Custom Form Validation

IIHT

Time To Complete: 10 to 12 hr

CONTENTS

1 Project Abstract

In modern front-end development, creating dynamic and reusable form components with built-in validation is essential for delivering flexible and user-friendly applications. This project focuses on building a customizable form validation system using **React** with **built-in custom validation rules**. The aim is to implement a **Dynamic Form** component capable of rendering various input types based on a configuration object, while applying appropriate validation rules without hardcoding field-specific logic.

The objective is to design a reusable, component-driven form system in React where form fields are generated dynamically and validated based on a flexible configuration. This implementation emphasizes the separation of concerns through reusable components (`DynamicForm`, `DynamicInput`), clean validation logic, and scalable state management using React hooks. The project also aims to highlight how to enforce **custom validation rules** via a utility module, improving maintainability and extensibility of form behavior.

2 Problem Statement

You are tasked with building a dynamic form rendering and validation system using **React**. The application must consist of multiple components that:

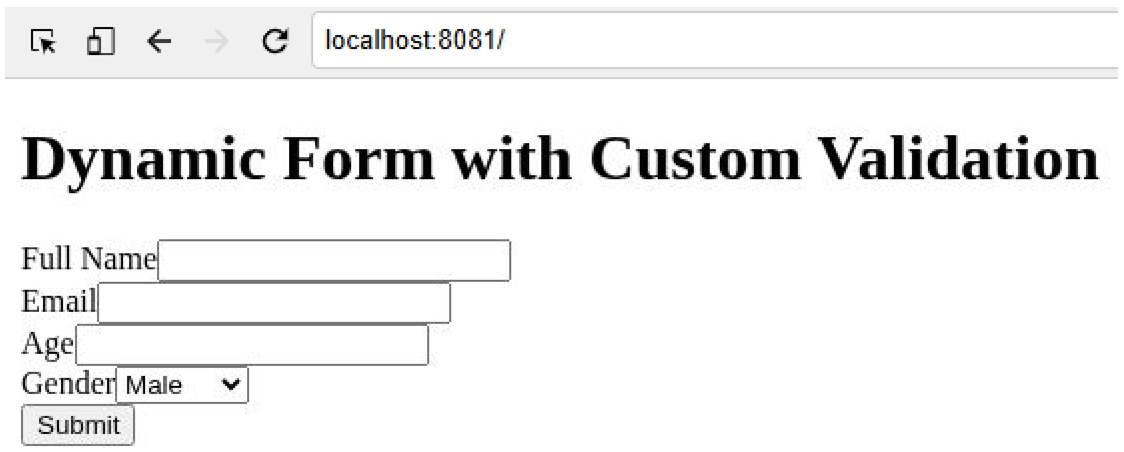
- Dynamically render input fields based on a form configuration.
- Handle user input and maintain internal state.
- Apply custom validation logic for different field types (e.g., text, email, number, dropdown).
- Display appropriate validation error messages on blur or form submission.
- Allow flexible extension of field types and validation rules.

The focus is on **modular design** using reusable components and centralized validation utilities, enabling a scalable form system suitable for real-world applications.

3 Proposed Custom Form Validation Application Wireframe

UI needs improvisation and modification as per given use case and to make test cases passed.

3.1 ScreenShots


localhost:8081/

Dynamic Form with Custom Validation

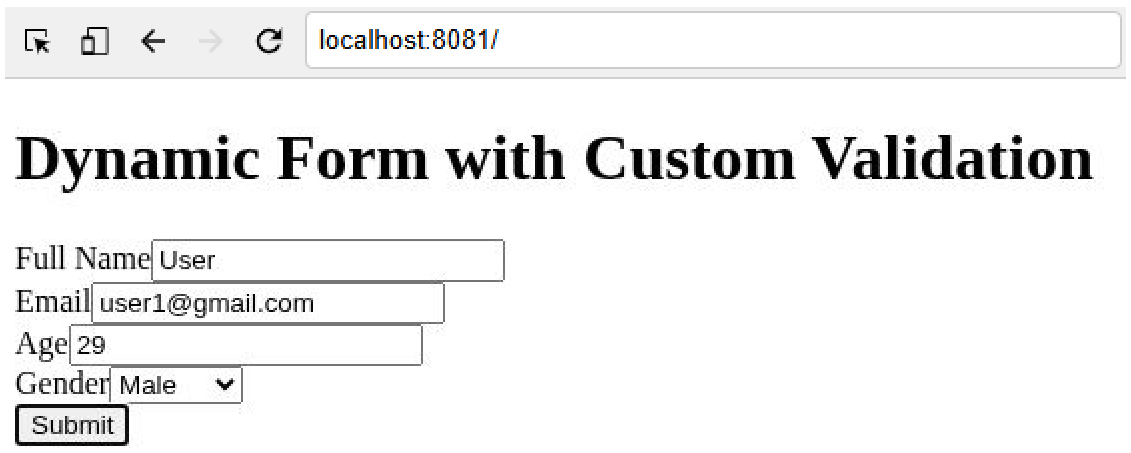
Full Name

Email

Age

Gender Male

Submit


localhost:8081/

Dynamic Form with Custom Validation

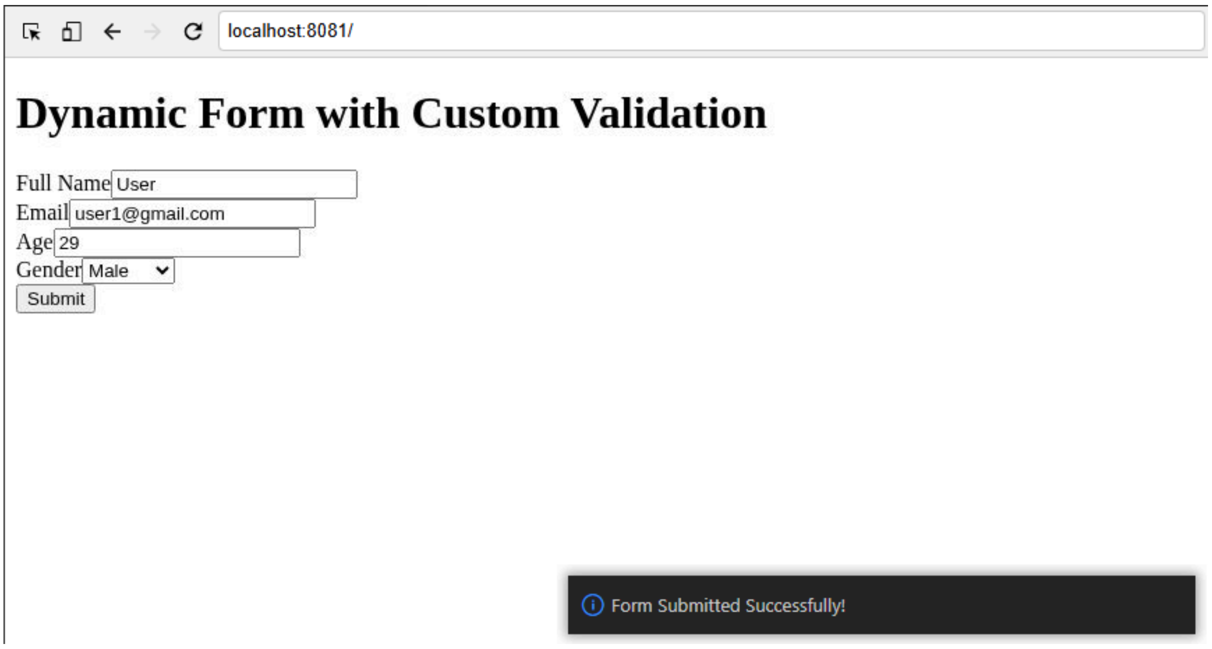
Full Name User

Email user1@gmail.com

Age 29

Gender Male

Submit

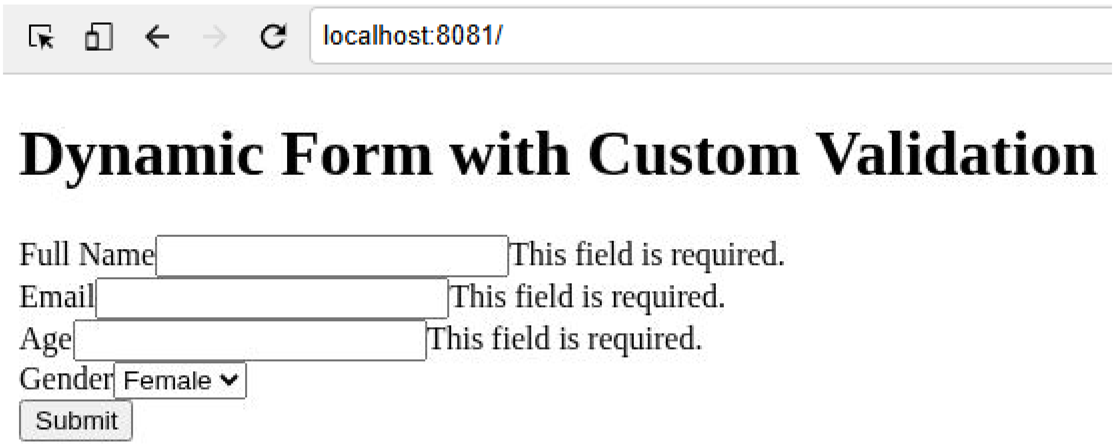

localhost:8081/

Dynamic Form with Custom Validation

Full Name User
Email user1@gmail.com
Age 29
Gender Male
Submit

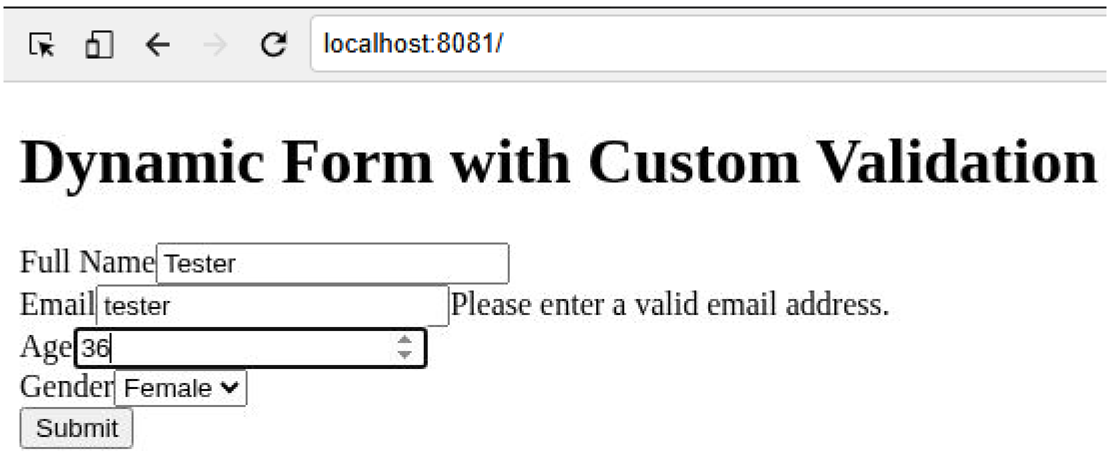
Form Submitted Successfully!

Validation


localhost:8081/

Dynamic Form with Custom Validation

Full Name This field is required.
Email This field is required.
Age This field is required.
Gender Female
Submit


localhost:8081/

Dynamic Form with Custom Validation

Full Name Tester

Email tester Please enter a valid email address.

Age 36

Gender Female

4 Business-Requirement:

As an application developer, develop the Custom Form Validation (Single Page App) with below guidelines:

User Story #	User Story Name	User Story
US_01	Welcome Page	As a user I should be able to visit the welcome page as the default page. **App Component** **Acceptance criteria:** 1. Display the heading: "Dynamic Form with Custom Validation" using `<h1>`. 2. Render the `DynamicForm` component underneath the heading. **HTML Structure:** 1. Use a top-level `<div>` to wrap the content. 2. Inside the `<div>`: ● Add a `<h1>` with the text: "Dynamic Form with Custom Validation". ● Render the `<DynamicForm />` component.

DynamicForm Component

Acceptance criteria:

1. Render a form dynamically using a configuration array.
2. For each field in the configuration:
   - Render an appropriate input based on its type.
   - Apply validation based on the rules provided.
   - Show an error message below the field if validation fails.
3. On form submit:
   - Validate all fields.
   - If validation passes, show a success alert.
   - If validation fails, show relevant errors next to each field.

Form Configuration Object (`formConfig`):

1. Create an array with the name “formConfig” which defines the fields that should appear in the form. Each item is an object with:
   - `type`: Field type (`text`, `email`, `number`, or `dropdown`)
   - `label`: Field label shown in UI
   - `name`: Unique identifier used in form state
   - `validation`: Object describing rules to apply
   - `value`: Initial value (usually `""`)
   - `options`: Only for dropdowns
2. **Example Fields to Implement:**
   - Full Name → required text
   - Email → required email
   - Age → required number
   - Gender → required dropdown (Male, Female)
3. Store this array as a constant called `formConfig`.

State Management:

State & Props Overview
State Variables:

1. `formValues (object)` –
   - Holds the current value for each field.

- Initialized using `formConfig` by mapping field names to default values.

2. `formErrors (object)` –
   - Stores error messages per field name.
   - Example: `{ name: "This field is required." }`

Handling Input Changes:

Functions & Responsibilities:

1. `handleInputChange(name, value)`: Updates the value of the given field in the `formValues` state.
2. `handleBlur(name, errorMessage)`: Updates the `formErrors` state with the given error for the specified field when it loses focus.
3. `handleSubmit(e)`: Prevents default form behavior. Validates all fields in the form.
   - Loops through `formConfig` to validate each field using `validateField`.
   - If all validations pass:
     - ➔ Clears errors.
     - ➔ Shows a success alert: “Form Submitted Successfully!”.
   - If validation fails:
     - ➔ Populates `formErrors` with appropriate error messages.
4. `validateField(field, value)`: Checks the input value against the field's validation rules.
   - Required:
     - ➔ If `required` is `true` and value is empty
     - ➔ Error: **"This field is required."**
   - Email:
     - ➔ If `email` is `true` and email is invalid
     - ➔ Error: **"Please enter a valid email address."**
   - Number:

- ➔ If `number` is `true` and value is not a number
- ➔ Error: **"Please enter a valid number."**

- Min Length:
  - ➔ If value is shorter than `minLength`
  - ➔ Error: **"Minimum length is X characters."**
- Max Length:
  - ➔ If value is longer than `maxLength`
  - ➔ Error: **"Maximum length is X characters."**
- No Error:
  - ➔ Returns empty string if all validations pass.

HTML Structure:

1. Use a `<form>` element to wrap the entire form.
2. Loop through the `formConfig` array.
3. For each field:
   - Render a `DynamicInput` component.
   - Pass props:

     · `inputConfig` – configuration for the current input

     · `value` – current value from `formValues`

     · `onChange` – triggers `handleInputChange`

     · `onBlur` – triggers `handleBlur` with error message

     · `error` – validation message from `formErrors`
4. At the bottom, include a `<button>` to submit the form.
   - Label: “Submit”
   - Type: submit

Expected Behavior

1. On Initial Render:
   - All fields are shown based on the `formConfig`.
   - No error messages are displayed initially.
2. On Input Blur:
   - The specific input is validated using the `handleBlur` logic.
   - Error is displayed under the input if validation fails.
3. On Submit:
   - All fields are validated using `validateField`.

- Error messages are shown for invalid fields.
- If all validations pass:
    - ➔ Show alert: **“Form Submitted Successfully!”**

DynamicInput Component

Acceptance criteria:

1. Render a labeled input or dropdown based on the `type` in config.
2. Support input types:
    - `text`
    - `email`
    - `number`
    - `dropdown`
3. Validate the input on `blur` and return error message if invalid.
4. Show error message if one is provided via props.

Props Received:

1. `inputConfig` – Configuration for this input (label, name, type, options, validation rules).
2. `value` – Current value of the field.
3. `onChange(value)` – Callback for when the input value changes.
4. `onBlur(name, error)` – Callback triggered on blur with validation result.
5. `error` – The current error message for this field.

Functions & Responsibilities:

1. handleValidation(value):
    - Runs through all validation rules defined in `inputConfig.validation`.
    - Returns the appropriate error message if any rule fails.
2. handleBlur():
    - When an input loses focus, runs validation and passes the error to `onBlur`.

HTML Structure:

1. Use a top-level `<div>` to wrap each input field and its label/error.
2. Add a `<label>` with the text from `inputConfig.label`.
3. If the field type is `dropdown`:
    - Render a `<select>` element.
    - Populate options from `inputConfig.options`.
4. Otherwise:
    - Render an `<input>` of the specified type.
5. Show an error message below the field if `error` exists:
    - Use a `<span>` element with an appropriate class for styling.

Dynamic Behavior:

1. User input updates the parent form state using `onChange`.
2. On blur, the field is validated:
    - If invalid, error is shown.
    - If valid, error is cleared.
3. Dropdown options are generated dynamically.

Validation Utility (`validation.js`):

You have to use this file in DynamicForm component

Create below defined functions as:

1. validateRequired(value):
    - Checks if the input is not empty (after trimming whitespace).
    - Returns `true` if the value is present.
2. validateEmail(value):
    - Validates if the input follows a standard email pattern.
    - Returns `true` if valid.
3. validateNumber(value):
    - Checks if the input consists only of numeric characters.
    - Returns `true` if valid.
4. validateMinLength(value, minLength):
    - Checks if the value length is at least `minLength`.
    - Returns `true` if valid.
5. validateMaxLength(value, maxLength):

		● Checks if the value length is no more than `maxLength`. ● Returns `true` if valid. **** Kindly refer to the screenshots for any clarifications. ****

5 CONSTRAINTS

1. You should be able to press the “TAB” key and “SHIFT + TAB” to navigate from top field to bottom field and vice-versa.

6 MANDATORY ASSESSMENT GUIDELINES

1. **All actions like build, compile, running application, running test cases will be through Command Terminal.**
2. **To open the command terminal the test takers, need to go to**
   Application menu (Three horizontal lines at left top) -> Terminal ->New Terminal.
3. **This editor Auto Saves the code.**
4. **These are time bound assessments the timer would stop if you logout and while logging in back using the same credentials the timer would resume from the same time it was stopped from the previous logout.**
5. **This is a web-based application, to run the application on a browser, use the internal browser in the workspace. Click on the second last option on the left panel of IDE, you can find Browser Preview, where you can launch the application.**
   Note: The application will not run in the local browser
6. **You can follow series of command to setup React environment once you are in your project-name folder:**
   a. **npm install -> Will install all dependencies -> takes 10 to 15 min.**
   a. **npm run start -> To compile and deploy the project in browser. You can press <Ctrl> key while clicking on localhost:8080/8081 to open project in browser -> takes 2 to 3 min.**

    a. **npm run jest -> to run all test cases and see the summary. It takes 5 to 6 min to run.**
    b. **npm run test -> to run all test cases. It is mandatory to run this command before submission of workspace -> takes 5 to 6 min.**

7. **Once you are done with development and ready with submission, you may navigate to the previous tab and submit the workspace. It is mandatory to click on "Submit Assessment" after you are done with code.**