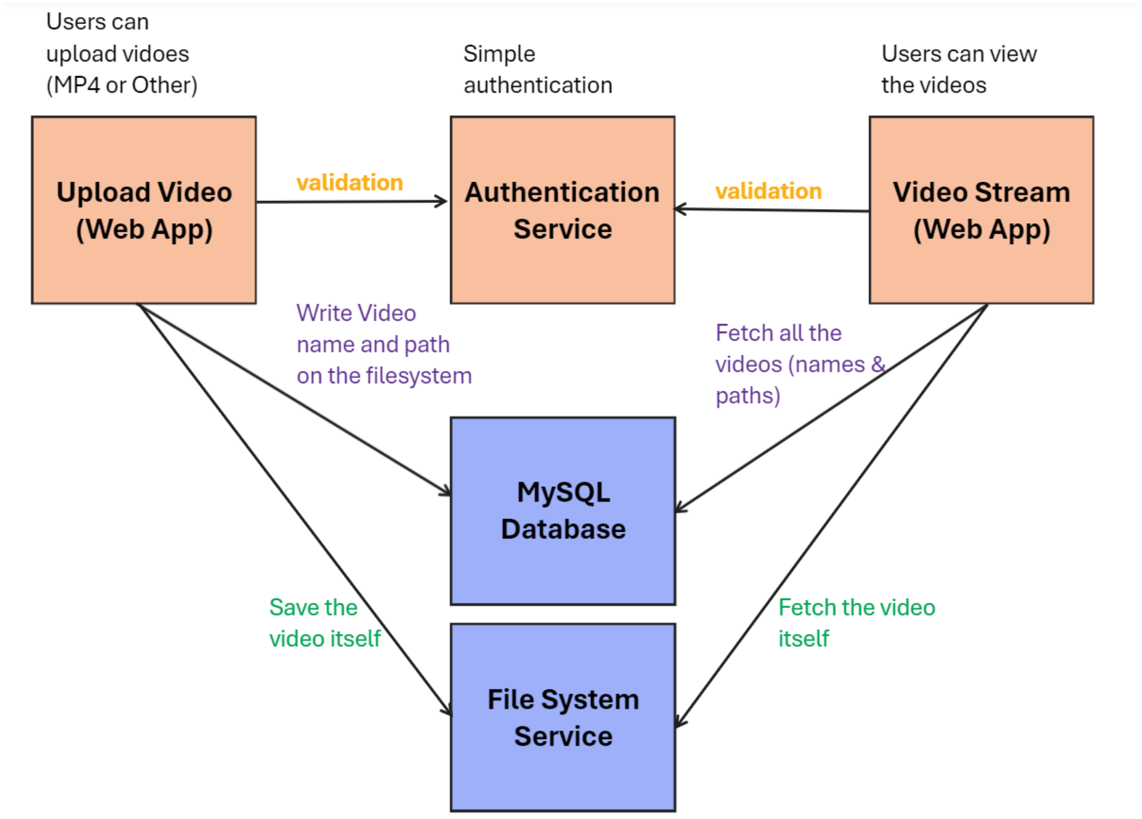
Video Streaming System

Containerized Video Streaming Architecture Using Docker

Outlines:

Introduction

We will construct the following Video Streaming architecture:



Which contains the following Microservices:

- **Video Uploading App**
  - Which allows the **authorized users** to **upload** a video on the **filesystem** service and save the data of the video on the **MySQL** service
- **Video Streaming App**
  - Which allows the **authorized users** to **view** all the videos that have been uploaded using the Video Uploading app
- **Authentication service**
  - A simple **authentication** service (**uses the MySQL service in my case**) to authenticate the users before accessing the **Uploading** and **Streaming** web apps
- **Filesystem service**
  - We can **upload** and **fetch** the videos themselves from this filesystem service.
- **MySQL Database**

Note: We will be using **Node.js** to build the Web Apps, Filesystem and Auth. services.

Microservices Explanation

Next, we will explain each of the microservices in details:

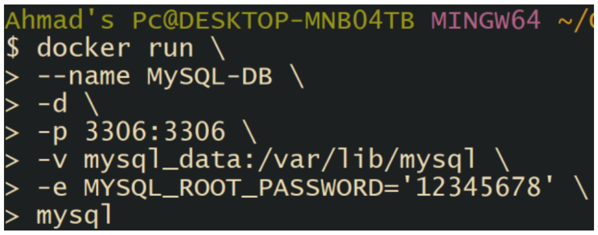
1. MySQL database:

- We will use **MySQL** database to store the video names and video paths, with the database **video_data** and the table **video_info** and all the other services will talk to this database and this table.

- Note that we will first build the whole system and after that we will consider the **containerization**, **dockerfiles** and **docker-compose**, but for this service, we will depend on docker image (**MySQL** image).

- To build this service, we will need:
  - **MySQL** docker **image**.
  - **mysql_data** docker **volume**.

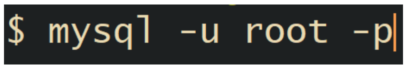
- So, we will use the following command to create the volume:



```
Ahmad's Pc@DESKTOP-MNB04TB MINGW64 ~/
$ docker run \
> --name MySQL-DB \
> -d \
> -p 3306:3306 \
> -v mysql_data:/var/lib/mysql \
> -e MYSQL_ROOT_PASSWORD='12345678' \
> mysql
```

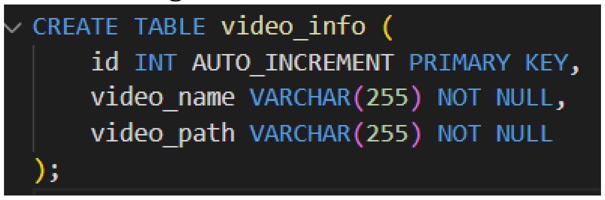
  With this command we've created a docker container from the image (**mysql**) in port **3306**, and with the volume (**mysql_data**) at the directory **/var/lib/mysql**, and with a root password = "**12345678**" in detached mode.

- After that we will enter the mysql database using the command:



```
$ mysql -u root -p
```

- And then we will create the database (**video_data**) and the table (**video_info**) with the following:



```
CREATE TABLE video_info (
    id INT AUTO_INCREMENT PRIMARY KEY,
    video_name VARCHAR(255) NOT NULL,
    video_path VARCHAR(255) NOT NULL
);
```

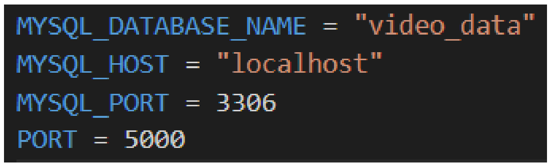
  With this, our database is **ready**!

2. Authentication Service:

- This simple **authentication** service will check if the user (using the entered **username** and **password**) has access to the database (have the **right privileges** to the **Video_Data** database that we've created).

- As I mentioned above, we will use **Node.js** to build this service, so after we've installed **Node.js** and configure it, we will use the command (**npm init**) to initialize the packages and the dependencies and after that we will install the following using **NPM** (run the command **npm i <module>**):
    - **express**, **mysql2**, **dotenv**, **path**, **url**

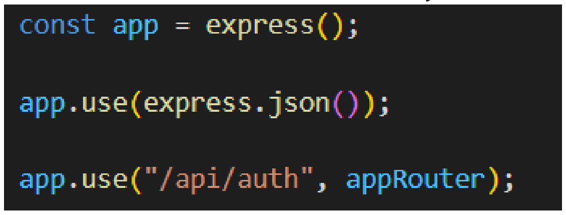
- And after that we will create file (**.env**) for the database and other configurations:



```
MYSQL_DATABASE_NAME = "video_data"
MYSQL_HOST = "localhost"
MYSQL_PORT = 3306
PORT = 5000
```

  As we can see it contains the database name, and the database host (currently **localhost**, we will later use the **host.docker.internal**), and the port of the database, and finally the port of the authentication microservice.

- Then, we will create the **index.js**:



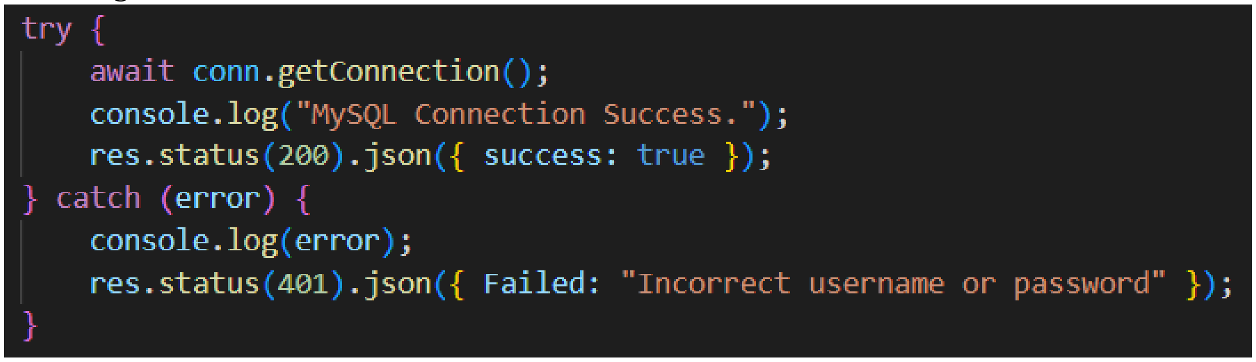
```
const app = express();

app.use(express.json());

app.use("/api/auth", appRouter);
```

  Which sets up the express server, router (**appRouter**) and defines a route (**/api/auth**) which will listen on port **5000** (port that we've provide in the **.env** file).

- After that, we need to specify the routes for the express router and the following:
    - **(/)**: which is the endpoint http://localhost:5000/api/auth that will **render an HTML** page to enter the username and password and then send these data to the (/login) endpoint to handle the authentication.
    - **(/login)**: which will check the **authentication** of the user data (username and password) at the http://localhost:5000/api/auth/login.
    - **(/get & /save)**: these endpoints will help us to **save the username and password** and retrieve them later when needed in the other services

- With these endpoints, we can **authenticate the users** when trying to access the uploading or streaming video services.

- Checking the **authentication**:



```
try {
    await conn.getConnection();
    console.log("MySQL Connection Success.");
    res.status(200).json({ success: true });
} catch (error) {
    console.log(error);
    res.status(401).json({ Failed: "Incorrect username or password" });
}
```

We will simply check if we can **connect to the database or not**, if the connection success, then the entered username and password **have the privileges to the database**, otherwise this user will be **unauthorized**.

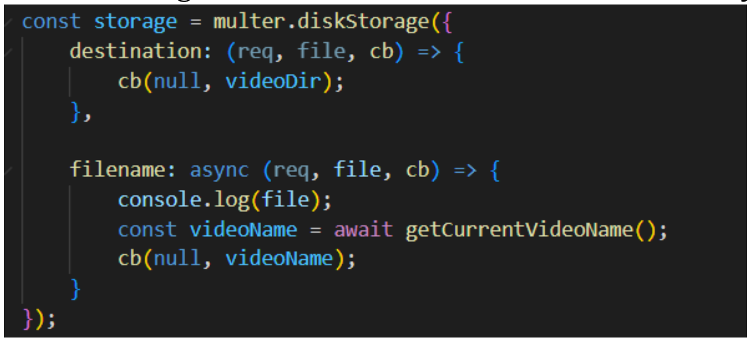
3. Filesystem service:

- In this service, we will use modules like "**multer**" and "**fs**" to make the filesystem service (also use command **npm i multer**, and **npm i fs**)

- We will configure the **.env** file to have **PORT = 5005**.

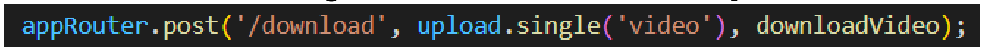
- The routes of the filesystem service will be:
  - **(/download)**: which will use **multer** module to save the video inside the filesystem (in the directory **/mnt/filesystem**)
  - **(/video/:name):** which will take a video **name** with the **path** and fetch it from the filesystem (return the saved **video itself**).

- We can configure multer to save the videos in the **filesystem** as the following:



```
const storage = multer.diskStorage({
    destination: (req, file, cb) => {
        cb(null, videoDir);
    },

    filename: async (req, file, cb) => {
        console.log(file);
        const videoName = await getCurrentVideoName();
        cb(null, videoName);
    }
});
```

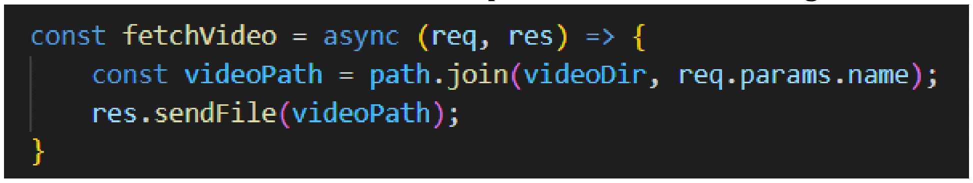
After that use this storage to save the videos when requested as the following:



```
appRouter.post('/download', upload.single('video'), downloadVideo);
```

The **upload.single('video')** will save the sent video in the filesystem, and after that we will simply notify the client that the video has been added successfully to the filesystem.

- If we need to fetch a video from the filesystem, we will just construct the video path, and then send the video file as a response, like the following:



```
const fetchVideo = async (req, res) => {
    const videoPath = path.join(videoDir, req.params.name);
    res.sendFile(videoPath);
}
```

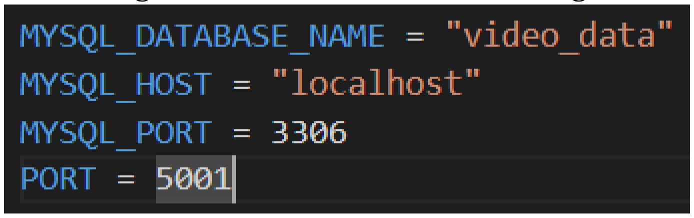
- And with this, our filesystem service is **ready**!

4.Video Uploading Web App:

- Next, we will explain the process of uploading the video using this service, but first let us talk about the used modules and some configurations.

- We will be use npm modules like:
    - **path**: provides utilities for **working with file** and directory paths.
    - **The other previous modules.**

- Will configure the **.env** file as the following:



```
MYSQL_DATABASE_NAME = "video_data"
MYSQL_HOST = "localhost"
MYSQL_PORT = 3306
PORT = 5001
```

We will use port **5001** for this service and the remaining is the database configurations.

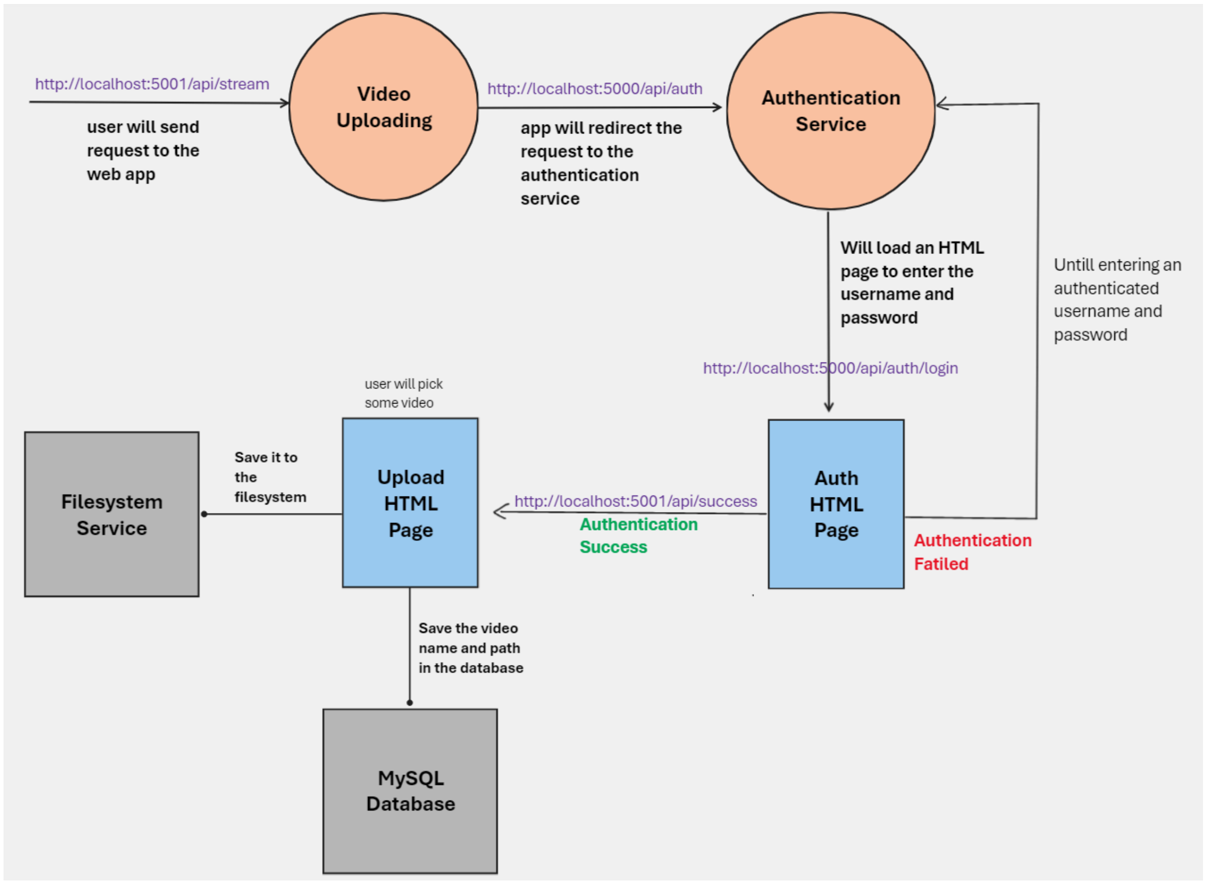
- This web app will deal with the authentication service, the filesystem service and **mysql** database.

- When the user enters his/her **credentials**, the authentication service will check these credentials, if authenticated, then the authentication service will get back to the uploading video we app and then continue, otherwise, user will **not go to the web app**.

- If the user is authenticated, an **HTML** page will be rendered to the users so that he can pick a video to upload.

- If the user clicks "**submit**", a response then will be sent to the filesystem to save the video, and the data of the video will be sent to the database.

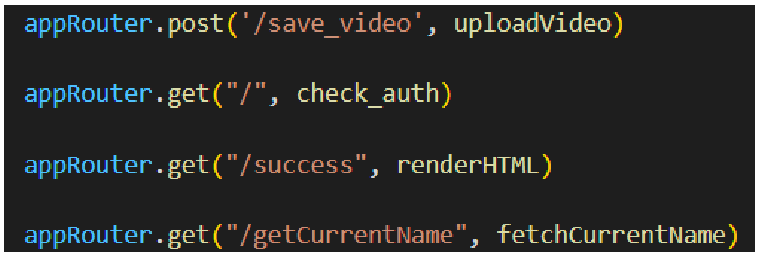
- The following explains the **process of uploading the video** using the video uploading web app:



http://localhost:5001/api/stream
user will send request to the web app
Video Uploading
http://localhost:5000/api/auth
app will redirect the request to the authentication service
Authentication Service
Will load an HTML page to enter the username and password
Untill entering an authenticated username and password
http://localhost:5000/api/auth/login
user will pick some video
Filesystem Service
Save it to the filesystem
Upload HTML Page
http://localhost:5001/api/success
Authentication Success
Auth HTML Page
Authentication Fatiled
Save the video name and path in the database
MySQL Database

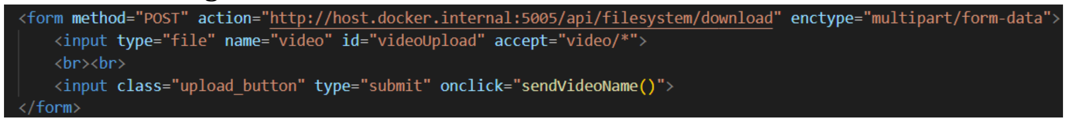
  - First, we will go to the endpoint **5001/api/upload**
  - Then the video upload service will redirect the response to the authentication service at the endpoint **5000/api/auth**
  - This endpoint will render and **HTML** page to **enter the data**
  - After that, a response to **5000/api/auth/login** will be sent to check the authentication.
  - If authenticated user, then a response will be sent to **5001/api/upload/success**, otherwise, a message will appear to the user to enter valid data, and then the user will be able to upload the videos on this endpoint.

- Next, let us have a look to the code:



```
appRouter.post('/save_video', uploadVideo)

appRouter.get("/", check_auth)

appRouter.get("/success", renderHTML)

appRouter.get("/getCurrentName", fetchCurrentName)
```

  The above is the routes of the video uploading service.
  - **(/save_video)**: will save the video information in the database, and the video itself will be saved through a request to the filesystem from the **HTML** itself as the following:

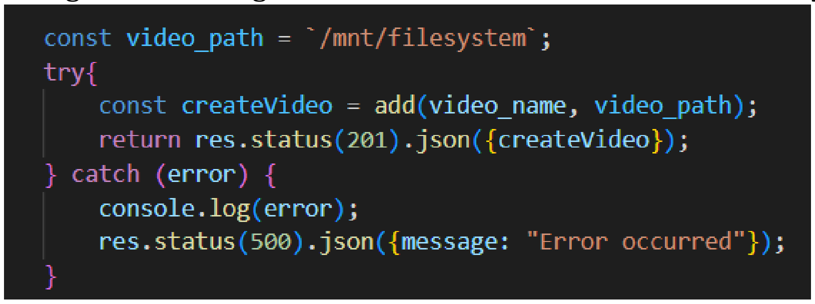
    ```
    <form method="POST" action="http://host.docker.internal:5005/api/filesystem/download" enctype="multipart/form-data">
        <input type="file" name="video" id="videoUpload" accept="video/*">
        <br><br>
        <input class="upload_button" type="submit" onclick="sendVideoName()">
    </form>
    ```

    Here, we have the action attribute which will send the request to the filesystem (after the user clicked the submit button) and the **filesystem will save the video**.
    Also, if the user clicked the **submit** button, the function **sendVideoName()** will be executed (which will send a request to "**/api/upload/save_video**"
  - **(/)**: will send a request to the authentication service at "**5000/api/auth/**" to check the authentication.

  - **(/succcess)**: this will render an **HTML** page to the user so that he can choose a video from his/her device and upload it.

  - **(/getCurrentName)**: will return the **encrypted name** of the uploaded video and the filesystem service will use this endpoint.

- Using the following method, we will save the video in the **MySQL** database:



```
const video_path = `/mnt/filesystem`;
try{
    const createVideo = add(video_name, video_path);
    return res.status(201).json({createVideo});
} catch (error) {
    console.log(error);
    res.status(500).json({message: "Error occurred"});
}
```

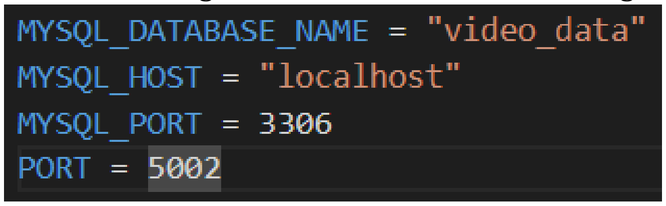
  add method is a query to the **MySQL** database and we will send the video **name** and video **path** to this method.

5. Streaming Video Web App:

- Finally, we will talk about the **Streaming** Video Web App, which will allow the authenticated users to view the videos that have been uploaded on the filesystem and the database.

- For this service, we will use the following modules:
  - **The other previous modules.**
  - **Node-fetch**: which supports promises and can be used to fetch data from APIs or interact with other services

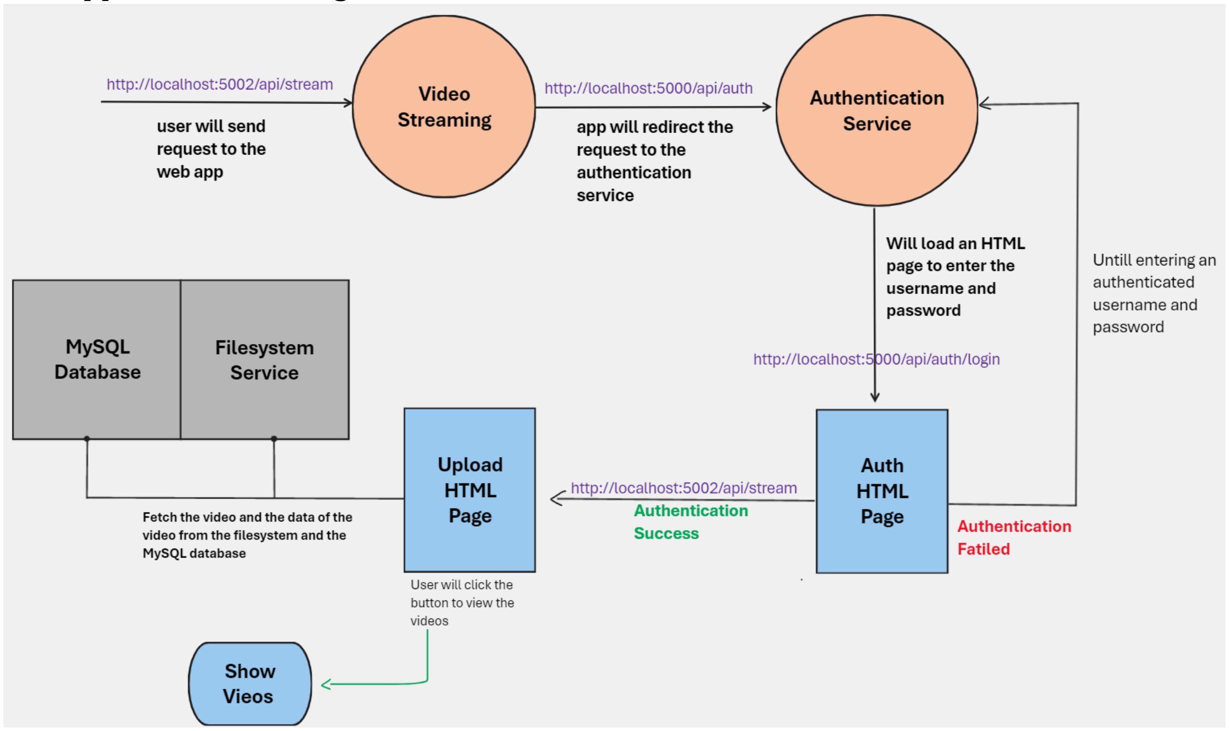
- The **.env** configuration will be as the following:



```
MYSQL_DATABASE_NAME = "video_data"
MYSQL_HOST = "localhost"
MYSQL_PORT = 3306
PORT = 5002
```

  We will be using port **5002** for this service, and the remaining are database configurations.

- The **workflow** of this service (web app) will be very similar to the uploading video web app, as the following:


http://localhost:5002/api/stream
user will send request to the web app
Video Streaming
http://localhost:5000/api/auth
app will redirect the request to the authentication service
Authentication Service
Will load an HTML page to enter the username and password
http://localhost:5000/api/auth/login
Untill entering an authenticated username and password
MySQL Database
Filesystem Service
Upload HTML Page
http://localhost:5002/api/stream
Authentication Success
Auth HTML Page
Authentication Fatiled
Fetch the video and the data of the video from the filesystem and the MySQL database
User will click the button to view the videos
Show Vieos

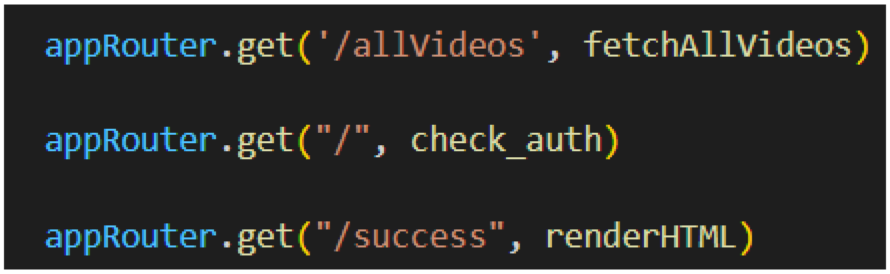
  This web app will also deal with the **authentication** service, the **filesystem** service and **mysql** database.

- When the user goes to the endpoint **5002/api/stream**, a request will be sent to the authentication service to make sure that the user has the right access, and the user is authenticated user

- If the user is authenticated, then a request to the endpoint **5002/api/stream/success** will be sent, which will render an **HTML** page that allows the user to show the videos when clicking a **button** on this **HTML.**

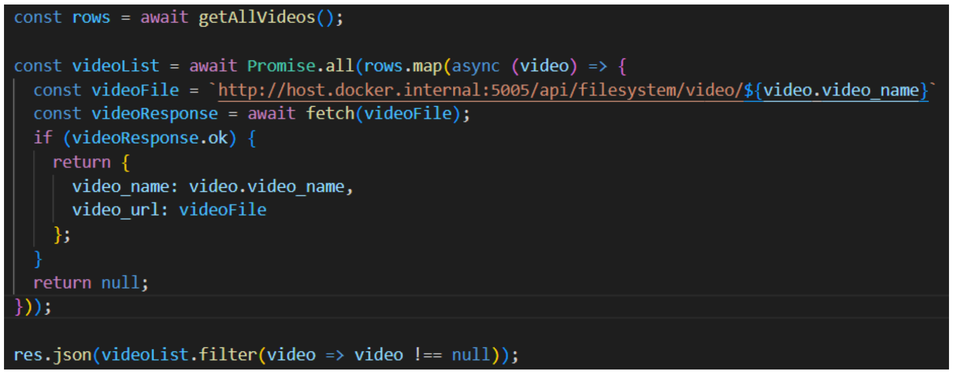
- If some videos in the database **don't exist in the filesystem**, a message will be logged that these videos do not exist in the filesystem.

- The routes of this service will be the following:



```
appRouter.get('/allVideos', fetchAllVideos)

appRouter.get("/", check_auth)

appRouter.get("/success", renderHTML)
```

  - **('/allVideos')**: which will fetch all the videos from the database (**names** and **paths**) and request these videos from the filesystem, as the following:



```
const rows = await getAllVideos();

const videoList = await Promise.all(rows.map(async (video) => {
  const videoFile = `http://host.docker.internal:5005/api/filesystem/video/${video.video_name}`
  const videoResponse = await fetch(videoFile);
  if (videoResponse.ok) {
    return {
      video_name: video.video_name,
      video_url: videoFile
    };
  }
  return null;
}));

res.json(videoList.filter(video => video !== null));
```

  The value of the "**rows**" will be all the current rows in the table **video_info** (which is all the videos that we've uploaded so far), we will iterate over all the rows, and for each row we will have "**video**" which is the current row, then from the filesystem we will fetch this video.

  After that we will return the list of all the videos, and the null videos will be filtered out.

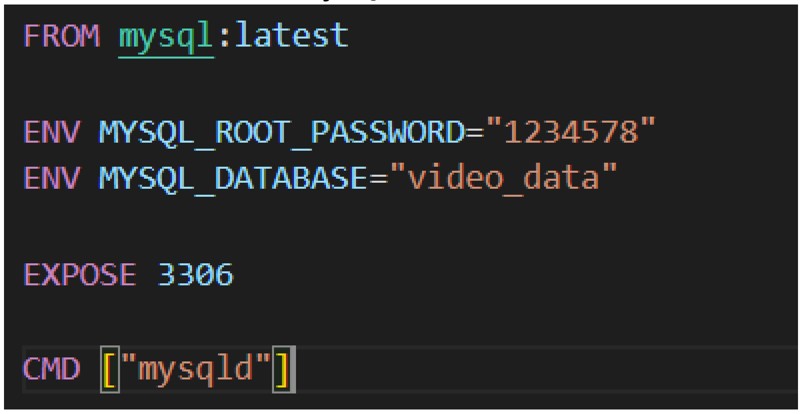
  - **('/' & '/success')**: these two endpoints will just do the same thing as the video uploading service, which is first authentication on **'/'** and then the **HTML** on **'/success'**

Docker Files

Next, we will look at the Dockerfile of each service and explain it:

1. MySQL

The Dockerfile of the **MySQL** database will be as the following:

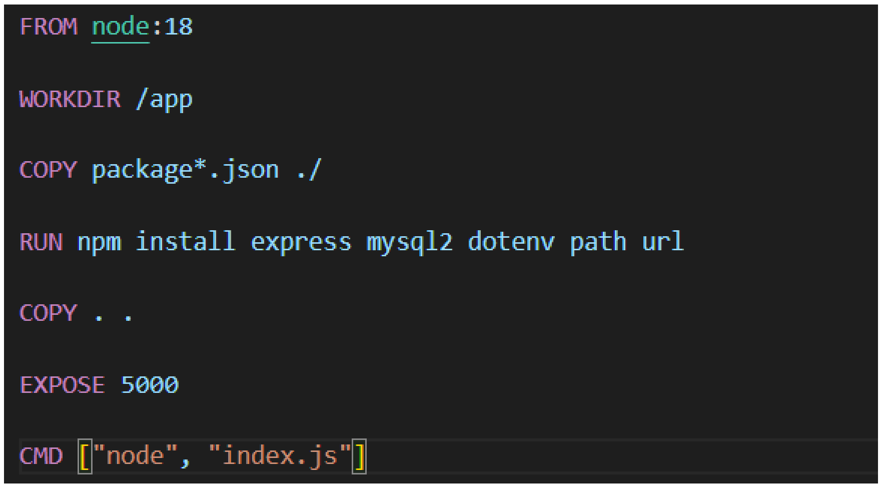


```
FROM mysql:latest

ENV MYSQL_ROOT_PASSWORD="1234578"
ENV MYSQL_DATABASE="video_data"

EXPOSE 3306

CMD ["mysqld"]
```

We will use the image **mysql** to create our database, and then we will just configure the **environment variables**, inform that this docker container will lister on port **(3306)** and finally the command that will run when the container starts which is "**mysqld**"

2. Authentication

The Dockerfile of the **Authentication** service will be as the following:



```
FROM node:18

WORKDIR /app

COPY package*.json ./

RUN npm install express mysql2 dotenv path url

COPY . .

EXPOSE 5000

CMD ["node", "index.js"]
```

We will use the image **node:18** as the base image of our authentication service, and then we will set up a working directory to be **/app**, in this directory, we will just copy all the **package*.json** (all the dependencies of our app) to the working directory (which is **/app**)

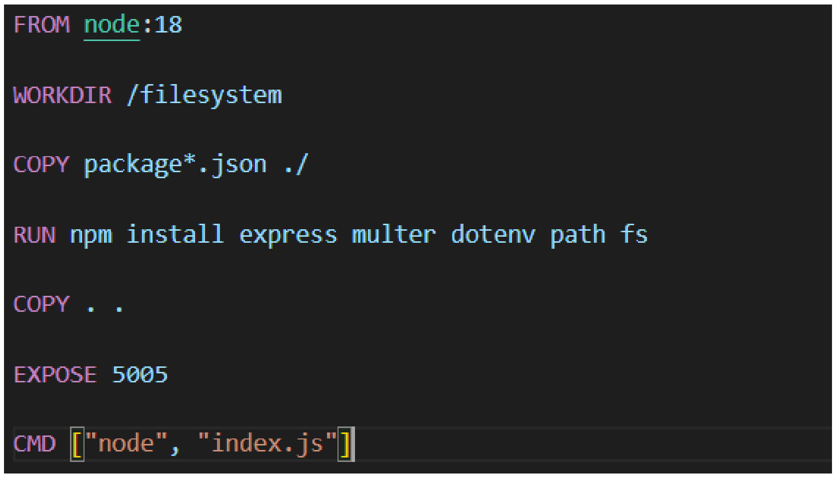
Then we install the following modules (using **npm install <module>**):

- **express**: Node.js framework
- **mysql2**: MySQL database client library
- **dotenv**: for handling the environment variables
- **path**: for handling the file paths
- **url**: for working with the **URLs.**

And then we will expose the port (**5000**) to tell that the container is listening on this port, and finally when the container starts, "**node index.js**" command will be executed.

3. Filesystem:

The Dockerfile of the **Filesystem** service will be as the following:



```
FROM node:18

WORKDIR /filesystem

COPY package*.json ./

RUN npm install express multer dotenv path fs

COPY . .

EXPOSE 5005

CMD ["node", "index.js"]
```

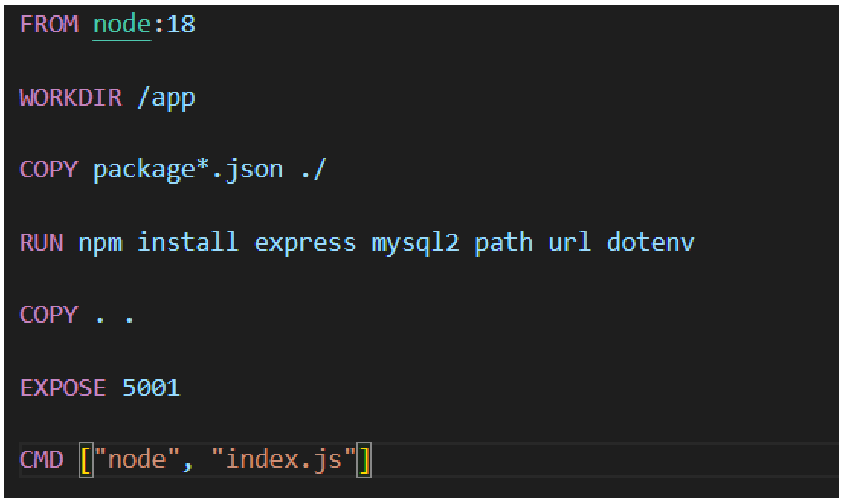
We will also use the image **node:18** as the base image of our filesystem service, and the working directory will be **/filesystem**, in this directory, we will copy all the **package.json** and **package-lock.json** to the working directory, then we install the following npm modules (also using **npm install <module>**):

- **express**: Node.js framework
- **multer**: middleware used for handling multipart/form-data (primarily used for uploading files)
- **fs**: provides functions for interacting with the file system (we will use it just for the read).
- **dotenv**: for handling the environment variables
- **path**: for handling the file **paths**

Then we will expose port (**5005**) and finally when the container starts, "**node index.js**" command will be executed.

4. Video Uploading:

The Dockerfile of the **Video Uploading** service will be as the following



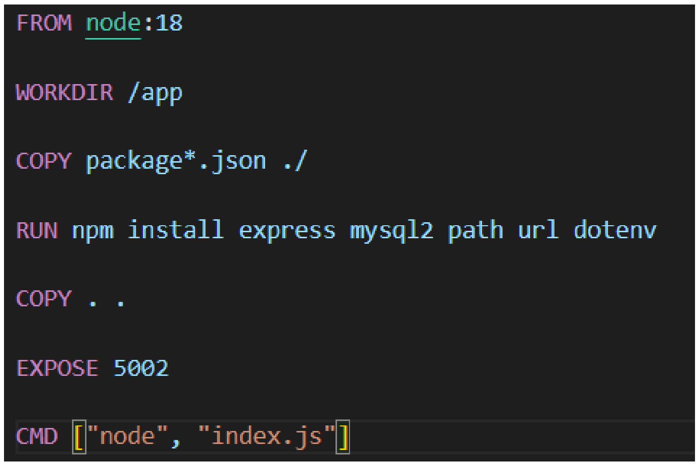
```
FROM node:18

WORKDIR /app

COPY package*.json ./

RUN npm install express mysql2 path url dotenv

COPY . .

EXPOSE 5001

CMD ["node", "index.js"]
```

This Dockerfile will be very **similar** to the previous Dockerfile, but the npm modules that we need for this service is:

- **express**: Node.js framework
- **mysql2**: MySQL database client library
- **dotenv**: for handling the environment variables
- **path**: for handling the file paths
- **url**: for working with the URLs.

Then we will export the port (**5001**) for this service to tell that this container is listening on this port.

5. Video Streaming:



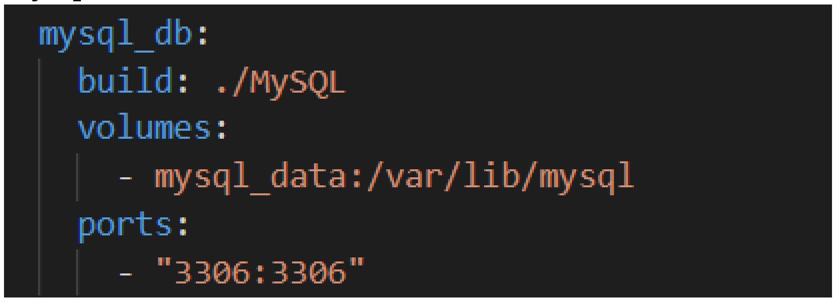
```
FROM node:18

WORKDIR /app

COPY package*.json ./

RUN npm install express mysql2 path url dotenv

COPY . .

EXPOSE 5002

CMD ["node", "index.js"]
```

Same as the previous Dockerfile (**Dockerfile of the Video Uploading**).

Docker Compose

And now, we will see the docker compose file and explain it:

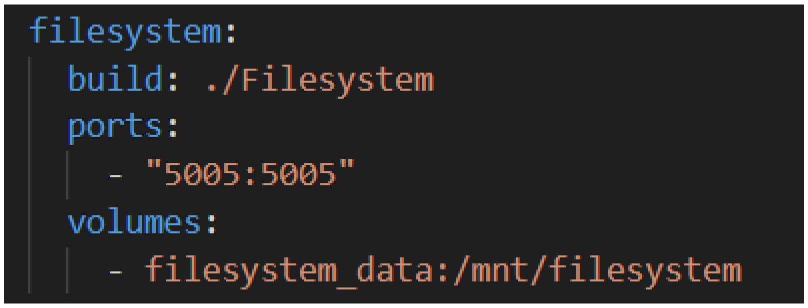
- **mysql_db**



```yaml
mysql_db:
  build: ./MySQL
  volumes:
    - mysql_data:/var/lib/mysql
  ports:
    - "3306:3306"
```

We will just determine the **path of the Dockerfile** of the mysql database located, and then add a volume to the mysql (**mysql_data**) so that the data will be saved even after the **termination**, and finally we will determine the **port** that this service will be listening to.

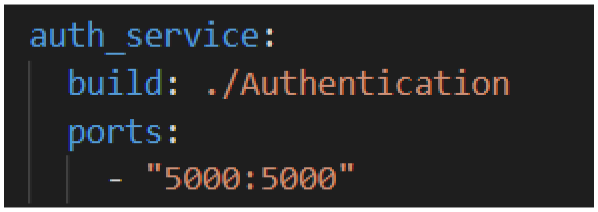
- **Filesystem**



```yaml
filesystem:
  build: ./Filesystem
  ports:
    - "5005:5005"
  volumes:
    - filesystem_data:/mnt/filesystem
```

Same as previously, but we will determine the **port** of this service to be **5005**, and the **volume** to be (**filesystem_data**) to persist the uploaded videos (**making it stateful**).

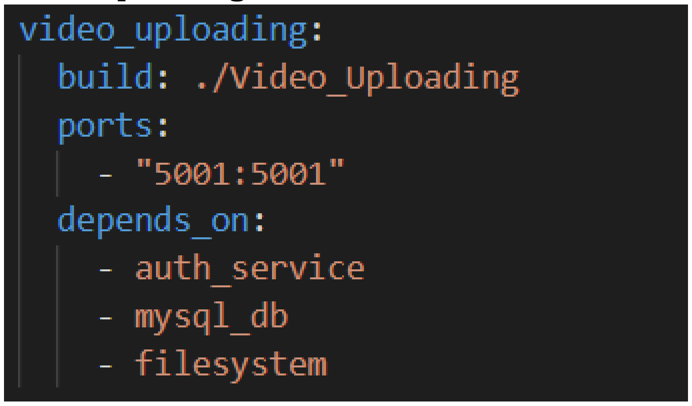
- **auth_service**



```yaml
auth_service:
  build: ./Authentication
  ports:
    - "5000:5000"
```

Same as previously, we will just determine the **location** of the **Dockerfile** of the authentication service and determine the port that will be listening to.

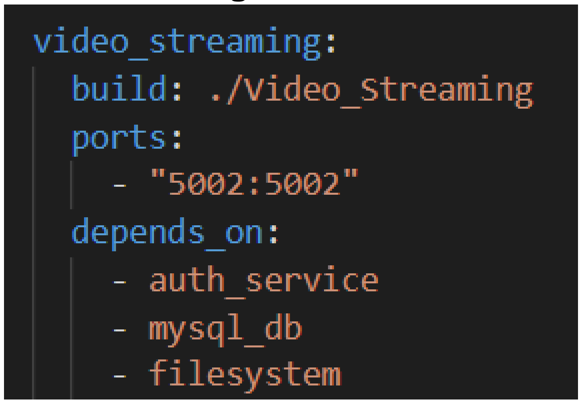
- **video_uploading**



```
video_uploading:
  build: ./Video_Uploading
  ports:
    - "5001:5001"
  depends_on:
    - auth_service
    - mysql_db
    - filesystem
```

This service will be created after the **auth_service**, **mysql_db** and **filesystem** are all created, then we will create this service and that is because this service depends on them.

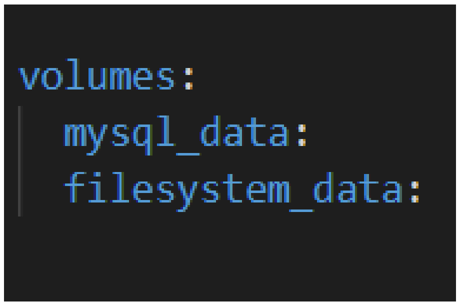
- **video_streaming**



```
video_streaming:
  build: ./Video_Streaming
  ports:
    - "5002:5002"
  depends_on:
    - auth_service
    - mysql_db
    - filesystem
```

Same as the previous Dockerfile, we will make it **depend on the filesystem, mysql_db and auth_servic**e.

- **volumes**



```
volumes:
  mysql_data:
  filesystem_data:
```

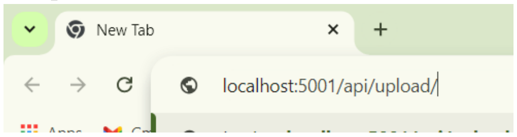
Finally, there are the volumes that we will create for each of the **filesystem** and **mysql_db**.

Also, a network will be automatically created by Compose for those services so that they can talk to each other.

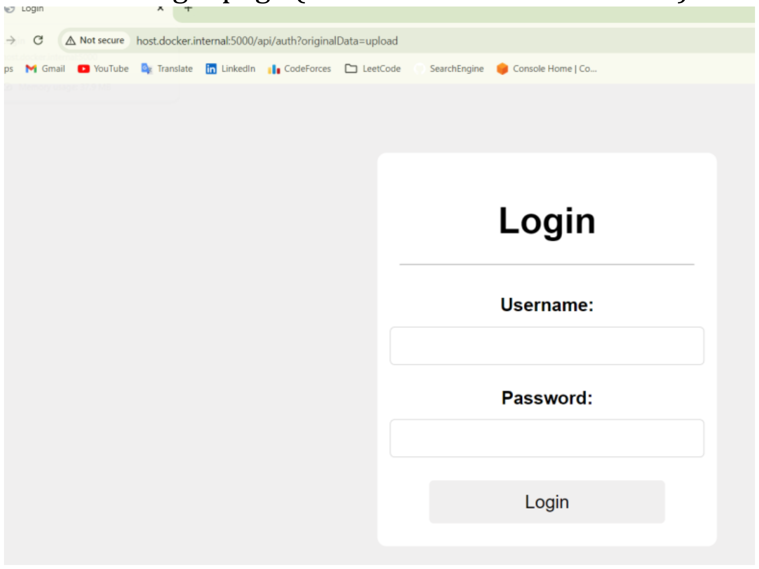
Testing

Finally, let see how to flow of the uploading and the streaming:

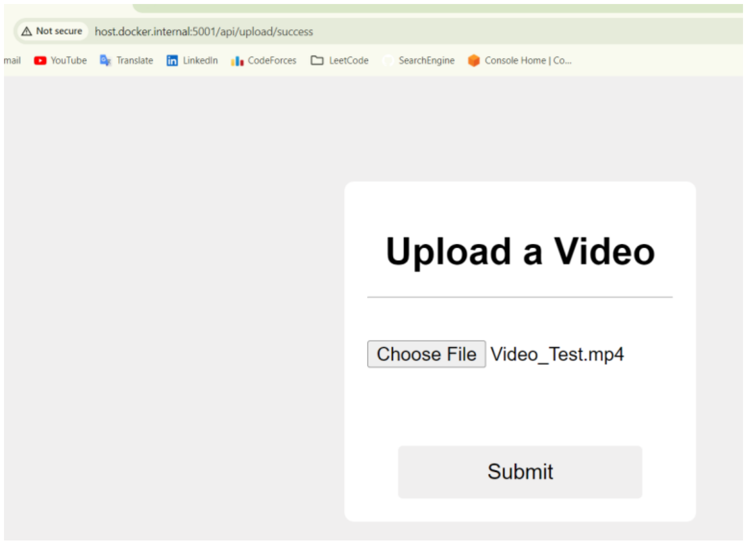
- **Video Uploading:**
  First, we will go to this endpoint:



Which will take us to the login page (the **authentication** service):



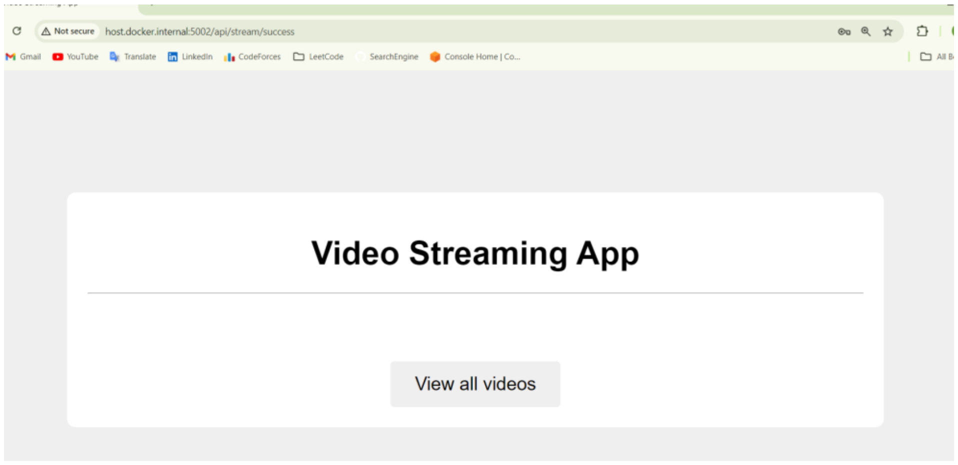
And after the authentication, we will **get back to the uploading** service like the following:



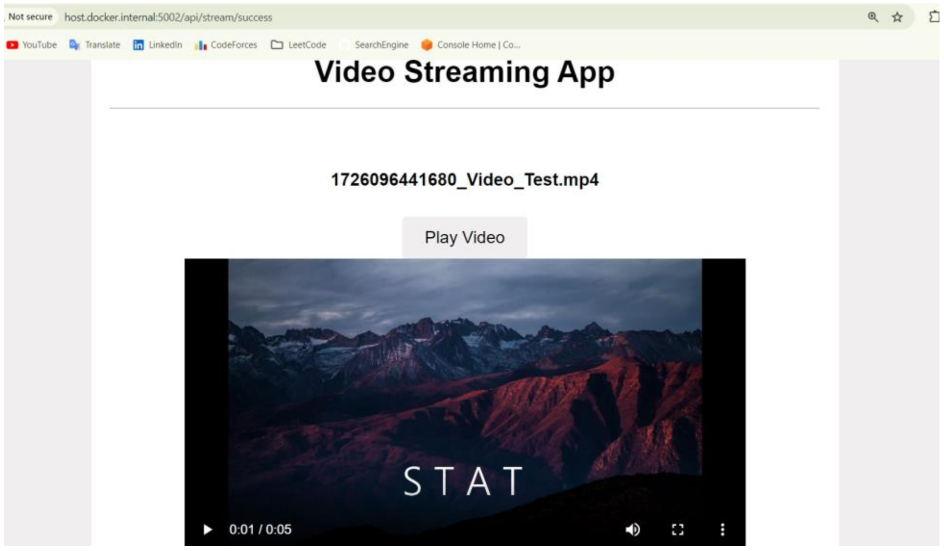
And after submitting, the **video will be saved** on the filesystem, and the data of the video will be saved on the **MySQL** database.

- **Video Uploading:**

  We will go to the endpoint http://localhost:5002/api/stream and after the **validation**, this window will be rendered:


Not secure host.docker.internal:5002/api/stream/success
Gmail YouTube Translate LinkedIn CodeForces LeetCode SearchEngine Console Home | Co...
Video Streaming App
View all videos

And clicking on the "**View all videos**" will show the list of all videos that have been uploaded (*we only upload one video so far*):


Not secure host.docker.internal:5002/api/stream/success
YouTube Translate LinkedIn CodeForces LeetCode SearchEngine Console Home | Co...
Video Streaming App
1726096441680_Video_Test.mp4
Play Video
STAT
0:01 / 0:05

Thanks

Ahmad Nabeel Al-Jaber