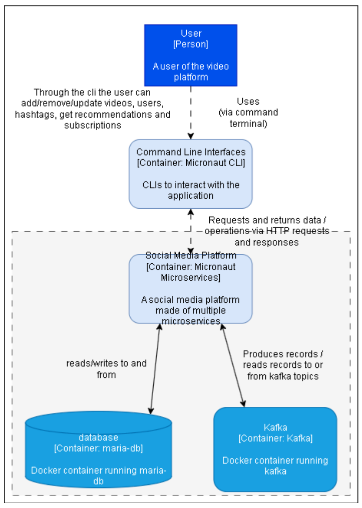
Question 2.1.1 - Architecture

Context Diagram


User
[Person]
A user of the video platform
Through the cli the user can add/remove/update videos, users, hashtags, get recommendations and subscriptions
Uses (via command terminal)
Command Line Interfaces
[Container: Micronaut CLI]
CLIs to interact with the application
Requests and returns data / operations via HTTP requests and responses
Social Media Platform
[Container: Micronaut Microservices]
A social media platform made of multiple microservices
reads/writes to and from
Produces records / reads records to or from kafka topics
database
[Container: maria-db]
Docker container running maria-db
Kafka
[Container: Kafka]
Docker container running kafka

To define the architecture of my video application I have split the architecture into three kinds of **C4 diagrams [1]** increasing in granularity. To see these diagrams in a better resolution, I have included them as .png files alongside the report submission in a folder called q1_architecture_diagrams.

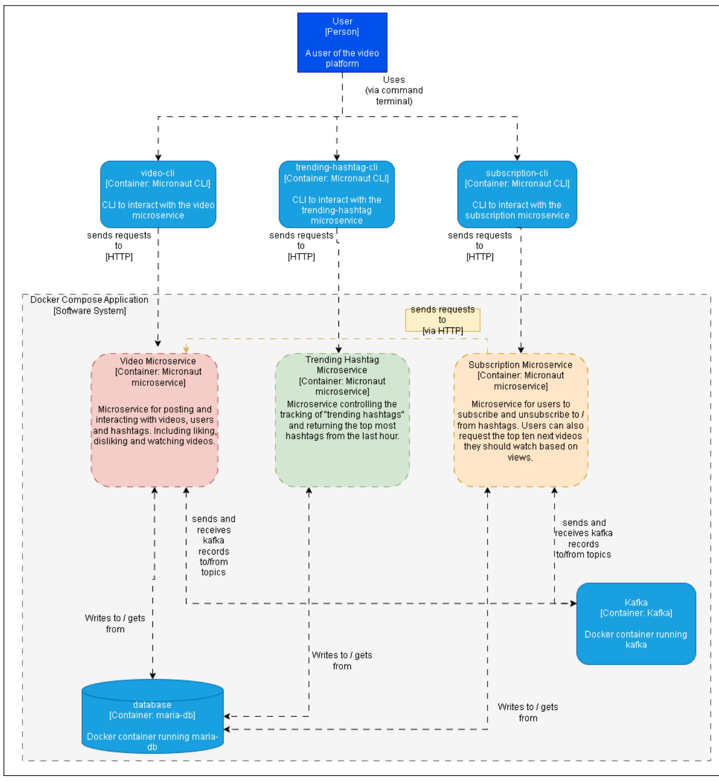
A **context diagram**, explaining the high level 'big picture', functionality landscape of the system as a whole

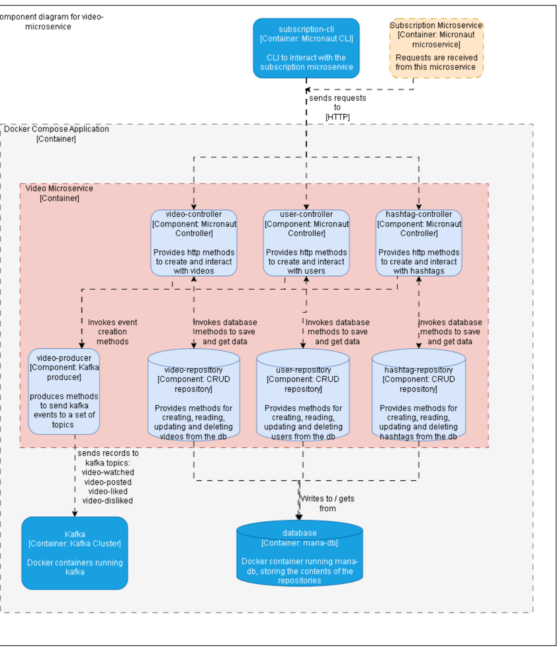
- A **container diagram** that elaborates on the context diagram, showing the detail of the application and how its inner components work together and interact with each other.
- And finally, three **component diagrams** (along the bottom of this page), showing the detail inside each subscription microservices and the processes and flows that it takes to answer requests and listen / produce events.

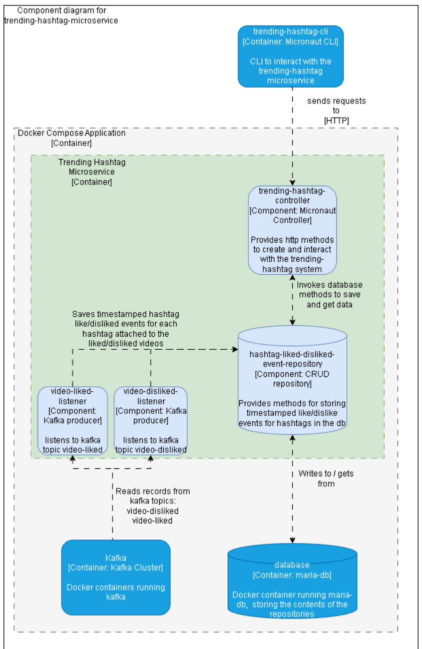
To elaborate on some of the diagrams, the following steps explain the processes and overall architecture in the container diagram / video-microservice component diagram:

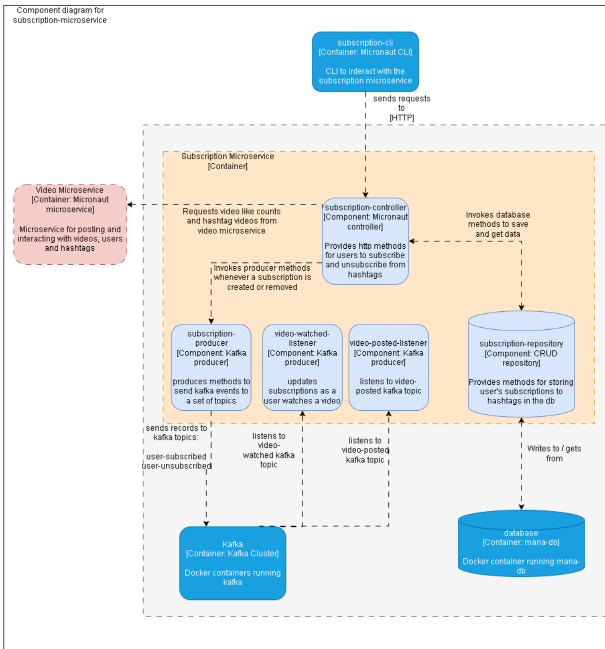
- The user runs commands such as 'add-video' using the correct command line interface (video-cli for video-microservice).
- This command uses HTTP requests to communicate with the video-microservice running as part of the Docker compose application.
- These requests are handled by the relevant controllers which interact with the necessary database entries to assemble responses back to the user.
- The microservice may also interact with the kafka cluster running in the Docker compose setup in order to listen to or produce events to a Kafka topic.
- The user receives the response which is printed to the terminal for the user.

Container Diagram


User
[Person]
A user of the video platform
Uses (via command terminal)
video-cli
[Container: Micronaut CLI]
CLI to interact with the video microservice
trending-hashtag-cli
[Container: Micronaut CLI]
CLI to interact with the trending-hashtag microservice
subscription-cli
[Container: Micronaut CLI]
CLI to interact with the subscription microservice
sends requests to [HTTP]
sends requests to [HTTP]
sends requests to [HTTP]
Docker Compose Application
[Software System]
sends requests to [via HTTP]
Video Microservice
[Container: Micronaut microservice]
Microservice for posting and interacting with videos, users and hashtags. Including liking, disliking and watching videos.
Trending Hashtag Microservice
[Container: Micronaut microservice]
Microservice controlling the tracking of "trending hashtags" and returning the top most hashtags from the last hour.
Subscription Microservice
[Container: Micronaut microservice]
Microservice for users to subscribe and unsubscribe to / from hashtags. Users can also request the top ten next videos they should watch based on views.
sends and receives kafka records to/from topics
sends and receives kafka records to/from topics
Kafka
[Container: Kafka]
Docker container running kafka
Writes to / gets from
Writes to / gets from
Writes to / gets from
database
[Container: maria-db]
Docker container running maria-db


Component diagram for video-microservice
subscription-cli
[Container: Micronaut CLI]
CLI to interact with the subscription microservice
Subscription Microservice
[Container: Micronaut microservice]
Requests are received from this microservice
sends requests to [HTTP]
Docker Compose Application
[Container]
Video Microservice
[Container]
video-controller
[Component: Micronaut Controller]
Provides http methods to create and interact with videos
user-controller
[Component: Micronaut Controller]
Provides http methods to create and interact with users
hashtag-controller
[Component: Micronaut Controller]
Provides http methods to create and interact with hashtags
Invokes event creation methods
Invokes database methods to save and get data
Invokes database methods to save and get data
Invokes database methods to save and get data
video-producer
[Component: Kafka producer]
produces methods to send kafka events to a set of topics
video-repository
[Component: CRUD repository]
Provides methods for creating, reading, updating and deleting videos from the db
user-repository
[Component: CRUD repository]
Provides methods for creating, reading, updating and deleting users from the db
hashtag-repository
[Component: CRUD repository]
Provides methods for creating, reading, updating and deleting hashtags from the db
sends records to kafka topics: video-watched video-posted video-liked video-disliked
Writes to / gets from
Kafka
[Container: Kafka Cluster]
Docker containers running kafka
database
[Container: maria-db]
Docker container running maria-db, storing the contents of the repositories


Component diagram for trending-hashtag-microservice
trending-hashtag-cli
[Container: Micronaut CLI]
CLI to interact with the trending-hashtag microservice
sends requests to [HTTP]
Docker Compose Application
[Container]
Trending Hashtag Microservice
[Container]
trending-hashtag-controller
[Component: Micronaut Controller]
Provides http methods to create and interact with the trending-hashtag system
Invokes database methods to save and get data
Saves timestamped hashtag like/disliked events for each hashtag attached to the liked/disliked videos
hashtag-liked-disliked-event-repository
[Component: CRUD repository]
Provides methods for storing timestamped like/dislike events for hashtags in the db
video-liked-listener
[Component: Kafka producer]
listens to kafka topic video-liked
video-disliked-listener
[Component: Kafka producer]
listens to kafka topic video-disliked
Reads records from kafka topics: video-disliked video-liked
Writes to / gets from
Kafka
[Container: Kafka Cluster]
Docker containers running kafka
database
[Container: maria-db]
Docker container running maria-db, storing the contents of the repositories


Component diagram for subscription-microservice
subscription-cli
[Container: Micronaut CLI]
CLI to interact with the subscription microservice
sends requests to [HTTP]
Subscription Microservice
[Container]
Video Microservice
[Container: Micronaut microservice]
Microservice for posting and interacting with videos, users and hashtags
Requests video like counts and hashtag videos from video microservice
subscription-controller
[Component: Micronaut controller]
Provides http methods for users to subscribe and unsubscribe from hashtags
Invokes database methods to save and get data
Invokes producer methods whenever a subscription is created or removed
subscription-producer
[Component: Kafka producer]
produces methods to send kafka events to a set of topics
video-watched-listener
[Component: Kafka producer]
updates subscriptions as a user watches a video
video-posted-listener
[Component: Kafka producer]
listens to video-posted kafka topic
subscription-repository
[Component: CRUD repository]
Provides methods for storing user's subscriptions to hashtags in the db
sends records to kafka topics: user-subscribed user-unsubscribed
listens to video-watched kafka topic
listens to video-posted kafka topic
Writes to / gets from
Kafka
[Container: Kafka Cluster]
Docker containers running kafka
database
[Container: maria-db]
Docker container running maria-db

Ability to scale to user demands - The system has the ability to scale to increasing user demands without much additional effort. At the simplest level, the capacity for the system to store data permanently through its maria db container can be expanded through vertical scaling i.e. adding additional resources to the container.

Increasing the number of instances of each microservice would improve the ability of the system to answer requests and read kafka events (providing the system has enough network speed, cpu speed, ram etc. and is not bottlenecked by these). This is possible as the microservices are stateless and all persistence is done through databases and inter-service communication through kafka topics. Running additional instances also helps to increase the reliability of the system, this is because if a container running a microservice crashes, there will be others that can continue answering http requests, therefore, there should be no loss in availability for the user.

However, when running multiple instances of a microservice, there will be multiple services potentially accessing the same database entries. This has been considered for my system and any editing of the database is marked as @transactional, so actions are performed in isolation before others are performed, this ensures that shared data spaces such as the database can remain synchronised and the fetched data is reliable and up-to-date.

With more effort, the system can be equipped with a load balancing **[2]** to ensure that traffic is divided evenly between all instances of the microservice to optimally use all the available resources and to avoid overwhelming any one of the instances with high traffic levels.

Finally, the last measure in place to help deal with scaling user demands is automatic restarting of all docker containers in the service. This ensures that if either kafka, maria-db or one of the microservice suffers a random crash, it will be restarted without the need for manual intervention. These kinds of crashes are more likely to occur with larger user numbers, so this is beneficial for the system and reduces downtime of the system.

Ability to adapt to future requirements - The system will be able to adapt to future requirements well, for example, the addition of a recommendation system could be implemented either as a fourth microservice in the system or as an addition to the video microservice. Notably, the recommendation could leverage the methods for tracking views, likes and dislikes that are present in the video microservice (vm) through its client, or it could utilise some of the kafka events being produced by vm to maintain an up-to-date list of recommendations.

Furthermore, due to the modelling work performed in part 2 of this assessment, aspects of the new microservice or new functionality could be modelled using the domain specific language created for this task. By doing this, the developer adapting the system can automatically generate sections of code using the model to text transformations produced in question 2.2.4, which saves time and reduces the chances for the developer making an error manually setting up these parts of the code by ensuring code is uniform across the microservices.

Other requirements, including additional interfaces and even visual interfaces can be met by utilising the http controllers provided by the microservices to interact with the microservices.

<u>Question 2.1.2 - Microservices</u> - Below is a description of the high level design of the microservices and any important implementation details and decisions.

video-microservice (VM) - Provides functionality for working with Videos, Users and Hashtags.

- ***Videos*** - Videos have a creator which is a User entity and a set of Hashtag entities linked to it. All of which can be accessed with the CLI.
- ***Users*** - Users have a username and a set of Videos they have created that can be retrieved with the CLI.
    - likes/dislikes/views - Video likes, dislikes and views are implemented as a many to many relationship between User and Video. This ensures that no duplicate likes/dislikes/views for videos can happen. The CLI provides ways to get the users who liked/disliked/viewed a specific video or to get the videos a user has liked/disliked/viewed.
- ***Hashtags*** - When creating a video, the service receives a string of comma separated hashtags. The controller automatically determines if those hashtags already exist as entities in the database, retrieving them and linking them to the newly created video. Or if they don't exist, the controller will create, save and link a new Hashtag entity.
    - These Hashtag entities have a many to many relationship with a set of videos which the controller and CLI make available, this allows users to see what videos are posted with a specific hashtag or to get the hashtags a video has.

trending-hashtag-microservice (THM) - Provides the functionality to receive the top ten trending hashtags.

- ***Criteria for a hashtag to be trending*** - Trending hashtags are considered to be the hashtags that have received the highest likes in the last hour. The service also tracks the dislikes a hashtag has received in the last hour and takes that into account (i.e. a hashtag with 3 likes and 2 dislikes will rank lower than a video with 2 likes).
- ***Storing like / dislike events*** - To track these events, two listeners receive events from the VM on the topics ‘video-liked’ and ‘video-dislikes’, these events are stored as a HashtagLikedDislikedEvent, which simply stores the timestamp and whether it was a like (+1) or dislike (-1), these events are then saved to a repository for use later.
- ***Requesting the trending hashtags*** - When the trending hashtags are requested from the CLI, the controller performs a search for any HashtagLikedDislikedEvents with timestamps from the last one hour and assembles a ranked list of trending hashtags using those events based on their total (like - dislikes) count.

subscription-microservice (SM) - Provides functionality for a User to subscribe to / unsubscribe from hashtags and to receive the top ten next video recommendations for that subscription.

- ***Recommendations*** - The recommendations for the top ten next videos are the top 10 videos that that user has not seen from the subscribed hashtag, they are ranked based on the total number of views each video has. So the top recommendation is the most viewed video that the user has not seen. To obtain an up-to-date number of views the microservice uses a HTTP client to request the views for a video from the VM.
- ***Updating recommendations*** - The subscription keeps track of three lists, the hashtag’s videos not seen by the user, the hashtag’s videos that the user has seen and the videos posted since the subscription was created. When the subscription is created, the HTTP client gets all the videos linked to the hashtag to populate the list of unseen videos. From there, the microservice’s listeners update the seen videos by listening for relevant events from Kafka topic “video-watched” and another listener updates the videos posted since the subscription was created by listening for relevant events from the Kafka topic “video-posted”.
    - It would have been possible to implement the full functionality of this microservice with HTTP requests to the vm, however, the exam states that it should use events from the VM, hence the two listeners updating the lists.

Command Line Interface (CLI) usage - To interact with the above microservices, each one has a CLI stored under the microservices/clients folder. Commands can be ran using ./gradlew run –args ‘<command> <args>’ and help can be printed by adding the -h flag in the args. Prior to using the cli, the system must be launched by using the docker compose.yml file included in the microservices folder, see the readme.md file for instructions on running it for the first time as additional setup is needed. **Note** - Any command that references a specific entity takes this as an ID of an entity that already exists and the user of the CLI is expected to work this way. The user can get IDs for entities with the provided get-Xs methods across the CLIs (e.g. get-users). Also note that similar commands are grouped together where their usages are similar and multiple commands may be displayed as prefix-<suffix/alternative> instead of listing two separate commands to save space.

video-cli commands - For a full list of all commands run './gradlew run –args "help"		
Command(s)	**Usage example**	**Explanation**
add-video	add-video 'vid-title' 'creator-id' 'tag1,tag2'	**vid-title** - the title of the video **creator-id** - the id of the user who the video belongs to **tags** - a comma-separated string of hashtags e.g. 'tag1,tag2'
add-user	add-user 'username'	**username** - username of the user
delete-video, delete-user	delete-video 'video-id' delete-user 'user-id'	**video-id / user-id** - the id of the video or user to delete
update-video	update-video 'new-vid-title' 'new-creator-id" 'newtags'	**new-vid-title** - the title of the video **new-creator-id** - the id of the user who the creator **newtags** - comma-separated hashtags e.g.'tag1,tag2'
update-user	update-user 'new-username'	**new-username** - the new username of the user
get-video, get-user, get-hashtag	get-video 'video-id' get-user 'user-id'	**video-id / user-id -** the id of the video / user to get
get-users, get-videos, get-hashtags	get-videos get-users	No parameters. Gets all of a kind of entity. (i.e. User, Video, Hashtag) depending on command
add-video-<viewer/liker/disliker>	add-video-viewer 'video-id' 'user-id'	**video-id -** the id of the video to watch / like / dislike **user-id -** the id of the user watching /liking / disliking
get-video-<viewers/likers/dislikers>	get-video-viewers 'video-id'	**video-id -** the id of the video to get the viewers for
get-<views/likes/dislikes>	get-views 'video-id'	**video-id -** the id of the video to get the views / likes / dislikes for
get-creator-videos	get-creator-videos 'user-id**'**	**user-id -** the id of the user to get created videos for
get-hashtag-videos	get-hashtag-videos 'hashtag-id'	**hashtag-id -** the id of the hashtag to get the tagged videos for
get-user-<watched/liked/disliked>-videos	get-user-watched-video 'user-id'	**user-id -** the id of the user to get the videos they have watched / liked / disliked
trending-hashtag-microservice-cli commands - For a list of all commands run './gradlew run –args "help"		
get-trending-hashtags	get-trending-hashtags	retrieves the top ten trending hashtags of the last hour
subscription-microservice-cli - For a list of all commands run './gradlew run –args "help"		
subscribe-to-hashtag	subscribe-to-hashtag 'user-id' 'hashtag-id'	**user-id -** id of user subscribing **hashtag-id** - id of hashtag (must exist prior to sub)
unsubscribe-from-hashtag	unsubscribe-from-hashtag 'user-id' 'hashtag-id'	**user-id -** id of user subscribing **hashtag-id** - id of hashtag user is subscribing to
get-subscription-recommendations	get-subscription-recommendations 'user-id' 'hashtag-id'	**user-id -** id of user who is subscribed to hashtag **hashtag-id** - id of hashtag user is subscribing to

__Question 2.1.3 - Containerisation__

The system has been packaged as a single docker compose.yml file inside the microservices folder. This allows the system to be brought up in a single 'docker compose up -d' command after some first time setup (see readme.md). The solution brings up all three of the microservices as docker images created by './gradlew dockerBuild' running on unique ports as well as three mariadb instances (one for each microservice, running on unique ports) and a kafka cluster.

How the solution can scale up to larger numbers of users -

- Kafka Cluster - The solution makes use of multiple kafka nodes to make a kafka cluster, by doing this, we can provide greater availability of data and fault tolerance (I'll discuss this later) as well as support for more consumers for each Kafka topic.In this solution, kafka topic data is split into a number of partitions, the number used for this solution is six partitions, with each of the three kafka nodes acting as the leader for two of the partitions. This solution can serve up to six consumers per topic. At present there are nowhere near this many consumers for any topic, meaning that if multiple instances of the microservice were to be run in parallel, this would not be an issue for our Kafka setup.

- Potential for horizontal scalability - With additional effort, multiple instances of the microservices could be implemented alongside load balancers to balance the traffic being sent to each, which would contribute to a more reliable and fast user experience.

- Vertical scalability - With additional effort, more processing power and storage could be added to the machine running the system which would be utilised by the system.

- *Multiple database instances* - Each microservice has its own mariadb instance, this is beneficial as users grow and their demands grow. This reduced the chance for databases to be overwhelmed with requests when compared with the alternative which is a single database setup. Each of these instances could be scaled vertically according to the storage / processing needs of each microservice to reach a cost efficient solution.

How the solution is resilient to failures -

- *Kafka topic partition replication* - The Kafka topics served by the Kafka cluster all utilise the replication factor setting for kafka topics. By setting this to three, we are ensuring all kafka topic partitions are duplicated and kept in sync across all three of the kafka nodes, this means that partitions are replicated to provide some fault tolerance in the event that a node fails, there will still be two other copies of the data in the cluster's other nodes.

- *Container restarting* - All containers within the compose file are set-up to automatically restart unless they are stopped manually. This ensures that if a container fails randomly it will be restarted automatically. This improves the fault tolerance and availability of the system as a whole and cuts down on the human intervention needed to maintain this system.

- *Reliable IP addressing* - The solution uses inter-service communication through HTTP requests to interact with the database and the services they need to perform their functionality. To make these requests, the microservices use URLs passed through from the compose file as environment variables using the service name e.g. 'video-microservice:8080'. This could reduce the possibility of failure due to mis-configuration or network addressing related problems.

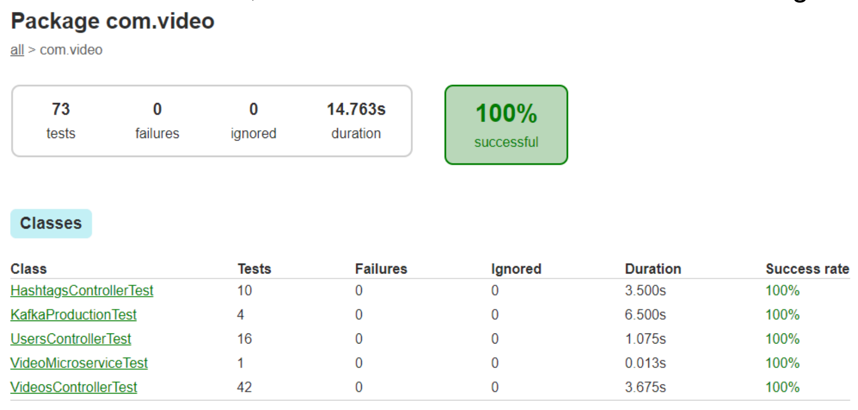
- *Database persistence and Kafka data persistence* - The mariadb contents are stored in a volume persistently, by doing this, the database data is not lost between restarts or failures of the system, this is important because the loss of all saved videos, users, hashtags and subscriptions would be devastating for real users. Additionally, the kafka cluster has its own volumes in order to save data persistently, this way no topic information and configuration is lost between restarts or failures of the system.

- *Unnecessary exposed ports have been removed* - Improving the security of the solution, reducing the chance of failures that come from potential exploits.

2.1.4 Quality Assurance

Running the tests - The tests for each microservice can be ran using './gradlew test', however, the docker containers for the kafka cluster, video-db, thm-db and subscription-db must be running (run 'docker compose up -d kafka-0 kafka1 kafka2 video-db thm-db subscription-db'). Upon completion of this action jacoco, a code coverage tool, will produce a coverage report stored at: 'build/reports/jacoco/test/html/index.html'.

Test Reports - Before explaining the tests performed, here are the test reports and coverage metrics. At the time of submission, all of the 100 tests I've written are running and passing.



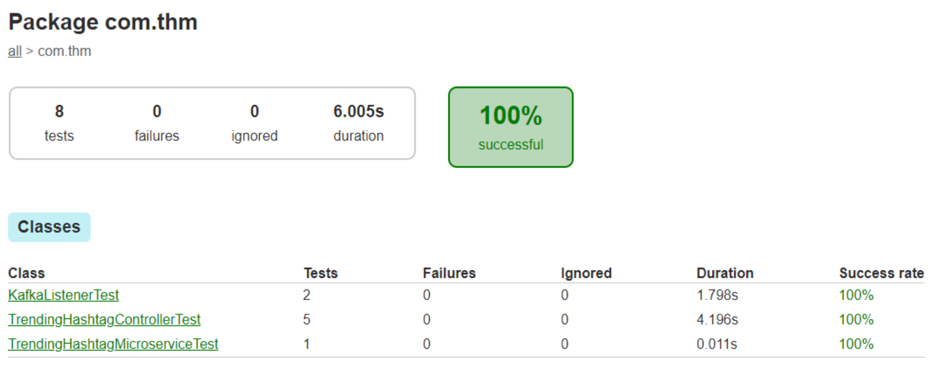
Package com.video

all > com.video

73	0	0	14.763s	100%
tests	failures	ignored	duration	successful

Classes

Class	Tests	Failures	Ignored	Duration	Success rate
HashtagsControllerTest	10	0	0	3.500s	100%
KafkaProductionTest	4	0	0	6.500s	100%
UsersControllerTest	16	0	0	1.075s	100%
VideoMicroserviceTest	1	0	0	0.013s	100%
VideosControllerTest	42	0	0	3.675s	100%



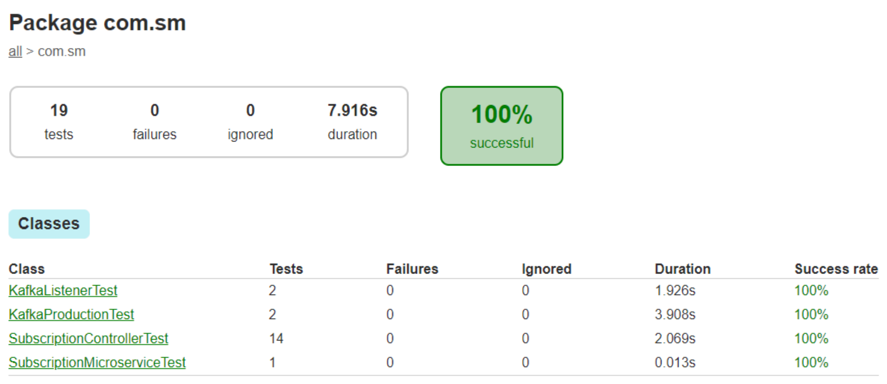
Package com.thm

all > com.thm

8	0	0	6.005s	100%
tests	failures	ignored	duration	successful

Classes

Class	Tests	Failures	Ignored	Duration	Success rate
KafkaListenerTest	2	0	0	1.798s	100%
TrendingHashtagControllerTest	5	0	0	4.196s	100%
TrendingHashtagMicroserviceTest	1	0	0	0.011s	100%



Package com.sm

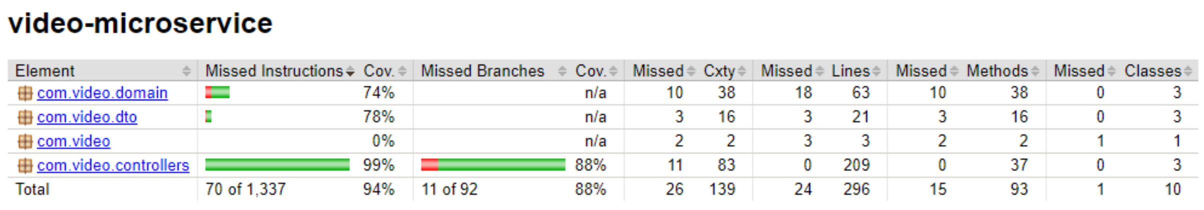
all > com.sm

19	0	0	7.916s	100%
tests	failures	ignored	duration	successful

Classes

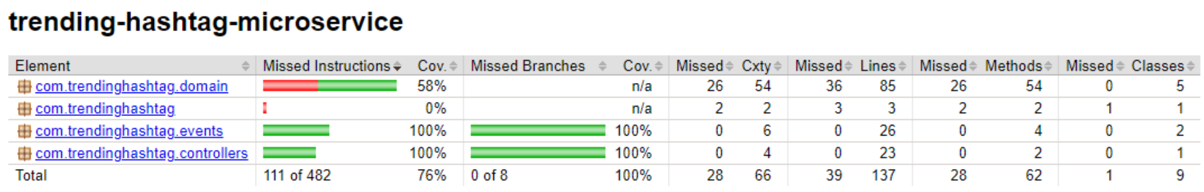
Class	Tests	Failures	Ignored	Duration	Success rate
KafkaListenerTest	2	0	0	1.926s	100%
KafkaProductionTest	2	0	0	3.908s	100%
SubscriptionControllerTest	14	0	0	2.069s	100%
SubscriptionMicroserviceTest	1	0	0	0.013s	100%

Overall Coverage Reports -



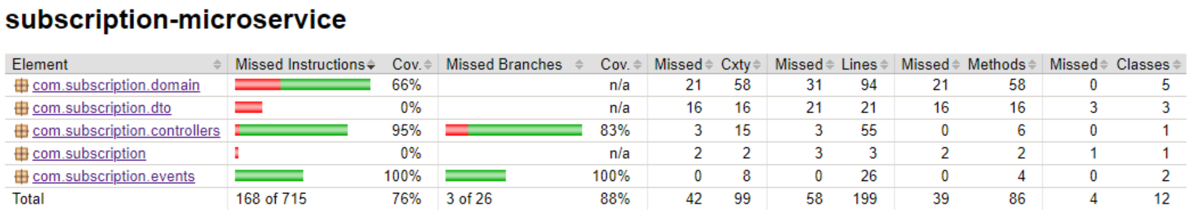
video-microservice

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.video.domain		74%		n/a	10	38	18	63	10	38	0	3
com.video.dto		78%		n/a	3	16	3	21	3	16	0	3
com.video		0%		n/a	2	2	3	3	2	2	1	1
com.video.controllers		99%		88%	11	83	0	209	0	37	0	3
Total	70 of 1,337	94%	11 of 92	88%	26	139	24	296	15	93	1	10



trending-hashtag-microservice

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.trendinghashtag.domain		58%		n/a	26	54	36	85	26	54	0	5
com.trendinghashtag		0%		n/a	2	2	3	3	2	2	1	1
com.trendinghashtag.events		100%		100%	0	6	0	26	0	4	0	2
com.trendinghashtag.controllers		100%		100%	0	4	0	23	0	2	0	1
Total	111 of 482	76%	0 of 8	100%	28	66	39	137	28	62	1	9



subscription-microservice

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.subscription.domain		66%		n/a	21	58	31	94	21	58	0	5
com.subscription.dto		0%		n/a	16	16	21	21	16	16	3	3
com.subscription.controllers		95%		83%	3	15	3	55	0	6	0	1
com.subscription		0%		n/a	2	2	3	3	2	2	1	1
com.subscription.events		100%		100%	0	8	0	26	0	4	0	2
Total	168 of 715	76%	3 of 26	88%	42	99	58	199	39	86	4	12

While instruction and branch coverage is not an absolute metric for good testing, I have achieved a high percentage on both across the microservice, with my calculated average across the three being 82% instruction coverage and 92% branch coverage.

Microservice Testing - The most simple type of tests performed are the Microservice test. This simply checks that the application is working by checking if it is running. If this test fails, then there is a larger problem with the microservice e.g. configuration errors.

Controller Testing - The most comprehensive testing in the suite comes from the controller tests. With almost 80 controller tests, spanning close to 1500 lines of test code between the three microservices, I believe these tests cover a broad, near comprehensive set of scenarios and edge cases. My logic for producing these controller tests was:

- If a requested entity can be missing, test it when it is present in the database and when it is not, capturing any combinations of these missing entities as well.
- If a method can be influenced by the state of the database, entities or situations happening before the controller method is called, test them.
- If a method is expected to produce a kafka event, capture this with a mocked producer and confirm that the logic to send the event has been triggered by the method. e.g. posting a video sends an event to the topic "video-posted".
- The slate should be wiped clean before all each test. Meaning all captured events and relevant repositories should be wiped clean with a @beforeeach method, ensuring that previous tests can not interfere with the next test even if they have failed or crashed.

The subscription microservice proved to be tougher than the other two to test, this was because the controller was making calls to the video-microservice, which is not running when the tests are running. To get around this, I injected a mocked version of the video, user and hashtag clients so that the controller could access the resources from the injected repositories instead of attempting to make a invalid HTTP call to the inactive video-microservice which would crash the test.

Kafka Producer Tests - As part of the controller tests they perform some kafka event testing through the form of a mocked kafka producer that records key-value pairs that would have been turned into an event, later checking those values match what is expected. However, this testing cuts kafka out, so I've performed producer tests, one for each method that would produce an event to a topic, which is four tests in the video microservice and two in the subscription microservice. These tests trigger the controller method that creates an event e.g. adding a video, and then waiting 30 seconds for that record to be captured from kafka and verified.

Kafka Listener Testing - The listener tests invoke listener methods to ensure that correct updates are performed in the microservice. There are two in the thm, checking that liked / disliked video records are turned into LikedDislikedEvents and stored to the repository and there are two more in the sm, checking that posted and viewed videos are updated in the subscription entities in the repository. I also attempted to implement listener tests by sending records to kafka and waiting for the listeners to react, unfortunately, I could not get this to work in the time available.

CLI Testing - Due to time constraints and the complexity of the tests, I have made the decision to not test my command line clients. With more time I would have produced tests for each command to ensure that the functionality is working as intended and is printed to the terminal as expected. The benefit of this is that mismatches between the clients and the microservice controllers could be caught and logical errors between the printed output and the received output could be picked up more clearly during testing.

Overall - I have produced a high number of tests (100), covering a range of scenarios and edge cases, which I believe is a good test suite for hypothetical further development of the services.

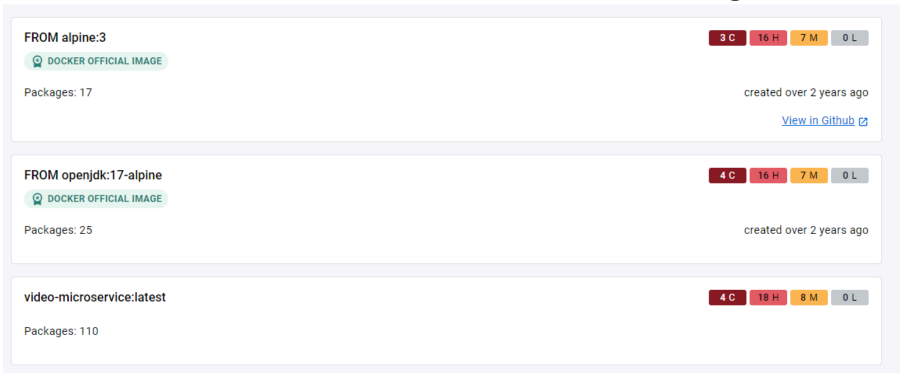
2.1.4 Quality Assurance - Docker Scans

Using docker scout I have scanned each of the images used in my docker compose solution. The following table states the number of vulnerabilities before and after any action taken, this is done in the form of an image taken from docker scout. They are formed of a number of **C**ritical, **H**igh, **M**edium and **L**ow vulnerabilities.

Image Name	Vulnerabilities before action	Actions taken	Vulnerabilitie s after action
bitnami-kafka:3.5	1 C 1 H 1 M 34 L	Moved to bitnami-kafka:3.6.1 - resolving the 1 critical issue. The high criticality issue cannot be resolved without a package update to Jetty.	1 H 0 M 34 L
mariadb:11	2 C 28 H 11 M 11 L	Most of the issues for this image come from the use of stdlib No later versions of mariadb exist to move to, however, I could roll back to an earlier version of mariadb that has less vulnerabilities, but I have opted not as I don't fully understand the impact this may have.	2 C 28 H 11 M 11 L
Video-microservice:latest	4 C 18 H 8 M 0 L	None - See below	4 C 18 H 8 M 0 L
trending-hashtag-microservice:latest	4 C 18 H 8 M 0 L	None - See below	4 C 18 H 8 M 0 L
trending-hashtag-microservice:latest	4 C 18 H 8 M 0 L	None - See below	4 C 18 H 8 M 0 L

For my own microservices, the scans all look like the image below. As you can see, my implementation has introduced 2 high criticality vulnerabilities, which come from the use of netty in the microservice runtime. This is something that I am not going to attempt to resolve as I dont have the knowledge or time.

Breakdown for the video-microservice image



FROM alpine:3 — 3 C 16 H 7 M 0 L
DOCKER OFFICIAL IMAGE
Packages: 17 — created over 2 years ago
View in Github

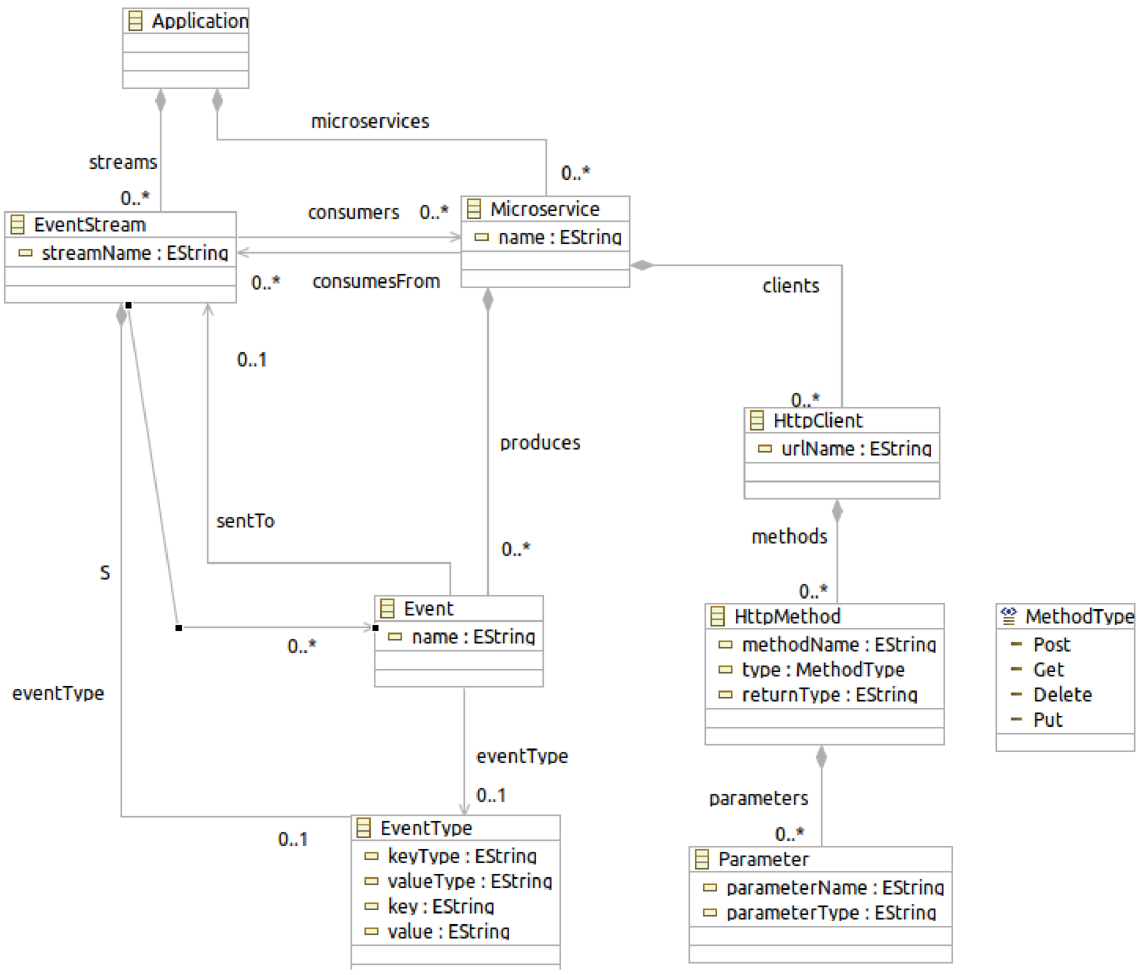
FROM openjdk:17-alpine — 4 C 16 H 7 M 0 L
DOCKER OFFICIAL IMAGE
Packages: 25 — created over 2 years ago

video-microservice:latest — 4 C 18 H 8 M 0 L
Packages: 110

The rest of the vulnerabilities come from alpine:3, an official image that is automatically applied as a dependency of the generated image. alpine:3 is the most up to date version of that image **[3],** I will not be doing anything about this.

2.2.1 Metamodel -

I've produced the following metamodel to describe the microservices produced in the application. It serves as a model for defining models of microservice applications like the ones designed in the earlier questions. The following class diagram represents the model and the relationships between its elements.



Application - There can only be one application in any given model and it is the root element of the model. The application holds the Microservices that make up the solution and the EventStreams (Kafka topics) that the application needs.

Microservices - An application may have zero or many Microservices, these unsurprisingly represent micronaut microservices. Each one has:

- A name, e.g. video-microservice
- A reference to zero or many **EventStreams** in the consumesFrom relationship.
- A variable "clients" that holds zero or many **HttpClient** objects
- A variable "produces" that holds zero or many **Event** objects. These are analogous to Kafka records and hold a reference to an EventStream (Kafka topic) that the record is sent to.

EventStreams - The application can have zero or many **EventStreams**, these are analogous to Kafka's topics, e.g. video-watched from vm, as these topics are owned by and created in Kafka. I made the decision to make the application the owner of them as I disliked the idea of representing kafka as a node in the metamodel. The EventStream also holds the reverse of Microservice's consumesFrom relationship, this represents the consumers of an EventStream.

EventType - Both EventStream and Event objects hold an EventType object, this represents the Kafka record key and value pair and their types. This information can be used later for setting up Kafka topics and records.

HttpClient, HttpMethod, MethodType and Parameter - A Microservice can have zero or many **HttpClient** which are analogous to a micronaut controller. These HttpClients expose zero or many **HttpMethods** on the HttpClient's url. Each of these has a method name, return type and a http method type (POST, GET, DELETE, PUT). These http method types are represented by an enumeration of the four method types that is used by the HttpMethod object. This enumeration ensures that only valid http methods types can be attached to the methods, a choice that I believe reduces the chance for errors in the model to text transformations later on.

Each one of these methods may take zero or more **Parameters**, each of which have their own name and type.

Earlier in the design of this metamodel I debated reducing the amount of modelling for the HttpClient and its methods, originally not including the Parameter object, however, later on during the model to text development I added this in to produce a fairly comprehensive HttpClient to Java code template that will produce detailed skeleton code for the HttpClient instances in the form of Micronaut controllers.

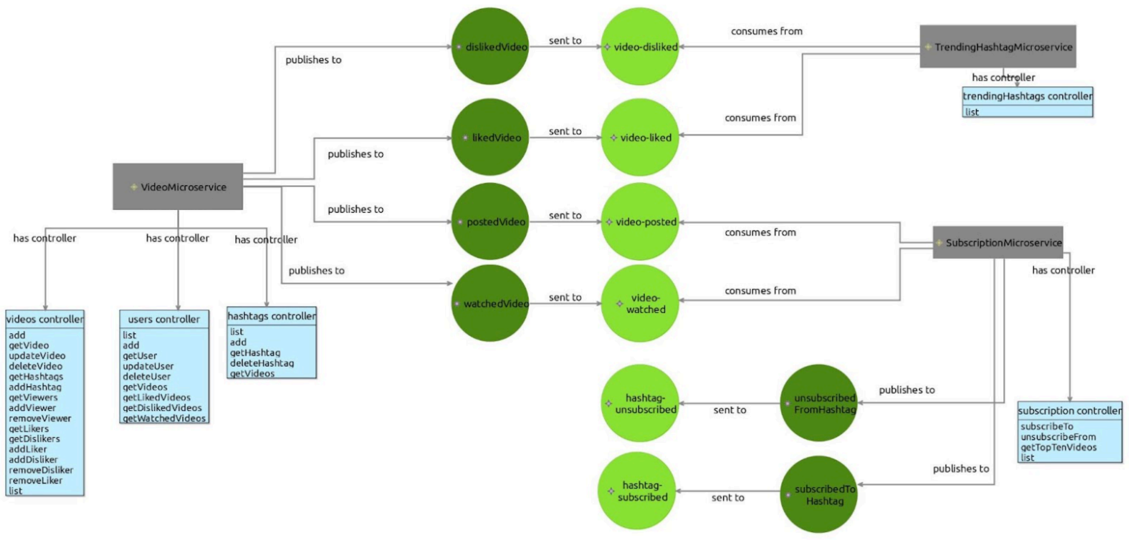
Usage of the model - Users of the model should find it easy to plan new microservices in a way that provides enough detail for those implementing the model while still being abstract enough to not slow down any design.

2.2.2 Graphical Concrete Syntax -

Using the graphical concrete syntax - The design project for the syntax can be found under modelling/metamodel/design.

- Prior to creation of a diagram the Y3887302.ecore must be registered as an EPackage (right click -> register EPackages).
- Then from the design project, open the project.aird file and add the microserviceArchitecture metamodel, and add the acme.model from the filesystem.
- Enable the viewpoint
- Create a new diagram, selecting the top level Application node from the model.

The following is a diagram produced from the model stored at modelling/models/acme.model, with some manual movement of elements to make the diagram more presentable. The diagram is organised with the three microservices (grey) and their HttpClients / controllers (blue) around the Events / Kafka records (dark green) and the EventStreams / Kafka topics (light green).



Syntax for Microservices - Microservices are represented by a simple rectangle with the microservice name as its label. Each microservice has many relationships to HttpClients, Events and EventStreams. These are automatically created using arrows set to and from the source / target element, each one of these arrows are labelled with the name of the relationship or association from the node.

Syntax for Kafka EventStreams and Events - Kafka EventStreams are represented by light green circles on the diagram with the name of the topic as its label. Events are represented by dark green circles, they also have a relationship to the EventStream, represented by an arrow showing where the Event is being sent to. Both of these elements are represented by different kinds of green to indicate they are related as they are both related to Kafka constructs.

Syntax for HttpClients (Micronaut Controllers) and HttpMethods - My metamodel includes a detailed breakdown of HttpClients, their methods and the methods parameters. If I were to include each of these details as an individual node, the diagram would become far too bloated to be useful and the other details of the microservice would be dwarfed. Therefore, I've simplified the presentation of the HttpClient to a container holding each of the methods, not showing their parameters or type.

Earlier in design I tried to split the methods into four sub-containers, each holding the methods for each method type (GET, POST, DELETE, PUT), but I could not get this working nicely.

Strengths -

- Controllers are easy to read and provide an easy to comprehend overview of the methods and functionality provided by the microservice.
- The diagram is clear, and it is easy to follow where events are sent to and consumed from.
- By default, all graphical elements are sized too small to see the text on the shape. I've made sure that all elements in the model are resized from the default size to make them more readable, eliminating the need for the user to manually resize elements.
- Groups of colours help to distinguish the different types of elements, and similar colours indicate similarity in domain (Kafka Event and EventStream).
- All labels match the names used in the metamodel, which may help users who worked on the model comprehend the visual syntax.

Potential weaknesses -

- In the future, if I were to create a graphical editor for this syntax, it may be difficult to edit controllers and add methods as the HttpMethod objects are placed inside the HttpClient (controller) objects at the time the graph is generated, making it potentially difficult to implement a "drag and drop" style editor to add methods to that client.
- Similarly, if I were to produce a graphical editor, there would be no way to add parameters to the HttpMethods as I made the decision to not include this detail in the diagram. Therefore it would not be as powerful as building a model in the Eclipse exceed editor.
- It may be better to use a different image for EventStreams (topics) as they look slightly too similar to Events, the original intent was for them to be similarly coloured to represent them being Kafka related objects, but maybe they are too similar.
- There is no way to view the parameters or method type (GET, POST, DELETE, PUT) of a http method.

Overall, the syntax serves its purpose of representing a model in a graphical way, however, reflecting on it, I would probably look at differentiating some of the Kafka objects a bit more, and maybe using other shapes or adding icons to blocks to improve visual comprehension. I would also look into a nice way to represent methods in more detail, perhaps, simply showing the number of parameters, or the names of them. Alternatively, they could be represented as their full method signature, e.g. *methodName(param1type param1name, param2type param1name).*

2.2.3 Model Validation

Running the constraints - Load and use the launch configuration in modelling/configs/ model-constraints.launch to run the constraints in Eclipse Epsilon under “Run As” -> “run configurations” -> model-constraints. If this fails, ensure that the configuration is pointing to the correct model and the EPackage for the metamodel is registered and that model is present.

Constraints and Critiques - I have produced four checks, each of which is based on the requirements in the exam and is made of one or more constraints or critiques. Constraints are requirements that the model *must* adhere to. Whereas critiques are advisory checks that the model *should* adhere to, but may not. I’ll briefly explain them and their rationale here.

- **Check 1** - *Constraint* - “There should be at least one microservice”
  - Implementation: This runs one constraint to checks that the top level Application object has at least one microservice attached to it.
  - Rationale: It does not make sense to model a system with no microservices, or else it could only be an application of EventStreams which are not interacted with i.e. there is no code.
- **Check 2** - *Critique* - “Every event should be used in at least one event stream”
  - Action: Checks that every Event object in the model has its “sent to” property defined.
  - Rationale: It does not make sense to define an event without sending it to an event stream. So using a critique this is flagged up to the user, who could consider where to send the event.
- ***Check 3*** - *Two Critiques* - “Every event stream should have at least one publisher and one subscriber”
  - Action: The two critiques check that every EventStream object in the model has a publisher and a subscriber included in their eventsRecieved and consumers references.
  - Rationale: While event streams can exist without receiving any events or being read from. Ideally an event stream should have both. Without them, it does bring into question whether the stream is redundant, so flagging it up as a critique to the user allows them to take action on this.
- ***Check 4*** - *Two Constraints* - “Every microservice needs at least one health resource using the HTTP GET method and taking no parameters, for reporting if it is working correctly”
  - Action: The two constraints check that each microservice has a HttpClient and then runs a second constraint against that HttpClient to make sure that there is at least one no-parameter, GET method. This is done by iterating over all methods to find at least one that has no parameters and its MethodType is GET.
  - Rationale: A health method such as this can be used to confirm that the microservice controller is working and http requests can be answered. This is enforced as a constraint to guarantee some way to check a microservice controller. These health methods could be used in a docker health method to confirm the container is running correctly.

Overall, these constraints and critiques offer a quick way to sanity check that a model is rational and that common mistakes are flagged up to the user. When ran against my current model, it flags two critiques as the hashtag-subscribed and hashtag-unsubscribed topics are not consumed by any microservice. This is expected.

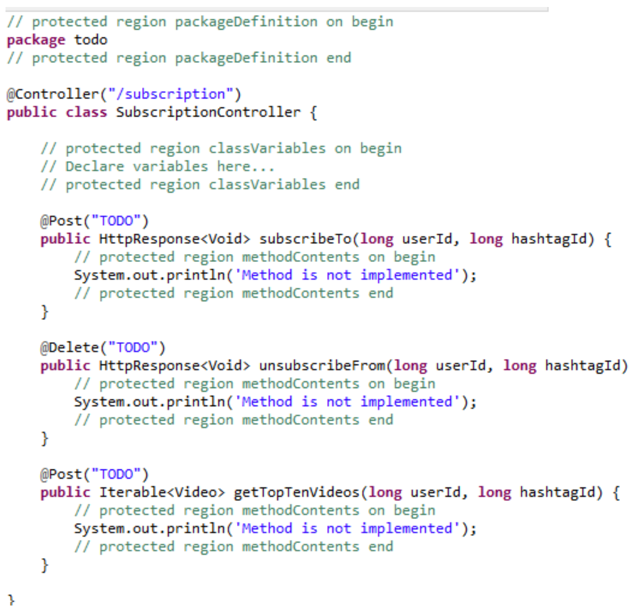
2.2.4 Model-to-Text Transformation

Running the model-to-text transformations - Load and use the launch configuration in modelling/configs/Y3887302.launch to run the EGX file that applies the model to text transformations. Do this in Eclipse Epsilon under "Run As" -> "run configurations" -> Y3887302. If this fails, ensure that the configuration is pointing to the correct model and the EPackage for the metamodel is registered and that model is present.
Note: My EGX file does not seem to work when ran on a network share, such as the ones used on university PCs e.g. //userfs/… due to a file not found error. But it runs fine when ran on a local drive e.g. C/: See this eclipse bug here.

Model-to-text transformations - I have produced an egx file (modelling/m2t/Y3887302.egx) that runs two transformations for every microservice, producing multiple micronaut controller classes, one for every HttpClient, and also a producer class for any microservice that produces an event to a topic. I'll discuss each below.

client2java - During my implementation I found the most tedious task to be the implementation of controllers. So I have produced a transformation that takes a HttpClient and produces a class called <microservice_name>Controller.

- Annotates the class with an @Controller annotation with the url name stored in the HttpClient object.
- Adds protected regions to put the method implementation in and another for the class variables (e.g. repositories, clients etc.) and for the package definition.
- Produces a method for every HttpMethod inside the HttpClient marked as public with its return type obtained from the HttpMethod object.
- Annotates each method with either @Get, @Delete, @Post, @Put depending on their MethodType attribute. The url for these annotations is marked as TODO, as the metamodel does not contain the information to fill this in, so it is left to the user to do this manually.
- The Parameters of the HttpMethod are filled in, including their types and name, each separated by a comma.
- The necessary dependencies are included.


```
// protected region packageDefinition on begin
package todo
// protected region packageDefinition end

@Controller("/subscription")
public class SubscriptionController {

    // protected region classVariables on begin
    // Declare variables here...
    // protected region classVariables end

    @Post("TODO")
    public HttpResponse<Void> subscribeTo(long userId, long hashtagId) {
        // protected region methodContents on begin
        System.out.println('Method is not implemented');
        // protected region methodContents end
    }

    @Delete("TODO")
    public HttpResponse<Void> unsubscribeFrom(long userId, long hashtagId)
        // protected region methodContents on begin
        System.out.println('Method is not implemented');
        // protected region methodContents end
    }

    @Post("TODO")
    public Iterable<Video> getTopTenVideos(long userId, long hashtagId) {
        // protected region methodContents on begin
        System.out.println('Method is not implemented');
        // protected region methodContents end
    }

}
```

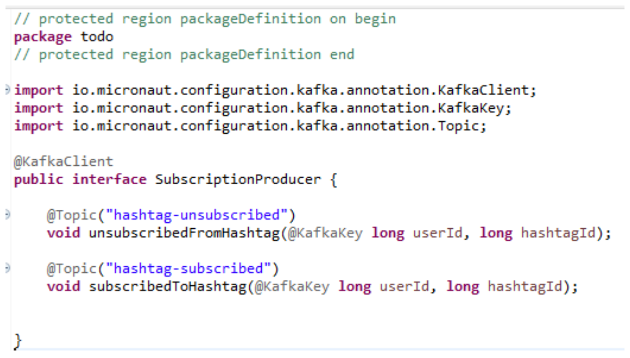
See the image to the right for an example of this transformation run against the subscription microservice HttpClient.

When this is ran as part of the EGX file all HttpClient objects are gathered from each microservice and this transformation is ran against them, placing the output in a file called <microservice_name>Controller.java in the m2t_output folder in the relevant microservice folder.

producer2java - My second transformation takes every instance of the Microservice from the model and produces a single <microservice_name>Producer Kafka producer interface for each microservice for each one the following is done:

- The interface is tagged with @KafkaClient
- For each topic the microservice is subscribed to, the template makes a method signature.
- The method signature is annotated to read from the relevant topic using the @Topic annotation with the name of the EventStream from the microservice's consumesFrom attribute.
- The methods parameters are filled in by using the EventStream's EventType, which holds the Kafka key and value.
- The necessary dependencies are included.

See the image to the right for an example of the template ran against the sm.


```
// protected region packageDefinition on begin
package todo
// protected region packageDefinition end

import io.micronaut.configuration.kafka.annotation.KafkaClient;
import io.micronaut.configuration.kafka.annotation.KafkaKey;
import io.micronaut.configuration.kafka.annotation.Topic;

@KafkaClient
public interface SubscriptionProducer {

    @Topic("hashtag-unsubscribed")
    void unsubscribedFromHashtag(@KafkaKey long userId, long hashtagId);

    @Topic("hashtag-subscribed")
    void subscribedToHashtag(@KafkaKey long userId, long hashtagId);

}
```

Organisation of the code - All code is sent to a folder called m2t_output inside of the relevant microservice. e.g. <microservice_name>/m2t_output/src/main/videos/ <controller/events>. The user of the generated code is expected to copy and paste the code from this location to the corresponding location in their src/main and adapt it to their liking. They are kept separate to avoid overwriting some of the non-protected regions of the code without the user's knowledge. This way the user must put their implementation back into the folder and overwrite it knowingly when running the generator.

Usage of the code - I have generated the code for all three microservices, which can be found in the m2t_output folders for each microservice. I used this code to help implement the final microserve, the subscription microservice, which helped speed up the controller development.

Alternative considerations - While I have only implemented two transformations, I believe that there is enough complexity in them for this question, especially the client2java transformation. However, I have considered and discounted some other alternatives during development.

- I would have liked to attempt an inheritance based approach to using the generated code. e.g. have my implemented controllers inherit the method signatures from an abstract class which are implemented
- With more time I was planning on implementing a third transformation listener2java that would produce Kafka listener classes for each topic that a microservice listens from. However, this proved to be a bit more complicated than the producer2java as my metamodel did not fit the task as easily.
- I would have also liked to have tried to automate some of the creation of the docker compose sections for each of the microservices, providing a db for each and filling in the environment variables and names as needed. However, to do this I would have had to go back to my metamodel and add more object and detail, which I was too far along in the assessment to do.

References

[1] S. Brown. C4 models. [Online]. Available: https://c4model.com/

[2] Splunk Maintainers. Splunk. [Online]. Available: https://www.splunk.com/en us/blog/learn/microservices-load-balancing.html#:~:text=At%20its%20core%2C%20load%20balancing,service%20instances%20are%20used%20effectively

[3] Docker. Alpine Image. [Online]. Available: https://hub.docker.com/_/alpine