

MEDIA STREAMING WITH IBM CLOUD VIDEO STREAMING

Phase 4: Development Part 2

Team Members :

960621104030 : Prashanna VC

960621104021 : Gold Lidiya S

960621104012 : Arockia Sreeja A

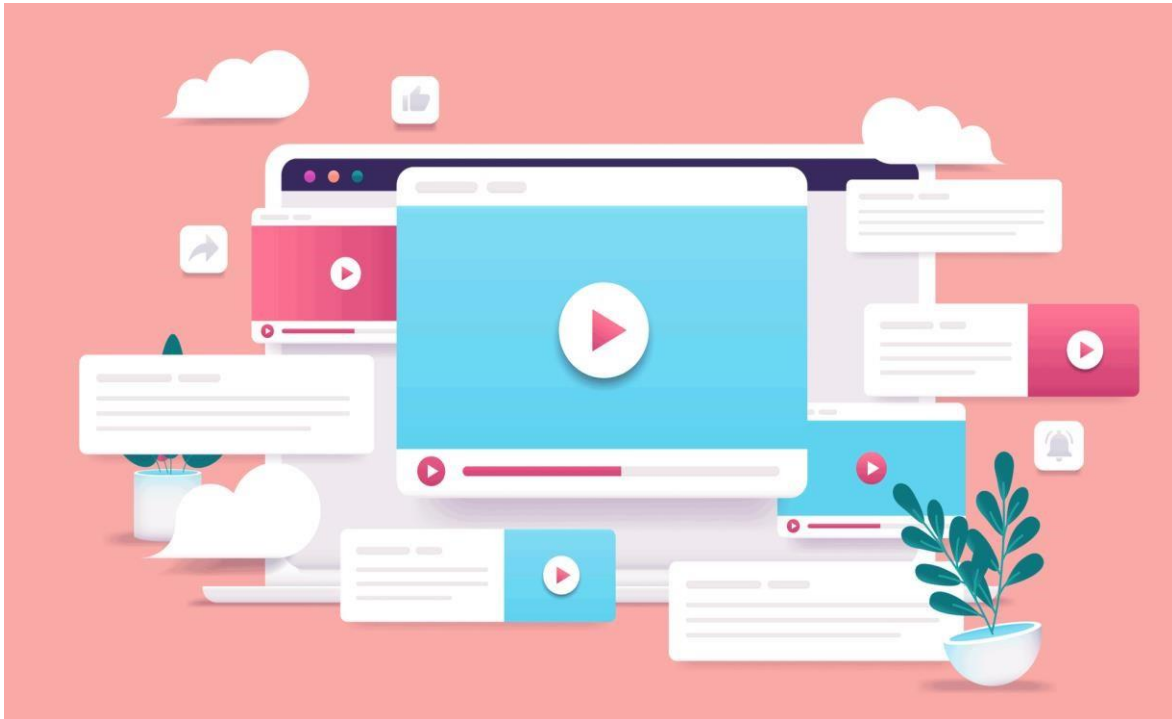
960621104007 : Annie Christina A

960621104003 : Abitha J

960621104017 : Brintha R

Introduction:

- ✓ Our project aims to create a cutting-edge video streaming platform that offers users a seamless and immersive video playback experience.
- ✓ In an age where digital video content is king, our platform aspires to be the go-to destination for content creators and viewers alike.
- ✓ We recognize the increasing demand for high-quality video streaming services, and this project is our response to that need.



Platform Features:

A platform's features refer to the functionalities and capabilities it offers to users. These may include user registration, content creation, communication tools, and more.

1. User Registration:

- Allow users to sign up with their email address or social media accounts.
- Collect essential information during registration, such as name, email, and password.
- Send a verification email to confirm the user's email address.

2. User Profile:

- Let users edit and update their profiles.
- Include profile pictures and personal information.
- Provide an option to set privacy settings for profile visibility.

3. Dashboard:

- Display personalized content and information.
- Show recent activity, notifications, and updates.

4. Search and Discovery:

- Enable users to search for content, users, or items.
- Implement filters, categories, and sorting options for ease of navigation.

5. Content Creation and Sharing:

- Allow users to create, upload, or post content (e.g., text, images, videos).
- Include options for adding tags and descriptions.
- Provide sharing options, including public, private, and restricted sharing

6. Interactions

- Support likes, comments, and shares on user-generated content.
- Enable direct messaging and communication between users.

7. Security and Privacy:

- Implement encryption for data transmission.
- Allow users to set privacy preferences for their content and profile.
- Regularly update security measures to protect user data.

8. Notifications:

- Send notifications for new messages, comments, likes, and relevant updates.
- Provide notification settings for user customization.

Intuitive User Interface:

An intuitive user interface is a design that is user-friendly and easy to understand. It ensures that users can navigate the platform effortlessly, with clear visuals, easy-to-use menus, and a logical layout.

1. Clean and Minimalistic Design:

- Use a clean and intuitive design with a consistent color scheme.
- Prioritize readability and accessibility.

2. Navigation:

- Place a navigation menu or sidebar for easy access to different sections of the platform.
- Use clear icons and labels for each section.

3. User Profile:

- Include a user profile picture and basic information.
- Show a list of recent activity and posts on the user's profile.

4. Content Feed:

- Display a personalized content feed with a mix of posts and recommendations.
- Use responsive card-based layouts for posts with images and captions.

5. Search and Discovery:

- Provide a search bar prominently at the top of the interface.
- Offer filtering and sorting options for search results.

6. Content Creation:

- Include a user-friendly content creation interface with options for adding media and tags.
- Use a WYSIWYG editor for text-based content.

7. Notifications:

- Place a notification icon or dropdown for instant access to alerts.
- Highlight unread notifications.

User Registration and Authentication Mechanisms:

1. Registration:

- Use HTTPS to encrypt data during registration.
- Verify email addresses by sending a confirmation link.
- Implement CAPTCHA or similar mechanisms to prevent bots.

2. Authentication:

- Use secure password hashing and salting techniques.
- Offer multi-factor authentication (MFA) options like SMS codes, email codes, or authenticator apps.
- Implement account lockout mechanisms after multiple failed login attempts to prevent brute force attacks.

3. Session Management:

- Use secure session tokens and set session timeouts.
- Provide a "Remember Me" option for convenience during subsequent logins.

4. Security Measures:

- Regularly update the platform to patch security vulnerabilities.
- Implement rate limiting to prevent login attempts from the same IP.

5. Privacy Controls:

- Allow users to set privacy preferences for their profiles and content.
- Clearly explain privacy settings to users.

6. Data Protection:

- Comply with data protection laws (e.g., GDPR, CCPA) and secure user data.

Database Integration

Database Selection:

- For the efficient storage of video information, user data, and other essential details, we've chosen to implement a relational database management system.
- MySQL, a widely-used open-source database, will serve as the backbone of our data storage and management.

Database Schema:

- To organize and manage data effectively, we've designed a comprehensive database schema.
- It includes the following key tables and their associated attributes:

Users Table:

- User ID
- Username
- Email
- Password (hashed and salted)
- User Roles/Permissions

Videos Table:

- Video ID
- Title
- Description
- Video URL
- Thumbnail URL
- Uploader (User ID)
- Upload Date
- Views
- Likes/Dislikes
- Comments

Video Categories Table:

- Category ID
- Category Name
- Video IDs (linked to videos in relevant categories)

User Favorites Table:

- User ID
- Favorite Video IDs

User History Table:

- User ID
- Watched Video IDs
- Timestamps

Database Functionality:

- **Data Retrieval:** The database will be queried to retrieve video details, user information, and other relevant data for seamless platform functionality.
- **Data Storage:** The database will store user profiles, video metadata, viewing history, and preferences.
- **Data Relationships:** Tables are linked to establish relationships between users, videos, categories, and user activities (such as favorites and history).

Database Security:

- To protect sensitive user information, password hashing and salting will be implemented to safeguard user credentials.
- Proper user authentication and authorization mechanisms will ensure secure data access.

Scalability and Performance:

The database architecture is designed to scale efficiently as the platform grows, ensuring consistent and fast access to data, even during periods of increased user activity.

Backup and Recovery:

Regular automated backups and data recovery procedures will be in place to prevent data loss in case of unexpected events.

On-Demand Playback**Video Player Development:**

To enable on-demand video playback, a user-friendly video player component has been developed as an integral part of our platform. This player is designed to provide a captivating viewing experience for our users. Key features of the video player include:

HTML5 Video Player: Our platform employs HTML5 video players to ensure compatibility with a wide range of devices and browsers, allowing users to watch videos seamlessly on desktop and mobile devices.

Adaptive Streaming: We've integrated adaptive streaming technology to automatically adjust the video quality based on the user's internet connection, providing a buffer-free viewing experience.

Fullscreen Mode: Users have the option to switch to fullscreen mode for an immersive viewing experience.

Playback Controls: Standard playback controls such as play, pause, volume, and progress bar are incorporated for easy navigation.

Thumbnail Previews: Hovering over the video progress bar displays thumbnail previews, making it easier for users to jump to specific parts of the video.

On-Demand Fetching:

Our platform is designed to fetch and play videos on-demand. When a user selects a video, the following steps occur to ensure seamless playback:

1. **User Request:** When a user selects a video for playback, the platform sends a request to the video streaming service to retrieve the video data.
2. **Video Fetching:** The video streaming service provides the video content, which is delivered via adaptive streaming protocols such as HLS (HTTP Live Streaming) or DASH (Dynamic Adaptive Streaming over HTTP).
3. **Local Caching:** To optimize performance, the video content may be locally cached on the user's device, reducing the need for repeated fetching of the same video.
4. **Video Playback:** The video player component on our platform then takes the fetched video data and presents it to the user, ensuring smooth and high-quality playback.

Seamless Transition Between Videos:

Our platform provides a seamless transition between videos, whether users are watching a series of episodes, a playlist, or related content. Users can easily navigate to the next video in the sequence, enhancing their viewing experience.

User Experience Enhancement:

On-demand playback is at the core of our platform, allowing users to watch content at their convenience. Whether it's catching up on missed live events or exploring a vast library of content, the on-demand playback functionality ensures users have the flexibility to enjoy videos on their terms.

User Interface (HTML/CSS/JavaScript):

1. HTML (index.html):

```
<!DOCTYPE html>
<html>

<head>

  <link rel="stylesheet" type="text/css" href="styles.css">
```

```
</head>
<body>
  <header>
    <h1>Welcome to My Streaming Platform</h1>
  </header>
  <main>
    <video id="video-player" controls></video>
  </main>
  <footer>
    <button id="start-stream">Start Streaming</button>
    <button id="stop-stream">Stop Streaming</button>
  </footer>
  <script src="script.js"></script>
</body>
</html>
```

1. CSS (styles.css):

```
body {
  font-family: Arial, sans-serif;
}

header {  text-align: center;
  background-color: #333;
  color: #fff;      padding:
  10px;
}

main {
  text-align: center;
  padding: 20px;
}
```

```
footer {  text-align:
center;  background-
color: #333;  color: #fff;
padding: 10px;
}
```

1. JavaScript (script.js):

```
const videoPlayer = document.getElementById('video-player'); const
startStreamButton = document.getElementById('start-stream'); const
stopStreamButton = document.getElementById('stop-stream');
```

```
// Event listeners for streaming buttons
startStreamButton.addEventListener('click', startStreaming);
stopStreamButton.addEventListener('click', stopStreaming);
```

```
function startStreaming() {  const
request = require('request');
```

```
// Replace these with your IBM Video Streaming API credentials
const apiKey = 'YOUR_API_KEY'; const
accountId = 'YOUR_ACCOUNT_ID';
const accessToken = 'YOUR_ACCESS_TOKEN';
```

```
// Set up the endpoint for starting a stream
const startStreamEndpoint =
`https://api.video.ibm.com/streaming/v1/accounts/${accountId}/streams`;
```

```
// Define your streaming parameters
const streamParams = {  name:
'MyStream',  broadcasting: true,
transcoding_profile: 'hd_4mbps',
};
```

```
// Create an HTTP POST request to start the stream
const options = {  url:
startStreamEndpoint,
method: 'POST',  json:
true,  body:
streamParams,
headers: {
```

```
    'Authorization': `Bearer ${accessToken}`,
    'Content-Type': 'application/json',
    'Accept': 'application/json',
  },
};
```

```
// Send the request to start the stream
request(options, (error, response, body) => {
  if (!error && response.statusCode === 201) {
    console.log('Stream started successfully');
    console.log('Stream ID:', body.id);
  } else {
    console.error('Error starting the stream:', error);
  }
});
}
```

```
function stopStreaming() {  const
  request = require('request');
```

```
// Replace these with your IBM Video Streaming API credentials and stream
ID
```

```
const apiKey = 'YOUR_API_KEY'; const
accountId = 'YOUR_ACCOUNT_ID'; const
accessToken = 'YOUR_ACCESS_TOKEN';
const streamId = 'YOUR_STREAM_ID';
```

```
// Set up the endpoint to stop the stream const
stopStreamEndpoint =
`https://api.video.ibm.com/streaming/v1/accounts/${accountId}/streams/${s
treamId}`;
```

```
// Create an HTTP DELETE request to stop the stream
const options = {  url: stopStreamEndpoint,
  method: 'DELETE',  headers: {
    'Authorization': `Bearer ${accessToken}`,
    'Accept': 'application/json',
  },
};
```

```
// Send the request to stop the stream
request(options, (error, response, body) => {
```

```
if (!error && response.statusCode === 204) {  
  console.log('Stream stopped successfully');  
} else {  
  console.error('Error stopping the stream:', error);  
}  
});  
}
```

User Registration:

```
const express = require('express'); const  
bodyParser = require('body-parser'); const  
app = express();
```

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

```
const PORT = process.env.PORT || 3000;  
app.listen(PORT, () => {  
  console.log(`Server is running on port ${PORT}`);  
});
```

```
const express = require('express'); const  
bodyParser = require('body-parser'); const  
mongoose = require('mongoose');  
const app = express();
```

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

```
const PORT = process.env.PORT || 3000;  
app.listen(PORT, () => {  
  console.log(`Server is running on port ${PORT}`);  
});
```

```
// Connect to your MongoDB database  
mongoose.connect('mongodb://localhost/your-database-name', {  
  useNewUrlParser: true, useUnifiedTopology: true });
```

```
// Create a user schema  
const userSchema = new mongoose.Schema({
```

```

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

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

// User registration endpoint app.post('/register',
async (req, res) => {  const { username,
password, email } = req.body;
  try
  {
    const user = new User({ username, password, email });
    await user.save();
    res.status(201).json({ message: 'User registered successfully', user });
  } catch (error) {
    console.error('Error registering user:', error);
    res.status(500).json({ message: 'User registration failed' });
  } });
const users = [];

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

  const express = require('express'); const
bodyParser = require('body-parser'); const
app = express();
const pgp = require('pg-promise')();

app.use(bodyParser.json());

const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  console.log(` Server is running on port ${PORT}`);
});

// Define the PostgreSQL database connection
const db = pgp({  user:
'your-username',
password: 'your-password',
  host: 'localhost',
port: 5432,

```

```

    database: 'your-database-name',
  });

// Create a user table schema const
createTable = `
  CREATE TABLE IF NOT EXISTS users (
    id SERIAL PRIMARY KEY,  username
    VARCHAR(255) NOT NULL,  password
    VARCHAR(255) NOT NULL,
    email VARCHAR(255) NOT NULL
  );
`;

db.none(createTable)
  .then(() => {
    console.log('User table created successfully');
  })
  .catch((error) => {
    console.error('Error creating user table:', error);
  });

// User registration endpoint app.post('/register',
async (req, res) => {  const { username,
password, email } = req.body;
  try
  {
    await db.none('INSERT INTO users(username, password, email)
VALUES($1, $2, $3)', [username, password, email]);
    res.status(201).json({ message: 'User registered successfully' });
  } catch (error) {
    console.error('Error registering user:', error);
    res.status(500).json({ message: 'User registration failed' });
  } });
  const user = { username, password, email };
  users.push(user);

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

```


MySQL connector library for Python

```
pip install mysql-connector-python
```

Code for connecting to a MySQL database

```
import mysql.connector

# Connect to the database db =
mysql.connector.connect(
    host="your_host",
    user="your_username",
    password="your_password",
    database="your_database"
)

# Create a cursor cursor
= db.cursor()

# Create a table (if it doesn't exist)
create_table_query = """ CREATE TABLE IF
NOT EXISTS videos ( id INT
AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR(255),
    url VARCHAR(255)
)
"""

cursor.execute(create_table_query)
```

```
# Insert data into the table insert_query = "INSERT INTO videos  
(title, url) VALUES (%s, %s)" video_data = ("Video Title",  
"video_url.mp4") cursor.execute(insert_query, video_data)
```

```
# Commit changes to the database db.commit()
```

```
# Read data from the table select_query  
= "SELECT * FROM videos"  
cursor.execute(select_query) videos =  
cursor.fetchall()
```

```
for video in videos:
```

```
    print(f"Video ID: {video[0]}, Title: {video[1]}, URL: {video[2]}")
```

```
# Update data in the table update_query = "UPDATE videos SET  
title = %s WHERE id = %s" new_title = "New Video Title"  
video_id = 1  
cursor.execute(update_query, (new_title, video_id)) db.commit()
```

```
# Delete data from the table delete_query = "DELETE  
FROM videos WHERE id = %s" video_id_to_delete = 2  
cursor.execute(delete_query, (video_id_to_delete,))  
db.commit()
```

```
# Close the cursor and the database  
connection cursor.close() db.close()
```

On-Demand Playback

```
<!DOCTYPE html>
<html>
<head>
  <title>On-Demand Video Playback</title>
</head>
<body>
  <h1>Video Title</h1>
  <video id="videoPlayer" controls>
    <source src="video_url.mp4" type="video/mp4">
```

Your browser does not support the video tag.

```
</video>
<button id="playButton">Play</button>
<button id="pauseButton">Pause</button>

<script>    const videoPlayer =
document.getElementById('videoPlayer');    const playButton =
document.getElementById('playButton');    const pauseButton
= document.getElementById('pauseButton');

    // Event listener for the play button
playButton.addEventListener('click', () => {
videoPlayer.play(); // Start playing the video
});
```

```
    // Event listener for the pause button
    pauseButton.addEventListener('click', () => {
    videoPlayer.pause(); // Pause the video

    });
```

// You can add more features like seeking, volume control, etc., as needed.

```
    // Example: Event listener for video end
    videoPlayer.addEventListener('ended', () => {

        // You can perform actions when the video ends, e.g., load the next video.
        // For instance: window.location.href = "next_video_url.html";

    });
</script>
</body>
</html>
```