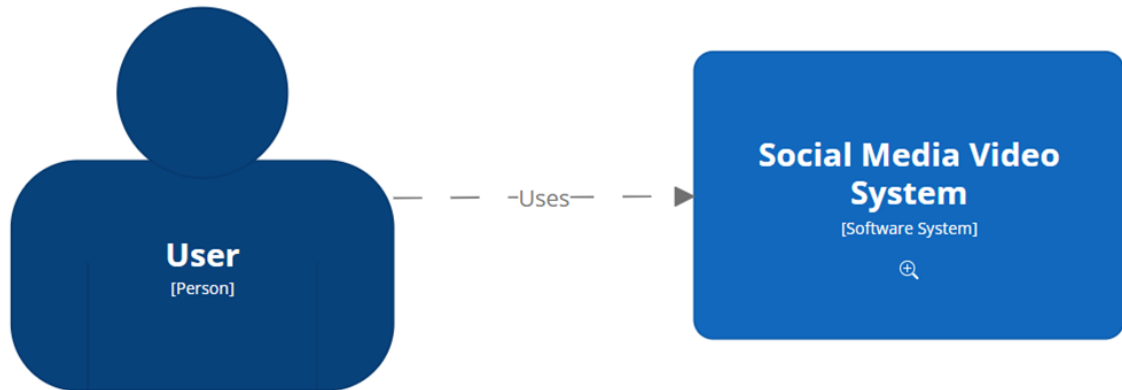


Engineering 2 Assessment

2.1.1 Architecture

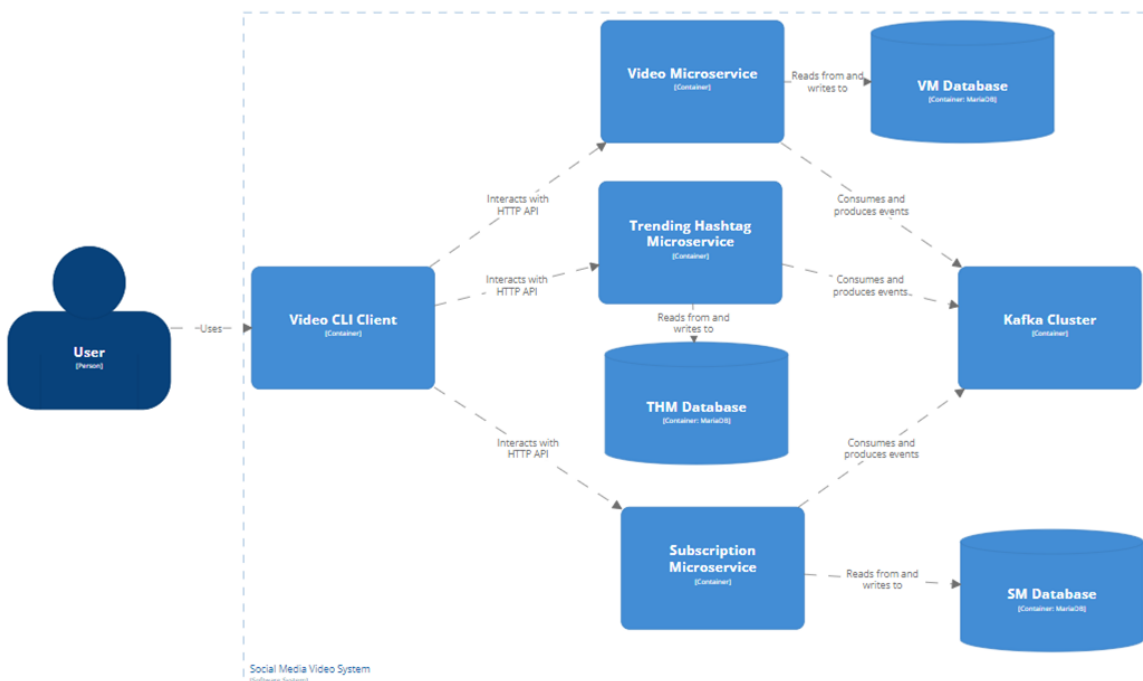
Below are the C4 diagrams and different abstraction levels:

Context:



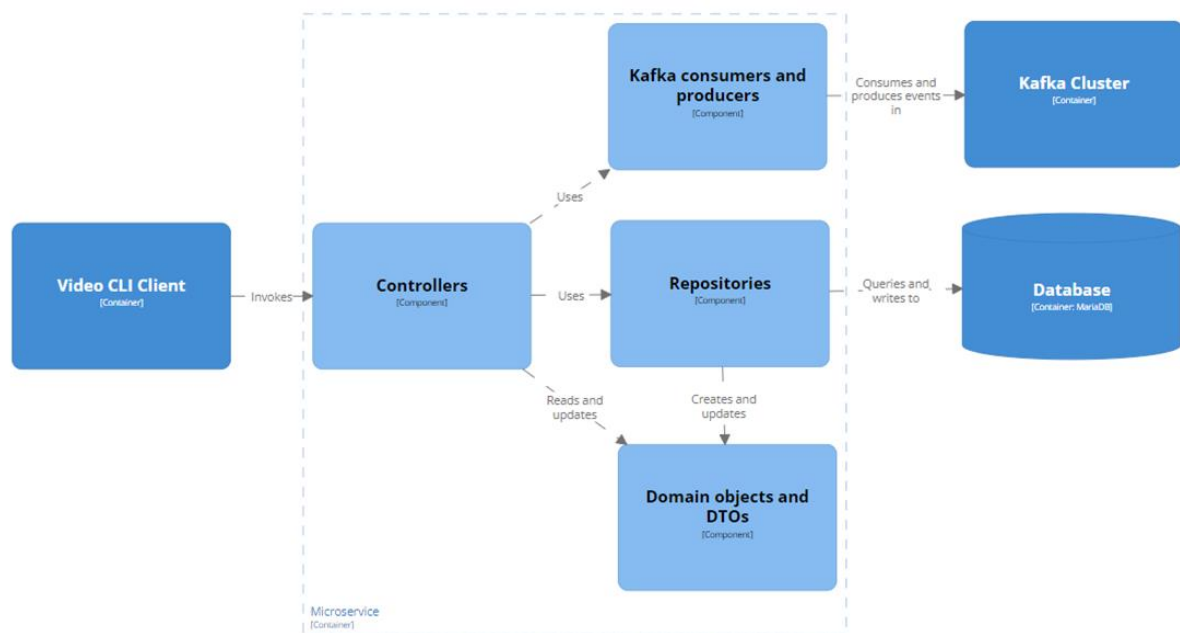
This shows the user interacting with the system

Container:



This shows the user interacting with the client which uses HTTP API's to interact with a microservice which all have their own database and interact over a shared kafka cluster

Component:



This shows the inside of a microservice, you can assume it looks the same for all of the microservices. The cli invokes methods on the controller using the HTTP API **[TO UPDATE]**

One way that this design scales with increasing user demands is that each microservice can be deployed separately on individual nodes and so requests from the user (through the CLI Client) are spread to the different nodes resulting in less load per node. However, this may be a marginal gain as the microservices are unlikely to have an equal amount of requests. This can be counteracted by adding spinning up more nodes with a copy of the more frequently used microservices. The bottleneck would then be the communication between the microservices of the same type (something that hasn't been implemented) as they would need to make sure the databases are kept up to date with any changes.

A good feature of this design is that new requirements can be implemented with relative ease as a new microservice can be created to facilitate this new requirement without needing to change anything about the other microservices. This makes it much less prone to faults and also easier to implement without too much knowledge of the current system.

2.1.2 Microservices

Video Microservice

The Video Microservice is the core of the social media platform. This is where users can register themselves. After this they can add a video and then hashtags to those videos. There is also functionality to watch and react to other people's videos.

User – This is the object used to represent the people that use the microservice. This contains the name of the user and an identification number. It also has videos the user authored and the reactions the user has had to videos.

Video - This represents the videos that user post and watch and contains a title and an id. This also has the user that authored it, the reactions users have had to it and any hashtags associated with it.

Hashtags – These are the text hashtags that an author can associate with the video. They consist of a name an id and the videos that it associates with.

Reactions – this represents the most recent interaction a user has with a video. It contains an id, the reaction the user had and also the user and video its associated with. This can represent a user watching, liking or disliking a video as it's implied that a user has watched a video when liking or disliking a video.

Having the reactions class making things interesting especially with the intricacies of Micronaut databases. However, I think it makes sense to have them as the alternatives seem to be worse. E.g. having liked and watched as relation to users I'm not sure if even possible to have more than 1 relation between 2 entities using Micronaut. Having them as attributes disable the user association meaning that a user can like a video multiple time. While this might be acceptable in this assessment as no authentication needs to happen, I think its against the spirit of a social media platform.

Trending Hashtag Microservice

The purpose of this microservice is to give the top ten most liked hashtags over the last hour. This is implemented by listening to events produced from the VM (such as a video being added or reacted to), storing the relevant information and then parsing it and displaying it when needed.

This uses the same Hashtags and reactions domain objects as the VM as it makes communication between them simpler. It would be might be more efficient to have custom data objects but I argue that the trade-off of having it simpler for an expense in efficiency is worth it.

Subscription Microservice

This microservice deals with users subscribing to a hashtag and getting recommended videos. It works by listening to the events of the video microservice in order to be informed about what videos, users and hashtags there are, as well as what videos each user has watched. It uses this information to be able to suggest the next 10 videos a user should watch for a given hashtag. The users can then use the microservice to subscribe and unsubscribe to hashtags. Its ability to maintain a list of the 10 next videos to watch for each subscription is handled by implicitly by being able to suggest the next 10 videos a user should watch for a given hashtag. While there isn't any explicit data persistence for maintaining a list for every subscription, this is done by computing it on request.

It also uses the same data objects as the VM minus some details like video title and username as this can be gotten from the VM after the ids have been received from the SM

User Interface

There is 1 command line interface that a user can use to interact with the system.

Command	Parameters	Purpose
add-video	userId, title	adds a video authored by a user
get-video	id	gets the videos information from an id
get-videos-by-hashtag	hashtagId	gets the videos associated with a hashtag
get-videos-by-user	userId	gets the videos authored by a user
get-videos		gets all videos
update-video	id, title	updates the title of a video
add-user	username	adds a new user
get-user	id	gets the user information from an id
get-users		gets all users
watch-video	videoId, userId	records that this user has watched this video
get-reaction	id	gets the reaction information by id
react-to-video	videoId, userId, reaction	records a user liking or disliking a video
add-hashtag	videoId, tagName	adds a hashtag to a video
get-hashtags		gets all hashtags added to videos
get-video-microservice-health		returns ok if the VM is running
get-top-ten-hashtags		gets the top ten most liked hashtags within the past 60 minutes
get-trending-hashtag-microservice-health		returns ok if the THM is running
get-suggestions	userId, hashtag	gets a list of 10 videos that have this hashtag that the SM recommends to the user
get-subscriptions	userId	gets the hashtags a user has subscribed to
subscribe-to-video	userId, hashtag	subscribes the user to a hashtag
unsubscribe-to-video	userId, hashtag	unsubscribes a user to a hashtag
get-subscription-microservice-health		returns ok if the SM is running

Some things to note when using this platform is that:

- Videos cannot be added without a user
- Hashtags are only referred to by name not by id
- Users have to watch a video before liking or disliking a video
- Top ten hashtags records like's that happen to videos that currently have hashtags so if a hashtag is added retroactively it will not count
- Users can subscribe to hashtags have not been added to any video
- Users can get suggestions from any hashtag including ones they have not subscribed to

2.1.3 Containerisation

One way that this solution can scale up larger numbers of users, is by increasing the resources allocated to containers that need them. This may only help marginally as the main bottleneck could be caused by the sheer volume of requests rather than how expensive each request is. This can be counteracted by adding more containers which can share the request load, decreasing the individual load. It can be resilient to failures partially by having a separation of functionality meaning that if all the containers or nodes that are used for some part of the functionality, then users can still use the other parts. Having multiple containers for the same functionality will also help as if one fails then the other can pick up the requests that would have been directed to the failed one. Another way it is resilient is that by restarting the nodes that fail. It minimises downtime and so while it has failed, the impact to the user experience will be less.

In order to run the social media platform, you should: open docker desktop -> open each of the microservices top-level folders in git bash -> run `./gradlew dockerBuild` in each of them -> open git bash in the microservices directory (the one containing all microservices) -> run `./compose-prod-up.sh`







2.1.4 Quality Assurance

Automated Testing








Testing is a complex issue with many opposing views such as whether you should only test the interface or should you test the implementation as well. With this in mind its important use a variety of test methods to manage risk to a suitable level. This something I believe my implementation of tests both succeeds and fails at simultaneously. I say this because while it tests the controllers and the domain of the microservices thoroughly, there are large, and sometimes not simple, portions that are left untried. In most cases this would be unacceptable. However, I argue that in this case, this is acceptable. My argument is that (in regards to imperative programming) perfect testing is inherently unattainable. You cannot test for every scenario. Because of this testing becomes trade off between how much risk you are willing to put up with vs how much time you are willing to spend to achieve it. Kafka producers were not tested as spying on producers only works when there is 1 abstract method per class and actually listen to Kafka can take a long time, this combined with producing event being relatively simple in my view makes it not worth the time. Testing Kafka consumers would be much more of time and effort as they can have complex custom functionality which may easily go wrong, however testing these was never mentioned. Considering these microservices will never go into production (making the risk level a lot lower), I think that it is acceptable to allow the harder to test areas to go untested.

These are test coverage reports from Jococo:








videomicroservice

Element	Missed Instructions	Cov.	Missed Branches	Cov.
assessment.events		63%		n/a
assessment.controllers		92%		92%
assessment.domain		83%		n/a
assessment		0%		n/a
assessment.dto		100%		n/a
Total	150 of 990	84%	4 of 50	92%

trendinghashtagmicroservice

Element	Missed Instructions	Cov.	Missed Branches	Cov.
assessment.events		34%		0%
assessment.dto		0%		n/a
assessment		0%		n/a
assessment.controllers		96%		90%
assessment.domain		94%		n/a
Total	158 of 397	60%	3 of 12	75%

subscriptionmicroservice

Element	Missed Instructions	Cov.	Missed Branches	Cov.
assessment.events		20%		0%
assessment.domain		70%		n/a
assessment.controllers		95%		94%
assessment		0%		n/a
assessment.dto		62%		n/a
Total	285 of 694	58%	14 of 46	69%

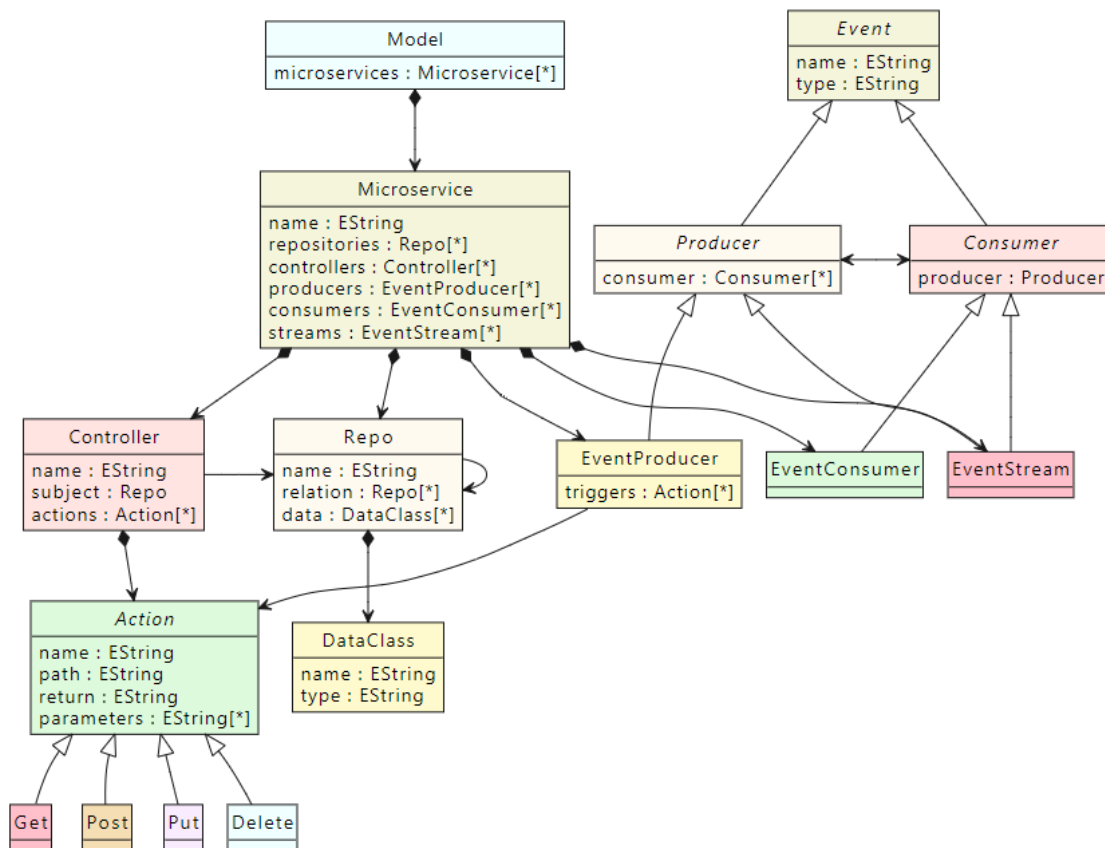
None my tests failed but as I've said there are large areas that have not been tested. This also does not mean that the bits that have been tested will work in all scenarios as there are a lot more variables that could be changed and could potentially cause an error of some sort.

Image Vulnerabilities

Since all the microservices share all the same packages, they all have the same vulnerabilities. Out of these, 4 of them are critical, 18 are high and 7 are medium all of which are fixable. All of these vulnerabilities are caused by using outdated images and packages. Most are from the alpine image all of which would be fixed by updating to 3.3.3p1-r3 which was released on the 24/10/2022, over a year ago. The other 2 were from the netty package which was fixed in version 4.1.100.Final released on 10/10/2023. Therefore, if this was to go into production I would update these packages to at least the version where the vulnerabilities were fixed, if not all the way to the latest stable build. I did not do this since the development was done on the lab pc's which have a fixed environment.

2.2.1 Metamodel

This is my metamodel:



This design was mostly guided by a mix of: the typical structure of this kind of microservice, the criteria for what must be included, what would make automated generation easier and how easily a none-domain expert would be able to use it. While typically a domain expert would be making the model, I decided to try and make the metamodel accessible as this allows experts to present and explain their model to others such as developers or other stakeholders, something that is important in a productive work environment.

The first design choice I discounted was having the Microservice as the top-level class rather than the Model. I initially liked the idea of modelling each microservice separately as separation of concerns is generally a very good design principle. However, this would limit the constraints I could put on it and more importantly disable the ability to view the event stream connections which would have made it harder to develop and present a group of microservices. While I haven't stated any positive of my decision but rather only negatives of the alternative, there are not any perfect solutions and so often it is better to opt for the 'least worst' solution.

An interesting design choice I made is the inheritance of the events. It does look needlessly complex, something I really tried to avoid, however I argue that: when making the model it isn't any more complex than having no inheritance, when viewing a model graphical syntax, it is more up to the syntax design to including any complexity and when presenting or explaining a model again if anything it helps as it allows you to give an appropriate level of abstraction. I chose this design mainly for the ease to have producers reference consumers and consumers reference produces, something that could have been difficult when you have a type that is both a producer and

consumer. Having them inherit from the Event class as well doesn't have much functional impact as the Event attributes could just be inserted into each derived concrete class, however it does enable new views in the graphical syntax should a designer want. The Microservice class could have contained all the concrete classes as Events, however this was discounted as it makes designing code generation more complex and it doesn't allow for a more heavily implied separation of concerns.

It was stated that modelling a microservice domain is not necessary for this assessment, and while this may be the case, I argue that a microservice (in the context of the ones we are creating for this assessment) only contain 3 things: a database (the domain), a public interface (the controller) and a private interface (the events). While all 3 of these can be optional and a microservice can function with only 2 of these, I would say that they are the quintessential building blocks of a microservice and so not modelling 1 of these would be under representative at the least and could be potentially damaging. This would also leave it database design up to the developer when it's more in the architect's responsibility.

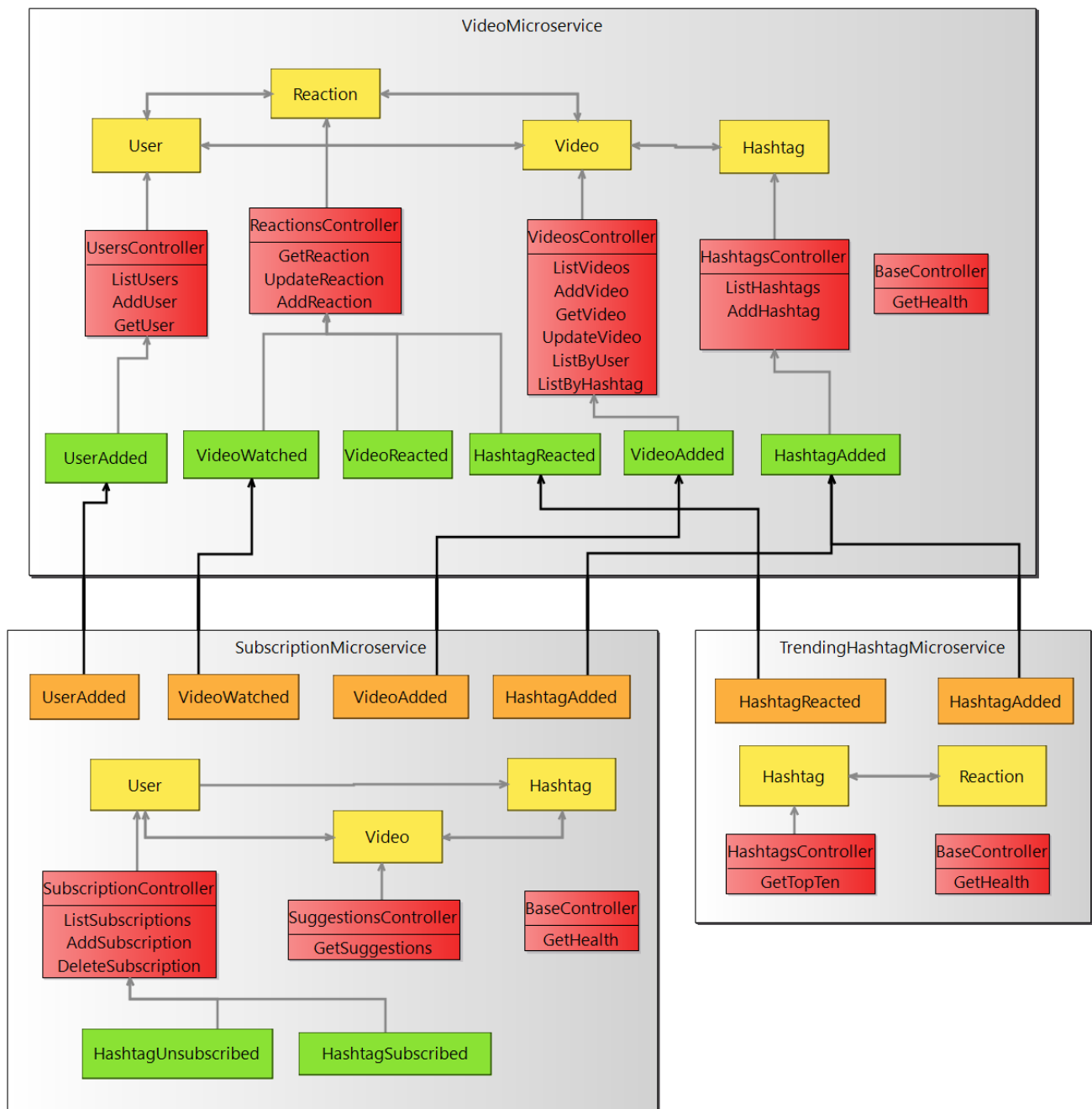
I initially wanted to model the domain further and include more detail in the Repository - Repository relation. I would have added the type of relation (such as ManyToOne and containment) as a lot of difficulties when developing the microservice were from domain issues. However even though I said it should be the architect's responsibility I decided to leave it more to the developer as I was conscious about modelling too much of the microservice or overcomplicating it which relation classes can often do.

The Action class (HTTP methods) have the attributes they do either because it made it easier to generate code and extend any generated code or because it required as part of the assessment. Action could have been a concrete class and have the different HTTP method types as an Enum that Action has as an attribute, however having it this way so that the methods are concrete types that inherit from action makes it easier for to view separately in a graphical syntax should it be designed that way for little if any change in complexity.

Another thing worth mentioning is that the EventProducer – Action relation has no impact on generated code, its purpose is to inform developers of where event should be triggered from and so that it can be shown in a graphical syntax to highlight the data flow.

2.2.2 Graphical Concrete Syntax

This is a screenshot of my model in its graphical syntax:



I aimed to give a clear overview for microservices with my graphical syntax; I wanted to be able to an image that you can present to others and they would get some value out of it (especially none-domain experts). My philosophy for doing this was mainly focused on logical grouping and on data flow as this help people understand how it structured and an insight to how it works.

In terms of grouping a lot of it is not enforced and so users can change the positions of objects within bounds. All the objects within a microservice have to be contained within the bounds of the microservice, but other grouping are up to the user. I have set up this representation to be clear by grouping EventConsumer -> Repo -> Controller -> EventProducer from top to bottom for each Microservice. This gives a clear positional separation for the types to reduce the graphical complexity and making it more cognitively manageable.

This particular grouping has the bonus that the relations don't cross each other too much and now make a clear data flow showing that data goes from the domain to the controller to the producer to the consumer. This isn't always true as the controller will both request and push data to and from the domain. You could also point out that in most cases consumers will be directly interacting with the domain but since that relation isn't in the metamodel, it would involve some guessing as to which domain it is interacting with.

You may notice that there aren't any EventStreams visible. This is mainly because my model of the microservices doesn't have any EventStreams. However, if one was my model I would have 2 main choices in regards to its design in the graphical syntax. I could have a different object for EventStreams and position it below producers and above consumers as it would likely be linked to a producer in a different microservice and a consumer in its own. Or I could model it as a consumer and a producer and so would appear as 2 different objects. I think the second option would look better from a data flow point of view and it may be easier to understand as a non-domain expert. However I chose to implement the first option as I think it's important to have it as its own object. Having it as 2 separate objects could cause big problems, imagine a developer who sees there is a supposed to be a stream from the generated code but can't find it on the diagram.

A decision I made that I'm not sure was the right decision is the direction of the relations (specifically the producer - controller and producer - consumers ones). This is because it might be better if they were directed the other way to show the direction of data transfer. I don't think it matters too much as the relation isn't meant to say there is definitely some data being transferred as this would make the Repo - Repo relation nonsensical but instead is meant to just represent that that is some sort of relation between the two objects. So maybe they should be in the direction that the data flows but that might give the wrong impression.

I did make another view for a more detailed view which expands the controllers to hold actions in the same way as the Microservice holds its objects instead of a list. This also allows the Producer - Controller relation to be more refined and actually show it to be Producer - Action. This is disabled by default as it makes the controllers a lot bigger for not much benefit.

All the objects are different colours so that people can easily differentiate between them however if we include the streams it would be at 6 different colours which is the limit on the human span of absolute judgment.

2.2.3 Model Validation

HasAtLeastOneMicroservice – Makes sure the model has a microservice, otherwise the whole model would be empty which would be pointless.

HasName – This is very common, and is used to make sure that the ones that need a name for the code generation do have a name

HasNoWhiteSpace – Also very common, makes sure the name has no whitespace so that the code generation is better

NameStartsWithUC – Also very common, suggests that the name starts with an uppercase so that code generation is better

HasController – makes sure the microservice has a controller as this is a necessary for this kind of microservice

HasHealthCheck – makes sure the microservice has a get method for checking the microservice is running correctly. It does have the possibility of failing as it uses the name so if a health check was under a different name or an unrelated method had 'health' in its name it could give an undesired result

InSameMicroservice – makes sure that all the repos that the repo is related to are in the same microservice as its possible to get confused when they have the same name

HasActions – makes sure the controller has at least 1 method and it would be pointless without

HasPath – makes sure all the HTTP methods have a path as otherwise the method won't work

PathStartsWithSlash – makes sure all the paths start with a '/' as otherwise the method won't work

NotTheSamePathAndType – tries to make sure that there are no actions that have the same full path (including port)

HasReturn – makes sure that all HTTP method have a return type for code generation

ProducerConsumer – suggests that all producers should have a consumer linked

ProducerHasTrigger – makes sure that every EventProducer is triggered by at least 1 action

ConsumerHasProducer – makes sure every consumer is linked to a producer

ConsumerHasSameNameAsProducer – suggests that the EventConsumer should have the same name as the producer its linked to for code generation

NotCircularStream – makes sure a stream isn't linked to itself

2.2.4 Model-To-Text Transformation

My m2t transformations were organised from experience from writing microservices in Micronaut. My process was for each type of file I needed in a microservice I would make an egl template for what was supposed to be in there and then create a rule such that it would get called for every object it relates to. E.g. for the controllers I copied a previously made controller, looked saw that they need certain repos producers and methods. Since I didn't know what repos would be needed I decided to inject all of them as it makes the developers job easier. Same with the producer although I knew there was only ever 1 so could just inject that straight away. For the methods I can use the controllers Actions to generate a template of what it would look like. I then looked at what model component this would correspond to which was of course the Controller class. I could then create a rule so that for every Controller in the model a micronaut controller class was created making sure to pass in the controller and the corresponding microservice so that the template could generate the appropriate repos and methods.

I tried to have a separation of hand written and generated code and succeeded in some areas, such as the controllers and consumers, however for the domain and repository classes there were some things that I couldn't override by extending and so had to use protected regions. Apart from those 4 sections almost all the microservice is generated from the m2t transformation. I could have included the docker setups however I was changing these fairly often and didn't want to change the egl every time. I thought about doing a few tests or at least the template but I realised that to write tests well, you have understand the meaning and intent of what the method is supposed to do.