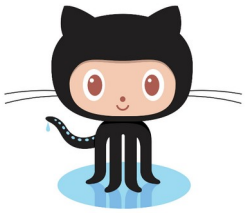
Desarrollo web en entorno cliente

UD – 5

Formularios & Axios

Contenidos



ÍNDICE

1. INTRODUCCIÓN

Los formularios son una parte fundamental en el desarrollo de aplicaciones web, permitiendo la interacción del usuario y la recolección de datos. Con el paso del tiempo, la gestión de formularios ha evolucionado significativamente, desde las primeras versiones de HTML hasta las modernas bibliotecas como React.

Los formularios en HTML **tienen sus raíces** en los primeros días de la web. En sus primeras versiones, HTML proporcionaba elementos como `<input>`, `<textarea>`, `<select>`, entre otros, para permitir a los usuarios enviar información a través de la web. Sin embargo, la gestión de estos formularios *era bastante estática*. Los datos enviados por los usuarios eran procesados en el servidor, y los formularios no tenían mecanismos integrados para validaciones dinámicas ni para interactuar con el usuario de manera fluida.

Formularios controlados VS Formularios no controlados.

En React, existen dos enfoques principales para manejar los formularios:

1. **Formularios Controlados:**
   En este enfoque, el valor de cada campo del formulario está completamente gestionado por el estado de React. Esto significa que cada vez que el usuario interactúa con un campo (por ejemplo, escribiendo en un input), el estado del componente se actualiza, lo que permite tener un control total sobre los datos del formulario. Los formularios controlados son muy útiles para validaciones en tiempo real y la manipulación de datos de manera estructurada.
2. **Formularios No Controlados:**
   En los formularios no controlados, el estado del formulario no está ligado directamente al estado de React. En cambio, se accede al valor de los campos a través de referencias (refs). Este enfoque es útil en situaciones donde no es necesario realizar un seguimiento constante del estado de cada campo o cuando se desea un rendimiento mejorado, ya que no hay actualizaciones frecuentes del estado.

¿Cuál elegir? ¿Controlados? ¿No controlados?

En este curso, nos centraremos en los formularios **controlados**, ya que proporcionan un control más robusto sobre los datos del formulario y son ideales para aplicaciones interactivas con validación dinámica y control de estado en tiempo real.

Existen herramientas y bibliotecas que pueden facilitar y mejorar la experiencia de trabajo con formularios en React. Algunas de las más populares incluyen:

- **Formik:**
  Formik es una de las bibliotecas más populares para el manejo de formularios en React. Proporciona una forma sencilla de manejar el estado del formulario, la validación y la gestión de errores. Formik abstrae la lógica repetitiva de los formularios y mejora la organización del código, lo que resulta en un desarrollo más rápido y eficiente.
- **React Hook Form:**
  React Hook Form es una biblioteca basada en hooks que facilita el manejo de formularios en React. Permite trabajar con formularios de manera más eficiente en términos de rendimiento, ya que minimiza las re-renderizaciones innecesarias. Es ligera, fácil de integrar y ofrece una API simple para manejar validaciones y envíos de formularios.

¿Elección de la Herramienta?

Aunque React proporciona una excelente solución para manejar formularios controlados de manera nativa, bibliotecas como Formik y React Hook Form pueden simplificar aún más la gestión de formularios, especialmente en aplicaciones más grandes o con formularios complejos.

2. GESTIONAR EL CAMBIO

La función **gestionarCambio** (*o handleChange en inglés)* es una función comúnmente usada en formularios en React. Su propósito es actualizar el estado del componente cuando un usuario interactúa con un campo de formulario (como un input, textarea, select, etc.).

Cómo funciona:

- **Eventos:** Se usa un manejador de eventos como onChange, que se ejecuta cada vez que el valor de un campo cambia.
- **Desestructuración:** Se extraen las propiedades “name, value, type, y checked” del evento para identificar y gestionar el valor de cada campo correctamente.
- **Actualización del estado:** Se actualiza el estado del formulario con el nuevo valor usando setState, asegurando que el formulario esté sincronizado con el estado.

```
const gestionarCambio = (e) => {
  const { name, value, type, checked } = e.target;
  setForm({
    ...form,
    [name]: type === 'checkbox' ? checked : value,
  });
};
```

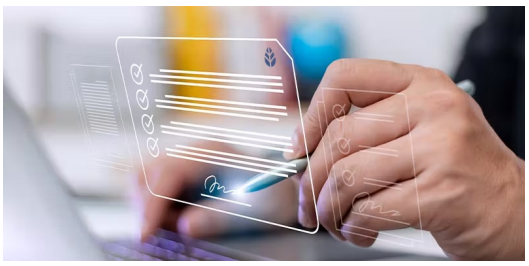
3. CAMPO GENÉRICO

Un input de tipo texto (<input type="text">) es el campo más comúnmente usado en formularios web para capturar datos de texto, como un nombre. Este campo es parte fundamental de cualquier formulario de entrada de datos.

Características clave:

- **name:** El atributo name se usa para identificar el campo dentro del formulario y es crucial cuando se gestionan cambios y se validan los datos.
- **value:** El atributo value está vinculado al estado del componente. Controla el valor que se muestra en el campo de entrada.
- **onChange:** Se utiliza para escuchar los cambios en el campo y actualizar el estado del formulario con el nuevo valor.

```
<label htmlFor="nombre">Nombre</label>
<input
  id="nombre"
  type="text"
  name="nombre"
  value={form.nombre}
  onChange={gestionarCambio}
  placeholder="Escribe tu nombre"
/>
```



4. VALIDAR

La función **validar** es responsable de revisar que los datos introducidos por el usuario sean correctos según las reglas que se definan (como longitud, formato, campos obligatorios, etc.).

Cómo funciona:

- **Lógica de validación:** Se define una serie de condiciones para cada campo del formulario. Si un campo no cumple con una condición, se agrega un mensaje de error en un objeto de errores.
- **Devolución de errores:** Si hay errores en los campos, la validación devolverá un objeto con los mensajes de error correspondientes. Si no hay errores, devuelve true.

```
const validar = () => {
  const nuevosErrores = {};
  if (!form.nombre.trim()) {
    nuevosErrores.nombre = 'El nombre es obligatorio';
  }
  if (form.descripcion && (form.descripcion.length < 10 || form.descripcion.length
    nuevosErrores.descripcion = 'La descripción debe tener entre 10 y 100 caractere
  }
  setErrores(nuevosErrores);
  return Object.keys(nuevosErrores).length === 0;
};
```

Es importante mostrar mensajes de error de manera clara para que los usuarios sepan qué está mal:

```
return (
  <form onSubmit={enviarFormulario}>
    {/* Campo de texto para nombre */}
    <label htmlFor="nombre">Nombre</label>
    <input
      id="nombre"
      type="text"
      name="nombre"
      value={form.nombre}
      onChange={gestionarCambio}
      placeholder="Escribe tu nombre"
    />
    {errores.nombre && <p className="error">{errores.nombre}</p>}
```

5. “SUBMIT” DEL FORMULARIO

El envío de formularios en React generalmente se maneja utilizando el evento onSubmit, que se asocia a un formulario HTML (<form>). Este evento es disparado cuando el usuario intenta enviar el formulario (por ejemplo, presionando un botón de tipo submit o presionando Enter en un campo de texto).

```
const enviarFormulario = (e) => {
  e.preventDefault();

  // Validar antes de enviar
  if (validar()) {
    console.clear();
    console.log('Formulario Enviado', form);
  }
};
```

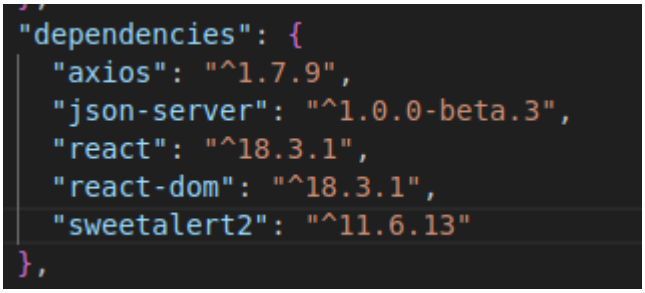
6. AXIOS

Axios es una biblioteca de JavaScript basada en promesas que facilita las solicitudes HTTP. Es ampliamente utilizada en proyectos React para interactuar con APIs, ya que ofrece una sintaxis simple y características avanzadas como interceptores, cancelación de solicitudes y manejo automático de JSON.

Primero, instala Axios en tu proyecto React:

npm install axios

Automáticamente es añadida como dependencia al proyecto actual, en el archivo package.json:



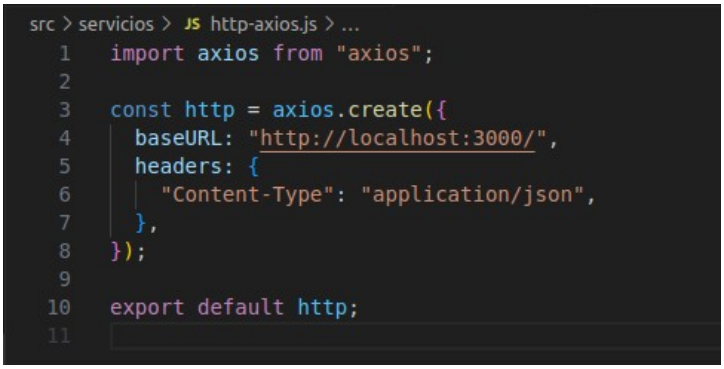
6.1 Características principales de Axios

Entre las muchas características de axios, las principales son:

- **Soporte para Promesas:** Facilita trabajar con operaciones asíncronas utilizando **.then()** y .**catch()**.
- **Soporte para asincronía:** Permite un código más limpio y fácil de leer.
- **Intercepción de solicitudes y respuestas:** Puedes agregar lógica como autenticación o logs de errores.
- **Transformación de datos:** Permite modificar datos de entrada o salida.
- **Compatibilidad con navegadores:** Maneja automáticamente problemas como CORS y convierte respuestas JSON.

6.2 Configuración como servicio: http-axios

Este archivo configura una **instancia personalizada de Axios** para centralizar la configuración y facilitar las solicitudes HTTP en un proyecto.



6.3 Servicio de aficiones

Este archivo define una clase llamada ServicioAficiones, que encapsula las operaciones relacionadas con el recurso "aficiones" en una API. Utiliza la instancia personalizada de Axios (http) para realizar solicitudes HTTP. Su propósito es centralizar y reutilizar las llamadas a la API, manteniendo el código más limpio y

```
import http from "./http-axios.js";

class ServicioAficiones {
  getAll() {
    return http.get("/aficiones");
  }

  get(id) {
    return http.get(`/aficiones/${id}`);
  }

  create(data) {
    return http.post("/aficiones", data);
  }

  update(id, data) {
    return http.put(`/aficiones/${id}`, data);
  }

  delete(id) {
    return http.delete(`/aficiones/${id}`);
  }
}

export default new ServicioAficiones();
```

organizado.

6.4 Ejemplo de uso

Este fragmento de código utiliza React para gestionar el estado de una lista de aficiones obtenida desde una API al montar el componente. Aquí está el desglose:

- **aficiones** es una variable de estado que almacena la lista de aficiones y **setAficiones** es la función a utilizar para actualizar el valor de aficiones.
- **useEffect**: Ejecuta el código de su interior cuando el componente se monta. [], es utilizado para indicar que solo se ejecute una vez.

```
const Informe = () => {
  const [aficiones, setAficiones] = useState([]);

  useEffect(() => {
    ServicioAficiones.getAll()
      .then((response) => {
        setAficiones(response.data);
      })
      .catch((error) => {

        Swal.fire({
          title: "¿Tienes Internet?",
          text: "No consigo descargar las aficiones :(",
          icon: "question"
        });
      });
  }, []);
```

7. JSON-SERVER

JSON Server es una herramienta que permite crear un servidor RESTful de manera rápida y sencilla, utilizando un archivo JSON como base de datos. Es muy útil para prototipos y pruebas en el desarrollo de aplicaciones front-end cuando aún no se tiene un backend real, ya que emula un servidor API con un comportamiento similar al de un servidor real.

public > {} data.json > [ ] aficiones > {} 2

```json
{
  "aficiones": [
    { "id": "1", "nombre": "futbol", "descripcion": "Los partidos de la liga andaluza" },
    { "id": "2", "nombre": "cocina", "descripcion": "cocina tradicional de una madre" },
    { "id": "3", "nombre": "correr", "descripcion": "Salir a correr por Fuente del Berro" }

  ]
}
```

Necesitamos instalarlo y lanzarlo, especificando el archivo de BD correspondiente:

npm install -g json-server

npx json-server public/data.json

Si todo ha funcionado bien, recibirás las urls para interactuar con la API: