Formularios y Validaciones en React

Parte 1: Manejo de Formularios en React

En esta primera parte, vamos a aprender cómo manejar formularios en React mediante **formularios controlados**. Los formularios controlados permiten manejar el estado de cada campo directamente en React, lo que facilita realizar validaciones y actualizaciones en tiempo real.

¿Qué es un formulario controlado en React?

En un **formulario controlado**, el valor de cada campo de entrada (input) es controlado por el estado de React. Esto significa que cada vez que el usuario escribe en un campo, el estado del componente se actualiza y refleja el valor del campo de entrada.

Ventajas de los formularios controlados:

- Permiten un control completo sobre el valor de los campos.
- Facilitan la validación en tiempo real y la manipulación de los datos antes de su envío.
- Mejoran la experiencia de usuario al poder mostrar mensajes de error en el momento en que el usuario completa el campo.

Paso a Paso: Creación de un Formulario Controlado

Vamos a crear un formulario básico en React que capture el nombre y el correo electrónico del usuario.

Archivo: src/Formulario.jsx

1. **Crear un nuevo archivo** en src/ llamado Formulario.jsx.
2. Implementa el siguiente código:

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

function Formulario() {
  // Estado para los campos del formulario
  const [nombre, setNombre] = useState('');
  const [correo, setCorreo] = useState('');

  // Funciones para manejar el cambio en los inputs
  const handleNombreChange = (event) => {
    setNombre(event.target.value);
  };

  const handleCorreoChange = (event) => {
    setCorreo(event.target.value);
  };

  return (
    <form>
      <div>
        <label>Nombre:</label>
        <input
          type="text"
          value={nombre}
          onChange={handleNombreChange}
        />
      </div>
      <div>
```

```
        <label>Correo:</label>
        <input
          type="email"
          value={correo}
          onChange={handleCorreoChange}
        />
      </div>
    </form>
  );
}

export default Formulario;
```

Explicación del código:

1. **Estado del formulario**:
    - Usamos el hook useState para crear el estado de los campos nombre y correo.
    - nombre y correo se inicializan como cadenas vacías.
2. **Manejo de eventos onChange**:
    - Cada campo de entrada (input) tiene un atributo value que se asigna a su valor correspondiente en el estado.
    - La función onChange en cada campo actualiza el estado cada vez que el usuario escribe en el campo. Así, el valor del campo y el estado están sincronizados en todo momento.
3. **Formulario básico**:
    - El formulario contiene dos campos: uno para el nombre y otro para el correo.

- Los valores de los campos se muestran en tiempo real, lo cual permite controlar y validar la entrada del usuario.

Agregar el Formulario a App

Para ver el formulario en acción, debemos incluirlo en el componente principal.

Archivo: src/App.js

1. Abre App.js.
2. Importa el componente Formulario y agrégalo a la estructura del componente.

```
import React from 'react';
import Header from './Header';
import Footer from './Footer';
import Usuario from './Usuario';
import Contador from './Contador';
import Toggle from './Toggle';
import LoginControl from './LoginControl';
import Formulario from './Formulario';

function App() {
  return (
    <div>
      <Header titulo="¡Bienvenidos a mi aplicación React!" />
      <main>
        <p>Esta es la sección principal de la página.</p>
        <Usuario nombre="Juan" edad={30} />
        <Contador />
        <Toggle />
```

```
        <LoginControl />
        <Formulario />
      </main>
      <Footer year={new Date().getFullYear()} />
    </div>
  );
}

export default App;
```

Resultado esperado

1. Ejecuta la aplicación (npm start) y abre http://localhost:3000 en tu navegador.
2. Deberías ver el formulario con dos campos: **Nombre** y **Correo**.
3. Al escribir en cualquiera de los campos, el valor del estado en React se actualizará, reflejándose en tiempo real en el campo correspondiente.

Resumen

En esta primera parte, hemos cubierto:

- **Formularios controlados en React**: Cómo sincronizar el estado de un campo de entrada con React usando el hook useState.
- **Manejo de eventos onChange**: Cómo actualizar el estado cada vez que el usuario escribe en un campo, manteniendo el formulario controlado por React.

Parte 2: Manejo del Estado de Formularios Complejos

En la primera parte, vimos cómo usar el hook useState para controlar campos individuales. Sin embargo, cuando un formulario tiene varios campos, es más eficiente manejar todos los valores en un solo estado.

Al usar un único objeto en useState para almacenar todos los campos, podemos centralizar la lógica y reducir la cantidad de funciones de manejo de eventos.

Ejemplo: Manejo del Estado en Formularios Complejos

Vamos a mejorar nuestro formulario para que gestione ambos campos (nombre y correo) en un único objeto de estado.

Archivo: src/Formulario.jsx

1. Abre el archivo Formulario.jsx.
2. Modifica el código para que utilice un solo estado para todos los campos:

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


function Formulario() {
  // Estado único para todos los campos del formulario
  const [formData, setFormData] = useState({
    nombre: '',
    correo: '',
  });


  // Función de manejo de eventos para actualizar el estado de
cualquier campo
  const handleChange = (event) => {
    const { name, value } = event.target;
    setFormData((prevData) => ({
      ...prevData,
      [name]: value,
```

```
    }));
  };

  return (
    <form>
      <div>
        <label>Nombre:</label>
        <input
          type="text"
          name="nombre"
          value={formData.nombre}
          onChange={handleChange}
        />
      </div>
      <div>
        <label>Correo:</label>
        <input
          type="email"
          name="correo"
          value={formData.correo}
          onChange={handleChange}
        />
      </div>
    </form>
  );
}
```

```
export default Formulario;
```

Explicación del código

1. **Objeto de estado**: Creamos un objeto formData en el estado para almacenar todos los campos del formulario. Este objeto inicializa cada campo (nombre y correo) con una cadena vacía.
2. **Función handleChange**:
    - Usamos una sola función para manejar el evento onChange de todos los campos.
    - La función toma el nombre y el valor del campo actual (name y value de event.target) y actualiza el estado del campo correspondiente en formData.
    - La sintaxis [name]: value permite acceder dinámicamente al campo correspondiente del estado.
3. **Atributo name en los inputs**:
    - El uso de name en cada <input> es crucial para que handleChange sepa qué campo se está actualizando. El valor de name coincide con las propiedades en el estado (nombre y correo).

Ventajas del Manejo Centralizado del Estado

- **Simplicidad**: Usar un solo estado para múltiples campos reduce la cantidad de variables y funciones necesarias.
- **Flexibilidad**: Permite agregar más campos en el futuro sin tener que crear una función handleChange específica para cada uno.
- **Escalabilidad**: Es especialmente útil en formularios largos, ya que simplifica el manejo de datos en comparación con la creación de múltiples variables de estado.

Probar la Aplicación

Para ver este formulario en acción, asegúrate de que la aplicación esté ejecutándose (npm start) y abre http://localhost:3000. Al escribir en cada campo, deberías ver que el valor cambia en tiempo real y se actualiza en el estado del componente Formulario.

Parte 3: Validaciones en Formularios de React

Las validaciones en los formularios permiten:

- Asegurar que los campos tengan los valores correctos.
- Proporcionar retroalimentación al usuario cuando la entrada no cumple con los requisitos.

Existen diferentes tipos de validaciones:

1. **Validaciones básicas**: Por ejemplo, verificar si un campo está vacío o si el formato de un correo electrónico es correcto.
2. **Validaciones personalizadas**: Cualquier lógica que desees aplicar a los campos, como verificar si un nombre tiene una longitud mínima o si un número está dentro de un rango permitido.

Ejemplo: Validación de Campos en Tiempo Real

Vamos a modificar el formulario para añadir validaciones en tiempo real en los campos **nombre** y **correo**. Mostraremos mensajes de error debajo de cada campo cuando los valores no cumplan con los criterios.

Archivo: src/Formulario.jsx

1. Abre Formulario.jsx.
2. Modifica el código para incluir la lógica de validación:

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


function Formulario() {
  const [formData, setFormData] = useState({
    nombre: '',
```

```
  correo: '',
});

const [errors, setErrors] = useState({
  nombre: '',
  correo: '',
});

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

  // Realizar la validación en tiempo real
  validateField(name, value);
};

const validateField = (name, value) => {
  let errorMessage = '';

  // Validación del campo "nombre"
  if (name === 'nombre') {
    if (value.trim() === '') {
      errorMessage = 'El nombre es obligatorio.';
    } else if (value.length < 3) {
```

```
        errorMessage = 'El nombre debe tener al menos 3
caracteres.';
      }
    }

    // Validación del campo "correo"
    if (name === 'correo') {
      const correoRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
      if (value.trim() === '') {
        errorMessage = 'El correo es obligatorio.';
      } else if (!correoRegex.test(value)) {
        errorMessage = 'El correo no es válido.';
      }
    }

    // Actualizar el estado de errores
    setErrors((prevErrors) => ({
      ...prevErrors,
      [name]: errorMessage,
    }));
  };

  return (
    <form>
      <div>
        <label>Nombre:</label>
        <input
```

```
          type="text"
          name="nombre"
          value={formData.nombre}
          onChange={handleChange}
        />
        {errors.nombre && <p
className="error">{errors.nombre}</p>}
      </div>
      <div>
        <label>Correo:</label>
        <input
          type="email"
          name="correo"
          value={formData.correo}
          onChange={handleChange}
        />
        {errors.correo && <p
className="error">{errors.correo}</p>}
      </div>
    </form>
  );
}

export default Formulario;
```

Explicación del código

1. **Estado para errores**:

- Creamos un estado adicional errors para almacenar mensajes de error por cada campo.
- Este estado tiene la misma estructura que formData, donde cada propiedad representa un campo del formulario.

2. **Función validateField**:
   - Esta función realiza la validación en tiempo real de cada campo.
   - Si el campo nombre está vacío o tiene menos de 3 caracteres, se muestra un mensaje de error.
   - Para el campo correo, usamos una expresión regular para verificar el formato del correo electrónico.
3. **Mostrar mensajes de error**:
   - Debajo de cada campo, mostramos el mensaje de error correspondiente si existe, usando errors.nombre y errors.correo.

Ventajas de la Validación en Tiempo Real

- **Interactividad**: El usuario recibe retroalimentación inmediata mientras completa el formulario.
- **Experiencia de Usuario**: Mejora la experiencia al permitirle corregir errores en el momento, en lugar de esperar hasta enviar el formulario.

Probar la Validación

Para ver la validación en acción:

1. Asegúrate de que la aplicación esté ejecutándose (npm start) y abre http://localhost:3000.
2. Intenta dejar los campos vacíos, ingresar un nombre corto o un correo con formato incorrecto. Los mensajes de error deberían aparecer debajo del campo correspondiente.

Parte 4: Gestión de Eventos en el Envío del Formulario

En React, el evento submit nos permite realizar acciones cuando el usuario envía el formulario. A través de este evento podemos:

1. Realizar una **validación completa** antes de procesar los datos.
2. **Evitar el comportamiento predeterminado** del formulario de recargar la página.
3. **Procesar** o **enviar los datos** capturados a una API o al backend.

Ejemplo: Gestión del Evento submit en un Formulario Controlado

Vamos a modificar el formulario actual para que valide todos los campos cuando el usuario haga clic en el botón de enviar y mostrar un mensaje de éxito si todos los campos son válidos.

Archivo: src/Formulario.jsx

1. Abre Formulario.jsx.
2. Añade el manejo del evento submit y la validación final al código:

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

function Formulario() {
  const [formData, setFormData] = useState({
    nombre: '',
    correo: '',
  });

  const [errors, setErrors] = useState({
    nombre: '',
    correo: '',
  });
```

```
  const [isSubmitted, setIsSubmitted] = useState(false);

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

    // Validación en tiempo real
    validateField(name, value);
  };

  const validateField = (name, value) => {
    let errorMessage = '';

    if (name === 'nombre') {
      if (value.trim() === '') {
        errorMessage = 'El nombre es obligatorio.';
      } else if (value.length < 3) {
        errorMessage = 'El nombre debe tener al menos 3
caracteres.';
      }
    }

    if (name === 'correo') {
```

```
      const correoRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
      if (value.trim() === '') {
        errorMessage = 'El correo es obligatorio.';
      } else if (!correoRegex.test(value)) {
        errorMessage = 'El correo no es válido.';
      }
    }

    setErrors((prevErrors) => ({
      ...prevErrors,
      [name]: errorMessage,
    }));
  };

  const handleSubmit = (event) => {
    event.preventDefault(); // Evita que el formulario recargue la
página
    let isValid = true;

    // Validación final de todos los campos
    Object.keys(formData).forEach((field) => {
      validateField(field, formData[field]);
      if (errors[field]) {
        isValid = false;
      }
    });
```

```
    // Verificar si el formulario es válido
    if (isValid) {
      setIsSubmitted(true); // Indica que el formulario se ha
enviado con éxito
    } else {
      setIsSubmitted(false);
    }
  };

  return (
    <form onSubmit={handleSubmit}>
      <div>
        <label>Nombre:</label>
        <input
          type="text"
          name="nombre"
          value={formData.nombre}
          onChange={handleChange}
        />
        {errors.nombre && <p
className="error">{errors.nombre}</p>}
      </div>
      <div>
        <label>Correo:</label>
        <input
          type="email"
          name="correo"
```

```
          value={formData.correo}
          onChange={handleChange}
        />
        {errors.correo && <p
classNameName="error">{errors.correo}</p>}
      </div>
      <button type="submit">Enviar</button>
      {isSubmitted && <p classNameName="success">Formulario enviado
con éxito</p>}
    </form>
  );
}

export default Formulario;
```

Explicación del Código

1. **handleSubmit**:
    - Esta función se ejecuta cuando el formulario se envía (evento submit).
    - Primero, previene el comportamiento predeterminado (event.preventDefault()), evitando que la página se recargue.
    - Luego, realiza una validación final de todos los campos. Si todos los campos son válidos, cambia el estado isSubmitted a true para indicar que el formulario fue enviado exitosamente.
2. **Estado isSubmitted**:
    - Este estado indica si el formulario ha sido enviado correctamente. Si isSubmitted es true, se muestra un mensaje de éxito.
3. **Mensaje de éxito**:
    - Si isSubmitted es verdadero, el mensaje "Formulario enviado con éxito" se muestra debajo del botón de envío.

Ventajas de Manejar el Evento submit

- **Validación final**: Puedes asegurarte de que todos los campos sean válidos antes de procesar los datos.
- **Control total**: Al evitar el comportamiento predeterminado del formulario, puedes manejar los datos de manera personalizada, como enviarlos a una API.
- **Flexibilidad**: Puedes proporcionar retroalimentación personalizada al usuario sobre el éxito o error en el envío del formulario.

Probar la Aplicación

Para ver esta funcionalidad en acción:

1. Asegúrate de que la aplicación esté ejecutándose (npm start) y abre http://localhost:3000.
2. Completa los campos del formulario y haz clic en **Enviar**.
3. Si el formulario está correctamente completado, deberías ver el mensaje de éxito. Si no, asegúrate de que los mensajes de error se muestren para los campos inválidos.