UTN FACULTAD REGIONAL CÓRDOBA
INGENIERÍA EN SISTEMAS DE INFORMACIÓN

Desarrollo de Software
Notas de Clase

Clase 24 - React: Formularios con React Hook Form

Para trabajar con formularios de una manera mas sencilla, vamos a utilizar la librería React Hook Form (https://react-hook-form.com/). Dicha librería se utiliza para simplificar y optimizar la creación y gestión de formularios en aplicaciones React, proporcionando herramientas para la validación, el manejo de errores y el rendimiento.

Esta librería utiliza hooks de React para gestionar la lógica del formulario y brindar un rendimiento optimizado.

Algunas de las principales ventajas y usos de React Hook Form incluyen:

1. **Validación de formularios**: React Hook Form facilita la validación de los campos de entrada utilizando reglas personalizadas o bibliotecas de validación como Yup o Joi.
2. **Rendimiento**: La librería es ligera y utiliza eventos de cambio mínimos para reducir las actualizaciones de componentes innecesarias, lo que mejora el rendimiento de la aplicación.
3. **Facilitar el manejo de errores**: React Hook Form simplifica el proceso de mostrar mensajes de error y gestionar los errores en el formulario.
4. **Control de formularios personalizado**: Puedes crear componentes personalizados de entrada y fácilmente conectarlos a la lógica de tu formulario con la ayuda de esta librería.
5. **Integración con bibliotecas de diseño**: React Hook Form se integra fácilmente con bibliotecas populares de diseño de componentes, como Material-UI, Ant Design y Bootstrap.
6. **Desarrollo rápido**: Al manejar la lógica de los formularios de una manera más simple y concisa, React Hook Form permite a los desarrolladores crear y mantener formularios en sus aplicaciones de manera más rápida y eficiente.

Componentes principales de react-hook-form

Sus componentes principales son hooks y funciones que facilitan la creación, validación y gestión de formularios. Algunos de los componentes principales de React Hook Form incluyen:

- **useForm**: Es el hook principal de React Hook Form. Se utiliza para inicializar la lógica del formulario y proporciona varias funciones y objetos útiles para gestionar campos de entrada, validación, errores y envío del formulario.
- **register**: Es una función que se obtiene del hook useForm. Se utiliza para registrar campos de entrada en el formulario y establecer reglas de validación. La función register también conecta los campos de entrada a la lógica del formulario, lo que permite rastrear su estado y validarlos.

- **handleSubmit**: Es otra función que se obtiene del hook useForm. Se encarga de manejar la lógica de envío del formulario y la validación de los campos. La función handleSubmit debe ser asignada al atributo onSubmit del elemento form.
- **formState**: Es un objeto que se obtiene del hook useForm y contiene información sobre el estado del formulario, como errores de validación, estado de envío y estado de modificación. La propiedad errors dentro de formState almacena los errores de validación de los campos y se puede utilizar para mostrar mensajes de error.
- **setError**: Es una función que se obtiene del hook useForm y permite establecer errores manualmente en campos específicos del formulario. Puede ser útil cuando se necesitan errores personalizados o errores que provienen de una fuente externa, como un servidor.
- **watch**: Es una función que se obtiene del hook useForm y permite observar y obtener el valor actual de uno o varios campos en tiempo real. Puede ser útil para implementar lógicas condicionales basadas en los valores de los campos de entrada.
- **reset**: Es una función que se obtiene del hook useForm y permite restablecer los valores y el estado de los campos del formulario a sus valores iniciales o a nuevos valores proporcionados.

Validaciones

React Hook Form realiza la validación de formularios intergrándos con los estándar de validación de formularios de HTML.

Reglas de validación soportadas:

- required
- min
- max
- minLength
- maxLength
- pattern
- validate

Ejemplo

Para poder utilizar esta librería se requiere instalar la misma en el proyecto de react:

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

Luego creamos un componente llamado MiFormulario.js con el siguiente código:

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

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

```
  const onSubmit = (data) => {
    console.log('Datos del formulario:', data);
  };

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <label htmlFor="name">Nombre:</label>
      <input
        id="name"
        type="text"
        {...register('name', { required: 'El campo nombre es requerido' })}
      />
      {errors.name && <p>{errors.name.message}</p>}
      <br/>
      <label htmlFor="email">Correo electrónico:</label>
      <input
        id="email"
        type="email"
        {...register('email', { required: 'El campo correo electrónico es
requerido' })}
      />
      {errors.email && <p>{errors.email.message}</p>}
      <br/>
      <button type="submit">Enviar</button>
    </form>
  );
}

export default MiFormulario;
```

Observe lo siguiente:

- La inicialización de useForm:

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

  Se utiliza el hook useForm para inicializar la lógica del formulario y obtener varias funciones y objetos útiles, como register, handleSubmit y errors.

- Función onSubmit: Esta función se ejecutará cuando el formulario se envíe correctamente (sin errores de validación). En este caso, simplemente imprime los datos del formulario en la consola

  ```
  const onSubmit = (data) => {
      console.log('Datos del formulario:', data);
  };
  ```

- Renderizado del formulario y campos de entrada: Se renderiza el elemento form con un atributo onSubmit que utiliza la función handleSubmit para manejar el envío del formulario. Se crean dos campos de entrada, "Nombre" y "Correo electrónico", y se utilizan las funciones register y errors para gestionar la validación y mostrar mensajes de error cuando corresponda.

```
return (
  <form onSubmit={handleSubmit(onSubmit)}>
    <label htmlFor="name">Nombre:</label>
    <input
      id="name"
      type="text"
      {...register('name', { required: 'El campo nombre es requerido' })}
    />
    {errors.name && <p>{errors.name.message}</p>}
    <br />
    <label htmlFor="email">Correo electrónico:</label>
    <input
      id="email"
      type="email"
      {...register('email', { required: 'El campo correo electrónico es requerido'
})}
    />
    {errors.email && <p>{errors.email.message}</p>}
    <br />
    <button type="submit">Enviar</button>
  </form>
);
```

- **...register**: los tres puntos significa la sintaxis de propagación (spread syntax) en JavaScript. La función register de React Hook Form devuelve un objeto que contiene varias propiedades y funciones necesarias para manejar la lógica del campo de formulario correspondiente. La sintaxis de propagación permite que todas las propiedades y funciones en el objeto devuelto por register se asignen automáticamente al elemento de entrada en el que se utiliza.

  ```
  {...register('email', { required: 'El campo correo electrónico es requerido'
  })}
  ```

  La función register en este en este caso específico, está registrando el campo de entrada 'email' con una regla de validación que indica que el campo es obligatorio.

- Para el **manejo de errores** React Hook Form proporciona un objeto **errors** para mostrar los errores en el formulario. El objeto *errors* devolverá los errores que se hayan producido dadas las restricciones de validación. El siguiente ejemplo muestra los errores de una regla de validación requerida y sino coincide el formato.

  ```
  {errors.email && <p>{errors.email.message}</p>}
  ```

Reglas de validación

La función register de React Hook Form permite aplicar diferentes reglas de validación para los campos de entrada. Algunas de las reglas de validación más comunes que puedes utilizar son las siguientes:

1. **required**: Indica que el campo de entrada es obligatorio. Puedes pasar un mensaje de error personalizado como valor.

```
{...register('name', { required: 'El campo nombre es requerido' })}
```

2. **minLength**: Establece la longitud mínima de caracteres que debe tener el campo de entrada. Debes proporcionar un objeto con la propiedad value (el número mínimo de caracteres) y la propiedad message (un mensaje de error personalizado).

```
{...register('password', { minLength: { value: 8, message: 'La contraseña
debe tener al menos 8 caracteres' } })}
```

3. **maxLength**: Establece la longitud máxima de caracteres que puede tener el campo de entrada. Al igual que minLength, debes proporcionar un objeto con las propiedades value y message.

```
{...register('username', { maxLength: { value: 20, message: 'El nombre de
usuario no puede tener más de 20 caracteres' } })}
```

4. **validate**: Permite aplicar una función de validación personalizada al campo de entrada. La función debe devolver true si la validación es exitosa, o un mensaje de error personalizado en caso contrario.

```
{...register('passwordConfirmation', { validate: (value) => value ===
password || 'Las contraseñas no coinciden' })}
```

Otro ejemplo de validate, podría ser para validar un email

```
const emailValidator = (email) => {
const emailRegex = /^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z]{2,4}$/i;
return emailRegex.test(email) || 'Ingrese un correo electrónico válido';
};
```

```
{...register('email', {
    required: 'El campo correo electrónico es requerido',
    validate: emailValidator,
})}
```

5. **pattern** Para validar un campo de entrada de correo electrónico, puedes utilizar la regla pattern con una expresión regular que verifique si el valor ingresado en el campo cumple con el formato de un correo electrónico.

```
{...register('email', {
required: 'El campo correo electrónico es requerido',
pattern: {
    value: /^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z]{2,4}$/i,
    message: 'Ingrese un correo electrónico válido',
},
})}
```

Estas son algunas de las reglas de validación más comunes que puedes utilizar con la función register de React Hook Form. Puedes combinar varias reglas para un único campo de entrada para aplicar múltiples validaciones. Además, también es posible crear funciones de validación personalizadas y más complejas si lo necesitas.

Expresiones regulares y pattern

Una expresión regular, también conocida como regex o regexp, es una secuencia de caracteres que define un patrón de búsqueda. Las expresiones regulares se utilizan principalmente para encontrar y manipular cadenas de texto que coincidan con dicho patrón. Las expresiones regulares son una herramienta muy poderosa y versátil en el procesamiento de texto y se utilizan en muchas tareas, como la validación de entradas, la búsqueda y sustitución de texto, y el análisis de datos, entre otros.

Las expresiones regulares están compuestas por caracteres literales (como letras, números y símbolos) y metacaracteres (símbolos especiales que tienen un significado particular en el contexto de la expresión regular). Estos metacaracteres permiten crear patrones más complejos y flexibles.

Algunos ejemplos de metacaracteres comunes en expresiones regulares son:

Algunos ejemplos de metacaracteres comunes en expresiones regulares son:

- **^**: Coincide con el inicio de la cadena.
- **$**: Coincide con el final de la cadena.
- **.**: Coincide con cualquier caracter, excepto un salto de línea.
- *****: Coincide con cero o más repeticiones del caracter o subpatrón anterior.
- **+**: Coincide con una o más repeticiones del caracter o subpatrón anterior.
- **?**: Coincide con cero o una repetición del caracter o subpatrón anterior.
- **[...]**: Define un conjunto de caracteres. Coincide con un caracter si está dentro del conjunto.
- **(...)**: Agrupa caracteres o subpatrones. Puede utilizarse para aplicar cuantificadores a varios caracteres o para capturar subcadenas coincidentes.

La mayoría de los lenguajes de programación, incluido JavaScript, proporcionan funciones y métodos para trabajar con expresiones regulares, lo que facilita su uso para buscar, extraer y manipular cadenas de texto que coincidan con un patrón específico.

Ejemplos sencillos de expresiones regulares:

- Coincidir con cualquier número de dígitos:

```
/\d+/;
```

  La expresión regular /\d+/ es una expresión que busca coincidencias con una secuencia de uno o más dígitos numéricos (0-9). Veamos en detalle cada parte de esta expresión:

  - /: Los caracteres / al inicio y al final de la expresión regular son delimitadores que indican el comienzo y el final de la expresión en JavaScript. No forman parte del patrón en sí.
  - \d: El metacaracter \d es un carácter de escape que representa un dígito numérico (0-9). Es equivalente al conjunto [0-9].
  - +: El metacaracter + es un cuantificador que indica una o más repeticiones del carácter o subpatrón inmediatamente anterior. En este caso, se aplica al metacaracter \d.

  Entonces, la expresión regular /\d+/ busca secuencias de uno o más dígitos numéricos en una cadena. Por ejemplo, coincidirá con cualquier número entero positivo, como "42" o "12345", pero no coincidirá con números que contengan puntos decimales, símbolos u otros caracteres no numéricos.

- Coincidir con una cadena que contiene solo letras y números (sin espacios ni símbolos):

```
/^[A-Za-z0-9]+$/;
```

  Veamos en detalle cada parte de esta expresión:

  - /: Los caracteres / al inicio y al final de la expresión regular son delimitadores que indican el comienzo y el final de la expresión en JavaScript. No forman parte del patrón en sí.
  - ^: El metacaracter ^ indica el inicio de la cadena. La expresión solo coincidirá si el patrón comienza al inicio de la cadena.
  - [A-Za-z0-9]: Este conjunto de caracteres coincide con cualquier letra mayúscula (A-Z), letra minúscula (a-z) o número (0-9).
  - +: El metacaracter + es un cuantificador que indica una o más repeticiones del carácter o subpatrón inmediatamente anterior. En este caso, se aplica al conjunto [A-Za-z0-9].
  - $: El metacaracter $ indica el final de la cadena. La expresión solo coincidirá si el patrón termina al final de la cadena.

formState

En React Hook Form, formState es un objeto que contiene información sobre el estado actual del formulario. Proporciona datos útiles sobre las interacciones del usuario y el estado de validación de los campos del

formulario. Puedes utilizar formState para realizar acciones o renderizar componentes condicionalmente según el estado del formulario.

Algunas de las propiedades más comunes de formState incluyen:

- **isDirty**: Indica si el usuario ha interactuado con alguno de los campos del formulario (es decir, si se han modificado los valores iniciales de los campos).
- **isValid**: Indica si todos los campos del formulario cumplen con las reglas de validación.
- **isSubmitting**: Indica si el formulario está actualmente en proceso de envío (es decir, si se ha llamado a la función handleSubmit y está en ejecución).
- **errors**: Es un objeto que contiene información sobre los errores de validación de cada campo. Puedes utilizar esta información para mostrar mensajes de error personalizados a los usuarios.

Para utilizar formState con React Hook Form, primero se debe desestructurar del objeto devuelto por la función useForm:

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

Luego, puedes acceder a las propiedades de formState en tu componente y utilizarlas según sea necesario. Por ejemplo, podrías deshabilitar un botón de envío mientras se envía el formulario:

```
<button type="submit" disabled={formState.isSubmitting}>Enviar</button>
```

O mostrar un mensaje de error personalizado si hay errores de validación en un campo específico:

```
{formState.errors.name && <p>{formState.errors.name.message}</p>}
```

El objeto formState es una herramienta útil para mejorar la experiencia del usuario

Por ejemplo:

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

function MiFormulario() {
  const { register, handleSubmit, formState } = useForm();
  const { isDirty, isValid, isSubmitting, isSubmitted, errors, touchedFields } =
formState;

  // Función que devuelve una promesa que se resuelve después de una demora
específica
  const sleep = (ms) => new Promise((resolve) => setTimeout(resolve, ms));

  const onSubmit = async (data) => {
    console.log('Datos del formulario:', data);
```

```
    // Simular una demora de 3 segundos usando await y sleep
    await sleep(3000);
    console.log('Formulario enviado después de 3 segundos');
  };

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <label htmlFor="name">Nombre:</label>
      <input
        id="name"
        type="text"
        {...register('name', { required: 'El campo nombre es requerido' })}
      />
      {errors.name && touchedFields.name && <p>{errors.name.message}</p>}
      <br />
      <label htmlFor="email">Correo electrónico:</label>
      <input
        id="email"
        type="email"
        {...register('email', { required: 'El campo correo electrónico es
requerido' })}
      />
      {errors.email && touchedFields.email && <p>{errors.email.message}</p>}
      <br />
      <button type="submit" disabled={!isValid || isSubmitting}>
        {isSubmitting ? 'Enviando...' : 'Enviar'}
      </button>
      {isDirty && isSubmitted && !isValid && <p>Por favor, completa correctamente
todos los campos.</p>}
    </form>
  );
}

export default MiFormulario;
```

En este ejemplo:

- **isDirty**: Se utiliza para verificar si el usuario ha interactuado con alguno de los campos del formulario. Si el formulario está "sucio" (isDirty === true) y no es válido (!isValid), se muestra un mensaje al usuario para que complete correctamente todos los campos.
- **isValid**: Se utiliza para habilitar o deshabilitar el botón de envío en función de si el formulario es válido o no. Si el formulario no es válido, el botón de envío estará deshabilitado.
- **isSubmitting**: Se utiliza para mostrar un texto diferente en el botón de envío mientras se procesa el envío del formulario. Si el formulario se está enviando, el texto del botón cambiará a "Enviando...".
- **errors**: Se utiliza para mostrar mensajes de error personalizados cuando hay errores de validación en los campos específicos.
- **isSubmitted**: Esta propiedad es un valor booleano que indica si el formulario se ha enviado al menos una vez. Es útil cuando deseas realizar acciones o mostrar mensajes condicionalmente después de que el usuario haya intentado enviar el formulario. Por ejemplo, podrías mostrar un mensaje de éxito después de que se haya enviado el formulario correctamente.