UNIDAD 5

Formularios y Validación en React

Parte I: El problema de capturar información del usuario

¿Por qué existen los formularios?

Una aplicación necesita permitir que el usuario proporcione información.

Ejemplos:

- Iniciar sesión
- Registrarse
- Realizar una compra
- Completar una encuesta
- Modificar un perfil
- Enviar un mensaje

¿Qué es un formulario?

Es un conjunto de controles que permiten capturar información proporcionada por el usuario.

Nombre: Ariel

Email: ariel@mail.com

Contraseña: ••••••••

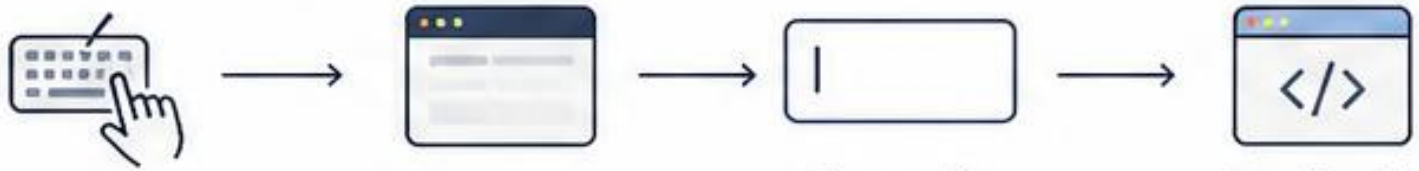
Iniciar sesión

Modelo conceptual

Idea clave: En los formularios tradicionales, el navegador administra el estado del formulario.

¿Qué sucede cuando escribimos?

Cuando un usuario escribe en un campo, ocurren varios eventos:



Usuario presiona tecla

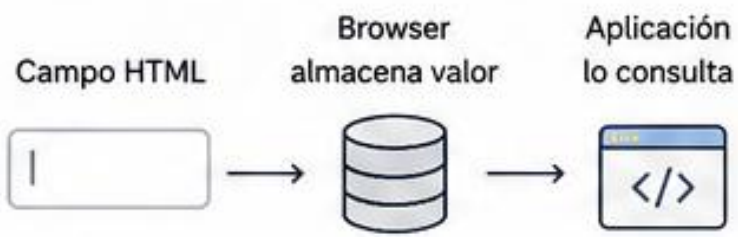
El navegador detecta el evento

El contenido del campo cambia

La aplicación puede leer el nuevo valor

Formularios en HTML tradicional

El navegador almacena el valor y la aplicación lo consulta cuando es necesario.



El problema en aplicaciones modernas

La aplicación necesita conocer constantemente qué está escribiendo el usuario para:

- Mostrar validaciones en tiempo real
- Contar caracteres
- Habilitar o deshabilitar botones
- Mostrar mensajes de error
- Calcular precios automáticamente
- Actualizar información en pantalla

Si React se basa en el estado, ¿cómo hacemos para que los datos del formulario formen parte de ese estado?

Dos enfoques diferentes

FORMULARIO TRADICIONAL (HTML)

ENFOQUE EN REACT

Idea clave

React propone que toda la información que afecta a la interfaz (incluida la que escribe el usuario) sea parte del estado de la aplicación.

UI = f(estado)

PARTE II

Inputs Controlados y No Controlados

BLOQUE 1

El problema: ¿Quién administra el valor?

Cuando un usuario escribe en un campo de un formulario, el valor cambia. Pero surge una pregunta clave:

¿Dónde se almacena ese valor?
¿Quién tiene el control?

En React existen dos enfoques posibles.
Antes de ver las soluciones, entendamos el problema.

¿Qué ocurre cuando el usuario escribe?

1. Usuario
2. Escribe en el campo de texto
3. El valor del campo cambia
4. ¿Quién almacena ese valor?

Pregunta clave: ¿El navegador guarda ese valor?
¿O es la aplicación (React) quien lo administra?

Ejemplo visual

Nombre: Ariel

El usuario está escribiendo...

Pero aún no sabemos dónde se guarda ese valor.

¿Por qué es importante resolver este problema?

- Para validar datos en tiempo real
- Para mostrar mensajes de error
- Para contar caracteres o calcular valores
- Para habilitar o deshabilitar botones
- Para guardar información

Objetivo de esta sección

Entender las dos estrategias disponibles en React para manejar formularios:
inputs no controlados (administrados por el navegador) e inputs controlados (administrados por el estado de React).

PARTE II

Inputs Controlados y No Controlados

BLOQUE 2

Formulario tradicional: el navegador administra el valor

En HTML tradicional (sin React), el valor que escribe el usuario es almacenado y controlado por el navegador.

Ejemplo: Formulario HTML tradicional

```
1  <form>
2    <label>Nombre:</label>
3    <input type="text" name="nombre"
4      placeholder="Escribe tu nombre" />
5    <br /><br />
7    <button type="submit">Enviar</button>
9  </form>
```

¿Cómo funciona?

1. Usuario escribe en el campo
2. El navegador detecta el evento y actualiza el campo
3. El valor se almacena dentro del navegador (no en React)
4. El valor se consulta recién cuando se envía el formulario

Hasta que el usuario presiona “Enviar”, la aplicación no tiene acceso directo al valor.

¿Dónde se almacena el valor?

El usuario escribe “Ariel” → El navegador guarda ese valor internamente → La aplicación lo recibe solo al enviar el formulario

React todavía no participa en este proceso.

Ventajas

- Código simple.
- Se comporta como cualquier formulario tradicional.
- Útil para formularios básicos o páginas simples.

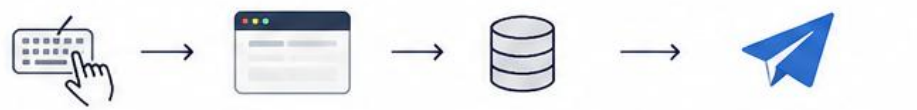
Limitaciones

- No podemos validar mientras el usuario escribe.
- No podemos mostrar contadores en tiempo real.
- No podemos habilitar/deshabilitar botones dinámicamente.
- No podemos mostrar mensajes de error inmediatamente.
- La aplicación no “conoce” el valor hasta el envío.

Idea clave

En los formularios tradicionales, el navegador administra el estado del formulario.

Resumen visual del flujo



Usuario escribiendo → Navegador actualiza el campo → Valor guardado en el navegador → Valor enviado a la aplicación al hacer submit

En React existe otra forma mejor

Para lograr aplicaciones interactivas y reactivas, React propone que sea el estado quien administre el valor. Esto lo veremos en el siguiente bloque **(inputs controlados)**.

PARTE II

Inputs Controlados y No Controlados

BLOQUE 3

Formularios modernos: la aplicación necesita conocer el valor

Las aplicaciones modernas no solo capturan datos al final del formulario.
Necesitan reaccionar mientras el usuario escribe.

El problema

En un formulario tradicional (no controlado), el valor queda dentro del navegador.

La aplicación no puede conocerlo hasta que el usuario envía el formulario.

Esto limita muchas funcionalidades importantes.

El valor permanece en el navegador

¿Qué necesitan las aplicaciones modernas?

Validaciones en tiempo real

Verificar si el dato es correcto mientras el usuario escribe.

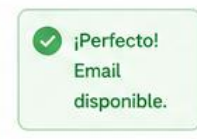
Contadores en tiempo real

Mostrar la cantidad de caracteres ingresados.

Habilitar / deshabilitar botones

Activar el botón solo cuando el formulario es válido.

Mensajes y feedback inmediato



Dar feedback instantáneo para mejorar la experiencia del usuario.

Cálculos y resultados dinámicos

Actualizar valores, precios o resultados según los datos ingresados.

Para lograr todo esto, la aplicación debe conocer el valor del campo en todo momento.
Necesitamos que React sea quien administre ese valor.

Comparación visual

Formulario tradicional (no controlado)

Usuario escribe → Valor en el campo → Guardado en el navegador → Se obtiene solo al enviar

La aplicación NO tiene acceso al valor mientras se escribe.

Formulario moderno (controlado)

Usuario escribe → Valor en el campo → React actualiza el estado → La aplicación conoce el valor al instante

La aplicación **Sí** conoce el valor en tiempo real.

Ejemplo práctico: Login

- Validación de email en tiempo real
- Indicador de fuerza de contraseña
- Botón **INICIAR SESIÓN** se habilita cuando todo es válido
- Mensajes de feedback inmediato

Solo se habilita cuando el formulario es válido

Idea clave de este bloque

Las aplicaciones modernas necesitan conocer el valor de los campos en todo momento para poder validar, calcular, mostrar mensajes y mejorar la experiencia del usuario. Por eso, React propone usar **inputs controlados**.

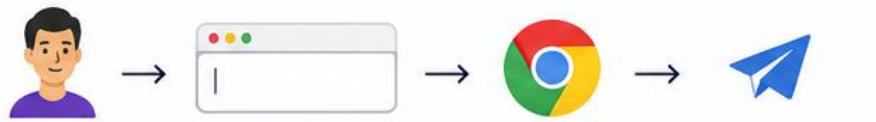
PARTE II

Inputs Controlados y No Controlados

BLOQUE 4

Comparación visual: ¿Quién administra el valor?

INPUT NO CONTROLADO
(El navegador administra el valor)



1. Usuario escribe
2. Campo de texto (HTML)
3. El navegador guarda el valor
4. La aplicación obtiene el valor al enviar (submit)

¿Dónde vive el valor?

Dentro del navegador (DOM).
React no lo conoce mientras el usuario escribe.

Ejemplo de código (HTML tradicional)

```
<form>
  <input type="text" name="nombre" />
  <button type="submit">Enviar</button>
</form>

```

Idea clave

El valor está en el DOM. React no puede usarlo hasta que se envía el formulario.

INPUT CONTROLADO
(React administra el valor mediante estado)

1. Usuario escribe
2. onChange se dispara
3. El estado de React se actualiza
4. React vuelve a renderizar
5. El input muestra el nuevo valor

¿Dónde vive el valor?

Dentro del estado de React.
React lo conoce y puede usarlo en todo momento.

Ejemplo de código (React - Input controlado)

```
const [nombre, setNombre] = useState("");

<input
  type="text"
  value={nombre}
  onChange={(e) => setNombre(e.target.value)}
/>
```

Idea clave

El valor está en el estado de React. Por eso la aplicación puede usarlo en cualquier momento.

Idea clave del bloque

La diferencia fundamental no está en el campo de texto. sino en quién administra el valor.

NO CONTROLADO

El navegador guarda el valor. La aplicación no lo conoce hasta el envío.

VS

CONTROLADO

React guarda el valor en su estado. La aplicación lo conoce y puede usarlo en tiempo real.

Objetivo

Usaremos inputs controlados para poder validar, mostrar mensajes, habilitar botones y mejorar la experiencia del usuario.

PARTE II

Inputs Controlados y No Controlados

BLOQUE 5

Primer input controlado en React

Veamos cómo crear nuestro primer campo controlado.

Login.jsx

```jsx
import { useState } from "react";

const Login = () => {
  const [nombre, setNombre] = useState("");

  return (
    <div>
      <label>Nombre:</label>
      <input
        type="text"
        value={nombre}
        onChange={(e) => setNombre(e.target.value)}
      />
    </div>
  );
};
```

1 Estado

Creamos una variable de estado llamada `nombre`. Inicialmente está vacía (“”).

2 value={nombre}

El valor que se muestra en el input proviene del estado.

3 onChange

Cuando el usuario escribe, actualizamos el estado con el nuevo valor del input.

¿Qué vemos en pantalla inicialmente?

Componente Login (resultado inicial)

Nombre: [ ]

Idea clave

El input no tiene un valor propio guardado por el navegador.
Su valor depende del estado de React.

Relación entre estado e input

Estado React		Input de texto
`nombre = ""` (useState)	⟷ Sincronizados en todo momento	`value={nombre}` (lo que se muestra)

El estado es la única fuente de verdad (single source of truth).

Resultado visual del componente

1. Render inicial — Nombre: [ ] — `nombre = ""`
→ 2. El usuario escribe “A” — Nombre: [A] — `nombre = "A"`
→ 3. El usuario escribe “Ari” — Nombre: [Ari] — `nombre = "Ari"`
→ 4. El usuario escribe “Ariel” — Nombre: [Ariel] — `nombre = "Ariel"`

Observa

Lo que el usuario ve en el input siempre refleja el valor actual del estado `nombre`.

Idea clave del bloque

En un input controlado, el valor mostrado en pantalla proviene del estado de React (`value={nombre}`) y se actualiza cada vez que el usuario escribe (onChange → `setNombre(e.target.value)`).

PARTE II

Inputs Controlados y No Controlados

BLOQUE 6

Flujo interno de un input controlado en React

Veamos qué ocurre paso a paso desde que el usuario escribe hasta que la interfaz se actualiza.

Código de ejemplo (Login.jsx)

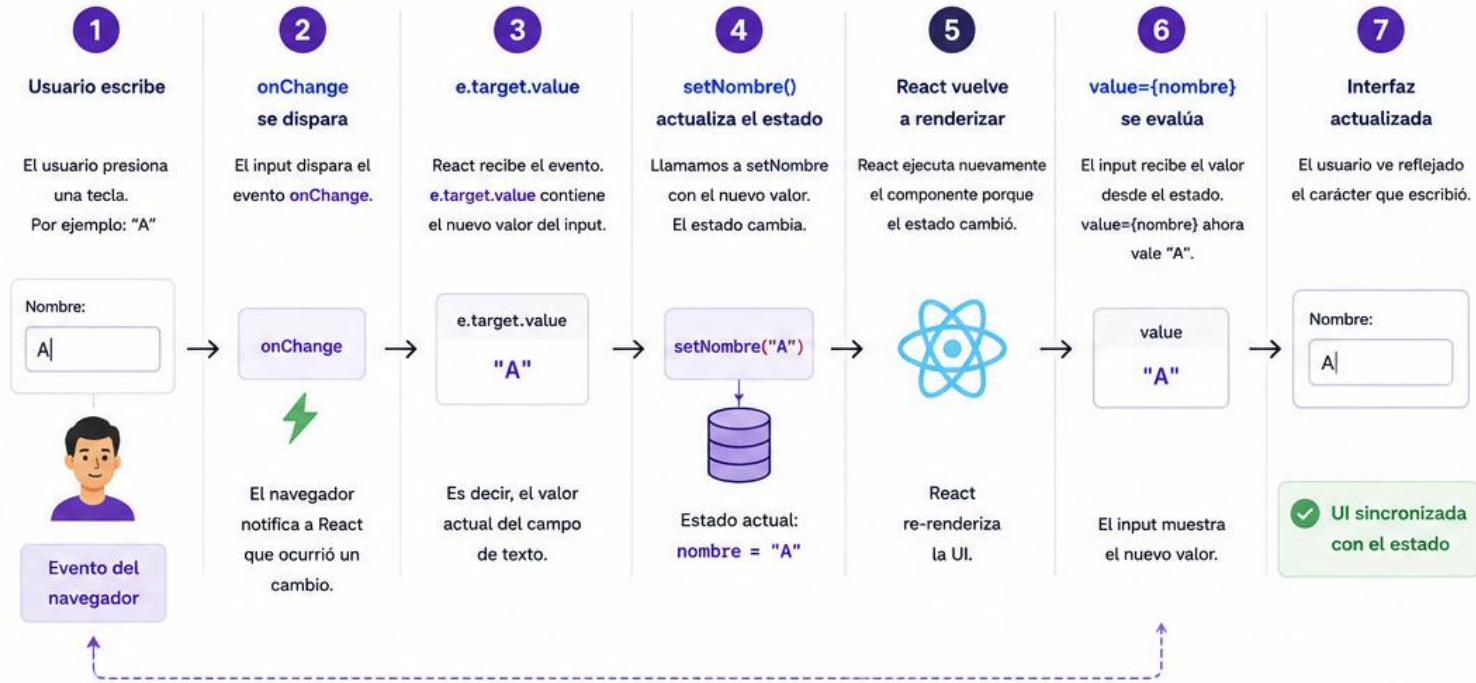
```jsx
import { useState } from "react";

const Login = () => {
  const [nombre, setNombre] = useState("");

  return (
    <div>
      <label>Nombre:</label>
      <input
        type="text"
        value={nombre}
        onChange={(e) => setNombre(e.target.value)}
      />
    </div>
  );
};
```

Idea clave

El input está "controlado" por el estado `nombre`.
La interfaz es una función del estado:

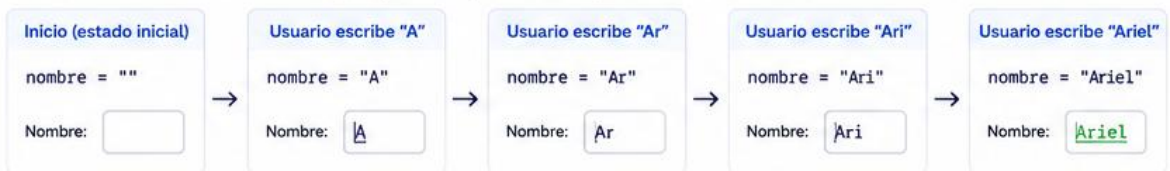
`UI = f(estado)`

¿Qué sucede cuando el usuario escribe "Ariel"?



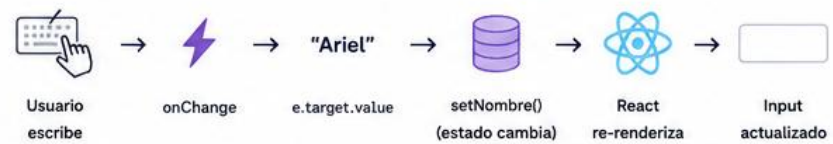
Este ciclo se repite por cada carácter que el usuario escribe.

Ejemplo completo del flujo escribiendo "Ariel"



En todo momento, el valor mostrado en el input proviene del estado y React es quien decide qué se muestra en pantalla.

Resumen del flujo (visión general)



Idea clave

En un input controlado, el estado es la única fuente de verdad.
El usuario escribe → el estado cambia → React vuelve a renderizar → la UI se actualiza.

PARTE II

Inputs Controlados y No Controlados

BLOQUE 7

Aplicación real: React utiliza el estado para construir experiencias dinámicas

Veamos un ejemplo completo donde un único input controla varios elementos de la interfaz.

Código del componente (Login.jsx)

```jsx
import { useState } from "react";

const Login = () => {
  const [usuario, setUsuario] = useState(""); // 1

  return (
    <div className="login">
      <label>Usuario:</label>
      <input
        type="text"
        value={usuario} // 2
        onChange={(e) => // 3
          setUsuario(e.target.value)
        }
      />
      <p>Usuario actual: <strong>{usuario}</strong></p>
    </div>
  );
};
```

1. Creamos el estado `usuario`. Inicialmente está vacío.
2. El valor del input proviene del estado `usuario`.
3. Cuando el usuario escribe, actualizamos el estado con el nuevo valor.

¿Qué ocurre cuando el usuario escribe?

Usuario escribe:	Usuario: / Usuario actual:	Estado React
(inicio)		usuario = ""
A	A / Usuario actual: A	usuario = "A"
An	An / Usuario actual: An	usuario = "An"
Ana	Ana / Usuario actual: Ana	usuario = "Ana"

Este ciclo se repite por cada carácter que el usuario escribe.
La interfaz siempre refleja el estado actual.

Gracias al estado, la aplicación puede actualizar muchos elementos a la vez

1. Mostrar el valor actual
   - Usuario actual: Ana
2. Contar caracteres
   - Caracteres: 3
3. Validar longitud mínima (mínimo 3)
   - Longitud válida
4. Habilitar botón según el estado
   - Iniciar sesión
   - (activo porque hay contenido)

Todos estos elementos se actualizan en el mismo instante porque dependen del mismo estado: `usuario`.

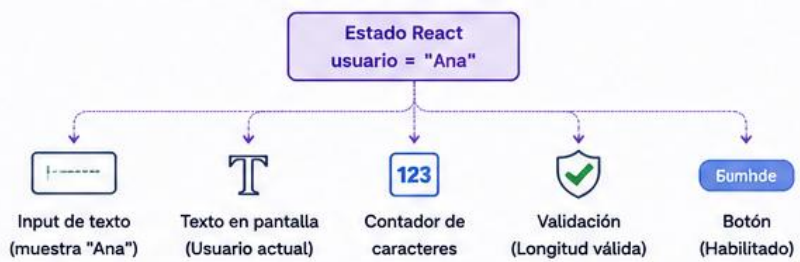
Idea clave del bloque

Los inputs controlados no solo permiten capturar datos.

Permiten que toda la aplicación reaccione inmediatamente a los cambios realizados por el usuario.

Por eso constituyen la base de los formularios modernos en React.

Un único estado puede actualizar múltiples partes de la interfaz



Síntesis final de la Parte II

INPUT NO CONTROLADO

- Usuario
- Campo
- Browser administra valor

INPUT CONTROLADO

- Usuario
- onChange
- Estado React
- Render (re-renderizado)
- Aplicación reacciona

La interfaz responde y se actualiza automáticamente

Los inputs controlados nos permiten construir interfaces interactivas, reactivas y dinámicas. Inputs controlados → Formularios completos → onSubmit → Validaciones → Formik → Yup

PARTE III

Manejo de eventos y envío de formularios

BLOQUE 1

El problema del envío

En la sección anterior aprendimos a capturar los datos del usuario y a mantenerlos sincronizados en el estado de React.

IDEA CLAVE

Capturar datos ≠ Procesar datos

Un formulario no existe solo para que el usuario escriba información.

El objetivo final es hacer algo útil con esos datos.

Nuestro nuevo objetivo: entender qué ocurre cuando el usuario decide enviar el formulario y cómo React toma el control para procesar la información sin recargar la página.

PARTE III

Manejo de eventos y envío de formularios

BLOQUE 2

El comportamiento tradicional de HTML

En HTML tradicional, los formularios fueron diseñados para enviar información al servidor y obtener una nueva página como respuesta.

Ejemplo de formulario HTML

```
<form action="/login" method="POST">

  <label>Usuario:</label>
  <input type="text" name="usuario" />

  <label>Contraseña:</label>
  <input type="password" name="clave" />

  <button type="submit">Enviar</button>
</form>
```

¿Qué ocurre cuando el usuario presiona "Enviar"?

El navegador interpreta que debe enviar el formulario al servidor indicado en el atributo `action`.

Idea clave

Este comportamiento es completamente válido y fue el estándar de la Web durante muchos años.
Sin embargo, implica que la página se recarga por completo cada vez que se envía un formulario.

Flujo tradicional de un formulario HTML

PARTE III

Manejo de eventos y envío de formularios

BLOQUE 3

El problema en una aplicación React (SPA)

En una Single Page Application (SPA), no queremos que la página se recargue cada vez que el usuario envía un formulario.

El comportamiento tradicional del navegador **entra en conflicto con la filosofía de React.**

HTML tradicional

Usuario completa el formulario

↓

El navegador envía el formulario

↓

El servidor procesa la información

↓

El navegador carga una nueva página

La página se recarga por completo

React (SPA)

Usuario completa el formulario

↓

React captura el evento `submit`

↓

La aplicación procesa la información

↓

La página se actualiza sin recargarse

La aplicación continúa activa

¿Qué significa SPA?

SPA (Single Page Application) es una aplicación web que funciona en una **única página** HTML.

Todo sucede dentro de esa página **sin recargas completas**, mejorando la experiencia del usuario.

Idea clave

Para que React pueda decidir qué hacer con los datos del formulario, necesitamos **tomar el control del envío** y evitar que el navegador recargue la página.

En los próximos bloques veremos cómo React logra ese control.

PARTE III

Manejo de eventos y envío de formularios

BLOQUE 4

El evento onSubmit

Para detectar cuando el usuario intenta enviar un formulario, utilizamos el evento **onSubmit** del formulario.

¿Dónde se usa?

```jsx
function Login() {
  const manejarEnvio = (e) => {
    // Código para procesar el formulario
  };

  return (
    <form onSubmit={manejarEnvio}>
      <input type="text" name="usuario" />
      <input type="password" name="clave" />
      <button type="submit">Ingresar</button>
    </form>
  );
}
```

Cuando el formulario sea enviado, ejecutar la función **manejarEnvio**.

¿Cómo funciona?

1. El usuario presiona el botón de envío.

2. El formulario genera el evento **submit**.

3. React detecta el evento **submit** en el formulario.

4. React ejecuta la función asociada a **onSubmit**.

5. La función manejarEnvio() se ejecuta.

Puntos clave

- El evento **onSubmit** pertenece **al formulario** completo (**<form>**), no al botón.
- Cualquier elemento dentro del formulario (un botón tipo **submit** o presionar **Enter** en un **input**) puede disparar el evento **submit**.
- **onSubmit** es el punto de entrada para procesar la información del formulario en React.

Importante

`onSubmit` solo detecta el envío. **Todavía no evita la recarga de la página.** Eso lo haremos en el siguiente bloque con `preventDefault()`.

Resumen del flujo

Idea clave

onSubmit nos permite **tomar el control** del envío del formulario y decidir qué hacer con los datos en lugar de dejar que el navegador recargue la página.

PARTE III

Manejo de eventos y envío de formularios

BLOQUE 5

El problema de la recarga automática

Aunque ya usamos **onSubmit** para detectar el envío del formulario,
todavía existe un **comportamiento heredado de HTML.**

¿Qué ocurre si no usamos `preventDefault()`?

1. El usuario presiona el botón **Enviar.**
2. El formulario genera el evento **submit.**
3. React ejecuta la función asociada a `onSubmit`.
4. El navegador continúa con su **comportamiento predeterminado: enviar el formulario.**
5. El navegador **recarga la página completa.**

React pierde el control de la experiencia.

¿Cuál es el resultado?

Problemas que genera:

- Se pierde el estado de la aplicación.
- El usuario ve una recarga completa.
- La experiencia se interrumpe.
- No podemos mostrar mensajes ni **validar antes de enviar.**

Necesitamos evitar este comportamiento para que React pueda procesar la información sin recargar la página.

Idea clave

El navegador tiene una acción automática cuando se envía un formulario.
Si queremos que React tome el control, debemos detener esa acción.

Objetivo del siguiente bloque: usar `preventDefault()` para evitar la recarga y permitir que React decida qué hacer con los datos del formulario.

```
e.preventDefault();
```

PARTE III

BLOQUE 6

Manejo de eventos y envío de formularios

preventDefault(): React toma el control

Para evitar la recarga automática de la página, usamos `e.preventDefault()` dentro de la función **onSubmit**.

¿Qué hace preventDefault()?

Le indica al navegador que **NO** ejecute su comportamiento predeterminado. En un formulario, ese comportamiento es: enviar los datos y recargar la página.

Ejemplo de código

```
1  function manejarEnvio(e) {
2    e.preventDefault();
3
4    // Aquí procesamos los datos
5    console.log('Formulario enviado');
6    // Validar, enviar a API, mostrar mensaje, etc.
7  }
8
```

Con esta línea, evitamos que el navegador recargue la página.

SIN preventDefault()

1. Usuario presiona **Enviar**
2. React ejecuta **onSubmit**
3. El navegador ejecuta su comportamiento predeterminado
4. Se envían los datos al servidor
5. La página se recarga completa

React pierde el control.
La experiencia se interrumpe.

CON preventDefault()

1. Usuario presiona **Enviar**
2. React ejecuta **onSubmit**
3. `e.preventDefault()` evita el comportamiento predeterminado
4. React mantiene el control de la aplicación
5. Procesamos la información (validar, enviar a API, mostrar mensaje, etc.)
6. La página permanece activa y actualizamos solo lo necesario

React mantiene el control.
La experiencia es fluida.

Comparación rápida

Sin preventDefault()
El navegador recarga la página.

VS

Con preventDefault()
React procesa los datos sin recargar.

Idea clave

Con preventDefault(), React puede decidir qué hacer con la información y ofrecer una mejor experiencia al usuario.

PARTE III

Manejo de eventos y envío de formularios

BLOQUE 7

Flujo completo de un formulario en React

Un formulario en React combina captura de datos, detección del envío, control del comportamiento y procesamiento personalizado.

1. Captura de datos (en tiempo real)

1. Usuario escribe en el input
2. `onChange` se ejecuta en cada cambio
3. React actualiza el estado
4. El componente se vuelve a renderizar con el nuevo valor

2. Envío y procesamiento

5. Usuario presiona el botón Enviar
6. El formulario genera el evento `submit`
7. React detecta `onSubmit` y ejecuta la función
8. `e.preventDefault()` evita la recarga de la página
9. Procesamos la información: validar, enviar a API, mostrar mensaje, etc.
10. Mostramos el resultado y la página permanece activa

Conceptos clave integrados

- **onChange**: captura cada cambio del usuario.
- **Estado React**: mantiene los datos sincronizados.
- **onSubmit**: detecta el momento del envío.
- **preventDefault()**: evita la recarga de la página.
- **Procesamiento**: React decide qué hacer con la información.

Ejemplo de código completo

```jsx
import { useState } from "react";

const Login = () => {
  const [usuario, setUsuario] = useState("");
  const [mensaje, setMensaje] = useState("");

  const manejarEnvio = (e) => {
    e.preventDefault();

    if (usuario.trim() === "") {
      setMensaje("El usuario es obligatorio");
      return;
    }

    // Aquí podríamos llamar a una API
    setMensaje(`Bienvenido, ${usuario}`);
  };

  return (
    <form onSubmit={manejarEnvio}>
      <input
        type="text"
        value={usuario}
        onChange={(e) => setUsuario(e.target.value)}
        placeholder="Usuario"
      />
      <button type="submit">Ingresar</button>
      {mensaje && <p className="mensaje">{mensaje}</p>}
    </form>
  );
};
```

Resumen visual del flujo completo

Escribe → onChange → Estado React → Render

Presiona Enviar → submit → onSubmit → preventDefault() → Procesamiento → Resultado

Idea clave

React captura los datos mientras el usuario escribe, detecta el envío del formulario, evita la recarga de la página y decide qué hacer con la información para ofrecer **una experiencia fluida y controlada.**

¿Para qué sirve todo esto?

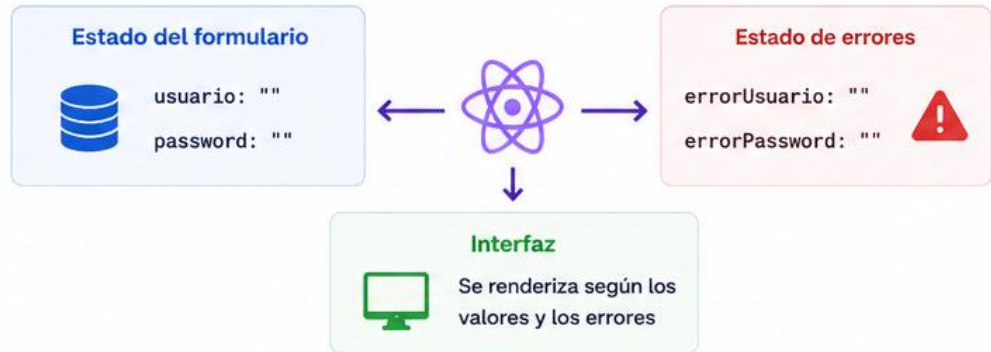
Nos permite validar datos, consultar APIs, mostrar mensajes y actualizar la interfaz sin salir nunca de nuestra aplicación.

PARTE IV

Validación de formularios en React

1 Validar en React: datos y errores como estado

En React, tanto los valores del formulario como los errores se guardan en el estado. La interfaz se actualiza automáticamente según esos estados.



Ejemplo básico

```jsx
import { useState } from "react";

function Login() {
  const [usuario, setUsuario] = useState("");
  const [password, setPassword] = useState("");
  const [errorUsuario, setErrorUsuario] = useState("");
  const [errorPassword, setErrorPassword] = useState("");

  return (
    <div>
      {/* El formulario irá aquí */}
    </div>
  );
}
```

Idea clave

React renderiza en función del estado. Si el estado cambia, la interfaz cambia.

Guardamos los datos y los errores en el estado para poder mostrarlos, ocultarlos y decidir qué hacer con ellos.

2 Validación durante la escritura con onChange

Podemos validar mientras el usuario escribe.
Así mostramos los errores en tiempo real y mejoramos la experiencia.

Usuario

a

El usuario debe tener al menos 3 caracteres

Contraseña

••

La contraseña debe tener al menos 6 caracteres

¿Qué ocurre aquí?

1. El usuario escribe en el input.
2. **onChange** actualiza el estado.
3. Se ejecuta la validación.
4. El estado de errores cambia.
5. React muestra u **oculta** el mensaje.

Ejemplo de validación en onChange

```jsx
const [usuario, setUsuario] = useState("");
const [errorUsuario, setErrorUsuario] = useState("");

const validarUsuario = (valor) => {
  if (valor.trim() === "") {
    setErrorUsuario("El usuario es obligatorio");
  } else if (valor.length < 3) {
    setErrorUsuario("El usuario debe tener al menos 3 caracteres");
  } else {
    setErrorUsuario("");
  }
};

// En el input:
<input
  type="text"
  value={usuario}
  onChange={(e) => {
    const valor = e.target.value;
    setUsuario(valor);
    validarUsuario(valor); // validamos mientras escribe
  }}
  placeholder="Usuario"
/>
{errorUsuario && <p className="error">{errorUsuario}</p>}
```

Beneficios

- El usuario recibe retroalimentación inmediata.
- Se evitan envíos con datos incorrectos.
- La interfaz refleja siempre el estado actual.

Validar en onChange no reemplaza la validación al enviar, pero mejora mucho la experiencia del usuario.

3 Validación al enviar con onSubmit

También podemos validar cuando el usuario intenta enviar el formulario.
Aquí revisamos todos los campos antes de procesar los datos.

Flujo al enviar

Ejemplo de validación en onSubmit

```jsx
const [usuario, setUsuario] = useState('');
const [password, setPassword] = useState("");
const [errores, setErrores] = useState({});

const manejarEnvio = (e) => {
  e.preventDefault();

  const nuevosErrores = {};

  if (usuario.trim() === "") {
    nuevosErrores.usuario = "El usuario es obligatorio";
  }
  if (password.length < 6) {
    nuevosErrores.password =
      "La contraseña debe tener al menos 6 caracteres";
  }
  setErrores(nuevosErrores);

  // Si no hay errores, seguimos con el envío
  if (Object.keys(nuevosErrores).length === 0) {
    console.log("Formulario válido, enviamos los datos");
    // Aquí iría la lógica para enviar datos (API, etc.)
  }
};
```

¿Qué hacemos?

1. Creamos un objeto para guardar los errores.
2. Validamos cada campo y agregamos su error si corresponde.
3. Actualizamos el estado de errores.
4. Si no hay errores, procesamos los datos.

Ventajas

- ✓ Comprobamos todo antes de enviar.
- ✓ Evitamos envíos con datos incorrectos.

Validar al enviar es fundamental para asegurar que los datos sean correctos antes de procesarlos.

4 Renderizado condicional de mensajes de error

React nos permite mostrar mensajes de error solo cuando existen.
Usamos renderizado condicional para mantener la interfaz limpia.

Ejemplo de interfaz

Idea clave

Solo mostramos los mensajes cuando existe un error para ese campo.

Cómo lo hacemos en React

```jsx
{/* Usuario */}
<input
  type="text"
  value={usuario}
  onChange={(e) => setUsuario(e.target.value)}
  placeholder="Usuario"
/>
{errores.usuario && (
  <p className="error">{errores.usuario}</p>
)}

{/* Contraseña */}
<input
  type="password"
  value={password}
  onChange={(e) => setPassword(e.target.value)}
  placeholder="Contraseña"
/>
{errores.password && (
  <p className="error">{errores.password}</p>
)}
```

`{errores.usuario && (...)}`

Si existe `errores.usuario` (mensaje de error para el usuario), mostramos el `<p>`.

`{errores.password && (...)}`

Si existe `errores.password` (mensaje de error para la contraseña), mostramos el `<p>`.

Sin este patrón

Tendríamos que mostrar siempre el `<p>` y usar estilos para ocultarlo, lo que ensucia la interfaz.

El renderizado condicional nos ayuda a mostrar solo lo necesario y mejora la experiencia del usuario.

5 Habilitar o deshabilitar botones según validez

Podemos usar el estado de errores para decidir si el formulario es válido.
Así evitamos envíos innecesarios y guiamos al usuario.

Idea clave

El botón refleja el estado del formulario.

Si hay errores, no permitimos enviar.

Ejemplo en React

```jsx
const [email, setEmail] = useState("");
const [password, setPassword] = useState("");
const [errores, setErrores] = useState({});

const validar = () => {
  const nuevosErrores = {};
  if (!email.trim()) nuevosErrores.email = "El email es obligatorio";
  else if (!/^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(email))
    nuevosErrores.email = "Email inválido";

  if (password.length < 6)
    nuevosErrores.password = "La contraseña debe tener al menos 6 caracteres";

  setErrores(nuevosErrores);
  return Object.keys(nuevosErrores).length === 0;
};

const esValido = Object.keys(errores).length === 0;
```

¿Qué hacemos?

1. Validamos y obtenemos los errores.
2. Si no hay errores, el formulario es válido.
3. Usamos esa información para habilitar o deshabilitar el botón.

En el JSX

```jsx
<button type="submit" disabled={!esValido}>
  Crear cuenta
</button>
```

Ventajas

- Mejora la experiencia del usuario.
- Evita envíos con datos incorrectos.
- Da feedback inmediato.

Usar el estado de errores para controlar el botón hace que la interfaz sea más clara y el formulario más robusto.

6 Validación de varios campos

En formularios reales tenemos varios campos y cada uno puede tener sus reglas.
Podemos manejar los valores y los errores usando objetos en el estado.

Ejemplo de formulario

Reglas de validación

- **Nombre:** obligatorio y mínimo 2 caracteres.
- **Email:** obligatorio y con formato válido.
- **Edad:** debe ser un número mayor que 0.

Ejemplo en React

```
const [datos, setDatos] = useState({
   nombre: "",
   email: "",
   edad: ""
});
const [errores, setErrores] = useState({});

const manajarChange = (e) => {
   const { name, value } = e.target;
   setDatos({ ...datos, [name]: value });
};

const validar = () => {
  const nuevosErrores = {};

  if (!datos.nombre.trim() || datos.nombre.trim().length < 2)
    nuevosErrores.nombre = "El nombre debe tener al menos 2 caracteres";

  if (!datos.email.trim())
    nuevosErrores.email = "El email es obligatorio";
  else if (!/^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(datos.email))
    nuevosErrores.email = "Email inválido";

  const edadNum = Number(datos.edad);
  if (isNaN(edadNum) || edadNum <= 0)
    nuevosErrores.edad = "La edad debe ser mayor que 0";

  setErrores(nuevosErrores);
  return Object.keys(nuevosErrores).length === 0;
};
```

¿Qué logramos?

- Mantenemos todos los valores en un solo objeto.
- Mantenemos todos los errores en otro objeto.
- Cada campo tiene su propia regla.
- El formulario escala mejor a medida que crece.

Idea clave

Organizar valores y errores en objetos nos permite manejar múltiples campos de forma ordenada.

La validación por campo nos permite dar mensajes específicos y mantener el formulario claro y fácil de mantener.

PARTE V

Formularios profesionales con Formik y Yup

BLOQUE 1 El problema que intentan resolver Formik y Yup

La validación manual funciona bien en formularios simples, pero se vuelve difícil de mantener cuando el formulario crece.

Formulario simple (login)

Usuario

ej: ana

Contraseña

•••••••

Ingresar

Con React puro

- useState(usuario)
- useState(password)
- errorUsuario
- errorPassword
- onChange
- onSubmit
- validar()

Resultado

- Fácil de entender
- Fácil de mantener

La aplicación crece...

- Más campos
- Más validaciones
- Más reglas

Formulario real (registro completo)

Nombre

ej: Ana

Apellido

ej: González

Email

ej: ana@mail.com

Contraseña

••••••••

Confirmar contraseña

••••••••

Teléfono

ej: 11 1234 5678

Provincia

Seleccionar...

Ciudad

Seleccionar...

Dirección

ej: Av. Siempre Viva 123

Fecha de nacimiento

dd / mm / aaaa

Registrarse

Con React puro

- useState(...)
- useState(...)
- useState(...)
- useState(...)
- useState(...)
- useState(...)
- ...
- errores (muchos)
- onChange (muchos)
- onSubmit
- validar() con muchas condiciones

Resultado

- Código repetitivo
- Difícil de mantener
- Más posibilidades de error

El problema escala rápidamente

Más campos → Más estados → Más validaciones → Más complejidad

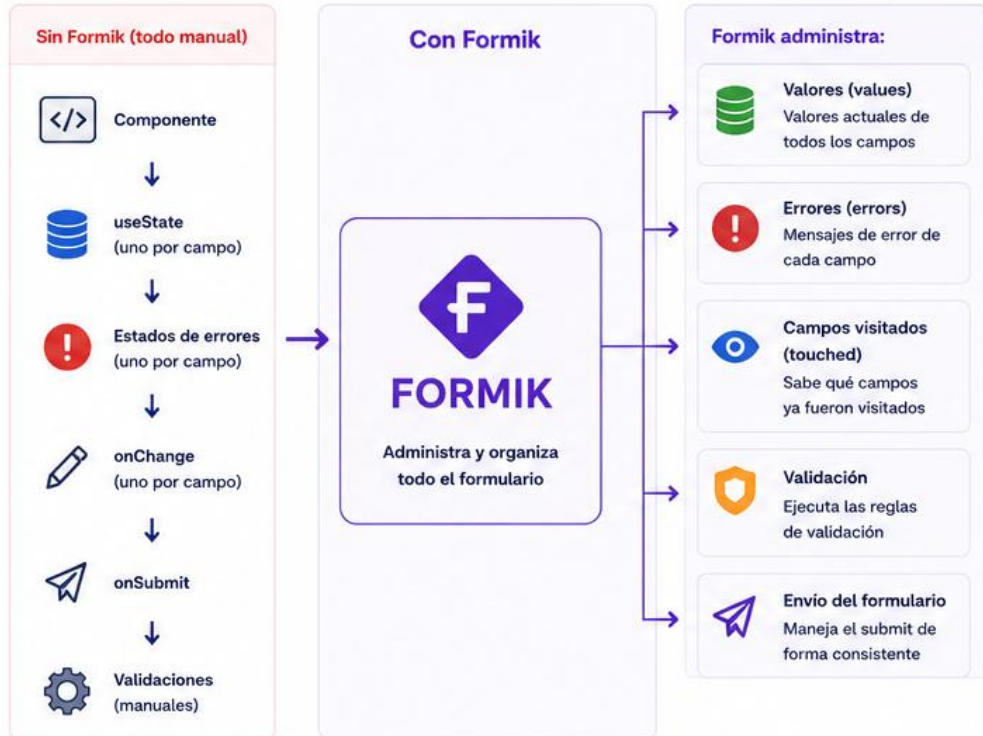
Idea clave

Formik y Yup no reemplazan a React.
Ayudan a organizar formularios cuando comienzan a crecer.

BLOQUE 2

¿Qué es Formik?

Formik es una librería de React que administra y organiza todo lo relacionado con un formulario en un solo lugar.



¿Cómo funciona?

Conceptos clave en Formik

- **values**
  → valores actuales del formulario
- **errors**
  → errores encontrados en los campos
- **touched**
  → campos que el usuario ya visitó o interactuó

Idea clave

Formik centraliza toda la información relacionada con un formulario.
De esta manera evitamos manejar múltiples estados y lógica repetitiva.

BLOQUE 3

Primer formulario con Formik

Formik nos permite construir formularios de forma declarativa y sencilla.
Con muy pocas líneas de código ya tenemos valores, cambios y envío mamejados.

Antes (React puro)

Mucho código repetitivo

- useState por cada campo
- Manejo de errores manual
- onChange por cada input
- Validación manual

Ejemplo: formulario de login

Usuario

ej: ana

Contraseña

•••••••

Ingresar

Con Formik

Menos código

- Valores manejados por Formik
- Errores manejados por Formik
- onChange automático
- Integración simple con validación

Código básico con Formik

```
import { Formik, Form, Field } from "formik";

function LoginForm() {
  const handleSubmit = (values) => {
    console.log("Datos del formulario:", values);
  };
  return (
    <Formik
      initialValues={{
        usuario: "",
        password: ""
      }}
      onSubmit={handleSubmit}
    >
      {({ isSubmitting }) => (
        <Form>
          <div>
            <label htmlFor="usuario">Usuario</label>
            <Field id="usuario" name="usuario" placeholder="ej: ana" />
          </div>
          <div>
            <label htmlFor="password">Contraseña</label>
            <Field id="password" name="password" type="password" placeholder="••••••" />
          </div>
          <button type="submit" disabled={isSubmitting}>
              Ingresar
          </button>
        </Form>
      )}
  )</Formik>
} ;
export default LoginForm;
```

¿Qué logramos con muy poco código?

Formik maneja los valores del formulario.

Formik maneja los errores.

No necesitamos escribir onChange en cada input.

onSubmit recibe todos los valores juntos.

Idea clave

Con Formik escribimos menos código y nos enfocamos en la estructura del formulario, no en la lógica repetitiva.

BLOQUE 4

El estado del formulario dentro de Formik

Formik mantiene toda la información del formulario en un solo objeto.
Para acceder a ella usamos sus propiedades principales.

values

Contiene los valores actuales de todos los campos del formulario.

```
{
  usuario: "ana",
  password: "123456"
}
```

errors

Contiene los mensajes de error de cada campo (si existen).

```
{
  usuario: "El usuario
    es obligatorio",
  password: "Mínimo
    6 caracteres"
}
```

touched

Indica qué campos fueron visitados o interactuaron el usuario.

```
{
  usuario: true,
  password: false
}
```

¿Cómo accedemos a esta información?

Formik nos da acceso a todo a través de props y helpers.

Dentro de <Formik>

```
{({ values, errors, touched,
  handleChange, handleBlur,
  handleSubmit, isSubmitting })=>(
  <Form> ... </Form>
)}
```

→

FORMIK

Administra todo el estado del formulario

→

La UI se actualiza automáticamente

Cuando cambia values, errors o touched, React vuelve a renderizar los mensajes y campos.

Visualizando el flujo

1. Usuario escribe en un campo
2. Formik actualiza values
3. Formik (o Yup) detecta errores
4. Formik marca campos como touched
5. React renderiza mensajes y estados

Idea clave

Formik centraliza todo el estado del formulario: valores, errores y campos visitados.
Esto evita manejar múltiples useState y lógica repetitiva.

BLOQUE 5

¿Qué es Yup?

Yup es una librería para definir esquemas de validación de forma declarativa.
En lugar de escribir muchas condiciones, describimos las reglas.

Validación manual (imperativa)

```
if (nombre.trim() === "") {
  error = "El nombre es obligatorio";
} else if (nombre.length < 2) {
  error = "El nombre debe tener al menos 2 caracteres";
} else if (!email.includes("@")) {
  error = "El email no es válido";
} else if (password.length < 6) {
  error = "La contraseña debe tener al menos 6 caracteres";
} ...
```

Mucho código, repetitivo y difícil de mantener

Yup (declarativo)

- nombre: obligatorio y mínimo 2 caracteres
- email: obligatorio y con formato válido
- password: obligatorio y mínimo 6 caracteres
- confirmarPassword: debe coincidir
- edad: debe ser mayor o igual a 18

...

Reglas claras, organizadas y fáciles de mantener

¿Cómo ayuda Yup?

Describe reglas
Escribimos qué debe cumplir cada campo.

Reutilizable
El esquema puede usarse en distintos formularios.

Escalable
A medida que crecen los formularios, es más fácil agregar reglas.

Integración perfecta
Se integra de forma simple con Formik.

Ejemplo: esquema Yup simple

```
import * as Yup from "yup";

const schema = Yup.object({
  nombre: Yup.string()
    .required("El nombre es obligatorio")
    .min(2, "El nombre debe tener al menos 2 caracteres"),
  email: Yup.string()
    .required("El email es obligatorio")
    .email("El email no es válido"),
  password: Yup.string()
    .required("La contraseña es obligatoria")
    .min(6, "La contraseña debe tener al menos 6 caracteres"),
});
```

Conceptos principales

`.required()`
→ campo obligatorio

`.min()`
→ longitud mínima

`.email()`
→ formato de email

`.matches()`
→ patrón personalizado

`Yup.object()`
→ objeto con reglas para cada campo

Idea clave

Yup nos permite describir las reglas de validación de forma declarativa y reutilizable, evitando mucho código condicional.

BLOQUE 6

Primer esquema Yup completo

Definimos un esquema con varias reglas y lo usamos para validar el formulativa.
Este esquema luego lo conectaremos con Formik.

Ejemplo: esquema de registro

```
import * as Yup from "yup";

const schema = Yup.object({
  nombre: Yup.string()
    .required("El nombre es obligatorio")
    .min(2, "El nombre debe tener al menos 2 caracteres")
    .max(30, "El nombre debe tener como máximo 30 caracteres"),
  email: Yup.string()
    .required("El email es obligatorio")
    .email("El email no es válido"),
  password: Yup.string()
    .required("La contraseña es obligatoria")
    .min(6, "La contraseña debe tener al menos 6 caracteres"),
  confirmarPassword: Yup.string()
    .oneOf([Yup.ref("password"), null], "Las contraseñas deben coincidir")
    .required("Debes confirmar tu contraseña"),
  edad: Yup.number()
    .required("La edad es obligatoria")
    .min(18, "Debes ser mayor de 18 años")
    .max(120, "Edad no válida"),
  terminos: Yup.boolean()
    .oneOf([true], "Debes aceptar los términos y condiciones"),
});
```

Tipos de reglas comunes

- **string()** → para textos
- **number()** → para números
- **boolean()** → para valores booleanos
- **required()** → campo obligatorio
- **min() / max()** → rangos de longitud o valores
- **oneOf()** → debe coincidir con otro valor

¿Qué validamos en este esquema?

nombre
- Obligatorio
- Mínimo 2 caracteres
- Máximo 30 caracteres

email
- Obligatorio
- Debe ser un email válido

password
- Obligatoria
- Mínimo 6 caracteres

confirmarPassword
- Debe coincidir con password

edad
- Obligatoria
- Mínimo 18
- Máximo 120

terminos
- Debe aceptar los términos

Nota

El esquema es independiente del formulario.

Solo describe las reglas.

Formik se encargará de ejecutarlas en el momento adecuado.

Idea clave

Con Yup definimos todas las reglas de validación en un solo lugar.
El esquema es claro, reutilizable y fácil de mantener.

BLOQUE 7

Integrando Formik + Yup

Formik se encarga del estado del formulario y Yup de validar los valores.
Trabajan juntos para darnos validación automática y organizada.

¿Qué sucede cuando el usuario escribe?

1. Usuario escribe en un campo
2. Formik actualiza el valor en values
3. Yup valida según el esquema
4. Formik actualiza errors (si hay errores)
5. React re-renderiza los mensajes de error y el estado de la UI
6. El usuario recibe feedback inmediato

Arquitectura Formik + Yup

Ejemplo de integración

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

const schema = Yup.object({
  email: Yup.string().email("Email inválido").required("Email obligatorio"),
  password: Yup.string().min(6, "Mínimo 6 caracteres").required("Obligatoria"),
});
function LoginForm() {
  return (
    <Formik
      initialValues={{ email: "", password: "" }}
      validationSchema={schema}
      onSubmit={(values) => console.log(values)}
    >
      <Form>
        <div>
          <label>Email</label>
          <Field name="email" type="email" />
          <ErrorMessage name="email" component="div" className="error" />
        </div>
        <div>
          <label>Contraseña</label>
          <Field name="password" type="password" />
          <ErrorMessage name="password" component="div" className="error" />
        </div>
        <button type="submit">Ingresar</button>
      </Formik>
);
```

¿Qué logramos?

- Validación automática mientras el usuario escribe.
- Mensajes de error claros y consistentes.
- Menos código repetitivo.
- Estado del formulario centralizado.
- Fácil de mantener y escalar.

Idea clave

Formik maneja el estado del formulario y Yup las reglas de validación.
Juntos nos dan una solución poderosa, clara y escalable.

BLOQUE 8

Flujo completo de un formulario moderno

Veamos el flujo completo desde que el usuario escribe hasta que ve el resultado en pantalla.
Este es el ciclo que ocurre en cada interacción.

1. Usuario

Escribe o interactúa con un campo.

Ej: escribe "ana" en el campo Nombre.

2. Field (Input)

El valor del input cambia.

onChange se ejecuta.

3. Formik

Actualiza el valor en values y el campo en touched.

4. Yup

Valida el valor según el schema definido. Devuelve errores si corresponde.

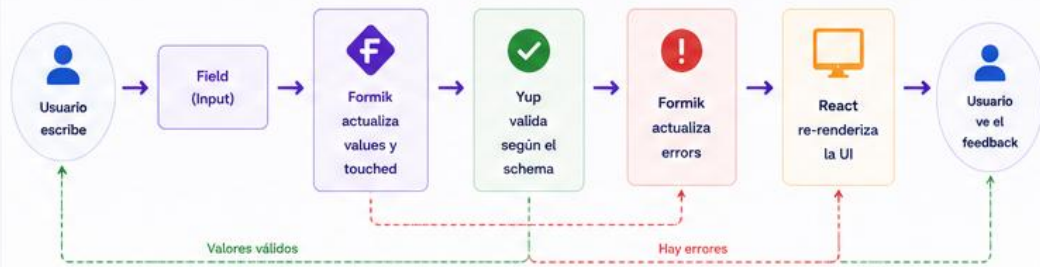
5. Errors

Formik actualiza el objeto errors con los erroelve encontrados.

6. React (UI)

Re-renderiza la UI mostrando mensajes de error y estados actualizados.

Diagrama del flujo completo



Ejemplo visual

1 Usuario escribe

Nombre

a

2 Formik actualiza values

```
values = {
  nombre: "a"
}
```

3 Yup valida

Regla: mínimo 2 caracteres

✕ No válido

4 Formik actualiza errors

```
errors = {
  nombre: "Mínimo
  2 caracteres"
}
```

5 React muestra error

Nombre

a

Mínimo 2 caracteres

6 Usuario ve el mensaje

Nombre

a

Mínimo 2 caracteres

Beneficios de este flujo

- Feedback inmediato
- Validación consistente
- Menos código repetitivo
- Escalable y mantenible
- Mejor experiencia para el usuario

Idea clave

Formik y Yup trabajan juntos en un ciclo continuo que mantiene el formulario sincronizado, validado y con feedback claro para el usuario.

BLOQUE 9

Comparación final: React puro vs Formik + Yup

Formik y Yup no reemplazan a React. Son herramientas que nos permiten escribir formularios complejos de forma más organizada, simple y escalable.

ANTES: React puro (validación manual)

¿Qué tenemos que manejar?

- Muchos useState() uno por cada campo
- Manejo manual de errores
- onChange por cada input
- Validación manual (condicionales, if, etc.)
- onSubmit y lógica de envío

Resultado

- Mucho código repetitivo
- Más posibilidades de error
- Difícil de mantener y escalar
- Más tiempo de desarrollo

JSX / JavaScript

```jsx
const [nombre, setNombre] = useState("");
const [email, setEmail] = useState("");
const [password, setPassword] = useState("");

const [errores, setErrores] = useState({});

const handleChange = (e) => {
  const { name, value } = e.target;
  // lógica para cada campo...
};

const validar = () => {
  const nuevosErrores = {};
  if (!nombre.trim()) {
    nuevosErrores.nombre = "El nombre es obligatorio";
  }
  if (!email.includes("@")) {
    nuevosErrores.email = "El email no es válido";
  }
  if (password.length < 6) {
    nuevosErrores.password = "Mínimo 6 caracteres";
  }

  setErrores(nuevosErrores);
  return Object.keys(nuevosErrores).length === 0;
};

const handleSubmit = (e) => {
  e.preventDefault();
  if (validar()) {
    // enviar datos
  }
};
```

VS

DESPUÉS: Formik + Yup

¿Qué se simplifica?

- Todo el estado del formulario centralizado en Formik
- Manejo automático de values, errors y touched
- Menos código en onChange y onBlur
- Validación declarativa con Yup
- Envío del formulario simple y consistente

Resultado

- Menos código más claro y legible
- Menos errores y más consistencia
- Fácil de mantener y escalar
- Desarrollo más rápido y productivo

JSX / JavaScript

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

const schema = Yup.object({
  nombre: Yup.string().required("El nombre es obligatorio"),
  email: Yup.string().email("Email no válido").required("Email obligatorio"),
  password: Yup.string().min(6, "Mínimo 6 caracteres").required("Obligatoria"),
});

function RegistroForm() {
  return (
    <Formik
      initialValues={{ nombre: "", email: "", password: "" }}
      validationSchema={schema}
      onSubmit={(values) => {
        // enviar datos
      }}
    >
      {({ isSubmitting }) => (
        <Form>
          <div>
            <label>Nombre</label>
            <Field name="nombre" placeholder="ej: Ana" />
            <ErrorMessage name="nombre" component="div" className="error" />
          </div>
          <div>
            <label>Email</label>
            <Field name="email" type="email" placeholder="ej: ana@mail.com" />
            <ErrorMessage name="email" component="div" className="error" />
          </div>
          <div>
            <label>Contraseña</label>
            <Field name="password" type="password" placeholder="Mínimo 6 caracteres" />
            <ErrorMessage name="password" component="div" className="error" />
          </div>
          <button type="submit" disabled={isSubmitting}>
            Registrarse
          </button>
        </Form>
      )}
    </Formik>
  );
}
```

Conceptos clave que aprendimos

Formik

Administra el estado del formulario: values, errors, touched, submit, etc.

values · errors · touched · submitCount · isSubmitting · handleChange · handleBlur

Yup

Permite definir reglas de validación de forma declarativa y reutilizable.

string() · number() · email() · required() · min() · max() · matches() · oneOf() · etc.

Integración

Formik usa el esquema de Yup para validar automáticamente los valores.

¿Cuándo usar cada uno?

React puro

- Formularios muy simples (1 a 3 campos)
- Validaciones básicas
- Casos específicos y muy controlados
- Aprender y entender cómo funciona React internamente

Ideal para empezar y para formularios chicos.

Formik + Yup

- Formularios medianos y grandes
- Muchas validaciones y reglas
- Equipos de desarrollo
- Proyectos reales y profesionales

Ideal para producción, escalabilidad y mantenimiento.

Resumen final

Formik y Yup nos permiten construir formularios profesionales.

- Código más limpio y organizado
- Validación declarativa y reutilizable
- Mejor experiencia para el usuario
- Menos bugs y más productividad

Menos esfuerzo, mejores resultados.

Idea clave

Dominar formularios con Formik y Yup es fundamental en React. Te permite enfocarte en la lógica de tu aplicación y dejar que las herramientas hagan el trabajo pesado.