VIDEO STREAMING SERVICE

CSE.687: Object Oriented Design

Akhil Kumar Ajitha

SUID: 275746663

aajitha@syr.edu

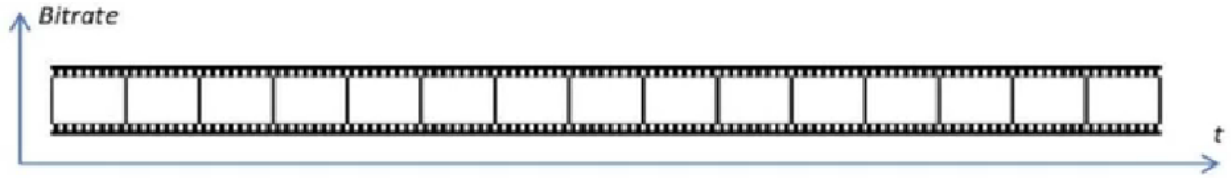
Table of Contents

INTRODUCTION/PROBLEM STATEMENT

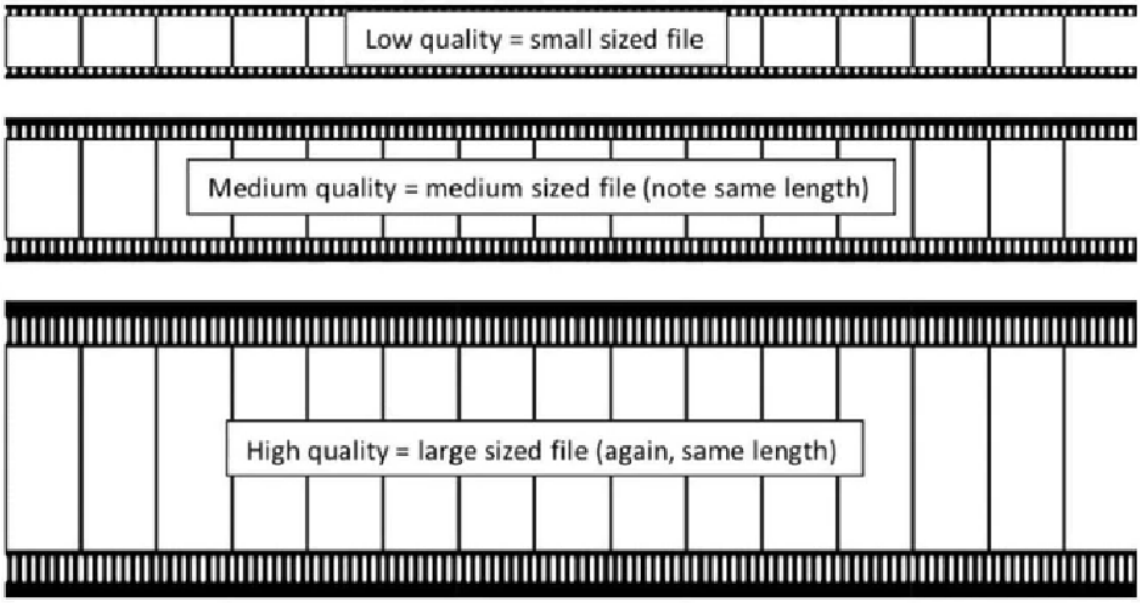
This project is about providing a service that'll allow true streaming capabilities also known as buffer less streaming. Through this project I have showcased the Object-Oriented design principles like SOLID Principles and various design patterns. To achieve this, I have used AWS services. It mimics the streaming provided by various major platforms like YouTube, Netflix, etc. which doesn't require the entire video file to be downloaded prior to streaming. Below are the various approaches takes for playing a video, in this project I have used the third approach i.e., Streaming. This service can be plugged into any website to allow buffer less streaming.

Various approaches and their advantages and disadvantages:

1) **Download**
   In this approach the entire media file must be downloaded before being played. In this approach the entire media file is stored in the database as a BLOB and is retrieved on user's request. The downloading of the entire media file consumes time and is not the best approach when the media file is of medium or large size.
2) **Progressive Download**
   In this approach the media file is split into segments, and hence each segment has to be downloaded before playing the respective sections. This allows the user to start watching the video as soon as the first segment is downloaded.
   Disadvantages:
   - Switching between various qualities of the media file is not supported. This leading to longer buffering time when the network bandwidth or strength drops.
   - Forwarding or jumping to other segments in the media will require all the previous segments to be downloaded before playing.



Media file Split into segments

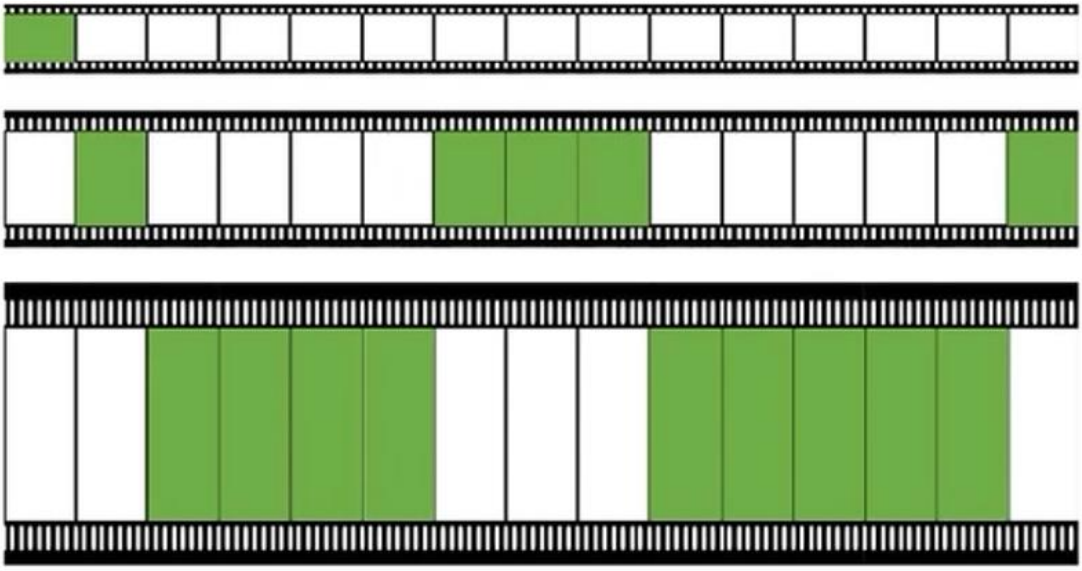


Same media file of varying quality split into segments

3) **Streaming *Implemented in this project***

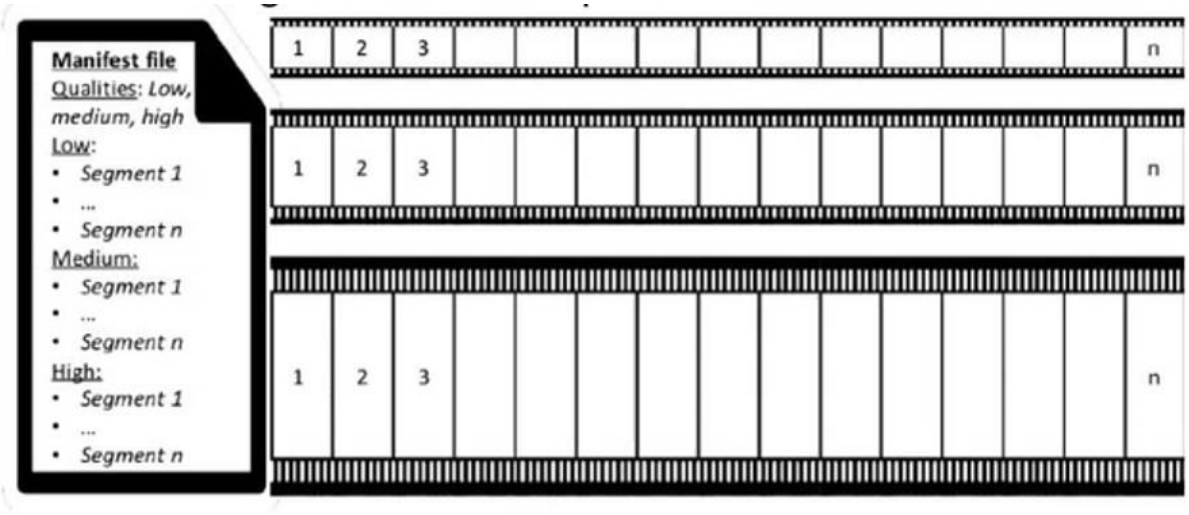
Streaming is like progressive download wherein every media file is split into various segments. The below figure shows the media file of three different qualities and the green color indicates the segments downloaded during the playback of the media file. Streaming unlike progressive download, allows seamless switching of the qualities during playback. This allows the client device to pick the best quality that matches the network bandwidth during playback.

It also allows the forwarding or jumping into a segment without the overhead of downloading the previous segments.



Downloaded segments of three different qualities are indicated from left to right.

While creating the media segments we need to create a manifest file which contains all the segments listed in the order of occurrence for all the quality types. This manifest file is used as a metadata during the playback of the media file which allows the quality switching and forwarding of the media. The extension of the manifest file is .m3u8. A pictorial representation of the manifest file along with its segments is shown below.



Media file with different qualities having a shared manifest file

There are various formats used in enable streaming which supports adaptive bit rate. In this project I have used Apple HLS format to which the media file is converted which supports adaptive bitrate and supports streaming.

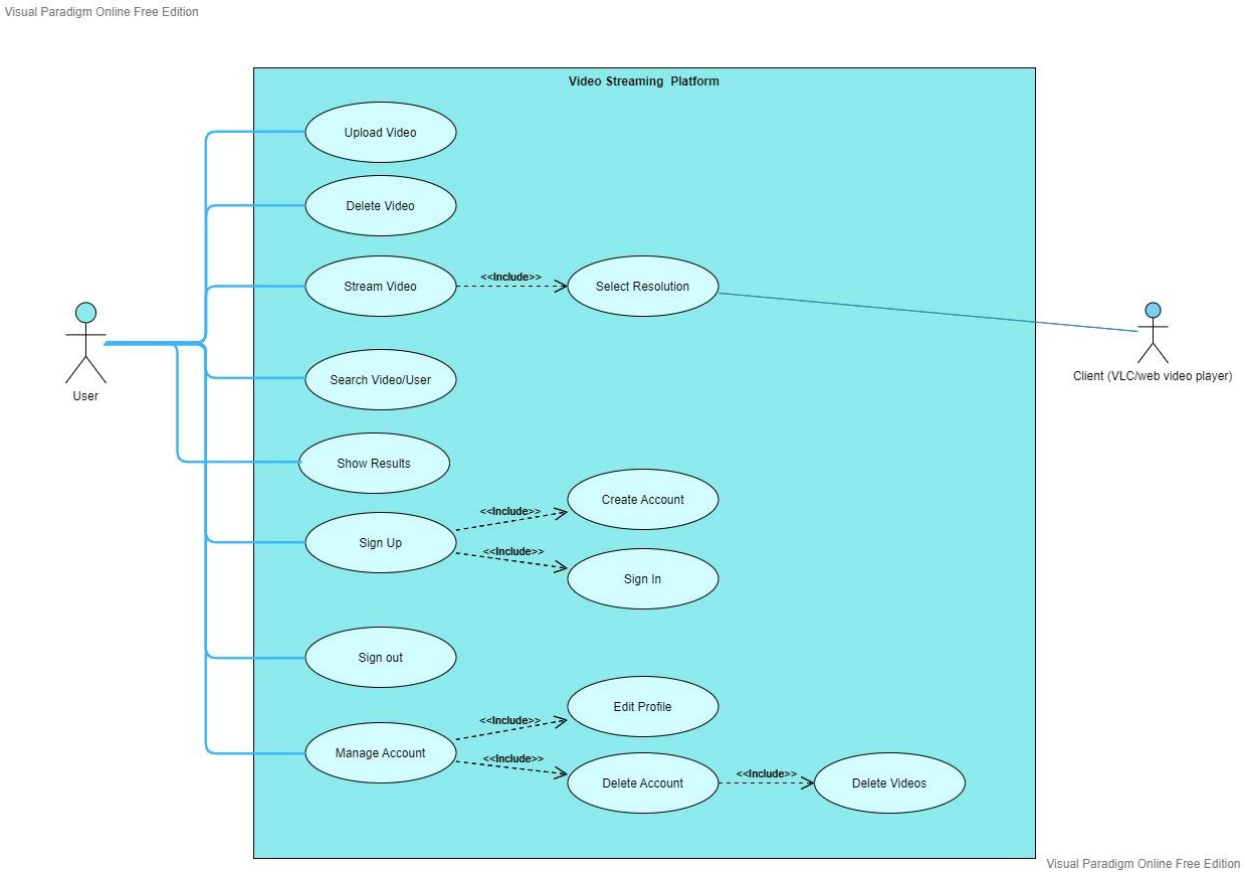
In this project I have implemented the service to enable streaming of the media file, using the 3rd approach mentioned above.

TECHNOLOGY STACK

I have used the below tech stack in the development of the project:

1) **Java**
   Object Oriented Programming language used in the development of the backend service.
2) **Spring Boot**
   Open-source Java-based framework used to create micro services.
3) **Maven**
   Dependency management and build automation tool.
4) **Python**
   Used by the AWS Lambda to trigger the Media Convert to convert the video into Apple HLS format.
5) **React**
   It is a JS library . In this project it is used to play the segmented video stored in AWS S3 bucket using the cloud front distribution URL.
6) **MySQL**
   Database used to store relational data of the user and metadata of the video.
7) **AWS Java SDK**
   Used to access AWS services from Java. In this project it is used to upload and delete videos from the S3 bucket.
8) **AWS S3**
   S3 is a NO SQL database used to store the videos uploaded by the user until it is processed by media convert. Post processing the video is stored into another output bucket which will contain the segmented video file in Apple HLS format.
9) **AWS Lambda**
   It is an event driven serverless computing platform that responds to the object created event in the S3 bucket and schedules jobs in the media convert.
10) **AWS Media Convert**
    It is a file-based video transcoding service, which is used in this project to convert any video into a segmented Apple HLS format.
11) **AWS Cloud Front**
    It is a content delivery network, which is used in this project to allow the distribution of the converted video through unique URL generated by AWS improving download speed without exposing the S3 bucket to the client. Hence adding a layer of abstraction.

USECASE



The actor is the user who will have the ability to upload, stream, and delete the video without any buffering, hence enabling true streaming capability.

The user will also be able to look up videos by searching by user's name or by video's title.

The user will be able to create/update and delete an account.

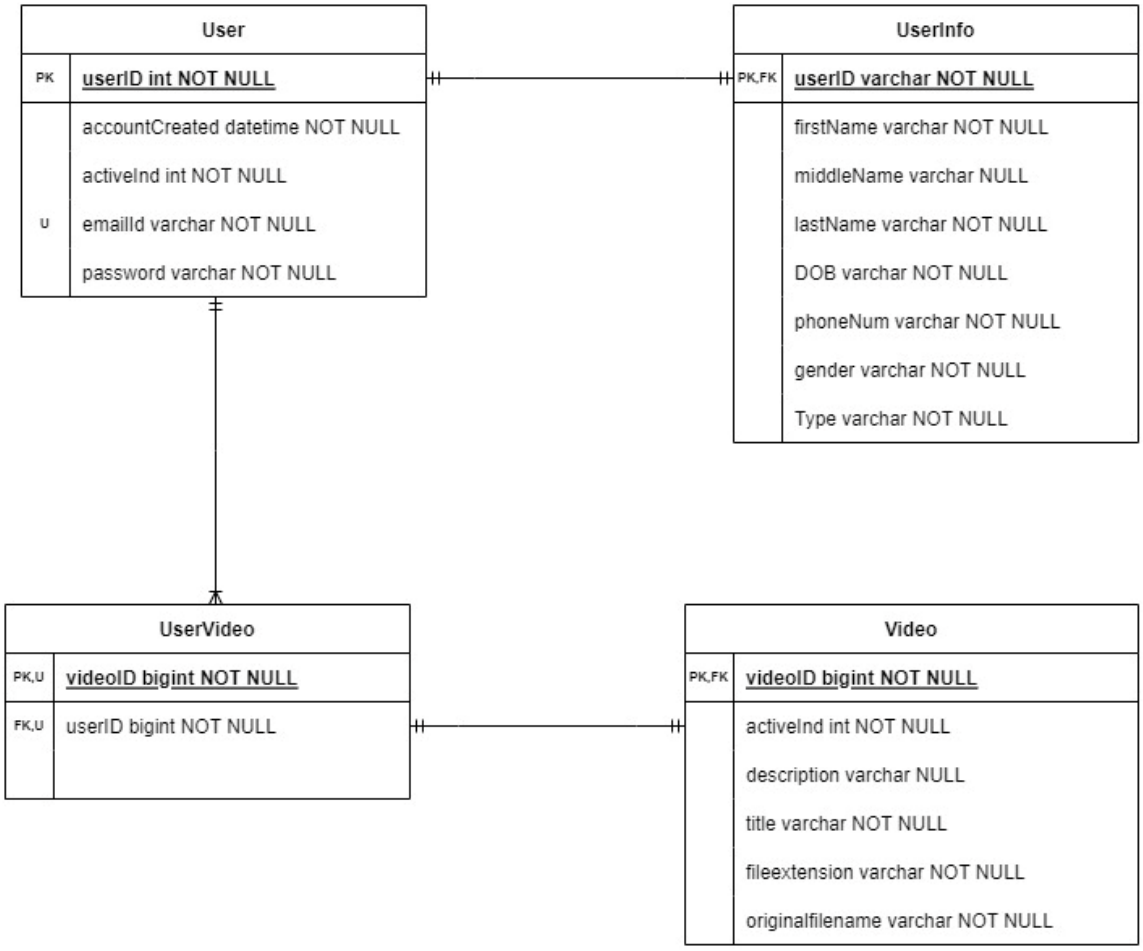
On deleting the account all the associated videos uploaded by the user will be removed from the system.

The second actor here is the client on which the video is being played which could be a VLC player or any web-based video player like video.js.

All the use cases have not been implemented due to time constraints. I have demoed a POC in this project, and various other services can be added and built upon. This project just shows the gist of the streaming microservice.

DATABASE STRUCTURE

1) Entity Relationship Diagram (MySQL Relational DB)



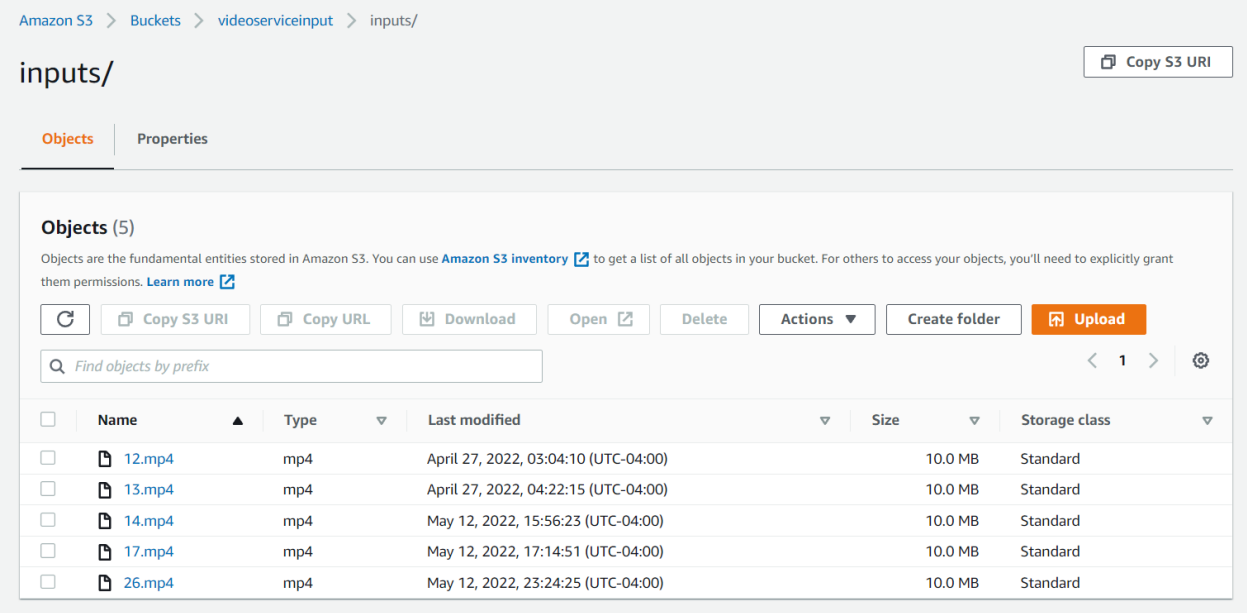
In the above relational schema, every user will have one userInfo. Every User will have many userVideos. And every UserVideo is associated to one Video through the foreign key videoID.

I have not implemented the userInfo entity due to time constraints. But a similar approach can be taken to implement it.

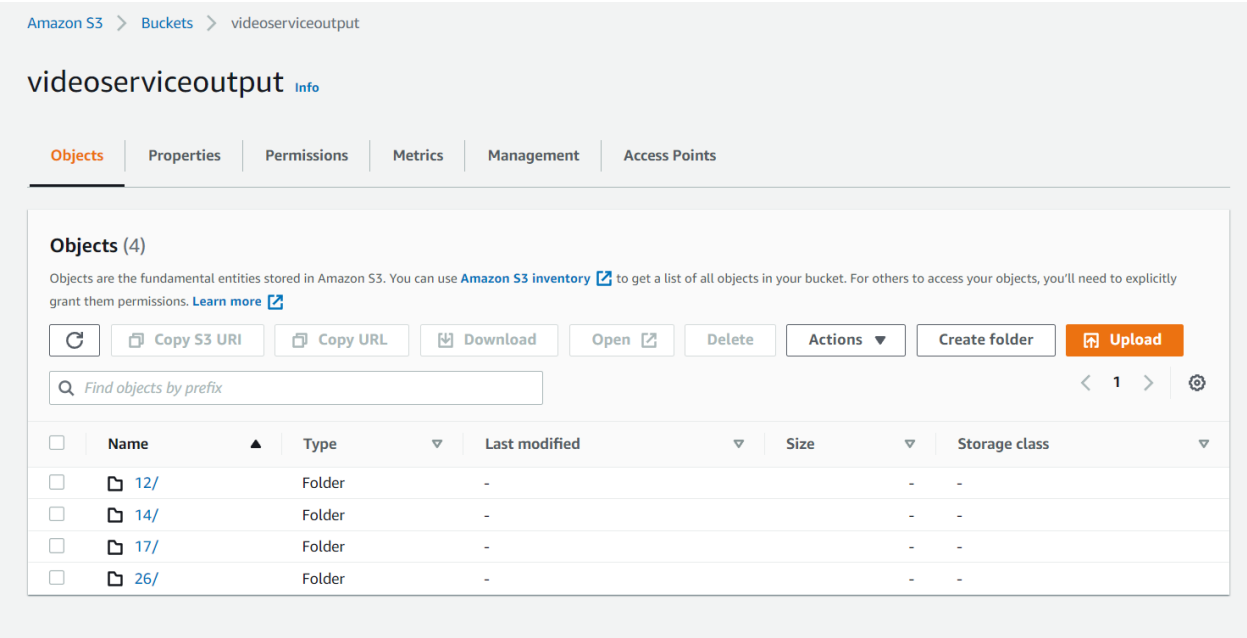
2) Amazon S3 (No SQL DB)

In the Amazon S3, we have two buckets.

- **Input bucket:** In which the media file uploaded by the user using the REST endpoints are placed. Below is the screenshot of the input S3 bucket. My input bucket is called "**videoserviceinput**".

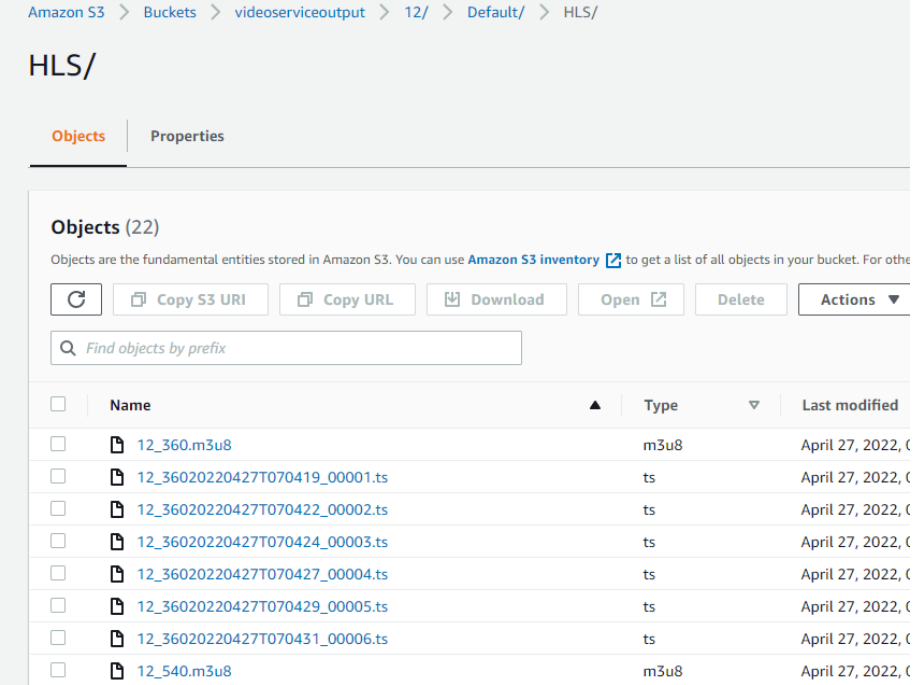


- **Output bucket:** In which the media file converted into Apple HLS format is placed by the Media Convert. Below is the screenshot of the output S3 bucket. My output bucket is called "**videoserviceoutput**".

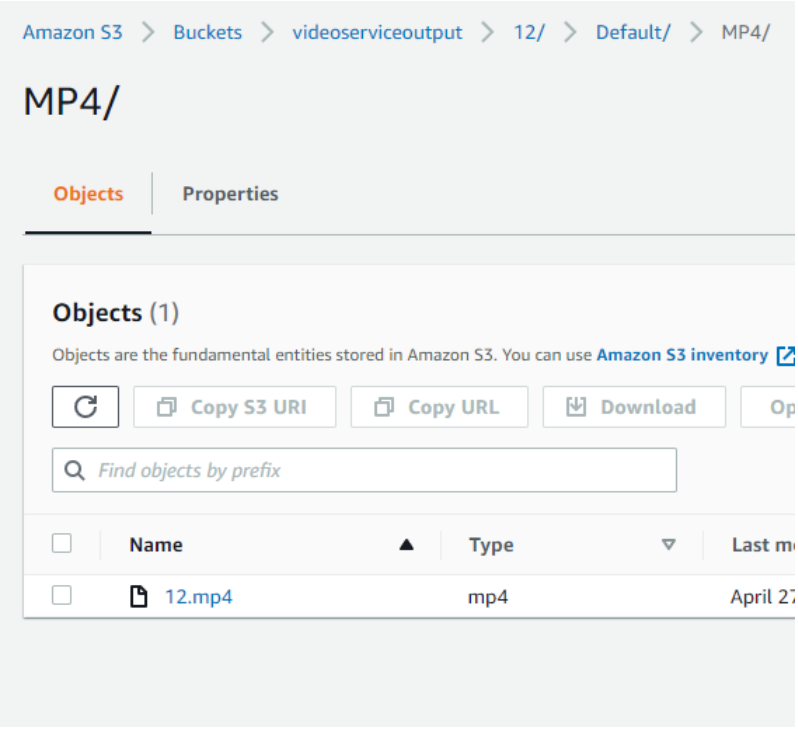


Every video file will have the path of **“<videoID”>/Default/”**. Under the default path there are three sections as listed below:

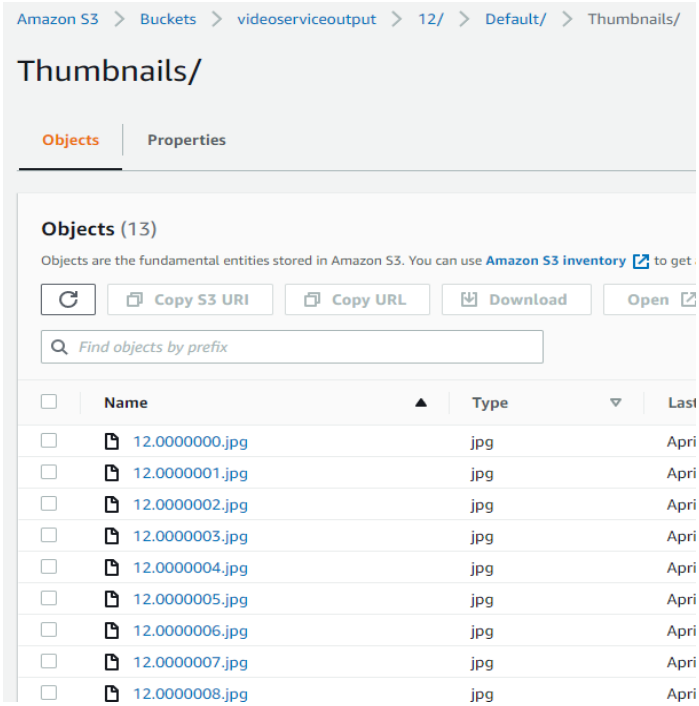
- **HLS:** this contains the segmented video files, along with the manifest file for 3 different qualities i.e., 360p, 540p, and 720p as shown below:



- **MP4:** This contains the original video file uploaded by the user.

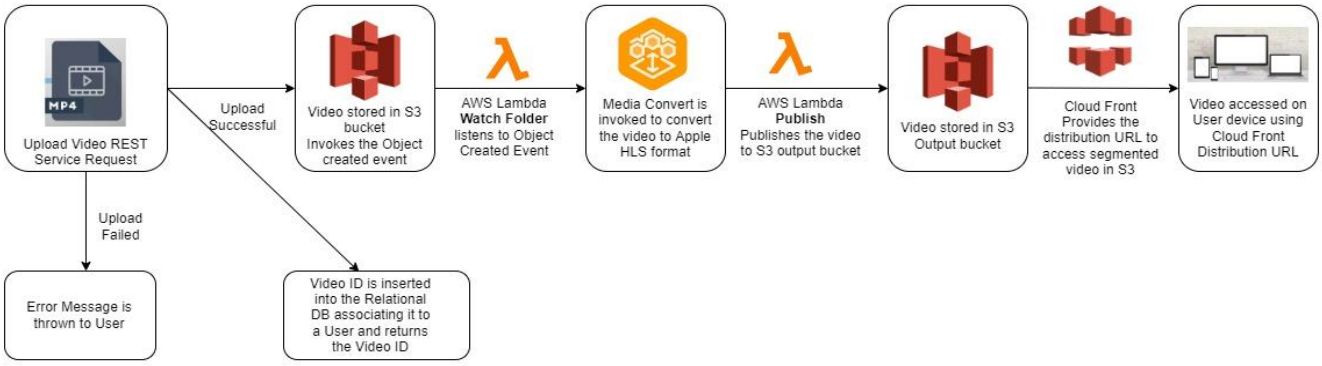


- **Thumbnails**: This contains images captured from the videos at a specified interval to be used as thumbnails for the video in the client's media player.



WORKFLOW

Below diagram shows the workflow after the users makes a request to upload a media file.



1) Once the video upload request is made through the REST service, an asynchronous call is made to the S3 bucket to upload the media file.
2) If the upload fails, the user is notified with an error message.
3) If the upload is successful, a unique ID is generated and inserted into the relational DB with the metadata of the video like description, title etc. As shown in the below snapshot.

	videoId	activeInd	description	fileExtension	originalFileName	title
▶	14	1	sample video akhil	mp4	SampleVideo_1280x720_10mb.mp4	test

Record inserted in the Video table

	videoId	userId
▶	14	1

Record inserted into the uservideo table mapping a video to a user

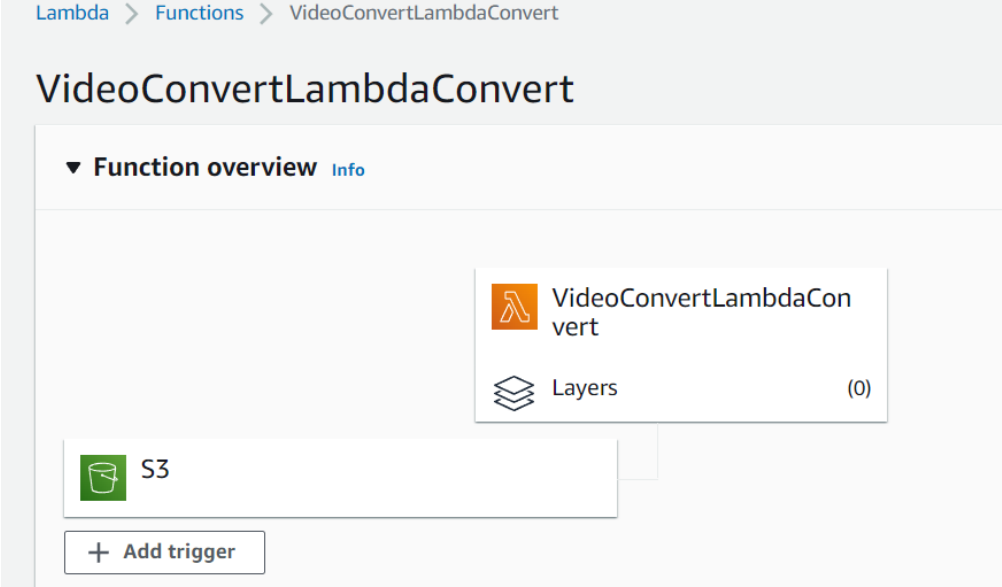
	userId	accountCreationDate	activeInd	emailId	password
▶	1	2022-04-27 02:25:42.636280	1	aajitha@syr.com	akhil123

User table having user data along with the ID

4) Once the file is uploaded in the S3 input bucket, an object created event is triggered.

Video file uploaded to the S3 input bucket which triggered an object created event

5) AWS Lambda Watch folder reacts on this event and create a job and adds it to the queue of Media Convert to process the media file into Apple HLS format.



Lambda function have a watch folder on the S3 input bucket.

On object created event, it executes the python script “convert.py” which uses the “job.json” to instruct the Media Convert to convert the media file to Apple HLS format.

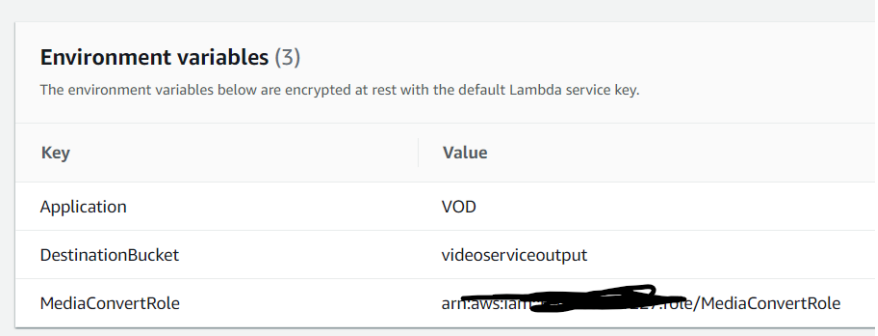
convert.py | job.json | Execution results

```
{
    "OutputGroups": [
      {
        "CustomName": "HLS",
        "Name": "Apple HLS",
        "Outputs": [
          {
            "ContainerSettings": {
              "Container": "M3U8",
              "M3u8Settings": {
                "AudioFramesPerPes": 4,
                "PcrControl": "PCR_EVERY_PES_PACKET",
                "PmtPid": 480,
                "PrivateMetadataPid": 503,
                "ProgramNumber": 1,
                "PatInterval": 0,
                "PmtInterval": 0,
                "Scte35Source": "NONE",
                "TimedMetadata": "NONE",
                "VideoPid": 481,
                "AudioPids": [
                  482,
                  483,
                  484,
                  485,
                  486,
```

Job.json file containing video encoding instructions(job settings)

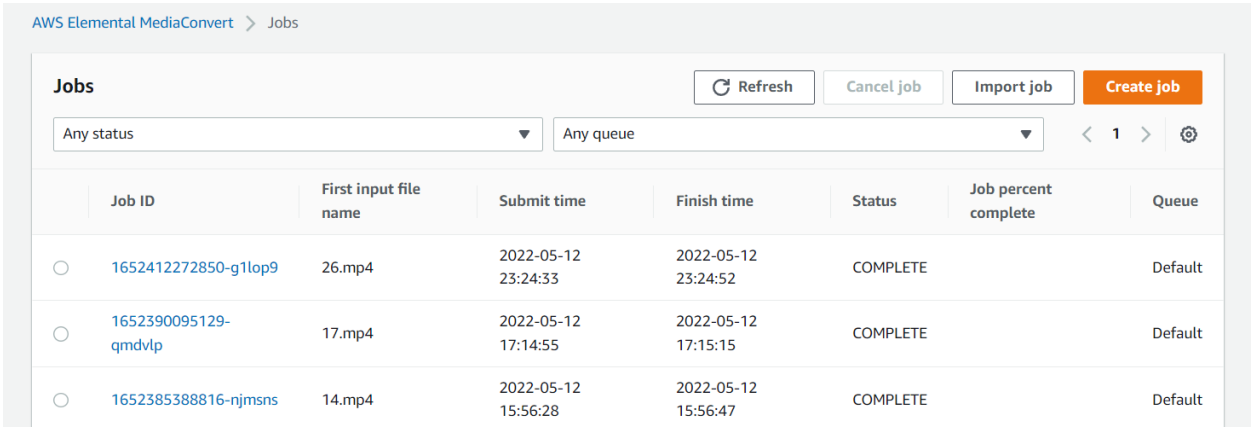
Below code in covert.py creates a job in the media convert using the job settings.

```
# Convert the video using AWS Elemental MediaConvert
job = client.create_job(Role=mediaConvertRole, UserMetadata=jobMetadata, Settings=jobSettings)
```

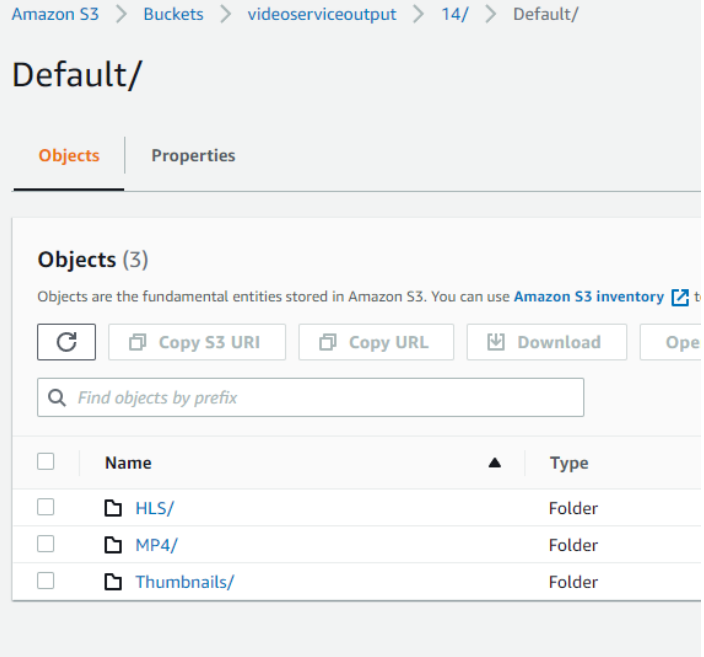


Environment variables in the AWS Lambda specifying the destination bucket and the ARN of the media convert

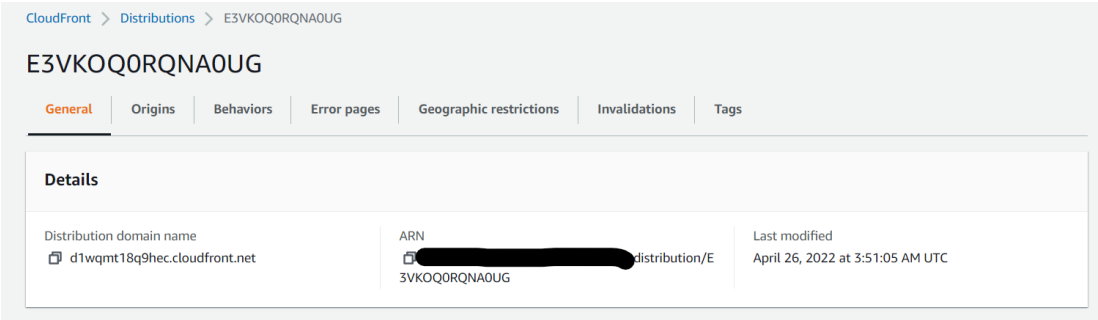
Media Convert works on the producer consumer architecture. Where jobs are added to the queue and are processed one at a time. Below is the snapshot of the jobs that were processed.



6) Once the media file is processed, the lambda function is notified which pushes the media file to the respective output S3 bucket.



7) Cloud front is configured to provide a distribution link which is used to access the media file from the S3 bucket. Below is the snapshot of the Cloud front distribution.



The URL of the distribution is **"d1wqmt18q9hec.cloudfront.net"**

8) The video is played on the client device using the distribution URL. For the demo purpose I have used a code from online source. The client here is "video.js" which supports playing of adaptive bit rate media file like Apple HLS by using the manifest file.

The distribution URL of this video is https://d1wqmt18q9hec.cloudfront.net/14/Default/HLS/14.m3u8 where the first half of URL is the cloud front domain name which is followed by the S3 bucket's video path structure. Here we can see that S3 bucket's domain name is not exposed to the client.

```
  <source
src="https://d1wqmt18q9hec.cloudfront.net/14/Default/H
LS/14.m3u8" type="application/x-mpegURL">
</video>
</div>
```

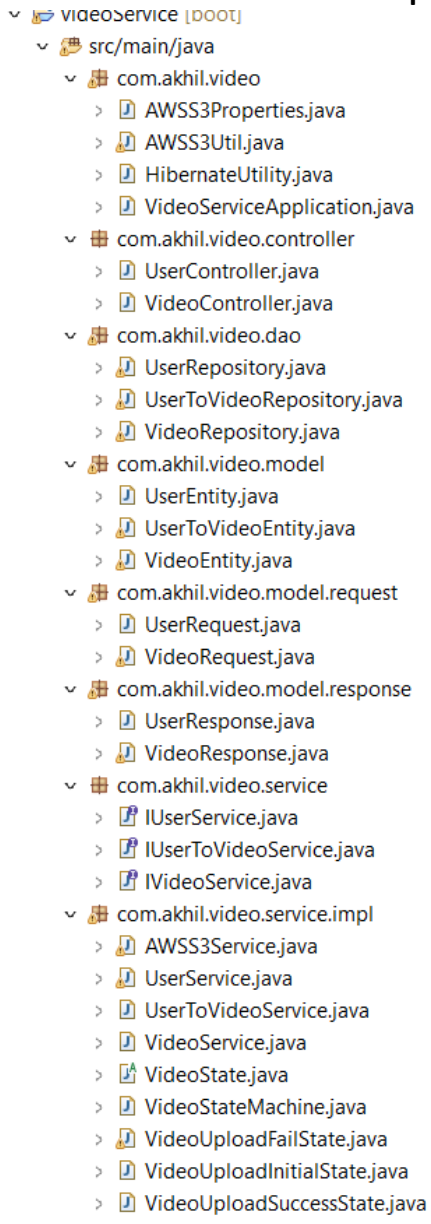
Snippet of the JS code to play the video

Code for this is sourced from: https://codepen.io/a4h8/pen/XLRoJQ?editors=1010

DESIGN PATTERNS

1) **MVC Pattern:** This pattern separates the model, view, and controller such that the model updates the view, controller manipulates the model, user sees the changes made to the view by the model, and the user requests changes to the model through the controller. Since this project contains only the backend rest services and doesn't have any UI element, it doesn't contain the View part of the logic. The View part can be implemented easily with the existing model and controller project structure.

Below is the screenshot of the project structure:



- **AWSS3Properties** -> is used to fetch all the configuration properties from the configuration.properties file which contains token to connect to S3 bucket and use aws services.
- **AWSS3Util** -> It's a singleton which returns an aws client meant to access aws s3 services.
- **HibernateUtil** -> Creates a singleton instance of the session factory.
- **VideoServiceApplication** -> This is the starting point of the spring boot application.
- **com.akhil.video.controller** package -> this contains the user and video controllers, which responds to the user requests through REST endpoints.
- **com.akhil.video.dao** package -> These are data access objects meant to interact with the backend database. There is one class for each of the entities.
- **com.akhil.video.model** package -> These contains POJO classes which represents the object representation of the entities in the Database.
- **com.akhil.video.model.request** package -> These are meant to act as a layer of abstraction, so every user request is mapped to the request object. Then these objects are manipulated and mapped to the Entity models to perform transactions with the database.
- **com.akhil.video.model.response** package -> These are meant to act as a layer of abstraction. All the responses sent back to the user is first mapped from entity class to response class, so that the user doesn't know the underlying implementation.
- **com.akhil.video.service** package -> These contain the interfaces of the services, which are to be implemented.
- **com.akhil.video.service.impl** package -> Contains all the business logic and all the services are present in this package. The controller interacts with these services to perform the business logic. These services in turn interacts with the models.

2) **Singleton:** It's a creational design pattern which creates only a single object of the class/type throughout the lifecycle of the application. In the context of the project, we are building a REST service performing CRUD operations on the underlying database. In this scenario creating a connection to the database i.e., creating a session is a performance intensive operation. So, it would be advantageous to create only a single session object and be shared across the application.
Singleton pattern has been applied twice in this project:

- **Creation of session factory by Spring Boot**

  In this project I have used Spring Boot and the session factory is created as a bean, which is managed by the Spring IoC container. As per the definition it would create only a single instance of the bean per container throughout the lifecycle of the application. Here the session factory is a bean and only a single instance of it would be created by Spring.

  Below is the snippet of code showing the creation of the Session Factory and Transaction Manager. Refer to `com.akhil.video.HibernateUtility` class for code containing the creation of these objects.

```java
/**
 * Singleton class to get a Hibernate session factory. Spring manages the
 * creation of the instance and will create only one session per container,
 * achieving singleton pattern.
 *
 * @author akhil
 */
@Configuration
@EnableTransactionManagement
public class HibernateUtility {
    @Value("${db.driver}")
    private String jdbcDriver;

    @Value("${db.password}")
    private String dbPassword;

    @Value("${db.url}")
    private String dbUrl;

    @Value("${db.username}")
    private String dbUserName;

    @Value("${hibernate.dialect}")
    private String hibernateDialect;

    @Value("${hibernate.show_sql}")
    private String hibernateShowSql;

    @Value("${hibernate.hbm2ddl.auto}")
    private String hibernateDdl;

    @Value("${entitymanager.packagesToScan}")
    private String entityManagerPackages;
```

```java
/**
 * Builds a hibernate transaction manager for a single Hibernate SessionFactory.
 * Binds a Hibernate Session from the specified factory to the
 * thread,potentially allowing for one thread-bound Session per factory.
 *
 * @return The hibernate transaction manager.
 */
@Bean
public HibernateTransactionManager transactionManager() {
    HibernateTransactionManager transactionManager = new HibernateTransactionManager();
    transactionManager.setSessionFactory(sessionFactory().getObject());
    return transactionManager;
}

/**
 * Factory Bean that creates a Hibernate SessionFactory. It is shared across the
 * spring application context achieving a singleton object per container.
 *
 * @return The Session Factory Bean.
 */
@Bean
public LocalSessionFactoryBean sessionFactory() {
    LocalSessionFactoryBean sessionFactory = new LocalSessionFactoryBean();
    sessionFactory.setDataSource(buildDataSource());
    sessionFactory.setPackagesToScan(entityManagerPackages);
    sessionFactory.setHibernateProperties(buildHibernateProperties());

    return sessionFactory;
}
```

- **Creation of Amazon S3 client**

  S3 client is a heavy object, and a singleton of this client will improve the performance. Below are the screenshots for the Amazon S3 client singleton. Refer to `com.akhil.video.AWSS3Util` class for detailed implementation.

```java
/**
 * Singleton instance of the amazon client.
 *
 * @author akhil
 */
@Component
public class AWSS3Util {
    private static AmazonS3 awsS3Client;

    /**
     * Private class to prevent instantiation.
     */
    private AWSS3Util() {
    }

    /**
     * Returns the instance of AWS S3 client. Creates the client only if it's not
     * created before.
     *
     * @return The {@link AmazonS3} instance to access S3 bucket.
     */
    public static synchronized AmazonS3 getAWSClient() {
        if (awsS3Client == null) {
            awsS3Client = createAWSS3Client(createAWSCredentials(), Regions.US_EAST_1);
        }

        return awsS3Client;
    }
```

3) **State Pattern:** It's a creational design pattern that allows an object to alter its behavior when the internal state changes. It looks like if the object changed its class.
In this project, I have used the state pattern while uploading the video to the S3 bucket. Before uploading the video, a unique video ID must be generated. This is done by inserting a record into the relational DB, which would return a video ID. Now the file is uploaded to the S3 bucket, if the upload fails, then the record inserted into the relational DB must be deleted or else it should return the video ID. Here we can see that there are three states, i.e., the initial state before the upload for which `VideoUploadInitialState.class` has been created. The initial class has a reference to the state machine, as it'll transition to success or fail state and the same must be updated as the current state. Then there are two possible states to which it might transition into i.e., the success or fail for which `VideoUploadSuccessState.class` and `VideoUploadFailState.class` has been created respectively. These classes don't have a reference to the state machine as these are the end states. All these classes extend an abstract class `VideoState.class` and override the `handleRequest` method. Now there's a state machine called `VideoStateMachine.class` which is maintains and transitions between these states.
The Initial State class on handle request, will try to upload the file to S3 bucket, and if it succeeds then it moves to Success state or else it'll move to fail state and invoke it's handleRequest method which will delete the record of the video from the relational DB. The state pattern helps reduce the cyclomatic complexity.
The same pattern can also be applied while deleting the objects from S3 bucket. Not implemented here due to time constraints.
Below are the screenshots showing the implemented pattern. Refer to the above classes in `com.akhil.video.service.impl` package for detailed implementation.

```java
/**
 * Abstract class to maintain the state of the video.
 *
 * @author akhil
 */
public abstract class VideoState {
    final static Logger logger = Logger.getLogger(VideoState.class);

    /**
     * Takes the file and process it based on the state.
     *
     * @param file    The file to be processed.
     * @param video   The video entity.
     * @param videoId The video Id of the video.
     */
    public void handleRequest(MultipartFile file, VideoEntity video, Long videoId) {
        logger.error("This state should not be accessible. Error occured.");
    }
}
```

```
 */
@Component
public class VideoStateMachine {
    VideoState videoUploadInitialState;
    VideoState videoUploadSuccessState;
    VideoState videoUploadFailState;

    VideoState videoState;

    /**
     * Creates all the states and then sets the initial state of the machine.
     */
    public VideoStateMachine() {
        videoUploadInitialState = new VideoUploadInitialState(this);
        videoUploadSuccessState = new VideoUploadSuccessState();
        videoUploadFailState = new VideoUploadFailState();

        videoState = videoUploadInitialState;
    }

    /**
     * Setter to set the video state.
     *
     * @param videoState The video state to be set as the current state.
     */
    public void setVideoState(VideoState videoState) {
        this.videoState = videoState;
    }
```

```
/**
 * The class representing the initial state of the video upload.
 *
 * @author akhil
 */
public class VideoUploadInitialState extends VideoState {
    private VideoStateMachine videoStateMachine;

    public VideoUploadInitialState(VideoStateMachine videoStateMachine) {
        this.videoStateMachine = videoStateMachine;
    }

    @Override
    public void handleRequest(MultipartFile file, VideoEntity video, Long videoId) {
        AWSS3Service awsS3Service = new AWSS3Service();
        try {
            if (awsS3Service.upload(videoId.toString(), video.getFileExtension(), file.getInputStream(),
                    file.getSize())) {
                logger.info("Video has been uploaded successfully. Moving it to success state.");
                videoStateMachine.setVideoState(videoStateMachine.getVideoUploadSuccessState());
            }
            else
            {
                logger.info("Video has failed to upload. Moving it to failed state");
                videoStateMachine.setVideoState(videoStateMachine.getVideoUploadFailState());
                videoStateMachine.getCurrentState().handleRequest(file, video, videoId);
            }
        } catch (Exception exception) {
            logger.error(exception);
            logger.info("Video has failed to upload. Moving it to failed state");
            videoStateMachine.setVideoState(videoStateMachine.getVideoUploadFailState());
            videoStateMachine.getCurrentState().handleRequest(file, video, videoId);
        }
    }
}
```

```java
/**
 * The class representing the failed state of video upload.
 *
 * @author akhil
 *
 */
public class VideoUploadFailState extends VideoState {
    @Override
    public void handleRequest(MultipartFile file, VideoEntity video, Long videoId) {
        logger.info("Video has been uploaded successfully.");
        VideoRepository videoRepository = new VideoRepository();
        videoRepository.hardDelete(videoId);
    }
}

/**
 * The class representing the success state of the video upload.
 *
 * @author akhil
 *
 */
public class VideoUploadSuccessState extends VideoState {
    @Override
    public void handleRequest(MultipartFile file, VideoEntity video, Long videoId) {
        logger.info("Video has been uploaded successfully.");
    }
}
```

Usage:

```java
@Override
public long addVideo(String title, String description, MultipartFile file, long userId) {
    VideoEntity video = VideoEntity.builder().title(title).description(description)
            .originalFileName(file.getOriginalFilename())
            .fileExtension(FilenameUtils.getExtension(file.getOriginalFilename())).activeInd(1).build();
    Long videoId = videoRepository.addVideo(video);

    VideoStateMachine stateMachine = new VideoStateMachine();
    stateMachine.getCurrentState().handleRequest(file, video, videoId);

    if (stateMachine.getCurrentState() instanceof VideoUploadFailState) {
        logger.error("Video upload failed. Video entries will be deleted from DB.");
        return -1;
    } else {
        userToVideoService.addUserToVideo(userId, videoId);
    }
    return videoId;
}
```

4) **Builder Pattern:** It's a creational design pattern that allows creating complex objects step by step. It produces different types and representation of the object using the same construction code.
In this project, it is used to build the entities without using telescoping constructors or repeated setter calls. Using the builder will let us build the object step by step, improving readability and maintainability.
Below is the snippet of the builder pattern in the `com.akhil.video.model.VideoEntity.class.`

```
/**
 * Create {@link VideoEntity} from the builder.
 *
 * @param builder The builder.
 */
private VideoEntity(VideoEntityBuilder builder) {
    this.videoId = videoId;
    this.title = builder.title;
    this.description = builder.description;
    this.originalFileName = builder.originalFileName;
    this.fileExtension = builder.fileExtension;
    this.activeInd = builder.activeInd;
}

/**
 * Returns the builder for {@link VideoEntity}.
 *
 * @return The builder.
 */
public static VideoEntityBuilder builder() {
    return new VideoEntityBuilder();
}

/**
 * Create a builder with the passed in video entity.
 *
 * @param videoEntity The video entity.
 * @return The builder.
 */
public static VideoEntityBuilder builder(VideoEntity videoEntity) {
    return new VideoEntityBuilder(videoEntity);
}
```

```java
/**
 * Builder for the Video Entity class.
 *
 * @author akhil
 */
public static class VideoEntityBuilder {
    private long videoId;
    private String title;
    private String description;
    private String originalFileName;
    private String fileExtension;
    private int activeInd;

    public VideoEntityBuilder() {
    }

    public VideoEntityBuilder(VideoEntity videoEntity) {
        this.videoId = videoEntity.videoId;
        this.title = videoEntity.title;
        this.description = videoEntity.description;
        this.originalFileName = videoEntity.originalFileName;
        this.fileExtension = videoEntity.fileExtension;
        this.activeInd = videoEntity.activeInd;
    }

    public VideoEntity build() {
        return new VideoEntity(this);
    }

    public VideoEntityBuilder title(String title) {
        this.title = title;
        return this;
```

The same has been implemented for the other entity classes `UserEntity` and `UserToVideoEntity.`

The **usage** is shown in `com.akhil.video.service.impl.VideoService class.`

```java
@Override
public long addVideo(String title, String description, MultipartFile file, long userId) {
    VideoEntity video = VideoEntity.builder().title(title).description(description)
            .originalFileName(file.getOriginalFilename())
            .fileExtension(FilenameUtils.getExtension(file.getOriginalFilename())).activeInd(1).build();
```

Other than the builder implemented by me, I have used the String builder class provided by java.lang to build the URL of the video step by step. Below is the code snippet for it from the VideoService class.

```
@Override
public String getVideoUrlById(long videoId) {
    StringBuilder urlBuilder = new StringBuilder();
    urlBuilder.append(awss3Properties.getAWSProperties().get("aws.cloudfront.url"));
    urlBuilder.append("/").append(String.valueOf(videoId));
    urlBuilder.append("/Default/HLS/");
    urlBuilder.append(String.valueOf(videoId));
    urlBuilder.append(".m3u8");

    return urlBuilder.toString();
}
```

The same builder pattern is also used to build the response entity object in the controller.

5) **Strategy Pattern:** It's a behavioral pattern that defines a family of algorithms that puts each of them into a separate class and make their objects interchangeable.
   This pattern can be seen being used by that javax.persistance's GeneratedValue which changes the strategy to be used for generating the ID field of the entity. Below is the snippet of the usage in `UserEntity` class. It's found in the `com.akhil.video.model` package.

```
@Entity
@Table(name = "user", uniqueConstraints = { @UniqueConstraint(colum
public class UserEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE)
    private long userId;
```

6) **Adapter Pattern:** It's a structural design pattern that allows objects with incompatible types to collaborate with each other.
   I have used it through the `org.modelmapper.ModelMapper` class. This map the Entity to response objects and request to Entity objects. Below is the snippet of code from the `com.akhil.video.service.impl.UserService` class.

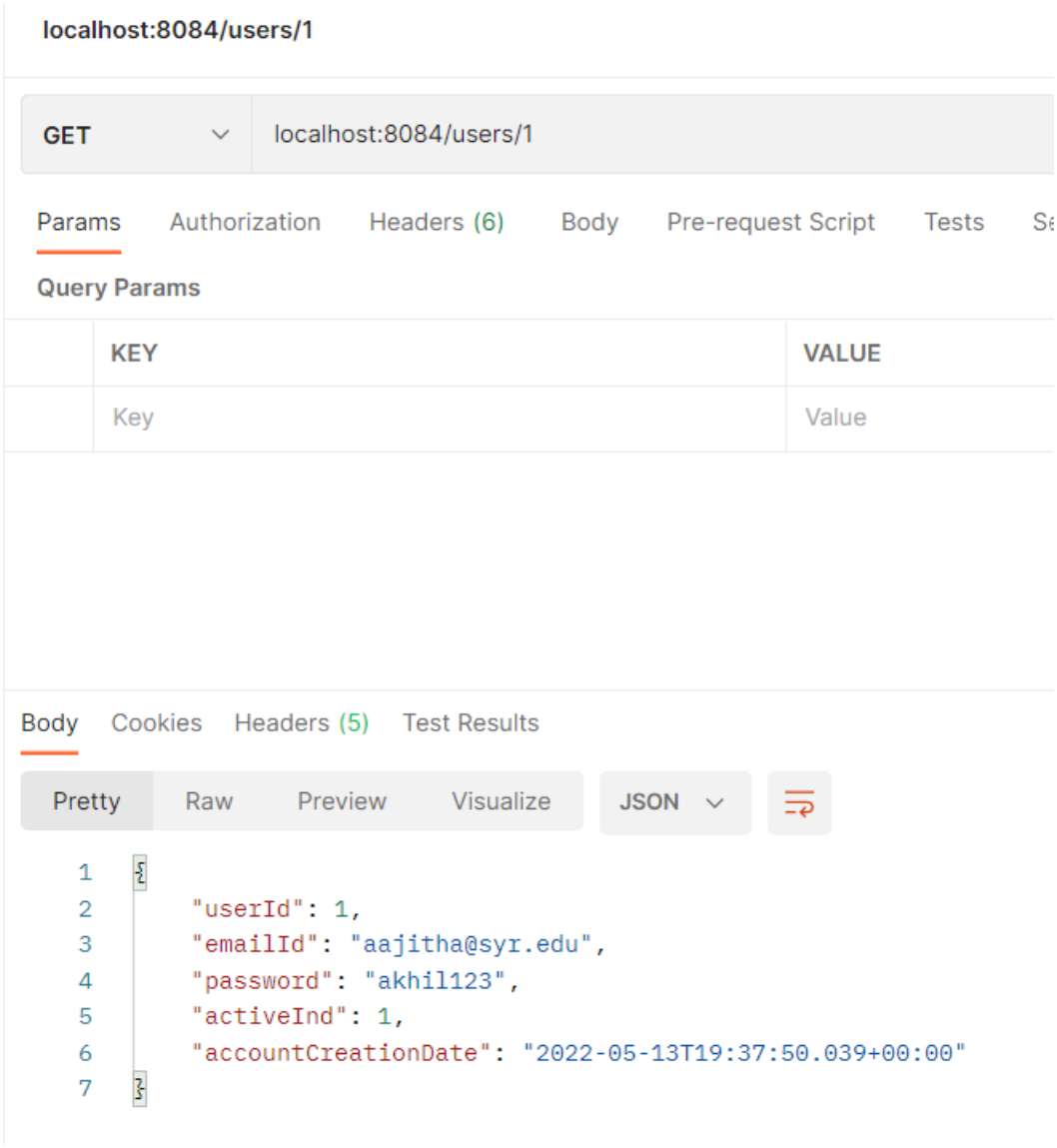
```
@Override
public UserResponse getUserByID(long userId) {
    UserEntity userEntity = userRepository.getUserById(userId);
    if (userEntity != null) {
        ModelMapper modelMapper = new ModelMapper();
        UserResponse userResponse = modelMapper.map(userEntity, UserResponse.class);
        return userResponse;
    }

    return null;
}
```

As seen above the UserEntity object is mapped to the UserResponse object using the model mapper, which internally uses the adapter pattern.
Another example of it is java.util.Arrays.asList(..) method, which I have used in the project.

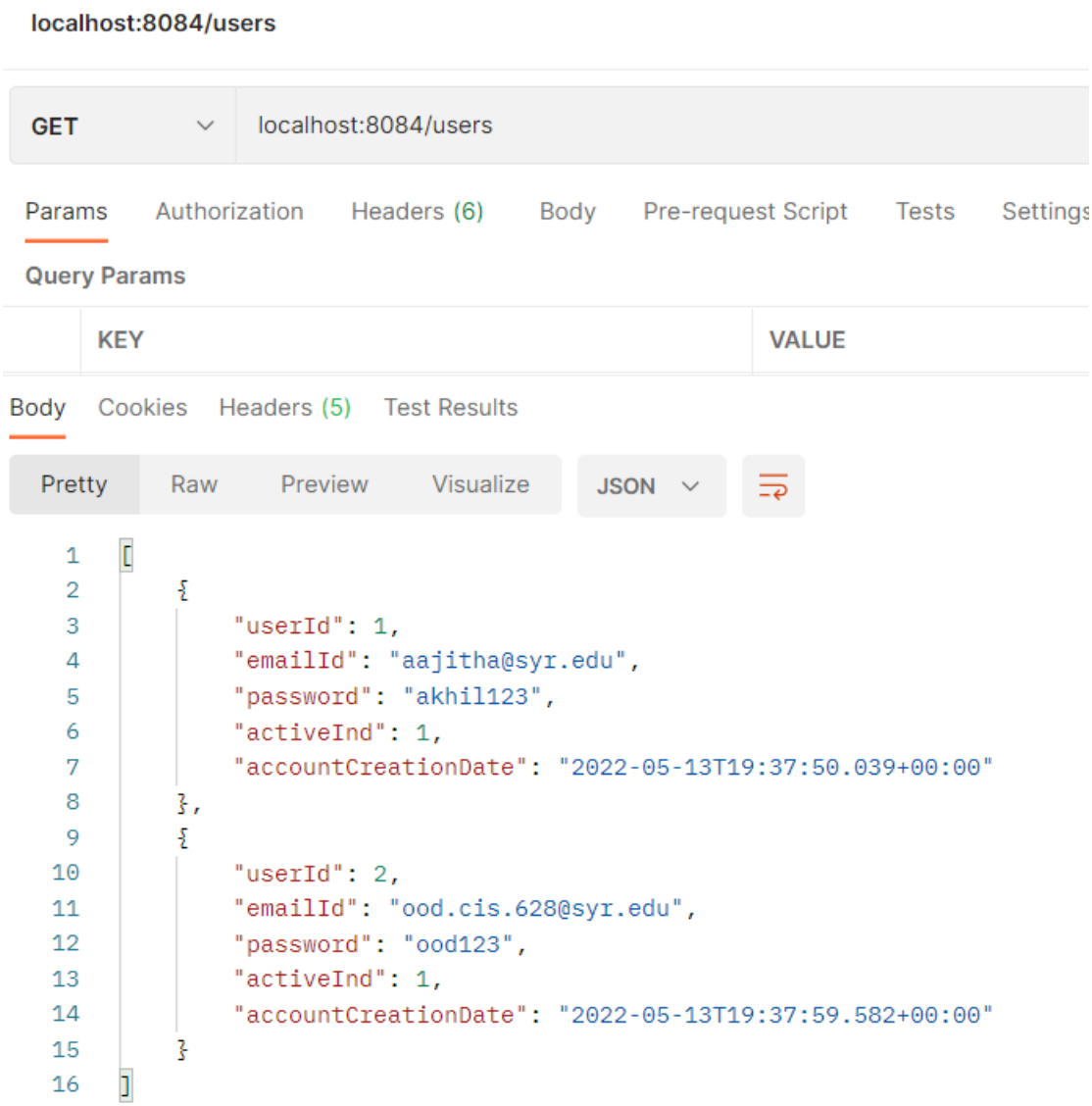
REST ENDPOINTS AND POSTMAN EVIDENCE

Below are the REST endpoints categorized by controllers. All the controllers are present in `com.akhil.video.controller` package.

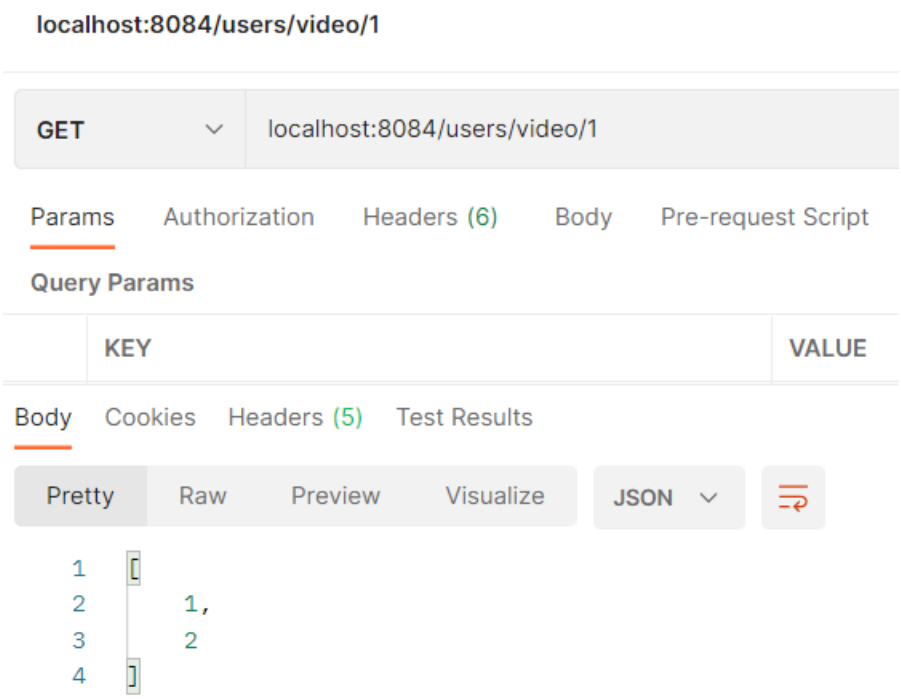
1) **UserController**
   - **Get Requests:**
     - **To get the user details by User ID.** It takes the userID as a path param.
       The Rest endpoint URL example: **localhost:8084/users/1**
       Returns the user details for the respective userID.



     - **To fetch all the users.**
       The Rest endpoint URL example: **localhost:8084/users**
       Returns the list of users in the database



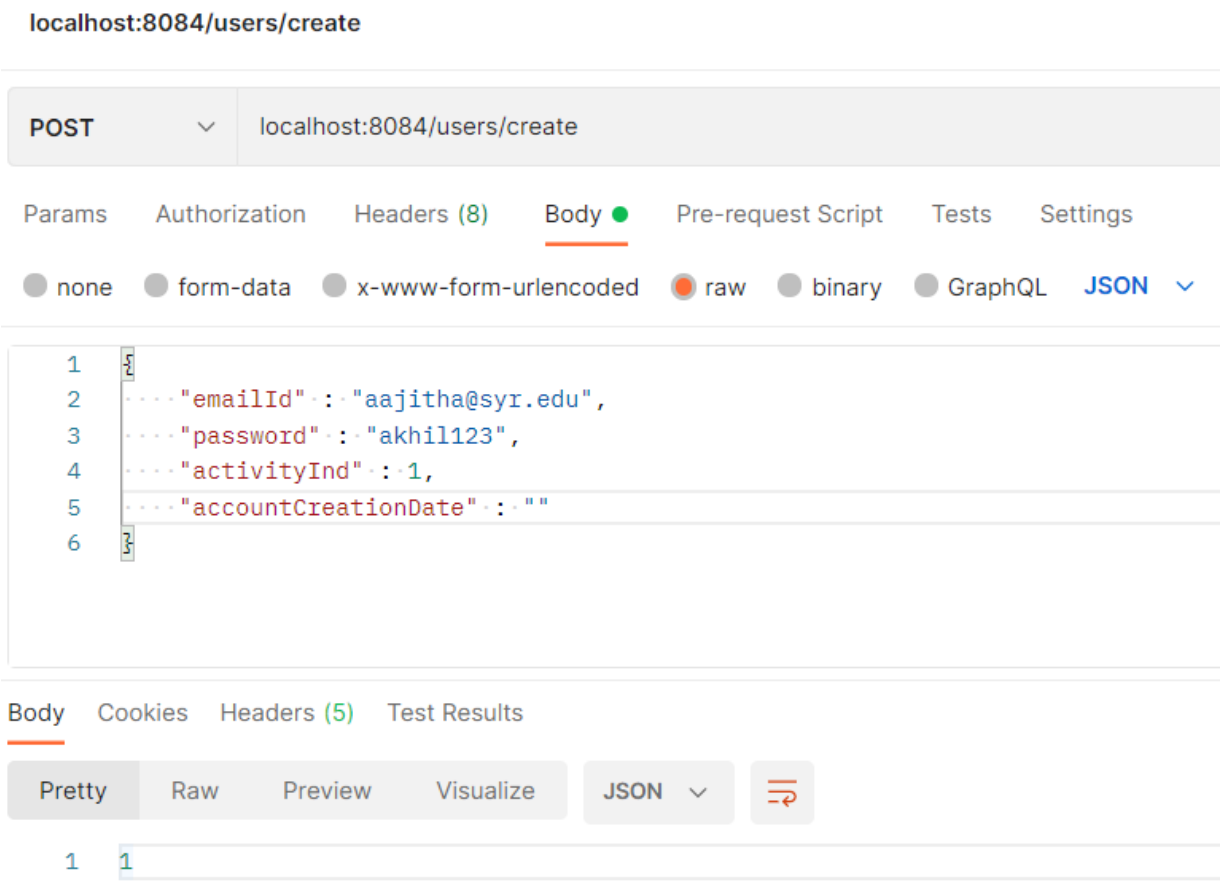
- **To fetch video IDs associated to a User.**
  It takes user Id as the path param.
  Rest endpoint URL example: **localhost:8084/users/video/1**
  Returns the video IDs, associated to the user.



- **Post requests:**
  - To create a user.
    Rest endpoint URL example: **localhost:8084/users/create**
    This takes a json body with a set of properties. Like:

```
{
    "emailId" : "aajitha@syr.edu",
    "password" : "akhil123",
    "activityInd" : 1,
    "accountCreationDate" : ""
}
```

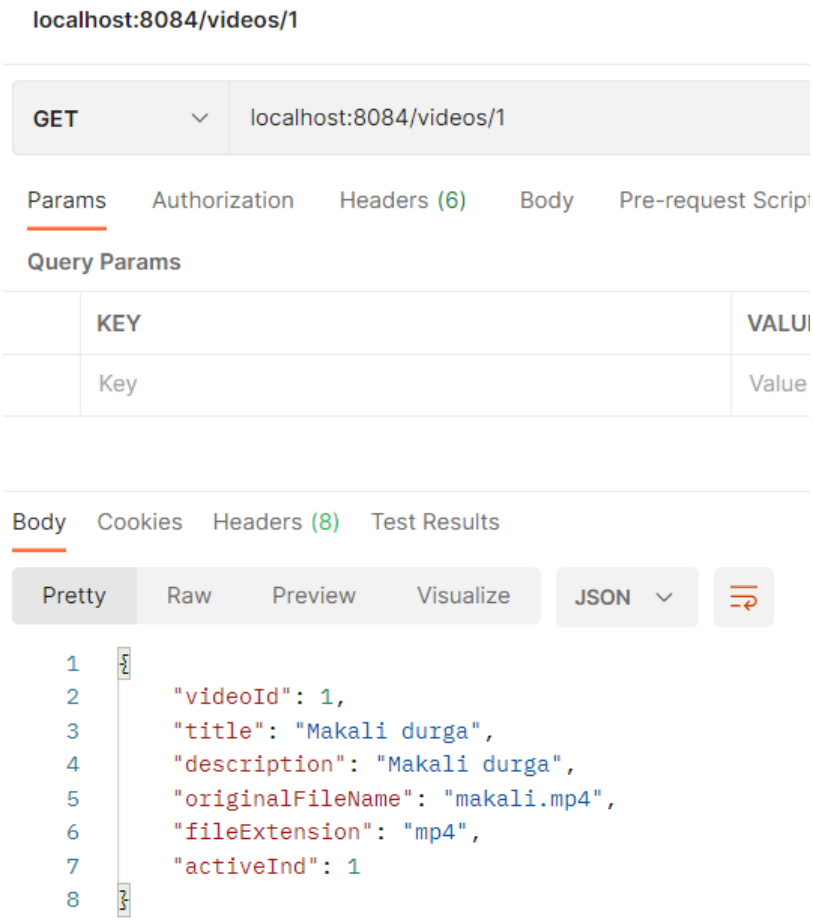
    The post request returns the ID of the newly created user.



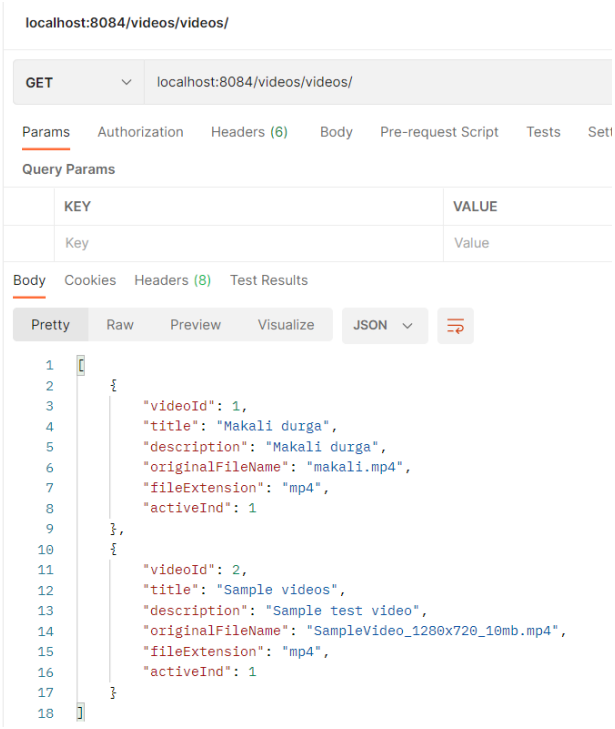
2) **VideoController**

- **Get Requests:**
  - **Gets videos details by video ID**
    Takes video ID as the path variable.
    Returns the video details as JSON object.

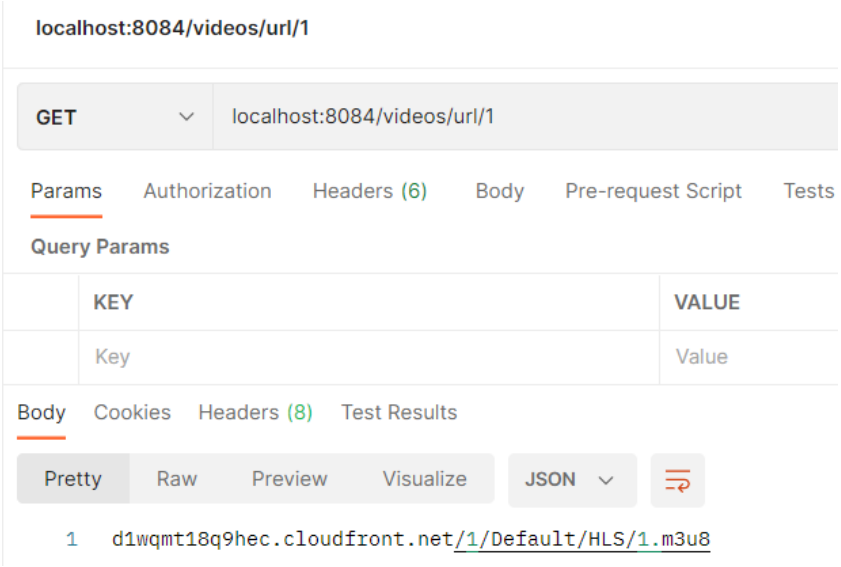
REST endpoint URL: **localhost:8084/videos/1**



- **Get all videos in the system**
  Rest endpoint URL: **localhost:8084/videos/videos/**
  Returns all the videos in the system.

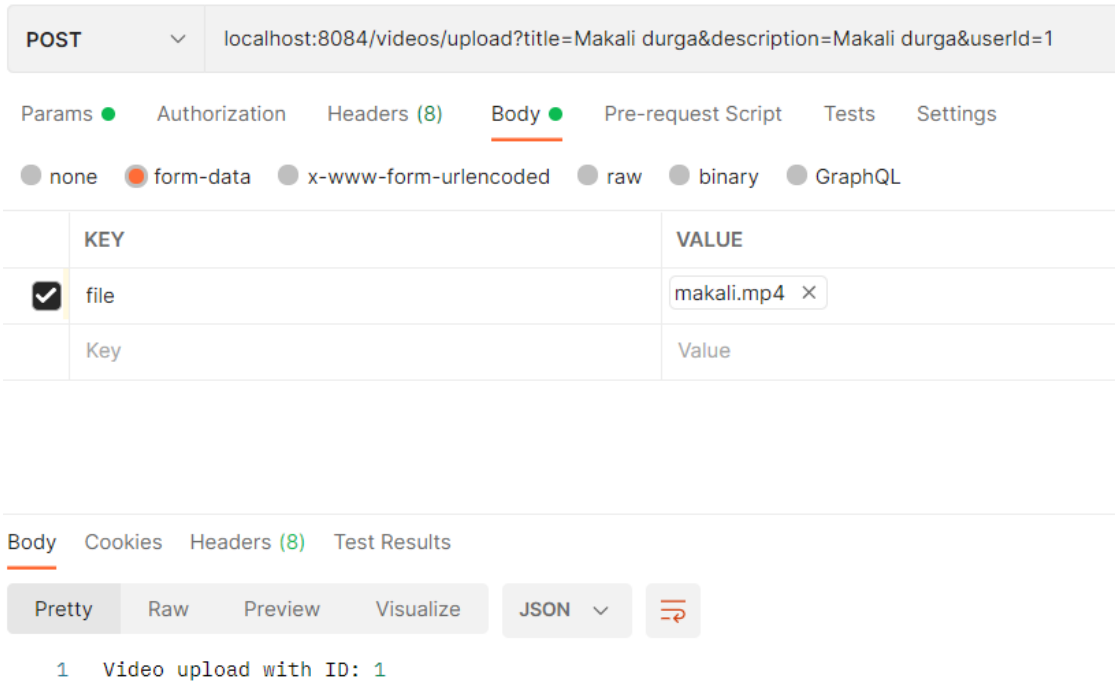


- **Get video URL by ID**
  Generates the cloud front distribution URL for a video by it's ID.
  Rest endpoint URL: **localhost:8084/videos/url/1**
  Returns the cloud front distribution URL for the video.



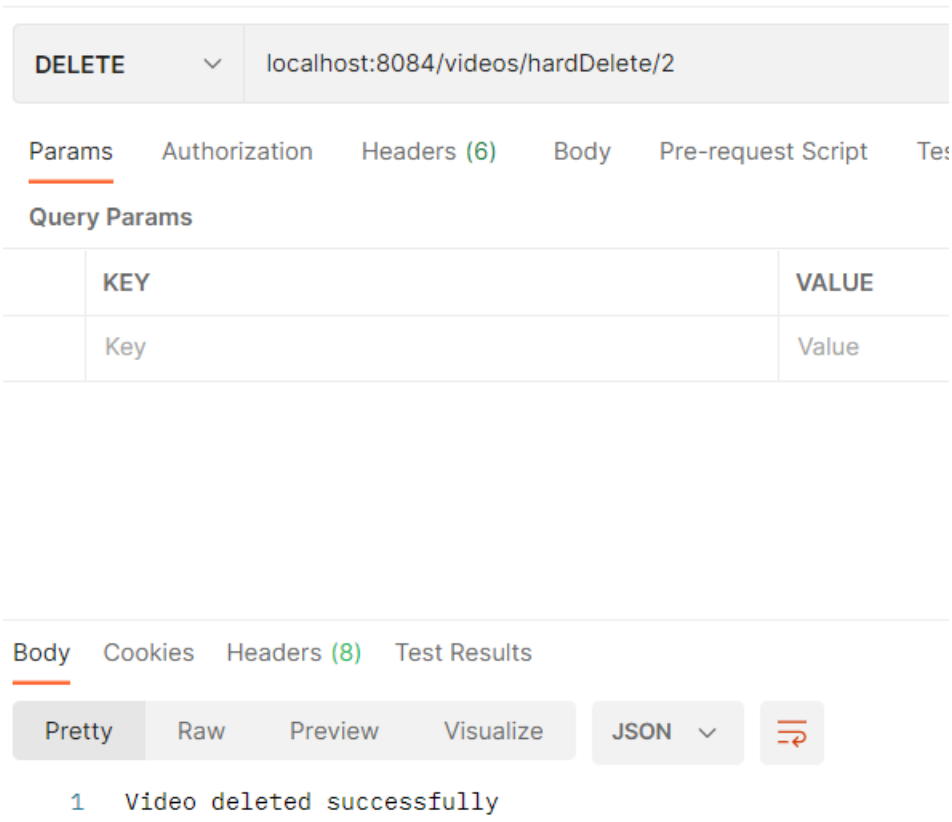
- **Post Requests:**
  - **Upload video.**
    It takes 3 request params for title, description and the userID. It also takes a multipart file from the form data.
    Rest endpoint URL will look like:
    localhost:8084/videos/upload?title=Makali durga&description=Makali durga&userId=1
    It will also take a form data as shown in the below screenshot.
    Returns the video ID of the newly created video.

localhost:8084/videos/upload?title=Makali durga&description=Makali durga&userId=1



- **Delete Requests:**
  - **Performs a hard delete for the video**
    It deletes the video from the S3 bucket and its associated data in the relational DB.
    Takes the video Id as path param.
    Returns the status of delete.

localhost:8084/videos/hardDelete/2



There are a couple more rest which I haven't shown in this report, because these rest calls show the working of the prototype system.

FUTURE WORK

The below will be implemented going forward to make the system viable for use:

1) Factory pattern can be implemented to switch between various cloud providers.
2) Spring Security
3) OAuth to be integrated with Front-end
4) Complete the REST services to make the system work for all CRUD operations.
5) Implement the UserInfo entity and Services.

STEPS TO EXECUTE THE PROJECT

1) Have mysql DB installed.
2) Set this `hibernate.hbm2ddl.auto` value to `create` in application.properties.
3) Build the project using maven install command.
4) Run the springboot project.
5) I have included the access key and token, but it'll be deactivated soon for AWS access.