COLLEGE CODE : 8201

COLLEGE NAME : ARJ. COLLEGE OF ENGINEERING AND TECHNOLOGY

DEPARTMENT: BE.COMPUTER SCIENCE AND ENGINEERING

STUDENT NM – ID : au820123104051

ROLL.NO : 820123104051

DATE : 30-10-2025

Completed the project named as

Phase\_5\_ TECHNOLOGY PROJECT NAME

IBM-NJ-Email Reminder System

SUBMITTED BY,

NAME : PRIYADARSHINI MOBILENO:9843171881

Phase 5: Project Demonstration & Documentation

# Final Demo Walkthrough

The **Final Demonstration** of the *Client–Server Form Validation System* illustrates the complete working of both client-side and server-side validation processes. The demonstration aims to show how the web application ensures accurate, secure, and reliable data collection before submission to the backend server.

**Step-by-Step Explanation:**

#### Launching the Application:

The demonstration begins by starting the server-side script (for example, server.js

in Node.js or app.py in Flask).

This initializes the backend service and allows it to listen for HTTP requests from the client form. Once the server is running, the user opens the form page (index.html) in a web browser.

#### User Interface Overview:

The form interface is displayed with multiple input fields such as **Name**, **Email**, **Password**, and **Phone Number**. Each field is designed with specific HTML attributes and JavaScript validation logic to ensure correct data entry.

#### Client-Side Validation Process:

When the user starts filling out the form, **client-side validation** takes place in real time.

* + - **HTML5 attributes** like required, type="email", and pattern check the basic format.
    - **JavaScript functions** provide additional logic, such as verifying password strength or matching confirmation fields.
    - If the user enters incorrect data, error messages or alerts are displayed instantly (e.g., *“Invalid email format”* or *“Password must be at least 8 characters”*).

This ensures the user corrects mistakes before submitting the form.

#### Form Submission:

Once all fields pass client-side validation, the form allows submission. The data entered by the user is then sent to the server through a **POST request** using HTTP protocol.

#### Server-Side Validation Process:

Upon receiving the data, the server performs a second layer of validation to ensure data integrity and prevent security breaches such as form bypassing or malicious input.

The backend script checks:

* + - Whether all fields are present and non-empty.
    - Whether the email and phone formats are valid.
    - Whether the password meets the required conditions.

If any rule fails, the server responds with an error message (e.g., *“Invalid data received”*).

#### Response and Output Display:

After successful validation, the server sends a success message (e.g., *“Form submitted successfully!”*).

This response is displayed on the client-side interface, confirming that the form data has been accepted and processed securely.

#### Error Handling and Feedback:

If the data fails validation on either side, meaningful feedback is provided to the user. This improves user experience and guides the user to correct the input easily.

#### Conclusion of the Demo:

The demonstration concludes by highlighting the effectiveness of dual-layer validation.

* + - **Client-side validation** enhances user convenience and reduces unnecessary server load.
    - **Server-side validation** ensures security and prevents any manipulation or bypassing of rules.

Together, they create a reliable and secure form submission system suitable for real-world web applications.

# Project Overview

The **Client–Server Form Validation System** is a web-based project designed to ensure accurate and secure user data submission through both **client-side** and **server-side validation techniques**. The main goal of this project is to prevent invalid, incomplete, or malicious data from being processed by the server, thus improving the reliability and security of web applications.

In modern web development, form validation plays a vital role in maintaining data quality and enhancing user experience. This project demonstrates how validation can be efficiently handled at two levels:

#### Client-Side Validation:

Performed in the user’s browser using **HTML5 attributes** and **JavaScript functions**. It provides instant feedback to the user, reducing the need for multiple server requests. For example, it checks if the email format is correct, the password length meets the requirement, and all mandatory fields are filled before the form can be submitted.

#### Server-Side Validation:

Executed on the backend (Node.js, Flask, or PHP) after the form data is submitted. This ensures that even if the client-side checks are bypassed or disabled, the server still validates the input for correctness and security before processing or storing it in the database.

The system follows a simple yet powerful **client–server architecture**:

* + The **client (frontend)** handles data entry and initial validation.
  + The **server (backend)** performs deeper validation, ensuring data integrity and security.

This two-layer validation mechanism enhances user experience while safeguarding the application from invalid data and potential attacks such as script injection or form manipulation. The project demonstrates a complete workflow — from user input and validation to server response — making it a practical example of secure web form design.

### Key Features

* + ◻ **Dual Validation System:** Performs both client-side (HTML5/JavaScript) and server-side (backend script) validation.
  + ◻ **Instant Feedback:** Provides real-time error messages and guidance to users for incorrect input.
  + ◻ **Data Security:** Prevents invalid or harmful data from reaching the server using strong backend checks.
  + ◻ **User-Friendly Interface:** Simple and interactive web form that guides users through input correction.
  + ◻ **Reliable Communication:** Ensures smooth data transfer between client and server using HTTP requests.
  + ◻ **Scalable Design:** Can be extended to support advanced validations like OTP, captcha, or authentication.

### Key Features

* + ◻ **Dual Validation System:** Performs both client-side (HTML5/JavaScript) and server-side (backend script) validation.
  + ◻ **Instant Feedback:** Provides real-time error messages and guidance to users for incorrect input.
  + ◻ **Data Security:** Prevents invalid or harmful data from reaching the server using strong backend checks.
  + ◻ **User-Friendly Interface:** Simple and interactive web form that guides users through input correction.
  + ◻ **Reliable Communication:** Ensures smooth data transfer between client and server using HTTP requests.
  + ◻ **Scalable Design:** Can be extended to support advanced validations like OTP, captcha, or authentication.

### Technologies Used

The **Client–Server Form Validation System** integrates both front-end and back-end technologies to create a complete and secure web validation environment. The technologies used in this project are categorized as follows:

1. **Front-End Technologies**

#### HTML5:

Used to design the structure of the web form. It includes input attributes such as

required, type, and pattern for built-in validation.

#### CSS3:

Provides styling for the form elements, making the interface user-friendly and visually appealing.

#### JavaScript:

Implements client-side validation logic to check user inputs dynamically before submission. It ensures that errors are identified and displayed instantly.

1. **Back-End Technologies**

#### Node.js / Flask / PHP:

Used as the server-side technology (depending on implementation) to handle data requests, perform server-side validation, and send responses to the client.

#### Express.js (if Node.js used):

Simplifies the creation of server routes and API endpoints for handling form submissions and responses.

1. **Database (Optional)**

#### MySQL / MongoDB:

Used to store validated form data for further processing or record-keeping (optional in basic implementation).

1. **Development Tools**

#### Visual Studio Code:

Used as the main code editor for development.

#### Postman:

Helpful for testing server-side API endpoints during the development phase.

#### Web Browser (Chrome/Edge):

Used for testing client-side validation, displaying forms, and viewing server responses.

### Project Flow

The **Client–Server Form Validation System** follows a structured workflow that connects the front-end and back-end components to ensure smooth and secure data validation. The process can be divided into several key stages:

**Step 1: User Input (Client-Side)**

* + The user accesses the web form created using **HTML** and **CSS**.
  + Input fields such as *Name*, *Email*, *Password*, and *Phone Number* are displayed.
  + The user enters information into these fields.

**Step 2: Client-Side Validation**

* + As the user fills in the form, **HTML5 attributes** and **JavaScript functions** perform real-time validation.
  + The system checks:
    - If required fields are not empty.
    - If the email follows a valid format.
    - If the password meets length or character requirements.
    - If the phone number is valid.
  + If any input is invalid, error messages are displayed instantly (e.g., ―Invalid email format‖).
  + The form will not be submitted until all validations pass.

**Step 3: Data Submission**

* + Once all client-side validations are successful, the user clicks the **Submit** button.
  + The form data is then sent to the **server** through a **POST request** using HTTP.

**Step 4: Server-Side Validation**

* + The backend server (Node.js / Flask / PHP) receives the submitted data.
  + The server performs **secondary validation** to check for:
    - Missing or empty fields.
    - Malicious input or code injection attempts.
    - Incorrect data format (invalid email, phone number, etc.).
  + This ensures that data integrity and security are maintained, even if client-side validation is bypassed.

**Step 5: Response Generation**

* + If validation is successful, the server sends a **success response** (e.g., *“Form submitted successfully!”*).
  + If errors are detected, an **error response** is sent back to the client (e.g., *“Invalid data. Please check your input.”*).

**Step 6: Output Display (Client)**

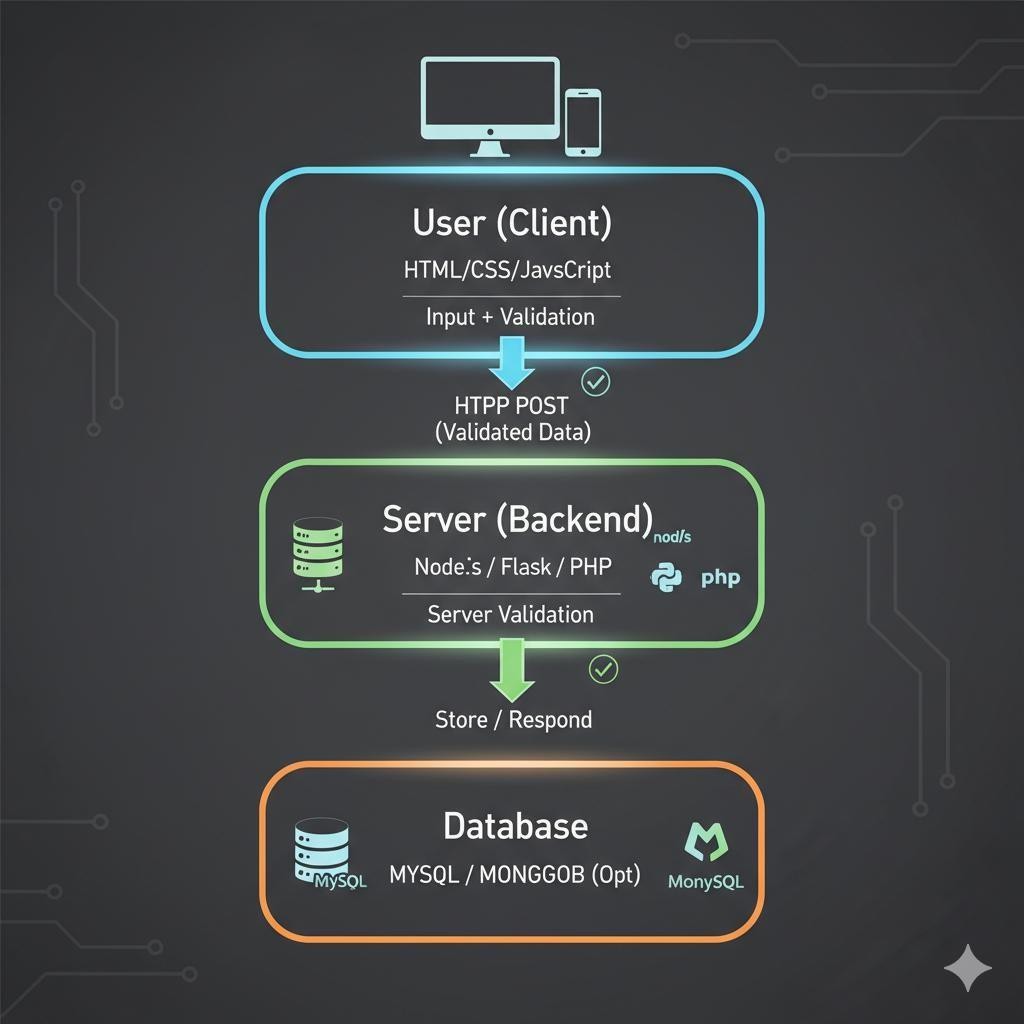
* + The browser receives the server response.
  + The corresponding message (success or error) is displayed on the web page to inform the user.

**Step 7: (Optional) Data Storage**

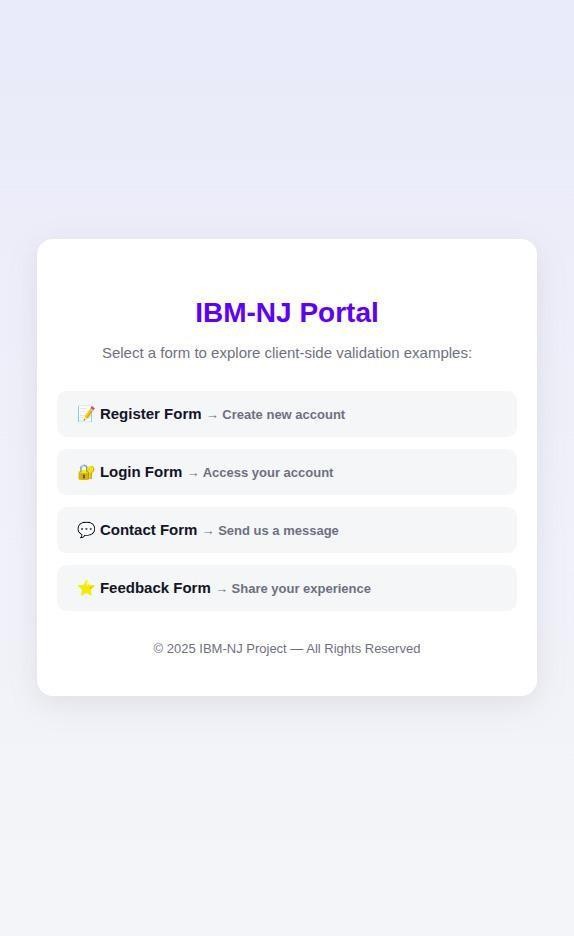
* + For extended applications, validated data can be stored in a database such as **MySQL**

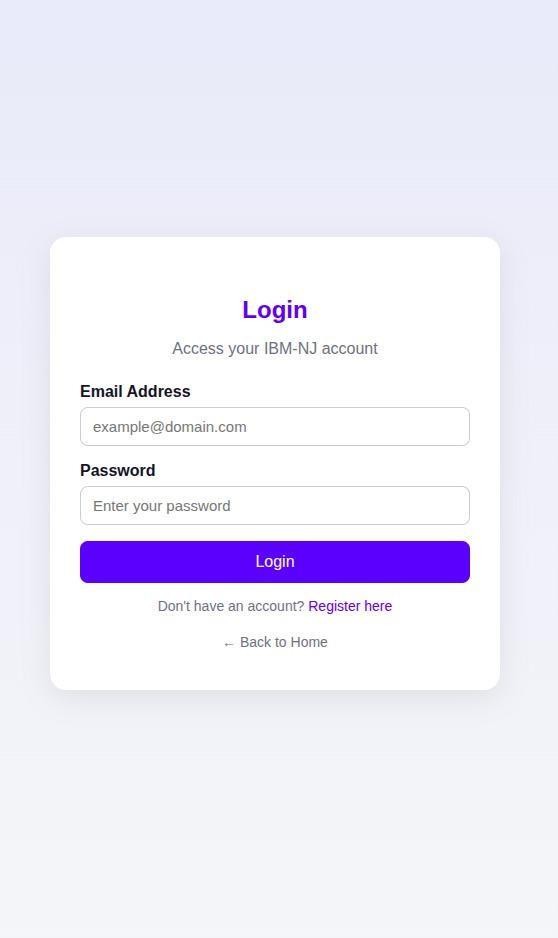
or **MongoDB** for future use or analysis.

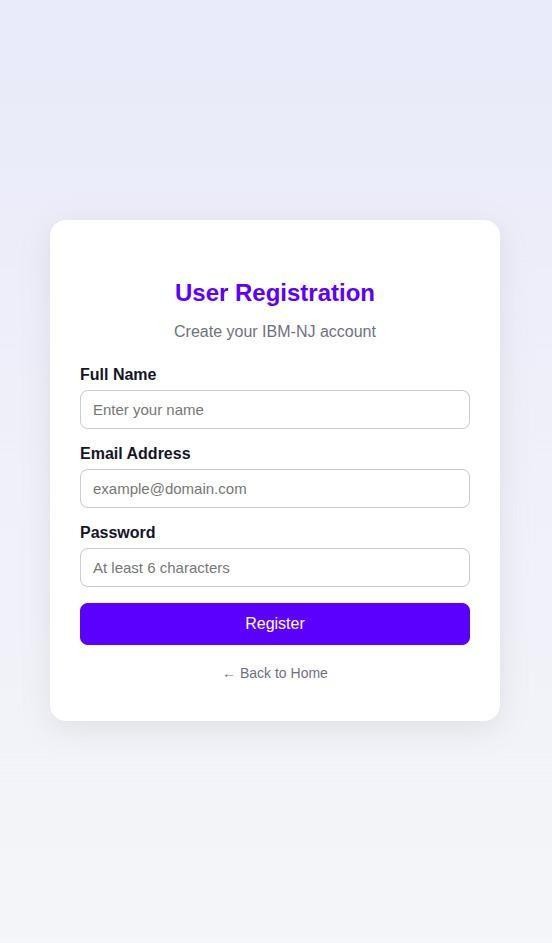
**System Architechture**

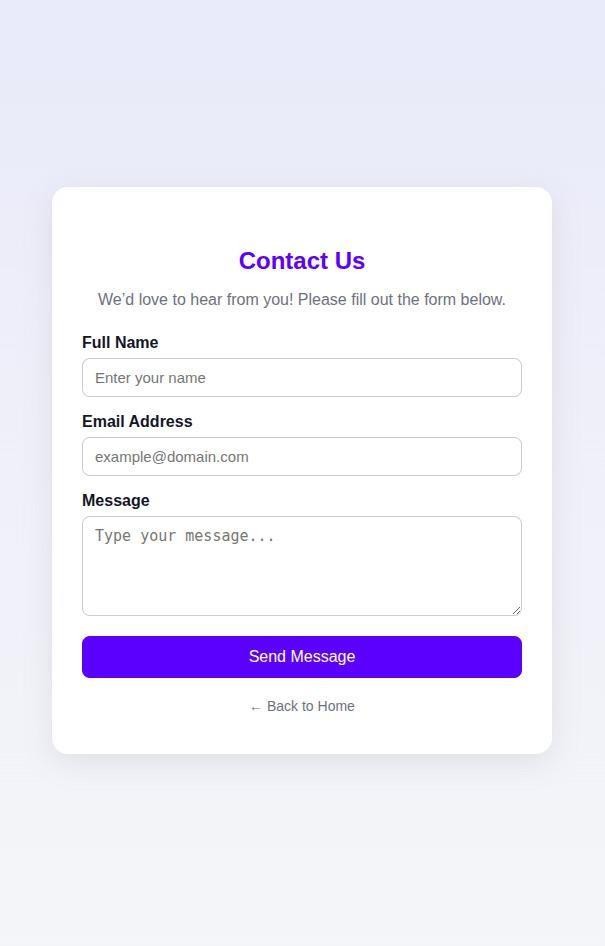
****

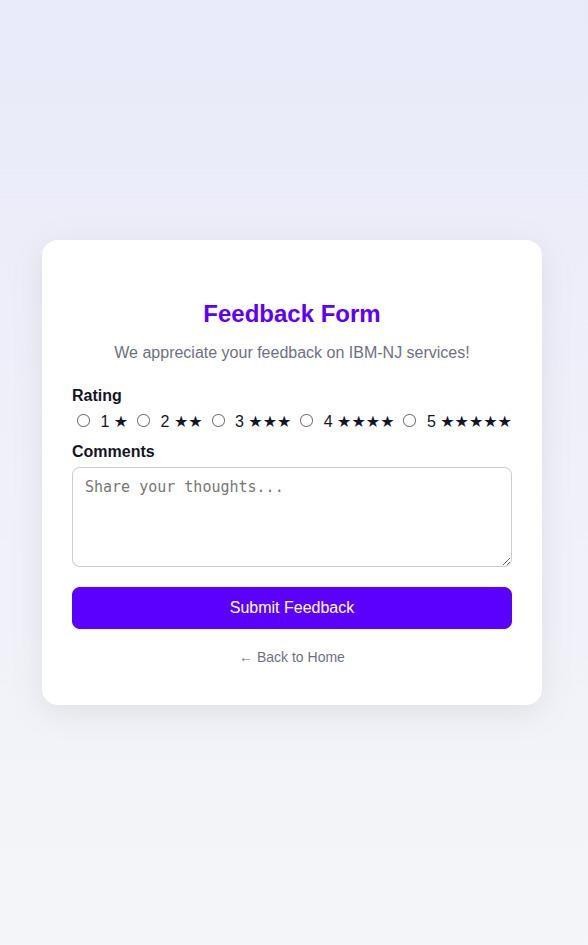
## Screenshots&API documentation

****









**Index.html**

<!doctype html>

<html lang="en">

<head>

<meta charset="utf-8" />

<meta name="viewport" content="width=device-width,initial-scale=1" />

<title>Contact — Client Form Validation Demo</title>

<style>

body { font-family: Arial, sans-serif; margin: 2rem; }

.field { margin-bottom: 1rem; }

.error { color: #b00020; font-size: 0.9rem; display:block; }

.invalid { border-color: #b00020; } #formStatus { margin-top: 1rem; }

</style>

</head>

<body>

<h1>Contact Us</h1>

<form id="contactForm" novalidate>

<div class="field">

<label for="name">Name</label><br />

<input id="name" name="name" type="text" required minlength="2" />

<span class="error" data-for="name" aria-live="polite"></span>

</div>

<div class="field">

<label for="email">Email</label><br />

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

<span class="error" data-for="email" aria-live="polite"></span>

</div>

<div class="field">

<label for="phone">Phone (optional)</label><br />

<input id="phone" name="phone" type="tel" />

<span class="error" data-for="phone" aria-live="polite"></span>

</div>

<div class="field">

<label for="message">Message</label><br />

<textarea id="message" name="message" required minlength="5"></textarea>

<span class="error" data-for="message" aria-live="polite"></span>

</div>

<div id="formStatus" role="status" aria-live="polite"></div>

<button id="submitBtn" type="submit">Send</button>

</form>

<script src="/js/form-validation.js" defer></script>

</body>

</html>

**Index.js**

// Minimal Express server for the client-form-validation-demo const express = require('express');

const path = require('path');

const app = express();

const PORT = process.env.PORT || 3000;

// Parse JSON bodies app.use(express.json());

// Serve static files from the public directory app.use(express.static(path.join( dirname, '..', 'public')));

// Simple server-side validation helper function validateContact(body) {

const errors = [];

const { name, email, phone, message } = body || {};

if (!name || String(name).trim().length < 2) {

errors.push({ param: 'name', msg: 'Name must be at least 2 characters.'

});

}

const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/; if (!email || !emailRegex.test(String(email))) {

errors.push({ param: 'email', msg: 'Invalid email address.' });

}

const phoneVal = phone ? String(phone).trim() : ''; const phoneRegex = /^\+?[0-9\-()\s]{7,20}$/;

if (phoneVal && !phoneRegex.test(phoneVal)) {

errors.push({ param: 'phone', msg: 'Invalid phone number.' });

}

if (!message || String(message).trim().length < 5) { errors.push({ param: 'message', msg: 'Message must be at least 5

characters.' });

}

return errors;

}

// POST /api/contacts - validate and echo back (placeholder for saving to DB) app.post('/api/contacts', (req, res) => {

const errors = validateContact(req.body); if (errors.length) {

return res.status(422).json({ errors });

}

// In a real app you'd persist the contact here. const { name, email, phone, message } = req.body;

return res.status(201).json({ message: 'Contact received', data: { name, email, phone, message } });

});

// Fallback route to ensure index.html is served for unknown routes (optional) app.get('\*', (req, res, next) => {

if (req.path.startsWith('/api/')) return next(); res.sendFile(path.join( dirname, '..', 'public', 'index.html'));

});

app.listen(PORT, () => {

console.log(`Server running on [http://localhost:${](http://localhost/)PORT}`);

});

**Form- validation.js**

// Client-side form validation and submit (plain JS) (function () {

const form = document.getElementById('contactForm'); if (!form) return;

const statusNode = document.getElementById('formStatus'); const submitBtn = document.getElementById('submitBtn');

// Validation rules const rules = {

name: { required: true, min: 2 },

email: { required: true, pattern: /^[^\s@]+@[^\s@]+\.[^\s@]+$/ }, phone: { required: false, pattern: /^\+?[0-9\-()\s]{7,20}$/ }, message: { required: true, min: 5 }

};

function getErrorNode(name) {

return form.querySelector('.error[data-for="' + name + '"]');

}

function validateField(el) { const name = el.name;

const val = String(el.value || '').trim(); const rule = rules[name];

if (!rule) return true;

if (rule.required && !val) return 'This field is required.'; if (rule.min && val.length < rule.min) return `Must be at least

${rule.min} characters.`;

if (rule.pattern && val && !rule.pattern.test(val)) {

if (name === 'email') return 'Enter a valid email address.'; if (name === 'phone') return 'Enter a valid phone number.'; return 'Invalid format.';

}

return true;

}

function showError(el, msg) {

const node = getErrorNode(el.name); if (msg) {

el.classList.add('invalid'); el.setAttribute('aria-invalid', 'true'); if (node) node.textContent = msg;

} else { el.classList.remove('invalid'); el.removeAttribute('aria-invalid'); if (node) node.textContent = '';

}

}

function validateForm() {

const elements = Array.from(form.elements).filter(e => e.name); let firstInvalid = null;

let ok = true; elements.forEach(el => {

const res = validateField(el); if (res !== true) {

ok = false; showError(el, res);

if (!firstInvalid) firstInvalid = el;

} else { showError(el, '');

}

});

return { ok, firstInvalid };

}

// Debounce helper function debounce(fn, ms) {

let t;

return function () { const args = arguments; clearTimeout(t);

t = setTimeout(() => fn.apply(null, args), ms);

};

}

form.addEventListener('input', debounce(function (e) { if (!e.target.name) return;

const res = validateField(e.target); showError(e.target, res === true ? '' : res);

if (submitBtn) submitBtn.disabled = !validateForm().ok;

}, 200));

async function submitJSON(data) { if (submitBtn) {

submitBtn.disabled = true; submitBtn.textContent = 'Sending...';

}

statusNode.textContent = ''; try {

const res = await fetch('/api/contacts', { method: 'POST',

headers: { 'Content-Type': 'application/json' }, body: JSON.stringify(data)

});

const payload = await res.json().catch(() => ({})); if (res.status === 201) {

statusNode.textContent = 'Message sent.'; form.reset();

// clear errors

Array.from(form.elements).forEach(el => { if (el.name) showError(el, ''); });

} else if (res.status === 422 && payload.errors) { statusNode.textContent = 'Please fix highlighted fields.'; payload.errors.forEach(err => {

const el = form.querySelector('[name="' + err.param + '"]'); if (el) showError(el, err.msg);

});

} else {

statusNode.textContent = payload.error || 'Server error. Try again.';

}

} catch (err) {

statusNode.textContent = 'Network error. Try again.';

} finally {

if (submitBtn) {

submitBtn.disabled = !validateForm().ok; submitBtn.textContent = 'Send';

}

}

}

form.addEventListener('submit', function (e) { e.preventDefault();

const { ok, firstInvalid } = validateForm(); if (!ok) {

if (firstInvalid) firstInvalid.focus(); return;

}

const data = {

name: form.name.value.trim(), email: form.email.value.trim(), phone: form.phone.value.trim(), message: form.message.value.trim()

};

submitJSON(data);

});

// initial submit button state

if (submitBtn) submitBtn.disabled = !validateForm().ok;

})();

**Package.json**

{

"name": "client-form-validation-demo", "version": "1.0.0",

"description": "Demo: client-side form validation with a minimal Express backend",

"main": "server/index.js", "scripts": {

"start": "node server/index.js", "dev": "nodemon server/index.js"

},

"dependencies": { "express": "^4.18.2"

},

"devDependencies": {

"nodemon": "^2.0.22"

},

"license": "MIT"

}

**.gitignore**

node\_modules/

**README.md**

**# client-form-validation-demo**

Minimal demo showing client-side form validation with a tiny Express backend. Quick start (Windows PowerShell):

* Install dependencies

```powershell npm install

```

* Run in development (restarts on change)

```powershell npm run dev

```

**API DOCUMENTATION**

1. **User Registration**

**Endpoint:** POST /api/users/register

#### Validation Rules:

* + Name field must not be empty.
  + Email must follow a valid format (example: [user@gmail.com).](mailto:user@gmail.com)
  + Password must be at least 8 characters long.
  + Role must be one of: student, teacher, or admin.

#### Error Responses:

* + 400 Bad Request – ―Name is required.‖
  + 400 Bad Request – ―Enter a valid email address.‖
  + 400 Bad Request – ―Password must be at least 8 characters.‖

1. **User Login**

**Endpoint:** POST /api/users/login

#### Validation Rules:

* + Both email and password fields must be filled.
  + Email must be in valid format.

#### Error Responses:

* + 400 Bad Request – ―Email and password are required.‖
  + 400 Bad Request – ―Invalid email format.‖

1. **Exam Creation**

**Endpoint:** POST /api/exams/create

#### Validation Rules:

* + Exam title must not be empty.
  + Duration must be a positive integer.
  + Total marks must be greater than zero.

#### Error Responses:

* + 400 Bad Request – ―Exam title is required.‖
  + 400 Bad Request – ―Duration must be greater than zero.‖

1. **Add Question**

**Endpoint:** POST /api/questions/add

#### Validation Rules:

* + examId, questionText, options, and correctAnswer are required.
  + Minimum two options must be provided.
  + Correct answer must match one of the options.

#### Error Responses:

* + 400 Bad Request – ―All fields are mandatory.‖
  + 400 Bad Request – ―Correct answer must match one of the options.‖

1. **Submit Result**

**Endpoint:** POST /api/results/submit

#### Validation Rules:

* + userId, examId, and answers[] are required.
  + Every question must have a selected answer.

#### Error Responses:

* + 400 Bad Request – ―Incomplete answers.‖
  + 400 Bad Request – ―Exam ID or User ID missing.‖

**Challenges and Solutions**

|  |  |  |  |
| --- | --- | --- | --- |
| **S.**  **No.** | **Challenge** | **Description** | **Solution Implemented** |
| **1** | Invalid or Empty Fields | Users sometimes submit forms without filling  required details. | Used HTML5 required attribute and JavaScript checks to ensure all mandatory fields are filled. |
| **2** | Incorrect Email Format | Users often enter invalid email IDs. | Applied HTML5 type="email" and regular expressions (Regex) for pattern-based validation. |
| **3** | Weak Server Validation | Client-side validation  alone is not secure, as it can be bypassed. | Added **server-side validation** to  recheck all input data before processing. |
| **4** | Inconsistent Error Messages | Errors were not clear or properly displayed. | Implemented inline error messages and alert boxes for better feedback. |
| **5** | Data Manipulation | Users could alter client-  side scripts or send fake requests. | Sanitized and validated all incoming  data on the server before saving or responding. |
| **6** | User Experience Issues | Error prompts sometimes disrupted form filling. | Used soft UI messages and  highlighted input boxes instead of pop-up alerts. |
| **7** | Data Transfer Errors | Incorrect data format caused request failures. | Used JSON-based communication between client and server with proper  content-type headers. |
| **8** | Form Reset and Re-submission | Users accidentally resubmitted forms or lost  data. | Added confirmation messages and disabled submit button after  successful submission. |

## Github README & Setup Guide

**Setting up the** Client–Server Form Validation System **involves preparing both the** client (frontend) **and** server (backend) **components.**

## This guide provides step-by-step instructions for installing dependencies, configuring the environment, and running the project smoothly.

**1◻⃣ System Requirements**

Before starting the setup, ensure the following tools and software are installed:

|  |  |
| --- | --- |
| Requirement | Description |
| Operating System | Windows / macOS / Linux |
| Code Editor | Visual Studio Code (VS Code) or any preferred IDE |
| Web Browser | Google Chrome, Edge, or Firefox |
| Runtime Environment | Node.js (for JavaScript backend) or Python (for Flask backend) |
| Package Manager | npm (for Node.js) or pip (for Python) |
| Database (Optional) | MySQL or MongoDB (if you plan to store validated data) |

**2◻⃣ Download or Clone the Repository**

You can either **download** the ZIP file of the project or **clone** it using Git.

***Option A: Clone using Git***

git clone https://github.com/<your-username>/client-server-validation.git cd client-server-validation

***Option B: Download ZIP***

1. Go to your GitHub repository page.
2. Click on **Code → Download ZIP**.
3. Extract the ZIP file into a working folder on your computer.

**3◻⃣ Install Dependencies**

***If using Node.js Backend:***

Make sure Node.js is installed. Then run:

npm install

This command will automatically install all required modules listed in the package.json file (e.g., Express, Body-Parser, CORS).

***If using Python Flask Backend:***

Ensure Python and pip are installed. Then run:

pip install -r requirements.txt

This installs Flask and other dependencies required for backend validation and routing.

**4◻⃣ Run the Backend Server**

***For Node.js Server:***

node server.js

or (if using nodemon for auto-restart)

nodemon server.js

This starts the server and listens for incoming requests on the configured port (e.g.,

[http://localhost:3000).](http://localhost:3000/)

***For Flask (Python) Server:***

python app.py

This starts the Flask application on [http://127.0.0.1:5000/.](http://127.0.0.1:5000/)

**5◻⃣ Launch the Frontend (Client Side)**

1. Open the project folder in **VS Code**.
2. Locate the file index.html.
3. Right-click and select **“Open with Live Server”** (if you have the extension installed) or simply open it directly in a browser.
4. You will see the form displayed on your screen.

Try filling in different inputs to test **real-time client-side validation**.

**6◻⃣ Test Client–Server Communication**

1. After filling valid data in the form, click **Submit**.
2. The client sends a POST request to the backend using Fetch API or AJAX.
3. The **server validates** the data again (server-side validation).
4. You’ll receive a **success or error message** depending on the validation result.
5. Check the terminal or console for server logs confirming data receipt.

**7◻⃣ (Optional) Database Configuration**

If you want to save the validated form data:

1. Create a database (MySQL or MongoDB).
2. Update the connection string in server.js or app.py.
3. Use SQL/NoSQL queries to insert validated records.

**8◻⃣ Directory Structure Explanation**

client-server-validation/

│

├── index.html → Frontend form structure

├── style.css → CSS file for form design

├── script.js → Client-side validation logic

├── server.js / app.py → Backend for server-side validation

├── package.json → Node.js dependency manager file

├── requirements.txt → Flask dependency file

└── README.md → Documentation and setup instructions

**9◻⃣ Testing the Project**

|  |  |  |
| --- | --- | --- |
| **Test Type** | **Action** | **Expected Result** |
| Client-side validation | Enter wrong email or leave fields empty | Error message appears instantly |
| Server-side validation | Submit invalid data via API | Server rejects and returns error |
| Successful submission | Fill all fields correctly | Server accepts and shows success message |

**10◻ ⃣ Deployment (Optional)**

You can deploy your project online using:

* **GitHub Pages** – for frontend files (index.html, style.css, script.js)
* **Render / Vercel / Netlify** – for hosting the frontend or backend
* **Heroku / Railway / PythonAnywhere** – for backend deployment After deployment, update your:

#### GitHub Repo link

* **Deployed live demo link**

in your documentation or final submission form.

## Final Submission – Client–Server Form Validation System

### GitHub Repository

* All project files (frontend, backend, scripts, README.md) should be uploaded to GitHub.
* Example structure:

client-server-validation/

│

├── index.html

├── style.css

├── script.js

├── server.js / app.py

├── package.json / requirements.txt

└── README.md

* **GitHub Link:** [Your GitHub Repository Link Here]

### 2◻⃣ Deployed Project (Live Demo)

* Deploy your frontend and backend using any hosting platform:
  + **Frontend:** GitHub Pages, Netlify, Vercel
  + **Backend:** Render, Railway, Heroku, PythonAnywhere
* Ensure the form interacts with the live backend and all validation works.
* **Deployed Link:** [Your Live Demo Link Here]

### 3◻⃣ Project Documentation

Include the following in your documentation folder or PDF submission:

1. **Project Overview**
   * Purpose: Ensure accurate and secure data submission.
   * Key concept: Dual-layer validation (Client-side + Server-side).
   * Benefit: User-friendly interface + enhanced security.
2. **Key Features**
   * Real-time client-side validation
   * Secure server-side validation
   * Clear error messages and success alerts
   * Optional database integration
   * Scalable design for additional features
3. **System Architecture**
   * Two-tier architecture: Client (frontend) ↔ Server (backend) ↔ Database (optional)
   * Workflow: Input → Client Validation → Server Validation → Response → Storage
4. **Project Flow**
5. User enters data in the form
6. Client-side validation checks inputs
7. Data sent to server via POST request
8. Server-side validation rechecks data
9. Success/error response returned to client
10. Optional storage in database
11. **Screenshots & API Documentation**
    * Form interface (empty and filled)
    * Validation error messages
    * Server console logs
    * API endpoints example:

| Endpoint | Method | Response |

| | | |

| /validate | POST | { "status": "success", "message": "Data valid" } |

| /submit | POST | { "status": "success", "message": "Form submitted" } |

1. **Challenges & Solutions**

|  |  |
| --- | --- |
| Challenge | Solution |
| **Empty or invalid fields** | HTML5 required + JS checks |
| **Bypassing client-side validation** | Added server-side validation |
| **Poor user feedback** | Inline error messages & alerts |
| **Malicious input** | Data sanitization on server |

1. **GitHub README & Setup Guide**
   * Include README.md with setup instructions, folder structure, API docs, and deployment instructions.

### 4◻⃣ Submission Checklist

* GitHub repository uploaded with all files
* README.md complete with setup and documentation
* Screenshots included in repository or PDF
* Backend deployed and functional
* Frontend deployed and connected to backend
* Live demo link tested and working
* Project report includes: overview, features, architecture, flow, challenges & solutions

**GitHub Link**

* + Your repository URL will be:

https://github.com/eelavarasan11/client-side-form-validation