FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Video lectures indexing service
Student:	Bc. Jiří Zdvomka
Supervisor:	doc. Ing. Tomáš Vitvar, Ph.D.
Study program:	Informatics
Branch / specialization:	Web Engineering
Department:	Department of Software Engineering
Validity:	until the end of winter semester 2022/2023

Instructions

One of the positive points of the COVID era was the mass production of materials from online education. The goal of the diploma project is to develop a service that will be used to index video lectures with help of various supplementary resources (e.g. slides, TOCs, manual input, etc.). The below steps describe the project's methodology:

1. Describe the current state of the art in the area of video indexing including existing tools, technologies, services and algorithms.
2. Design the service to index video from lectures with selected tools and algorithms. Use AM1/AM2 online materials and discuss a general approach to index any lecture video under certain constraints or limitations. The service should provide both API and UI that can be used to manage the indexing process and deal with various anomalies in a (semi) automated way.
3. Develop the service in a selected technology.
4. Deploy the service as a container in a selected environment and perform the testing and evaluation.

Electronically approved by Ing. Jaroslav Kuchař, Ph.D. on 31 May 2021 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Video lectures indexing service

Bc. Jiří Zdvomka

Department of Web Engineering
Supervisor: doc. Ing. Tomáš Vitvar, Ph.D.

January 4, 2022

Acknowledgements

I want to thank my supervisor, doc. Ing. Tomáš Vitvar, PhD, for meaningful feedback and insights at consultations. I also want to thank my partner and parents for their support in my education.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46 (6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work for nonprofit purposes only, in any way that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on January 4, 2022 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Zdvomka, Jiří. *Video lectures indexing service.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Abstrakt

Indexování studijních video materiálů, které není tak časté, může vést ke zkvalitnění výuky. Tato práce se zaměřuje na implementaci webové služby pro indexování video přednášek s pomocí podpůrných materiálů a cílí na předměty vyučované na FIT ČVUT. Výsledkem práce je nasazená webová služba pro indexování video přednášek s uspokojivou přesností. Webová služba poskytuje API i uživatelské rozhraní a modul s algoritmem jako Python balíček.

Klíčová slova video, přednáška, index, web, služba, kontejner

Abstract

Video learning material indexes are rare and could lead to a better studying experience for students. This thesis aims to implement a web service for video lecture indexing with the help of supplementary materials as user input with focus on FIT CTU courses. The result is a deployed web service capable of video lecture indexing with satisfactory accuracy on the testing dataset, providing API, UI and core module with algorithm as a Python package.

Keywords video, lecture, indexing, web, service, container

Contents

List of Figures

List of Tables

Introduction

The COVID pandemic has led to the mass production of learning materials. Especially lectures have had to be presented online via video calls. While text-based learning materials are commonly indexed, and there are well-known methods for indexing and retrieval, video indexing is rare. The ability to search terms in videos and directly skip to the relevant part in a video could lead to students' better understanding of the topic, saving them time manually finding it. There have been attempts to create generic video indexing platforms, and they are studied in this work, but few of them specialise directly on the recently produced video lectures with presented slides. Such specialisation, as well as supplementary user input, could lead to video indexing with good performance. Furthermore, providing the video indexing and retrieval capabilities through a web-based user interface could make it more accessible and adopted by both teachers and students, not only on the Faculty of Information Technologies of CTU.

The goal of this master thesis is to design and implement a web service for video lecture indexing based on prior research in the field, providing both API and UI. It should specialise on available video lectures from AM1 and AM2 courses. The service user can provide additional input like text content area in the video or table of contents for higher accuracy. Technical-wise, the platform should follow Service-oriented architecture and be deployed into a cloud environment as application containers. The system should be modular and extensible.

The thesis is organised as follows. The first chapter (State of the art) analyses the current research in video indexing and describes a few selected existing methods. The second chapter (Algorithm) presents the designed algorithm for video indexing based on the findings from the analysis. The third chapter (Requirements) defines requirements for the system. The fourth chapter (Design) presents a high-level design of the system, its components and their responsibilities. The fifth chapter (Implementation) describes the system's low-level design and implementation details, including selected technologies. It also

presents the UI as a client web application. The sixth chapter (Verification) measures the core algorithm's precision and time performance on the test dataset of video materials. It also presents results from automatic end-to-end testing of the web platform. The last seventh chapter (Deployment) describes running the system's components as application containers and deployment in a cloud environment.

Chapter 1

State of the art

This chapter introduces video indexing and commonly used methods specialized for video lectures. Related works with various approaches ranging from Optical Character Recognition to document clustering or collaborative filtering are studied, including a production proprietary solution.

1.1 Video Indexing

Video indexing is a process of adding a retrieval capability over a collection of videos by tagging and organizing them [1]. Conventionally, this is achieved with the help of video metadata like title, description or categories, which are usually annotated **manually**. For example, on large video platforms like YouTube, users upload their videos, annotate them, and other users search for videos based on the annotations.

On the contrary, *content-based video indexing* creates indices by automated processing of the actual video content [1], enabling detailed retrieval capacity more appropriate for the context of video lectures. Users can search for a specific topic and retrieve only the relevant part of a video. These methods are based on text, audio, colours, shapes, objects or scene transitions [2]. This work focuses on video lectures, where presented slides or notes partially or fully occupy the whole video scene. More appropriate methods for the video lectures domain are text-based, audio-based methods or a combination of both.

Based on an analysis of literature overview [2] and research papers on the topic [3, 4, 5], which are studied in 1.5, common steps for state-of-the-art methods for lecture video indexing are:

1. segmentation and key-frame detection,
2. information extraction by text or sound,
3. indexing of the extracted information.

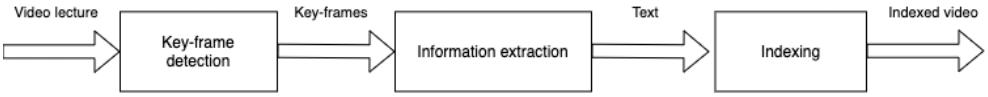

Figure 1.1: A common video indexing pipeline

1.2 Key-frame detection

A video is a combination of audio and visual presentation of content. In order to analyse the visual part and extract information from it, segmenting the video into individual frames is the first step. For performance reasons, a reasonable rate is one frame every one to three seconds [3]. A *key-frame* is a sequence of the same or almost the same frames. In the context of lecture videos, it is usually a slide. In order to effectively analyse the visual content, key-frame detection is essential. The studied works use various approaches described in section 1.5.

1.3 Information extraction

As lecture videos usually contain lecture slides or notes, the information is extracted from the text. The majority of analysed works use Optical Character Recognition (OCR), a process of converting input text into a machine-readable format [6]. The quality of the converted text is crucial. Therefore other support methods are applied, such as Merged Bounding Boxes detection [4] or image preprocessing like resizing, sharpening or blurring [7].

A different approach for information extraction from a video is from the spoken word. As well as optical frames, the audiotrack is partitioned into segments, which are then converted to a transcript with Automatic Speech Recognition (ASR) [4]. A contemporary example of such a service is Google Speech-to-Text [1].

1.4 Indexing

An output from the information extraction discussed in 1.3 is text. Therefore, classic approaches for text document indexing are appropriate. Representing the text content of slides as Bag of Words [8] and calculating Term Frequency Inverse Document Frequency (TF-IDF) score, reflecting the importance of a term to a document [9], is an example of such method. The document is either a video or a single key-frame, and document terms are extracted from the video content. An advantage of indexing individual key-frames is

[1] https://cloud.google.com/speech-to-text

granularity. A search query can be responded with an exact time occurrence of the search term in a video.

1.5 Related work

In [3], Yang and Meinel presented a content-based lecture video retrieval framework using speech and video text information. Compared to older works (e.g., [10]) using pixel-level-differencing metrics for key-frame detection vulnerable to noise, they focused on a more robust solution dealing with slide animations and build-ups. They segmented videos using differencing metrics with Connected Components (CCs). For example, those are text lines, figures or tables. Combined with other support methods like Support Vector Machine classifier (SVM) or image intensity histogram features, they achieved good results even for videos with scene switching (e.g. slide and speaker scene). In the second stage, they detect text areas in the key-frames and use a standard OCR engine to extract the frame's text. Afterwards, they applied spell check to filter poorly recognised words. In the third stage, they constructed the key-frame outline with a premise that the slide's title and subtitle are more significant than the slide text. They used text stroke width and geometry analysis to classify titles and subtitles. The authors applied Automatic Speech Recognition (ASR) to construct a speech transcript as a complementary method using a custom acoustic model trained on a video lectures dataset. However, the word error rate of ASR is generally still high, around 40 - 80 %. The OCR and ASR extracted keywords were combined by aligning them in an appropriate timestamp in the video and computing the TFIDF score with different weights for OCR and ASR as ASR keywords are less accurate.

Medida and Ramani [4] introduced a content-based indexing framework based on audio and video as well, but with a different approach for individual steps of the indexing pipeline. For text content on a key-frame, they detect merged bounding boxes and apply OCR for each one. For the key-frame detection, the authors calculate image hashes and filter similar frames with the hash difference. They partition the audio content into segments and use Google Speech Recognition technology for extracting the transcript. At this point, the authors perform summarization and keyword extraction from the whole video lecture document. Furthermore, the document is assigned a category based on Naive Bayes classifier modelled on a prepared dataset. With categories for each document, the dataset is clustered with K-means method to reduce the search time. When searching, a user either assigns one of the predefined categories to the query manually, or the classifier automatically assigns the category. Then classic TFIDF method with Cosine Similarity is used, but only within the category cluster. The authors achieved relevancy 95% with an average search time of 1.42 seconds on their dataset.

A more generic approach focused not only on lecture videos was introduced by Wang et al. [5] as inVideo platform. They do not list concrete implemented methods and only state it uses artificial intelligence and machine learning. Keyword search from audio analysis, image references and object recognition is incorporated as well as image search. The platform supports multilanguage videos. It is designed for usage in a cloud with decentralized indexing and search and thus scales well. Collaborative filtering, a process leveraging user feedback for improved accuracy, is a unique feature of the platform. Users can comment on videos, tag keywords and even play in-place quizzes. The authors state that this interactive enhancement improved the accuracy of video indexing search and increased an average grade of a university course for which the system was deployed.

Last but not least, a representative of proprietary software, Microsoft Video Analyzer [2] is a complex cloud-based tool for extracting insights from video built upon artificial intelligence. Among 30+ features listed on the tool's website, it can extract timestamped keywords, identify topics, classify named entities or even detect emotions from video. The service offers to process up to 10 hours of video content for free. Otherwise, the pricing is based on the duration of the input file. The exact numbers are $0.04 / minute for audio analysis and $0.15 / minute for video analysis, approximately $18 for a 1.5-hour video. The service provides REST API for usage in third-party applications. To explore the tool, I indexed [11] video lecture. The result was 12 topics and 30 keywords indexed in 36 minutes, which does not include all lecture content as some slide titles were not indexed.

[2] https://vi.microsoft.com/

Chapter 2

Algorithm

This chapter describes the algorithm for video lecture indexing used in this work, including auxiliary user input like text content area or table of contents as a support material. The indexing process follows the common steps described in section 1.1 - key-frame detection, information extraction and indexing. The input to the algorithm is a video file with presented lecture slides. The output is a list of lecture topics identified by a timestamp of occurrence in the video, a title and text content. An indexing engine further indexes the list with TFIDF method. The chapter is closed with a discussion of algorithm constraints and possible improvements to index any video lecture.

2.1 Key-frame detection

The first step in the key-frame detection is to convert a video file into frames and associate a time of occurrence in the video with the frame. A reasonable interval for one frame in video lectures is one to three seconds, as a lecture slide is usually presented for a few minutes. A shorter interval would be more accurate in detecting the exact timestamp of a lecture topic, which also means more frames, thus more processing time. A deviation of a few seconds is acceptable. The most suitable value for this parameter based on measurements is evaluated in section 6.2 of this work.

Now that we have a list of image frames associated with the timestamp of appearance, the next step is to filter the same frames with image similarity methods. There are many ways to compare images based on individual pixels, histograms [12], SIFT descriptors [13] or convolutional neural networks [14]. However, I chose a perceptual image hashing method, which generates a fingerprint of the image. Compared to classic hash functions like SHA-256 it has the property of *perceptual robustness*, which means that perceptually identical images should have similar fingerprints [15]. Perceptual hashing is prone to scaling, aspect ratios or colour differences.

The frame filtering algorithm 1 iterates over a list of all video frames. It fingerprints each frame as well as the successive one and calculates normalized Levenshtein similarity between them. If the similarity is greater or equal to a defined threshold (e.g. 90%), the algorithm discards it from the list. At the end of the loop, the list contains only distinct frames. However, it can happen that two frames of the same lecture slide still pass through this filter either because of hashing function error or inadequate threshold value. The next step, text extraction, handles these anomalies.

Algorithm 1 Similar frames filter

```
L ← list of video frames
t ← similarity threshold
for c ← current frame of L do
    n ← the successive frame of c
    h_c ← fingerprint of c
    h_n ← fingerprint of h
    if normalized Levenhstein similarity of h_c and h_n ≥ t then
        remove c from L
    end if
end for
L contains all distinct frames
```

2.2 Text extraction

A video lecture scene usually consists of an area with the teacher written notes on a whiteboard and lecture slides. Sometimes, it contains teacher scratches on a tablet in an online lecture. Figure 2.1 is an example of such scene. The lecture slide area is important for the algorithm as it contains the presented content as a text. The algorithm does not use speech as the source of information and does not work for lectures without slides.

It is wise to preprocess the key-frames prior to detecting text and applying OCR for better accuracy. Two basic preprocessing techniques are applied - *binarization* and *noise removal*.

Binarization means converting a coloured frame into grayscale, which can be achieved by establishing a colour threshold value from 0 to 255 and all pixels greater than the threshold are converted to black and less the threshold to white. *Adaptive thresholding* improves the binarization by establishing different thresholds for smaller parts of the whole image [16].

Noise removal should smoothen the image and remove small dots and patches. A *median filter* method is applied in the preprocessing phase after binarization [17]. An example of these transformations applied on a video lecture frame is depicted in figure 2.2

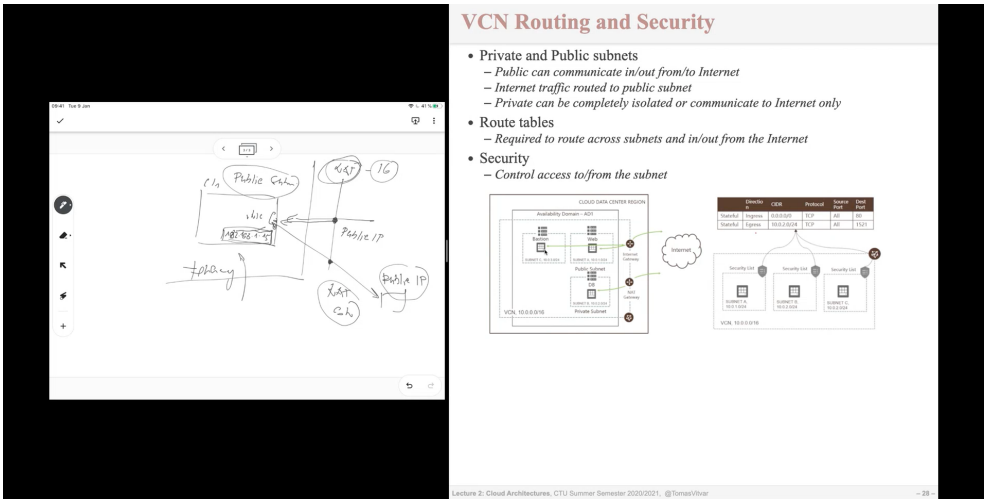

Figure 2.1: An example of a video lecture scene consisting of a slide and presenter scratches (snapshoted from a recording of [11])

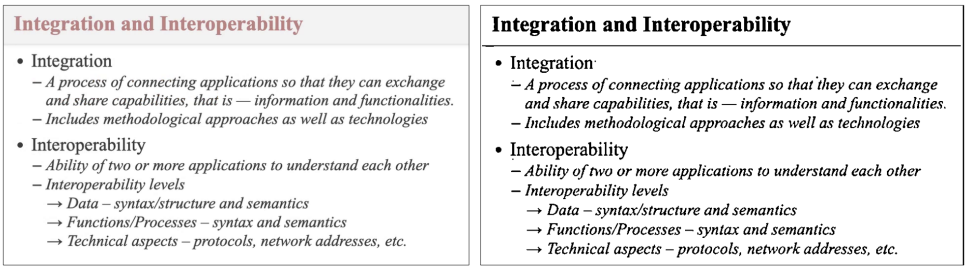

Figure 2.2: An example of an original slide (on the left) and the binarised version with noise removal (on the right)

Finally, with the preprocessed key-frame, the algorithm applies OCR on the key-frame and filters out newlines from the extracted text. We are interested in extracting the slide's title as well as the rest of text content. The title is usually located at the top of the slide, so we take only the first line from the extracted text. The algorithm also compares the title with a title from the previous key-frame to double-check that they are indeed different, addressing issues with some false positive key-frames mentioned in 2.1. This phase's output is final key-frames, extracted slide titles and text, ready for indexing and providing full-text search. An example of such output in JSON format is listed in 2.1.

```
[
  { "second": 4, "title": "Overview" },
  { "second": 62, "title": "Compute Instances" },
  { "second": 212, "title": "Image" },
  { "second": 342, "title": "Autoscaling" },
  { "second": 598, "title": "Load Balancer" },
  { "second": 1752, "title": "Overview" },
  { "second": 1768, "title": "Object Storage" }
]
```

Listing 2.1: Example algorith output without the slide text content

2.3 Indexing

Designing a complete TFIDF indexing process is outside of the scope of this work. Therefore, an existing indexing solution capable of indexing classic text documents as a Bag of Words is used. In this context, a document is a collection of key-frames from a lecture. Each key-frame's text content in the lecture is indexed and associated with a timestamp, so we have information when the key-frame is retrieved for a search query. A user of the index can search for any topic that appeared in lecture slide titles or text and directly jump to the moment of the lecture when it is presented. Skipping directly to the exact moment when a topic is presented is the most significant benefit of this work.

2.4 User input

The slide text content extraction approach does not handle all cases well. Sometimes, there can be unmeaningful characters or words on lines preceding the title. Typically, this can be a header of the slide or a system time on a presenting device. A possible solution is Natural Language Processing and text correction. That can filter out the corrupted content and fix poorly converted words. Instead of increasing processing complexity with such techniques, I designed two methods for handling such situations with the help of complementary user input - a location of text content in the video and a lecture's table of contents.

2.4.1 Text content area

If we only care for the text content of a slide, not the whole scene, and OCR output can be noisy, what if we let a user choose the content area on the scene, defined by origin coordinates (x, y), width and height? The content area is usually static throughout the lecture. For example in figure 2.1 the text

occupies only half of the scene. If we are not interested in all content but only the title, the area is even smaller.

The input can be applied in the first phase of the algorithm when the video is converted to frames. The frames can be cropped only to the input area. That decreases frame size, thus improving the performance of computing image hashes, reducing noise in the image and increasing the precision of OCR.

2.4.2 Table of contents

Another valuable user input is a table of contents (ToC), more precisely a sequence of topics appearing in slides, which serves as a list of anticipated titles. With ToC, we know successive slides at any moment of the video and compare a current extracted text content with it. If it is similar to the expected title, we have identified a new key-frame and can use text from the ToC instead of the OCR output. This approach's advantage is that we can skip some slides we do not want to have in the index, such as an overview slide or a part of the lecture, when a presenter switches to a command line window.

However, there could be an issue as well. If the algorithm is processing a frame and there is a problem with extracting the title, it can happen that the frame will be skipped entirely. At that point, the algorithm expects the already passed slide from the ToC, which will never be there. Comparing a current frame's title with not only one but two or more anticipated titles may help to solve the issue.

The situation may be more clear from an example. Let A, B be two frames with different content and T_A, T_B the corresponding slide titles from a ToC. Processing of frame A fails (e.g. the OCR accuracy is poor), but B succeeds. The algorithm compares content extracted from A with T_A and decides the content is not similar to T_A. It continues and compares content extracted from B with T_A, which is not similar either and therefore, the algorithm will continue comparing all successive frames with T_A resulting in a corrupted index. However, if the algorithm compares B with T_A and then T_B, it finds a match with T_B, marks T_B as processed, and continues with successive titles from ToC.

In reality, a lecturer does not always present a lecture from start to end but can finish one lecture and start another halfway. That can cause problems if we have a way to automatically provide ToC to the tool for a whole lecture. For such lectures, the ToC would have to be adjusted manually or not used at all.

2.5 Generic approach

This work is constrained to video lectures from AM1 and AM2 coursers on FIT CTU, which have materials available in English. The algorithm is capable of indexing video lectures not only from these courses but any course with

English slides presented on a screen. In order to index other languages, their support would have to be configured in the OCR engine, as it usually supports only English by default.

The algorithm's content extraction quality depends on the presented text quality and OCR engine performance. If a video is of low quality or there are handwritten notes on a whiteboard instead of slides, the algorithm does not perform well. What could also help to improve the extraction performance is post OCR text correction with a dictionary.

Furthermore, what this work is not focused on is Automatic Speech Recognition. Adding it to the algorithm would also generalize its usage and result in more complex indexed content. Usually, important information is included in the verbal content as well as in the text content. However, ASR would not work very well on AM1 / AM2 lectures because their verbal content is presented in Czech, whereas text content is in English.

Chapter 3

Requirements

A goal of this work is to introduce a web service for indexing video lectures with the help of various supplementary resources. The service should implement the algorithm described in chapter 2 and provide both API and UI. This chapter defines the functional and nonfunctional requirements of the service as well as its actors.

3.1 Actors

There are two actors in the system based on a user role. A user represents any of these two actors.

1. **Teacher** - has access to video lectures, is eligible to index and publish them in the system.
2. **Student** - can view published video indexes and search them. The user with role student **is not** authorized to publish videos.

3.2 Functional requirements

F1: A user can sign in to the system with his FIT CTU identity.

F2: A user can sign out of the system.

F3: A teacher can upload a video lecture file and index it. He can attach supplementary resources like a table of contents and slide content area in the video scene.

F4: A teacher can display a video processing progress.

F5: A teacher can download the video index in JSON format.

F6: A teacher can display a list of his video indices.

F7: A teacher can update the title or indexed topics of his uploaded video index.

F8: A teacher can remove an uploaded video index.

F9: A user can display video indices published to the platform in chronological order.

F10: A user can play a published video lecture, display indexed topics and skip the video to a specific time when a topic is discussed.

F11: A user can search published videos by a name or topic present within the video. The results are ordered by relevance to the query.

3.3 Non-functional requirements

NF1: The system is implemented as a web platform with frontend and backend.

NF2: The system uses FIT CTU OAuth 2.0 server [3] for authentication.

NF3: The system's architecture is designed as service-oriented architecture (SOA) [18].

NF4: The system must be easily extensible, meaning the core algorithm is portable and independent on the rest of the system.

NF5: The system's services run in application containers with an orchestrating system.

NF6: The system is deployed to a virtual machine.

[3] https://rozvoj.fit.cvut.cz/Main/oauth2

Chapter 4

Design

This chapter describes the high-level design of the system. The indexing web service consists of three main components - an indexing module, a backend and a frontend. The indexing module is an implementation of the algorithm described in chapter 2. The frontend is a client web application. The backend provides REST API is decomposed into multiple isolated components, including tiny services, databases and proxy servers.

The system's architecture follows service-oriented architecture (SOA) principles such as loose coupling, encapsulation, or contracting. A single service in the system only solves a limited set of operations, usually around a single resource [18]. For instance, Video Service only handles uploading, removing and serving of videos, nothing more. The services exchange information via HTTP protocol or asynchronous messaging.

Figure 4.1 depicts the overall system architecture. There are four services - *UserService*, *VideoService*, *IndexService* and *JobService*. The only component accessible from the outside is API Gateway, which routes clients' HTTP requests to the receiving services. Furthermore, it performs authentication with User Service before forwarding them on. There are multiple types of data storage - a document database for storing indices, disk storage for storing video files, a key-value database for caching and a full-text index engine providing search capability over lecture topics.

4.1 Indexing module

The indexing module's responsibility is to run the video indexing process described in chapter 2. The module is independent of the rest of the services. It provides both API and command-line interface (CLI).

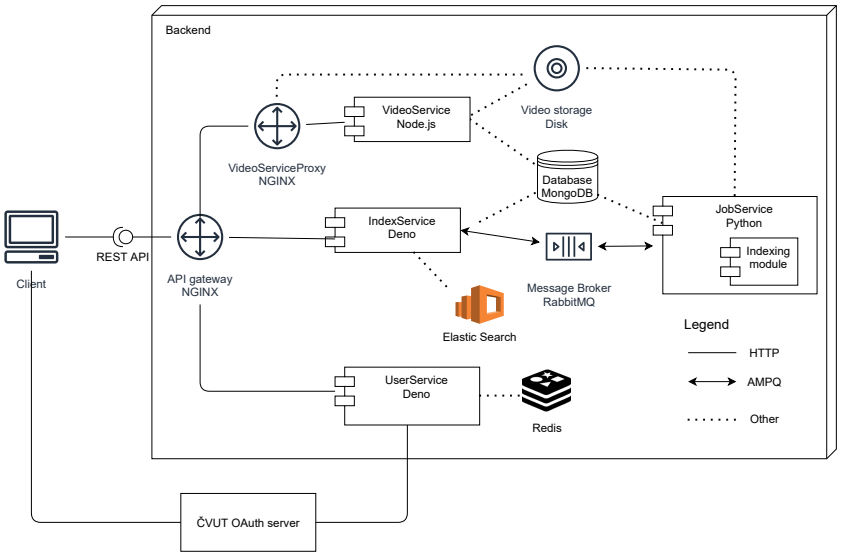

Figure 4.1: Architecture diagram of the indexing service components

4.2 Backend

4.2.1 API Gateway

The API Gateway serves as the only entry point to the REST API. It has two purposes. First, it routes incoming requests to destination services [19]. Second, it performs authentication with User Service before forwarding the requests. Details are described in section 4.2.2. Apart from that, it configures CORS policy [20].

4.2.2 User Service

The primary responsibility of User Service is authentication of requests incoming to protected API endpoints in cooperation with the API gateway. The secondary responsibility is providing an endpoint for the user resource.

Figure 4.2 illustrates the complete authentication flow for an incoming request.

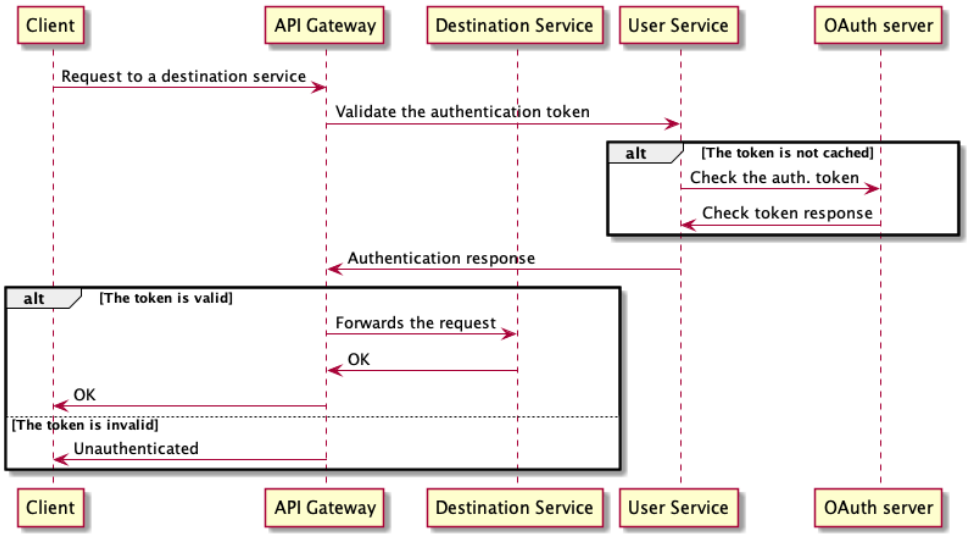

Figure 4.2: A sequence diagram illustrating the authentication flow for an incoming request

1. The entry point of the request is the API gateway.

2. The API gateway sends an authentication subrequest to User Service with a request's access token as its payload before routing it to the destination service [21].

3. User Service validates the received access token with the school's OAuth 2.0 [4] server [22]. There is a caching mechanism to avoid a high number of requests to the OAuth server. The service first checks the access token with the cache and only sends a request to the OAuth server if there is no cache hit.

4. If the access token is not valid, User Service notifies the API gateway that the access token is invalid, which forwards it to the client. If the access token is valid, User Service issues a new JSON Web Token (JWT) containing user info and a role as its payload. The JWT is used for inter-system authentication and authorization by other services. The main advantage of JWT is the complexity of token verification. There is no need to make an additional request to the OAuth server, as each service can verify the JWT with a private key [23]. The service replaces the received token with the new JWT and responds to the API gateway.

[4] https://rozvoj.fit.cvut.cz/Main/oauth2

5. If the original request is authenticated, the API gateway finally routes it to the destination service, which authorizes the request based on the user role in the JWT payload and responds.

6. If the original request is not authenticated, the API gateway directly responds to the client with `Unauthenticated` status.

4.2.3 Video Service

The Video Service's responsibility is to handle video files upload, store the files on a disk, save video info in the database, serve videos to clients and delete them. In the architecture diagram 4.1, a proxy server is placed before the service to handle file uploads and video serving efficiently.

4.2.4 Index Service

Index Service provides CRUD operations API for the video index resource as well as the index search endpoint. It also provides an endpoint for reading progress of the video indexing process, called a job. Compared to the other services, this one does quite a lot of work in the background.

For inter-system communication, the service uses asynchronous messaging pattern. Asynchronous messages are unidirectional, meaning a sender does not expect a direct response from a receiver. In fact, it does not have to know the receiver at all, or there may be multiple receivers [24]. Asynchronous messaging makes Index Service loosely coupled because the service does not care which service or an instance of service indexes the video. Figure 4.3 illustrates the message flow of Index Service.

1. When creating a new index, the service sends an asynchronous message to the message queue to start indexing the video.

2. The service also subscribes to messages about the indexing progress so that it can forward it to clients.

3. When the service receives an indexing completed message, it indexes the textual content of each slide in the video with TF-IDF along with a timestamp of occurrence in the presentation.

4.2.5 Job Service

Job Service's responsibility is to run video indexing jobs. It subscribes to index video messages emitted to the message queue and starts processing the video for each message. It monitors the progress of the job and publishes messages with the progress information. After the job finishes, it saves the newly created index into the database and publishes a success message. In case of an error,

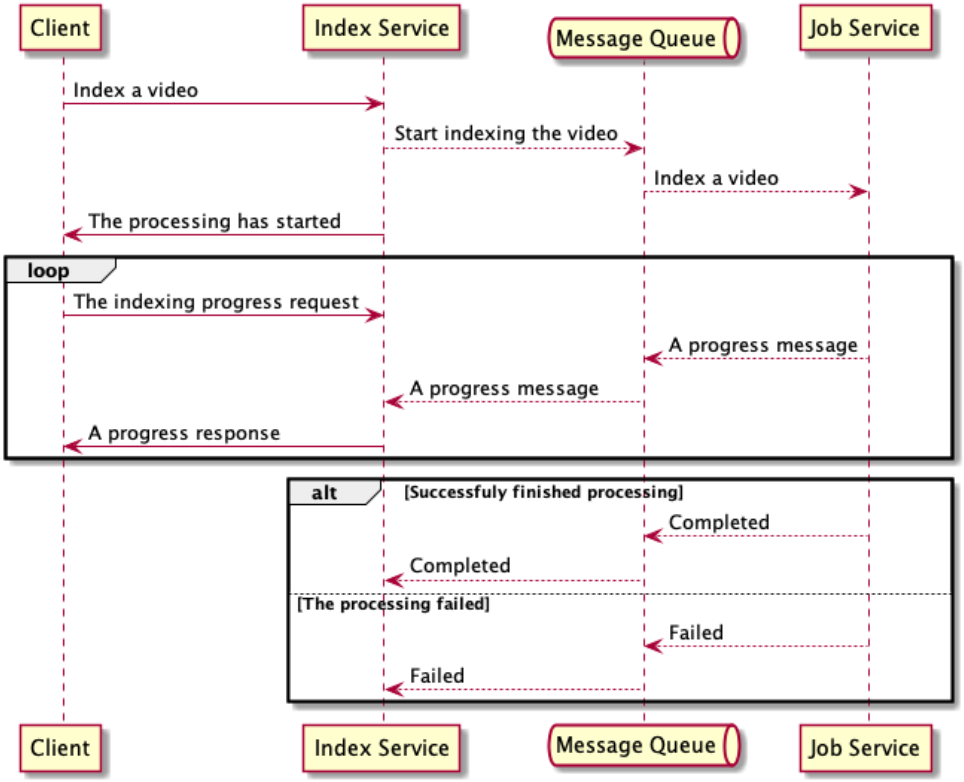

Figure 4.3: A sequence diagram illustrating the flow of messages between Index Service, message queue and Job Service

it emits an error message and retries the job with exponential backoff. Figure 4.3 illustrates the message flow.

4.3 Frontend

The frontend is an isomorphic web application, which provides user interface for the system. An isomorphic web application is a combination of statically rendered Single Page Application (SPA) and server-side code [25].

The reason for choosing both server and client-side code for the frontend is the authentication. FIT CTU OAuth server only supports Code Authentication Grant [22] with a client secret. Storing the secret in a client-side JavaScript application is a security risk as anyone can see the secret in the source code and disguise as our application. However, the server can securely store it. The proper OAuth flow for client-side JavaScript applications is Authorization Code Flow with Proof Key for Code Exchange (PKCE), which does not operate with a client secret [26]. Apart from authentication,

there is no reason for server-side code, since there are no public pages requiring search engine optimizations.

Chapter 5

Implementation

This chapter describes the low-level design and implementation of the system components in selected technologies and some of the implementation details. As mentioned in chapter 4, there are three main components in the system - the indexing module, backend and frontend.

5.1 Indexing module

Indexing module is a Python package implementing the algorithm described in chapter 2. Python programming language has been chosen because the Python ecosystem provides a rich set of libraries required to implement the algorithm. Used libraries are listed in table 5.1 The source code is stored in a public Github repository [5] or can be found at `src/fit-lecture-indexer` directory of the enclosed memory card. The package is also published into Python Package Index [6].

5.1.1 API

The module exports `LectureVideoIndexer` class, which can be used for indexing videos in any Python program. Code 5.1 presents the interface and code 5.2 usage example. The class constructor accepts two parameters - `config` and `progress_callback`. The first parameter is an object with settings for the algorithm. The second is a function that is called every time video progress changes, which is useful for displaying indexing progress. Parameters of the configuration object are listed in table 5.2.

The class has `index` method which can index a video lecture. It has the following parameters:

- `video_path` - path to a video file

[5] https://github.com/jstorm31/fit-lecture-indexer
[6] https://pypi.org/project/fit-lecture-indexer/

Library name	Description
imagehash	Perceptual hashing library
strsimpy	String comparison package implementing normalised Levenshtein distance used as a similarity measure
opencv-python-headless	OpenCV python library used for video conversion to frames and image preprocessing before application of OCR
pytesseract	Python port of tesseract binary for OCR
tqdm	A package for displaying processing progress in a command line

Table 5.1: Table of libraries used in Indexing module

Parameter	Description
`frame_step`	create a frame every x seconds of the video
`img_sim_threshold`	a threshold when two images are considered similar
`txt_sim_threshold`	a threshold when two strings are considered similar
`hash_size`	number of bytes for image *phash* x seconds of the video

Table 5.2: Algorithm parameters

- `skip_converting` - skips conerting the video into frames (useful when running the method multiple times or the conversion is executed externally)

- `crop_region` - a crop region for frames a slide content is expected to be located, a tuple in format `(x_from, x_to, y_from, y_to)`

- `toc` - a path to a table of contents file in JSON format with an array of slide titles

```
LectureVideoIndexer(config: Config, progress_callback:
    ProgressCallback)

  index(video_path: os.PathLike, skip_converting: bool,
    crop_region: CropRegion, toc: TableOfContents = None) ->
    VideoIndex
```

Listing 5.1: LectureVideoIndexer class interface

```
from indexer import LectureVideoIndexer, CropRegion

indexer = LectureVideoIndexer()
index = indexer.index(video_path='video/example.mp4', crop_region
    =CropRegion(890, 1700, 0, 80), toc='toc.json')
```

Listing 5.2: Example usage of Indexing module

5.1.2 CLI

The package also provides a command-line interface. It can index a video by running `python cli.py -i example.mp4` command in the package's directory. Availabel parameters can be printed by running `python cli.py -h`.

5.2 Backend

5.2.1 Architecture

The backend consists of multiple individual web services that communicate through HTTP protocol or asynchronous messages via RabbitMQ message broker [7]. The services also provide RESTful API [27]. Apart from JobService, which is specific, all backend services follow two-layer architecture, which consists of *controller* and *service* layer [8].

The *controller* layer handles HTTP request-response communication, provides routes for resources, connects them with services and converts errors to correct response status codes. Code 5.3 shows an example of controller function to return index detail. The *service* layer contains business logic and directly communicates with databases. Code 5.4 shows an example of a service function that queries a database for indices. There is usually also a *data* layer, often following *Repository Pattern*, which is an abstraction for accessing databases [28]. However, since the backend services are not very complex, they access databases directly on the service layer.

[7] https://www.rabbitmq.com

[8] We must distinguish a web service as a high-level design system's component and service as a layer of architecture - code implementing some business logic.

```
import * as indexService from '../services/index.service.ts'

const getIndex = async (ctx: Context) => {
    const { id } = helpers.getQuery(ctx, { mergeParams: true })
    const index = await indexService.getIndex(id)

    if (!index) {
        ctx.response.status = 404
    } else {
        ctx.response.status = 200
        ctx.response.body = index
    }
}

router.get('/indices/:id', getIndex)
```

Listing 5.3: An example of a route handler on *controller* layer

```
import { Index } from '../db.ts'

export const getIndices = async (params: GetIndicesParams = {})
    => {
    const options: Record<string, unknown> = {}
    const page = params.page ?? 1
    const limit = params.limit ?? 10

    if (params.user) {
        options.username = params.user
    }

    if (params.q) {
        options['video.name'] = { $regex: `${params.q}`, $options
    : 'i' }
    }

    const total = await Index.count(options)
    const indices = await Index.find(options)
        .sort({ _id: -1 })
        .limit(limit)
        .skip((page - 1) * limit)
        .toArray()

    return {
        data: indices,
        total,
    }
}
```

Listing 5.4: An example of a function on *service* layer querying a database for indices

Furthermore, on the *controller* layer, the app also uses *middleware* for sharing some common logic. A *middleware* is a function executed on every request. It receives the request and response entity, can modify them, and return to other *middleware* in the chain, ending with a controller [29]. Typical use cases for *middleware* are logging or authentication/authorization. Code 5.5 shows an example of a middleware that extracts JWT token from the request Authorization header and verifies it. If the token is valid, it extracts user information from the JWT payload, adds it to the HTTP context object and calls the next middleware. If the token is invalid, it directly returns 401 HTTP response status code and does not call the next middleware.

```
import * as jwtService from '../services/jwt.service.ts'
import { extractToken } from '../utils.ts'

export const authenticate = async (
    ctx: Context,
    next: () => Promise<unknown>
) => {
    const token = extractToken(ctx)

    if (!token) {
        ctx.response.status = 401
        ctx.response.body = { error: 'missing_token' }
        return
    }

    try {
        ctx.user = await jwtService.verifyToken(token)
        await next()
    } catch (error) {
        ctx.response.status = 401
        ctx.response.body = {
            error: 'invalid_token',
            message: error.message,
        }
    }
}
```

Listing 5.5: An example of authentication middleware on *controller* layer

All services apart from Job Service are implemented in TypeScript [9], a strongly typed programming language, which extends JavaScript and makes it safer with compile-time type checks. In some runtimes, it needs to be transpiled into JavaScript (browser or Node.js), whereas others support it directly (Deno).

As might be visible from the code examples, the programming style is functional, not object-oriented. It comes from the nature of JavaScript, where

[9] https://www.typescriptlang.org

functions are first-class, meaning they can be assigned to a variable, passed to a function or even returned by one (higher-order function) [30]. Furthermore, JavaScript modules allow encapsulation of variables and functions in a module and can be imported into other modules [31]. Therefore, the object-oriented paradigm is not necessary, although possible in both JavaScript and TypeScript.

5.2.2 API documentation

Documentation is an integral part of every HTTP API so consumers know which resources they can access, how to use the API or what errors it returns. OpenAPI is a widely used documentation standard for HTTP APIs [32]. It is programming-language agnostic, therefore, suitable for any API.

The OpenAPI version 3.1 specification for the platform's REST API in YAML format can be found at `src/fit-stream/docs` directory of the enclosed memory card. It is also published as an HTML page [10] powered by Swagger UI [11].

5.2.3 API Gateway

For the implementation of API gateway, I chose Nginx [12] HTTP server, which offers rich capabilities, high performance and simple configuration [33]. The configuration defines publicly accessible HTTP locations, which are forwarded to appropriate platform services by the Nginx server. Furthermore, the configuration sets CORS policy to allow requests only from known origins. Last but not least, it uses `http_auth_request_module` to authenticate incoming requests based on subrequest results from User Service [21]. Detailed flow of the authentication is presented in diagram 4.2. The source code is stored in a GIT repository and can be found at `src/fit-stream/api-gateway` directory of the enclosed memory card.

Code 5.6 shows an example configuration for `/auth/user` location. First, it sends an authentication subrequest to User Service with `Authorization` header from the original request and replaces its value with a newly issued JWT token obtained from the subrequest response header. Then it forwards the request to `http://user_service:8003` located on a private virtual network, keeping the original `Host` header and IP address.

5.2.4 User Service

The selected technology for User Service implementation is Deno, an asynchronous, event-drivend JavaScript runtime for server-side code [13] similar to

[10] http://10.38.5.10:81/docs/
[11] https://swagger.io/tools/swagger-ui/
[12] https://www.nginx.com/
[13] https://deno.land

```
location /auth/user {
  auth_request /auth/token/check;
  auth_request_set $authorization $upstream_http_authorization;
  proxy_set_header Authorization $authorization;

  proxy_pass http://user_service:8003;
  proxy_set_header Host $host;
  proxy_set_header X-Real-IP $remote_addr;
}
```

Listing 5.6: Nginx proxy configuration for a location

Node.js. The service uses in-memory Redis database [14] for caching of user authentication tokens and roles to improve performance by avoiding requests send to the OAuth server. The source code is stored in a GIT repository and can be found at `src/fit-stream/user-service` directory of the enclosed memory card.

5.2.4.1 API endpoints

- `GET /auth/user` - returns a user info object including roles
- `GET /auth/token/check` - non-public endpoint used by API gateway to check an OAuth authentication token and return an internal JWT token

5.2.4.2 Services

- `jwt.service.ts` - creates and verifies JWT tokens
- `oauth.service.ts` - communicates with the OAuth server, can check and refresh an OAuth authentication token
- `user.service.ts` - can fetch user info and user roles

5.2.5 Video Service

The selected technology for Video Service implementation is Node.js [15] because Deno does not have a stable library for parsing `multipart/form-data` content type of HTTP request at the time of writing. The source code is stored in a GIT repository and can be found at `src/fit-stream/video-service` directory of the enclosed memory card.

The proxy server placed before Video Service is an Nginx server same as the API gateway. The proxy server handles file uploads and forwards the requests to the Node.js server [34], which performs related business logic.

[14] https://redis.io
[15] https://nodejs.org

5.2.5.1 API endpoints

- `POST /videos` - upload a video
- `DELETE /videos/:id` - delete an uploaded video

5.2.5.2 AMQP messages

- `DeleteIndexMessage` (consumer) - delete an uploaded video (requests from within the system)

5.2.5.3 Services

- `videoService.ts` - creates a video database entry, generates image thumbnails for a video, deletes a video from the database

5.2.6 Index Service

Index Service is implemented in Deno and TypeScript, same as User Service. The source code is stored in a GIT repository and can be found at `src/fit-stream/index-service` directory of the enclosed memory card.

It stores data in two types of storage. Primary, indexed content of a video is stored in a MongoDB database along with video data [16] as a collection of denormalized documents. Storing indexed content and video data together in a document without strict schema allows flexibility [35, slide 48]. Secondary, Index Service uses Elastic Search [17] engine for indexing video content, including all text content of slides. Elastic Search provides full-text search, and for a given query, it can return video lectures sorted by relevance to the query. The search results include timestamps of a match for the query when it occurs in the lecture.

The service provides an endpoint for real-time progress of the indexing process implementing server-sent events (SSE). SSE is a technique for pushing messages from a server to client. The client makes only the initial request and then receives messages automatically [36]. While SSE from server to client directly works well, there can be issues [18] with proxies between server and client, which can buffer the connection, breaking the real-time delivery of the messages. Fortunately, Nginx in the role of API gateway can be configured not to buffer or cache the connection for the progress endpoint.

5.2.6.1 API endpoints

- `GET /indices` - fetch indices, supports pagination and filter by username

[16] https://www.mongodb.com
[17] https://www.elastic.co
[18] https://community.cloudflare.com/t/server-sent-events-buffering/179526

- `GET /indices/:id` - fetch index detail
- `POST /indices` - create a new index for an uploaded video and start the indexing process
- `PUT /indices/:id` - replace an index
- `DELETE /indices/:id` - delete an index
- `GET /indices/search` - full-text search indices for a given query
- `GET /indices/:id/progress` - provides real-time progress information about indexing progress for a video

5.2.6.2 AMQP messages

- `JobMessage` (producer) - starts video indexing process
- `JobProgressMessage` (consumer) - contains information about video indexing progress
- `JobCompletedMessage` (consumer) - signals a video indexing process has been finished
- `JobErrorMessage` (consumer) - reports a failed video indexing process

5.2.6.3 Services

- `index.service.ts` - provides CRUD operations over index resource
- `job.service.ts` - publishes `JobMessage` to start video indexing process, observes a job progress, updates Elastic Search index after a job finish
- `jwt.service.ts` - verifies JWT token
- `search.service.ts` - can index or delete a video document in Elastic Search and search video documents for a given full-text query

5.2.7 Job Service

The selected technology for Job Service is Python to be compatible with Indexing module, which is implemented as a Python package, and the service uses it. The source code is stored in a GIT repository and can be found at `src/fit-stream/job-service` directory of the enclosed memory card.

Job Service has a different architecture than the other services. It does not provide API endpoints for a client and only publishes or subscribes to asynchronous messages via the RabbitMQ broker. Therefore it does not have controller and service layer. The service is implemented using Dramatiq Python package [19] for background task processing. It only contains a so-called Actor,

[19] https://dramatiq.io/

which is a function executed on a worker process [37]. The Actor subscribes to start indexing messages, runs the video indexing process via Indexing module and publishes a message for completed or error state. Dramatiq library handles job scheduling, retrying and error handling.

The service saves the created index directly in the Mongo database and does not send it in a message payload. That creates tight coupling with Index Service indirectly through the database. It does not have to be a good practice on a large scale, but it is generally not advised to send large payloads of data through messages [24].

5.3 Frontend

The frontend is implemented in TypeScript and Next.js framework [20], which supports hybrid static and server rendering. Next.js is based on React [21], a JavaScript library for building user interfaces. The source code is stored in a GIT repository and can be found at `src/fit-stream/frontend` directory of the enclosed memory card.

5.3.1 Architecture

React is a component-based library. A *component* is a simple JavaScript / TypeScript function that returns a part of a user interface (UI) for input properties and an internal state of the component [38]. Components can be composed together into complex UIs. Code 5.7 presents an example of `Loader` component.

Next.js framework supports file-based routing, meaning each file in `pages` project's directory exporting a React component is an individual web page [39]. The frontend follows modular architecture. A *module* is a part of the application focusing on a complete isolated functionality. For instance, a list of videos can be implemented as a module called `index-list` and contain a set of React components, CSS styles and business logic for fetching data from the backend REST API. The module exports one component that is rendered in a page file. The modular architecture encourages loose coupling between modules.

The frontend uses several libraries:

- Next Auth - authentication library supporting OAuth 2.0,
- SWR - fetching data from the backend REST API and caching,
- Ant Design - a design system UI library,
- React Player - a video player library for React,

[20] https://nextjs.org
[21] https://reactjs.org

- React Region Select - a library for selecting a rectangular area for React.

```
import { Spin, SpinProps } from 'antd';

import styles from './Loader.module.scss';

export interface LoaderProps extends SpinProps {
    loading?: boolean;
}

const Loader = ({ loading = false, ...props }: LoaderProps) =>
    loading ? (
        <div className={styles.loader}>
            <Spin {...props} />
        </div>
    ) : null;

export default Loader
```

Listing 5.7: An example of React Loader component

5.3.2 User Interface

This section describes the front-end web application's user interface, including screenshots.

5.3.2.1 Login

There is nothing complex on the login page, just a *login* button that starts the OAuth flow redirects the user to the OAuth provider page.

5.3.2.2 Videos

Video page contains a grid of indexed video previews. Each preview contains a thumbnail from the video and a title. By clicking on a video preview, the user navigates to Video detail page. Initially, twelve videos are loaded, and the user can load more with a button at the bottom of the page. Above the grid, there is a search text field, where a user can search any query that will match the content of indexed videos. At the top in the navigation bar, the user can go to Index video page, User Videos page or sign out.

5.3.2.3 Index video

The index video page contains a file upload area where a user can drag and drop a video file, which gets uploaded. The application starts uploading the video directly after it has been dropped, signalling upload progress by

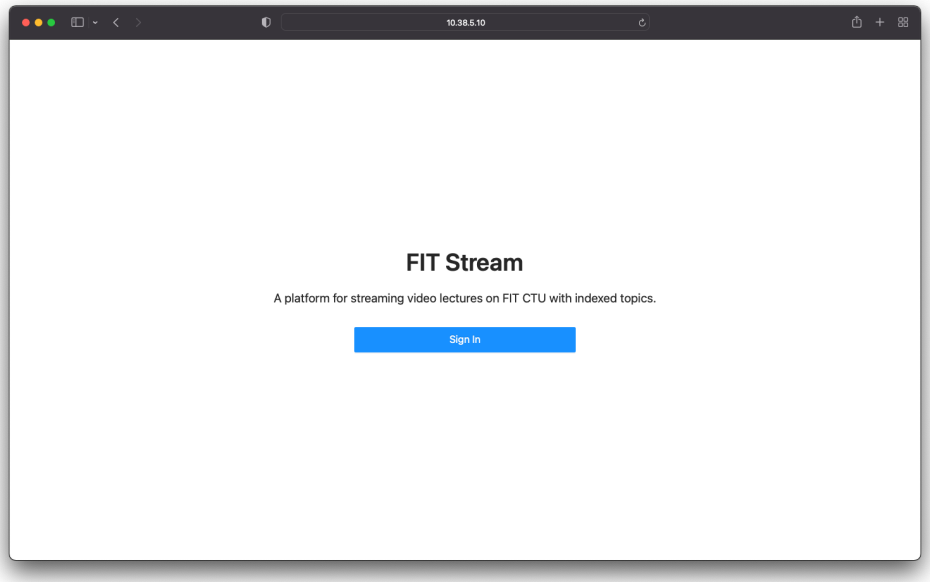

Figure 5.1: Login page

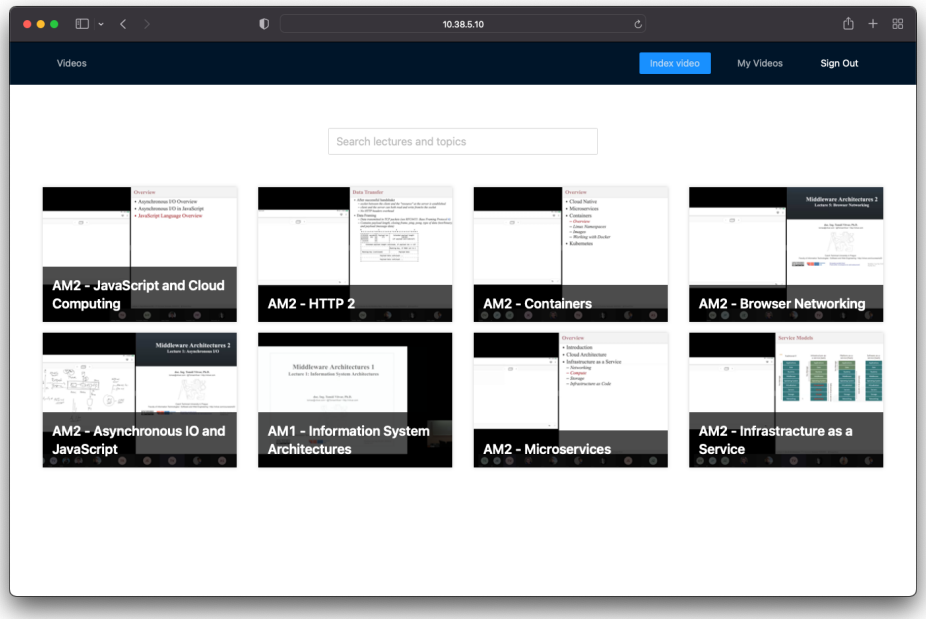

Figure 5.2: Videos page

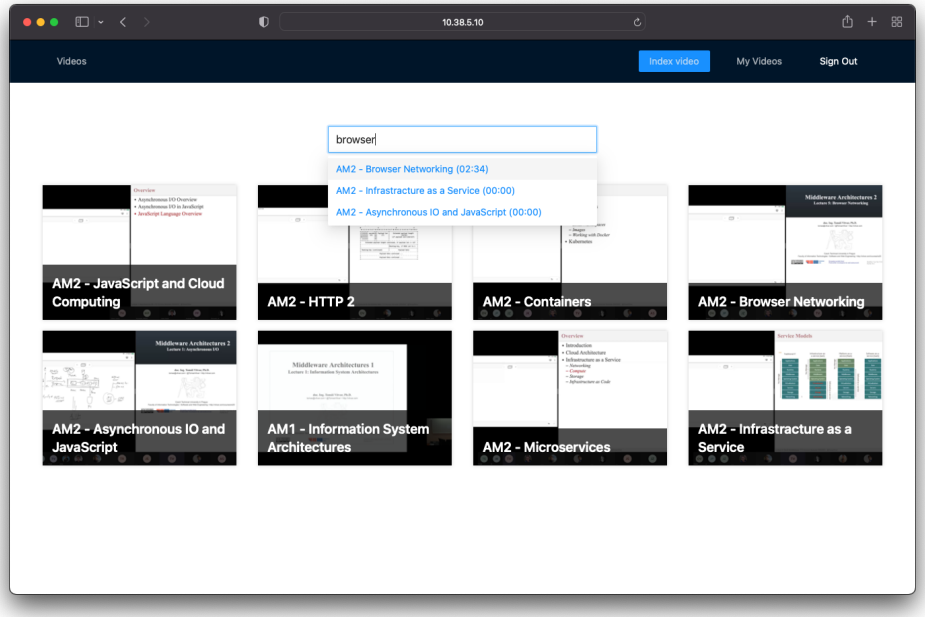

Figure 5.3: Search indices

a progress bar. The application starts uploading the video directly after it has been dropped. Under the upload area, there is a video name text field and *continue* button, which navigates user to the next step - Mark text content area page.

5.3.2.4 Mark text content area

In the second step of the index creation process, the user is prompted to mark a region with text content in the slides presented in the video. He can play the video to locate the text area conveniently. He can either hit the *continue* button or *clear* the video draft.

5.3.2.5 Upload table of contents

At the last step of the index creation process, a user can attach a table of contents (TOC) in JSON format into another file upload area. The TOC format is validated instantly on the client. If it is valid, the user can start the indexing process by hitting *Index video* button. Then he is redirected to Indexing progress page.

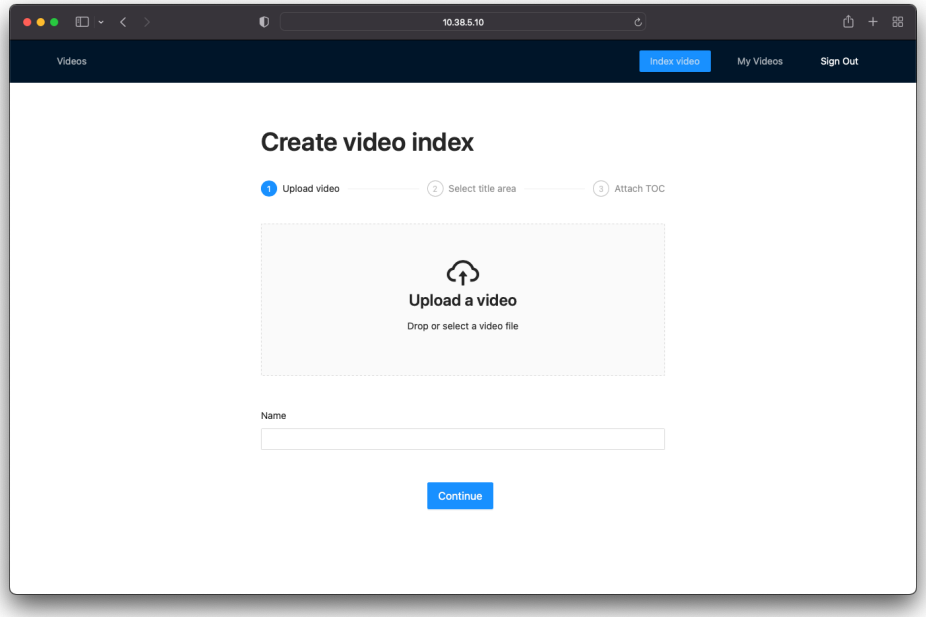

Figure 5.4: Index video page

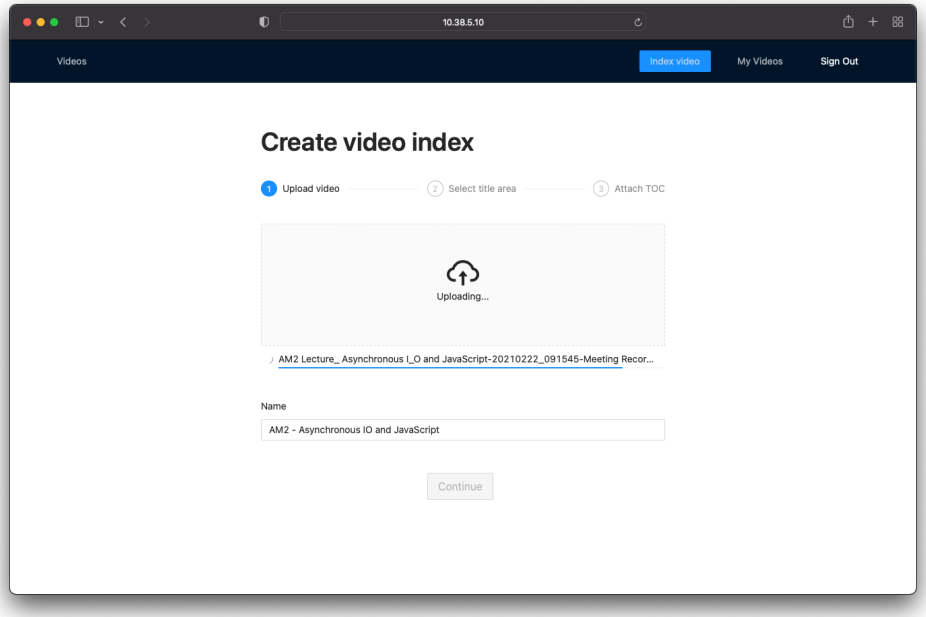

Figure 5.5: Uploading video page

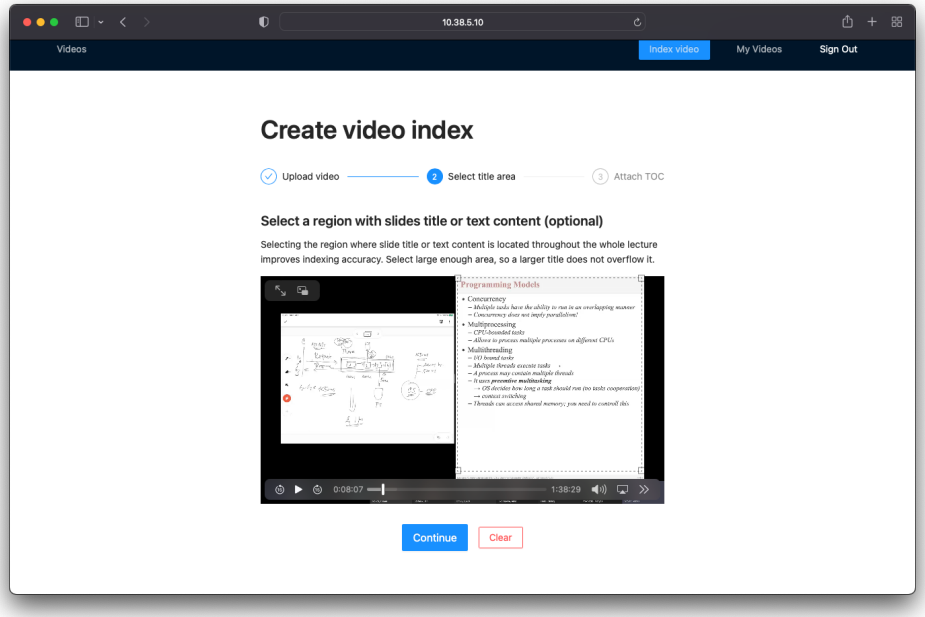

Figure 5.6: Mark text content area page

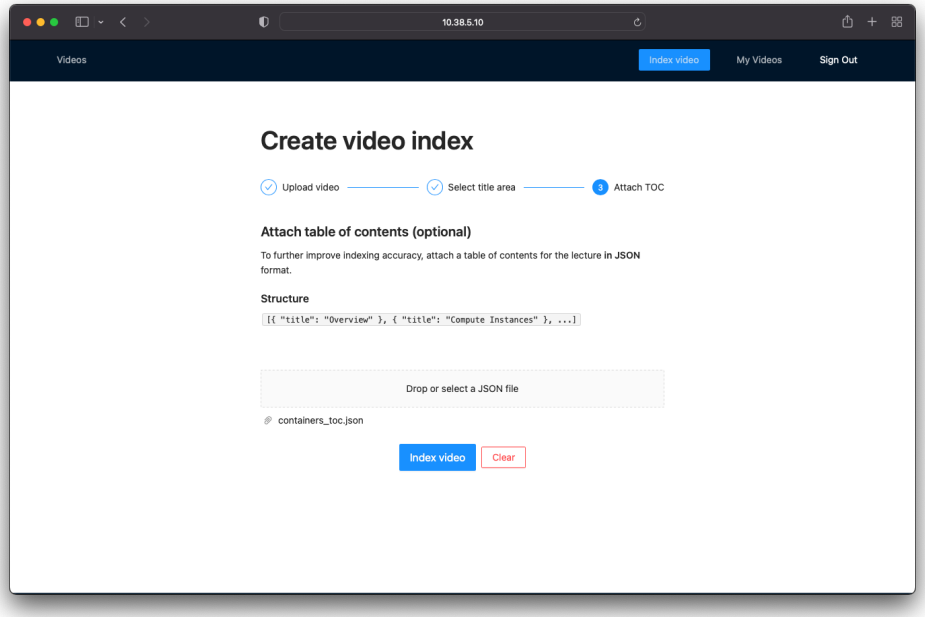

Figure 5.7: Upload table of contents page

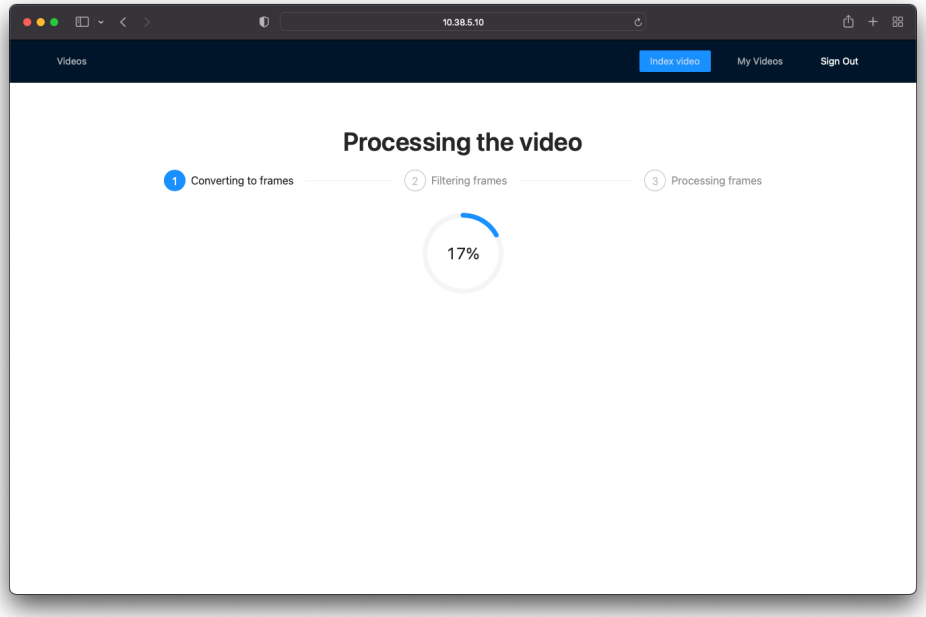

Figure 5.8: Mark text content area page

5.3.2.6 Indexing progress

Indexing progress page contains three steps with a circular progress bar for each indexing process step. The processing progress is displayed in real time. After the processing has finished, the user is redirected to Edit index page.

5.3.2.7 Video detail

At the top of Video detail page, there is a video title on the left and actions on the right - *Download index as JSON*, *Edit* and *Delete* buttons. Under the title, there is the video player. On the right side of the video player, there is a list of the video topics and a search field. If the user clicks on a topic, the video is skipped to the timestamp of the topic. The search works on the client-side, and the user can only search for words occurring among the topics.

5.3.2.8 Edit index

At the Edit index page, a user can change the name of the video, video topics (e.g. fix mistakes in topic detection) and their timestamps. The user can also delete selected index entries if they were detected incorrectly. There is a video player so the user can verify detected topics with the video.

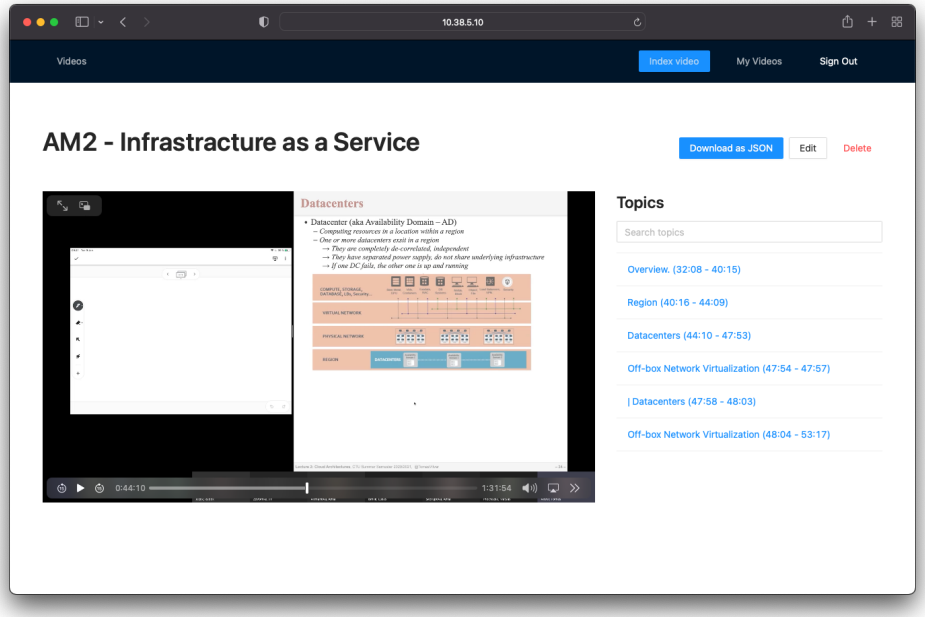

Figure 5.9: Video detail page

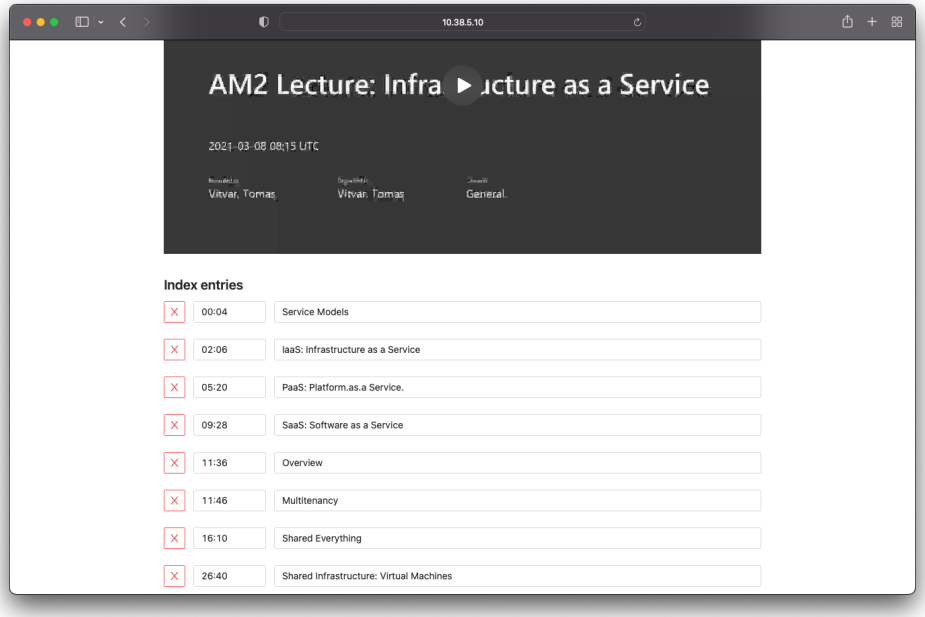

Figure 5.10: Edit index page

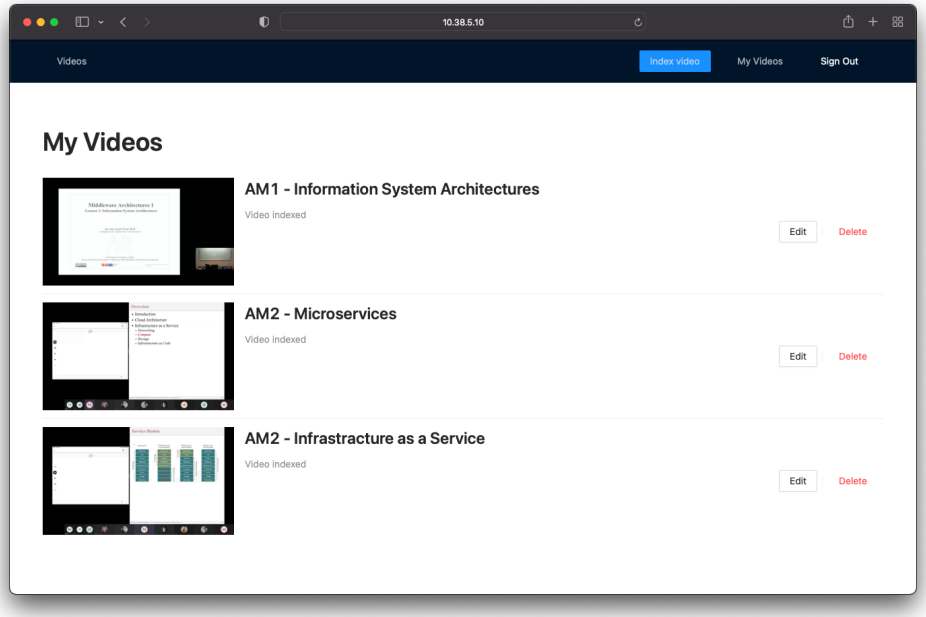

Figure 5.11: My Videos page

5.3.2.9 User videos

User videos page contains a list of videos the user uploaded, including videos in processing. In this list, he can delete a video or go to Edit index page.

Chapter 6

Verification

In this chapter, I first introduce a custom-designed precision metric to evaluate the indexing algorithm. The precision and running time is then used to find the best possible algorithm parameters (listed in table 5.2). The algorithm is evaluated on a dataset of video lectures from AM1 and AM2 courses at FIT CTU. Lastly, I describe how the indexing web platform was tested with end-to-end tests to verify correct functionality.

6.1 Precision metric

The algorithm's primary task is to correctly extract timestamps of presented slides from a video lecture. Therefore its output is an array of timestamps. To measure the accuracy of the presented algorithm, I designed a specific *precision* metric, which compares the algorithm's output with a reference solution.

Let E be a set of extracted timestamps, R a set of correct reference timestamps and I their intersection. The *precision* p is defined as

$$p = \frac{|I| - |E - I| * r}{|R|}, \tag{6.1}$$

where $r \in \langle 0, 1\rangle$ is a constant expressing a weight of surplus timestamps in the output.

The denominator takes a count of correctly extracted timestamps and subtract the number of surplus timestamps (not in the reference solution) weighted by a constant r. Missing timestamps are more critical in the solution than extra timestamps, which can be easily removed, but the missing ones are harder to find. That is the reason for weighting the number of extra slides by r, which is set to 0.5 in the tests. The value of the denominator is divided by the reference solution timestamps count, which makes p relative to the size of the solution.

For example let

$$E = \{4, 275, 455\}, R = \{4, 275, 300\}, I = \{4, 275\}, r = 0.5 \tag{6.2}$$

Two timestamps are extracted correctly, one is missing and one is extra. The *precision* is then

$$p = \frac{3 - 1 * \ 0.5}{3} \approx 0.83. \tag{6.3}$$

Because the extracted timestamp values depend on `frame_step` parameter of the algorithm from table 5.2, they can be extracted with an error and shifted up to `frame_step` forward or backward. Therefore, the set intersection is calculated with a custom equivalence operator e considering the error.

$$e(x, y) = |x - y| \leq frame\_step \tag{6.4}$$

6.2 Parameters

In order to have the best possible precision and performance tradeoff, I ran a set of tests to find appropriate values for the algorithm's parameters listed in table 5.2. The tests are run over a reference collection of video indexes and focus on two observed metrics - *precision* introduced in 6.1 and *processing time*. The reference collection consists of three different lecture videos from AM1/AM2 courses and corresponding indexes evaluated manually. Each video has a slightly different layout.

There are four test cases for each algorithm parameter. The algorithm is executed with different values for the observed parameter in each test case while keeping the other parameters fixed. The observed value is an average *precision* and *processing time* in seconds from all videos. Each test case is executed twice, and the results are averaged.

6.2.1 Frame step

Chart 6.1 depicts results from `frame_step` parameter test. The tested values were $\{1, 2, 4, 8\}$. The precision is more or less the same. The results are as expected for the running time - value 1 has the highest running time and 8 the lowest. There are more frames to process with a lower value of the parameter. Choosing the highest `frame_step` value would be natural. However, that would lead to inaccurate timestamps. A video keyframe with some topic could start at time t, but the algorithm would return timestamp $t \pm frame\_step$. That is acceptable for up to 2 or 4 seconds, not more.

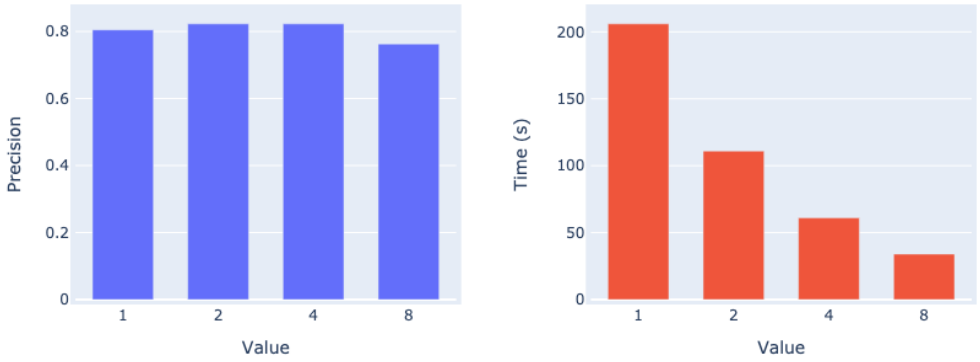

Figure 6.1: Frame step parameter test results

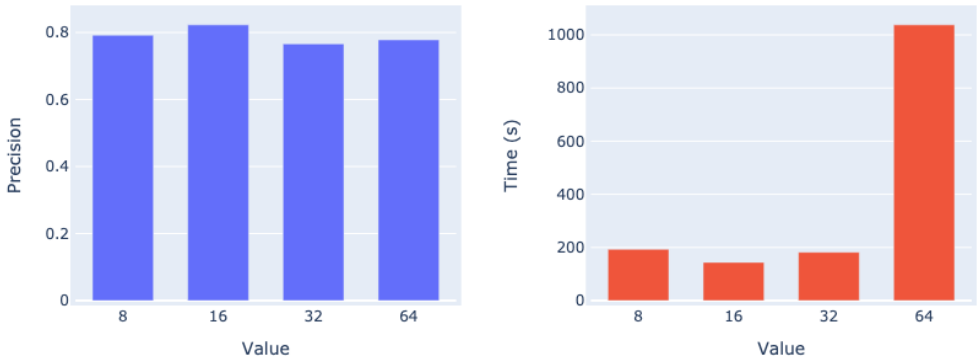

Figure 6.2: Hash size parameter test results

6.2.2 Hash size

The results for `hash_size` parameter are similar to `frame_step` parameter when looking at the precision - no value has significantly higher precision than the other (depicted in chart 6.2). Therefore, we can conclude that the size of a hash does not significantly impact the precision for the testing data. It is slightly different with observed running time. The largest hash size value runs significantly longer than smaller sizes, which have similar running times.

6.2.3 Image similarity threshold

Chart 6.4 depicts results for `image_similarity_threshold`. The best precision was achieved for the lowest value 0.7, altought the differences are not significant. The four lowest values from 0.7 to 0.9 ran the fastest. The slowest time, almost three times higher, was achieved for value 0.99.

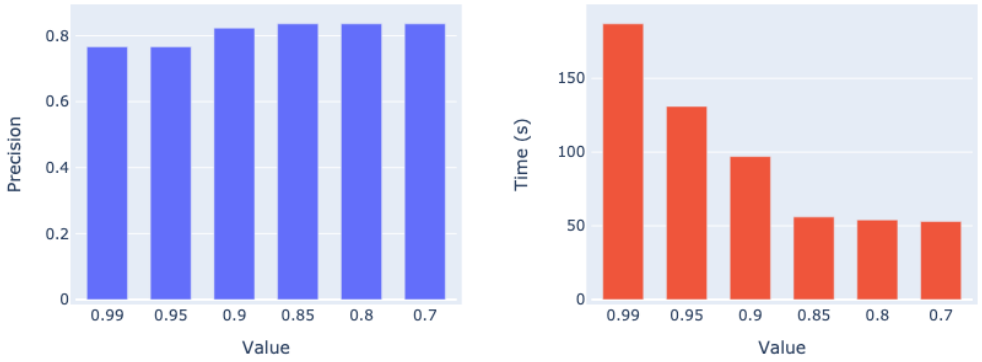

Figure 6.3: Image similarity threshold parameter test results

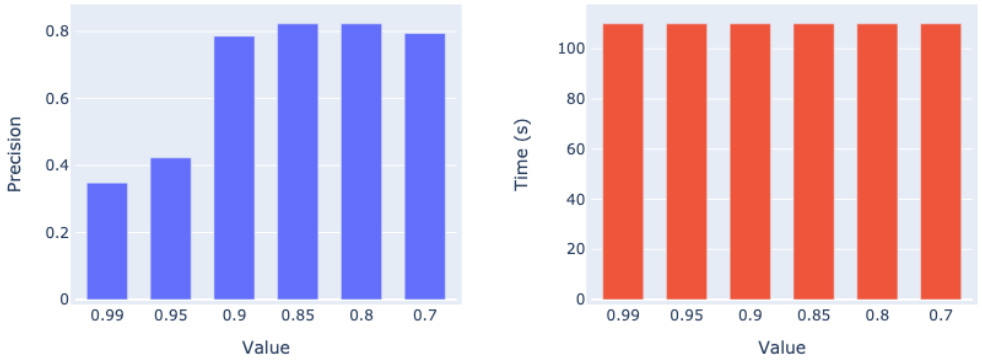

Figure 6.4: Text similarity threshold parameter test results

6.2.4 Text similarity threshold

For `text_similarity_threshold` parameter, the precision is higher for lower parameter values (chart 6.4). For this case, running time is the same for all values.

6.3 Evaluation

The measurements in section 6.2 show that various parameter values do not have a very high impact on the precision (with the given testing data). What they have an impact on, though, is speed. Based on the results, I chose a set of used parameters listed in table 6.1 and ran a set of tests on a complete dataset. The full dataset contains ten video lectures from AM1 and AM2 courses. The goal of the tests was to evaluate the precision and speed of the algorithm.

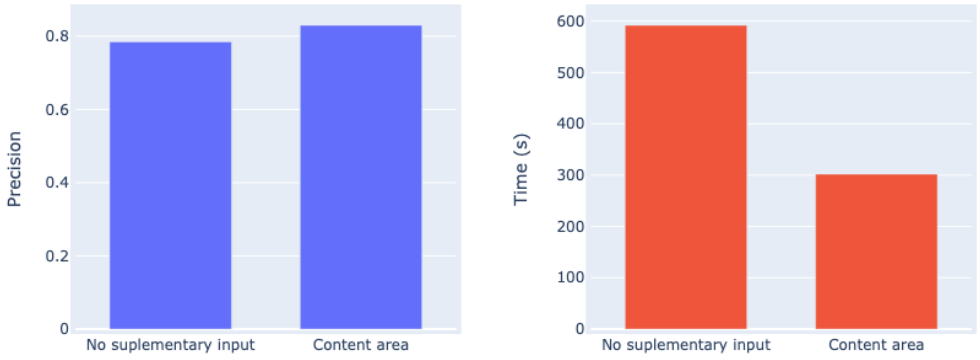

Figure 6.5: Full dataset test results

Parameter	Chosen value
`frame_step`	2
`img_sim_threshold`	0.85
`txt_sim_threshold`	0.85
`hash_size`	16

Table 6.1: Chosen algorithm parameters

The first test ran without a text content area as supplementary input and the second with the text content area input to compare the two methods.

The test results are presented in chart 6.5. The average precision for the test without supplementary input is 0.785 and 0.83 for the test with a text content area. The average running time for the first test is 9 minutes 52 seconds, whereas it is 5 minutes for the second test. The first test has worse precision because the algorithm can detect noise from the non-content area of frames. The second test is faster because, after the first phase of converting a video to frames, only the content area of frames is passed to the other stages of the algorithm, which naturally process fewer data. The tests have proven that supplementary material improves both precision and running time.

6.4 End-to-end tests

Generally, there are three types of tests from "how large part of a system they test" point of view - *unit*, *integration* and *end-to-end*. Unit tests cover small isolated units (typically functions), integration tests cover composing these units together, and end-to-end tests cover the whole system in the same way

a user experiences it [40]. Since the indexing platform consists of the frontend with UI and the backend, I chose to test it with automated end-to-end tests using the UI because it covers the system as a whole and verifies its functionality and requirements presented in chapter 3.

6.4.1 Cypress

Cypress[22] is a testing engine for web applications. It provides rich capabilities for visual testing, intercepting network requests, debugging and overall developer experience. Cypress can be used both for testing a frontend application in isolation by intercepting requests to a backend and defining custom mocked responses or, as a whole, connected to the backend. I chose the latter way for testing the indexing system.

Tests in Cypress are written in JavaScript or TypeScript and are organized into test cases (individual files). Cypress provides an interface for interaction with a website, like finding elements on a page, triggering a button click or waiting for a network request to complete. The tests are a codification of what a real user would do on a page. An example of a test case is presented in code 6.1. Cypress opens a web browser, executes the sequence of test case steps and reports on it. An example report of a successful test run is depicted in figure 6.6. It can also run in a command line environment, which is usefull for automation and Continues Integration systems [41].

```
it('a user can search published videos by a topic name', () => {
    cy.authenticate()
    cy.visit('/');

    cy.get('input[type="search"]').type('architecture');
    cy.getByTestId('search-option')
      .should('have.length.gte', 2)
      .eq(1)
      .click();

    cy.location().should((loc) => {
        expect(loc.pathname).to.match(/\/video\/*/);
        expect(loc.search).to.match(/t=*/);
    });
});
```

Listing 6.1: Cypress test case

However, there are some challenges in running automated UI tests. First, they must be executed in a dedicated testing environment with seeded data not to interfere with development or staging environments. Second, there are usually issues with authentication, especially with OAuth 2.0 integration, and

[22] https://www.cypress.io

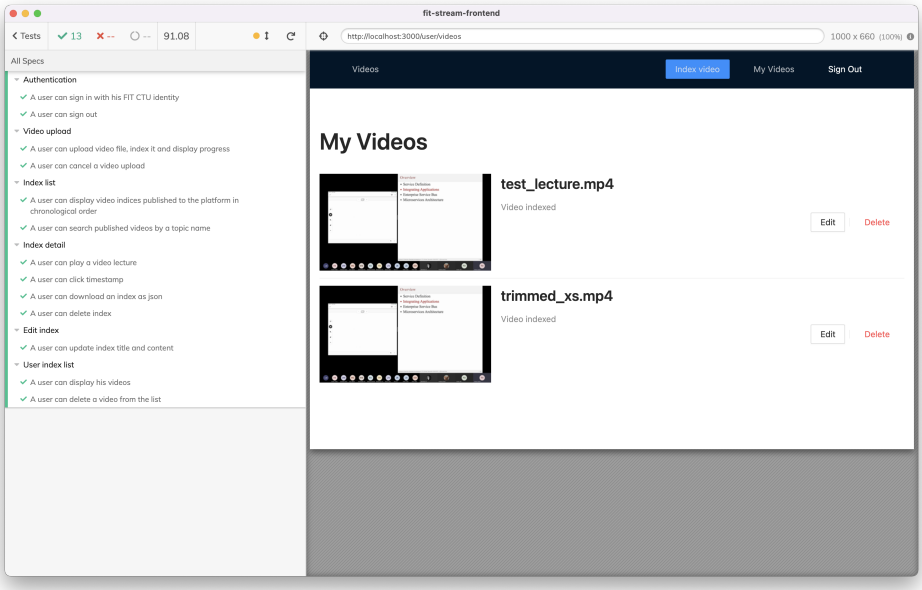

Figure 6.6: Cypress tests example

signing in with UI is not recommended in tests due to performance reasons [42]. Last but not least, to keep each test case isolated and independent of others, the test engine clears browser cookies before each run, which can also cause problems with authentication. One can then authenticate before each test or preserve cookies with a special Cypress command [43].

6.4.2 Seeding test data

The testing environment needs to be seeded with some initial data in order for the tests to work correctly. Concretely, there have to be some video indexes. There are two options for data seeding -

1. add the test data directly to the database, storage and Elastic Search,
2. create them via the provided REST API.

I chose the latter to decouple the test from the internal implementation and the need to communicate with multiple services. That way, the seeding script only makes two HTTP requests for uploading a video and indexing it. The downside is, we have to wait for the indexing process to finish, which adds approximately 20 seconds of test execution time. The seeding script is written directly in Cypress and executed before test running.

Test case	Functional requirements
Authentication	F1, F2
Video upload	F3, F4
Index list	F9, F11
Index detail	F5, F10, F8
User index list	F6, F8
Edit index	F7

Table 6.2: Cypress test cases mapping to the functional requirements

6.4.3 Authentication

For a programmatic sign-in, I leveraged the refresh token endpoint of the OAuth server to obtain a new access token and use it in HTTP requests. The programmatic way is faster than using the UI to sign in, plus the OAuth server website for signing in does not redirect back to the application after successful authentication with automated software. In order to save calling the refresh token endpoint multiple times for each test case, the access token is stored in the browser's Local Storage and preserved throughout the test run. The authentication script is also written directly in Cypress and executed before a test run.

6.4.4 Test cases

The test cases are part of the frontend GIT repository, which can be found at `src/fit-stream/frontend/cypress` directory of the enclosed memory card. Table 6.2 presents the mapping from the functional requirements listed in chapter 3 to Cypress test cases. The tests verify that all functional requirements have been met.

Chapter 7

Deployment

The first section of this chapter explains application containers, their advantages compared to the traditional deployment and how they can be practically used with Docker and Docker Compose. The second section presents how the video lectures indexing platform is deployed on a virtual machine as application containers.

7.1 Containers

Conventionally, software used to be deployed on a physical or virtual server. There could be many applications running on the same server. Administrators had to ensure the application was correctly configured for each application and environment pair, and the server had all required dependencies like libraries or a database installed. That is fine for simple applications running in one environment that have a few dependencies. However, how does such a way of deployment scale for multiple environments? What if we upgrade a library version for one application but keep the older version installed on the system for another application? Deployment management becomes a nightmare for developers and administrators with an increased number of applications, dependencies, and environments.

Recently, the deployment of applications using containers has become the new standard. "*A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another [44].*" Containers eliminate the problem of dependencies because each application has its own installed in the container, and containers do not conflict among themselves. Combined with Virtual Machines (VM), they consume system resources efficiently while keeping the isolation [45]. The downside of containers and their orchestrators is a steep learning curve and configuration complexity. Therefore, automation is a must and pays off in the long term.

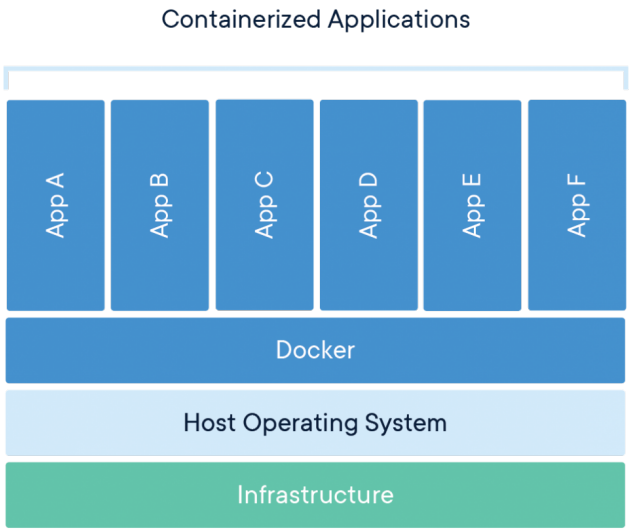

Figure 7.1: Containerized application scheme powered by Docker [44]

7.1.1 Docker

Docker[23] is the most popular container engine for deploying applications. It caused market disruption by deciding to resolve the problem of application deployment. Docker standardized the format and distribution of containers. Using Docker brings developers and administrators easy way to manage

- application dependencies (libraries, frameworks, services),
- multiple environments (local, testing, staging and production) and migration among them,
- automatic deployment on any host (VM, physical machine),
- scaling and reproducibility [46].

In Docker, there is a concept of a layered filesystem called *a container image*. An image contains the container's filesystem and includes everything needed to run an application - binaries, libraries, configuration, environment variables, and other dependencies. It also specifies a default command to run after starting the container [47]. The individual layers are shared across all

[23] https://www.docker.com

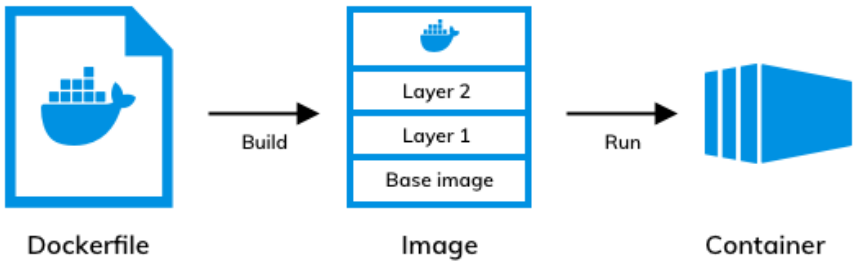

Figure 7.2: Relationship between Dockerfile, Image and Container

images and can be arbitrarily extended. For example, if we have one image for running a Java application and another for a database, they can share the operating system layer or libraries. This approach of sharing layers is space-efficient.

An image is defined in Dockerfile. Each line of Dockerfile is an individual filesystem layer. If we build an image and run it with Docker, the result is a running container - a process running in the isolated filesystem [45]. Another advantage of the layered filesystem is that it can be cached. After building a container and then changing its Dockerfile, only the changed layers are rebuilt.

```
1 FROM denoland/deno:alpine-1.12.0
2 EXPOSE 8004
3 WORKDIR /app
4 COPY deps.ts .
5 RUN deno cache deps.ts
6 ADD . .
7 RUN deno cache app.ts
8 CMD ["run", "--allow-net", "--allow-env", "app.ts"]
```

Listing 7.1: Index Service Dockerfile

Code example 7.1 is Dockerfile for Index Service. Each line is a filesystem layer and does the following:

1. defines a base image including OS and a version,
2. exposes a TCP port on which the server is running,
3. changes the working directory,
4. copies a dependency file into the working directory,

5. installes dependencies,

6. adds source code into the working directory,

7. builds the source code,

8. specifies the default command run when the container is started.

We can build the image by running `docker build -t index-service .` in a directory with the Dockerfile and then start a container with `docker run -p 8004:8004 -it index-service`. The running server TCP port `8004` in the container is mapped to the same port on localhost.

7.1.2 Docker Compose

While Docker supports communication between containers via various networking options [48] other orchestration tasks like restarting containers or managing dependencies among them have to be managed manually. Docker Compose is a tool that automates complex configuration of multiple container applications. It uses YAML file for application configuration and can run multiple containers with a single command. Docker Compose also supports multiple configuration files for different environments and can share a common configuration [49].

Generally, Docker Compose is a good solution for local development. There are more advanced orchestration solutions for production, like Kubernetes [24] supporting multiple running instances of a service, rollouts and rollbacks, load balancing, autoscaling and many more. For a simple deployment on one physical or virtual machine, Docker Compose can usually serve well too.

Code listing 7.2 shows an example configuration of two services - a backend server and a database. For each service, we specify either `image`, which is an existing image pulled from a Docker repository or a path to a local directory with a custom Dockerfile as `build` option. The services communicate over a virtual network `net_mongo` Docker Compose automatically creates [50]. We can specify `restart` policy for the containers in case of an error or a container stops [51]. Some services usually need to wait for others before starting. Docker Compose allows specifying these dependencies with `depends_on` option. Last but not least, the database service needs to persist data. Defining `volumes` ensures the data are persisted even when the container is stopped [49]. All services can be built with `docker compose build` command and then started on the background with `docker compose up -d`.

[24] https://kubernetes.io/

```
version: '3.8'
services:
  index_service:
    restart: always
    build:
      context: ./index-service
    depends_on:
      - mongodb
    networks:
      - net_mongo
  mongodb:
    image: mongo
    restart: always
    networks:
      - net_mongo
    volumes:
          - mongodb_data:/data/db
networks:
  net_mongo
volumes:
  mongodb_data
```

Listing 7.2: Docker Compose configuration for two containers

7.2 Platform deployment

Each service described in chapter 4 has its Dockerfile in a GIT repository with the source code and runs as a container. The repositories can be found on the memory card attached to this work. Docker Compose manages the containers with both local and production configurations. Environment specific configuration, like URLs or API keys, is stored in a `.env` file and provided to Docker Compose. The indexing platform, including all services, databases and frontend, is deployed on a virtual machine provided by FIT CTU on their private cloud [25]. The virtual machine is accessible through SSH and Virtual Private Network (VPN).f

Docker Context is employed to manage the remote Docker node from the local CLI for the deployment process. It uses SSH to connect to the remote server, build and run commands are executed locally, but have effect remotely [52]. For example, to start the services on the VM, we execute the following command on a local machine: `docker --context fit-stream-vm compose --env -f docker-compose.production.yml up -d`.

The platform needs persistent storage for video files and image thumbnails. That is achieved by mounting a disk connected to the VM to individual containers that work with it as a Compose volume. Compose volumes are also created for Mongo DB, Redis and Elastic Search.

[25] https://cloud.fit.cvut.cz/

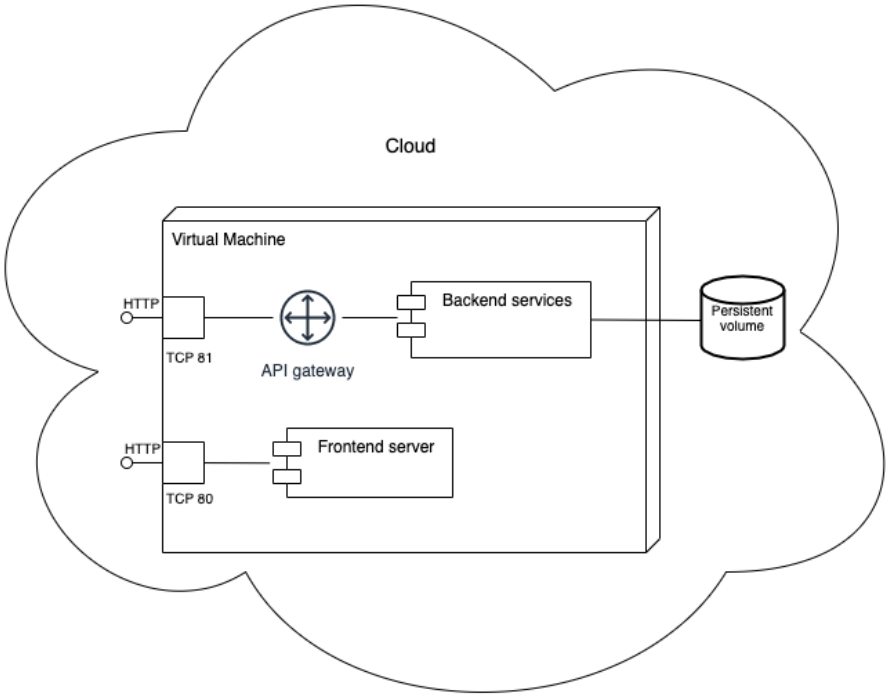

Figure 7.3: Deployment diagram of the indexing platform

To access both the frontend and backend from a machine other than the VM itself, we need to map TCP ports exposed by the containers correctly in Compose configuration. The frontend is mapped to the default HTTP port 80, so it is accessible without port specification in a browser. The backend is mapped to port 81. The application runs on IP address `10.38.5.10`. Note that a user must be still connected to the VPN.

The next steps for production deployment are mapping a domain to the server's IP address and setting up TLS for both the frontend and backend due to security concerns. These steps are outside the scope of this work.

Conclusion

This thesis aimed to create a web service capable of indexing video lectures with the help of supplementary resources, mainly for courses on FIT CTU. Among the thesis' goals were: analyse the current research in video indexing, design the web service, develop it including API and UI, deploy it as application containers, test it and evaluate its accuracy and performance.

The video lecture indexing service is valuable for students in providing them with a possibility to search topics in a video lectures collection and skip directly to the relevant parts. It all happens in a semi-automated way, saving teachers time in creating video indexes.

The service was designed as multiple components - core algorithm as an independent module, backend services providing a REST API and a frontend with UI. It was developed in both Python and TypeScript. The evaluation was performed with good precision on the testing dataset, and the service was tested with end-to-end tests, verifying the requirements were met. Last but not least, the service was deployed on a virtual machine as application containers.

Apart from successfully indexing video lectures, one of the work's findings was that providing a text content area as supplementary input to the indexing algorithm helps improve accuracy and processing time. Providing a lecture's table of contents solves a problem with a teacher skipping some slides or jumping back in slides.

The service can be further extended by incorporating lecture speech processing with Automatic Speech Recognition and including it into the indexed content. Another extension could be integrating the service into FIT CTU learning portals like Courses.

Bibliography

[1] Patel, B. V.; Meshram, B. B. Content Based Video Retrieval Systems. *International Journal of UbiComp (IJU), Vol. 3, No. 2*, 4 2012.

[2] Chand, D.; H., O. Content-Based Search in Lecture Video: A Systematic Literature Review. *3rd International Conference on Information and Computer Technologies (ICICT]*, 2020.

[3] Yang, H.; Meinel, C. Content Based Lecture Video Retrieval Using Speech and Video Text Information. *IEE Transactions On Learning Technologies, vol. 7, no. 2*, 2014.

[4] Medida, L. H.; Ramani, K. An Optimized e-Lecture Video Search and Indexing framework. *International Journal of Computer Science and Network Security, vol. 21, No. 8*, 8 2021.

[5] Wang, S. P.; Xiaolong, C.; et al. InVideo: An Automatic Video Index and Search Engine for Large Video Collections. 2017, ISBN 978-1-61208-559-3.

[6] Tappert, C.; Suen, C.; et al. The state of the art in online handwriting recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, volume 12, no. 8, 1990: pp. 787–808, doi:10.1109/34.57669.

[7] Brisinello, M.; Grbić, R.; et al. Improving optical character recognition performance for low quality images. In *2017 International Symposium ELMAR*, 2017, pp. 167–171, doi:10.23919/ELMAR.2017.8124460.

[8] Skopal, T. Lecture 3: Text-based and bag-of-features models. In *NI-VMM*, 2020, p. 12, [cit. 2021-09-21]. Available from: https://moodle-vyuka.cvut.cz/pluginfile.php/428983/course/section/70484/lecture03.pdf

[9] Rajaraman, A.; Ullman, J. D. *Mining of Massive Datasets*. 2011, ISBN 978-1-139-05845-2, 1-17 pp.

[10] Adcock, J.; M., C.; et al. TalkMiner: A Lecture Webcast Search Engine. *Proc. ACM Int. Conf. Multimedia*, 2010.

[11] Vitvar, T. Lecture 2: Cloud Architectures. In *Middleware Architectures 2*, 2021, [cit. 2021-09-17]. Available from: https://w20.vitvar.com

[12] Lee, S. M.; Xin, J. H.; et al. Evaluation of Image Similarity by Histogram Intersection. *Color Research and Application*, volume 30, 2005: pp. 265–274.

[13] Kang, L.-W.; Hsu, C.-Y.; et al. Feature-Based Sparse Representation for Image Similarity Assessment. *IEEE Transactions on Multimedia*, volume 13, no. 5, 2011: pp. 1019–1030, doi:10.1109/TMM.2011.2159197.

[14] Wang, J.; song, Y.; et al. Learning Fine-Grained Image Similarity with Deep Ranking. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 04 2014: pp. 1386–1393, doi:10.1109/CVPR.2014.180.

[15] Lv, X.; Wang, Z. J. Perceptual Image Hashing Based on Shape Contexts and Local Feature Points. *IEEE Transactions on Information Forensics and Security*, volume 7, no. 3, 2012: pp. 1081–1093, doi:10.1109/TIFS.2012.2190594.

[16] Reddy, S. Pre-Processing in OCR. *Towards Data Science*, 3 2019, [cit. 2021-10-08]. Available from: https://towardsdatascience.com/pre-processing-in-ocr-fc231c6035a7

[17] Sekhon, M. Image Filters in Python. *Towards Data Science*, 10 2019, [cit. 2021-10-08]. Available from: https://towardsdatascience.com/image-filters-in-python-26ee938e57d2

[18] Vitvar, T. Lecture 3: Service Oriented Architecture. In *Middleware Architectures 1*, 2020, [cit. 2021-09-10]. Available from: https://mdw.vitvar.com

[19] Richardson, C. Pattern: API Gateway / Backends for Frontends. [cit. 2021-10-04]. Available from: https://microservices.io/patterns/apigateway.html

[20] Mozilla; individual contributors. Cross-Origin Resource Sharing (CORS]. 10 2021, [cit. 2021-10-04]. Available from: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

[21] F5, Inc. Authentication Based on Subrequest Result. [cit. 2021-09-28]. Available from: https://docs.nginx.com/nginx/admin-guide/security-controls/configuring-subrequest-authentication

[22] D. hardt, E. The OAuth 2.0 Authorization Framework. RFC, 2012, [cit. 2021-09-21]. Available from: https://datatracker.ietf.org/doc/html/rfc6749

[23] Jones, M.; Bradley, J. JSON Web Token (JWT). RFC, 2015. Available from: https://datatracker.ietf.org/doc/html/rfc7519

[24] Rodger, R. The Tao of Microservices. chapter 3.1.1, 3.5.7, Manning Publications Co., 2017, ISBN 9781617293146. Available from: https://www.manning.com/books/the-tao-of-microservices

[25] Gordon, E. K. Isomorphic Web Applications. chapter 1, Manning Publications Co., 2018, ISBN 9781617294396. Available from: https://livebook.manning.com/book/isomorphic-web-applications

[26] Sakimura, N. E.; Bradley, J.; et al. Proof Key for Code Exchange by OAuth Public Clients. RFC, 9 2015. Available from: https://datatracker.ietf.org/doc/html/rfc7636

[27] Vitvar, T. Lecture 5: Representational State Transfer. In *Middleware Architectures 1*, 2020, [cit. 2021-09-17]. Available from: https://mdw.vitvar.com

[28] Spasojevic, M. ASP.NET Core Web API – Repository Pattern. 6 2021, [cit. 2021-11-22]. Available from: https://code-maze.com/net-core-web-development-part4/

[29] StrongLoop, IBM, and other expressjs.com contributors. Using middleware. 2017, [cit. 2021-11-22]. Available from: https://expressjs.com/en/guide/using-middleware.html

[30] Mozilla and individual contributors. *First-class Function*. [cit. 2021-11-27]. Available from: https://developer.mozilla.org/en-US/docs/Glossary/First-class_Function

[31] Mozilla and individual contributors. *JavaScript modules*. [cit. 2021-11-27]. Available from: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules

[32] D. Miller, E. *OpenAPI Specification v3.1.0*. 2 2021. Available from: https://spec.openapis.org/oas/v3.1.0.html

[33] *Nginx documentation*. [cit. 2021-11-17]. Available from: https://nginx.org/en/docs/

[34] F5, Inc. NGINX upload module. [cit. 2021-10-04]. Available from: https://www.nginx.com/resources/wiki/modules/upload/

[35] Svoboda, M. Lecture 1: Introduction. In *Advanced Database Systems*, 2020, [cit. 2021-03-11]. Available from: https://www.ksi.mff.cuni.cz/~svoboda/courses/201-MIE-PDB/lectures/MIEPDB16-Lecture-01-Introduction.pdf

[36] Richardson, C. Server-sent events. [cit. 2021-10-04]. Available from: https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events

[37] Popa, B. Dramatiq: User Guide. [cit. 2021-11-28]. Available from: https://dramatiq.io/guide.html

[38] Meta Platforms, Inc. *Components and Props*. [cit. 2021-11-30]. Available from: https://reactjs.org/docs/components-and-props.html

[39] Vercel, Inc. *Routing*. [cit. 2021-11-30]. Available from: https://nextjs.org/docs/routing/introduction

[40] Testim Inc. End-to-End Testing vs Integration Testing. 3 2021, [cit. 2021-12-21]. Available from: https://www.testim.io/blog/end-to-end-testing-vs-integration-testing

[41] Cypress Inc. Command Line. [cit. 2021-12-21]. Available from: https://docs.cypress.io/guides/guides/command-line

[42] Cypress Inc. Testing Your App. [cit. 2021-12-21]. Available from: https://docs.cypress.io/guides/getting-started/testing-your-app

[43] Cypress Inc. Cypress.Cookies. [cit. 2021-12-21]. Available from: https://docs.cypress.io/api/cypress-api/cookies

[44] Docker, Inc. Use containers to Build, Share and Run your applications. [cit. 2021-11-14]. Available from: https://www.docker.com/resources/what-container

[45] Vitvar, T. Lecture 3: Cloud Native and Microservices. In *Middleware Architectures 2*, 2020, [cit. 2021-09-10]. Available from: https://w20.vitvar.com

[46] Vondra, T. Lecture 8: Docker. In *Virtualization and Cloud Computing*, 2021, [cit. 2021-09-17]. Available from: https://courses.fit.cvut.cz/NI-VCC/

[47] Docker, Inc. Docker: Orientation and setup. [cit. 2021-11-16]. Available from: https://docs.docker.com/get-started

[48] Docker, Inc. Networking overview. [cit. 2021-11-17]. Available from: https://docs.docker.com/network/

[49] Docker, Inc. Overview of Docker Compose. [cit. 2021-11-17]. Available from: https://docs.docker.com/compose/

[50] Docker, Inc. Networking in Compose. [cit. 2021-11-17]. Available from: https://docs.docker.com/compose/networking/

[51] Docker, Inc. Start containers automatically. [cit. 2021-11-17]. Available from: https://docs.docker.com/config/containers/start-containers-automatically/

[52] Docker, Inc. Docker Context. [cit. 2021-11-17]. Available from: https://docs.docker.com/engine/context/working-with-contexts/

Appendix A

Contents of enclosed memory card

- `readme.txt` ..... the file with memory card contents description
- `src` ..... the directory of source codes
  - `installation.md` ..... Installation guide
  - `fit-lecture-indexer` ..... Core algorithm package
  - `fit-stream`
    - `api-gateway` ..... API gateway configuration
    - `docs` ..... API documentation
    - `frontend` ..... Frontend
    - `index-service` ..... Index Service
    - `video-service` ..... Video Service
    - `user-service` ..... User Service
    - `job-service` ..... Job Service
    - `data` ..... Data storage
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `thesis.pdf` ..... the thesis text in PDF format