Phase 3: Virtual cinema platform using IBM Cloud Video Streaming

MOHAMMED MOOSHA

Building a virtual cinema platform using IBM Cloud Video Streaming involves several steps. Below, I'll outline the process in detail, including defining features, designing the user interface, and setting up user registration and authentication.

Step 1: Define Platform Features

1. User Registration and Authentication:

   - Allow users to create accounts and log in securely.

2. Virtual Cinema Listings:

   - Display a list of available movies, including title, description, genre, and release date.

3. Movie Details:

   - Provide detailed information about each movie, including trailers, cast, and synopsis.

4. Virtual Cinema Rooms:

   - Create virtual rooms for each movie screening.

5. Chat and Interaction:

   - Enable users to interact with each other in real-time during the movie screening.

6. Payment Integration:

   - Allow users to purchase tickets for movie screenings.

7. User Profiles:

   - Let users manage their account details, preferences, and view transaction history.

8. Search and Filter Options:

- Provide a search and filter feature to help users find movies easily.

9. Review and Rating System:

   - Allow users to leave reviews and ratings for movies.

Step 2: Design User Interface

- Create wireframes and mockups for the platform using tools like Adobe XD, Sketch, or Figma.
- Design the layout for the homepage, movie listings, movie details page, user profile, and virtual cinema room.
- Focus on an intuitive and user-friendly interface for easy navigation.

Step 3: Set Up User Registration and Authentication

1. Backend (Node.js with Express.js):

   - Set up a MongoDB database to store user information.
   - Implement user registration and login routes.
   - Use bcrypt to securely hash and store passwords.
   - Generate JWT tokens for secure authentication.

2. Frontend (React.js):

   - Create registration and login forms.
   - Use Axios or Fetch API to send requests to the backend.
   - Store JWT tokens securely (e.g., in HTTP cookies or local storage).

Step 4: Integrate IBM Cloud Video Streaming

1. Create an IBM Cloud Account:

   - Sign up for an IBM Cloud account if you haven't already.

2. Set Up IBM Cloud Video Streaming Service**:

   - Follow IBM's documentation to create a new Video Streaming service instance.

3. Generate API Keys:

   - Obtain API keys to authenticate requests to the Video Streaming service.

4. Integrate Video Streaming API:

   - Use the API keys to make requests to the Video Streaming service for creating virtual cinema rooms, streaming videos, etc.

   - Follow the IBM Cloud Video Streaming API documentation for integration details.

Step 5: Connect Backend and Frontend

- Implement the necessary API endpoints on the backend to interact with the frontend (e.g., fetching movie listings, creating virtual rooms, handling user interactions).

Step 6: Implement Additional Features

- Develop features like movie listings, movie details, virtual cinema rooms, chat functionality, payment integration, user profiles, search, and reviews.

Step 7: Testing and Deployment

- Thoroughly test the platform, including user registration, authentication, and video streaming functionality.

- Deploy the backend (Node.js) and frontend (React.js) on platforms like Heroku or IBM Cloud.

Please note that this is a high-level overview and the actual implementation will require detailed coding, configuration, and testing. Always consult the official documentation for IBM Cloud Video Streaming and any other technologies you use.

Implementation of Virtual Cinema Platform

```
npm init -y

npm install express mongoose bcrypt jsonwebtoken socket.io

mkdir virtual-cinema

cd virtual-cinema

mkdir client server

cd server

touch server.js

npm init -y

npm install express mongoose bcrypt jsonwebtoken socket.io
```

```
// server/server.js

const express = require('express');

const mongoose = require('mongoose');

const bcrypt = require('bcrypt');

const jwt = require('jsonwebtoken');

const socketIo = require('socket.io');

const http = require('http');


const app = express();

const server = http.createServer(app);

const io = socketIo(server);


app.use(express.json());
```

```
mongoose.connect('mongodb://localhost/virtual_cinema', { useNewUrlParser: true,
useUnifiedTopology: true });


// Define User Schema
const userSchema = new mongoose.Schema({
  username: String,
  password: String
});


const User = mongoose.model('User', userSchema);


// Routes
app.post('/register', async (req, res) => {
  const { username, password } = req.body;
  const hashedPassword = await bcrypt.hash(password, 10);


  const user = new User({
    username,
    password: hashedPassword
  });


  await user.save();


  res.json({ message: 'User registered successfully' });
});


app.post('/login', async (req, res) => {
  const { username, password } = req.body;
  const user = await User.findOne({ username });


  if (!user) {
    return res.status(400).json({ message: 'Invalid username or password' });
  }

  const isPasswordValid = await bcrypt.compare(password, user.password);

  if (!isPasswordValid) {
    return res.status(400).json({ message: 'Invalid username or password' });
  }

  const token = jwt.sign({ username }, 'secret_key'); // Replace with your own secret key
  res.json({ token });
});

// Socket.IO for real-time chat
io.on('connection', (socket) => {
  console.log('A user connected');

  socket.on('disconnect', () => {
    console.log('A user disconnected');
  });

  socket.on('chat message', (msg) => {
    io.emit('chat message', msg);
  });
});

server.listen(3000, () => {
  console.log('Server is running on http://localhost:3000');
});
cd ../client
mkdir public
```

```
cd public
touch index.html
cd ..
mkdir src
cd src
touch main.js
```

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Virtual Cinema</title>
</head>
<body>
  <h1>Welcome to Virtual Cinema</h1>
  <input type="text" id="username" placeholder="Username">
  <input type="password" id="password" placeholder="Password">
  <button id="register">Register</button>
  <button id="login">Login</button>

  <div id="chat">
    <ul id="messages"></ul>
    <input id="message" autocomplete="off">
    <button id="send">Send</button>
  </div>

  <script src="/main.js"></script>
</body>
</html>
```

```javascript
document.addEventListener('DOMContentLoaded', () => {
  const socket = io();
  const registerBtn = document.getElementById('register');
  const loginBtn = document.getElementById('login');
  const messages = document.getElementById('messages');
  const messageInput = document.getElementById('message');
  const sendBtn = document.getElementById('send');

  registerBtn.addEventListener('click', async () => {
    const username = document.getElementById('username').value;
    const password = document.getElementById('password').value;

    const response = await fetch('http://localhost:3000/register', {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json'
      },
      body: JSON.stringify({ username, password })
    });

    const data = await response.json();
    alert(data.message);
  });

  loginBtn.addEventListener('click', async () => {
    const username = document.getElementById('username').value;
    const password = document.getElementById('password').value;

    const response = await fetch('http://localhost:3000/login', {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json'
      },
      body: JSON.stringify({ username, password })
    });

    const data = await response.json();
    alert('Login successful!\nToken: ' + data.token);
  });

  sendBtn.addEventListener('click', () => {
    const message = messageInput.value;
    socket.emit('chat message', message);
    messageInput.value = '';
  });

  socket.on('chat message', (msg) => {
    const li = document.createElement('li');
    li.textContent = msg;
    messages.appendChild(li);
  });
});
cd ../..
node server/server.js
```