

# Nifty tool-chain for CQRS application development with read-model projection



Simon Dietschi (ELCA)

Follow



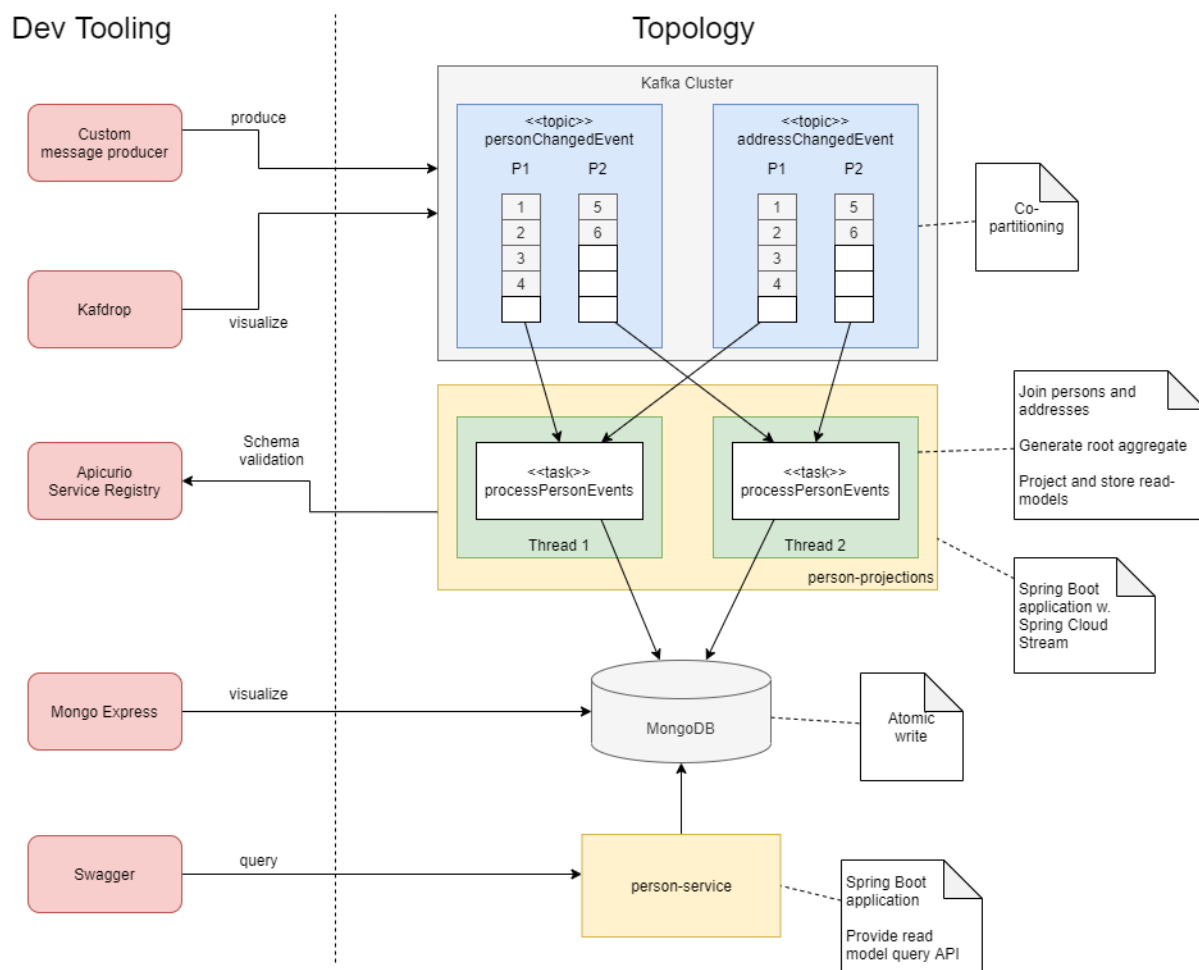
Jan 4 · 4 min read

Event-driven applications are constantly gaining in popularity across industries, especially in areas where large legacy systems provide the core domain of a business, such systems often cannot be replaced, but still, need to interact with modern architectures. To achieve this, a well-known pattern is the command query responsibility segregation (CQRS). Because this pattern adds extra complexity due to the asynchronous behavior, in a recent project here at ELCA we were looking for a good solution on how to support local development, testing, and debugging through a nifty tool-chain.



Photo by [Tom Conway](#) on [Unsplash](#)

We have selected Spring Cloud Stream for the event processing, Kafka as the streaming platform, and MongoDB for storing the read models. In this post, I would like to outline some interesting implementation details based on a condensed scenario focusing on the read model projection part of CQRS.



Illustrative architecture (Image by Author)

## A closer look at the topology

Condensed from the real project, there is one Kafka topic receiving change events as soon as a person record is being modified in the core system, where the particular event is being generated through change data capture. Because the core system stores one or more addresses separately and there is no guarantee that a person has at least one address created at the same time, we need to join the persons and their addresses creating a root aggregate for further read model projections. Also worth mentioning is each event is expected to contain not only the state change but also the full history of previous changes (maybe not exactly by the book).

For those folks being interested, the join method may look very similar to the one below:

```

1  @Bean
2  public BiConsumer<KTable<String, PersonChangedEvent>, KTable<String, AddressChangedEvent>>
3      ProjectorService projectorService,
4      MappingContext mappingContext) {
5
6      return (personChangedEvents, addressChangedEvents) -> {
7
8          personChangedEvents
9              .mapValues(personChangedEvent -> PersonAggregateRoot.empty().apply(personCh
10             .join(addressChangedEvents, ((root, addressChangedEvent) -> root.apply(addr
11             .mapValues(projectorService::project)
12             .mapValues(PersonAggregateRoot::getPersonId)
13             .toStream()
14             .foreach((k, v) -> log.info("Read model updated for key '{}'" and personId '
15     };
16 }

```

BiConsumerConfig.java hosted with ❤ by GitHub

[view raw](#)

As mentioned, every event also contains the history, hence we can optimize and configure the involved Kafka topics to be compacted (no event sourcing needed) — this enables switching from KTables to KStreams in our join method, doing an in-memory conversion to internal KTables where no more physical helper-topics and unnecessary data duplication will need to be created by Spring Cloud Streams.

To boost performance with lightweight scaling, as well as overcoming the latency for message schema validation against the registry and writing the models into MongoDB within the particular worker thread, the co-partitioning feature is being used to have multiple threads processing the person-address joins. Here is a short look into the Spring settings configuration:

```

1  spring:
2    cloud:
3      stream:
4        bindings:
5          processPersonEvents-in-0:
6            destination: person-changed-events
7          processPersonEvents-in-1:
8            destination: address-changed-events
9        default:
10         resetOffsets: true
11         startOffsets: earliest
12       kafka:
13         streams:

```

```
14         binder:
15         configuration:
16             num.stream.threads: 2
17     ...
```

application.yml hosted with ❤ by GitHub

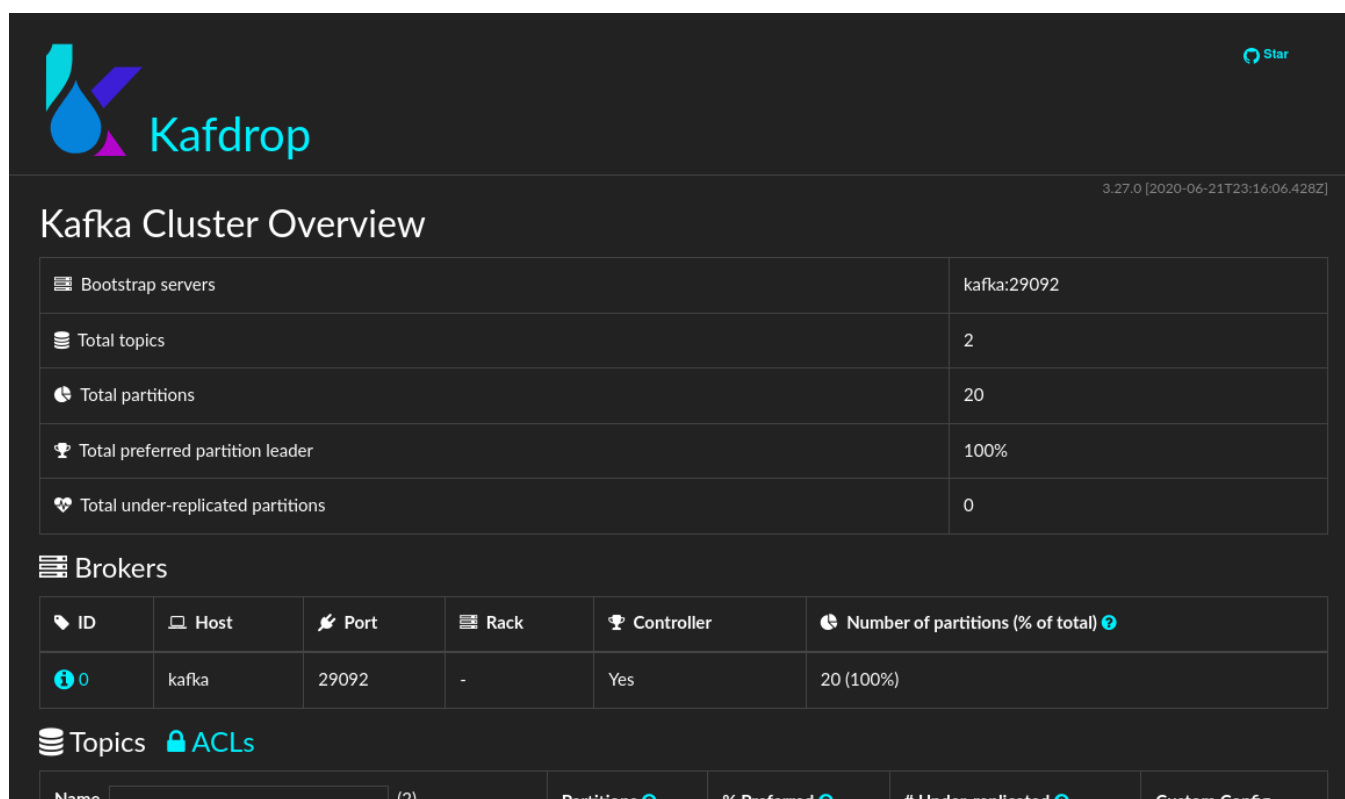
[view raw](#)

## The tool-chain

For the impatient 😊, an example declaration of the tool-chain can be found here: [docker-compose.yml](#)

Coming back to the illustrative architecture view from above — As the change data capture part heavily depends on a large legacy system is reconsidered too expensive to be set up locally, a custom message producer tool has been implemented. Using the same local Avro schema registry, this custom tool integrating the Avro client library allows producing new events locally as well as being flexible in testing changes to the event schemas not to be published to the real registry right now. Concerning the Avro schemas, these are being curated in a separate code repository where for local development a separate maven plugin is being used to upload the schema files into the local registry.

For an easy exploration of the Kafka topics without having to query them per CLI, Kafdrop has been added to the tool-chain. Furthermore, this tool provides useful insights about topic partitions, offsets, events, and much more.



The screenshot shows the Kafdrop web interface for a Kafka cluster. The top header features the Kafdrop logo and a 'Star' button. Below the header, the title 'Kafka Cluster Overview' is displayed with a version and timestamp '3.27.0 [2020-06-21T23:16:06.428Z]'. The main content area is divided into two sections: 'Bootstrap servers' and 'Brokers'. The 'Bootstrap servers' section shows a single entry 'kafka:29092'. The 'Brokers' section is a table with columns for ID, Host, Port, Rack, Controller, and Number of partitions (% of total). It contains one entry for broker '0' on host 'kafka' at port '29092', which is a controller and has 20 partitions (100% of total). Below the 'Brokers' section, there are tabs for 'Topics' and 'ACLs'. The 'Topics' tab is active, showing a table with columns for Name, Partitions, % Preferred, # Under-replicated, and Custom Config. The first row shows a topic with 2 partitions.

Bootstrap servers	
kafka:29092	

Brokers					
ID	Host	Port	Rack	Controller	Number of partitions (% of total)
0	kafka	29092	-	Yes	20 (100%)


Topics				
Name	Partitions	% Preferred	# Under-replicated	Custom Config
(2)				

Topic	Partitions	%1 Referred	# Order Replicated	Custom Config
address-changed-events	10	100%	0	Yes
person-changed-events	10	100%	0	Yes

+ New

Kafdrop with topics (Image by Author)









To be able to verify the read models being projected and stored in the MongoDB, the official Mongo Express tool is being used, offering interesting document and query statistics, but also a wide range of other useful features making it very easy to analyze and debug stored information.


Mongo Express
Database ▾

## Mongo Express

Databases

+ Create Database

 View	admin	 Del
 View	config	 Del
 View	local	 Del
 View	read-model-projections	 Del

Server Status

Hostname	3ece972fceca	MongoDB Version	4.4.3
Uptime	978 seconds	Server Time	Sun, 12 Dec 2021 23:29:01 GMT
Current Connections	4	Available Connections	838856
Active Clients	0	Queued Operations	0
Clients Reading	0	Clients Writing	0
Read Lock Queue	0	Write Lock Queue	0
Disk Flushes		Last Flush	
Time Spent Flushing	ms	Average Flush Time	ms
Total Inserts	0	Total Queries	5
Total Updates	0	Total Deletes	0

Mongo Express (Image by Author)

Another may be rather small but useful detail to mention, is about the creation of the initial topics in Kafka — so once the local infrastructure is up and running, a simple

text file containing the needed topic configurations can be used to declare these topics as compacted:

```
1 # Explanation: topic name, partition count to be a multiple of the number of stream thro
2 person-changed-events:10:1:cleanup.policy=compact
3 address-changed-events:10:1:cleanup.policy=compact
```

topics.txt hosted with ❤ by GitHub

[view raw](#)

And last but not least, the Makefile to specify the needed helper command for topic creation:

```
1 #!make
2
3 KAFKA_BOOTSTRAP_SERVER=localhost:9092
4 KAFKA_TOPICS_FILE=./config/kafka/topics.txt
5
6 kafka-topics-create:
7     awk -F':' '{ system("kafka-topics.sh --create --bootstrap-server ${KAFKA_BOOTSTRAP_
8         ${KAFKA_TOPICS_FILE}
9
10 .PHONY: kafka-topics-create
```

makefile hosted with ❤ by GitHub

[view raw](#)

## Conclusion

Even if this blog post contains project-specific aspects and adoptions, it may still provide a good starting point for your own streaming application development. However, regarding testing and debugging, unit and integration tests often are not enough. Due to the complexity of asynchronous messaging, a supporting tool-chain is essential. Furthermore, such a tool-chain could also be deployed into one of the available container orchestration platforms, like OpenShift which is widely used at ELCA.

For those folks looking for a working demonstrator, while not caring too much if it is Java or Kotlin, please have a look at my [read-model-projections](#) example.

Java

Cqrs

Event Driven Architecture

Kafka

It Architecture