COLLEGE CODE: 8223

COLLEGENAME: VANDAYAR ENGINEERING COLLEGE

DEPARTMENT : COMPUTER SCIENCE ENGINEERING STUDENT

NM-ID:3EB40D10185B9B4BFD14530102F3457A

ROLL NO: 822323104013

DATE:23/10/2025

Repository link: https://github.com/lathika/lathika-user-authentication-system

Deployed link: https://user-authentication-system.vercel.app

TECHNOLOGY PROJECT NAME : User Authentication System

SUBMITTED BY:

Completed the project named as phase 5

Name : AKASH.M[Team leader]
Mobile No: 63694 89528 Name :
NIJITHA.BMobile No: 93847
88850Name : KEERTHIKA.M
Mobile No: 70106 13062
Name :DIVYA.M
Mobile No:+91 93612 83989
Name : VELMURUGAN.S
Mobile No: 9578258284

USER AUTHENTICATION SYSTEMS

FINAL DEMO WALKTHROUGH:

1. **Project Overview**

The User Authentication System is designed to provide secure access control for users through modern authentication and authorization mechanisms. The system allows users to register, log in, log out, and manage their profiles securely. The demo showcases the complete flow of user interactions, from signup to protected route access, ensuring data privacy and application integrity. It uses technologies like Node.js, Express.js, MongoDB, and JWT (JSON Web Token) for backend implementation, with a simple and intuitive frontend interface.

2. **User Registration (Sign-Up)**

The demo begins with the registration page, where new users can create an account by entering their details such as username, email, and password. The system performs real-time validation to check if the email is already registered and ensures that passwords meet the security criteria. Once the details are verified, the data is securely stored in the database with password encryption using bcrypt. After successful registration, the user receives a confirmation message and is redirected to the login page.

3. **User Login**

Next, the demo proceeds to the login process, where registered users can authenticate themselves using their email and password. The backend verifies the credentials and generates a JWT token, which is stored in local storage or cookies for maintaining the session. Invalid login attempts display appropriate error messages, ensuring a smooth user experience. Upon successful login, the user gains access to the dashboard or home page, indicating that the authentication is successful.

4. **Accessing Protected Routes**

The demo highlights how only authenticated users can access protected routes such as the dashboard or profile page. If a user tries to access these routes without a valid token, the system automatically redirects them to the login page. This step demonstrates how middleware authentication ensures that unauthorized users cannot access sensitive areas of the application.

5. **User Profile Management**

Once logged in, the user can view and update their profile information. The system allows users to change their password, update their name, or modify their email securely. All updates are validated and reflected in real time. The frontend interacts with secure API endpoints to handle these requests efficiently.

6. **Logout Functionality**

The logout feature ends the user session by removing the stored token, ensuring that no unauthorized access can occur after the user signs out. The user is redirected to the login screen, confirming a clean logout process. This enhances the overall security and user control over sessions.

7 **Final Output and Demonstration**

The final part of the demo showcases the complete working flow — from user registration to logout. The smooth transitions between pages, secure login mechanism, and responsive UI confirm that the authentication system works flawlessly. The interface is clean, user-friendly, and provides a professional user experience

PROJECT REPORT :

1. **Introduction**

The User Authentication System is an essential web application component that ensures secure access control for users. It verifies user identity before granting access to protected areas of an application. The system uses modern web technologies such as Node.js, Express.js, MongoDB, and JWT (JSON Web Token) to authenticate and authorize users effectively. This project focuses on creating a simple, user-friendly, and highly secure login and registration system that can be integrated into any web-based platform.

2. **Objective**

The main objective of this project is to develop a secure authentication mechanism that:

- Allows users to register, log in, and log out safely.
- Protects sensitive user data through encryption and validation.
- Provides restricted access to authorized users only.
- Demonstrates practical implementation of JWT authentication and middleware protection.

3. System Design

The system is divided into two main components — the frontend and backend.**Frontend**:Designed using HTML, CSS, and JavaScript for a responsive and interactive user interface. It includes pages for registration, login, profile management, and dashboard display.**Backend**:Built with Node.js and Express.js, it handles all authentication logic, API routing, and database connectivity. MongoDB serves as the database to store user information securely.

4. **Workflow of the System**

User Registration:

- The user enters their name, email, and password. The password is encrypted before being stored in the database.

User Login:

- The user provides valid credentials, and a JWT token is generated for maintaining the session securely.

Protected Routes:

- Only users with valid tokens can access certain routes like dashboard or profile pages. Unauthorized users are redirected to the login page.

Profile Management:

- Authenticated users can view and update their details, change passwords, or log out securely.

5. **Features**

- Secure registration and login system.
- Encrypted password storage using bcrypt.
- JWT-based authentication and authorization.
- Profile update and logout functionality.
- Error handling for invalid login and registration attempts.
- Protected routes using middleware verification.

SCREENSHOT /API DOCUMENTATION:

1. **Registration Page**

The registration page allows new users to create an account by entering their username, email, and password. Input validation ensures that all fields are properly filled and that the email ID is not already in use. Upon successful registration, the user receives a success message and is redirected to the login page.

Key Highlights:

- Input validation for all fields.
- Password encryption before saving to the database.
- Clear success/error messages for user guida

2. **Login Page**

The login page is designed for registered users to authenticate themselves using their email and password. Once verified, the backend generates a JWT token to maintain the user's session. Invalid login attempts trigger appropriate error messages.

Key Highlights:

- Secure login process using JWT tokens.
- Validation for incorrect credentials.
- Smooth transition to the dashboard upon success.

3. **Dashboard (Protected Route)**

The dashboard is a protected route that can only be accessed after successful authentication. It displays user-specific information fetched using the JWT token. If an unauthenticated user attempts access, the system automatically redirects them to the login page.

Key Highlights:

- Route protection using middleware.
- Display of authenticated user data.
- Automatic redirection for unauthorized users.

4. **Profile Page**

The profile page allows authenticated users to view and update their personal information such as username, email, and password. The system validates updates and reflects changes in the database securely.

Key Highlights:

- Profile editing and password change functionality.
- Secure database update with validation.
- Clean and user-friendly interface.

5. **Logout Page**

The logout page or button ends the current session by removing the JWT token from local storage. This ensures that the user's session is terminated safely and no unauthorized access occurs after logout.

Key Highlights:

- Token deletion to end session securely.
- Redirection to login page post logout.
- Enhanced user control and session safety.

API Documentation

- The User Authentication System API consists of a set of endpoints designed to handle user registration, authentication, profile management, and logout functionalities securely. Each endpoint interacts with the database and ensures safe communication between the frontend and backend using JSON Web Tokens (JWT) for authentication.

- The first endpoint, /api/register, is used for user registration. When a new user submits their details such as name, email, and password, the backend validates the input and checks if the email is already registered. The password is then encrypted using bcrypt before being stored in the database to ensure security. After successful registration, the system returns a success message indicating that the user account has been created successfully.

- The second endpoint, /api/login, is responsible for user authentication. When the user enters valid credentials, the backend verifies them against the stored encrypted password. If the credentials are correct, a JWT token is generated and sent back to the client. This token is used to maintain the user's session and grant access to protected routes. Invalid credentials result in an appropriate error message, ensuring the system's reliability and security.

- The third endpoint, /api/profile, is a protected route that allows authenticated users to view their stored information. This route can only be accessed when the user provides a valid JWT token in the request header. The token is verified through middleware before granting access. If the token is missing or expired, the system denies access and returns an unauthorized error response.

- The fourth endpoint, /api/profile/update, allows authenticated users to update their profile information, such as name, email, or password. The backend validates the updated data, encrypts the new password if changed, and saves the new details in the database. This endpoint ensures that user information is kept accurate and secure at all times.

- Finally, the /api/logout endpoint handles user logout. When triggered, it removes or invalidates the stored JWT token, effectively ending the user's session. This ensures that no further access to protected routes can occur without re-authentication, maintaining complete session security.

- Each API endpoint returns clear and meaningful responses with appropriate HTTP status codes such as 200 for success, 400 for invalid inputs, 401 for unauthorized access, and 500 for internal server errors. This structured and secure API design ensures reliable user authentication, efficient data

CHALLENGE AND SOLUTIONS:

- During the development of the User Authentication System, several challenges were encountered in terms of security, data handling, and system design. Each challenge was carefully analyzed and addressed with an effective solution to ensure that the final system was secure, reliable, and user-friendly.

- One of the major challenges was ensuring data security, especially in the handling of user passwords and personal details. Storing plain-text passwords posed a significant security risk. To overcome this, the solution involved implementing bcrypt hashing, which encrypts passwords before storing them in the database. This ensures that even if the database is compromised, user credentials remain protected.

- Another challenge was maintaining secure session management. It was crucial to ensure that only authenticated users could access protected pages and resources. The solution was to integrate JWT (JSON Web Tokens) for session handling. JWT tokens provide a secure and stateless way to authenticate users, making it easy to verify their identity without storing sensitive session data on the server.

- A further challenge arose in handling unauthorized access and token validation. Users without valid tokens should not be able to view restricted content or perform certain actions. To address this, middleware authentication was implemented. The middleware checks the validity of tokens on every protected route, ensuring that unauthorized users are automatically redirected to the login page.

- User experience was also an important concern. Initially, the system lacked clear feedback during login or registration errors, which confused users. This was solved by adding real-time form validation and meaningful success/error messages for all API responses. This improvement enhanced usability and made the application more interactive and user-friendly.

- Another technical issue involved ensuring smooth communication between frontend and backend. Sometimes API requests failed due to improper configuration or missing headers. This was resolved by using proper CORS (Cross-Origin Resource Sharing) settings and structured API design, ensuring that all endpoints worked seamlessly with the frontend interface.

GITHUB README & SETUP GUIDE :

- The User Authentication System repository contains all the source code, configuration files, and documentation required to run the project successfully. It is structured to help developers easily understand the setup process, dependencies, and overall functionality of the system. The repository includes folders for the frontend, backend, and database configuration, making it simple to navigate and modify as needed.

- To begin, users should clone the repository from GitHub using the command git clone <repository_link>. Once the repository is cloned, navigate to the project directory and install the required dependencies by running the command npm install in both the frontend and backend folders. This step ensures that all necessary packages such as Express.js, Mongoose, bcrypt, and jsonwebtoken are installed properly.

- Before running the server, users must configure their environment variables. In the backend folder, create a .env file and add the required variables such as PORT, MONGO_URI, and JWT_SECRET. The MONGO_URI should contain the connection string of your MongoDB database, and the JWT_SECRET should be a secure random string used for token generation and validation. Proper configuration of these variables is crucial for secure authentication and database connectivity.

- After the environment setup is complete, the backend server can be started using the command npm start. The server will initialize and connect to MongoDB, displaying a success message in the terminal once it is ready. Similarly, the frontend can be started by navigating to the frontend directory and running the command npm start, which launches the user interface in a web browser.

- Once both servers are running, users can access the application through their local host (for example, http://localhost:3000). From here, they can register new accounts, log in, and explore protected routes to experience the full functionality of the system. The application ensures data encryption, secure token handling, and smooth navigation between pages.

- The GitHub repository also includes documentation and code comments for better understanding of the logic and flow. Developers can easily modify or extend the features, such as adding email verification, password reset functionality, or role-based access control (RBAC). The code is organized in a modular way, allowing for scalability and easier maintenance.

FINAL SUBMISSION :

```
require('dotenv').config();
const express = require('express');
const mongoose = require('mongoose');
const bcrypt = require('bcrypt');
const jwt = require('jsonwebtoken');
const cors = require('cors');

// Express app setup
const app = express();
app.use(cors());
app.use(express.json());

//  Environment variables
const PORT = process.env.PORT || 5000;
const MONGO_URI = process.env.MONGO_URI || "mongodb://localhost:27017/user_auth_db";
const JWT_SECRET = process.env.JWT_SECRET || "my_super_secret_key";
const TOKEN_EXPIRES_IN = process.env.TOKEN_EXPIRES_IN || "7d";

// MongoDB Connection
mongoose.connect(MONGO_URI, { useNewUrlParser: true, useUnifiedTopology: true })
  .then(() => console.log("MongoDB Connected Successfully"))
  .catch((err) => console.log(" MongoDB Connection Error:", err));

// Mongoose Schema
const userSchema = new mongoose.Schema({
  username: { type: String, required: true, trim: true },
  email:    { type: String, required: true, unique: true, lowercase: true, trim: true },
```

```
  password: { type: String, required: true },
  createdAt:{ type: Date, default: Date.now }
});
const User = mongoose.model("User", userSchema);

// Middleware for JWT verification
const authMiddleware = (req, res, next) => {
  const authHeader = req.headers.authorization || req.headers.Authorization;
  if (!authHeader) return res.status(401).json({ message: "No token provided" });

  const parts = authHeader.split(" ");
  if (parts.length !== 2 || parts[0] !== "Bearer")
    return res.status(401).json({ message: "Invalid token format" });

  const token = parts[1];
  try {
    const decoded = jwt.verify(token, JWT_SECRET);
    req.user = decoded; // { id, email, iat, exp }
    next();
  } catch (err) {
    res.status(401).json({ message: "Token invalid or expired" });
  }
};

//  ROUTES

// REGISTER
app.post("/api/register", async (req, res) => {
  try {
    const { username, email, password } = req.body;
    if (!username || !email || !password)
```

```
      return res.status(400).json({ message: "All fields are required" });

    const exists = await User.findOne({ email });
    if (exists) return res.status(400).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) {
    console.error("Registration error:", err);
    res.status(500).json({ message: "Registration failed" });
  }
});

// LOGIN
app.post("/api/login", async (req, res) => {
  try {
    const { email, password } = req.body;
    if (!email || !password)
      return res.status(400).json({ message: "Email and password required" });

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

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

    const payload = { id: user._id, email: user.email };
            const token = jwt.sign(payload, JWT_SECRET, { expiresIn:
```

```
TOKEN_EXPIRES_IN });

    res.json({ message: "Login successful", token });
  } catch (err) {
    console.error("Login error:", err);
    res.status(500).json({ message: "Login failed" });
  }
});

// PROFILE VIEW
app.get("/api/profile", authMiddleware, async (req, res) => {
  try {
    const user = await User.findById(req.user.id).select("-password");
    if (!user) return res.status(404).json({ message: "User not found" });
    res.json(user);
  } catch (err) {
    console.error("Profile fetch error:", err);
    res.status(500).json({ message: "Failed to fetch profile" });
  }
});

//  PROFILE UPDATE
app.put("/api/profile/update", authMiddleware, async (req, res) => {
  try {
    const { username, email, password } = req.body;
    const updates = {};

    if (username) updates.username = username;
    if (email) updates.email = email;
    if (password) updates.password = await bcrypt.hash(password, 10);
```

```
    const updatedUser = await User.findByIdAndUpdate(req.user.id, updates, { new: true }).select("-password");
    res.json({ message: "Profile updated successfully", user: updatedUser });
  } catch (err) {
    console.error("Profile update error:", err);
    res.status(500).json({ message: "Profile update failed" });
  }
});
```