08-form & form validation

**How to Build Forms in React**

Forms play an essential role in modern web applications. They enable users to share information, complete tasks and provide feedback.

Without forms, many of the tasks that we take for granted on the web, such as logging in, signing up, or making purchases, would not be possible.

As such, learning how to create effective and user-friendly forms is essential for developers looking to build engaging and interactive web applications.

With its extensive collection of built-in hooks, React provides several features and techniques for creating and managing forms, including state management, event handling, and form validation.

The purpose of this guide is to provide a comprehensive and in-depth look at creating forms in React.

**Getting Started...**

In React, there are two ways of handling form data:

* **Controlled Components:** In this approach, form data is handled by React through the use of hooks such as the useState hook.
* **Uncontrolled Components:** Form data is handled by the Document Object Model (DOM) rather than by React. The DOM maintains the state of form data and updates it based on user input.

To better understand the difference between controlled and uncontrolled components, consider there are two ways of riding a bike.

In the first approach, you let the bike take control. You sit on the bike and let it decide the direction and speed. You might try to make it go in a certain direction by leaning your body, but ultimately, the bike decides where to go.

This is similar to uncontrolled components in React. You place a form element in the component, and the DOM takes control of it. The DOM decides the state of the input element and updates it based on a user's input.

In the second approach, you take control of the bike. You hold the handlebars and pedal, and you decide where to go and how fast to ride.

You can easily slow down or speed up as needed.

This is similar to controlled components where a React component takes control of the form data, and maintains the state of form elements. The component decides when and how to update the state, and it re-renders itself based on the state changes.

In the upcoming sections, we will expound upon the distinction between controlled and uncontrolled components and provide practical examples to illustrate how each operates.

# ****Controlled Components in React****

In React, a controlled component is a component where form elements derive their value from a React state.

When a component is controlled, the value of form elements is stored in a state, and any changes made to the value are immediately reflected in the state.

To create a controlled component, you need to use the value prop to set the value of form elements and the onChange event to handle changes made to the value.

The value prop sets the initial value of a form element, while the onChange event is triggered whenever the value of a form element changes. Inside the onChange event, you need to update the state with the new value using a state update function.

Here's an example:

import {useState} from 'react';

export default function ControlledComponent() {

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

const handleChange = (event) => {

setInputValue(event.target.value);

};

return (

<form>

<label>Input Value:

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

</label>

<p>Input Value: {inputValue}</p>

</div>

)};

​

In this example:

The useState hook defines a state variable (inputValue) and a state update function (setInputValue).

The value prop sets the initial value of the input element to the value of inputValue.

Also, the onChange event handles changes made to the input value. The handleChange function updates the inputValue state with the new value of the input element, and the updated value is immediately reflected in the state and displayed on the screen.

As the user types into the input field, the handleChange function updates the state variable using the "setInputValue" function. The component is then re-rendered, and the input field's value attribute is updated to reflect the new value of inputValue.

The value of the input field and the text displayed below it are always in sync, making it a controlled component.

### **How to handle dropdowns and checkboxes in Controlled Components**

Just like with input elements, the value of a dropdown can be set by using the value prop in conjunction with the onChange event handler to update the state of the component.

For example, to handle a dropdown menu, you can define the initial value of the dropdown menu within the state of the component, then update the state when the value of the dropdown changes:

import { useState } from "react";

export default function Dropdown() {

const [selectedOption, setSelectedOption] = useState("option1");

const handleDropdownChange = (event) => {

setSelectedOption(event.target.value);

};

return (

<div>

<label>

Select an option:

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

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

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

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

</select>

</label>

<p>Selected option: {selectedOption}</p>

</div>

);

}

Similarly, you can handle checkboxes by setting the checked prop of the checkbox input element based on the state of a component, and then updating the state when a checkbox is clicked.

Here's an example:

import { useState } from "react";

function Checkbox() {

const [isChecked, setIsChecked] = useState(false);

const handleChange = (event) => {

setIsChecked(event.target.checked);

};

return (

<form>

<label htmlFor="color">

<input type="checkbox" name="color" checked={isChecked} onChange={handleChange}/>

Blue

</label>

{isChecked && <div>Blue is selected!</div>}

</form>

);

}

export default Checkbox;

In this example, we have defined a state variable isChecked to keep track of whether the checkbox is checked or not. When the checkbox is clicked, the handleChange function is called, and it updates the isChecked state variable to a new value (true or false.).

The isChecked variable controls the checked attribute of the checkbox input and conditionally renders a message indicating that the checkbox is selected.

### **How to handle multiple form fields**

When working with forms in React, it's common to have several form elements, such as text inputs, checkboxes, radio buttons, and others.

To manage the state of these form elements, you can define the values for the input fields as an object using a single state variable and update each respective state variable using the onChange event.

As an example, suppose you wish to create a form with the following fields:

* Text input for the user's name
* An email field for the user's email
* A textarea field for the user's message

Here's how you could handle these fields:

import { useState } from "react";

export default function Multiple() {

const [formData, setFormData] = useState({name: "",email: "",message: ""});

const handleChange = (event) => {

const { name, value } = event.target;

setFormData((prevFormData) => ({ ...prevFormData, [name]: value }));

};

const handleSubmit = (event) => {

event.preventDefault();

alert(`Name: ${formData.name}, Email: ${formData.email}, Message: ${formData.message}`);

};

return (

<form onSubmit={handleSubmit}>

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

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

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

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

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

<textarea id="message" name="message" value={formData.message} onChange={handleChange}/>

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

</form>

);

}

In the example code:

The useState hook defines a state object named formData that contains three properties: name, email, and message, each initialized to an empty string.

The handleChange function is called whenever a user types in one of the form fields. It extracts the name and value of the form field that has changed using the event.target object and then updates the formData state variable using the setFormData function.

The setFormData function uses the spread operator (...) to copy the previous formData object. Then it updates the value of the changed form field by setting its value prop with the new value.

By using an object to manage form data, we can easily keep track of the values of multiple form elements. This makes it easier to manage and manipulate the state of our form data, especially when dealing with complex forms with many form elements.