Full Stack Development with MERN Project Documentation format

# Introduction

* + **Project Title:** iMovies
  + **Team Members:**

**1.Aniket Ranjan (Team Leader):**

* + Set up Express server and MongoDB connection
  + Create User/Admin schemas
  + Implement login, register with JWT
  + Add authentication middleware and input validation

**2.Aprajita:**

* + Create Movie, Theatre, Show, Favourite, Booking schemas
  + Develop routes for shows, bookings, favourites
  + Handle seat availability and booking logic
  + Add admin routes for movie/show management

**3.Komal Saraf:**

* + Set up React project and routes
  + Design and style pages (Home, Login, Register, etc.)
  + Create reusable components (Forms)
  + Ensure responsiveness and dark mode support

**4.Arpita Srivastava:**

* + Connect frontend with backend APIs
  + Handle data binding, state management, JWT handling
  + Implement search, filtering, and role-based routing

# Project Overview

* + **Purpose:**

The purpose of an online movie ticket booking system is to provide users with a convenient platform to browse movies, check showtimes, select seats, and purchase tickets without visiting a physical counter.

* 1. User Engagement – Offer promotions, discounts, and personalized recommendations.
  2. Operational Efficiency – Help theaters manage bookings, occupancy, and sales analytics.
  3. Convenience & Accessibility**:** Allow users to book tickets anytime, anywhere.
  4. Time-Saving**:** Reduce waiting times at theaters by enabling advance bookings
  5. Operational Efficiency – Help theaters manage bookings, occupancy, and sales analytics.
  + **Features:** An online movie ticket booking system provides a seamless and user-friendly platform for customers to book tickets, select seats, and make payments online. It also offers extensive management functionalities for theater administrators to control operations efficiently. The following are the key features and functionalities categorized into user-centric and system-centric aspects.

User-Centric Features-

1. Movie Selection: Users browse movies, view trailers, read descriptions, and filter by genre, language, or rating for easy selection.
2. Showtime Selection: Users choose preferred theaters, showtimes, and formats (2D, 3D, IMAX) with real-time availability updates.
3. Seat Reservation: Interactive seat maps allow users to select seats based on availability, pricing, and personal preference.
4. E-Ticket Generation: Digital tickets with QR codes are generated after successful booking, eliminating the need for physical copies

System-Centric Features:

1. Movie management: Administrators can add, edit, or remove movies from the platform, ensuring an up-to-date catalog of available films. They can upload posters, trailers, descriptions, cast details, and language options to provide users with complete movie information. This feature also includes managing different formats such as 2D, 3D, and IMAX.
2. Showtime Management: The system allows theater operators to schedule, modify, or cancel movie showtimes as needed. Real-time updates ensure that users see accurate availability when booking tickets.
3. Theare management: Admins select theaters, assign movies, schedule showtimes, set pricing, manage seating, update availability, and track performance analytics in real-time.

# Architecture

**Frontend:**

1. Setup React Application

This is the foundational step where you initialize your React project and prepare the environment.

• Create a React app in the client folder:

Open a terminal and run:

bash

npx create-react-app client

This will scaffold a new React application in a folder named client.

• Install required libraries:

Install libraries that will help in routing, API handling, styling, etc. Some common ones

bash

cd client

npm install react-router-dom axios bootstrap

Other useful libraries may include:

redux and @reduxjs/toolkit – for state management

formik or react-hook-form – for form handling

yup – for form validation

react-icons – for adding icons

• Create required pages and components and add routes:

Set up a basic folder structure:

css

src/

├── components/

├── pages/

├── App.js

└── index.js

Example pages:

Home.js

Login.js

Register.js

Dashboard.js

Set up routing in App.js:

jsx

import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';

import Home from './pages/Home';

import Login from './pages/Login';

import Register from './pages/Register';

import Dashboard from './pages/Dashboard';

function App() {

return (

<Router>

<Routes>

<Route path="/" element={<Home />} />

<Route path="/login" element={<Login />} />

<Route path="/register" element={<Register />} />

<Route path="/dashboard" element={<Dashboard />} />

</Routes>

</Router>

);

}

export default App;

2. Design UI Components

This stage focuses on the look and structure of your app.

• Create Components:

Reusable UI components help organize your UI efficiently. Some common examples:

Navbar.js

Footer.js

FormInput.js

Button.js

Loader.js

Keep them in the components/ folder for modularity.

• Implement layout and styling:

Use CSS, SCSS, Tailwind CSS, or libraries like Bootstrap or Material-UI.

Example: Create a global stylesheet and apply layout grids or Flexbox.

css

body {

font-family: 'Arial', sans-serif;

margin: 0;

padding: 0;

background-color: #f4f4f4;

}

.container {

max-width: 1200px;

margin: auto;

padding: 20px;

}

Create a consistent layout using wrapper components like Layout.js.

• Add navigation:

Build a Navbar component using react-router-dom's Link or NavLink.

jsx

import { Link } from 'react-router-dom';

function Navbar() {

return (

<nav>

<Link to="/">Home</Link>

<Link to="/login">Login</Link>

<Link to="/register">Register</Link>

</nav>

);

}

export default Navbar;

Include Navbar in App.js or in a Layout component.

3. Implement Frontend Logic

This step adds functionality by integrating the frontend with backend APIs.

• Integration with API endpoints:

Use axios or fetch to call backend services.

Example:

jsx

import axios from 'axios';

useEffect(() => {

axios.get('http://localhost:5000/api/users')

.then(res => setUsers(res.data))

.catch(err => console.log(err));

}, []);

For POST requests:

jsx

axios.post('http://localhost:5000/api/login', { username, password })

.then(res => console.log(res.data))

.catch(err => console.error(err));

• Implement data binding:

Use useState and useEffect for managing and binding data.

Example for form data:

jsx

const [email, setEmail] = useState('');

const [password, setPassword] = useState('');

const handleSubmit = (e) => {

e.preventDefault();

axios.post('/api/login', { email, password })

.then(response => {

// handle success

})

.catch(error => {

// handle error

});

};

Use controlled components to bind form inputs:

jsx

<input

type="text"

value={email}

onChange={(e) => setEmail(e.target.value)}

/>

**Backend:**

**Setup Express Server**

This step sets up the backend application using Express.js.

**Create index.js file in the server (backend) folder:**

Inside your project root, create a server/ directory:

bash

mkdir server && cd server

npm init -y

npm install express mongoose cors body-parser dotenv jsonwebtoken bcryptjs

touch index.js .env

**Define port number, MongoDB connection string, and JWT key in .env file:**

Create a .env file inside the server/ folder with the following:

env

PORT=5000

MONGO\_URI=mongodb://localhost:27017/myappdb

JWT\_SECRET=my\_super\_secret\_key

Make sure to load environment variables in index.js:

js

require('dotenv').config();

**Configure the server by adding CORS and body-parser:**

Here’s how your index.js should look:

js

const express = require('express');

const mongoose = require('mongoose');

const cors = require('cors');

const bodyParser = require('body-parser');

require('dotenv').config();

const app = express();

// Middleware

app.use(cors());

app.use(bodyParser.json());

// MongoDB connection

mongoose.connect(process.env.MONGO\_URI, {

useNewUrlParser: true,

useUnifiedTopology: true,

})

.then(() => console.log('MongoDB connected'))

.catch(err => console.error('MongoDB connection error:', err));

// Routes (to be added later)

app.get('/', (req, res) => {

res.send('Server is running...');

});

// Error handling middleware (added later)

const PORT = process.env.PORT || 5000;

app.listen(PORT, () => console.log(`Server running on port ${PORT}`));

**Add Authentication**

Set up user registration and login functionality.

**Create routes/auth.js and models/User.js:**

**models/User.js**

js

const mongoose = require('mongoose');

const UserSchema = new mongoose.Schema({

name: { type: String, required: true },

email: { type: String, required: true, unique: true },

password: { type: String, required: true },

});

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

**routes/auth.js**

js

const express = require('express');

const bcrypt = require('bcryptjs');

const jwt = require('jsonwebtoken');

const User = require('../models/User');

const router = express.Router();

// Register

router.post('/register', async (req, res, next) => {

try {

const { name, email, password } = req.body;

const existingUser = await User.findOne({ email });

if (existingUser) return res.status(400).json({ message: 'User already exists' });

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

const newUser = new User({ name, email, password: hashedPassword });

await newUser.save();

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

} catch (err) {

next(err);

}

});

// Login

router.post('/login', async (req, res, next) => {

try {

const { email, password } = req.body;

const user = await User.findOne({ email });

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

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

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

const token = jwt.sign({ id: user.\_id }, process.env.JWT\_SECRET, { expiresIn: '1h' });

res.json({ token, user: { id: user.\_id, name: user.name, email: user.email } });

} catch (err) {

next(err);

}

});

module.exports = router;

**Connect routes to server in index.js:**

js

const authRoutes = require('./routes/auth');

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

**Error Handling:**

Create middleware to catch and respond to errors in a unified way.

**Add error handling middleware at the end of index.js:**

js

app.use((err, req, res, next) => {

console.error(err.stack);

const statusCode = res.statusCode === 200 ? 500 : res.statusCode;

res.status(statusCode).json({

message: err.message,

stack: process.env.NODE\_ENV === 'production' ? '' :err.stack,

});

});

**Benefits:**

Ensures all uncaught errors are logged and properly responded to.

Prevents server crash from unhandled exceptions.

Helps in debugging with stack traces (optional in production).

**Database: Database Development for Movie Ticket Booking System**

**Step 1: Create MongoDB Database in the Cloud**

**• Use MongoDB Atlas:**

* Go to [MongoDB Atlas](https://www.mongodb.com/cloud/atlas).
* Create an account or log in.
* Create a **new project** and **cluster** (shared/free tier is fine for development).
* Once the cluster is created:
  + Click on **Database Access** → Add a new database user with a password.
  + Click on **Network Access** → Add your current IP or allow access from anywhere (0.0.0.0/0) for testing.
  + Go to **Clusters** → Click **Connect** → Choose "Connect your application" → Copy the connection string.

**Reference Materials:**

* Video: [Create DB in Cloud (Google Drive)](https://drive.google.com/file/d/1CQil5KzGnPvkVOPWTLP0h-Bu2bXhq7A3/view)
* Video: Connect Node to MongoDB
* Docs: [MongoDB Atlas Tutorial](https://www.mongodb.com/docs/atlas/tutorial/connect-to-your-cluster/)

**Step 2: Install Mongoose & Create Database Connection**

**• Installation:**

Run this in your server/ directory:

bash

npm install mongoose dotenv

**• Setup .env:**

env

MONGO\_URI=mongodb+srv://<username>:<password>@cluster0.mongodb.net/moviedb?retryWrites=true&w=majority

**• Create config/db.js:**

js

const mongoose = require('mongoose');

require('dotenv').config();

const connectDB = async () => {

try {

await mongoose.connect(process.env.MONGO\_URI, {

useNewUrlParser: true,

useUnifiedTopology: true,

});

console.log('MongoDB Connected');

} catch (error) {

console.error('DB Connection Error:', error.message);

process.exit(1);

}

};

module.exports = connectDB;

**• Update index.js:**

js

const connectDB = require('./config/db');

connectDB();

**Step 3: Create Data Schemas using Mongoose**

Each schema represents a core feature in the movie booking system.

**User Schema**

**File:** models/User.js

js

const mongoose = require('mongoose');

const userSchema = new mongoose.Schema({

username: { type: String, required: true, minlength: 3, maxlength: 30 },

email: { type: String, required: true, unique: true },

password: { type: String, required: true, minlength: 6 },

});

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

**Purpose**: Stores end-user account info and secures login data.

**Theater Schema**

**File:** models/Theater.js

js

const mongoose = require('mongoose');

const theaterSchema = new mongoose.Schema({

name: String,

theaterId: { type: String, unique: true },

location: String,

seatLayout: Array, // e.g., [["A1", "A2"], ["B1", "B2"]]

seatPrices: Map, // e.g., { "VIP": 300, "Regular": 150 }

});

module.exports = mongoose.model('Theater', theaterSchema);

**Purpose**: Manages theater info like seat layouts and pricing.

**Show Schema**

**File:** models/Show.js

js

const mongoose = require('mongoose');

const showSchema = new mongoose.Schema({

adminEmail: String,

movieId: { type: mongoose.Schema.Types.ObjectId, ref: 'Movie' },

theaterName: String,

showDate: String,

showTime: String,

ticketDetails: Array, // each entry can store seat number and status

});

module.exports = mongoose.model('Show', showSchema);

**Purpose**: Stores movie showtimes, theater, and booking details.

**Movie Schema**

**File:** models/Movie.js

js

const mongoose = require('mongoose');

const movieSchema = new mongoose.Schema({

name: String,

description: String,

genres: [String],

releaseDate: Date,

runtime: Number,

certification: String,

mediaFormat: String, // e.g., 2D, 3D, IMAX

showIds: [{ type: mongoose.Schema.Types.ObjectId, ref: 'Show' }],

});

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

**Purpose**: Stores metadata about movies available for booking.

**Favorite Schema**

**File:** models/Favorite.js

js

const mongoose = require('mongoose');

const favoriteSchema = new mongoose.Schema({

userEmail: String,

movieId: { type: mongoose.Schema.Types.ObjectId, ref: 'Movie' },

});

module.exports = mongoose.model('Favorite', favoriteSchema);

**Purpose**: Allows users to bookmark favorite movies.

**Booking Schema**

**File:** models/Booking.js

js

const mongoose = require('mongoose');

const bookingSchema = new mongoose.Schema({

bookingId: { type: String, unique: true },

userEmail: String,

showId: { type: mongoose.Schema.Types.ObjectId, ref: 'Show' },

tickets: Array, // includes selected seat numbers and price

});

module.exports = mongoose.model('Booking', bookingSchema);

**Purpose**: Tracks user reservations for shows.

**Admin Schema**

**File:** models/Admin.js

js

const mongoose = require('mongoose');

const adminSchema = new mongoose.Schema({

username: { type: String, required: true },

email: { type: String, required: true, unique: true },

password: { type: String, required: true },

});

module.exports = mongoose.model('Admin', adminSchema);

**Purpose**: Stores admin account details for privileged operations.

# Setup Instructions

**Prerequisites:** To develop a full-stack e-commerce app using React JS, Node.js, and MongoDB, there are several prerequisites you should consider. Here are the key prerequisites for developing such an application:

**Node.js and npm:**

Install Node.js, which includes npm (Node Package Manager), on your development machine. Node.js is required to run JavaScript on the server

side.

• Download: https://nodejs.org/en/download/

• Installation instructions: https://nodejs.org/en/download/package-manager/

**MongoDB:** Set up a MongoDB database to store hotel and booking information. Install MongoDB locally or use a cloud-based MongoDB service. • Download: https://www.mongodb.com/try/download/community

• Installation instructions: https://docs.mongodb.com/manual/installation/

**Express.js:** Express.js is a web application framework for Node.js. Install Express.js to handle server-side routing,middleware, and API development. • Installation:

Open your command prompt or terminal and run the following command: **npm install express**

**React.js**: React.js is a popular JavaScript library for building user interfaces. It enables developers to create interactive and reusable UI components, making it easier to build dynamic and responsive web applications. To install React.js, a JavaScript library for building user interfaces, follow the installation guide:

https://reactjs.org/docs/create-a-new-react-app.html

**HTML, CSS, and JavaScript:** Basic knowledge of HTML for creating the structure of your app, CSS for styling,and JavaScript for client-side interactivity is essential.

**Database Connectivity:** Use a MongoDB driver or an Object-Document Mapping (ODM) library like Mongoose to connect your Node.js server with the MongoDB database and perform CRUD (Create, Read, Update, Delete) operations.

**Front-end Framework:** Utilize Angular to build the user-facing part of the application, including product listings, booking forms, and user interfaces for the admin dashboard.

**Version Control**: Use Git for version control, enabling collaboration and tracking changes throughout the development process. Platforms like GitHub or Bitbucket can host your repository.

• Git: Download and installation instructions can be found at: https://git scm.com/downloads

**Development Environment:** Choose a code editor or Integrated Development Environment (IDE) that suits your preferences, such as Visual Studio Code, Sublime Text, or WebStorm.

• Visual Studio Code: Download from https://code.visualstudio.com/download • Sublime Text: Download from https://www.sublimetext.com/download • WebStorm: Download from https://www.jetbrains.com/webstorm/download

**To Connect the Database with Node JS go through the below provided link:**

Link: https://www.section.io/engineering-education/nodejs

mongoosejs-mongodb

To add images into Firebase Storage, you'll need to set up Firebase in your project and configure Firebase Storage. Here are the prerequisites:

1. Create a Firebase Project:

- Go to the Firebase Console: https://console.firebase.google.com/ - Click on "Add Project" and follow the prompts to create a new Firebase project.

2. Set Up Firebase in Your Web Application:

- Install Firebase SDK in your project using npm or yarn:

npm install firebase

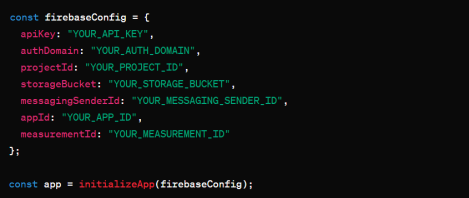
or

yarn add firebase

- Import Firebase in your application:

import { initializeApp } from "firebase/app";

- Initialize Firebase with your project configuration:



3. Enable Firebase Storage:

- In the Firebase Console, navigate to your project.

- Go to the "Storage" tab.

- Click on "Get Started" and follow the prompts to enable Firebase Storage for your project.

4. Set Up Firebase Storage Functions:

- Install Firebase Storage SDK:

import { getStorage, ref, getDownloadURL, uploadBytesResumable } from "firebase/storage";

- Get a reference to the Firebase Storage service:

const storage = getStorage(app);

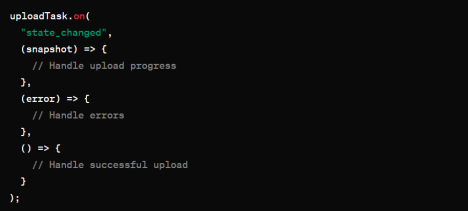
5. Upload Images to Firebase Storage:

- Use the `uploadBytesResumable` function to upload images:

const storageRef = ref(storage, 'path/to/image.jpg');

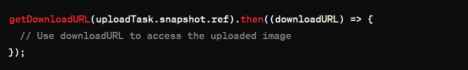
const uploadTask = uploadBytesResumable(storageRef, file);

- Monitor the upload progress using event listeners on `uploadTask`:



6. Retrieve Image URLs:

- Once the upload is complete, retrieve the download URL using `getDownloadURL`:

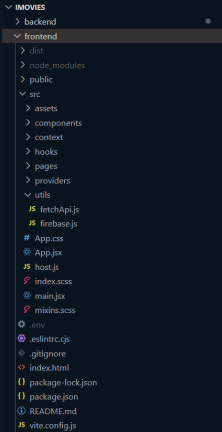
By following these steps, you should be able to upload images to Firebase Storage from your web application. Make sure to replace placeholders like `YOUR\_API\_KEY`, `YOUR\_AUTH\_DOMAIN`, etc., with your actual Firebase project credentials.

# Folder Structure:

# 

The first image is of Backend part which is showing all the files and folders that have been used in backend development

The second image is of frontend part which is showing all the files and folders that have been used in UI development



# Running the Application

**Start Frontend (React App)**

Navigate to the client/ directory:

bash

cd client

Install dependencies (only needed once or if you add new packages):

bash

npm install

Start the React development server:

bash

npm start

This will typically run your app at: <http://localhost:3000>

**Start Backend (Express Server)**

Navigate to the server/ directory:

bash

cd server

Install backend dependencies (only needed once or if you add new packages):

bash

npm install

Start the server:

bash

npm start

If you're using nodemon, you can also use:

bash

npx nodemon index.js

This usually runs at: <http://localhost:5000> (depending on your defined port)

# API Documentation

.

**Backend API Endpoint Documentation**

**Authentication Endpoints**

**1. Register User**

* **URL**: /api/auth/register
* **Method**: POST
* **Body Parameters**:

json

{

"username": "aprajita",

"email": "aprajita@example.com",

"password": "securepass123"

}

* **Success Response**:

json

{

"message": "User registered successfully",

"user": {

"\_id": "643f80a...",

"username": "aprajita",

"email": "aprajita@example.com"

}

}

**2. Login User**

* **URL**: /api/auth/login
* **Method**: POST
* **Body Parameters**:

json

{

"email": "aprajita@example.com",

"password": "securepass123"

}

**Success Response**:

json

{

"message": "Login successful",

"token": "eyJhbGciOiJIUzI1NiIs..."

}

**User Endpoints**

**3. Get User Profile**

* **URL**: /api/users/:id
* **Method**: GET
* **Headers**: Authorization: Bearer <JWT\_TOKEN>
* **Success Response**:

json

{

"\_id": "643f80a...",

"username": "aprajita",

"email": "aprajita@example.com"

}

**Movie Endpoints**

**4. Get All Movies**

* **URL**: /api/movies
* **Method**: GET
* **Success Response**:

json

[

{

"\_id": "movieId1",

"name": "Inception",

"genres": ["Sci-Fi", "Thriller"],

"releaseDate": "2024-03-01",

"runtime": 148,

...

},

...

]

**5. Get Movie by ID**

* **URL**: /api/movies/:id
* **Method**: GET
* **Success Response**:

json

{

"\_id": "movieId1",

"name": "Inception",

"description": "A mind-bending thriller...",

...

}

**Theater Endpoints**

**6. Get All Theaters**

* **URL**: /api/theaters
* **Method**: GET
* **Success Response**:

json

[

{

"\_id": "theaterId1",

"name": "PVR Velachery",

"location": "Chennai",

"seatLayout": [["A1", "A2"], ["B1", "B2"]],

"seatPrices": { "VIP": 300, "Regular": 150 }

}

]

**Show Endpoints**

**7. Get Shows for a Movie**

* **URL**: /api/shows/movie/:movieId
* **Method**: GET
* **Success Response**:

json

[

{

"\_id": "showId1",

"theaterName": "PVR Velachery",

"showDate": "2024-04-20",

"showTime": "18:00",

...

}

]

**8. Get Show by ID**

* **URL**: /api/shows/:id
* **Method**: GET

**Booking Endpoints**

**9. Create Booking**

* **URL**: /api/bookings
* **Method**: POST
* **Headers**: Authorization: Bearer <JWT\_TOKEN>
* **Body Parameters**:

json

{

"userEmail": "aprajita@example.com",

"showId": "showId1",

"tickets": [

{ "seat": "A1", "price": 300 },

{ "seat": "A2", "price": 300 }

]

}

* **Success Response**:

json

{

"message": "Booking successful",

"bookingId": "BK983823"

}

**10. Get User Bookings**

* **URL**: /api/bookings/user/:email
* **Method**: GET
* **Success Response**:

json[

{

"bookingId": "BK983823",

"showId": "showId1",

"tickets": [ ... ]

}

]

**Favorites Endpoints**

**11. Add to Favorites**

* **URL**: /api/favorites
* **Method**: POST
* **Body**:

json

{

"userEmail": "aprajita@example.com",

"movieId": "movieId1"

}

* **Success Response**:

json

{ "message": "Added to favorites" }

**12. Get User Favorites**

* **URL**: /api/favorites/:email
* **Method**: GET

**Admin Endpoints**

**13. Add New Movie**

* **URL**: /api/admin/movies
* **Method**: POST
* **Headers**: Authorization: Bearer <ADMIN\_TOKEN>
* **Body**:

json

{

"name": "Avatar 3",

"description": "Sci-fi epic",

"genres": ["Action", "Fantasy"],

...

}

**14. Add New Show**

* **URL**: /api/admin/shows
* **Method**: POST
* **Headers**: Authorization: Bearer <ADMIN\_TOKEN>

# Authentication

**Authentication & Authorization in the Project**

Authentication and authorization are crucial for securing the application, especially when managing users, bookings, and admin functionalities.

**1. Authentication**

Authentication verifies **who the user is**.

**How It Works:**

* Users log in with their **email and password**.
* On successful login, the server generates a **JWT (JSON Web Token)**.
* This token is sent back to the client and is used for authenticating future requests.

**Login Flow:**

1. Client sends login request to:

bash

POST /api/auth/login

1. Server verifies email and password using bcrypt.compare().
2. On success, a **JWT** is created using:

javascript

jwt.sign({ userId, email, role }, JWT\_SECRET, { expiresIn: '1h' });

1. JWT is returned to the client and stored in **localStorage** or **sessionStorage**.

**2. Authorization**

Authorization determines **what the user is allowed to do**.

**How It Works:**

* Protected routes require a **valid JWT token** to access.
* The token is included in the **Authorization header** of the request:

makefile

Authorization: Bearer <JWT\_TOKEN>

* A **middleware function** decodes and verifies the token:

const token = req.headers.authorization?.split(" ")[1];

jwt.verify(token, JWT\_SECRET, (err, decoded) => {

if (err) return res.status(401).json({ message: "Unauthorized" });

req.user = decoded;

next();

});

**Roles & Access Levels:**

* Regular users can:

View movies, shows, and theaters

Make bookings

Add to favorites

* Admins can:

Add/edit/delete movies, shows, and theaters

Access admin routes secured with **admin role checks**

**Token Lifecycle**

* Tokens typically expire in **1 hour**.
* After expiration, the user must **log in again** or use a **refresh token system** (optional for your app).

**Security Measures**

* **Passwords** are securely hashed using bcrypt before being saved in MongoDB.
* JWT is signed using a secret key defined in .env:

ini

JWT\_SECRET = 'yourSecretKeyHere'

Only authorized users with valid tokens can access **protected API endpoints**.

# User Interface

# 

# 

# 

# 

# Testing

**1. Testing Strategy**

The project follows a **multi-level testing approach**, including:

**a. User Acceptance Testing (UAT)**

* **Purpose**: Ensure the application meets end-user requirements and expectations.
* **Process**: Users/testers execute predefined test cases and verify if the actual results match expected outcomes.
* **Coverage**:
* Registration and login functionality
* Movie and show browsing
* Booking process and seat selection
* Favourites and user profile actions
* Admin features (adding movies, shows)

**b. Functional Testing**

* Verifies that each feature works as intended.
* Focused on core modules like authentication, booking logic, and role-based access.

**c. Manual Testing**

All test scenarios are executed manually using test cases outlined in the UAT document.

**d. Bug Tracking**

Bugs are logged using a bug table that includes:

* Bug ID
* Description
* Steps to reproduce
* Severity
* Status (Open, In Progress, Closed)

# Screenshots or Demo

* + Provide screenshots or a link to a demo to showcase the application.

# Known Issues

1. **Duplicate Bookings on Double-Click**

* Users may accidentally create duplicate bookings by double-clicking the "Book Now" button.
* Suggested fix: Disable the button after the first click until a response is received.

1. **JWT Expiration Handling**

* There is no automatic handling of expired JWT tokens.
* Users might encounter authorization errors without being redirected to the login page.
* Suggested fix: Add token expiry checks and auto-logout on the frontend.

1. **Movie Poster/Image Upload Not Implemented**

* Admins cannot upload images for movies.
* Movies may show broken or placeholder images if URLs are not correctly provided.

1. **Search and Filter Limitations**

* Movie filtering and search functionality is basic or missing.
* Users may find it hard to locate specific content quickly.

1. **Mobile Responsiveness Issues**

* Some pages, especially booking and seat layout views, are not fully mobile-friendly.
* Layouts may break or become difficult to navigate on small screens.

1. **No Role-Based Access Enforcement in UI**

* Frontend pages meant for admins (like add movie/show) are accessible via direct URL even by normal users.
* Backend still prevents unauthorized actions, but UI visibility is not restricted.

1. **State Sync Issues After Booking**

* Seat availability updates are not reflected immediately after a booking.
* A page reload is required to view changes.

1. **No Email Verification or Password Reset**

* User registration does not include email verification.
* There is no “Forgot Password” or password reset functionality.

# Future Enhancements

**Image Upload Functionality**

Enable admins to upload movie posters using services like Cloudinary or Firebase Storage.

Improve movie detail pages with high-quality visuals.

**Search and Filter Enhancements**

Add advanced filters (genre, language, release date, location).

Implement fuzzy search or auto-suggestions for better UX.

**Real-Time Seat Booking Updates**

Integrate WebSockets or polling to show real-time seat availability during booking.

**Email Verification and Password Reset**

Add email verification during registration.

Implement "Forgot Password" feature with secure token-based email reset.

**Role-Based Frontend Access Control**

Hide or disable admin pages/routes for regular users on the frontend UI.

Improve route guards and access-based rendering.

**Mobile App Integration**

Create a mobile app using React Native for iOS/Android support.

Sync functionality with the existing backend APIs.

**Payment Gateway Integration**

Add secure online payment support using Razorpay, Stripe, or PayPal.