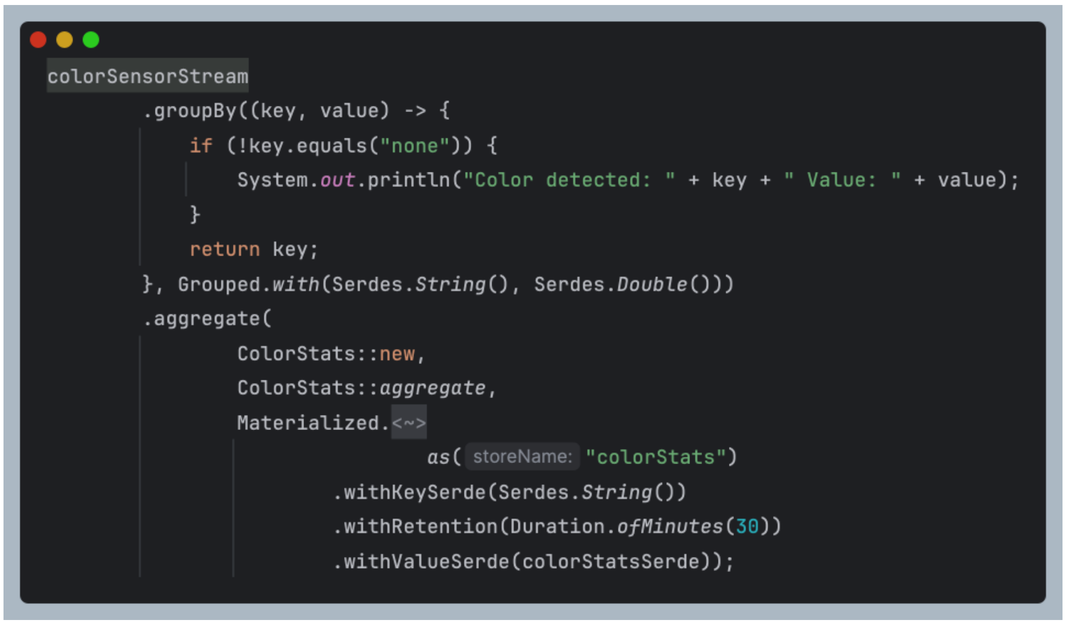
The resulting "XYZTypedFilteredStream" can then be sent to the event sink that is being consumed by the respective factory station microservices.

The PreviousEventFilterXYZ implementation has to specify a state store at object initialization, to define where the intermediate event should be persisted during the transform operation. Furthermore, the class overrides the "transform()" method. Here, we define that for VGR events it filters events that have identical discretized color readings from the previous one. For HBW events it filters events that have identical stock readings and identical light barrier readings.

Note: For this implementation we don't need to materialize the table, because we don't perform interactive queries on it. We just redirect the events to the corresponding event sinks.

Color Stats (Lecture 9)

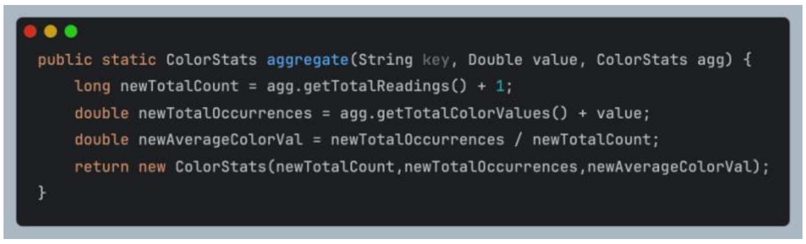
The monitoring service is predestined to track aggregated metrics of the behavior of the factory. Therefore, we formulated the use case that we want to analyze the color readings of the data stream. More specifically, we want to calculate the total count of all color readings, the total value of all color readings, and the average value of all the color readings, to ultimately display them in the frontend.



```
colorSensorStream
        .groupBy((key, value) -> {
            if (!key.equals("none")) {
                System.out.println("Color detected: " + key + " Value: " + value);
            }
            return key;
        }, Grouped.with(Serdes.String(), Serdes.Double()))
        .aggregate(
                ColorStats::new,
                ColorStats::aggregate,
                Materialized.<~>
                                as( storeName: "colorStats")
                        .withKeySerde(Serdes.String())
                        .withRetention(Duration.ofMinutes(30))
                        .withValueSerde(colorStatsSerde));
```

Figure 29: Grouping & Materializing

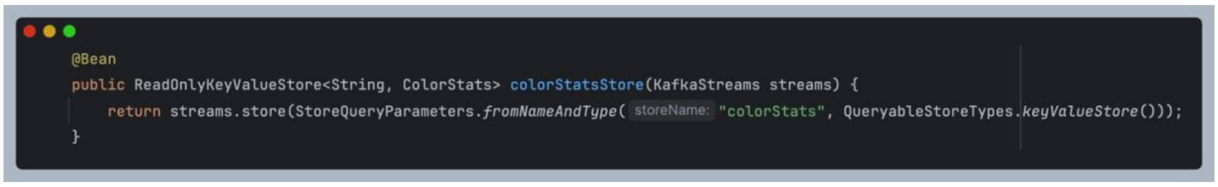
We can build up on the prepared “colorSensorStream”, which contains events with the discretized color reading (“blue”, “red”, “white”) as the key, and the actual color reading value in the payload. First, the stream needs to be grouped in a “key-based” manner. Afterwards, we utilize the “aggregate()” method that has three parameters: (1) The first parameter is the “initializer”, that creates a new ColorStats object for each group, setting the color metrics on zero. This ColorStats manifests the custom, un-materialized state store. (2) The second parameter is the “adder”, that is invoked for each new event in the event stream. The “aggregate()” method from the ColorStats object, is implemented in a straightforward way, see figure below.



```
public static ColorStats aggregate(String key, Double value, ColorStats agg) {
    long newTotalCount = agg.getTotalReadings() + 1;
    double newTotalOccurrences = agg.getTotalColorValues() + value;
    double newAverageColorVal = newTotalOccurrences / newTotalCount;
    return new ColorStats(newTotalCount,newTotalOccurrences,newAverageColorVal);
}
```

Figure 30: Aggregation

(3) Finally, the third parameter specifies how the stats should be saved in the materialized state store ColorStats.



```java
@Bean
public ReadOnlyKeyValueStore<String, ColorStats> colorStatsStore(KafkaStreams streams) {
    return streams.store(StoreQueryParameters.fromNameAndType( storeName: "colorStats", QueryableStoreTypes.keyValueStore()));
}
```

Figure 31: ColorStats Store

By creating this Bean of the ColorStats state store, we can access the state store wherever we like within the monitoring service (This, and the materialization of the state store is needed because otherwise the state store would only be accessible within the stream processing topology). We query this state store for example in the MonitoringRestController class to retrieve the current color reading metrics, and to display them in the frontend.

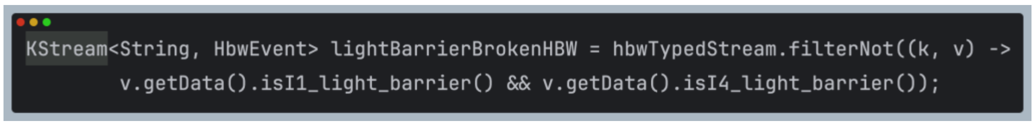
Note: This is an example for non-windowed stateful stream processing.

Time Difference Aggregation (Lectures 9 & 10)

Now, we get to the windowed stateful stream processing operations.

In the HBW factory station, there are two light barriers of interest for us. If the "i4" light barrier is broken (false), the warehouse transport unit has dropped off the workpiece on the delivery band, where it gets driven to the second light barrier "i1". If the light barrier "i1" is broken (false), the gripper of the VGR station gets informed to pickup the workpiece from the delivery band. To calculate an additional metric for our factory, we are interested in finding out the time difference between those two events.

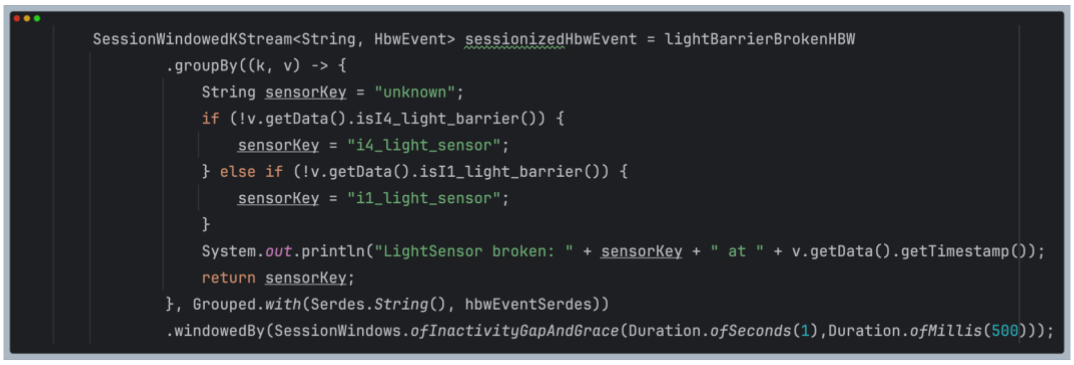
To accomplish this use case, in the monitoring service, we filter for events that have a broken light barrier, see figure 31. Remember: A broken light barrier is denoted with "false".



```java
KStream<String, HbwEvent> lightBarrierBrokenHBW = hbwTypedStream.filterNot((k, v) ->
        v.getData().isI1_light_barrier() && v.getData().isI4_light_barrier());
```

Figure 32: Filtering

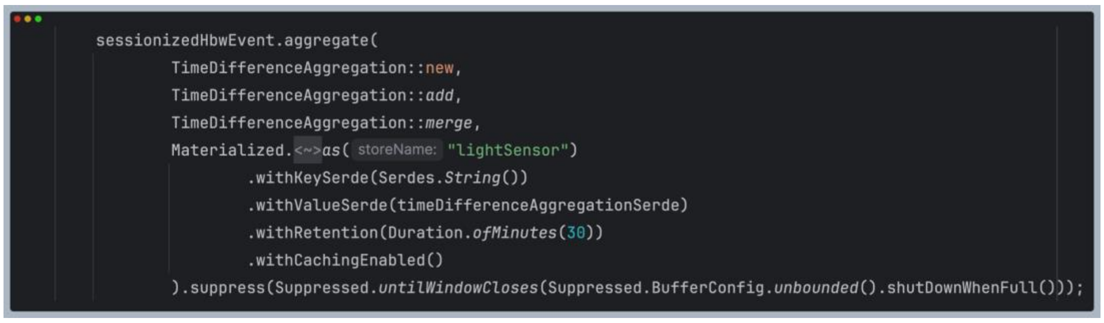
Next, we group the "lightBarrierBrokenHBW" stream by the light barrier type, either "i4" or "i1". Because we want to find out the time difference of the light barrier break event for every workpiece that passes from the HBW to the VGR station, we apply a session window on the grouped stream of events. The decision of a session window is straightforward because only with this window type, we can catch all events belonging to a given workpiece, even if this process takes a variable amount of time. If there is an inactivity gap of 1 second, the session window gets closed, until a new session window gets created by the event of a broken light barrier.



```java
SessionWindowedKStream<String, HbwEvent> sessionizedHbwEvent = lightBarrierBrokenHBW
        .groupBy((k, v) -> {
            String sensorKey = "unknown";
            if (!v.getData().isI4_light_barrier()) {
                sensorKey = "i4_light_sensor";
            } else if (!v.getData().isI1_light_barrier()) {
                sensorKey = "i1_light_sensor";
            }
            System.out.println("LightSensor broken: " + sensorKey + " at " + v.getData().getTimestamp());
            return sensorKey;
        }, Grouped.with(Serdes.String(), hbwEventSerdes))
        .windowedBy(SessionWindows.ofInactivityGapAndGrace(Duration.ofSeconds(1),Duration.ofMillis(500)));
```

Figure 33: Windowed Stream

Now we have the preprocessed relevant events grouped per light barrier and by session window. The “sessionizedHbwEvent” stream can now be aggregated. Therefore, we specify the initializer to be a new instance of the custom TimeDifferenceAggregation class. This creates an object with a timestamp for the first timestamp of the respective light barrier and a timestamp for the last timestamp of that same light barrier, initialized with “null”. Each next event will be applied to the TimeDifferenceAggregation::add method, that is setting the new “firstTimestamp”, or “lastTimestamp” attribute respectively (if appropriate). The aggregations for light barriers “i4” and “i1” are now to be merged, to calculate the time difference between the first “i4” timestamp, and the last “i1” timestamp. This implementation can be found in TimeDifferenceAggregation::merge. The result of this merge operation is then stored in the state store “lightSensor” on which we can perform interactive queries similar to the implementation of ColorStats.



```java
sessionizedHbwEvent.aggregate(
        TimeDifferenceAggregation::new,
        TimeDifferenceAggregation::add,
        TimeDifferenceAggregation::merge,
        Materialized.<~>as( storeName: "lightSensor")
                .withKeySerde(Serdes.String())
                .withValueSerde(timeDifferenceAggregationSerde)
                .withRetention(Duration.ofMinutes(30))
                .withCachingEnabled()
        ).suppress(Suppressed.untilWindowCloses(Suppressed.BufferConfig.unbounded().shutDownWhenFull()));
```

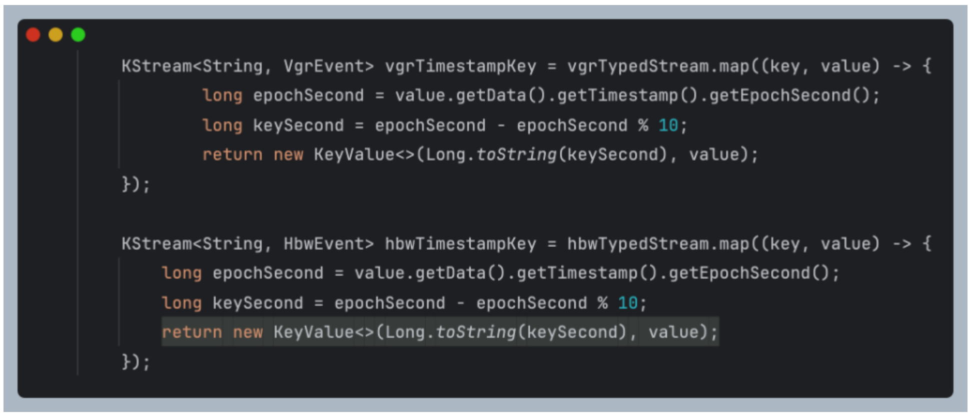
Figure 34: Aggreagation of sessonized events

Joining of Streams / Session Window (Lectures 9 & 10)

Another example in our implementation for a windowed stateful stream processing operation is the joining of streams. Here, we want to take events from both typed streams HBW and VGR, and join them on the timestamp, to have one event that represents the state of the complete factory for one specific point in time.

For this operation we leverage our work that we did with the customized timestamp extractor, because the timestamp of those events is already specified with the event time semantic – i.e. the timestamp, when the factory emitted the respective MQTT message.

For both event streams HBW and VGR, because we want to join on the timestamp, we need to map the values, such that the event timestamp is taken as the key. But because it is highly unlikely that two timestamps will be identical, we truncate all timestamps by fixed 10 seconds, see figure 34.



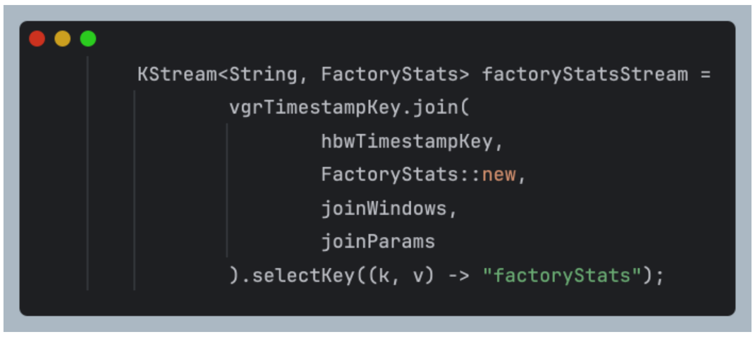
```
KStream<String, VgrEvent> vgrTimestampKey = vgrTypedStream.map((key, value) -> {
        long epochSecond = value.getData().getTimestamp().getEpochSecond();
        long keySecond = epochSecond - epochSecond % 10;
        return new KeyValue<>(Long.toString(keySecond), value);
});

KStream<String, HbwEvent> hbwTimestampKey = hbwTypedStream.map((key, value) -> {
    long epochSecond = value.getData().getTimestamp().getEpochSecond();
    long keySecond = epochSecond - epochSecond % 10;
    return new KeyValue<>(Long.toString(keySecond), value);
});
```

Figure 35: Truncated Timestamps

Next, we defined the “joinParams” and the “JoinWindows” parameters. Therefore, we specified that the join window should be of length 5 seconds with a grace period of 1 second. That means that the maximum allowable difference in time between two records will be 5 seconds. If any record arrives up to 1 second after the window and time, it will still be considered for joining.

In figure 35 we then join the two streams and use the wrapper class FactoryStats for this – because Kafka Streams can not join streams of different types directly. The FactoryStats object then holds two references to the two events for the HBW and the VGR stations.

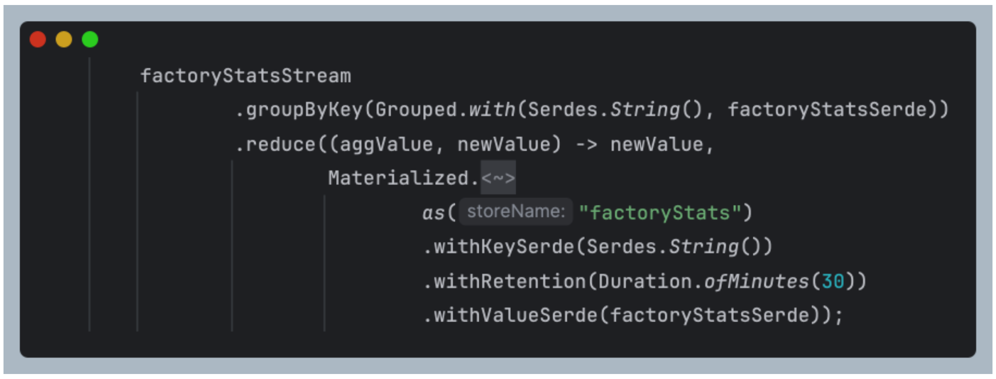
This joining operation results in a table of lots of joined events, but because we only want to have the most recent joined event (to most recent status of the factory), we rekey all the joined events to the fixed literal "factoryStats". This has the effect, that we now, only have events with this key.



```java
KStream<String, FactoryStats> factoryStatsStream =
        vgrTimestampKey.join(
                hbwTimestampKey,
                FactoryStats::new,
                joinWindows,
                joinParams
        ).selectKey((k, v) -> "factoryStats");
```

Figure 36: Inclusion of relevant keys

Now, we can group these events by their key. By applying the "reduce()" method on this group as we did, this results in having only the most recent joined event in our table. Finally, we can materialize the table as "factoryStats", to use the information in the same way as we did as for the ColorStats state store. Note: The retention time specifies how long the records in the materialized state store should be retained.



```java
factoryStatsStream
        .groupByKey(Grouped.with(Serdes.String(), factoryStatsSerde))
        .reduce((aggValue, newValue) -> newValue,
                Materialized.<~>
                        as( storeName: "factoryStats")
                        .withKeySerde(Serdes.String())
                        .withRetention(Duration.ofMinutes(30))
                        .withValueSerde(factoryStatsSerde));
```

Figure 37: Configuration oft he factoryStatsStream

Architectural Decisions

Whenever relevant, architectural decisions have been incorporated into the report. We faced architectural decisions mainly in the first part of the lectures where the systems architecture has been developed (microservices, workflow engine, etc.) (ADR-1 to ADR-7) where in the second part of the lectures, where we considered stream processing, we decided on the streaming topology (ADR-8) as well as on the message format (ADR-9). The Architecture Decision Records (ADR) that have been decided on can be found under:

https://github.com/jonaslanzlinger/EDPO-Group1-Project/tree/master/docs/adrs

Reflections & Lessons learned

Messaging

Effective messaging systems often incorporate multiple patterns to ensure smooth and efficient communication. One common and straightforward approach is event notification, which allows systems to quickly notify other components of changes or updates. Event-Driven Architecture (EDA) is a double-edged sword; it can both simplify and complicate systems. While EDA helps in decoupling components, making them more modular and scalable, it can also introduce complexity in terms of maintaining and monitoring the system. Visual tools, such as graphical representations of messaging flows, are invaluable in maintaining a clear mental model of the system, ensuring all team members understand how messages traverse through the system. With the use of Camunda the visualization of the processes and the tokens enabled us to gain a shared understanding and to iterate fast during development.

Kafka

Using Kafka with its out-of-the-box configuration can streamline the setup process, allowing for quick deployment and testing. Managing a Kafka cluster is made simpler with Zookeeper, which handles the coordination and configuration of Kafka brokers. However, consumer lag can become a significant issue, potentially leading to the retransmission of messages. Therefore, we found out that it is crucial to configure retention and cleanup policies carefully to avoid data loss due to offset misconfigurations in our case the out-of-the-box configuration was already sufficient. Deploying multiple Kafka instances enhances redundancy, ensuring that message processing remains reliable even in the event of individual broker failures but in our case where everything is running on a single host, multiple brokers would not give us more availability.

Specific learnings more in depth

One of the most important learnings was how vital it is to have additional UUIDs, or other identifiers separate to the ones given by Camunda or Kafka. Even though the platform given ones are enough in most cases, having the ability to fall back onto our own identifiers was very helpful. In the case of Camunda, it was evident as soon as we started working with subprocesses. Since Camunda will give new Instance Keys, there was no real way to link a parent and a subprocess together. Here we found that just having an OrderId in our own object sufficed to be able to do subprocesses. Also with Kafka streams, we found in the case of, for example, the timestamps, that the ones given by the service were not always enough. Here we utilized the timestamps that came from the MQTT messages from the factory.

Behavior of different stores in Kafka

Another challenge that we faced was the integration of different stores to enable interactive querying. While we already had successfully implemented a KeyValueStore that just returned aggregated current statistics, we aimed to be able to report the length of time a light barrier is broken. To enable this functionality, we needed a sessionedWindow and the resulting sessionStore to query the data. The implementation was analog to the previous just with slightly different classes. But once we had the endpoint running, we noticed that the response time was rather slow (between 5 – 10 seconds). Since this is an unusually long time, we were sure it had to be some unwanted behavior. Through experimenting and logging the times for the specific operations, such as the initial "fetch()" operation or the inidividual "next()" operations and other related operations. We noticed that the initial "fetch()" was the reason for the slow response, as all other methods were done in milliseconds. We then compared it to the "backwardsFetch()" operation, which does the same as the "fetch()" but returns the newest session first! What we observed was that here the initial "backwardsFetch()" was not slow, neither were the "next()" operations. Not until we reached the last (we know it to be the last) session. For some reason, the last "next()" takes a long time to return the information to us, that there are no more sessions left. We also posted on the Confluent forum about our problem: https://forum.confluent.io/t/sessionstore-fetch-somekey-is-very-slow/10723/2

In the end, we implemented a (deprecated) endpoint, that runs each "next()" operation in a separate thread and just kills it if it steps over a time limit. While this is not really the best practice, it enables the wanted performance and still provides all known to be valid sessions. Since in the forum, they described the slowness of the "fetch()" for sessionStores as an expected behavior, we left the implementation as is. But it is definitely a learning on Kafka and its specific behavior, and forced us to consult the docs as well as many google results.

Plotting of Station Data

To understand the change of the values emitted by each station, we plotted the data against the time. The data that we have used has been recorded by us in a laboratory session where we recorded all MQTT messages that have been emitted by the stations. The following figure shows in (1) that the HBW_1 ("High Bay Warehouse") initiates the collection of the workpiece, next in (2) we see that the workpiece has been loaded onto the assembly line and breaks the first light barrier. Between (2) and (3) we see that the workpiece container breaks the second light barrier and is then grabbed by the VGR_1 ("Vacuum Gripper Arm") and deposited to the color sensor. In (4) we see that the gripper arm has grabbed the workpiece again and transported it to the delivery station.

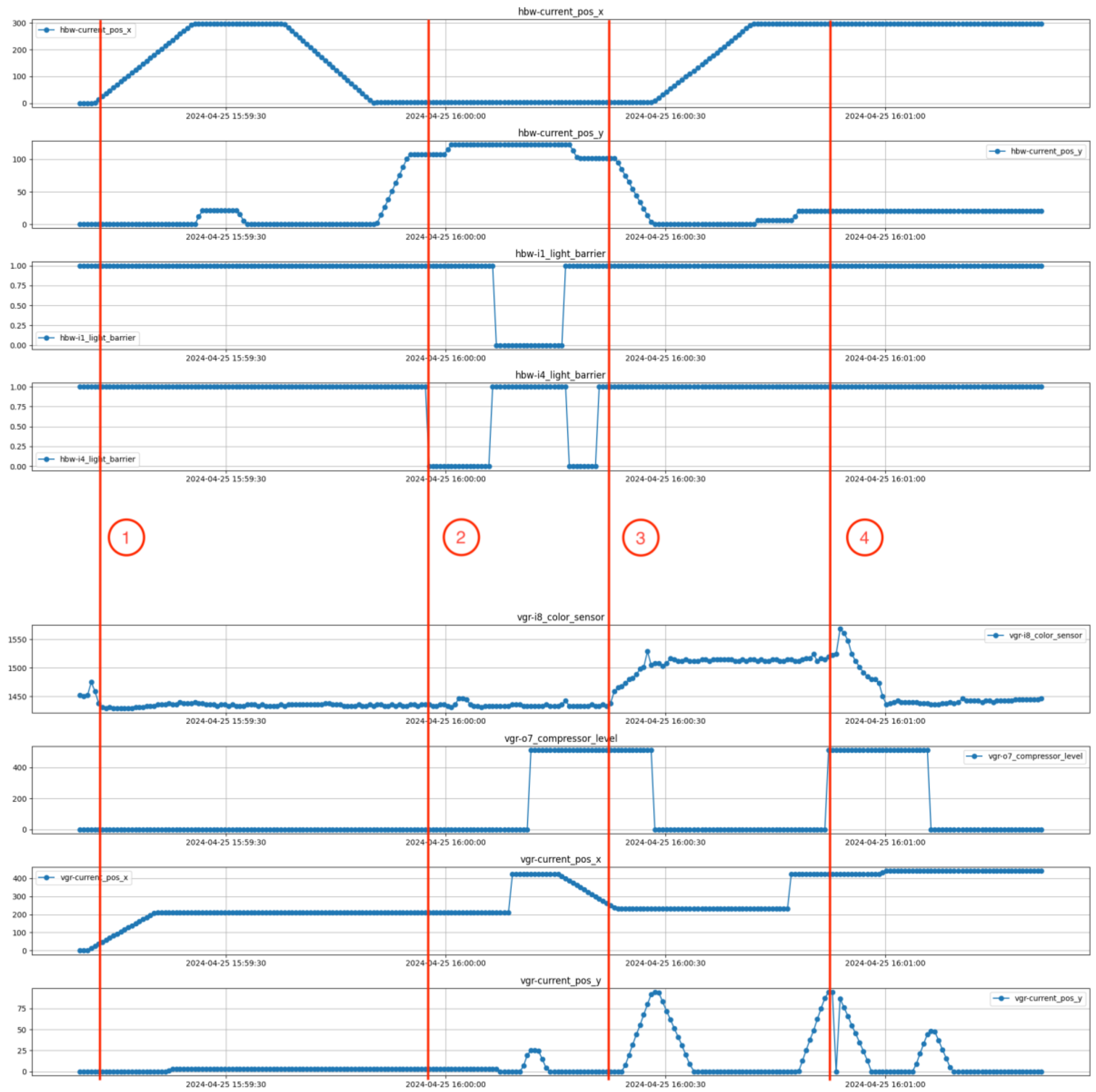


Figure 38: Plotted sensor data agains time

Work Distribution

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