# user.model.js

```
import mongoose from "mongoose";
import jwt from "jsonwebtoken";
import bcrypt from "bcrypt";

const userSchema = new mongoose.Schema(
  {
    watchHistory: [
      {
        type: mongoose.Schema.Types.ObjectId,
        ref: "Video",
      },
    ],
    username: {
      type: String,
      required: true,
      unique: true,
      lowercase: true,
      trim: true,
      index: true,
    },
    email: {
      type: String,
      required: true,
      unique: true,
      lowercase: true,
      trim: true,
    },
    fullName: {
      type: String,
      required: true,
      unique: true,
      index: true,
    },
    avatar: {
      type: String, //cloudinary url
      required: true,
    },
    coverImage: {
      type: String, //cloudinary url
    },
    password: {
      type: String,
      required: [true, "Password is required"],
    },
    refreshToken: {
      type: String,
    },
  },
  { timestamps: true }
);
userSchema.pre("save", async function (next) {
  if (!this.isModified("password")) return next();
  this.password = bcrypt.hash(this.password, 10);
  next();
```

```
});

userSchema.methods.isPasswordCorrect = async function (password) {
  return await bcrypt.compare(password, this.password);
};

userSchema.methods.generateAccessToken = function () {
  return jwt.sign(
    {
      _id: this._id,
      email: this.email,
      username: this.username,
      fullName: this.fullName,
    },
    process.env.ACCESS_TOKEN_SECRET,
    { expiresIn: process.env.ACCESS_TOKEN_EXPIRY }
  );
};
userSchema.methods.generateRefreshToken = function () {
  return jwt.sign(
    {
      _id: this._id,
    },
    process.env.REFRESH_TOKEN_SECRET,
    { expiresIn: process.env.REFRESH_TOKEN_EXPIRY }
  );
};
export const User = mongoose.model("User", userSchema);
```

# Explaination of new parts

# Mongoose Middleware for Password Hashing

The following code snippet represents a middleware function in a Node.js application using Mongoose, a MongoDB object modeling library.

```
userSchema.pre("save", async function (next) {
  if (!this.isModified("password")) return next();
  this.password = bcrypt.hash(this.password, 10);
  next();
});
```

### Middleware Setup:

- `userSchema.pre("save", ...)` : This line sets up a Mongoose middleware that will be triggered before the "save" operation on a document of the `userSchema` model.

### Async Function Parameter:

- `async function (next) { ... }` : The middleware function is asynchronous and takes a `next` callback function as a parameter. The `next` function is called when the middleware has completed its operations, allowing the application to continue with the saving process.

### Check for Password Modification:

- `if (!this.isModified("password")) return next();` : This line checks whether the "password" field of the document is modified. If it's not modified (indicating that the document is being saved for reasons other than updating the password), the function exits early by calling `next()` . This ensures that the save operation proceeds without rehashing the password.

### Password Hashing:

- `this.password = bcrypt.hash(this.password, 10);` : If the password is modified, this line hashes the password using the bcrypt hashing algorithm with a cost factor of 10. The hashed password is then assigned back to the `password` field of the document. This is a security measure to store passwords securely in the database.

### Call to `next()` :

- `next();` : Finally, the `next()` function is called, indicating that the middleware has completed its tasks and the save operation can proceed.

### Summary:

This middleware ensures that the password is hashed before saving a document, but only if the password field has been modified during the save operation. If the password remains unchanged, the middleware does not interfere with the saving process. This is a common practice for securing passwords in a database using bcrypt hashing in a Node.js/Mongoose environment.

# Mongoose Method for Password Verification

The following code extends the functionality of a Mongoose schema by adding a custom method `isPasswordCorrect` . This method is designed to verify whether a given password matches the stored hashed password in the document.

```
userSchema.methods.isPasswordCorrect = async function (password) {
  return await bcrypt.compare(password, this.password);
};
```

# Code Explanation:

### Method Definition:

- `userSchema.methods.isPasswordCorrect` : This line adds a custom method named `isPasswordCorrect` to the `userSchema` model. Methods added to the `methods` property of a Mongoose schema become available on each document created from the model.

### Async Function Parameter:

- `async function (password) { ... }` : The method is asynchronous and takes a `password` parameter. It compares the provided password with the hashed password stored in the document.

### Password Verification:

- `return await bcrypt.compare(password, this.password);` : The method uses `bcrypt.compare` to compare the provided `password` with the hashed password ( `this.password` ) stored in the document. The `compare` function returns a promise, and the `await` keyword is used to wait for the comparison to complete.

## Summary:

This custom method, `isPasswordCorrect` , allows for password verification by comparing a given password with the hashed password stored in a Mongoose document. The use of `bcrypt.compare` ensures a secure and efficient comparison, typically used in scenarios such as user authentication.

## Mongoose Methods for JWT Token Generation

The following code extends a Mongoose schema by adding two custom methods, `generateAccessToken` and `generateRefreshToken` . These methods are designed to generate JSON Web Tokens (JWT) for authentication purposes.

```
userSchema.methods.generateAccessToken = function () {
  return jwt.sign(
    {
      _id: this._id,
      email: this.email,
      username: this.username,
      fullName: this.fullName,
    },
    process.env.ACCESS_TOKEN_SECRET,
    { expiresIn: process.env.ACCESS_TOKEN_EXPIRY }
  );
};

userSchema.methods.generateRefreshToken = function () {
  return jwt.sign(
    {
      _id: this._id,
    },
    process.env.REFRESH_TOKEN_SECRET,
    { expiresIn: process.env.REFRESH_TOKEN_EXPIRY }
```

```
  );
};
```

# Code Explanation:

## Generate Access Token:

- `userSchema.methods.generateAccessToken` : This line adds a custom method named `generateAccessToken` to the `userSchema` model. This method generates an access token using the `jwt.sign` method.

## Access Token Payload:

- The payload includes properties such as `_id` , `email` , `username` , and `fullName` extracted from the current document ( `this` ).

## Signing and Expiry:

- `jwt.sign(..., process.env.ACCESS_TOKEN_SECRET, { expiresIn: process.env.ACCESS_TOKEN_EXPIRY })` : The `jwt.sign` function is used to sign the token with a secret key ( `ACCESS_TOKEN_SECRET` ) and set an expiration time defined by `ACCESS_TOKEN_EXPIRY` from environment variables.

## Generate Refresh Token:

- `userSchema.methods.generateRefreshToken` : This line adds a custom method named `generateRefreshToken` to the `userSchema` model. This method generates a refresh token using the `jwt.sign` method.

## Refresh Token Payload:

- The payload includes only the `_id` property from the current document ( `this` ).

## Signing and Expiry:

- `jwt.sign(..., process.env.REFRESH_TOKEN_SECRET, { expiresIn: process.env.REFRESH_TOKEN_EXPIRY })` : The `jwt.sign` function is used to sign the token with a secret key ( `REFRESH_TOKEN_SECRET` ) and set an expiration time defined by `REFRESH_TOKEN_EXPIRY` from environment variables.

## Summary:

These custom methods, `generateAccessToken` and `generateRefreshToken` , facilitate the generation of access and refresh tokens for user authentication. The use of JWT allows for secure and stateless token-based authentication in a Node.js/Mongoose environment.

## video.model.js

```
import mongoose from "mongoose";
import aggregatePaginate from "mongoose-aggregate-paginate-v2";

const videoSchema = new mongoose.Schema(
  {
    videoFile: {
      type: String, //cloudinary url
      required: true,
    },
    thumbnail: {
      type: String, //cloudinary url
      required: true,
    },
    owner: {
      type: mongoose.Schema.Types.ObjectId,
      ref: "User",
    },
    title: {
      type: String,
      required: true,
    },
    description: {
      type: String,
      required: true,
    },
    duration: {
      type: Number, //cloudinary or aws etc
      required: true,
    },
    views: {
      type: Number,
      default: 0,
    },
    isPublished: {
      type: Boolean,
      default: true,
    },
  },
  { timestamps: true }
);

videoSchema.plugin(aggregatePaginate);
export const Video = mongoose.model("Video", videoSchema);
```

# Mongoose Plugin Usage

The provided code applies a Mongoose plugin, `aggregatePaginate`, to a `videoSchema`. This plugin enhances the schema with pagination capabilities tailored for MongoDB aggregate queries.

```
videoSchema.plugin(aggregatePaginate);
```

# Code Explanation:

- `videoSchema` : Represents a Mongoose schema designed to handle video-related data, commonly used for defining the structure of a MongoDB collection.
- `aggregatePaginate` : A Mongoose plugin named `aggregatePaginate` is applied to the `videoSchema` . Plugins in Mongoose provide a way to extend a schema's functionality.
- **Applying the Plugin**: The `plugin` method is used to apply the `aggregatePaginate` plugin to the `videoSchema` . This incorporation grants the schema access to additional methods and features introduced by the plugin.

# Summary:

By applying the `aggregatePaginate` plugin to the `videoSchema` , the schema gains pagination capabilities specifically tailored for MongoDB aggregate queries. This proves useful when dealing with large datasets and implementing efficient paginated results, especially in scenarios involving complex queries and transformations using the MongoDB aggregation framework.