# Backend Creation (by Manjeet)

first run npm init

npm install -D nodemon prettier,

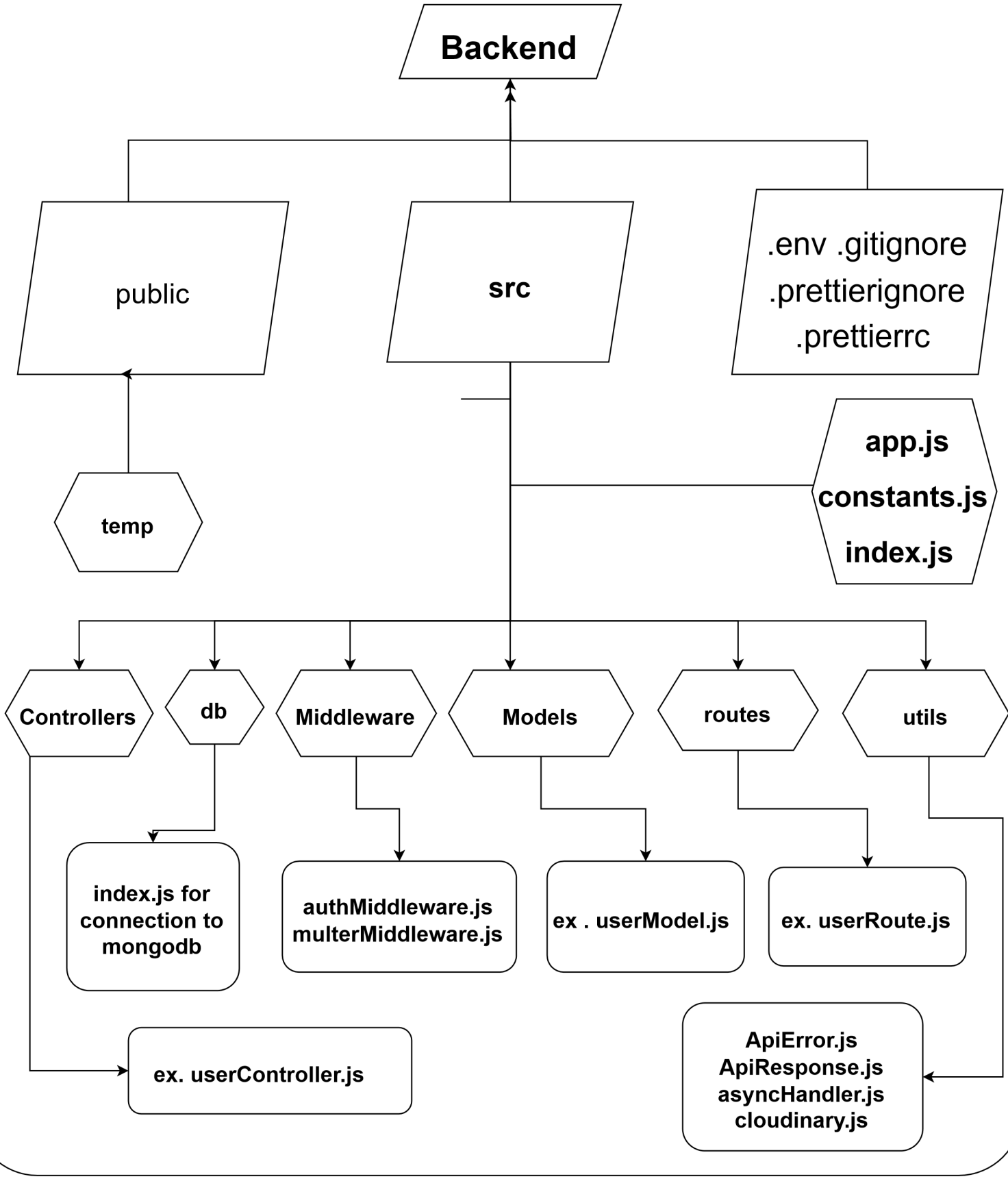
npm install bcrypt  cloudinary cookie-parser cors dotenv express
jsonwebtoken mongoose mongoose-aggregate-paginate-v2 multer

open package.json file and edit :-

1.) under scripts : -

"dev": "nodemon -r dotenv/config --experimental-json-modules src/index.js"

# Make files and folders : -



## .gitignore

**put the file name that you want to ignore while pushing the code into github ex. .env**

prettierrc

```
{
        "singleQuote":false,
        "bracketSpacing":true,
        "tabWidth":2,
        "trailingComma":"es5",
        "semi": true
}
```

## prettierignore

```
/.vscode
/node_modules
./dist
```

## .index.js

```
import dotenv from 'dotenv' 6.8k (gzipped: 3k)
import connectDB from "./db/index.js"
import {app} from "./app.js"
dotenv.config({
        path : './.env'
})

connectDB()
.then(() =>{
        app.listen( 8000 || process.env.PORT , () =>{
                console.log(`server is running at port :
                        ${process.env.PORT}`);

        })
})
.catch((err) => {
        console.log("server Problem.....",err);
})
```

## . / db / index.js

```
import mongoose from "mongoose"  870k (gzipped: 232.8k)
import {DB_NAME} from "../constants.js"

const connectDB = async() =>{
        try {
                await mongoose
                .connect(`${process.env.MONGO_URI}/${DB_NAME}`)
                console.log("server is running......");

        } catch (error) {
                console.log("Mongoose Connection error : ",error);
                process.exit(1)
        }
}
export default connectDB;
```

## . / app.js

```
import express from 'express'
import cors from 'cors'  5k (gzipped: 2.1k)
import cookieParser from 'cookie-parser'  4.6k (gzipped: 1.8
const app = express()
app.use(cookieParser())


app.use(cors({
        origin : process.env.ORIGIN,
        credentials: true
}))

app.use(express.json({limit : "16kb"}))
app.use(express.urlencoded({extended:true,limit:"16kb"}))
app.use(express.static("public"))


import userRouter from './routes/userRoutes.js'

app.use("/api/v1/users",userRouter)


export { app }
```

## . / routes / userRotes.js

```
import {Router} from 'express'
import {
        changeCurrentPassword,
        getCurrentUser,
        loginUser,
        logoutUser,
        refreshAccessToken,
        registerUser,
        updateAccountDetails,
        updateUserAvatar,
        updateUserCoverImage
} from '../controller/userController.js'

import { upload } from '../middleware/multer.js'
import { verifyJWT } from '../middleware/auth.js'


const router = Router()

router.route("/register").post(
        upload.fields([
                {
                        name : "avatar",
                        maxCount : 1
                },
                {
                        name : "coverImage",
                        maxCount : 1
                }
        ]),
        registerUser
)

router.route("/login").post(loginUser)

router.route("/logout").post(verifyJWT,logoutUser)

router.route("/refresh-token").post(refreshAccessToken)

router.route("/change-password").post(verifyJWT,changeCurrentPassword)

router.route("/current-user").get(verifyJWT,getCurrentUser)

router.route("/update-account").patch(verifyJWT,updateAccountDetails)

router.route("/update-avatar").patch(verifyJWT,upload.single("avatar"),
updateUserAvatar)

router.route("/update-cover-image").patch(verifyJWT,upload.single("coverImage"),
updateUserCoverImage)
```

. / src / middlewares / authmiddlewares.js

```
import { User } from "../model/userModel.js";
import { ApiError } from "../utils/ApiError.js";
import { asyncHandler } from "../utils/asyncHandler.js";
import jwt from "jsonwebtoken" 53.9k (gzipped: 16.1k)

export const verifyJWT = asyncHandler(async(req,res,next) => {
        try {
                const token = req.cookies?.accessToken ||
                req.header("Authorization")?.replace("Bearer ","")

                if(!token ){
                        throw new ApiError(401,"unaouthorized request ... ")
                }
                const decodedToken = jwt.verify(token,process.env.ACCESS_TOKEN_SECRET)

                const user = await User.findById(decodedToken?._id)
                .select(" -password -refreshToken")

                if(!user){
                        throw new ApiError(401,"Invalid access token...")
                }

                req.user = user;
                next()


        } catch (error) {
                throw new ApiError(401 , " Invalid Access Token ... ")
        }


})
```

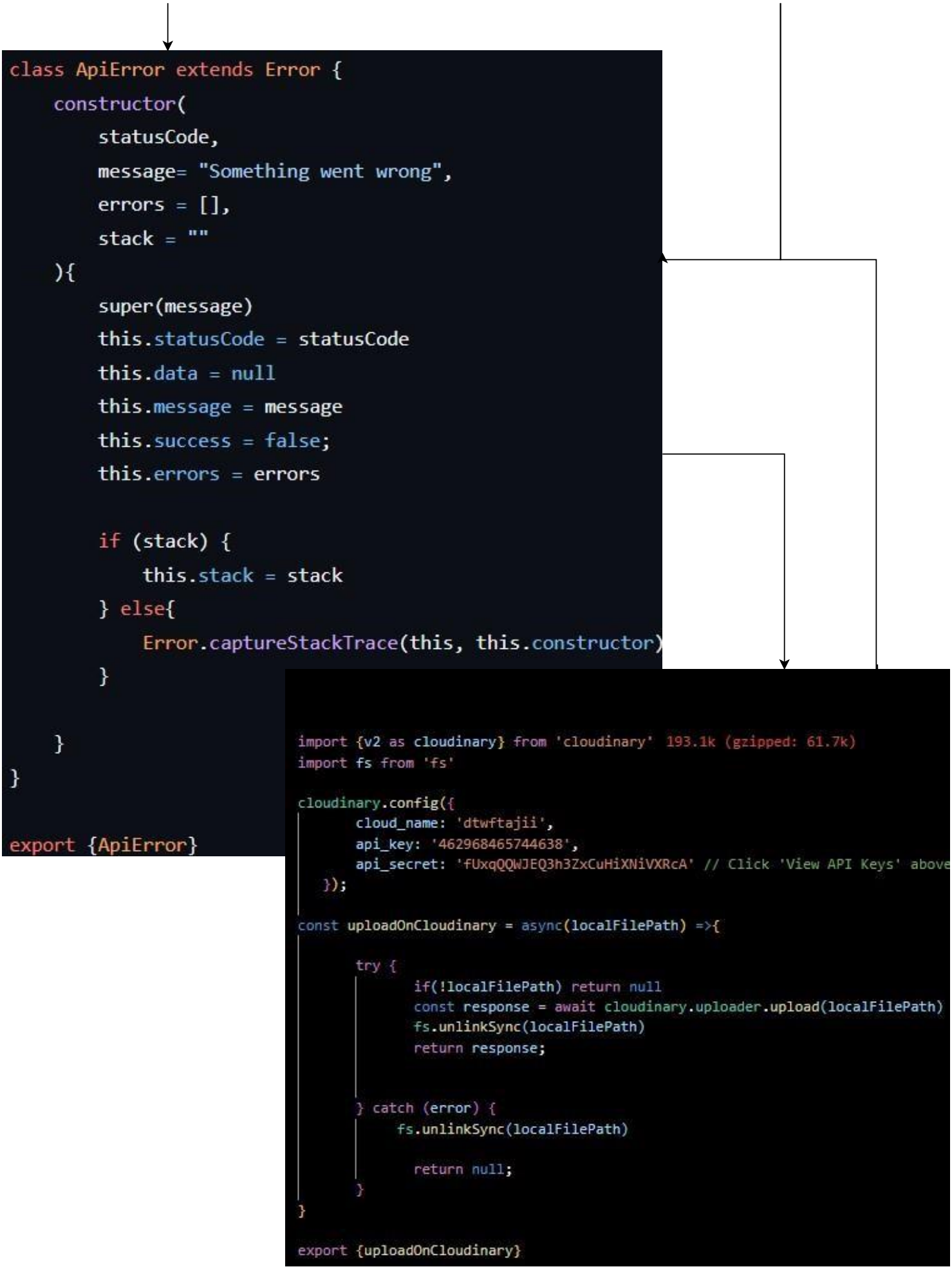
./ src / middlewares / multerMiddlewares.js

```
import multer from "multer"  241.8k (gzipped: 47.6k)


const storage = multer.diskStorage(
        {
                destination : function(req,file,cb){
                        cb(null,"./public/temp/")
                },
                filename : function(req,file,cb){
                        cb(null,file.originalname)
                }
        }
)

export const upload = multer({storage : storage})
```

.utils

```
class ApiResponse {
    constructor(statusCode, data, message = "Success"){
        this.statusCode = statusCode
        this.data = data
        this.message = message
        this.success = statusCode < 400
    }
}

export { ApiResponse }
```

```
const asyncHandler = (requestHandler) =>{
        return (req,res,next) => {
                Promise.
                resolve(requestHandler(req,res,next))
                .catch((error) => next(error))
        }
}

export {asyncHandler}
```

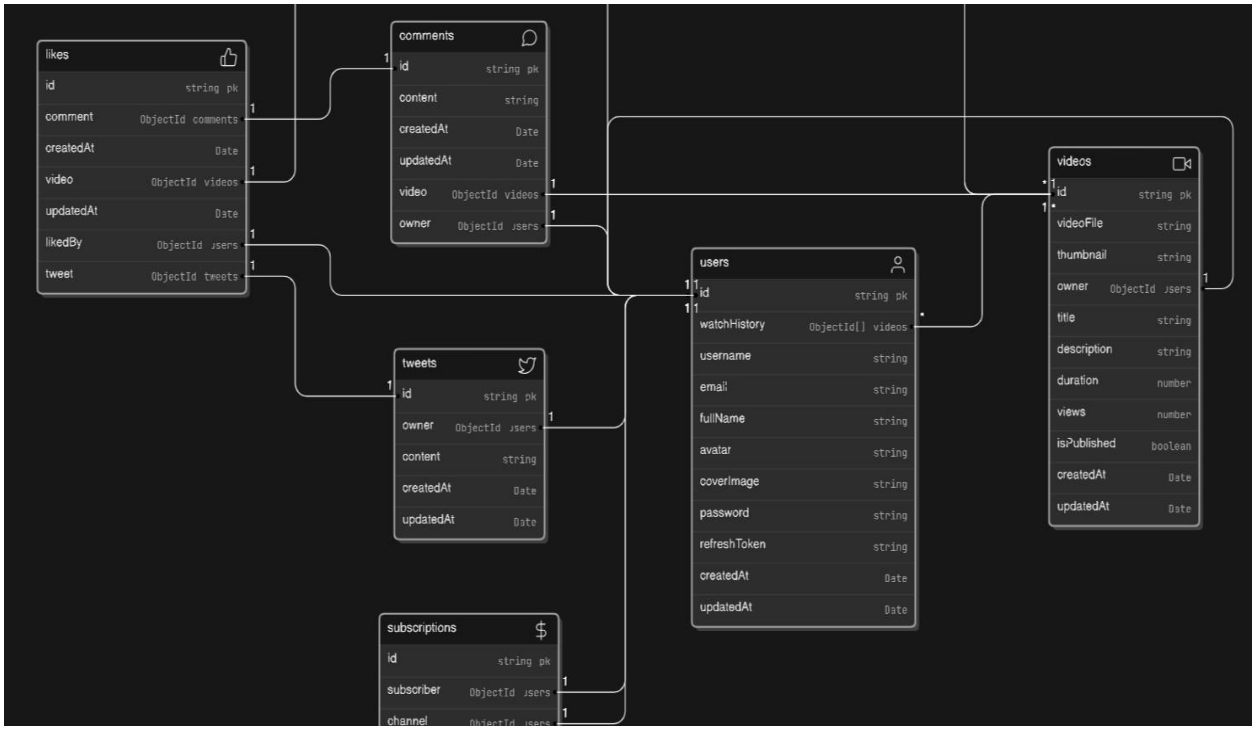


```
class ApiError extends Error {
    constructor(
        statusCode,
        message= "Something went wrong",
        errors = [],
        stack = ""
    ){
        super(message)
        this.statusCode = statusCode
        this.data = null
        this.message = message
        this.success = false;
        this.errors = errors

        if (stack) {
            this.stack = stack
        } else{
            Error.captureStackTrace(this, this.constructor)
        }

    }
}

export {ApiError}
```

```
import {v2 as cloudinary} from 'cloudinary' 193.1k (gzipped: 61.7k)
import fs from 'fs'

cloudinary.config({
      cloud_name: 'dtwftajii',
      api_key: '462968465744638',
      api_secret: 'fUxqQQWJEQ3h3ZxCuHiXNiVXRcA' // Click 'View API Keys' above
   });

const uploadOnCloudinary = async(localFilePath) =>{

      try {
            if(!localFilePath) return null
            const response = await cloudinary.uploader.upload(localFilePath)
            fs.unlinkSync(localFilePath)
            return response;


      } catch (error) {
          fs.unlinkSync(localFilePath)

            return null;
      }
}

export {uploadOnCloudinary}
```

# Data Model :-

- Data modeling is the process of creating a visual representation of a system's data and its relationships, often used in databases, software systems, or business processes.
- It involves defining how data is structured, stored, and retrieved, which helps in designing databases that are efficient, consistent, and scalable.




# Express.js :-

- Express.js is a minimal and flexible web application framework for Node.js, designed to simplify building web and mobile applications.
- It provides robust tools for handling HTTP requests and responses, routing, middleware, and more, while being lightweight enough to give you full control over the structure and behavior of your application.

## Custom Api :-

- A **custom API** (Application Programming Interface) is an interface that you design and build specifically to meet the needs of your application or system.
- **Custom APIs provide more control, flexibility, and optimization compared to using general-purpose or third-party APIs.**

## Nodemon:-

- **Nodemon is a popular tool for Node.js developers that automatically restarts your application when file changes in the directory are detected.**

## Bcrypt :-

- **A popular library used for hashing passwords in Node.js applications.**
- **It provides an efficient way to securely store user passwords by generating cryptographic hashes.**
- **Basic Usage of bcrypt :-** ○ **Hashing a password, Verifying a password:**

## JWT :-

- **WT (JSON Web Token) is a compact, URL-safe token used for securely transmitting information between parties as a JSON object. It is commonly used for authentication and authorization in web applications.**
- **JWTs allow servers to verify the identity of clients without maintaining session state, making them particularly useful in stateless environments like RESTful APIs.**

## Structure of a JWT:

**A JWT consists of three parts separated by dots (.):**

**1.) Header :-**

**Contains metadata about the token, including the type of token and the algorithm used for signing (e.g., HS256).**

**2.) Payload :-**

**Contains the claims, which are statements about an entity (typically the user), like user ID, roles, or any other relevant data.**

**3.) Signature:-**

Verifies that the sender of the JWT is who it says it is and that the message wasn't altered.

## Access Token :-

- An Access Token is a short-lived token that is used to authorize and authenticate requests to protected resources (e.g., APIs, web services).
- Access tokens typically have a short expiration time.

## Refresh Token :-

- A Refresh Token is a long-lived token that is used to obtain a new access token after the current one expires.
- Refresh tokens usually have a much longer expiration time (e.g., days, weeks, or even months).
- When an access token expires, the client can send the refresh token to the authorization server to obtain a new access token without requiring the user to log in again.

## cookie-parser:-

- cookie-parser is a middleware for Express.js that parses cookies attached to client requests.
- It simplifies working with cookies by automatically reading and parsing the Cookie header, making it easier to manage session data, track user activity, and store information client-side.

## CORS (Cross-Origin Resource Sharing):-

- A mechanism that allows restricted resources (like fonts, images, or API responses) on a web page to be requested from another domain outside the domain from which the resource originated.
- In an Express.js application, you can enable CORS by using the cors middleware. This package allows you to control how cross-origin requests are handled in your app.

## Cloudinary:-

- Cloudinary is a cloud-based image and video management service that provides comprehensive solutions for uploading, storing, manipulating, and delivering media assets.

## Multer :-

- Multer is a node.js middleware for handling multipart/form-data, which is primarily used for uploading files.
- It simplifies the process of handling file uploads, enabling you to easily receive files from HTML forms and manage them on the server.

## Router : -

- A router in the backend is responsible for defining and routing incoming HTTP requests (e.g., GET, POST, PUT, DELETE) to the appropriate controller methods.
- It acts as the traffic director, mapping URL endpoints (or routes) to specific logic that should be executed in response.
- Responsibilities of a Router :- ○ Routing Requests, URL Mapping, Middleware Integration.

## Controller:

- A controller contains the logic for handling incoming requests and interacting with the database or services.
- It's where the business logic resides and acts as the intermediary between the model (database) and the view (client-side response).
- The controller's role is to process the request, perform actions (e.g., CRUD operations), and return a response, typically in the form of JSON (in APIs).
- Responsibilities of a Controller:
  - Request Handling, Logic implementation, Response Generation, Error Handling.

### Relationship Between Router and Controller:

- **Router: Defines the endpoints and HTTP methods (GET, POST, PUT, DELETE) and maps those endpoints to the appropriate controller functions.**
- **Controller: Contains the business logic that executes when a specific route is hit.**

# MongoDB aggregation pipeline :-

- **The MongoDB Aggregation Pipeline is a powerful framework for data aggregation, allowing you to perform operations such as filtering, transforming, and analyzing your MongoDB data in a flexible and efficient manner.**
- **Stages in the pipeline performs a specific operation on the data. Some common stages include $match, $group, $sort, $project, $limit, and more.**

## Common Aggregation Pipeline Stages:

- **$match**: Filters the documents to pass only those that match the specified condition.
  - Equivalent to a find() query.
  - Example: { $match: { status: "active" } }
- **$group**: Groups documents by a specified key and performs operations like counting, averaging, summing, etc., on grouped data. ○ Example: { $group: { _id: "$status", count: { $sum: 1 } } } (groups by status and counts the number of documents per status).
- **$sort**: Sorts documents by the specified fields.
  - Example: { $sort: { age: -1 } } (sorts by age in descending order).
- **$project**: Reshapes the documents by specifying the fields to include or exclude. ○ Example: { $project: { name: 1, age: 1, _id: 0 } } (includes name and age but excludes _id).
- **$limit**: Limits the number of documents passed through the pipeline. ○ Example: { $limit: 5 } (limits to 5 documents).
- **$lookup**: Performs a join operation by querying another collection and embedding the related data into the documents.

- Example: { $lookup: { from: "orders", localField: "customerId", foreignField: "_id", as: "orders" } }

- **$unwind**: Deconstructs an array field into multiple documents, one for each element of the array. ○ Example: { $unwind: "$orders" } (creates a separate document for each element in the orders array).
- **$addFields**: Adds or modifies fields in the documents.
  - Example: { $addFields: { fullName: { $concat: ["$firstName", " ", "$lastName"] } } }
- **$count**: Outputs the total number of documents.
  - Example: { $count: "total" } (counts the number of documents)

## Sub Pipeline :-

- **In MongoDB, a sub-pipeline refers to the use of an embedded aggregation pipeline within certain stages of the main aggregation pipeline, such as `$lookup`, `$facet`, and `$graphLookup`.**

- **Each facet represents a different sub-pipeline, allowing for different analyses on the same set of input documents.**

## mongoose-aggregate-paginate-v2 :-

- **a MongoDB pagination library for Mongoose that helps you paginate aggregation results.**
- **It provides an easy way to paginate complex MongoDB queries when you are using the Mongoose ODM (Object Data Modeling) library.**
- **The library works on top of MongoDB's aggregation pipeline, making it suitable for complex queries, filters, joins ($lookup), and more.**

1. **Pagination for Aggregation Pipelines**: You can apply pagination directly on an aggregation pipeline result.
2. **Customizable Options**: Options for page size, sorting, and projection.
3. **Easy Integration with Mongoose Aggregation Framework**: Simplifies working with large data sets in combination with Mongoose.
4. **Promises and Callbacks Support**: Works with both callback-based and promise-based approaches.