#Identify a Real-World Problem or Need
--Movie viewers often lack a centralized platform to access trustworthy, diverse user reviews and ratings, making it hard to decide which movies to watch.

Define Clear Objectives and Expected Outcomes
• Build a user-friendly movie review platform where users can submit reviews and ratings.
• Enable browsing of movies with aggregated ratings and detailed user comments.
• Provide personalized movie suggestions based on user preferences.
• Expected outcome: An interactive, community-driven movie review app that helps users make informed viewing choices.

Apply Concepts Learned in the Course to Design the Solution
--Utilize programming logic to handle user input, data structures for managing reviews and ratings, and design modular components for user registration, review submission, and recommendation features.

Use Appropriate Tools,
Libraries, or Programming Techniques Based on the Subject • Backend development: Flask or Django (Python) or Java Spring Boot. • Frontend: HTML/CSS/JavaScript or React for dynamic UI. • Database: SQLite, MySQL, or MongoDB for storing user and movie data. • Algorithms: Data filtering and sorting for recommendations. • Version control: Git and GitHub.

Follow a Structured Development Process--

Problem Definition:
--In the current digital entertainment landscape, movie lovers face a challenge in making informed decisions about what films to watch.
While numerous platforms provide movie reviews and ratings, they often suffer from issues such as biased opinions, superficial reviews, or lack of diversity in viewpoints.
Aggregated ratings can be confusing or misleading without detailed user feedback.
Additionally, many users find it difficult to discover movies tailored to their personal tastes due to a fragmented and non-personalized review ecosystem. This lack of a centralized, reliable, user-driven movie review platform creates a gap between available content and audience needs. Users require an easy-to-navigate application that allows them to:
• Read authentic, diverse movie reviews from a wide community of users.
• Submit their own opinions and ratings for movies they've watched.
• Access consolidated ratings that reflect community consensus.
• Discover new movies recommended based on their preferences and past interactions.

Requirement Analysis:
--Requirement Analysis for Movie Review App To develop an effective movie review application, the requirements can be divided into functional and non-functional categories:

Functional Requirements

1. User Registration and Authentication: • Users should be able to create accounts, log in, and log out securely. • Password recovery functionality should be included.
2. Movie Database Access:
• The system should store a comprehensive list of movies with details like title, genre, release year, director, and poster images.
• Support for adding new movies (admin or authorized users).
3. Review and Rating Submission: •
Registered users can submit reviews and rate movies on a defined scale (e.g., 1 to 5 stars).
• Reviews must support text input and possibly multimedia attachments.
4. Aggregate Rating Display:
• Display average ratings and total number of reviews for each movie.
• Show individual user reviews with date and username.
5. Search and Filtering:
• Users should find movies by title, genre, release year, or rating filters.
6. Recommendation System:
• Provide personalized movie recommendations based on user ratings and preferences.
7. User Profile Management:
• Users can update profile information and view their submitted reviews.
8. Admin Panel (Optional):
• Manage users, moderate reviews, and update movie listings.
Non-Functional Requirements
. Performance:
• App should perform efficiently with quick loading of movie lists and reviews.
2. Usability:
• The interface should be intuitive and responsive for both desktop and mobile users.
3. Security:
• Protect user data with encryption and secure authentication.
• Prevent injection attacks and ensure data validation.
4. Scalability:
• Design should allow future expansion with more movies, users, and features.

Top-Down Design / Modularization:
--Level 0: Movie Review Application
• The complete system that provides movie reviews, ratings, and recommendations.
Level 1: Main Modules
1. User Management Module (Handles all user-related functionalities like registration, login, profile management, and authentication.
2. Movie Management Module (Manages movie data including storing movie details, searching, and filtering movies.
3. Review and Rating Module (Allows users to submit, edit, and view reviews and ratings of movies.
4. Recommendation Engine Module (Analyzes user ratings and preferences to provide personalized movie recommendations.
5. User Interface Module (Frontend components responsible for displaying data and interacting with users.
Level 2: Submodules 1. User Management Module
• Registration & Login
• Profile Update
• Authentication & Authorization
2. Movie Management Module
• Movie Database Access
• Movie Search & Filter
• Movie Addition/Update (Admin)
3. Review and Rating Module
• Review Submission
• Edit/Delete Reviews
• Aggregate Ratings Calculation
• View Reviews & Ratings
4. Recommendation Engine Module
• Analyze User Ratings
• Generate Recommendations
• Provide Similar Movie Suggestions
5. User Interface Module
• Home Page
• Movie Details Page
• User Profile Page
• Review Submission Form
• Recommendation Display Section
Level 3: Function Decomposition Examples
• Registration & Login:
Input validation
• Store user credentials securely
• Manage session/cookies
• Review Submission:
• Input review text and rating
• Validate input
• Store in database
• Update movie’s overall rating

Design algorithms for calculating average movie ratings, sorting movies by rating/popularity, and basic recommendation logic.
--1. Algorithm for Calculating Average Movie Ratings Input: Set of user ratings for a movie:

Output: Average rating Steps:
1. Initialize
2. For each rating , add to
3. Compute average , where = total number of ratings
4. Return Implementation:

--2. Algorithm for Sorting Movies by Rating or Popularity Input: List of movies , each with attributes rating and popularity Output: Movies sorted in descending order based on the chosen attribute (rating or popularity) Steps:
1. Choose sorting attribute (e.g., average rating or number of reviews)
2. Apply sorting algorithm (e.g., quicksort, mergesort) to reorder movies by that attribute descending
3. Return sorted movie list

--3. Basic Recommendation Logic (Content-Based Filtering) Input: User ’s rated movies and preferences, movie database Output: List of recommended movies not yet rated by Approach:
1. Identify user’s favorite genres or tags based on highly rated movies
2. Filter movies in matching these genres/tags that user hasn’t rated
3. Sort filtered movies by overall rating or popularity 4. Return top N recommendations

Develop modules independently and integrate into a cohesive web application using chosen tools. --1. Develop Modules Independently • User Management Module:
• Implement user registration, login, logout, and profile management through RESTful API endpoints (e.g., /register , /login).
• Secure authentication using JWT or sessions.
• Movie Management Module:
• Create APIs for fetching movie lists, details, and search functionalities (e.g., /movies , /movies/ , /search).
• Include admin functions for adding or updating movies if required.
• Review and Rating Module: • Develop endpoints to submit reviews and ratings (e.g., /movies//review). • Store and retrieve user reviews and calculate average ratings dynamically.
• Recommendation Engine Module: • Implement recommendation logic as a separate service or library function. • Expose recommendations through API endpoints (e.g., /recommendations/).
• Frontend Module:
• Design UI views/pages for login, registration, movie listing, movie details with reviews, and recommendations. • Use frameworks like React, Angular, or simple HTML/CSS/JS to consume backend APIs.
2. Integration into a Cohesive Web Application API Integration:
• Connect frontend UI elements with backend APIs using AJAX/fetch or frontend HTTP clients to exchange data seamlessly.
• Handle asynchronous data loading and error management gracefully.
• Database Connectivity:
• Ensure the backend modules interact consistently with the chosen database for data storage and retrieval.
• Cross-Module Communication:
• For example, after user login, maintain session or token to authorize review submissions
• Pass user data to the recommendation engine to personalize suggestions.

--Testing:• Conduct unit testing on individual modules to ensure correctness.
• Perform integration testing to verify modules work together as expected.
• Use tools like Postman for API testing and Selenium for UI testing.

--Deployment:
• Host the backend on platforms like Heroku, AWS, or local servers.
• Serve the frontend via web hosting or as part of the backend framework.
• Ensure environment variables and API endpoints are correctly configured.

ALGORITMN CODES:-

```
1. function calculateAverageRating(ratings):
if ratings is empty:
return 0
sum = 0
for rating in ratings:
sum += rating
average = sum / length(ratings)
return average
```

```
2.function sortMovies(movies, attribute):
sortedMovies = sort movies by attribute in descending order
return sortedMovies
```

```
3.function recommendMovies(user, movies, topN):
favoriteGenres = get genres from user's highest rated movies
candidateMovies = []
for movie in movies:
if movie not rated by user and movie.genre in favoriteGenres:
candidateMovies.append(movie)
sortedCandidates = sortMovies(candidateMovies, 'averageRating')
return first topN movies from sortedCandidates
```

CODES FOR THAT PROJECT:-

1ST. User Registration API

```
const express = require('express');
const bcrypt = require('bcrypt');
const router = express.Router();
const User = require('./models/User'); // Mongoose user model

router.post('/register', async (req, res) => {
try {
const { username, email, password } = req.body;
const hashedPassword = await bcrypt.hash(password, 10);
const newUser = new User({ username, email, password: hashedPassword });
await newUser.save();
res.status(201).json({ message: 'User registered successfully' });
} catch (err) {
res.status(500).json({ error: err.message });
}
});
```

2ND. Login API

```
const jwt = require('jsonwebtoken');

router.post('/login', async (req, res) => {
try {
const { email, password } = req.body;
const user = await User.findOne({ email });
if (!user) return res.status(400).json({ message: 'User not found' });
const validPassword = await bcrypt.compare(password, user.password);
if (!validPassword) return res.status(400).json({ message: 'Invalid password' });
const token = jwt.sign({ id: user.id }, 'secretkey', { expiresIn: '1h' });
res.json({ token, user: { id: user._id, username: user.username, email } });
} catch (err) {
res.status(500).json({ error: err.message });
}
});
```

3RD-3. Submit Review API

```
const Review = require('./models/Review'); // Mongoose review model
const Movie = require('./models/Movie');

router.post('/movies/:movieId/reviews', async (req, res) => {
try {
const { movieId } = req.params;
const { rating, reviewText } = req.body;
const userId = req.user.id; // from auth middleware
const review = new Review({ userId, movieId, rating, reviewText });
await review.save();

// Update movie average rating
const reviews = await Review.find({ movieId });
const avgRating = reviews.reduce((acc, r) => acc + r.rating, 0) / reviews.length;
await Movie.findByIdAndUpdate(movieId, { averageRating: avgRating });

res.status(201).json({ message: 'Review submitted successfully', review });
} catch (err) {
res.status(500).json({ error: err.message });
}
});
```

Frontend (React)

1. Movie List Component

```
import React, { useEffect, useState } from 'react';

function MovieList() {
const [movies, setMovies] = useState([]);

useEffect(() => {
fetch('/api/movies')
.then(res => res.json())
.then(data => setMovies(data));
}, []);

return (
<div>
<h2>Movies</h2>
<ul>
{movies.map(movie => (
<li key={movie._id}>
{movie.title} - Rating: {movie.averageRating.toFixed(1)}
</li>
))}
</ul>
</div>
```

```
);
}

export default MovieList;
```

2. Review Submission Form

```
import React, { useState } from 'react';

function ReviewForm({ movieId, onReviewSubmitted }) {
const [rating, setRating] = useState(0);
const [reviewText, setReviewText] = useState('');

const handleSubmit = async (e) => {
e.preventDefault();
const token = localStorage.getItem('token');
await fetch(`/api/movies/${movieId}/reviews`, {
method: 'POST',
headers: {
'Content-Type': 'application/json',
Authorization: `Bearer ${token}`
},
body: JSON.stringify({ rating, reviewText })
});
onReviewSubmitted();
};

return (
<form onSubmit={handleSubmit}>
<label>
Rating (1-5):
<input type="number" min="1" max="5" value={rating} onChange={e =>
setRating(parseInt(e.target.value))} required />
</label>
<br />
<label>
Review:
<textarea value={reviewText}onChange={e => setReviewText(e.target.value)}required />
</label>
<br />
<button type="submit">Submit Review</button>
</form>
);
}

export default ReviewForm;
```

```
#Movie.js
const mongoose = require('mongoose');
const movieSchema = new mongoose.Schema({
title: String,
genre: [String],
releaseYear: Number,
averageRating: { type: Number, default: 0 }
});
module.exports = mongoose.model('Movie', movieSchema);

#Review.js
const mongoose = require('mongoose');
const reviewSchema = new mongoose.Schema({
userId: { type: mongoose.Schema.Types.ObjectId, ref: 'User' },
movieId: { type: mongoose.Schema.Types.ObjectId, ref: 'Movie' },
rating: Number,
reviewText: String,
createdAt: { type: Date, default: Date.now }
});
module.exports = mongoose.model('Review', reviewSchema);
```

output:-

```
#Backend Console Output
Connected to MongoDB
Server running on port 3000
Postman / API Request Examples
• POST /api/auth/register

#Request JSON:
{
"username": "john",
"email": "john@example.com",
"password": "password123"
}
```

#{
"message": "User registered successfully"
}

#Request JSON:
{
"userId": "605c72a5f1d2a913a04f9b87",
"rating": 5,
"reviewText":"Awesome movie!"
}

#Response:
{
"message": "Review submitted successfully",
"review": {
"_id": "605c731bf1d2a913a04f9b89",
"userId": "605c72a5f1d2a913a04f9b87",
"movieId": "605c72f7f1d2a913a04f9b88",
"rating": 5,
"reviewText":"Awesome movie!",
"createdAt": "2025-11-22T00:00:00.000Z"
}
}

MAIN OUTPUT:-
#React UI Output
- Movie List:
- Inception - Rating: 4.8
- Interstellar - Rating: 4.7
- No ratings yet (for movies without reviews)
- Review Form:
- A simple form to add rating (1-5) and text review and submit.

This example covers end-to-end flow with backend connection, data models, APIs, and basic React UI with sample output expectation.

Backend Node.js + Express + Mongoose

1. Setup and Connect to MongoDB

```javascript
// server.js
const express = require('express');
const mongoose = require('mongoose');
const app = express();

app.use(express.json());

// Connect to MongoDB
mongoose.connect('mongodb://localhost:27017/cinecritique', {
  useNewUrlParser: true,
  useUnifiedTopology: true
});
const db = mongoose.connection;
db.on('error', console.error.bind(console, 'MongoDB connection error:'));
db.once('open', () => {
  console.log('Connected to MongoDB');
});

// Start server
app.listen(3000, () => {
  console.log('Server running on port 3000');
});
```

2. Define Schemas & Models

```javascript
// models/User.js
const mongoose = require('mongoose');

const userSchema = new mongoose.Schema({
  username: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  password: { type: String, required: true }
});

module.exports = mongoose.model('User', userSchema);
```

```javascript
// models/Movie.js
const mongoose = require('mongoose');

const movieSchema = new mongoose.Schema({
  title: { type: String, required: true },
  genre: { type: [String], required: true },
  releaseYear: Number,
  averageRating: { type: Number, default: 0 }
}, { timestamps: true });

module.exports = mongoose.model('Movie', movieSchema);
```

```javascript
// models/Review.js
const mongoose = require('mongoose');

const reviewSchema = new mongoose.Schema({
  userId: { type: mongoose.Schema.Types.ObjectId, ref: 'User', required: true },
  movieId: { type: mongoose.Schema.Types.ObjectId, ref: 'Movie', required: true },
  rating: { type: Number, required: true, min:1, max:5 },
  reviewText: String
}, { timestamps: true });

module.exports = mongoose.model('Review', reviewSchema);
```

3. User Registration API

```javascript
// routes/auth.js
const express = require('express');
const bcrypt = require('bcrypt');
const User = require('../models/User');
const router = express.Router();

router.post('/register', async (req, res) => {
  try {
    const { username, email, password } = req.body;
    const userExists = await User.findOne({ email });
    if (userExists) return res.status(409).json({ message: 'Email already registered' });

    const hashedPassword = await bcrypt.hash(password, 10);
    const user = new User({ username, email, password: hashedPassword });
    await user.save();
    res.status(201).json({ message: 'User registered successfully' });
  } catch (err) {
    res.status(500).json({ error: err.message });
  }
});

module.exports = router;
```

4. Submit Review API with Movie Average Rating Update

```javascript
// routes/reviews.js
const express = require('express');
const Review = require('../models/Review');
const Movie = require('../models/Movie');
const router = express.Router();

router.post('/movies/:movieId/reviews', async (req, res) => {
  try {
    const { movieId } = req.params;
    const { userId, rating, reviewText } = req.body; // Assuming userId is passed in body for simplicity

    const review = new Review({ userId, movieId, rating, reviewText });
    await review.save();

    const reviews = await Review.find({ movieId });
    const avgRating = reviews.reduce((acc, r) => acc + r.rating, 0) / reviews.length;
    await Movie.findByIdAndUpdate(movieId, { averageRating: avgRating });

    res.status(201).json({ message: 'Review submitted successfully', review });
  } catch (err) {
    res.status(500).json({ error: err.message });
  }
});

module.exports = router;
```

5. Sample Running Server Code with Routes

```javascript
// server.js continued
const authRoutes = require('./routes/auth');
const reviewRoutes = require('./routes/reviews');

app.use('/api/auth', authRoutes);
app.use('/api', reviewRoutes);

// Example endpoint to get all movies
const Movie = require('./models/Movie');
app.get('/api/movies', async (req, res) => {
  const movies = await Movie.find();
  res.json(movies);
});
```

Frontend React Components

1. MovieList Component (Output Movie Titles and Ratings)

```jsx
import React, { useEffect, useState } from 'react';

function MovieList() {
  const [movies, setMovies] = useState([]);

  useEffect(() => {
    fetch('/api/movies')
      .then(res => res.json())
      .then(setMovies)
      .catch(console.error);
  }, []);

  return (
    <div>
      <h2>Movie List</h2>
      <ul>
        {movies.map(movie => (
          <li key={movie._id}>
            {movie.title} - Rating: {movie.averageRating ? movie.averageRating.toFixed(1) : 'No ratings yet'}
          </li>
        ))}
      </ul>
    </div>
  );
}

export default MovieList;
```

2. ReviewForm Component

```jsx
import React, { useState } from 'react';

function ReviewForm({ movieId, userId, onReviewSubmitted }) {
  const [rating, setRating] = useState(1);
  const [reviewText, setReviewText] = useState('');

  async function handleSubmit(e) {
    e.preventDefault();

    try {
      const resp = await fetch(`/api/movies/${movieId}/reviews`, {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({ userId, rating, reviewText }),
      });
      if (resp.ok) {
        onReviewSubmitted();
        setRating(1);
        setReviewText('');
      } else {
        alert('Review submission failed');
      }
    } catch {
      alert('Error submitting review');
    }
  }

  return (
    <form onSubmit={handleSubmit}>
      <label>
        Rating (1-5):
        <input type="number" min="1" max="5" value={rating} onChange={e =>
setRating(Number(e.target.value))} required />
      </label>
      <br/>
      <label>
        Review:
        <textarea value={reviewText} onChange={e => setReviewText(e.target.value)} required />
      </label>
      <br/>
      <button type="submit">Submit Review</button>
```

Sample Outputs

Backend Console Output

```
Connected to MongoDB
Server running on port 3000
```

Postman / API Request Examples

- POST /api/auth/register

Request JSON:

```json
{
  "username": "john",
  "email": "john@example.com",
  "password": "password123"
}
```

Response:

```json
{
  "message": "User registered successfully"
}
```

- POST /api/movies/605c72f7f1d2a913a04f9b88/reviews

Request JSON:

```json
{
  "userId": "605c72a5f1d2a913a04f9b87",
  "rating": 5,
  "reviewText": "Awesome movie!"
}
```

Response:

```json
{
  "message": "Review submitted successfully",
  "review": {
    "_id": "605c731bf1d2a913a04f9b89",
    "userId": "605c72a5f1d2a913a04f9b87",
    "movieId": "605c72f7f1d2a913a04f9b88",
    "rating": 5,
    "reviewText": "Awesome movie!",
    "createdAt": "2025-11-22T00:00:00.000Z"
  }
}
```

React UI Output

- Movie List:
  - Inception - Rating: 4.8
  - Interstellar - Rating: 4.7
  - No ratings yet (for movies without reviews)
- Review Form:
  - A simple form to add rating (1-5) and text review and submit.

This example covers end-to-end flow with backend connection, data models, APIs, and basic React UI with sample output expectation.