1. Play videos on client
    - Any YouTube URL
    - Steam your video, audio
    - Add video to s3 bucket and play video using the s3 url

2. Create Upload service
    - Upload media on s3 and test API using postman

3. Send upload request from client to service with file
4. Extract file from request in service and upload that to s3

1. **Play YouTube video / Stream your video using NextJS**

https://www.npmjs.com/package/react-player

```
'use client'
import React, { useState } from 'react'
import dynamic from 'next/dynamic'
const ReactPlayer = dynamic(() => import("react-player"), { ssr: false });

const Room = () => {

   const [userStream, setUserStream] = useState();

   const callUser = async () => {
       const stream = await navigator.mediaDevices.getUserMedia({
           audio: true,
           video: true
       })
       setUserStream(stream);
   }

   return (
       <div>
           <div className='m-10'>
               <ReactPlayer
                   width="1280px"
                   height="720px"
                   url="https://www.youtube.com/watch?v=e2fKYP_7B_Y&t=1s"
                   controls={true}
               />
           </div>
           <div className='m-10'>
               <ReactPlayer
                   width="1280px"
                   height="720px"

url="https://hhld-classes.s3.ap-south-1.amazonaws.com/Day+9.mp4"
                   controls={true}
               />
           </div>
           <button type="button"
               onClick={callUser}
               class="text-white bg-blue-700 hover:bg-blue-800 focus:ring-4
focus:ring-blue-300 font-medium rounded-lg text-sm px-5 py-2.5 me-2 mb-2
dark:bg-blue-600 dark:hover:bg-blue-700 focus:outline-none
dark:focus:ring-blue-800 m-10">Stream</button>
           <div className='m-10'>
```

```
                <ReactPlayer
                    width="1280px"
                    height="720px"
                    url={userStream}
                    controls={true}
                />
            </div>
        </div>
    )
}

export default Room
```

2. **Create Upload service -** https://www.npmjs.com/package/aws-sdk

   (The filepath and file key is hardcoded)

```
// index.js

import express from "express"
import uploadRouter from './routes/upload.route.js'
import dotenv from "dotenv"
import cors from "cors"

dotenv.config();
const app = express();
const port = process.env.PORT || 8080;

app.use(cors({
   allowedHeaders: ["*"],
   origin: "*"
}));
app.use(express.json());
app.use('/upload', uploadRouter);


app.get('/', (req, res) => {
   res.send('HHLD YouTube')
})

app.listen(port, () => {
   console.log(`Server is listening at http://localhost:${port}`);
})
```

```
// routes/upload.route.js
import express from "express"
import uploadFileToS3 from "../controllers/upload.controller.js";

const router = express.Router();
router.post('/', uploadFileToS3);
```

```
export default router;

// controllers/upload.controller.js

import AWS from 'aws-sdk';
import fs from 'fs'

const uploadFileToS3 = async(req, res) => {

 const filePath = '/Users/keertipurswani/GitHub/HHLD
Projects/hhld-youtube/server/assets/dsa.png';

   // Check if the file exists
   if (!fs.existsSync(filePath)) {
       console.log('File does not exist: ', filePath);
       return;
   }

   AWS.config.update({
       region: 'ap-south-1',
       accessKeyId: process.env.AWS_ACCESS_KEY_ID,
       secretAccessKey: process.env.AWS_SECRET_ACCESS_KEY
   });

   const params = {
       Bucket: process.env.AWS_BUCKET,
       Key: 'dsa.png',
       Body: fs.createReadStream(filePath)
   };

   const s3 = new AWS.S3();

   // Upload the file to S3
   s3.upload(params, (err, data) => {
       if (err) {
           console.log('Error uploading file:', err);
           res.status(404).send('File could not be uploaded!');
```

```
        } else {
            console.log('File uploaded successfully. File location:',
data.Location);
            res.status(200).send('File uploaded successfully');
        }
    });
}

export default uploadFileToS3;
```

3. Basic FrontEnd UI to select and send File in Upload Req

```
// pages/uploadPage.jsx

import UploadForm from '../components/uploadForm';

const UploadPage = () => {
 return (
   <div className='m-10'>
     <h1>HHLD YouTube - Upload Page</h1>
     <UploadForm/>
   </div>
 );
};

export default UploadPage;
```

```
// components/uploadForm.jsx
"use client"
import React, {useState}  from 'react'
import axios from 'axios';

const UploadForm = () => {

   const [selectedFile, setSelectedFile] = useState(null);

   const handleFileChange = (e) => {
       setSelectedFile(e.target.files[0]);
   };

   const handleSubmit = (e) => {
       e.preventDefault();
     handleFileUpload(selectedFile);
   };
```

```
   const handleFileUpload = async (file) => {
       try {
         const formData = new FormData();
         formData.append('file', file);
         console.log('Going to upload file to server');
         const res = await axios.post('http://localhost:8080/upload',
formData, {
           headers: {
             'Content-Type': 'multipart/form-data'
           }
         });
         console.log(res.data);
       } catch (error) {
         console.error('Error uploading file:', error);
       }
     };

 return (
   <div>
       <form onSubmit={handleSubmit}>
           <input type="file" onChange={handleFileChange} />
           <button type="submit"
                   className="text-white bg-gradient-to-br from-purple-600
to-blue-500 hover:bg-gradient-to-bl focus:ring-4 focus:outline-none
focus:ring-blue-300 dark:focus:ring-blue-800 font-medium rounded-lg text-sm
px-5 py-2.5 text-center me-2 mb-2">
           Upload
           </button>
       </form>
   </div>
 )
}

export default UploadForm
```

4. **Extract file from request in service and upload that to s3**

https://www.npmjs.com/package/multer

```
// upload.route.js

import multer from 'multer';
const upload = multer();

router.post('/', upload.single('file'), uploadFileToS3);
```

```
import AWS from 'aws-sdk';

const uploadFileToS3 = async(req, res) => {
   console.log('Upload req received');

   if (!req.file) {
       console.log('No file received');
       return res.status(400).send('No file received');
   }
   const file = req.file;

   AWS.config.update({
       region: 'ap-south-1',
       accessKeyId: process.env.AWS_ACCESS_KEY_ID,
       secretAccessKey: process.env.AWS_SECRET_ACCESS_KEY
   });

   const params = {
       Bucket: process.env.AWS_BUCKET,
       Key: file.originalname,
       Body: file.buffer
   };

   const s3 = new AWS.S3();
   s3.upload(params, (err, data) => {
       if (err) {
           console.log('Error uploading file:', err);
           res.status(404).send('File could not be uploaded!');
       } else {
           console.log('File uploaded successfully. File location:', data.Location);
           res.status(200).send('File uploaded successfully');
       }
   });}
```

```
export default uploadFileToS3;
```

5. Login using OAuth - Google SignIn on Client

OAuth 2.0 enables secure access to user data without exposing credentials. It is commonly used for third-party application access to user data on platforms like Google and Facebook.

How it Works:

- Client Registration: Applications (clients) register with the authorization server.
- User Authentication: Users log in and grant permissions to the client.
- Authorization Grant: The client receives an authorization grant.
- Token Request: The client exchanges the grant for an access token.
- Access Token: The access token is used to access protected resources on the resource server.

https://next-auth.js.org/configuration/providers/oauth

https://www.npmjs.com/package/next-auth
https://next-auth.js.org/getting-started/example
https://console.cloud.google.com
https://next-auth.js.org/providers/google

- npm install next-auth
- Google cloud console -
  - Create Project
  - Go to Project
  - API and services (on left) > OAuth consent screen
  - Choose external and create
  - Add details, save and continue
  - Go to API and services > credentials > Create credentials > OAuth Client ID
  - Choose Web App, give name
  - Add redirect url from documentation (https://next-auth.js.org/providers/google) - `http://localhost:3000/api/auth/callback/google`
  - Add google client ID and secret to .env

```
// app/api/auth/[...nextauth]/route.js
import NextAuth from "next-auth"
import GoogleProvider from "next-auth/providers/google";
const handler = NextAuth({
  providers: [
      GoogleProvider({
          clientId: process.env.GOOGLE_CLIENT_ID,
          clientSecret: process.env.GOOGLE_CLIENT_SECRET,
        })
  ]
})
```

```
export { handler as GET, handler as POST }
```

```
"use client"
import React from 'react'
import { useSession, signIn, signOut } from "next-auth/react"

const AuthPage = () => {
   const { data } = useSession()
   console.log('session data : ', data);

   const signin = () => {
       console.log("Signing in Google");
       signIn("google");
   }

   const signout = () => {
       console.log("Signing out of Google");
       signOut();
   }

   return (
       <div className='m-10'>
           <button type="submit"
               onClick={signin}
               className="text-white bg-gradient-to-br from-purple-600 to-blue-500 hover:bg-gradient-to-bl
focus:ring-4 focus:outline-none focus:ring-blue-300 dark:focus:ring-blue-800 font-medium rounded-lg text-sm px-5
py-2.5 text-center me-2 mb-2">
               Sign In
           </button>
           <button type="submit"
               onClick={signout}
               className="text-white bg-gradient-to-br from-purple-600 to-blue-500 hover:bg-gradient-to-bl
focus:ring-4 focus:outline-none focus:ring-blue-300 dark:focus:ring-blue-800 font-medium rounded-lg text-sm px-5
py-2.5 text-center me-2 mb-2">
               Sign Out
           </button>
       </div>
   )
}
export default AuthPage
```

```
//layout.js
import { Inter } from "next/font/google";
import "./globals.css";
import SessionProviderAuth from "./components/sessionProviderAuth";

const inter = Inter({ subsets: ["latin"] });

export const metadata = {
 title: "Create Next App",
 description: "Generated by create next app",
};

export default function RootLayout({ children }) {
 return (
   <html lang="en">
     <SessionProviderAuth>
```

```
      <body className={inter.className}>{children}</body>
    </SessionProviderAuth>
  </html>
 );}
```

```
// components/sessionProviderAuth.jsx

"use client";
import { SessionProvider } from "next-auth/react";

export default function SessionProviderAuth({ children }) {
 return <SessionProvider>{children}</SessionProvider>;
}
```

- Creating API Routes on NextJS Server

https://nextjs.org/docs/app/api-reference/file-conventions/route

```
// app/api/getuser/route.js

import { NextResponse } from 'next/server'
export async function GET(request) {
 return NextResponse.json({ msg: 'Hello from server' })
}
```

- **SSO (Single Sign-On):**

SSO is an authentication process that allows users to access multiple applications with a single set of credentials. It streamlines user access to multiple applications within an organization. For example, an employee logs in once and can access email, cloud storage, and other tools without repeated logins.

How it Works:

- User Authentication: Users log in once to the identity provider (IdP).
- Token Generation: Upon successful authentication, the IdP generates tokens.
- Token Validation: Tokens are validated by service providers (SPs) when users access their applications.

Access Granted: Users gain access to SPs without re-entering credentials.

Kafka

- Distributed streaming platform for real-time data streaming and processing.
- Ideal for high-throughput, responsive applications, surpassing the capabilities of JMS, RabbitMQ, and AMQP.

Key Kafka Concepts in E-commerce Example

- **Producer**: Generates and pushes records (messages) into topics. E.g., Order Management Service creates order messages.
- **Consumer**: Reads data from Kafka topics. E.g., Delivery Service processes order messages.
- **Topic**: A category for records where multiple consumers can subscribe. E.g., "Orders" topic for order messages.
- **Broker**: Servers storing and managing data in a Kafka cluster.
- **Cluster**: A set of brokers, scalable without downtime.
- **Partition**: Divides topics for organization and scalability. Hosted on different servers. - achieves parallelism
- **Offset**: Unique record identifier in a partition.
- **Replica**: Copies of partitions for fault tolerance.
- **Consumer Group:** A group of consumers that collaboratively process data.

How Consumer Consumes and Tracks

- Periodic heartbeat updates to Kafka with the latest offset.
- Multiple consumers are organized into consumer groups.
- Each consumer group reads from specific partitions, ensuring each message is delivered to only one consumer.

Replication of Partition

- Kafka replicates each partition across multiple brokers for data reliability and fault tolerance.
- One broker is the "leader," handling data requests, while others are "followers" duplicating the leader's data.
- Replicas are distributed across brokers, ensuring data availability during broker failures.

2 services -

upload service-> producer

transcoder -> consumer

https://www.npmjs.com/package/kafkajs
https://kafka.js.org/docs/getting-started

https://aiven.io/

- Create Apache Kafka Service on Aiven - Free tier - skip all the steps
  - Disable kaService Settings (on left) - scroll down to Advanced Configuration > Configure
  - fka_authentication_methods.certificate
  - Enable kafka_authentication_methods.sasl (to allow login using username and pwd)
- Overview (on left) - Copy Service URI, user and pwd to your code
- Download CA certificate and add to root path of services
- Create topic on aiven

```
// kafka/kafka.js - all services where you want to use kafka (either produce or consume)
import {Kafka} from "kafkajs"
import fs from "fs"
import path from "path"

class KafkaConfig {
   constructor(){
       this.kafka = new Kafka({
           clientId: "youtube uploader",
           brokers: ["kafka-1162169e-hhldwitheducosys-5bb4.d.aivencloud.com:17399"],
           ssl: {
               ca: [fs.readFileSync(path.resolve("./ca.pem"), "utf-8")]
           },
           sasl: {
               username: "avnadmin",
               password: "AVNS_umwmJ24HPenZhmOVtfN",
               mechanism: "plain"
           }
       })
       this.producer = this.kafka.producer()
       this.consumer = this.kafka.consumer({groupId: "youtube-uploader"})
   }

   async produce(topic, messages){
       try {
           const result = await this.producer.connect()
           console.log("kafka connected... : ", result)
           await this.producer.send({
               topic: topic,
               messages: messages
           })
       } catch (error) {
           console.log(error)
       }finally{
           await this.producer.disconnect()
       }  }

   async consume(topic , callback){
       try {
           await this.consumer.connect()
           await this.consumer.subscribe({topic: topic, fromBeginning: true})
           await this.consumer.run({
```

```
                eachMessage: async({
                    topic, partition,message
                }) =>{
                    const value = message.value.toString()
                    callback(value)
                }
            })
        } catch (error) {
            console.log(error)
        }
    }
}
export default KafkaConfig;

// upload service
// kafkapublisher.controller.js

import KafkaConfig from "../kafka/kafka.js";

const sendMessageToKafka = async (req, res) => {
    console.log("got here in upload service...")
    try {
        const message = req.body
        console.log("body : ", message)
        const kafkaconfig = new KafkaConfig()
        const msgs = [
            {
                key: "key1",
                value: JSON.stringify(message)
            }
        ]
        const result = await kafkaconfig.produce("transcode", msgs)
        console.log("result of produce : ", result)
        res.status(200).json("message uploaded successfully")

    } catch (error) {
        console.log(error)
    }
}
export default sendMessageToKafka;
```

```
//routes - kafkapublisher.route.js

import express from "express"
import sendMessageToKafka from "../controllers/kafkapublisher.controller.js";

const router = express.Router();
router.post('/', sendMessageToKafka);

export default router;
```

```
// index.js
app.use('/publish', kafkaPublisherRouter);
```

```
// consumer service - transcoder service
// consume data from kafka - index.js
const kafkaconfig =  new KafkaConfig()
kafkaconfig.consume("transcode", (value)=>{
```

```
   console.log("got data from kafka : " , value)
})
```

- Send video in chunks from client to server to S3

```
//client - uploadForm.jsx

"use client"
import React, {useState}  from 'react'
import axios from 'axios';

const UploadForm = () => {

  const [selectedFile, setSelectedFile] = useState(null);

  const handleFileChange = (e) => {
      setSelectedFile(e.target.files[0]);
  };

  const handleSubmit = (e) => {
      e.preventDefault();
      handleFileUpload(selectedFile);
  };

  const handleFileUpload = async (file) => {
      try {

       const chunkSize = 100*1024*1024; // 100mb chunks
       const totalchunks = Math.ceil(file.size/chunkSize);
       console.log(file.size);
       console.log(chunkSize);
       console.log(totalchunks);

       let start = 0;

       for(let chunkIndex = 0; chunkIndex < totalchunks; chunkIndex++) {
         const chunk = file.slice(start, start + chunkSize);
         start += chunkSize;

         const formData = new FormData();
         formData.append('filename', file.name);
         formData.append('chunk', chunk);
         formData.append('totalChunks', totalchunks);
         formData.append('chunkIndex', chunkIndex);

         console.log('Uploading chunk', chunkIndex + 1, 'of', totalchunks);

         const res = await axios.post('http://localhost:8080/upload', formData, {
           headers: {
             'Content-Type': 'multipart/form-data'
           } });
         console.log(res.data);
       }
      } catch (error) {
        console.error('Error uploading file:', error);
      }
    };
```

```
return (
  <div className='m-10'>
      <form onSubmit={handleSubmit} encType="multipart/form-data">
 <input type="file" name="file" onChange={handleFileChange} />
 <button
   type="submit"
   className="text-white bg-gradient-to-br from-purple-600 to-blue-500 hover:bg-gradient-to-bl focus:ring-4
focus:outline-none focus:ring-blue-300 dark:focus:ring-blue-800 font-medium rounded-lg text-sm px-5 py-2.5
text-center me-2 mb-2"
 >
   Upload
 </button>
</form>
  </div>
)
}

export default UploadForm
```

```
// routes/kafkapublisher.route.js

import express from "express"
import uploadFileToS3 from "../controllers/upload.controller.js";
import multer from 'multer';
const upload = multer();


const router = express.Router();
router.post('/', upload.fields([{ name: 'chunk' }, { name: 'totalChunks' }, { name: 'chunkIndex' }]),
uploadFileToS3);

export default router;
```

Week 5

1. Multipart upload from backend to S3 - fixed file
2. Sending chunks from frontend to backend in sequence and then multipart upload to S3
3. Send chunks from frontend to backend in parallel
4. Create watch service, Get Signed URL from S3 and test from postman
5. Create VideoMetaData DB (Postgres) using Prisma in Upload Service
6. Add route in watch service to list all videos from DB
7. Add frontend code to get list of all videos and on clicking - play video
8. Only Authorized users should be able to upload (OAuth 2)

Ways to Upload Data on S3 -

- Frontend to Backend to S3 without Chunking - Slow, not efficient
- Frontend to S3 without Chunking - Processing like transcoding isn't possible
- Frontend to Backend to S3 with Chunking - Faster, processing is possible and retry, resume, abort mechanisms are possible
- Uploading using pre signed URLs from Frontend to S3 - Processing like transcoding isn't possible

Ways to Watch videos -

- Directly from S3 to frontend - Security issues, s3 bucket will have to be public
- Videos from S3 to backend to frontend - Slow, no processing is required on backend
- Directly from S3 to frontend using pre-signed URL

1. Multipart upload from backend to S3 - fixed file

https://docs.aws.amazon.com/AmazonS3/latest/userguide/mpuoverview.html

https://aws.amazon.com/blogs/compute/uploading-large-objects-to-amazon-s3-using-multipart-upload-and-transfer-acceleration/

https://docs.aws.amazon.com/sdk-for-javascript/v2/developer-guide/s3-node-examples.html

```
// multipartupload.controller.js

import AWS from 'aws-sdk';
import fs from 'fs';

const multipartUploadFileToS3 = async (req, res) => {
   console.log('Upload req received');

   const filePath = '/Users/keertipurswani/Downloads/HHLD_videos/day8_end.mp4';

   // Check if the file exists
   if (!fs.existsSync(filePath)) {
       console.log('File does not exist: ', filePath);
       return res.status(400).send('File does not exist');
   }

   AWS.config.update({
       region: 'ap-south-1',
       accessKeyId: process.env.AWS_ACCESS_KEY_ID,
       secretAccessKey: process.env.AWS_SECRET_ACCESS_KEY
   });

   const s3 = new AWS.S3();
   const uploadParams = {
       Bucket: process.env.AWS_BUCKET,
       Key: "trial-key",
       ACL: 'public-read',
       ContentType: 'video/mp4'
   };

   try {
       console.log('Creating MultiPart Upload');
       const multipartParams = await s3.createMultipartUpload(uploadParams).promise();
       const fileSize = fs.statSync(filePath).size;
       const chunkSize = 5 * 1024 * 1024; // 5 MB
       const numParts = Math.ceil(fileSize / chunkSize);
```

```
        const uploadedETags = []; // Store ETags for uploaded parts

        for (let i = 0; i < numParts; i++) {
            const start = i * chunkSize;
            const end = Math.min(start + chunkSize, fileSize);

            const partParams = {
                Bucket: uploadParams.Bucket,
                Key: uploadParams.Key,
                UploadId: multipartParams.UploadId,
                PartNumber: i + 1,
                Body: fs.createReadStream(filePath, { start, end }),
                ContentLength: end - start
            };

            const data = await s3.uploadPart(partParams).promise();
            console.log(`Uploaded part ${i + 1}: ${data.ETag}`);

            uploadedETags.push({ PartNumber: i + 1, ETag: data.ETag });
        }

        const completeParams = {
            Bucket: uploadParams.Bucket,
            Key: uploadParams.Key,
            UploadId: multipartParams.UploadId,
            MultipartUpload: { Parts: uploadedETags }
        };

        console.log('Completing MultiPart Upload');
        const completeRes = await s3.completeMultipartUpload(completeParams).promise();
        console.log(completeRes);

        console.log('File uploaded successfully');
        res.status(200).send('File uploaded successfully');
    } catch (err) {
        console.log('Error uploading file:', err);
        res.status(500).send('File could not be uploaded');
    }
};

export default multipartUploadFileToS3;
```

```
import express from "express"
//import uploadFileToS3 from "../controllers/upload.controller.js";
import multer from 'multer';
import multipartUploadFileToS3 from "../controllers/multipartupload.controller.js";
const upload = multer();


const router = express.Router();
//router.post('/', upload.fields([{ name: 'chunk' }, { name: 'totalChunks' }, { name:
'chunkIndex' }]), uploadFileToS3);
router.post('/', multipartUploadFileToS3);
export default router;
```

https://docs.aws.amazon.com/AmazonS3/latest/userguide/cors.html

```
// Bucket CORS under permissions
[
    {
        "AllowedHeaders": [
            "*"
        ],
        "AllowedMethods": [
            "PUT",
            "POST",
            "HEAD",
            "GET"
        ],
        "AllowedOrigins": [
            "*"
        ],
        "ExposeHeaders": [
            "x-amz-server-side-encryption",
            "x-amz-request-id",
            "x-amz-id-2",
            "ETag"
        ],
        "MaxAgeSeconds": 3000
    }
```

```
]
```

2. **Sending chunks from frontend to backend in sequence and then multipart upload to S3**

```
import AWS from 'aws-sdk';

// Initialize upload
export const initializeUpload = async (req, res) => {
   try {
       console.log('Initialising Upload');
       const {filename} = req.body;
       console.log(filename);
       const s3 = new AWS.S3({
           accessKeyId: process.env.AWS_ACCESS_KEY_ID,
           secretAccessKey: process.env.AWS_SECRET_ACCESS_KEY,
           region: 'ap-south-1'
       });
       const bucketName = process.env.AWS_BUCKET;

       const createParams = {
           Bucket: bucketName,
           Key: filename,
           ContentType: 'video/mp4'
       };

       const multipartParams = await s3.createMultipartUpload(createParams).promise();
       console.log("multipartparams---- ", multipartParams);
       const uploadId = multipartParams.UploadId;

       res.status(200).json({ uploadId });
   } catch (err) {
       console.error('Error initializing upload:', err);
       res.status(500).send('Upload initialization failed');
   }
};

// Upload chunk
export const uploadChunk = async (req, res) => {
   try {
       console.log('Uploading Chunk');
       const { filename, chunkIndex, uploadId } = req.body;
       const s3 = new AWS.S3({
```

```
            accessKeyId: process.env.AWS_ACCESS_KEY_ID,
            secretAccessKey: process.env.AWS_SECRET_ACCESS_KEY,
            region: 'ap-south-1'
        });
        const bucketName = process.env.AWS_BUCKET;

        const partParams = {
            Bucket: bucketName,
            Key: filename,
            UploadId: uploadId,
            PartNumber: parseInt(chunkIndex) + 1,
            Body: req.file.buffer,
        };

        const data = await s3.uploadPart(partParams).promise();
        console.log("data------- ", data);
        res.status(200).json({ success: true });
    } catch (err) {
        console.error('Error uploading chunk:', err);
        res.status(500).send('Chunk could not be uploaded');
    }
};

// Complete upload
export const completeUpload = async (req, res) => {
    try {
        console.log('Completing Upload');
        const { filename, totalChunks, uploadId } = req.body;
        const uploadedParts = [];

        // Build uploadedParts array from request body
        for (let i = 0; i < totalChunks; i++) {
            uploadedParts.push({ PartNumber: i + 1, ETag: req.body[`part${i + 1}`] });
        }

        const s3 = new AWS.S3({
            accessKeyId: process.env.AWS_ACCESS_KEY_ID,
            secretAccessKey: process.env.AWS_SECRET_ACCESS_KEY,
            region: 'ap-south-1'
        });
        const bucketName = process.env.AWS_BUCKET;

        const completeParams = {
            Bucket: bucketName,
            Key: filename,
```

```
            UploadId: uploadId,
        };

        // Listing parts using promise
        const data = await s3.listParts(completeParams).promise();

        const parts = data.Parts.map(part => ({
            ETag: part.ETag,
            PartNumber: part.PartNumber
        }));

        completeParams.MultipartUpload = {
            Parts: parts
        };

        // Completing multipart upload using promise
        const uploadResult = await
s3.completeMultipartUpload(completeParams).promise();

        console.log("data----- ", uploadResult);
        return res.status(200).json({ message: "Uploaded successfully!!!" });

    } catch (error) {
        console.log('Error completing upload :', error);
        return res.status(500).send('Upload completion failed');
    }
};
```

```
import express from "express";
import { initializeUpload, uploadChunk, completeUpload } from
"../controllers/multipartupload.controller.js";
import multer from 'multer';
const upload = multer();

const router = express.Router();

// Route for initializing upload
router.post('/initialize', upload.none(), initializeUpload);

// Route for uploading individual chunks
router.post('/', upload.single('chunk'), uploadChunk);
```

```
// Route for completing the upload
router.post('/complete', completeUpload);

export default router;
```

```
// client

"use client"
import React, { useState } from 'react';
import axios from 'axios';

const UploadForm = () => {
 const [selectedFile, setSelectedFile] = useState(null);

 const handleFileChange = (e) => {
   setSelectedFile(e.target.files[0]);
 };

 const handleUpload = async () => {
   try {
     const formData = new FormData();
     formData.append('filename', selectedFile.name);
     const initializeRes = await axios.post('http://localhost:8080/upload/initialize',
formData, {
       headers: {
         'Content-Type': 'multipart/form-data'
       }
     }
     );
     const { uploadId } = initializeRes.data;
     console.log('Upload id is ', uploadId);

     const chunkSize = 5 * 1024 * 1024; // 5 MB chunks
     const totalChunks = Math.ceil(selectedFile.size / chunkSize);

     let start = 0;

     for (let chunkIndex = 0; chunkIndex < totalChunks; chunkIndex++) {

       const chunk = selectedFile.slice(start, start + chunkSize);
       start += chunkSize;
```

```
      const chunkFormData = new FormData();
      chunkFormData.append('filename', selectedFile.name);
      chunkFormData.append('chunk', chunk);
      chunkFormData.append('totalChunks', totalChunks);
      chunkFormData.append('chunkIndex', chunkIndex);
      chunkFormData.append('uploadId', uploadId);

      await axios.post('http://localhost:8080/upload', chunkFormData, {
        headers: {
          'Content-Type': 'multipart/form-data'
        }
      });
    }

    const completeRes = await axios.post('http://localhost:8080/upload/complete', {
      filename: selectedFile.name,
      totalChunks: totalChunks,
      uploadId: uploadId
    });

    console.log(completeRes.data);
  } catch (error) {
    console.error('Error uploading file:', error);
  }
};

return (
  <div className='m-10'>
    <form encType="multipart/form-data">
      <input type="file" name="file" onChange={handleFileChange} />
      <button
        type="button"
        onClick={handleUpload}
        className="text-white bg-gradient-to-br from-purple-600 to-blue-500
hover:bg-gradient-to-bl focus:ring-4 focus:outline-none focus:ring-blue-300
dark:focus:ring-blue-800 font-medium rounded-lg text-sm px-5 py-2.5 text-center me-2
mb-2"
      >
        Upload
      </button>
    </form>
  </div>
);
};
```

```
export default UploadForm;
```

3. Send chunks from frontend to backend in parallel

```
// Before for loop for chunking -

const uploadPromises = [];
```

```
// In for loop

const uploadPromise = axios.post('http://localhost:8080/upload', chunkFormData, {
         headers: {
           'Content-Type': 'multipart/form-data'
         }
       });
       uploadPromises.push(uploadPromise);
```

```
// After for loop and before completion

await Promise.all(uploadPromises);
```

4. Create watch service, Get Signed URL from S3 and test from postman

https://docs.aws.amazon.com/AmazonS3/latest/userguide/ShareObjectPreSignedURL.html

```
// watch.controller.js

import AWS from "aws-sdk"

async function generateSignedUrl(videoKey) {

   const s3 = new AWS.S3({
       accessKeyId: process.env.AWS_ACCESS_KEY_ID,
       secretAccessKey: process.env.AWS_SECRET_ACCESS_KEY,
       region: process.env.AWS_REGION
   });

   const params = {
       Bucket: process.env.AWS_BUCKET,
       Key: videoKey,
       Expires: 3600 // URL expires in 1 hour, adjust as needed
   };

   return new Promise((resolve, reject) => {
       s3.getSignedUrl('getObject', params, (err, url) => {
           if (err) {
               reject(err);
           } else {
               resolve(url);
           }
       });
   });
}

const watchVideo = async (req, res) => {
   try {
       const videoKey = req.query.key; // Key of the video file in S3
       const signedUrl = await generateSignedUrl(videoKey);
       res.json({ signedUrl });
   } catch (err) {
       console.error('Error generating pre-signed URL:', err);
       res.status(500).json({ error: 'Internal Server Error' });
   }
}
```

```
export default watchVideo;
```

```
// watch.route.js

import express from "express"
import watchVideo from "../controllers/watch.controller.js";

const router = express.Router();

router.get('/', watchVideo);

export default router;
```

```
// index.js

import express from "express";
import dotenv from "dotenv"
import cors from "cors"
import watchRouter from "./routes/watch.route.js"

dotenv.config();

const port = process.env.PORT || 8082;
const app = express();

app.use(cors({
   allowedHeaders: ["*"],
   origin: "*"
}));

app.use(express.json());

app.use('/watch', watchRouter);

app.get('/', (req, res) => {
   res.send('HHLD YouTube Watch Service')
})

app.listen(port, () => {
```

```
   console.log(`Server is listening at http://localhost:${port}`);
})
```

5. Create VideoMetaData DB (Postgres) using Prisma in Upload Service

Main differences between Vitess and PostgreSQL:

- **Purpose**: Vitess is designed for horizontal scaling and sharding of MySQL databases, while PostgreSQL is a traditional relational database management system (RDBMS) known for its reliability and feature-rich capabilities.
- **Scaling**: Vitess scales out horizontally by sharding data across multiple nodes, whereas PostgreSQL typically scales vertically by adding more resources to a single node.
- **Cloud-Native vs. Traditional**: Vitess is designed to be cloud-native and works well with containerized environments like Kubernetes, whereas PostgreSQL is a traditional database system that can be deployed on various platforms.
- **Query Routing**: Vitess abstracts the sharding complexity and intelligently routes queries to the appropriate shards, whereas PostgreSQL does not natively support sharding and relies on other scaling techniques like replication and partitioning.
- **Consistency**: Both Vitess and PostgreSQL provide strong consistency guarantees and support ACID transactions, but they achieve this in different ways due to their architectural differences.

https://vitess.io/
https://www.prisma.io/
https://www.prisma.io/docs/getting-started/setup-prisma/start-from-scratch/relational-databases-typescript-postgresql
https://www.postgresql.org/download/
https://aiven.io/

- npm install prisma
- npx prisma init
    - creates a new directory called `prisma` that contains a file called `schema.prisma`, which contains the Prisma schema with your database connection variable and schema models
    - creates the `.env` file in the root directory of the project, which is used for defining environment variables (such as your database connection)
- Install Prisma extension on VSCode
- set DATABASE_URL in .env - get your details from aiven.io project

```
model VideoData {
id          Int    @id @default(autoincrement())
title       String
description String?
author      String
url         String
}
```

- npx prisma format

- npx prisma migrate dev –name init
- npm i @prisma/client
- npx prisma generate

```
// db/db.js

import { PrismaClient } from '@prisma/client'
const prisma = new PrismaClient()

export async function addVideoDetailsToDB(title, description, author, url) {
 const videoData = await prisma.videoData.create({
  data: {
      title: title,
      description: description,
      author: author,
      url: url
  } })
 console.log(videoData);
}
```

- Test using postman

```
//controller

export const uploadToDb = async (req, res) => {
   console.log("Adding details to DB");
   try {
       const videoDetails = req.body;
       await addVideoDetailsToDB(videoDetails.title, videoDetails.description,
videoDetails.author, videoDetails.url);
       return res.status(200).send("success");
   } catch (error) {
       console.log("Error in adding to DB ", error);
       return res.status(400).send(error);
   }
}
```

```
//route
```

```
router.post('/uploadToDB', uploadToDb);
```

```
// Postman Body

{
   "title": "js",
   "description": "learn js",
   "author": "keerti",
   "url": "https://hhld-classes.s3.ap-south-1.amazonaws.com/day8_end.mp4"
}
```

- Send Video Details from frontend during completion of upload and add to DB

```
 const [title, setTitle] = useState('');
 const [description, setDescription] = useState('');
 const [author, setAuthor] = useState('');
```

```
// handleUpload - inside try

if (!title || !author) {
       alert('Title and Author are required fields.');
       return;
}
```

```
const completeRes = await axios.post('http://localhost:8080/upload/complete', {
       filename: selectedFile.name,
       totalChunks: totalChunks,
       uploadId: uploadId,
       title: title,
       description: description,
       author: author
});
```

```
<div className='container mx-auto max-w-lg p-10'>
    <form encType="multipart/form-data">
      <div className="mb-4">
        <input type="text"
               name="title"
               placeholder="Title"
               value={title}
               onChange={(e) => setTitle(e.target.value)}
               required
               className="px-3 py-2 w-full border rounded-md focus:outline-none
focus:border-blue-500" />
      </div>
      <div className="mb-4">
        <input type="text"
               name="description"
               placeholder="Description"
               value={description}
               onChange={(e) => setDescription(e.target.value)}
```

```
              className="px-3 py-2 w-full border rounded-md focus:outline-none
focus:border-blue-500" />
      </div>
      <div className="mb-4">
        <input type="text"
              name="author"
              placeholder="Author"
              value={author}
              onChange={(e) => setAuthor(e.target.value)}
              required
              className="px-3 py-2 w-full border rounded-md focus:outline-none
focus:border-blue-500" />
      </div>
      <div className="mb-4">
        <input type="file"
              name="file"
              onChange={handleFileChange}
              className="px-3 py-2 w-full border rounded-md focus:outline-none
focus:border-blue-500" />
      </div>
      <button
        type="button"
        onClick={handleUpload}
        className="text-white bg-gradient-to-br from-purple-600 to-blue-500
hover:bg-gradient-to-bl focus:ring-4 focus:outline-none focus:ring-blue-300
dark:focus:ring-blue-800 font-medium rounded-lg text-sm px-5 py-2.5 text-center"
      >
        Upload
      </button>
    </form>
  </div>
```

```
// upload service - controller - complete upload

console.log('Completing Upload');
const { filename, totalChunks, uploadId, title, description, author } = req.body;

////////////////////////////////

const uploadResult = await s3.completeMultipartUpload(completeParams).promise();

console.log("data----- ", uploadResult);

console.log("Updating data in DB");
```

```
const url = uploadResult.Location;
console.log("Video uploaded at ", url);

await addVideoDetailsToDB(title, description, author, url);
return res.status(200).json({ message: "Uploaded successfully!!!" });
```

6. Add route in watch service to list all videos from DB

- npm i prisma
- npx prisma init
- Set DATABASE_URL in .env
- npm i @prisma/client
- npx prisma generate

```
//home.controlller.js

import {PrismaClient} from "@prisma/client"

const getAllVideos = async(req, res) => {
    const prisma = new PrismaClient();
    try {
        const allData = await prisma.$queryRaw`SELECT * FROM "VideoData"`;
        console.log(allData);
        return res.status(200).send(allData);
      } catch (error) {
        console.log('Error fetching data:', error);
        return res.status(400).send();
      }
}

export default getAllVideos;
```

```
router.get('/home', getAllVideos);
```

- Test using postman

7. Add frontend code to get list of all videos and on clicking - play video

```
"use client"
import React, { useEffect, useState } from 'react';
import axios from "axios"
import dynamic from 'next/dynamic'
const ReactPlayer = dynamic(() => import("react-player"), { ssr: false });

const YouTubeHome = () => {
   const [videos, setVideos] = useState([]);
   const [loading, setLoading] = useState(true);

   useEffect(() => {

       const getVideos = async () => {
           try {
               const res = await axios.get('http://localhost:8083/watch/home');
               console.log(res);
               setVideos(res.data);
               setLoading(false); // Set loading to false when videos are fetched
           } catch (error) {
               console.log('Error in fetching videos : ', error);
               setLoading(false);
           }
       }
       getVideos();

   }, []);

   return (
       <div>
           {loading ? (
               <div className='container mx-auto flex justify-center items-center
h-screen'>Loading...</div>
           ) : (
               <div className="grid grid-cols-1 sm:grid-cols-2 md:grid-cols-3
lg:grid-cols-4 xl:grid-cols-5 gap-4 m-10">
                   {videos.map(video => (
                       <div key={video.id}
                           className="border rounded-md overflow-hidden">
                           <div>
                               <ReactPlayer url={video.url}
                                   width="360px"
                                   height="180px"
```

```
                                controls={true}
                            />
                        </div>
                        <div className="p-4">
                            <h2 className="text-lg font-semibold
mb-2">{video.title}</h2>
                            <p className="text-gray-700">Author -
{video.author}</p>
                            <p className="text-gray-700">{video.description}</p>
                        </div>
                    </div>
                ))}
            </div>
        )}
    </div>
  );
};

export default YouTubeHome;
```

8. Only Authorized users should be able to upload (OAuth 2) - Add NavBar in Frontend

```
import React from 'react'
import { useSession, signIn, signOut } from "next-auth/react"
import { useRouter } from 'next/navigation'

const NavBar = () => {
   const router = useRouter()
   const { data } = useSession()
   console.log('data---------- ', data);

   const goToUpload = () => {
       router.push('/upload')
   }
   return (
       <div>
           <nav class="bg-white border-gray-200 dark:bg-gray-900">
               <div class="max-w-screen-xl flex flex-wrap items-center justify-between
mx-auto p-4">
                   <span class="self-center text-2xl font-semibold whitespace-nowrap
dark:text-white">Educosys YouTube</span>
                   <div class="hidden w-full md:block md:w-auto" id="navbar-default">
                       {
                           data ? (
                               <div className='flex'>
                                   <button
                                       type="button"
                                       onClick={goToUpload}
                                       className="text-white bg-gradient-to-br
from-purple-600 to-blue-500 hover:bg-gradient-to-bl focus:ring-4 focus:outline-none
focus:ring-blue-300 dark:focus:ring-blue-800 font-medium rounded-lg text-sm px-5
py-2.5 text-center m-2"
                                   >
                                       Upload
                                   </button>
                                   <button
                                       type="button"
                                       onClick={signOut}
                                       className="text-white bg-gradient-to-br
from-purple-600 to-blue-500 hover:bg-gradient-to-bl focus:ring-4 focus:outline-none
focus:ring-blue-300 dark:focus:ring-blue-800 font-medium rounded-lg text-sm px-5
py-2.5 text-center m-2"
                                   >
                                       Sign Out
                                   </button>
```

```
                                    <span className='my-5'>
                                        Hello {data.user.name}
                                    </span>
                                    <div className='m-3'>
                                        <img class="w-10 h-10 rounded-full"
src={data.user.image} alt="" />
                                    </div>
                                </div>

                            ) : (
                                <button
                                    type="button"
                                    onClick={signIn}
                                    className="text-white bg-gradient-to-br
from-purple-600 to-blue-500 hover:bg-gradient-to-bl focus:ring-4 focus:outline-none
focus:ring-blue-300 dark:focus:ring-blue-800 font-medium rounded-lg text-sm px-5
py-2.5 text-center"
                                >
                                    Sign In
                                </button>
                            )
                        }
                    </div>
                </div>
            </nav>


        </div>
    )
}

export default NavBar
```

Redirecting from Upload page to YouTube Home if Data Not Found - Handling on client

```
const {data} = useSession();
useEffect(() => {
   console.log('data------- ', data);
   if(!data) {
     console.log('redirecting');
     redirect("/");
   }
 }, [])
```

Adaptive Bitrate Streaming

- **Resolution**: Resolution refers to the clarity and detail of the video image and is achieved by varying the number of pixels in the width and height of the video frame. It's usually expressed in terms of pixels, like 720p (1280x720 pixels), 1080p (1920x1080 pixels - Full HD) or 3840x2160 (4K). Higher resolutions generally provide sharper images but require more bandwidth to stream.

- **Format**: The format of a YouTube video typically refers to the encoding method used to compress and store the video data. YouTube supports various formats, but the most common ones are MP4 (H.264 video codec, AAC audio codec) and WebM (VP9 video codec, Opus audio codec). - Codec is the software or hardware tool that compresses and decompresses digital media files like videos or audios. It shrinks file sizes for storage or streaming and then restores them for playback or editing. Codecs follow specific standards for encoding and decoding media, ensuring compatibility across devices and software.

- **Bitrate**: Bitrate refers to the amount of data transferred per second in the video stream. It's usually measured in bits per second (bps) or kilobits per second (kbps) for video streaming. Higher bitrates result in better video quality but require more bandwidth to stream without buffering. YouTube automatically adjusts the bitrate based on the viewer's internet connection speed to ensure smooth playback.

Transport Layer - UDP, TCP
TCP- 3 way handshake - reliable, ordering -> tradeoff - speed
UDP- Speed -> tradeoff - not reliable

WebRTC (Web Real-Time Communication): (on top of UDP)
- Enables real-time communication directly between web browsers and mobile apps.
- Utilizes peer-to-peer connections for communication.
- Supports various types of media, including audio, video, and data.
- Doesn't require plugins, making it suitable for modern web applications.
- Offers encryption and security features for privacy.

RTMP (Real-Time Messaging Protocol):
- Developed by Adobe for streaming audio, video, and data.
- Commonly used for live streaming applications.
- Provides low-latency streaming capabilities.
- Requires dedicated RTMP servers for content delivery.
- Usage has declined in favor of other protocols like HLS and DASH.

- Both HLS and DASH are adaptive streaming protocols used for delivering multimedia content over HTTP.
- They segment media content into small files for efficient streaming and provide mechanisms for adaptive bitrate streaming to ensure a smooth playback experience.
- HLS was developed by Apple, while DASH is an open standard developed by a consortium of companies including Microsoft, Netflix, and Google.

HLS (HTTP Live Streaming):

1. File Extensions:
- .m3u8: Playlist file containing URLs of media segments.
- .ts: Transport Stream files containing the actual media segments.

2. How it Works:
- Segments the media content into small, downloadable files (typically 2-10 seconds long).
- Creates a master playlist (.m3u8) which references variant playlists or media playlists.
- Variant playlists contain links to multiple renditions of the same content at different bitrates.
- Media playlists contain references to the individual media segments.
- The client (e.g., web browser) requests the master playlist, selects an appropriate bitrate rendition, and then requests the corresponding media playlist and segments.
- Adaptive bitrate streaming allows switching between different quality levels based on available bandwidth and device capabilities.

3. Advantages:
- Widely supported by various platforms and devices.
- Simple setup and deployment.
- Works well with existing HTTP infrastructure.
- Supports adaptive bitrate streaming for a better user experience.

DASH (Dynamic Adaptive Streaming over HTTP):

1. File Extensions:
- .mpd: Media Presentation Description file, which acts as the manifest file.
- .m4s: Media Segment files containing the actual media segments.

2. How it Works:
- Divides the media content into smaller segments.
- Utilizes a manifest file (.mpd) to describe the media presentation, including available bitrates, segment URLs, and other metadata.
- The client requests the manifest file, selects a suitable representation based on network conditions, and then fetches segments directly from the server.
- Supports various codecs and container formats, providing flexibility in content creation and delivery.
- Offers features like trick play (fast forward, rewind) and DRM (Digital Rights Management) support.

HLS uses manifest files (M3U8) to organize and deliver video content in small segments (TS files) over HTTP. The adaptive bitrate streaming allows the video player to dynamically adjust the quality of the stream based on the available network bandwidth and the capabilities of the playback device.

Manifest File (M3U8):

A **manifest file** in the context of HLS is typically an M3U8 file. An M3U8 file is a plaintext file with the .m3u8 extension that serves as a playlist for the video stream.

It contains information about the available video segments, their URLs, and other metadata necessary for playback. The manifest file specifies the different video qualities (bitrates) available for the stream, allowing the player to adaptively switch between them based on the user's device and network conditions.

File Extensions:

.m3u8: This is the file extension for HLS manifest files. It contains a list of URLs to the video segments in the stream.
.ts: Transport Stream (TS) is a container format used for multiplexing audio, video, and other data. In the context of HLS, video segments are often stored as TS files. Each TS file represents a small segment of the video stream.

Video Encoding:

Before being streamed via HLS, the video content needs to be encoded into a format suitable for streaming. This typically involves compressing the video using a codec (such as H.264 or H.265) to reduce file size and bit rate while maintaining acceptable visual quality. Audio may also be compressed using codecs like AAC. The encoded video is then segmented into small chunks (usually a few seconds long), each represented by a TS file. These TS files, along with the manifest file, are served over HTTP by a server to clients for playback.

https://ffmpeg.org/
https://www.npmjs.com/package/fluent-ffmpeg
https://www.npmjs.com/package/ffmpeg-static

In Transcoder Service-

- npm i ffmpeg-static
- npm i fluent-ffmpeg

Encoding on Backend-

```
import ffmpeg from "fluent-ffmpeg"
import ffmpegStatic from "ffmpeg-static"
import fs from "fs"
ffmpeg.setFfmpegPath(ffmpegStatic)

const convertToHLS = async()=>{
    const resolutions = [
        {
            resolution: '320x180',
            videoBitrate: '500k',
            audioBitrate: '64k'
        },
        {
            resolution: '854x480',
            videoBitrate: '1000k',
            audioBitrate: '128k'
        },
        {
            resolution: '1280x720',
            videoBitrate: '2500k',
            audioBitrate: '192k'
        }
    ];

    const mp4FileName = 'test.mp4';
    const variantPlaylists = [];

    for (const { resolution, videoBitrate, audioBitrate } of resolutions) {
        console.log(`HLS conversion starting for ${resolution}`);
        const outputFileName = `${mp4FileName.replace(
            '.',
            '_'
        )}_${resolution}.m3u8`;
```

```
      const segmentFileName = `${mp4FileName.replace(
          '.',
          '_'
      )}_${resolution}_%03d.ts`;
      await new Promise((resolve, reject) => {
          ffmpeg('test.mp4')
              .outputOptions([
                  `-c:v h264`,
                  `-b:v ${videoBitrate}`,
                  `-c:a aac`,
                  `-b:a ${audioBitrate}`,
                  `-vf scale=${resolution}`,
                  `-f hls`,
                  `-hls_time 10`,
                  `-hls_list_size 0`,
                  `-hls_segment_filename output/${segmentFileName}`
              ])
              .output(`output/${outputFileName}`)
              .on('end', () => resolve())
              .on('error', (err) => reject(err))
              .run();
      });
      const variantPlaylist = {
          resolution,
          outputFileName
      };
      variantPlaylists.push(variantPlaylist);
      console.log(`HLS conversion done for ${resolution}`);
  }
  console.log(`HLS master m3u8 playlist generating`);
  let masterPlaylist = variantPlaylists
      .map((variantPlaylist) => {
          const { resolution, outputFileName } = variantPlaylist;
          const bandwidth =
              resolution === '320x180'
                  ? 676800
                  : resolution === '854x480'
                  ? 1353600
                  : 3230400;
          return
`#EXT-X-STREAM-INF:BANDWIDTH=${bandwidth},RESOLUTION=${resolution}\n${outputFileName}`
;
      })
      .join('\n');
  masterPlaylist = `#EXTM3U\n` + masterPlaylist;
```

```
    const masterPlaylistFileName = `${mp4FileName.replace(
        '.',
        '_'
    )}_master.m3u8`;
    const masterPlaylistPath = `output/${masterPlaylistFileName}`;
    fs.writeFileSync(masterPlaylistPath, masterPlaylist);
    console.log(`HLS master m3u8 playlist generated`);
}

export default convertToHLS
```

HLS Streaming on Client -

```
"use client"
import React, { useRef, useEffect } from 'react';
import Hls from 'hls.js';

const VideoPlayer = () => {
    const videoRef = useRef(null);
    const src =
"https://hhld-classes.s3.ap-south-1.amazonaws.com/output/test_mp4_master.m3u8";

    useEffect(() => {
        const video = videoRef.current;

        if (Hls.isSupported()) {
            console.log("HLS is supported");
                console.log(src);
                const hls = new Hls();
                hls.attachMedia(video);
                hls.loadSource(src);
                hls.on(Hls.Events.MANIFEST_PARSED, function () {
                    console.log("playing video");
                    video.play();
                });
        } else {
            console.log('HLS is not supported');
            // Play from the original video file
        }
    }, [src]);
```

```
    return <video ref={videoRef} controls />;
};

export default VideoPlayer;
```

Dash Conversion Code for reference-

```
import ffmpegStatic from "ffmpeg-static"
import ffmpeg from "fluent-ffmpeg"

ffmpeg.setFfmpegPath(ffmpegStatic)
const inputPath = "input.mp4"
const outputPath = "output/output.mpd"

const scaleOptions = [
    "scale=1280:720",
    "scale=640:320",
    "scale=1920:1080",
    "scale=854:480"
]

const videoCodec = "libx264"
const x264Options = "keyint=24:min-keyint=24:no-scenecut"
const videoBitrates = ['500k', '1000k', '2000k', '4000k']

ffmpeg().input(inputPath)
.videoFilters(scaleOptions)
.videoCodec(videoCodec)
.addOptions("-x264opts", x264Options)
.outputOptions("-b:v", videoBitrates[0])
.format('dash')
.outputPath(outputPath)
.on('end', ()=>{
    console.log("DASH encoding completed.")
})
.run();
```

Transcoder Service - Pick up Video from S3, transcode it and push it back
what we did - we took file in local and transcoded and manually uploaded output folder to s3

what we are going to do - give s3 url - our transcoder service should read from s3, transcode and upload to s3

```
import dotenv from "dotenv";
import AWS from 'aws-sdk';
import fs from "fs";
import path from "path";
import ffmpeg from "fluent-ffmpeg"
import ffmpegStatic from "ffmpeg-static"
ffmpeg.setFfmpegPath(ffmpegStatic)

dotenv.config();

const s3 = new AWS.S3({
   accessKeyId: process.env.AWS_ACCESS_KEY_ID,
   secretAccessKey: process.env.AWS_SECRET_ACCESS_KEY
});

const mp4FileName = 'trial1.mp4';
const bucketName = process.env.AWS_BUCKET;
const hlsFolder = 'hls';

const s3ToS3 = async () => {
   console.log('Starting script');
   console.time('req_time');
   try {
       console.log('Downloading s3 mp4 file locally');
       const mp4FilePath = `${mp4FileName}`;
       const writeStream = fs.createWriteStream('local.mp4');
       const readStream = s3
           .getObject({ Bucket: bucketName, Key: mp4FilePath })
```

```
        .createReadStream();
    readStream.pipe(writeStream);
    await new Promise((resolve, reject) => {
        writeStream.on('finish', resolve);
        writeStream.on('error', reject);
    });
    console.log('Downloaded s3 mp4 file locally');

    const resolutions = [
        {
            resolution: '320x180',
            videoBitrate: '500k',
            audioBitrate: '64k'
        },
        {
            resolution: '854x480',
            videoBitrate: '1000k',
            audioBitrate: '128k'
        },
        {
            resolution: '1280x720',
            videoBitrate: '2500k',
            audioBitrate: '192k'
        }
    ];

    const variantPlaylists = [];
    for (const { resolution, videoBitrate, audioBitrate } of resolutions) {
        console.log(`HLS conversion starting for ${resolution}`);
        const outputFileName = `${mp4FileName.replace(
            '.',
            '_'
        )}_${resolution}.m3u8`;
        const segmentFileName = `${mp4FileName.replace(
            '.',
            '_'
        )}_${resolution}_%03d.ts`;
```

```
        await new Promise((resolve, reject) => {
            ffmpeg('./local.mp4')
                .outputOptions([
                    `-c:v h264`,
                    `-b:v ${videoBitrate}`,
                    `-c:a aac`,
                    `-b:a ${audioBitrate}`,
                    `-vf scale=${resolution}`,
                    `-f hls`,
                    `-hls_time 10`,
                    `-hls_list_size 0`,
                    `-hls_segment_filename hls/${segmentFileName}`
                ])
                .output(`hls/${outputFileName}`)
                .on('end', () => resolve())
                .on('error', (err) => reject(err))
                .run();
        });
        const variantPlaylist = {
            resolution,
            outputFileName
        };
        variantPlaylists.push(variantPlaylist);
        console.log(`HLS conversion done for ${resolution}`);
    }
    console.log(`HLS master m3u8 playlist generating`);
    let masterPlaylist = variantPlaylists
        .map((variantPlaylist) => {
            const { resolution, outputFileName } = variantPlaylist;
            const bandwidth =
                resolution === '320x180'
                    ? 676800
                    : resolution === '854x480'
                    ? 1353600
                    : 3230400;
            return
`#EXT-X-STREAM-INF:BANDWIDTH=${bandwidth},RESOLUTION=${resolution}\n${outputFile
```

```
Name}`;
            })
            .join('\n');
        masterPlaylist = `#EXTM3U\n` + masterPlaylist;

        const masterPlaylistFileName = `${mp4FileName.replace(
            '.',
            '_'
        )}_master.m3u8`;
        const masterPlaylistPath = `hls/${masterPlaylistFileName}`;
        fs.writeFileSync(masterPlaylistPath, masterPlaylist);
        console.log(`HLS master m3u8 playlist generated`);

        console.log(`Deleting locally downloaded s3 mp4 file`);

        fs.unlinkSync('local.mp4');
        console.log(`Deleted locally downloaded s3 mp4 file`);

        console.log(`Uploading media m3u8 playlists and ts segments to s3`);

        const files = fs.readdirSync(hlsFolder);
        for (const file of files) {
            if (!file.startsWith(mp4FileName.replace('.', '_'))) {
                continue;
            }
            const filePath = path.join(hlsFolder, file);
            const fileStream = fs.createReadStream(filePath);
            const uploadParams = {
                Bucket: bucketName,
                Key: `${hlsFolder}/${file}`,
                Body: fileStream,
                ContentType: file.endsWith('.ts')
                    ? 'video/mp2t'
                    : file.endsWith('.m3u8')
                    ? 'application/x-mpegURL'
                    : null
            };
```

```
            await s3.upload(uploadParams).promise();
            fs.unlinkSync(filePath);
        }
        console.log(
            `Uploaded media m3u8 playlists and ts segments to s3. Also deleted
locally`
        );

        console.log('Success. Time taken: ');
        console.timeEnd('req_time');
    } catch (error) {
        console.error('Error:', error);
    }
};

export default s3ToS3;
```

- **Finishing the flow -**
  Transcoder Service Consumes the msg, picks file from S3, encodes and sends it back

```
// Transcoder Service

const kafkaconfig = new KafkaConfig()
kafkaconfig.consume("transcode", async (message) => {
   try {
       console.log("Got data from Kafka:", message);

       // Parsing JSON message value
       const value = JSON.parse(message);

       // Checking if value and filename exist
       if (value && value.filename) {
           console.log("Filename is", value.filename);
           await s3ToS3(value.filename); // Make this change in controller
       } else {
           console.log("Didn't receive filename to be picked from S3");
       }
   } catch (error) {
       console.error("Error processing Kafka message:", error);
       // You might want to handle or log this error appropriately
   }
});
```

- In Upload service, just pass filename in pushVideoForEncodingToKafka (url not needed)

CDN - Cloudfront

https://tools.keycdn.com/performance

ElasticSearch

- **Apache Lucene's Role:**
  - Apache Lucene is a Java library that provides indexing and search functionalities.
  - It's known for its efficiency and is widely used for building search engines, databases, and information retrieval systems.

- **Limitations and Need for Elasticsearch:**
  - Lucene operates on a single node, meaning it's limited to the resources of that single machine.
  - As data volumes grow, a single-node setup becomes insufficient for handling indexing and search operations efficiently.
  - Additionally, a single node is a single point of failure, jeopardizing system reliability.

- **Here comes Elasticsearch:**
  - Elasticsearch builds upon Lucene, extending its capabilities to distributed environments.
  - It's designed to handle large-scale data processing and querying across multiple nodes in a cluster.

- Key Terms:
  - **Nodes**: Each instance of Elasticsearch running in a cluster is a node. Nodes can be added or removed dynamically. They are responsible for storing data, executing queries, and participating in cluster coordination.
  - **Shards**: Elasticsearch divides indices into smaller, manageable units called shards. Each shard is essentially a Lucene index in itself. It's a self-contained unit that can be distributed across nodes for parallel processing. By default, each index in Elasticsearch is divided into multiple primary shards.
  - **Replica Shards:** To ensure data availability and resilience, Elasticsearch allows for replica shards. These are copies of primary shards distributed across nodes. Replica shards provide redundancy and fault tolerance. They are used to serve read requests in case the primary shard or its node fails.

- **Indexing Process in Lucene:**
  - Lucene indexes documents by tokenizing, analyzing, and storing textual information.
  - Textual content is broken down into tokens based on rules specified by analyzers.
  - Analyzers process tokens to normalize them (lowercasing, removing stopwords, etc.).
  - Processed tokens are then stored in an inverted index structure, allowing for **efficient full-text search.**

- **Horizontal Scaling in Elasticsearch:**
  - Elasticsearch achieves horizontal scaling by distributing shards across multiple nodes in a cluster.
  - When indexing documents, Elasticsearch hashes each document's ID to determine which shard it belongs to.

    - Queries are executed in parallel across shards, and results are aggregated, allowing for efficient distributed search operations.
    - As data or query loads increase, additional nodes can be added to the cluster, and Elasticsearch dynamically redistributes shards to balance the workload.

- **Handling Failures:**
    - Elasticsearch monitors node health and automatically redistributes shards in the event of node failures.
    - Replica shards provide redundancy, ensuring that data remains available even if a node goes down.
    - By distributing data and workload across multiple nodes, Elasticsearch enhances system reliability and fault tolerance.

- Communication between Nodes:
    - **Transport Layer:** Elasticsearch nodes communicate with each other using a binary protocol over TCP (Transport Control Protocol). This transport layer is responsible for handling cluster coordination, data exchange, and cluster management tasks.
    - **Discovery Mechanisms**: Elasticsearch employs various discovery mechanisms to detect and join nodes to form a cluster.
    - **Cluster State**: Nodes exchange information about the cluster state, including the list of available nodes, cluster settings, index mappings, and shard allocation. This information is distributed among all nodes in the cluster to ensure consistency and coordination.
    - **Shard Routing**: When executing queries or indexing documents, nodes route requests to the appropriate shards based on the hashed document ID or the query's routing parameters. This routing mechanism ensures that requests are distributed evenly across nodes and shards for efficient parallel processing.

  https://aws.amazon.com/what-is/elk-stack/

- **What is the ELK Stack?**
    - The ELK Stack is a combination of three open-source projects: Elasticsearch, Logstash, and Kibana.
    - Originally developed by Elastic, the ELK Stack is commonly used for log aggregation, search, and visualization.

- **Components of the ELK Stack:**
    - **Elasticsearch**: A distributed search and analytics engine that stores and indexes data for searching, analysis, and visualization.
    - **Logstash**: A data processing pipeline that ingests, transforms, and enriches data from multiple sources before sending it to Elasticsearch.
    - **Kibana**: A visualization and exploration tool that provides a user-friendly interface for querying, analyzing, and visualizing data stored in Elasticsearch.

- **Why is the ELK Stack Needed?**
    - **Log Management**: Organizations generate vast amounts of log data from various systems, applications, and devices.

    - **Centralized Logging**: Centralizing log data enables easier monitoring, troubleshooting, and analysis of system behavior and performance.
    - **Search and Visualization**: Elasticsearch provides powerful full-text search capabilities, while Kibana enables users to visualize and explore data through dashboards, charts, and graphs.
    - **Real-time Insights**: The ELK Stack allows for real-time analysis of log data, enabling organizations to detect issues, anomalies, and trends as they occur.

- **Key Use Cases:**
    - **Infrastructure Monitoring**: Monitoring system and application logs for performance metrics, errors, and warnings.
    - **Security Analytics**: Analyzing security logs for threat detection, incident response, and forensic analysis.
    - **Business Intelligence**: Extracting insights from application logs, transaction data, and user behavior for decision-making and optimization.
    - **DevOps and IT Operations**: Facilitating collaboration between development, operations, and support teams by providing a centralized platform for log management and analysis.

- **Deployment and Scalability:**
    - The ELK Stack can be deployed on-premises, in the cloud, or in hybrid environments.
    - Elasticsearch provides horizontal scalability, allowing organizations to scale their log storage and analysis infrastructure as their data volumes grow.
    - Logstash pipelines can be scaled horizontally to handle high volumes of data ingestion from multiple sources.

In the ELK stack, "Beats" and "X-Pack" are additional components that enhance the capabilities of the core ELK components -

Beats:

- Beats are lightweight data shippers developed by Elastic. They are designed to collect various types of data from different sources and send them to Elasticsearch or Logstash for indexing.
- Beats are used for collecting different types of data such as logs (Filebeat), metrics (Metricbeat), network data (Packetbeat), and uptime monitoring (Heartbeat).
- Beats are lightweight, efficient, and easy to set up, making them ideal for collecting data from distributed systems or containers.

X-Pack (Now called Elastic Stack features):

- X-Pack was a set of commercial extensions for the ELK stack provided by Elastic. However, as of version 6.3 of the Elastic Stack, X-Pack features are no longer offered as a separate package. Instead, the individual components and features of X-Pack have been integrated directly into the Elastic Stack.
- X-Pack provided various features such as security (authentication, authorization, encryption), monitoring, alerting, reporting, machine learning, and more.
- With X-Pack features integrated into the Elastic Stack, users can now easily enable and configure these features directly from within Elasticsearch, Kibana, or other components.

Index in Elasticsearch:

- An index in Elasticsearch is a logical namespace that stores a collection of JSON documents. It's similar to a database in a traditional SQL system or a collection in MongoDB.
- An index is the primary unit of data storage and retrieval in Elasticsearch. It organizes and partitions data, allowing for efficient storage, distribution, and querying.
- Each index may contain multiple documents, and each document is represented as a JSON object with its own set of fields and values.

Inverted Indices:

- Inverted indices are a fundamental data structure used by Elasticsearch (and many other search engines) to enable fast full-text searches. They are built during the indexing process and store a mapping from terms to the documents that contain those terms.
- When you index a document in Elasticsearch, the text fields within the document are analyzed and tokenized. The resulting tokens are then stored in the inverted index along with references to the documents that contain them.
- This allows Elasticsearch to quickly determine which documents contain a particular term during a search query. Instead of scanning through all documents, Elasticsearch can directly look up the term in the inverted index and retrieve the relevant documents.

Lambda

- **What is AWS Lambda?**
  - AWS Lambda is a serverless computing service provided by Amazon Web Services (AWS).
  - It allows you to run code without provisioning or managing servers. You only pay for the compute time consumed by your code.

- **Key Components:**
  - **Functions**: AWS Lambda operates on the concept of functions, which are small units of code that can be triggered by various events.
  - **Event Sources**: Functions can be triggered by events from various AWS services (e.g., S3, DynamoDB, API Gateway) or custom events.
  - **Compute Environment**: AWS Lambda manages the compute environment for executing functions, automatically scaling resources based on demand.
  - **Integration**: Lambda integrates seamlessly with other AWS services, allowing you to build serverless applications and workflows.

- **Why is AWS Lambda Needed?**
  - **Serverless Architecture**: With Lambda, you can focus on writing code without worrying about server provisioning, scaling, or maintenance.
  - **Cost Efficiency**: Lambda offers a pay-as-you-go pricing model, where you're billed only for the compute time consumed by your functions, with no charges for idle resources.
  - **Scalability**: Lambda automatically scales to handle incoming requests, whether it's a single request or thousands of concurrent requests.
  - **Event-Driven Programming**: Lambda enables event-driven programming, where functions are triggered by events such as file uploads, database changes, or HTTP requests.

- **Workflow with AWS Lambda:**
  - **Develop**: Write your function code in supported languages (e.g., Node.js, Python, Java).
  - **Deploy**: Upload your function code to AWS Lambda along with any required dependencies.
- **Trigger**: Define triggers for your functions, specifying the events that should invoke them (e.g., S3 bucket notifications, API Gateway requests).
- **Execute**: When triggered, AWS Lambda provisions the necessary compute resources and executes your function code.
- **Scale**: Lambda automatically scales resources based on incoming requests, ensuring high availability and performance.

- **Pushing Data to OpenSearch**

https://www.npmjs.com/package/@opensearch-project/opensearch

```
import express from "express"
import dotenv from "dotenv"
import cors from "cors"
import { Client } from "@opensearch-project/opensearch";

dotenv.config();
const app = express();
const port = process.env.PORT || 8080;

app.use(cors({
   allowedHeaders: ["*"],
   origin: "*"
}));
app.use(express.json());


// Route for uploading a video
app.post('/upload', async (req, res) => {
   try {
       console.log('Inside upload call');
       // Process video upload and extract metadata
       const { title, description, author, videoUrl } = req.body;

       var host =
"https://name:pwd@search-trial-jqtlus4ksz5xo3aqihkpv3pxny.ap-south-1.es.amazonaws.com"
;
       var host_aiven =
"https://avnadmin:AVNS_ByB7-ssBS4rc_XOzkP4@os-25088be6-hhldwitheducosys-5bb4.l.aivencl
oud.com:17386";

       var client = new Client({
           node: host_aiven
       });

       var index_name = "video";
       var document = {
           title: title,
           author: author,
           description: description,
           videoUrl: videoUrl
```

```
        };

        var response = await client.index({
            id: title, // id should ideally be db id
            index: index_name,
            body: document,
            refresh: true,
        });

        console.log("Adding document:");
        console.log(response.body);

        // Respond with success message
        res.status(200).json({ message: 'Video uploaded successfully' });
    } catch (error) {

        // Respond with error message
        console.log(error.message)
        res.status(500).json({ error: 'Internal server error' });
    }
});

app.get('/', (req, res) => {
    res.send('HHLD OpenSearch Demo')
})
app.listen(port, () => {
    console.log(`Server is listening at http://localhost:${port}`);
})
```

https://opensearch.org/docs/latest/dashboards
https://opensearch.org/docs/latest/dashboards/dev-tools/run-queries/
https://opensearch.org/docs/latest/tuning-your-cluster/

OpenSearch > Dev Tools > Query
https://opensearch.org/docs/latest/query-dsl/full-text/index/

```
GET _search
{
  "query": {
    "match_all": {}
  }
}
```

```
GET _cluster/health
```

```
GET _cat/indices
```

```
GET _cat/nodes?v
```

```
GET _cat/indices?v
```

```
GET _search
{
  "query": {
    "match": {
      "title": "hhld course"
    }
  }
}
```

```
GET _search
{
  "query": {
    "simple_query_string": {
      "query": "keerti teaches hhld",
      "fields": ["title", "author", "description", "videoUrl"]
    }
  }
}
```

```
GET _search
{
  "query": {
    "match": {
      "author": {
        "query": "keetri",
        "fuzziness": "AUTO"
      }
    }
  }
}
```

```
GET _search
{
  "query": {
    "bool": {
      "should": [
        {
          "simple_query_string": {
            "query": "keetri",
            "fields": ["title", "author", "description", "videoUrl"]
          }
        },
        {
          "bool": {
            "should": [
              { "fuzzy": { "title": { "value": "keetri", "fuzziness": "AUTO" } } },
              { "fuzzy": { "author": { "value": "keetri", "fuzziness": "AUTO" } } },
              { "fuzzy": { "description": { "value": "keetri", "fuzziness": "AUTO" } }
},
              { "fuzzy": { "videoUrl": { "value": "keetri", "fuzziness": "AUTO" } } }
            ]
          }
        }
      ]
    }
  }
}
```

- **Query Data using Lambda**

```
import express from 'express';
import { Client } from "@opensearch-project/opensearch";
import serverless from 'serverless-http';
const app = express();

// Route for handling search requests
app.get('/search', async (req, res) => {
 try {
   console.log('Inside search query');
   // Extract query parameter from the request
   const searchTerm = req.query.q || '';

   console.log('search term is ', searchTerm);
   // Example search query

   var host =
"https://name:pwd@search-trial-jqtlus4ksz5xo3aqihkpv3pxny.ap-south-1.es.amazonaws.com"
;
   var host_aiven =
"https://avnadmin:AVNS_ByB7-ssBS4rc_XOzkP4@os-25088be6-hhldwitheducosys-5bb4.l.aivencl
oud.com:17386";

   var client = new Client({
       node: host_aiven
   });

   const { body } = await client.search({
     index: 'video', // Index name in OpenSearch
     body: {
       query: {
           "simple_query_string": {
               "query": searchTerm,
               "fields": ["title", "author", "description", "videoUrl"]
           }
       }
     }
   });

   // Process search results
   const hits = body.hits.hits;
   console.log(hits);
```

```
    res.status(200).json(hits);
  } catch (error) {
    console.error(error);
    res.status(500).json({ message: 'Internal Server Error' });
  }
});

// Wrap the Express app with serverless-http
const wrappedApp = serverless(app);

// Export the wrapped app for Serverless Framework
export const handler = wrappedApp;
```

Adding Search on YouTube Project

```
// upload service - opensearch/pushToOpenSearch.js

import { Client } from "@opensearch-project/opensearch";

const PushToOpenSearch =  async (title, description, author, videoUrl) => {
   try {

       console.log('Pushing to Open Search');
       // Process video upload and extract metadata
       var auth = "keerti:HHLDisawesome1!"; // For testing only. Don't store
credentials in code.
       var host =
"https://username:pwd@search-trial-jqtlus4ksz5xo3aqihkpv3pxny.ap-south-1.es.amazonaws.
com";
       var host_aiven =
"https://avnadmin:AVNS_ByB7-ssBS4rc_XOzkP4@os-25088be6-hhldwitheducosys-5bb4.l.aivencl
oud.com:17386";

       var client = new Client({
           node: host_aiven
       });

       var index_name = "video";
       var document = {
           title: title,
           author: author,
           description: description,
           videoUrl: videoUrl
       };

       var response = await client.index({
           id: title, // id should ideally be db id
           index: index_name,
           body: document,
           refresh: true,
       });
       console.log("Adding document:");
       console.log(response.body);

   } catch (error) {
       // Respond with error message
       console.log(error.message)
```

```
    }
};
export default PushToOpenSearch;
```

```
// After Completion of MultiPart Upload

// Pushing original file for now - should push the master manifest file
PushToOpenSearch(title, description, author, uploadResult.Location);
```

Add Search Code On Client -

```
//components/searchbar.jsx

"use client"
import React, { useState } from 'react'
import axios from 'axios';
import { useVideosStore } from '../zustand/useVideosStore'

const SearchBar
    = () => {
        const [searchText, setSearchText] = useState('');
        const { updateSearchedVideos } = useVideosStore();

        const searchVideos = async () => {
            try {
                const res = await
axios.get('https://fffiqlakmg.execute-api.ap-south-1.amazonaws.com/search', { params:
{ q: searchText } });
                console.log('Data received - ', res.data);
                updateSearchedVideos(res.data);
                res.data.map((data) => {
                    console.log(data);
                    console.log('video url', data._source.videoUrl);
                })
            } catch (error) {
                console.log("Error in searching : ", error.message)
            }
        }
        return (
            <div>

                <label for="default-search" class="mb-2 text-sm font-medium
```

```
text-gray-900 sr-only dark:text-white">Search</label>
                <div class="relative">
                    <div class="absolute inset-y-0 start-0 flex items-center ps-3
pointer-events-none">
                        <svg class="w-4 h-4 text-gray-500 dark:text-gray-400"
aria-hidden="true" xmlns="http://www.w3.org/2000/svg" fill="none" viewBox="0 0 20 20">
                            <path stroke="currentColor" stroke-linecap="round"
stroke-linejoin="round" stroke-width="2" d="m19 19-4-4m0-7A7 7 0 1 1 1 8a7 7 0 0 1 14
0Z" />
                        </svg>
                    </div>
                    <input type="search"
                        id="default-search"
                        className="block w-full p-4 ps-10 text-sm text-gray-900 border
border-gray-300 rounded-lg bg-gray-50 focus:ring-blue-500 focus:border-blue-500
dark:bg-gray-700 dark:border-gray-600 dark:placeholder-gray-400 dark:text-white
dark:focus:ring-blue-500 dark:focus:border-blue-500"
                        placeholder="Search..."
                        value={searchText}
                        onChange={(e) => setSearchText(e.target.value)}
                        required />
                    <button type="submit"
                        onClick={searchVideos}
                        className="text-white absolute end-2.5 bottom-2.5 bg-blue-700
hover:bg-blue-800 focus:ring-4 focus:outline-none focus:ring-blue-300 font-medium
rounded-lg text-sm px-4 py-2 dark:bg-blue-600 dark:hover:bg-blue-700
dark:focus:ring-blue-800">Search</button>
                </div>

            </div>
        )
    }

export default SearchBar
```

```
// zustand/useVideosStore.js

import { create } from 'zustand'

export const useVideosStore = create((set) => ({
   searchedVideos: [],
   updateSearchedVideos: (videos) => set({searchedVideos: videos})
}))
```

```jsx
// youtubehome.jsx

import { useVideosStore } from '../zustand/useVideosStore'

const { searchedVideos } = useVideosStore();


<div className="grid grid-cols-1 sm:grid-cols-2 md:grid-cols-3 lg:grid-cols-4
xl:grid-cols-5 gap-4 m-10">
                        {searchedVideos.map(video => (
                            <div key={video._source._id}
                                className="border rounded-md overflow-hidden">
                                <div>
                                    <ReactPlayer url={video._source.videoUrl}
                                        width="360px"
                                        height="180px"
                                        controls={true}
                                    />
                                </div>
                                <div className="p-4">
                                    <h2 className="text-lg font-semibold
mb-2">{video._source.title}</h2>
                                    <p className="text-gray-700">Author -
{video._source.author}</p>
                                    <p
className="text-gray-700">{video._source.description}</p>
                                </div>
                            </div>
                        ))}
                    </div>
```

CICD for Lambda using GitHub Actions-

// aws.yml code just like chat system

```
name: Deploy #Name of the Workflow

on:  #Name of the GitHub event that triggers the workflow
  push:   #On Push Event We need to take action
    branches:  #Now we need to specify the branch name
    - master

jobs:  #Workflow run is made up of one or more jobs
  deploy_lambda:
    runs-on: ubuntu-latest  #Through which Server OS we need to Work (type of machine to run
the job on)
    steps:
    #Using versioned actions
      - uses: actions/checkout@v2  # --> Reference a specific version
      - uses: actions/setup-node@v2 # --> To Setup the Server With Node Env
        with:
          node-version: '14' #--> Specify the Version of the Node
      - name: Configure AWS Credentials
        uses: aws-actions/configure-aws-credentials@v1 #--> Setup the Credential for the AWS
cli
        with:
        # Created the Secrets Under the Repo only with These Variables
          aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
          aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
          aws-region: ap-south-1 #--> Define Region of the AWS-CLI
      - name: npm install
        env:
          CI: true
        run: |
          npm ci
      - name: deploy
        run: |
          zip -j deploy.zip ./* #--> Zip the Code As we know lambda function accept the zip
file.
          aws lambda update-function-code --function-name=test --zip-file=fileb://deploy.zip
```