ENG2 - REPORT

Y3877930

1.1 Architecture

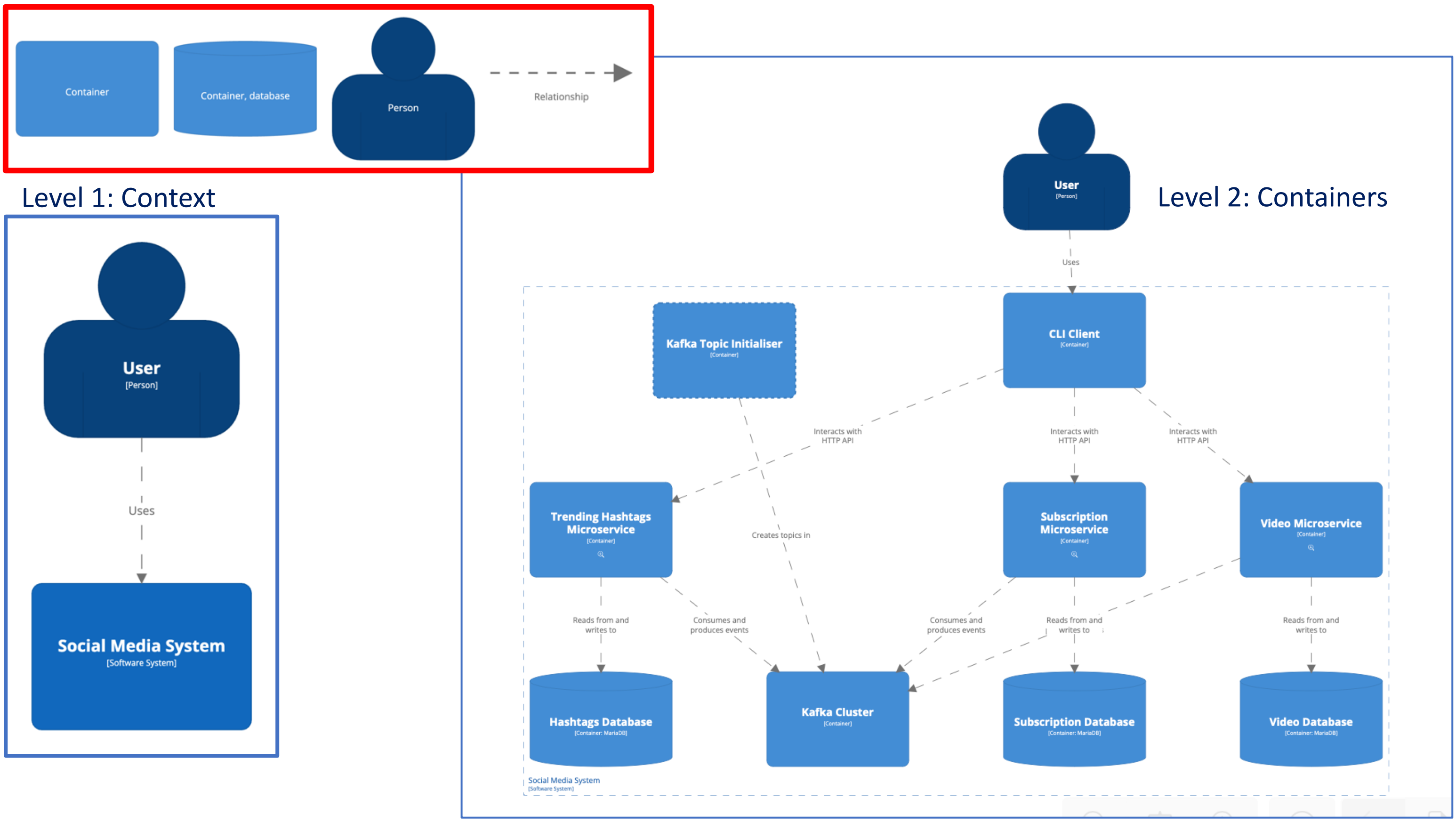
Define the overall architecture using recognised notations (e.g. UML component/deployment diagrams or C4 diagrams). Justify how the architecture can scale with increasing user demands, and be adapted to new requirements in the future (e.g. a recommendation system). [15 marks, max 2 pages]

The architecture of my system is that there is 3 Kafka nodes and 3 microservices, each with its own database. There is also an init-kafka node that creates the kafka topics. This allows me to disable automatic topic creation which increases security. Furthermore there are no exposed ports other than microservices. This helps to scale with increasing user demands, as more users mean that security is more important.
Having 3 kafka nodes is important because kafka uses replication to ensure the service can be maintained even if a node is removed or goes down. Each partition of each topic is replicated across the nodes, and having three nodes allows for a majority quorum to be maintained even if one nodes goes down. It is also is a balance between the fault tolerance of the system and performance given it is being run locally on a single machine. This helps the architecture be scalable because the multiple nodes balances the processing and storage requirements
The microservices having its own database and storing some information in its own database allows for the microservice to be loosely coupled, so that if one microservice, and/or its database, goes down, the other microservices may still be able to provide functionality that would reduce downtime for users.

The architecture used is a microservice architecture rather than a Monolithic architecture. As each microservice can be deployed independently and therefore scaled independently, particularly if there is increased user demands then microservices handling core functions such as authentication or main data throughput can be scaled both horizontally (adding more docker containers for example) or vertically (increasing the CPU power or memory available). Furthermore, each microservice can be worked on separately by different teams as each is responsible for a different function and is only loosely coupled, so this can increase how quickly additional features can be added to a system.

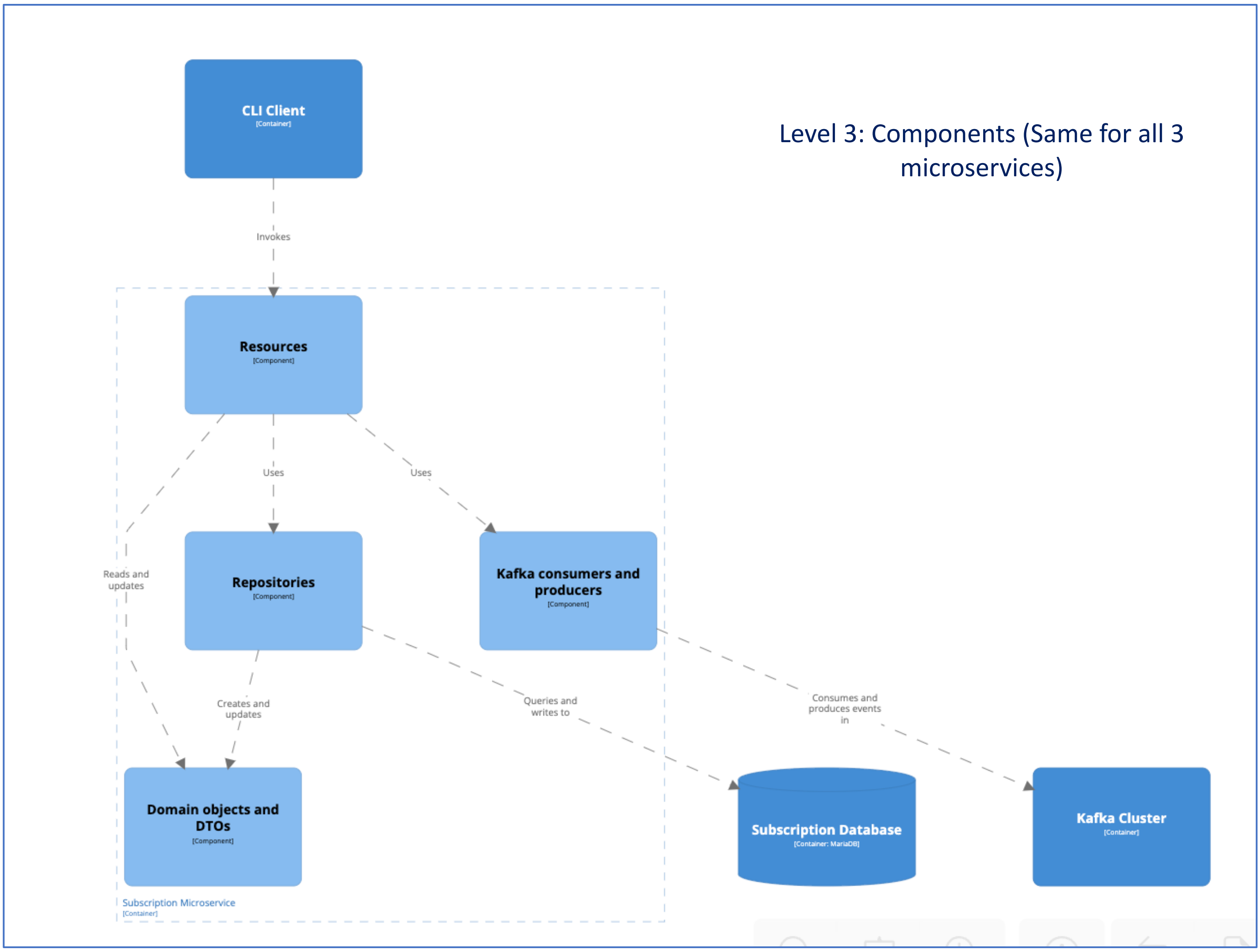
The database choice of MariaDB and docker compose means that it can be scaled horizontally in the same with as the microservices, adding additional nodes to the cluster would mean that the MariaDB could handly more I/O operations at once.

In terms of adding new features to the service, the microservice architecture means that to add new features, a new microservice can be added to the architecture. The new microservice can consume and produce events and do api calls without effecting existing microservices and features.



Level 1: Context

Level 2: Containers



Level 3: Components (Same for all 3 microservices)

1.2 Microservices

It is important to note that I did not focus on UX when designing this system. Pariticularly the CLI. In hindsight I should have made the usernames unique so that an end user can use their username to perform actions rather than their user ID.

Video microservice

This microservice has a database that stores all users, videos and hashtags as entities. It exposes endpoints to be able to create, retrieve and update these entities, such as marking a video as watched by a user and liking a video.

Video microservice exposes 18 endpoints, under the paths /user, /video and /hashtag. (Listed and described fully in microservices/README.md)

This microservice does not interact with any other microservice and does not consume any events. This means that it does not rely on anything else and will continue working if other microservices go down.

However, it does produce events to kafka topics:

- new-hashtag – upon a hashtag being created (Key: Hashtag ID, Value: Hashtag Object)
- new-user – upon a user being created (Key: User ID, Value: User Object)
- watch-video – upon a video being marked as watched by a user (Key: User ID, Value: Video Object)
- new-video – upon a video being created (Key: User Posted ID, Value: Video)
- like-video – upon a video being liked by a user (Key: User Liked ID, Value: Video)
- dislike-video – upon a video being disliked by a user. (Key: User Liked ID, Value: Video)

Architecture design decisions:

- Separate entities for hashtags means that they can be stored in a database separately and have relationships to other classes. This is due to the first normal form of a database being that each cell contains only one piece of information and should not contain relationships. We could have used a comma separated string to store hashtags in a video entity, but this would break the first normal form. Instead a new table is created that stores the relation between videos and hashtags.
- Each entity has a controller that exposes endpoints that create, or make changes to, the respective entity. UserController, HashtagController and VideoController. This is similar for event producers, UserProducer, HashtagProducer and VideoProducer. This is to improve readability of the system for developers.
- Each microservices has as little duplication as possible, particular for its persistence stores. For example the video entities stored in subscription microservices database does not store the postedBy value or the watchers (instead the are retrieved in the stream), which means two database tables are not needed.

Trending Hashtag Microservice

The Trending Hashtag Service is responsible for calculating the trending hashtags in the system. It does this by consuming from the `video-liked` topic in the Kafka cluster and keeping track of the trending hashtags accordingly in a 1-hour sliding time window. It also keeps track of the hashtags id's that have been created in a database. This is so that during the Kafka stream, it will check every hashtag that has been created for occurrences of likes in the stream on every update.

It exposes an endpoint /trendingHashtags which lists the id's of the top 10 trending hashtags.

The stream in this microservice, upon every event received, looks at the previous hour of events and counts how many occurrences of each hashtag there is, this is then stored in a materialized store.

If this microservice goes down, as long as the video microservice is still producing events, when it comes back on, it will be able to resume after the next event it receives as the kafka topic would persist.

This microservice does not directly communicate with any other, instead it only listens to kafka topic events, this means that the updates are slightly delayed.

Design decisions:

- The microservice consumes 'liked-videos' events in order to decide on trending hashtags, and also keeps a record of the hashtags that have been created by consuming the 'new-hashtag' event and stores in a database. On each new video-like, the stream breaks a video up into its respective hashtags, has a value of 1 assigned to the hashtag, and adds an event for each other hashtag that exists and gives it a value of 0. This is so that when the stream is windowed, it updates the count for all of the hashtags rather than just ones that are liked. An aggregate is used to count all likes within the rolling 1 hour window.
- Event streams were used rather than simple producers and consumers as it handles partition rebalancing and is less computationally intensive to do it in this way rather than storing in a database with timestamps to allow for windowing.
- A copy of the databases are stored in a separate database to VM so that the microservices stay loosely coupled. Again, using events rather than api requests also allows the microservices to stay loosely coupled.
-

Subscription Microservice

This microservice is responsible for allowing users to subscribe and unsubscribe from a hashtag. Subscribing to the hashtag starts the stream keeping track of videos that are posted. It also consumes videos from the watch-video topic.

If a user watches a video, it is removed from their list of videos in their respective subscriptions. This is done with two streams, the one that consumes from the new-video topic produces an event per subscription (user hashtag pare if the user is subscribed) for each hashtag in each video that is posted. Another stream keeps track of the videos that a user has watched for each subscription. The key at this stage is a subscriptionIdentifier, which contains a userId and a hashtagId. Both streams are grouped by subscriptionIdentifier and then aggregated to create a list of videos. The two streams are then merged, with the watched videos being removed from the new-videos list and then added to a materialized store. The store is accessed in the endpoint. The watch video stream also keeps track of a count of how many times each is watched. Lastly before the video list for the subscription is returned through the endpoint, it is sorted by number of watched, then the top 10 is taken.

Design decisions:

- The entities in this service are as minimal as possible. The postedBy value of a video is not stored by this services database.
- Due to the nature of the stream, the subscriptions do not update unless a video is watched by any user. This would not be a problem with a high usage of the social media system.

CLI Client Usage

The client can be run with either the dev mode of `./gradlew run args=”<command> <args>”` or after running the ./gradlew dockerbuild command in the client directory, can be run with `./videoCLI.sh <command> <args>.` This is a wrapper around a docker run command that runs the client docker container with the command given.

The CLI client can be used to interact with the microservices. It can be run as above and the following commands are available:

- `add-user <name>` - Adds a new user with the given name to the system.
- `get-user -id <id>` or `get-user -name <name>` - Gets the user with the given id from the system or by name.
- `list-users` - Lists all users in the system.
- `post-video <title> <id> <hashtags (space seperated)…>` Adds a new video with the given title, user posted by id and (optional) hashtags to the system.
- `get-video -id <id>` or `get-video -postedBy <userId>` or `get-video -hashtag <hashtagName> -` get videos by id, user posted or hashtag
- `list-videos` - Lists all videos in the system.
- `like-video <videoID> <userID>` - Marks the video with the given id as liked by a user.
- `dislike-video <videoID> <userID>` - Marks the video with the given id as disliked by a user.
- `watch-video <userID> <videoID>` - Marks the video with the given id as watched by a user.
- `get-trending-hashtags` - Gets the top 10 trending hashtags in the system.
- `subscribe <userID> -name <hashtagName>` **TBD** - Subscribes a user with the given id to the hashtag with the given name.
- `unsubscribe <userID> -name <hashtagName>` **TBD** - Unsubscribes a user with the given id from the hashtag with the given name.

1.3 Containerisation

Discuss how the solution can scale up to larger numbers of users, and be resilient to failures (e.g. of a container, or a node)
Each microservice has its own container, as well as each database and all 3 kafka nodes. Also there is a kafka-init container that adds the topics to the kafka nodes. These are all run with the orchestrator docker compose.
Lastly the client (CLI) is containerised and is started briefly upon each command.

As each part (resources and services) are containerised separately, this means that they can be scaled separately.
They can be scaled up to a larger number of users by:

- Multiple containers – multiple containers of each microservice can easily be added to ensure that there is constant availability. Also additional kafka nodes could be added. This means that if any containers go down, the others may be able to continue regardless.
- Database replication – MariaDB supports database replication in multiple ways including master-slave whereby changes made on the master server are asynchronously replicated to one or more slave servers. This means that if a database volume becomes corrupted or if a database node goes down, the system will carry on.
- Load Balancers – The addition of load balancers would be able to distribute requests evenly over multiple microservice containers to reduce risk of crashes due to too many requests.
- Scaling up – Giving more resources to the services such as RAM, disk space, CPU availability.

For the system to be resilient to failures the following could be/has been added:

- Automated restart containers (added) – The restart: unless-stopped stategy has been added to all containers to ensure that if they stop for any reason other than being stopped, they will restart.
- Monitoring – monitoring of the services and containers could be added with the addition of a Prometheus container which could send data to either an on-prem or cloud Grafana in order to easily configure alerts and create dashboards. This would allow a fast response to problems such as highload or a full disk.
- Healthchecks(added) – Implemented on each microservice and resource to aid with monitoring
- Loose coupling (added) – Where possible, microservices rely on kafka events rather than direct request based communication. This reduces reliance on each microservice and means that the microservices are more resilient to failures.

2.1.4 Quality Assurance

Client

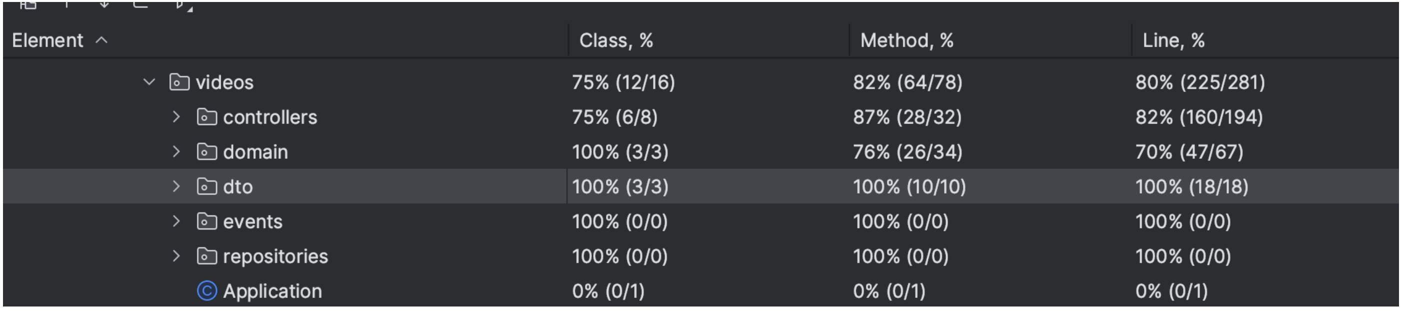
The client tests use docker-compose test containers to create isolated instances of the test environment, including all microservices and their resources, in order to run the tests. (see compose file in microservices/client/src/tests/resources. These containers only last for the running of a single class. The 36 tests takes around 8 minutes to run. Although this is longer than what is optimal, the fact that these containers are isolated means that the issue of not being able to clear databases between tests is not so much of a concern. Instead we must only be careful on the tests inside the classes eg. Use different hashtag names across the test classes methods. These were unstable but after adding `Awaitility.await().atMost(30, TimeUnit.SECONDS).until(ctx::isRunning);` the stability of the tests is increased as it ensures that the client is up and running fully before running a command. I also needed to add some sleep() statements to allow time for the events to propogate to the kafka streams. Although this helps, the tests are still occasionally unstable, Client tests cover all commands that the CLI supports, testing conditions where applicable, for example, a user doesn't exist. The tests for the Client are Integration and feature tests. As each CLI command is a feature of the system, the tests can be seen to be feature tests. As the tests use actual microservices and resources with test containers rather than using Mockito or similar to mock the complexities and services. This means that the CLI tests can give a good idea of how the services will behave in production.

The line coverage for the tests for this project is X%:

The tests for this in terms of feature and integration are relatively complete. However, if I had more time, I would want to mock the Client to be able to set the return value and then test only business logic without needing the resources. This would mean that the tests would also be faster as it would mean we would not need any sleep() or Awaitility() statements.

Video Microservice

Video microservices has 82% line coverage, of which the gaps in coverage are all non-business logic, eg. The application class. All tests pass and the 31 tests takes less than 30 seconds to run. Although these tests are faster and more granular than the client ones, again, if I had more time I would want to be able to mock the repositories to stop the tests using a database.



Element ^	Class, %	Method, %	Line, %
videos	75% (12/16)	82% (64/78)	80% (225/281)
controllers	75% (6/8)	87% (28/32)	82% (160/194)
domain	100% (3/3)	76% (26/34)	70% (47/67)
dto	100% (3/3)	100% (10/10)	100% (18/18)
events	100% (0/0)	100% (0/0)	100% (0/0)
repositories	100% (0/0)	100% (0/0)	100% (0/0)
Application	0% (0/1)	0% (0/1)	0% (0/1)

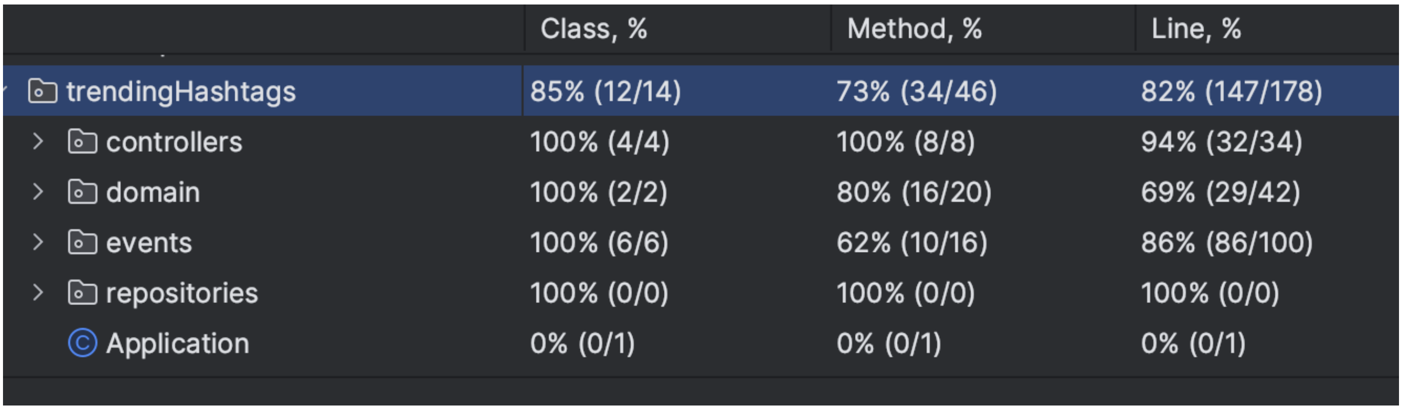
Tests for the video microservice are integration tests and unit tests. There is a test class for each controller and one for the kafka producer class. The tests interact with an actual database as it would in a production system, however some things are mocked like a unit test. In a controller, a mockbean is used to mock the kafka producers to enable the capturing of events in the tests without having to query kafka. The tests also focus on smaller bits of logic, like how specific endpoints react to invalid data as well as valid data in isolation, hence these controller tests are also unit tests.

Trending Hashtag Microservice

Trending hashtag Microservice has two main test classes with 9 tests that take ~2.5 minutes to run, all of which pass. TrendingControllerTest and TrendingStreamsDoubleTest. The controller tests are integration tests that tests that given a particular scenario, eg. There being 11 hashtags that have been created and only 10 are in a video that have been liked, the one that wasn't liked doesn't show up in the trending list and the rest do. If I had more time I would want to mock the trending stream results (materialized store) and test the logic and code inside of the endpoint body, rather than needing to use specific test cases that do not effect eachother due to the test containers being the same across the methods and therefore the kafka topics retaining previous events.

Note: the trending controller uses Junits OrderAnnotation to ensure that the ordering is consistent to avoid occasional failures when the tests are run in particular orders as a result of events being in the topics.

The coverage of this project is:



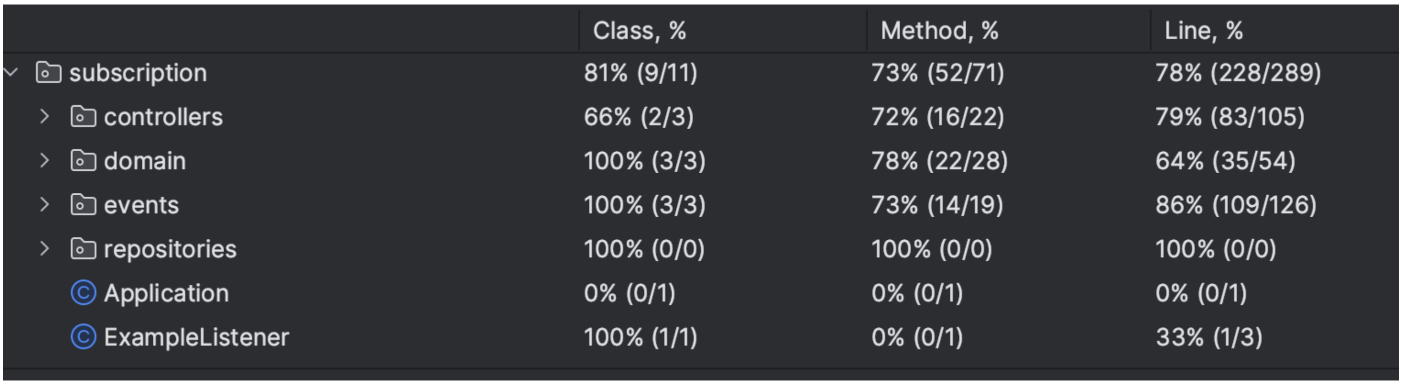
	Class, %	Method, %	Line, %
trendingHashtags	85% (12/14)	73% (34/46)	82% (147/178)
controllers	100% (4/4)	100% (8/8)	94% (32/34)
domain	100% (2/2)	80% (16/20)	69% (29/42)
events	100% (6/6)	62% (10/16)	86% (86/100)
repositories	100% (0/0)	100% (0/0)	100% (0/0)
Application	0% (0/1)	0% (0/1)	0% (0/1)

Subscription Microservice

The tests for subscription microservice, similar to the trending one as it also uses Kstreams, has test classes SubscriptionControllerTest and SubscriptionStreamsDoubleTest. The Controller has an additional function that is not an endpoint, sortedNextVideos that deals with the sorting of the videos in a users subscription by watches and then returns the top 10. It takes a list of Longs and a ReadOnlyKeyValueStore which is from a Materialized store. Pulling this functions logic out of the endpoint bodies allowed me to test it. To do so I create a ReadOnlyKeyValueStore and define the returns to allow the sorting of the values given. This allows the unit testing of this important part of the logic of the system. The endpoints are tested with integration tests similar to trending hashtag microservice.

These tests require some significant length sleeps, 20-30 seconds to ensure that they are stable. This is because the microservice gets information from 2 topics and then merges the streams together. This means that the information can sometimes be delayed.

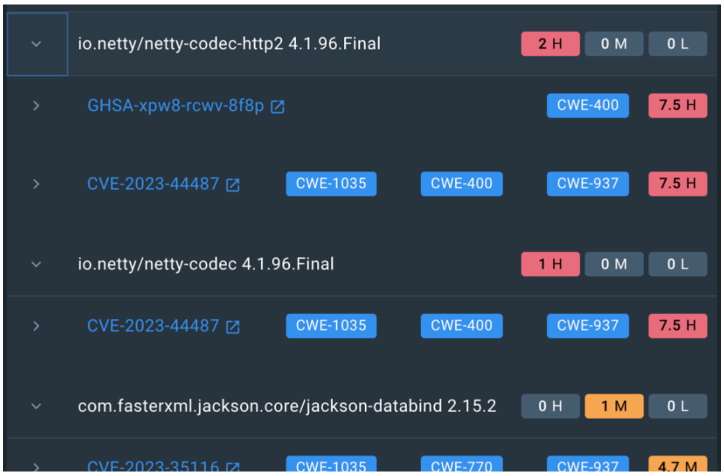
For this there are



	Class, %	Method, %	Line, %
subscription	81% (9/11)	73% (52/71)	78% (228/289)
controllers	66% (2/3)	72% (16/22)	79% (83/105)
domain	100% (3/3)	78% (22/28)	64% (35/54)
events	100% (3/3)	73% (14/19)	86% (109/126)
repositories	100% (0/0)	100% (0/0)	100% (0/0)
Application	0% (0/1)	0% (0/1)	0% (0/1)
ExampleListener	100% (1/1)	0% (0/1)	33% (1/3)

Docker Image inspecting

When scanning my microservice images with docker scout, the following vulnerabilities where found:



io.netty/netty-codec-http2 4.1.96.Final 2 H 0 M 0 L

GHSA-xpw8-rcwv-8f8p CWE-400 7.5 H

CVE-2023-44487 CWE-1035 CWE-400 CWE-937 7.5 H

io.netty/netty-codec 4.1.96.Final 1 H 0 M 0 L

CVE-2023-44487 CWE-1035 CWE-400 CWE-937 7.5 H

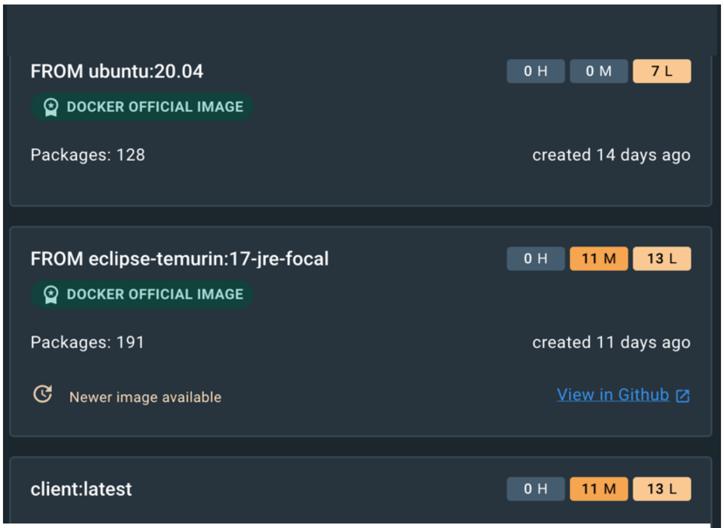
com.fasterxml.jackson.core/jackson-databind 2.15.2 0 H 1 M 0 L

CVE-2023-35116 CWE-1035 CWE-770 CWE-937 4.7 M

Figure 1 - Microservice docker scout scan before fix

The first 2 are dependencies from the micronaut-http-client dependency defined in my build.gradle and are were found in all 3 microservices. (This was traced using the dependency task in gradle). In order to resolve this, I found the earliest point in which these security advisories were resolved, and overrode the version to io.netty:netty-codec-http2:4.1.100.Final. The 3rd is a dependency of kafka-streams, I overrode the dependency to "com.fasterxml.jackson.core:jackson-databind:2.16.0" which also resolved it.

Ideally I would have traced back to the version of kafka streams or micronaut that caused the dependency version however I did not have enough time to feasibly do this and ensure there were no breaking changes in my microservices.



FROM ubuntu:20.04 0 H 0 M 7 L

DOCKER OFFICIAL IMAGE

Packages: 128 created 14 days ago

FROM eclipse-temurin:17-jre-focal 0 H 11 M 13 L

DOCKER OFFICIAL IMAGE

Packages: 191 created 11 days ago

Newer image available View in Github

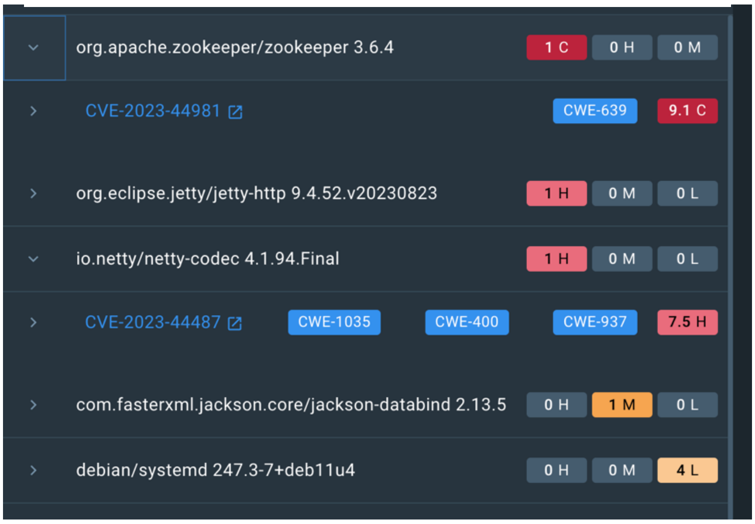
client:latest 0 H 11 M 13 L

Figure 2 - Client (CLI) image docker scout scan before fix

My client container uses ubuntu:20:04 and eclipse temurin as the base images which has many vulnerabilities:

In order to resolve this I have changed the base image to "amazoncorretto:17-alpine3.17" which **resolved all vulnerabilities**.

There were 51 vulnerabilities in MariaDB containers, the highest risk ones were related to parsing javascript which is mostly unrelated to my system. This many vulnerabilities cannot feasibly be resolved in the time given as the MariaDB version is the most up to date at the time.



org.apache.zookeeper/zookeeper 3.6.4 1 C 0 H 0 M

CVE-2023-44981 CWE-639 9.1 C

org.eclipse.jetty/jetty-http 9.4.52.v20230823 1 H 0 M 0 L

io.netty/netty-codec 4.1.94.Final 1 H 0 M 0 L

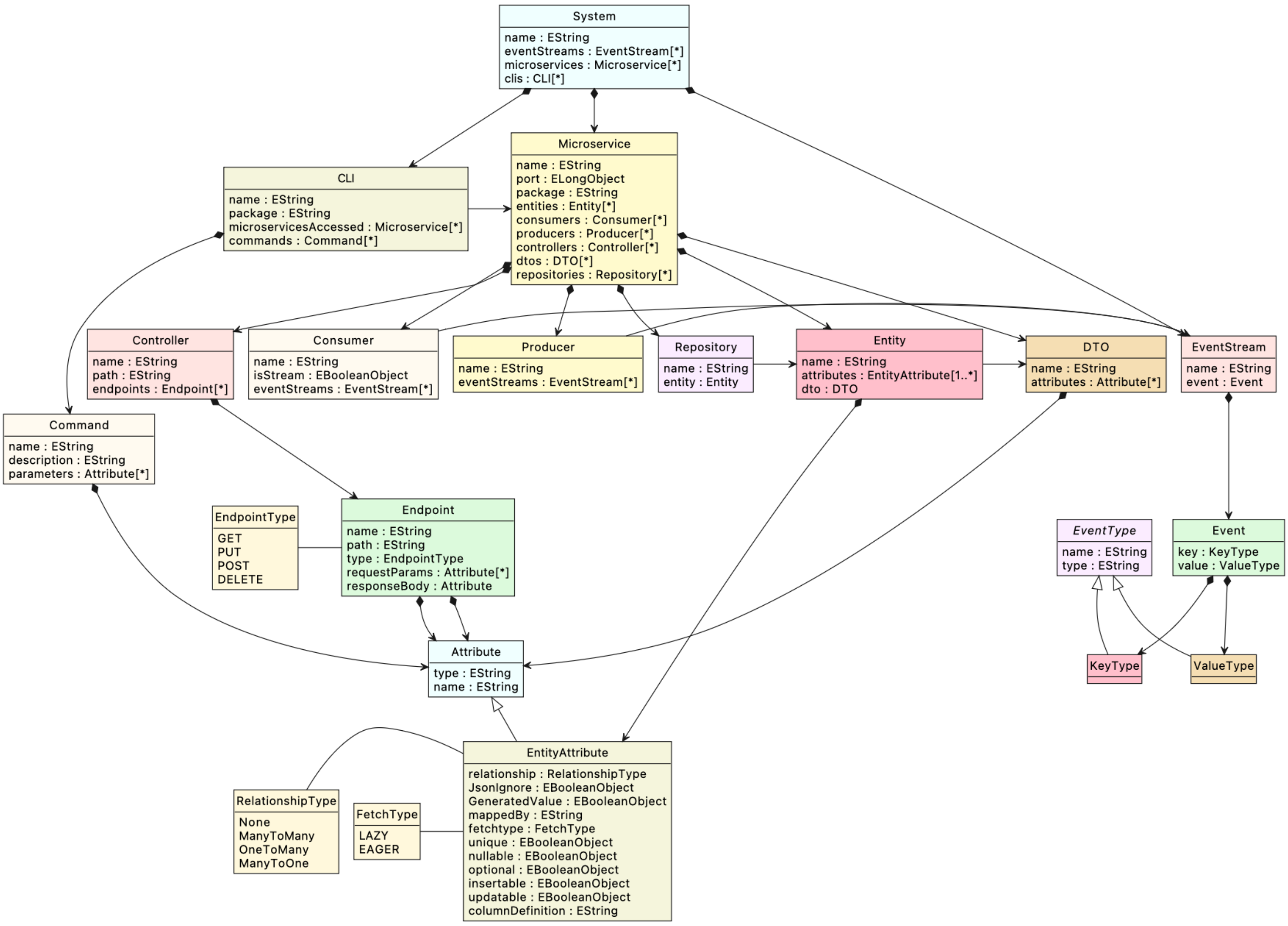
CVE-2023-44487 CWE-1035 CWE-400 CWE-937 7.5 H

com.fasterxml.jackson.core/jackson-databind 2.13.5 0 H 1 M 0 L

debian/systemd 247.3-7+deb11u4 0 H 0 M 4 L

There were 36 vulnerabilities in the kafka containers, in order to fix the most critical, I have upgraded my kafka image version to 3.6.1. This fixes the critical vulnerability although that relates to zookeeper, which my kafka cluster does not use. It also resolves a netty-code vulnerability as before which is a dependency in the image. This is good because these vulnerabilities were critical and high respectively.

2.1.1 Metamodel



From https://eclipse.dev/epsilon/playground/
At the top level of my metamodel, there is a "System" class encapsulating all elements within models using this metamodel. The "System" includes "EventStreams," representing Kafka topics for event production and consumption. Each "EventStream" has a name and an event type, further categorized into "KeyType" and "ValueType." This design choice allows for the creation of EventStreams that may not be immediately utilized by microservices. This feature aids in visually assessing whether topics are actively used or if adjustments are needed. It also helps improve extendability, if EventStreams were needed to be used by additional services inside the system that are not yet modelled. This was a decision made rather than having EventStreams in microservices but this means that when microservices are removed, any eventStream that is a val in it is also removed. Furthermore, in an actual system, EventStreams or Kafka Topics are not actually contained within microservices but instead are in the event streaming platform or Broker.

The "System" class encompasses "CLI" instances, representing command-line interfaces interacting with microservices. Multiple CLIs can interact with either a subset or all microservices, indicated by a reference to microservices in the CLI class. Each CLI has a name and includes commands represented by the "Command" class, featuring a name, description, and parameters represented by the "Attributes" class.

Furthermore, the "System" class includes "Microservices," each having a name, port, and package representation. Microservices consist of entities with names, attributes, and associated DTOs. Additionally, microservices involve consumers and producers, where consumers have a Boolean flag indicating whether they are streams. Both consumers and producers reference "EventStreams" for topic interaction. This design decision improves the visual representation and understanding of how microservices interact with event streams.

Microservices also incorporate controllers, featuring a name, a path (base URL path for controller endpoints), and API endpoints represented by instances of the "Endpoint" class. Endpoints include a name, URL path, and an enumerated endpoint type (GET, PUT, POST, DELETE). This simplifies the representation of RESTful APIs and enhances the DSL's ability to describe various API resource types. Endpoints may have request parameters and a response body, adhering to DSL specifications for API resource descriptions.

Microservices can possess DTOs, each with a name and attributes. The "Entity" class represents the domain within microservices, featuring a name and one or more "EntityAttributes" with type and name attributes. Entities also reference DTOs, graphically linking them. This association also aids in maintaining a concise and organized structure, fostering better readability and understanding of the relationships between entities and their corresponding DTOs."EntityAttributes" extend the standard attribute class and include additional attributes like JsonIgnore, GeneratedValue, uniqueness, nullability, optionality, insertability, upgradability, mappedBy, columnDefinition, fetch type (LAZY or EAGER), and a relationship type (ManyToMany, OneToMany, ManyToOne). Including a comprehensive set of attributes (e.g., JsonIgnore, GeneratedValue, uniqueness) in the "EntityAttribute" class allows for a detailed specification of entity attributes and allows for the accurate generation of entities code.

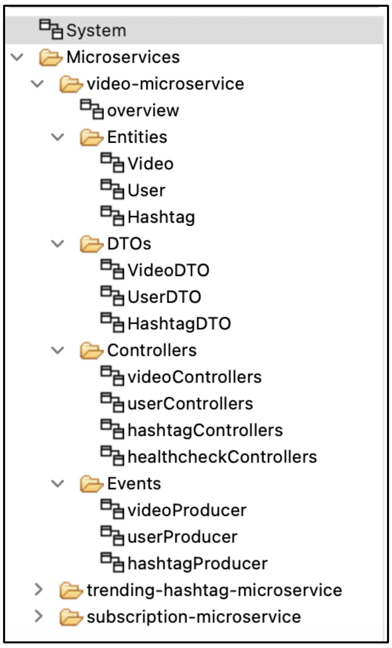
Lastly, each microservice includes a "Repository" with a name associated with the entity it manages. The separation of Repositories rather than assuming a repository for every entity, allows persistence of entities to be optional. Rather than having a Boolean for if an entity is persisted or not, a repository class promotes a clean and modular architecture where data access operations are encapsulated in repository classes.

I have assumed that each eventStream will only have one type of key and one type of value entering it. Also that a producer/consumer can interact with multiple eventStreams, eg. In a java and kafka representation, a class is a producer that has multiple methods that produce to different topics and can be called separately.

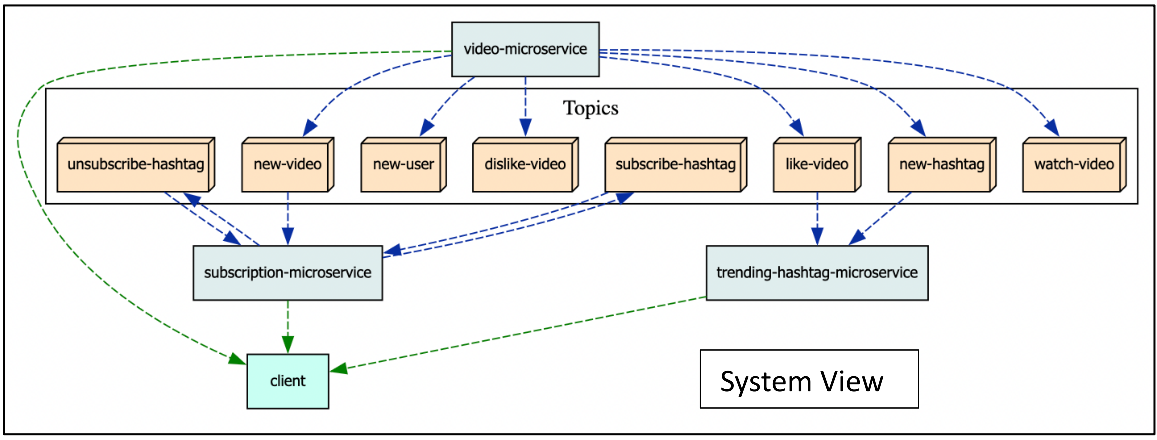
One of the decisions I made was whether to have DTO's and entities be shared across my microservices. Due to the way in which my project is set up, ie. Without it being a multi gradl e project and instead each microservice being a separate gradle project, DTO's and Entitys, although they can be the same across multiple projects, are separate in my model. This is to ensure it is an accurate representation of the architecture and also ensure that they can be generated as code separately into each project where needed. It also allow different services to drop some attributes of classes, allowing services to only hold a subset of entities and repositories. For example, the trending hashtag microservice does not need the relationship of "watcher" between video and user.

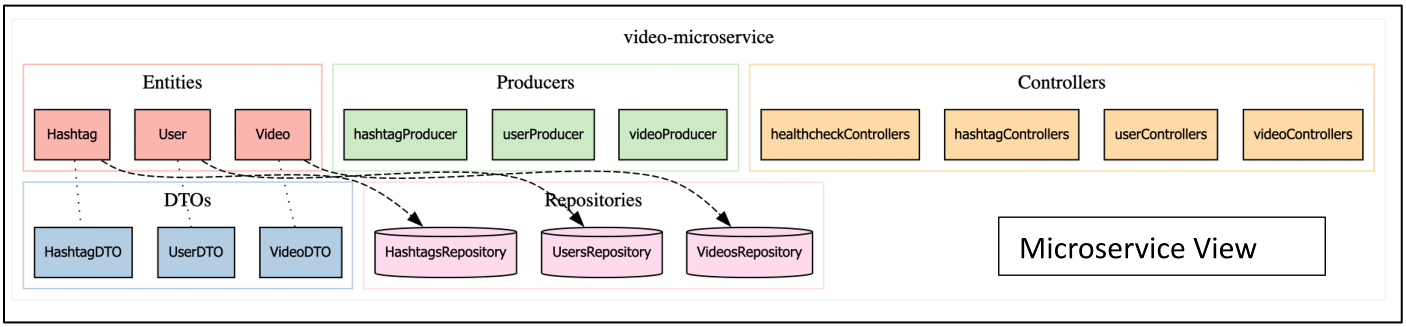
2.2.2 Graphical Concrete Syntax

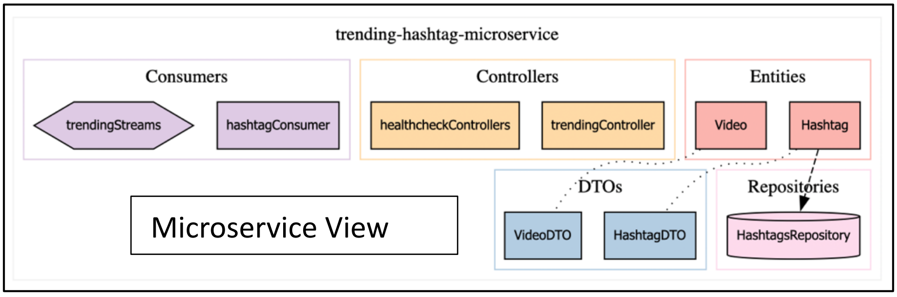
As my model is in picto and I have decided to break it up into smaller models as the picto language allows, I have provided a subset of my diagrams which shows all types of graphical syntax.

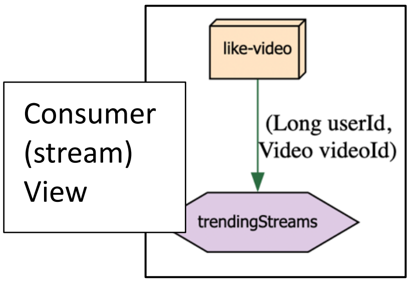


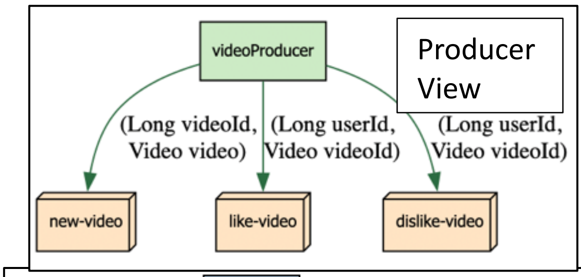
- System
- Microservices
  - video-microservice
    - overview
    - Entities
      - Video
      - User
      - Hashtag
    - DTOs
      - VideoDTO
      - UserDTO
      - HashtagDTO
    - Controllers
      - videoControllers
      - userControllers
      - hashtagControllers
      - healthcheckControllers
    - Events
      - videoProducer
      - userProducer
      - hashtagProducer
  - trending-hashtag-microservice
  - subscription-microservice

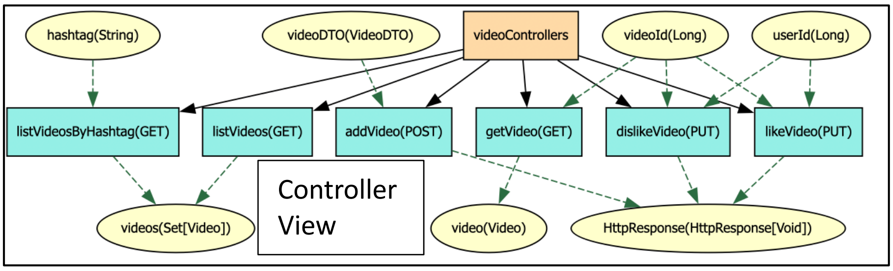


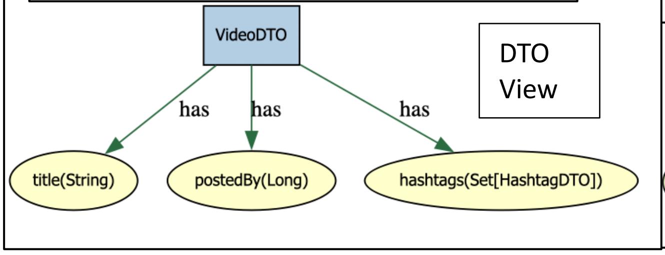


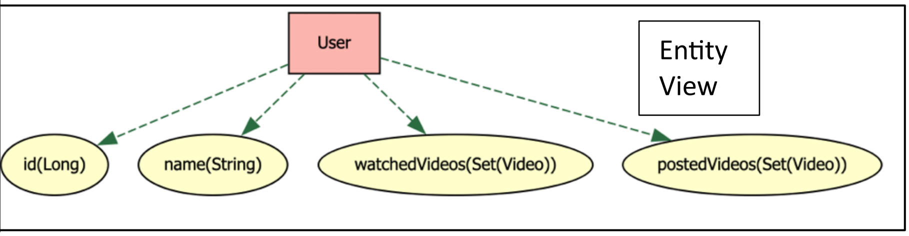





videoProducer
Producer View
(Long videoId, Video video)
(Long userId, Video videoId)
(Long userId, Video videoId)
new-video
like-video
dislike-video







The graphical implementation can generate diagrams of the system as a whole, and singular microservice, entity, dto, consumer and producer. This is implemented using picto.

- In the **highest-level system view**, microservices are represented as boxes, and event streams are depicted as 3D orange boxes connected by blue dashed lines, indicating the flow of information. The use of 3D boxes for event streams enhances semantic transparency, conveying that they hold data. The consistency of dashed lines throughout the graphical syntax maintains a clear representation of input/output flows. Arrows indicate the direction of information flow, aligning with users' natural cognitive processes. CLI's are showed as blue boxes, linked to the respective microservices that it interacts with by another dashed line.
- The next level of granularity is the **microservice view**. This view shows all components of the system, including Controllers, Consumers, Producers, Entities, DTO's and Repositories.
    - These components are in a container which represents the microservice as a whole and each component type has its own subgraph with an edge that represents the grouping by type. One of the important parts of this view of the system is that it shows the links between the Entities and its representation as a DTO. This is shown by a dotted black edge without an arrowhead, as there is no direction to the relationship, only that there is a link between them. Furthermore, this view also shows a dashed arrow between the entity and its respective repository. This shows that the entity is stored within the repository for the entities persistence. The dashed line, as previously, shows the flow of data. The database, similar to the KafkaStream, is a 3d shape, in the form of a cylinder, which represents again, that information is stored within it. Although it is technically stored within the associated database, the repository provides the CRUD interface for it and to non technical users of the graphical syntax, this is a logical representation. The cylinder was also chosen as it is the way in which databases are typically represented in C4 diagrams and other system diagrams.
    - Component colors and shapes denote distinct types, simplifying identification however a disadvantage of this is that it relies on the user being able to distinguish between colours. A hexagonal consumer shows that it is a stream rather than a simple consumer.
- **Entities and DTOs** are shown in smaller diagrams, with yellow oval attributes and green dashed lines between them. The yellow oval attributes are consistent across the diagrams as attributes. Attribute nodes show the name of the attribute and its type. However, the entity attributes do not show all of the additional features allowed in the 'EntityAttribute' class because there is a lot of information that can be added which is not necessary to understand the system. Similarly, the relationships are not shown in the microservice view either, this is to stop there being too much clutter and to reduce complexity, particularly when there are multiple types of relationship between two entities.
    - The decision to omit relationship details in the microservice view may limit the comprehensiveness of the representation for users who require a more detailed understanding of entity relationships. An extension to my graphical syntax might be to add a toggleable layer showing the relationships between entities and the additional details of the Entity Attributes.
- Attributes are used in the **controller view** to show the request parameters and response body from each endpoint. They use the same yellow oval and dashed arrow. The dashed arrow shows whether the attribute is a request parameter (arrow goes towards endpoint) or response body (away from endpoint).
- **Lastly, consumers and producers** are shown with another diagram view each, with the consumer/producer node being the same colour/shape as it is in the microservice view to show if they consume or produce and show if they are a stream or not. Furthermore they also show the various EventStreams they interact with with them being shown in the same 3d box form as in the system view.

Picto was chosen for its simple usage as my development of my model was iterative, I wanted the process of adjusting my graphical model to be simple and clearly readable in the diff of version controle for if I wanted to revert back. Furthermore the lightweight nature of it and the fact that I was already relatively familiar with graphviz, made it more appealing from a developers point of view. Also the fact that it is useable in other IDE's means that it is more accessible to other stakeholders. However, an advantage of using Sirius would be that it provides many additional features customization and extensibility, allowing the creation of graphical editors for stakeholders to interact with.

2.2.3 Model Validation

Use the Epsilon Validation Language to implement any validation constraints specified in Section 1.2, which cannot be expressed in the metamodel itself. Briefly explain the rationale and implementation of each constraint. [5 marks] (max 1 page)

The DSML must support automated validation. It must check these properties:

• There should be at least one microservice.

- This is checked in the EVL. Although it could have been expressed in the metamodel with the usage of [+] rather than [*], the use of the phrase 'should' suggests that only a warning should be given rather than an error. Therefore the EVL rule AtleastOneMicroservice checks whether the size of 'microservices' in the system is atleast one. If there isn't, it throws a warning rather than an error.

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

- This is checked automatically with the design of my metamodel. An event cannot exist unless it is contained within an EventStream as a value rather than a reference. Although this could be checked in an EVL, it would always be true and therefore a waste to implement or run unless the design of the metamodel changes.

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

- Similar to the first check, this is also in EVL form and Is also a critique and therefore gives a warning. (assumption based on clarification on the VLE) This check is implemented by two rules, hasProducer and hasConsumer. These rules, for every eventStream, checks there exists a producer/consumer that references this eventStream. It does this by looping through, checks if any of them reference it and then checks whether there is any of the producers/consumers that are true using exists().

• 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 check is implemented in EVL by a rule HasHealthcheck, which for every microservices, it checks that there exists a controller such that there exists a healthcheck endpoint in a controller. To do this, I implemented a function isHealthcheck that checks if an endpoint fulfils all of the following conditions:
    - EndpointType is GET
    - The name of the endpoint starts with "health" (converted to lowercase first so as to cover health, healthcheck, Healthcheck etc.)
    - There are no request Paramaters
    - The responseBody type starts with HttpResponse

  This is an error due to the word "needs".

2.2.4 Model-to-Text Transformation

My generated Java code is generated into src-gen folders within each microservice directory. This allows each microservice to have different generated code rather than sharing a src-gen folder. This decision was made rather than generating into src and using protected regions for manually written code. A dedicated src-gen folder allows us to maintain a clear separation between the automatically generated code and code that was manually written. This is important because it means generated code is less likely to be modified directly. It also means that src-gen can be marked as a generated source in an IDE which means it would be read-only, preventing changes entirely. Although protected regions would allow for integration between generated and manual code, it can make it confusing to a developer. The use of src-gen also allows easy ignoring of the folder in version control systems if needed.

The classes are sorted into separate packages to aid in reducing the complexity of the code for developers. DTO's are in .dto, entities in .domain, controllers in .controller and repositories in .repository.

- **DTOs** - including the getter and setter for each attribute.
- **Entities** - To enable all of the annotations for each attribute in the entity to be added, the models EntityAttribute class has needed to become very detailed in order to include all of the information needed for the code generation.
- **Controllers** are generated as interfaces that can be implemented by a developer.
    - It includes the annotation in each method to show the http type (GET, PUT etc.) and the uri path ("/", "/video").
    - The interfaces act as a blueprint for a class, allowing developers to focus on the implementation of business logic rather than spending time on the initial setup. This accelerates development by providing a starting point for coding that aligns with the overall architecture.
    - Standardized interfaces make collaboration among team members more straightforward. If I had more time, I might also have added some commonly used methods or properties, eliminating the need for developers to rewrite them for each class.
    - This was originally generated as abstract classes, with methods that return null so that the controller can extend the abstract. This is more of a scaffold as it allows the running of the microservice without anything being added. This template is still left in, as I could not decide which was preferable.
- **Repositories** have been created as standard interfaces without any additional defined methods. To add additional methods, it would need to be extended to create new interfaces with additional functionality.
    - I considered adding the overrides for common methods (eg. findById) as standard with the additional annotations that may be needed, like joins, calculated from the entities and their attributes types depicting joins between entities.
- **`docker-compose-gen.yml` and `docker-compose-secrets.yml`**. This has the yaml configuration for all microservices in the system. If a microservice has a repository configured then the generation assumes it needs a database. It also generates a suggest docker compose secrets file with a generic username and password set and the name of the DB is the name of the microservice it is meant to be used by. The secret is left with 'todo' to show that it needs to be changed. The use case would be the file to be copied and the password changed before starting the services.

If I was to have more time, generating the client interface for microservices into test folders and the cli would be beneficial for cutting down on the amount of work a developer needs to do in order to integrate the microservices into a CLI and also to test. This would be useful because when more endpoints are added etc, if they are added to the model, they will be added to the client automatically upon the next model to text transformation.

The model project is set up as gradle project that runs the model to text generation. There is also a javaGenBuild.xml that is an ant build that also runs the m2t generation along with a launch config Y3877930-m2t-egx.launch.