ENG2 Open Individual Assessment

Exam number: Y3884331

Part 1 – Data Intensive Systems

2.1.1 – Architecture

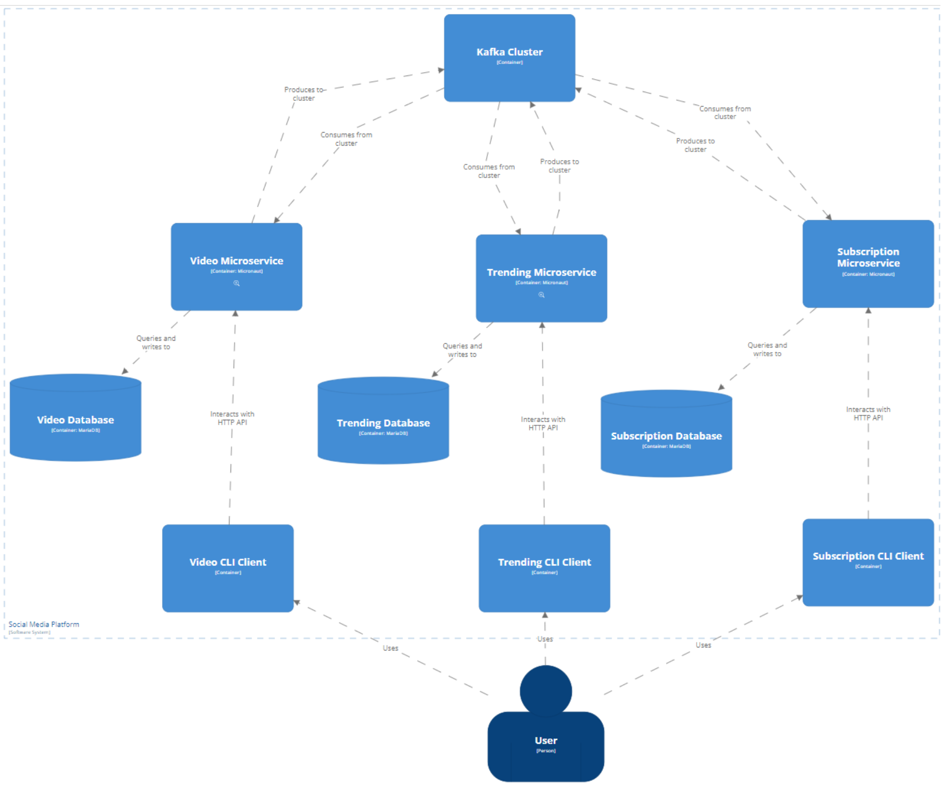


Figure 1: The overall architecture of the system

The overall architecture of the system is based around three microservices that maintain their own persistence, and communicate with each other within an event driven paradigm. The principles of scalable, data intensive systems were considered in the architecture phase of the project. The Kafka cluster, which is responsible for the event streaming features, receives and sends events to microservices that require them, which in turn change their database state depending on their internal logic. There is a separate CLI client associated with each microservice for interactivity and sending of information with a HTTP API. Each microservice maintains its own database based on the domain that they are associated with.

In particular, as the system was assembled to maintain only the necessary aspects of a video based social media platform, the ease of being able to add microservices further down the development pipeline was highlighted in the high-level design process. To aid this endeavour, each microservice publishes events to the Kafka Cluster for most processes it undergoes that change the state of the database. For example, the video microservice publishes events each time a user, video, or hashtag is added. It also publishes events each time any changes are made to the entities stored in the database, for example a video being liked or disliked, or a new watcher being added to a video. This means that the Kafka cluster always maintains events and topics pertaining to changes of state in each respective microservice database. Therefore, when new microservices are added, all they will need to do is subscribe to the relevant topics already created in the Kafka cluster and quickly be able to utilise the relevant information regarding the state of the overall system. Kafka enables multiple consumers to consume the same event due to the function of the "Consumer Group" [1].

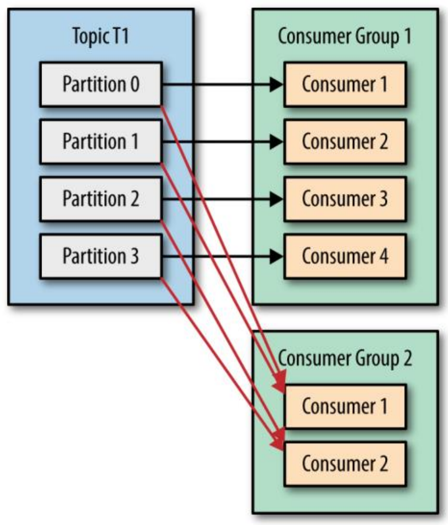


Figure 2: Apache Kafka Consumer Groups

Fig. 2 shows the concept of the consumer group, which means that new microservices will simply be able to create a new consumer group and consume the same events that may already be consumed by other microservices in the system.

Another aspect of scalability within the architecture is the ability for it to deal with a changing feature set. Other than its deployment on Docker containers, which is discussed in more detail in section 2.1.3, the architecture maintains scalability in this regard by having "loose coupling", [2] a facet of independent scalability [3] – which means that each microservice can be modified independently without affecting others. For example, the Video domain object can be internally modified to have a description field, but as long as the video value produced by the Kafka producer in the microservice does not contain the description, the subscription and trending hashtag microservices will not be affected in their operation.

In addition, the architecture must be able to cope with the demands of additional users effectively. One way this is accomplished in my architecture is through the use of decoupled databases; user queries to different microservices access their own databases, which mitigates issues with database access scaling. Instead of a single database needing to be accessed by many users, the requests are split between each individual microservice. An additional benefit of a decentralised data store is resilience – the accessing of the top trending hashtags or subscribing to a hashtag are not affected by a database outage in the video microservice.

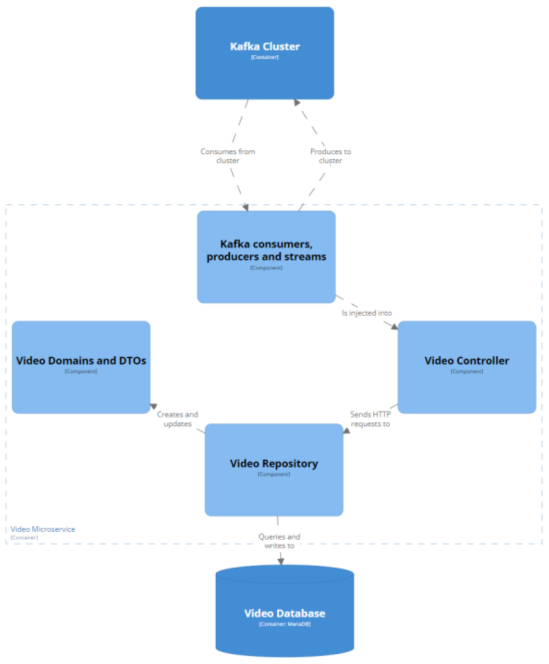


Figure 3: Architecture of the video microservice. Other microservices have identical structures.

2.1.2 – Microservices

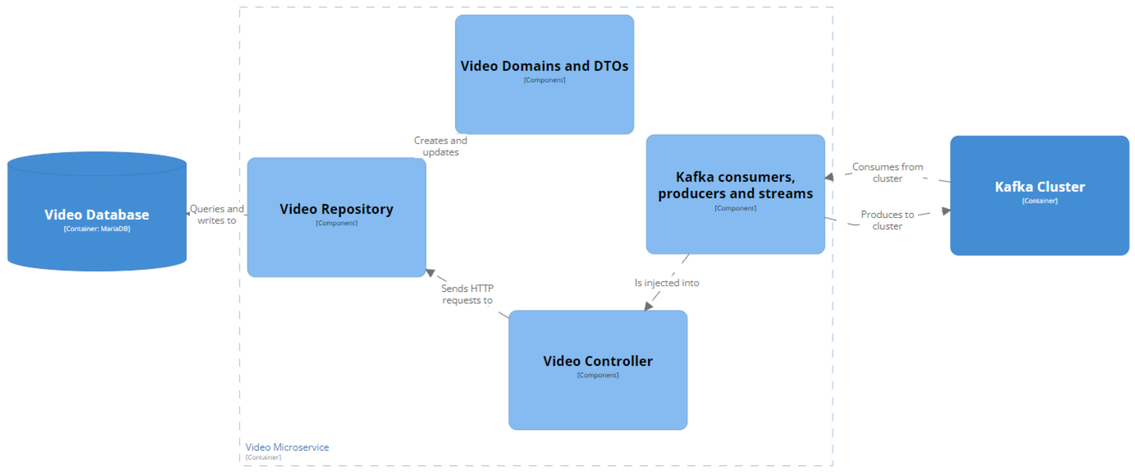


Figure 4 - Microservice overall structure. Trending/Subscription microservice structure is identical, their diagrams are located in "report" folder in submission for reference

The overall structure of the Micronaut based classes in the microservice follow a "domain centric approach" – which means that each domain element has its own associated repository and controller. The Kafka integration is based around each microservice having three classes for interfacing with the Kafka cluster – a producer class, a consumer class and a Kafka Streams factory class. Instances of the Kafka related classes are injected into the controller for any stream-based interaction that needs to occur, for example when a video is liked, the related method in the Video Controller that updates the like count for that particular video will also utilise the injected producer to produce an event into the "video-liked" topic in the cluster. An outline of the relationship between a domain element, a repository and a controller is outlined in figure 4 below using the example of the "User" class in the video microservice. A similar relationship is present within all the other domain elements in this microservice, and indeed for all of the other domain elements in the other respective microservices.

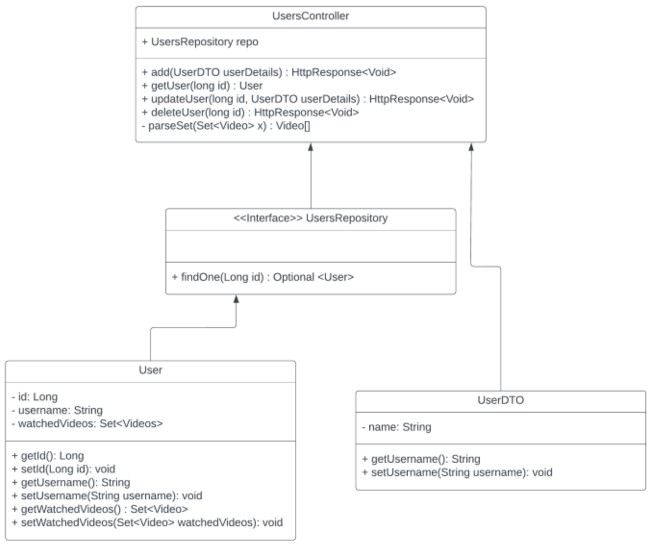

UsersController
+ UsersRepository repo
+ add(UserDTO userDetails) : HttpResponse<Void>
+ getUser(long id) : User
+ updateUser(long id, UserDTO userDetails) : HttpResponse<Void>
+ deleteUser(long id) : HttpResponse<Void>
- parseSet(Set<Video> x) : Video[]
<<Interface>> UsersRepository
+ findOne(Long id) : Optional <User>
User
- id: Long
- username: String
- watchedVideos: Set<Videos>
+ getId(): Long
+ setId(Long id): void
+ getUsername(): String
+ setUsername(String username): void
+ getWatchedVideos() : Set<Video>
+ setWatchedVideos(Set<Video> watchedVideos): void
UserDTO
- name: String
+ getUsername(): String
+ setUsername(String username): void

Figure 5 - The relationships between domain objects and their respective repository and controller

The Domain Transfer Object (DTO) is used to remove certain details about certain elements in the domain, which is especially useful with the command line interface; one does not wish the User to care for the id of a video when they create it, so a DTO is used in a variety of scenarios such as when a new user is created into the database.

Each microservice created is paired up with its own command line interface, utilising the Picocli framework.

The feature-set within the subscription microservice is based on requiring most of the information from the video microservice in order to be able to subscribe users to hashtags and display videos from a hashtag. For this reason, I made a design decision to mirror the database state between the video microservice and subscription microservice. The overall structure of the controllers in both microservice remain close in structure in order to aid ease of making mirrored changes in both class files. This database mirroring is accomplished by using Kafka consumers in the subscription microservice which consume the events produced by changes in state within the controllers of the video microservice. This mechanism is outlined below in figure 5.

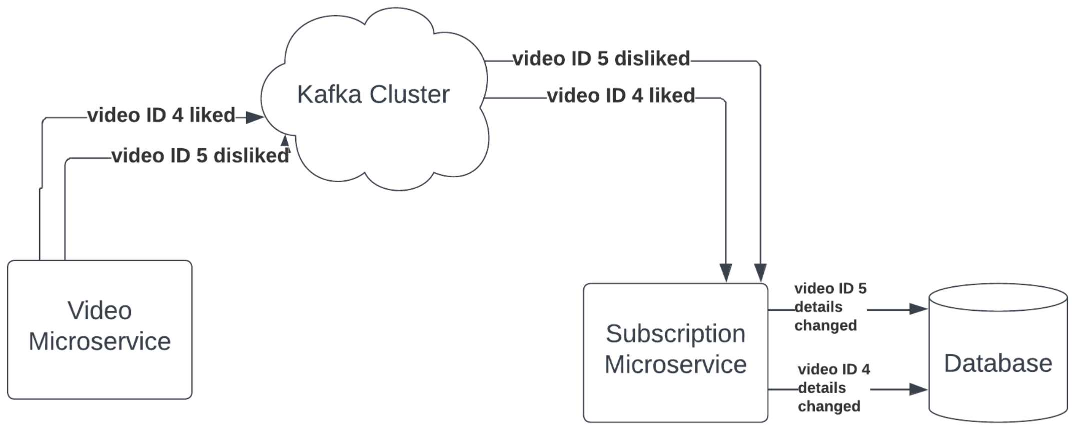


Figure 6 - How data parity is achieved between Video and Subscription microservice. NB: Video Microservice's database is updated in parallel, just not shown on this diagram.

Design decisions were also made within the domain class as well, particularly with scalability in mind. In particular, time complexity of access time was of key importance for domain attributes that potentially stored a large number of items. For example, the set of a user's watched videos is stored within a HashSet. This is of particular importance regarding scalability as for future microservices, it may be important to check if a user has watched a particular video in order to recommend them another one – this can be accomplished by using the ".contains" method on the hashset of watched videos. This access operation is far quicker – O(1) with Java HashSets than with other data structures such as ArrayLists, which are O(n).

Command line client usage

An important aspect of the video command line to note is that a video cannot be added without first creating a hashtag (with add-tag) and a user (with add-user), as a video must be initialised with an author and an initial hashtag. Additionally, most commands that reference specific domain values within the microservice will reference them via ID. For example, adding a watcher to a video is accomplished by referencing first the video ID, then a user ID.

The trending hashtag microservice client has a single command, which retrieves the top 10 most liked hashtags, which are stored within its own database.

The subscription microservice client has commands to subscribe and unsubscribe a user to a hashtag, and to view all the videos associated with a particular subscription. The persistence of the subscription microservice's domain entities mirrors that of the video microservice, so any IDs referenced can be assumed to be the same as the ones used in the video microservice. So, User 4 in the video microservice refers to the same user within the subscription microservice's internal representation.

A list of commands for each interface is located in a readme file in each respective directory.

2.1.3 - Containerisation

A number of design decisions were made with the composition and orchestration of docker containers for the overall system that made the system scalable and resilient.

Scalability

An important aspect of scalability is the distribution of workload. Docker enables developers to deploy multiple instances of a container, so that it would be able to handle an increased throughput. For example, if the number of users attempting to subscribe to hashtags became too high, this could be scaled by utilising horizontal scaling. The number of partitions within the Kafka Cluster is set to 6, as "more partitions leads to higher throughput" [4].

Resilience

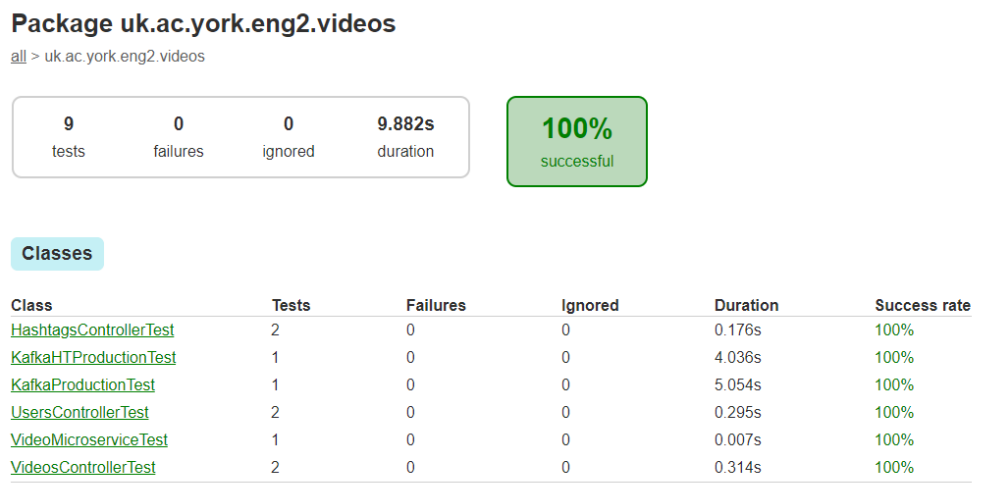
The Kafka implementation maintains three separate brokers. The main reason for this is due to resilience – one node is denoted as the leader, with other brokers replicate the leader to maintain data consistency in the case of a broker failing for any reason. Docker also maintains the resilience by offering a high degree of isolation – the fact that the container runs independently means it does not cause problems for other images running on the same server.

Security

With containers, there is a heightened level of security as only the necessary ports are accessible to the end user. In my case, it is 8080, 8081 and 8082 which are mapped to each particular microservice. No other ports are shown.

2.1.4 – Quality Assurance

A variety of approaches were taken in order regarding quality assurance in the software project. Every aspect of the microservice and Kafka microservice was tested, from the domain and various HTTP methods to the command line interfaces themselves. Below are outlined some of the results of the tests run.



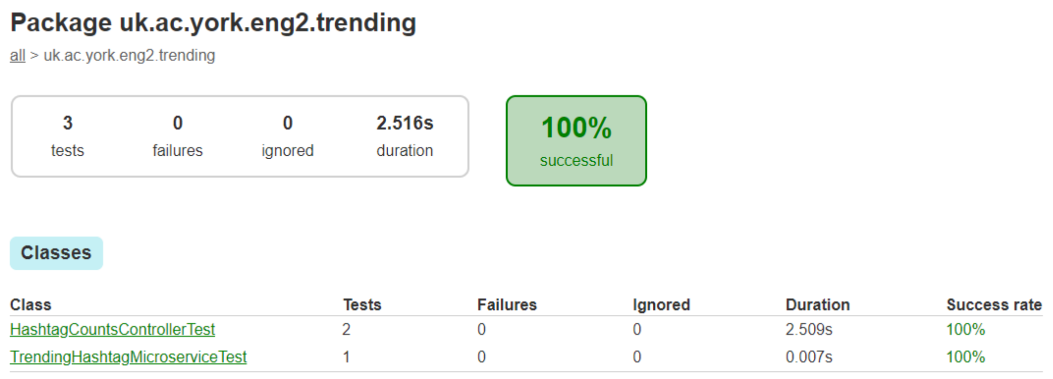
Package uk.ac.york.eng2.videos

all > uk.ac.york.eng2.videos

9	0	0	9.882s	100%
tests	failures	ignored	duration	successful

Classes

Class	Tests	Failures	Ignored	Duration	Success rate
HashtagsControllerTest	2	0	0	0.176s	100%
KafkaHTProductionTest	1	0	0	4.036s	100%
KafkaProductionTest	1	0	0	5.054s	100%
UsersControllerTest	2	0	0	0.295s	100%
VideoMicroserviceTest	1	0	0	0.007s	100%
VideosControllerTest	2	0	0	0.314s	100%



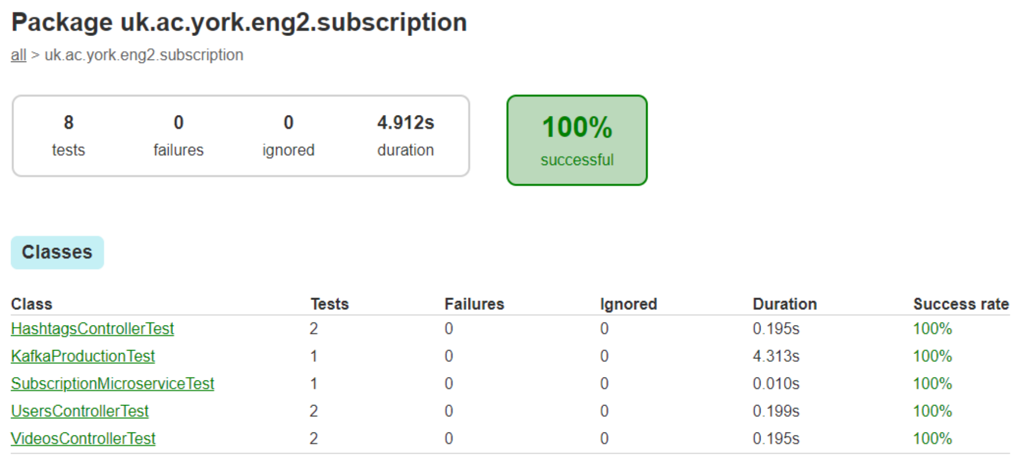
Package uk.ac.york.eng2.trending

all > uk.ac.york.eng2.trending

3	0	0	2.516s	100%
tests	failures	ignored	duration	successful

Classes

Class	Tests	Failures	Ignored	Duration	Success rate
HashtagCountsControllerTest	2	0	0	2.509s	100%
TrendingHashtagMicroserviceTest	1	0	0	0.007s	100%



Package uk.ac.york.eng2.subscription

all > uk.ac.york.eng2.subscription

8	0	0	4.912s	100%
tests	failures	ignored	duration	successful

Classes

Class	Tests	Failures	Ignored	Duration	Success rate
HashtagsControllerTest	2	0	0	0.195s	100%
KafkaProductionTest	1	0	0	4.313s	100%
SubscriptionMicroserviceTest	1	0	0	0.010s	100%
UsersControllerTest	2	0	0	0.199s	100%
VideosControllerTest	2	0	0	0.195s	100%

Testing the API

For each microservice, each API present in the microservice was tested. For example, within the video microservice, the VideosController, UsersController and HashtagsController were tested. The general formula was to create a test class for the controller and utilise a HTTP client to retrieve the necessary information, and test the input received.

```
@Test
public void UserTests() {


    UserDTO newuser = new UserDTO();
    newuser.setUsername("jack");

    HttpResponse<Void> response = client.add(newuser);

    assertTrue(addedUsers.containsKey((long) 1));
    assertEquals(HttpStatus.CREATED, response.getStatus(), "Creation should be successful");
    assertEquals(client.list().iterator().next().getId(), (long)1, "Author ID should be 1");
    assertEquals(client.list().iterator().next().getUsername(), "jack", "Author username should be jack");
```

Figure 7 - A test carried out on the User Controller of the Video Microservice

There were some additional steps in testing if a video was created, due to the fact that a user and a hashtag also needed to be present in the database before one could be created. In those cases, I had to instantiate the UsersClient and HashtagsClient and utilise them to create the relevant user and hashtag.

Testing Kafka

Kafka is tested in a variety of methods in the microservices. The main way was to check the production of events. As the trending hashtag microservice did not produce events, this testing was only undertaken in the video and subscription microservices.

Testing CLI

The CLIs were tested in a semi-automated fashion. The Video CLI, as it was based upon creating and getting information about the dataset, was easier to test automatically as an entity was able to be created then received. However, the trending hashtag microservice and subscription microservice populate their respective databases based on Kafka events that they receive, and the CLI cannot create these entities as the commands for them are not present as they are not necessary within the context of the application. For this reason, testing was done semi-automatically. Namely, video-cli has two test bash scripts, trending-test.bash and subscription-test.bash. The trending-test.bash works by creating multiple videos and multiple hashtags, liking them, then this populates the trending hashtags database. Then, one issues the "get-trending" command manually and checks the output. The way subscription-test.bash works is by issuing a variety of commands from the video-cli. The test has two phases – first, one must open up an adminer instance to make sure that the database state is identical between both databases, then you run the various subscription-based commands and check for output.

Conclusion

Ultimately, I was able to get good testing coverage with my microservices, and I was able to test extremely extensively in a manual way. I am confident the functionality works as intended. However, my *automated* CLI testing in particular remained constrained by time, and I would have liked to add more of this type of test. This would be something to improve upon in future iterations of this software.

NB: To run these tests, the application.ymls must be changed to localhost:3306/videos for the video microservice, localhost:3307/hashtags in the trending microservice and localhost:3308/subscriptions in the subscription microservice. If necessary, run the compose.yml provided instead of the production level compose-prod.yml as well. Additionally, the Kafka tests can sometimes arbitrarily fail. Usually, running the test again once or twice will initialise Kafka properly so that it will not fail.

Docker Image Analysis

Upon analysis of the three images that I created for the microservices, each one had a total of 29 vulnerabilities. In each case, 26 of these vulnerabilities came from the alpine image it ran on, one additional one from JDK17. My own actual microservices had 2 vulnerabilities each - GHSA-xpw8-rcwv-8f8p and CVE-2023-44487.

The GHSA- vulnerability is related to the vulnerability of the system regarding susceptibility to DDOS attacks. This is due to issuing frequent "RST frames". An RST frame is a TCP reset – when an unexpected TCP packet arrives at a host. The RST bit is set in the TCP header flags. The fix is to upgrade the netty version that I am using (4.1.94 to 4.1.100.Final or beyond.)

The CVE- vulnerability is related to preventing a HTTP/2 rapid reset attack. The functionality for this is to tell the server that a previous stream can be cancelled by sending an RST_STREAM frame. The fix is to upgrade the netty version that I am using (4.1.94 to 4.1.100.Final or beyond.)

Both of these issues, in each microservice (total of 6 *HIGH* level vulnerabilities) were solved by implementing a newer version of netty within build.gradle:

```
runtimeOnly("io.netty:netty-all:4.1.106.Final")
```

The other vulnerabilities were due to the version of Alpine Linux that was running on the containers. Unfortunately, the fact that these images were running on the lab PCs which have limited configuration options, these vulnerabilities could not be solved. However, the ones specific to the images I created were free of all vulnerabilities.

Another image that I used was the version of OpenJDK 17 specific to Alpine, which has 27 vulnerabilities (26 of which come from Alpine:3). These would have been resolved just by updating Alpine which as I have shown above is not feasible.

MariaDB has 56 vulnerabilities – 14 of these are from Ubuntu 22.04, most of which do not have fixes available yet however are classified as "low risk" by Docker Scout, so these are not of the most importance to fix. The majority of the vulnerabilities with MariaDB comes from the version of stdlib that it is running on – 1.18.2, and it generates many high level and two "critical" risk vulnerabilities.

Kafka 3.5 has 36 vulnerabilities, with 34 of these as low risk but with one critical and one high level risk. The critical risk is due to the version of zookeeper, which should be upgraded to 3.7.2 and Eclipse Jetty, which should be updated to 10.0.16

Part 2 – Application of Model Driven Engineering

2.2.1 – Metamodel

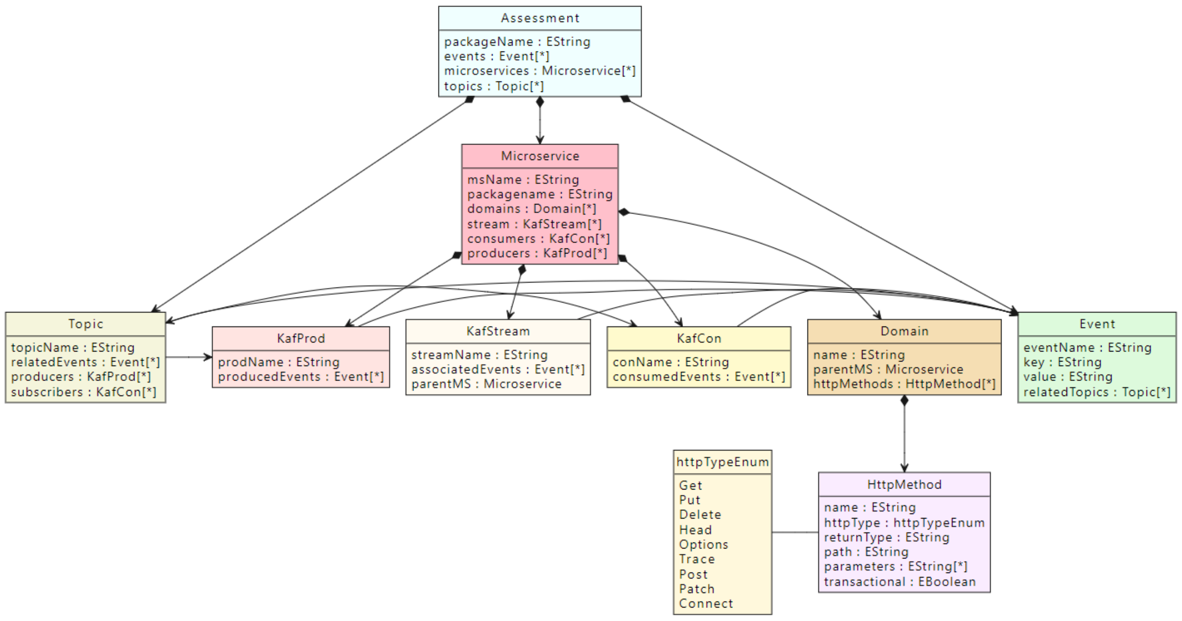


Figure 8 - Structure of the Metamodel

The basis of the metamodel was to create a *prescriptive model* that could define any microservices based data intensive architecture that utilised an event driven framework for fulfilling its event driven processing requirements. The structure of the metamodel is based on establishing the required details for the microservices and the event driven aspects of the system.

The microservice is described in the metamodel as: its domain elements that it stores for persistence and the three types of Kafka related classes that each microservice can contain (producer, consumer and stream). The Kafka aspects are described as: a series of topics that are stored on the cluster and the relevant events that correspond to those topics. The way these two parallel systems interact are by relations defined between the producer/consumer/stream classes and the events and topics. For example, Microservice A has a producer which is related to Event X (a part of Topic J), and Microservice B has a consumer which is related to Event Y (a part of Topic J). These Microservices are now able to communicate with each other. As one microservice has a producer (*KafProd*) to a topic that has a consumer (*KafCon*), this defines a link.

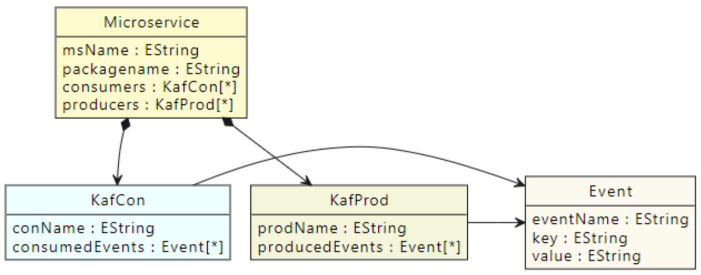


Figure 9 - Details of how a microservice is linked to consumers, producers, which are in turn linked to events

Design decisions were made to maintain the highest levels of abstraction in this metamodel so that it would be able to generalise to any implementation details, which would be left to the developers implementing it into an actual use case. Fine details of the domain elements such as the attributes that they would be populated with are not present, only the ways in which these elements interact with their own databases (located in the HttpMethod class). In fact, the only non-Kafka / non event-driven aspects of the microservice is the Domain and its child elements of the HTTP API related classes. Other key features to note is the use of an Enum in httpType, this was changed early on from a basic string field as there is a finite amount of HTTP methods that are available.

The method of making the model was a top down approach - I defined all of the model elements based on *generalising* to have the right amount of information that is required to describe an architecture rather than requiring only the model elements that have a functional purpose with my specific Micronaut/Kafka system.

Decisions were also made to create a system that could be utilised by domain experts, to follow the examples and types of use-cases presented throughout the teaching of the module. Details about table joins to describe relations between domain elements are removed, with the only aspects describing the inter-relationship between domain elements being present in the HTTP methods. These may take, for example, a User as an argument when requesting record(s) from the Video repository. The HttpMethod name should be descriptive enough in this case to be able to parse the nature of this method and therefore the nature of the relationship, for example if the model has a Video HTTP method called "getAuthorVideo" then the implementation team will be able to understand that the Video entity itself must have a relationship with User to describe an Author. This removes the necessity for domain experts to define the entity relationships such as "many to many", etc. Similar decisions are taken throughout the entire metamodel.

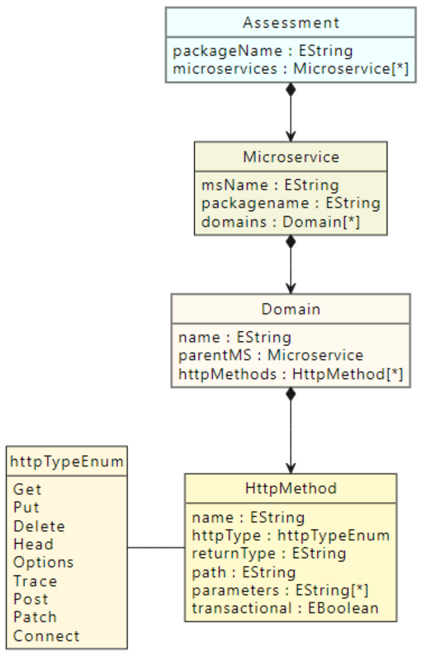


Figure 10 - The relationship between the different microservice related aspects of the class

2.2.2 – Graphical Concrete Syntax

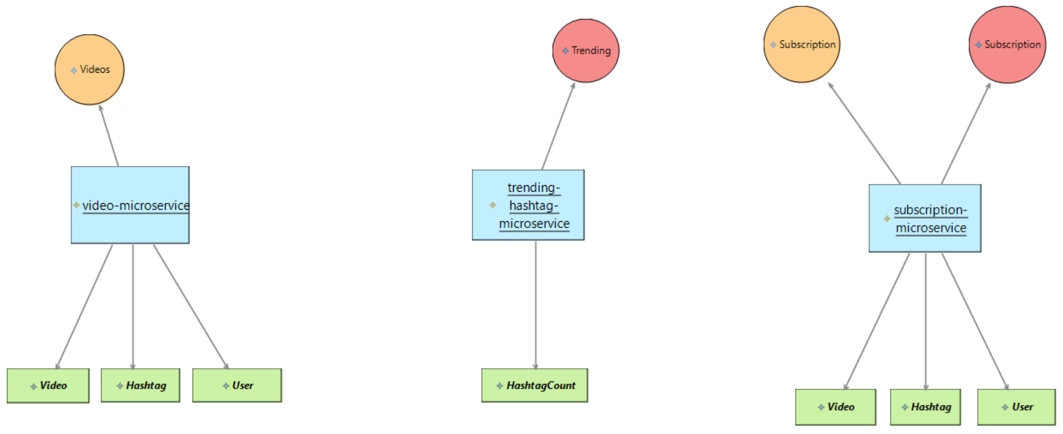


Figure 11 - High level overview of my model – further views below

The graphical concrete syntax was made in a way that would allow implementation experts to be able to view the final model in a high level, event streaming centric and domain centric layer. Figure 10 contains the high-level default layer, which is each individual microservice isolated from each other, with their Kafka elements and domains. The event streaming centric layer shows how each producer and consumer communicates with each other (Figure 11), and the domain centric layer shows all the different HTTP methods used by all the microservices (Figure 12). Due to the number of HTTP methods, this view is off by default as it creates a very large network which can obfuscate the relative importance of each element in the syntax, but it remains in the diagram as I believed that it was necessary in certain edge cases. For example, it allows you to view from a high level the number of methods that are associated with each domain element, then zoom into the diagram to view each one in greater detail. This is to make sure that complexity is well-managed.

The theory behind this usage of layers is to deal with complexity management – this reduces cognitive overload by initially giving the user only the necessary information in the default layer, then the user being able to add complexity when they feel necessary, rather than it being part of the default view.

The syntax was based around exploiting the brain's visual system. There were certain design decisions that were utilised throughout, such as the separation of microservice based classes into rectangle shapes, and event streaming based classes into circular shapes. This aids in giving the model semantic transparency, as one can view that the overall design of these two types of shapes (square and circle) are due to a difference in function.

The spatial positioning of the domain and its related elements was also by design. The domain elements are low level details that define interaction with the database and reflect the finer points of the overall system. As you go "upwards" through the diagram until the microservice, it represents a higher level container for the specific microservice. Then, at the top of the diagram are the event streaming based elements such as the producers, topics and consumers. These represent the networking between microservices, and in the case of Kafka, these access the cluster in order to be passed to different microservice. The top layer of topics (Figure 11) represents the cluster, in a symbolic way.

Other positionings within my syntax are by design; related to each microservice, the producer is always to the upper left of the microservice while the consumer is to the upper right.

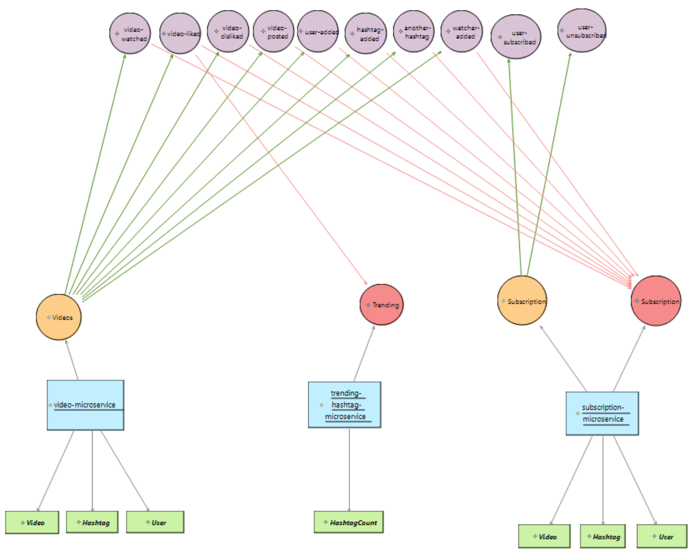


Figure 12 - The interaction of each microservice with the event streams

Colour theory was also utilised to make an effective syntax. All the elements used light colours rather than dark colours so that text would always be visible, while arrows remained dark in colour so that they could be easily discriminable to clearly view the source and target. This is especially important in some views where there are many arrows. The producers are all in orange, while the consumers are all in red. This was done to indicate a similarity due to being different hues within the red-yellow range, but a difference in function. The connection between these producers and consumers and the topics themselves, also follows a design decision – producing an event is green, while the consumption of an event from a topic is red.

Additionally, the directionality of arrows also had a purpose – the arrows going from consumers to topics to producers were all in the direction of event transfer, the arrow from the microservice to the consumers, producers and domain were also directional to show the originator of those microservice elements.

The decision was made to not include Events within this graphical syntax, as they do not aid in showing the overall communication between microservices – as events are components heavily related to topics, simply showing the topics and their connections was enough.

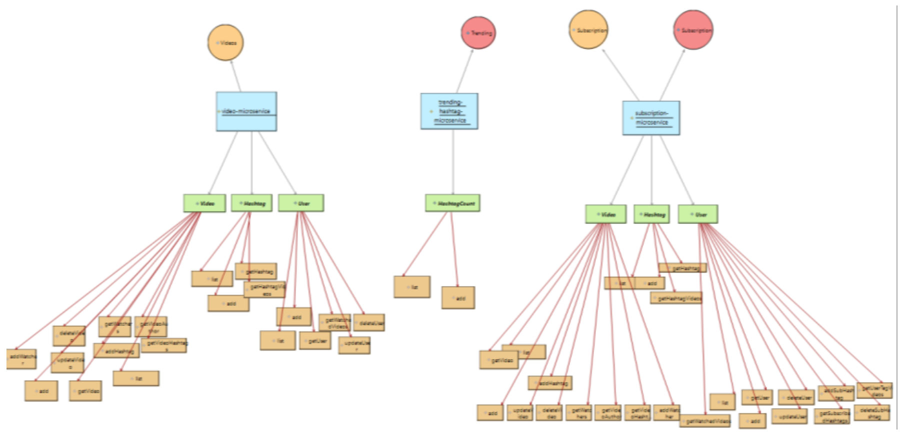


Figure 13 - Domain centric layer of the syntax

2.2.3 – Model Validation

Validation was accomplished by using the Epsilon Validation Language, and making sure that the model conforms to a variety of standards that must be upheld by any model of this type. Broadly speaking, the types of validation rules can be broken up into several different types, which are described as follows:

Field Presence

An important aspect of any of the model elements was the presence of a name. This is especially important for identifying different elements, and making sure that when a particular part of the element causes a validation error then the error message gives a descriptive error message. For example, if a topic does not have a consumer and subscriber, the error message calls the topicName eAttribute to identify the erroring topic – this is especially important when there may be many topics in a model. Other than names, other fields that needed presence were the return types of HTTP methods, and paths for a method.

Uniqueness

Certain aspects of my model necessitated checking that they were unique from an identifier perspective – in particular, each event must have a unique name, and each microservice must have an event name. The uniqueness of a microservice name is particularly important in the code generation part of the task, as otherwise two different microservices with the same name would overwrite each other's code as the file path for the microservice related code utilises microservice name to decide where to generate the src-gen folder.

Formatting

Due to various constraints in each class, I imposed certain formatting constraints throughout particular EAttribute fields. For example, the key and value in an event cannot have spaces, because they are representative of a datatype in a programming language. The vast majority of languages do not support spaces in their datatypes, so this constraint was enforced. For similar reasons, a domain was forbidden from starting with a number. Other formatting constraints are in the HTTP method class, where a path must begin with a "/", and similarly the return type cannot have a space.

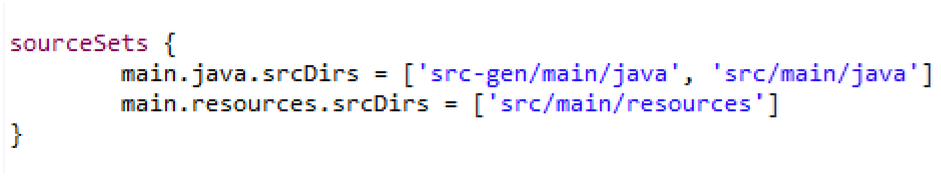
Other constraints

Other constraints were also used, especially those to meet the specification required in the task:

- Presence of at least one microservice is checked
- Presence of at least one event stream is checked
- A critique is present to ensure that each event stream has a consumer and producer
- A constraint is *not* present to check if a health resource is present, as this is handled by the EGX – it ensures that as long as microservice is created, a health resource will be automatically generated every time. Therefore, this specification is met outside of the metamodel/model.

2.2.4 – Model-to-Text Transformation

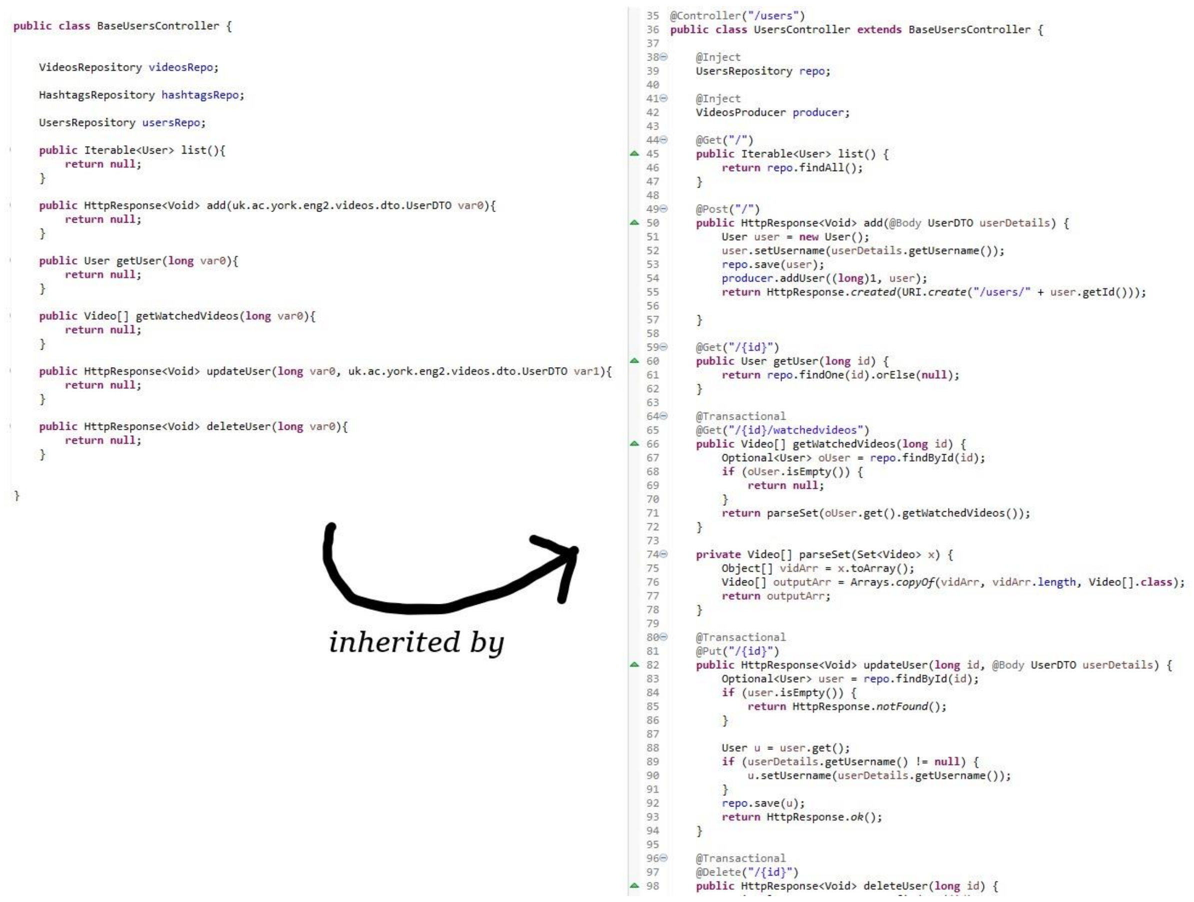
The model to text effectively transformed my final model into Java code. The structure of the generated code is that it is all generated into a src-gen folder in each microservice root folder. The build.gradle is set to look for the src-gen files as a source, and my handwritten classes inherit from this generated code.



```
sourceSets {
        main.java.srcDirs = ['src-gen/main/java', 'src/main/java']
        main.resources.srcDirs = ['src/main/resources']
}
```

Figure 14 - Declares the source for the newly generated code (in src-gen)

For example, UsersController inherits from BaseUsersController, with BaseUsersController containing every single method that is used in UsersController. This is showcased below:



```
public class BaseUsersController {


    VideosRepository videosRepo;

    HashtagsRepository hashtagsRepo;

    UsersRepository usersRepo;

    public Iterable<User> list(){
        return null;
    }

    public HttpResponse<Void> add(uk.ac.york.eng2.videos.dto.UserDTO var0){
        return null;
    }

    public User getUser(long var0){
        return null;
    }

    public Video[] getWatchedVideos(long var0){
        return null;
    }

    public HttpResponse<Void> updateUser(long var0, uk.ac.york.eng2.videos.dto.UserDTO var1){
        return null;
    }

    public HttpResponse<Void> deleteUser(long var0){
        return null;
    }


}
```

```
@Controller("/users")
public class UsersController extends BaseUsersController {

    @Inject
    UsersRepository repo;

    @Inject
    VideosProducer producer;

    @Get("/")
    public Iterable<User> list() {
        return repo.findAll();
    }

    @Post("/")
    public HttpResponse<Void> add(@Body UserDTO userDetails) {
        User user = new User();
        user.setUsername(userDetails.getUsername());
        repo.save(user);
        producer.addUser((long)1, user);
        return HttpResponse.created(URI.create("/users/" + user.getId()));

    }

    @Get("/{id}")
    public User getUser(long id) {
        return repo.findOne(id).orElse(null);
    }

    @Transactional
    @Get("/{id}/watchedvideos")
    public Video[] getWatchedVideos(long id) {
        Optional<User> oUser = repo.findById(id);
        if (oUser.isEmpty()) {
            return null;
        }
        return parseSet(oUser.get().getWatchedVideos());
    }

    private Video[] parseSet(Set<Video> x) {
        Object[] vidArr = x.toArray();
        Video[] outputArr = Arrays.copyOf(vidArr, vidArr.length, Video[].class);
        return outputArr;
    }

    @Transactional
    @Put("/{id}")
    public HttpResponse<Void> updateUser(long id, @Body UserDTO userDetails) {
        Optional<User> user = repo.findById(id);
        if (user.isEmpty()) {
            return HttpResponse.notFound();
        }

        User u = user.get();
        if (userDetails.getUsername() != null) {
            u.setUsername(userDetails.getUsername());
        }
        repo.save(u);
        return HttpResponse.ok();
    }

    @Transactional
    @Delete("/{id}")
    public HttpResponse<Void> deleteUser(long id) {
```

Figure 15 - Users Controller inheritance. The green ^ arrows to the left of the methods indicate overriding

A key thing to note is the nature of the code generation. Unfortunately, due to the nature of Java inheritance, Java annotations cannot be inherited by child classes. This means that even though my module is detailed enough to annotated each controller method with the appropriate @Post/@Get method, each Kafka consumer as a @KafkaListener, these aspects were not included in the final generated code, as they could not be inherited. However, this is a strength of my model because it gives future EGX developers the freedom to generate the type of code that they want; they can either go the same route my code did, where I generate the skeleton to be inherited, or they can generate the actual classes themselves and utilise protected regions – this means that Java annotations can be kept.

In addition to the class files, I also made the decision to be able to generate additional files. These are not inherited by gradle directly in my project, but rather are supplementary files which can be utilised by developers. This includes the compose.yml file for the project, build.gradle files for each microservice and application.yml files for each microservice. These are all fully functional files, with no overwriting needing to be done – and can be utilised.

modeling\m2t\uk.ac.york.eng2.assessment.y3884331.m2t\other-generated-files

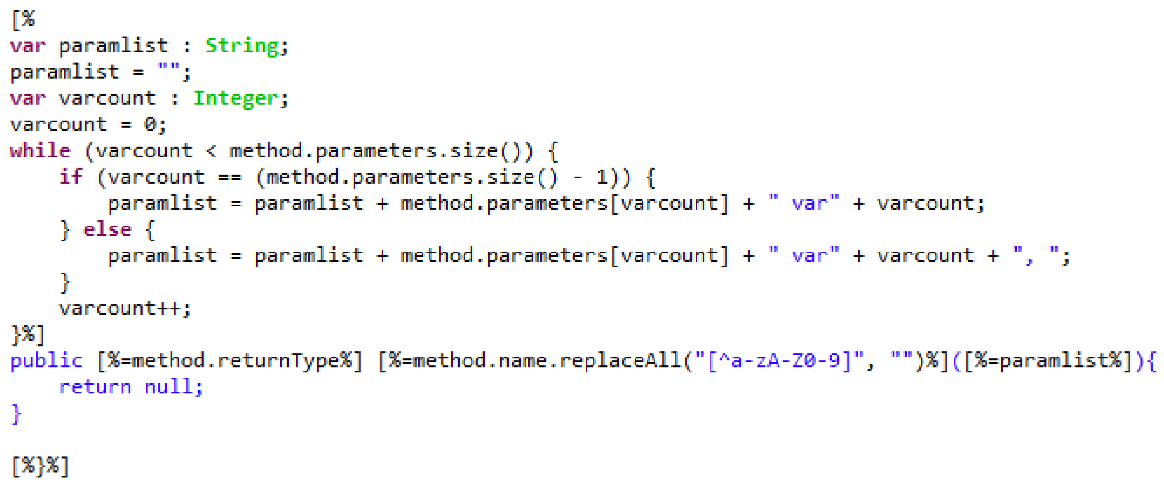
Things of interest within my code generation

Due to the nature of expanding on my model to create fully fledged Java code, there were a number of design and implementation decisions that I undertook, which I will now outline.

The function of much of my EGL code doubled as an extra layer of validation. For example, for any java class name, I always used the ftuc() method (first to upper case) to ensure that the first letter was upper case, as is the syntactic requirement in Java. I also made sure that class names could not have any hyphens by using a String.replaceAll function and pattern matching with regex.

In order to make sure that my code generation would be “portable”, my .egx file utilises relative filepaths which moves up the modelling/m2t directory and then into the actual active microservices. This ensures that this code can be generated across different machines.

Another challenge was generating variable names for parameters. I did not wish to put variable names as part of the metamodel itself as that would be unnecessary overhead for domain experts to deal with, so I generated variable names within the EGL files by using a loop to increment a counter, and using this to generate the parameters in the controller method. This is shown below.



```
[%
var paramlist : String;
paramlist = "";
var varcount : Integer;
varcount = 0;
while (varcount < method.parameters.size()) {
    if (varcount == (method.parameters.size() - 1)) {
        paramlist = paramlist + method.parameters[varcount] + " var" + varcount;
    } else {
        paramlist = paramlist + method.parameters[varcount] + " var" + varcount + ", ";
    }
    varcount++;
}%]
public [%=method.returnType%] [%=method.name.replaceAll("[^a-zA-Z0-9]", "")%]([%=paramlist%]){
    return null;
}

[%}%]
```

Figure 16: Generating the methods in each controller

References

[1] Confluent *Kafka Consumer: Confluent Documentation* [Online]. Available: URL https://docs.confluent.io/platform/current/clients/consumer.html#consumer-groups

[2] Nordic APIs *How to design loosely coupled Microservices* [Online]. Available: URL https://nordicapis.com/how-to-design-loosely-coupled-microservices/

[3] Tanzu *Should That Be a Microservice? Part 4: Independent Scalability* [Online]. Available: URL https://tanzu.vmware.com/content/blog/should-that-be-a-microservice-part-4-independent-scalability

[4] Confluent *How to choose the number of topics/partitions in a Kafka Cluster* [Online]. Available: URL https://www.confluent.io/en-gb/blog/how-choose-number-topics-partitions-kafka-cluster/