

Guía de ejercicios 2 - Estado de los componentes y eventos



¡Hola! Te damos la bienvenida a esta nueva guía de ejercicios.

¿En qué consiste esta guía?

En la siguiente guía podrás trabajar los siguientes aprendizajes:

- Imprimir variables del estado declaradas con `useState` en los componentes.
- Modificar variables del estado a partir de interacciones con el usuario.
- Crear formularios en React que contengan múltiples campos.
- Gestionar la información de formularios en React con el hook `useState()`.

En este material de estudio, aprenderemos a manejar estados sobre componentes en React. Para ello, comprenderemos inicialmente qué son los **hooks** y cuáles son sus ventajas y aportes en aplicaciones que necesiten mostrar cierta información a partir de estados definidos. Entendamos los estados como la forma de almacenar información dentro de un componente y este puede cambiar según acciones o eventos que sean ejecutados por los usuarios.

¡Vamos con todo!



Tabla de contenidos

Guía de ejercicios 2 - Estado de los componentes y eventos	1
¿En qué consiste esta guía?	1
Tabla de contenidos	2
¿Qué son los estados en React?	3
Setup del proyecto contador	3
Incorporando eventos	7
Incorporando el estado	8
¿Qué es el estado?	8
Agreguemos el estado a nuestra componente	9
El estado de los componentes y eventos en React	10
Validando un formulario con un input	11
Setup del proyecto	11
Detectando cambios en un input	14
Validando el formulario	17
Validando un formulario con múltiples inputs	22
Setup del proyecto	22
Gestionando los estados de nuestros inputs del formulario	26
Para profundizar	32
Preguntas de cierre	32



¡Comencemos!

¿Qué son los estados en React?

Los estados en React nos permiten almacenar, utilizar y mostrar información actualizada en nuestros componentes. El manejo de los estados permiten construir componentes que sean interactivos y de este modo mostrar valores o datos a partir de eventos y acciones que realicen los usuarios en el sitio web.

Para aprender sobre estados realizaremos un proyecto con un botón y un contador que incrementa cada vez que se hace click. Este contador inicia su valor en 0

Setup del proyecto contador

- **Paso 1:** creamos un nuevo proyecto React bajo el nombre `contador-simple`.

```
npx create-react-app contador-simple
```

- **Paso 2:** realizaremos limpieza de nuestro proyecto una vez creado, dicho esto veremos dentro del directorio `/src` el siguiente listado de archivos que instala por defecto React.
 - App.css ✗
 - App.js ✓
 - App.test.js ✗
 - index.css ✓
 - index.js ✓
 - logo.svg ✗
 - reportWebVitals.js ✗
 - setupTests.js ✗

Marcamos con una ✗ los archivos que eliminaremos y con ✓ los que se mantendrán para trabajar con nuestro ejercicio, la estructura del directorio `/src` quedaría de la siguiente forma como se muestra en la imagen:

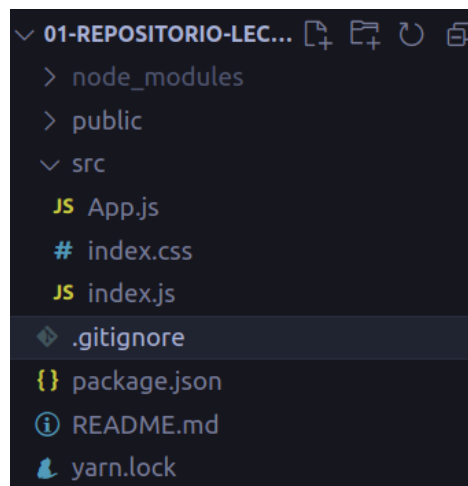


Imagen 1. Estructura limpia de directorio /src
Fuente: Desafío Latam



Esta limpieza consiste en utilizar los archivos que realmente necesitamos para lograr el objetivo de nuestro ejercicio. React instala por defecto múltiples archivos que no siempre utilizaremos o no son necesarios hasta este punto del curso.

- **Paso 3:** observemos lo que sucede al levantar el servidor del proyecto con el comando `npm run start`

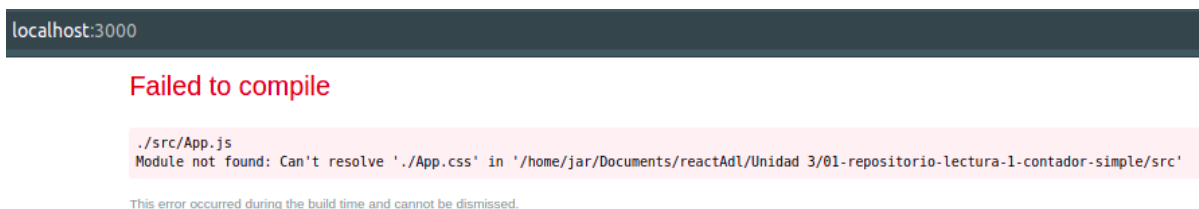


Imagen 2. Error de compilación
Fuente: Desafío Latam

- **Paso 4:** haremos la corrección en el proyecto dado que en nuestros `App.js` e `index.js` están importando archivos que no existen o fueron eliminados. Deberán quedar como se muestra a continuación.

Limpiamos el archivo `App.js`

```
//App.js
import React from 'react'
```

```
function App() {  
  return (  
    <div className="App">  
      <h1>Estado</h1>  
    </div>  
  );  
}  
  
export default App;
```

Limpiamos el archivo index.js

```
import React from 'react';  
import ReactDOM from 'react-dom/client';  
import './index.css';  
import App from './App';  
  
const rootElement = document.getElementById('root');  
const root = ReactDOM.createRoot(rootElement);  
  
root.render(  
  <React.StrictMode>  
    <App />  
  </React.StrictMode>  
);
```

Una vez modificados estos dos archivos veremos en nuestro navegador que el error ha sido solucionado y se imprime el título que incorporamos en el `App.js`



Estados con Hook useState

Imagen 3. Error solucionado

Fuente: Desafío Latam

¡Muy bien! sigamos con nuestro ejercicio para trabajar con los estados.

- **Paso 5:** en la carpeta `/src` crearemos un nuevo directorio llamado `/components`. Dentro de este, agregaremos un archivo llamado `Contador.js`.

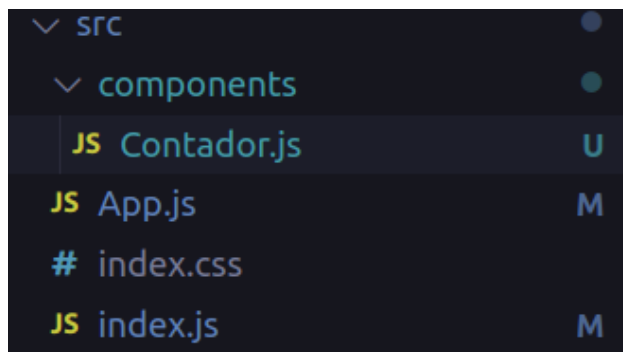


Imagen 4. Componente Contador.js
Fuente: Desafío Latam

- **Paso 6:** En el componente contador se mostrará un título `<h3>` con el texto "Componente Contador inicia en: 0".

```
// components/Contador.js
const Contador = () => {
  return(
    <h3> Componente Contador inicia en: 0 </h3>
  )
}

export default Contador
```

- **Paso 7:** importamos nuestro componente en el App.js para visualizarlo en el navegador.

```
//App.js
import React from 'react'
import Contador from './components/Contador';

function App() {
  return (
    <div className="App">
      <h1>Estados</h1>

      <Contador />
    </div>
  );
}

export default App;
```

- **Paso 8:** comprobamos que la aplicación funcione bien mostrándose en pantalla el siguiente contenido



Estados con Hook useState

Componente Contador inicia en: 0

Imagen 5. Componente Contador cargado
Fuente: Desafío Latam

Incorporando eventos

En React al igual que en Javascript tenemos una variedad de eventos que podemos utilizar algunos de ellos son:

- ❖ `onClick()`
- ❖ `onSubmit()`
- ❖ `onChange()`
- ❖ Entre otros.

Para nuestro contador utilizaremos `onClick()`. El resto de los eventos los estudiaremos más adelante en esta guía



En React los eventos debemos definirlos con camelCase y siempre comenzarán con la palabra "on".

- **Paso 9:** en nuestro componente `Contador.js` agregaremos en la etiqueta `<h3>` el evento `onClick()` para ejecutar la función `console.log("Hola")`. Veamos cómo debe quedar el código del componente.

```
// Contador.js
const Contador = () => {
  return(
    <h3 onClick={() => (console.log("hola"))}>
      Componente Contador inicia en: 0
    </h3>
  )
}
```

```
}
```

```
export default Contador
```

- **Paso 10:** comprobamos que al dar click sobre el texto del `<h3>` se imprime en la consola del navegador el texto "hola". Debe mostrarse como en la siguiente imagen:



Imagen 6. Impresión en consola evento onClick
Fuente: Desafío Latam

Incorporando el estado

El siguiente paso de nuestro ejercicio consiste en modificar el contador. En React para guardar valores que puedan cambiar necesitamos hacer uso del estado.

¿Qué es el estado?

El estado es un objeto interno del componente, donde podemos guardar todos los valores necesitemos. Para usar estados ocuparemos una función especial llamada **useState** (Traducido es literalmente usar estado).

Para utilizar useState necesitamos:

- a. Importar useState al inicio del componente.

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

- b. Definir un nombre de variable que representará el estado que queremos guardar.
- c. Definir un nombre de una función que cambiará el estado.
- d. Un valor inicial para el estado, el cual puede ser un valor de tipo *number*, *string*, *array*, *object*, *boolean*.

```
const [state, setState] = useState(valor_inicial);
```


Agreguemos el estado a nuestra componente

Continuemos con el ejercicio e incorporemos `useState()`.

- **Paso 11:** importamos `useState()` en el componente `Contador.js`, luego definimos una variable llamada `state` que almacenará nuestro estado y el modificador lo llamaremos `setState()`, el estado inicial será 0.

```
// components/Contador.js
import {useState} from 'react'

const Contador = () => {
  const [cuenta, setCuenta] = useState(0) // Definimos un estado
  llamado cuenta con valor 0, la función setCuenta podrá cambiar este
  estado.
  return(
    <h3 onClick={() => (console.log("hola"))}> Componente Contador
    inicia en: {cuenta} </h3>
  )
}

export default Contador
```

Ahora dentro de nuestra componente tenemos una cuenta con valor inicial cero y una función para cambiar el valor que ocuparemos en el próximo paso

- **Paso 12:** Ahora hacemos uso de la función `setCuenta` para aumentar el valor de la cuenta. Para ello cambiaremos el código del click para que lo haga.

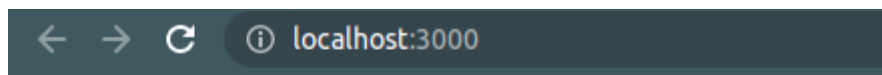
```
// Contador.js
import {useState} from 'react'

const Contador = () => {
  const [cuenta, setCuenta] = useState(0)

  return(
    <h3 onClick={() => setCuenta(cuenta + 1)}> Componente Contador:
    {cuenta} </h3>
  )
}

export default Contador
```

- **Paso 13:** observemos en el navegador el comportamiento esperado al darle click al texto del `<h3>`



Estados con Hook useState

Componente Contador: 1

Imagen 7. Resultado useState
Fuente: Desafío Latam



Nota: podrás acceder al repositorio de este ejercicio en el LMS con el nombre **Repositorio Ejercicio - Contador de clics en React**.



Recuerda que para usar el repositorio debes descargarlo, abrirlo en la terminal y correr el comando `npm install`.



¡Felicitaciones! Hasta aquí ya vimos a través de un ejercicio simple cómo implementar el Hook `useState()` y cómo hacerlo funcionar a través de un contador de clicks.

Cada vez que un estado de una componente se modifica se vuelve a renderizar la aplicación, de esta forma el usuario ve la información que corresponde al estado. Para manejar los estados en React utilizaremos el Hook `useState`.

Es normal que el primer uso de `useState` nos parezca poco familiar, hasta ahora no habíamos ocupado funciones de esta forma y tampoco habíamos trabajado con una función como `useState` que devuelve una función, revisando algunos ejercicios más lo dominaremos.

El estado de los componentes y eventos en React

En este material de estudio, seguiremos trabajando con `useState()` pero en esta ocasión lo utilizaremos para manejar estados en respuesta al cambio, envío de inputs y formularios, de forma de utilizarlo en conjunto con otro tipo de eventos.

Validando un formulario con un input

Uno de los usos más frecuentes es la validación de inputs, para aprender sobre esto crearemos un proyecto nuevo llamado `ejercicio-de-validacion` donde construiremos un componente que será un formulario con un campo de tipo input y un botón de envío.

Mostraremos un mensaje de error al usuario cuando el input esté vacío.

¡Veamos el siguiente ejercicio!: estados y formularios

Setup del proyecto

- **Paso 1:** setup del proyecto, creamos una nueva aplicación en React.

```
npx create-react-app estados-formularios
```

- **Paso 2:** limpiamos el proyecto para que nuestra carpeta `/src` contenga solo los archivos
 - App.js
 - index.js
 - index.css

Recuerda depurar los archivos que no se están importando o mostrando para que la aplicación funcione correctamente.

- **Paso 3:** en `/src` creamos una carpeta `/components` y agregamos un componente `Formulario.js`.
- **Paso 4:** dentro del archivo `App.js` dejamos un código básico para iniciar:

```
// App.js
import Formulario from './components/Formulario';

function App() {
  return (
    <Formulario/>
  );
}

export default App;
```

- **Paso 5:** en el componente `Formulario.js` retornaremos un formulario con un input y un botón de enviar.

```
// components/Formulario.js
const Formulario = () => {
  return (
    <form>
      <input name="Nombre"/>
      <button type="submit">Enviar</button>
    </form>
  )
}

export default Formulario
```

- **Paso 6:** en el archivo `index.css` agregaremos margen a la aplicación y ancho máximo a la etiqueta input del formulario.

```
body {
  margin: 70px;
}

input {
  max-width: 30%;
}
```

- **Paso 7:** importamos los estilos de bootstrap mediante su [CDN](#). Esto lo agregamos en el archivo `index.html` ubicado en la carpeta `/public`.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <link rel="icon" href="%PUBLIC_URL%/favicon.ico" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <meta name="theme-color" content="#000000" />
    <meta
      name="description"
      content="Web site created using create-react-app"
    />
    <link rel="apple-touch-icon" href="%PUBLIC_URL%/logo192.png" />
```

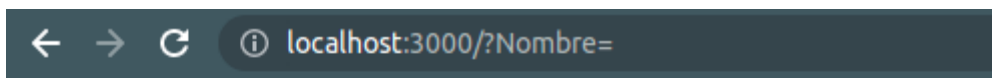
```
<link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
<link
  href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css"
  rel="stylesheet"
  integrity="sha384-1BmE4kWBq78iYhF1dvKuhfTAU6auU8tT94WrHftjDbrCEXSU1oBoqy
  12QvZ6jIW3"
  crossorigin="anonymous"
/>
<title>React App</title>
</head>
<body>
  <noscript>You need to enable JavaScript to run this app.</noscript>
  <div id="root"></div>
</body>
</html>
```

- **Paso 8:** agregamos clases de bootstrap al formulario, tanto al input como al botón.

```
// components/Formulario.js
const Formulario = () => {
  return (
    <form>
      <div className="form-group">
        <input className="form-control" name="Nombre"/>
        <button className="btn btn-dark mt-3" type="submit">
Enviar</button>
      </div>
    </form>
  )
}

export default Formulario
```

- **Paso 9:** probamos nuestro sitio en el navegador y deberíamos ver el input y el botón creado con las configuraciones de Bootstrap.



Ejercicio formulario parte 1

Imagen 8. Aplicación con estilos bootstrap
Fuente: Desafío Latam

¡Felicitaciones! Hasta este punto has creado una aplicación en React incorporando un formulario y estilos con Bootstrap mediante su CDN.

Detectando cambios en un input

Buscamos validar si el input está vacío y según eso mostrar un mensaje en caso de que lo esté. Para lograrlo, guardaremos el contenido del input como estado con `useState`, el estado inicial será un string vacío ya que el input comenzará vacío, cuando se modifique el input guardaremos el nuevo valor, para capturar el evento de que el input fue modificado utilizaremos el evento `onChange()`.

Continuemos el ejercicio anterior con los siguientes pasos:

- **Paso 10:** agregamos un estado al input, utilizaremos `useState()` donde definiremos:
 - Una variable de estado llamado `nombre`.
 - Un setter llamado `setNombre`.
 - Un valor inicial que será un string vacío.
 - También agregaremos el listener de `onchange` al input

```
// components/Formulario.js
import {useState} from 'react'

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

  return (
    <form>
      <div className="form-group">
        <input className="form-control" name="Nombre" onChange={(e)
```

```
=> console.log(e))/>
    <button className="btn btn-dark mt-3" type="submit">
Enviar</button>
    </div>
  </form>
)
}

export default Formulario
```

- **Paso 11:** si observamos la consola del navegador para verificar el comportamiento del evento `onChange()` y la función `console.log()` que está recibiendo, veremos que si escribimos caracteres en el input se imprimen cada uno de los eventos de esta acción

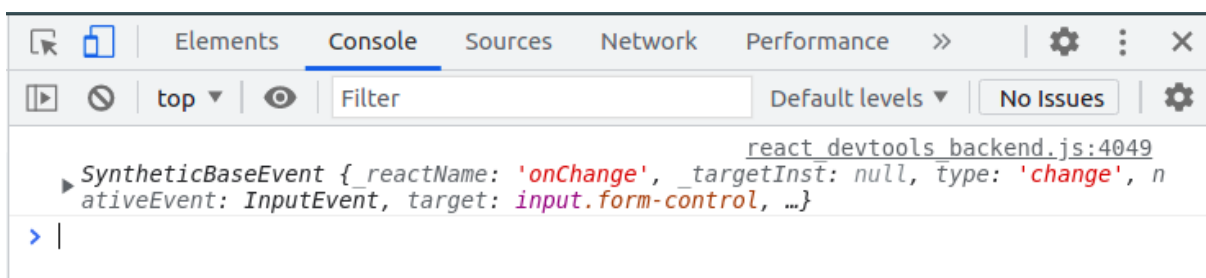


Imagen 9. Evento onChange en consola
Fuente: Desafío Latam

- **Paso 12:** modifiquemos el `console.log()` definido en el `onChange()` y en vez de pasar el evento, pasaremos `e.target.value`. Esto nos debería mostrar directamente los caracteres que se están agregando al input del formulario.

```
// components/Formulario.js
import {useState} from 'react'

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

  return (
    <form>
      <div className="form-group">
        <input className="form-control" name="Nombre" onChange={
=> console.log(e.target.value)}/>
        <button className="btn btn-dark mt-3" type="submit">
Enviar</button>
      </div>
    </form>
  )
}
```

```
    </form>
  )
}

export default Formulario
```

- **Paso 13:** observemos qué ocurre ahora en la consola del navegador si escribimos en el input la palabra hola.

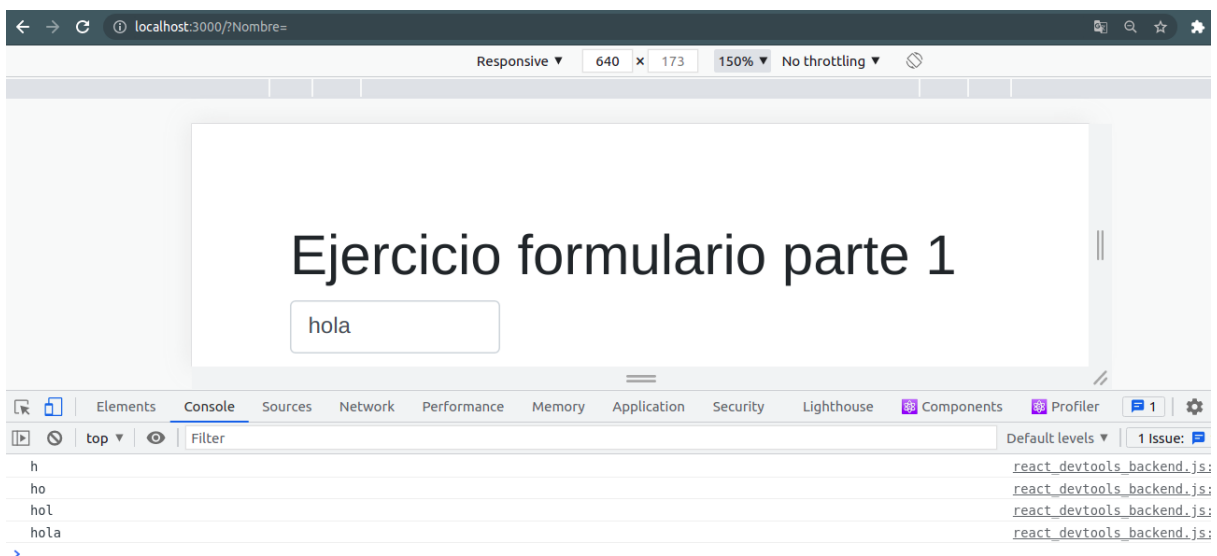


Imagen 10. Muestra caracteres del input
Fuente: Desafío Latam

- **Paso 14:** mostremos en pantalla nuestro estado inicial y modifiquemos su valor con el `setNombre`. Nuestro componente `Formulario.js` quedará de la siguiente manera:

```
// components/Formulario.js
import {useState} from 'react'

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

  return (
    <form>
      <h1>{nombre}</h1>
      <div className="form-group">
        <input className="form-control" name="Nombre" onChange={(e)
=> setNombre(e.target.value)}>
        <button className="btn btn-dark mt-3" type="submit">
```



```
Enviar</button>
  </div>
</form>
)
}

export default Formulario
```

- **Paso 15:** si observamos los cambios en el navegador, veremos que al escribir en el input se muestra antes del input nombre lo que el usuario ingresa. Veamos la siguiente imagen:

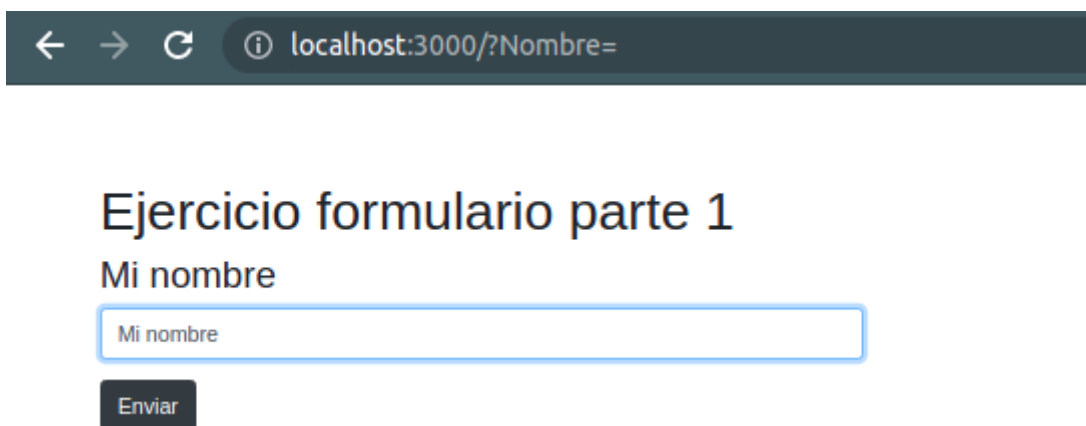


Imagen 11. Formulario y sitio dinámico
Fuente: Desafío Latam

El texto del input lo estamos mostrando temporalmente. Ahora tenemos que validar el formulario al momento de que el usuario intente enviarlo, si cumple las reglas, o sea en este caso que el input no esté vacío, lo enviaremos, en caso contrario mostraremos un mensaje de error. Para esto utilizaremos tanto la etiqueta form con `onSubmit()`.

Validando el formulario

- **Paso 16:** en el componente `Formulario.js`, agregamos a la etiqueta `<form>` de apertura el evento `onSubmit()`. Esta recibirá una función que llamaremos `validarInput`. Esta función la definiremos antes del `return()` y debajo de nuestro state en la cual vamos a mostrar un `alert()` de JavaScript para verificar que al oprimir el botón se dispare. Veamos el código a continuación y verifiquemos el funcionamiento en el navegador.

```
// components/Formulario.js
import {useState} from 'react'
```

```
const Formulario = () => {
  const [nombre, setNombre] = useState("")

  const validarInput = () => {
    alert('evento form')
  }

  return (
    <form onSubmit={validarInput}>
      <h3>{nombre}</h3>
      <div className="form-group">
        <input className="form-control" name="Nombre" onChange={(e)
=> setNombre(e.target.value)}>
        <button className="btn btn-dark mt-3" type="submit">
Enviar</button>
      </div>
    </form>
  )
}

export default Formulario
```

En el navegador debe mostrarse nuestro `alert()` de la siguiente manera:

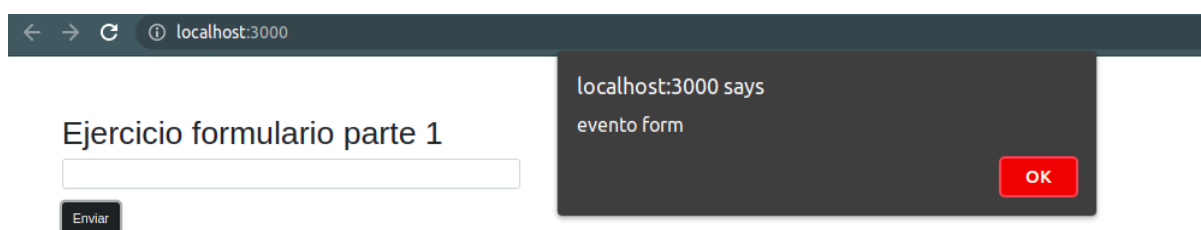


Imagen 12. Evento onSubmit
Fuente: Desafío Latam

¡Excelente! logramos acceder al evento cuando se oprime el botón de enviar, sigamos con la validación.



Si observas el comportamiento del formulario verás que el navegador realiza una petición GET y recarga el sitio. Este comportamiento por defecto del botón de tipo `submit` podemos prevenirla con el método `e.preventDefault()`.

- **Paso 17:** implementemos dentro de la función `validarInput` la lógica de JavaScript correspondiente para que el formulario no pueda ser enviado sin datos ingresados.

```
// components/Formulario.js
import {useState} from 'react'

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

  const validarInput = (e) => {
    // Prevenimos el comportamiento por defecto
    e.preventDefault()

    // Validación input
    if(nombre === '') {
      alert('Debes agregar tu nombre')
    }
  }

  return (
    <form onSubmit={validarInput}>
      <h3>{nombre}</h3>
      <div className="form-group">
        <input className="form-control" name="Nombre" onChange={e}
=> setNombre(e.target.value)}>
        <button className="btn btn-dark mt-3" type="submit">
Enviar</button>
      </div>
    </form>
  )
}

export default Formulario
```

De esta forma le estamos diciendo al sistema, cuando el formulario intente ser enviado con el campo nombre vacío, levanta un alerta que avise que “Debes agregar tu nombre”. Quizás hacer esto con un `alert()` no sea lo conveniente, sino mostrar en su defecto un mensaje de error, veamos cómo hacerlo.

- **Paso 18:** creemos un nuevo estado para mostrar errores, lo llamaremos `[error, setError]`. Este estado mostrará en pantalla un mensaje al usuario si intenta enviar el formulario con el campo input vacío. Veamos el siguiente código y luego analicemos lo que se implementó.

```
// components/Formulario.js
import {useState} from 'react'

const Formulario = () => {
  const [nombre, setNombre] = useState("")
  const [error, setError] = useState(false)

  const validarInput = (e) => {
    //Prevenimos el comportamiento por defecto
    e.preventDefault()

    //Validación input
    if(nombre === '') {
      setError(true)

      return
    }
    //Eliminar mensaje de error
    setError(false)
  }

  return (
    <form onSubmit={validarInput}>
      {error ? <p className="error">Debes ingresar tu nombre</p> :
null}
      <h3>{nombre}</h3>
      <div className="form-group">
        <input className="form-control" name="Nombre" onChange={e)
=> setNombre(e.target.value)}>
        <button className="btn btn-dark mt-3" type="submit">
Enviar</button>
      </div>
    </form>
  )
}

export default Formulario
```

El mensaje de error lo estamos manejando mediante un estado inicial definido como `false`. Una vez que el formulario es enviado con el campo nombre vacío, React modificará el estado a través del `setError(true)`, cambiando de falso a verdadero.

Seguidamente, en la parte inicial de nuestro formulario estamos validando mediante operador ternario que, en caso de que error sea `true`, imprime un párrafo con el texto “Debes ingresar tu nombre”, en caso contrario no muestres nada con la instrucción `null`.

Si continuamos analizando, en nuestra validación inicial y debajo del `setError(true)` estamos pasando un `return`. Esto lo hacemos dado que en caso de que si existe el error no queremos que se siga ejecutando la lógica siguiente. Seguidamente, por fuera del `if` le estamos diciendo al sistema que en caso de que el formulario si reciba los datos en el input entonces elimine el mensaje de error.

Nuestra aplicación se verá de la siguiente manera:

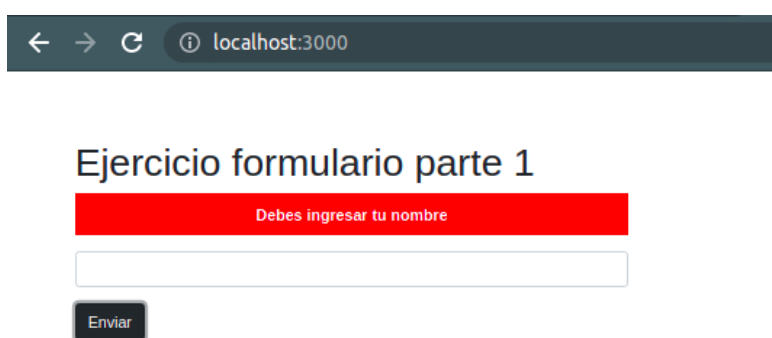


Imagen 6. Formulario con error true
Fuente: Desafío Latam

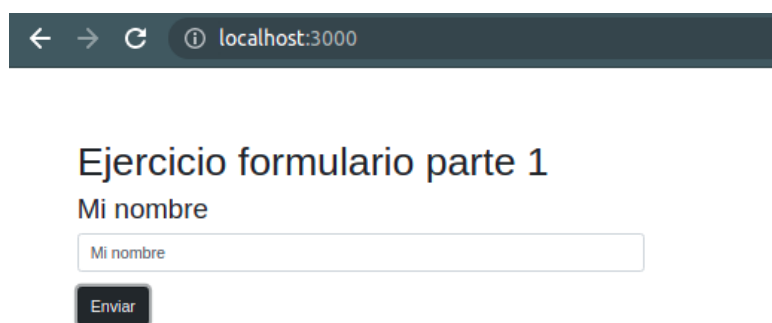


Imagen 13. Formulario con error en false
Fuente: Desafío Latam

- **Paso 19:** Para los estilos de la etiqueta párrafo con el mensaje de error puedes utilizar el siguiente código de nuestro archivo `index.css`.

```
body {  
  margin: 70px;  
}
```

```
input {  
  max-width: 30%;  
}  
  
.error {  
  padding: 10px;  
  max-width: 30%;  
  color: white;  
  background-color: red;  
  text-align: center;  
  font-weight: bold;  
}
```



Nota: podrás acceder al repositorio de este ejercicio en el LMS con el nombre **Repositorio Ejercicio - Estados y Formularios**.



¡Felicitaciones! En esta ocasión aprendiste a desarrollar un pequeño formulario implementando dos eventos importantes para gestionar el comportamiento por defecto de los mismos. Estos eventos fueron `onChange()` y `onSubmit()`. Son muy utilizados y es por ello que en los materiales de estudio siguientes, continuaremos ejercitando.

Validando un formulario con múltiples inputs

A continuación seguiremos ejercitando los conceptos adquiridos hasta el momento. Trabajaremos con formularios y, en esta ocasión, veremos cómo manejar el estado de nuestros componentes asignando como valor inicial un objeto con distintos valores por defecto. Estos valores pasados en el estado serán los campos de nuestro formulario.

Setup del proyecto

¡Veamos el siguiente ejercicio!: validando múltiples campos de formularios.

- **Paso 1:** setup del proyecto, creamos una nueva aplicación en React.

```
npx create-react-app formularios-react
```

- **Paso 2:** limpiamos el proyecto y dejamos solo los siguientes archivos dentro de `/src`
 - App.js
 - index.js
 - index.css



Modifica los archivos eliminando las referencias a los archivos borrados para que la aplicación funcione correctamente

- **Paso 3:** en `/src` creamos una carpeta `/components` y agregamos un componente `Formulario.js`.
- **Paso 4:** dentro del archivo `App.js` dejamos un código básico para iniciar:

```
// App.js
import Formulario from './components/Formulario';

function App() {
  return (
    <Formulario />
  );
}

export default App;
```

- **Paso 5:** en el componente `Formulario.js` retornaremos un formulario que contendrá los campos de nombre, apellido, edad y email. Para cada uno de estos campos vamos a definir un label correspondiente. Un `label` para el Nombre, Apellido, Edad y Email. Veamos cómo quedaría el componente.

```
import React from 'react'

const Formulario = () => {
  return (
    <form>
      <div className="form-group">
        <label>Nombre</label>
        <input type="text" className="form-control" />
      </div>
      <div className="form-group">
        <label>Apellido</label>
        <input type="text" className="form-control" />
      </div>
      <div className="form-group">
        <label>Edad</label>
        <input type="text" className="form-control" />
      </div>
      <div className="form-group">

```

```
        <label>Email</label>
        <input type="text" className="form-control" />
      </div>
      <button type="submit" className="btn
btn-primary">Submit</button>
    </form>
  )
}

export default Formulario
```

- **Paso 6:** en el archivo `index.css` agregaremos margen a la aplicación y ancho máximo a la etiqueta input del formulario.

```
body {
  margin: 70px;
}

.formulario {
  max-width: 50%;
}
```

- **Paso 7:** importamos los estilos de bootstrap mediante su [CDN](#). Esto lo agregamos en el archivo `index.html` ubicado en la carpeta `/public`.

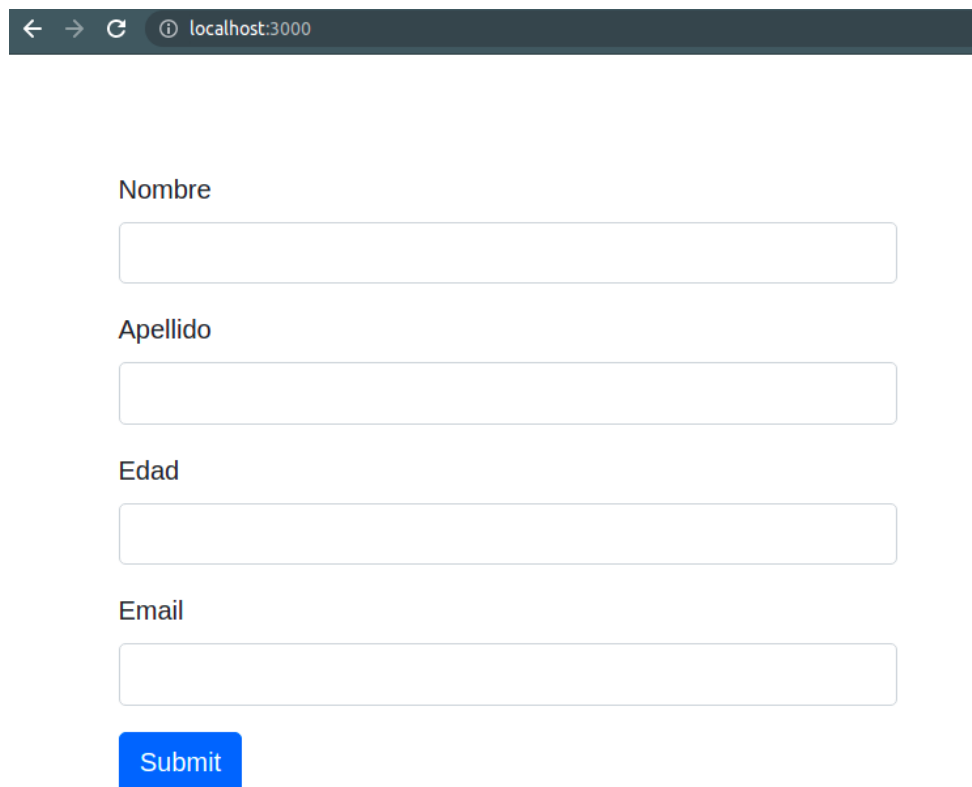
```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <link rel="icon" href="%PUBLIC_URL%/favicon.ico" />
    <meta name="viewport" content="width=device-width, initial-scale=1"
  />
    <meta name="theme-color" content="#000000" />
    <meta
      name="description"
      content="Web site created using create-react-app"
    />
    <link rel="apple-touch-icon" href="%PUBLIC_URL%/logo192.png" />
    <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
    <link
      href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.mi
```



```
n.css"
  rel="stylesheet"

integrity="sha384-1BmE4kWBq78iYhF1dvKuhfTAU6auU8tT94WrHftjDbrCEXSU1oBoqy
12QvZ6jIW3"
  crossorigin="anonymous"
/>
<title>React App</title>
</head>
<body>
  <noscript>You need to enable JavaScript to run this app.</noscript>
  <div id="root"></div>
</body>
</html>
```

- **Paso 8:** probamos nuestro sitio en el navegador y deberíamos ver el input y el botón creado con las configuraciones de Bootstrap.



← → ↻ ⓘ localhost:3000

Nombre

Apellido

Edad

Email

Submit

Imagen 14. Formulario Bootstrap
Fuente: Desafío Latam

¡**Felicitaciones!** Hasta este punto has creado una aplicación en React incorporando un formulario y estilos con Bootstrap mediante su CDN.

Gestionando los estados de nuestros inputs del formulario

Para cada campo del formulario asignaremos un estado, esto nos permitirá controlar y almacenar los valores que sean ingresados por los usuarios.

- **Paso 9:** agregamos el estado datos al componente `Producto.js`, el cual recibirá las propiedades definidas en nuestros campos del formulario:

```
//Formulario.jsx
import React, { useState } from 'react';

const Formulario = () => {
  //Estados del formulario
  const [nombre, setNombre] = useState('');
  const [apellido, setApellido] = useState('');
  const [edad, setEdad] = useState('');
  const [email, setEmail] = useState('');
```

- **Paso 10:** agregamos a través del evento `onChange()` la función correspondiente que detectará cuando el usuario escriba en los campos del formulario. Este evento va a recibir un callback que captura a través del *event* los datos que sean ingresados en cada input:

```
import {useState} from 'react'

const Formulario = () => {
  //Estados del formulario
  const [nombre, setNombre] = useState('');
  const [apellido, setApellido] = useState('');
  const [edad, setEdad] = useState('');
  const [email, setEmail] = useState('');

  const actualizarDatos = (e) => {
    console.log('Escribiendo')
  }

  return (
    <form className="formulario">
      <div className="form-group">
        <label>Nombre</label>
        <input
          type="text"
```

```
        name="nombre"
        className="form-control"
        onChange={(e) => setNombre(e.target.value)}
        value={nombre}
      />
    </div>
    <div className="form-group">
      <label>Apellido</label>
      <input
        type="text"
        name="apellido"
        className="form-control"
        onChange={(e) => setApellido(e.target.value)}
        value={apellido}
      />
    </div>
    <div className="form-group">
      <label>Edad</label>
      <input
        type="text"
        name="edad"
        className="form-control"
        onChange={(e) => setEdad(e.target.value)}
        value={edad}
      />
    </div>
    <div className="form-group">
      <label>Email</label>
      <input
        type="email"
        name="email"
        className="form-control"
        onChange={(e) => setEmail(e.target.value)}
        value={email}
      />
    </div>
    <button type="submit" className="btn btn-primary">
      Submit
    </button>
  </form>
)
}
```

```
export default Formulario
```

Si inspeccionamos nuestro componente con *React Developer Tools*, veremos que por cada información ingresada en los campos del formulario, se va agregando a nuestro estado.

Observemos la siguiente imagen extraída de React Developer Tools

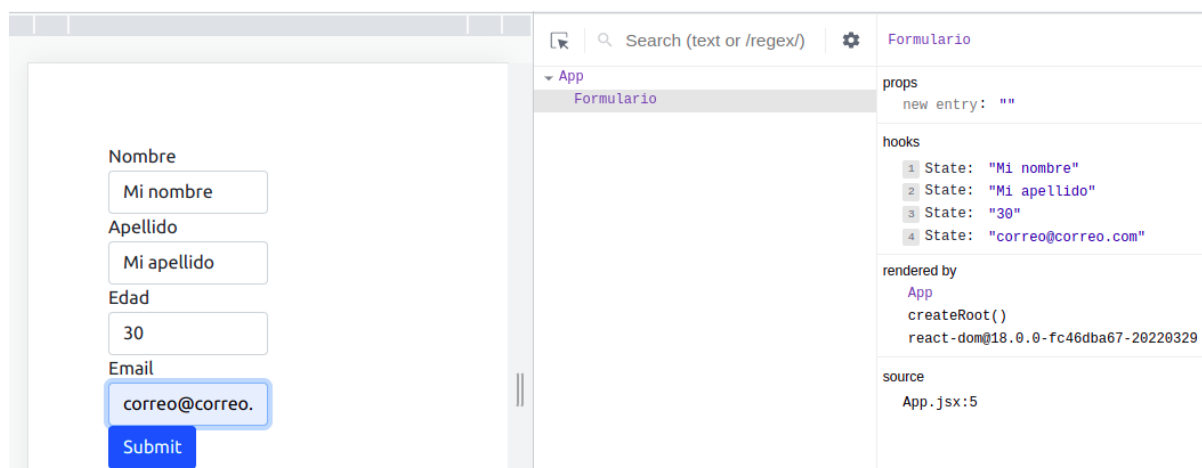


Imagen 15. Estado campo nombre
Fuente: Desafío Latam

Continuemos con nuestro ejercicio, ahora vamos a validar cada input del formulario.

- **Paso 11:** validamos el evento `onSubmit()` cuando el usuario envíe el formulario. Esta validación la haremos para evitar que los campos estén vacíos. Para ello crearemos una función que se llamará `validarDatos()`. Dicha función estará incorporada en la etiqueta `<form>` de apertura.



Recuerda prevenir la acción por defecto en la función `validarDatos()` con el `e.preventDefault()`.

El cuerpo de nuestra función será de la siguiente manera:

```
//Función antes de enviar el formulario
const validarDatos = (e) => {
  e.preventDefault()

  //Validación
}
```

- **Paso 12:** dentro de nuestra función `validarDatos()` haremos a continuación la validación condicional haciendo uso del condicional `if` de JavaScript.

```
//Función antes de enviar el formulario
const validarDatos = (e) => {
  e.preventDefault()

  //Validación
  if(nombre === '' || apellido === '' || edad === '' || email ===
  '') {
    alert('Todos los campos son obligatorios')

    return
  }
}
```

Observa este comportamiento en el navegador y si los campos del formulario se envían vacíos se dispara la función `alert()` y el mensaje definido en su interior.

- **Paso 13:** como vimos en el ejercicio de la lectura anterior, podemos gestionar errores a través de estado. Configuremos nuestro componente para implementar errores en la validación. Crearemos un nuevo estado `[error, setError]`. Este queremos que se active cuando los campos estén vacíos y se desactive cuando sean completados los campos faltantes.

```
//Formulario.jsx
import React, { useState } from 'react';

const Formulario = () => {
  //Estados del formulario
  const [nombre, setNombre] = useState('');
  const [apellido, setApellido] = useState('');
  const [edad, setEdad] = useState('');
  const [email, setEmail] = useState('');

  //Estado para los errores
  const [error, setError] = useState(false);

  //Función antes de enviar el formulario
  const validarDatos = (e) => {
    e.preventDefault();

    //Validación;
    if (nombre === '' || apellido === '' || edad === '' || email === '')
    {
      setError(true);
    }
  }
}
```

```
    return;
  }
};

return (
  <div>
    <form className="formulario" onSubmit={validarDatos}>
      {error ? <p>Todos los campos son obligatorios</p> : null}
      <div className="form-group">
        <label>Nombre</label>
        <input
          type="text"
          name="nombre"
          className="form-control"
          onChange={(e) => setNombre(e.target.value)}
          value={nombre}
        />
      </div>
      <div className="form-group">
        <label>Apellido</label>
        <input
          type="text"
          name="apellido"
          className="form-control"
          onChange={(e) => setApellido(e.target.value)}
          value={apellido}
        />
      </div>
      <div className="form-group">
        <label>Edad</label>
        <input
          type="text"
          name="edad"
          className="form-control"
          onChange={(e) => setEdad(e.target.value)}
          value={edad}
        />
      </div>
      <div className="form-group">
        <label>Email</label>
        <input
          type="email"
          name="email"
          className="form-control"

```

```
        onChange={(e) => setEmail(e.target.value)}
        value={email}
      />
    </div>
    <button type="submit" className="btn btn-primary">
      Submit
    </button>
  </form>
  <hr />
  <h1>Datos ingresados</h1>
  {nombre} - {apellido} - {edad} - {email}
</div>
);
};

export default Formulario;
```

Imaginemos un caso ¿Qué tendríamos que configurar para que los campos del formulario una vez que el usuario lo envía se vuelvan a mostrar vacíos?

- ¿Te imaginas cómo podría ser esta configuración?
- ¿Identificas con facilidad el código necesario para esto?

Si no logras dar con la solución tranquilo, estás en proceso de aprendizaje y poco a poco estos casos imaginativos van a ser solucionados con práctica. A continuación sigamos con los pasos.

- **Paso 14:** después del `setError(false)`, le diremos a nuestro formulario que vuelva al estado inicial pasando cada una de las funciones setter asociadas a cada campo del formulario a un string vacío, tal como lo definimos desde un inicio. Veamos cómo queda entonces el cuerpo de nuestra función `validarDatos()`.

```
//Función antes de enviar el formulario
const validarDatos = (e) => {
  e.preventDefault();

  //Validación;
  if (nombre === '' || apellido === '' || edad === '' || email === '')
  {
    setError(true);
    return;
  }
}
```

```
// Si el formulario se envía correctamente devolvemos todos nuestros  
estados al inicial y reseteamos el formulario  
setError(false);  
setNombre('');  
setApellido('');  
setEdad('');  
setEmail('');  
};
```



Nota: podrás acceder al repositorio de este ejercicio en el LMS con el nombre **Repositorio Ejercicio - Validando múltiples campos de formularios.**



Recuerda que para usar el repositorio debes descargarlo, abrirlo en la terminal y correr el comando `npm install`.



Para profundizar

Tomemos unos minutos para autoreflexionar los conocimientos adquiridos hasta el momento respondiendo las siguientes preguntas:

- ¿Cuánto comprendí de los conceptos vistos hasta este punto?
- ¿Cómo puedo relacionar esta información con los conocimientos previos durante mi ciclo formativo?
- ¿Qué aprendí que no quiero olvidar sobre lo visto en este material?

Preguntas de cierre

- ¿Cuáles son los distintos tipos de datos que pueden recibir los estados de nuestros componentes?
- ¿Para qué nos sirve el *spread operator*?
- Si tenemos un campo, el atributo `name=nombre`, `name=apellido` en el formulario, ¿Cómo se deberá llamar nuestra propiedad en el estado?