React. Inputs y formularios

Desarrollo Web Cliente.

React.Inputs y Formularios

Inputs

Para utilizar un input con React y tener control sobre los cambios del usuario podemos utilizar el evento onChange. Este evento recibe como parámetro el evento en sí, y a través de event.target.value podemos acceder a su valor.

Para que el valor del input concuerde con el estado del componente será necesario actualizarlo en onChange.

En onChange tenemos que pasar la referencia a una función.

Así si tenemos el estado inputValue declarado así:

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

El input definido dentro del JSX que devuelve nuestro componente así:

<input id="searchInput" value ={inputValue} onChange={handleChange}></input>

Deberemos definir la función:

const handleChange = (event) => {

console.log(event)

```
setInputValue(event.target.value);
}
```

En esta función además podremos incluir las validaciones u acciones que necesitemos llevar a cabo. De esta manera cuando llegue el evento onClick el valor del input estará en nuestra variable de estado.

Useref

Otra forma de trabajar con los inputs es con el hook useRef.

Se utiliza para tener una referencia a una variable y se puede utilizar en dos escenarios:

- Para referenciar un elemento del DOM y poder accedera los métodos de su API a través de .current

- Para tener una variable que se inicializa la primera vez, pero que a diferencia de los estados no va a provocar nuevos renderizados del componente.

Para utilizar useRef:

Import {useRef} from 'react'

const miref = useRef(valor inicial)

El valor de la referencia estará almacenado en current, así que para leer o modificar el valor tenemos que acceder:

miRef.current = valor que queremos dar

var valor = miRef.current (obtenemos el valor de miRef y se lo pasamos a otra variable, función...)

Para poder referenciar un elemento del DOM en JSX tenemos que añadir la propiedad: ref = {miref} al componente.

Si tenemos esta referencia:

```
const inputValue= useRef('');
```

Y en el JSX que devuelve el componente enlazamos esa referencia con:

<input id="searchInput" ref={inputValue} ></input>

Asumiendo que tenemos un evento onClick en un botón manejado por la función:

```
const search = () => {
  console.log(inputValue.current);
  searchImage(inputValue.current.value);
  inputValue.current.value = '';
}
```

Así podriamos obtener el valor del input y pasárselo a la función que deba realizar la gestión esperada con ese valor.

Formik

Cuando tenemos un único input es bastante manejable, pero cuando tenemos formularios más o menos grandes que queremos enviar al servidor es útil el uso de una librería. Vamos a ver dos de ellas, formik y React Hook Form.

Formik es más sencilla pero con formularios grandes puede tener problemas de rendimiento, mientras que React Hook Form es más compleja pero tiene mejor performance.

Instalación:

Para la instalación es necesario ejecutar:

```
npm install formik
```

Componente básico:

Lo primero que tenemos que hacer es importar Formik dentro de nuestro componente:

import {Formik} from 'formik'

Vamos a empezar pasando a Formik las props: initialValues y onSubmit

InitialValues --> será un objeto con el nombre de cada campo y el valor inicial

Por ejemplo:

```
initialValues={{ email: '', password: '' }}
```

```
onSubmit={(values) => console.log(values)}
```

Dentro de Formik tenemos que pasar una función, que recibirá cmo parámetros un objeto con las propiedades values, handleSubmit y handleChange

({ values, handleSubmit, handleChange })

Y que devolverá el formulario JSX. Es importante que el nombre de los inputs concuerde con el pasado anteriormente en initialValues. El form deberá tener asociado la funcion apropiada a on Submit. También deberá haber un botón tipo submit.

```jsx
import { Formik } from 'formik'

export default function BasicForm() {
    return (
        <Formik
            initialValues={{ email: '', password: '' }}
            onSubmit={(values) => console.log(values)}
        >{
                ({ values, handleSubmit, handleChange }) => (
                    <form onSubmit={handleSubmit}>
                        <input
                            type="email"
                            name="email"
                            value={values.email}
                            onChange={handleChange}
                        />
                        <input
                            type="password"
                            name="password"
                            value={values.password}
                            onChange={handleChange}

                        />
                        <button type="submit" >
                            Submit
                        </button>
                    </form>
                )
            }
        </Formik>
    )
}
```

Validación de errores:

Para añadir validación de errores debemos pasarle otra prop a Formik llamada validate. Esta prop consiste en una función que recibe los valores por parámetro y que devuelve un objeto con los errores.

```
validate={values => {
    const errors = {};
    if (!values.email) {
      errors.email = 'Required';
    } else if (
      !/^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,}$/i.test(values.email)
    ) {
      errors.email = 'Invalid email address';
    }
    return errors;
  }}
```

Formik por defecto valida los campos al introducir un cambio, al perder el foco y antes de enviar el formulario. Si hay errores el formulario no se enviará.

Ahora si obtenemos también errors de la función que va a devolver el formulario podemos usar errors para mostrar o no el error. Por ejemplo:

```
({ values, handleSubmit, handleChange, errors }) => (
    <form onSubmit={handleSubmit}>
        <input
            type="email"
            name="email"
            value={values.email}
            onChange={handleChange}
        />
        {errors.email}
```

Y lo mismo para password.

Si queremos que el error solo se muestre cuando se pierde el foco, podemos controlarlo con la propiedad touched y asignar al evento onBlur la función handleBlur. Podemos acceder a touched para cada elemento del formulario. Si añadimos algo de estructura con label y un div para el error tendríamos:

```
({ values, handleSubmit, handleChange, handleBlur, errors, touched }) => (
    <form onSubmit={handleSubmit} >
        <div className="form-group">
            <label htmlFor="email">Email</label>
            <input
                type="email"
                id="email"
                name="email"
                value={values.email}
                onChange={handleChange}
                onBlur={handleBlur}
            />
            <div className="error-message"> {touched.email && errors.email}</div>
        </div>
```

Nota: Es necesario añadir el campo En la siguiente página el ejemplo completo:

```
export default function BasicForm() {
    return (
        <Formik
            initialValues={{ email: '', password: '' }}
            onSubmit={(values) => console.log(values)}
            validate={values => {
                const errors = {};
                if (!values.email) {
                    errors.email = 'Required';
                } else if (
                    !/^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,}$/i.test(values.email)
                ) {
                    errors.email = 'Invalid email address';
                }
                if (!values.password) errors.password = 'Required'
                else if (values.password.length < 4) errors.password = 'Minimal length: 4';
                return errors;
            }} >
            {
                ({ values, handleSubmit, handleChange, handleBlur, errors, touched }) => (
                    <form onSubmit={handleSubmit} >
                        <div className="form-group">
                            <label htmlFor="email">Email</label>
                            <input
                                type="email" id="email" name="email"
                                value={values.email} onChange={handleChange} onBlur={handleBlur}
                            />
                            <div className="error-message"> {touched.email && errors.email}</div>
                        </div>
                        <div className="form-group">

                            <label htmlFor="password">Password</label>
                            <input
                                type="password" id="password" name="password"
                                value={values.password} onChange={handleChange} onBlur={handleBlur}
                            />
                            {<div className="error-message">{touched.password && errors.password}</div>}
                        </div>
                        <button type="submit" >
                            Submit
                        </button>
                    </form>
                )
            }
        </Formik>
    )
}
```

Componentes de Formik.

Para reducir el código repetido podemos hacer uso de los componentes de Formik.

```
import { Formik, Form, Field, ErrorMessage } from 'formik'

export default function CompleteForm() {
  return (
    <Formik
      initialValues={{ email: '', password: '' }}
      onSubmit={(values) => console.log(values)}
      validate={values => {
        const errors = {};
        if (!values.email) {
          errors.email = 'Required';
        } else if (
          !/^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,}$/i.test(values.email)
        ) {
          errors.email = 'Invalid email address';
        }
        if (!values.password) errors.password = 'Required'
        else if (values.password.length < 4) errors.password = 'Minimal length: 4';
        return errors;
      }} >
      {
        () => (
          <Form>
            <div className="form-group">
              <label htmlFor="email">Email</label>
              <Field name="email" type="email" />
              <ErrorMessage name="email" component="div" />
            </div>
            <div className="form-group">
              <label htmlFor="password">Password</label>
              <Field name="password" type="password" />
              <ErrorMessage name="password" component="div" />
            </div>
            <button type="submit" >
              Submit
            </button>
          </Form>
        )
      }
    </Formik>
  )
}
```

setSubmitting

Normalmente queremos saber cuándo el formulario está en proceso de envío, para bloquear las acciones del usuario o para darle feedback, para ello podemos añadir en la función que llamamos onSubmit un objeto con la propiedad setSubmitting. Formik pondrá la propiedad a true directamente al empezar a enviar el formulario y nosotros tendremos que controlar cuando queremos que pase a fale con setSubmitting(false)

```
export default function CompleteForm() {
  return (
    <Formik
      initialValues={{ email: '', password: '' }}
      onSubmit={(values, { setSubmitting }) => {
        setTimeout(() => {
          console.log(values)
          setSubmitting(false);
        }, 5000);
      }}
      validate={values => {
        const errors = {};
        if (!values.email) {
          errors.email = 'Required';
        } else if (
          !/^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,}$/i.test(values.email)
        ) {
          errors.email = 'Invalid email address';
        }
        if (!values.password) errors.password = 'Required'
        else if (values.password.length < 4) errors.password = 'Minimal length: 4';
        return errors;
      }} >
      {
        ({isSubmitting}) => (
          <Form>
            <div className="form-group">
              <label htmlFor="email">Email</label>
              <Field name="email" type="email" />
              <ErrorMessage name="email" component="div" />
            </div>
            <div className="form-group">
              <label htmlFor="password">Password</label>
              <Field name="password" type="password" />
              <ErrorMessage name="password" component="div" />
            </div>
            <button type="submit" disabled={isSubmitting} >
              Submit
            </button>
          </Form>
        )
      }
    </Formik>
  )
}
```

Yup

Podemos simplificar y unificar las validaciones usando yup. Para ello lo primero que tenemos es que definir es instalarlo e importarlo.

En la raíz de nuestro proyecto:

```
npm install yup
```

En el componente importamos todo yup, o las partes que vayamos a usar:

```
import * as Yup from 'yup'
```

A continuación tenemos que definir un esquema. Para ello llamamos a la función object() de Yup y le pasamos un objeto con los campos del formulario que queramos validar y las validaciones específicas para cada caso. Un ejemplo, para nuestro mail y password:

```
const SignSquema = Yup.object({
  email: Yup.string().email().required(),
  password: Yup.string().min(4).max(10).required()
})
```

Ahora, en el componente Formik cambiamos la función validate por validationSqueme y le pasamos el esquema. Si sacamos fuera la función que controla el envío nos quedarían como props de Formik :

```
<Formik
  initialValues={{ email: '', password: '' }}
  onSubmit={(values, {setSubmitting}) => handleSubmit(values, setSubmitting)}
  validationSchema={SignSquema} >
```

Por defecto yup nos va a devolver unos mensajes de error, si queremos definir nuestros propios mensajes simplemente debemos modificar el esquema para añadirlos:

```
const SignSquema = Yup.object({
  email: Yup.string().email('No es un email válido').required(),
  password: Yup.string().min(4, 'Minimo 4 caracteres').max(10, 'Máximo 10 caracteres').required()
})
```

Finalmente podríamos tener algo así:

```jsx
import { Formik, Form, Field, ErrorMessage } from 'formik'
import * as Yup from 'yup'

const SignSquema = Yup.object({
  email: Yup.string().email('No es un email válido').required(),
  password: Yup.string()
    .min(4, 'Minimo 4 caracteres')
    .max(10, 'Máximo 10 caracteres').required()
})

export default function ValidationForm() {

  const handleSubmit = (values, setSubmitting) => {
    setTimeout(() => {
      console.log(values)
      setSubmitting(false);
    }, 5000);
  }
  return (
    <Formik
      initialValues={{ email: '', password: '' }}
      onSubmit={(values, {setSubmitting}) => handleSubmit(values, setSubmitting)}
      validationSchema={SignSquema} >
      {
        ({isSubmitting}) => (
          <Form>
            <div className="form-group">
              <label htmlFor="email">Email</label>
              <Field id="email" name="email" type="email" />
              <ErrorMessage name="email" component="div" />
            </div>
            <div className="form-group">
              <label htmlFor="password">Password</label>
              <Field id="password" name="password" type="password" />
              <ErrorMessage name="password" component="div" />
            </div>
            <button type="submit" disabled={isSubmitting} >
              Submit
            </button>
          </Form>
        )
      }
    </Formik>
  )
}
```

React Hook Form

Instalación:

Para la instalación es necesario ejecutar:

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

Formulario básico

Lo primero que tenemos que hacer es importar el hook useForm:

```
import { useForm } from 'react-hook-form'
```

UseForm devuelve un objeto, de ese objeto vamos a obtener las propiedades o funciones que nos interesen. De momento vamos a empezar utilizando register y handleSubmit:

Con handleSubmit vamos a controlar el envío, para eso tenemos que pasarle a la función onSubmit del formulario handleSubmit(*Mifuncion)*, siendo *Mifuncion* la función que utilizaremos para controlar el envío.

```
<form onSubmit={handleSubmit(onSubmit)}>
```

Con register vamos a registrar todos los inputs. Al llamar a la función register pasándole una cadena devuelve un objeto con los métodos necesarios para controlar el input: name, ref, onChange, onBlur. Usando el spread operator podemos hacerlo de manera bastante concisa:

```
<input {...register('email')} placeholder="Introduce email"></input>
```

Nuestro primer ejemplo quedaría así:

```
import { useForm } from 'react-hook-form'

export default function SignForm () {

  const{register, handleSubmit} = useForm();

  function onSubmit(data) {
    console.log(data);
  }

  return (

    <form onSubmit={handleSubmit(onSubmit)}>
      <input {...register('email')} placeholder="Introduce email"></input>
      <input {...register('password')} placeholder="Introduce el password"></input>
      <button>Aceptar</button>
    </form>
  )
}
```

Reset

Si le damos al botón de envío deberíamos observar que en data tenemos un objeto con los valores, pero al terminar de enviar el formulario seguimos teniendo los inputs rellenos. Para evitar esto recogemos de useForm la funcion reset. Llamando a esa función conseguiremos vaciar lo formularios.

```
import { useForm } from 'react-hook-form'

export default function SignForm () {

  const{register, handleSubmit, reset} = useForm();

  function onSubmit(data) {
    console.log(data);
    reset();
  }

  return (

    <form onSubmit={handleSubmit(onSubmit)}>
      <input {...register('email')} placeholder="Introduce email"></input>
      <input {...register('password')} placeholder="Introduce el password"></input>
      <button>Aceptar</button>
    </form>
  )
}
```

Validaciones

Al igual que en formik podemos aplicar validaciones a los campos del formulario. Cuando hacemos el registro, podemos añadir un segundo parámetro con un objeto con las validaciones que queremos hacer. Por ejemplo:

```
<form onSubmit={handleSubmit(onSubmit)}>
  <input {...register('email', {required: true})} placeholder="Introduce email"></input>
  <input {...register('password', {required: true, minLength: 8})} placeholder="Introduce el password"></input>
  <button>Aceptar</button>
</form>
```

Para ver con detalle las opciones disponibles acceder a la documentación:

https://www.react-hook-form.com/api/useform/register/

Mensajes de error

Para dotar de mensajes de error por un lado utilizamos el objeto errors de la prop formState

```
const { register, handleSubmit, reset, formState: { errors } } = useForm();
```

Por otro lado añadimos mensajes a la validación. Para ello modificamos los objetos y en vez de directamente el valor tenemos para cada propiedad el campo value y el campo message. Para el campo required es suficiente cambiar el true por el mensaje:

```
<form onSubmit={handleSubmit(onSubmit)}>
  <input {...register('email', { required: 'Obligatorio' })} placeholder="Introduce email"></input>
  {errors.password &&  <p>{errors.password.message}</p>}
  <input {...register('password', {
    required: 'Obligatorio',
    minLength: {
      value: 8,
      message: 'el password debe ocupar mas de 8 caracteres '
  }})} placeholder="Introduce el password"></input>
  {errors.password &&  <p>{errors.password.message}</p>}
  <button>Aceptar</button>
</form>
```

Ahora como vemos en el ejemplo anterior podemos mostrar el mensaje a partir de la variable errors.

Yup

Al igual que con formik también podemos utilizar un esquema de yup para la validación. Esta vez además de instalar yup debemos instalar @hookform/resolvers.

Una vez que está instalado necesitamos importar tanto yup como yupResolver (que será la conexión entre react hook form y yup). Para ello:

```
'react-hook-form';
rom '@hookform/resolvers/yup';
import * as Yup from "yup";
```


A continuación definimos el esquema de Yup:

```
const SignSquema = Yup.object({
  email: Yup.string().email('No es un email válido').required(),
  password: Yup.string()
    .min(4, 'Minimo 4 caracteres')
    .max(10, 'Máximo 10 caracteres').required()
})
```

Y por último integramos el esquema en el formulario con el resolver:

```
const { register, handleSubmit, reset, formState: { errors } } = useForm({
  resolver:yupResolver(SignSquema)
});
```

El ejemplo completo sería:

```
import { useForm } from 'react-hook-form';
import { yupResolver } from '@hookform/resolvers/yup';
import * as Yup from "yup";

const SignSquema = Yup.object({
  email: Yup.string().email('No es un email válido').required(),
  password: Yup.string()
    .min(4, 'Minimo 4 caracteres')
    .max(10, 'Máximo 10 caracteres').required()
})

export default function SignForm() {

  const { register, handleSubmit, reset, formState: { errors } } = useForm({
    resolver:yupResolver(SignSquema)
  });

  function onSubmit(data) {
    console.log(data);
    reset();
  }

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <input {...register('email')} placeholder="Introduce email"></input>
      {errors.email &&  <p>{errors.email.message}</p>}
      <input {...register('password')} placeholder="Introduce el password"></input>
      {errors.password &&  <p>{errors.password.message}</p>}
      <button>Aceptar</button>
    </form>
  )
}
```