# **Contact Form with Validation – Full Documentation**

This project creates a contact form that validates user input on the client side and then sends the data to a backend server. The backend API, built with Express.js, receives the form data and saves it into a MongoDB database. The project is divided into three phases:

1. **Week 1 (Frontend):** Create and style the contact form using HTML and CSS.
2. **Week 2 (JavaScript Logic):** Validate the form data using JavaScript.
3. **Week 3 (Backend Integration & API):** Build an API with Express.js that accepts form submissions and stores them in MongoDB.

## **Project Structure**

contact-form-project/

├── public/

│ ├── index.html // Main HTML file for the contact form.

│ ├── style.css // CSS file for styling the contact form.

│ └── app.js // JavaScript file for handling form validation and API calls.

└── server.js // Node.js server file for backend API and MongoDB integration.

## **Architecture Overview**

Below is a diagram that illustrates the architecture of the project:

+-----------------------------+

| Web Browser |

| (Client: HTML, CSS, JS) |

+-------------+---------------+

|

User fills & submits the form

|

v

+-----------------------------+

| Frontend (app.js) |

| Validates and sends data |

| via Fetch API to /api/contact |

+-------------+---------------+

|

v

+-----------------------------+

| Express.js API |

| (server.js) |

| - Receives POST requests |

| - Processes JSON data |

+-------------+---------------+

|

Save data to MongoDB database

|

v

+-----------------------------+

| MongoDB |

| (contactDB, Contacts Collection) |

+-----------------------------+

**Explanation of the Architecture:**

* **Client (Web Browser):** Users interact with a responsive contact form built with HTML and styled with CSS. The client-side JavaScript (app.js) validates user input and then uses the Fetch API to send data to the backend.
* **Frontend (app.js):** The JavaScript code checks for required fields, validates the email format using a regular expression, and ensures the message does not exceed 250 characters. When validated, it sends a POST request to the API endpoint /api/contact.
* **Backend API (server.js):** The Express.js server handles incoming POST requests on /api/contact. It uses middleware to parse JSON data and then creates a new document using Mongoose to store the data in MongoDB. If successful, it returns a status code of 200.
* **Database (MongoDB):** MongoDB stores the contact messages in a database named contactDB with a collection managed via the Contact model.

## **1. Frontend – HTML & CSS**

### **public/index.html**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

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

<title>Contact Us</title>

<link rel="stylesheet" href="style.css">

</head>

<body>

<div class="container">

<h2>Contact Us</h2>

<!-- Contact Form -->

<form id="contactForm">

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

<input type="text" id="name" name="name" placeholder="Your Name" required>

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

<input type="email" id="email" name="email" placeholder="Your Email" required>

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

<textarea id="message" name="message" placeholder="Your Message" maxlength="250" required></textarea>

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

<p id="errorMsg"></p>

</form>

</div>

<script src="app.js"></script>

</body>

</html>

**Explanation:**

* The HTML structure sets up the contact form with fields for **Name**, **Email**, and **Message**.
* The <p> element with the ID errorMsg is reserved for displaying any validation errors.
* The file includes the app.js script at the end to manage the form’s behavior.

### **public/style.css**

/\* Basic Reset \*/

\* {

margin: 0;

padding: 0;

box-sizing: border-box;

}

body {

font-family: Arial, sans-serif;

background: #f9f9f9;

padding: 20px;

display: flex;

justify-content: center;

align-items: center;

height: 100vh;

}

/\* Container Styling \*/

.container {

max-width: 400px;

width: 100%;

background: #fff;

padding: 20px;

border-radius: 8px;

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

}

h2 {

margin-bottom: 20px;

text-align: center;

}

/\* Form Elements \*/

label {

display: block;

margin: 10px 0 5px;

}

input, textarea {

width: 100%;

padding: 10px;

margin-bottom: 10px;

border: 1px solid #ccc;

border-radius: 5px;

}

textarea {

resize: none;

height: 100px;

}

button {

width: 100%;

padding: 10px;

background: #007BFF;

color: white;

border: none;

border-radius: 5px;

cursor: pointer;

}

button:hover {

background: #0056b3;

}

#errorMsg {

color: red;

margin-top: 10px;

text-align: center;

}

**Explanation:**

* Provides a reset for default browser styles.
* Uses flexbox to center the container, ensuring the form is responsive.
* Styles for form elements improve usability and visual appeal.

## **2. Client-Side JavaScript (Validation & API Call)**

### **public/app.js**

document.addEventListener('DOMContentLoaded', () => {

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

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

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

e.preventDefault(); // Prevent the default form submission

// Retrieve and trim values from input fields

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

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

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

// Client-side validation:

if (!name || !email || !message) {

errorMsg.textContent = "All fields are required.";

return;

}

if (!isValidEmail(email)) {

errorMsg.textContent = "Invalid email address.";

return;

}

if (message.length > 250) {

errorMsg.textContent = "Message must be 250 characters or less.";

return;

}

// Clear error message if validations pass

errorMsg.textContent = "";

// API Call: Send data to the backend

try {

const response = await fetch('/api/contact', {

method: 'POST',

headers: {

'Content-Type': 'application/json'

},

body: JSON.stringify({ name, email, message })

});

// Check if response is OK

if (response.ok) {

alert('Message sent successfully!');

form.reset();

} else {

alert('Failed to send message.');

}

} catch (error) {

alert('Error: ' + error.message);

}

});

// Helper function to validate email using regex

function isValidEmail(email) {

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

return emailRegex.test(email);

}

});

**Explanation:**

* **Event Listener:** Waits for the DOM to load before attaching a submit handler to the form.
* **Validation:** Checks that all fields are filled, the email is in the proper format, and the message length does not exceed 250 characters. Errors are shown in the <p id="errorMsg">.
* **API Call:** Uses the Fetch API to send a POST request to /api/contact with the form data in JSON format. On a successful response, the user is alerted, and the form is reset.

## **3. Backend API & MongoDB Integration**

### **server.js**

const express = require('express');

const mongoose = require('mongoose');

const bodyParser = require('body-parser');

const path = require('path');

const app = express();

const PORT = 3000;

// Connect to MongoDB using Mongoose

mongoose.connect('mongodb://localhost:27017/contactDB', {

useNewUrlParser: true,

useUnifiedTopology: true

});

// Define a Mongoose schema for the contact form data

const contactSchema = new mongoose.Schema({

name: String,

email: String,

message: String

});

// Create a Mongoose model based on the schema

const Contact = mongoose.model('Contact', contactSchema);

// Middleware: parse JSON bodies from incoming requests

app.use(bodyParser.json());

// Serve static files from the public directory (HTML, CSS, JS)

app.use(express.static('public'));

/\*\*

\* API Endpoint: /api/contact

\* Method: POST

\* Description: Receives contact form submissions, validates the data, and stores it in MongoDB.

\*/

app.post('/api/contact', async (req, res) => {

try {

// Extract data from request body

const { name, email, message } = req.body;

// Create a new document based on the Contact model

const newContact = new Contact({ name, email, message });

// Save the document to the MongoDB database

await newContact.save();

// Respond with a success status and message

res.status(200).send('Message saved successfully');

} catch (error) {

console.error('Error saving message:', error);

res.status(500).send('Internal Server Error');

}

});

// Serve the main HTML file on root route

app.get('/', (req, res) => {

res.sendFile(path.join(\_\_dirname, 'public', 'index.html'));

});

// Start the Express server

app.listen(PORT, () => {

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

});

**Explanation:**

* **Express Setup:** Creates an Express server and sets it to listen on port 3000.
* **MongoDB Connection:** Uses Mongoose to connect to a local MongoDB database named contactDB.
* **Schema & Model:** Defines a Mongoose schema (contactSchema) and model (Contact) to represent contact form submissions.
* **Middleware:** bodyParser.json() is used to parse incoming JSON data.  
   express.static('public') serves the static HTML, CSS, and JavaScript files.
* **API Endpoint (/api/contact):** The POST endpoint receives the contact form data.  
   It creates a new Contact document and saves it to the database.  
   On success, it sends back a 200 status with a success message. On error, it returns a 500 status with an error message.
* **Serving the Frontend:** The root route (/) sends the index.html file so that the application is accessible via the browser.

## **4. How to Run the Project**

**Install Dependencies:** Ensure you have [Node.js](https://nodejs.org/) and [MongoDB](https://www.mongodb.com/) installed.  
  
 npm init -y

npm install express mongoose body-parser

1. **Start MongoDB:** Make sure MongoDB is running on your system (default port 27017).

**Run the Server:** Start the backend server with:  
  
 node server.js

You should see a message in the console:  
  
 Server is running on http://localhost:3000

1. **Access the Application:** Open your browser and navigate to<http://localhost:3000>.
2. **Test the Contact Form:** Fill in the form and submit. If validation passes, the data is sent to the API and stored in MongoDB. A success alert confirms that the message was sent.

## **5. Summary**

* **Frontend (HTML/CSS):** A responsive contact form with fields for Name, Email, and Message.
* **Client-Side JavaScript:** Validates the form data and sends a POST request to the API endpoint using the Fetch API.
* **Backend API (Express.js):** Receives the form data, processes it, and saves it in MongoDB using Mongoose.
* **Architecture Drawing:** Visualizes how the client, frontend JavaScript, API, and MongoDB interact.