ACIT 3495 - Project 1 Technical Report

Group 19

Jaguar Perlas

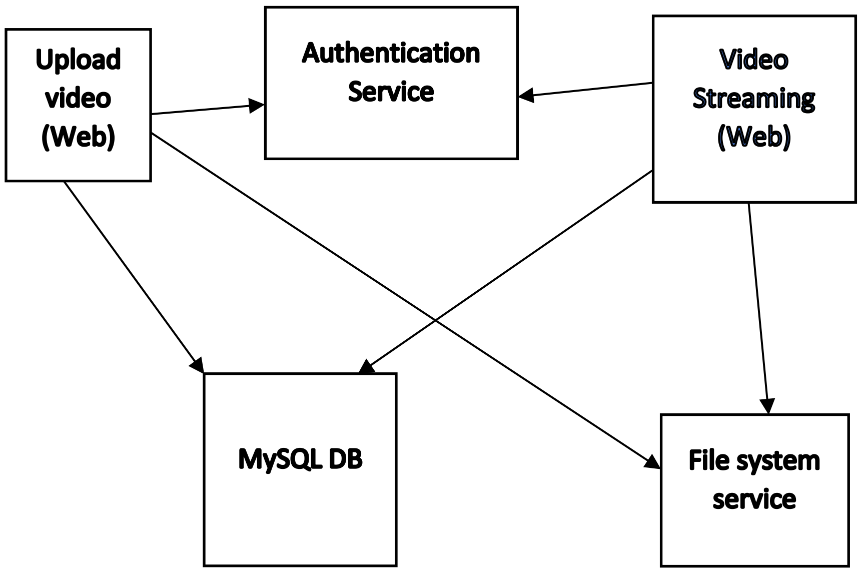
Roy Ortega

Felix Ruttan

Contents

Overview

For this Docker project, we have decided to use the following diagram to build our solution:



We have decided to proceed with a Video Streaming System consisting of 5 services. For each of the service shown in the diagram, we have used the following technologies:

Upload video (Web) - Node.JS

MySQL DB - MySQL DB

Authentication Service - Python with Flask

Video Streaming (Web) - Node.JS

File system service - Python with Flask

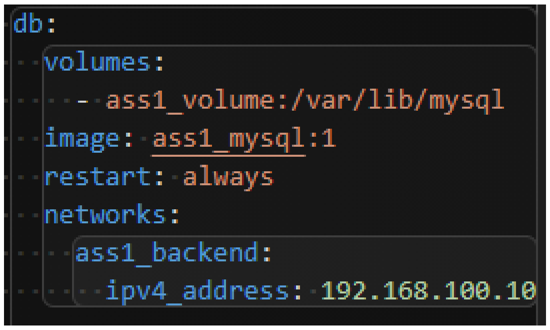
Each service will have their own *Dockerfile* to build their own images. Each image will then come together with a *docker-compose.yml* file to containerized and communicate with each other. The next sections will explain how our system works and how Docker is configured.

Docker Configuration

docker-compose.yml

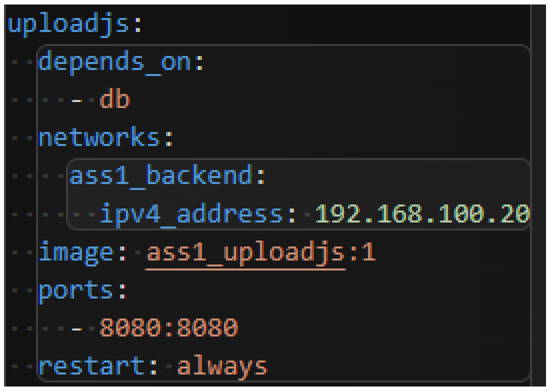
Our *docker-compose.yml* file contains 5 services, 1 volume and 1 network:

db - Our MySQL service responsible for storing file names and paths. This has a volume attached named *ass1_volume* with a path of */var/lib/mysql*. This service is also attached to a network named *ass1_backend* with a static IP address of *192.168.100.10*. The image to be run is named *ass1_mysql:1*.



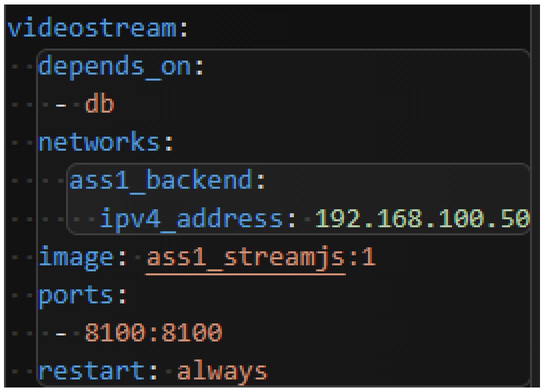
```
db:
  volumes:
    - ass1_volume:/var/lib/mysql
  image: ass1_mysql:1
  restart: always
  networks:
    ass1_backend:
      ipv4_address: 192.168.100.10
```

uploadjs - Our Node.JS service responsible for receiving video files from external clients. This service is attached to a network named *ass1_backend* with a static IP address of *192.168.100.20*. Since this service will be outfacing, it has an external port of 8080. The image to be run is named *ass1_uploadjs:1.*



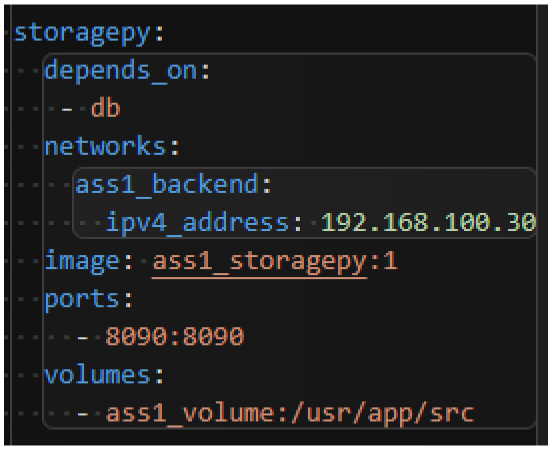
```
uploadjs:
  depends_on:
    - db
  networks:
    ass1_backend:
      ipv4_address: 192.168.100.20
  image: ass1_uploadjs:1
  ports:
    - 8080:8080
  restart: always
```

videostream - Our Node.JS service responsible for streaming video files from our storage for external clients to watch. This service is attached to a network named *ass1_backend* with a static IP address of *192.168.100.50*. Since this service will be outfacing, it has an external port of 8090. The image to be run is named *ass1_streamjs:1.*



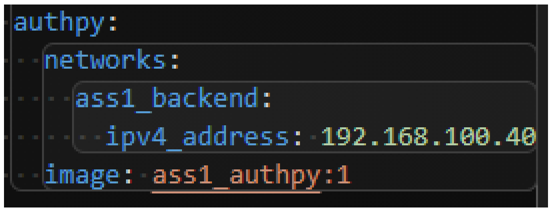
```yaml
videostream:
  depends_on:
    - db
  networks:
    ass1_backend:
      ipv4_address: 192.168.100.50
  image: ass1_streamjs:1
  ports:
    - 8100:8100
  restart: always
```

storagepy - Our Python service responsible for storing video files. This service is attached to a network named *ass1_backend* with a static IP address of 192.168.100.30. This service has a volume attached named *ass1_volume* with a path of /usr/app/src.



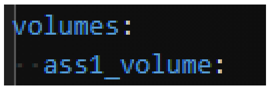
```yaml
storagepy:
  depends_on:
    - db
  networks:
    ass1_backend:
      ipv4_address: 192.168.100.30
  image: ass1_storagepy:1
  ports:
      - 8090:8090
  volumes:
      - ass1_volume:/usr/app/src
```

authpy - Our Python service responsible for authenticating users at login. This service is attached to a network named *ass1_backend* with a static IP address of 192.168.100.40. The image to be run is named *ass1_authpy:1*



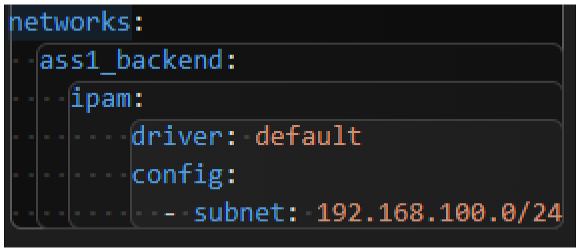
```yaml
authpy:
  networks:
    ass1_backend:
      ipv4_address: 192.168.100.40
  image: ass1_authpy:1
```

ass1_volume - Our volume to be used for storing MySQL data and video files



```yaml
volumes:
  ass1_volume:
```

ass1_backend - Our network to be used for communication between services. Each service has their own static IP address within the subnet *192.168.100.0/24.*

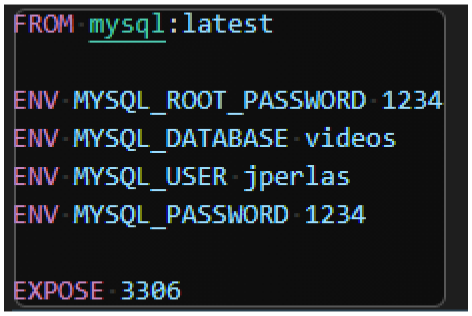


```yaml
networks:
  ass1_backend:
    ipam:
        driver: default
        config:
          - subnet: 192.168.100.0/24
```

Dockerfiles

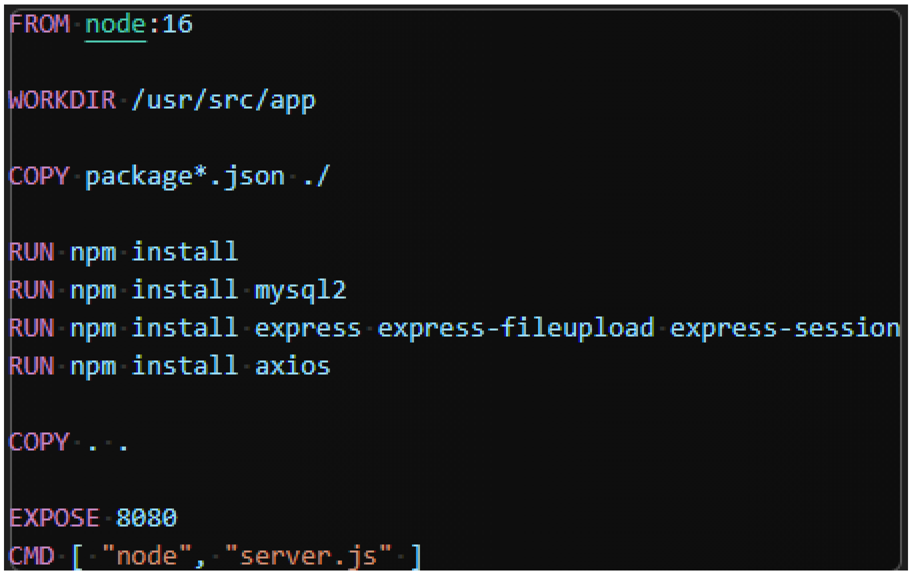
Each service has their own *Dockerfile* to build their own images.

db - A simple *Dockerfile*, containing MySQL version, environment variables, and a port to expose.



```dockerfile
FROM mysql:latest

ENV MYSQL_ROOT_PASSWORD 1234
ENV MYSQL_DATABASE videos
ENV MYSQL_USER jperlas
ENV MYSQL_PASSWORD 1234

EXPOSE 3306
```

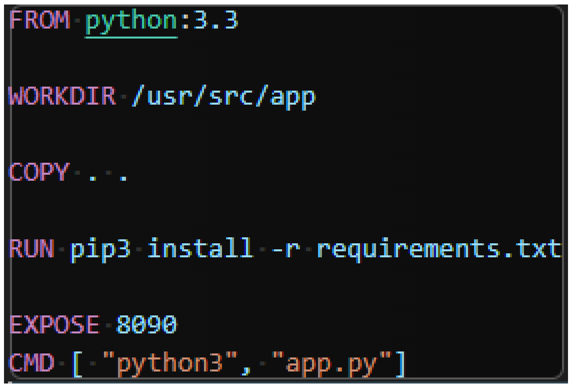
uploadjs - This *Dockerfile* installs Node.JS version 16, uses a work directory of */usr/src/app*, copies files from our local machine to the container, installs Node.JS dependencies, exposes port 8080, and runs *server.js*.



```
FROM node:16

WORKDIR /usr/src/app

COPY package*.json ./

RUN npm install
RUN npm install mysql2
RUN npm install express express-fileupload express-session
RUN npm install axios

COPY . .

EXPOSE 8080
CMD [ "node", "server.js" ]
```

videostream - This *Dockerfile* is the same as our *uploadjs,* except exposing port 8100.

storagepy - This *Dockerfile* installs Python 3.3, uses a work directory of */usr/src/app*, copies files from our local machine to the container, installs Python libraries from *requirements.txt*, exposes port 8090, and runs *app.py*.



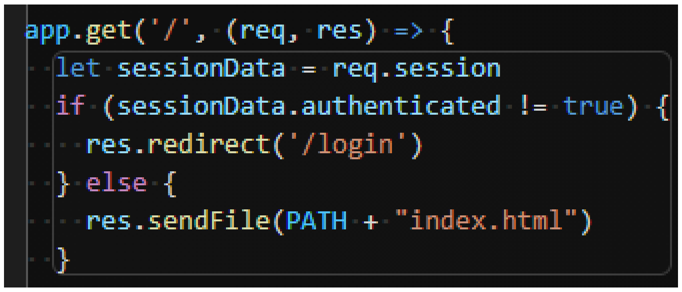
```
FROM python:3.3

WORKDIR /usr/src/app

COPY . .

RUN pip3 install -r requirements.txt

EXPOSE 8090
CMD [ "python3", "app.py"]
```

authpy - This *Dockerfile* is the same as *storagepy*, except exposing port 8110.

Services And How Our System Works

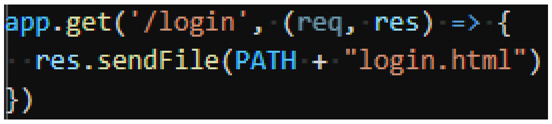
uploadjs

Our system first starts with our *uploadjs* service. As users first visit http://localhost:8080, the service will check their browser session if they are already authenticated. This is done by using the *express-session* module, allowing us to store session data and check if a variable named *authenticated* is not True. If that variable is not True, they will be redirected to http://localhost:8080/login. Otherwise, the service will render the root *index.html* file.

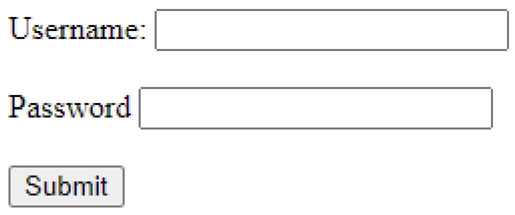


```
app.get('/', (req, res) => {
  let sessionData = req.session
  if (sessionData.authenticated != true) {
    res.redirect('/login')
  } else {
    res.sendFile(PATH + "index.html")
  }
```

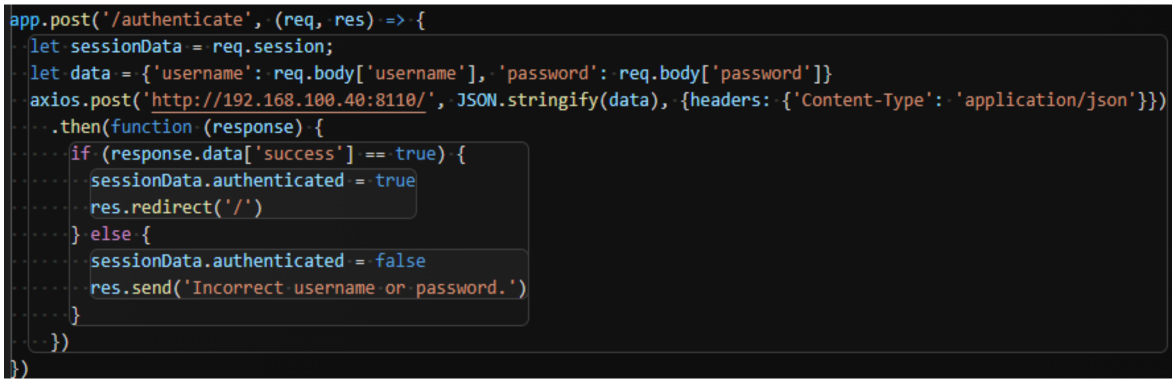
Assuming the user has not been authenticated, the service will send a GET request to */login* and render the *login.html* file, containing two simple text fields where users can enter their username and password.



```
app.get('/login', (req, res) => {
  res.sendFile(PATH + "login.html")
})
```

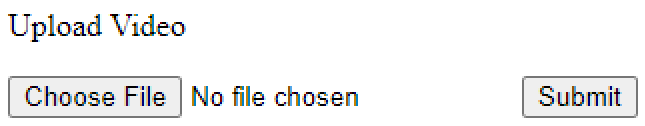


Filling out username and password fields and clicking *Submit* will send a POST request to */authenticate.* Grabbing the username and password from the request, our service will send a POST request to http://192.168.100.40:8110/, our authpy service, passing username and password into a JSON object. Our *authpy* service will return a Boolean, indicating whether the user is authenticated or not. Our *uploadjs* service will then grab the Boolean and store it into the session variable *authenticated* and either grant access or inform the user about incorrect credentials.



```
app.post('/authenticate', (req, res) => {
  let sessionData = req.session;
  let data = {'username': req.body['username'], 'password': req.body['password']}
  axios.post('http://192.168.100.40:8110/', JSON.stringify(data), {headers: {'Content-Type': 'application/json'}})
    .then(function (response) {
      if (response.data['success'] == true) {
        sessionData.authenticated = true
        res.redirect('/')
      } else {
        sessionData.authenticated = false
        res.send('Incorrect username or password.')
      }
    })
})
```

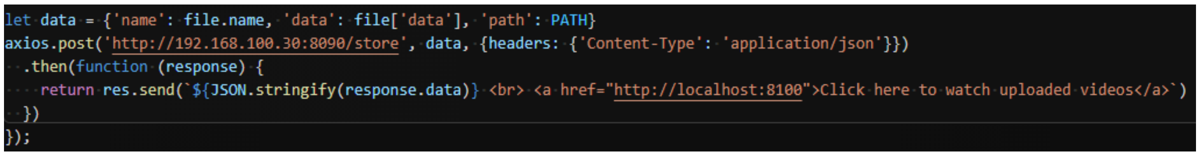
After authentication, the user will now be able to upload a video file to our system.



Submitting a video file will send a POST request to */upload*, responsible for querying our *db* service and sending the file to our *storagepy* service. Our *uploadjs* service will first make a connection to the MySQL database using host as *192.168.100.10*, user as *jperlas*, and password as *1234*. Then it will make three SQL queries in order, dependent on the query above to succeed:

1. *CREATE DATABASE IF NOT EXISTS videos*
2. *CREATE TABLE IF NOT EXISTS videos.files (id INT(6) AUTO_INCREMENT PRIMARY KEY, name VARCHAR(255) NOT NULL, path VARCHAR(255) NOT NULL)*
3. *INSERT INTO videos.files (name, path) VALUES ("${file.name}", "${PATH}")*

After these SQL queries have succeeded, *uploadjs* will send a POST request to *http://192.168.100.30:8090/store*, sending the file name, path (hard coded to be */usr/src/app)*, and the binary array of the video file. On success, *uploadjs* will dynamically generate HTML informing the user of success and a link to watch uploaded videos.



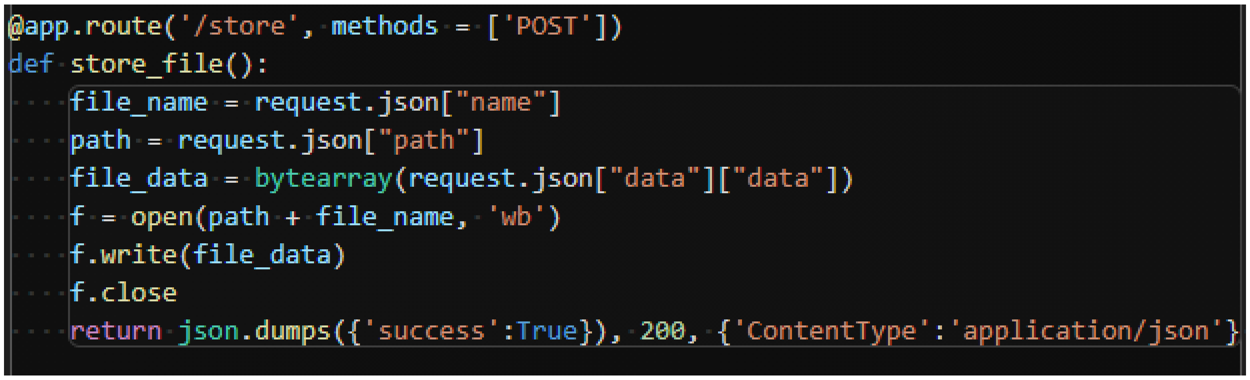
```
let data = {'name': file.name, 'data': file['data'], 'path': PATH}
axios.post('http://192.168.100.30:8090/store', data, {headers: {'Content-Type': 'application/json'}})
  .then(function (response) {
    return res.send(`${JSON.stringify(response.data)} <br> <a href="http://localhost:8100">Click here to watch uploaded videos</a>`)
  })
});
```

storagepy

This service acts upon receiving video files from *uploadjs* service and getting video files for *videostream* service.

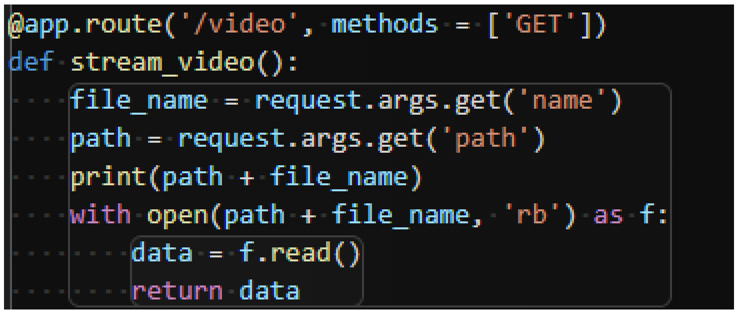
If *storagepy* service receives a POST request from *uploadjs* service, *storagepy* will extract the file name, path, and binary array from the received JSON data and creates a new file with the same name as the video and writing the binary array to said file. The file will be written to path /usr/src/app, a path that was passed from *uploadjs.*

The binary array is the video content of the sent video file. Writing this binary array to another file will contain the same video, in other words, copying it.



```
@app.route('/store', methods = ['POST'])
def store_file():
    file_name = request.json["name"]
    path = request.json["path"]
    file_data = bytearray(request.json["data"]["data"])
    f = open(path + file_name, 'wb')
    f.write(file_data)
    f.close
    return json.dumps({'success':True}), 200, {'ContentType':'application/json'}
```

If *storagepy* receives a GET request from *videostream* service, it will grab the file according to the file name and path query parameters in the URL. The file is then read and contents are returned.

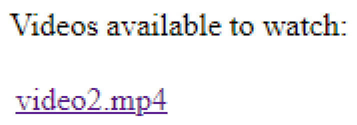


```
@app.route('/video', methods = ['GET'])
def stream_video():
    file_name = request.args.get('name')
    path = request.args.get('path')
    print(path + file_name)
    with open(path + file_name, 'rb') as f:
        data = f.read()
        return data
```

In the *videostream* service, the data is streamed/piped as a response, resulting in video streaming from *storagepy*. Explained in the next section.

videostream

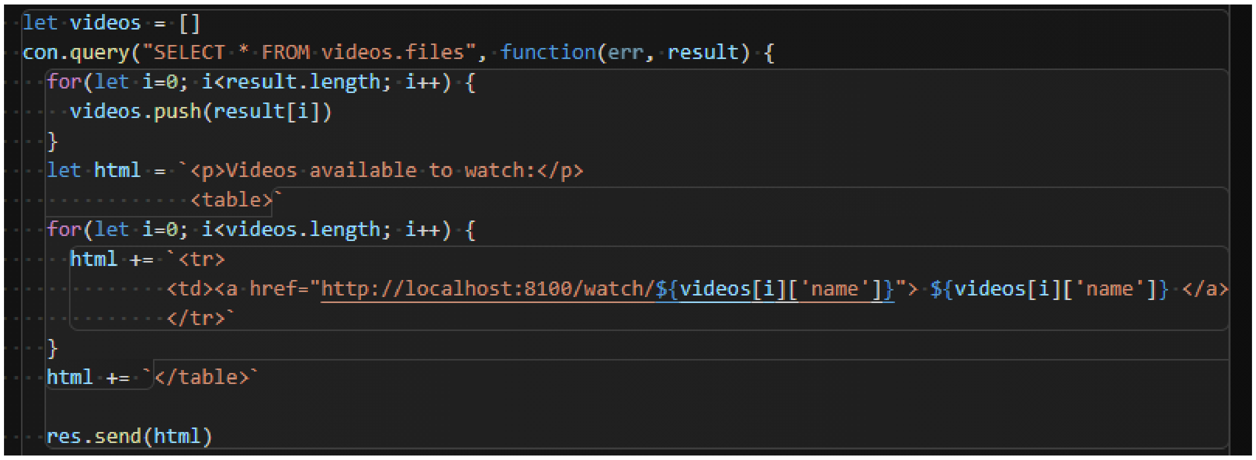
After uploading a video to *uploadjs*, users can be directed to the *videostream* service. Users are required to login once again, following the same process as explained in *uploadjs.* After authentication users can access uploaded videos.



Videos available to watch:

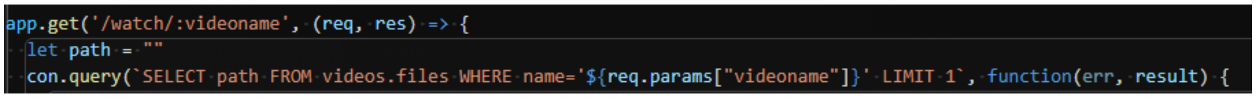
video2.mp4

Upon reaching “/”, the service queries the *db* service using “*SELECT * FROM videos.files”* to grab all uploaded videos. The service *videostream* then uses a for loop to generate HTML code (because Node.JS only serves static HTML) to show all videos and display it to the user.



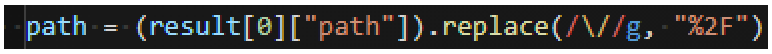
```
let videos = []
con.query("SELECT * FROM videos.files", function(err, result) {
  for(let i=0; i<result.length; i++) {
    videos.push(result[i])
  }
  let html = `<p>Videos available to watch:</p>
            <table>`
  for(let i=0; i<videos.length; i++) {
    html += `<tr>
            <td><a href="http://localhost:8100/watch/${videos[i]['name']}"> ${videos[i]['name']} </a>
            </tr>`
  }
  html += `</table>`

  res.send(html)
```

The generated HTML code also generates links that are unique to each video uploaded. Upon clicking a generated link, *videostream* will parse through the link to grab parameters and use it towards an SQL query to grab the stored path.



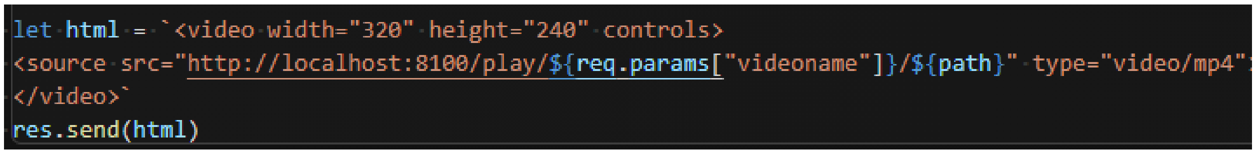
```
app.get('/watch/:videoname', (req, res) => {
  let path = ""
  con.query(`SELECT path FROM videos.files WHERE name='${req.params["videoname"]}' LIMIT 1`, function(err, result) {
```

After grabbing the path, *videostream* replaces all forward slashes "/" with "%2F". This is because the next GET request handler relies on URL query parameters. Having forward slashes will cause the browser to think is part of the URL path, redirecting users to the wrong website. "%2F" is the URL encoding for forward slash.



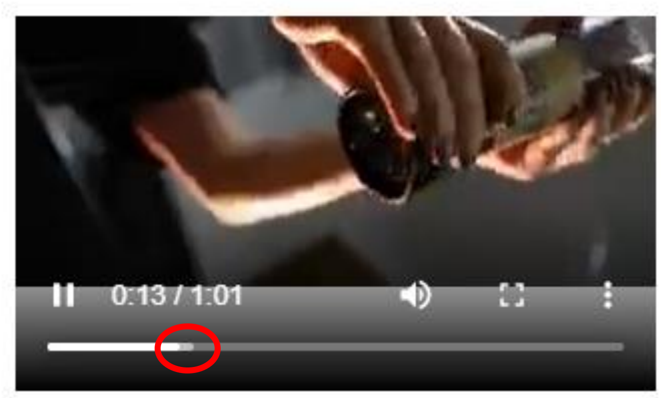
```
path = (result[0]["path"]).replace(/\//g, "%2F")
```

The service then dynamically generates HTML code to show a video player which has a source of a URL that automatically sends a GET request upon loading.



```
let html = `<video width="320" height="240" controls>
<source src="http://localhost:8100/play/${req.params["videoname"]}/${path}" type="video/mp4"
</video>`
res.send(html)
```

The *videostream* service handles the GET request and sends a GET request to the *storagepy* service. *Storagepy* will stream/pipe the binary array of the appropriate video file to *videostream* and *videostream* will display the already received binary data to the client while still receiving binary data from *storagepy*. Notice the buffering in the below screenshot.



This results in video streaming from the *storagepy* service to the *videostreaming* service.

authpy

Finally, our *authpy* service is a simple authentication service. Due to the project scope largely on Docker and functionality, security is not a priority.

Authpy receives POST requests on "/" and will check if received values match the username *"test"* and password *"123"*. If both username and password meet the requirements, *authpy* will return *{'success':True}, 200*. If username and password don't meet the requirements, *authpy* will send the opposite, False.

db

The *db* service is a simple MySQL database. This service communicates with the *uploadjs* service and the *videostream* service through SQL queries.

Running Our System

Each service needs to be built using a specific name. In their respective folders, build the image with their names as:

db - ass1_mysql:1

uploadjs - ass1_uploadjs:1

videostream - ass1_streamjs:1

storagepy - ass1_storagepy:1

authpy - ass1_authpy:1

After building the images, run the docker-compose.yml file with *docker-compose up -d* in the root folder. Then in a browser, go to *http://localhost:8080* to upload your first video file.

Default username and password for authentication:

Username: test

Password: 123