FRONTEND:

Formularios y peticiones HTTP

En esta unidad veremos la interacción del usuario a través de formularios destacando bibliotecas populares como Formik y React Hook Formy. Estas herramientas nos facilitan la validación de formularios, el manejo de estados y la administración de eventos. Para construir aplicaciones verdaderamente dinámicas, necesitamos la capacidad de interactuar con servicios externos, por eso aprenderemos a cómo se gestionan las peticiones HTTP pudiendo hacer que nuestras aplicaciones se conecten con el mundo exterior.

Librerías para la creación de formularios

Los formularios se merecen un tema independiente ya que son una de las herramientas más importantes cuando trabajamos con una aplicación de Front. Se trata de la única herramienta con la cual podemos recuperar información directa del usuario permitiéndole introducir los datos que necesite sin influir en dicha introducción.

Hasta el momento hemos usado formularios a través de un doble enlace entre una variable y el campo de texto correspondiente, pero para poder dar un paso más necesitamos crear formularios de manera diferente.

En esta unidad vamos a analizar dos de las librerías más potentes para la creación de formularios en React, Formik y React Hook Form.

1. Formik

Para poder usar la librería, lo primero que debemos hacer, sobre un proyecto nuevo es la instalación:

```
npm install formik
```

A partir de dicha instalación ya podemos empezar a generar componentes de tipo formulario con los elementos propios de la librería Formik. El funcionamiento básico es la utilización del hook `useFormik`, que será el encargado de gestionar los campos, las validaciones y el envío del formulario.

src/components/Contacto.jsx

```
import { Field, Form, Formik, useFormik } from 'formik';
const Contacto = () => {
const formik = useFormik({
initialValues: { name: '', surname: '', email: '' },
onSubmit: values => {
console.log(values);
}
})
return <>
<h3>Formulario de contacto</h3>
<form onSubmit={formik.handleSubmit}>
<div className="form-group">
<label>Nombre</label>
<input
type="text"
name="name"
id="name"
onChange={formik.handleChange}
value={formik.values.name} />
</div>
<div className="form-group">
<label>Apellidos</label>
<input
type="text"
name="surname"
id="surname"
onChange={formik.handleChange}
value={formik.values.surname} />
</div>
<div className="form-group">
<label>Email</label>
<input
type="text"
name="email"
id="email"
onChange={formik.handleChange}
value={formik.values.email} />
</div>
<button type="submit">Enviar</button>
</form>
</>;
}
export default Contacto;
```

- En primera instancia hacemos una llamada al hook `useFormik` donde definimos dos propiedades importantes:

    - **initialValues**: se trata del objeto donde estamos generando los valores iniciales para todos los campos del formulario.
    - **onSubmit**: la función donde vamos a enviar los valores del formulario cuando pulsemos en el botón enviar.
- A partir de esa ejecución retornamos dentro del JSX del componente los diferentes campos del formulario.
    - En la etiqueta form enlazamos el método submit.
    - Dentro de cada uno de los campos de texto enlazamos el valor correspondiente generado por formik, así como la gestión del evento onChange.

Con esta serie de configuraciones previas el formulario ya estaría funcionando, pero podemos incluir más capas para comprobar, por ejemplo, las diferentes validaciones que necesitemos implementar.

Vamos a generar una función sencilla de validación donde retornaremos un objeto con todos aquellos errores que vayamos encontrando:

src/components/Contacto.jsx

```
import { Field, Form, Formik, useFormik } from 'formik';
const validate = values => {
const errors = {};
if (!values.name) {
// Comprobamos si el campo nombre está incluido
errors.name = 'Required';
} else if (values.name.length > 15) {
// Comprobamos si el campo nombre tiene como máximo 15 caracteres
errors.name = 'Must be 15 characters or less';
}
if (!values.surname) {
errors.surname = 'Required';
} else if (values.surname.length > 20) {
errors.surname = 'Must be 20 characters or less';
}
if (!values.email) {
errors.email = 'Required';
} else if (!/^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}$/i.test(values.email)) {
// Comprobamos mediante una expresión regular si el email es válido
errors.email = 'Invalid email address';
}
return errors;
};
const Contacto = () => {
const formik = useFormik({
initialValues: { name: '', surname: '', email: '' },
validate,
onSubmit: values => {
console.log(values);
```

```
}
})
return <>
<h3>Formulario de contacto</h3>
<form onSubmit={formik.handleSubmit}>
<div className="form-group">
<label>Nombre</label>
<input
type="text"
name="name"
id="name"
onChange={formik.handleChange}
value={formik.values.name} />
{formik.errors.name && <div>{formik.errors.name}</div>}
</div>
<div className="form-group">
<label>Apellidos</label>
<input
type="text"
name="surname"
id="surname"
onChange={formik.handleChange}
value={formik.values.surname} />
{formik.errors.surname && <div>{formik.errors.surname}</div>}
</div>
<div className="form-group">
<label>Email</label>
<input
type="text"
name="email"
id="email"
onChange={formik.handleChange}
value={formik.values.email} />
{formik.errors.email && <div>{formik.errors.email}</div>}
</div>
<button type="submit">Enviar</button>
</form>
</>;
}
export default Contacto;
```

- Generamos una función `validate` fuera del componente que recibe por parámetro los valores del formulario y devuelve un objeto con los posibles errores que podamos encontrar a partir de esos valores.
- Para hacer las validaciones podemos utilizar todas las herramientas que necesitemos, como condicionales, expresiones regulares...
- Si el objeto `errors` devuelto tiene alguna clave quiere decir que tenemos algún tipo de error en el formulario y por tanto el método definido dentro de la propiedad `onSubmit` no se ejecutará.

- Para que el método validate funcione con nuestro formulario debemos incluirlo en la llamada al hook useFormik.
- Para cada uno de los campos, a través del objeto formik.errors podemos comprobar si está afectado por alguno de los errores que tenemos predefinidos.
- De ser así colocamos un DIV con el error debajo del propio campo de texto.

Podemos seguir incrementando la funcionalidad de nuestro formulario. En este caso vamos a interactuar con el evento onBlur del formulario para así enterarnos de cuando el usuario ha estado en el campo concreto y lo ha abandonado

De esta manera podemos ser más concretos a la hora de mostrar los errores:

src/components/Contacto.jsx

```
import { Field, Form, Formik, useFormik } from 'formik';
const validate = values => {
const errors = {};
if (!values.name) {
// Comprobamos si el campo nombre está incluido
errors.name = 'Required';
} else if (values.name.length > 15) {
// Comprobamos si el campo nombre tiene como máximo 15 caracteres
errors.name = 'Must be 15 characters or less';
}
if (!values.surname) {
errors.surname = 'Required';
} else if (values.surname.length > 20) {
errors.surname = 'Must be 20 characters or less';
}
if (!values.email) {
errors.email = 'Required';
} else if (!/^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}$/i.test(values.email)) {
// Comprobamos mediante una expresión regular si el email es válido
errors.email = 'Invalid email address';
}
return errors;
};
const Contacto = () => {
const formik = useFormik({
initialValues: { name: '', surname: '', email: '' },
validate,
onSubmit: values => {
console.log(values);
}
})
return <>
<h3>Formulario de contacto</h3>
<form onSubmit={formik.handleSubmit}>
<div className="form-group">
```

```
<label>Nombre</label>
<input
type="text"
name="name"
id="name"
onBlur={formik.handleBlur}
onChange={formik.handleChange}
value={formik.values.name} />
{formik.touched.name && formik.errors.name ? (
<div>{formik.errors.name}</div>
) : null}
</div>
<div className="form-group">
<label>Apellidos</label>
<input
type="text"
name="surname"
id="surname"
onBlur={formik.handleBlur}
onChange={formik.handleChange}
value={formik.values.surname} />
{formik.touched.surname && formik.errors.surname ? (
<div>{formik.errors.surname}</div>
) : null}
</div>
<div className="form-group">
<label>Email</label>
<input
type="text"
name="email"
id="email"
onBlur={formik.handleBlur}
onChange={formik.handleChange}
value={formik.values.email} />
{formik.touched.email && formik.errors.email ? (
<div>{formik.errors.email}</div>
) : null}
</div>
<button type="submit">Enviar</button>
</form>
</>;
}
export default Contacto;
```

- Para cada uno de los campos incluimos la gestión del evento `onBlur`.
- Con la acción anterior tenemos acceso al objeto `formik.touched` que nos indica qué campos han sido tocados por el usuario.

- Modificamos los mensajes de error para que la validación sea mucho más precisa, únicamente cuando el campo sea inválido y además el usuario haya pasado por el mismo.

Las validaciones personalizadas pueden incluir cualquier tipo de condición, pero si necesitamos un poco más de orden podemos utilizar alguna herramienta que nos facilite el trabajo.

Para poder generar esquemas que nos permitan validar de manera más completa nuestros formularios podemos usar la librería **YUP**.

El primer paso es instalar la librería en el proyecto:

```
npm install yup
```

A partir de ahí podemos definir un esquema de validación dentro de la llamada al hook useFormik:

src/components/Contacto.jsx

```
import { useFormik } from 'formik';
import * as Yup from 'yup';
const Contacto = () => {
const formik = useFormik({
initialValues: { name: '', surname: '', email: '' },
validationSchema: Yup.object({
name: Yup.string()
.max(15, 'Must be 15 characters or less')
.required('Required'),
surname: Yup.string()
.max(20, 'Must be 20 characters or less')
.required('Required'),
email: Yup.string().email('Invalid email address').required('Required'),
}),
onSubmit: values => {
console.log(values);
}
})
return <>
<h3>Formulario de contacto</h3>
<form onSubmit={formik.handleSubmit}>
<div className="form-group">
<label>Nombre</label>
<input
type="text"
name="name"
id="name"
onBlur={formik.handleBlur}
onChange={formik.handleChange}
```

```
value={formik.values.name} />
{formik.touched.name && formik.errors.name ? (
<div>{formik.errors.name}</div>
) : null}
</div>
<div className="form-group">
<label>Apellidos</label>
<input
type="text"
name="surname"
id="surname"
onBlur={formik.handleBlur}
onChange={formik.handleChange}
value={formik.values.surname} />
{formik.touched.surname && formik.errors.surname ? (
<div>{formik.errors.surname}</div>
) : null}
</div>
<div className="form-group">
<label>Email</label>
<input
type="text"
name="email"
id="email"
onBlur={formik.handleBlur}
onChange={formik.handleChange}
value={formik.values.email} />
{formik.touched.email && formik.errors.email ? (
<div>{formik.errors.email}</div>
) : null}
</div>
<button type="submit">Enviar</button>
</form>
</>;
}
export default Contacto;
```

Utilizando la propiedad `validationSchema` e integrando las diferentes acciones que nos permite llevar a cabo Yup podemos mejorar mucho el código que estamos desarrollando.

Si necesitamos entrar más en detalle sobre esta última librería, podemos recuperar sus métodos más importantes del siguiente enlace: https://github.com/jquense/yup

Por último, para dejar nuestro código lo más limpio posible y una vez hemos visto cómo podemos recuperar las diferentes propiedades de los campos de texto, podemos utilizar el método `getFieldProps` para aplicarlas todas de una vez:

src/components/Contacto.jsx

```
import { useFormik } from 'formik';
import * as Yup from 'yup';
const Contacto = () => {
const formik = useFormik({
initialValues: { name: '', surname: '', email: '' },
validationSchema: Yup.object({
name: Yup.string()
.max(15, 'Must be 15 characters or less')
.required('Required'),
surname: Yup.string()
.max(20, 'Must be 20 characters or less')
.required('Required'),
email: Yup.string().email('Invalid email address').required('Required'),
}),
onSubmit: values => {
console.log(values);
}
})
return <>
<h3>Formulario de contacto</h3>
<form onSubmit={formik.handleSubmit}>
<div className="form-group">
<label>Nombre</label>
<input
type="text"
name="name"
id="name"
{...formik.getFieldProps('name')} />
{formik.touched.name && formik.errors.name ? (
<div>{formik.errors.name}</div>
) : null}
</div>
<div className="form-group">
<label>Apellidos</label>
<input
type="text"
name="surname"
id="surname"
{...formik.getFieldProps('surname')} />
{formik.touched.surname && formik.errors.surname ? (
<div>{formik.errors.surname}</div>
) : null}
</div>
<div className="form-group">
<label>Email</label>
<input
type="text"
name="email"
id="email"
```

```
{...formik.getFieldProps('email')} />
{formik.touched.email && formik.errors.email ? (
<div>{formik.errors.email}</div>
) : null}
</div>
<button type="submit">Enviar</button>
</form>
</>;
}
export default Contacto;
```

En esta versión de nuestro componente, la librería formik queda muy bien integrada y podemos controlar al máximo cualquiera de los aspectos que sucedan sobre nuestro formulario.

2. React Hook Form

La otra alternativa que te vamos a presentar es una de las librerías más modernas y podríamos decir que recién salida del horno.

Mediante la librería React Hook Form vamos a poder generar formularios con menos código y con un alto rendimiento. Es labor del desarrollador decidir qué librería cumple mejor con lo que necesite para cada proyecto.

El primer paso para poder usar la librería es su instalación:

```
npm install react-hook-form
```

A partir de aquí podemos empezar a definir un nuevo componente con otro formulario:

src/components/Registro.jsx

```
import { useForm } from "react-hook-form";
const Registro = () => {
const { register, handleSubmit } = useForm();
const onSubmit = data => console.log(data);
return (
<form onSubmit={handleSubmit(onSubmit)}>
<div>
<label>Username</label>
<input {...register('username')} />
</div>
<div>
<label>País</label>
<select {...register('country')}>
<option value="es">España</option>
<option value="it">Italia</option>
<option value="pt">Portugal</option>
</select>
```

```
</div>
<div>
<label>Dirección</label>
<input {...register('address')} />
</div>
<input type="submit" />
</form>
);
}
export default Registro;
```

- El primer paso es ejecutar el hook `useForm` y de dicha ejecución recuperar los métodos `register` y `handleSubmit`.
- El primero de ellos lo usamos dentro de cada uno de los campos para registrar su valor y todos los eventos asociados.
- El método `handleSubmit` lo usamos para gestionar el evento `onSubmit` dentro del formulario y poder asignarle de igual manera una función donde gestionar el envío del formulario.
- Podemos rellenar los campos del formulario y observamos por consola sus valores.

En este caso, las validaciones que necesitemos realizar sobre los diferentes campos de nuestro formulario las hemos de hacer dentro de la ejecución del método `register`:

src/components/Registro.jsx

```
import { useForm } from "react-hook-form";
const Registro = () => {
const { register, handleSubmit } = useForm();
const onSubmit = data => console.log(data);
return (
<form onSubmit={handleSubmit(onSubmit)}>
<div>
<label>Username</label>
<input {...register('username', { required: true, maxLength: 20 })} />
</div>
<div>
<label>País</label>
<select {...register('country', { required: true })}>
<option value="es">España</option>
<option value="it">Italia</option>
<option value="pt">Portugal</option>
</select>
</div>
<div>
<label>Dirección</label>
<input {...register('address', { required: true })} />
</div>
<input type="submit" />
</form>
```

```
);
}
export default Registro;
```

De esta manera, el formulario no ejecuta el método submit hasta que no se cumpla con todos los validadores que definimos dentro del JSX.

Una vez realicemos las validaciones, podemos mostrar los errores dentro del JSX:

src/components/Registro.jsx

```
import { useForm } from "react-hook-form";
const Registro = () => {
const { register, formState: { errors }, handleSubmit } = useForm();
const onSubmit = data => console.log(data);
return (
<form onSubmit={handleSubmit(onSubmit)}>
<div>
<label>Username</label>
<input {...register('username', { required: true, maxLength: 20 })} />
{errors.username?.type === 'required' && "Username es requerido"}
{errors.username?.type === 'maxLength' && "Username debe tener menos de 20
caracteres"}
</div>
<div>
<label>País</label>
<select {...register('country', { required: true })}>
<option value="es">España</option>
<option value="it">Italia</option>
<option value="pt">Portugal</option>
</select>
{errors.country?.type === 'required' && "Country es requerido"}
</div>
<div>
<label>Dirección</label>
<input {...register('address', { required: true })} />
{errors.address?.type === 'required' && "Dirección es requerida"}
</div>
<input type="submit" />
</form>
);
}
export default Registro;
```

- Lo primero es recuperar el objeto `errors` dentro de la llamada al hook.
- Para cada uno de los campos podemos gestionar la posibilidad de que tenga errores como hemos hecho en el ejemplo anterior.

En este segundo ejemplo el código queda más limpio que en el anterior, pero el objetivo alcanzado es el mismo.

Peticiones HTTP

Es el momento de salir hacia el exterior. Hasta ahora hemos trabajado nuestras aplicaciones de manera interna dándoles forma y concretando las diferentes herramientas que nos proporcionan los componentes.

A partir de ahora empezaremos a hablar de cómo podemos recoger datos de manera externa a nuestros componentes y de esta manera tener una interacción completa con nuestros sistemas de trabajo Full Stack.

1. useEffect

Mediante el uso de este hook podemos, dentro de los componentes funcionales, ejecutar una serie de sentencias justo después de renderizar el código JSX correspondiente.

Lo vemos en el siguiente ejemplo:

src/components/Mensaje.jsx

```
import { useEffect, useState } from "react";
const Mensaje = () => {
const [num, setNum] = useState(0);
useEffect(() => {
console.log('[Mensaje] useEffect');
});
return (
<div>
<p>Este es el mensaje</p>
<button onClick={() => setNum(num + 1)}>Aumenta</button>
</div>
);
};
export default Mensaje;
```

El hook `useEffect` se ejecuta en cada actualización del estado o de las props. Si pulsamos el botón observamos como se ejecuta el método `useEffect` cada vez que aprecia un cambio en el estado del número.

Es importante controlar cuándo se ejecuta el hook para evitar acciones inesperadas y sobre todo entrar en bucles innecesarios que nos lleven a rebajar el rendimiento de nuestras aplicaciones.

Si necesitamos que se ejecute una única vez en el ciclo de vida del componente, podemos pasar al hook un segundo parámetro. En este caso sería un array vacío.

Podemos comprobarlo sobre el ejemplo anterior:

src/components/Mensaje.jsx

```
import { useEffect, useState } from "react";
const Mensaje = () => {
const [num, setNum] = useState(0);
useEffect(() => {
console.log('[Mensaje] useEffect');
}, []);
return (
<div>
<p>Este es el mensaje</p>
<button onClick={() => setNum(num + 1)}>Aumenta</button>
</div>
);
};
export default Mensaje;
```

Con este segundo ejemplo, si pulsamos el botón y, por lo tanto, modificamos el estado, ya no ejecutamos después del render el método dentro del hook useEffect.

- Dentro del array que pasamos como segundo parámetro al hook podemos incluir todas las dependencias que tendrá dicho hook.
- Podemos colocar ciertas variables como dependencias para que el hook únicamente se lance cuando las modifiquemos.

src/components/Mensaje.jsx

```
import { useEffect, useState } from "react";
const Mensaje = () => {
const [num, setNum] = useState(0);
const [mensaje, setMensaje] = useState('Mensaje inicial');
useEffect(() => {
console.log('[Mensaje] useEffect');
}, [mensaje]);
return (
<div>
<p>{mensaje}</p>
<p>{num}</p>
<button onClick={() => setNum(num + 1)}>Aumenta</button>
<input type="text" onChange={(e) => setMensaje(e.target.value)} />
</div>
);
};
export default Mensaje;
```

Hemos modificado el ejemplo anterior. Además del número hemos colocado dentro del estado un string (`mensaje`).

- Dentro del array asociado a la ejecución de `useEffect` hemos agregado la variable `mensaje`. Podemos observar que únicamente se ejecuta dicho método cuando modificamos `mensaje`.
- Si necesitamos realizar alguna acción relacionada con los cambios producidos en la variable `num` podemos hacer otra llamada a `useEffect` poniendo como dependencia dicha variable.

Dentro de este método, como veremos más adelante durante la unidad, tiene sentido lanzar todo tipo de acciones que cuadren después de mostrar nuestra vista (petición a una API, modificación del DOM...).

2. Peticiones HTTP

A partir de esta sección vamos a gestionar la comunicación de datos entre el servidor y la aplicación de React. Uno de nuestros objetivos finales dentro del curso y paso previo a entender la comunicación completa en un sistema Full Stack es saber cómo podemos recuperar información de sistemas externos a través de diferentes herramientas.

En este caso, como estaremos comunicando con diferentes APIs, no será una transferencia de datos al uso como la que podemos estar acostumbrados cuando navegamos por internet. Los datos que recibamos y procesemos dentro de nuestras aplicaciones de React habitualmente tendrán un formato JSON.

Para poder realizar esta comunicación vamos a utilizar una herramienta muy potente que no está relacionada con React pero que casi forma parte de su ecosistema, axios. Podemos instalarla dentro de cualquier proyecto de la siguiente forma:

```
npm install axios
```

Vamos a intentar a partir de ahora replicar todos los tipos de peticiones que ya vimos cuando estábamos creando nuestras APIs en NodeJS.

📌 Hoy en día, está muy extendido el uso de fetch (integrado en el ecosistema) en vez de axios (requiere de instalación), por lo que veremos ejemplos comentados. Además, tendremos el apartado 3 para fetch.

📌 Para nuestras pruebas vamos a usar diferentes APIs. Para todas aquellas pruebas que no impliquen ningún tipo de actualización y únicamente recuperación de datos, usaremos la swapi (https://swapi.dev/) Cuando necesitemos más interacción usaremos JSONPlaceholder (https://jsonplaceholder.typicode.com/)

2.1 Petición GET

Por norma general, este tipo de peticiones las realizaremos dentro del método que se ejecuta justos después de renderizar el componente. En los componentes de clase lo

haremos dentro de componentDidMount y en los componentes funcionales dentro de una llamada al hook **useEffect**.

El uso de axios (así como de fetch) es muy sencillo ya que dispone de una serie de métodos que replican los principales verbos HTTP con los que vamos a interactuar. Empezamos realizando un ejemplo sobre el método get.

src/components/PeopleList.jsx

```
import axios from 'axios';
import { useEffect, useState } from 'react';
const PeopleList = () => {
const [people, setPeople] = useState([]);
useEffect(() => {
axios.get('https://swapi.dev/api/people')
.then(response => {
setPeople(response.data.results);
})
/** ejemplo con fetch:
fetch('https://swapi.dev/api/people')
.then(response => response.json())
.then(data => { setPeople(data.results); })
 .catch(error => {
console.error('Error fetching data:', error);
})
*/
}, []);
let listPeople = null;
if (people.length > 0) {
listPeople = people.map(person => (
<div className="person">
<h3>{person.name}</h3>
<p>Año Nacimiento: {person.birth_year}</p>
<p>Núm. Películas: {person.films.length}</p>
</div>))
}
return (
<div className="people">
{listPeople}
</div>
);
}
export default PeopleList;
```

- Dentro del método useEffect hacemos la llamada al método get de axios (o fetch).. Este método devuelve una Promise y la resolvemos a través de su método then.
- Dentro de la resolución de la promesa actualizamos el array people con los datos recuperados en la llamada.

- El resto del componente se encarga de gestionar, a partir del array de people, la lista que vamos a representar dentro del JSX a renderizar.
- Antes de hacer la llamada a setPeople podríamos realizar todo tipo de modificaciones sobre los datos que vamos a almacenar en el estado.

Nuestro siguiente objetivo para ver cómo podemos utilizar en toda su magnitud el hook useEffect es intentar recuperar el detalle de uno de los personajes cuando seleccionemos su nombre en la lista. Para ello, vamos a generar un nuevo componente que reciba a través de las props la url de la cual va a descargar la información.

src/components/PersonDetail.jsx

```
import { useEffect, useState } from "react";
import axios from 'axios';
export default function PersonDetail({ url }) {
const [person, setPerson] = useState(null);
useEffect(() => {
async function fetchData() {
const response = await axios.get(url);
console.log(response);
setPerson(response.data);
/* con fetch
try {
const response = await fetch(url);
const data = await response.json();
console.log(data);
setPerson(data);
} catch(error){
console.error('Error fetching data:', error);
}
*/
}
if (url) fetchData();
}, [url]);
if (!person) {
return <h2>Ninguna persona seleccionada</h2>;
} else {
return (<div>
<h3>{person.name}</h3>
</div>)
}
}
```

- Podemos comprobar en este ejemplo el uso que hacemos de la prop url.
- Dentro del método useEffect ponemos esta variable como dependencia, por lo que cuando se modifique desde el componente padre, se volverá a ejecutar todo el contenido del hook, recuperando, por tanto, los datos de la nueva persona seleccionada.

- Para evitar un warning de React que nos indica que la función definida como parámetro del hook `useEffect` no puede ser asíncrona, generamos una función dentro y ahí asignamos el modificador async. Posteriormente, en el interior del hook, ejecutamos dicha función.

Debemos modificar también el componente `PeopleList` para poder usar el detalle de cada personaje.

src/components/PeopleList.jsx

```
import axios from 'axios';
import { useEffect, useState } from 'react';
import PersonDetail from './PersonDetail';
const PeopleList = () => {
const [people, setPeople] = useState([]);
const [urlSelected, setUrlSelected] = useState('');
useEffect(() => {
axios.get('https://swapi.dev/api/people')
.then(response => {
setPeople(response.data.results);
})
}, []);
let listPeople = null;
if (people.length > 0) {
listPeople = people.map(person => (
<div className="person">
<h3 onClick={() => setUrlSelected(person.url)}>{person.name}</h3>
<p>Año Nacimiento: {person.birth_year}</p>
<p>Núm. Películas: {person.films.length}</p>
</div>))
}
return (
<div>
<div className="people">
{listPeople}
</div>
<PersonDetail url={urlSelected} />
</div>
);
}
export default PeopleList;
```

📌 Sería interesante tomar un tiempo para asignar estilos a estos dos componentes y que así el trabajo con los mismos sea más agradable.

2.2 Petición POST

Del mismo modo podemos lanzar peticiones POST a partir de las herramientas que nos proporciona Axios.

src/components/DataPost.jsx

```
import axios from "axios";
const DataPost = () => {
const handleClick = () => {
const body = {
title: 'Prueba de título',
body: 'Cuerpo del post',
author: 'Antonio Robles'
};
axios.post('https://jsonplaceholder.typicode.com/posts', body)
.then(response => console.log(response.data))
.catch(error => console.log(error));
/** con fetch
fetch('https://jsonplaceholder.typicode.com/posts',
{
method: 'POST',
headers: { 'Content-Type': 'application/json' },
body: JSON.stringify(body) })
 .then(response => response.json())
.then(data => console.log(data))
 .catch(error =>console.log(error));
*/
}
return (<div>
<button onClick={handleClick}>Pulsa para POST</button>
</div>)
}
export default DataPost;
```

2.3 Petición UPDATE

Probamos a lanzar una petición de actualización de datos.

src/components/DataUpdate.jsx

```
import axios from "axios";
const DataUpdate = () => {
const handleClick = async () => {
const body = {
```

```
title: 'Título nuevo',
body: 'Nuevo contenido',
author: 'Rodrigo Lafuente'
}
const response = await axios.put('https://jsonplaceholder.typicode.com/posts/1', body);
console.log(response.data);
/** con fetch
const response = await fetch('https://jsonplaceholder.typicode.com/posts/1',
{
method: 'PUT',
headers: { 'Content-Type': 'application/json' },
body: JSON.stringify(body)
});
const data = await response.json();
console.log(data);
} catch (error) {
console.error('Error updating data:', error);
}
*/
}
return <div>
<button onClick={handleClick}>Pulsa para UPDATE</button>
</div>
}
export default DataUpdate;
```

La dinámica es la misma que para el método POST pero cambiando la petición en función del servidor.

2.4 Petición DELETE

Por último, definimos un ejemplo para poder lanzar una petición con el método DELETE.

src/components/DataDelete.jsx

```
import axios from "axios";
const DataDelete = () => {
const handleClick = () => {
axios.delete('https://jsonplaceholder.typicode.com/posts/1')
.then(response => console.log(response.data))
.catch(error => console.log(error));
}
return <div>
<button onClick={handleClick}>Pulsa para DELETE</button>
</div>
}
export default DataDelete;
```

```
//Con fetch
const DataDelete = () => {
async function deleteData() {
try {
const response = await fetch('https://jsonplaceholder.typicode.com/posts/1',
{ method: 'DELETE', }
);
const data = await response.json();
console.log(data);
} catch (error) {
console.error('Error deleting data:', error); }
}
const handleClick = () => deleteData();
return <div>
<button onClick={handleClick}>Pulsa para DELETE</button>
</div>
}
export default DataDelete;
```

2.5 Gestión de Errores

Debemos determinar una manera óptima para la inclusión de los errores dentro de la interfaz gráfica de nuestros componentes. De alguna manera, el usuario debe quedar informado de lo que vaya sucediendo según vayamos completando las peticiones contra el servidor.

Dentro del ejemplo de `PeopleList` vamos a notificar de errores dentro del JSX que renderiza el componente.

Para forzar el error, simplemente hay que colocar la URL mal.

src/components/PeopleList.jsx

```
import axios from 'axios';
import { useEffect, useState } from 'react';
import PersonDetail from './PersonDetail';
const PeopleList = () => {
const [people, setPeople] = useState([]);
const [urlSelected, setUrlSelected] = useState('');
const [error, setError] = useState(false);
useEffect(() => {
axios.get('https://swapi.dev/api/peoplessss')
.then(response => {
setPeople(response.data.results);
})
.catch(error => setError(true));
}, []);
let listPeople = null;
```

```
if (error) listPeople = <p style={{ textAlign: 'center' }}>Ha ocurrido un error</p>;
if (!error && people.length > 0) {
listPeople = people.map(person => (
<div className="person">
<h3 onClick={() => setUrlSelected(person.url)}>{person.name}</h3>
<p>Año Nacimiento: {person.birth_year}</p>
<p>Núm. Películas: {person.films.length}</p>
</div>))
}
return (
<div>
<div className="people">
{listPeople}
</div>
<PersonDetail url={urlSelected} />
</div>
);
}
export default PeopleList;
```

Lo único que incluimos es la gestión del error a través del catch en la gestión de la promesa y la actualización del estado para indicar que existe error y por lo tanto mostrar ciertos bloques de código dentro del JSX.

2.6 Interceptors

Aparte de realizar las acciones básicas sobre un recurso en concreto, con axios podemos gestionar algunas características extra de las peticiones que realicemos sobre las diferentes APIs.

En el caso de los interceptors vamos a poder definir de manera global una serie de funciones para interactuar con las peticiones de salida y respuestas entrantes a nuestra aplicación.

Los podemos definir de la manera más general posible dentro del fichero **index.js**

src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';
import axios from 'axios';
axios.interceptors.request.use(request => {
console.log(request);
// tenemos que devolver la petición para no bloquearla
// Se ejecuta para cada petición
```

```
return request;
}, error => {
console.log(error);
return Promise.reject(error);
});
ReactDOM.render(
<React.StrictMode>
<App />
</React.StrictMode>,
document.getElementById('root')
);
reportWebVitals();
```

Con esta implementación podemos interactuar con la petición en cualquier momento antes de que se envíe al servidor de destino.

También podemos colocar un interceptor para la respuesta.

```
axios.interceptors.response.use(response => {
console.log(response);
return response;
}, error => {
console.log(error);
return Promise.reject(error);
})
```

Antes de que la respuesta llegue al componente donde la vamos a utilizar podemos transformar los datos para aplicarles acciones globales.

3. Peticiones HTTP con Fetch

Dentro de las alternativas que tenemos disponibles para lanzar peticiones HTTP desde nuestras aplicaciones, una de las más potentes es **Fetch**, una API Javascript nativa para realizar dicha interacción.

Vamos a ver a partir de un ejemplo las posibilidades que nos ofrece este api.

src/components/PeticionesFetch.jsx

```
const PeticionesFetch = () => {
const handleGET = () => {
fetch('https://jsonplaceholder.typicode.com/posts')
.then((response) => response.json())
.then((json) => console.log(json));
}
const handlePost = () => {
fetch('https://jsonplaceholder.typicode.com/posts', {
method: 'POST',
body: JSON.stringify({
```

```
title: 'foo',
body: 'bar',
userId: 1,
}),
headers: {
'Content-type': 'application/json; charset=UTF-8',
},
})
.then((response) => response.json())
.then((json) => console.log(json));
}
const handlePut = () => {
fetch('https://jsonplaceholder.typicode.com/posts/1', {
method: 'PUT',
body: JSON.stringify({
id: 1,
title: 'foo',
body: 'bar',
userId: 1,
}),
headers: {
'Content-type': 'application/json; charset=UTF-8',
},
})
.then((response) => response.json())
.then((json) => console.log(json));
}
const handleDelete = () => {
fetch('https://jsonplaceholder.typicode.com/posts/1', {
method: 'DELETE',
});
}
return <div>
<button onClick={handleGET}>GET</button>
<button onClick={handlePost}>POST</button>
<button onClick={handlePut}>PUT</button>
<button onClick={handleDelete}>DELETE</button>
</div>
}
export default PeticionesFetch;
```

Las acciones son las mismas, lo único que modificamos es la manera de recuperar la información y de estructurar la llamada.

EJERCICIO: Reto Fitness ¡Acompáñanos en tu viaje hacia una vida más saludable! 🏋️

En **FitLife**, nuestro objetivo es ayudarte a alcanzar tus metas de fitness de una manera divertida y sostenible. Creemos que el primer paso para un estilo de vida saludable es dar el paso, y por eso queremos facilitarte el proceso de unirte a nuestro gimnasio.

Tu misión:

Desarrollar un formulario de registro intuitivo y atractivo para que los nuevos miembros puedan unirse a FitLife con facilidad. El formulario debe ser atractivo y fácil de usar, reflejando la vibra moderna y acogedora de nuestro gimnasio.

Requisitos del formulario:

- **Componentes:**
  - Crear componentes reutilizables para cada sección del formulario (datos personales, información de contacto, preferencias de entrenamiento, etc.).
  - Implementar la lógica de cada sección como un componente independiente.
- **Props y State:**
  - Pasar datos entre componentes utilizando props.
  - Gestionar el estado interno de cada componente con useState.
- **Manejo de eventos:**
  - Validar la entrada del usuario en tiempo real.
  - Mostrar mensajes de error y éxito.
- **Comunicación hijo-padre:**
  - Enviar datos desde el componente hijo al componente padre al completar el formulario.
- **Condicionales y Listas:**
  - Mostrar u ocultar elementos del formulario según las opciones del usuario.
  - Mostrar una lista de opciones de entrenamiento disponibles.
- **Aplicación de estilos:**
  - Aplicar estilos CSS a los componentes para una interfaz atractiva.
  - Usar clases CSS para diferentes estados de los elementos (activo, inactivo, error).
- **Estilos dinámicos:**
  - Cambiar estilos dinámicamente según las opciones del usuario.
- **Styled Components:**
  - Implementar styled components para una mejor organización y mantenimiento del código CSS.
- **CSS Modules:**

    - Usar CSS Modules para evitar conflictos de nombres de clase.
- **Peticiones HTTP:**
    - Enviar datos del formulario al servidor para registrar al nuevo miembro.
- **Librerías:**
    - Implementar una librería de gestión de formularios como Formik o React Hook Form para facilitar la validación y el manejo de errores.

Recursos adicionales:

- Documentación de React: https://es.reactjs.org/docs/
- Formik: https://formik.org/
- React Hook Form: https://react-hook-form.com/

Consejos:

- Divide el formulario en secciones pequeñas y manejables.
- Comienza con una versión simple del formulario y ve agregando funcionalidades gradualmente.
- Prueba el formulario con diferentes casos de uso para asegurarte de que funciona correctamente.
- Pide feedback a tus compañeros o profesores para mejorar la experiencia del usuario.

¡Esperamos tu formulario y ayudarte a empezar tu viaje hacia una vida más saludable!

Guía para realizar el ejercicio:

1. Planificación y Diseño:

- **Definir las secciones del formulario:**
    - Datos personales (nombre, correo electrónico, teléfono)
    - Información de contacto (dirección, ciudad, código postal)
    - Preferencias de entrenamiento (tipo de entrenamiento, objetivos, disponibilidad)
    - Datos de pago (método de pago, información de la tarjeta)
- **Diseñar la interfaz del formulario:**
    - Elegir una paleta de colores y tipografía que represente la marca FitLife.
    - Crear un diseño atractivo y fácil de usar para cada sección del formulario.
    - Utilizar imágenes y vídeos para mostrar el ambiente del gimnasio y las clases disponibles.

2. Implementación de React:

- **Componentes:**

    - Crear un componente principal para el formulario que contenga los demás componentes.
    - Desarrollar componentes independientes para cada sección del formulario.
    - Pasar datos entre componentes utilizando props.
- **Props y State:**
    - Definir el estado interno de cada componente con useState.
    - Pasar datos dinámicos a los componentes como props.
- **Manejo de eventos:**
    - Validar la entrada del usuario en tiempo real.
    - Mostrar mensajes de error y éxito.
    - Implementar eventos para enviar el formulario.

3. Funcionalidades avanzadas:

- **Comunicación hijo-padre:**
    - Enviar datos desde el componente hijo al componente padre al completar el formulario.
- **Condicionales y Listas:**
    - Mostrar u ocultar elementos del formulario según las opciones del usuario.
    - Mostrar una lista de opciones de entrenamiento disponibles.
- **Aplicación de estilos:**
    - Aplicar estilos CSS a los componentes para una interfaz atractiva.
    - Usar clases CSS para diferentes estados de los elementos (activo, inactivo, error).
- **Estilos dinámicos:**
    - Cambiar estilos dinámicamente según las opciones del usuario.

4. Integración de librerías:

- **Formik o React Hook Form:**
    - Implementar una librería de gestión de formularios para facilitar la validación y el manejo de errores.
    - Configurar las reglas de validación para cada campo del formulario.
    - Mostrar mensajes de error personalizados.

5. Peticiones HTTP:

- **Enviar datos del formulario al servidor:**
    - Implementar una API para registrar al nuevo miembro.
    - Enviar una petición HTTP al servidor con los datos del formulario.
    - Mostrar un mensaje de éxito o error al completar la solicitud.

Consejos de integración de Formik y peticiones HTTP en tu formulario FitLife

1. Instalación de Formik:

Comienza instalando las librerías necesarias:

```
npm install formik yup
```

2. Implementación básica de Formik:

1. Componente principal (ejemplo):

```
import { useFormik } from 'formik';

import Yup from 'yup';

const MyForm = () => {
  const formik = useFormik({
    initialValues: {
      nombre: '',
      email: '',
      telefono: '',
    },
    validationSchema: Yup.object({
      nombre: Yup.string().required('El nombre es obligatorio'),
      email: Yup.string().email('Introduce un email válido').required('El email es obligatorio'),
      telefono: Yup.string().required('El teléfono es obligatorio'),
    }),
    onSubmit: (values) => {
      // Enviar datos a la API
      console.log('Formulario enviado:', values);
    },
  });

  return (
    <form onSubmit={formik.handleSubmit}>
      <input type="text" name="nombre" value={formik.values.nombre}
onChange={formik.handleChange} />
      {formik.errors.nombre && <p className="error">{formik.errors.nombre}</p>}
      ... (similar para email y telefono)
      <button type="submit">Enviar</button>
    </form>
  );
};
```

- `useFormik` crea un hook que nos permite gestionar el estado del formulario.
- `initialValues` define los valores iniciales de los campos del formulario.
- `validationSchema` define las reglas de validación para cada campo.
- `onSubmit` se ejecuta cuando se envía el formulario.

3. Envío de datos a una API:

1. Función para enviar la petición POST (ejemplo):

```
const enviarDatosAPI = async (datos) => {
  const respuesta = await fetch('https://api.fitlife.com/registro', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify(datos),
  });

  if (respuesta.ok) {
    // Mostrar mensaje de éxito
    console.log('Usuario registrado correctamente');
  } else {
    // Mostrar mensaje de error
    console.log('Error al registrar usuario');
  }
};
```

2. Integración en el componente:

```
const MyForm = () => {

 ...

  onSubmit: (values) => {
    enviarDatosAPI(values);
  },
  ...
};
```

- enviarDatosAPI envía una petición POST a la API con los datos del formulario.
- Se comprueba la respuesta de la API para mostrar un mensaje de éxito o error.

CUESTIONARIO

---

1. En React, ¿cuál es la función principal de los métodos componentDidMount y componentDidUpdate en el contexto de las peticiones HTTP?
a) Iniciar el proceso de solicitud al servidor.
b) Actualizar el estado del componente después de una solicitud.
c) Realizar operaciones de limpieza después de montar o actualizar el componente.
d) Ninguna de las anteriores.
Respuesta correcta: c) Realizar operaciones de limpieza después de montar o actualizar el componente.

2. ¿Cuál de las siguientes declaraciones describe mejor el propósito de un interceptor en Axios?
a) Manejar errores de red en las peticiones HTTP.
b) Modificar globalmente las solicitudes o respuestas antes de que se envíen o procesen.
c) Controlar la autenticación de usuarios en el servidor.
d) Gestionar la presentación de datos en la interfaz de usuario.
Respuesta correcta: b) Modificar globalmente las solicitudes o respuestas antes de que se envíen o procesen.

3. ¿Qué método de Axios se utiliza para enviar una solicitud HTTP tipo POST al servidor?
a) axios.post()
b) axios.get()
c) axios.request()
d) axios.send()
Respuesta correcta: a) axios.post()

4. En React, ¿cuál es la forma más común de manejar las respuestas de las peticiones HTTP?
a) Utilizando la función fetch() proporcionada por React.
b) Actualizando directamente el estado del componente con los datos recibidos.
c) Utilizando el hook useEffect() para actualizar el estado del componente.
d) Utilizando callbacks dentro de los métodos then() y catch().
Respuesta correcta: d) Utilizando callbacks dentro de los métodos then() y catch().

5. ¿Qué es CORS (Cross-Origin Resource Sharing) y cómo afecta a las peticiones HTTP en React?
a) CORS es una técnica para compartir recursos entre diferentes dominios, y puede causar problemas al realizar solicitudes HTTP a servidores con políticas de CORS restrictivas.
b) CORS es una característica de React que permite compartir recursos entre diferentes componentes del mismo dominio.
c) CORS es un tipo de autenticación para las solicitudes HTTP en React que requiere credenciales de usuario.
d) CORS es un formato de datos utilizado para transmitir información entre el cliente y el servidor en React.
Respuesta correcta: a) CORS es una técnica para compartir recursos entre diferentes dominios, y puede causar problemas al realizar solicitudes HTTP a servidores con políticas de CORS restrictivas.

6. ¿Cuál de las siguientes afirmaciones describe mejor el propósito del método DELETE en una solicitud HTTP?
a) Actualiza los datos en el servidor.
b) Recupera datos del servidor.
c) Elimina datos en el servidor.
d) Envía datos al servidor.
Respuesta correcta: c) Elimina datos en el servidor.

7. En el contexto de React, ¿cuál es la ventaja principal de utilizar axios sobre fetch para realizar peticiones HTTP?

a) axios ofrece una sintaxis más simple y concisa.
b) axios proporciona una gestión más avanzada de errores.
c) fetch está obsoleto y ya no se recomienda su uso en React.
d) fetch tiene mejor soporte para peticiones asíncronas.
Respuesta correcta: b) axios proporciona una gestión más avanzada de errores.

8. ¿Qué hook de React se utiliza comúnmente para manejar la autenticación de usuarios antes de enviar peticiones HTTP?
a) useEffect
b) useState
c) useContext
d) useReducer
Respuesta correcta: c) useContext

9. ¿Cuál es el propósito de la función fetch() en JavaScript cuando se usa para realizar peticiones HTTP?
a) Enviar solicitudes a servidores remotos y recibir respuestas.
b) Manipular datos en la base de datos local.
c) Ejecutar operaciones matemáticas en el navegador.
d) Ninguna de las anteriores.
Respuesta correcta: a) Enviar solicitudes a servidores remotos y recibir respuestas.

10. ¿Cuál de las siguientes opciones es un código válido para realizar una petición HTTP "update" con Axios y actualizar el estado del componente?

```
const [data, setData] = useState([]);

const handleUpdate = () => {
  axios.put(`/api/users/${id}`, {
    name: "Nuevo nombre",
    email: "nuevo@email.com",
  }).then((response) => {
    setData(response.data);
  });
};
```

a) Sí, es un código válido.
b) No, es un código válido. Falta el hook useEffect.
c) No, es un código válido. Falta el hook useReducer.
d) No, es un código válido. Falta el hook useMemo.

Respuesta correcta: a) Si, es un código válido.

PARA SABER MÁS

Formik:
Guía de Formik - Librería para Formularios en React

React Hook Form:
Tutorial Completo de React Hook Form

Peticiones HTTP:
Axios.js - Curso Practico de Peticiones HTTP