MEDIA STREAMING WITH IBM VIDEO STREAMING

We'll use **FLASK, IBM-DB2** for the **backend** and **HTML, CSS, JAVASCRIPT** for the **frontend**, and a basic in-**memory data structure** for the product catalog with **IBM CLOUD OBJECT STORAGE** using **DOCKER and KUBERNETES** in containers.

Media Streaming Features:

1. **Live Streaming:** Ability to broadcast live events in real-time to audiences.
2. **On-Demand Streaming:** Store and stream pre-recorded content for users to access at their convenience.
3. **Multi-bitrate Streaming:** Deliver content at different quality levels to accommodate various internet connection speeds.
4. **Content Management:** Organize and categorize videos into playlists or libraries for easy access.
5. **Monetization Options:** Support for subscription models, pay-per-view, or ad-based monetization.
6. **Analytics and Insights:** Track viewer engagement, audience demographics, and video performance.
7. **Custom Branding:** Customize the video player and interface with your brand's logo, colors, and themes.
8. **Security Features:** Implement content protection measures such as encryption, access control, and DRM (Digital Rights Management).
9. **Live Chat and Interaction:** Engage with viewers through live chat, comments, and interactive features.
10. **Mobile Compatibility:** Support for streaming across various devices and operating systems.

User Interface Design:

An intuitive user interface (UI) for IBM Video Streaming can include the following elements:

1. **Homepage:** Featuring categorized sections for live events, recommended content, and trending videos.
2. **Navigation Menu:** Clear and simple navigation for Live, On-Demand, Categories, Search, and User Profile sections.
3. **Video Player:** Clean, customizable, and responsive player with play/pause, volume control, and playback options.
4. **Search Bar:** Easy-to-use search functionality to find specific content.
5. **User Profile:** Allows users to manage their subscriptions, preferences, watch history, and account settings.
6. **Content Categories:** Sections dedicated to different types of content (e.g., sports, news, entertainment) for easy browsing.
7. **Call to Action Buttons:** Prominently placed buttons for signing up, subscribing, or accessing premium content.

User Registration and Authentication Mechanisms:

To ensure secure access to the platform, robust registration and authentication mechanisms should be in place:

1. **Registration:** Users sign up with an email, username, and password.
2. **Email Verification:** Send a verification link to the user's provided email for account confirmation.
3. **Two-Factor Authentication (2FA):** Offer an additional layer of security with 2FA via SMS or authenticator apps.
4. **Account Management:** Allow users to reset passwords, update account information, and manage security settings.
5. **Access Control:** Implement user roles and permissions to manage access to different features or content based on user types (e.g., viewers, administrators).
6. **Encryption:** Employ encryption protocols to secure data transmission and storage.

7. **OAuth Integration:** Enable social media or single sign-on options for simplified login.

To Setup User Registration and Authentication:

Frontend Code of HTML and CSS is used

```
<!DOCTYPE html>
<html>
<head>
  <title>User Registration</title>
  <style>
    /* Basic CSS for styling the form */
    /* Add your own styling as needed */
    .form-container {
      width: 300px;
      margin: 0 auto;
    }
    .form-group {
      margin-bottom: 15px;
    }
  </style>
</head>
<body>
  <div class="form-container">
```

```
    <h2>User Registration</h2>
    <form id="registrationForm">
      <div class="form-group">
        <label for="username">Username:</label>
        <input type="text" id="username" name="username" required>
      </div>
      <div class="form-group">
        <label for="email">Email:</label>
        <input type="email" id="email" name="email" required>
      </div>
      <div class="form-group">
        <label for="password">Password:</label>
        <input type="password" id="password" name="password"
required>
      </div>
      <div class="form-group">
        <button type="submit">Register</button>
      </div>
    </form>
  </div>

  <script>
```

```
    // JavaScript for handling form submission

document.getElementById('registrationForm').addEventListener('submit', function(event) {
        event.preventDefault();
        const username = document.getElementById('username').value;
        const email = document.getElementById('email').value;
        const password = document.getElementById('password').value;

        // Here, you can perform further actions like sending this data to the backend for registration process
        // For the example, you can log the data to the console
        console.log('Username: ', username);
        console.log('Email: ', email);
        console.log('Password: ', password);

        // Additional steps: Use Fetch API or other means to send data to the backend for user registration
        // This frontend code simulates the form submission and logging the entered data.
    });
  </script>
</body>
</html>
```

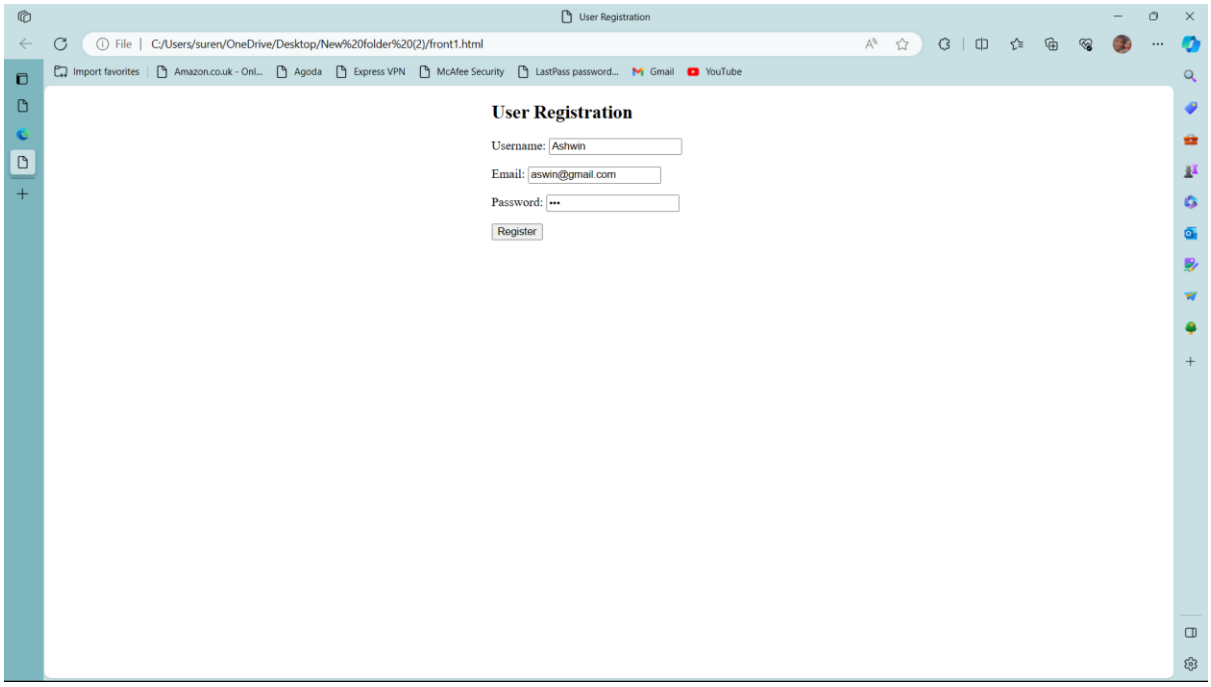
Backend of Javascript is used:

```
// Required dependencies
const express = require('express');
const bodyParser = require('body-parser');
const mongoose = require('mongoose');
const bcrypt = require('bcrypt'); // For password hashing

const app = express();

// Body parser middleware
app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());

// MongoDB connection setup
mongoose.connect('mongodb://localhost:27017/mydatabase', {
  useNewUrlParser: true,
  useUnifiedTopology: true,
});

// User model schema
const User = mongoose.model('User', {
```

```
  username: String,
  email: String,
  password: String,
});

// User registration endpoint
app.post('/register', async (req, res) => {
  const { username, email, password } = req.body;

  // Check if the user already exists
  const existingUser = await User.findOne({ email });

  if (existingUser) {
    return res.status(409).json({ message: 'User already exists' });
  }

  // Hash the password
  const hashedPassword = await bcrypt.hash(password, 10);

  // Create a new user
  const newUser = new User({
```

```
    username,
    email,
    password: hashedPassword,
  });

  await newUser.save();

  res.status(201).json({ message: 'User registered successfully' });
});

// User login endpoint
app.post('/login', async (req, res) => {
  const { email, password } = req.body;

  // Find the user by email
  const user = await User.findOne({ email });

  if (!user) {
    return res.status(404).json({ message: 'User not found' });
  }
```

```
  // Compare hashed password
  const passwordMatch = await bcrypt.compare(password, user.password);

  if (!passwordMatch) {
    return res.status(401).json({ message: 'Invalid password' });
  }

  res.status(200).json({ message: 'Login successful' });
});

// Server setup
const PORT = 3000;
app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
});vv
```



THEREFORE, LOGIN PAGE HAS BEEN DEPLOYED SUCCESSFULLY