COLLEGE CODE : 9222

COLLEGE NAME : THENI KAMMAVAR SANGAM COLLEGEOFTECHNOLOGY

DEPARTMENT : B.tech(IT)

STUDENT NM-ID : 7EB86FC4663BF6A07E4BA5E0BA8B673

B ROLL NO : 23it010

DATE : 26-09-2025

Completed the project named as

Phase__2 TECHNOLOGY PROJECT

NAME : **Client side form validation**

SUBMITTED BY,

NAME : A Gogulpandi

MOBILE NO : 9025086742

Solution Design & Architecture

1. Tech Stack Selection

Frontend Framework:

- **React.js** (Recommended for interactive UIs)
- **Vue.js** (Alternative to React, lightweight for form handling)
- **HTML/CSS** (Basic for form structure and design)

Form Validation Libraries:

- **Formik**: A popular form library for React that simplifies form handling, validation, and submission.
- **React Hook Form**: Another popular React form library that offers lightweight form management with built-in validation support.
- **Yup**: A schema validation library that can be used with Formik or React Hook Form to define validation rules (e.g., required fields, pattern matching).
- **Validator.js**: A lightweight library for validating common form inputs (e.g., email, phone number).
- **Custom Validation**: For simple scenarios, custom JavaScript validation logic can be directly written.

State Management (Optional):

- **Redux** or **Context API** for managing form states, especially in complex forms with dynamic fields or multi-step forms.

Other Useful Tools:

- **Styled Components** or **Tailwind CSS** for styling forms and showing validation messages dynamically.

2.UI Structure / API Schema Design

UI Structure:

A typical form layout includes:

- **Input Fields**:
  - Text inputs (name, email, password)
  - Radio buttons (for gender selection)
  - Checkboxes (terms and conditions)
  - Select dropdowns (for country, age group, etc.)
  - File upload (profile picture)
- **Validation Indicators**:
  - Error messages displayed below fields.
  - Inline validation (e.g., red border or icon next to invalid fields).
- **Submit Button**:
  - Disabled until all fields are validated.

Example UI:

```
-------------------------------------------------------
| Name:        [_______________] [Error message]     |
-------------------------------------------------------
```

```
| Email:       [_______________] [Error message]     |
-------------------------------------------------------
| Password:    [_______________] [Error message]     |
-------------------------------------------------------
| Submit Button [Submit] (disabled until valid)       |
-------------------------------------------------------
```

Form API Schema:

- **GET /api/formFields** (Optional): If the form fields are dynamic (e.g., fetched from a server), this API endpoint can provide the field definitions (e.g., field names, types, validation rules).
- **POST /api/submitForm**: When the form is valid and submitted, this endpoint receives the form data.

3. Data Handling Approach for Client-Side Validation

Client-side validation happens in the browser, before the form data is submitted to the server. The process involves the following:

Step-by-Step Approach:

1. **Initialization**:
    a. Define form fields and their validation rules.
    b. Use Formik or React Hook Form to handle form state and validation.
2. **User Interaction**:
    a. As the user types in a field, the form validates the input in real-time (e.g., checks if the email is valid or if the password meets complexity requirements).
3. **Validation Rules**:
    a. **Required Fields**: Ensure fields like name and email are not empty.

b. **Pattern Matching**: Use regex or built-in libraries (e.g., Yup or Validator.js) to validate formats like emails, phone numbers, etc.

c. **Password Strength**: Validate if passwords are strong (e.g., at least 8 characters, contains numbers, uppercase letters, etc.).

d. **Min/Max Length**: Check if text inputs (e.g., username) meet length requirements.

4. **Error Handling**:
    a. Track which fields have errors using state or libraries like Formik.
    b. Display error messages when validation fails.
5. **Form Submission**:
    a. Once all fields are validated successfully, allow the form to be submitted.
    b. If the form is invalid, prevent submission and highlight errors.

Client-Side Validation Example (Yup + Formik):

```
import { Formik, Field, Form, ErrorMessage } from 'formik';

import * as Yup from 'yup';

const validationSchema = Yup.object({

  name: Yup.string().required('Name is required'),

  email: Yup.string().email('Invalid email address').required('Email is required'),

  password: Yup.string().min(8, 'Password must be at least 8 characters').required('Password is required'),

});

const MyForm = () => {

  return (

    <Formik

      initialValues={{ name: '', email: '', password: '' }}
```

```
      validationSchema={validationSchema}
      onSubmit={(values) => {
        console.log(values);
      }}
    >
      <Form>
        <div>
          <label htmlFor="name">Name</label>
          <Field type="text" id="name" name="name" />
          <ErrorMessage name="name" component="div" />
        </div
<div>
          <label htmlFor="email">Email</label>
          <Field type="email" id="email" name="email" />
          <ErrorMessage name="email" component="div" />
        </div>

        <div>
          <label htmlFor="password">Password</label>
          <Field type="password" id="password" name="password" />
          <ErrorMessage name="password" component="div" />
        </div>
```

```
        <button type="submit">Submit</button>
      </Form>
    </Formik>
  );
};
```

4. Component / Module Diagram

The **Component Diagram** outlines how different parts of the form interact:

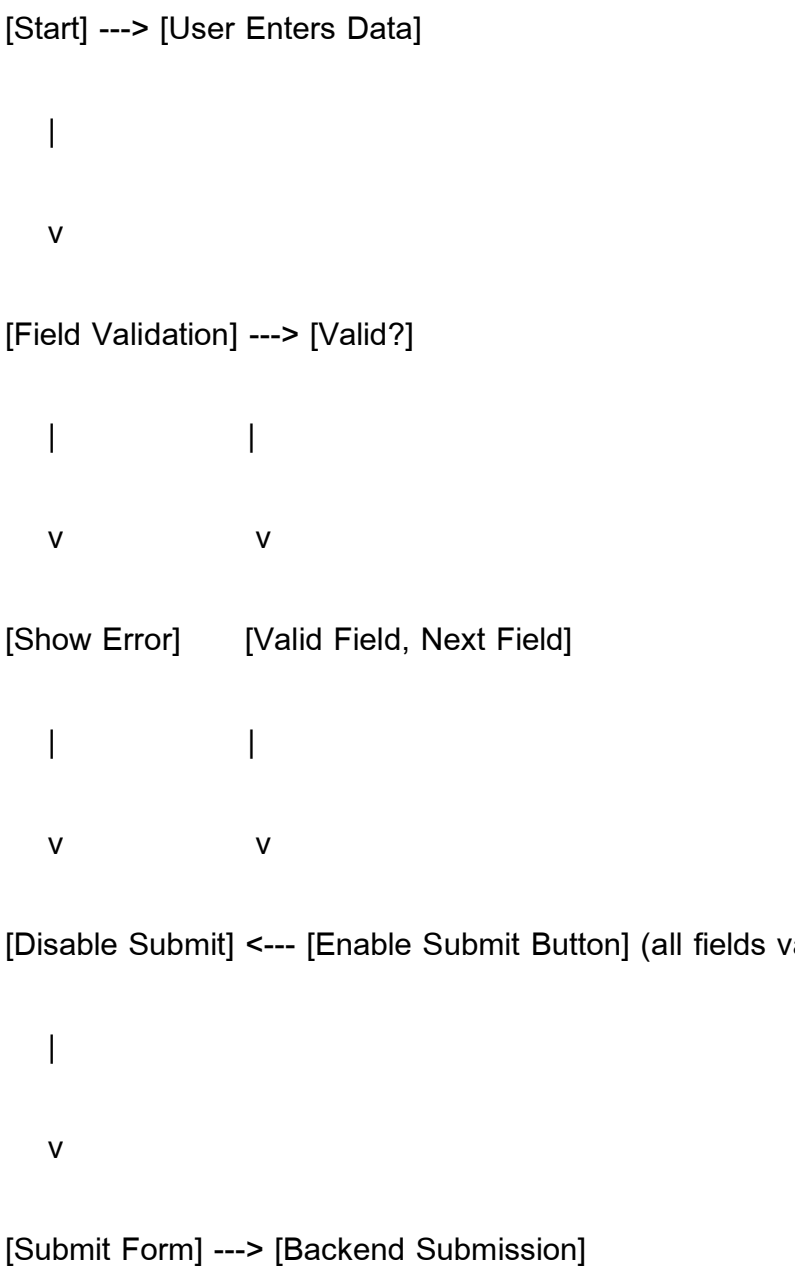
1. **Form Component**:
    a. Contains individual input fields and validation logic.
    b. Manages state for each field (via Formik or React Hook Form).
    c. Validates the input on user interaction (e.g., on blur, on change).
2. **Validation Component**:
    a. A validation schema (e.g., Yup) or custom validation functions applied to each field.
    b. Provides real-time feedback on validation errors (e.g., via `ErrorMessage` in Formik).
3. **Submit Button**:
    a. Disabled when the form is not valid.
    b. Triggers the form submission when valid.
4. **Error Message Display**:
    a. Displays error messages if the field input doesn't meet validation criteria.

```
[Form Component]
     |
[Field 1] ----> [Validation]
     |              |
[Field 2] <----> [Error Messages]
     |
[Submit Button]
     |
  [Form Submit]
```

5. Basic Flow Diagram for Client-Side Validation

Here's the **Basic Flow Diagram** for the form validation process:

1. **User Inputs Data** into form fields (e.g., name, email, password).
2. **Field Validation** occurs on input change, blur, or submit:
    a. If the data is valid, the field will show a green checkmark or remain without error.
    b. If the data is invalid, the field will show an error message.
3. **Submit Action**:
    a. The submit button is disabled until all fields are valid.
    b. Once valid, the user clicks the **Submit** button, triggering a form submission.
4. **Final Validation**:
    a. If there are any errors upon form submission, the user is prompted to correct them.
5. **Data Sent to Backend** (if all validations pass).



Conclusion:

This design ensures that the form handles client-side validation effectively, enhancing user experience by catching errors early and providing clear feedback. The architecture is modular, with distinct components responsible for handling input fields, validation, and

form submission. This modularity helps maintain scalability for complex forms in the future.

Would you like to dive into any specific area of this design or need more detailed code examples for any part?