COLLEGE CODE: 8203

COLLEGE NAME: A.V.C. College of Engineering

DEPARTMENT: Information Technology

STUDENT NM-ID: 2CF9A0E1E49C80C86124EED021004FAD

ROLL NO: 23IT114

DATE: 22/09/2025

Completed the project named as Phase 3

TECHNOLOGY PROJECTN NAME: *USER AUTHENTICATION SYSTEM*

SUBMITTED BY,

VEDIKA.D

NAME: Vedika.D

MOBILE NO: 9486786011

Project Setup – User Authentication System

1. Prerequisites

- Install Node.js (v16+ recommended)
- Install MongoDB (local or cloud: MongoDB Atlas)
- Tools: Postman / Swagger → API testing
    - VS Code → Development IDE
    - Git → Version control

2. Project Structure

```
user-authentication-system/
|— backend/              # Node.js + Express API
|   ├— config/           # DB connection, JWT secret, env vars
|   ├— controllers/      # Business logic (login, signup, profile)
|   ├— middlewares/      # Auth middleware, validation
|   ├— models/           # Mongoose schemas (User, Logs)
|   ├— routes/           # API routes (auth, admin, profile)
|   ├— utils/            # Helper functions (logger, validators)
|   ├— app.js            # Express server
|   └— package.json
|
|— frontend/             # React.js or simple HTML-CSS-JS
|   ├— public/
|   ├— src/
|   |   ├— pages/        # Login, Signup, Dashboard
|   |   ├— components/   # Reusable UI (forms, navbar)
|   |   └— App.js
|   └— package.json
|
└— README.md
```

3. Backend Setup

1. Initialize project:
2. mkdir backend && cd backend
3. npm init -y
4. npm install express mongoose bcrypt jsonwebtoken cors helmet morgan joi dotenv
5. npm install --save-dev nodemon
6. Create **.env file**:
7. PORT=5000
8. MONGO_URI=mongodb://localhost:27017/userAuth
9. JWT_SECRET=superSecretKey123
10. TOKEN_EXPIRES=1h
11. Example **User Model (models/User.js)**:
12. const mongoose = require('mongoose');
13.
14. const userSchema = new mongoose.Schema({
15. name: { type: String, required: true },
16. email: { type: String, required: true, unique: true },
17. password: { type: String, required: true },
18. role: { type: String, enum: ['user', 'admin', 'moderator'], default: 'user' },
19. }, { timestamps: true });
20.
21. module.exports = mongoose.model('User', userSchema);
22. **Routes (e.g., routes/auth.js)**:
    - POST /api/signup → Register
    - POST /api/login → Authenticate + JWT
    - GET /api/logout → Invalidate token/session
    - GET /api/profile → Fetch logged-in profile
    - GET /api/admin → Admin-only route

- PUT /api/admin/role → Update role

4. Frontend Setup

1. Initialize React app:
2. npx create-react-app frontend
3. cd frontend
4. npm install axios react-router-dom
5. Pages:
    - **Login Page** → Fields: Email, Password; Features: error handling, show/hide password.
    - **Signup Page** → Fields: Name, Email, Password, Confirm Password, Role.
    - **Dashboard** → Role-based UI:
        - User → Profile
        - Admin → User management, logs
        - Moderator → Limited tools

5. Security Configurations

- **bcrypt** → Hash + salt passwords before saving
- **JWT** → Stored in **HTTP-only cookies** or **localStorage**
- **Helmet + CORS** → Protect against XSS, CSRF
- **Validation (Joi)** → Enforce strong passwords & email format
- **Logging (Morgan/Winston)** → Track login attempts, errors

6. API Testing

- Use **Postman** to test endpoints:
    - Register
    - Login → receive JWT
    - Access /api/profile with JWT
    - Try admin route with non-admin user

7. Deployment

- **Backend**: Host on **Heroku / Render / AWS EC2**
- **Database**: Use **MongoDB Atlas**
- **Frontend**: Deploy via **Netlify / Vercel**
- Configure **CORS** for cross-origin requests.

Core Features Implementation

1. User Signup (Registration)

- Validates email & password
- Hashes password with **bcrypt**
- Saves new user in MongoDB

```
// controllers/authController.js
const User = require('../models/User');
const bcrypt = require('bcrypt');
const jwt = require('jsonwebtoken');

exports.signup = async (req, res) => {
  try {
    const { name, email, password, role } = req.body;

    // validation
    if (!email || !password) return res.status(400).json({ msg: "Email & password required" });

    const existingUser = await User.findOne({ email });
    if (existingUser) return res.status(400).json({ msg: "Email already registered" });

    // password hashing
```

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

    const user = new User({ name, email, password: hashedPassword, role: role || 'user' });
    await user.save();

    res.status(201).json({ msg: "Signup successful", userId: user._id });
  } catch (err) {
    res.status(500).json({ msg: "Server error", error: err.message });
  }
};
```

2. User Login (JWT Authentication)

- Verifies credentials
- Issues a **JWT token** (1h expiry)
- Token can be stored in **cookies/localStorage**

```
exports.login = async (req, res) => {
  try {
    const { email, password } = req.body;

    const user = await User.findOne({ email });
    if (!user) return res.status(400).json({ msg: "Invalid credentials" });

    const match = await bcrypt.compare(password, user.password);
    if (!match) return res.status(400).json({ msg: "Invalid credentials" });

    // generate JWT
    const token = jwt.sign(
      { id: user._id, role: user.role },
      process.env.JWT_SECRET,
```

```
    { expiresIn: process.env.TOKEN_EXPIRES || '1h' }
   );

   res.status(200).json({
    msg: "Login successful",
    token,
    role: user.role
   });
  } catch (err) {
   res.status(500).json({ msg: "Server error", error: err.message });
  }
};
```

3. Middleware for JWT Verification

Protects routes by verifying token & role.

```
// middlewares/auth.js
const jwt = require('jsonwebtoken');

exports.verifyToken = (req, res, next) => {
  const token = req.headers['authorization']?.split(" ")[1]; // Bearer token
  if (!token) return res.status(401).json({ msg: "Access denied" });

  try {
   const decoded = jwt.verify(token, process.env.JWT_SECRET);
   req.user = decoded; // {id, role}
   next();
  } catch (err) {
   res.status(401).json({ msg: "Invalid or expired token" });
  }
```

```
};

exports.isAdmin = (req, res, next) => {
  if (req.user.role !== "admin") return res.status(403).json({ msg: "Admin access only" });
  next();
};
```

4. Logout

- On frontend → remove JWT from localStorage/cookies
- (Optional hybrid) Invalidate session on server

```
exports.logout = async (req, res) => {
  res.status(200).json({ msg: "Logout successful, clear token on client" });
};
```

5. User Profile (Protected Route)

- Fetch logged-in user info

```
exports.getProfile = async (req, res) => {
  try {
    const user = await User.findById(req.user.id).select("-password");
    if (!user) return res.status(404).json({ msg: "User not found" });

    res.json(user);
  } catch (err) {
    res.status(500).json({ msg: "Server error" });
  }
};
```

6. Role-Based Access Control (RBAC)

Admin-only API example.

```
exports.getAllUsers = async (req, res) => {
  try {
    const users = await User.find().select("-password");
    res.json(users);
  } catch (err) {
    res.status(500).json({ msg: "Server error" });
  }
};

// route example
router.get('/admin/users', verifyToken, isAdmin, getAllUsers);
```

7. Validation (Joi Example)

```
const Joi = require('joi');
exports.validateSignup = (req, res, next) => {
  const schema = Joi.object({
    name: Joi.string().min(3).required(),
    email: Joi.string().email().required(),
    password: Joi.string().min(8).pattern(new RegExp("^(?=.*[A-Z])(?=.*[0-9])")).required(),
    role: Joi.string().valid("user", "admin", "moderator").optional()
  });
  const { error } = schema.validate(req.body);
  if (error) return res.status(400).json({ msg: error.details[0].message });
  next();
};
```

8. Logging (Winston Example)

```
const winston = require('winston');
const logger = winston.createLogger({
```

```
  level: 'info',
  format: winston.format.json(),
  transports: [
    new winston.transports.File({ filename: 'logs/error.log', level: 'error' }),
    new winston.transports.File({ filename: 'logs/combined.log' })
  ]
});
module.exports = logger;
// usage in controller
logger.info(`User login attempt: ${email}`);
```

Summary of Core Features Implemented

1. **Signup** → with bcrypt & validation
2. **Login** → JWT issued (1h expiry)
3. **Logout** → token invalidation (client-side)
4. **Profile** → protected route
5. **RBAC** → role-based route guards (admin/moderator)
6. **Validation** → Joi for secure inputs
7. **Logging** → Winston for auth events.

Data Storage (Local State / Database)

1. Database Layer – MongoDB (Backend)

MongoDB is used to store **persistent authentication data**.

User Schema (example from docs)

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

const userSchema = new mongoose.Schema({
  name: { type: String, required: true },
```

```
  email: { type: String, unique: true, required: true },
  password: { type: String, required: true },   // stored hashed with bcrypt
  role: { type: String, enum: ['user', 'admin', 'moderator'], default: 'user' },
  createdAt: { type: Date, default: Date.now },
  updatedAt: { type: Date, default: Date.now }
});
module.exports = mongoose.model('User', userSchema);
```

Stored in MongoDB:

- User details → name, email
- Hashed password
- Role (user, admin, moderator)
- Timestamps (createdAt, updatedAt)

Session / Token Storage

- **JWT tokens** → usually stored **client-side** (not in DB).
- (Optional) A **sessions collection** can be used if you want hybrid (persistent sessions).

```
// Example (optional sessions)
{
  userId: ObjectId,
  token: String,
  expiresAt: Date
}
```

Logs (Optional Table)

For security monitoring:

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

const logSchema = new mongoose.Schema({
```

```
  email: String,
  action: { type: String, enum: ['login_success', 'login_failed', 'logout'] },
  timestamp: { type: Date, default: Date.now },
  ip: String
});

module.exports = mongoose.model('AuthLog', logSchema);
```

2. Local State (Frontend – React / Browser)

On the **client-side**, authentication state is stored temporarily.

Options

1. **Local Storage**
2. localStorage.setItem("token", jwtToken);
3. localStorage.setItem("role", userRole);

Pros: Persistent across refreshes
Cons: Vulnerable to XSS

4. **Session Storage**
5. sessionStorage.setItem("token", jwtToken);

Pros: Safer, cleared when tab closes
Cons: Lost when tab closes

6. **HTTP-Only Cookies** (Recommended for production)
    - Store JWT as **httpOnly cookie** (not accessible by JS).
    - More secure against XSS.

Frontend Example (React – Auth Context)

```
import { createContext, useState } from "react";

export const AuthContext = createContext();

export const AuthProvider = ({ children }) => {
  const [user, setUser] = useState(null);  // local state for user
  const [token, setToken] = useState(localStorage.getItem("token"));
```

```
  const login = (userData, jwtToken) => {
    setUser(userData);
    setToken(jwtToken);
    localStorage.setItem("token", jwtToken);
  };

  const logout = () => {
    setUser(null);
    setToken(null);
    localStorage.removeItem("token");
  };

  return (
    <AuthContext.Provider value={{ user, token, login, logout }}>
      {children}
    </AuthContext.Provider>
  );
};
```

3. Data Flow

1. **Signup/Login** → User details stored in **MongoDB**
2. **Login Success** → JWT token issued → stored in **localStorage/session/cookies**
3. **Requests** → Client attaches JWT in Authorization: Bearer <token> header
4. **Backend** → Validates JWT → fetches user profile from **MongoDB**
5. **Logout** → Token cleared from local storage/cookies

Testing Core Features – User Authentication System

1. Tools for Testing

- **Postman** (manual API testing)
- **cURL** (quick command-line testing)
- **Mocha/Chai/Jest** (automated testing – optional)

2. Test Cases for Each Feature

Signup API (POST /api/signup)

Input:

```
{
  "name": "Alice",
  "email": "alice@example.com",
  "password": "Password123",
  "role": "user"
}
```

Expected:

- Response 201 Created
- Message: "Signup successful"
- User stored in **MongoDB** with **hashed password**

Negative Tests:

- Duplicate email → "Email already registered"
- Weak password → "Password must contain..."

Login API (POST /api/login)

Input:

```
{
  "email": "alice@example.com",
  "password": "Password123"
}
```

Expected:

- Response 200 OK
- Returns JWT token + role

{

"msg": "Login successful",

"token": "<JWT_TOKEN>",

"role": "user"

}

Negative Tests:

- Wrong email → "Invalid credentials"
- Wrong password → "Invalid credentials"

Profile API (GET /api/profile)

Setup:
Pass token in header:

Authorization: Bearer <JWT_TOKEN>

Expected:

- Response 200 OK
- Returns logged-in user info (without password)

{

"_id": "64adf...",

"name": "Alice",

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

"role": "user"

}

Negative Test:

- No token → 401 Unauthorized
- Expired token → 401 Invalid or expired token

Logout API (GET /api/logout)

Expected:

{ "msg": "Logout successful, clear token on client" }

Post-condition:

- Frontend must clear token from **localStorage/cookies**

Admin Route (RBAC Test) (GET /api/admin)

Setup:

- Login as admin → Get JWT
- Call endpoint with token

Expected:

- If role = admin → 200 OK with admin data
- If role = user → 403 Admin access only

Role Update API (PUT /api/admin/role)

Input (admin only):

{

"userId": "64adf...",

"role": "moderator"

}

Expected:

- Response 200 OK → "Role updated successfully"
- User role updated in MongoDB

3. Testing with Postman – Example Steps

1. Create **collection "UserAuth API"**
2. Add requests:
    - POST /api/signup
    - POST /api/login
    - GET /api/profile

- GET /api/logout
- GET /api/admin
- PUT /api/admin/role

3. Add **Authorization tab → Bearer Token** for protected routes.
4. Save environment variables:
   - {{baseUrl}} = http://localhost:5000
   - {{token}} = auto-updated after login

4. Automated Testing (Optional – Jest Example)

```
const request = require("supertest");
const app = require("../app");

describe("Auth System", () => {
  let token;

  it("should register a new user", async () => {
    const res = await request(app)
      .post("/api/signup")
      .send({ name: "Test", email: "test@test.com", password: "Password123" });
    expect(res.statusCode).toEqual(201);
  });

  it("should login and return token", async () => {
    const res = await request(app)
      .post("/api/login")
      .send({ email: "test@test.com", password: "Password123" });
    expect(res.statusCode).toEqual(200);
    token = res.body.token;
  });
```

```
  it("should get profile with valid token", async () => {
    const res = await request(app)
      .get("/api/profile")
      .set("Authorization", `Bearer ${token}`);
    expect(res.statusCode).toEqual(200);
  });
});
```

Version Control (GitHub) Setup:

URL- https://github.com/vedikadurai/NM-IBM-AVCCE.git.