COLLEGE CODE:[8223]

COLLEGE NAME:[Vandayar Engineering College]

DEPARTMENT:[Computer Science And Engineering]

STUDENT NM ID: 3EB40D10185B9B4BFD14530102F3457A

ROLL NO:822323104013

DATE:[26.09.2025]

Completed the project named as Phase-3

TECHNOLOGY PROJECT NAME:USER AUTHENDICATION SYSTEM

SUBMITTED BY, NAME:[KEERTHIKA M]

MOBILE NO:[7010613062]

Phase -3

TECHNOLOGY PROJECT NAME:USER AUTHENDICATION SYSTEM

Project Setup

The project setup stage is the foundation of the entire User Authentication System. During this phase, the initial environment required for development will be created. This involves choosing the technology stack (e.g., Node.js, Express, MongoDB, React), creating the project directory structure, and installing the required dependencies such as authentication libraries (JWT, bcrypt, passport.js, etc.).

The environment setup includes configuration of package managers like npm or yarn, and ensuring all team members have the same versions of tools installed. This step also covers establishing environment variables (.env file) to store sensitive data like database credentials or secret keys securely.

Additionally, coding standards and guidelines will be documented. Linting tools and prettier will be set up to maintain uniform code formatting across the project. By the end of this step, the team will have a stable and ready-to-use development environment.

Core Features Implementation

The core features form the backbone of the User Authentication System. This stage focuses on implementing the basic yet most crucial modules:

1. User Registration: Users can create an account by providing details such as username, email, and password. Passwords will be hashed before storing in the database.
2. Login: Registered users can log in using their email and password. Validation will ensure that only legitimate users gain access.
3. Logout: A secure logout mechanism will invalidate the session or token to prevent unauthorized reuse.
4. Password Reset: The system will allow users to reset their password through secure mechanisms like email-based OTP or reset links.
5. Authentication & Authorization: JWT (JSON Web Token) or session-based authentication will be integrated to ensure secure access. Authorization ensures that only specific roles (e.g., admin) can access restricted routes.

By implementing these features, the project will achieve its core objective of providing a secure and user-friendly authentication process.

Data Storage (Local State / Database)

For storing and managing user data, both local state (for temporary operations) and databases (for persistent storage) will be utilized. During development and testing, local state may be used to simulate user data without depending on external systems. However, for real implementation, a robust database will be required.

A database such as MongoDB (NoSQL) or MySQL/PostgreSQL (SQL) will be set up to store user details including usernames, emails, hashed passwords, and session information. Security measures such as salting and hashing passwords using bcrypt will be implemented to protect sensitive user credentials.

Additionally, the system will ensure data integrity and scalability. The database schema will be designed to accommodate additional features in the future, such as multi-factor authentication or user roles. Connection pooling and indexing will be used to improve performance when handling a large number of concurrent users.

Testing Core Features

Testing ensures the reliability and security of the User Authentication System. This phase will include multiple levels of testing:

1. Unit Testing: Each function, such as password hashing or token generation, will be tested individually.
2. Integration Testing: The interaction between modules, such as login and database verification, will be tested.
3. Functional Testing: End-to-end scenarios like registration, login, and logout will be validated to ensure features work as expected.
4. Security Testing: Specific focus will be given to vulnerabilities such as SQL injection, XSS attacks, and brute force attempts.
5. User Acceptance Testing (UAT): The system will be tested from a user's perspective to ensure it meets functional requirements.

. Unit Testing

Unit testing ensures that individual functions, methods, and classes within the authentication system behave correctly in isolation.

Objectives

- Validate that each module works as intended.
- Catch errors at the earliest stage before integration.
- Ensure correctness of algorithms like password hashing, encryption, and token validation.

Scope

- **Password hashing function**: Verify that the same password always results in the same hash (with salts applied correctly).
- **Password verification function**: Confirm that entered passwords match the stored hash.
- **Token generation function**: Ensure tokens are unique, time-bound, and securely generated.
- **Session handling functions**: Validate that session data is created, stored, and destroyed correctly.
- **Input validation functions**: Check edge cases for username and password formats.

Examples

- Test that hashing the string "password123" never produces the same result without a salt.
- Test that expired tokens are rejected by the validation function.
- Test input validation rejects empty, null, or overly long usernames.

Tools

- Jest / Mocha (JavaScript testing frameworks)
- PyTest (if using Python)
- JUnit (if using Java)

Expected Outcomes

- Individual components perform exactly as expected.
- High code coverage achieved to ensure minimal blind spots.

---

2. Integration Testing

Integration testing ensures that different modules interact correctly when combined.

Objectives

- Validate smooth communication between the authentication system and the database, APIs, or third-party services.
- Detect issues caused by module interactions that cannot be found in unit testing.

Scope

- Login module ↔ Database verification.
- Registration module ↔ User storage system.
- Password reset module ↔ Email/OTP service.
- Session/token module ↔ Frontend client.

Examples

- Test that entering a valid username/password fetches the correct record from the database.
- Verify that a new registration is properly inserted into the user database.
- Ensure password reset requests trigger an email/OTP to the registered address.
- Test token validation between frontend API requests and backend server.

Tools

- Postman (API integration testing).
- Selenium (UI + backend integration).
- SuperTest (Node.js integration tests).

Expected Outcomes

- Modules interact smoothly.
- Database queries work correctly.
- APIs return expected responses.
- No broken links between services.

---

3. Functional Testing

Functional testing validates whether the system works **end-to-end** as expected, based on functional requirements.

Objectives

- Confirm that the authentication features meet business and user requirements.
- Validate real-world user scenarios from start to finish.

Scope

- **Registration**: Users should be able to create accounts with valid details.
- **Login**: Users should be able to log in with correct credentials.
- **Logout**: Sessions should terminate properly.
- **Password reset**: Forgotten password flows should work correctly.
- **Multi-factor authentication (MFA)**: If enabled, the user should complete the second layer of security.

Examples

- Test that a user can register with a valid email, username, and password.
- Attempt login with incorrect password and verify proper error messages are displayed.
- Ensure logout destroys the session and prevents unauthorized reuse.
- Validate that MFA codes expire after a set time.

Tools

- Selenium (end-to-end testing).
- Cypress (modern E2E testing tool).
- Manual black-box testing by QA team.

Expected Outcomes

- Core authentication functions work perfectly.
- System behaves consistently across browsers and devices.
- No broken flows in user interactions.

---

4. Security Testing

Security testing is the most critical part of authentication systems since they protect sensitive data.

Objectives

- Identify vulnerabilities and close security gaps.
- Ensure resilience against common attacks.
- Validate compliance with security best practices.

Scope

- **SQL Injection**: Ensure queries are parameterized to block injections.
- **XSS (Cross-Site Scripting)**: Input fields must sanitize user input.
- **Brute force attacks**: Limit login attempts and implement CAPTCHA after multiple failures.
- **Session hijacking**: Ensure secure cookie handling (HttpOnly, Secure flags).
- **Password storage**: Ensure passwords are stored as salted hashes, not plaintext.
- **Token handling**: Verify token expiration, refresh handling, and revocation work correctly.

Examples

- Attempt to inject malicious SQL code during login.
- Enter <script>alert("hacked")</script> in username fields to test XSS prevention.
- Run brute-force scripts and check if account lockout occurs after multiple failed attempts.
- Capture session cookies and attempt reuse (should fail if secure).

Tools

- OWASP ZAP (security vulnerability scanner).
- Burp Suite (penetration testing).
- Hydra (brute force testing).
- Custom penetration test scripts.

Expected Outcomes

- Authentication system is hardened against external attacks.
- Sensitive data is safe and encrypted.
- Meets OWASP Top 10 security standards.

Testing will be automated wherever possible using tools like Jest, Mocha, or Postman. Bug tracking systems will be used to document and resolve any issues encountered.

Version Control (GitHub)

Version control plays a critical role in collaborative software development. For this project, GitHub will be used as the main version control platform.

Key activities include:
- Initializing a Git repository and pushing the project structure to GitHub.
- Establishing branching strategies such as ‘main’, ‘development’, and feature-specific branches to maintain clean workflows.
- Using pull requests (PRs) for code reviews to ensure high-quality code.
- Documenting commits with meaningful messages for traceability.
- Leveraging GitHub Actions for continuous integration (CI) to automate testing after each commit.

Version control also ensures backup and disaster recovery. In case of local failures, the project remains safe on GitHub. It will also enable multiple team members to work on different features simultaneously without overwriting each other’s code.

Description

PORAGRAM

```
<!DOCTYPE html>

<html lang=”en”>

<head>

 <meta charset=”UTF-8”>

 <meta name=”viewport” content=”width=device-width, initial-scale=1.0”>

 <title>Advanced To-Do List</title>

 <style>

  Body {

   Font-family: Arial, sans-serif;

   Background: #f3f4f6;

   Display: flex;
```

```
  Justify-content: center;
  Align-items: center;
  Height: 100vh;
}
.todo-container {
  Background: white;
  Padding: 20px;
  Border-radius: 12px;
  Box-shadow: 0px 4px 15px rgba(0,0,0,0.2);
  Width: 380px;
}
H2 {
  Text-align: center;
  Margin-bottom: 10px;
}
.input-row {
  Display: flex;
  Gap: 5px;
  Margin-bottom: 15px;
}
Input {
  Flex: 1;
  Padding: 8px;
  Border: 1px solid #ccc;
  Border-radius: 6px;
}
```

```
Button {
  Padding: 8px;
  Border: none;
  Background: #4CAF50;
  Color: white;
  Border-radius: 6px;
  Cursor: pointer;
}
Ul {
  List-style: none;
  Padding: 0;
  Margin: 0;
}
Li {
  Background: #e5e7eb;
  Margin: 6px 0;
  Padding: 8px;
  Border-radius: 6px;
  Display: flex;
  Justify-content: space-between;
  Align-items: center;
  Cursor: pointer;
}
li.completed {
  text-decoration: line-through;
  color: gray;
```

```
    }
    Li input.edit-box {
      Flex: 1;
      Padding: 6px;
      Border: 1px solid #ccc;
      Border-radius: 6px;
    }
    Li span {
      Cursor: pointer;
      Color: red;
      Font-weight: bold;
      Margin-left: 10px;
    }
    .clear-btn {
      Margin-top: 10px;
      Width: 100%;
      Background: #f59e0b;
    }
  </style>
</head>
<body>
  <div class=”todo-container”>
    <h2>📝 My To-Do List</h2>
    <div class=”input-row”>
      <input type=”text” id=”taskInput” placeholder=”Enter task…”>
      <button onclick=”addTask()”>Add</button>
```

```
  </div>
  <ul id=”taskList”></ul>
  <button class=”clear-btn” onclick=”clearCompleted()”>Clear Completed</button>
</div>

<script>
  Let tasks = JSON.parse(localStorage.getItem(“tasks”)) || [];

  Function saveTasks() {
    localStorage.setItem(“tasks”, JSON.stringify(tasks));
  }

  Function renderTasks() {
    Let list = document.getElementById(“taskList”);
    List.innerHTML = “”;
    Tasks.forEach((task, index) => {
      Let li = document.createElement(“li”);

      If (task.editing) {
        // Edit mode
        Let editBox = document.createElement(“input”);
        editBox.type = “text”;
        editBox.value = task.text;
        editBox.className = “edit-box”;
        editBox.onblur = () => finishEdit(index, editBox.value);
        editBox.onkeydown = € => {
```

```
    if (e.key === “Enter”) finishEdit(index, editBox.value);
    if (e.key === “Escape”) cancelEdit(index);
   };
   li.appendChild(editBox);
   editBox.focus();
  } else {
   // Normal mode
   li.textContent = task.text;
   if (task.completed) li.classList.add(“completed”);

   // Toggle complete on click
   li.onclick = () => {
    tasks[index].completed = !tasks[index].completed;
    saveTasks();
    renderTasks();
   };

   // Double click for edit
   li.ondblclick = € => {
    e.stopPropagation();
    tasks[index].editing = true;
    renderTasks();
   };

   // Delete button
   Let span = document.createElement(“span”);
```

```
      Span.textContent = " ×";
      Span.onclick = € => {
        e.stopPropagation();
        tasks.splice(index, 1);
        saveTasks();
        renderTasks();
      };
      li.appendChild(span);
    }

    List.appendChild(li);
  });
}

Function addTask() {
  Let input = document.getElementById("taskInput");
  Let taskText = input.value.trim();
  If (taskText === "") return;
  Tasks.push({ text: taskText, completed: false, editing: false });
  Input.value = "";
  saveTasks();
  renderTasks();
}

Function finishEdit(index, newText) {
  Tasks[index].editing = false;
```

```
  If (newText.trim() !== “”) {
    Tasks[index].text = newText.trim();
  } else {
    Tasks.splice(index, 1);
  }
  saveTasks();
  renderTasks();
}

Function cancelEdit(index) {
  Tasks[index].editing = false;
  renderTasks();
}

Function clearCompleted() {
  Tasks = tasks.filter(task => !task.completed);
```

FLOW DIAGRAM