STREAMSTER

Video streaming application

Michaela Golhová - 266099

Matej Michálek - 266827

Michał Karol Pompa - 266494



Supervisor: Behnam Boujarzadeh

VIA University College

VIA University College

86148 characters

Software engineering

7th semester

18.12.2020

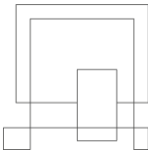
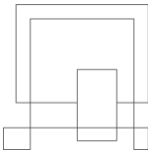
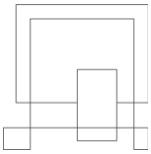
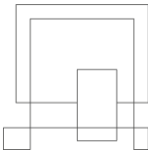


Table of content

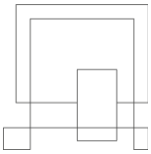






Abstract

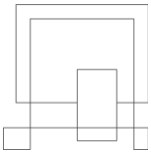
The following document describes the development of a platform with a purpose to improve the experience of video streaming solutions. It is divided into eight parts. In the first one, the background of the project is considered and the current state of the existing solutions is discussed. In the next part, the system's needs were analyzed and the requirements were stated. Moreover, the distribution of the functionalities between the actors was illustrated using diagrams. Furthermore, the recommendation algorithm was brought to a focus. In the following part, the system's design was discussed and choices about the architecture and technologies to be used were considered. Additionally, core functionalities such as video streaming or file storage were given more attention. After the Design, the next topic is Implementation. In this chapter, all relevant implementation details are described, such as details of the recommendation algorithm, challenges of video streaming and file upload or characteristics of Bloc pattern. Afterwards, the topic of the tests is brought up, presenting the testing methodology used to verify the completion of the requirements and tools used for Unit Testing the implementation. The document is concluded with chapters where project's results are discussed and its possible future considered.



1 Introduction

In February of 2005, a new era of video entertainment started when the YouTube platform was launched. In contrast to traditional television, it offered watching videos at any time from a wide selection of user-created content. It allowed the type of freedom, that perfectly suited new generations, that had grown up surrounded by internet and multimedia. 15 years later, YouTube is the biggest source of video content in the world. Alongside with Netflix, HBO GO, Amazon Prime, and other streaming services. It is dominating over television in attracting user's attention. As the research made by Defy Media shows, young people between the ages of 13 and 24 are spending around 20 hours a week on online video and streaming platforms, compared to 8 hours spent on watching TV (DEFY Media, 2015). Those statistics clearly show that the future of video entertainment is in the online video streaming platforms and their importance in young people's life cannot be neglected. Another important factor is the portability of the new media. The freedom of watching your favorite videos wherever you are is superior to a need to take a seat next to a TV set. According to Google, "3 in 4 adults report watching YouTube at home on their mobile devices". This points out that the mobile part of the audience cannot be overlooked (Google, 2020).

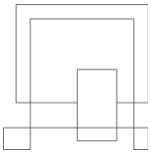
However, the existing platforms are not perfect. Taking into consideration YouTube, it has changed significantly over the past years. It is no longer a platform for sharing videos with friends and family as it was before. Right now, its functionality is focused on earning money on the high-quality content materials made by its community. As the platform started to support the professionals more and more, its function to simply share videos has been moved aside. YouTube started to delete videos from channels that do not fit its policy. However, it is not the humans that are checking the videos, but the algorithm called ContentID. As shown in the article by Maximillian Laumeister, in many cases, the aforementioned algorithm is not treating the videos correctly, deleting or disabling monetization on those videos that are original and do not violate copyrights. He brings up a case of a Music Sheet Boss channel, whose channel was demonetized for the reason of "repetitive content" with no way of appeal (Maximillian Laumeister, 2019). Moreover, its video promotion algorithm is more likely to show you a video that has many views than a video uploaded by your friend, since it is not popular. Other victims of that



functionality are small passionates that create content for a certain niche of viewers. They do not seek great popularity and all they require is a fair way to share their videos, not depending on the number of views they make. In addition, one of YouTube's biggest strengths is also one of its biggest weaknesses. While offering the biggest selection of videos in the world, it does not provide an advanced way of searching for desired content. It does not allow to narrow down the search results using filters or additional criteria. Another issue that disturbs the watching experience is the advertisements that have recently become harsher and more aggressive than ever. In this matter, YouTube gets very similar to traditional TV by serving ads before and during the video.

As shown above, the demand for video streaming platforms is high. However, there is also a room for improvement. A new platform would have to solve the problems of existing solutions to confront the consumer's needs. Moreover, with a great number of similar platforms, a new platform, to be successful, it must differentiate itself from other solutions existing on the market, offering functionalities that can change the way the people are watching videos online. It is also required that it treats every user equally. No matter how popular their channel is, their videos will always reach its viewers. It must provide a simple and reliable solution for sharing videos within communities. Moreover, it must have a flexible search engine that helps to find desired materials and it should not distract a viewer from watching by playing advertisements. In addition, it should serve as a cross-platform solution that allows usage of the platform on desktop and mobile devices. At last, as the expected audience is counted in millions of viewers, it must be resilient to the great number of users using the platform simultaneously.

Further details about the project background can be found in Project Description – Appendix A.



2 Analysis

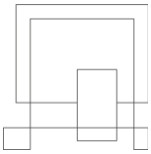
The first stage in the process of software development is the Analysis. There are three main deliveries that are produced during this stage – Requirements, Domain Model and Use Case Diagram with particular Use Case Descriptions.

2.1 Requirements

Based on the description of the desired functionalities and needs of the selected personas, a list of requirements has been made. The requirements have been formulated in Software Requirements Specification (SRS) using IEEE Standard 830-1998 (IEEE Recommended Practice for Software Requirements Specifications, 1998). SRS document can be found in Appendix B. However, to give an overview of the project scope, a following list of non-functional requirements and user stories has been created based on SRS.

2.1.1 Non-functional requirements

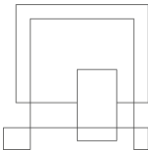
1. The system must be able to allow at least 1000 users to watch video at the same time.
2. The system must be functional for a user with an internet connection with a speed of at least 10 Mb/s.
3. The system must have a 95% availability.
4. The system must store all passwords encrypted.
5. The system must ensure that all network traffic between client applications (Android application and web application) is secured using TLS.
6. The Android application must support Android in version 8.0 or higher.
7. The web application must support Chrome browser in version 81 or higher.



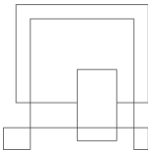
2.1.2 Functional requirements – user stories

From the software requirement specification were created functional requirements as user stories that represent all features that the application provides. They are separated into 3 categories: MUST HAVE, SHOULD HAVE and COULD HAVE. All MUST HAVE features are critical part of proof of concept and must be implemented. The actual Product Backlog of this project is shown in the table below.

1	**Account and profile management**		**PRIORITY**
	1.1	As a user, I want to be able to create an account so that I can use the system	MUST HAVE
	1.2	As a user, I want to be able to change my profile information so that my profile is updated	COULD HAVE
	1.3	As an administrator I want to be able to change a user role to teacher or student so that I can grant or remove teacher rights to a user	MUST HAVE
	1.4	As a user, I want to be able to delete my account so that my personal data and videos are deleted	COULD HAVE
2	**Login / Logout**		**PRIORITY**
	2.1	As a user, I want to be able to log in so that I have access to my account	MUST HAVE
	2.2	As a user, I want to be able to log out from my account so that nobody can access it	SHOULD HAVE
	2.3	As a user, I want to be able to reset my password so that I can log in when I forget it	COULD HAVE
3	**Uploading videos**		**PRIORITY**
	3.1	As a teacher, I want to be able to upload a video to my profile so that I can share it	MUST HAVE
4	**Managing uploaded videos**		**PRIORITY**



	4.1	As a video owner, I want to be able to change video attributes so that the attributes are updated	SHOULD HAVE
	4.2	As a video owner, I want to be able to delete a video so that nobody can access it	COULD HAVE
5	**Watching videos**		**PRIORITY**
	5.1	As a user, I want to be able to watch video	MUST HAVE
	5.2	As a user, I want to be able to interact with the video so that I can pause it, move to a specific timestamp, or change the volume	MUST HAVE
6	**Searching for videos**		**PRIORITY**
	6.1	As a user, I want to be able to search for a video by search term so that I can find it	MUST HAVE
	6.2	As a user, I want to be able to use advanced search so that I can specify multiple searching criteria	SHOULD HAVE
7	**Feedback on videos**		**PRIORITY**
	7.1	As a user, I want to be able to like/dislike the video so that I can share my reaction	SHOULD HAVE
	7.2	As a user, I want to be able to edit like/dislike in the video so that it is updated	COULD HAVE
	7.3	As a user, I want to be able to comment on the video so that I can share my opinion	COULD HAVE
	7.4	As a user, I want to be able to delete my comment so that nobody can see it	COULD HAVE
	7.5	As an owner of the video, I want to be able to delete the comments of my video so that nobody can see them anymore	COULD HAVE
8	**Groups**		**PRIORITY**
	8.1	As a teacher, I want to be able to create a group so that I can share videos between users	SHOULD HAVE



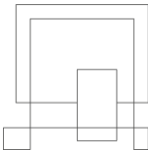
	8.2	As a group owner or teacher, I want to be able to add users so that they will be notified when I share a video to the group	SHOULD HAVE
	8.3	As a group owner or teacher, I want to be able to update group roles of members so that students gain instructor role or instructors are degraded to student	COULD HAVE
	8.4	As a group owner, I want to be able to delete a group so that all videos are not shared anymore within the group	COULD HAVE
	8.5	As a teacher or instructor I want to be able to share the video so that members of the group are notified	SHOULD HAVE
	8.6	As a group owner or teacher I want to be able to remove a user from the group so that they are no longer group members	COULD HAVE
	8.7	As a group member, I want to be able to see the list of videos shared in the group so that I can watch them.	SHOULD HAVE
	8.8	As a video owner, I want to be able to stop sharing the video in the group so it is no longer visible in the group	COULD HAVE
9	**Recommending videos**		**PRIORITY**
	9.1	As a user, I want to get the recommendation of videos based on my behavior so that I can watch a video that fits my preferences	MUST HAVE
	9.2	As a user, I want to be able to edit my personal preferences so that system can make a better recommendation for me	MUST HAVE

Table 1 - Product Backlog

2.2 Domain Model

The Domain Model is used to define all essential entities in the system and indicate the ways how they are connected with each other. In the case of this project, the Domain Model diagram shown below (Figure 1) can be divided into 5 main parts:

1. **User** – there are 3 main types of users:
    a. **Student** – a basic user that can search and watch videos



   b. **Teacher** – a user with an additional right to upload video and create groups
   c. **Instructor** – a special kind of Student that has a right to upload video in a particular group where this user acquired the Instructor role

   The Admin is the superuser that manages all the users.
2. **Group** – is used by Teachers to gather specific students into one common place to reach a selected audience while sharing videos
3. **Feedback** – once a user has watched the video, it is possible to give feedback as:
   a. **Rate** – user can like or dislike
   b. **Comment** – user can leave positive or negative feedback

   The purpose of Feedback is not only to provide the possibility to share user's reactions on video about its content and quality, but it is one of the key entities used for personalized recommendations.

4. **Recommendation** – consists of the following entities:
   a. **Preferences** – each user can set their own preferences that are to be used when generating the personalized recommendations
   b. **Behavior** – is used to gather user actions like searching and watching a video or leaving feedback.
   c. **RecommendationEngine** – is analyzing the data obtained from Preferences and Behaviour and eventually generates a VideoRecommendation
   d. **VideoRecommendation** – is the list of videos that are recommended to the user based on their Preferences and Behaviour in the system.

5. **Video** – is the core of this system as it is used and referenced by most of the entities above.

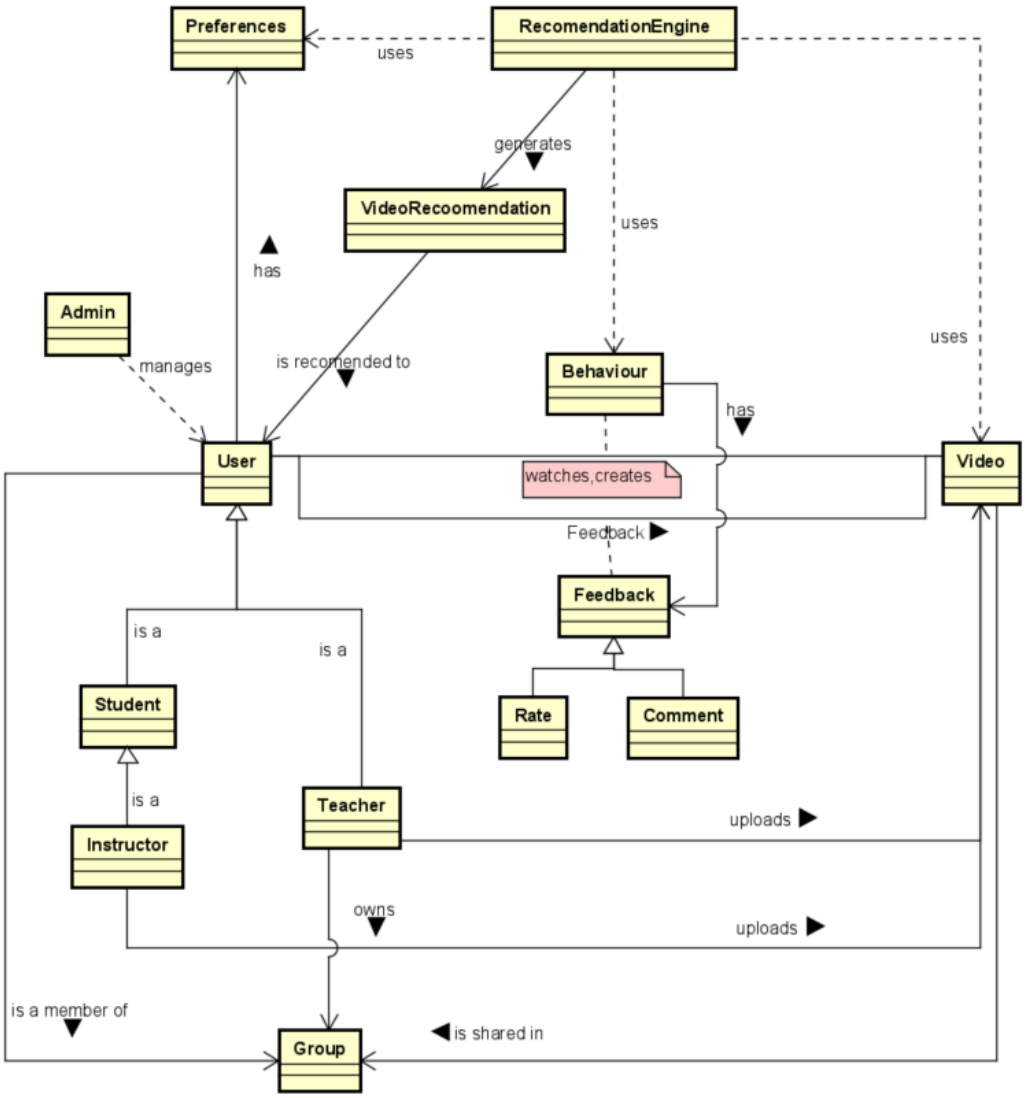
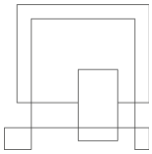


Figure 1 - Domain Model



2.3 Use Case Diagram

The Use Case Diagram is the second essential delivery in the Analysis phase. In the diagram below (Figure 2), there are defined all actors who can interact with the system and corresponding use cases that a particular actor can perform. These use cases are made from the functional requirements (see section 2.1.2 - Functional requirements – user stories) and each of them contains its Use Case Description.

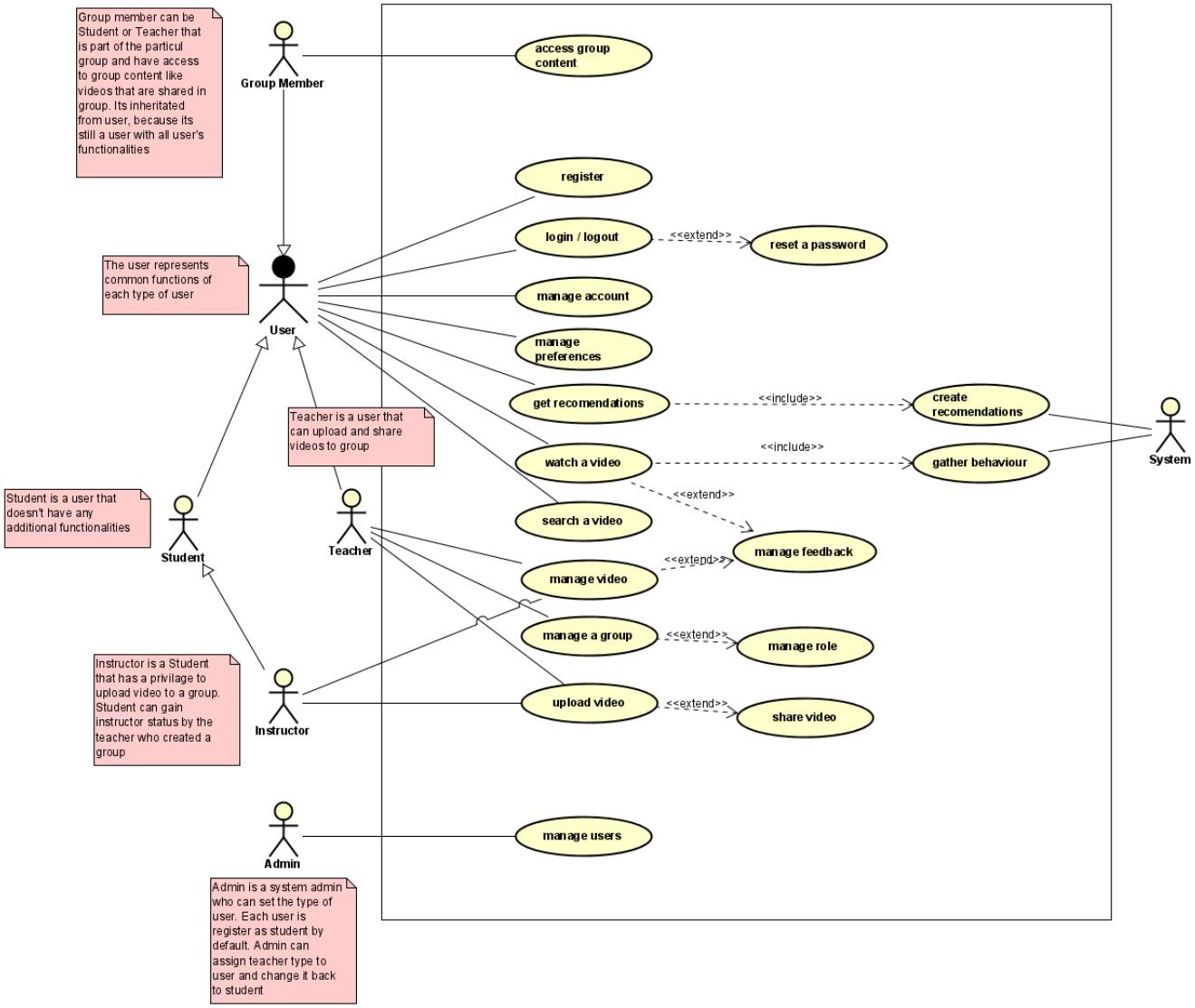
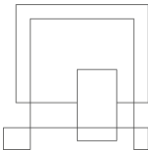


Figure 2 - Use Case Diagram

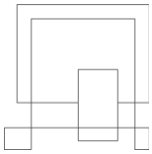


User and types of user

- **The user** is an abstract user that is representing all functionalities that are shared between several types of users. **The student** is a user that does not have any additional functionalities. **The teacher** is a user that can upload video, create a group and share videos. When a teacher creates a group and adds students or other teachers to it, those users become **group members** that have access to group content. Teacher can also manage roles in the group and can assign an **Instructor** role to a student, so they can also upload videos to the group. In fact, the instructor is a student with additional capabilities.
- **Admin** can change a type of user. By default, each user is a Student when they register. Admin can assign to them the Teacher role or degrade Teacher to Student.
- **System** role is important for recommendation feature, it has two important functionalities: gather user's behaviour and prepare recommendations for a user based on it.

This separation of actors is chosen to make the diagram easier to read by clearly showing which kind of actor can do which task. In addition to the actors, the Use Case Diagram contains all 18 use cases of the system. Each use case has a use case description that helps the reader to understand how the specific use case works in the application. The base sequence is describing the sequence of actions that are taken in a specific feature.

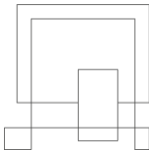
For example, the use case description of the upload video use case, that is shown in the diagram below (Figure 3) has a short summary of use case, actors that are able to perform this use case, and a postcondition that explains what has happened after this use case is completed. The Base Sequence is demonstrating a sunny scenario and the Branch Sequence shows an alternative sunny scenarios which are other possible actions that can be performed in a given use case. The Exception Sequence describes how the system responds when the user tries to make an incorrect action or an error happens.



ITEM	VALUE
UseCase	upload video
Summary	The teacher or instructor uploads a new video by selecting a video file with supported file format from the device file system and specifying information in mandatory and optional video attributes.
Actor	Teacher, Instructor
Precondition	
Postcondition	The video is saved with its attributes in the teacher's profile
Base Sequence	1. The teacher navigates to profile and clicks upload video button 2. The system displays upload video page 3. The teacher selects a file, enters title, study programs, tags and thumbnail 4. The teacher clicks upload button 5. The system checks video format and attributes 6. The system stores video to database, and shows the uploading progress bar. When the video is stored, the system sends notification with a message 'Your video was uploaded successfully'
Branch Sequence	3. The teacher selects a file, chooses video thumbnail, enters title, description, study programs, tags, language and group access
Exception Sequence	3.1 IF teacher selects a video file with unsupported format THEN 3.2 The system displays an error message 'Video format is not supported' 3.3 IF teacher does not specify any tag THEN 3.4 The system displays an error message 'At least one tag must be specified' 3.5 IF teacher does not specify any program THEN 3.6 The system displays an error message 'At least one study program must be specified' 3.7 IF teacher does not specify any title THEN 3.8 The system displays an error message 'Video title must be specified' 3.9 IF teacher does not upload a video thumbnail THEN 3.10 The system displays an error message 'No thumbnail uploaded'
Sub UseCase	notification
Note	

Figure 3 - Example of Use Case Description - Upload video

All the sequences stated in the particular Use Case Description are to be used for the evaluation of User Acceptance Tests (see section 5.2 - User Acceptance Tests). Moreover, the full documentation of Use Cases and their descriptions can be found in the Appendix C.



2.4 Data schemas

In the scope of this project, modelling of data schema was divided into:

- schema for Business Entities - document based MongoDB database
- schema for Recommendations - Neo4j graph database

2.4.1 MongoDB – Business Entities

The base for MongoDB data schema was taken from Domain Model and Use Cases. The data schema for MongoDB presented below can be divided in two parts – one that is related to a User and one that is related to a Video (File Storage). Entities with green background are the actual collections with the attribute descriptions of their corresponding document. Entities with yellow background represent documents which are embedded into the main documents like User or Video.

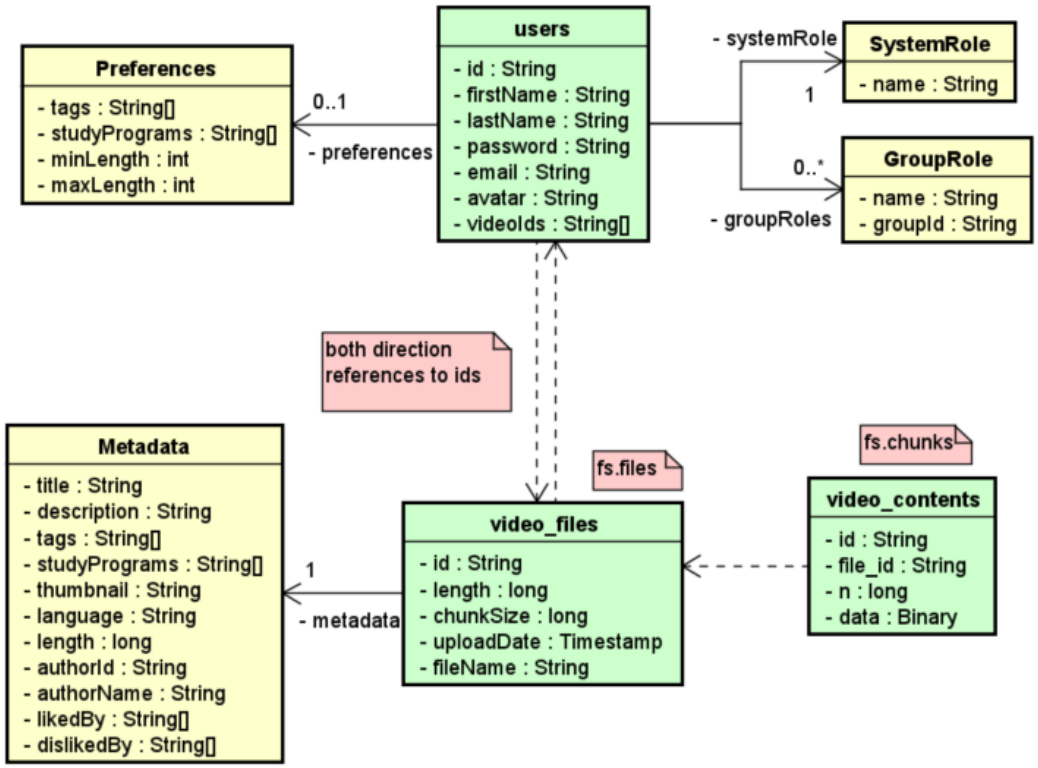
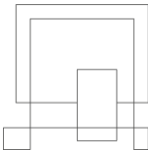


Figure 4 - Data Schema - document model

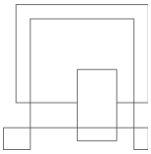


In the case of *users* collection, it holds documents that store the following data:

1. user's data relevant for:
    a. authentication – email and password attributes
    b. authorization – in form of embedded documents of type SystemRole and GroupRole
2. user's personal information – firstName, lastName and avatar attributes
3. user's preferences – consist of preferred user settings which can be of following categories:
    a. List of preferred tags - attribute in Preferences document
    b. List of preferred study programs - attribute in Preferences document
    c. Preferred interval of video length - attributes *minLength* and *maxLength* which define the interval

The second part of this data schema is concerning the storage of a video content and its metadata. There are two relevant collections in this section:

1. video_files – holds the following data about video file:
    a. id - id of video
    b. length – length of file
    c. chunkSize – number of chunks that the file is divided into
    d. uploadDate – actual date when video was uploaded
    e. fileName – name of uploaded file
    f. metadata – user inserted data about video in form of embedded object of type Metadata that consists of attributes as follows:
        i. title
        ii. description
        iii. tags – list of assigned tags to the uploaded video
        iv. studyPrograms – list of assigned studyPrograms to the uploaded video
        v. thumbnail – image representation of video that is encoded into Base64 String
        vi. language – defined language of video
        vii. length – length of video in milliseconds



viii. authorId – id of a user that uploaded a video
ix. authorName – first and last name of user that uploaded a video
x. likedBy - ids of users that liked the video
xi. dislikedBy - ids of users that disliked the video

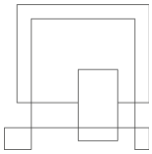
2. video_contents – stores video content in the form of chunks of predefined chunkSize. The attributes are the following:
    a. id – id of a chunk
    b. file_id – id of a file that the chunk belongs to. It is a reference to the id attribute in video_files collection.
    c. n – sequence number of a chunk for a specific video file
    d. data – actual binary data that is stored inside the specific chunk

The connection between these two parts of MongoDB data schema is done with both direction references between *users* and *video_files* collections. On the one hand, each document in the *video_files* collection stores authorId as well as likedBy and dislikedBy in the Metadata embedded object, which is a reference to the id attribute in *users* collection. On the other hand, documents in the *users* collection have videoIds attribute which is an array that stores ids of videos that the user uploaded and they are referenced to the id attribute in *video_files* collection.

It is important to notice that there is one more connection between users and video_files and that is the name of the user that can be updated in the system and therefore this attribute has to be synchronized always when a user changes their firstName or lastName. There are the following advantages and disadvantages of this decision:

Advantages:

- faster read operations when querying videos, because otherwise the name of user would need to be additionally requested from *users* collection
- better fitting model for making full-text search since all necessary indexes can be found in one embedded document – Metadata



Disadvantage:

- synchronization of data when user changes their name
- slower write operation as it needs to update metadata of all videos that a user uploaded to the system

Bearing in mind that it is important to have fast search operations, it was decided to store user's name inside video metadata even though it slows down write operation.

2.4.2 Neo4j – Recommendations

In graph databases, business entities are represented as nodes (bubbles) which can hold multiple attributes that are used to store details about the specific node. The nodes are distinguished by their labels, in simple words by their type. In this sample diagram shown below (Figure 5), different labels are represented by different colors. What makes graph databases different from other database paradigms are the connections between nodes. They are represented as edges which join nodes together and establish relationships between them. All these form a network of connected nodes which is often a case in the real world scenarios and that is why graph databases are often used to reflect such a kind of networks.
Moreover, since edges can hold properties in the same way as nodes, it is possible to create weighted models in graph databases. It can be done by using an attribute of edges for storing the distance or degree of similarity between nodes. (What is a Graph Database, 2020)

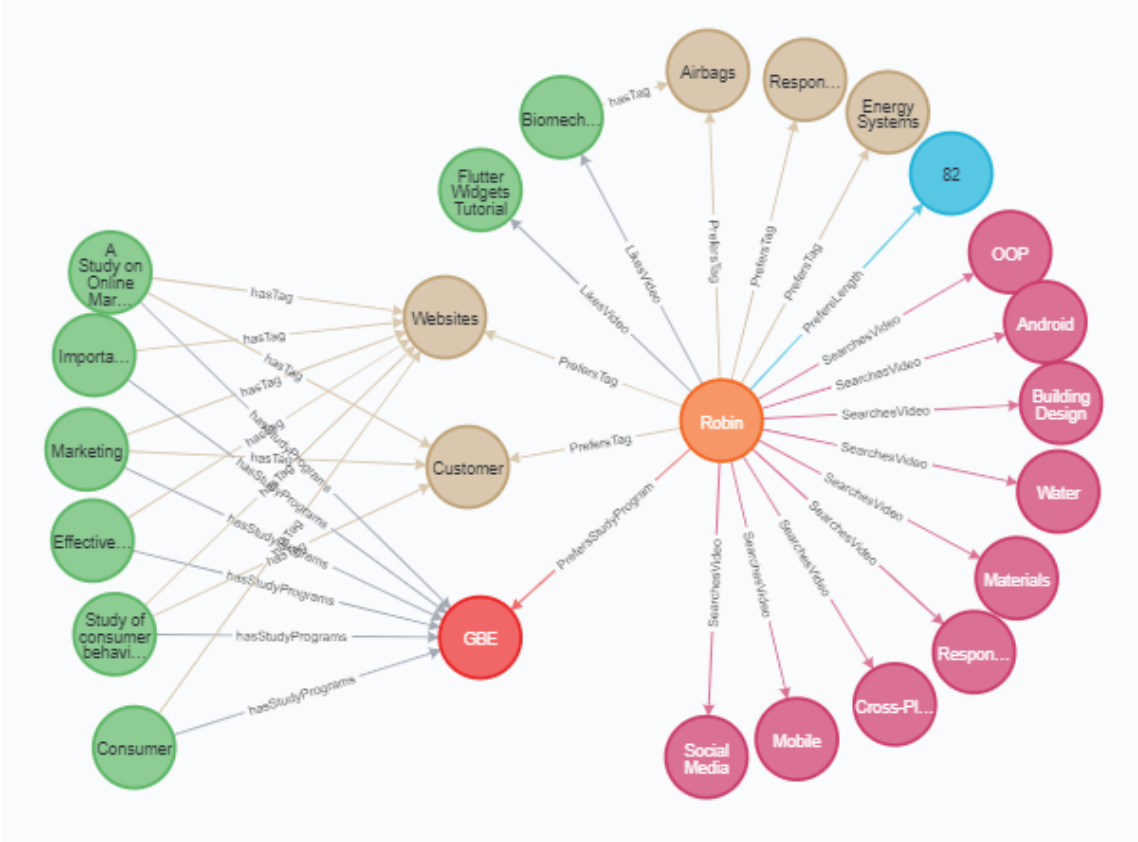


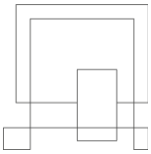
Figure 5 - Example of graph model

In case of creating schema for recommendations, the graph database offers needed functionalities for this project, which is mostly the need to:

1. store history data about different actions that a user performs in the system
2. store relationships between related entities or entities that have similar attributes
3. evaluate relationships by the degree of similarity between two entities - nodes

The sample model shown above (Figure 5) can be used as a demonstration or proof of concept that a graph database solution can be used in this project for recommendations.

Prior to actual data modelling, it was important to find out how to investigate which user actions are important to users in order to be able to assign adequate weight to each user action. To find that out, a short survey was created where potential users of the system responded to a couple of questions that helped us to discover based on what attributes



they are searching for videos, what actions they take after they watch a video or if they have some requirements for a video that they are going to watch.
Below there are shown some of the survey questions (Figure 6), however all the results from the survey can be found in Appendix D.

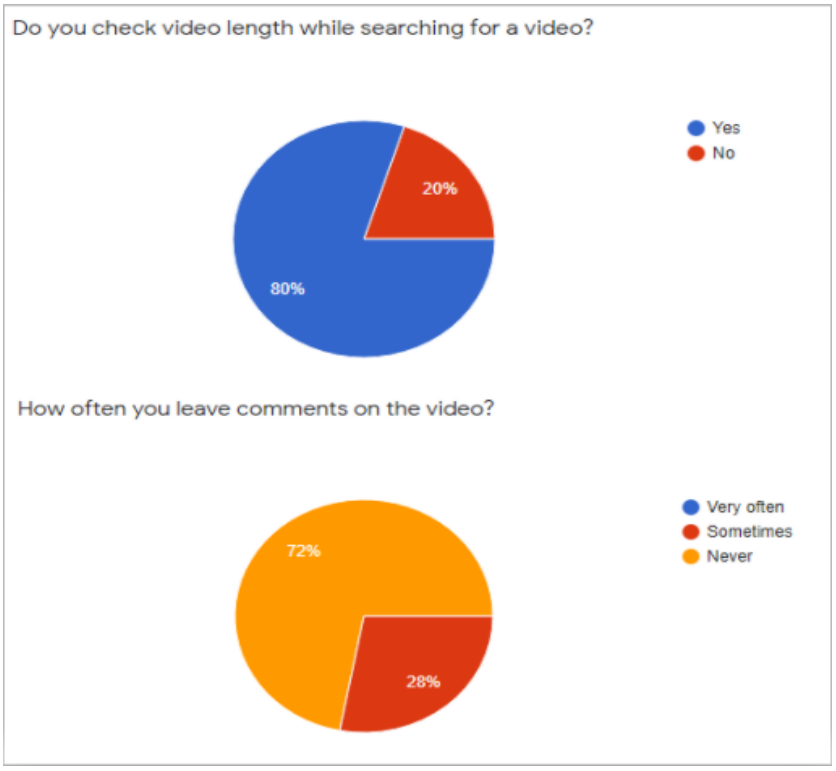
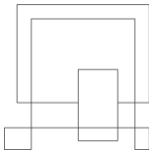


Figure 6 - Examples of charts generated from survey results

From the result of the survey, it is visible that for many of them it is very important the length of the video, that is why it was decided to prioritize this attribute and the user will be able to set an interval length that they like. Then there was a question if the author of the video is important, and as 40% responded no, it was decided that videos will not be recommended based on the author as people are mostly putting attention to the content of a video. Also, 96% were choosing the 'Title' as a preferable option for picking a video that strongly points out that the content is more important. In the case of commenting and rating videos, many people are usually not leaving comments, therefore we took a decision that this action should be considered with a high priority as it happens rarely.



This survey definitely helped to prioritize user actions and construct a recommendation algorithm, that is using user preferences and their actions to select the best matching videos. In the following table is listed what attributes the user can set and what user actions are part of the algorithm with its priority number.

User Preferences	
Action	Priority
Study programs	25
Tags	15
Interval length	20
User actions	
Action	Priority
Searching video	12
Watching video	12
Commenting video positively	18
Commenting video negatively	-18
Like video	6
Dislike video	-12

Table 2 - Recommendation engine attributes priority table

It is important to mention that every action has a date when it was created. This date is also very important for algorithm as it is deducting priority points based on how old a particular action is. In particular, each passed month sets a penalty of 1 to the initial priority defined by type of user action.

Bearing in mind all actions that need to be tracked and priorities they have, the following data schema was created for a graph database (Figure 7).

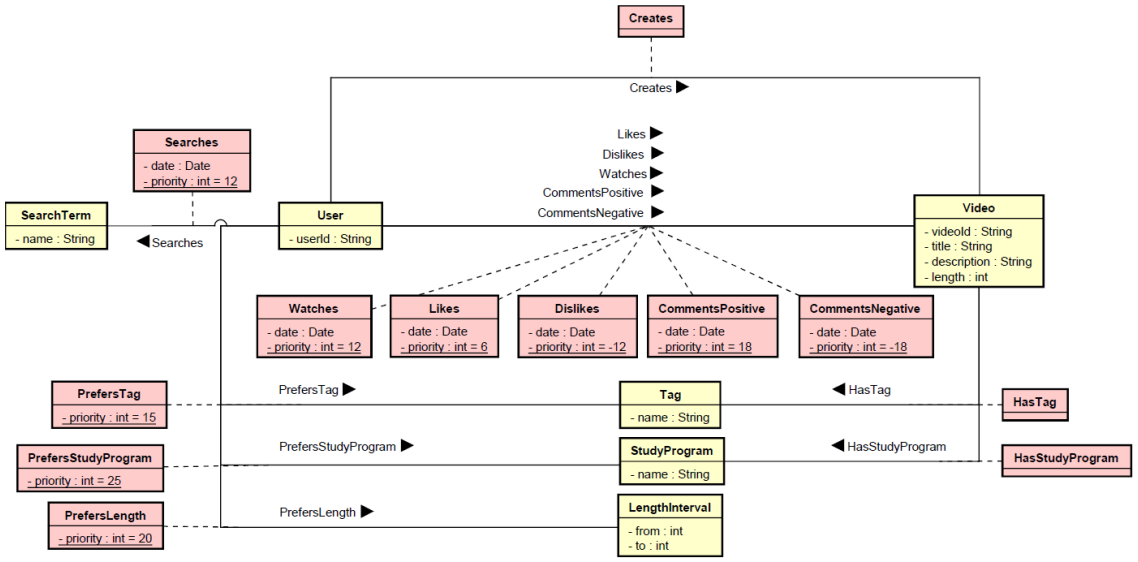


Figure 7 - Data Schema - graph model

The schema presented above, as the other graph database schemas, consists of several nodes and edges. In the diagram, there are in total 6 nodes (entities with yellow background) – User, Video, Tag, StudyProgram, LengthInterval and SearchTerm.

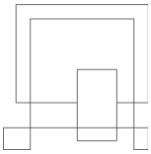
Aforementioned nodes are connected by edges (entities with red background) which can be separated in three categories – edges related to:

- User preferences - PrefersTag, PrefersStudyProgram and PrefersLength
- User behaviour (actions) - Creates, Searches, Watches, Likes, Dislikes, CommentsPositive and CommentsNegative
- Video metadata - HasTag and HasStudyProgram

Edges related to user preferences and behaviour have a *priority* attribute which is used for deciding which video better match.

Edges related to user behaviour hold important attribute *date* which is used as a penalty for priority in the way that action that happened 1 year ago does not have that big impact as the action that happened recently.

In the case of edges related to user preferences, their *priority* attribute is time independent so that there is no time penalty needed.



To sum up, based on relationships between nodes, it is possible to find best matches of videos for a given user. With a combination of user relationships to their preferred tags, study programs, interval and their behaviour, the final priority of a video is calculated.

2.4.2.1 Recommendations algorithm

In order to understand better the algorithm for recommendations, the Activity Diagram at the end of this section is provided (Figure 8).

The whole flow starts when a user requests to get recommended videos. The RecommendationEngine firstly checks if the user has created or watched some videos, because they are at the end used to filter the final video list so the user will not get recommended video which they created or have already watched.

Then it checks the user's preferences (Tags,StudyPrograms and LengthInterval) and finds videos that share at least one of the preferences which means that there is a connection between video recommendation candidate and nodes that reflect the user's preferences.

After that it finds videos which share at least one Tag or StudyProgram with one of videos that the user has either watched or liked or disliked or commented.

Then it comes to the search. The RecommendationEngine finds all searchTerms that the user has used and does a full-text search on tags, study programs and on videos which contain the specific searchTerm in their title or description. If tags or study programs match, the engine finds videos that share at least one of those tags or study programs. Otherwise, if a specific video matches, then no additional action is needed and can be directly added into the list of recommended videos from search action.

When videos are successfully collected from each of user actions and their preferences, they are filtered as already mentioned and eventually grouped by their ids and priorities of videos with the same ids are summed. At this moment, the final video list can be sorted by priority and responded back to the user.

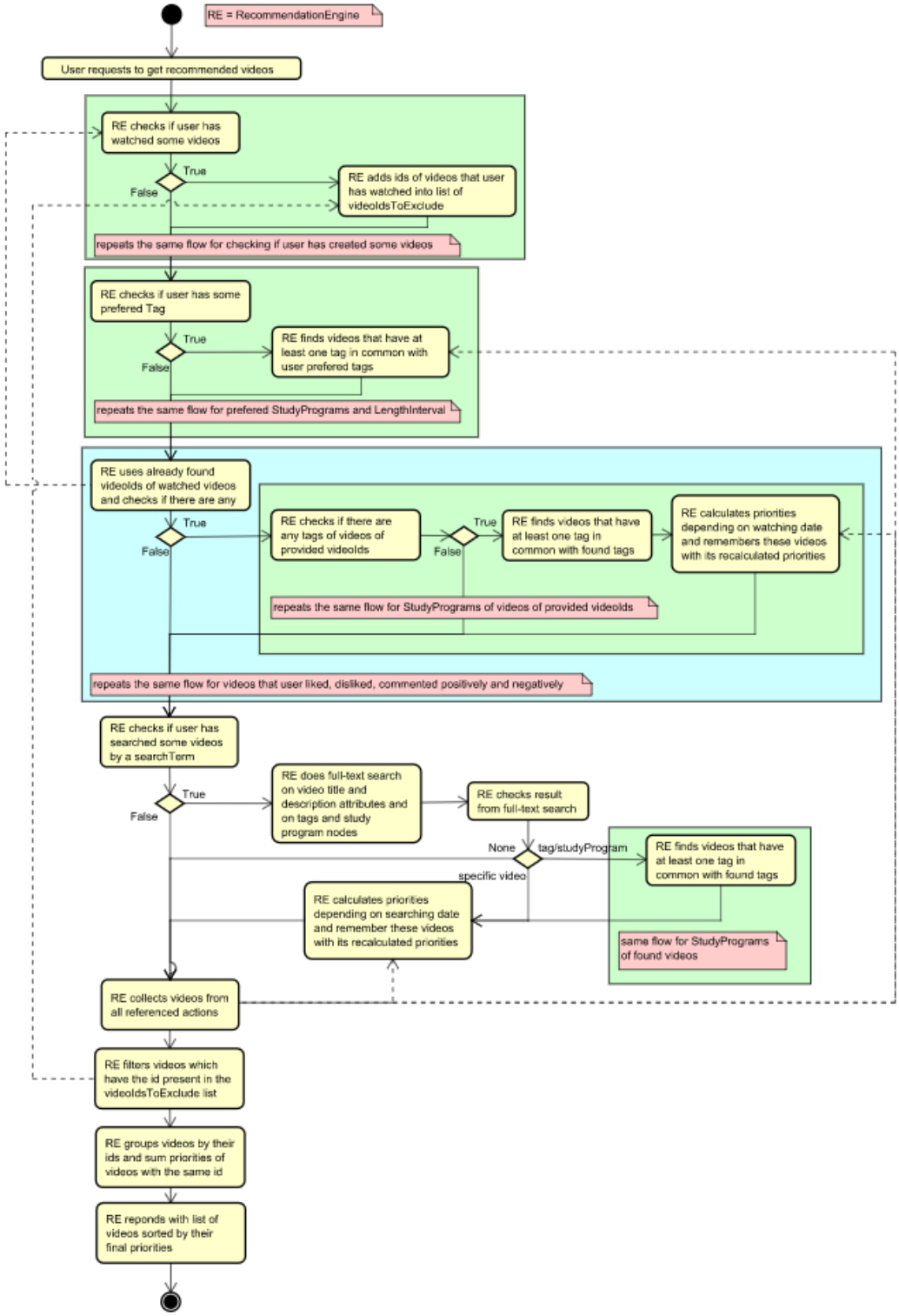
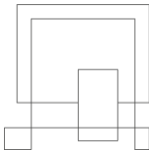


Figure 8 - Activity Diagram of recommendation algorithm



3 Design

3.1 Architecture

Based on the analysis, the design of the system can be made. In this part, the requirements are converted into the shape of the system. All the components are identified and communication between them is defined. This step is crucial, as good design and architecture are fundamental to the successful implementation.

To comply with the requirement of high availability and possible big amount of simultaneous users, microservices architecture has been chosen with a combination of 3-Tier architecture. The system is divided into 3 layers: UI, Service and Persistence. UI layer is responsible for interaction with users through website and android application GUI. The service layer handles business logic of the system and persistence layer is used to store data needed by the upper layers. The architecture of the system is visualized on the diagram below (Figure 9)

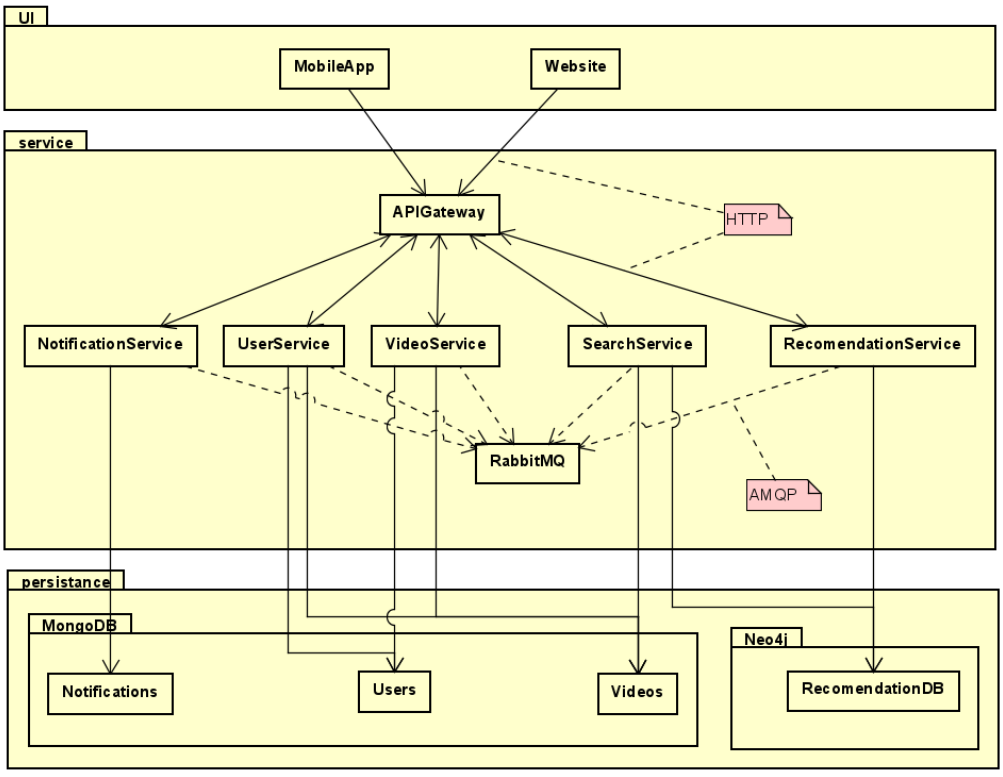


Figure 9 - Architecture Diagram

3.1.1 UI Layer

The UI layer is where the user interacts with the system. This can be done using two different applications, web application and Android application. They both provide user interface with the same functionality. Both applications communicate with the service layer to provide the data to be displayed to the user.

3.1.1.1 Frontend – Bloc Pattern

For the flutter application, the Bloc pattern was chosen as it is a recommended pattern for designing Flutter app architecture. This pattern allows us to easily separate the presentation layer from the business logic. The code is testable and reusable that makes implementation faster. The goal of this pattern is to define events and their states that force the programmer to cover all possible scenarios of the given event. Bloc attempts to make state changes predictable by regulating when a state change can occur and enforcing a single way to change state throughout an entire application. (Angelov, 2020)

That makes this pattern to be a good solution for state management even for the complex application by composing them of smaller components. In our project the components are functionalities of the application, that are separated into packages. Each functionality (package) has its own bloc, repository and view (Figure 10).

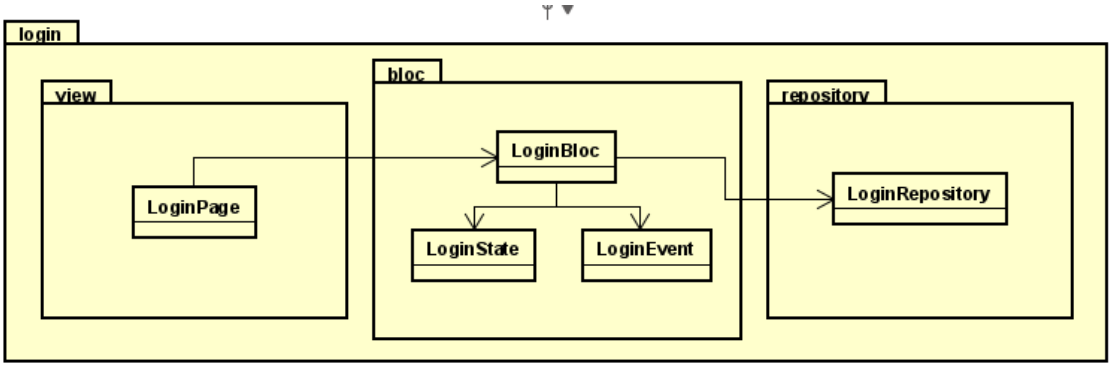
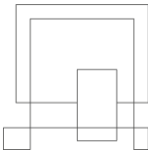


Figure 10 - Structure of Bloc Pattern



The LoginPage in the view package prepares the user interface using Flutter widgets. It also triggers events that are passed to the LoginBloc. The LoginBloc handles the events and maps them to states. To do that, it is using a LoginRepository that makes requests to the backend. LoginPage listens to LoginBloc for incoming states and updates the UI with new data.

3.1.1.2 Class Diagram

The structure of the frontend application is split into feature-specific packages. In the class diagram of feature-specific package (Figure 11), each package contains bloc, repository, view, and in some cases model packages. Inside of the common package are model classes and repositories that are shared by many features. For example, a client is initialized in its own Service package inside the common package. User repository is also in commons because it is needed by many features as they are using user data or displaying them in the application. There is also a shared widget, an Avatar, that predefines the view for the user image. There are also some features that do not have a typical bloc structure, that is home, logout, and video. All of those packages represent only a widget that is part of some feature or other widget. For example, Video prepares the video widget and initializes the video controller provider. It is then used in the My Videos feature or in the Search feature. Full version of the Frontend Class Diagram, shown in the Figure 12 can be found in Appendix C.

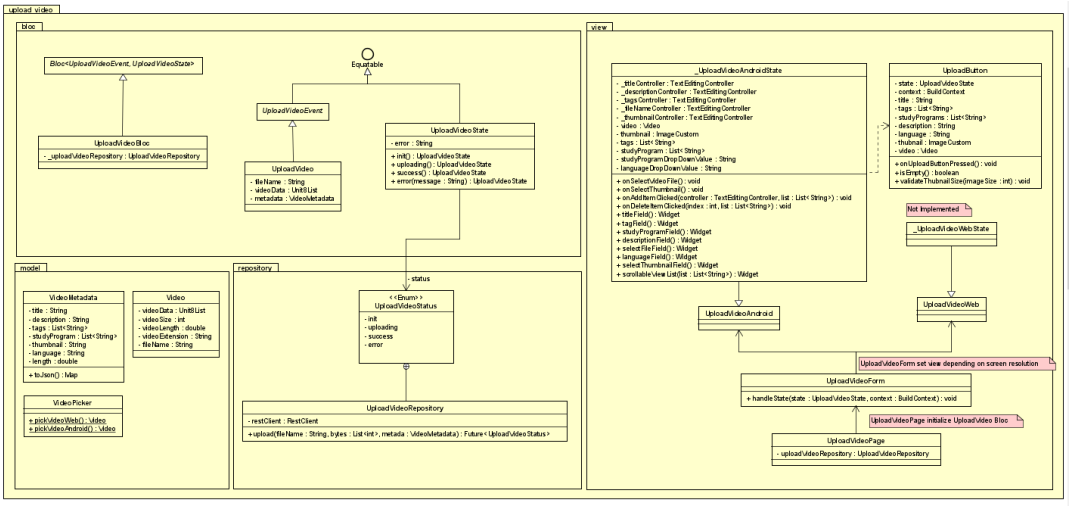


Figure 11 - Example of a feature package in Flutter application

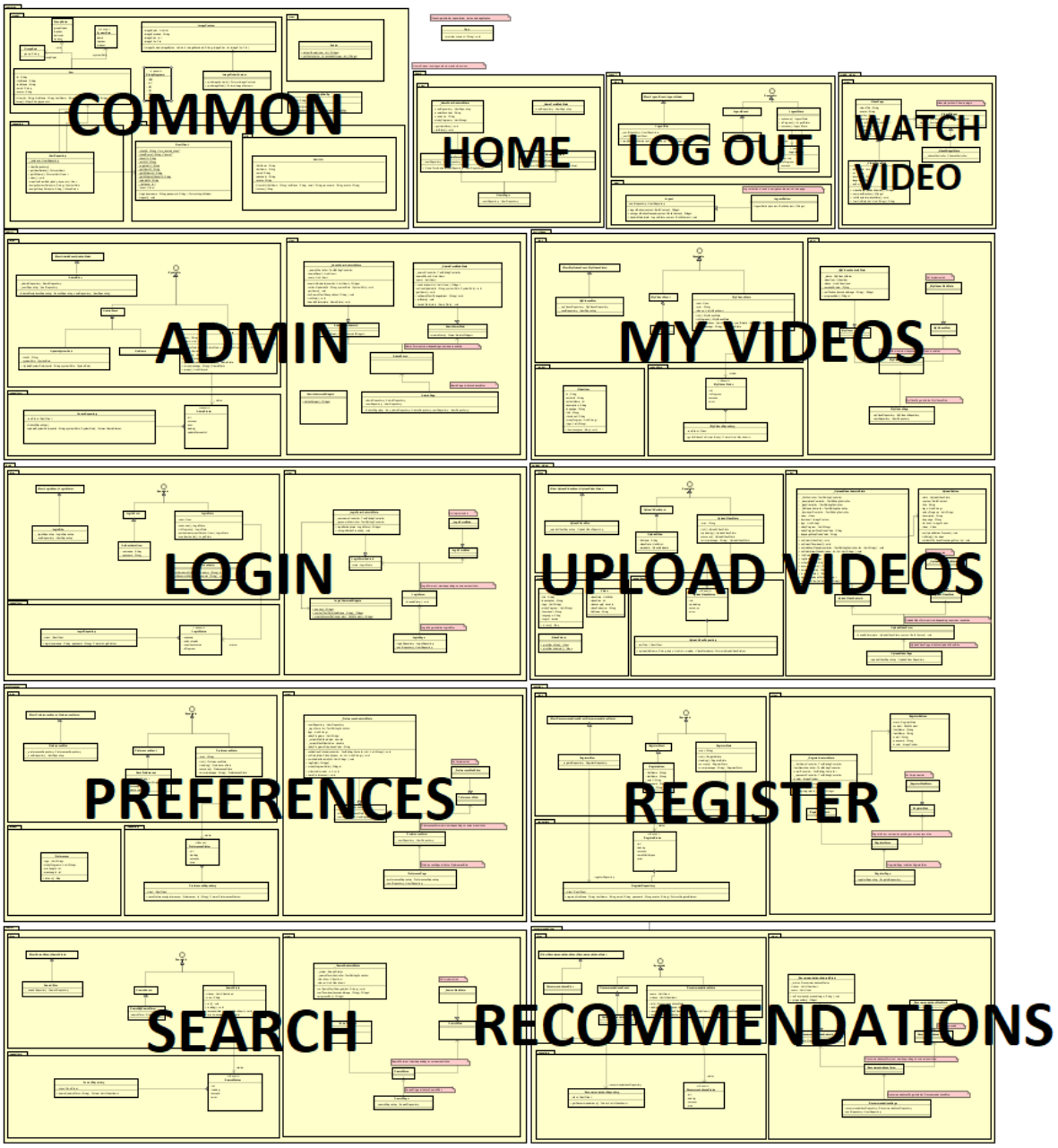
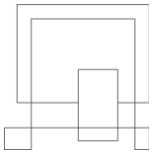


Figure 12 - Frontend Class Diagram



3.1.2 Service Layer

The system business logic layer is divided into small services (microservices) that are focusing on only specific parts of the system domain, so that the development can be more agile, as the coupling between the components is weak. There are following services in the system

- UserService handles authentication and authorization as well as managing the information about the users in the system, for example, system roles and group membership
- VideoService handles operations like uploading videos to the system and streaming the videos to the users.
- SearchService responsibility is to provide search results for the videos in the fastest possible way.
- RecommendationService purpose is to map the messages from other services to user behavior and generate personalized recommendations.
- NotificationService encapsulates the logic of sending push notifications to the users.
- The API Gateway is a component that is responsible for forwarding HTTP requests from the frontend to the backend services so that the system exposes a single point of entry. It does not contain any business logic and serves as an infrastructure component.

Even though SearchService and VideoService are using the same data in the database, they are separated to ensure the performance of those services and to be able to scale them separately as those two services might be the two most affected by the high number of users.

3.1.2.1 Class diagram

The class diagram shown below represents the structure of the service layer (Figure 13). It is formed by 6 separate class diagrams - one for each service described in the previous section and one called *commons*. The class diagram for NotificationService is not present in the diagram as it was a low priority task which was not possible to fit into the scope of this project.

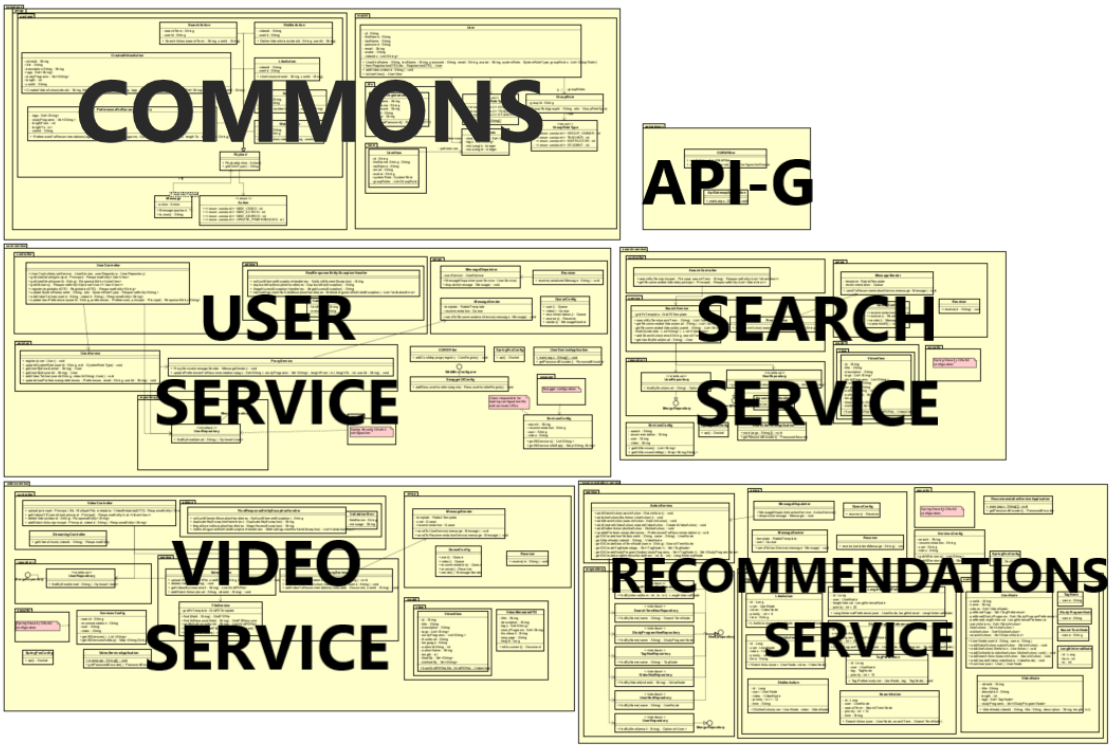
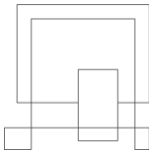


Figure 13 - Backend Class Diagram

The purpose of *commons* is that all services in the service layer need to have one and the same implementation of User related classes and also classes related to AMPQ (Advanced Message Queuing Protocol). Therefore, these classes were placed into *commons* module in order to avoid the duplication of them in each of the services. Further details about AMPQ can be found in the section 3.1.6.1 - Messaging design.



Below, there is a class diagram of VideoService (Figure 14), which is divided into the following main packages:

- *controller* - package for classes which provide required API endpoints relevant to the Video business entity such as *getVideosOfCurrentUser*, *upload* in VideoController class or *getVideoByChunks* in StreamingController class which is called for streaming a video content to the application.
- *service* - package which classes are responsible to transform requests from endpoints into requests to the persistence layer - retrieving, updating, storing or deleting data. In the case of VideoService there are 3 services inside service package:
    - VideoService which is a bridge between controllers and FileService
    - FileService uses GridFSTemplate to store and retrieve video content and its metadata
    - ProxyService which uses MessageSender class from *amqp* package to send asynchronous messages through message queues to other services. For instance, when a user creates a video, UserService and RecommendationService have to be notified about new video besides storing the video into MongoDB.
- *model* - package that holds DTOs (Data Transfer Objects) and business entities
- *repository* - package where are placed interfaces which are used to access persistence layer

Moreover, there are packages like *security* needed for configuration of Spring Security OAuth2 and *amqp* package needed for asynchronous communication between services.

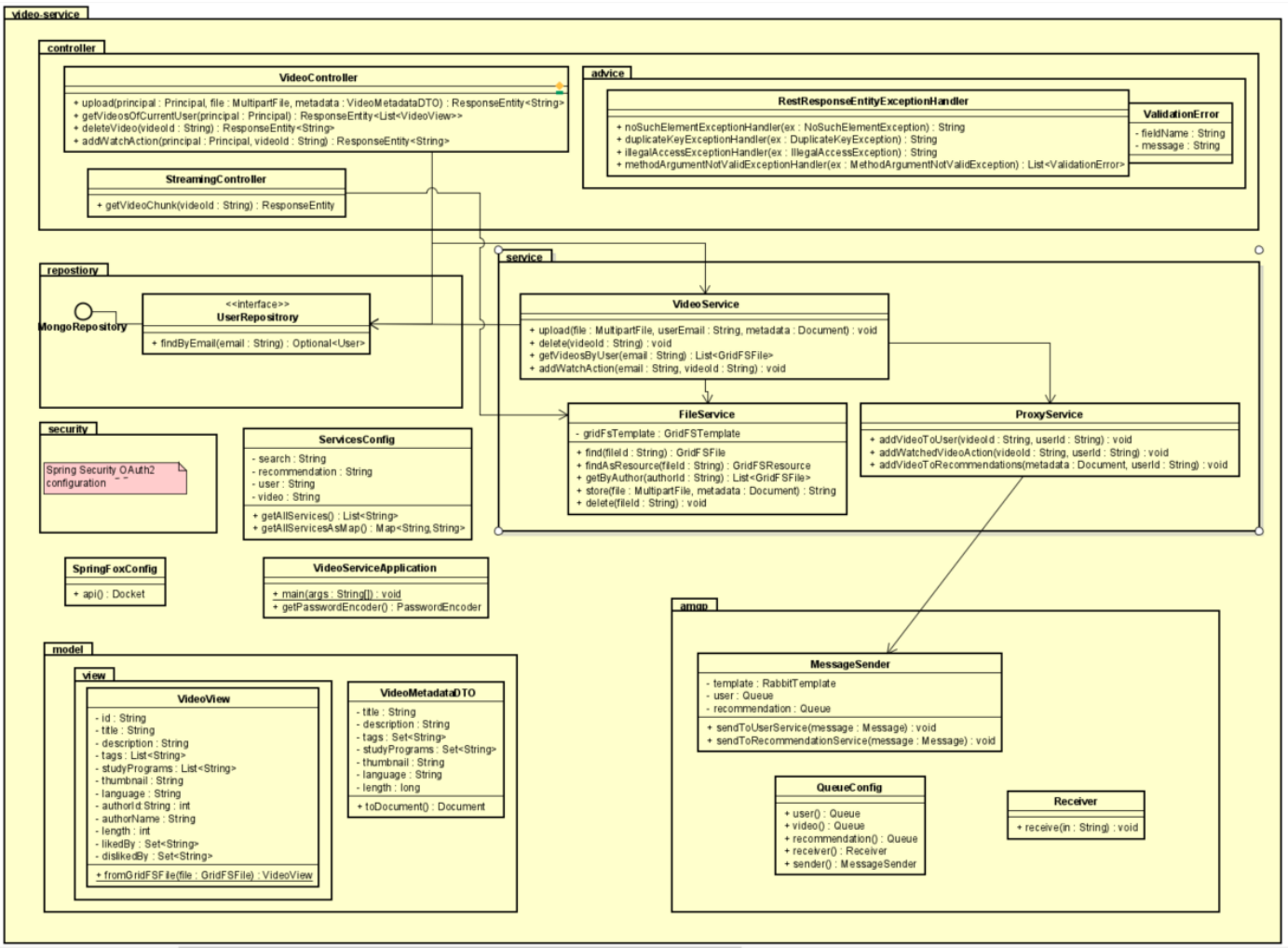
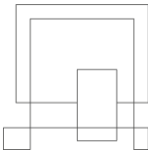


Figure 14 - Example of service structure - VideoService class diagram

The structure and packages presented in the VideoService are the same in the rest of the services (UserService,SearchService and RecommendationService) and for that reason they are not going to be shown in this document. However, to an interested reader all class diagrams can be found in Appendix C.



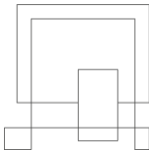
3.1.3 Persistence Layer

In the persistence layer there are two database types. First one, MongoDB handles data about users, groups and videos. Moreover, the actual video files are stored in MongoDB using the GridFS system, so that it is easy to extract a certain file chunk, without loading the whole file from the file system. The second database is Neo4J that is responsible for storing information about user's behaviour that are to be used by recommendation algorithm to generate recommendations.

The reason for having two database types in the system is to use the right tool for a problem. In order to achieve high availability of core system functions and fast response time for the most used queries, MongoDB was chosen as its object model allows for query optimised schema. Moreover, out of the box support for the replica set and clustering makes it a very good choice when availability is priority. (Introduction to MongoDB, 2020)
On the other hand, to implement recommendations algorithms, it is not required for the database to be fast and highly available. Much more important is the ability to represent recommendation schema, preferably using graph structure. Therefore, Neo4j, a graph database was chosen for that task. Even though it would be possible to implement this schema using MongoDB or some other SQL database, the task would have been harder than simply using Neo4j. As in this project, our main constraint was time, not resources, it was more efficient to use another database then try to fit the graph model into an object or relational model. (What is a Graph Database, 2020)

3.1.3.1 File storage

The storage of larger files was another important topic to cover in the design phase. Improper design could lead into various problems, such as file size limitation, compatible file types and efficient file retrieval. Moreover, it could cause obstacles related to the specific domain of this project which requires to store videos with its corresponding custom metadata.



For the reason of using a document-based database – MongoDB, it was decided to utilize and get advantage of MongoDB driver specification called GridFS which enables storing files larger than 16MB into MongoDB (GridFS - MongoDB Manual, 2020). The GridFS stores a file in two separate collections:

1. **fs.files** - collection used to store only file's metadata. Each document in the *fs.files* collection contains mandatory attributes like id, length, upload date, content type, etc.. In addition to the mandatory attributes, the optional *metadata* attribute can be used to store any additional domain specific information about the file.
2. **fs.chunks** - collection used to store file content by dividing the content into chunks of 255KB. This brings an advantage in case of file content retrieval as it prevents loading the whole file at once which could lead into memory issues. It does so by allowing to load only a specific part of the file – chunk. Each document in the *fs.chunks* collection contains chunk id, file id, the sequence number of the file's chunk and the payload of the chunk.

The following part of the Data Schema diagram shows how the aforementioned pattern is used in this project (Figure 15).

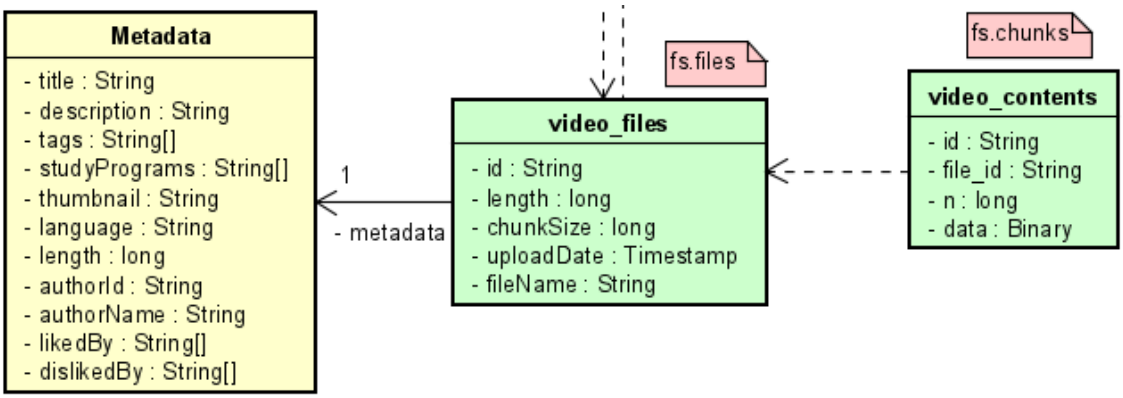
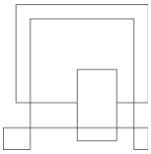


Figure 15 - Part of MongoDB data schema focused on video files

It can be observed that the *video_files* collection is the *fs.files* collection from GridFS and *video_contents* collection is the actual *fs.chunks* collection from GridFS. Moreover, each document in the *video_files* collection holds the embedded metadata document that is used as a storage for domain specific information about the file, such as title, description,



list of assigned tags, list of assigned study programs and the information about author (id and name) among other relevant attributes.

It is worth to mention two attributes that are important for effective file retrieval by chunks:

1. **chunkSize** attribute in documents from *video_files* collection. It represents the number of chunks into which the whole file was divided.
2. **n** attribute in documents from *video_contents* collection. It represents the sequence number of the file's chunk

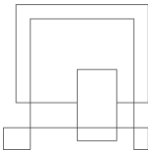
Using those two attributes, it is possible to easily retrieve a specific chunk of the file for streaming video to the client.

3.1.4 Architecture choice discussion

In comparison to monolithic architecture, microservices provide much better flexibility and lack of single point of failure, so that the application can continue to function when one of its components has failed. Moreover, microservices provide high agility of the development by clearly dividing the responsibilities in the system and introducing weak coupling between components that allows simultaneous development of features easily. However, most importantly, this architecture allows us to scale the individual components of the system as needed so that more hardware resources can be utilized more efficiently by deploying new instances of the needed services. That complies to the requirement of high availability of the system while used by a lot of users. The cost of using this approach is the high complexity of the implementation and technological challenge to incorporate technologies needed to successfully implement this architectural pattern. Moreover, the boundaries of the services must be defined carefully so that they are self-sufficient and can fulfil their responsibilities on their own.

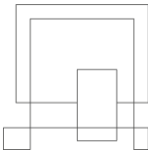
3.1.5 Cloud environment

As one of the requirements is that the system should be highly available, it was decided to deploy the system in the cloud environment. This solution removes the burden of maintaining hardware infrastructure from the project team to the cloud provider. It is then



the cloud provider's responsibility to fix a failed server or network connection, as their support team is specialized in that kind of problems.
Another benefit of using a cloud environment is the possibility to distribute the core components between multiple cloud servers by implementing microservices infrastructure. Thanks to that, the availability of the service can be increased, as in case when one of the servers fails, the system is able to function with limited functionality. Lastly, it is possible to spawn multiple instances of a particular server, depending on the current load of the application. Therefore, the system can be more responsive during heavy load, as the work is distributed amongst many instances. All aforementioned possibilities are much easier to implement using cloud services than hosting the servers on-premises and having to manage the infrastructure.

However, to make the system cloud-ready, necessary steps must be made. One of them was a requirement of carefully designed microservices architecture where dependencies between services cannot be very strong (that means that in case when one service fails, the others should be able to complete their tasks) and the communication between services must be asynchronous. Another requirement is that the services must be deployed to the cloud server without performing extensive manual tasks and configurations. In fact, the deployment process should be completely automated. To make that process easier, it was chosen to package the web application and web services into Docker containers, so that they can be run in any cloud environment without manual configuration of the actual servers. Moreover, this approach gives control over how the application is run and ensures that the runtime environment is the same, regardless of the used server. To make the deployment automation complete, TeamCity is used as CI/CD server. It listens for the commits to the project repository master branch and builds new docker images of all services. Next, thanks to integration with cloud provider, it pushes the docker images to the remote docker repository and deploys new services instances to the cloud. The details of this process are described in the section 4.7 - Cloud Deployment.



3.1.6 Communication

As the system has distributed architecture, components need to communicate with each other.
The frontend applications communicate with the backend using HTTP requests. To make the communication easier, API-Gateway has been established as a single point of entry to the system, so that frontend applications must know only its address. API-Gateway forwards requests from frontend to the backend services using HTTP requests.
In case of backend services, to make sure that the communication between them is decoupled in time and space, the communication is realized using AMQP protocol with RabbitMQ as a message broker. Thanks to that approach, the messages can be sent asynchronously utilizing the fact that the services can have multiple instances deployed at the same time and that some services might not be available at the moment of sending a message. On the contrary, using HTTP requests to communicate would require additional tools to discover the services in the cloud and ensure that the services are available. Moreover, a failsafe mechanism would have to be developed to guarantee message delivery when the message recipient is unavailable. All those problems can be solved by utilizing message queues provided by RabbitMQ. (RabbitMQ, 2020)

Communication between system components has been shown in the section 3.1 - Architecture on Figure 9.

3.1.6.1 Messaging design

As RabbitMQ provides only implementation of AMQP protocol, it is required to integrate the project structure with RabbitMQ API and provide its functionalities to the services classes. There are several components that need to be in place in each of the microservices, that is queue configuration, message receiver and message dispatcher. The Figure 16 demonstrates how RabbitMQ API was incorporated into the services.

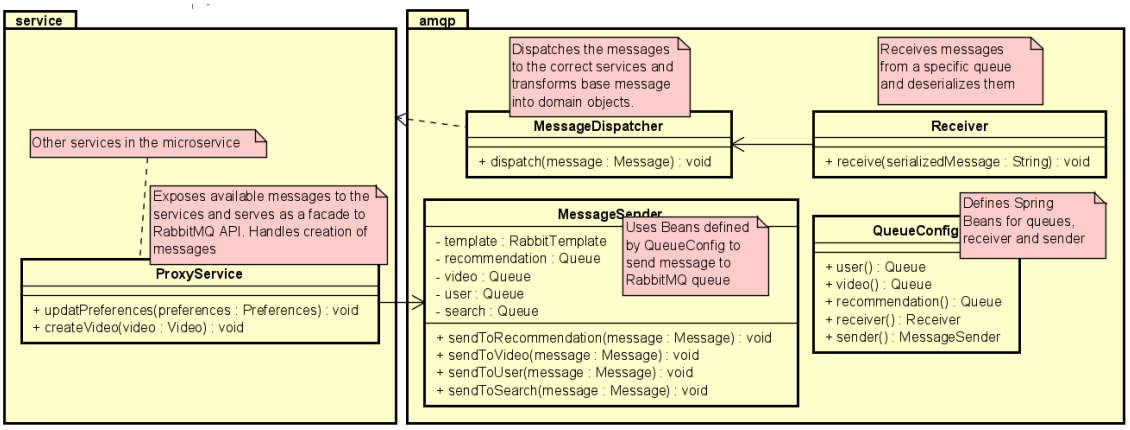


Figure 16 - Example of amqp package in backend services

To configure Spring Boot integration with RabbitMQ API, QueueConfig class defines the Spring Beans of type Queue that represents RabbitMQ queue and is used to send messages. Moreover, this class defines the sender bean as MessageSender class and receiver bean as Receiver class. In order to send a message to the queue, one of the services in the service package uses ProxyService and calls its methods. ProxyService is responsible for creating a message object from domain objects using classes defined in the *commons.amqp* package. When the message is created, it is passed to the MessageSender that serializes the message to JSON format and sends it to the appropriate queue. When a message is received from the queue, it first goes to the Receiver, which listens to all incoming messages and deserializes them from JSON format. Further, the message is passed to the MessageDispatcher where it is converted to the domain object and dispatched to the service that can handle the message.

Having considered the way of sending and receiving messages, the actual structure of the messages was designed. As during the project development, there might be a need for several new types of messages, their design must be easily extendable. Moreover, the recipient of the message must know what it contains so that the correct action can be made. To achieve it, a Message class was created that carries two things, action that the message represents, and the payload which serves as the data needed to complete the action. The action is represented by an Action enum that defines all possible actions. To create a payload, a new class that extends abstract Payload must be created with

fields for all required information and another action added to Action enum. Thanks to placing these classes in the *commons*, they are available for all other microservices, so that no code duplication is required and it is ensured that the same version of the classes are used everywhere. The structure of *commons.amqp* can be seen on the Figure 17 below.

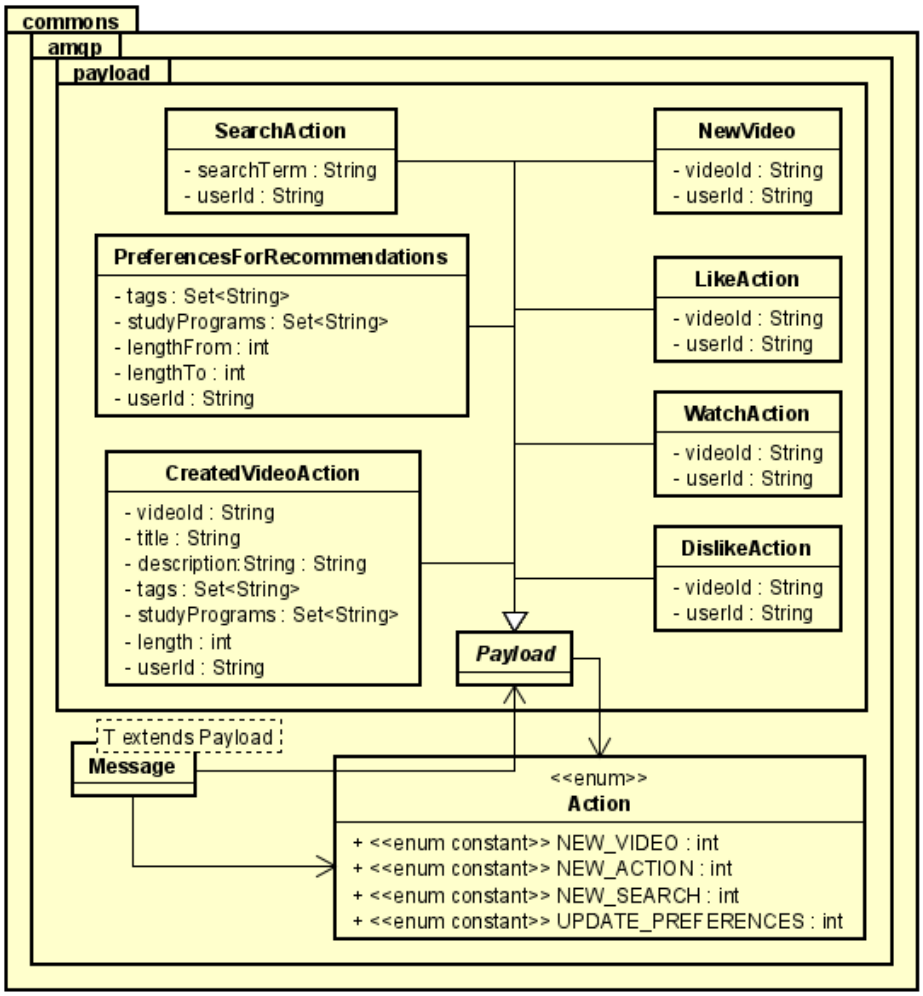
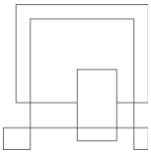


Figure 17 - Class diagram of amqp package in commons project



3.2 Technologies

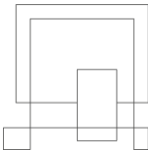
In this chapter, a discussion over choices of technologies is presented. Each subsection focuses on the specific technology and why it was chosen in comparison to others.

3.2.1 Frontend - Native vs Cross-Platform

As the system should support both website and mobile app, the first decision that had to be made was to choose between native or cross-platform approach. As the native approach gives more control over each component, it requires significantly more resources to develop an Android application and website simultaneously then creating one cross-platform app. On the other side, cross-platform solutions trade development speed and shared code base for better control over device and application. Some of the native features are yet not possible to reproduce in the cross-platform approach. Having considered those possibilities, it was decided to follow a cross-platform approach as the app is not going to benefit from native specific features (as sensors in case of mobile app). Most importantly, with given resources, developing one application instead of two is a much more feasible choice. (Klubnikin, 2017)

3.2.2 Cross-Platform App framework

To develop a Website and Android application, the Flutter framework has been chosen. As it is a stable framework developed by Google, its support, documentation, and amount of learning materials are on a very good level. Another considered choice was React-native as it also allows developers to develop cross-platform apps. However, this option has been rejected as this framework is not stable and its parts are changing quickly, possibly breaking the existing components. Moreover, it lacks good documentation of the API, which in the case of Flutter is extensive and up to date. Another reason is that React-native does not supply as many tools and components as Flutter, so that they would have to be created from scratch.



3.2.3 Backend language and framework

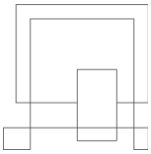
In the case of backend development, the Java language was chosen. It is well known by all group members and has proven as a viable choice over past semester projects. Spring Boot has been chosen as an application framework, as it provides support for all needed technologies (MongoDB and Neo4J data access, RabbitMQ, REST web services, OAuth2 authentication) and it was used by group members in the past. As the application should be deployed in the cloud environment, several other frameworks have been considered. As the main downside of the Spring is that it was not created to be used in the docker containers. Therefore, its startup time in the container environment is very big, even up to 30 seconds. Frameworks such as Micronout and Quarkus solve that problem, which makes them more suitable for the containerized environment. Nevertheless, the Spring Boot support for needed technologies and group member's experience with it was a stronger argument than what Micronout and Quarkus have to offer in terms of performance. Therefore, the Spring Boot framework has been chosen to develop microservices. (Graf, 2020)

3.2.4 Cloud provider

As a cloud provider, Google Cloud has been chosen, as it is the most known for the group members and its usage is not as complex as in the case of Azure. Moreover, Google Cloud provides 300$ credits for 1 year which is enough to develop the project without additional costs of the cloud hosting. This is not possible on Azure or AWS. Both Azure and AWS solutions were rejected due to the complexity of use and lack of experience with them as well as no viable option for no-cost development during project duration.

3.2.5 Databases

To store the data in the cloud, the choice was between PostgreSQL and MongoDB. MongoDB has been chosen as the main database solution. As PostgreSQL is a reliable



and stable solution, it does not provide the flexibility of MongoDB. The GridFS feature of MongoDB that allows to store whole video files in the database in a structured way and retrieve specific chunks of them easily is a great advantage. Moreover, to ensure the high availability of the database, MongoDB can be deployed in replica sets when needed, creating a database cluster where when one database instance fails, the cluster continues functioning and is able to serve data. As a similar result can be achieved with PostgreSQL, it is not as simple as in the case of MongoDB. Another advantage of MongoDB is the possibility to perform a full-text search over the document without changing the database schema, as in PostgreSQL. (Comparing MongoDB vs PostgreSQL, 2020)

The other database that is used in the system is Neo4j. It was chosen because it allows to represent data as a graph, which was needed to implement the recommendation algorithm. Moreover, it has API written in Java, and Spring Data has its own implementation of the Neo4J connector. Lastly, as the system must be running in the cloud, Neo4j fits into that requirement with a wide range of available Neo4j cloud instance providers. As the Neo4j fit perfectly to the needs of the system, other graph databases were not taken under consideration. (What is a Graph Database, 2020)

3.3 User Interface

The user interface should be kept simple and apply principles of the Material Design for the Android application. The diagram below (Figure 18) represents the navigation between separate views. If the user is already created, they can navigate to the Home Page that is different for Teacher and Student. Users of type Teacher can navigate to Upload Video Page and to My Video Page where is the list of their videos. Students can access this page only when a Teacher assigns them the role of Instructor. If the user is type of Administrator, it will navigate them to the Administrator page where is the list of all users and their can change the type of each user.

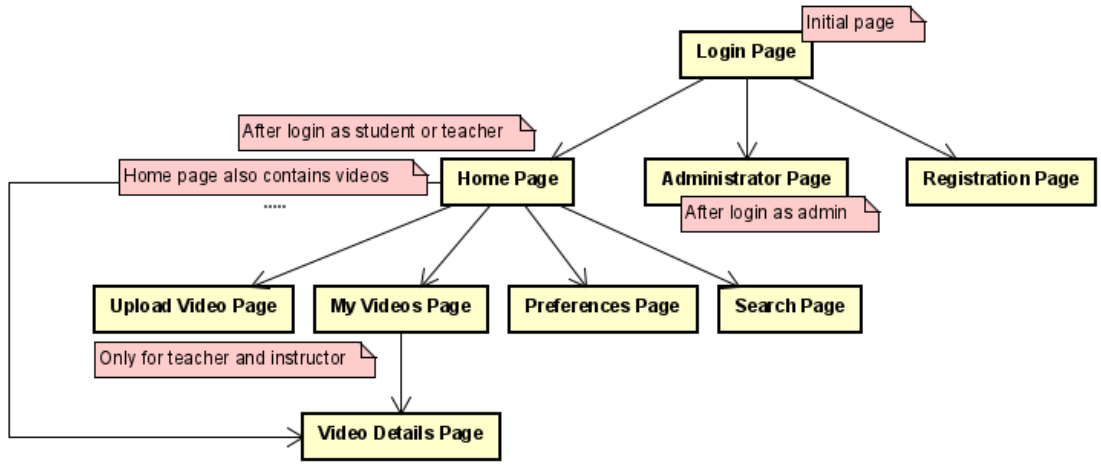


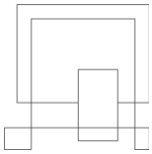
Figure 18 - Navigation tree

3.3.1 Android Application

The application uses material design with its Material icons. There were only three colors selected for application design: White, Material brown and Material GreyBlue

Figure 19 - Color schema

For Home Page and Video Details Page was created draft that represents widgets that will be part of view. (Preferences Page can be found in SRS – Appendix B).



3.3.1.1 Home Page

The home page uses a navigation drawer for navigation to another page or thumbnail of video to navigate to video details. First it is displaying the recommendations if the user has set their preferences, and if user searches videos it will replace the recommendation section to videos that are result of search.

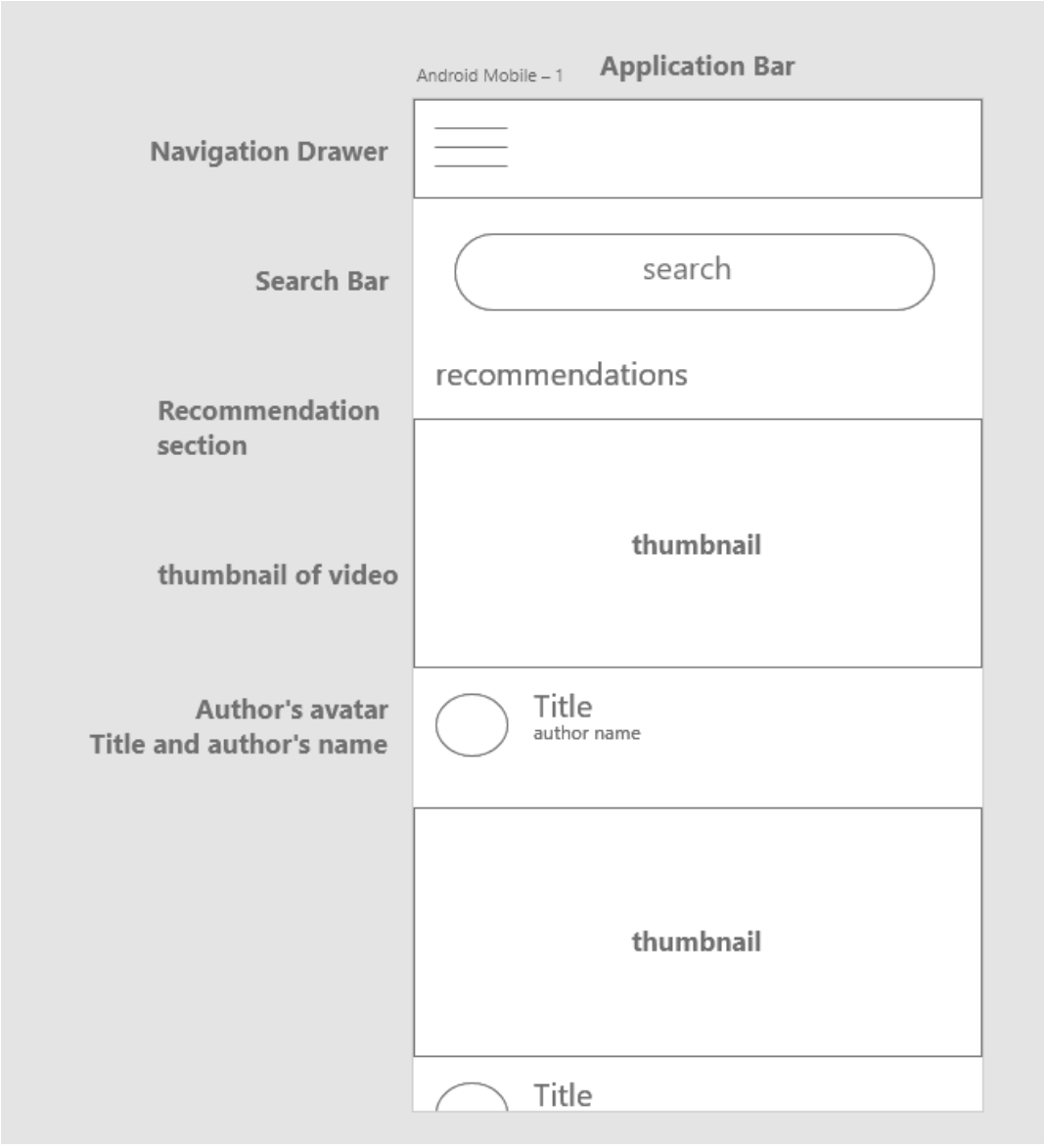
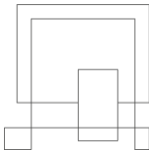


Figure 20 - Draft of application's Home Page



3.3.1.2 Video Details Page

The video details page has a video player that the user can maximize to full screen. The part under video has two states, the default one is showing information about like author's name, description, study programs, duration, tags and language. If user press the extendable arrow it will change this part to comment section where user can create a comment or leave like or dislike.

with video details

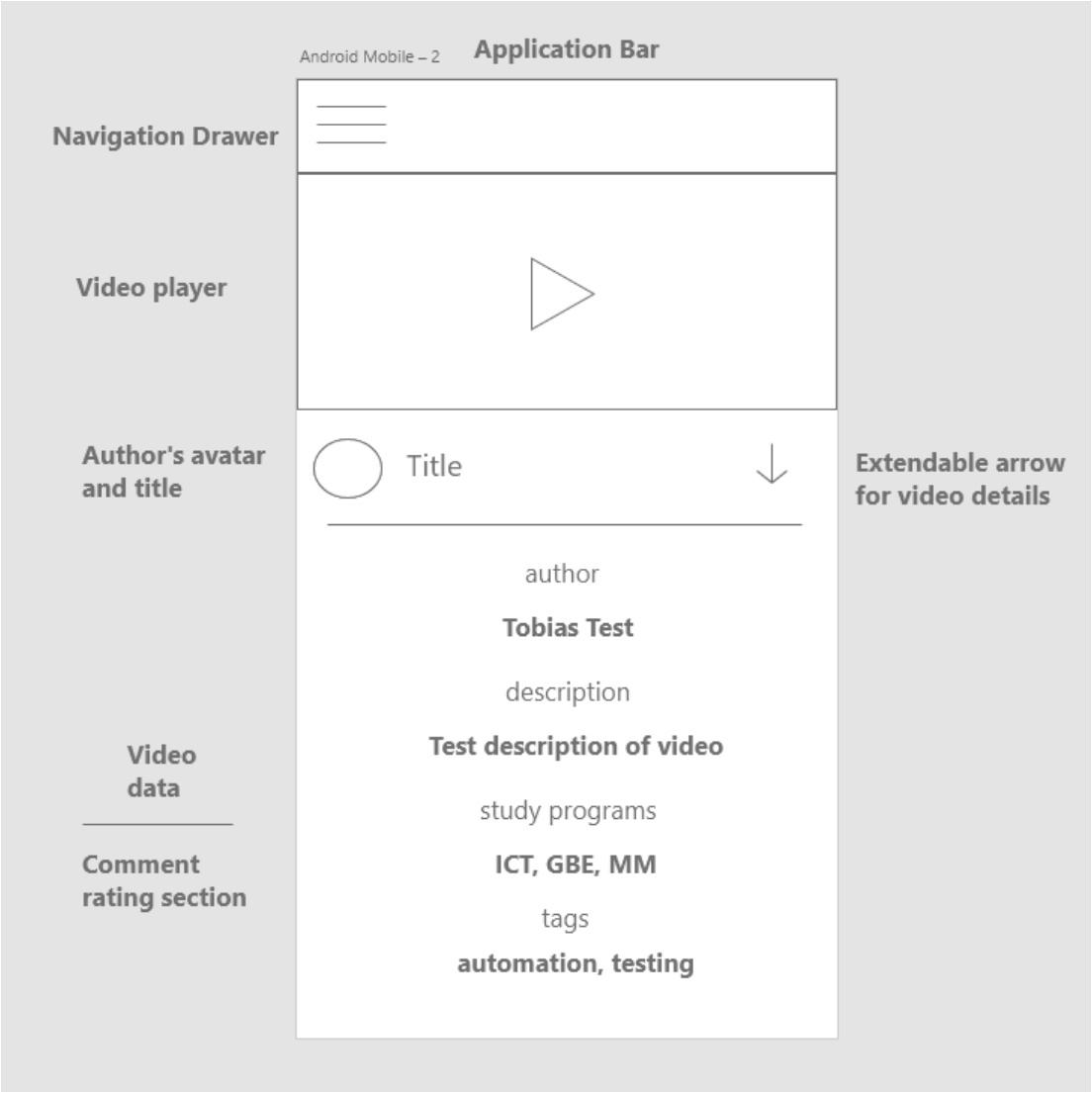


Figure 21 - Draft of application's Video Details page

with video comments section

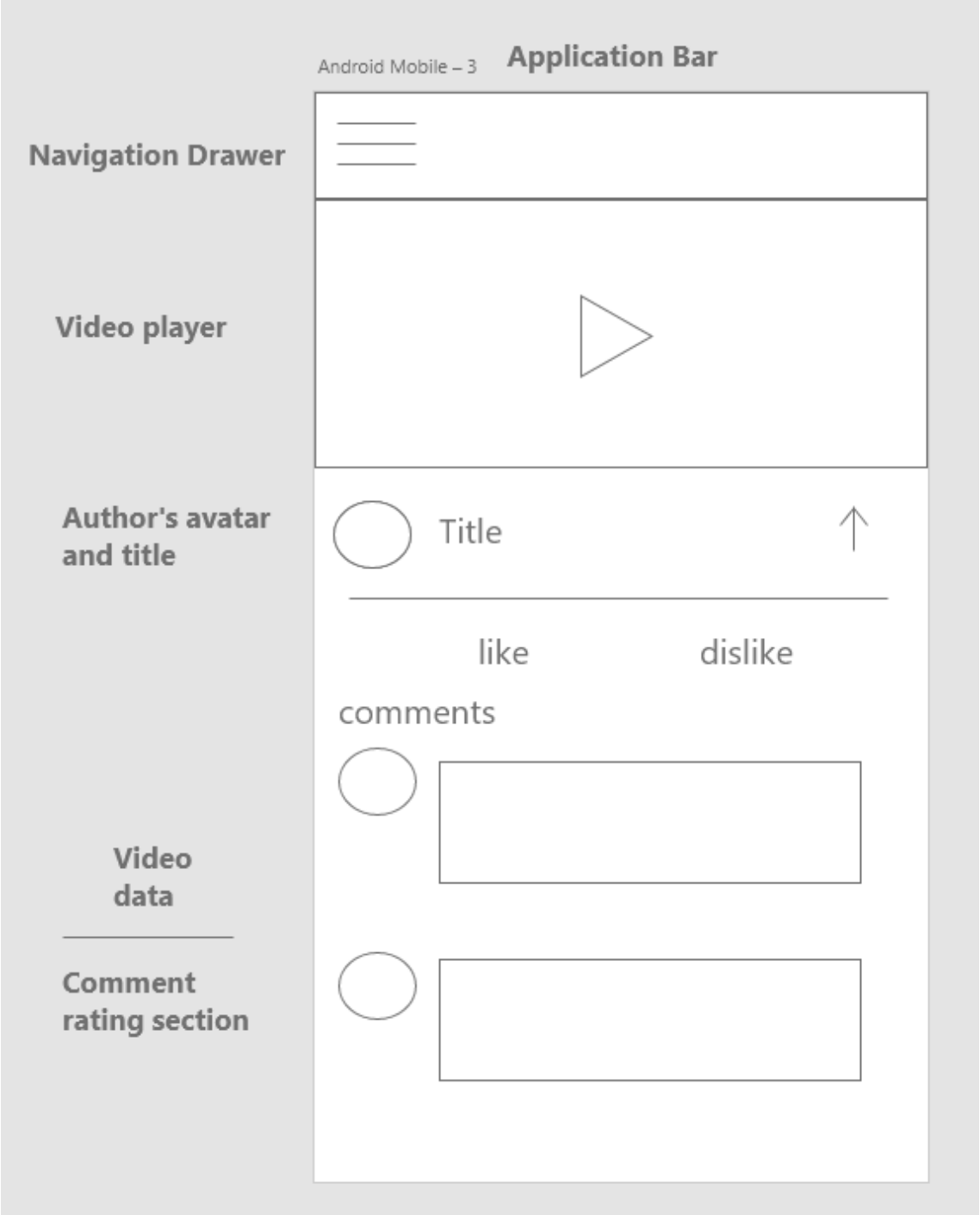
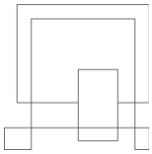


Figure 22 - Draft of Video Details page with comments section



3.3.1.3 Widgets

The upload video page is the most complex one, as it requires multiple widgets for fields like title, description, tags, selector for study program, language and file pickers for uploading thumbnail image and video. Each attribute has its own custom widget with its own text controller and data. Bellow diagram (Figure 24) represents a widget tree of upload video page:

Visualization of upload video widget

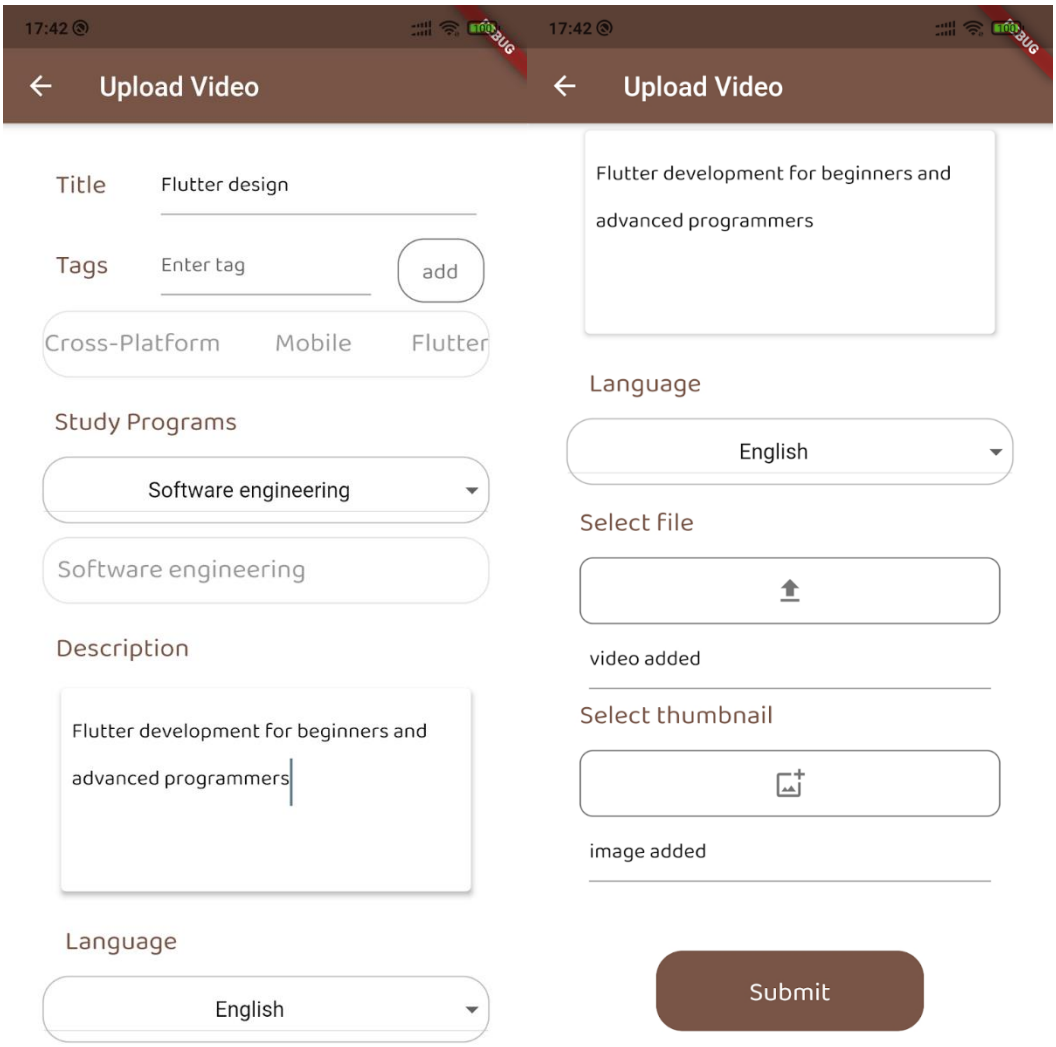


Figure 23 - Upload Video widget visualization

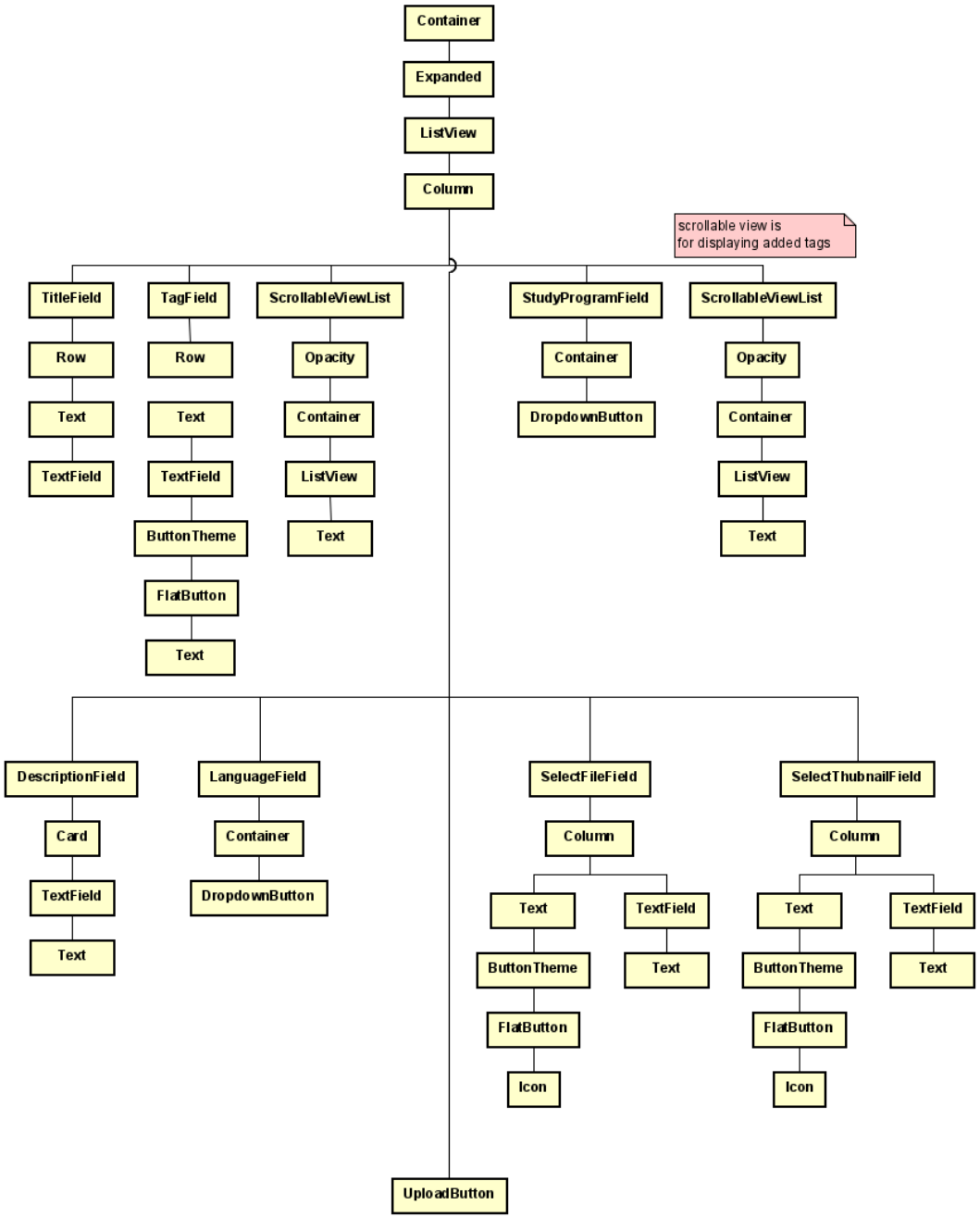
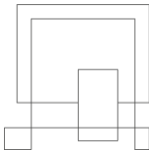


Figure 24 - Upload Video widget tree

The visualizations of rest of the widgets can be found in user guidelines – Appendix F.



3.4 Security

One of the most important aspects of software development is security and it must be considered during the design phase of the project execution. There are two aspects of security to be considered in scope of this project: authentication and authorization. In this chapter, only design of the implemented solution is presented. The topic is continued in Implementation.

3.4.1 Authentication

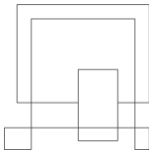
As the system is not open for everybody, but only for students and teachers of schools and universities, a way to limit the access to the platform must be implemented. In order to achieve it, an OAuth2 authentication mechanism has been used as it is safe, reliable, widely supported and easy to use. (Spring Boot and OAuth2, 2020)

In order to access the system, a user needs to login first using login and password. A request is sent with user's credentials to the authentication server and if the credentials match a specific user, the access to the system is granted and an access token is generated. The token is stored in the client application that the user is using. From now on, the token is attached to every request that is made to the service layer as a result of the user's actions, authenticating the user in the system and granting access to the server resources. To mitigate the possibility of stealing the token by a potential attacker, the token has an expiration time of 5000 seconds, after which the user must log in again.

3.4.2 Authorization

To guarantee that specific parts of the system can be accessed by only a subset of users, implementation of Role Based Access Control (RBAC) was needed. In order to do that, one has to first identify system roles within the system. The following system roles have been identified: Admin, Teacher and Student.

When a user is created in a system, they have one of those roles assigned to them. During the login process, when an access token is generated, it is associated with the



user representation in the database, that has information about the user's system role. Thanks to that, when a user's action triggers the client application to make a request to the server, the system retrieves the user's details from the database using a provided access token and verifies that the user making the request has a required role.

3.4.3 Authentication flow

In order to authenticate and authorize requests to the services, one of the services was chosen to be an Authorization Server. It is responsible for authenticating users, generating access tokens and providing information about active tokens to the other services. When a user logs in, a POST request is sent to the UserService to generate an access token. The token is saved in the memory of UserService, alongside with the details of the authenticated user. Next time, the user makes a request to one of the services, before the request is handled, the service makes a request to the Authorization Server(UserService) to validate the token and provide information about the authenticated user. At this point, the service can validate that the user has required roles and authorize the user. When both validations have been completed, the service handles the request. The authentication flow is presented in the Figure 25.

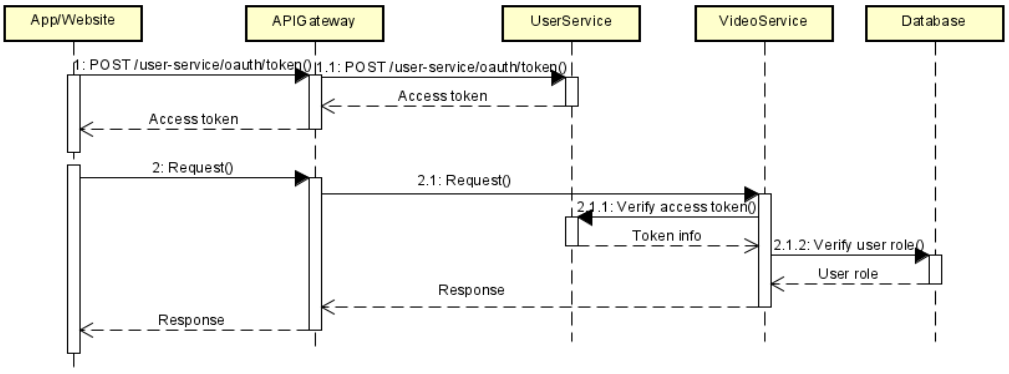
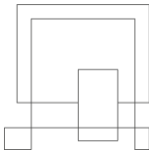


Figure 25 - Authentication flow diagram



3.5 Core features

3.5.1 Video uploading

The upload video feature is using bloc pattern as any other functionality. When the user picks a file that their wants to upload and presses the upload button, the UI triggers an upload video event that takes in the parameter name of the picked file, video in bytes list format, and video metadata that contains information about that video like title, tags, study program, description, length, language, and thumbnail. Then the bloc prepares all data for the repository which sends a request to Video Service. Depending on the response from the server an upload video status is returned to the upload video bloc that yields the corresponding state to UI. The flow of uploading video in frontend is demonstrated in the sequence diagram below (Figure 26).

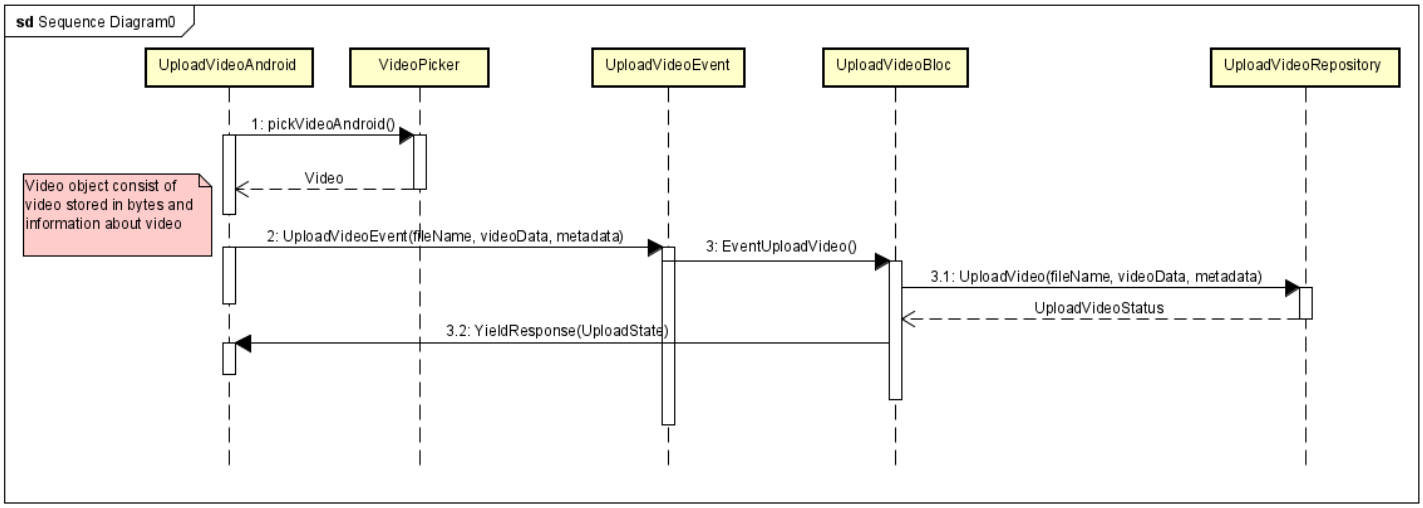


Figure 26 - Video uploading sequence diagram

3.5.2 Video streaming

Another topic that is worth mentioning is video streaming. In order for the client application to access and play the video from the network resource, both client and the server must be compatible. To utilize the file storage mechanism of MongoDB GridFS it was decided to expose videos as a Partial Content resource which is part of HTTP specification.

When a client wants to get the video file, it sends a GET request to the server. As the video file might have a size of hundreds of megabytes, it is required that the file must be loaded to the server memory to be sent in a response. This solution could lead to problems with filling server runtime memory and could cause crashes from reaching server memory limits. To avoid this problem, instead of returning the whole file in one response, the server returns only a part of the file, alongside with response code 206 Partial Content, that tells the client that it received only a chunk of the file. The client executes the next request using Content-Range response header to create a new request that specify the next chunk of the data. In this way, the client can download the whole video file from the server in chunks and avoid memory problems on both client and server side. Moreover, the client does not have to wait to receive the whole file in order to start playing video as the received chunks are ordered. The Request/Response sequence is shown in the Figure 27.

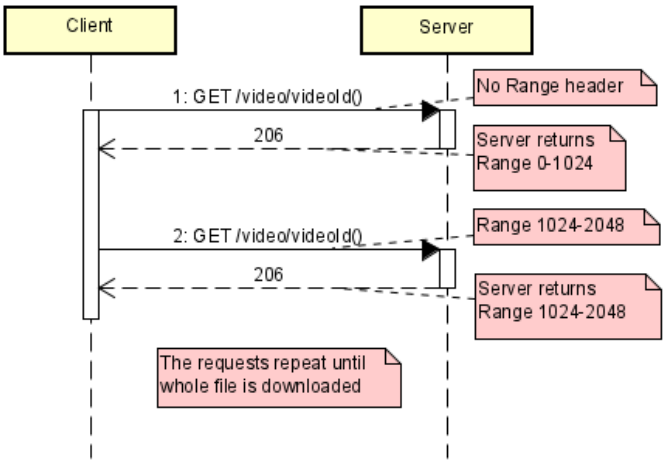
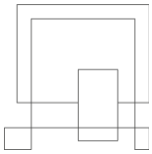


Figure 27 - Request-Response sequence diagram for video streaming



3.5.3 Recommendations

In order to align with the Neo4j model created during analysis (see section 2.4.2 - Neo4j – Recommendations), it is necessary to design effective flows through the systems between components that hold different resources. There are 4 services relevant in case of recommendations such as UserService, VideoService, SearchService and RecommendationService.

The System Sequence Diagram below (Figure 28) shows 2 selected scenarios - sequences and their relevant components.

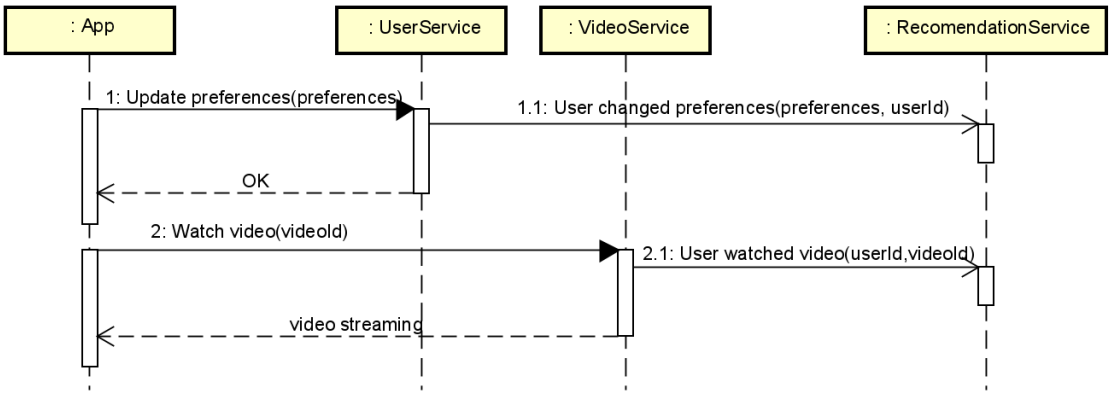


Figure 28 - Gathering behaviour and preferences sequence diagram

The first sequence is related to the use case when a user updates their preferences, where the flow is as follows:

1. App makes an HTTP request to a UserServices to update user's preferences.
2. UserService updates user preferences stored in *users* collection of MongoDB
3. UserService makes two actions:
   3.1. UserService sends asynchronous message to RecommendationService with updated preferences and userId
      3.1.1. RecommendationService stores updated preference in form of new edge - relationship
   3.2. Meanwhile the RecommendationService is storing updated preferences, the UserService responds back to the App with a successful message.

The second sequence relates to the scenario when a user requests to watch a video. It starts when the App requests a video content of specific video from VideoService by videoId parameter. When the VideoService is preparing for streaming requested video, it also sends an asynchronous message to RecommendationService which records the WatchAction, so that it creates *Watches* edge between User and Video nodes specified by passed userId and videoId parameters.

The following System Sequence Diagram (Figure 29) explains the use case when a user wants to get recommended videos. In this scenario, the App makes an HTTP Get request directly to the SearchService, which executes recommendation query to the RecommendationDB - Neo4j. This query uses the recommendation algorithm described in the section 2.4.2.1 - Recommendations algorithm, in order to find the best matching videos. It returns video ids back to SearchService. Once the videoIds are received, they are used to get actual videos by querying videos collection in MongoDB, which responds back with actual videos that are followingly returned back to the App.

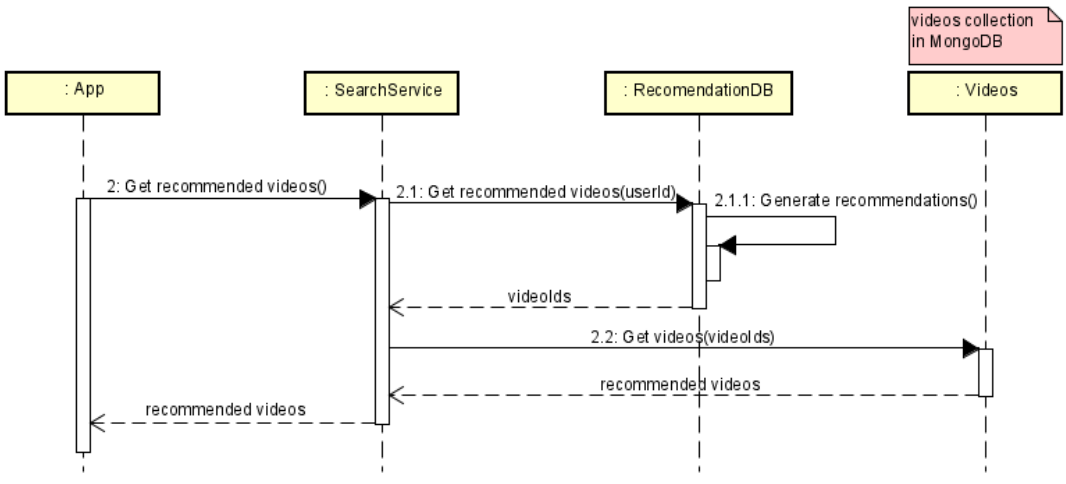
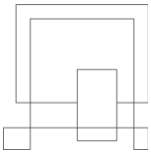


Figure 29 - Generating recommendations flow sequence diagram



4 Implementation

4.1 Recommendations

The logic behind the recommendation algorithm is enclosed in one query that is called from recommendation-service on Neo4j database.

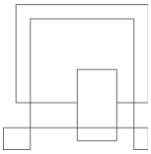
As mentioned in the data schema presented in section 2.4.2 - Neo4j – Recommendations, there are 8 different factors that have impact on video recommendations. Each of these factors is wrapped into its own call block which produces a list of the best matching videos considering only the specific factor.

In the code snippet below (Figure 30) is shown the call block that is finding the best matching videos based on the videos that a user has watched.

```
30 CALL {
31     WITH user,watchedVideos,videoIdsToExclude
32     UNWIND watchedVideos as watchedVideo
33     WITH user, watchedVideo.watchedVideo as watchedVideo, watchedVideo.wv as wv, videoIdsToExclude
34     MATCH (watchedVideo)-[:HasStudyProgram]→(spFromWatch:StudyProgram)
35     MATCH (watchedVideo)-[:HasTag]→(tagFromWatch:Tag)
36     OPTIONAL MATCH (videoFromWatchSP:Video)-[:HasStudyProgram]→(spFromWatch)
37         WHERE NOT videoFromWatchSP.videoId IN videoIdsToExclude
38     OPTIONAL MATCH (videoFromWatchTag:Video)-[:HasTag]→(tagFromWatch)
39         WHERE NOT videoFromWatchTag.videoId IN videoIdsToExclude
40     WITH collect(DISTINCT videoFromWatchSP)
41         +collect(DISTINCT videoFromWatchTag)
42             as videosFromWatch,wv
43     UNWIND videosFromWatch as videoFromWatch
44     WITH DISTINCT videoFromWatch,wv
45     WITH videoFromWatch,wv,wv.priority-duration.InMonths(date(wv.time),date()).months as priorityFromWatch
46     WHERE priorityFromWatch > 0
47     WITH videoFromWatch,Sum(priorityFromWatch) as priorityFromWatch
48     RETURN collect(DISTINCT {video:videoFromWatch,priority:priorityFromWatch}) as videosFromWatch
49 }
```

Figure 30 - Recommendation query - part for Watch action

Firstly, studyPrograms and tags of the watched video are identified (lines 34-35). Afterwards, the video candidates that have the same tag or studyProgram as the watched video are found. All the candidates have the same initial priority which is set to 12 in the case of “Watch” action. However, “Watch” action as well as the rest of actions have their priority decreased by the time where each passed month from the date of action represents a penalty of 1 to the initial priority (line 45). Eventually, (line 48) all candidates are collected into a list that is later joined with the lists from the rest of call blocks.



The action when a user searches videos by a searchTerm has to be implemented in a different way as the rest of user actions. In this case, it was required to make full-text search on title and description attribute from Video node and name attribute from StudyProgram and Tag nodes.

For the full-text search, the Apache Lucene indexing and search library was chosen. To execute the full-text search, an index on aforementioned attributes was created by the following procedure:

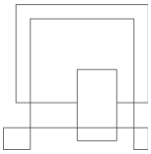
```
1 CALL db.index.fulltext.createNodeIndex("search",["Video","StudyProgram","Tag"],["title","description","name"])
```

Figure 31 - Query to create full-text search index in Neo4j

Once the index was created (Figure 31), it can be used in the recommendation query in order to find the best matching videos that were found by the full-text search (line 121 in Figure 32). Depending on a node, the result from full-text search is casted to the relevant attribute (line 125-127). In the case of Tag and StudyProgram, they are used to find videos that have either Tag or StudyProgram in common and in the case of Video, the actual video is passed into the final list of recommended videos from search action (line 131-133). At the end, the time penalty is applied to the priorities as it was described in the query description of Watch action (line 136).

```
118 CALL {
119     WITH user
120     MATCH (user)-[sv:SearchesVideo]→(searchTerm:SearchTerm)
121     CALL db.index.fulltext.queryNodes("search", searchTerm.name) YIELD node, score
122     WITH
123         score,
124         sv,
125         CASE node:Video WHEN true THEN node ELSE null END AS videoFromSearch,
126         CASE node:Tag WHEN true THEN node ELSE null END AS tagFromSearch,
127         CASE node:StudyProgram WHEN true THEN node ELSE null END AS spFromSearch
128
129     OPTIONAL MATCH (videoFromSearchSP:Video)-[:HasStudyProgram]→(spFromSearch)
130     OPTIONAL MATCH (videoFromSearchTag:Video)-[:HasTag]→(tagFromSearch)
131     WITH collect(DISTINCT videoFromSearchSP)
132         +collect(DISTINCT videoFromSearchTag)
133         +collect(videoFromSearch)
134         as videosFromSearch,sv
135     UNWIND videosFromSearch as videoFromSearch
136     WITH videoFromSearch,sv,sv.priority-duration.InMonths(date(sv.time),date()).months as priorityFromSearch
137     WHERE priorityFromSearch > 0
138     WITH videoFromSearch,Sum(priorityFromSearch) as priorityFromSearch
139     RETURN collect(DISTINCT {video:videoFromSearch,priority:priorityFromSearch}) as videosFromSearch
140 }
```

Figure 32 - Recommendation query - part for Search action



When videos from all the factors are gathered and joined into one list (lines 142-151 in Figure 33), it is needed to filter the list and remove all videos that the user either created or already watched (line 155). After that the final list of recommended videos can be constructed by grouping all entries by videoId and sums priorities of videos with the same id (lines 156-157).

```
142 WITH videosFromStudyProgram
143     +videosFromTag
144     +videosFromLength
145     +videosFromWatch
146     +videosFromLike
147     +videosFromDislike
148     +videosFromCommentPositive
149     +videosFromCommentNegative
150     +videosFromSearch
151     as videos,
152     videoIdsToExclude
153 UNWIND videos as v
154 WITH v,videoIdsToExclude
155 WHERE NOT v.video.videoId IN videoIdsToExclude
156 RETURN v.video.title,v.video.videoId, Sum(v.priority) as priority
157     ORDER BY priority DESC, v.video.videoId ASC
```

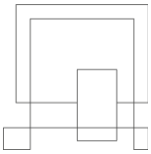
Figure 33 - Recommendation query - final part

4.2 File storage

Considering the design of the file storage presented in section 3.1.3.1 - File storage, it was decided to make use of the implementation of GridFS provided by Spring Data MongoDB called *GridFsTemplate*. In this system the GridFsTemplate is used to:

1. Store a new file with its metadata
2. Delete an existing file with its metadata and content
3. Find one or more files – documents stored in VideoFiles collection
4. Get content of a specific file – documents stored in VideoContents collection

The rest of this section is focused on the implementation of storing a new file into MongoDB GridFS.

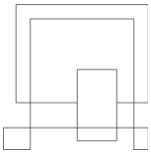


```java
@Log4j2
@RestController
@RequestMapping("/videos")
public class VideoController {

    private final VideoService videoService;

    public VideoController(VideoService videoService) { this.videoService = videoService; }

    @PreAuthorize("hasAuthority(T(com.streamster.videoservice.model.SystemRoleType).TEACHER)")
    @PostMapping("/upload")
    public ResponseEntity<String> upload(Principal principal,
                                         @RequestParam("video") MultipartFile file,
                                         @Valid @RequestPart VideoMetadataDTO metadata) {
        this.videoService.uploadVideo(file, principal.getName(), metadata.toDocument());
        return new ResponseEntity<>(HttpStatus.CREATED);
    }
```

Figure 34 - REST controller for uploading video

The VideoController class is the RESTful controller that provides endpoints for managing videos. The example of the endpoint for uploading the video can be seen in the code snippet above (Figure 34). The VideoController requires to:

1. provide the video in the form of MultipartFile object attached to the request body as a form-data
2. provide video's metadata attached in the request body as a JSON representation of a valid VideoMetadataDTO object – see the diagram below (Figure 35) where is shown the VideoMetadataDTO class with its fields and fields annotations that are used to validate the parameters passed to the constructor – annotations such as @NotNull, @NotEmpty or @Pattern which validates a regex expression.



```java
@Data
@AllArgsConstructor
@NoArgsConstructor
public class VideoMetadataDTO {
    @NotNull
    private String title;
    private String description;
    @NotNull @NotEmpty
    private Set<String> tags;
    @NotNull @NotEmpty
    private Set<String> studyPrograms;
    @Pattern(regexp = "^([A-Za-z0-9+/]{4})*([A-Za-z0-9+/]{3}=|[A-Za-z0-9+/]{2}==)?$",
            message = "thumbnail is not valid Base64 encoded string")
    private String thumbnail;
    private String language;
    @NotNull
    private Long length;
```

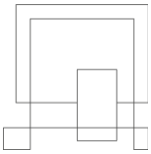
Figure 35 - Example of VideoMetadataDTO class

Afterwards, the VideoController calls the *uploadVideo* method on the videoService with passed video file, user's email taken from the Principal object that is retrieved from the token, and the video's metadata in a form of Document object required by GridFsTemplate implementation.

```java
public String uploadVideo(MultipartFile file, String userEmail, Document metadata) {
    var user :User = userRepository
            .findByEmail(userEmail)
            .orElseThrow(() -> new NoSuchElementException("Cannot be found user with email: " + userEmail));

    metadata.put("authorId", user.getId());
    metadata.put("authorName", user.getFirstName() + " " + user.getLastName());
    // Store file to the GridFS
    String fileId = fileService.store(file, metadata);
    metadata.put("videoId", fileId);
    // Update user service
    proxyService.addVideoToUser(fileId, user.getId());
    proxyService.addVideoToRecommendations(metadata, user.getId(), fileId);
    return fileId;
}
```

Figure 36 - Implementation of uploadVideo method in VideoService

The code snippet above (Figure 36) represents the *uploadVideo* method from VideoService class, which was called from VideoController.

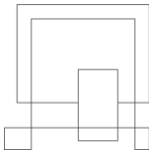


In this method it is important firstly to find the user in the userRepository to get the id and name of a user that is uploading a video. Once, the user is found and retrieved, the id and name are added to the metadata document. At this point, everything is ready to store the file with its metadata into MongoDB. As mentioned in the design section of File Storage, there is both-way referencing to ids between user and video entities. Therefore, when the video is successfully stored into MongoDB, it is important to add a new videoId into the user document. It is done by calling *addVideoToUser* method on the proxyService with passed ids of file and user. In the proxyService a message is created and sent to the *user* queue using RabbitMQ. The UserService listens for messages incoming to the *user* queue and processes them. The communication between services is asynchronous, therefore VideoService is not waiting for a response but returns a successful message back to the client with HTTP status 201 - Created. Full implementation of the file storage, as well as source code of the Streamster application, can be found in Appendix E.

4.3 Security

To implement authentication and authorization according to OAuth2 specification, Spring Security has been used. It distinguishes the services into Authorization Server and Resource Server. The UserService is configured to be an Authorization Server. All other services are Resource Servers, that are using Authorization Server to validate access tokens. The following steps are needed to configure Spring Security OAuth2:

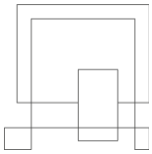
1. Spring Security OAuth2 must be added as a dependency to the service
2. Authorization server needs to be configured by extending AuthorizationServerConfigurerAdapter. In this class, the details of request to generate access tokens are specified. At this point, the Authorization Service is ready.



```
@Override
public void configure(AuthorizationServerSecurityConfigurer oauthServer) throws Exception {
    oauthServer.tokenKeyAccess("permitAll()")
            .checkTokenAccess("permitAll()");
}

@Override
public void configure(ClientDetailsServiceConfigurer clients) throws Exception {
    clients.inMemory() InMemoryClientDetailsServiceBuilder
            .withClient( clientId: "my-trusted-client") ClientDetailsServiceBuilder<B>.ClientBuilder
            .authorizedGrantTypes("client_credentials", "password", "authorization_code")
            .authorities(SystemRoleType.getAllRoles())
            .scopes("read", "write", "trust")
            .resourceIds("oauth2-resource")
            .accessTokenValiditySeconds(5000)
            .secret(passwordEncoder.encode( rawPassword: "secret"));
}
```

Figure 37 - Example of OAuth2 configuration for Spring Boot

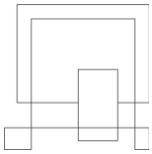
3. To allow Resource Servers to communicate with Authorization Server, a class that implements ResourceServerTokenServices is required that handles the call to the Authorization Server to validate the access token.
4. Next step is to configure Resource Server security. To do that, a configuration class that extends ResourceServerConfigurerAdapter is created. The main purpose of this class is to configure the authentication process and define which endpoints must be secured by authentication. The example of configuration can be seen below. On the Figure 38, it can be seen that all endpoints require authentication by specifying *.antMatchers("/**").authenticated()* with few exceptions. Each of the endpoints that does not require authentication is listed and configured as *permitAll().*



```
@Configuration
@EnableResourceServer
public class ResourceServerConfig extends ResourceServerConfigurerAdapter {

    @Override
    public void configure(HttpSecurity http) throws Exception {
        http
                .headers() HeadersConfigurer<HttpSecurity>
                .frameOptions() HeadersConfigurer<H>.FrameOptionsConfig
                .disable() HeadersConfigurer<HttpSecurity>
                .and() HttpSecurity
                .authorizeRequests() ExpressionUrlAuthorizationConfigurer<H>.ExpressionInterceptUrlRegistry
                .antMatchers( ...antPatterns: "/**").authenticated()
                .antMatchers( ...antPatterns: "/oauth/check_token",
                        "/register").permitAll()
                .antMatchers( ...antPatterns: "/swagger-ui.html",
                        "/v2/api-docs",
                        "/v2/websockets.json",
                        "/configuration/ui",
                        "/swagger-resources/**",
                        "/configuration/security",
                        "/swagger-ui.html",
                        "/v2/websockets.json",
                        "/webjars/**").permitAll();
    }

    @Override
    public void configure(ResourceServerSecurityConfigurer security) throws Exception {
        security.stateless(false);
    }
}
```

Figure 38 - Configuration of ResourceServer for OAuth2

5. The last step is to specify the source of user details. In this project, the user details are stored in the MongoDB in users collection. The required details are user email, password and roles. Standard MongoRepository that is bound to the users collection is used to achieve it.



4.4 Video streaming

To implement HTTP Partial Content video streaming both server and client must be supporting this protocol and be compatible with each other.

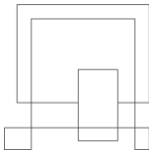
The server implementation utilizes Spring Framework to create Partial Content responses. The code is shown in Figure 39.

```
private final FileService fileService;

public StreamingController(FileService fileService) { this.fileService = fileService; }

@GetMapping("/{videoId}")
public ResponseEntity getVideoChunk(@PathVariable String videoId) throws IOException {
    GridFsResource gridFsResource = fileService.findAsResource(videoId);
    return ResponseEntity.ok().body(new InputStreamResource(gridFsResource.getInputStream()));
}
```

Figure 39 - Implementation of video streaming endpoint

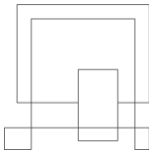
On the code snipped shown above it can be seen that firstly, a FileService is used to retrieve GridFSResource by given id of the video. FileService is a class responsible for encapsulating logic of accessing MongoDB GridFS file storage. Next, a new InputStreamResource is created from GridFSResource by passing its InputStream to the constructor. In this case, InputStreamResource means a HTTP Resource based on an InputStream. Finally, the created InputStreamResource is passed to the builder function as a body of ResponseEntity from the Spring framework. Then, the Spring framework handles interpreting the Request Headers and accessing the desired bytes from the InputStream and creating the response.

On the client side, to consume the server endpoint, a Chewie flutter package is used. It provides implementation of a video player widget and processing Partial Content video streaming. In order to use Chewie, a ChewieController object must be created. It is used to configure the video player with settings such as, aspect ratio, reference to video player widget which contains information about video source (in this case it is network url) and error handling. Once the controller is created, it is passed as a parameter to a Chewie widget. This process is demonstrated as a code snippet in Figure 40.



```dart
@override
void initState() {
  super.initState();
  _chewieController = ChewieController(
      videoPlayerController: widget.videoPlayerController,
      aspectRatio: 16 / 9,
      autoInitialize: true,
      looping: widget.looping,
      errorBuilder: (context, errorMessage) {
        return Center(
          child: Text(
            errorMessage,
            style: TextStyle(color: Colors.white),
          ),  // Text
        );  // Center
      }
  );  // ChewieController
}

@override
Widget build(BuildContext context) {
  return Padding(
    padding: const EdgeInsets.all(8.0),
    child: Chewie(
      controller: _chewieController,
    ),  // Chewie
  );  // Padding
}
```

Figure 40 - Implementation of video player widget



4.5 BLOC pattern

The implementation of BLoC pattern has 3 core packages:

1. **Data**

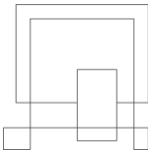
This package consists of a repository that is responsible for managing the feature's domain. In this example the registration repository exposes a stream of registration status in the register function, that will be used to notify the application about the result of an event that can be either successful or unsuccessful. The status init is an initial state of event and the loading status indicates that the event has been triggered and is waiting for the result state.

```
enum RegistrationStatus { init, loading, success, error, }
```

Figure 41 - Example of status enumeration class

```
Future<RegistrationStatus> register(
    {String firstName,
    String lastName,
    String email,
    String password,
    String avatar}) async {

  var user = UserInfo(firstName, lastName, email, password, avatar);

  var response = await http.post(RestClient.registerUrl,
      headers: {'Content-Type': 'application/json'},
      body: convert.jsonEncode(user.toJson()));

  if (response.statusCode == 200) {
    return RegistrationStatus.success;
  } else {
    return RegistrationStatus.error;
  }
}
```

Figure 42 - Implementation of register in UserRepository



2. Bloc

2.1. Events

Event is an abstract class that can be extended by multiple subclasses that defines the event and its necessary attributes. In this example (Figure 43), for the Register User event are necessary following attributes: first name, last name, email, password and avatar.

```
abstract class RegisterEvent extends Equatable {
  const RegisterEvent();
}

class RegisterUser extends RegisterEvent {
  final String firstName;
  final String lastName;
  final String email;
  final String password;
  final String image;
```

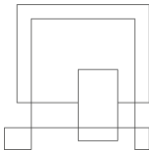
Figure 43 - Example of Event class

2.2. States

State indicates the output of the Bloc and will be consumed by the presentation layer.

```
const RegisterState.loading() : this._(status: RegistrationStatus.loading);

const RegisterState.success() : this._(status: RegistrationStatus.success);

const RegisterState.error(String message) : this._(status: RegistrationStatus.error, error: message);
```

Figure 44 - States returned by Bloc object



2.3. Bloc

Bloc maps events to its states and initializes the repository. This is the logic of the bloc pattern.

```
@override
Stream<RegisterState> mapEventToState(RegisterEvent event) async* {

  if (event is RegisterUser) {
    yield RegisterState.loading();

     var response = await _registerRepository.register(
        firstName: event.firstName,
        lastName: event.lastName,
        email: event.email,
        password: event.password,
       avatar: event.image
     );
     if(response == RegistrationStatus.success) {
       yield RegisterState.success();
    } else {
       yield RegisterState.error("error");
     }
  }
}
```

Figure 45 - Example of mapEventToState method in Bloc class

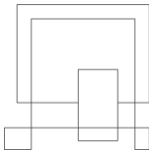
3. View

View is the presentation layer that listens for states, triggers events and builds the layout of the application.

```
return BlocListener<RegisterBloc, RegisterState>(
  listener: (context, state) {
    handleState(state);
  },
  child:
  BlocBuilder<RegisterBloc, RegisterState>(builder: (context, state)
```

Figure 46 - Example of BlocListener in one of the widgets

From the listener, we can retrieve state of an event and pass it to the function handleState that decides, how view will react on event's result.



```
} else if (state.status == RegistrationStatus.success) {
  Scaffold.of(context).showSnackBar(SnackBar(
    content: Text('success'),
    backgroundColor: Colors.green,
  )); // SnackBar
  Navigator.of(context).pushNamed('/login');
}
```

Figure 47 - Implementation of handleState method in one of the widgets

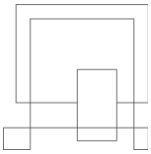
For example, if a registration is successful the application will display successful dialog and navigates to login page so user can login with their new account.

4.6 Uploading videos and images

Uploading videos and images were made by an external plugin ImagePicker, that allows choosing a file from the gallery or taking a new picture or video with the camera. This plugin is supported only on Android and iOS (ImagePicker Documentation, 2020), therefore the web application needed an alternative function for uploading videos or images. For the web was used universal HTML plugin, which allowed to pick a file from the file explorer.

```
static Future<Video> pickVideoAndroid() async {
  ImagePicker picker = ImagePicker();
  File _androidVideo;
  final pickedFile = await picker.getVideo(source: ImageSource.gallery);

  if (pickedFile != null) {
    _androidVideo = File(pickedFile.path);
    String fileName = _androidVideo.path.split('/').last;
    var videoExtension = p.extension(_androidVideo.path);
    var videoData = _androidVideo.readAsBytesSync();
    var videoSize = videoData.length;
    var videoLength = await getVideoLength(pickedFile.path);

    return new Video(
        videoData, videoSize, videoLength, videoExtension, fileName);
  }
```

Figure 48 - Implementation of pickVideoAndroid method



The logic of the function (Figure 48) is to read a file from a given source, convert the file to a byte list, get the size of the file and length of the video. The limitation for the video length is 1GB that is set by Spring and can be changed. In the case of video picker for web, the function is a little bit more complicated, first, it is reading the file as string base64, then it decodes data to a byte list.

Uploading images works in the same way, instead of getVideo function is used function getImage and then the image is encoded to base64 and send as String (Figure 49).

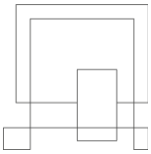
```
List<int> imageBytes = androidImage.readAsBytesSync();
String base64Image = base64Encode(imageBytes);
```

Figure 49 - Example of encoding image to base64

Then in the repository is created upload request that is using an HTTP multipart request that allows sending multiple fields of the different data types (Figure 50). The first field is the type of multipart/form-data that sends bytes of video and the second is application/JSON that sends JSON of metadata that contains information about the video.

```
Future<UploadVideoStatus> upload(String filename, List<int> bytes, VideoMetadata metadata) async {
  print("sending");
  var request = http.MultipartRequest(
      'POST', RestClient.uploadUrl);
  request.files
      .add(http.MultipartFile.fromBytes('video', bytes, filename: filename,contentType: MediaType.parse("multipart/form-data")));

  request.files.add(http.MultipartFile.fromString("metadata", convert.jsonEncode(metadata.toJson()),contentType:MediaType.parse("application/json")));

  var result = await restClient.client.send(request);
  if (result.statusCode == 201) {
    print("result: $result");
    return UploadVideoStatus.success;
  } else {
    return UploadVideoStatus.error;
  }
}
```

Figure 50 - Implementation of upload method in UploadVideoRepository

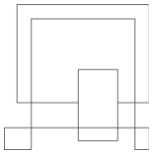


4.7 Cloud Deployment

To be able to deploy the system in the cloud environment, few tasks have been completed. Firstly, as the desired way of deploying the services was to build Docker containers and run them on the Google Cloud Run service, a way of creating Docker images must be chosen in the first place. The usual way to build a Docker image is to define a Dockerfile for the service where operations that are needed to build the image are specified. However, to avoid manual customization of the image and Dockerfiles, it was decided to make use of JIB, a plugin for Gradle - a Java build tool used to build the Java services (Jib - Containerize your Gradle Java project, 2020). By providing only necessary configuration to the JIB plugin in build.gradle file, it is possible to create a cloud-ready Docker image of the service by simply executing *gradle jib* command. The Figure 51 below shows the JIB configuration in the build.gradle file of one of the services. It can be seen that a base image is chosen and a name of the target image is specified. It is worth to notice that the name of the target image is specified as follows: *registry/project/image:tag*. Thanks to that, the JIB plugin can push the created image to the specified Docker registry with tag *latest*. JIB allows also to specify authentication methods for the target remote registry and to specify entry arguments for the created container, such as JVM flags.

```
jib {
    from {
        image = 'adoptopenjdk/openjdk15:jdk-15.0.1_9-alpine-slim'
    }
    to {
        image = 'gcr.io/streamster-289010/video-service:latest'
        auth {
            username = '_json_key'
            password = dockerPassword
        }
    }
    container{
        jvmFlags = ['-Dspring.profiles.active=prod', '--enable-preview']
    }
}
```

Figure 51 - Example of JIB Gradle plugin configuration



When the container image is created, it is needed to prepare the Google Cloud Run service where it can be deployed. Firstly, a service must be created by choosing its name and the region where it should be created.

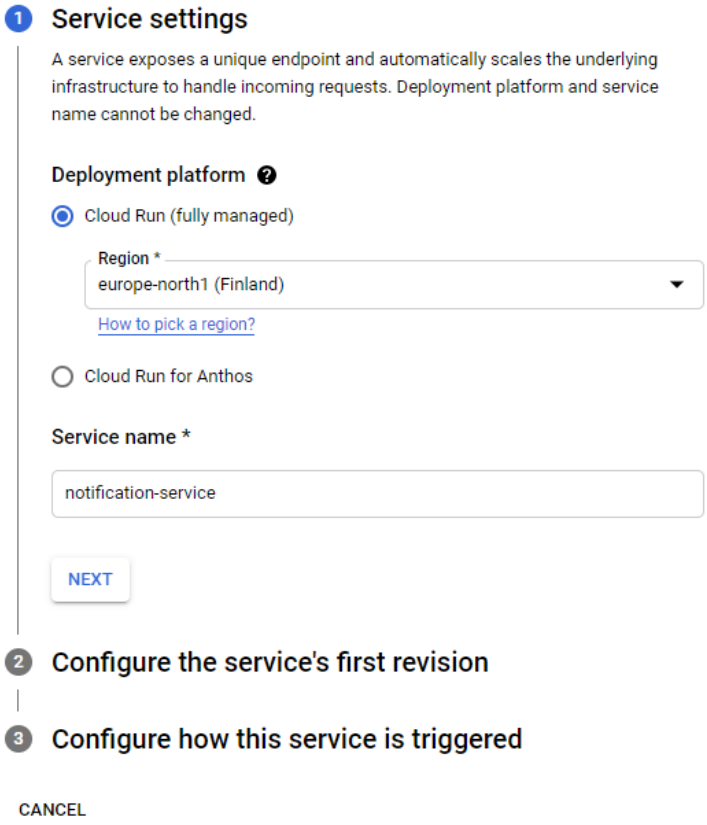
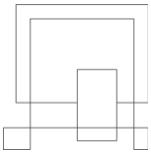


Figure 52 - Creating new service in Google Cloud Run

In the next step, a container image is chosen, and its configuration, such as open ports, memory limit and CPU cores are specified. Moreover, it is possible to set environment variables. In this case, they are used to safely pass secrets, such as database credentials to the container and the application running in it.



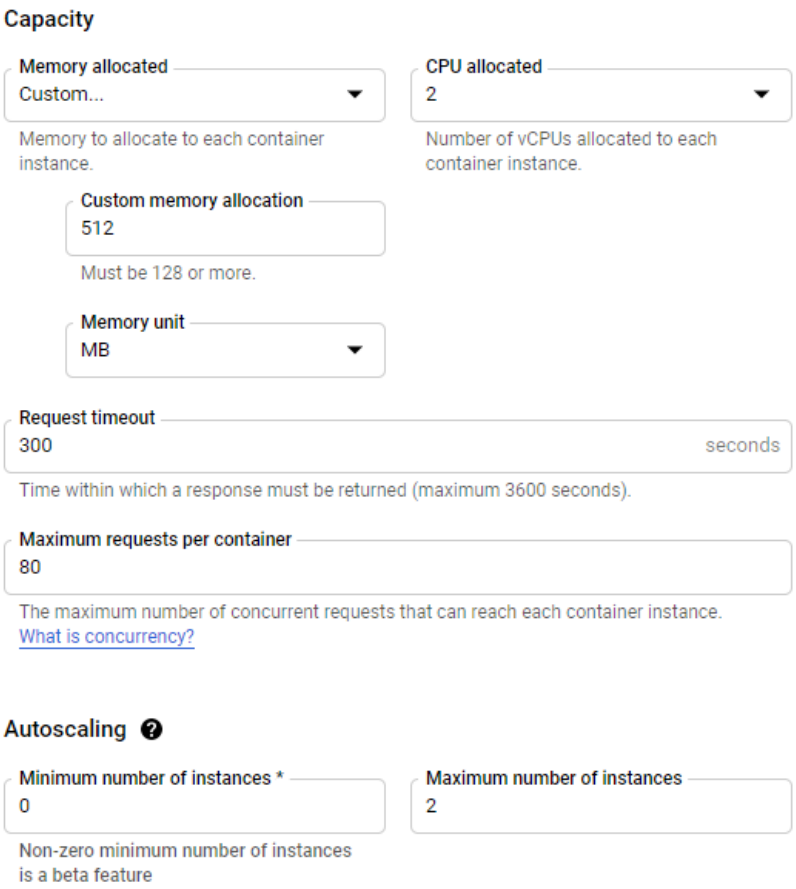


Figure 53 - Configuration of new service instance in Google Cloud Run

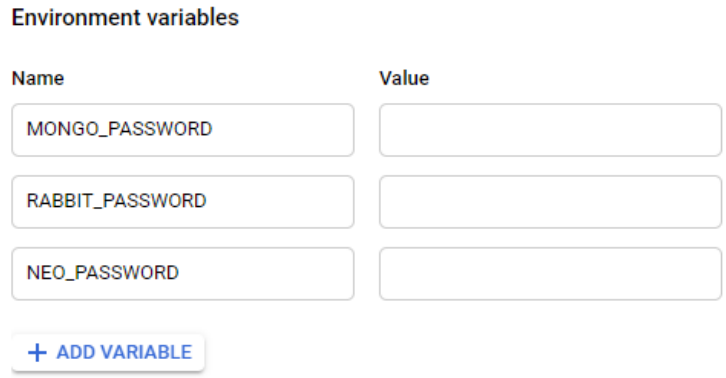
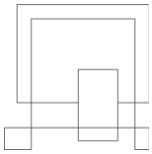


Figure 54 - Example of Environment variables configuration for Cloud Run service



When all the required options are specified, the service can be deployed. After a while, Google Cloud will start to serve traffic to it and the service can be accessed by the generated URL.

As this process must be repeated every time a new version of the service is created, for each of the services, it was decided to automate it using TeamCity as CI/CD server. TeamCity has been configured, so that it listens to the new commits pushed to the master branch of the project repository. It fetches new changes, builds all the services, creates the docker images and using Google Cloud CLI, deploys new revision of all services. Thanks to that, each time a feature is merged into the master branch, the cloud environment is updated with the newest changes. In the Figure 55 below, it can be seen a configuration of build pipeline for one of the services, however, the setup is similar for all the services.

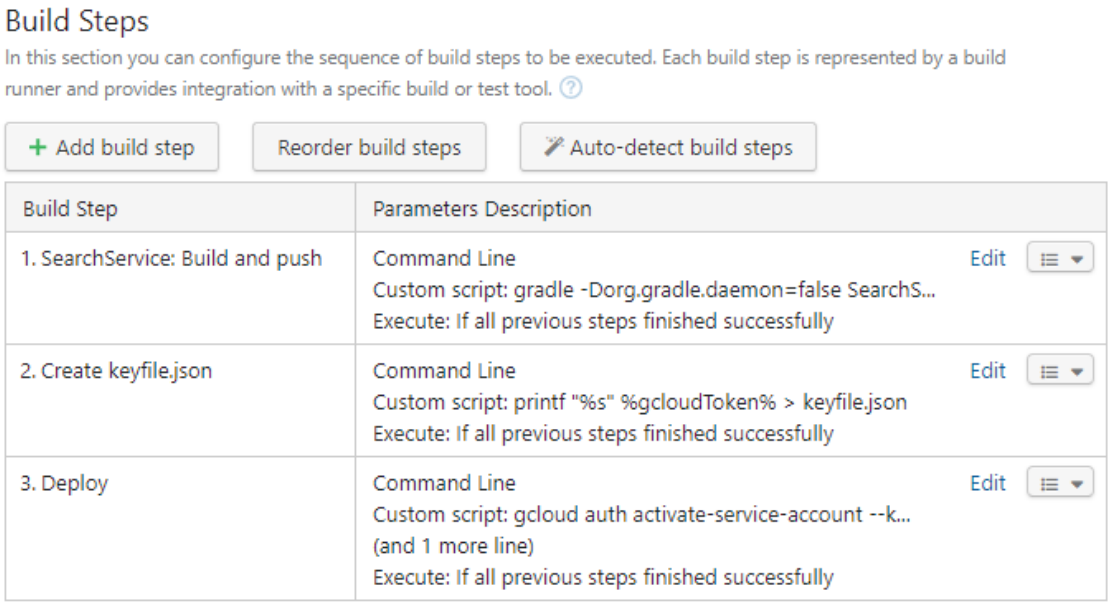
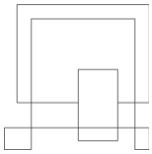


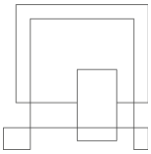
Figure 55 - Build steps configuration in TeamCity



Firstly, the service is built and a docker image is created and pushed to the Docker registry in Google Cloud project. Then, an access token for Google Cloud CLI is created to be used in the Deploy step. In the last step, the service is deployed using following command:

> *gcloud run deploy search-service --project=streamster-289010 --image=gcr.io/streamster-289010/search-service:latest --platform=managed --region=europe-north1 --memory=512M.*

This command performs the same configuration of the service as the one done using Web Interface of Google Cloud.

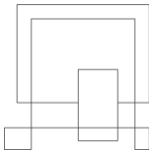


5 Test

One of the most important activities performed during the project execution is testing. To ensure that the created software functions according to the requirements and the code quality is high, the testing must be taken under consideration not only at the end of the project work, but also during it. A suite of unit and integration tests were created during work on the features, to verify that the internal implementation of core functionalities is correct. Moreover, a UAT (User Acceptance Test) was performed, to validate that the created software fulfills the requirements. In the following sections, these testing approaches will be described in detail.

5.1 Unit and Integration tests

To verify the implementation of backend services, it was decided to use the Spock framework. It provides great integration with Spring Boot and allows testing of Spring Web endpoints, mocking services and repositories, and use authenticated users that are compatible with Spring Security. Moreover, Spock framework uses Groovy as its language rather than Java. Thanks to that, there is less boilerplate code needed to write test cases and the structure of the tests is more readable (Niederwieser, Brünings and TheSpockFrameworkTeam, 2020).
It was decided to test only crucial elements of the system or the ones that can change often. The areas selected for automated testing were authentication and permissions, validation of user registration, video upload and database integration.
In the Figure 56 it is shown an example of a unit test written in Spock framework using Groovy language.



```
@WithMockUser(authorities = ['TEACHER'])
def "test upload validation invalid metadata invalid thumbnail"() {
    given:
    def json = "{\n" +
            "    \"title\": \"My video\",\n" +
            "    \"description\": \"This is my new video\",\n" +
            "    \"tags\": [\"Java\",\"OOP\"],\n" +
            "    \"studyPrograms\": [\"ICT\",\"GBE\"],\n" +
            "    \"thumbnail\": \"invalidThumbnail1\",\n" +
            "    \"language\": \"English\",\n" +
            "    \"length\": 45960\n" +
            "}"
    def metadata = new MockMultipartFile( name: "metadata",  originalFilename: "", MediaType.APPLICATION_JSON_VALUE,
            json.getBytes());
    when:
    def results = mvc.perform(multipart( urlTemplate: "/videos/upload")
            .file(video)
            .file(metadata)
            .contentType(MediaType.MULTIPART_FORM_DATA))
    then:
    results.andExpect(status().isBadRequest())
}
```

Figure 56 - Example of Spock test

This test verifies that when a user with Role Teacher uploads a video with invalid thumbnail format, the service responds with a BadRequest message. The test is divided into three, easily distinguishable sections: *given*, *when* and *then*. In the *given* part, test data is prepared. In this case, the video metadata is created and converted into bytes. In the *when* section, the actual action under test is performed. Using *mvc* object, that is provided by Spock Spring Boot integration, a video upload endpoint is called. In the *then* part, the result of the tested action is evaluated, in particular, the response status code is checked.

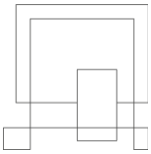
5.2 User Acceptance Tests

The User Acceptance Testing (UAT) was performed in order to test all implemented functionalities from the users perspective. For the evaluation, the sequences stated in Use Case Descriptions were followed and the results can be seen in the table below. Moreover, a Test Team Roles has been defined and distributed amongst the team members:

Matej - Test Lead - responsible for creating UAT Test Plan and defining test cases.

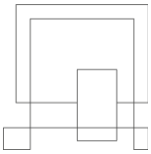
Michaela - Tester - responsible for execution of manual tests

Michał - Test Engineer - responsible for execution of automated tests

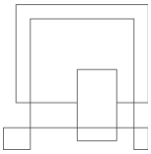


Features that have not been implemented in the scope of this project are not shown in the following table.

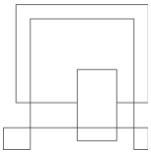
nº	Description	Result ✔ / X	error
register			
1	The user clicks the "create a new account" button. The system navigates to registration page with a registration form that contains fields for name, last name, password, email and avatar. The user fills all fields and uploads a picture. The user clicks the "Submit" button. The system registers a user with student role and displays successful message dialog.	✔	
2	The user clicks the "Register" button. The system navigates to registration page with a registration form that contains fields for name, last name, password, email and avatar. The user fills all fields and does not upload a picture. The user clicks the "Submit" button. The system registers a user with default picture and student role and displays successful message dialog.	✔	
3	The user clicks the "Register" button. The system navigates to registration page with a registration form that contains fields for name, last name, password, email and avatar. The user fills all fields but email that is already used. The user clicks the "Submit" button. The system displays "This email is already being used" message.	✔	
4	The user clicks the "Register" button. The system navigates to registration page with a registration form that contains fields for name, last name, password, email and avatar. The user fills all fields but an invalid email. The user clicks the "Submit" button. The system displays "Email is not valid" message.	✔	
5	The user clicks the "Register" button. The system navigates to registration page with a registration form that contains fields for name, last name, password, email and avatar. The user fills all fields and uploads a picture of invalid format. The user clicks the "Submit" button. The system displays "Invalid format" message.	✔	Not applicable on Android



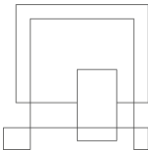
6	The user clicks the "Register" button. The system navigates to registration page with a registration form that contains fields for name, last name, password, email and avatar. The user fills except password. The user clicks the "Submit" button. The system displays "Password is empty" message	✔	
7	The user clicks the "Register" button. The system navigates to registration page with a registration form that contains fields for name, last name, password, email and avatar. The user fills all fields with password that contains less than 8 characters. The user clicks the "Submit" button. The system displays "Password must be at least 8 characters long" message	✔	
8	The user clicks the "Register" button. The system navigates to registration page with a registration form that contains fields for name, last name, password, email and avatar. The user fills all fields with password that does not contain digit. The user clicks the "Submit" button. The system displays "Password must contain at least one digit" message	✔	
9	The user clicks the "Register" button. The system navigates to registration page with a registration form that contains fields for name, last name, password, email and avatar. The user fills all fields with password that does not contain uppercase letter. The user clicks the "Submit" button. The system displays "Password must contain at least one uppercase letter" message	✔	
10	The user clicks the "Register" button. The system navigates to registration page with a registration form that contains fields for name, last name, password, email and avatar. The user fills all fields with password that does not contain lowercase letter. The user clicks the "Submit" button. The system displays "Password must contain at least one lowercase letter" message	✔	
login / logout			
1	The user enters their email and password to the login form. The user clicks "Login" button. The system authenticates the user. The system redirects user to the home page	✔	



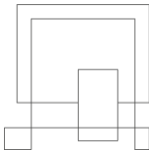
2	The user enters their email and invalid password to the login form. The user clicks "Login" button. Then the system does not authenticate the user and displays "The email or password is incorrect" message	✔	
3	The user enters invalid email and their password to the login form. The user clicks "Login" button. Then the system does not authenticate the user and displays "The email or password is incorrect" message	✔	
4	The user clicks "Logout" button. The system logs out the user. The system redirects the user to the login page	✔	
manage preferences			
1.	The user clicks preferences button. The system navigates to preferences page. The user specifies favourite study programs, tags, preferred minimal and maximal length of the video and clicks save button. The system updates user's personal preferences.	✔	
2.	The user clicks preferences button. The system navigates to preferences page. The user specifies favourite study programs, tags, preferred minimal and maximal length where maximum length is smaller than minimal and clicks save button. The system displays error message “Maximum length must be bigger than minimum”	✔	
upload video			
1.	The teacher navigates to the profile and clicks upload video button. The system displays upload video page.	✔	
2.	On upload video page, the teacher selects a file, enters title, study programs, tags and thumbnail. The teacher clicks upload button. The system checks video format and attributes. The system stores video to database, and shows the uploading progress bar. When the video is stored, the system sends notification with a message 'Your video was uploaded successfully'.	✔	
3.	On upload video page, the teacher selects a file, enters title, study programs, tags, thumbnail and optional language attribute and group access. The teacher	X	group feature is



	clicks upload button.The system checks video format and attributes. The system stores video to database, and shows the uploading progress bar. When the video is stored, the system sends notification with a message 'Your video was uploaded successfully'.		not implemented
4.	On upload video page, the teacher selects a file of unsupported format.The teacher clicks upload button. The system checks video format. The system displays an error message 'Video format is not supported'	X	system is not checking video format
5.	On upload video page, the teacher selects a file, and does not enter title. The system checks video format and attributes. The system displays an error message 'Video title must be specified'	✔	
6.	On upload video page, the teacher selects a file, and does not enter any tag. The system checks video format and attributes. The system displays an error message 'At least one tag must be specified'	✔	
7.	On upload video page, the teacher selects a file, and does not enter any study program. The system checks video format and attributes. The system displays an error message 'At least one study program must be specified'	✔	
8.	On upload video page, the teacher selects a file, and does not upload any thumbnail. The system checks video format and attributes. The system displays an error message 'No thumbnail uploaded'	✔	
watch video			
1.	The user opens a video page. The system displays video page and starts playing the video. The user watches the video	✔	
2.	While watching a video, the user clicks pause button. The system stops playing the video.	✔	
3.	While watching a video, the user changes the volume of the video using volume buttons / volume dial. The system changes the volume of played video	✔	
4.	While watching a video, the user taps on the video. The system shows video timeline. The user clicks on the	✔	



	specific place on the video timeline. The system starts playing the video from selected timestamp.		
5.	While watching a video, the user hovers the mouse on the video. The system shows video timeline. The user clicks on the specific place on the video timeline. The system starts playing the video from selected timestamp.	✔	
search video			
1.	The user enters search term into the search bar and clicks 'Search' button. The system displays videos matching search term	✔	
2.	User clicks 'Advanced search' button. The system displays options for advanced search. The user specifies one or multiple search criteria such as: video tags, creator of the video, length of the video or upload time. The system displays videos matching search criteria	X	advanced search is not implemented
manage feedback			
1.	The user clicks the like button. The system registers reaction and updates the amount of likes in the video.	✔	
2.	The user clicks the dislike button. The system registers reaction and updates the amount of dislikes in the video.	X	dislike is not implemented in frontend
recommendations			
1.	The user opens a home page. The system generates actual recommendations for a user using information about their behaviour (watch, search, like, dislike) and their defined preferences. The system displays home page with recommended videos	✔	
2.	The user watches a video. The system registers user behaviour.	✔	
3.	The user likes a video. The system registers user behaviour.	✔	

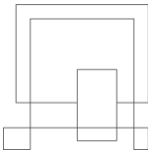


4.	The user dislikes a video. The system registers user behaviour.	X	dislike is not implemented in frontend
5.	The user searches by a searchTerm. The system registers user behaviour.	✔	
6.	The user changes their preferences. The system registers the change and updates changed preferences.	✔	
manage users			
1.	The admin selects user with student role and clicks 'Update to teacher' button. The system updates selected role, and displays message 'User roles has been updated'. The system notifies the user that their role has been changed.	✔	
2.	The admin selects user with teacher role and clicks 'Update to student' button. The system updates selected role, and displays message 'User roles has been updated'. The system notifies the user that their role has been changed	✔	

Table 3 - UAT Test cases descriptions table

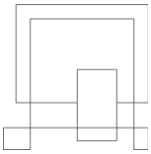
5.3 Testing Tool – Data Generator

For testing the recommendation algorithm it was needed to prepare a lot of dummy data of users, videos and user's behaviour to verify that the result from recommendations is correct. Instead of creating data manually, a python script was made that generates user data and videos that are pushed through services to the database. By using the system's API and not operating on the database itself, it also allowed to verify the communication between the services using RabbitMQ. The generator classes have a set of random names, surnames, emails, pictures, titles, study programs, tags, languages and videos that are randomly picked, encoded and sent to the database. Thanks to that approach, it is possible to easily generate large quantities of valid test data.



6 Results and Discussion

The general outcome of the project can be stated as positive. Even though not all planned features were implemented, the core functionalities of the system have been completed and they have passed the UAT with a good outcome. There are still few features that were not started, such as Group and Notifications. Moreover, a web version of the Streamster application was abandoned because of the time constraints and needs to be aligned with the Android application. A performance test, which was conducted while using the Data Generator, has shown that the system functions well under the stress and with low internet speed. That proves the completion of performance focused non-functional requirements. Furthermore, Data Generator was used to verify recommendation algorithm on larger data set. Even though the current algorithm is performing up to the expectations, the logic behind it is already quite robust. As it is expected to consider more and more user actions and after applying hyper-parameter tuning, it would be even more accurate.



7 Conclusions

The purpose of the project was to improve the experience of using video streaming platforms. It has been achieved by executing several steps that lead to completing this task.

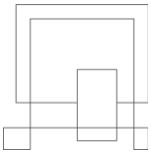
The first step was to study the background of the problem. It was crucial to understand where the current solutions fail the user's expectations and how to improve it. Moreover, to find out what are the needs of the users, that might not have been covered by existing platforms, two personas were created, a student and a teacher. Furthermore, their needs were used to create system requirements, gathered in the Software Requirements Specification. This document became a fundament of the project domain analysis. During the initial phase of the project work, a work schedule was defined and milestones have been set so that the progress of the project can be tracked.

The next step was Analysis, where the non-functional and functional requirements were stated and Product Backlog created. Based on the requirements, the domain could have been analysed by creating a Domain Model and Data Schemas and Use Case diagram. Moreover, during that phase of the project work, a recommendation algorithm has been analyzed, to identify the functionalities of the system required to accurately recommend a video to the user. In order to do that, a survey has been conducted to identify key parameters and to find out which of them are the most important for the users.

Having done the Analysis, one can proceed with Design of the system. In that part, the architecture of the system was defined, to match the requirements. Moreover, the needed technologies were chosen and each of the core system functionalities was reconsidered from the technical perspective.

After completing the Design, the next step was Implementation. It is here where all the work put into previous steps profited as implementation of the features strictly followed their design and were completed successfully.

To validate the implementation, a suite of Unit tests was created that checked the functionalities using white box approach. Moreover, a User Acceptance Test was conducted where based on the requirements and Use Cases the system's functionality was verified. Positive results of the test also proved correctness of the Design and Analysis.



8 Project future

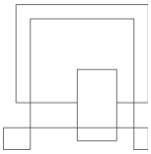
There are several areas of the system that could be improved in the project future. Some of the implemented features have only their must have requirements completed. Moreover, there are functionalities that were not started. The following functionalities should be finished:

1. Account and profile management - could have requirements
2. Login / Logout - could have requirements
3. Managing uploaded videos - all requirements
4. Searching for videos - should have requirements
5. Feedback on videos - could have requirements
6. Groups - all requirements

Alongside with the remaining features, a web application should be aligned with the Android app and have all the features implemented. Furthermore, it can be considered to create an iOS application, depending on the market needs.

Another aspect of the project that can be improved is the recommendation algorithm. As more and more users would use the system, more data can be gathered and a deeper analysis can be conducted to improve the recommendation accuracy.

At last, an agreement with schools or universities can be signed to deploy the system to be used by real users.



9 Sources of information

Angelov, F., 2020. *Bloc library for Dart*. [online] Available at: <https://bloclibrary.dev/#/gettingstarted> [Accessed 17 Dec. 2020].

Anon 1998. *IEEE Recommended Practice for Software Requirements Specifications*. *IEEE Std 830-1998*, .

Anon 2020. *Comparing MongoDB vs PostgreSQL*. [online] Available at: <https://www.mongodb.com/compare/mongodb-postgresql> [Accessed 18 Dec. 2020].

Anon 2020. *GridFS - MongoDB Manual*. [online] Available at: <https://docs.mongodb.com/manual/core/gridfs/> [Accessed 17 Dec. 2020].

Anon 2020. *ImagePicker Documentation*. [online] Available at: <https://pub.dev/packages/image_picker> [Accessed 17 Dec. 2020].

Anon 2020. *Introduction to MongoDB*. [online] Available at: <https://docs.mongodb.com/manual/introduction/> [Accessed 17 Dec. 2020].

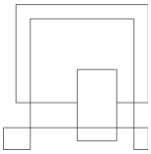
Anon 2020. *Jib - Containerize your Gradle Java project*. [online] Available at: <https://github.com/GoogleContainerTools/jib/tree/master/jib-gradle-plugin> [Accessed 17 Dec. 2020].

Anon 2020. *RabbitMQ*. [online] Available at: <https://www.rabbitmq.com/> [Accessed 17 Dec. 2020].

Anon 2020. *Spring Boot and OAuth2*. [online] Available at: <https://spring.io/guides/tutorials/spring-boot-oauth2/> [Accessed 18 Dec. 2020].

Anon 2020. *What is a Graph Database*. [online] Available at: <https://neo4j.com/developer/graph-database/> [Accessed 17 Dec. 2020].

DEFY Media, 2015. *Acumen Report Constant Content*. [online] Available at: <https://web.archive.org/web/20170417135757/http://sandbox.break.com/acumen/Acumen Constant Content__ExecSum Booklet_Final2.pdf>.



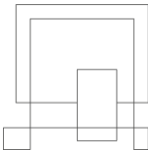
Google, 2020. *The latest YouTube stats on when, where, and what people watch*. [online] Available at: <https://www.thinkwithgoogle.com/data-collections/youtube-stats-video-consumption-trends/> [Accessed 20 Mar. 2020].

Graf, M., 2020. *Which Java Microservice Framework Should You Choose in 2020?* [online] Available at: <https://medium.com/better-programming/which-java-microservice-framework-should-you-choose-in-2020-4e306a478e58> [Accessed 18 Dec. 2020].

Klubnikin, A., 2017. *Cross-platform vs Native Mobile App Development*. [online] Available at: <https://medium.com/all-technology-feeds/cross-platform-vs-native-mobile-app-development-choosing-the-right-dev-tools-for-your-app-project-47d0abafee81> [Accessed 18 Dec. 2020].

Maximillian Laumeister, 2019. *YouTube is Deleting Your Favorite Videos, And They Won't Say Why*. [online] Available at: <https://www.maxlaumeister.com/articles/youtube-is-deleting-your-favorite-videos/> [Accessed 20 Mar. 2020].

Niederwieser, P., Brünings, L. and TheSpockFrameworkTeam, 2020. *Spock Framework Reference Documentation*. [online] Available at: <http://spockframework.org/spock/docs/1.3/index.html> [Accessed 18 Dec. 2020].



10 Appendices

Appendix A	Project Description – ProjectDescription.pdf
Appendix B	Software Requirements Specification – SRS.pdf
Appendix C	Astah diagrams – Diagrams.zip
Appendix D	Survey results – Survey.pdf
Appendix E	Source code of Streamster – Streamster-source-code.zip
Appendix F	User guidelines – UserGuidelines.pdf