Engineering 2: Automated Software Engineering (ENG2)

2.1.1 Architecture

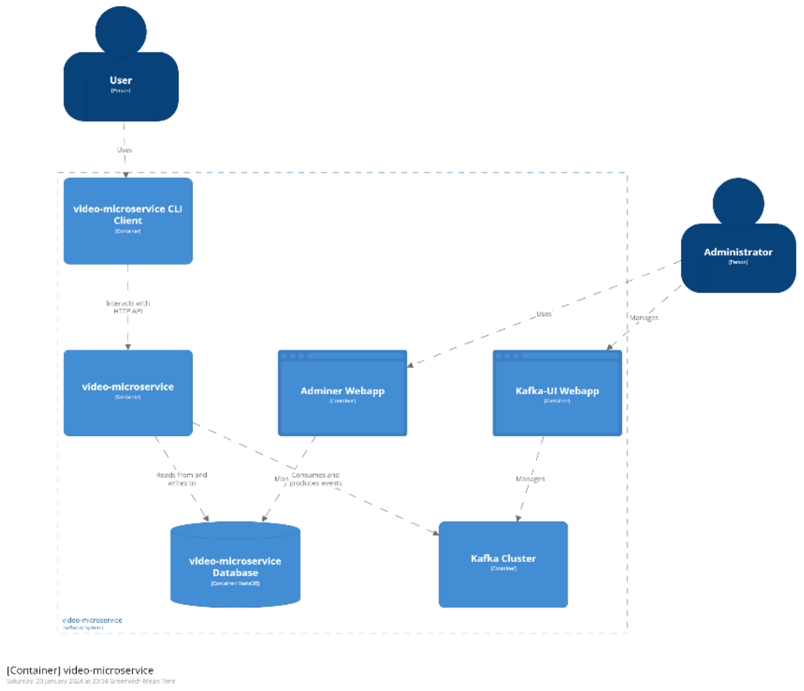


Figure 1

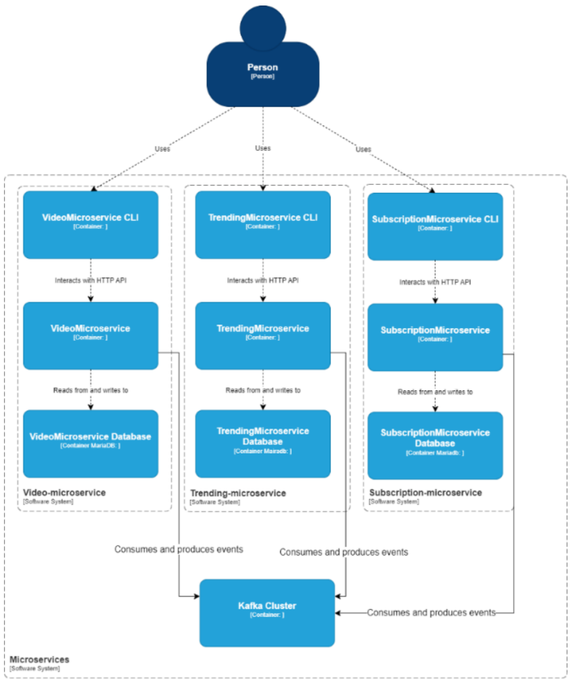


Figure 2

Diagrams overview

Each of the microservices, when they are separately deployed, have a similar structure to what's found in *figure 1* which is an architecture overview for the video-microservice. Where they each have a user that connects to a CLI which then connects to the microservice and, in turn, connects to the relevant database. Then they also have an administrator which can connect to do Kafka-UI, adminer and structurizr.

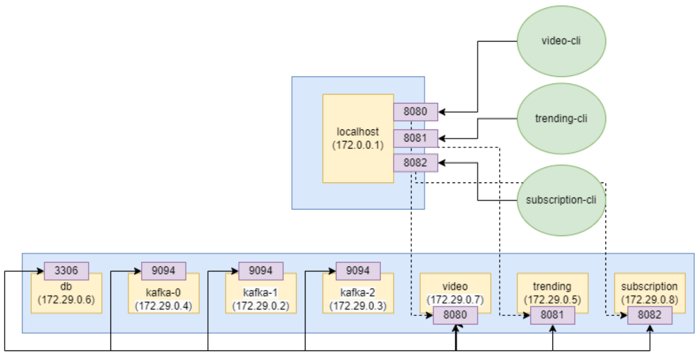


Figure 3

When the microservices are containerised and deployed as one it looks like *figure 2.* The network architecture of the microservices can also be seen in *figure 3.*

All of the C4 models can be found in: **.\microservices\structurizr\diagrams** and can be viewed when using
docker compose up -d at address: http://localhost:8081/
as stated in the readme.md.

Scaling architecture

One of the ways in which this architecture can scales with an increase in users is as each of the microservices are separate. If one of the microservices sees an increase in traffic, (e.g. video-microservice) it can be relatively easily allocated more processing based on its specific load to do its job without affecting any of the other microservices. As well as this more instances can be added of that service as well as load-balancing can be used to distribute the load across the system as a whole. So, if the video-microservice saw an increase in traffic, the partitions for the node could be increased, e.g. from 3 to 6, to allow linear scaling for both consumers and producer.

Another method which can be used to scale with an increase in demand is horizontal scaling in which databases/microservices are run on multiple machines where each machine is an independent node. This means that to scale the system if an increase in load occurs would be to add more instances of the microservices.

Adapting to new requirements

The system can be expanded upon in the future by creating new microservices to perform extra tasks, such as a recommendation system. The newly added microservice can subscribe to the Kafka producers which are already in place to get data out of the currently implemented microservices. It can then use the Kafka cluster which is already in use, and all which needs to be added is the correct setup in the docker compose file. A new CLI client would need to be made for the user to connect to, but this can run on the same service that the other CLI clients run on.

So, for a recommendation algorithm, the microservice can be initialised in the standard way, then to create a recommendation algorithm it could subscribe to the videos being posted (in a similar way to how it works the trending-microservice) to get a reference of all the videos, then also subscribe to likes and dislikes, then trending hashtags and what users are watching. With this data, the new recommendation-algorithm-microservice could then look at what videos a specific user has watched previously and what hashtags it contains, then recommend videos with those hashtags and have a good like-dislike ratio. Then a user could use a command to ask the microservice what video they should watch, given their id, and it would return the best 3.

2.1.2 Microservices

- Each of the microservices can be found in in the *microservices/* folder found in this .zip as this project was created with the template provided.
- Each of the implemented microservices are separately deployable using, *docker compose up -d* (Note: Not the prod .yml) and running each of them using *./gradlew run* or using the inbuilt eclipse functions.
- Each of the implemented cli clients can be located in microservices/client.
- Each of the microservices can run locally without any cloud-based resources, and the data base can run locally in the docker container.
- All of the microservices can be launched in a single command using *./buildDocker.sh* , for more launch option/information see README.md.

High level design documentation

Video-microservice

The microservice is built using the Micronaut framework, as such it has the standard file structure. Inside src/main/java… is where the code is located. It is split into five folders:

- Controllers, which contain the user and video controllers for the HTTP RESTful commands required for the CLI to function.
- Domain, which contains the user and video objects which contain the data structures and relevant functions.
- DTO, contains the data transfer objects for user and video, so that not all of the data is sent to the CLI.
- Events, contains the Kafka producers and consumers for things like posting a video and liking a video.
- Repositories, contains the repos for videos and users.

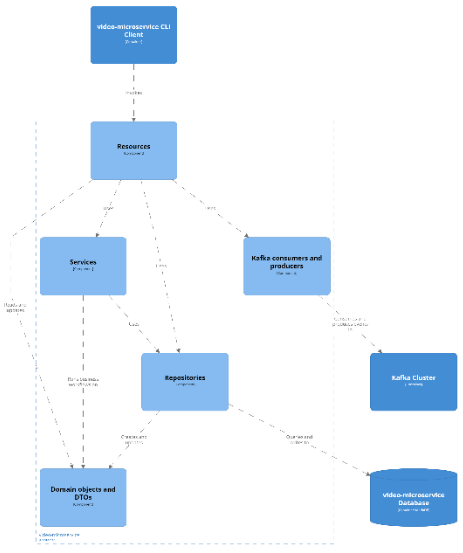


Figure 4

The high-level design of this microservice is first that the Video object contains the data for a video to contain a user, title, and hashtag. This can then be listed using the controller command in which each of the videos which have been added to the database using the CLI are reported back to the user. This is also the same for listing videos by user/hashtag but the function checks for the parameter entered and only returns those which fit. A user can also mark a video as viewed and like/dislike a video. The Kafka producers are designed to produce events which can be listed two by an external microservice. The c4-diagram shown in *Figure 4* for the video-microservice also gives a good overview of the high-level structure. Diagrams for each of the microservices can be found in**.\microservices\structurizr\diagrams.**

Trending-microservice

The microservices' file structure is set up the same as the video-microservice.
The high-level design of this microservice is that is primarily designed around the KStream consumer for the videos posted, found in /events/VideosStreams. This subscribes to the video-posted topic and uses a groupedStream with a sliding window to get all of the videos posted in the last hour. This is then consumed and the hashtags of each of these videos are collated and ranked by the top 10. Then when the user uses the command get-top-ten-hashtags in the CLI, it then displays it as a string.

Subscription-microservice

The microservices' file structure is set up the same as the video-microservice.
The high-level design of this microservice is that is primarily designed around the kafka subscriptions for each of the events they're listening to. So, the microservice has a listener for the video-posted topic, produced in video-microservice, and stores these in the database. Then when a user subscribes to a certain hashtag that is also stored. Then when the user gets the top-10 videos to watch next using the CLI, it checks which videos have the correct hashtag and displays them.

Usage of command cline client (CLI)

Video-cli

Command	**Description**	**Example usage with ./gradlew –args="…"**
get-videos	Fetches all the videos	"get-videos"
add-video	Addes a video to the database	"add-video Vid1 User1 hash1,hash2"
get-video	Get a specific vid with id	"get-video 1"
update-video	Updates the data of a video	"update-video 1 -t Vid2"
delete-video	Deletes a video with id	"delete-video" 1
add-user	Adds a user to database	"add-user User1"
get-users	Fetches all the users	"get-users"
get-viewers	Fetches all the viewers of a certain video with Id	"get-viewers 1"
add-viewer	Adds a viewer to a video	"add-viewer 1 2"
delete-viewer	Deletes a view with id	"delete-viewer 2"
add-like	Adds a like to a video given a video id and user id	"add-like 1 2"
add-dislike	Adds a dislike to a video from a user given video id and uId	"add-dislike 1 2"
get-dislikes-of-video	Gets users who disliked a video	"get-dislikes-of-video 1"
get-likes-of-video	Gets users who liked a video	"get-likes-of-video 1"
get-videos-by-user	Fetches the videos posted by a user	"get-videos-by-user User1"
get-videos-by-hashtag	Fetches the videos with a common hashtag	"get-videos-by-hashtag Hash1"

Trending-cli

Command	**Description**	**Example usage with ./gradlew –args="…"**
get-all-hashtags	Fetches all the hashtags in the database (development function)	"get-all-hashtags"
add-hashtag	Adds a hashtag to the database (development function, don't use)	"add-hashtag Hash1"
get-top-ten-hashtags	Fetches the top ten hashtags from videos posted in the last hour	"get-top-ten-hashtags"

Subscription-cli

Command	**Description**	**Example usage with ./gradlew –args="…"**
subscribe-to-hashtag	Allows the user to subscribe to a hashtag	"subscribe-to-hashtag User1 Hash1"
unsubscribe-to-hashtag	Allows the user to unsubscribe to a hashtag	"unsubscribe-to-hashtag User1 Hash1"
get-next-videos	Fetches the top ten hashtags from videos posted in the last hour	"get-next-videos User1"

2.1.3 Containerisation

For production, the system can be deployed in a single command using *./buildDocker.sh* as seen in *figure 5.* This first runs dockerBuild on each of the microservices to compile them, then brings up the kafka-clients before each of the topics for the microservices are initialised. Then finally the docker compose-prod file executed to start the microservices.

In *figure 6*, it illustrates the interconnections for each of the CLIs and the system as a whole.

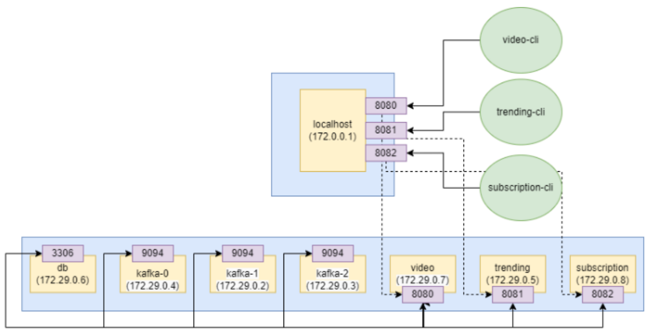
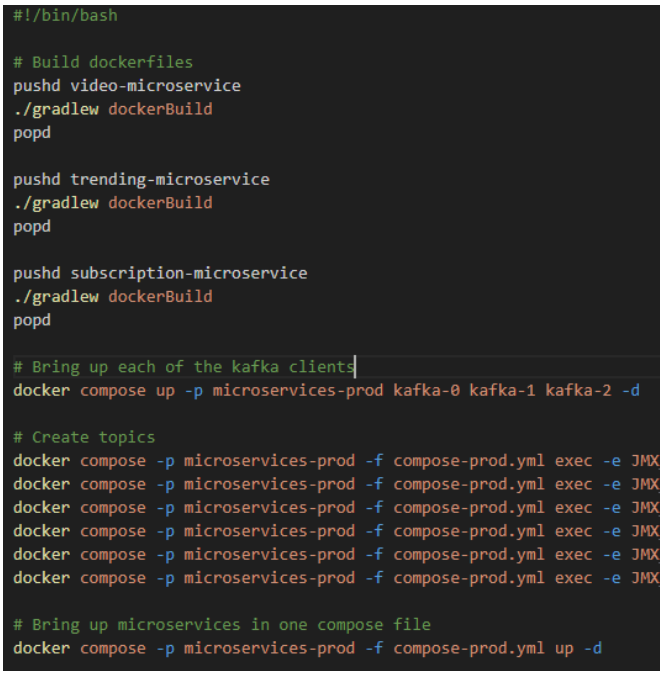


Figure 6



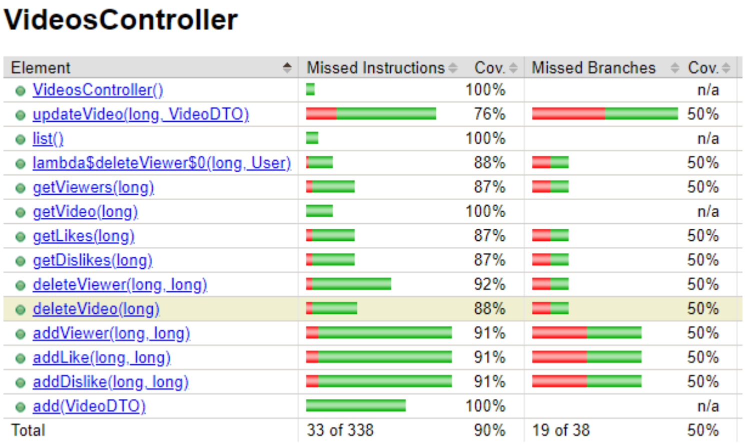
```
#!/bin/bash

# Build dockerfiles
pushd video-microservice
./gradlew dockerBuild
popd

pushd trending-microservice
./gradlew dockerBuild
popd

pushd subscription-microservice
./gradlew dockerBuild
popd

# Bring up each of the kafka clients
docker compose up -p microservices-prod kafka-0 kafka-1 kafka-2 -d

# Create topics
docker compose -p microservices-prod -f compose-prod.yml exec -e JMX
docker compose -p microservices-prod -f compose-prod.yml exec -e JMX
docker compose -p microservices-prod -f compose-prod.yml exec -e JMX
docker compose -p microservices-prod -f compose-prod.yml exec -e JMX
docker compose -p microservices-prod -f compose-prod.yml exec -e JMX
docker compose -p microservices-prod -f compose-prod.yml exec -e JMX

# Bring up microservices in one compose file
docker compose -p microservices-prod -f compose-prod.yml up -d
```

Figure 5

This solution can scale up to large numbers of users by scaling each of the microservices independently, as such increases the microservice in need without effecting the other services. This also means that there's focused resource allocation so that it can be more efficiently optimized. Also, as the users only connect to the microservices though REST interfaces, it means that using software the requests can be re-routed to another container running the microservices to balance the load on where the requests are coming from. Also when topics are created in the initialisation, the number of partitions can be increased to increase the parallelisation of the service, allowing more users to run the same commands at once.

This solution is also resilient to failures, as each of the microservices are separate, it means that if one of them fails the other microservices stay up, and if configured correctly, can restart, and recover automatically without human intervention which leads to increased uptime.
This solution also means that automated heath-checks can be run to make sure that the system is working as expected.
Containerisation also allows for increased security as they isolate their applications and dependencies as well as having enhanced security controls, which is much better than using a VM. Lastly it has more predicable behaviour as it behaves more consistently across different environments as it is always using the same alpine Linux environment in Docker.

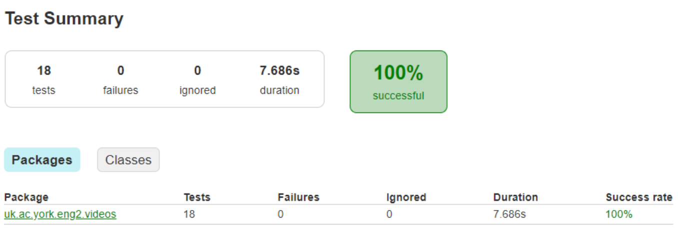
2.1.4 Quality Assurance



VideosController

Element	Missed Instructions	Cov.	Missed Branches	Cov.
VideosController()		100%		n/a
updateVideo(long, VideoDTO)		76%		50%
list()		100%		n/a
lambda$deleteViewer$0(long, User)		88%		50%
getViewers(long)		87%		50%
getVideo(long)		100%		n/a
getLikes(long)		87%		50%
getDislikes(long)		87%		50%
deleteViewer(long, long)		92%		50%
deleteVideo(long)		88%		50%
addViewer(long, long)		91%		50%
addLike(long, long)		91%		50%
addDislike(long, long)		91%		50%
add(VideoDTO)		100%		n/a
Total	33 of 338	90%	19 of 38	50%

Figure 7



Test Summary

18	0	0	7.686s	100%
tests	failures	ignored	duration	successful

Packages Classes

Package	Tests	Failures	Ignored	Duration	Success rate
uk.ac.york.eng2.videos	18	0	0	7.686s	100%

Figure 8

For each of the microservices, the main way that I initially tested them is though exploratory testing using the CLI created. This allowed for quick testing to be used both during and after they were developed.

To run the tests, start a microservice independently using *docker compose up -d* and go to the test folder for the given microservice. Then run *./gradlew test* or use the inbuilt commands within eclipse. This should then produce the test summary html found, like in *Figure 8*, and can found once produced in */build/reports/tests/test/index.html.*

The primary way that the microservices were tested was though unit tests. Though Junit, each of the elements in the controller were tested, using various methods, and as shown in *figure 7* using the Gradle JaCoCo plugin, it has a 90% code coverage for the VideosController and for the video-microservice overall its 84%.

Interaction based testing was also used when testing the production of Kafka records, by using a spy. This was used to test that the producers, such as the video-posted producer, were working as intended.

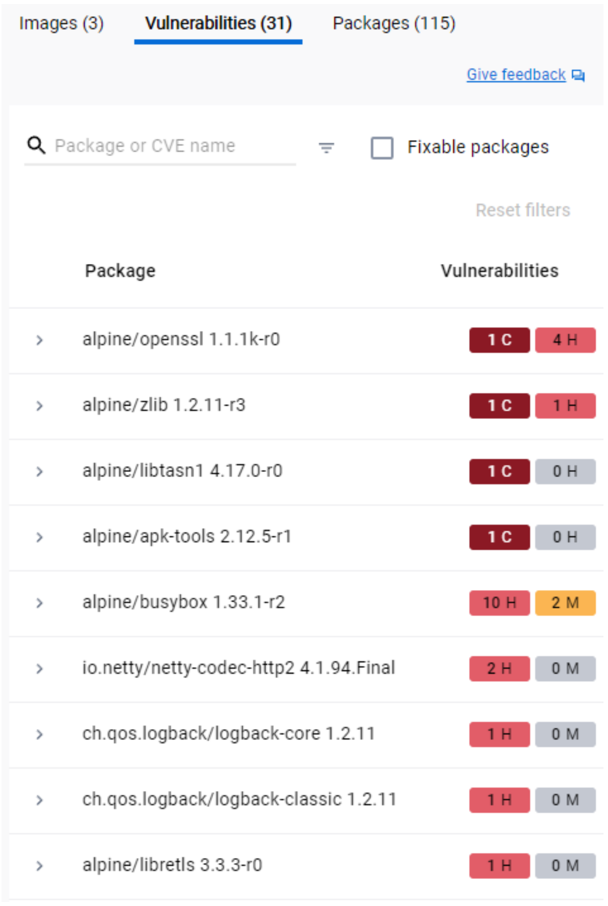
Awaitility was also used to test the asynchronous interactions, for example when a video is viewed and see if the database is updated with the correct data.

To test the Kafka streams used in the trending-microservice, integration testing was used as well. This was done by writing a custom Kafka listener to update events when a record is received from the stream.

With the current implementation, none of the test failed for the video-microservice. As such, the completeness the tests showed that 84% of the instructions were covered and approximately 50% of the branches were missed. The reason why 50% of the branches were missed was primarily due to all of them being used for exceptions, e.g. checking if the input is null. The reason why I did not test these was because it felt redundant as checking that the branches work correctly are correct also would check the error branches if they were incorrectly implemented.

Inspection of docker images

Using Docker Scout, a number of vulnerabilities were found in the docker images used for the microservices. For example, each of the microservices have 31 vulnerabilities. The most major of which (having a critical CVSS score) all have to do with libraries within Alpine Linux, as shown in *Figure 9*. Theses being libraries which alpine has installed by default and have been found to contain vulnerabilities inside them.


Images (3) | Vulnerabilities (31) | Packages (115)

Give feedback

Package or CVE name | Fixable packages

Reset filters

Package	Vulnerabilities	
alpine/openssl 1.1.1k-r0	1 C	4 H
alpine/zlib 1.2.11-r3	1 C	1 H
alpine/libtasn1 4.17.0-r0	1 C	0 H
alpine/apk-tools 2.12.5-r1	1 C	0 H
alpine/busybox 1.33.1-r2	10 H	2 M
io.netty/netty-codec-http2 4.1.94.Final	2 H	0 M
ch.qos.logback/logback-core 1.2.11	1 H	0 M
ch.qos.logback/logback-classic 1.2.11	1 H	0 M
alpine/libretls 3.3.3-r0	1 H	0 M

Figure 9

In the time available it did not seem feasible to fix theses security issues, as it would require individually updating the packages inside of Alpine which would take a large amount of time, would not necessarily fix the vulnerabilities and is not crucial to do for this assessment.
Also, the easier way to deal with this would be to get the newest version (or most recent known safe version) of Alpine. However, due to Micronaut 3.10.1 requiring this version of alpine, it means that a large amount of work around would need to be done to re-implement all of the functionalities done so far in the newer version and would mean that it would not necessarily work on the examiner’s computer, as it doesn’t include Java 11.

Inside adminer, and structurizr/lite there were only low severity vulnerability, as such was not worth the time investment to fix them. However, inside of mariadb:11 there were a couple of critical vulnerabilities, but they were in the standard library so to fix them would be unfeasible, and the only logical way to do this would be to update MariaDB to a more recent version, but was not possible within the timeframe.

2.2.1 Metamodel

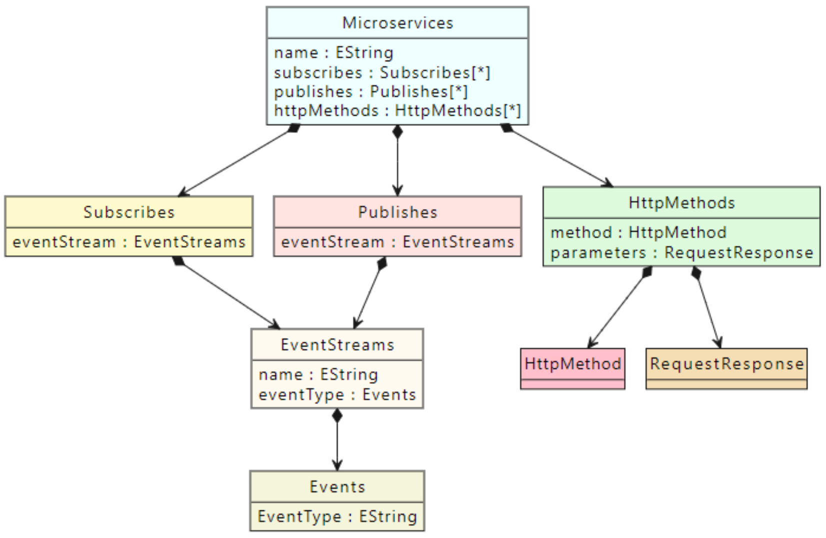


Figure 10

The meta-model can be found in *\modelling\metamodel\uk.ac.york.eng2.assessment.y3890370.* The meta-model shows the relationship between the microservice and it's Kafka subscribers and publishers. The model shows that for each Event/event stream the microservice needs a subscriber and publisher which is a constraint which can be placed on the system. The http methods are for the CLI API portion of the microservice, so each microservice needs to have a cli linked, as well as the HTTP methods requiring a specific type (e.g. Get, Post) as well as a response which contains the data, so the data which is inside the request type can be parsed correctly.

I have assumed that each of the events can be described as a String in the event type, in which the actual code will go into it to produce the event. I have also assumed that the HTTP method used is of a standard type, as such does not need a body. As well as for the request response that it will contain *http.ok* or some other form of response which is a pre-programmed type.

One of the alternative design decisions was to model the file/system structure of the microservice, as shown in *figure 11*. However, this was discounted due to it not sufficiently meeting the requirements set out in 1.2, as well as it not being of a high enough level of abstraction.

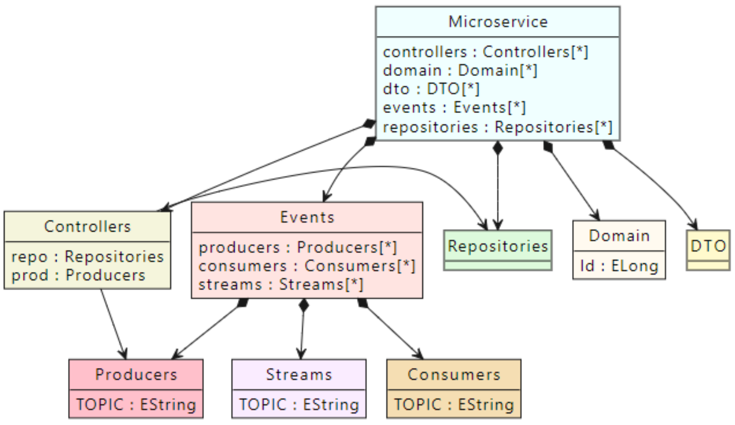


Figure 11