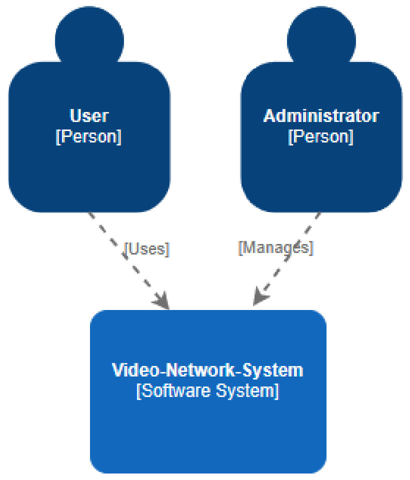
Exam no: Y3907647

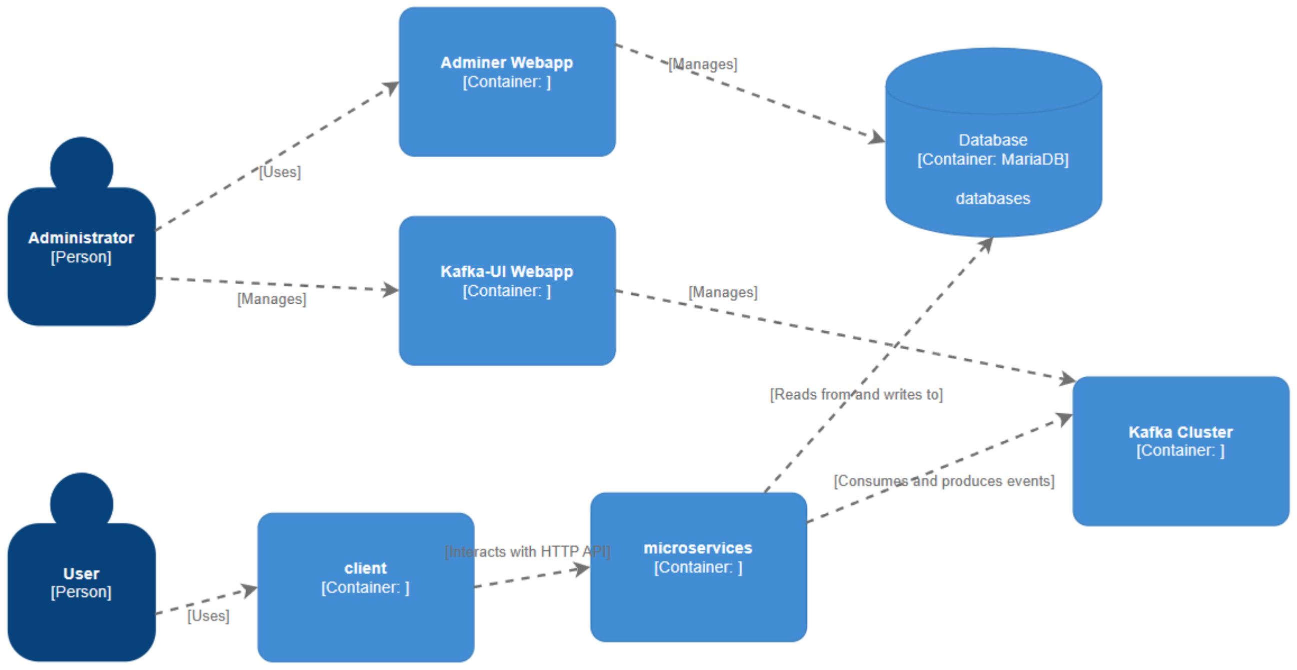
2.1 Part 1: Data-Intensive Systems

2.1.1 Architecture

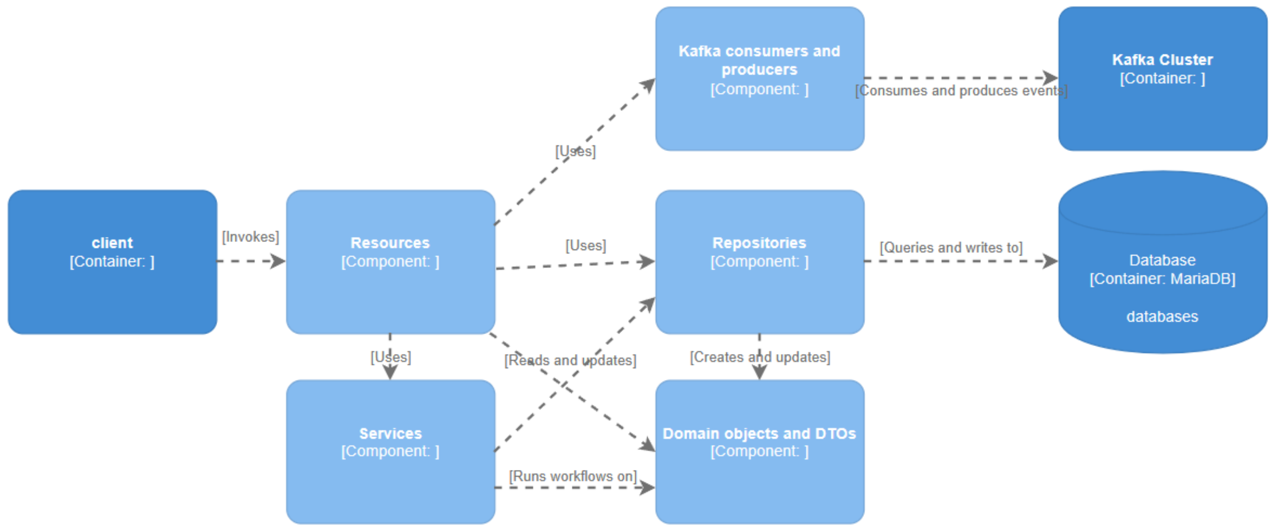
Level 1: Context



Level 2:Containers (simplified)



Level 3: Components of each microservice



The architecture can scale with increasing user demands in many ways. One being with the use of MariaDB which supports horizontal scaling, by distributing data across multiple MariaDB instances. Kafka also allows for new requirements in the future with its integration with other components. It will help in adapting the system without disrupting existing consumers.

An example of a new requirement would be to add another microservice. This involves creating a new database and client commands to invoke it.

2.1.2 Microservices

The high level design of my microservices include the use of kafka and RESTful APIs for secure messaging, MariaDB for databases, logging for performance tracking and scalability. Firstly the use of kafka and RESTful APIs together provides a powerful solution for secure messaging. Kafka is designed for high volumes of data and can retain messages for a long time. This is needed especially with the trending microservice where there are sliding time windows of one hour. RESTful APIs provide common endpoints such as GET, POST, PUT, making for easy design. Combining both creates a communication model which is well built for the high level design of the microservices.

In terms of database management, each microservice has its own database allowing for autonomy, which is in line with the separately deployable and scalable feature.

Throughout the microservices actions such as subscribing to topics are logged. This helps in maintaining the microservices by allowing certain aspects to be monitored and troubleshooted.

I had three microservices, first being the video microservice. This was for user and video creation. These users could then watch, like and dislike the videos. My command line client assisted this with the following commands: add-user, post-video, add-like, add-dislike, add-viewer. To help with management of users and videos there are also commands to list the all the users or videos such as: get-videos, get-viewers, list-by-creator, get-users.

The second microservice is the trending hashtag microservice. This can be used via the command line client with: get-all-hashtags, get-top-10-hashtags. The first command will list all of the hashtags that have been liked. So if a video with hashtags is uploaded, but has not been liked, the hashtags under that video will not be listed. For the second command, this lists the top 10 liked hashtags.

The third and final microservice is the subscription microservice. Here users can subscribe and unsubscribe from hashtags. After a user has subscribed to one or more hashtag, the subbed-top-10-next-videos-to-watch command with their user id, will list the next 10 videos to watch for each subscription. This is done via how many likes the video has as well as requiring that the video contains the corresponding hashtag.

A list of all the commands can be seen in the terminal by running gradlew run –args="—help".

For very repetitive tasks such as creating many users or uploading many videos, scripts can be made to automate this job.

2.1.3 Containerisation

In terms of scaling, integrating MariaDB and kafka allows the system to scale horizontally allowing for large numbers of users. By running multiple instances of containers and adding nodes to the kafka clusters, the increased user load can be accommodated. If more containers are needed, I can declare them within the docker compose file. With increased users comes an increase in messages sent between the microservices. The maintenance of this can be achieved by increasing the number of brokers and distributing the partitions across them.

For resilience I have implemented health checks for my containers. I will be notified if a container becomes unhealthy and is automatically restarted to maintain reliability.

To run bring up all of the microservices using the docker-compose file, you must first cd into each microservice directory and run a ./gradlew dockerBuild. After that you can run the docker-compose up -d to bring up the microservices. There is more information in the readme file.

2.1.4 Quality Assurance

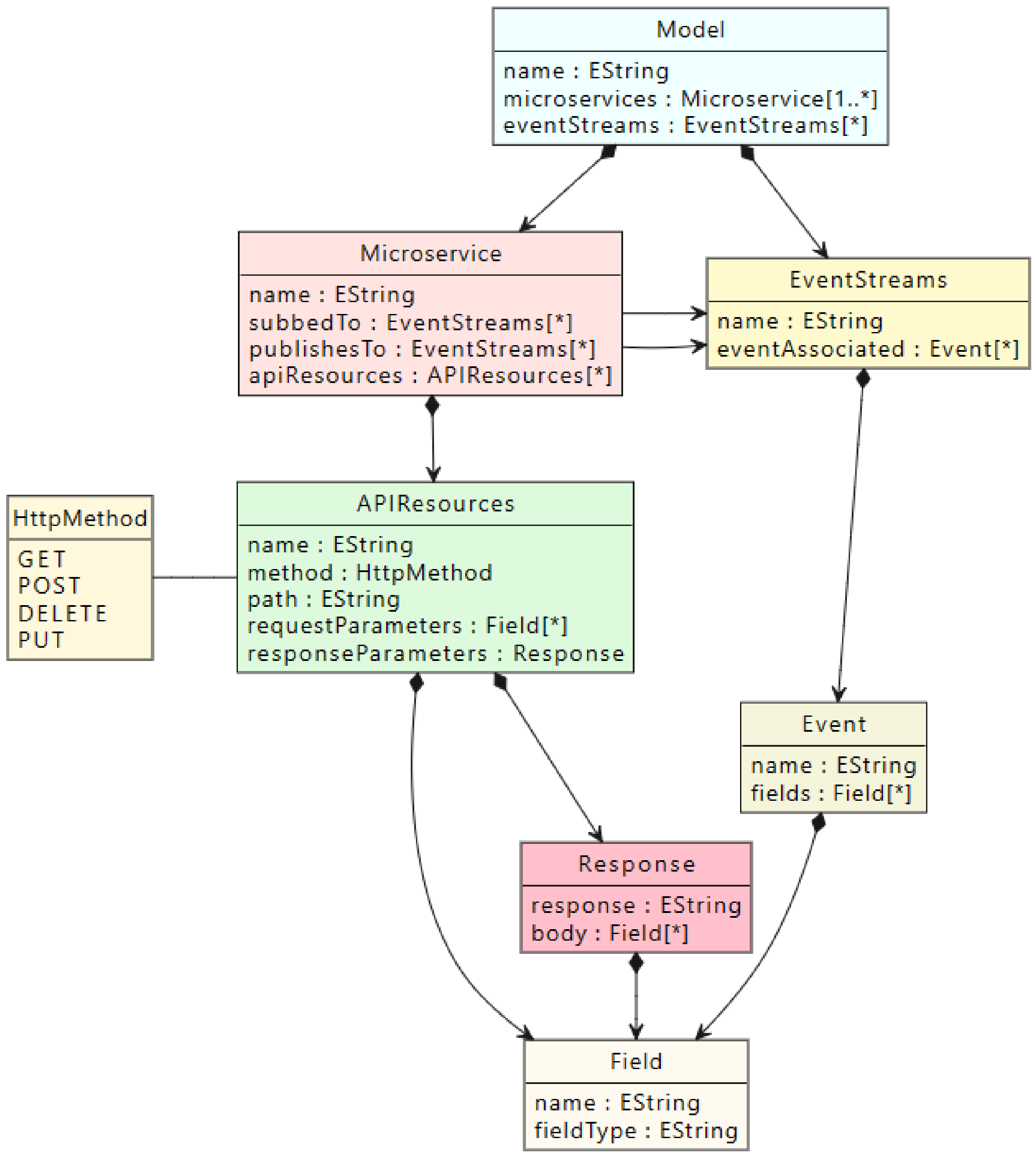
I created tests for my videos controller but unfortunately after running into initialization errors I was not able to see how my system performed against these tests. However during creation of my multiple microservices, I would manually test at each stage, to make sure that everything was working as intended. This included creating users and videos and testing the various commands to make sure it worked. I would ensure this by using kafka ui the localhost ports on the web and logs from the terminal. The main test which I did not verify working was for finding the top 10 current liked hashtags within a rolling time window of 1 hour. I was only able to find out the top 10 most liked hashtags of all time. For this test to succeed I would probably have to look into the logic of my kafka streams and ensure that it is working as intended.

Inspecting my Docker images, in terms of the microservices I built, all had 31 vulnerabilities, 4 being critical and 20 being a high vulnerability in each. The MariaDB image had 81 vulnerabilities with 37 high and 3 critical. With the time I had left I opted from dealing with the issues.

2.2 Part 2: Application of Model-driven Engineering

2.2.1 Metamodel

Class Diagram



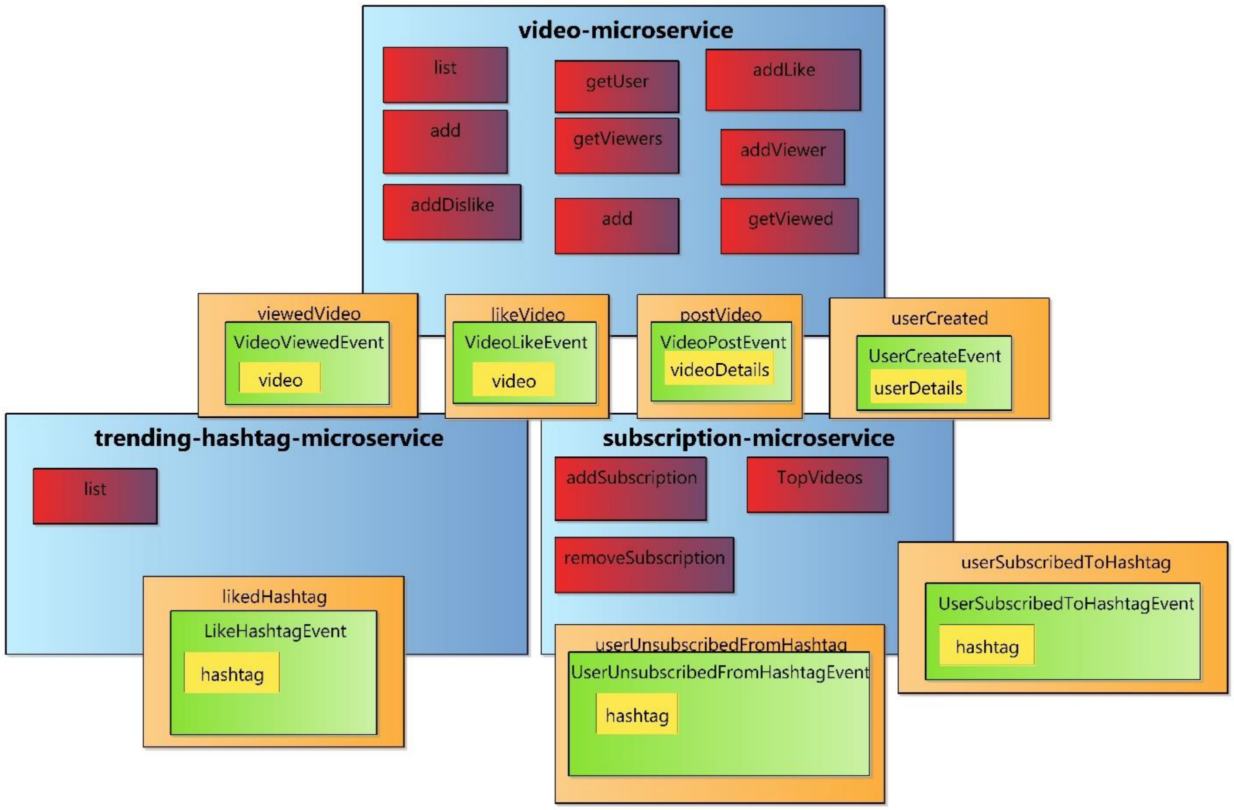
My metamodel, as shown by the class diagram, has 5 classes. The Microservice can contain multiple EventStreams which each have their own event type, and a list of fields.The APIResource includes the HttpMethod type, the path and the request and response parameters.

Some assumptions I have made are that there can be multiple microservices. Another is that fieldType is just a string, in actuality it would be defined as something more specific to the event. Also in the Field class, I have left out the key type as it would be of type long for all. In APIResources I have assumed that the correct string for path will be given.

Design decisions I had were to have the http method as just a string, however I decided to create a separate enumeration class to store all the different http methods I wanted.

Another design decision I had was to model the microservices around the event streams, having the microservices be apart of only the streams they subscribe or publish to. I discounted this as I thought the model was much clearer this way around as the microservices are the main part of the system.

2.2.2 Graphical Concrete Syntax



Firstly going over my syntax design, I have shown all three microservices in large blue containers. Inside each are the API resources, showing the name of each. The orange containers around the outside represent the event streams, with the event and request inside. How a microservice subscribes to an event stream is shown by the overlap. In my syntax the producers are also generally shown on the bottom of the microservice. I decided on this type of implementation as it shows a very abstract and simple overview of my architecture. All microservices and event streams are the first things that can be seen, as in my opinion, are the most important features.

This also ties into the main strengths of my design, having it be not as complex makes it easier to take in. There was a lot of information that I thought would crowd the graphics, including all the data seemed unnecessary and so I opted for the most important first. The use of color allows for more semantic clarity as all the boxes are different colors and refer to different classes. Containers allow for more information to be stored in a relevant location. This also led to higher levels of abstraction, the user can isolate one microservice and look at the stored resources within it, as well as ultimately looking at the model as a whole.

Some weaknesses of my design would be scaling. To add more and more microservices or event streams could eventually lead to it being very squished and complicated. In this case I would change it to have arrows, microservices pointing to event streams and vice versa, showing off publishers and subscribers. In this way my design shows a lack of adaptability and the alternative showing much better flexibility.

2.2.3 Model Validation

The first constraint is needed as there would be no model without at least one microservice. It is implemented by checking if the size of the microservices is greater than 0.

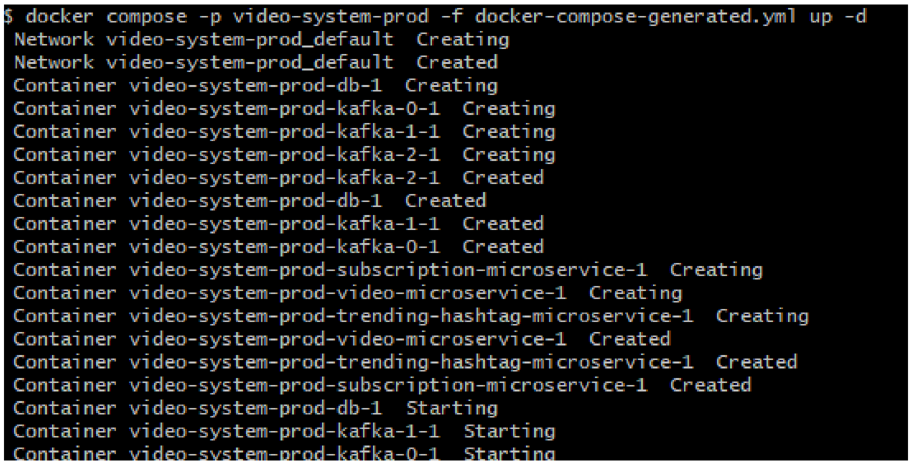
The second constraint is there to avoid a situation where events are defined but not associated with and event stream. I implemented this by checking the events and making sure they were all associated under an event stream.

The third constraint is there to check that there is a complete relationship between the microservices and event streams. However I did not implement this as some of the event streams where not consumed by another microservice. For example the user subscribed and unsubscribed from hashtags.

2.2.4 Model-To-Text Transformation

The model-to-text transformation I made was for container orchestration. Using my model I was able to iterate over each of the microservices and create a container within the docker compose file. I did this to allow for easier production of more microservices when it comes to deployment and creating the compose files.

I have organized the generated code with one .egx file and one .egl file. When run the egx file outputs to a m2tDocker folder with the docker-compose file inside.



```
$ docker compose -p video-system-prod -f docker-compose-generated.yml up -d
 Network video-system-prod_default  Creating
 Network video-system-prod_default  Created
 Container video-system-prod-db-1  Creating
 Container video-system-prod-kafka-0-1  Creating
 Container video-system-prod-kafka-1-1  Creating
 Container video-system-prod-kafka-2-1  Creating
 Container video-system-prod-kafka-2-1  Created
 Container video-system-prod-db-1  Created
 Container video-system-prod-kafka-1-1  Created
 Container video-system-prod-kafka-0-1  Created
 Container video-system-prod-subscription-microservice-1  Creating
 Container video-system-prod-video-microservice-1  Creating
 Container video-system-prod-trending-hashtag-microservice-1  Creating
 Container video-system-prod-video-microservice-1  Created
 Container video-system-prod-trending-hashtag-microservice-1  Created
 Container video-system-prod-subscription-microservice-1  Created
 Container video-system-prod-db-1  Starting
 Container video-system-prod-kafka-1-1  Starting
 Container video-system-prod-kafka-0-1  Starting
```

Here I run the generated docker compose file, which successfully starts the docker containers.