**Forms in React**

**What forms are in react?**

In React, forms are used to collect and manage user input.

They allow users to enter data, such as text, numbers, selections, and submit it to the application for processing.

React provides a flexible way to work with forms, allowing developers to handle form data and user interactions efficiently.

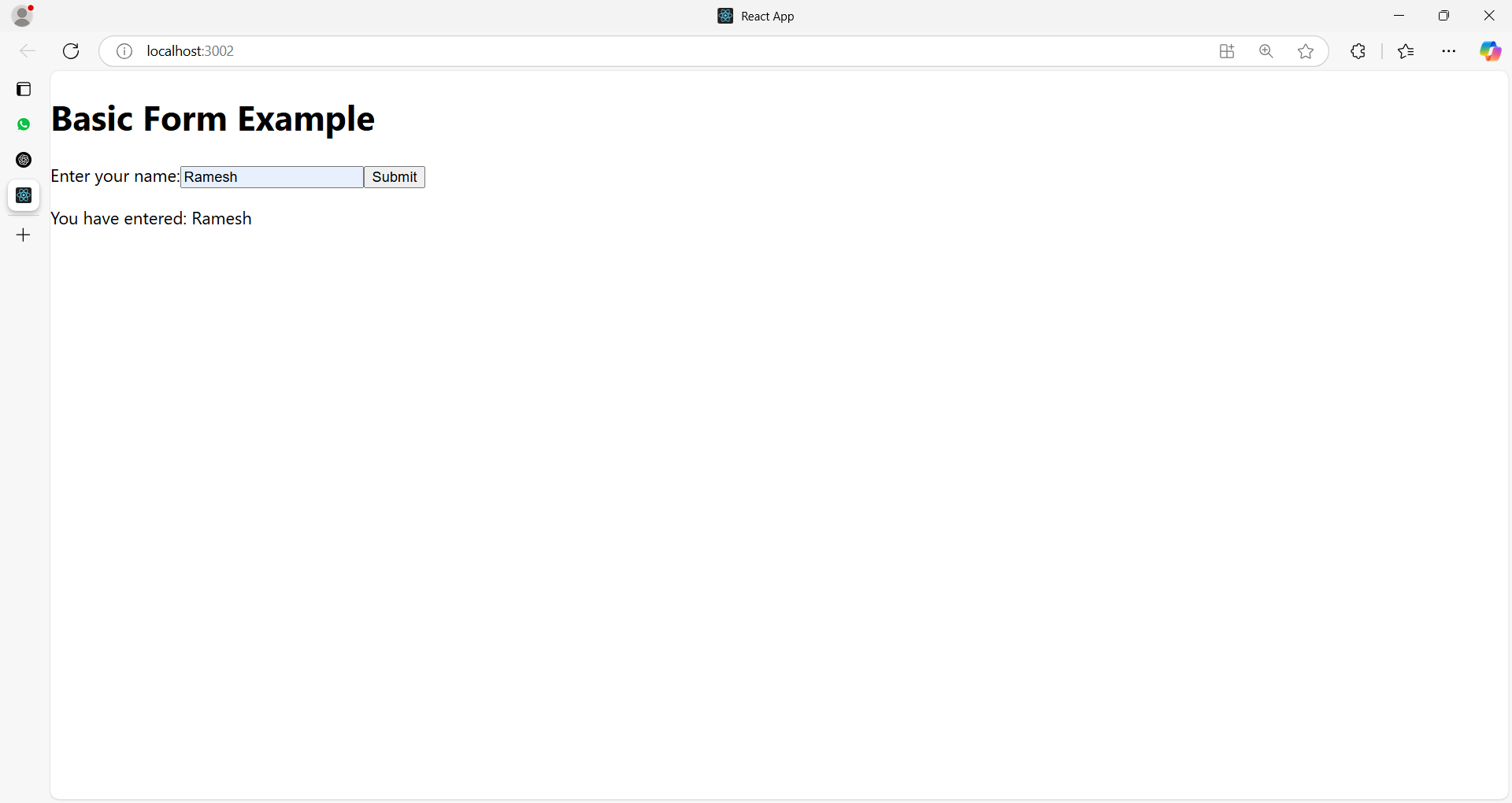
Forms in React typically consist of:

1. **Form Elements**:
   1. These include input fields, select boxes, radio buttons, checkboxes, and buttons.
   2. Each form element represents a piece of data that the user can input or select.
2. **State Management**:
   1. React components can maintain internal state to track the values of form elements.
   2. State is updated as the user interacts with the form, and changes are reflected in the UI.
3. **Event Handling**:
   1. React components use event handlers to respond to user interactions with form elements.
   2. For example, onChange event handlers are used to capture changes to input fields in real-time.
4. **Validation**:
   1. Form validation ensures that the data entered by the user meets certain criteria or constraints.
   2. React allows developers to implement custom validation logic and provide feedback to users when input is invalid.
5. **Submission Handling**:
   1. When the user submits the form, React components handle the form submission event.
   2. This typically involves sending the form data to a server or processing it within the application.

Forms are a fundamental part of most web applications, and React provides powerful tools for building interactive and dynamic forms that meet the needs of modern user interfaces.

**Example of programs**

1. **Basic Form with Input Field**:



**BasicForm.js**

import React, { useState } from 'react';

const BasicForm = () => {

const [inputValue, setInputValue] = useState('');

const [value, setValue] = useState('')

const handleChange = (e) => {

setInputValue(e.target.value);

};

const handleSubmit = (e) => {

e.preventDefault();

    setValue("You have entered: " + inputValue)

console.log('Form submitted with value:', inputValue);

};

return (

<div>

<form onSubmit={handleSubmit}>

<label>

Enter your name:

<input type="text" value={inputValue} onChange={handleChange} />

</label>

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

</form>

 <p id="result">{value}</p>

</div>

);

};

export default BasicForm;

**App.js**

import React from 'react';

import BasicForm from './BasicForm';

const App = () => {

return (

<div>

<h1>Basic Form Example</h1>

<BasicForm />

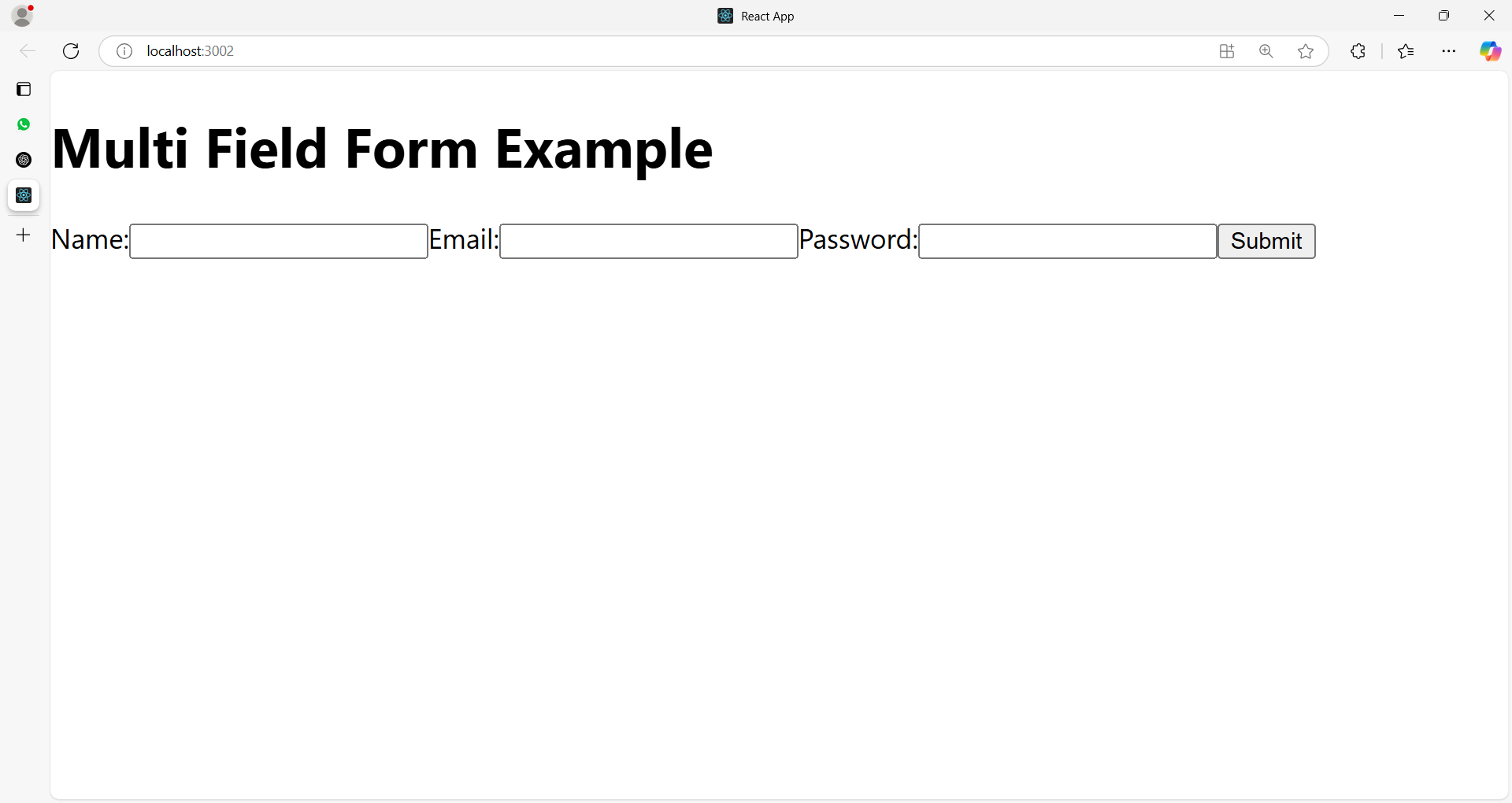
</div>

);

};

export default App;

1. **Form with Multiple Input Fields and Submit Handler**:



**MultiFieldForm.js**

import React, { useState } from 'react';

const MultiFieldForm = () => {

const [formData, setFormData] = useState({

name: '',

email: '',

password: '',

});

const handleChange = (e) => {

const { name, value } = e.target;

setFormData({ ...formData, [name]: value });

};

const handleSubmit = (e) => {

e.preventDefault();

console.log('Form submitted with data:', formData);

};

return (

<form onSubmit={handleSubmit}>

<label>

Name:

<input type="text" name="name" value={formData.name} onChange={handleChange} />

</label>

<label>

Email:

<input type="email" name="email" value={formData.email} onChange={handleChange} />

</label>

<label>

Password:

<input type="password" name="password" value={formData.password} onChange={handleChange} />

</label>

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

</form>

);

};

export default MultiFieldForm;

**App.js**

import React from 'react';

import MultiFieldForm from './MultiFieldForm';

const App = () => {

return (

<div>

<h1>Multi-field Form Example</h1>

<MultiFieldForm />

</div>

);

};

export default App;

1. **Form with Select Box and Textarea**:

**SelectForm.js**

import React, { useState } from 'react';

const SelectForm = () => {

const [selectedOption, setSelectedOption] = useState('');

const [textareaValue, setTextareaValue] = useState('');

const handleSelectChange = (e) => {

setSelectedOption(e.target.value);

};

const handleTextareaChange = (e) => {

setTextareaValue(e.target.value);

};

const handleSubmit = (e) => {

e.preventDefault();

console.log('Form submitted with data:', { selectedOption, textareaValue });

};

return (

<form onSubmit={handleSubmit}>

<label>

Choose an option:

<select value={selectedOption} onChange={handleSelectChange}>

<option value="">Select</option>

<option value="option1">Option 1</option>

<option value="option2">Option 2</option>

<option value="option3">Option 3</option>

</select>

</label>

<label>

Enter your message:

<textarea value={textareaValue} onChange={handleTextareaChange} />

</label>

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

</form>

);

};

export default SelectForm;

**App.js**

import React from 'react';

import SelectForm from './SelectForm';

const App = () => {

return (

<div>

<h1>Select Form Example</h1>

<SelectForm />

</div>

);

};

export default App;

**4. Registration Form**

**RegistrationForm.js**

// RegistrationForm.js

import React, { useState } from 'react';

import './RegistrationForm.css';

const RegistrationForm = () => {

  const [formData, setFormData] = useState({

    firstName: '',

    lastName: '',

    email: '',

    password: '',

    confirmPassword: '',

    age: '',

    gender: '',

    termsAgreed: false,

  });

  const [errors, setErrors] = useState({});

  const handleChange = (e) => {

    const { name, value, type, checked } = e.target;

    const val = type === 'checkbox' ? checked : value;

    setFormData({ ...formData, [name]: val });

  };

  const handleSubmit = (e) => {

    e.preventDefault();

    const validationErrors = validateForm(formData);

    if (Object.keys(validationErrors).length === 0) {

      // Form is valid, proceed with submission

      console.log('Form submitted with data:', formData);

    } else {

      // Form is invalid, display errors

      setErrors(validationErrors);

    }

  };

  const validateForm = (data) => {

    const errors = {};

    // Validate each field

    if (!data.firstName.trim()) {

      errors.firstName = 'First name is required';

    }

    if (!data.lastName.trim()) {

      errors.lastName = 'Last name is required';

    }

    if (!data.email.trim()) {

      errors.email = 'Email is required';

    } else if (!isValidEmail(data.email)) {

      errors.email = 'Invalid email address';

    }

    if (!data.password.trim()) {

      errors.password = 'Password is required';

    } else if (data.password.length < 8) {

      errors.password = 'Password must be at least 8 characters long';

    }

    if (data.password !== data.confirmPassword) {

      errors.confirmPassword = 'Passwords do not match';

    }

    if (!data.age.trim()) {

      errors.age = 'Age is required';

    } else if (isNaN(data.age) || parseInt(data.age) <= 0) {

      errors.age = 'Invalid age';

    }

    if (!data.gender.trim()) {

      errors.gender = 'Gender is required';

    }

    if (!data.termsAgreed) {

      errors.termsAgreed = 'You must agree to the terms';

    }

    return errors;

  };

  const isValidEmail = (email) => {

    // Check if the email contains an '@' character

    if (email.indexOf('@') === -1) {

      return false;

    }

    // Split the email at the '@' character

    const parts = email.split('@');

    // Check if there are exactly two parts

    if (parts.length !== 2) {

      return false;

    }

    // Check if the local part (before '@') is not empty

    if (parts[0].length === 0) {

      return false;

    }

    // Check if the domain part (after '@') contains a dot

    if (parts[1].indexOf('.') === -1) {

      return false;

    }

    // Check if the domain part is not empty

    if (parts[1].length === 0) {

      return false;

    }

    // If all checks pass, the email is considered valid

    return true;

  };

  return (

    <div className="form-container">

      <h2>Registration Form</h2>

      <form onSubmit={handleSubmit}>

        <div>

          <label>First Name:</label>

          <input type="text" name="firstName" value={formData.firstName} onChange={handleChange} />

          {errors.firstName && <div className="error">{errors.firstName}</div>}

        </div>

        <div>

          <label>Last Name:</label>

          <input type="text" name="lastName" value={formData.lastName} onChange={handleChange} />

          {errors.lastName && <div className="error">{errors.lastName}</div>}

        </div>

        <div>

          <label>Email:</label>

          <input type="email" name="email" value={formData.email} onChange={handleChange} />

          {errors.email && <div className="error">{errors.email}</div>}

        </div>

        <div>

          <label>Password:</label>

          <input type="password" name="password" value={formData.password} onChange={handleChange} />

          {errors.password && <div className="error">{errors.password}</div>}

        </div>

        <div>

          <label>Confirm Password:</label>

          <input type="password" name="confirmPassword" value={formData.confirmPassword} onChange={handleChange} />

          {errors.confirmPassword && <div className="error">{errors.confirmPassword}</div>}

        </div>

        <div>

          <label>Age:</label>

          <input type="number" name="age" value={formData.age} onChange={handleChange} />

          {errors.age && <div className="error">{errors.age}</div>}

        </div>

        <div>

          <label>Gender:</label>

          <select name="gender" value={formData.gender} onChange={handleChange}>

            <option value="">Select</option>

            <option value="male">Male</option>

            <option value="female">Female</option>

            <option value="other">Other</option>

          </select>

          {errors.gender && <div className="error">{errors.gender}</div>}

        </div>

        <div>

          <input type="checkbox" name="termsAgreed" checked={formData.termsAgreed} onChange={handleChange} />

          <label htmlFor="termsAgreed">I agree to the terms</label>

          {errors.termsAgreed && <div className="error">{errors.termsAgreed}</div>}

        </div>

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

      </form>

    </div>

  );

};

export default RegistrationForm;

**[**

The line **Object.keys(validationErrors).length** is used to **count the number of properties (keys) in the validationErrors object**.

**Breaking it Down**

1. Object.keys(validationErrors):
   * This returns an array of all the keys (property names) present in the validationErrors object.
2. .length:
   * This gets the total number of keys in the returned array.

**Example 1: Checking if an Object Has Errors**

const validationErrors = {

name: "Name is required",

email: "Invalid email format",

};

console.log(Object.keys(validationErrors)); // ["name", "email"]

console.log(Object.keys(validationErrors).length); // 2

Since there are **two properties** (name and email), the length is 2.

**Example 2: Checking If an Object is Empty**

If validationErrors is empty:

const validationErrors = {};

console.log(Object.keys(validationErrors)); // []

console.log(Object.keys(validationErrors).length); // 0

Since there are **no keys**, the length is 0.

**Use Case in Validation**

You can use this to check if there are any validation errors before submitting a form:

if (Object.keys(validationErrors).length > 0) {

console.log("Form has errors. Please fix them.");

} else {

console.log("Form is valid. Proceeding with submission.");

}

**Conclusion**

* Object.keys(obj).length **counts** how many properties exist in an object.
* It is **commonly used for validation**, checking if an object is empty, or tracking dynamic data

**]**

**RegistrationForm.css**

/\* RegistrationForm.css \*/

.form-container {

max-width: 600px;

margin: 0 auto;

padding: 20px;

border: 1px solid #ccc;

border-radius: 5px;

}

.form-container h2 {

margin-top: 0;

margin-bottom: 20px;

}

.form-container label {

display: block;

margin-bottom: 5px;

}

.form-container input[type='text'],

.form-container input[type='email'],

.form-container input[type='password'],

.form-container input[type='number'],

.form-container select,

.form-container textarea {

width: 100%;

padding: 8px;

margin-bottom: 10px;

}

.form-container .error {

color: red;

font-size: 12px;

margin-top: 5px;

}

.form-container button {

background-color: #007bff;

color: #fff;

border: none;

padding: 10px 20px;

cursor: pointer;

}

.form-container button:hover {

background-color: #0056b3;

}

.form-container input[type='checkbox'] {

margin-right: 10px;

}

**App.js**

// App.js

import React from 'react';

import RegistrationForm from './RegistrationForm';

const App = () => {

return (

<div>

<RegistrationForm />

</div>

);

};

export default App;

**Explanation**:

1. **Component Definition**: The **RegistrationForm** component is defined using functional components and React hooks to manage state.
2. **State Management**: State variables **formData** and **errors** are initialized using the **useState** hook to manage form data and validation errors, respectively.
3. **Event Handlers**: Event handlers like **handleChange** and **handleSubmit** are defined to handle form input changes and submissions, respectively.
4. **Form Validation**: The **validateForm** function is defined to validate form data based on certain rules.
5. **Render JSX**: The JSX code within the **return** statement renders the form elements, including input fields, error messages, and a submit button.
6. **CSS Styling**: The CSS file **RegistrationForm.css** contains styles for the registration form, including container styling, input fields, labels, buttons, etc.
7. **Component Export**: The **RegistrationForm** component is exported for use in other components.
8. **App Component**: In **App.js**, the **App** component renders the **RegistrationForm** component, which will be displayed in the application.
9. **CSS Import**: The CSS file is imported into the component file to apply styles to the form elements.

**Detailed Explanation of the code:**

**1.**

**import React, { useState } from 'react';**

**import './RegistrationForm.css';**

This lines of code imports the React library and the **useState** hook from React, as well as the CSS file named **RegistrationForm.css**. Let's break it down:

1. **import React from 'react';**: This imports the core React library, which is required for creating React components and using JSX syntax.
2. **{ useState }**: This syntax uses destructuring to import the **useState** hook from the React library. The **useState** hook is a function that allows functional components to manage state.
3. **from 'react';**: This specifies that the **useState** hook is being imported from the 'react' module.
4. **import './RegistrationForm.css';**: This line imports the CSS file named **RegistrationForm.css**. The **./** indicates that the file is located in the same directory as the current component file. The imported CSS styles will be applied to the component's elements when they are rendered.

In summary, this line imports React, the **useState** hook, and a CSS file into the current component file, allowing the component to utilize React functionality and apply CSS styles.

**2.**

**const RegistrationForm = () => {**

This line of code defines a functional component named **RegistrationForm**. Let's break it down:

1. **const**: This keyword is used to declare a constant variable in JavaScript. In this case, it's used to declare a constant variable named **RegistrationForm**.
2. **RegistrationForm**: This is the name of the constant variable and also the name of the functional component.
3. **()**: This set of parentheses indicates that **RegistrationForm** is a function.
4. **=>**: This arrow function syntax is used to define the function body.
5. **{}**: These curly braces contain the function body, which is the code that will be executed when the **RegistrationForm** component is rendered.

In summary, this line defines a functional component named **RegistrationForm** using arrow function syntax, which will be used to render a form for user registration.

**3.**

**const [formData, setFormData] = useState({**

**firstName: '',**

**lastName: '',**

**email: '',**

**password: '',**

**confirmPassword: '',**

**age: '',**

**gender: '',**

**termsAgreed: false,**

**});**

This line of code uses the **useState** hook to declare a state variable named **formData** and a function named **setFormData** to update that state. Let's break it down:

1. **const**: This keyword is used to declare a constant variable in JavaScript.
2. **[formData, setFormData]**: This is array destructuring syntax. It declares two variables **formData** and **setFormData** and assigns them the values returned by the **useState** hook.
3. **useState**: This is a React hook that allows functional components to manage state. It takes one argument, which is the initial state value, and returns an array containing the current state value and a function to update that state.
4. **{}**: This curly braces define the initial state object. It contains properties for each form field (**firstName**, **lastName**, **email**, **password**, **confirmPassword**, **age**, **gender**, **termsAgreed**) initialized with empty strings or **false** for the **termsAgreed** property.

In summary, this line initializes a state variable **formData** with an object representing the initial state of the form fields and a function **setFormData** to update that state.

4.  
**const [errors, setErrors] = useState({});**

This line of code is similar to the previous one, but it declares a state variable **errors** and a function **setErrors** to manage errors that occur during form validation. Let's break it down:

1. **const**: This keyword is used to declare a constant variable in JavaScript.
2. **[errors, setErrors]**: This is array destructuring syntax. It declares two variables **errors** and **setErrors** and assigns them the values returned by the **useState** hook.
3. **useState**: This is a React hook that allows functional components to manage state. It takes one argument, which is the initial state value, and returns an array containing the current state value and a function to update that state.
4. **{}**: This curly braces define the initial state object for errors. Initially, it's an empty object because there are no errors.

In summary, this line initializes a state variable **errors** with an empty object representing the initial state of form validation errors and a function **setErrors** to update that state. As the form is validated, this state will be updated with error messages for each field.

**5.**

**const handleChange = (e) => {**

**const { name, value, type, checked } = e.target;**

**const val = type === 'checkbox' ? checked : value;**

**setFormData({ ...formData, [name]: val });**

**};**

This function **handleChange** is responsible for handling changes in form inputs. Let's break down what it does:

1. **(e) => { ... }**: This is an arrow function syntax in JavaScript, which takes an event **e** as its parameter.
2. **const { name, value, type, checked } = e.target;**: This line uses destructuring assignment to extract properties from the **target** object of the event **e**. The **target** property refers to the element that triggered the event, in this case, an input element.
   * **name**: The **name** attribute of the input element, which identifies the property of the form data being updated.
   * **value**: The current value of the input element.
   * **type**: The type of the input element.
   * **checked**: For checkboxes, it indicates whether the checkbox is checked or not.
3. **const val = type === 'checkbox' ? checked : value;**: This line determines the value to be assigned to the **val** variable based on the type of the input element. If the type is a checkbox, it uses the **checked** property; otherwise, it uses the **value** property.
4. **setFormData({ ...formData, [name]: val });**: This line updates the form data state using the **setFormData** function. It spreads the current form data (**formData**) into a new object and then updates the property specified by **name** with the new value **val**. This effectively updates the corresponding field in the form data.

In summary, this function updates the form data state whenever there is a change in any input field by extracting the field name and value from the event object and then updating the form data state accordingly.

6.

**const handleSubmit = (e) => {**

**e.preventDefault();**

**const validationErrors = validateForm(formData);**

**if (Object.keys(validationErrors).length === 0) {**

**// Form is valid, proceed with submission**

**console.log('Form submitted with data:', formData);**

**} else {**

**// Form is invalid, display errors**

**setErrors(validationErrors);**

**}**

**};**

This **handleSubmit** function is called when the form is submitted. Let's break down its functionality:

1. **(e) => { ... }**: This is an arrow function syntax in JavaScript, which takes an event **e** as its parameter.
2. **e.preventDefault();**: This line prevents the default behavior of form submission, which is to reload the page. By calling **preventDefault()**, the form data is submitted using JavaScript without causing a page refresh.
3. **const validationErrors = validateForm(formData);**: This line calls the **validateForm** function to check if there are any validation errors in the form data. It passes the current form data (**formData**) as an argument to the **validateForm** function.
4. **if (Object.keys(validationErrors).length === 0) { ... }**: This condition checks if there are no validation errors. It uses **Object.keys(validationErrors)** to get an array of all the keys (property names) in the **validationErrors** object. If the length of this array is **0**, it means there are no validation errors.
   * If there are no validation errors, the form is considered valid, and the code inside the **if** block is executed.
     + **console.log('Form submitted with data:', formData);**: This line logs the form data to the console, indicating that the form submission was successful.
   * If there are validation errors, the code inside the **else** block is executed.
     + **setErrors(validationErrors);**: This line updates the **errors** state with the validation errors received from the **validateForm** function. These errors will be displayed to the user to indicate what fields need to be corrected.

In summary, this function handles form submission by first preventing the default behavior, then validating the form data. If there are no validation errors, it proceeds with the submission by logging the form data. Otherwise, it updates the **errors** state with the validation errors to display them to the user.

7.

**const validateForm = (data) => {**

**const errors = {};**

**// Validate each field**

**if (!data.firstName.trim()) {**

**errors.firstName = 'First name is required';**

**}**

**if (!data.lastName.trim()) {**

**errors.lastName = 'Last name is required';**

**}**

**if (!data.email.trim()) {**

**errors.email = 'Email is required';**

**} else if (!isValidEmail(data.email)) {**

**errors.email = 'Invalid email address';**

**}**

**if (!data.password.trim()) {**

**errors.password = 'Password is required';**

**} else if (data.password.length < 8) {**

**errors.password = 'Password must be at least 8 characters long';**

**}**

**if (data.password !== data.confirmPassword) {**

**errors.confirmPassword = 'Passwords do not match';**

**}**

**if (!data.age.trim()) {**

**errors.age = 'Age is required';**

**} else if (isNaN(data.age) || parseInt(data.age) <= 0) {**

**errors.age = 'Invalid age';**

**}**

**if (!data.gender.trim()) {**

**errors.gender = 'Gender is required';**

**}**

**if (!data.termsAgreed) {**

**errors.termsAgreed = 'You must agree to the terms';**

**}**

**return errors;**

**};**

This **validateForm** function is responsible for validating the form data. Let's break down its functionality:

1. **const validateForm = (data) => { ... }**: This defines an arrow function named **validateForm**, which takes the form data (**data**) as its parameter.
2. **const errors = {};**: This initializes an empty object named **errors** to store any validation errors encountered during the validation process.
3. Validation Logic:
   * For each field in the form data, it checks if the value meets the validation criteria.
   * If a field fails validation, it adds an entry to the **errors** object with the field name as the key and the error message as the value.
   * **if (!data.firstName.trim()) { ... }**: This checks if the **firstName** field is empty or contains only whitespace characters. If it is empty, it adds an error message indicating that the first name is required.
   * **if (!data.lastName.trim()) { ... }**: Similar to the previous check, this validates the **lastName** field.
   * **if (!data.email.trim()) { ... }**: This checks if the **email** field is empty or contains only whitespace characters. If it is empty, it adds an error message indicating that the email is required. Otherwise, it validates the email format using the **isValidEmail** function and adds an error message if the email is invalid.
   * **if (!data.password.trim()) { ... }**: This checks if the **password** field is empty or contains only whitespace characters. If it is empty, it adds an error message indicating that the password is required. Otherwise, it checks if the password length is less than 8 characters and adds an error message if it is.
   * **if (data.password !== data.confirmPassword) { ... }**: This checks if the **password** and **confirmPassword** fields match. If they don't match, it adds an error message indicating that the passwords do not match.
   * **if (!data.age.trim()) { ... }**: This checks if the **age** field is empty or contains only whitespace characters. If it is empty, it adds an error message indicating that the age is required. Otherwise, it checks if the age is a valid number and greater than 0.
   * **if (!data.gender.trim()) { ... }**: This checks if the **gender** field is empty or contains only whitespace characters. If it is empty, it adds an error message indicating that the gender is required.
   * **if (!data.termsAgreed) { ... }**: This checks if the **termsAgreed** field is not checked (false). If it is not checked, it adds an error message indicating that the user must agree to the terms.
4. **return errors;**: This returns the **errors** object containing any validation errors encountered during the validation process.

In summary, this function iterates over each field in the form data and validates its value based on specific criteria. If any field fails validation, an entry is added to the **errors** object with the corresponding error message. Finally, it returns the **errors** object containing any validation errors.

8.

**const isValidEmail = (email) => {**

**// Check if the email contains an '@' character**

**if (email.indexOf('@') === -1) {**

**return false;**

**}**

**// Split the email at the '@' character**

**const parts = email.split('@');**

**// Check if there are exactly two parts**

**if (parts.length !== 2) {**

**return false;**

**}**

**// Check if the local part (before '@') is not empty**

**if (parts[0].length === 0) {**

**return false;**

**}**

**// Check if the domain part (after '@') contains a dot**

**if (parts[1].indexOf('.') === -1) {**

**return false;**

**}**

**// Check if the domain part is not empty**

**if (parts[1].length === 0) {**

**return false;**

**}**

**// If all checks pass, the email is considered valid**

**return true;**

**};**

This **isValidEmail** function is used to validate email addresses based on certain criteria. Let's understand how it works:

1. **const isValidEmail = (email) => { ... }**: This defines an arrow function named **isValidEmail**, which takes an email address (**email**) as its parameter.
2. Email Validation Logic:
   * **if (email.indexOf('@') === -1) { ... }**: This checks if the email address contains an '@' character. If not found, it returns **false**, indicating that the email is invalid.
   * **const parts = email.split('@');**: This splits the email address into two parts, before and after the '@' character, creating an array named **parts**.
   * **if (parts.length !== 2) { ... }**: This checks if the email address has exactly two parts after splitting at the '@' character. If not, it returns **false**, indicating that the email is invalid.
   * **if (parts[0].length === 0) { ... }**: This checks if the local part (before the '@' character) is not empty. If it is empty, it returns **false**, indicating that the email is invalid.
   * **if (parts[1].indexOf('.') === -1) { ... }**: This checks if the domain part (after the '@' character) contains a dot ('.'). If not found, it returns **false**, indicating that the email is invalid.
   * **if (parts[1].length === 0) { ... }**: This checks if the domain part is not empty. If it is empty, it returns **false**, indicating that the email is invalid.
   * If all the above checks pass, it means that the email address has passed all validation criteria, and the function returns **true**, indicating that the email is considered valid.

In summary, this function checks various aspects of an email address, including the presence of an '@' character, the structure of the email address (local part and domain part), and the presence of a dot ('.') in the domain part. If all validation checks pass, it returns **true**, indicating that the email address is valid; otherwise, it returns **false**.

9.

**return (**

**<div className="form-container">**

1. **Return Statement**: The return keyword is used in a React functional component to specify what should be rendered to the screen. It indicates that the following JSX will be returned as the component's output.
2. **Opening <div> Element**: The <div> tag creates a generic container element in HTML. It is used to group and style other elements, and in this case, it contains the form and its associated elements.
3. **Class Attribute**: The className="form-container" attribute applies a CSS class named form-container to the <div>. This class can be used to apply specific styling rules to the container element.
4. **Container Purpose**: The <div className="form-container"> is likely used to organize and style the form elements, such as inputs, labels, and buttons, within a distinct section of the page.
5. **JSX Syntax**: The parentheses ( and ) surrounding the <div> tag are used to enclose JSX expressions, allowing for multiline rendering. This syntax is necessary in React components to ensure proper rendering and readability.

**<h2>Registration Form</h2>**

1. **Header Element**: The <h2> tag is an HTML header element. It defines a heading of the second level, which is typically used for sub-sections of a page.
2. **Text Content**: The text "Registration Form" between the opening and closing <h2> tags is the heading content. It provides a title or label for the section of the page where the form is located.
3. **Semantic Meaning**: Using an <h2> tag helps convey the hierarchical structure of the content. It indicates that this is a major heading for a subsection under a larger section, providing context for the form's purpose.
4. **Styling**: By default, headings like <h2> are rendered with a larger font size and bold text compared to normal paragraphs. This visual prominence helps users easily identify important sections of the page.
5. **Accessibility**: <h2> tags improve accessibility for users who rely on screen readers. They help these users navigate the content more effectively by defining the structure and sections of the page.

**<form onSubmit={handleSubmit}>**

1. **Form Element**: The <form> tag defines the start of an HTML form element. It groups together form controls such as input fields, checkboxes, and buttons, which are used to collect user input.
2. **onSubmit Attribute**: The onSubmit={handleSubmit} attribute specifies an event handler function handleSubmit that is triggered when the form is submitted. This function is responsible for processing or validating the form data.
3. **Form Submission**: When the user submits the form (typically by clicking a submit button or pressing Enter in a text field), the handleSubmit function will be called. This allows for custom logic, such as form validation, data handling, or making API calls.
4. **Prevent Default Behavior**: Often, inside the handleSubmit function, event.preventDefault() is used to prevent the default form submission behavior, which would normally cause the page to reload or navigate away. This allows handling the form submission within the client-side JavaScript.
5. **Integration with Form Controls**: All form controls (inputs, selects, buttons) enclosed within the <form> tags are part of this form and will be included in the submission process. The handleSubmit function will process the data from these controls when the form is submitted.

**<div>**

**<label>First Name:</label>**

**<input type="text" name="firstName" value={formData.firstName} onChange={handleChange} />**

**{errors.firstName && <div className="error">{errors.firstName}</div>}**

**</div>**

1. A div container holds the form elements related to the first name.
2. **<label>:** A label for the input field, indicating it's for the first name.
3. **<input type="text">**: An input field for text input with the name attribute set to **firstName**.
4. **value={formData.firstName}:** Binds the input's value to **formData.firstName**, which holds the current state for the first name.
5. **onChange={handleChange}:** Calls **handleChange** when the input value changes to update the state.
6. **{errors.firstName && <div className="error">{errors.firstName}</div>}:** Conditionally renders an error message if **errors**.**firstName** exists.

**<div>**

**<label>Last Name:</label>**

**<input type="text" name="lastName" value={formData.lastName} onChange={handleChange} />**

**{errors.lastName && <div className="error">{errors.lastName}</div>}**

**</div>**

1. **Container div**: The code starts with a <div> element, which acts as a container to group the elements related to the "Last Name" input field.
2. **Label Element**: The <label> tag is used to create a label for the input field. It displays the text "Last Name:" to inform the user what the corresponding input field is for.
3. **Input Field**: The <input type="text"> element is used to create a text input field where users can enter their last name. The name="lastName" attribute links the input field to the form's formData state for easy identification.
4. **State Binding**: The value={formData.lastName} binds the input field's value to the lastName property of the formData state object. This ensures that the input field displays the current value of formData.lastName.
5. **Error Handling**: The expression {errors.lastName && <div className="error">{errors.lastName}</div>} conditionally renders an error message if the errors.lastName property is set. If there is an error, it will be displayed inside a <div> element with the class error.

**<div>**

**<label>Email:</label>**

**<input type="email" name="email" value={formData.email} onChange={handleChange} />**

**{errors.email && <div className="error">{errors.email}</div>}**

**</div>**

1. **Container div**: The <div> element acts as a container that groups the label, input field, and potential error message related to the email input.
2. **Label Element**: The <label> tag provides a label with the text "Email:" for the input field, making it clear that the input is for the user's email address.
3. **Email Input Field**: The <input type="email"> element creates an input field specifically for email addresses. The type="email" ensures that the browser validates the input as a proper email format.
4. **State Binding**: The value={formData.email} attribute binds the input field to the email property of the formData state object. This ensures that the input field reflects the current value stored in formData.email.
5. **Error Handling**: The {errors.email && <div className="error">{errors.email}</div>} conditionally renders an error message if errors.email is set. If an error exists, the message will be displayed in a <div> element with the class error.

**<div>**

**<label>Password:</label>**

**<input type="password" name="password" value={formData.password} onChange={handleChange} />**

**{errors.password && <div className="error">{errors.password}</div>}**

**</div>**

1. **Container div**: The <div> element serves as a container that groups together the label, password input field, and any potential error message related to the password.
2. **Label Element**: The <label> tag provides the text "Password:" next to the input field, indicating that the user should enter their password in this field.
3. **Password Input Field**: The <input type="password"> element creates a password input field. The type="password" attribute ensures that the text entered in the field is masked (shown as dots or asterisks) to keep it hidden for security.
4. **State Binding**: The value={formData.password} attribute binds the input field's value to the password property of the formData state object. This keeps the input field in sync with the current value of formData.password.
5. **Error Handling**: The {errors.password && <div className="error">{errors.password}</div>} conditionally renders an error message if errors.password is set. If there is an error related to the password, it will be displayed in a <div> element with the class error.

**<div>**

**<label>Confirm Password:</label>**

**<input type="password" name="confirmPassword" value={formData.confirmPassword} onChange={handleChange} />**

**{errors.confirmPassword && <div className="error">{errors.confirmPassword}</div>}**

**</div>**

1. **Container div**: The <div> element acts as a container, grouping the label, confirm password input field, and any related error message, keeping the form structure organized.
2. **Label Element**: The <label> tag displays the text "Confirm Password:" next to the input field, instructing the user to re-enter their password to confirm it.
3. **Password Input Field**: The <input type="password"> element creates a password input field. The type="password" attribute ensures that the entered text is hidden (masked with dots or asterisks), suitable for entering sensitive information like a password.
4. **State Binding**: The value={formData.confirmPassword} binds the input field to the confirmPassword property of the formData state object. This ensures that the input field shows the current value of formData.confirmPassword.
5. **Error Handling**: The {errors.confirmPassword && <div className="error">{errors.confirmPassword}</div>} conditionally renders an error message if errors.confirmPassword is set. If there’s an error with the confirm password input, it will be displayed inside a <div> element with the class error.

**<div>**

**<label>Age:</label>**

**<input type="number" name="age" value={formData.age} onChange={handleChange} />**

**{errors.age && <div className="error">{errors.age}</div>}**

**</div>**

1. **Container div**: The <div> element serves as a container to group the label, age input field, and any related error message, keeping the form structure organized.
2. **Label Element**: The <label> tag provides the text "Age:" next to the input field, informing the user that this field is meant for entering their age.
3. **Number Input Field**: The <input type="number"> element creates an input field specifically for numerical input. The type="number" ensures that only numbers can be entered, which is appropriate for capturing the user's age.
4. **State Binding**: The value={formData.age} binds the input field's value to the age property of the formData state object. This keeps the input field in sync with the current value of formData.age.
5. **Error Handling**: The {errors.age && <div className="error">{errors.age}</div>} conditionally renders an error message if errors.age is set. If there is an error related to the age input, the error message will be displayed inside a <div> element with the class error.

**<div>**

**<label>Gender:</label>**

**<select name="gender" value={formData.gender} onChange={handleChange}>**

**<option value="">Select</option>**

**<option value="male">Male</option>**

**<option value="female">Female</option>**

**<option value="other">Other</option>**

**</select>**

**{errors.gender && <div className="error">{errors.gender}</div>}**

**</div>**

1. **Container div**: The <div> element groups the label, dropdown (select), and any related error message, keeping the form elements organized.
2. **Label Element**: The <label> tag displays the text "Gender:" next to the dropdown, indicating that the user should select their gender.
3. **Dropdown (Select) Element**: The <select> element creates a dropdown menu that allows the user to choose their gender. The name="gender" attribute helps identify the field, and value={formData.gender} binds the selected value to the gender property of the formData state object.
4. **Options**: Inside the <select> element, there are three <option> elements. The first option, with value="", serves as a placeholder prompting the user to select a gender. The other two options provide specific values, "Male," "Female," and "Other," as choices.
5. **Error Handling**: The {errors.gender && <div className="error">{errors.gender}</div>} conditionally renders an error message if errors.gender is set. If an error related to the gender selection occurs, the message will be displayed inside a <div> element with the class error.

**<div>**

**<input type="checkbox" name="termsAgreed" checked={formData.termsAgreed} onChange={handleChange} />**

**<label htmlFor="termsAgreed">I agree to the terms</label>**

**{errors.termsAgreed && <div className="error">{errors.termsAgreed}</div>}**

**</div>**

1. **Checkbox Input**: The <input type="checkbox"> element creates a checkbox input field. The name="termsAgreed" attribute identifies this field as the agreement to the terms.
2. **State Binding**: The checked={formData.termsAgreed} attribute binds the checkbox state (checked or unchecked) to the termsAgreed property of the formData state object. If formData.termsAgreed is true, the checkbox will be checked; otherwise, it will be unchecked.
3. **Change Handler**: The onChange={handleChange} attribute calls the handleChange function whenever the user checks or unchecks the checkbox. This updates the formData.termsAgreed state based on the user's interaction.
4. **Label Element**: The <label> tag, associated with the checkbox input, displays the text "I agree to the terms" next to the checkbox. The htmlFor="termsAgreed" attribute links the label to the checkbox input, improving accessibility.
5. **Error Handling**: The {errors.termsAgreed && <div className="error">{errors.termsAgreed}</div>} conditionally renders an error message if errors.termsAgreed is set. If there is an error related to the terms agreement (e.g., the user hasn't checked the box), the error message will be displayed inside a <div> element with the class error.

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

1. **Button Element**: The <button> element creates a clickable button on the form. It serves as the action trigger for form submission.
2. **Button Text**: The text "Submit" between the opening and closing <button> tags is displayed on the button, indicating to the user that clicking this button will submit the form.
3. **Type Attribute**: The type="submit" attribute specifies that this button will submit the form data when clicked. This action triggers the form's onSubmit event handler, if defined.
4. **Default Behavior**: When the button is clicked, the browser's default behavior is to submit the form to the action URL or call the form's onSubmit event handler if handled within the component.
5. **Styling**: Although not directly specified here, the button can be styled using CSS classes or inline styles to enhance its appearance, if desired. Without any additional classes or styles, it will render as a default HTML button.

**</form>**

**</div>**

**);**

1. **Closing </form> Tag**: The </form> tag marks the end of the form element. This indicates that all input fields, buttons, and other form-related elements enclosed within the <form> tags are part of the same form.
2. **Form Submission Handling**: Upon form submission (triggered by the Submit button or pressing Enter in an input field), the onSubmit event handler defined in the opening <form> tag will be executed.
3. **Closing </div> Tag**: The </div> tag closes the div container that wraps the entire form. This div was opened with the <div className="form-container"> tag, helping to style and structure the form content within the page layout.
4. **Component Return Statement**: The entire structure, including the div and form elements, is part of the JSX returned by the component. This return statement ensures that the form is rendered on the page as part of the component’s UI.
5. **Component Structure**: This marks the end of the component’s JSX code. After this point, the closing parentheses ()) and curly braces (}) will be used to complete the function or class definition of the component.

This JSX code defines the structure of the registration form component:

* Each input field is associated with its corresponding state variable (**formData**) and onChange event handler (**handleChange**).
* Error messages are displayed conditionally based on the presence of errors in the **errors** state object.
* When the form is submitted, the **handleSubmit** function is called to validate the form data.
* The form is styled using CSS classes defined in the **RegistrationForm.css** file.

**Example 05**

Here's a comprehensive example of a React application that includes routes, form elements, JSON integration, and form validations.

**Project Structure**

product-app/

├── public/

│ └── index.html

├── src/

│ ├── components/

│ │ ├── AddProduct.js

│ │ ├── EditProduct.js

│ │ ├── ProductForm.js

│ │ ├── ProductList.js

│ │ └── ProductDetails.js

│ ├── pages/

│ │ ├── Home.js

│ │ └── NotFound.js

│ ├── styles/

│ │ ├── App.css

│ │ ├── Form.css

│ │ ├── ProductList.css

│ │ ├── ProductDetails.css

│ ├── App.js

│ ├── index.js

│ └── products.json

├── .gitignore

├── package.json

└── README.md

**Step-by-Step Implementation**

**1. Setting Up Routing**

**src/App.js**

import React from 'react';

import { BrowserRouter as Router, Route, Routes } from 'react-router-dom';

import Home from './pages/Home';

import AddProduct from './components/AddProduct';

import EditProduct from './components/EditProduct';

import ProductDetails from './components/ProductDetails';

import NotFound from './pages/NotFound';

function App() {

return (

<Router>

<Routes>

<Route path="/" element={<Home />} />

<Route path="/add-product" element={<AddProduct />} />

<Route path="/edit-product/:id" element={<EditProduct />} />

<Route path="/product/:id" element={<ProductDetails />} />

<Route path="\*" element={<NotFound />} />

</Routes>

</Router>

);

}

export default App;

**[**

The expression **product?.name || ''** is using two JavaScript/TypeScript operators:

**1️. Optional Chaining (?.)**

* The **?.** operator safely accesses the name property of product without throwing an error if product is null or undefined.
* If product is null or undefined, the entire expression **returns undefined** instead of causing an error.

**Example:**

let product = null;

console.log(product?.name); // undefined (instead of throwing an error)

**2️. Logical OR (||)**

* The **|| (OR) operator** provides a **fallback/default value** if the left-hand side is **falsy** (null, undefined, '', 0, false, etc.).
* If product?.name is undefined or an empty string, it will return '' (an empty string) as a default.

**Example:**

let product = { name: 'Laptop' };

console.log(product?.name || 'Default Name'); // Output: 'Laptop'

let product2 = null;

console.log(product2?.name || 'Default Name'); // Output: 'Default Name'

**Summary**

name: product?.name || ''

**product?.name** ensures we don’t get an error if product is null or undefined.  
**|| ''** ensures that if name is missing or empty, we use an empty string '' instead.

**]**

**2. Creating the Product Form Component**

**src/components/ProductForm.js**

import React, { useState } from 'react';

function ProductForm({ product, onSave }) {

const [formData, setFormData] = useState({

name: product?.name || '',

price: product?.price || '',

category: product?.category || 'Electronics',

stock: product?.stock || '',

brand: product?.brand || '',

model: product?.model || '',

weight: product?.weight || '',

color: product?.color || '',

size: product?.size || '',

description: product?.description || ''

});

const [errors, setErrors] = useState({});

const handleChange = (e) => {

setFormData({

...formData,

[e.target.name]: e.target.value

});

};

const validateForm = () => {

const newErrors = {};

if (!formData.name) newErrors.name = 'Product Name is required';

if (!formData.price) {

newErrors.price = 'Price is required';

} else if (isNaN(formData.price) || formData.price <= 0) {

newErrors.price = 'Price must be a positive number';

}

if (!formData.stock) {

newErrors.stock = 'Stock is required';

} else if (isNaN(formData.stock) || formData.stock < 0) {

newErrors.stock = 'Stock must be a non-negative number';

}

if (!formData.brand) newErrors.brand = 'Brand is required';

if (!formData.model) newErrors.model = 'Model is required';

if (!formData.weight) newErrors.weight = 'Weight is required';

if (!formData.color) newErrors.color = 'Color is required';

if (!formData.size) newErrors.size = 'Size is required';

if (!formData.description) newErrors.description = 'Description is required';

setErrors(newErrors);

return Object.keys(newErrors).length === 0;

};

const handleSubmit = (e) => {

e.preventDefault();

if (validateForm()) {

onSave(formData);

}

};

return (

<form onSubmit={handleSubmit}>

<fieldset>

<legend>Product Details</legend>

<div>

<label>Product Name:</label>

<input

type="text"

name="name"

value={formData.name}

onChange={handleChange}

/>

{errors.name && <p style={{ color: 'red' }}>{errors.name}</p>}

</div>

<div>

<label>Price:</label>

<input

type="number"

name="price"

value={formData.price}

onChange={handleChange}

/>

{errors.price && <p style={{ color: 'red' }}>{errors.price}</p>}

</div>

<div>

<label>Category:</label>

<select

name="category"

value={formData.category}

onChange={handleChange}

>

<option value="Electronics">Electronics</option>

<option value="Clothing">Clothing</option>

<option value="Books">Books</option>

<option value="Home Appliances">Home Appliances</option>

</select>

</div>

</fieldset>

<fieldset>

<legend>Product Specifications</legend>

<div>

<label>Stock:</label>

<input

type="number"

name="stock"

value={formData.stock}

onChange={handleChange}

/>

{errors.stock && <p style={{ color: 'red' }}>{errors.stock}</p>}

</div>

<div>

<label>Brand:</label>

<input

type="text"

name="brand"

value={formData.brand}

onChange={handleChange}

/>

{errors.brand && <p style={{ color: 'red' }}>{errors.brand}</p>}

</div>

<div>

<label>Model:</label>

<input

type="text"

name="model"

value={formData.model}

onChange={handleChange}

/>

{errors.model && <p style={{ color: 'red' }}>{errors.model}</p>}

</div>

<div>

<label>Weight:</label>

<input

type="text"

name="weight"

value={formData.weight}

onChange={handleChange}

/>

{errors.weight && <p style={{ color: 'red' }}>{errors.weight}</p>}

</div>

<div>

<label>Color:</label>

<input

type="text"

name="color"

value={formData.color}

onChange={handleChange}

/>

{errors.color && <p style={{ color: 'red' }}>{errors.color}</p>}

</div>

<div>

<label>Size:</label>

<input

type="text"

name="size"

value={formData.size}

onChange={handleChange}

/>

{errors.size && <p style={{ color: 'red' }}>{errors.size}</p>}

</div>

<div>

<label>Description:</label>

<textarea

name="description"

value={formData.description}

onChange={handleChange}

></textarea>

{errors.description && <p style={{ color: 'red' }}>{errors.description}</p>}

</div>

</fieldset>

<button type="submit">Save Product</button>

</form>

);

}

export default ProductForm;

**3. Adding Product Logic**

**src/components/AddProduct.js**

import React from 'react';

import ProductForm from './ProductForm';

import { useNavigate } from 'react-router-dom';

function AddProduct() {

const navigate = useNavigate();

const handleSave = (product) => {

const products = JSON.parse(localStorage.getItem('products')) || [];

product.id = new Date().getTime(); // Assign a unique ID based on timestamp

products.push(product);

localStorage.setItem('products', JSON.stringify(products));

navigate('/');

};

return (

<div>

<h2>Add New Product</h2>

<ProductForm onSave={handleSave} />

</div>

);

}

export default AddProduct;

**4. Editing Product Logic**

**src/components/EditProduct.js**

import React from 'react';

import ProductForm from './ProductForm';

import { useParams, useNavigate } from 'react-router-dom';

function EditProduct() {

const { id } = useParams();

const navigate = useNavigate();

const products = JSON.parse(localStorage.getItem('products')) || [];

const product = products.find((p) => p.id === parseInt(id));

const handleSave = (updatedProduct) => {

const updatedProducts = products.map((p) =>

p.id === parseInt(id) ? { ...updatedProduct, id: parseInt(id) } : p

);

localStorage.setItem('products', JSON.stringify(updatedProducts));

navigate('/');

};

return (

<div>

<h2>Edit Product</h2>

<ProductForm product={product} onSave={handleSave} />

</div>

);

}

export default EditProduct;

**5. Displaying the Product List**

**src/components/ProductList.js**

import React from 'react';

import { Link } from 'react-router-dom';

function ProductList() {

const products = JSON.parse(localStorage.getItem('products')) || [];

return (

<div>

<h2>Product List</h2>

<Link to="/add-product">Add New Product</Link>

<ul>

{products.map((product) => (

<li key={product.id}>

<Link to={`/product/${product.id}`}>{product.name}</Link>

<Link to={`/edit-product/${product.id}`}>Edit</Link>

</li>

))}

</ul>

</div>

);

}

export default ProductList;

**[**

**useParams() in React Router**

useParams() is a **React Router Hook** that allows you to access **URL parameters** from the current route.

**How useParams() Works**

1. It retrieves **dynamic segments** (parameters) from the URL.
2. It returns an **object** where each key corresponds to a URL parameter.

**Example: Using useParams() in a Route**

Consider the following route:

<Route path="/product/**:id**" element={<ProductDetail />} />

* Here, :id is a **route parameter**.
* If the URL is **/product/123**, id will be **123**.

Now, inside the ProductDetail component, you can access the id using useParams():

import { useParams } from 'react-router-dom';

function ProductDetail() {

const { id } = useParams(); // Get the 'id' from the URL

return (

<div>

<h2>Product ID: {id}</h2>

</div>

);

}

export default ProductDetail;

**Output for /product/123**

Product ID: 123

**More Examples**

**Multiple Parameters**

<Route path="/category/:categoryId/product/:productId" element={<ProductPage />} />

To access both parameters:

const { categoryId, productId } = useParams();

console.log(categoryId, productId);

If the URL is **/category/5/product/10**, the values will be:

categoryId = "5"

productId = "10"

**Key Points**

**Only works inside components** (because it's a hook).  
**Returns an object** with parameters as key-value pairs.  
**Can handle multiple parameters** in the URL.

**]**

**6. Displaying Product Details**

**src/components/ProductDetails.js**

import React from 'react';

import { useParams } from 'react-router-dom';

function ProductDetails() {

const { id } = useParams();

const products = JSON.parse(localStorage.getItem('products')) || [];

const product = products.find((p) => p.id === parseInt(id));

if (!product) {

return <div>Product not found</div>;

}

return (

<div>

<h2>Product Details</h2>

<p><strong>Name:</strong> {product.name}</p>

<p><strong>Price:</strong> ${product.price}</p>

<p><strong>Category:</strong> {product.category}</p>

<p><strong>Stock:</strong> {product.stock}</p>

<p><strong>Brand:</strong> {product.brand}</p>

<p><strong>Model:</strong> {product.model}</p>

<p><strong>Weight:</strong> {product.weight}</p>

<p><strong>Color:</strong> {product.color}</p>

<p><strong>Size:</strong> {product.size}</p>

<p><strong>Description:</strong> {product.description}</p>

</div>

);

}

export default ProductDetails;

**7. Creating the Home and Not Found Pages**

**src/pages/Home.js**

import React from 'react';

import ProductList from '../components/ProductList';

function Home() {

return (

<div>

<h1>Welcome to the Product Management App</h1>

<ProductList />

</div>

);

}

export default Home;

**src/pages/NotFound.js**

import React from 'react';

import { Link } from 'react-router-dom';

function NotFound() {

return (

<div>

<h2>Page Not Found</h2>

<Link to="/">Go to Home</Link>

</div>

);

}

export default NotFound;

**8. Sample JSON Data**

**src/products.json**

[

{

"id": 1,

"name": "Laptop",

"price": 999.99,

"category": "Electronics",

"stock": 5,

"brand": "Dell",

"model": "XPS 15",

"weight": "2 kg",

"color": "Silver",

"size": "15 inches",

"description": "A high-performance laptop for professionals."

},

{

"id": 2,

"name": "Smartphone",

"price": 699.99,

"category": "Electronics",

"stock": 10,

"brand": "Samsung",

"model": "Galaxy S21",

"weight": "200 g",

"color": "Black",

"size": "6 inches",

"description": "A premium smartphone with top-notch features."

}

]

**CSS3**

**1. Global Styles (App.css)**

body {

font-family: Arial, sans-serif;

background-color: #f4f4f4;

margin: 0;

padding: 0;

}

h2 {

text-align: center;

color: #333;

}

button {

background-color: #007bff;

color: white;

padding: 10px 15px;

border: none;

cursor: pointer;

font-size: 16px;

}

button:hover {

background-color: #0056b3;

}

**2. Form Styles (Form.css)**

form {

width: 60%;

margin: 20px auto;

background: white;

padding: 20px;

border-radius: 8px;

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

}

fieldset {

border: none;

margin-bottom: 20px;

}

legend {

font-size: 18px;

font-weight: bold;

color: #007bff;

}

label {

display: block;

margin-bottom: 5px;

font-weight: bold;

}

input,

select,

textarea {

width: 100%;

padding: 10px;

border: 1px solid #ccc;

border-radius: 5px;

font-size: 16px;

margin-bottom: 10px;

}

textarea {

resize: vertical;

}

p {

color: red;

font-size: 14px;

margin-top: -5px;

}

button {

display: block;

width: 100%;

margin-top: 10px;

}

**3. Product List Styles (ProductList.css)**

.product-list {

width: 80%;

margin: 20px auto;

display: flex;

flex-wrap: wrap;

gap: 20px;

}

.product-card {

background: white;

padding: 15px;

border-radius: 5px;

box-shadow: 0 0 5px rgba(0, 0, 0, 0.1);

flex: 1 1 calc(33.333% - 20px);

}

.product-card h3 {

color: #333;

}

.product-card p {

font-size: 14px;

color: #555;

}

.product-card button {

margin-top: 10px;

}

**4. Product Details Styles (ProductDetails.css)**

.product-details {

width: 60%;

margin: 30px auto;

background: white;

padding: 20px;

border-radius: 8px;

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

}

.product-details h2 {

color: #007bff;

}

.product-details p {

font-size: 16px;

color: #555;

}

.product-details button {

margin-top: 15px;

}

**Features Covered**

1. **Routing**: The app uses React Router to navigate between different pages like the home page, add/edit product forms, and product details.
2. **Form Handling**: The ProductForm component handles both adding and editing products.
3. **JSON Integration**: Data is loaded from localStorage, simulating interaction with a JSON-based API. You can extend this to interact with an actual API.
4. **Form Validation**: Form validation ensures that all required fields are filled out correctly before submission.
5. **Dynamic Rendering**: The product list and details pages dynamically render content based on the state.

**Running the Application**

1. **Install Dependencies**: Make sure you have react-router-dom installed.

npm install react-router-dom

1. **Start the Application**:

npm start

Visit http://localhost:3000 to see the application in action.

This example provides a solid foundation for a React application that includes routing, form handling, validation, and JSON data management.