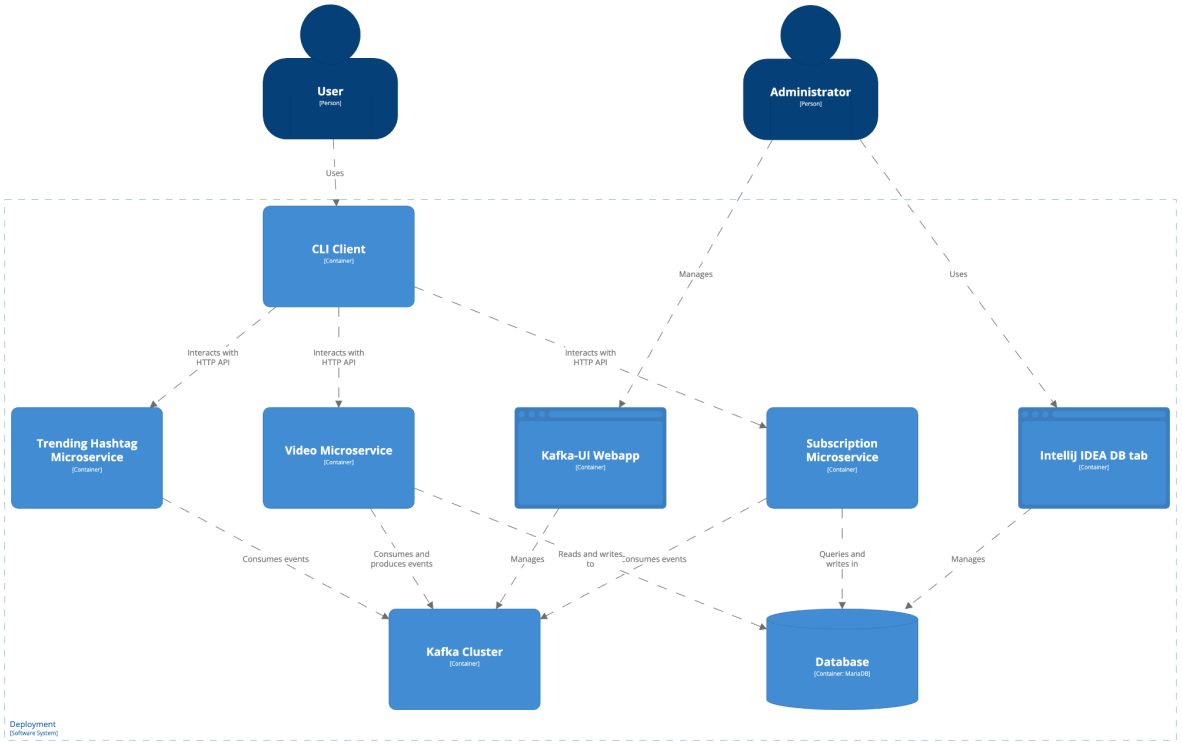
2.1.1 Architecture



High level architecture consists of three microservices with one replica each, communicating with three Kafka brokers and a single instance of MariaDB. While later clarifications by Antonio have underlined that we were expected to use a separate DB per microservice, this was unfortunately discovered by the author too late to implement (and the assessment spec doesn't indicate this). To the author's defence, a single DB minimises complexity and is a legitimate microservice pattern [1]. Additionally, in production scenarios, dedicated DevOps/DB Admin staff may be responsible for solving tricky problems like sharding the DB or ensuring reliable backups and fault-free schema migrations which fall on the Ops side and outside the purview of software engineering.

Aside from this, the user communicates with the microservices by way of a CLI client which accesses them via their HTTP API. The administrator manages Kafka via the Kafka UI brought up as part of the docker compose. Minimal manual interaction with the DB was necessary, so the built in DB interaction features from the IDE sufficed for those purposes.

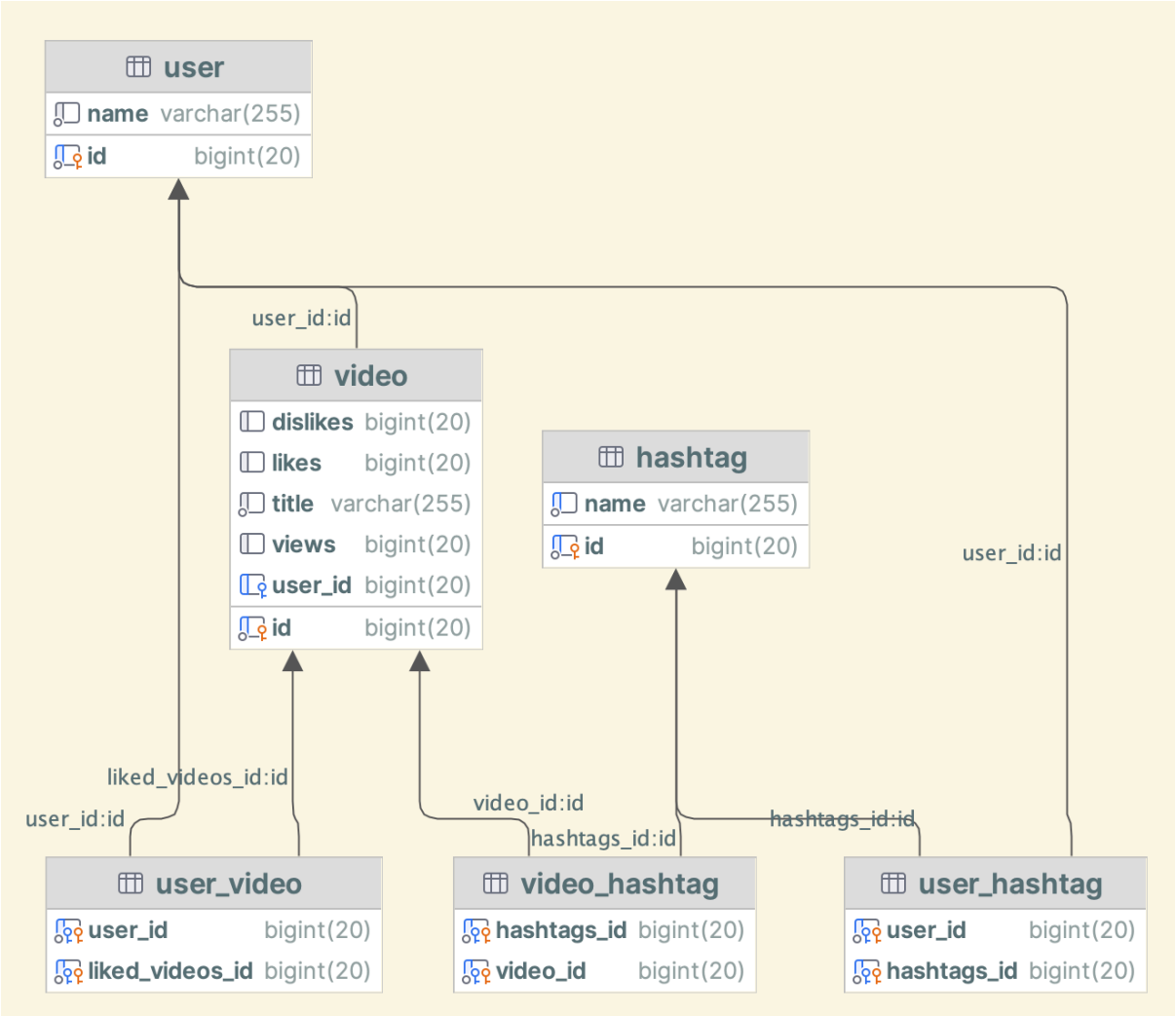
A detailed discussion of how this architecture can scale would need to touch on the individual microservice implementation and is thus further advanced in the next section. In general, we can increase the number of replicas for each microservice to handle additional demand, assuming that 1) we can load balance the requests between the microservices (addressed in 2.1.3), and 2) we do not offload heavy lifting to the poorly scalable RDBMS - which we don't, thanks to leveraging Kafka streams processing (discussed in detail in 2.1.2).

Kafka itself is inherently scalable. While our deployment features 3 brokers, we could increase the number of brokers to cope with higher demand. Additionally, we could increase

the number of partitions per topic up from our current setting of 6, allowing us to spread data processing across a greater number of microservice replicas.

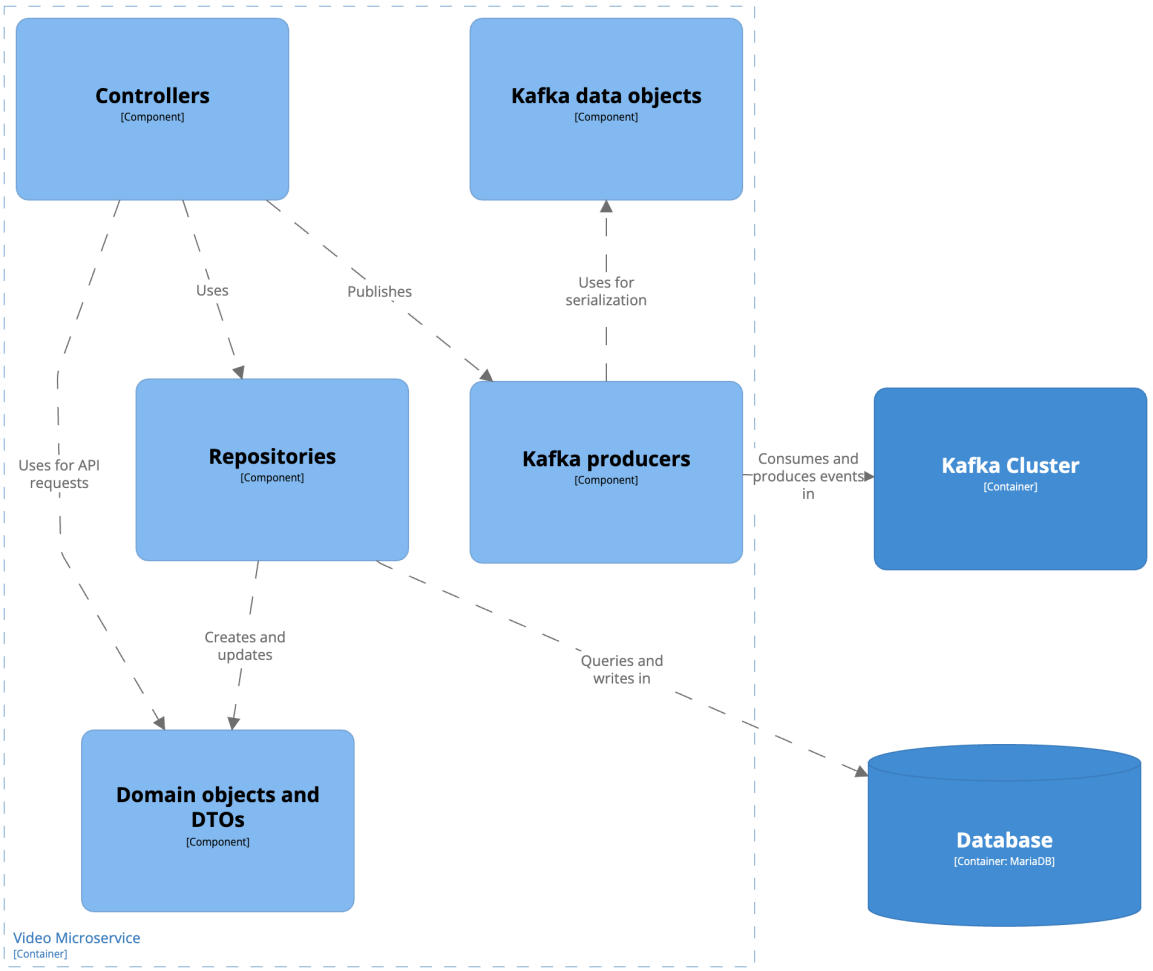
New requirements could be enabled by rolling out a separate microservice to fulfil them, thereby not impacting the functionality of the current microservices. CLI client would need updating to interact with the new service. Established Kafka topics could either be transformed within the service to accommodate its needs, or the topic schema could be adjusted across the microservices. In the latter case, adding something like the Kafka Schema Registry [2] could be very helpful to ensure error free migrations in line with new requirements.

For completeness, included below is a representation of the database schema used throughout our microservice deployment:

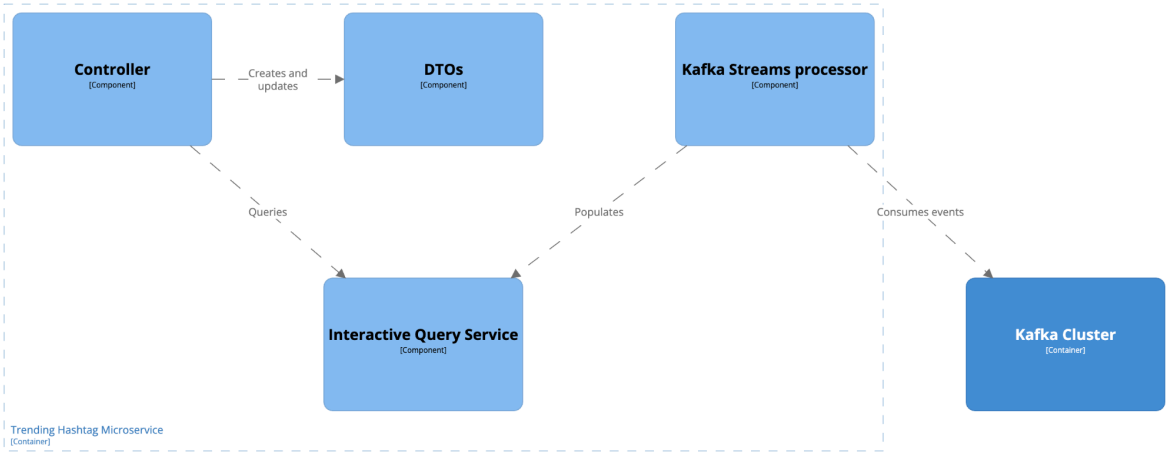


2.1.2 Microservices

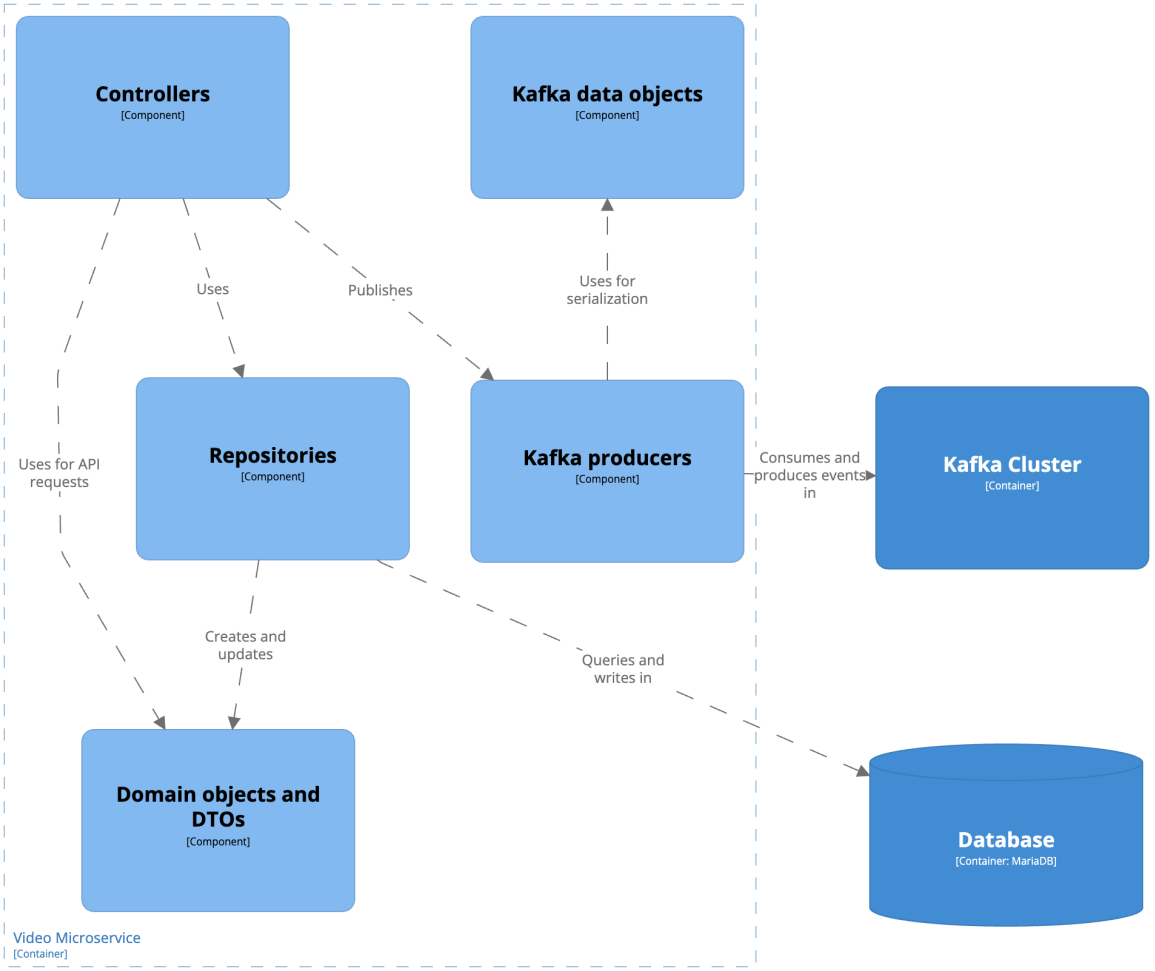
The microservice deployment consists three separate microservices, detailed below:



Video microservice facilitates all CRUD operations apart from (un)subscribing to hashtags. As such, it is heavily reliant on database reads and writes. The issues arising from this will be discussed later in this section. It is also the major publisher of Kafka events.



Trending hashtag microservice is very lightweight: it does not interact with the database at all, instead relying on Kafka stream processing to build trending lists on the fly.



Subscription microservice has minimal interactions with the database, updating user subscription lists and reading in lists of uploaded videos to populate new subscriptions. Like THM, it makes significant use of Kafka stream processing to build video suggestions dynamically.

The use of Kafka streaming benefits scalability significantly: deploying additional replicas of the microservices would allow each one to deal with a fraction of the data volume, as partition assignment would be spread between them. Each one would have access to a full set of computed results, since the interactive query service accesses a GlobalKTable that stores locally the final results of our stream processing.

RDBMs are difficult to scale horizontally and frequently still have a single write master even when they are sharded. Hence, services such as VM that commit many writes could encounter row lock contention under heavy use. We could rely further on inherently scalable Kafka brokers by doing away with DB writes and using JDBC Sink Connector[3] for persisting the streams, and minimising DB reads by serving data directly from Kafka stores, as is possible due to the stream-table duality [4].

2.1.3 Containerisation

Docker containers for the system are built using Micronaut's included Gradle tasks under build/dockerBuild. A docker-compose.yml file is provided for orchestrating the lifecycle of the container images together with the Kafka brokers and the MariaDB instance that they rely on.

As the microservices depend on their Kafka topics being created to function, an init container was added that brings up an additional broker instance in order to orchestrate these topics' creation. It consists of a simple shell script as its entrypoint, which uses the built-in Kafka shell script for the creation of topics. An important note is that Docker compose is not sophisticated enough to allow waiting for this init container to successfully terminate before launching the microservices. To mitigate this, a shell script was written that runs docker compose, waits for the topic creator to finish, then re-runs the compose to bring back the microservices which have terminated due to no topics available.

The usage of docker containers allows scalability by letting us bring up multiple independent copies of the microservices. Unfortunately, to facilitate error recovery or autoscaling, we would need a more advanced form of container orchestration such as Docker Swarm, Kubernetes, or other similar systems such as Nomad. These could monitor the containers' health check endpoints, re-creating any failed containers, as well as autoscaling the deployment based on resource usage telemetry.

In order to spread the user load across multiple containers, we would need another solution such as a reverse proxy like NGINX or Traefik, or a Kubernetes load balancer/ingress. This would allow balancing user requests across multiple microservice replicas.

Nevertheless, since THM & SM perform most of their data processing using Kafka Streams, the services are already scalable to a large degree without load balancing. The data processing they perform would scale with the number of replicas, so long as it is smaller or equal to the number of topic partitions. Each instance would then process the events only from the partition it was assigned, even if all HTTP requests are directed to a single instance.

2.1.4 Quality Assurance

Integration testing was carried out to verify correctness of important behaviours of the trending hashtag and subscription microservices. To enable this, two testing scripts were created under microservices/integration_testing, trending_integration.sh and watchlist_integration.sh. An example of the operation of both is shown below:

Trending hashtags:
1. *Re-create the environment by running compose down, then compose up*
2. *Setup a test user and upload 15 videos:*
$ ./trending_integration.sh setup 15
Creating video 1
…
Creating video 15

3. *Apply a random amount of likes between 1 and 10 to all 15 videos:*
$ ./trending_integration.sh like 10 15
Video 1 has 5 likes
Video 2 has 7 likes
Video 3 has 5 likes
Video 4 has 6 likes
Video 5 has 6 likes
Video 6 has 7 likes
Video 7 has 7 likes
Video 8 has 7 likes
Video 9 has 3 likes
Video 10 has 9 likes
Video 11 has 10 likes
Video 12 has 7 likes
Video 13 has 6 likes
Video 14 has 1 likes
Video 15 has 4 likes

4. *Wait 10s for the data to process via KStreams and query the trending endpoint:*
$ ./trending_integration.sh trending (minified below, but usually would be pretty-printed via jq):
{"top10Hashtags":[{"id":11,"likes":10,"rank":1},{"id":10,"likes":9,"rank":2},{"id":2,"likes":7,"rank":3},{"id":6,"likes":7,"rank":4},{"id":7,"likes":7,"rank":5},{"id":8,"likes":7,"rank":6},{"id":12,"likes":7,"rank":7},{"id":4,"likes":6,"rank":8},{"id":5,"likes":6,"rank":9},{"id":13,"likes":6,"rank":10}]}

5. *Check that a) we have a max of 10 videos returned b) the videos are ranked in order of the likes applied c) correct number of likes is registered.*
This test would also be redone with smaller window sizes to judge the effect of windowing on the results.

Watchlist (next recent videos to watch for hashtag):
1. *Re-create the environment by running compose down, then compose up*
2. *Set up a test user and add some videos with hashtags:*
$ ./watchlist_integration.sh setup
$ ./watchlist_integration.sh post2 1 2 # This creates a video w/ hashtag1 & hashtag2
$ ./watchlist_integration.sh post2 1 2
3. *Subscribe to one of the hashtags:*
$ ./watchlist_integration.sh sub 1
4. *Check the watchlist:*
./watchlist_integration.sh list 1

```
{
  "newVideos": [
    2,
    1
  ]
}
```

5. *Watch one of the videos:*
$ ./watchlist_integration.sh watch 1
6. *Verify that the video is off the watchlist:*
$ ./watchlist_integration.sh list 1

```
{
  "newVideos": [
    2
  ]
}
```


This testing would be further done to make sure that proper behaviour is exhibited, with the number of videos uploaded, hashtag matches, etc. One discovery made through this testing was that due to the implementation of KStreams & GlobalKTable queries, unsubscribing merely pauses the updates to the watchlist: the stale version can still be retrieved. This was deemed not to be an issue for the functionality, although a production version of the app would need some way of trimming old subscriptions.

Unit Testing
Unit testing proved unfeasible due to the domain design relying on separate generator tables. Exception of the following type were encountered:

Error executing DDL "drop sequence `video_id_generator`" via JDBC [(conn=53) Unknown SEQUENCE: 'videos.video_id_generator']

This behaviour also remained when mocking out repositories via Mockito.

Correcting this would involve refactoring the code, as well as breaking the integration tests that have served well in developing the KStreams aspects of the microservices. Had the deadlines allowed this, some solution would need to be implemented, as unit tests are invaluable for fixing the behaviour of the code in place and thus preventing breaking changes during further development.

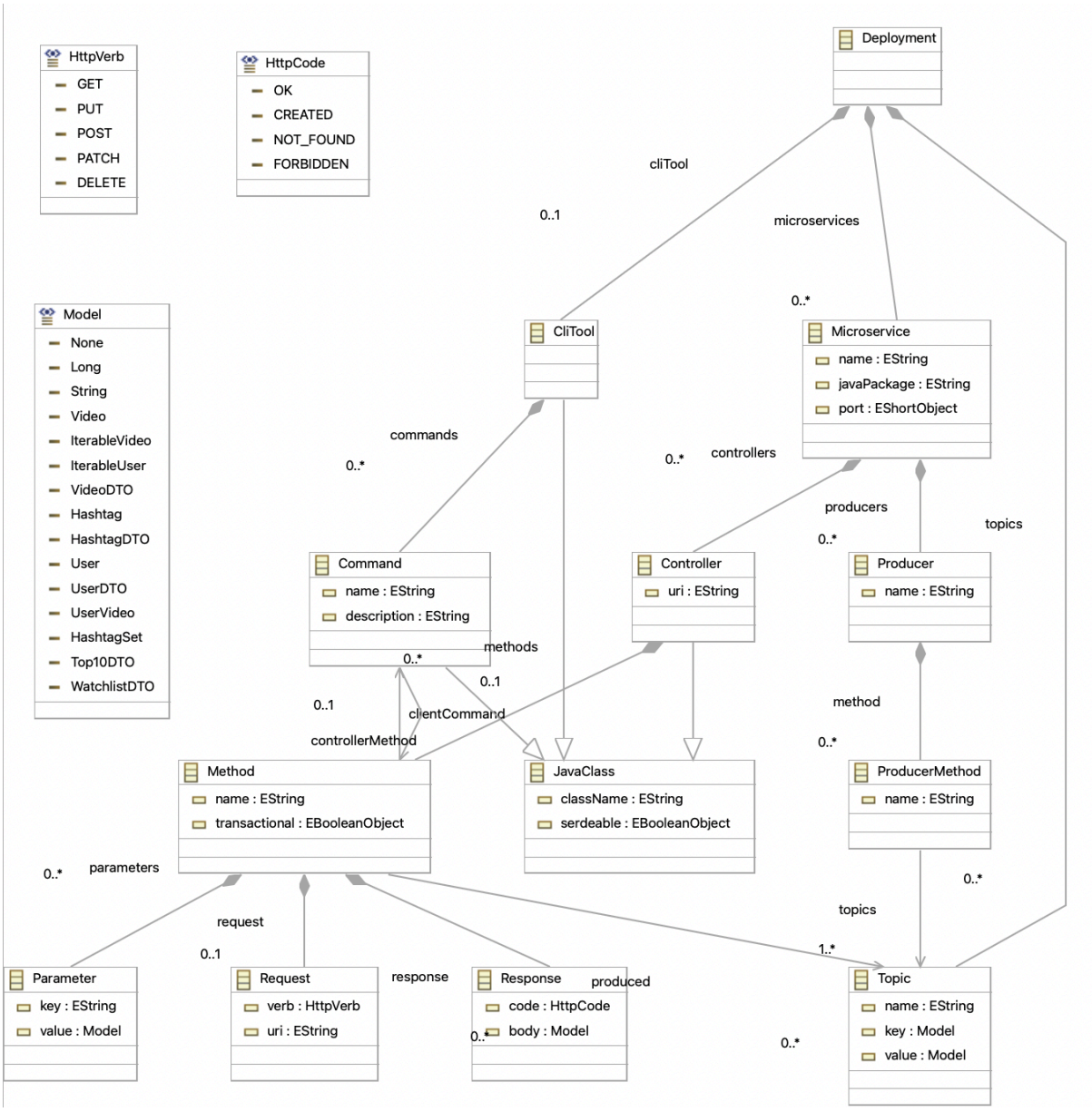
Docker vulnerability scanning

Microservice images were scanned using Docker Scout. A high severity vulnerability was found in all three containers affecting the logback package. After pinning the fixed version in build.gradle, reloading Gradle, and rebuilding the docker images, Docker Scout no longer reported this issue.

The rest of the vulnerabilities were medium and low priority. Majority did not yet have fix versions reported. Some, like the Sqlite CVE, would have required manually specifying the Dockerfile and changing layers to newer versions.

As the severity was less critical, and custom docker files would require specifying our own new Gradle tasks to build its image, and informing the user (reviewer) of how to go about this, the choice was made to allow these issues to remain.

2.2.1 Metamodel



The metamodel was chosen with regard to maximising its utility for code generation, as well as to allow us to visualise the possible behaviours of the microservice controllers in the graphical concrete syntax.

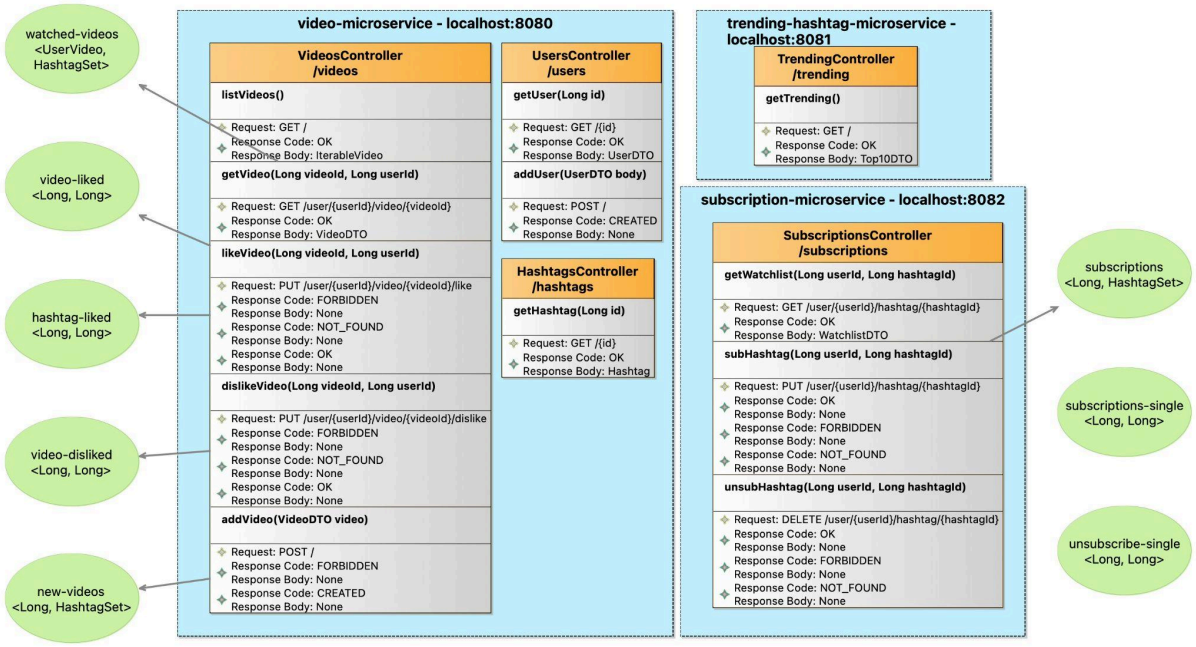
A conscious choice was made not to be all-encompassing in the modelling of the Kafka topics. I included all topics that figure within the microservice controllers; many more are present for the purposes of Kafka streaming and supplying global state stores. As these do not help elucidate the elements captured in the model, they were omitted from representation.

Metamodel was specifically crafted to enable model-to-text transformation for generating controller methods and the CLI tool functionality, as these two have a direct 1-1 correspondence between their methods.

Originally, it was also planned to enable modelling entity relations for the database layout. Indeed the code for this exists within the metamodel, although it is not instantiated in the concrete model. This was omitted for three reasons:

1. The database model is simple, consisting only of users, videos, hashtags, and the relationships between them. Not a lot was gained from modelling such a simple schema.
2. The database model is mainly driven by the code we write in Micronaut, rather than choosing a model and writing the code around it. Again, this was straightforward given the limited use made of the database for data processing.
3. IntelliJ IDEA, the author's IDE of choice, allows generating a reasonable graphical representation of the database tables, as seen in 2.1.1. This also reflects the reality directly, as it is taken directly from the DDL representations within MariaDB.

2.2.2 Graphical Concrete Syntax



Primary aim of the chosen syntax was to clearly illustrate the compositional relationship of the primary “moving parts” of the microservice controllers without overwhelming the viewer with information.

Microservice containers are coloured blue to reflect their somewhat ephemeral nature, given we do not directly interact with them (other than running docker compose), but with the code in the controllers.

Controllers are coloured orange to contrast and emphasise them within the chosen syntax, as they are the important functional units. Each controller is created as a container so as to hold info on its methods and their various parameters, which are displayed in a vertical stack reminiscent of the actual code within their Java files.

Controller methods were coloured grey to facilitate easy reading of their constituent members. The method signatures are bolded while the requests and responses are enumerated with bullet points (label icons) to further reinforce that they belong to the method above, like a bulleted list. It would’ve been preferable to remove the separating line between the method signature and the requests/responses, however I couldn’t find a way of rendering this within Sirius.

The detail provided about the methods allows the viewer at a glance to reason about valid response codes from each method endpoint: any other received response would indicate a malfunction in the code/microservice.

Topics were coloured green and rendered as ovals to keep them thematically separate from the microservice elements which are concrete parts of our implementation, rather than managed by Kafka outside the microservices. They were labelled with their key/value types

in a form instantly recognisable to anyone dealing with Java types. Arrows were drawn between the controller methods and the topics they publish to as part of their functionality. Unfortunately, a bug in Sirius resulted in some of the arrows on the right hand side steadfastly disappearing despite the many attempts to get them rendering. This is further illustrated in the Appendix A.

Some fairly intricate (at least for a beginner) AQL was needed to enable neatly printing labels consisting of multiple elements. The display of topic key/value types is one example. Another is the joining of HTTP verbs with the URI. It was a worthwhile effort, as it maintained legibility and salience for graphical syntax purposes, while still enabling enough flexibility for the metamodel to accommodate powerful code generation detailed in section 2.2.4.

While CLI tool also figures in our model and metamodel, it was not graphically rendered: it is effectively invisible to the implementation, being fully functional with generated code, without alterations other than sorting the odd missing import, which the IDE can handle automatically.

2.2.3 Model Validation

It is the author’s deep regret to state that model validation was omitted due to severe time constraints in implementing this assignment.

2.2.4 Model-to-Text Transformation

The 1-1 relationship between controller methods, and CLI client commands and HTTP client interfaces was chosen to be the focus for the model to text transformation work. Thanks to the model supplying sufficient information on the methods, their parameters, and their return types, I managed to fully generate:

- All method signatures for the microservice controllers
- All of the CLI HTTP client interface
- All of the necessary Micronaut annotations
- All CLI subcommands, along with their wiring to the main CLI command class
- Most of the necessary import statements for the classes and annotations involved

The fact that the CLI tooling can be fully generated and fully functional is a tremendous boon to productivity and a real credit to the power of model to text transformations.

For the microservice controller generation, it was chosen to rely on protected sections to enable adding hand written code. These were restricted to clear, logical areas: the list of controller's member fields at the top of the class, and the actual bodies of the controller methods.

While extension is generally preferred, and controller logic is often moved into a separate Service package (thereby turning MVC into MVCS), I chose to keep things simple: separating out a Service layer would mean coding it into the metamodel at a marginal benefit. Subclassing meanwhile would prevent us from generating the Micronaut annotations via M2T. Ultimately, this also fit well by allowing direct copy of the old, handwritten controller code within the protected sections. To the author's great surprise, nothing broke and there were no surprises, only business as usual.

One issue that was encountered (perhaps a skill issue?) lied in the behaviour of protected sections: it appeared that somewhat randomly, they would inhibit updates to the method signatures that laid directly above the protected sections resulting from updates to the concrete model to accommodate changes in data types. Sometimes, manual reconciliation of these changes was necessary (which would then not be overwritten, as the code was being made current with the modelled state).

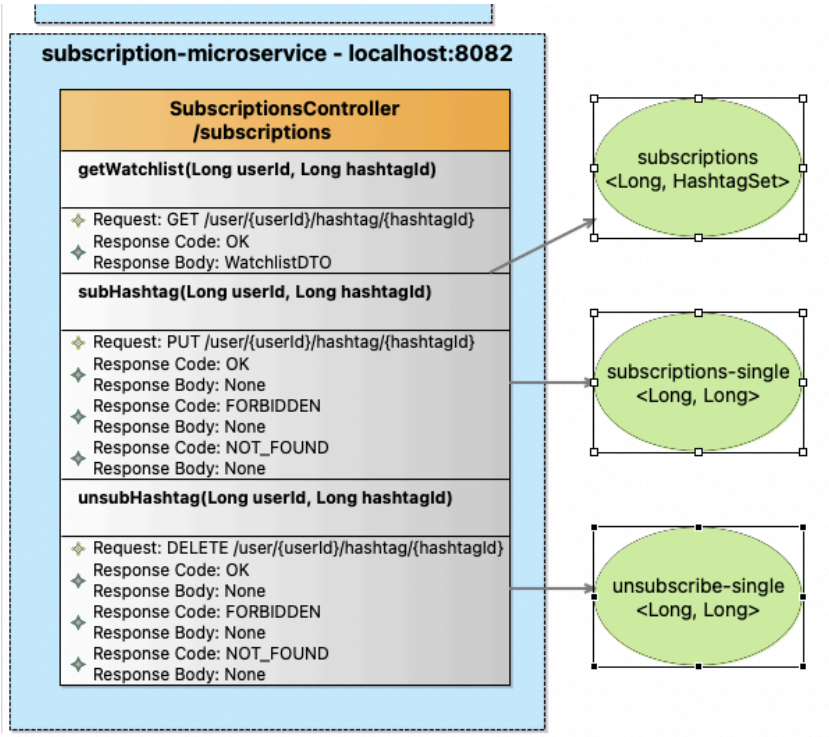
Implementing code generation for Kafka domain beans was considered. However, this would then entail modelling out the entirety of the Kafka topics for completeness, some of which were only used intermediately for Kafka stream processing. Given all Kafka streams code is handwritten, this part of code generation was omitted for simplicity and clear separation between handwritten and generated parts of the codebase.

References

[1] C. Richardson, “Microservices Pattern: Shared database,” *microservices.io*. https://microservices.io/patterns/data/shared-database.html (accessed Jan. 20, 2024).
[2] “Schema Registry Overview | Confluent Documentation,” *docs.confluent.io*. https://docs.confluent.io/platform/current/schema-registry/index.html (accessed Jan. 20, 2024).
[3] “JDBC Sink Connector for Confluent Platform | Confluent Documentation,” *docs.confluent.io*. https://docs.confluent.io/kafka-connectors/jdbc/current/sink-connector/overview.html (accessed Jan. 20, 2024).
[4] M. Noll, “Streams and Tables in Apache Kafka: A Primer | UK,” *Confluent*, Jan. 13, 2020. https://www.confluent.io/en-gb/blog/kafka-streams-tables-part-1-event-streaming/ (accessed Jan. 20, 2024).

Appendix A

Below is an illustration of how Eclipse Sirius displays relation based edges on the right hand side after the Topic nodes were selected and moved:



Below is what happens when you deselect the Topics: