Y3874118

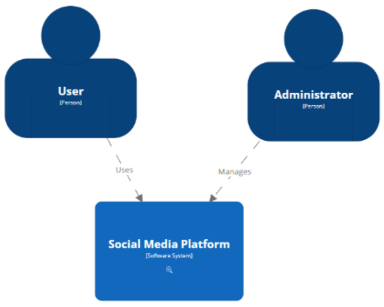
Engineering 2: Automated Software Engineering Report

Part 1

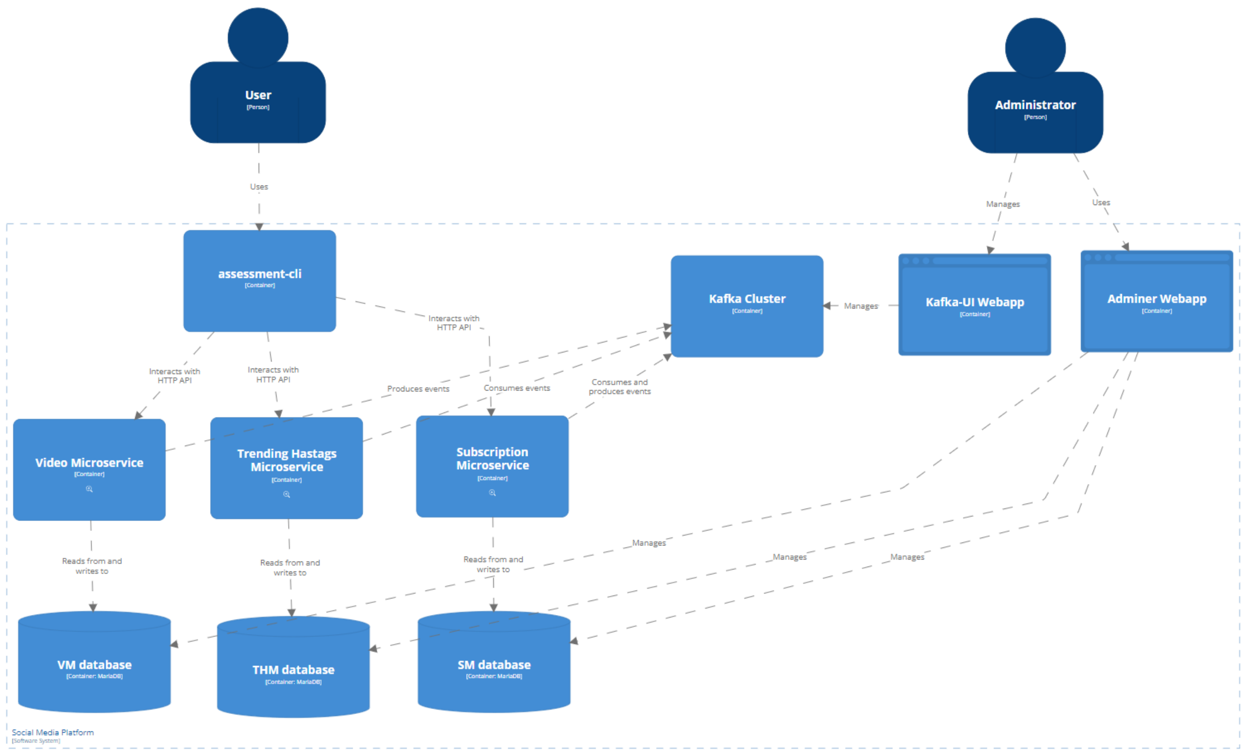
For this part, I assumed I do not have to talk about and include the HRM I developed in part 2.

2.1.1 Architecture:

At the contextual level, users and administrators are interacting with the social media platform. User can post videos, view them, like and dislike them, browse the currently trending hashtags, subscribe, or unsubscribe to hashtags, browse recommended videos. Administrators manage the system through secure channels.



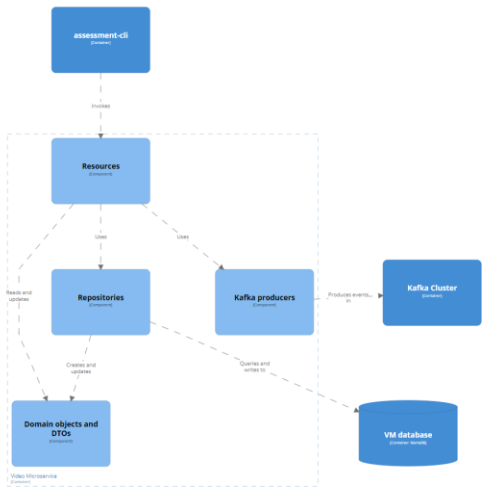
At the container level, the social media platform is distributed into microservices designed for scalability, each of them being responsible with a distinct facet of the application functionality. The four microservices are the Video Microservice (VM), the Trending Hashtag Microservice (THM), the Subscription Microservice (SM). Communication between microservices happen exclusively using a kafka cluster composed of three kafka containers that split the load of the messages. Each microservice uses its own database to promote modularization, fault isolation or data isolation. The user interacts with the social media platform through a Command Line Client (CLI). The CLI can forward the user's request to the corresponding microservice and display the result, acting as the front end of the application. The administrator manages resources through two webapp applications, for managing the kafka cluster or the databases respectively.

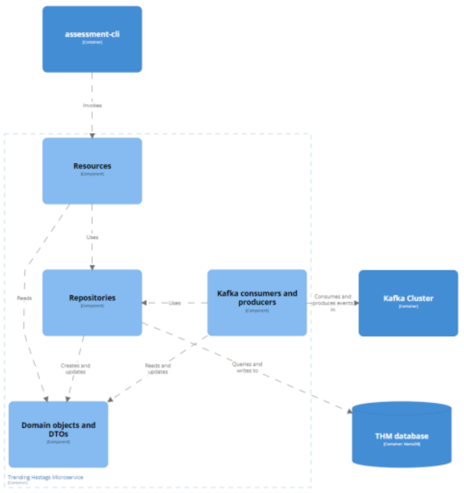


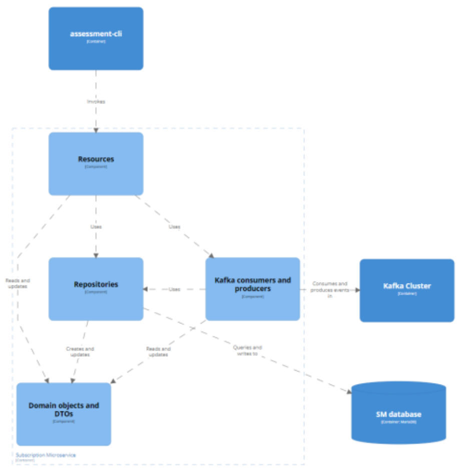
At the component level, the microservices are similar, but with slight implementation differences bases on their purpose or their dependencies. Each microservice has repositories for querying its database, domain objects and Data Transfer Objects (DTOs) for representing said data, and kafka producers or consumers for interacting with the kafka cluster.

Every microservice exposes at least one resource for the CLI, and those can either retrieve information from the database through the repositories or they can be used for updating domain objects.

VM: The entry point for creating users, posting videos, listing videos, and viewing/liking/disliking videos. It uses kafka producers to forward these events to various topic. Therefore, it's resources make use of repositories for reading/writing data, and of its producers for publishing records.
THM: Provides a list of top 10 most liked hashtags in the last hour. Its subscribers and publishers make use of the repositories to maintain the list updated in the database, computing it from the records published in the kafka cluster by the VM.
SM: Uses kafka assets to replicate users, videos and hashtags created in the VM database by subscribing to the events published by the VM. Provides front facing resources to for users subscribing and unsubscribing to hashtags. Therefore, in this microservice, both kafka streams and HTTP resources make use of the repositories. Subscription records are produced in the kafka cluster for further use by other microservices.







The scalability of the system is offered by its modularization, separation of concern and its distributed architecture. With an increase in user demands, the architecture can scale a subset of its functionalities, addressing only the overloaded subpart of the system, while leaving optimally used resources at an appropriate level to minimise cost. For example, if a lot of new users are interacting with the app, there is a lot of need for computing new users, new videos, new interactions with the videos and so on, but the demand of calculating trending hashtags or hashtag recommendations doesn't increase at the same rate. In this case, it is very beneficial to be able to only scale by only adding new VM nodes, instead of having to deploy an entire monolith architecture.

In addition, the architecture is suited for different types of scaling of each microservice. The VM and SM are more front facing, offering more resources to the users and those resources must also maintain databases. To minimize response time, the best suited strategy is scaling out, adding more nodes for better parallel processing of data. More requests can be handled synchronously, offering a better user experience. In the meantime, the THM is more appropriate for scaling up to more powerful nodes that can process data faster, computing trends quicker. A monolith architecture would not have this kind of flexibility. Elasticity is another benefit of the chosen architecture. Unexpected variations in loads are easier to deal with, because small, isolated nodes are quick to spin up and shut down on demand, again minimizing upkeep costs.

New requirements are easily enabled by the added maintainability of the distributed architecture. Features that make use of existing records can quickly be adopted in a new microservice or as part of the small, readable codebase of an existing microservice. This is significantly easier than adopting a new feature in a monolith architecture where dependencies are harder to detect. For example, adding a new recommendation feature for finding hashtags with similar subscribers would be easily facilitated by the records already created by the SM. A microservice for processing the records produced by the SM to expose such a resource would fit very well in the architecture and would. In another example, if we wanted to add the possibility for the user to download videos, this could be again implemented as part of a new microservice, which would be deployed on machines with high network bandwidth and good encoding capabilities. New features are also easy to add as part of existing microservices because the codebase is relatively small, and the separation of concerns means that developers have a lot less dependencies to worry about. For example, adding the possibility to add a comment to an existing video would be easily deployable as part of VM's codebase. In all the examples, there are good options for developing the architecture to meet new user requirements, while also adhering to the original design, such that future additions are just as easy.

2.1.2 Microservices:

Video Microservice (VM):

The video microservice handles user creation, video creation, viewing/liking/disliking videos and listing videos by user and hashtag. To enable all those resources, I created three domain entities, each of which is accompanied by a JPA repository with access to the database, and a DTO for secure communication to the CLI or for kafka stream serialization.

The User entity has id and a username fields, a set of videos representing the videos this user posted, which is a OneToMany relationship to Video, and a set for each of the watched, liked and disliked videos, which are ManyToMany relationships to the Video entity. Similarly, on the other end, the Video entity also defines these relationships to the User, in addition to a id field and a title field. It is important to note that the User is normally the owning entity of the relation, so the resources will normally be updating users not videos. The OneToMany relationship to Video is an exception because in this scenario the Video has to be the owning entity, but this works well because it is natural to add the user of a video when a video is created, instead of the other way around. The Video entity also has a ManyToMany relationship to the hashtag entity to store the hashtags of a video. The hashtag entity uses the fact that hashtags are unique to define a findByName() function in the HashtagRepository, which comes in handy when the user creates a video with hashtags and we must first check if those hashtags exist without having an id for them. This is only applicable for hashtags because I don't consider usernames or video titles as unique.

The VM has a lot of front facing resources for creating and updating these entities. I have created three separate http controllers to handle user requests. The main one of those is the UserController which handles all user specific requests through the following endpoints on the "/users" path:

- GET "/": Returns all the users in the DB.
- POST "/": Creates a user specified in the body of the request.
- GET "/{userId}": Returns user info for user with userID (doesn't include sensitive database specific information, like the id).
- POST "/{userId}/videos": Adds a video for the user.
- GET "/{userId}/videos": Returns the videos posted by the user
- GET "/{userId}/viewed": Returns the videos viewed by the user. Similarly for /liked and /disliked
- PUT "/{userId}/viewed/{videoId}": Adds video to set of videos viewed by the user. Same for /liked and /disliked

The HashtagController handles hashtag specific request on the"/hashtags" path:

- GET "/": Returns all the hashtags in the DB
- GET "/{hashtag}": Returns hashtag info for hashtag with hashtagId (doesn't include sensitive database id).
- GET "/hashtagId}/videos": Returns all the videos posted with this hashtag

The VideosController handles video specific requests on the "/videos" path:

- GET "/": Returns all the videos in the DB

Due to the relational MariaDB database, these resources are enough to satisfy all the requirements of the VM, since changes are reflected across all tables as mapped by the relation. For example, creating a video also updates the "videos" field of the User entity. The repositories are also augmented with the required "Join" annotations to make database fetching include related tables.

VM also includes a VideoProducer interface which is a kafka publisher that publishes messages to the following topics:

- video_posted: When a video is posted
- video_watched: When a video is watched
- video_liked: When a video is liked
- video_disliked: When a video is disliked
- user_created: When a user is created

The microservice uses Data Transfer Objects for all communications, except for the 'helthcheck' "/" resources of each controller. The Data Transfer Objects don't carry the id of the item, so they are more secure. In a real world application, the "/" paths would be secured or perhaps wouldn't exist but for the purpose of this assessment they are required because all resources are ID dependent and the CLI needs a way to access this information.

Trending Hashtags Microservice (THM):

This microservice handles the creation of a list of the top 10 trending hashtags based on the likes received the videos. To achieve this, THM uses two kafka streams consumers that compute the top 10 in a rolling window of one hour, a kafka subscriber that saves the list in the DB, and a kafka producer that is scheduled to update the rolling window every minute, even if there are no new liked videos. In more details, these are the workflow of THM:

1. HashtagsLikedStream subscribes to the video_liked topic, where VM is publishing messages. For each video liked, this streams a new record to "hashtag_liked" for each of the hashtags of that video. The key here is in the flatMapValues function which enables turning one "video_liked" record to multiple "hashtag_liked" records.

2. TrendingHashtagsStream subscribes to “hashtag_liked” and applies a grouping by key, followed by a rolling window of 1 hour, followed by a custom aggregator TopAggregator which aggregates the windows into maps of type <String, Integer> where the String is the name of the hashtag and the Integer is the number of likes. At this point the stream hold the information needed, as the key is the rolling window and the value is a map with all the hashtags and their number of likes. I apply two more operations to change the type of the key to the serdeable WindowIdentifier defined by THM and two sort the map and get only the top 10. After that the record contains the information needed and I publish it to the “trending_hashtags-by_hour” kafka topic.
3. The TrendingConsumer listens to “trending_hashtags-by_hour” upon receiving a record, parses it into a Trend entity. This entity holds the top hashtags and the hour, and is managed by TrendsRepository interface.
4. With the current approach, the rolling window in TrendingHashtagsStream wouldn’t be updated if users stop liking videos for a long time. This is why I created the HashtagLikedScheduler singleton which produces dummy records to the “video_liked” topic to keep the rolling window rolling. This approach is far from ideal because it pollutes the “video_liked” topic with lots of dummy messages that other microservices would have to learn to interpret. I could have been using a punctuator in TrendingHashtagsStream to schedule the update of the rolling window, but this required a lot more kafka customization. Alternatively I could have published the messages to a “internal” topic of THM and merge those messages into the “hashtag_liked” during processing.
5. Finally, a TrendsController exposes a GET method for retrieving the latest trend at “/trending/latest”.

Subscription Microservice (SM):

This microservice is more like VM, as it revolves around the use of the relational database to fulfil its requirements. By subscribing to “user_created” and “video_posted” I recreate all the records of the VM database but drop irrelevant information for this service. However, in SM’s database, Users have a new ManyToMany relation to hashtags which represents its subscriptions. The viewedVideos relation is also maintained because this is the metric used to recommend videos and we use the “video_viewed” topic to update the number of views of a video. With these three consumers defined to keep the two services in sync, SM provides five resources exposed by the SubscriptionController on the path “/user-subscriptions”:

- GET “/”: Returns all the users in SM’s database (used as a healthcheck)
- GET “/{userId}: Returns all the hashtags a user is subscribed to
- PUT “/{userId}/subscribe”: Subscribes the user to the hashtag specified in the body
- PUT “/{userId}/unsubscribe”: Unsubscribes from the hashtag specified in the body
- GET “/{userId/recommandations}”: Gets the recommendations for a given user and hashtag

The last resource is computes the list on demand, by iterating through a hashtag’s videos and returning a sorted list of the top 10 most viewed videos in that hashtag that the user is not the owner of and hasn’t seen yet.
SM publishes events to “user_subscribed” and “user_unsubscribed” which hold records of the userId and the hashtag entity they subscribed to.
The implementation of SM produces a very high risk of database records between it and VM falling out of order, if for example kafka messages are processed in the wrong order. This reduces the “isolation” of this microservice significantly, and ideally such a risk would be mitigated. One solution would be to remove @GeneratedValue annotation of the id and rely on the records of the topic to carry that information and replicate it more accurately in SM’s database. However, this would require the kafka topics to carry more database specific information, which I have been avoiding for the most part of the project, and so for the purpose of the assessment, I have accepted this risk.

Assessment-CLI:

The CLI provides commands for every resource of each microservice, except for the ‘health check’ resources of THM and SM. Since all the request paths are defined using IDs, we must make extensive use of the “/users” “/videos” and “/hashtags” resources of the VM. These are mapped by the following commands: get-users, get-videos, get-hashtags. In addition, the rest of the commands are:

- add-user: Adds a user
- add-video: Adds a video for a user
- get-user-videos: Gets a user's videos
- get-hashtag-videos: Gets a hashtag's videos
- view-video: User views a video
- like-video: User likes a video
- dislike-video: User dislikes a video
- get-trending-hashtags: Gets the current top10 trending hashtags
- subscribe: Subscribe a user to a hashtag
- unsubscribe: Unsubscribe a user from a hashtag
- get-recommended-videos: Lists the top10 recommanded videos for a given user and hashtag
- get-all-recommended-videos: Lists the top10 recommanded videos for a given user and all its subscriptions
- get-recommended-hashtags   For a given hashtag, lists the top 10 hashtags its subscribers subscribe to as well
-

2.1.3 Containerisation:

The microservices are packaged and deployed using Docker and I have provided the associated “compose.yml” file for orchestration. In this file, I define the four microservices which expose ports 8080, 8081, 8082, 8083 (VM,THM,SM,HRM), a MariaDB container which is the database (additional DB are defined in microservices/sql), The kafka cluster with 3 kafka containers called kafka-0 kafka-1 and kafka-2. In addition, the two containers for administrators are also defined here, kafka-ui for kafka management ad adminer for database management. “Internal” containers like the kafka clusters and the database do not have exposed ports and can only be accessed from inside the network.

In order to run the production environment of the system, follow these instructions from inside the microservices folder of the submission:

1. Use “gradlew dockerBuild” to build the docker images
2. User “docker compose up -d kafka-0 kafka-1 kafka-2 db” to bring up the kafka cluster and the db
3. Run the following commands to create all the kafka topics, replacing TOPIC_NAME with each of the following: video_posted, video_watched, video_liked, video_disliked, user_created, hashtag_liked, trending_hashtags_by_hour, user_subscribed, user_unsubscribed

   ```
   docker exec -e JMX_PORT= microservices-kafka-0-1 kafka-topics.sh --bootstrap-server kafka-0:9092 --create
   --topic TOPIC_NAME --replication-factor 3 --partitions 6
   ```

4. Run docker compose up to bring everything else up

The solution can scale through docker compose’s functionality for scaling services. For example, in the compose file we can specify a `scale` parameter to multiply the number of instances of a microservice that are being brought up. However, docker compose is limited and using more complex tools like Kubernetes would allow for better coordination between containers, specifically by enabling load balancing. Without load balancing, the multiple instances of a microservice wouldn’t receive an equal number of requests and some of them might get overloaded while others are being underused.

The current compose file offers fault resilience in the form of automatic restarts for crashed services. To improve fault tolerance, healthchecks could be performed on the startup of each service. For example, with the current setup, if we were to bring the entire environment up at once, the microservices would fail to start because the MariaDB database takes a while to initialize. Using healthchecks, we could instruct the microservice containers to wait for the MariaDB container to signal that it’s ready to process requests.

The project can also be execute in “development” mode through the compose-dev file provided in the same folder.

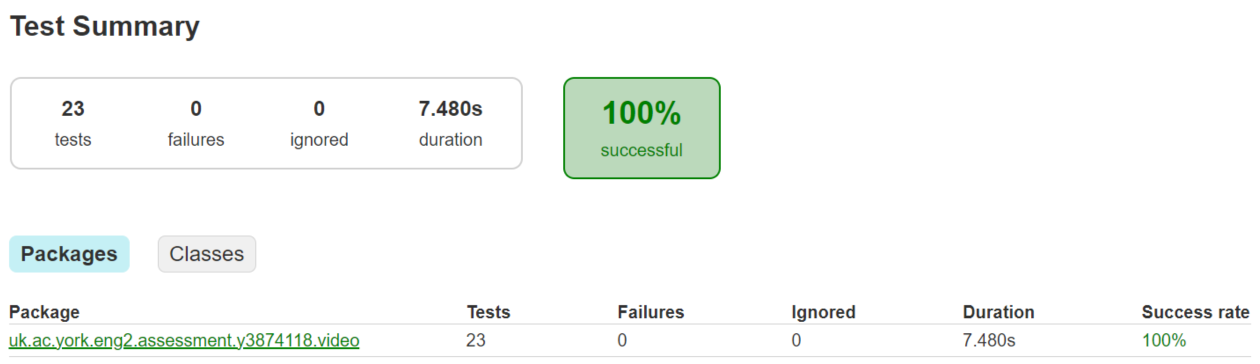
2.1.4 Quality Assurance:

Each microservice is tested, with the most common scenarios being considered. Where possible, I've considered both interactive-based and state-based testing.

Video Microservice (VM):

I performed tests for all of the available resources, creating a controllerTest class for each of the microservice's controllers. For each of the resource, I tried to incorporate common failing scenarions, like invalid IDs in the request path, or idempotent actions like viewing a video twice not having an impact on the state of the application. For the kafka producer, I created a Mock VideoProducer inside the UserControllerTest class, because this is the only controller that produces records. This is interaction-based testing, and it successfully tested how the controller interacts with the producer, not nothing more. This is why I also opted to implement a KafkaProducerTest, which performs state-based testing of the producer. These asynchronous interactions are slower than the interaction-based testing, but they verify that the application is interacting correctly with the kafka cluster. Because the time trade-off was not significant enough, I implemented state-based testing for all of VM's producers.

The test suite comes up to 23 successful tests for this microservice:

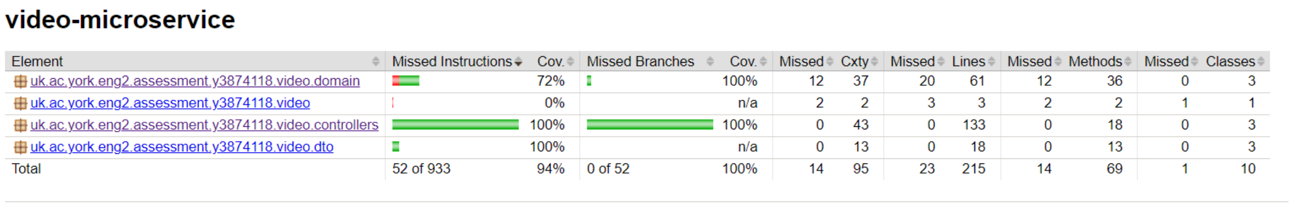


Test Summary

23	0	0	7.480s	100%
tests	failures	ignored	duration	successful

Package	Tests	Failures	Ignored	Duration	Success rate
uk.ac.york.eng2.assessment.y3874118.video	23	0	0	7.480s	100%

The code coverage is up at 94%. However, this is not a perfect indication of the code quality. While these tests cover the most common types of failures, and ensure a basic functionality, there could be edge cases and special scenarios that are not tested for. However, during development this test suite proved useful in signalling new bugs, and I haven't been able to find a failing case which is not covered by it yet.



video-microservice

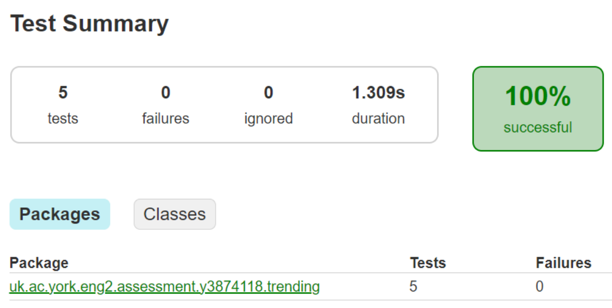
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
uk.ac.york.eng2.assessment.y3874118.video.domain		72%		100%	12	37	20	61	12	36	0	3
uk.ac.york.eng2.assessment.y3874118.video		0%		n/a	2	2	3	3	2	2	1	1
uk.ac.york.eng2.assessment.y3874118.video.controllers		100%		100%	0	43	0	133	0	18	0	3
uk.ac.york.eng2.assessment.y3874118.video.dto		100%		n/a	0	13	0	18	0	13	0	3
Total	52 of 933	94%	0 of 52	100%	14	95	23	215	14	69	1	10

Trending Hashtags Microservice(THM):

Because THM's implementation is highly based on kafka streams, the test suite for it is also quite different. The two main classes of this microservice, the HashtagLikedStream and the TrendingHashtagStream are tested using interactive-based kafka stream testing. For HashtagLikedStream, I create a mock input topic that creates mock records for the "video_liked" topic and then I check if the processing of those records is correct. I ensure that the crucial flatMapValues function produces multiple records from a single input message, and I also test for the correct extraction of hashtag names from the VideoDTO object that is published. For TrendingHashtagStream, I follow the same interactive-based approach and I test for the correctness of the produced Map object.

While this is testing is far from exhaustive, and even though it doesn't test if the microservice interacts correctly with the kafka stream, I consider that it covers the most crucial part of the Trending Hashtags Microservice workflow, and ensuring the correct manipulation of records with kafka streams is the most important thing here.

The suite also includes tests for the TrendsController which exposes the current trending hashtags and it's important to test thoroughly. The total number of tests is 5, and the coverage is up at 86%.

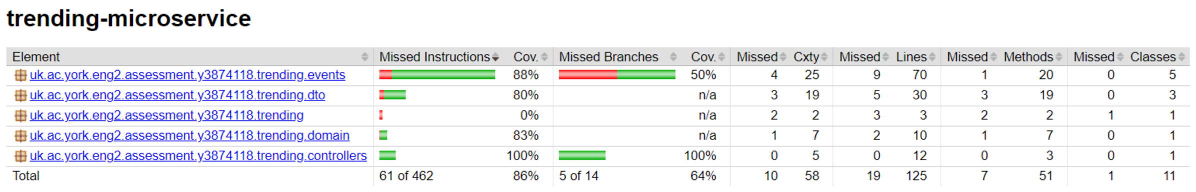


Test Summary

5	0	0	1.309s	100%
tests	failures	ignored	duration	successful

Packages | Classes

Package	Tests	Failures	Ignored	Duration	Success rate
uk.ac.york.eng2.assessment.y3874118.trending	5	0	0	1.309s	100%



trending-microservice

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
uk.ac.york.eng2.assessment.y3874118.trending.events		88%		50%	4	25	9	70	1	20	0	5
uk.ac.york.eng2.assessment.y3874118.trending.dto		80%		n/a	3	19	5	30	3	19	0	3
uk.ac.york.eng2.assessment.y3874118.trending		0%		n/a	2	2	3	3	2	2	1	1
uk.ac.york.eng2.assessment.y3874118.trending.domain		83%		n/a	1	7	2	10	1	7	0	1
uk.ac.york.eng2.assessment.y3874118.trending.controllers		100%		100%	0	5	0	12	0	3	0	1
Total	61 of 462	86%	5 of 14	64%	10	58	19	125	7	51	1	11

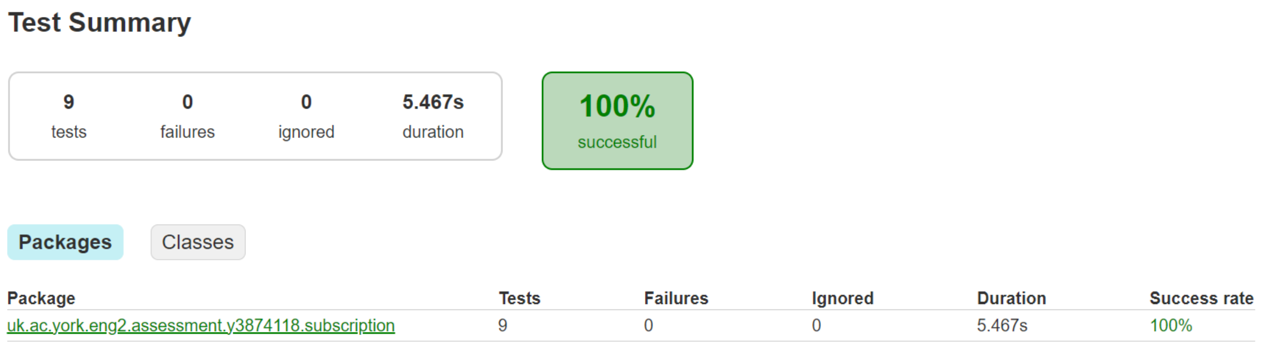
Subscriptions Microservice (SM):

For SM, it is very important to test the kafka consumers which are responsible with the data synchronization between SM and VM. Those are tested inside the SubscriptionConsumer test file, where I apply interactive-based testing to check if the records from the kafka topics are correctly reproduced into the database. The tests don't check the connection between the microservice and the cluster, but the interaction with the database is state-based. Here it is important to check that the users, videos and hashtags are created with the same ID that is passed to the consumer. The tests cannot fully mitigate the risk of SM and VM falling out of sync, because the records are slowly produced one by one. However, I imagine that applying load testing to this microservice, where a lot of records are being passed in quick succession would cause more failures.

For the SubscriptionController, I write test the subscribe/unsubscribe resources and I also cover a complex scenario of multiple users, videos and hashtags interacting with the server to retrieve lists of recommended videos to watch. Interactive-based testing of the producers is also executed here.

Lastly, for the kafka producers of this microservice I also perform state-based testing, making sure that the requests to "/subscribe" and "/unsubscribe" produce the right records in the "user_subscribed" and "user_unsubscribed" topics respectively.

The total number of tests is 9 and the coverage is 89%. In this case, I believe that the completeness of my tests is high, and I think most edge cases are covered, but the lack of load testing for the consumer is the main concern.

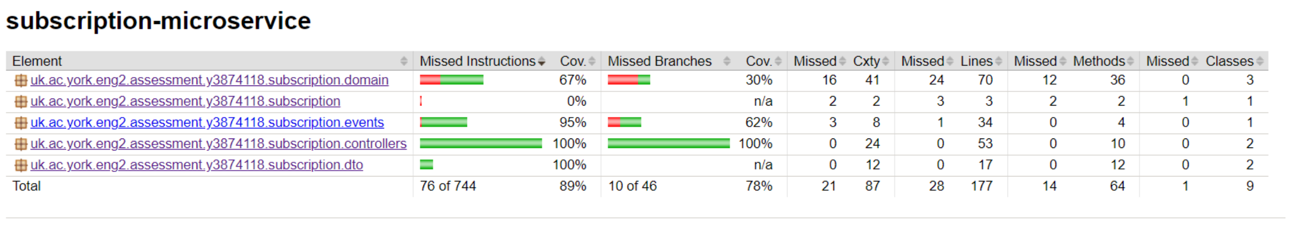


Test Summary

9	0	0	5.467s	100%
tests	failures	ignored	duration	successful

Packages | Classes

Package	Tests	Failures	Ignored	Duration	Success rate
uk.ac.york.eng2.assessment.y3874118.subscription	9	0	0	5.467s	100%



subscription-microservice

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
uk.ac.york.eng2.assessment.y3874118.subscription.domain		67%		30%	16	41	24	70	12	36	0	3
uk.ac.york.eng2.assessment.y3874118.subscription		0%		n/a	2	2	3	3	2	2	1	1
uk.ac.york.eng2.assessment.y3874118.subscription.events		95%		62%	3	8	1	34	0	4	0	1
uk.ac.york.eng2.assessment.y3874118.subscription.controllers		100%		100%	0	24	0	53	0	10	0	2
uk.ac.york.eng2.assessment.y3874118.subscription.dto		100%		n/a	0	12	0	17	0	12	0	2
Total	76 of 744	89%	10 of 46	78%	21	87	28	177	14	64	1	9

Docker inspection:

I have gone over the high severity vulnerabilities and addressed them as follows:

Image: provectuslabs/kafka-ui:latest

This image has a few vulnerabilities related to possible DDoS attacks, specifically a high severity (7.5H0 one called GHSA-xpw8-rcwv-8f8p which is caused by the netty-codec-http2 package. The vulnerability is patched in a release of netty from almost 3 months ago. My system could be at risk of DDoS attacks especially since I've decided to expose the port of the kafka-ui container. The recommended course of action is updating the netty package to the version with the fix. Since this is already a fixed vulnerability, I have decided not to address it myself as part of the assessment.

Another high severity vulnerability comes from the apache.avro package which allows the reader of a deserialized object to consume memory beyond the allowed constraints. This only affects Java applications using Apache Avro so it doesn't concern me currently

Image: mariadb:11

This image has multiple critical vulnerability caused by the stdlib 1.18.2. I am currently using version 11 of this image, and I looked into updating to the latest version, but that vulnerability hasn't been addressed yet even in that release. I cannot address this in any other way.

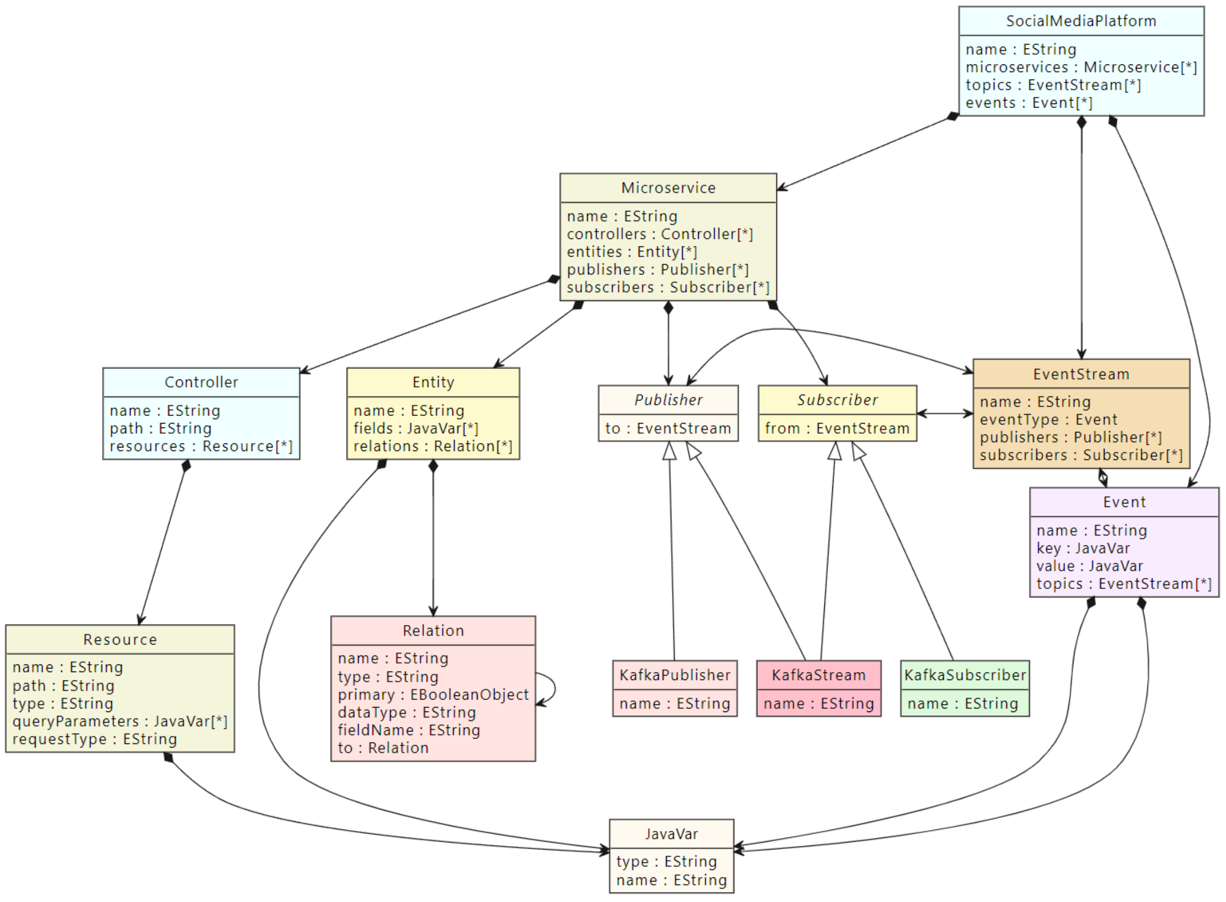
Image: adminer:4-standalone
This image only has very low vulnerabilities.

Image: VM,THM,SM and HRM
This image has the same high vulnerability as kafka-ui, produced by the netty package. I looked into updating the netty package in my build.gradle file but since it is a dependency of micronaut, I couldn't do it without causing conflicts.

Image: bitnami/kafka:3.5
This image couldn't be analyzed by docker scout

Part 2

2.2.1 Metamodel:



The metamodel is as shown above. The SocialMediaPlatform class is designed to be the root instance of models, containing microservices, topics and events.

The Microservice class contains:

- The name of the microservice
- Controllers, which are essentially collections of resources, and map to the java controllers described in Part 1. Each controller has a base path, and when concatenated with its resources paths, it forms the http endpoints of the microservice. On top of a path, resources also define the http method type, queryParameters and body request type. Those are used to uniquely identify the resources a microservices needs to provide.
- Entities, which again map to the java entities described in Part 1, and on an even higher abstraction level, these map to the DB tables. Entities contain fields, which represent columns in the name (e.g. video title for Video or username for User), and relations, which represent mappings to other entities (e.g. videos of a user).
    - Relations have a type, (e.g. ManyToMany or OneToMany), a datatype (User/Video/Hashtag), a fieldname and a boolean attribute called primary which describes weather or this is the "owning" side of the relation (i.e. The entity which owns this relation is the one that would have to be updated)
    - Perhaps non-intuitively, relations map to relations instead of mapping to Entities. This is because an entity could have multiple relations to another entity in micronaut, and relationships are not "mirrored" so we need to be aware of both sides of the relation. Mapping to another relation offers us that information and the parent entity is still accessible in EOL through the .eContainer() method.
- Publishers and Subscribers. This section is slightly trickier. Both of the publisher and subscriber classes are abstract, and I have defined the KafkaPublisher and a KafkaSubscriber concrete classes which implement them to represent kafka producers and kafka subscribers respectively. Publishers define a "to" relation to EvenStreams and Subscribers a "from" relation. This creates a clean mapping between microservices and the topics they communicate to. Kafka Streams are special in that they can both consume and produce to a topic, therefore the KafkaStream class implements both Publisher and Subscriber, which means it will inherit both the "to" and the "from" relation. This produces exactly the outcome I was looking for, but it comes at a slight cost. Firstly, the Publisher and Subscriber abstract classes cannot define the "name" attribute because

KafkaStreams would then inherit the same attribute twice, which means that this attribute needs to be defined for all subclasses of Publisher and Subscriber, which is fine but makes for a non-intuitive first look. Secondly, and perhaps more importantly, the “from” or “to” attributes inherited by KafkaStream are not mandatory, so we could in fact define KafkaStreams without a output topic, which is very good and desirable, but on the other side, defining KafkaStreams without a input topic makes little to no sense. Ideally, there would be a constraint on KafkaStreams having a “from” attribute.

The EventStream class contains:

- Publishers and subscribers mapping to a microservice publisher or subscriber, as described above.
- A relation to an Event class, which represents the type of record that is stored in this event stream. While I would have possibly preferred to represent this information as a contained value of EventStream, I assumed from the wording of the assessment that this design was required instead. With this approach, a OneToMany relation between EventStream and Event is created, where an EventStream can only have one type, but there can be multiple Streams of that type.

The Event class contains:

- As described above, a mapping to eventStreams of this type
- JavaVars for describing the key and value of this event.

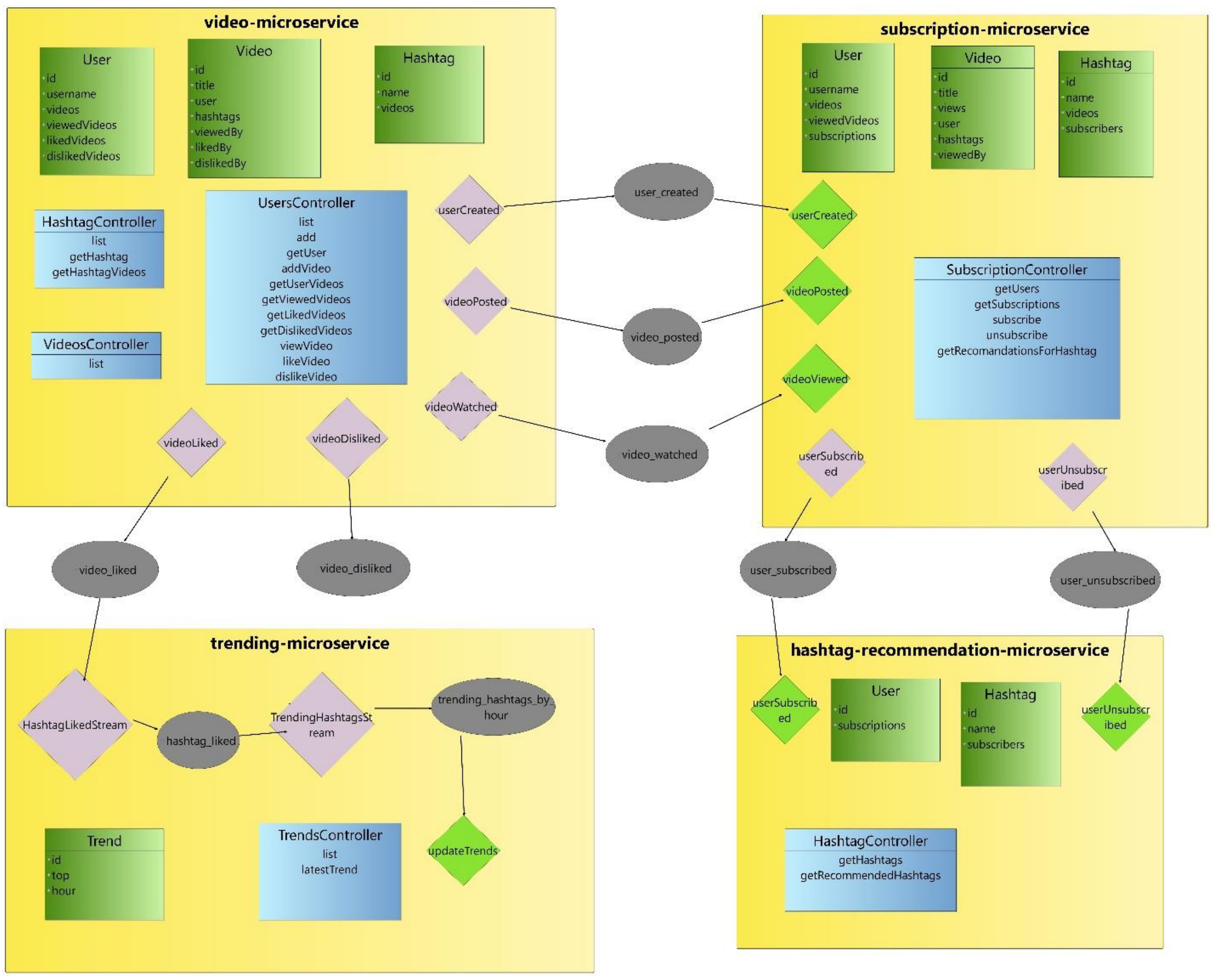
JavaVar is a representation of an object in Java. It contains a type and a name (e.g “String name” or “Long id”). This is a support class that makes it easier to represent such elements instead of creating two new attributes for a class. It is used to describe entity fields, resource query parameters, and event keys and values.

All concrete classes in this design have a “name” attribute for identification.

On top of the already discussed designed decisions, I will add the following alternatives designes I have discarded:

- Representing repositories. While repositories have a very important role in any microservice, it seemed a bit pointless to add it to this representation, where it wouldn’t have provided information that is not already available in the Entity class. Therefore, with the current simplified design, Entities can also be used to generate repositories.
- Relations between consumers and entities and controllers. It is very common in my design to have consumers update databases (e.g. SM consumers or the THM KafkaSubscriber). While this would have been very useful for human readers of the metamodel, it didn’t provide any benefit in code generation as far as I could see. I have decided not to add it in order to keep simplicity low, but ideally, I think this relation should exist.
- The JavaVar class makes the creation of models more straightforward, and I would argue that it improves readability of the metamodel, but it makes for a harder time accessing the information it contains (having to go through multiple classes to get this information isn’t always intuitive), and makes for harder representation in Sirius.
- I could have represented Topic Beans or DTOs as part of the metamodel to aid in code generation later on, but this info was often repetitive with the entity class and it created bloated models, so I discarded the idea.

2.2.2 Graphic Concrete Syntax



Syntax design justification:

- Firstly, I adhered to the concept of semiotic clarity, maintaining a one-to-one correspondence between symbols and their referent concept.
- In terms of graphic economy, I sticked to 3 types of shapes and 6 different colours, which was enough to represent all the elements in my model
- The SocialMediaPlatform root instance is not part of the model represented here because it didn't add any relevant information for this type of diagram. If I would have used a flow diagram, a table or a tree to represent my model, the SocialMediaPlatform class of the metamodel would have been useful.
- Not all classes of the metamodel are used in this diagram. Other diagrams could make use of other metamodel classes.
- In terms of perceptual discriminability:
  - Microservices are represented using big yellow boxes, which I rate at a high visual distance to any other object, because of its size, its unique colour, being an encompassing element, and even adding a gradient. They easily stand out as the main elements of the model, as they should be.
  - Entities live inside microservices, gives it semantic transparency, because we can easily infer that microservices have entities and not vice versa. The green colour with a strong gradient makes them easily differentiable inside the crowded microservice container.
  - Controllers are similar to Entities, but they are placed at a visual distance of one to them due to the change in color. This is fine, because at this abstraction level, controllers and entities do not differ much, but they are clearly differentiable inside a microservice. If we were to delve deeper into the inner workings of microservices, those two should place themselves at a higher visual distance.
  - Event streams are dark grey and oval shaped. Topics are nothing like any other component in the diagram so I placed them at the highest visual distance to anything else I could. They have a unique shape and a colour that stands out as a relational element. The colours of the arrows leading to and from even streams are matching the colour of the oval shape, which makes a lot of sense and makes

this element be very easily interpretable as a “connection” between other elements. I used a hierarchy to distinguish two of the event streams, “hashtag_liked” and “trending_hashtags_by_hour”. Those two are unique in that they are both created and consumed by the same microservice, and they could be considered “internal”. Having them be part of their parent microservice helps materialize this idea, and makes for a neater representation of THM’s workflow. This works great for this particular model, but it must be emphasised that all topics are equally public and the “hashtag_liked” topic, for example, could be used by future microservices, which is not very clear in my diagram.

    - Producers and subscribers are have a high visual distance to other objects, but a low visual distance to each other. Their shape distinguishes them from other elements in a microservice, and the connections to the topic offer semantic transparency. Their role in the communication between microservices is clear. One weakness I will point out is the colour choice of the subscribers which is a bit to similar to entities, which have no relation to them at all. Subscribers find themselves at equal visual distance to entities and producers, which is not ideal. However, this choice helps with graphic economy, since adding another colour would have pushed the syntax closer to the human span of absolute judgement.

- I’ve mostly covered the strengths of the syntax choice, so I will go over some additional weaknesses I’ve encountered:
    - Mainly, the lack of description of event types, which I could represent due to the metamodel decision of making events a class related to eventStreams, instead of belonging to eventStreams. Ideally (for this diagram), eventStreams should have been containers with Events in them. I couldn’t find a appropriate relational representation.
    - Like mentioned above in section 2.2.1, a lack of connection between consumers and controllers, which I have tried to create by placing those elements in close proximity.
    - Lack of semantic transparency for the arrow, like text or event type to accompany it

- In addition, I will also mention that I was striving to achieve a diagram that is not bloated with lots of information. I find the current level of graphic economy very good and I think it isn’t easy to justify the addition of a lot more elements.

2.2.3 Model Validation:

The following constraints are validated by Y1234.evl:

• There should be at least one microservice.

- It makes a lot of sense to validate for such a constraint. It would make little sense to have a application with no microservices in this metamodel. In addition, the metamodel does allow for the creation of a SocialMediaPlatform with no microservices, so this has to be checked.
- Implemented at the SocialMediaPlatform context, I apply a constraint that the number of microservices should be at least 1

• Every event should be used in least one event stream.

- With the current design of the metamodel, it is very easy to check for this important constraint. Dangling records should be avoided in any model representation
- At the Event context, I can simply get the related topics (eventStreams) and check that there is at least one of them.

• Every event stream needs to have at least one publisher and one subscriber.

- This is not a constraint but rather a critique. It is useful for my model where “video_disliked” doesn’t have a subscribers and so it raises this critique
- At the EventStream context, I have two critique ops, one check for publishers and the other for subscribers.

• Every microservice needs at least one “health” resource using the HTTP GET method and taking no parameters, for reporting if it is working correctly.

- This is important for any microservice but it is a bit trickier to check in my metamodel. The controller class stands between microservices and their resource. This is a useful aspect in code generation but it proves to be a hindrance for writing this important validation.
- Nonetheless, I have to find if there exists a controller such that one of its resources meet the “health” criteria defined above. EOL provides good tools for iterating and checking for null objects so I can do this in a relatively neat “one liner”.

2.2.4 Model-to-Text Transformation

I define an .egx transformation file called Y1234.egx that contains the following transformations:

- Entity2Java.egl:
    - This transforms Entity elements in the model into JPA entities used by micronaut. The target is a class that takes the name of the entity and all its fields and relations.
    - Then I iterate through the fields of the entity and create the java class fields with the column annotation. The Id is treated a bit different because it doesn't have the column annotation but the "Id" and "GeneratedValue" ones
    - In addition, fields are surrounded by protected regions because it is sometimes necessary to modify the annotations of a field in ways that the metamodel cannot reproduce. For example, adding the "unique" parameter to the column annotation of a hashtag's name, or removing the @GeneratedValue annotation from the entities of the Hashtag Recommendation Microservice.
    - Getters and setters than follow a similar approach, but those don't need protected regions.
    - There are two more protected regions in this file. One for additional functions at the end of the class, this could be custom operations for turning entities into dtos or overrides of hashCode or Equals or toString for better object manipulation in java. The other protected region is at the imports, because any new code in the other protected region would require new imports and those have to be protected when the file is recreated.
    - This transformation is very useful because entities in micronaut have a lot of repetitive code, have getters and setters which can be easily automated, and they adhere to a clear structure that rarely needs to be modified.
    - This was very helpful in generating the entities for the subscription microservice and the hashtag recommendation microservice, and while I was going back and forth on the design I was using the generator a lot to keep all related files in order. For example, it was easier to modify a relation in my .flexmi file and regenerate the code, then go to two java files and modify two different class fields which have to be correctly mapped in order for the application to work.

- Entity2Repository.egl
    - Like mentioned above, entities already include all the needed information to generate JPA repositories out of them, which is why the metamodel doesn't include a repository class
    - This generation is similar to that of Entity2Java.egl, but slightly shorter. I generate the repository implementing CrudRepository with the Entity type, and then I put a protected region inside of it, and over the imports.
    - This is because the metamodel cannot infer the needed additions to the repository, like custom find functions and @Join annotations, so the user will often have to manually polish these interfaces.

- Controller2Java.egl
    - This is a different approach to code generation. Instead of creating concrete classes with protected areas, I create an interface which I will then implement in my manually written code to override its methods.
    - This is the only case where I could take this approach because most other micronaut classes and interfaces have annotations that are not inherited. However, the controller's @Get @Put @Post etc annotation are inherited so I can define my interface with this annotations and then omit them from my class, which I thought would be very neat and particularly useful for newer micronaut developers that can make use of the boilerplate code.
    - In this transformation I create a interface with the Controller name and I iterate over its resources to create a method for each of them, with the required query parameters and body.
    - All methods here are of type HttpResponse<?>, which gives the user a lot of freedom to return whatever datatype he/she wants. This might not always be desirable, but at least it guarantees that all methods return a httpresponse. The alternative was to store the response type as part of the Controller class in my metamodel.
    -
- Kafkapublisher2Java.egl
    - This is generates a interface with the name of the microservice+Producer and with the @KafkaClient annotation that contains all methods for publishing records to topics, as represented in the model.
    - It consists of iterating through the publishers of a microservice and writing the specific method for it.

- KafkaSubscriber2Java.egl
    - Subscribers are a bit different because they are concrete classes not interfaces, they need a groupId which I defined as the name of the microservice, and they need custom implementation of their topic subscriptions

    - The transformation generates one Consumer class with all the subscriptions inside of it, which means that all kafka subscribers will be part of the same group, which might not always be desired but for the purpose of this assessment I thought it was ok.
    - Inside the consumer class there is a protected region for additional fields like repositories or clients
    - Then I iterate over the microservice subscribers and I create a empty void function for their specific topic. Inside the void functions there are protected regions where the developer can add manual implementation.

- KafkaStream2Java.egl
    - Kafka streams also contain a lot of repetitive code and predictable function types and annotations. Specifically, in this transformation I try to take advantage of the fact that I know what types of objects will be coming into the stream and what type it will be returning, therefore I can generate some of the code for deserializing the input topic into a kafka stream.
    - It all has to live inside protected regions however, because kafka streams are highly versatile and cannot be generated automatically more than this.

The code is **always** generated right inside the microservices folder, so developers should be weary of changes they make outside protected areas. However, interfaces for the Controllers are generated inside a new .gen package. This is because, like I said, controllers code is generate as an interface that is meant to be implemented by the developers. The transformer can handle the creation of the .gen package if it doesn’t exist. Code generation location is mostly computed in a pre {} block inside the Y1234.egx and it assumes that it is being executed from inside the “\modeling\m2t” folder.

In addition, it should be noted that the generated java files have dependencies to the .domain and .dto packages which should be manually implemented.

2.2.5 MDE-based Microservice

HRM makes use exclusively of the topics produced by SM to recreate a small subset of SM's database and provide a resource for computing recommended hashtags for a given hashtag. I used the DSML to add this microservice to my model, as seen in section 2.2.2. I created it such that it only contains two entities, Users and Hashtags, which are mapped by a single @ManyToMany relation representing the subscriptions of a user.

HRM has two consumers:

- One for "user_subscribed" which handles the creation of the user and the hashtag it subscribed to in the SM database. It is very important to note that entities in HRM don't have an autogenerated Id and so we need to set the Id from the incoming records. This helps better maintain the consistency between the data of the two services.
- One for "user_unsubscribed" which unsubscribes user from a given hashtag.

While the consumers keep the database updated, HRM controllers expose the path "/hashtags" to provide a resource for computing the recommended hashtags. This happens on demand and it is done by iterating through the subscribers of the given hashtag, and then iterating through the hashtags of each subscriber to find the most similar hashtags. This is not an ideal implementation as it doesn't scale well with added users and hashtags, but for the purpose of this assessment I thought it was good enough.

The DSML designed in sections 2.2.1 and 2.2.3 have been heavily used in designing both the subscription microservice and the hashtag recommendations microservice. The following evidence is available in the HRM:

- The .gen package with the generated HashtagControllerInterface is fully generated by the .egx transformer as described above. It contains the resources described in the model for the HRM microservice.
- The repository interfaces are also generated from the model, but I have added custom find* functions inside their protected regions because I needed @Join operations.
- The HashtagConsumer class in the events package is generated by the egx, but it has a lot of content inside the protected regions because the subscribers are very complex. However, the generator was very useful at modifying topic key and value type. During development I was often going back and forth on the value type of the "user_subscribed" topic, and it was very easy to modify my model and regenerate my code such that all references of that topic were updated.
- The Hashtag and User entities inside the domain package are also generate by the transformer. I modfied some of the annotations of the fields to make ids not be autogenerated, but other than that the code is mostly written by the .egx
- Rerunning the transformation and running the test will produce a positive outcome with all the tests passing, since the generator won't change any of the contents.

The testing process of HRM is as follows:

- There is a HashtagConsumerTest which does iteractive-based testing to check if the consumer and the database interact correctly. This does not check for the connection between the consumer and the real kafka cluster.
- There is a HashtagControllerTest which creates a complex scenario of multiple users subscribing to hashtags and then requesting recommended hashtags to subscribe to next. I think this scenario covers most of the use cases of the microservice but still there might be edge cases where it fails.

I scanned the container using docker scout and found the same vulnerability as the one seen in VM,SM and THM, related to the netty package which I couldn't fix.

I think DSML can save time for both new and experienced users. It helped me a lot with keeping things consistent between packages as the project was getting bigger and bigger. Especially when it came to modifying the key or value type of a kafka topic which was something I kept running into. Even for experienced developers it would be a pain to have to find every microservice that uses the given topic and slightly adapt it to the change. This is just an example but more cases like this happened while I was working on SM and HRM.

However, there have also been cases where the using DSML has been detrimental. Specifically for generating interface classes which can then be implemented by a manually written class. I could only take this approach for the controller classes, but the result was detrimental. Important parts of my code (the HTTP method annotation) now lived in the .gen package while the rest of the code lived in my .controllers package, so I often had to navigate between the two to get all the information I need, instead of having it in the same class in the first place. On the other hand, generating these interfaces without the important annotations doesn't seem beneficial to me at all, since it doesn't achieve more than type checking method types and parameters. However, this is heavily tailored to my experience on my minimal model, and I do believe most of these issues can be overcome by increasing the complexity of the model.

If the DSML contained more fine grain information about each component of the microservice, it could then generate files that don’t need any intervention by a human. Not to modify protected area and not through implementation of interfaces either.

I believe this can be well tailored for new users, where they can be given access to code that can be quickly modified into something deployable. It could be useful for creating tutorials and walkthroughs with complex generated projects, that only require small modifications to reach a state where the follower can experiment with new elements of a technology.

For experience users I think this could work best as a powerful type checking tool that ensures dependencies between services are met at a high level but don’t interfere with complex implementation. For example, defining the type of keys and values a topic requires and denning the modification of producers into a broken state.