COLLAGE CODE: 9604

COLLAGE NAME: CSI INSTITUTE OF TECHNOLOGY

DEPARTMENT: COMPUTER SCIENCE AND ENGINEERING

STUDENT NM ID: D3AAE798B05A3FA9FEA49AA542B7E1E0

ROLL NO: 960423104072

DATE: 23/09/2025

SUBMITTED BY,

NAME: Sree Aswin.S

MOBILE NO: 9342338715

PHASE 3-MVP IMPLEMENTATION

USER REGISTRATION WITH VALIDATION

Project Setup:

1. Define the Purpose

The project is meant to allow users to **register an account** safely, ensuring their information is valid and stored securely. Key requirements include:

- Collecting user information (name, email, password, etc.).
- Validating inputs to prevent errors or malicious data.
- Storing user data in a database securely.
- Giving feedback to the user (success or errors).

2. Identify Core Features

1. **User Input Fields**:
   - Full Name
   - Email Address
   - Password
   - Confirm Password
   - Optional fields: phone number, username, etc.
2. **Validation Rules**:
   - Name: cannot be empty.
   - Email: must be in correct email format, unique in the database.
   - Password: minimum length, strong enough.
   - Confirm Password: must match the password.
3. **User Feedback**:
   - Show errors if inputs are invalid.
   - Show success message after successful registration.

3. Choose the Tech Stack

- **Frontend (User Interface)**: React, Angular, Vue, or plain HTML/CSS/JS.
- **Backend (Server & Logic)**: Node.js with Express, Django, or any server-side language.
- **Database (Storage)**: MySQL, PostgreSQL, MongoDB, or any relational/non-relational DB.
- **Validation**:
  - Client-side: prevent bad inputs immediately.
  - Server-side: confirm inputs before saving to DB.

4. Organize the Project Structure

- **Frontend**:
  - Pages/components for registration form.
  - Input fields with validation logic.
  - Form submission to backend API.

- **Backend**:
    - API endpoint for registration (/register).
    - Input validation logic on server.
    - Password encryption before saving to database.
    - Database model/schema for users.

5. Setup Validation Workflow

- **Client-side Validation**:
    - Check inputs before sending to the server.
    - Display errors (like “invalid email” or “password too short”).
- **Server-side Validation**:
    - Verify the same rules again.
    - Ensure email is unique in database.
    - Encrypt password for security.
    - Save the user if everything is valid.
- **Database Rules**:
    - Ensure unique email addresses.
    - Store hashed passwords only.

6. Additional Considerations

- **Security**: Hash passwords, use HTTPS, prevent SQL injection.
- **User Experience**: Clear error messages, confirmation on successful registration.
- **Future Features**:
    - Email verification.
    - Captcha to prevent bots.
    - Login functionality.
    - Profile management.

CODE:

```
<!DOCTYPE html>

<html lang="en">

<head>

 <meta charset="UTF-8">

 <title>User Registration Form</title>

 <style>

   body {

     font-family: Arial, sans-serif;

     background: #f2f2f2;

     display: flex;

     justify-content: center;

     align-items: center;

     height: 100vh;

   }
```

```
.container {

  background: #fff;

  padding: 30px 40px;

  border-radius: 8px;

  box-shadow: 0 0 10px rgba(0,0,0,0.1);

  width: 350px;

}


h2 {

  text-align: center;

  margin-bottom: 20px;

  color: #333;

}


input[type=text],

input[type=email],

input[type=password] {

  width: 100%;

  padding: 10px;

  margin: 6px 0 12px 0;

  border: 1px solid #ccc;

  border-radius: 4px;

}


button {

  width: 100%;

  padding: 10px;

  background: #4CAF50;

  color: white;
```

```
      border: none;

      border-radius: 4px;

      cursor: pointer;

      font-size: 16px;

    }


    button:hover {

      background: #45a049;

    }


    .error {

      color: red;

      font-size: 14px;

      margin-bottom: 10px;

    }


    .success {

      color: green;

      font-size: 16px;

      text-align: center;

      margin-top: 10px;

    }

 </style>

</head>

<body>


<div class="container">

 <h2>Register</h2>

 <form id="registrationForm">

 <div class="error" id="nameError"></div>
```

```
    <input type="text" id="name" placeholder="Full Name">


    <div class="error" id="emailError"></div>

    <input type="email" id="email" placeholder="Email">


    <div class="error" id="passwordError"></div>

    <input type="password" id="password" placeholder="Password">


    <div class="error" id="confirmPasswordError"></div>

    <input type="password" id="confirmPassword" placeholder="Confirm Password">


  <button type="submit">Register</button>

  <div class="success" id="successMessage"></div>

  </form>

 </div>


 <script>

  const form = document.getElementById('registrationForm');

  const nameInput = document.getElementById('name');

  const emailInput = document.getElementById('email');

  const passwordInput = document.getElementById('password');

  const confirmPasswordInput = document.getElementById('confirmPassword');


  const nameError = document.getElementById('nameError');

  const emailError = document.getElementById('emailError');

  const passwordError = document.getElementById('passwordError');

  const confirmPasswordError = document.getElementById('confirmPasswordError');

  const successMessage = document.getElementById('successMessage');


  form.addEventListener('submit', function(e) {
```

```
e.preventDefault();


// Clear previous messages

nameError.textContent = '';

emailError.textContent = '';

passwordError.textContent = '';

confirmPasswordError.textContent = '';

successMessage.textContent = '';


let isValid = true;


//Name validation

if(nameInput.value.trim() === '') {

  nameError.textContent = 'Name is required';

  isValid = false;

}


//Email validation

const emailPattern = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;

if(emailInput.value.trim() === '') {

  emailError.textContent = 'Email is required';

  isValid = false;

}else if (!emailPattern.test(emailInput.value.trim())) {

  emailError.textContent = 'Enter a valid email';

  isValid = false;

}


//Password validation

if(passwordInput.value.trim() === '') {

  passwordError.textContent = 'Password is required';
```

```
      isValid = false;

    }else if (passwordInput.value.length < 6) {

      passwordError.textContent = 'Password must be at least 6 characters';

      isValid = false;

    }


    //Confirm password validation

    if(confirmPasswordInput.value.trim() === '') {

      confirmPasswordError.textContent = 'Confirm your password';

      isValid = false;

    }else if (confirmPasswordInput.value !== passwordInput.value) {

      confirmPasswordError.textContent = 'Passwords do not match';

      isValid = false;

    }


    // If all valid

    if (isValid) {

      successMessage.textContent = 'Registration successful!';

      form.reset();

    }

  });

</script>


</body>

</html>
```

OUTPUT:

1. Initial Form Load

- Centered white box on a light gray background.
- Title: **“Register”** at the top.
- Four input fields stacked vertically:
  1. Full Name
  2. Email
  3. Password

4. Confirm Password

- A green **Register** button at the bottom.
- No error or success messages initially.

2. Validation Errors

When the user clicks **Register** without filling fields or enters invalid data:

- Each invalid input shows a **red error message** just above the field.
    - Example:
        - Name empty → "Name is required"
        - Invalid email → "Enter a valid email"
        - Password too short → "Password must be at least 6 characters"
        - Passwords don't match → "Passwords do not match"
- Multiple errors can appear at once if multiple fields are invalid.

3. Successful Registration

When all inputs are valid:

- Red error messages disappear.
- A **green success message** appears below the Register button:
    - **"Registration successful!"**
- All input fields are **cleared automatically**.

4. Visual Layout Summary

```
-------------------------------

|         Register         |

-------------------------------

| Name                  |

| [______________________]    |

| Error: "Name is required" |

-------------------------------

| Email                 |

| [______________________]    |

| Error: "Enter a valid email" |

-------------------------------

| Password              |

| [______________________]    |

| Error: "Password too short"|

-------------------------------

| Confirm Password        |

| [______________________]    |
| Error: "Passwords do not match"|

-------------------------------

|      [Register Button]    |

-------------------------------

| Success: "Registration successful!" |

-------------------------------
```

- **Responsive**: The form is centered and stays neat on small screens.
- **Interactive**: Errors appear only when invalid, success appears when valid.

CORE FEATURES IMPLEMENTATION:

1. User Registration Form

- A **user registration form** is the interface where users enter their details to create an account.
- Typical fields include:
  - **Full Name**
  - **Email Address**
  - **Username**
  - **Password**
  - **Confirm Password**
  - Optional: Phone number, Date of Birth, etc.

2. Input Validation

- Validation ensures that users enter data correctly and securely.
- **Client-side validation (using JavaScript/HTML):**
  - **Required Fields**: Ensure no field is left empty.
  - **Email Format**: Check if the email has a valid structure (e.g., user@example.com).
  - **Password Rules**: Minimum length, uppercase/lowercase letters, numbers, special characters.
  - **Password Match**: Confirm Password must match the Password field.
  - **Username Rules**: Avoid spaces or special characters if not allowed.
- **Server-side validation (backend checks):**
  - Re-check all client-side rules for security.
  - Ensure **unique email and username** (no duplicates in the database).
  - Prevent malicious inputs (e.g., SQL injection, XSS attacks).

3. Password Security

- **Hashing**: Store passwords in a hashed format (e.g., bcrypt) instead of plain text.
- **Salting**: Add a unique random string to each password before hashing for extra security.

4. Database Storage

- User information is stored securely in a **database**.
- Example database fields:
  - id (Primary Key)
  - name
  - email
  - username
  - password_hash
  - created_at (timestamp)

5. Feedback & Error Handling

- Users should get **clear messages** for invalid inputs:

    - “Email already exists”
    - “Password must be at least 8 characters”
    - “Passwords do not match”
- On successful registration, show a **confirmation message** or redirect to login.

6. Optional Features

- **Email Verification**: Send a verification link to the user’s email before activating the account.
- **Captcha Integration**: To prevent bots from registering automatically.
- **Terms & Conditions Agreement**: User must agree before registration.

7. Flow of Registration with Validation

- User fills in the form.
- Client-side validation checks inputs.
- Form is submitted to the server.
- Server-side validation checks inputs and uniqueness.
- Password is hashed and stored in the database.
- User gets success or error message.
- Optional: Email verification link is sent.

DATA STORAGE:

1. Purpose of Data Storage

The main goal is to **store user information securely and efficiently** after registration, so that users can log in, retrieve their profiles, and the system can manage accounts properly.

2. Types of Data to Store

Typical information collected from a registration form includes:

Field	**Purpose**
Full Name	To identify the user
Email Address	Login credential and communication
Username	Unique identifier for the user
Password Hash	Secure authentication (never store plain passwords)
Phone Number	Optional, for verification or contact
Date of Birth	Optional, for age verification
Account Status	Active, pending verification, suspended
Created At	Timestamp of registration

3. Database Choice

- **Relational Databases (SQL)**: MySQL, PostgreSQL, SQLite
  - Pros: Structured data, supports constraints like unique email/username, easy to query
  - Tables example: users
- **NoSQL Databases**: MongoDB, Firebase
  - Pros: Flexible structure, scalable, useful for rapidly changing schemas
  - Collections example: users with JSON-like documents

4. Data Security

- **Password Storage**:
  - Use **hashing** algorithms like bcrypt or Argon2
  - Add **salt** to enhance security
- **Email & Personal Data**:
  - Encrypt sensitive information if required
- **Database Access**:
  - Use restricted access with proper authentication
  - Prevent SQL Injection using prepared statements or ORM

5. Validation Before Storage

- **Server-side validation** ensures that only correct and safe data is stored:
  - Unique email and username
  - Valid email format
  - Password strength
  - Required fields are filled
- Client-side validation is optional for better user experience but **server-side validation is mandatory** for security.

6. Data Flow for Registration

- User fills the registration form.
- Client-side validation checks inputs (optional but recommended).
- Data is sent to the server.
- Server-side validation checks inputs again.
- Password is hashed.
- User data is inserted into the database.
- Optional: Send email verification.
- Registration is complete, user can now log in.

7. Optional Storage Enhancements

- **Audit logs**: Track when a user registers or updates profile.
- **Verification tokens**: Store temporary tokens for email/phone verification.
- **Session storage**: Save session info if the user logs in immediately after registration.

TESTING CORE FEATURES:

1. Purpose of Testing

The main goal is to **ensure that the registration process works correctly, securely, and efficiently**. Testing verifies that users can register with valid information and that invalid or malicious inputs are properly handled.

2. Core Features to Test

- **Form Input Validation**
  - Check that **all required fields** are validated.
  - Test **email format** (user@example.com), invalid emails should be rejected.
  - Test **password strength** rules (length, numbers, uppercase/lowercase, special characters).
  - Ensure **password confirmation** matches the password field.
  - Validate **username rules** (no spaces or invalid characters).
- **Unique Constraints**
  - Attempt registration with an **already used email** → should fail.
  - Attempt registration with an **already used username** → should fail.
- **Password Security**
  - Ensure **passwords are hashed** in the database.
  - Test that plain passwords are not stored.
- **Database Storage**
  - Check that valid user data is **successfully stored**.
  - Confirm timestamps and status fields are correctly set (created_at, account_status).
- **Error & Success Messages**
  - Test that **clear, user-friendly messages** appear for:
    - Invalid inputs
    - Duplicate email/username
    - Weak password
  - Ensure **success message** appears on successful registration.
- **Optional Features**
  - Email verification: Check that verification emails are sent and links work.
  - Captcha integration: Test that bots cannot bypass the form.
  - Terms & Conditions: Ensure registration fails if checkbox is unchecked.

3. Types of Testing

- **Manual Testing**
  - Enter different combinations of inputs to check validations.
  - Try edge cases like extremely long usernames or special characters.

- **Automated Testing**
  - Write **unit tests** for backend validation functions.
  - Create **integration tests** to simulate full registration workflow.
  - Test database constraints automatically.
- **Security Testing**
  - Test for **SQL injection**, XSS, or other malicious inputs.
  - Verify that passwords and sensitive data are secure.

4. Sample Test Scenarios

Test Case	**Input**	**Expected Result**
Empty Fields	Leave all fields blank	Show error “Field is required”
Invalid Email	user@@mail	Show error “Invalid email format”
Weak Password	123	Show error “Password too weak”
Mismatched Password	Password=abc123 Confirm=abc124	Show error “Passwords do not match”
Duplicate Email	Email already in DB	Show error “Email already exists”
Valid Registration	Proper input for all fields	Show success message and save data in DB

5. Testing Workflow

- Open registration form.
- Fill in the form with **test inputs** (valid and invalid).
- Submit the form.
- Verify:
  - Validation messages
  - Data stored in database
  - Security rules applied
  - Optional features (email verification, captcha)
- Repeat with **edge cases** and **malicious inputs**.

VERSION CONTROL:

1. Purpose of Version Control

Version control is used to **track changes, manage code history, and collaborate safely** during the development of a user registration system.For user registration with validation, it ensures that any updates to form design, validation rules, or database schema are **well-documented and reversible**.

2. Key Version Control Features

- **Tracking Changes**
  - Keep a **history of all code changes** (HTML, CSS, JS, backend code, database scripts).
  - Allows developers to **compare different versions** to find bugs or revert mistakes.
- **Collaboration**
  - Multiple developers can work on **different features simultaneously** (e.g., frontend validation vs backend storage).
  - Merges changes into a single project without overwriting each other's work.
- **Branching**
  - Create **branches** for new features or bug fixes:
    - feature/validation-enhancement → adding stricter password rules.
    - bugfix/email-duplication → fixing duplicate email validation.
  - Once tested, merge into the **main branch**.
- **Commit Messages**
  - Use **clear messages** to describe changes:
    - Example: Added server-side email format validation
    - Example: Updated password hashing to bcrypt with salt
- **Tags & Releases**
  - Mark stable versions of registration system for deployment:
    - Example: v1.0-basic-registration
    - Example: v1.1-email-verification-added

3. Recommended Workflow

- **Initialize Version Control**
  - Use Git to track files.
  - Example: git init in project folder.
- **Create Repository**
  - Local repository or hosted on platforms like **GitHub, GitLab, or Bitbucket**.
- **Branching Strategy**
  - main → stable version
  - develop → ongoing development
  - feature/* → individual features like form validation, captcha, email verification
- **Committing Changes**
  - Make small, descriptive commits:
    - Example: git commit -m "Added client-side password validation"
- **Merging and Pull Requests**
  - After testing a feature branch, merge into main or develop branch.
  - Use **pull requests** for code review and approval.
- **Handling Conflicts**
  - Resolve conflicts carefully when multiple developers modify the same files (e.g., validation logic in JS or backend scripts).

4. Benefits for User Registration System

- **Traceability**: Know when a validation rule or database change was added.
- **Rollback**: Revert to a previous stable version if a bug occurs.
- **Collaboration**: Multiple developers can safely work on frontend, backend, and database.
- **Documentation**: Commit messages act as a record of feature additions, bug fixes, and updates.

CODE IMPLEMENTATION:

```
<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">
```

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

<title>User Registration All-in-One</title>

<style>

body {

 font-family: Arial, sans-serif;

    background-color: #f5f5f5;

}

.container {

 width: 400px;

 margin: 50px auto;

    padding: 30px;

    background: #fff;

    border-radius: 10px;

 box-shadow: 0 0 10px #aaa;

}

h2 {

    text-align: center;

}

label {

    display: block;

 margin-top: 10px;

}

input {

 width: 100%;

    padding: 8px;

 margin-top: 5px;

}

button {

 width: 100%;

    padding: 10px;
 margin-top: 15px;

   background-color: #28a745;

   border: none;

   color: #fff;

   font-size: 16px;

   cursor: pointer;

   border-radius: 5px;

}

#message {

 margin-top: 10px;

   text-align: center;

   color: red;

}

</style>

</head>

<body>

<div class="container">

   <h2>User Registration</h2>

 <form id="registrationForm">

       <label>Full Name</label>

       <input type="text" id="name" required>



       <label>Email</label>

       <input type="email" id="email" required>



       <label>Username</label>

       <input type="text" id="username" required>



       <label>Password</label>

       <input type="password" id="password" required>
```

```
        <label>Confirm Password</label>

        <input type="password" id="confirmPassword" required>


        <button type="submit">Register</button>

        <p id="message"></p>

 </form>

</div>


<script>

// Function to validate email

function validateEmail(email) {

 const re = /\S+@\S+\.\S+/;

 return re.test(email);

}


// Function to validate password strength

function validatePassword(password) {

    const re = /^(?=.*[a-z])(?=.*[A-Z])(?=.*\d).{8,}$/;

 return re.test(password);

}


// Function to hash password (simple hash for demo, not secure for production)

function simpleHash(str) {

    let hash = 0;

    for (let i = 0; i < str.length; i++) {

        hash = (hash << 5) - hash + str.charCodeAt(i);

        hash = hash & hash;

    }

    return hash.toString();
```

```
}


const form = document.getElementById('registrationForm');

const message = document.getElementById('message');


form.addEventListener('submit', (e) => {

   e.preventDefault();

 message.style.color = 'red';


   const name = document.getElementById('name').value.trim();

   const email = document.getElementById('email').value.trim();

   const username = document.getElementById('username').value.trim();

   const password = document.getElementById('password').value;

   const confirmPassword = document.getElementById('confirmPassword').value;


   //Client-side validation

   if(!name || !email || !username || !password || !confirmPassword) {

      message.textContent = "All fields are required";

      return;

   }


   if(!validateEmail(email)) {

      message.textContent = "Invalid email format";

      return;

   }


   if(password !== confirmPassword) {

      message.textContent = "Passwords do not match";

      return;

   }
```

```
if(!validatePassword(password)) {
   message.textContent = "Password must be at least 8 characters, include uppercase, lowercase, and number";
   return;
}

// Check if user already exists
let users = JSON.parse(localStorage.getItem('users') || '[]');
const emailExists = users.some(u => u.email === email);
const usernameExists = users.some(u => u.username === username);

if(emailExists) {
   message.textContent = "Email already exists";
   return;
}

if(usernameExists) {
   message.textContent = "Username already exists";
   return;
}

//Store user data in localStorage
const user = {
   name,
   email,
   username,
   passwordHash: simpleHash(password),
   createdAt: new Date().toISOString()
};
users.push(user);
```

```
    localStorage.setItem('users', JSON.stringify(users));


    message.style.color = 'green';

    message.textContent = "Registration successful!";


 // Reset form

 form.reset();

});

</script>

</body>

</html>
```

EXPECTED OUTPUT:

1. Initial View

When you open the page in a browser, you'll see:

```
---------------------------------------

|          User Registration          |

---------------------------------------

Full Name: [          ]

Email:    [          ]

Username: [          ]

Password: [          ]

Confirm Password: [    ]

[ Register Button ]

Message: (empty)

---------------------------------------
```

- A clean, centered registration form with all fields.
- No messages are shown initially.

2. Validation Error Messages

a) Empty Fields

- If you click **Register** without filling any field:

Message: "All fields are required" (in red)

b) Invalid Email

- If you enter user@@mail in the email field:

Message: "Invalid email format" (in red)

c) Password Mismatch

- If password and confirm password don’t match:

Message: "Passwords do not match" (in red)

d) Weak Password

- If password doesn’t meet the rules (less than 8 chars, no uppercase/lowercase/number):

Message: "Password must be at least 8 characters, include uppercase, lowercase, and number" (in red)

e) Duplicate Email or Username

- If an email or username already exists in localStorage:

Message: "Email already exists" (or "Username already exists") (in red)

3. Successful Registration

- If all fields are valid and the email/username is unique:

Message: "Registration successful!" (in green)

- The form is cleared automatically.
- User data is stored in the browser’s localStorage:

Example stored data (localStorage.getItem('users')):

```
[

 {

 "name": "Vanitha Mathavan",

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

 "username": "vanitha123",

 "passwordHash": "1234567890", // demo hash

   "createdAt": "2025-09-23T20:30:00.000Z"

 }

]
```

4. LocalStorage Behavior

- Every successful registration adds a new user to the users array.
- You can check it in the browser console:

```
JSON.parse(localStorage.getItem('users'))
```

- It will show all registered users with hashed passwords and timestamps.

5. Visual Summary

Action	**Expected Message**	**Color**
Empty fields	"All fields are required"	Red
Invalid email	"Invalid email format"	Red
Password mismatch	"Passwords do not match"	Red
Weak password	"Password must be at least 8 characters, include uppercase, lowercase, and number"	Red
Duplicate email	"Email already exists"	Red
Duplicate username	"Username already exists"	Red
Successful registration	"Registration successful!"	Green

This gives a **complete picture of what the user sees and how data is stored**.

FUTURE ENHANCEMENT:

Here's a detailed list of **future enhancements for a user registration system with validation**, going beyond the basic all-in-one implementation:

1. Enhanced Security

- **Secure Password Hashing**
  - Replace the demo hash with **bcrypt** or **Argon2** for production-level password security.
- **HT TPS**
  - Ensure all data is transmitted over HTTPS to protect sensitive user information.
- **Two-Factor Authentication (2FA)**
  - Add email or SMS-based OTP verification for login and registration.
- **ReCAPTCHA / Bot Protection**
  - Integrate Google reCAPTCHA to prevent automated bot registrations.
- **Password Strength Meter**
  - Show a visual indicator for password strength while typing.

2. Database & Backend Integration

- **Use a Real Database**
  - Store user data in MySQL, PostgreSQL, or MongoDB instead of localStorage.
- **Email Verification**
  - Send verification links to confirm user email before activating the account.

- **Username & Email Normalization**
  - Convert emails to lowercase to avoid duplicates and standardize usernames.
- **Role-Based Registration**
  - Support multiple roles like “user”, “admin”, or “moderator” with different privileges.

3. User Experience Enhancements

- **Real-Time Validation**
  - Show validation errors **as the user types**, instead of after form submission.
- **Show/Hide Password**
  - Allow users to toggle password visibility.
- **Responsive Design**
  - Improve layout for mobile, tablet, and desktop screens.
- **Auto-Fill & Suggestions**
  - Provide suggestions for username availability or generate a secure password.
- **Friendly Error Messages**
  - Make error messages more descriptive and helpful.

4. Analytics & Logging

- **Registration Metrics**
  - Track how many users register daily/weekly/monthly.
- **Error Logging**
  - Log failed registration attempts for security monitoring.
- **Audit Trail**
  - Track user registration, updates, and verification activities.

5. Scalability & Maintainability

- **API-Based Registration**
  - Implement RESTful or GraphQL endpoints for registration to support multiple frontends.
- **Version Control**
  - Use proper branching and release management for feature updates.
- **Microservices**
  - Separate registration, authentication, and email verification services for large-scale apps.
- **Unit & Integration Testing**
  - Automated tests for registration and validation to ensure system stability.

6. Optional Enhancements

- **Social Login**
  - Enable registration via Google, Facebook, or Apple accounts.
- **Profile Picture Upload**
  - Allow users to upload avatars during registration.
- **Localization / Multilingual Support**
  - Provide error messages and UI text in multiple languages.
- **Password Recovery**
  - Add “Forgot Password” workflow with secure reset link.