Universität St.Gallen

EDPO-Group 4 Assignment 2

Yasmin Lützelschwab
Stefan Meier
23.05.2024

Agenda

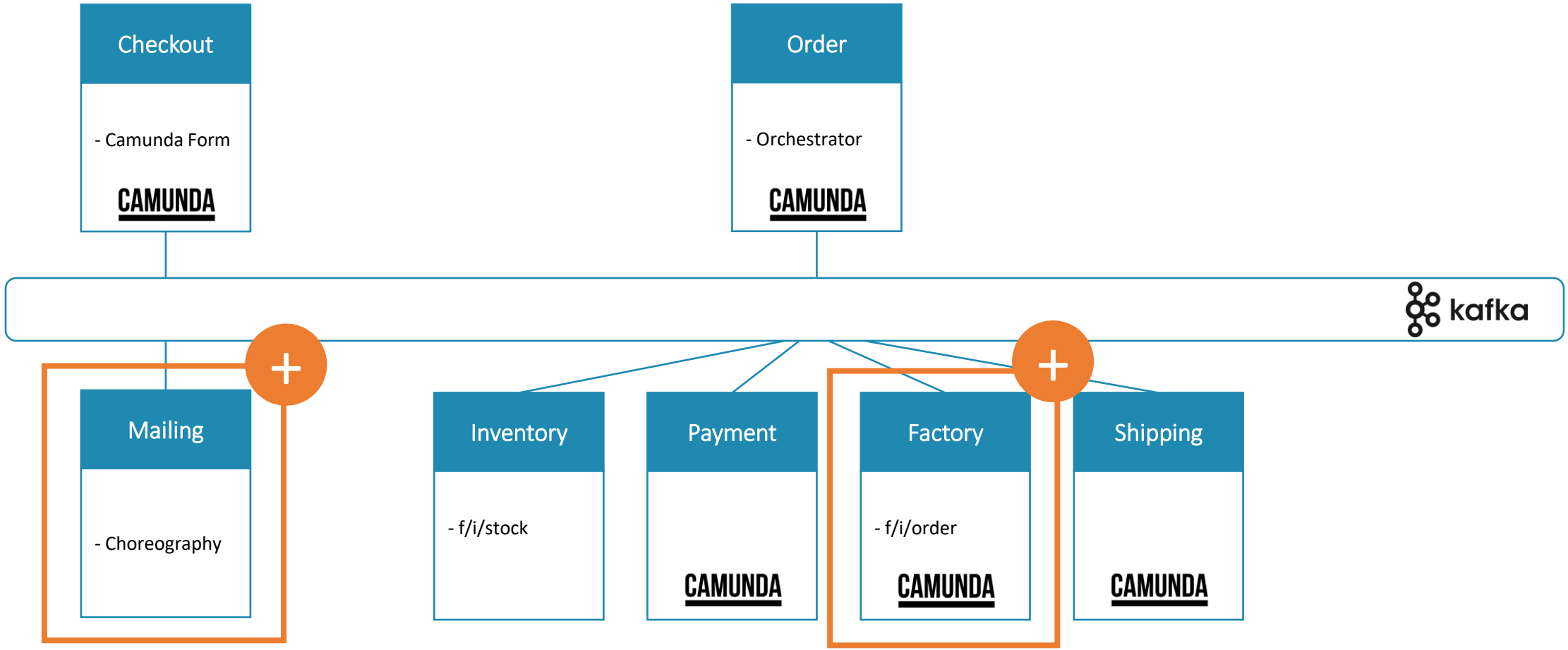
Project overview

Stream Processing Apps and Concepts

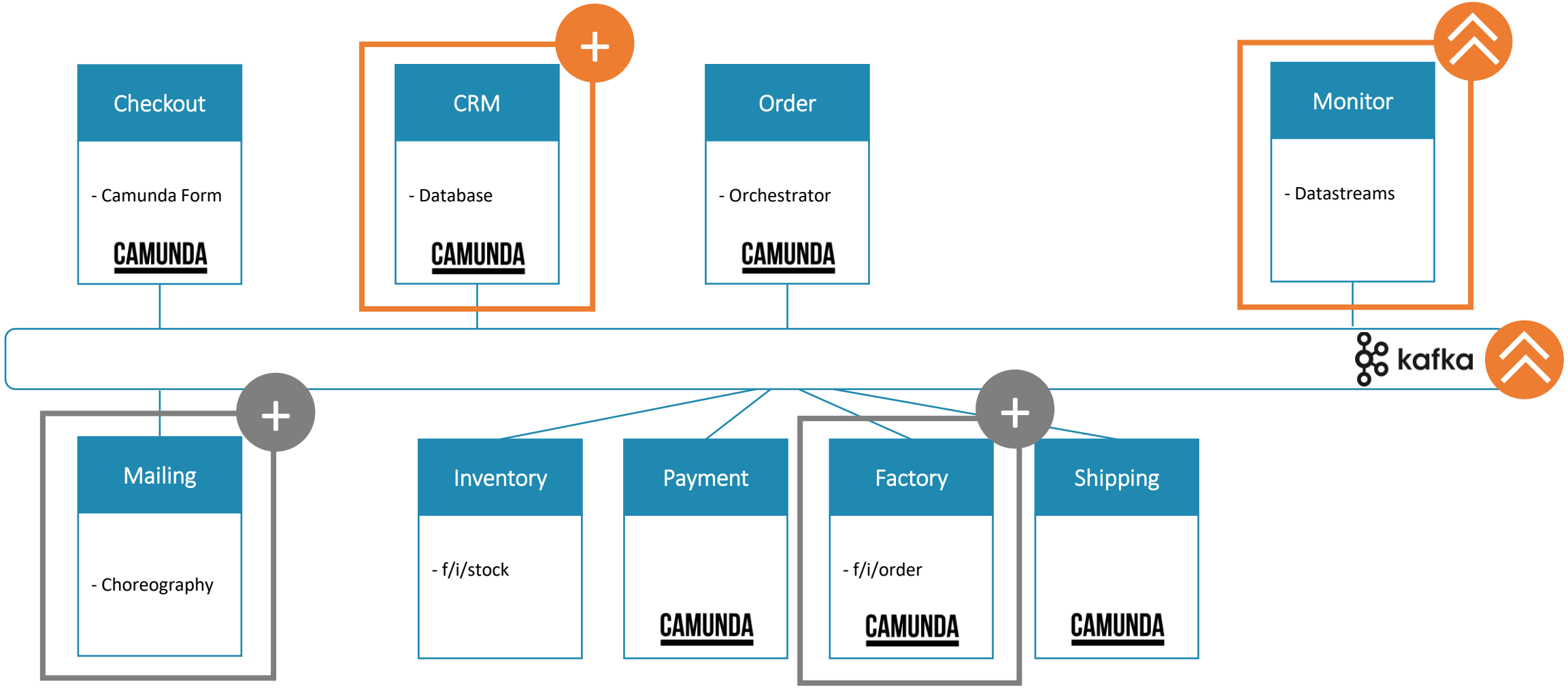
Demo

Q&A

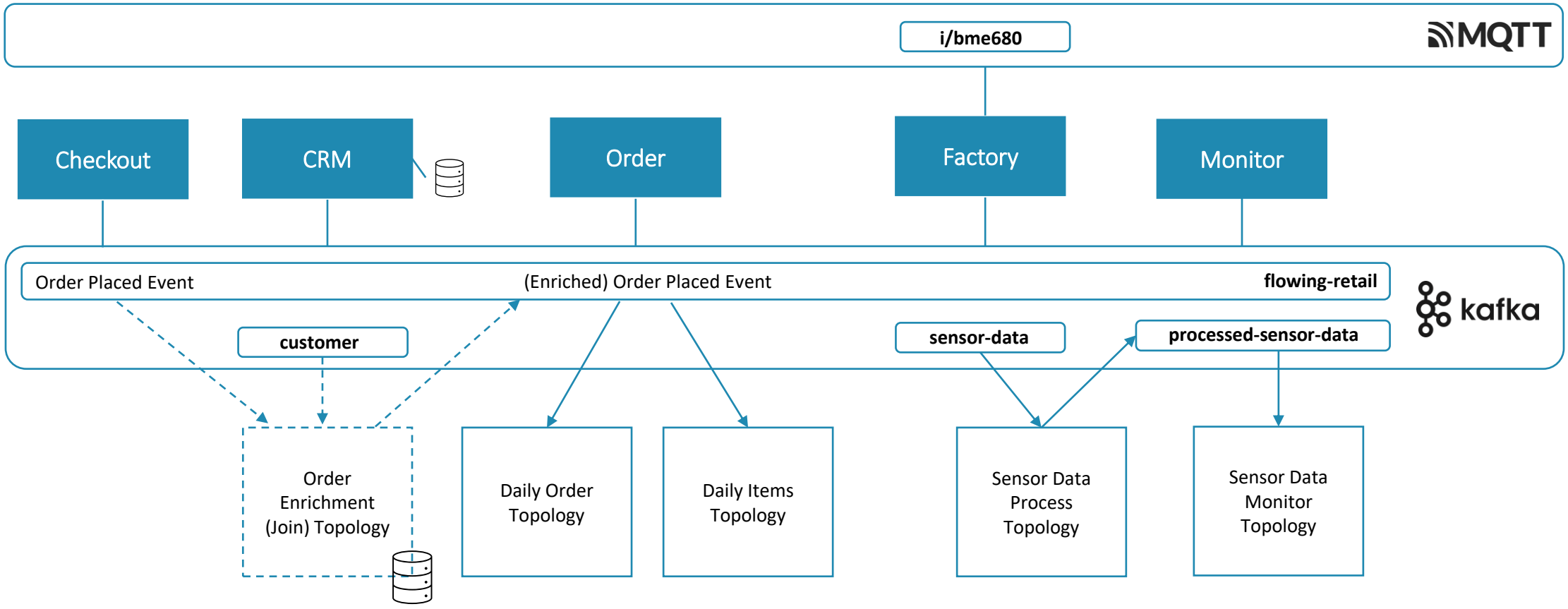
Project State of Assignment 1



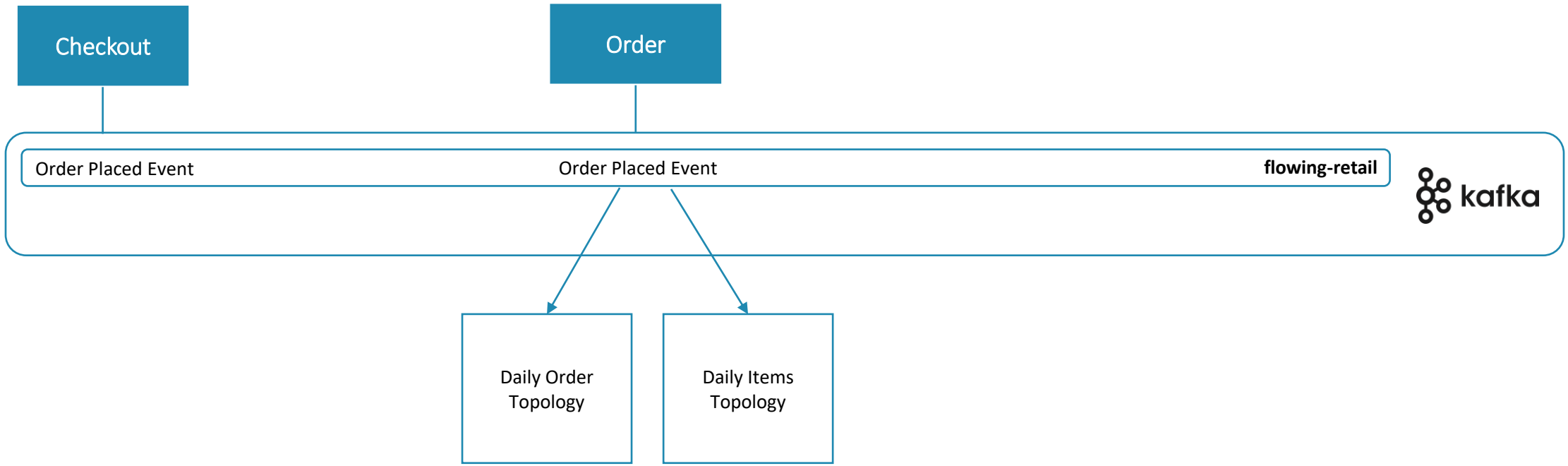
Project State of Assignment 2



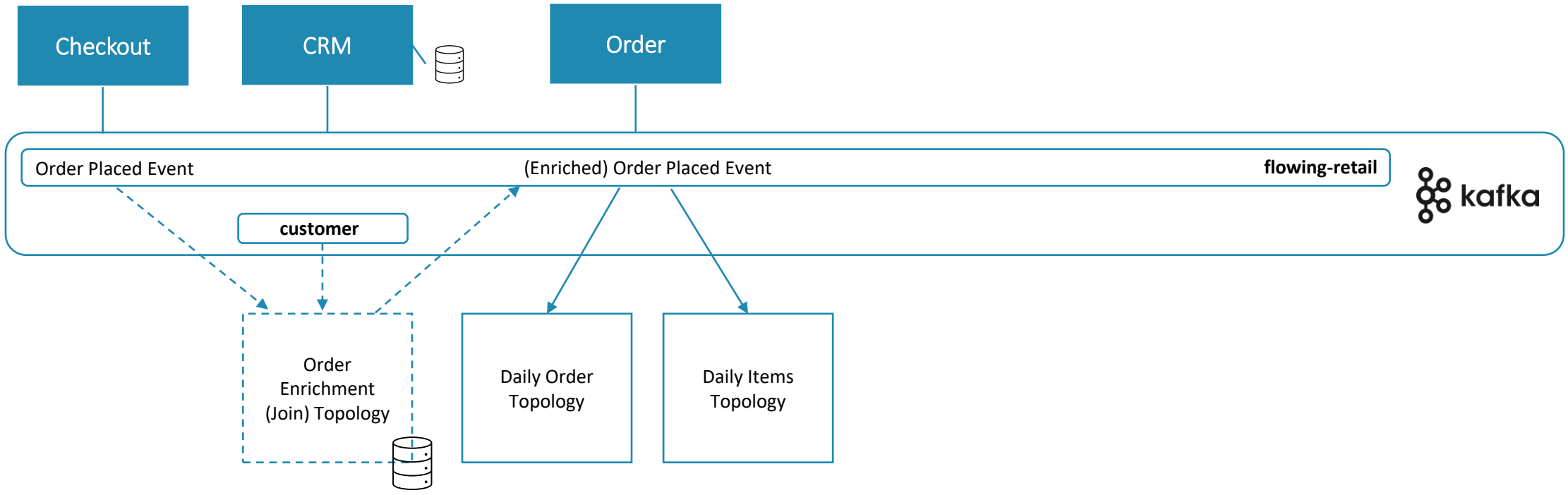
Stream Processing Apps



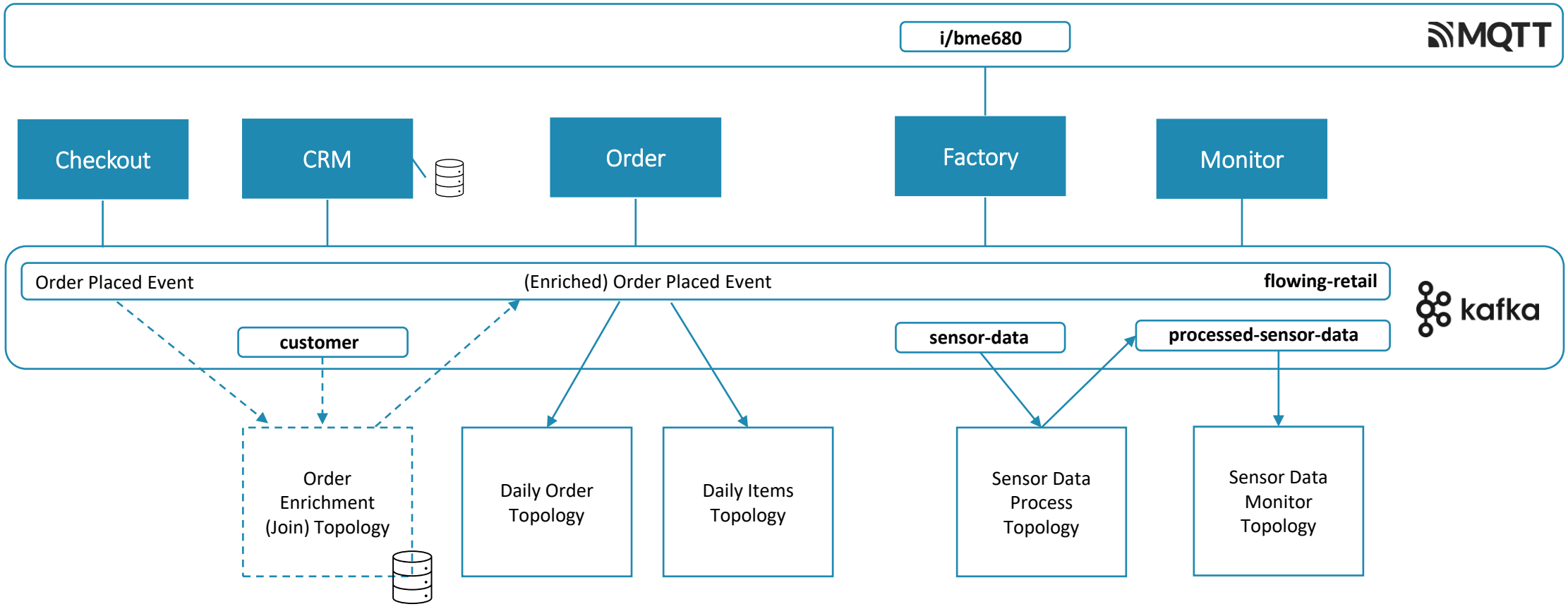
Stream Processing Apps



Stream Processing Apps

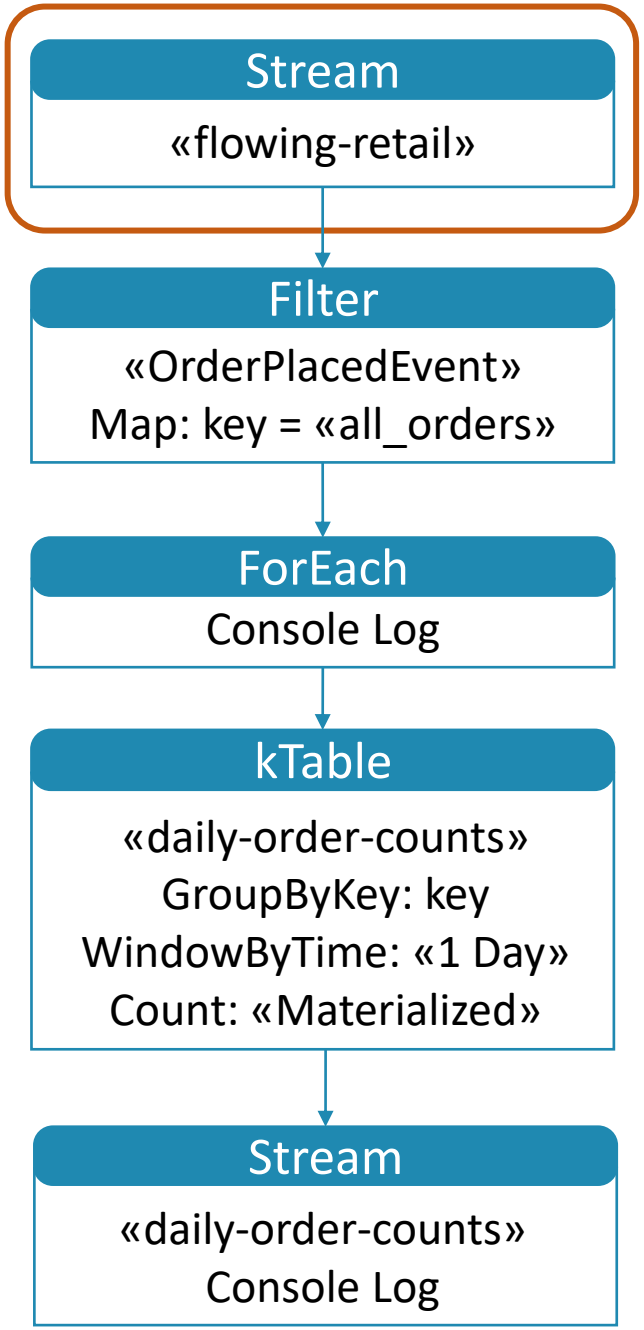


Stream Processing Apps



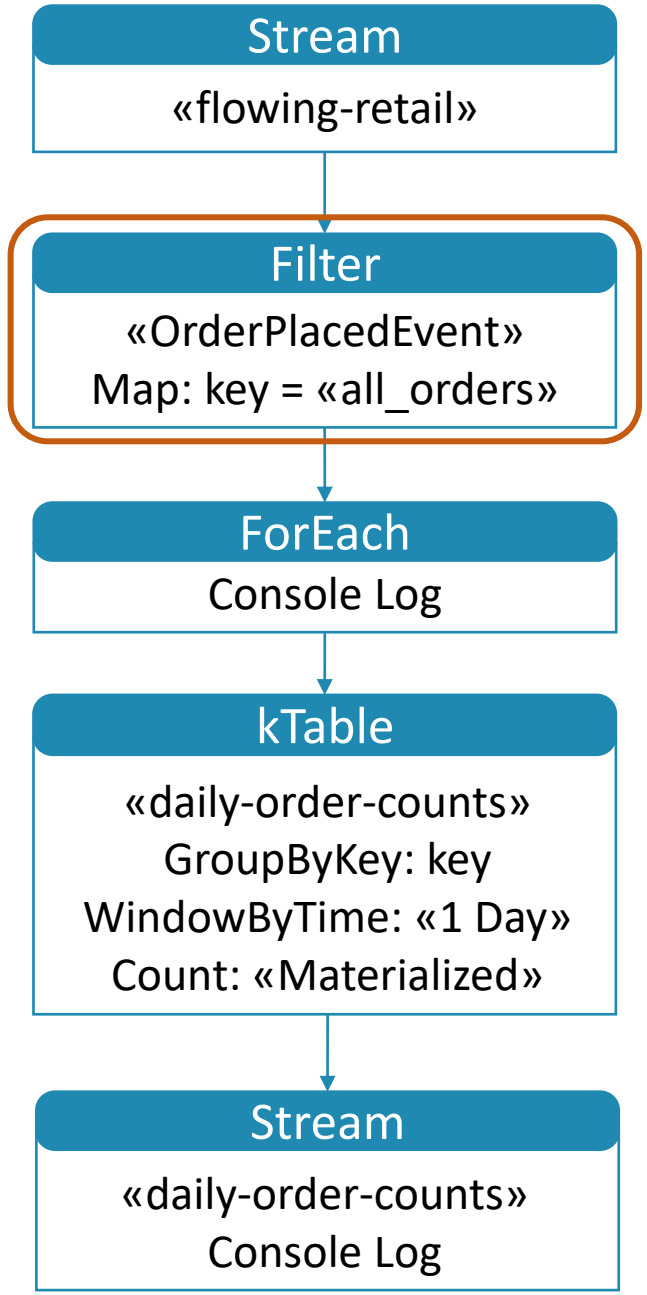
Daily Order Topology

```
// Create new itemsStream and consume from the "flowing-retail" topic
KStream<String, Message<Order>> ordersStream =
        builder.stream( topic: "flowing-retail",
                        Consumed.with(Serdes.String()
                        , new MessageOrderSerde()));
```



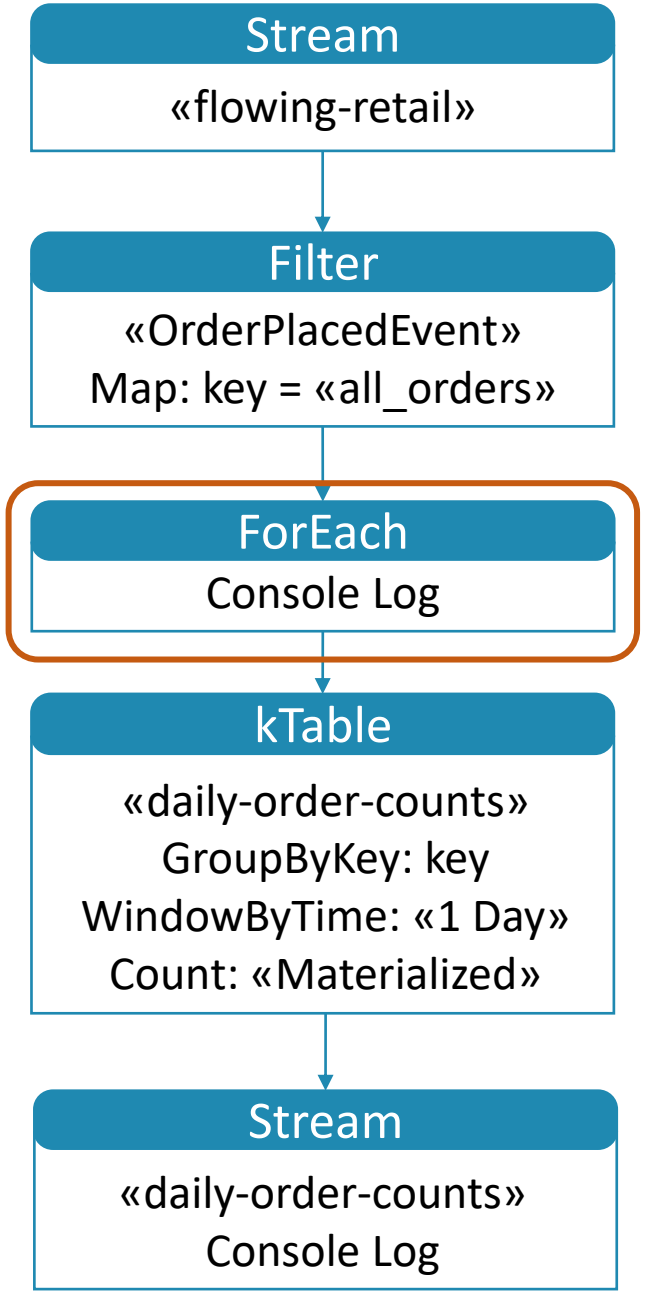
Daily Order Topology

```
// Filter for only "OrderPlacedEvent" and then map to set the new key to order ID
KStream<String, Order> orderPlacedEvents = ordersStream
        .filter((key, value) -> value != null && "OrderPlacedEvent".equals(value.getType()))
        .map((key, value) -> KeyValue.pair("all_orders", value.getData()));
```



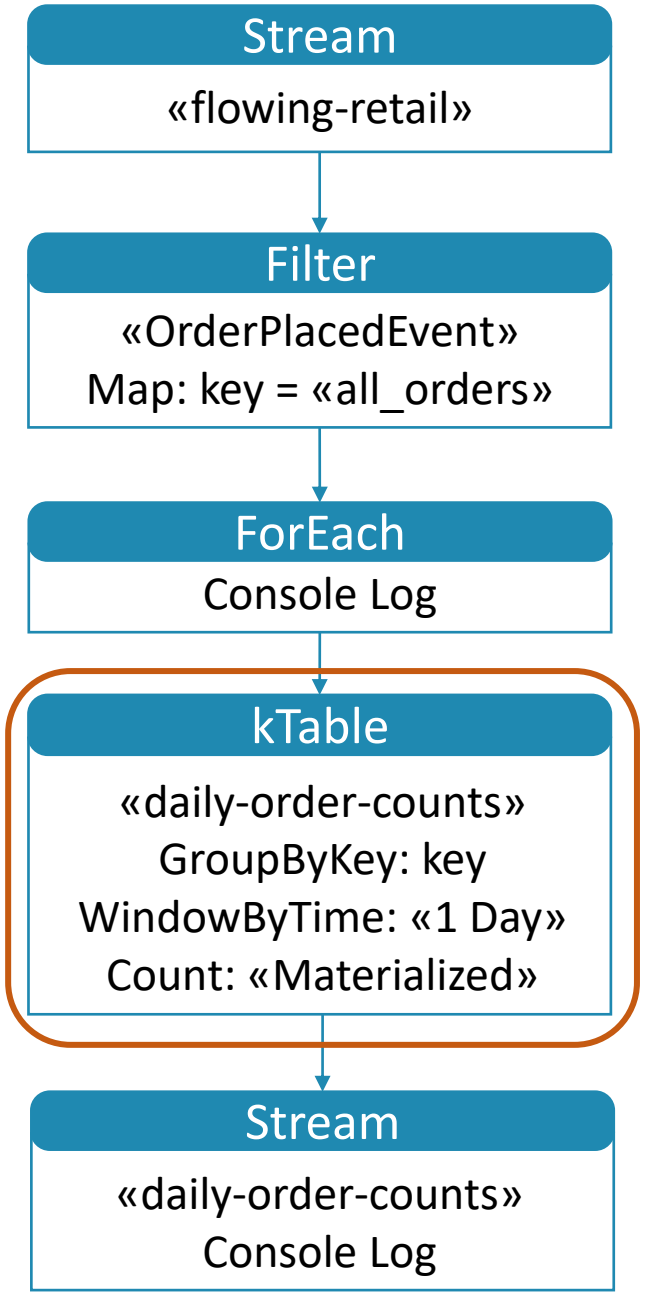
Daily Order Topology

```
// Print each valid message to the console
orderPlacedEvents.foreach((key, order) -> {
    System.out.println("Streamed Message Key: " + key + ", " +
            "Order ID: " + order.getId() + ", Items: " + order.getItems());
    order.getItems().forEach(item ->
            System.out.println("Item: " + item.getArticleId() +
                    ", Quantity: " + item.getAmount()));
});
```



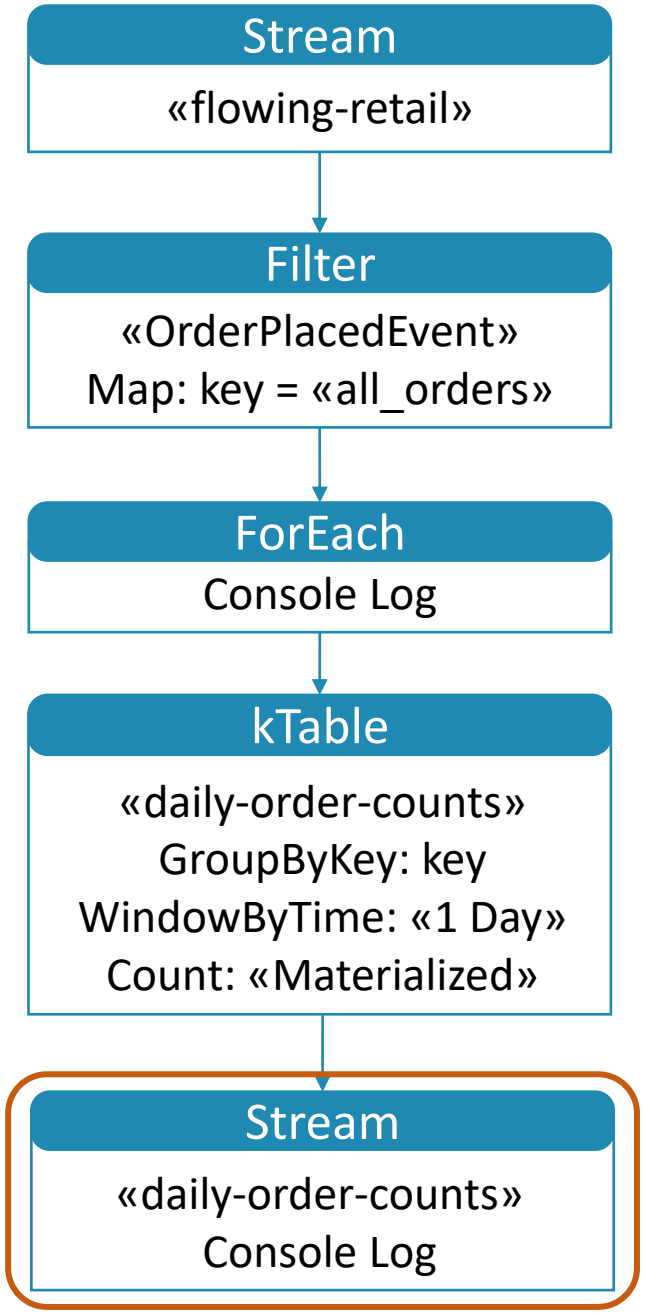
Daily Order Topology

```
// Group by key (order ID) and window by day, then count
KTable<Windowed<String>, Long> dailyOrderCounts = orderPlacedEvents
        .groupByKey(Grouped.with(Serdes.String(), new OrderSerde())) KGroupedStream<String, Order>
        .windowedBy(TimeWindows.of(Duration.ofDays(1))) TimeWindowedKStream<String, Order>
        .count(Materialized.<String, Long, WindowStore<Bytes,
                        byte[]>>as( storeName: "daily-order-counts")
                .withKeySerde(Serdes.String())
                .withValueSerde(Serdes.Long()));
```



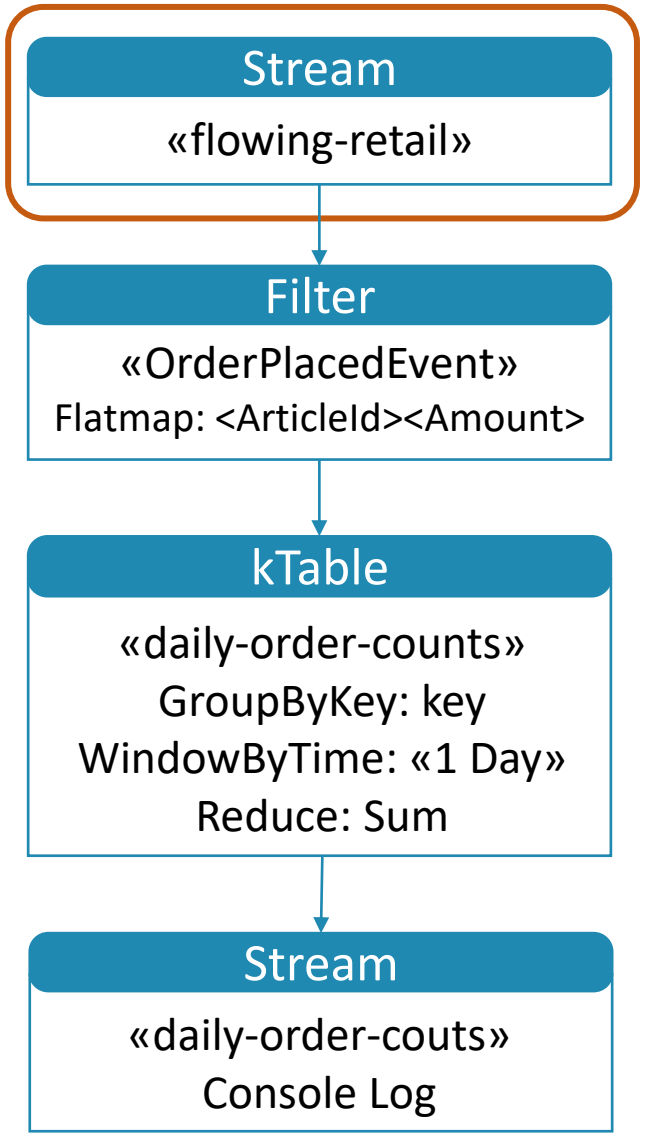
Daily Order Topology

```
// Print the daily counts
dailyOrderCounts.toStream().foreach((windowedKey, count) -> {
    System.out.println("**DailyOrdersTopology** \n" +
            "Window Start Time: " + windowedKey.window().startTime() +
            ", Order ID: " + windowedKey.key() +
            ", Total Orders Count: " + count);
});
```



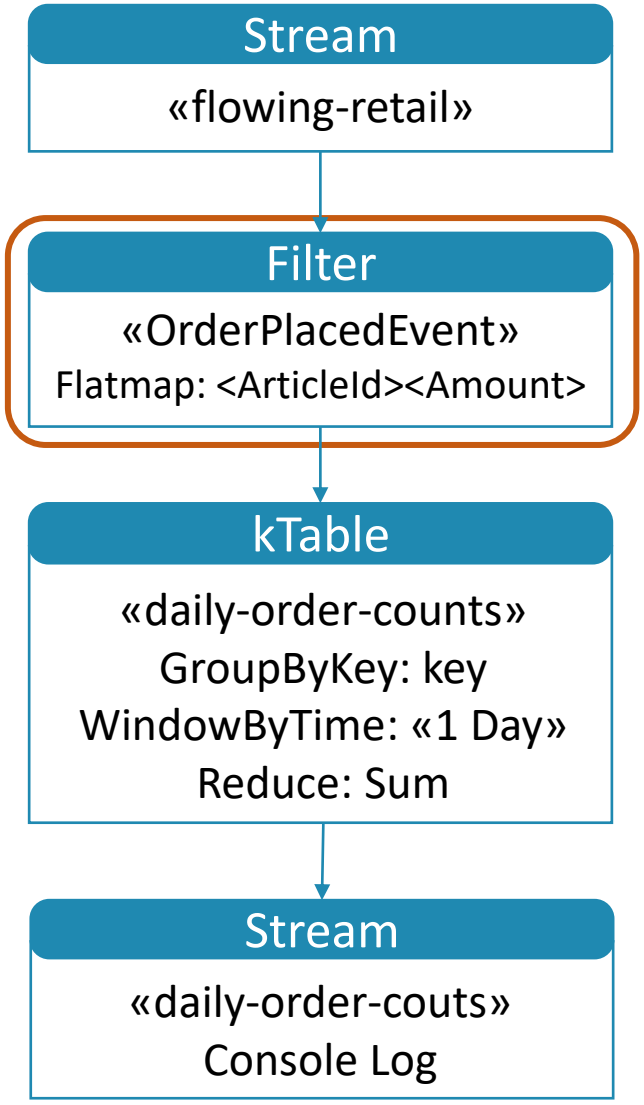
Daily Items Topology

```
// Create new itemsStream and consume from the "flowing-retail" topic
KStream<String, Message<Order>> ordersStream =
        builder.stream( topic: "flowing-retail",
                        Consumed.with(Serdes.String()
                        , new MessageOrderSerde()));
```



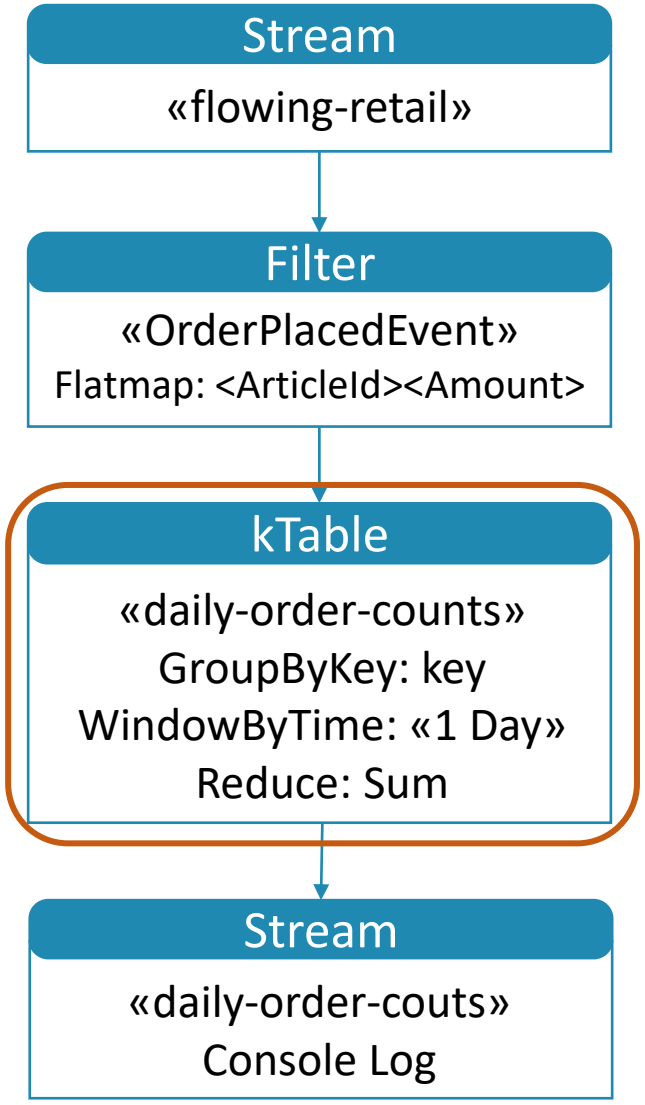
Daily Items Topology

```
// Filter for only "OrderPlacedEvent" and then map to set the new key to order ID
KStream<String, Long> itemsStream = ordersStream
        .filter((key, value) -> value != null && "OrderPlacedEvent".equals(value.getType()))
        // Use flatMap to transform the list of items in each order to a stream of color key-values
        .flatMap((key, value) -> {
            List<KeyValue<String, Long>> itemsColorList = new ArrayList<>();
            for (OrderItem orderItem: value.getData().getItems()) {
                itemsColorList.add(new KeyValue<>(orderItem.getArticleId()
                        , (long) orderItem.getAmount()));
            }
            return itemsColorList;
        });
```



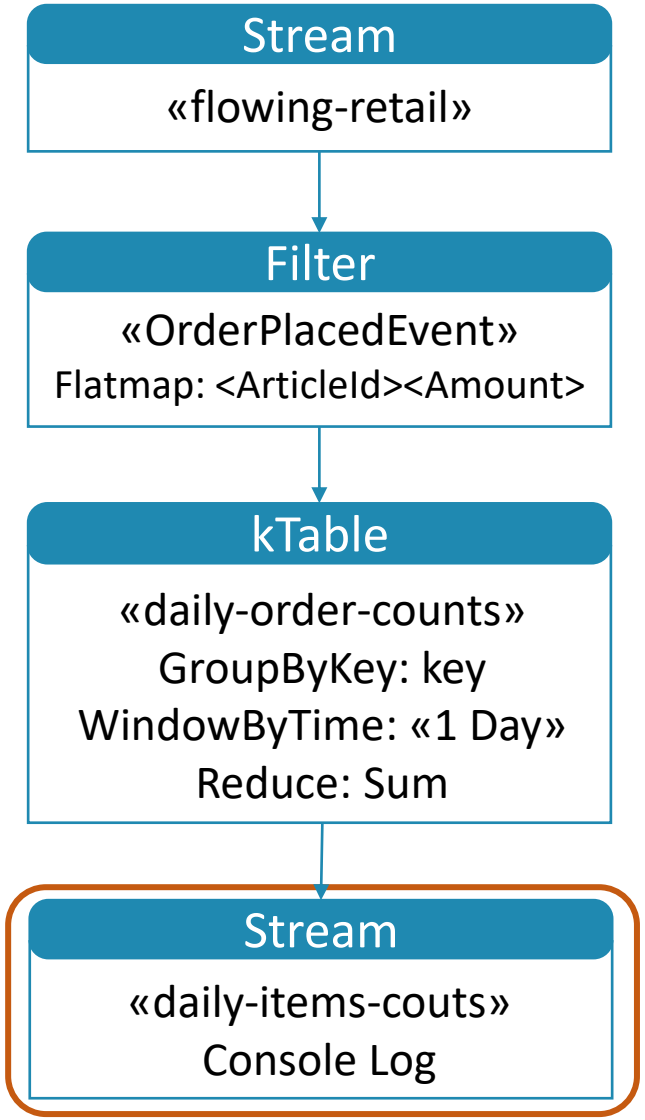
Daily Items Topology

```
// Group by item ID, window by day, and aggregate the count of each item
KTable<Windowed<String>, Long> dailyItemsCount = itemsStream
        .groupByKey(Grouped.with(Serdes.String(), Serdes.Long()))  KGroupedStream<String, Long>
        .windowedBy(TimeWindows.of(Duration.ofDays(1)))  TimeWindowedKStream<String, Long>
        .reduce(Long::sum)
        ;
```

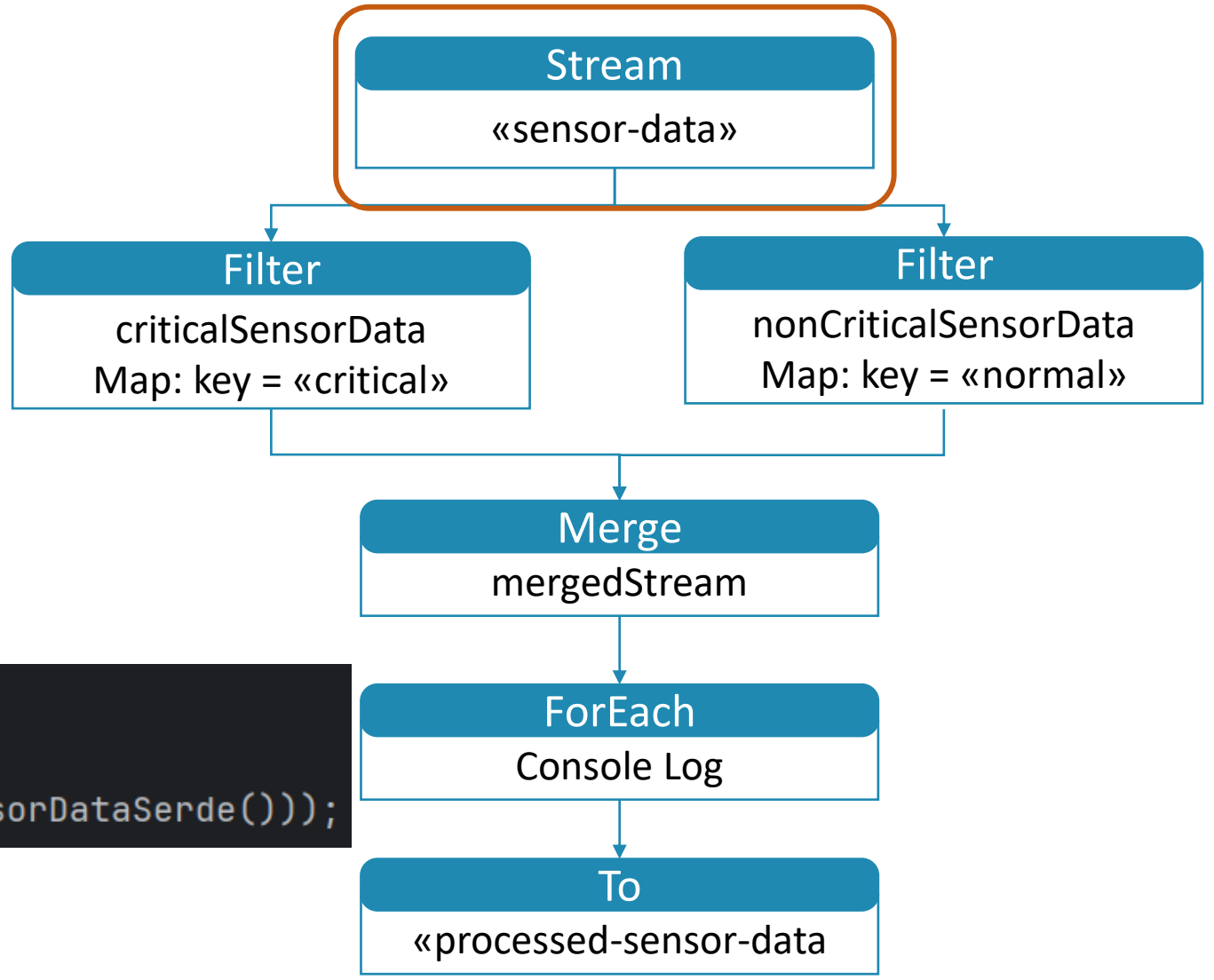


Daily Items Topology

```
// Print the daily item counts
dailyItemsCount.toStream().foreach((windowedKey, count) -> {
    System.out.println("**DailyItemsTopology** \n" +
            "Window Start Time: " + windowedKey.window().startTime() +
            ", Item ID: " + windowedKey.key() +
            ", Total Items Count: " + count);
});
```

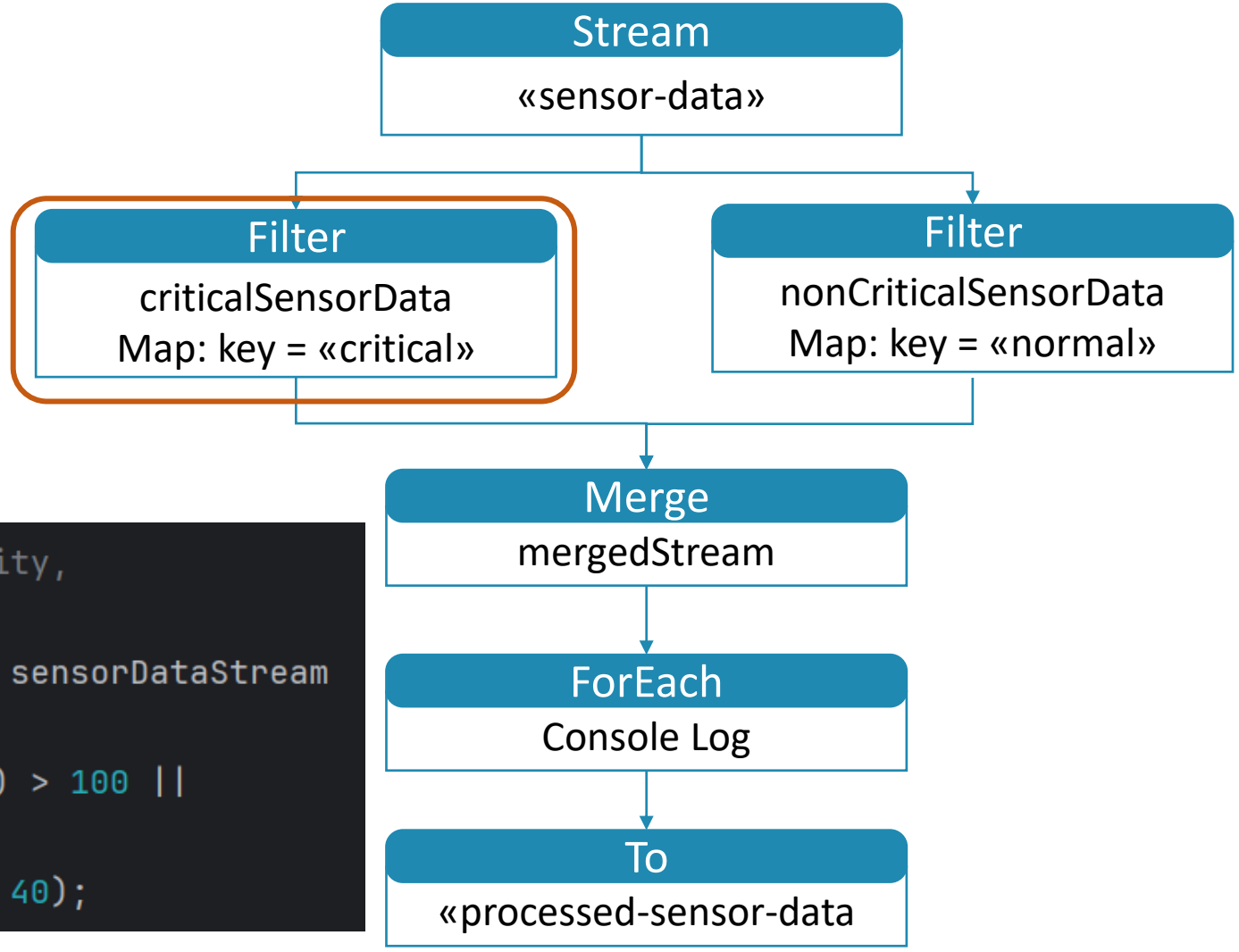


Sensor Data Process Topology




```
KStream<String, SensorData> sensorDataStream =
        builder.stream( topic: "sensor-data",
        Consumed.with(Serdes.String(), new SensorDataSerde()));
```

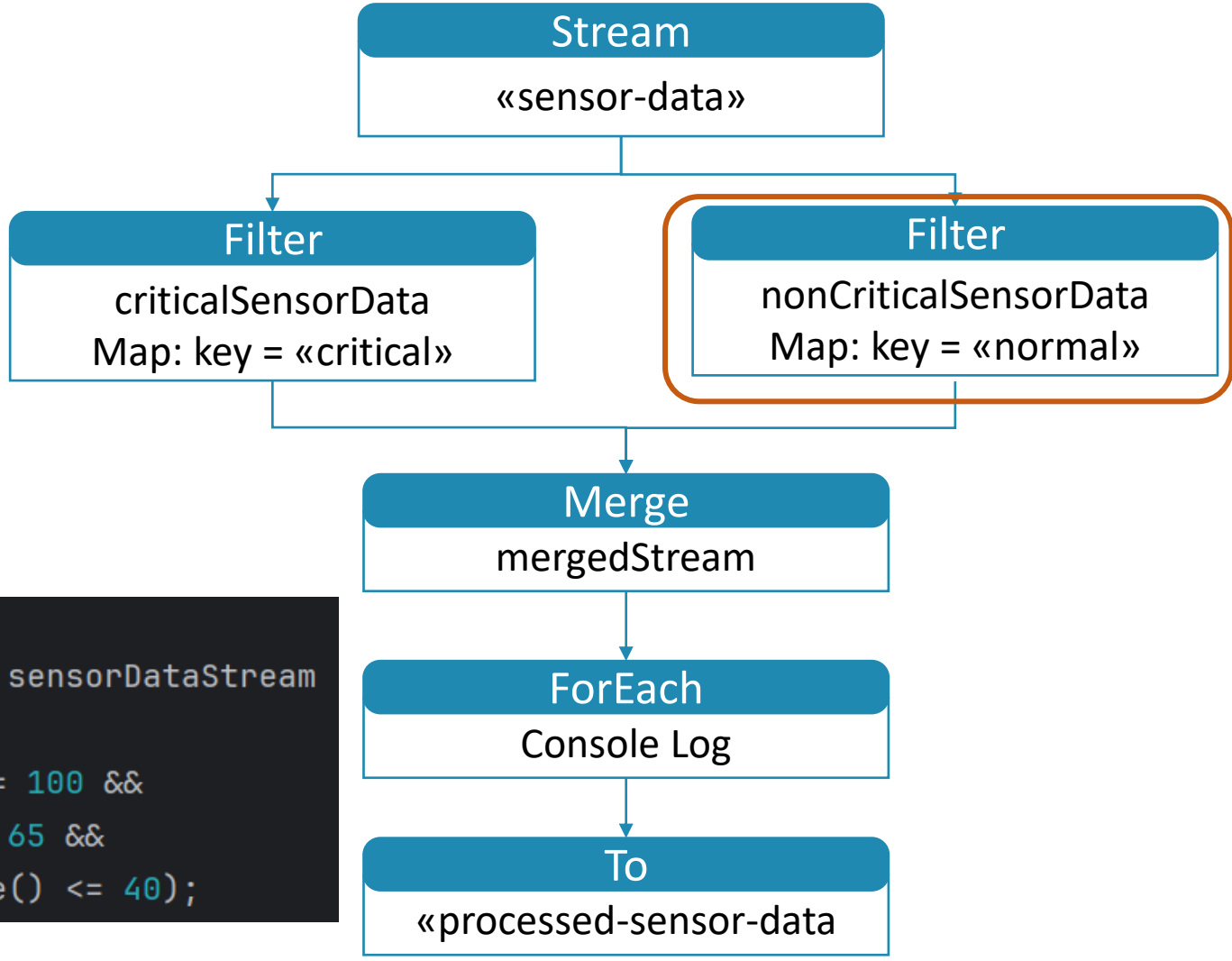
Sensor Data Process Topology




```
// Filter sensor readings based on poor air quality,
// very high humidity, and very high temperature
KStream<String, SensorData> criticalSensorData = sensorDataStream
        .filter((key, sensorData) ->
                sensorData.getIndexedAirQuality() > 100 ||
                sensorData.getHumidity() > 65 ||
                sensorData.getAirTemperature() > 40);
```

```
// Map critical sensor data to have the key "critical"
KStream<String, SensorData> criticalKeyedSensorData = criticalSensorData
        .map((key, sensorData) -> new KeyValue<>("critical", sensorData));
```

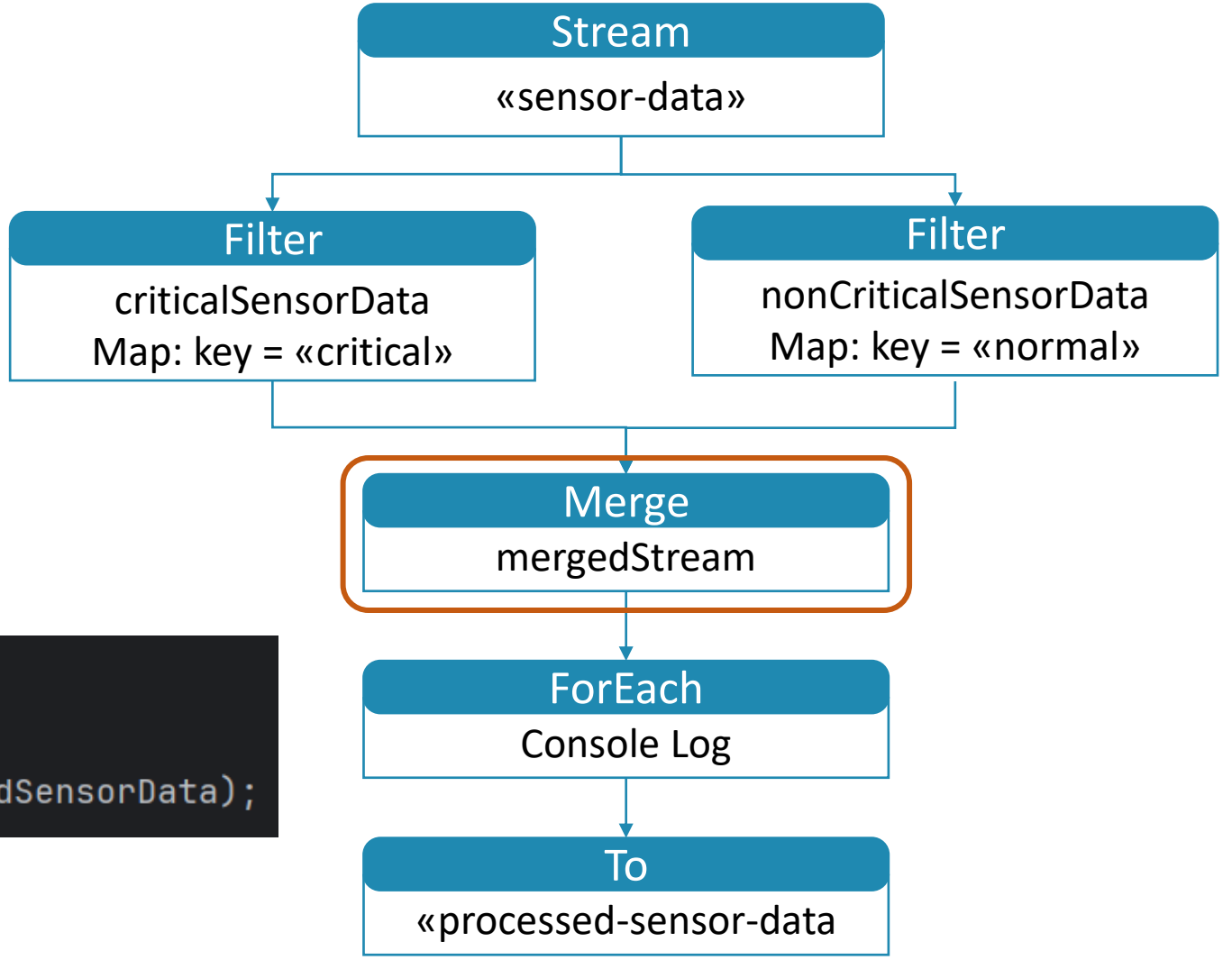
Sensor Data Process Topology




```
// Process the non-critical sensor data
KStream<String, SensorData> nonCriticalSensorData = sensorDataStream
        .filter((key, sensorData) ->
                sensorData.getIndexedAirQuality() <= 100 &&
                        sensorData.getHumidity() <= 65 &&
                        sensorData.getAirTemperature() <= 40);
```

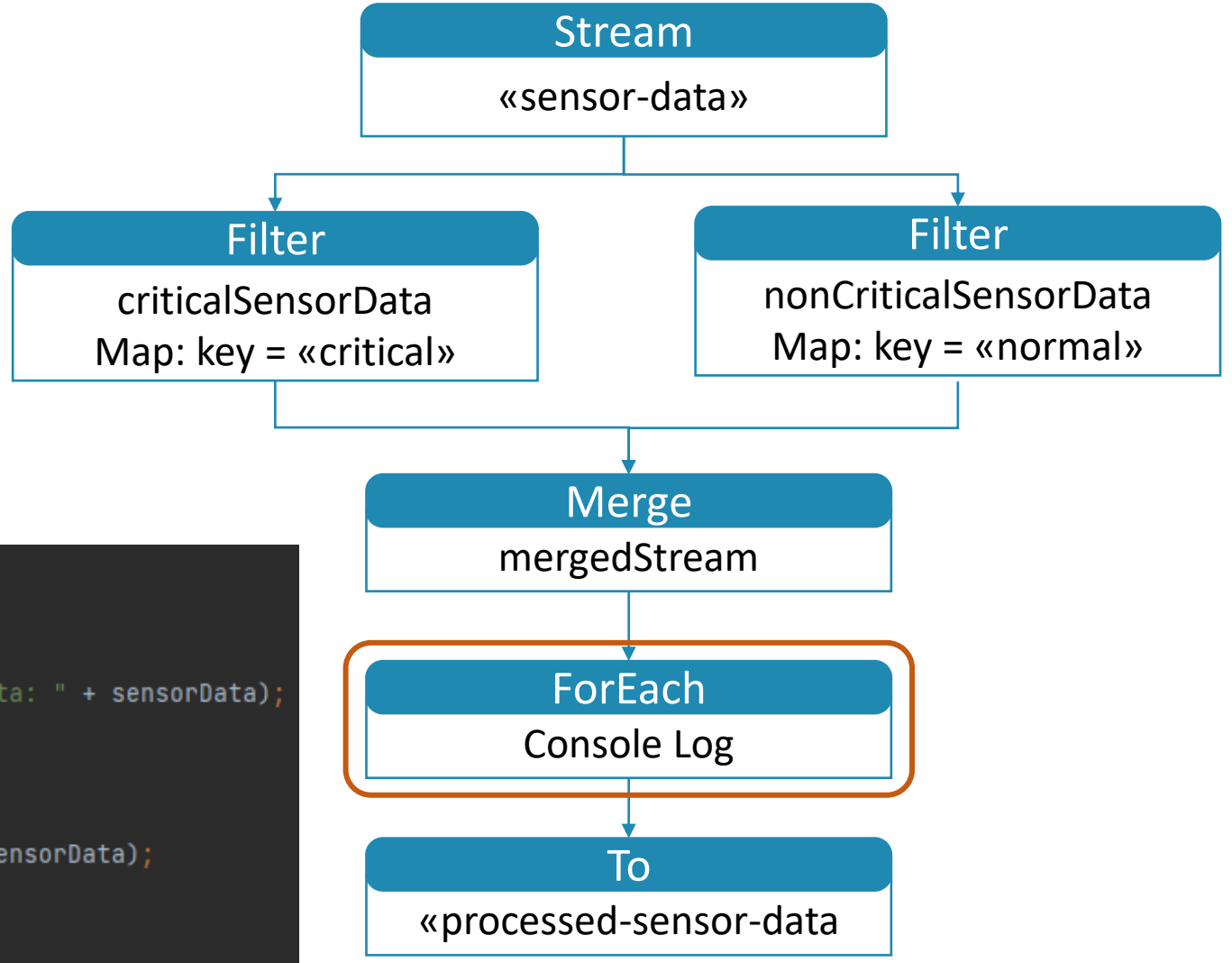
```
// Map non-critical sensor data to have the key "normal"
KStream<String, SensorData> normalKeyedSensorData = nonCriticalSensorData
        .map((key, sensorData) -> new KeyValue<>("normal", sensorData));
```

Sensor Data Process Topology



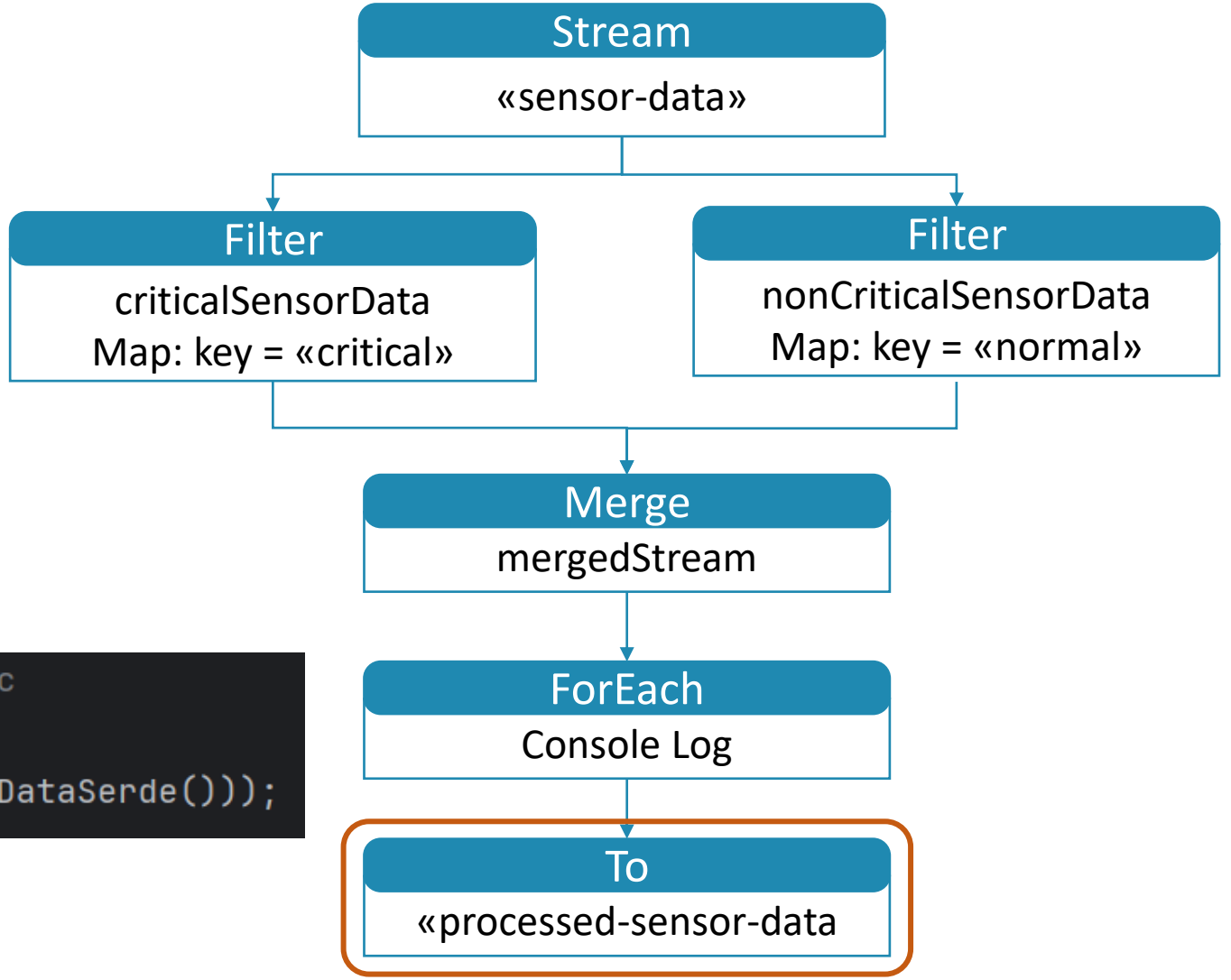

```
// Merge the two streams back together
KStream<String, SensorData> mergedStream =
        criticalKeyedSensorData.merge(normalKeyedSensorData);
```

Sensor Data Process Topology



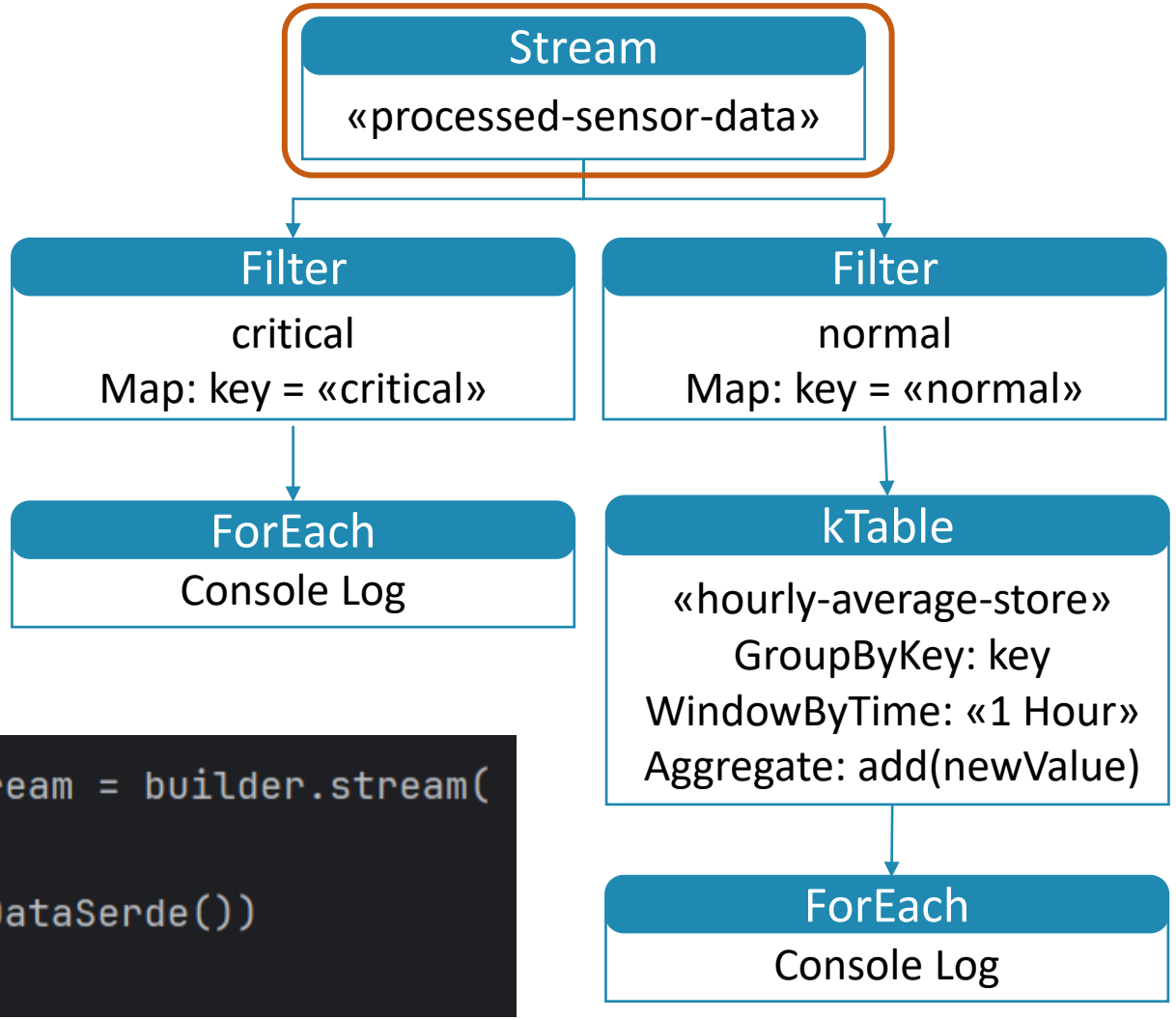

```
// Print the output for debugging purposes
mergedStream.foreach((key, sensorData) -> {
    if (debug) {
        System.out.println("Process-App: Key: " + key + ", SensorData: " + sensorData);
    } else {
        // only print critical sensor data
        if ("critical".equals(key)) {
            System.out.println("Key: " + key + ", SensorData: " + sensorData);
        }
    }
});
```

Sensor Data Process Topology



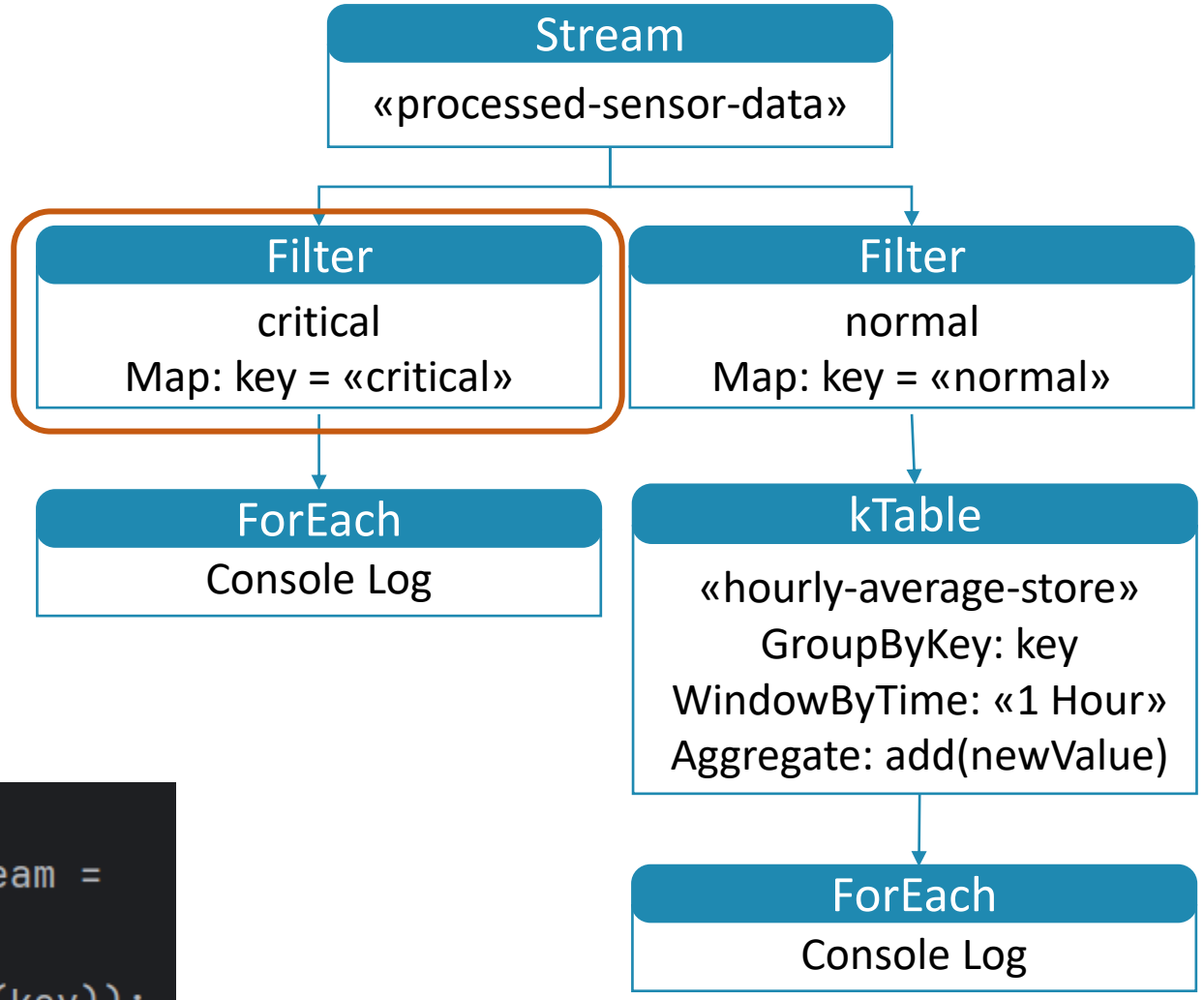

```
// Forward the processed stream to an output topic
mergedStream.to( s: "processed-sensor-data",
        Produced.with(Serdes.String(), new SensorDataSerde()));
```

Sensor Data Monitor Topology



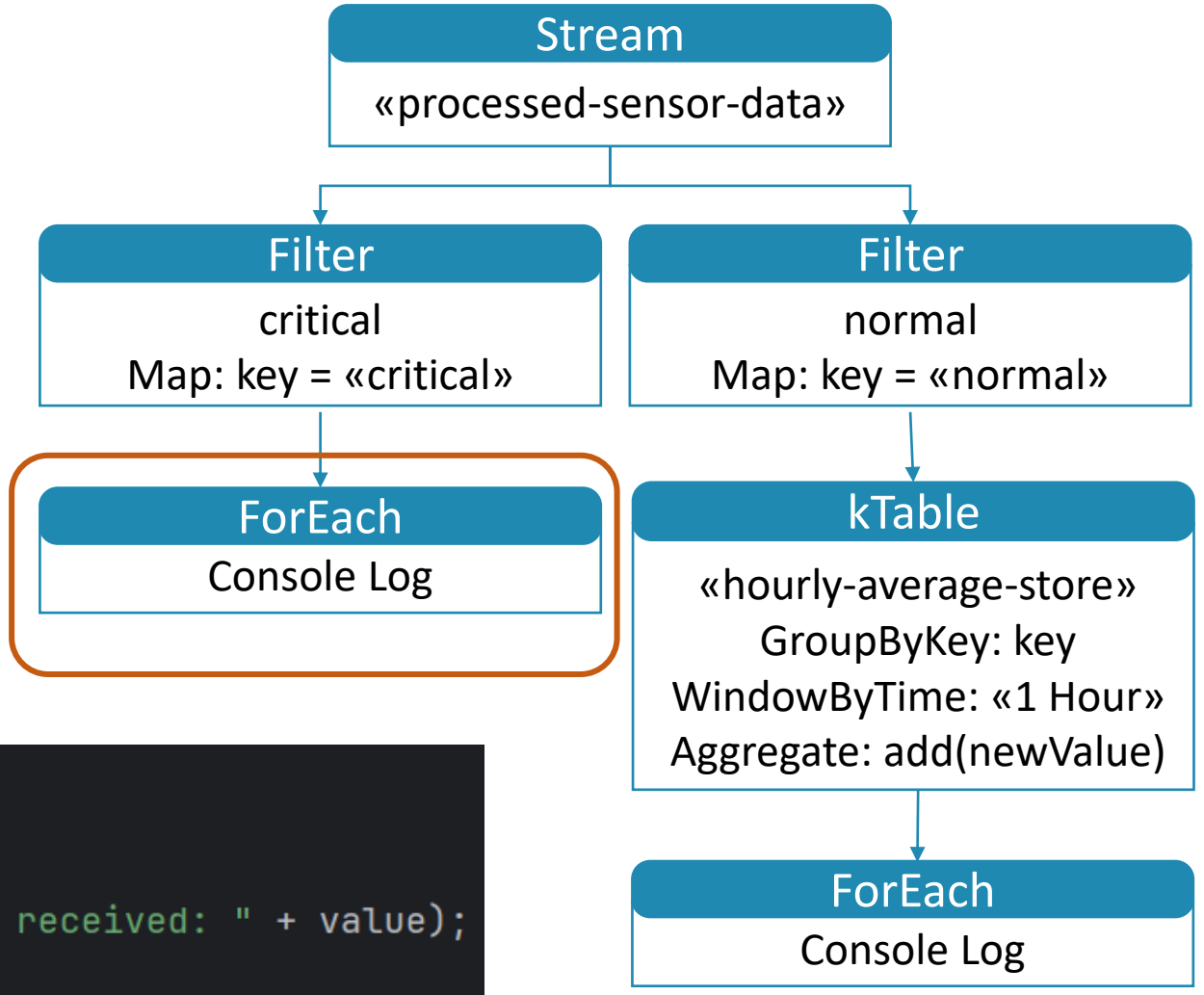

```
KStream<String, SensorData> processedSensorDataStream = builder.stream(
        topic: "processed-sensor-data",
        Consumed.with(Serdes.String(), new SensorDataSerde())
);
```

Sensor Data Monitor Topology



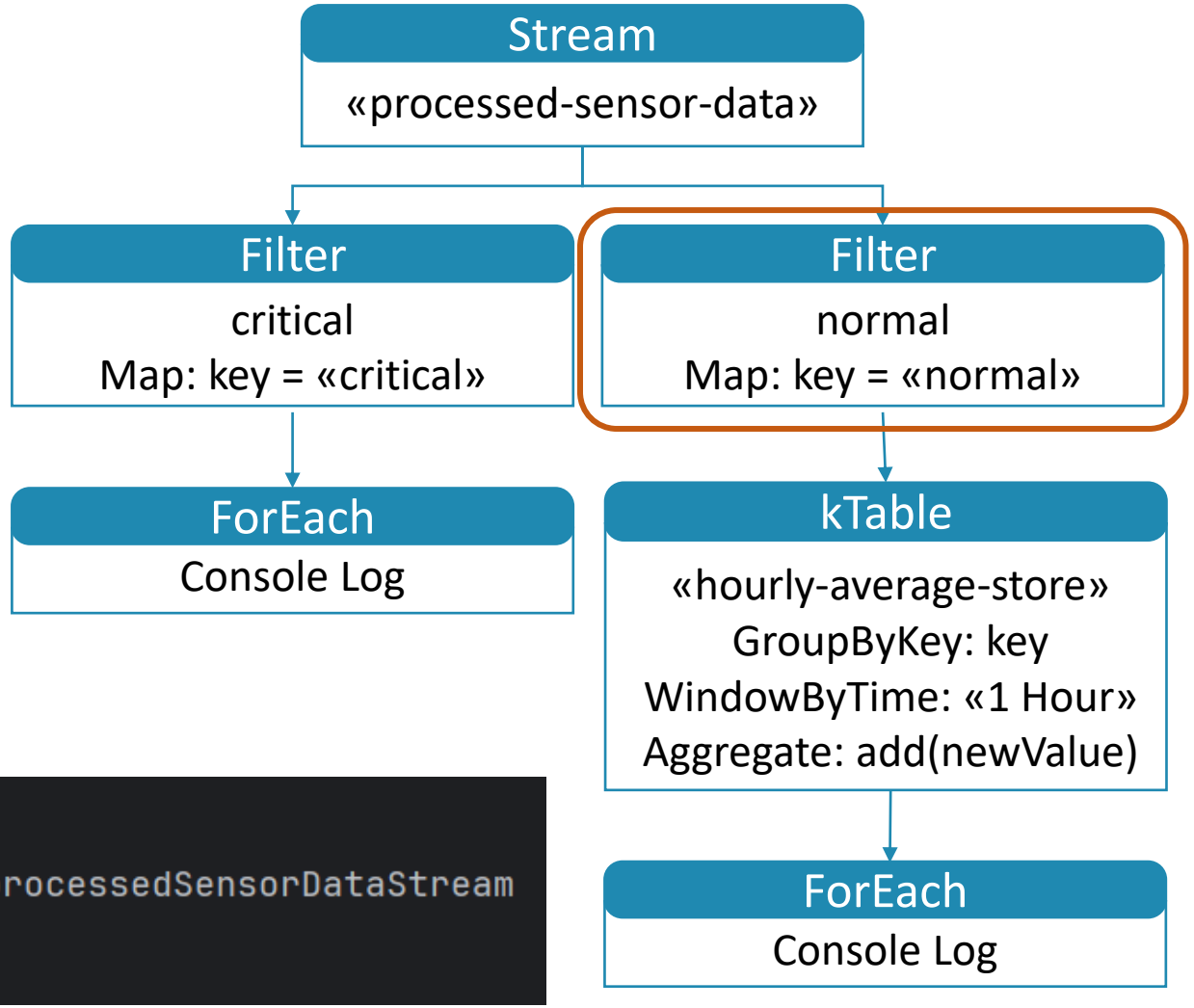

```
// Filter for critical sensor data
KStream<String, SensorData> criticalSensorDataStream =
        processedSensorDataStream
        .filter((key, value) -> "critical".equals(key));
```

Sensor Data Monitor Topology



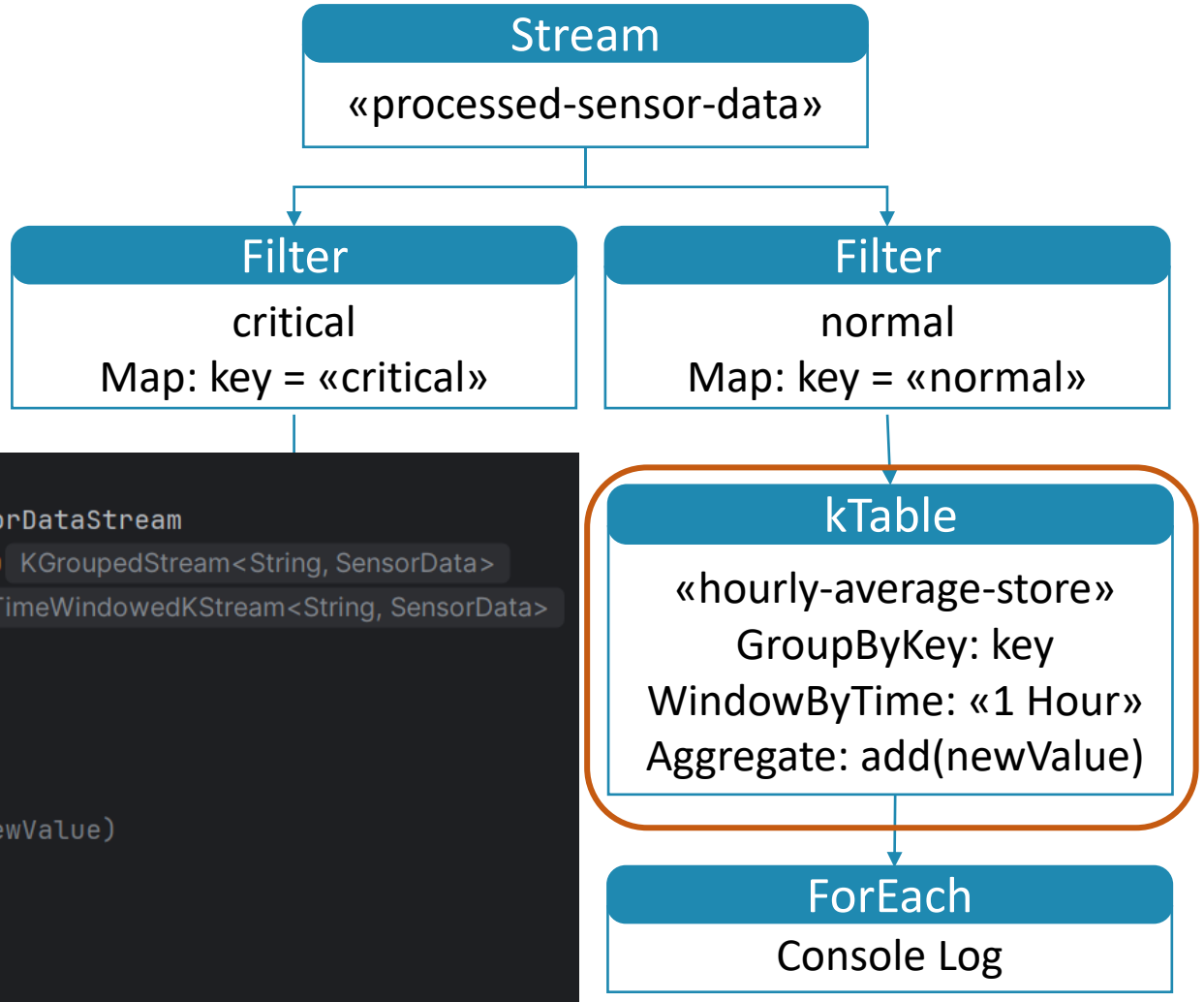

```
// Handle critical sensor data
criticalSensorDataStream.foreach((key, value) -> {
    System.out.println("ALERT! Critical sensor data received: " + value);
    kafkaStreamsService.sendCriticalData(value);
});
```

Sensor Data Monitor Topology



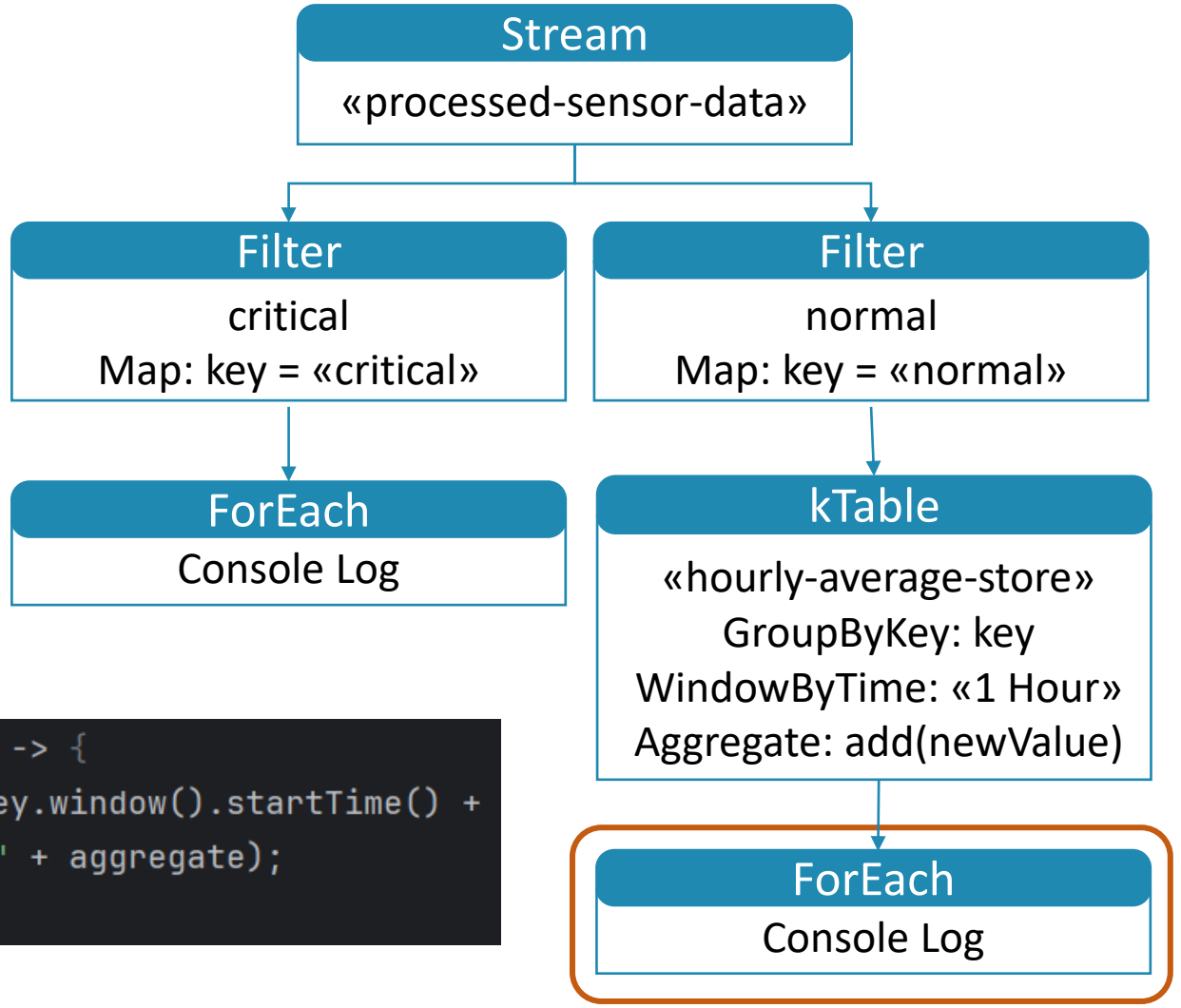

```
// Filter for normal sensor data
KStream<String, SensorData> normalSensorDataStream = processedSensorDataStream
        .filter((key, value) -> "normal".equals(key));
```

Sensor Data Monitor Topology




```
// Calculate average values per hour for normal data
KTable<Windowed<String>, SensorDataAggregate> hourlyAverage = normalSensorDataStream
        .groupByKey(Grouped.with(Serdes.String(), new SensorDataSerde()))  KGroupedStream<String, SensorData>
        .windowedBy(TimeWindows.ofSizeWithNoGrace(Duration.ofHours(1)))  TimeWindowedKStream<String, SensorData>
        .aggregate(
                SensorDataAggregate::new, // Initializer
                (key, newValue, aggregate) -> {
                    try {
                        // Aggregator defines how to add a new record (newValue)
                        // to the existing aggregation
                        return aggregate.add(newValue);
                    } catch (Exception e) {
                        e.printStackTrace();
                        return aggregate; // return the existing aggregate on error
                    }
                },
                Materialized.<~>as( storeName: "hourly-average-store")
                        .withKeySerde(Serdes.String())
                        .withValueSerde(new SensorDataAggregateSerde())
        );
```

Sensor Data Monitor Topology




```
hourlyAverage.toStream().foreach((windowedKey, aggregate) -> {
    System.out.println("Average values from " + windowedKey.window().startTime() +
            " to " + windowedKey.window().endTime() + ": " + aggregate);
});
```

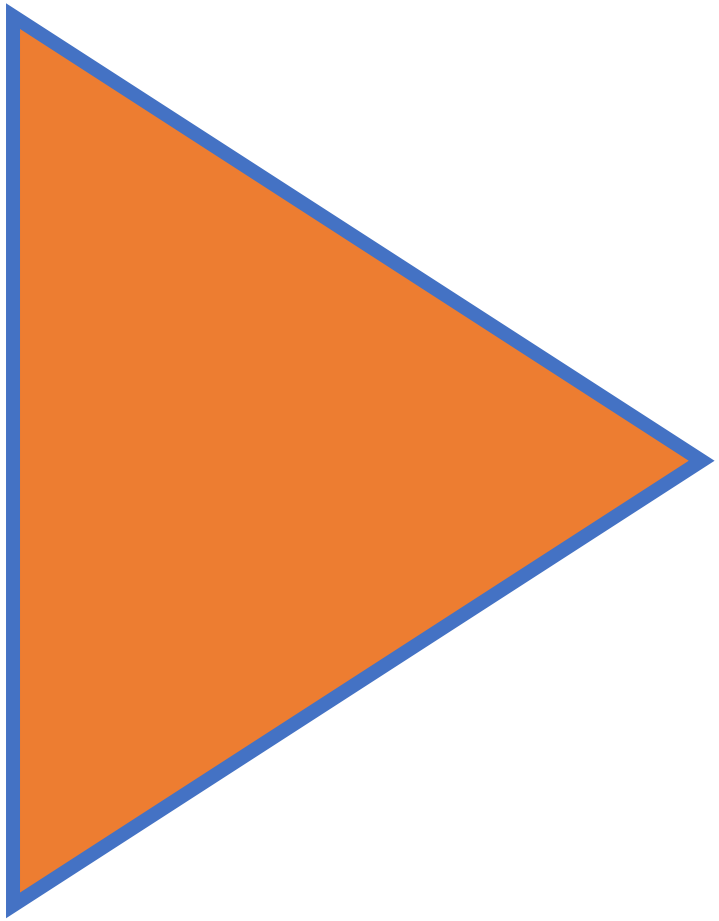
Sensor Data Monitor Topology

```
public ReadOnlyWindowStore<String, SensorDataAggregate> getHourlyAverageStore() {
    return streams.store(StoreQueryParameters.fromNameAndType( storeName: "hourly-average-store",
            QueryableStoreTypes.windowStore()));
}
```

```
ReadOnlyWindowStore<String, SensorDataAggregate> store = kafkaStreamsService.getHourlyAverageStore();
Instant now = Instant.now();
Instant from = now.minus(Duration.ofHours(24)); // Fetch data from the last 24 hours
```

```
// Fetch all entries within the time range
KeyValueIterator<Windowed<String>, SensorDataAggregate> iterator = store.fetchAll(from, now);
while (iterator.hasNext()) {
  KeyValue<Windowed<String>, SensorDataAggregate> entry = iterator.next();
  ZoneId targetTimeZone = ZoneId.of( zoneId: "Europe/Zurich");
  LocalDateTime windowStart = entry.key.window().startTime().atZone(targetTimeZone).toLocalDateTime();
  LocalDateTime windowEnd = entry.key.window().endTime().atZone(targetTimeZone).toLocalDateTime();
  sensorDataList.add(new SensorDataAggregateWithWindow(
          entry.value,
          windowStart,
          windowEnd
  ));
}
iterator.close();
```

DEMO



Additional Learnings

- Java Streams similarity to Kafka Streams
- Cloud Events require additional deserialization step
- Conflicting domain representations when migrating to Avro

Q & A