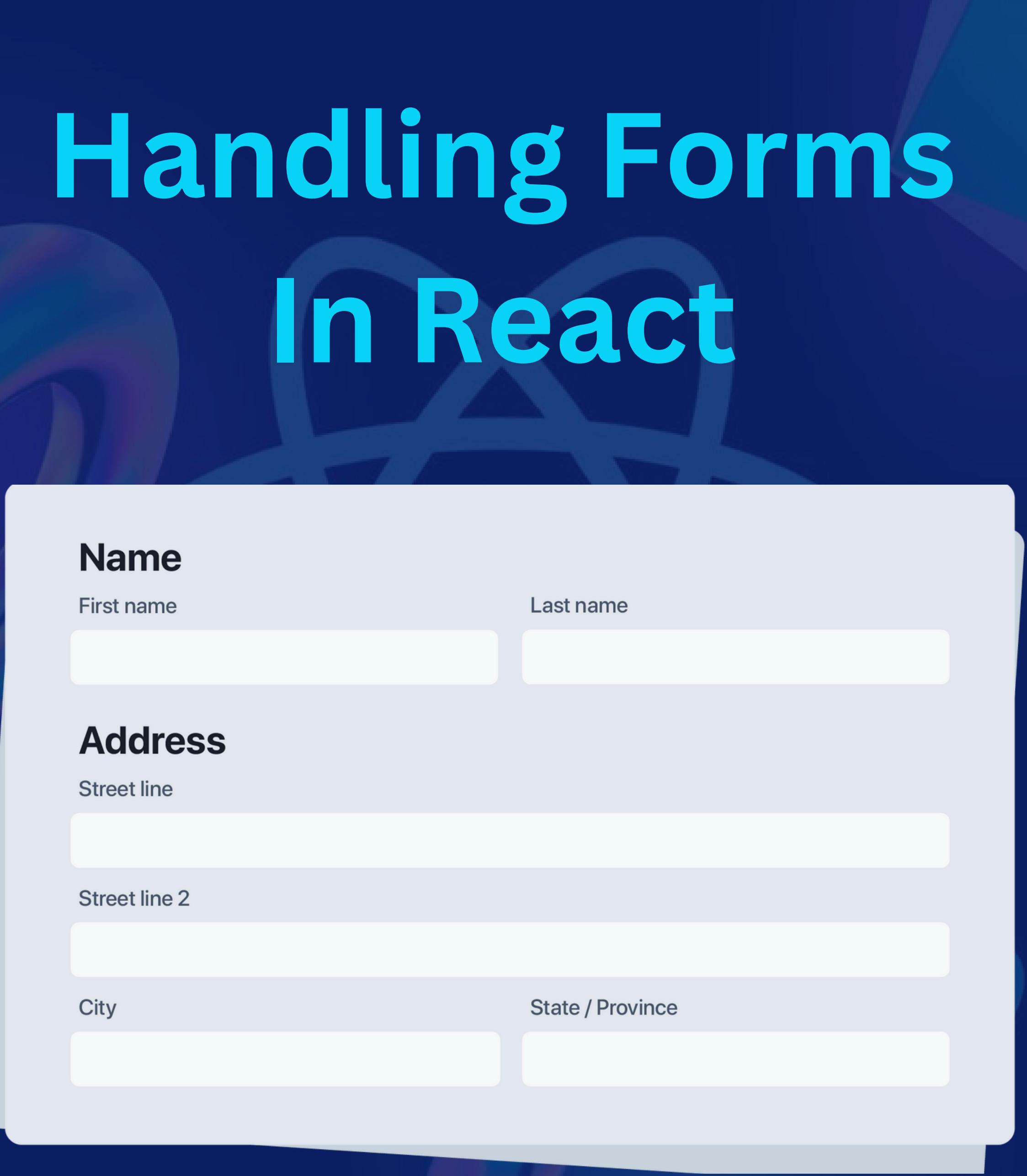
Day 12

Handling Forms In React



@oluwakemi Oluwadahunsi

Introduction

Handling forms in React.js is a crucial part of creating interactive web applications.

Forms allow users to input data that can be processed, validated, and submitted to a server or used within the application itself.

React offers a streamlined approach to form handling by utilizing components, state, and event management effectively.

@oluwakemi Oluwadahunsi

Understanding Controlled vs Uncontrolled Components

1. Controlled Components:

In React, form elements like **<input>**, **<textarea>**, and **<select>** are called **controlled components** when their value is controlled by the component's state.

The key idea is to bind the form data directly to the **state** of the component.

Any change in the **input** is immediately reflected in the component's state, giving you **full control** over the form values.

For Example:

@oluwakemi Oluwadahunsi

```jsx
import { useState } from 'react';

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

  const handleChange = (e) => {
    setInputValue(e.target.value);
  };

  const handleSubmit = (e) => {
    e.preventDefault();
    console.log('Form Submitted: ', inputValue);
  };

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="text"
        value={inputValue}
        onChange={handleChange}
        placeholder="Enter your name"
      />
      <button type="submit">Submit</button>
    </form>
  );
}

export default Form;
```

@oluwakemi Oluwadahunsi

In the example, the **inputValue** is stored in the **component's state**, and the **onChange** event updates it. When the form is submitted, you can access the value from the state, making controlled components **highly predictable**.

For Controlled Components:

- **React manages the form data:** In controlled components, the form elements (e.g., input, textarea, etc.) are controlled by React's state.

- **Single source of truth:** The component's value is updated through state using the setState or useState hooks, making React the single source of truth.

- **Immediate access to form data:** You can access the form data instantly because it's stored in React state.

@oluwakemi Oluwadahunsi

2. Uncontrolled Components:

Unlike controlled components, uncontrolled components store their state **internally within the DOM**, and you don't manage it with React's state. You can retrieve the values using **refs**.

```jsx
function UncontrolledComponent() {
  const inputRef = React.useRef(null);

  const handleSubmit = () => {
    console.log(inputRef.current.value);
  };

  return (
    <div>
      <input type="text" ref={inputRef} />
      <button onClick={handleSubmit}>Submit</button>
    </div>
  );
}
```

In this case, the **inputRef** is used to access the input's value when needed.

Uncontrolled components can be useful when dealing with **external libraries or legacy code**, but they provide **less control** over the form state compared to controlled components.

@oluwakemi Oluwadahunsi

For Uncontrolled Component:

- **DOM manages the form data:** In uncontrolled components, the form data is handled by the DOM itself, not React.

- **Refs to access form data:** You can access the form data only when you need it, using React refs to interact with the DOM.

- **Less React state management:** Since React doesn’t manage the input values, these components are simpler, but you have less control over the input’s state.

@oluwakemi Oluwadahunsi

Handling Multiple Inputs

When a form contains multiple inputs, it's important to manage the state effectively.

You can use a **single state objec**t to store the values of all inputs, updating each field dynamically based on the **input’s name attribute.**

For Example:

@oluwakemi Oluwadahunsi

```jsx
import { useState } from 'react';

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

  const handleChange = (e) => {
    const { name, value } = e.target;
    setFormData({
      ...formData,
      [name]: value
    });
  };

  const handleSubmit = (e) => {
    e.preventDefault();
    console.log('Form Submitted: ', formData);
  };

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="text"
        name="name"
        value={formData.name}
        onChange={handleChange}
        placeholder="Name"
      />
      <input
        type="email"
        name="email"
        value={formData.email}
        onChange={handleChange}
        placeholder="Email"
      />
      <textarea
        name="message"
        value={formData.message}
        onChange={handleChange}
        placeholder="Message"
      />
      <button type="submit">Submit</button>
    </form>
  );
}

export default MultiInputForm;
```

All input data is stored in a single state

- This **MultiInputForm** component handles multiple form inputs in React by using a **single state object (formData)** to store values for the **name**, **email**, and **message** fields.

- The **handleChange** function updates the respective field in the **formData** object dynamically using the **name attribute** from each input.

- When the form is submitted, the **handleSubmit** function prevents the default action.

By spreading the previous state and updating only the changed field, you can handle multiple form inputs effectively without creating separate state variables for each input.

@oluwakemi Oluwadahunsi

Handling File Uploads

Handling file uploads in React is a common requirement when working with forms, especially when dealing with **images**, **documents**, or other types of media.

React allows for handling file uploads, and you can manage file inputs by using the **onChange attribute** to capture the file and process it as needed.

Many finds this process a bit difficult to understand, but let's simplify this as much as possible:

Step 1: Set up File Input in Form:

In React, you can create a file input field using the **<input>** element with the **type="file"** attribute. This allows users to select files from their device.

@oluwakemi Oluwadahunsi

Step 2: Use State to Store Selected File:

When a user selects a file, you can store the file information in the component's state using **useState** for controlled components.

Step 3: Handle File Changes:

React will use a **handler function** to capture the file when the user selects it. This function reads the selected file from the input field and **updates** the state and it it is usually set as the value to the **onChange** attribute on the **<input>** element.

Step 4: Submit the File:

You can submit the file by sending it to an **API** using a tool like **FormData**, which allows you to send files along with other form data in a single request.

Bringing every step together in an example:

@oluwakemi Oluwadahunsi

```jsx
import { useState } from 'react';

function FileUploadForm() {
  const [file, setFile] = useState(null);

  const handleFileChange = (e) => {
    setFile(e.target.files[0]);
  };

  const handleSubmit = (e) => {
    e.preventDefault();

    const formData = new FormData();
    formData.append('file', file);

    // You can send formData to your backend using fetch or axios
    console.log('File to be uploaded:', file);
  };

  return (
    <form onSubmit={handleSubmit}>
      <input type="file" onChange={handleFileChange} />
      <button type="submit">Upload</button>
    </form>
  );
}

export default FileUploadForm;
```

step2: store selected file in a state

step3: handler function

step4: function to submit form

step1: set an input element with type attribute “file”

Here, the file input's **onChange** event updates the component state with the selected file, and the **handleSubmit** function can then handle the file (e.g., upload it to a server). @oluwakemi Oluwadahunsi

React file inputs can also be either **controlled** (from our last example) or **uncontrolled**, similar to text inputs.

In most cases, it’s better to use uncontrolled components for file inputs due to browser-specific behavior and the handling of large files. You can still access the selected file using refs.

```jsx
import { useRef } from 'react';

function UncontrolledFileUpload() {
  const fileInputRef = useRef(null);

  const handleSubmit = (e) => {
    e.preventDefault();
    if (fileInputRef.current.files[0]) {
      console.log('File Uploaded:', fileInputRef.current.files[0].name);
    }
  };

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="file"
        ref={fileInputRef}
      />
      <button type="submit">Upload</button>
    </form>
  );
}

export default UncontrolledFileUpload;
```

@oluwakemi Oluwadahunsi

React file inputs can also be either **controlled** (from our last example) or **uncontrolled**, similar to text inputs.

In most cases, **it's better to use uncontrolled components for file inputs** due to browser-specific behavior and the handling of large files. You can still access the selected file using refs. Example:

```jsx
import { useRef } from 'react';

function UncontrolledFileUpload() {
  const fileInputRef = useRef(null);

  const handleSubmit = (e) => {
    e.preventDefault();
    if (fileInputRef.current.files[0]) {
      console.log('File Uploaded:', fileInputRef.current.files[0].name);
    }
  };

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="file"
        ref={fileInputRef}
      />
      <button type="submit">Upload</button>
    </form>
  );
}

export default UncontrolledFileUpload;
```

Handling Form Submission

In React, form submission is handled via the onSubmit event, and typically the form's default behavior of refreshing the page on submission is prevented by calling **e.preventDefault()**.

This gives you control over what happens when the form is submitted, such as **performing form validation**, **sending data to an API**, or **clearing the form fields**.

A basic example:

Handle form logic like form validation, sending form data, etc...

```
1 const handleSubmit = (e) => {
2   e.preventDefault();
3   // Perform form submission logic here
4 };
```

@oluwakemi Oluwadahunsi

Sending Data to an API

React can handle form submission by sending the data to an API using **fetch** or **Axios**.

Here’s an example of a basic **React login form** that sends form data (**username and password**) to an **API endpoint**. In this case, we're simulating a login request by sending the form data via a **POST** request using the **fetch API**.

This is a Controlled component example.

@oluwakemi Oluwadahunsi

```jsx
import { useState } from 'react';

function LoginForm() {
  const [username, setUsername] = useState('');
  const [password, setPassword] = useState('');
  const [error, setError] = useState(null);

  const handleSubmit = async (e) => {
    e.preventDefault();
    try {
      const response = await fetch('https://explanation.com/api/login', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({ username, password })
      });

      if (!response.ok) {
        throw new Error('Login failed');
      }

      const data = await response.json();
      console.log('Form Submitted:', data);
    } catch (err) {
      setError(err.message);
    }
  };

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="text"
        placeholder="Username"
        value={username}
        onChange={(e) => setUsername(e.target.value)}
      />
      <input
        type="password"
        placeholder="Password"
        value={password}
        onChange={(e) => setPassword(e.target.value)}
      />
      {error && <p style={{ color: 'red' }}>{error}</p>}
      <button type="submit">Login</button>
    </form>
  );
}

export default LoginForm;
```

@oluwakemi Oluwadahunsi

In the Login form example:

- We use **useState** to manage the form inputs (username and password) and error tracking.

- The **handleChange** function updates the state whenever a user types in the input fields, making sure the form data is always up-to-date.

- The **handleSubmit** function prevents the form's default behavior (which would refresh the page), sends a **POST** request to the login API endpoint (**https://explanation.com/api/login**) with the form data in the request body.

- The **try-catch** blocks in the **handleSubmit** function handles errors to ensure that the user gets feedback in case something goes wrong during the form submission. These errors are displayed in **line 42** of the code.

@oluwakemi Oluwadahunsi

Best Practices for Handling Forms in React

- **Keep the form state minimal:** Only store values that are essential for the form. Avoid duplicating the state between the form inputs and state.

- **Debounce input changes:** If your form is connected to an API, you might want to debounce the input changes to avoid sending too many requests.

- **Use third-party form libraries when necessary:** Libraries like Formik, React Hook Form, and Yup provide advanced validation features and handle complex forms efficiently.

- **Handle form errors gracefully:** Always provide clear error messages to users and ensure the form is accessible and user-friendly.

@oluwakemi Oluwadahunsi

I hope you found this material useful and helpful.

Remember to:

Like

Save for future reference

&

Share with your network, be helpful to someone 👌

@oluwakemi Oluwadahunsi

Hi There!

Thank you for reading through

Did you enjoy this knowledge?

💼 Follow my LinkedIn page for more work-life balancing and Coding tips.

🌐 LinkedIn: Oluwakemi Oluwadahunsi

kodemaven-portfolio.vercel.app