*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 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. Furthermore, Kafka similarly allows scaling, it currently hads 3 nodes but more can easily be added. Kafka handles clustering very easily, having one leader node and the rest are in eventual consistency.

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.

A blue silhouette of a person

Description automatically generatedA diagram of a company

Description automatically generated

Level 2: Containers

Key

Level 1: Context

A screenshot of a computer screen

Description automatically generated

A diagram of a company

Description automatically generated

Level 3: Components (Same for all 3 microservices)

1.2 Microservices

**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)

**Trending Hashtag Microservice**

The Trending Hashtag Service is responsible for calculating the trending hashtags in the system. It does this by listening to 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.

**Subscription Microservice**

**TODO**

**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.
* trending-hashtags - Gets the top 10 trending hashtags in the system.
* subscribe **TBD** - Subscribes a user with the given id to the hashtag with the given name.
* unsubscribe **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 – 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 – Implemented on each microservice and resource to aid with monitoring
* Loose coupling – 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

**Docker Image inspecting**

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

**A screenshot of a computer

Description automatically generated**

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.

A screenshot of a video chat

Description automatically generatedThe 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.

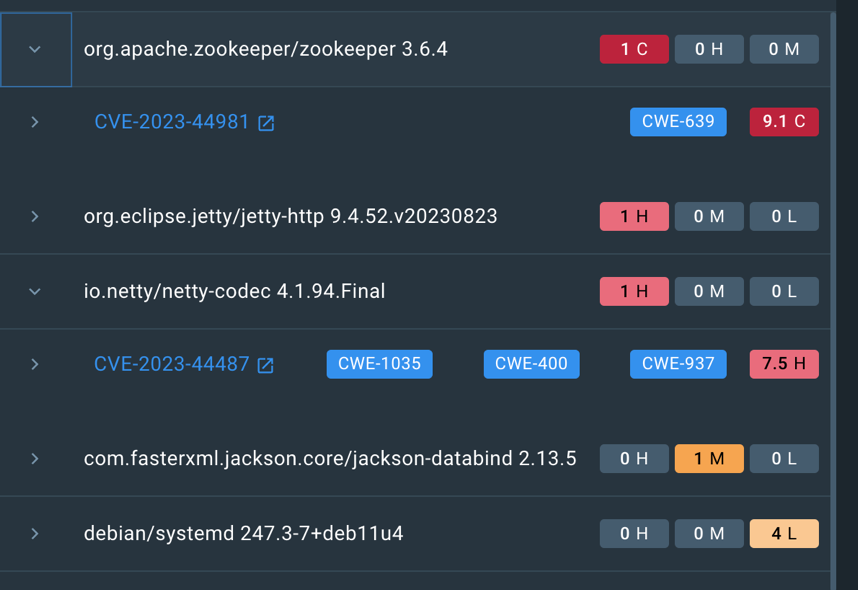
Figure 1 - Microservice docker scout scan before fix

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.

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

****

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.

A diagram of a computer

Description automatically generated

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.

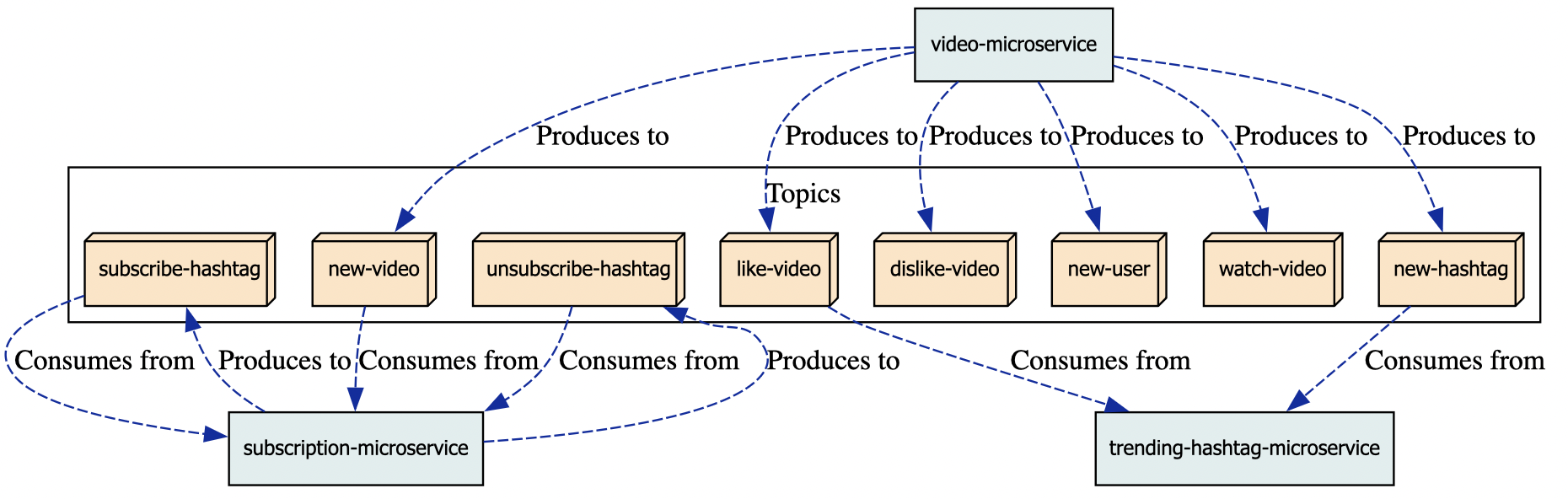
Microservices also incorporate controllers, featuring a name, a path (base URL path for controller endpoints), and API endpoints represented by instances of the "Endpoint" class. Endpoints include a name, URL path, and an enumerated endpoint type (GET, PUT, POST, DELETE). This simplifies the representation 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.

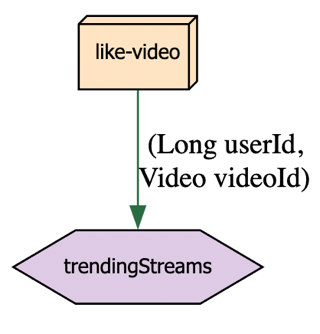
Lastly, each microservice includes a "Repository" with a name associated with the entity it manages. The separation of Repositories rather than assuming a repository for every entity, allows persistence of entities to be optional. Rather than having a Boolean for if an entity is persisted or not, a repository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

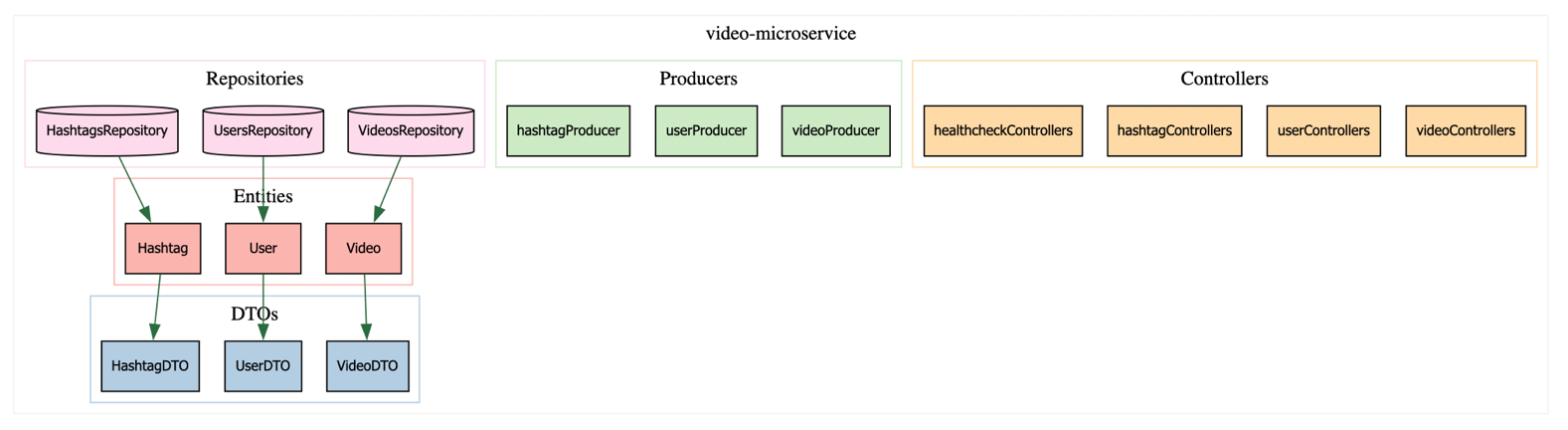
I have assumed that each eventStream will only have one type of key and one type of value entering it.

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.

A diagram of a diagram

Description automatically generatedA diagram of a video

Description automatically generatedA diagram of a user flow

Description automatically generatedA diagram of a video producer

Description automatically generatedTODO: REPLACE THESE IMAGES and label

The highest level of my graphical representation of a system shows only the microservices as a box and the topics as oranges 3dBoxes. Blue dashed lines shows the links between microservices and EventStreams, with the arrows direction showing if it produces to or consumes from the respective stream. This is because it gives an overarching view of event flow in the system, which is very important in an event driven architecture. The choice to show topics as 3d boxes, shows that it holds data which increases the semantic transparency of the graphical representation. The dashed line is used throughout the graphical syntax to show the input/output of information/data/events, so this creates consistency.

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. The important part 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.