# Guía de ejercicios 3 - Renderización dinámica de componentes

 ¡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:

- Renderizar dinámicamente componentes a partir de un conjunto de datos locales.
- Crear una aplicación en React que importe un JSON de datos y renderice una galería de productos.

En este material de estudio, abordaremos la renderización dinámica de información en componentes de React. Es muy importante la realización de las actividades, pues a través de ellas, reforzarás con casos comunes o semejantes a lo que será nuestro desempeño laboral.
Además, a partir de una base de datos en formato JSON, mostraremos información a los usuarios en el navegador. Es un apartado muy importante porque pondremos en práctica los conocimientos adquiridos hasta el momento en React y veremos que es posible mostrar datos dinámicos a partir de archivo JSON.

**¡Vamos con todo!**

# Tabla de contenidos



**¡Comencemos!**

# Mostrando una lista de elementos

En esta guía abordaremos cómo trabajar con conjuntos de datos, o sea arreglos y objetos, para construir una aplicación web a partir de una fuente de información estática como un arreglo, archivo CSV o un archivo JSON, y más adelante poder mostrar en nuestros componentes información traída a través de APIS

## Actividad 1: Mostrando un listado de elementos

- ***Paso 1:*** Setup del proyecto, creamos un nuevo proyecto llamado `lista-tareas`

```
npm create vite
```

**Recuerda:** Eliminar las importaciones y el código que no se utilizará, además de los archivos svg como el logo de React y Vite, ya que no serán utilizados.

- ***Paso 2:*** Dentro del directorio `src`, creamos el directorio `components`, y un componente llamado `Tareas.jsx`.
  Una vez creado, lo importamos dentro de nuestro componente principal `App.jsx`.

```
// App.jsx

import Tareas from './components/Tareas'
function App() {
  return (
    <>
      <Tareas />
    </>
  );
}

export default App
```

- ***Paso 3:*** Agregamos el siguiente código al componente `Tareas` :

```
// components/Tareas.jsx

const Tareas = () => {
  const listaTareas = ["Tarea 1", "Tarea 2", "Tarea 3"]

  return (
    <ul>
      {listaTareas.map(tarea => <li key={tarea}> {tarea} </li> )}
    </ul>
  )
}

export default Tareas
```

Entendamos el código del componente `Tarea.js`

1. Estamos definiendo en una constante `listaTareas` un arreglo con valores de tipo *string*.
2. Por cada uno de los elementos de la lista, escribimos <li key="nombre-tarea"> nombre-tarea </li>. En JSX podemos hacerlo con el método map.
3. Dentro del `return()` estamos pintando los datos de nuestra constante a través del método `.map()`.
4. Estamos asignando `key={tarea}` a cada uno de los elementos de la lista para cumplir con las recomendaciones de React al momento de mostrar un listado con información. Esto lo pondremos a prueba después de verlo funcionar

   Nuestra aplicación hasta este punto se verá de la siguiente manera:


Imagen 1. Muestra de app funcionando
Fuente: Desafío Latam

Prueba removiendo `key={tarea}` y recarga la página, deberás ver un error asociado a la falta de key en el inspector de elementos.

## Recordemos el método map.

Se utiliza en arreglos para transformar cada elemento en uno nuevo.
Toma el arreglo existente, en este caso listaTareas y crea un nuevo arreglo, aplicando una función a cada elemento de manera individual.

En el bloque de código anterior, solo utilizamos el método `map` para listar los elementos, pero podríamos modificar sus elementos sin la necesidad de modificar el arreglo original.

Veamos un ejemplo:

```
const listaTareas = [Tarea 1, Tarea 2, Tarea 3];

// Creamos un nuevo arreglo llamado lista, que contendrá el resultado de
la función del método map, en este caso convertir cada tarea a mayúscula

const lista = listaTareas.map((tarea) => tarea.toUpperCase());

console.log(lista); // Resultado: [TAREA 1, TAREA 2, TAREA 3]
```

*Esta conversión de las tareas a mayúsculas es solo un ejemplo, por lo que no es necesaria para el desarrollo del ejercicio.*

Te invitamos a reforzar el uso de map creando un nuevo proyecto (**Debe ser nuevo, ya que seguiremos utilizando el ejercicio** `lista-tareas`).
En este proyecto, crea un arreglo con 10 números (pueden ser aleatorios o los que tu prefieras) y realiza algunas operaciones como:

- Multiplicar cada número del arreglo por su mismo valor.
- Calcular el número al cuadrado de cada número del arreglo.
    - ***Investigar sobre el operador de exponente***
- Filtrar los números pares e impares.

## Agregando elementos a una lista

A continuación aprenderemos a trabajar agregando dinámicamente elementos a una lista, para eso seguiremos trabajando en el proyecto `lista-tareas`.
Entramos al proyecto y levantamos nuevamente el servidor.

## Actividad 2: Agregando una tarea a la lista

Para agregar una tarea nueva a lista necesitaremos 3 elementos: un input, un formulario y un botón submit:

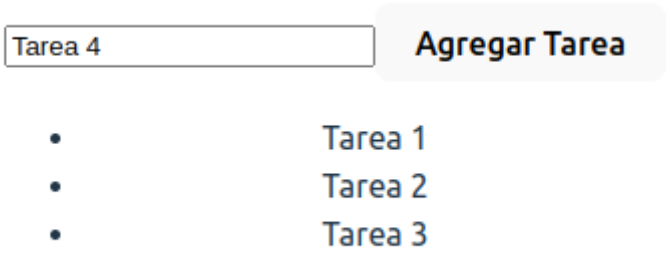


Imagen 2. Listado de tareas con elementos para agregar
Fuente: Desafío Latam

Para actualizar la lista necesitamos que el arreglo sea un estado, de esta forma solo con agregar un elemento nuevo la lista se actualizará. También necesitaremos ir guardando en un estado el valor del input dado que será este valor el que guardaremos en el arreglo.

- Cada vez que se modifique el input, actualizaremos el estado que guarda su información.
- Cada vez que se envíe el formulario, actualizaremos el estado que guarda el arreglo.

### Implementemos el código

- ***Paso 4:*** Nuestro siguiente paso será agregar un formulario con un input y botón en nuestro componente de `Tareas.jsx`. Por el momento, dejaremoa los eventos con `console.log`, y el código deberá quedar como se muestra a continuación:

```
// components/Tareas.jsx

const Tareas = () => {
    const listaTareas = ["Tarea 1", "Tarea 2", "Tarea 3"]
```

```
    // Función al enviar el formulario
    const enviarFormulario = (e) => {
        e.preventDefault()
        console.log('Enviando formulario')
    }

    //Función al escribir sobre el input del formulario
    const capturaInput = (e) => {
        console.log(e.target.value)
    }

    return (
      <>
        <div>
          <form onSubmit={enviarFormulario}>
            <input name="nombreTarea" onChange={capturaInput} />
            <button> Agregar Tarea </button>
          </form>

          <ul>
            {listaTareas.map(tarea => <li key={tarea}> {tarea} </li> )}
          </ul>
        </div>
      </>
    )
}

export default Tareas
```

Entendamos el código

1. En nuestro paso 4 agregamos al componente `Tareas.jsx` un formulario con input y botón de enviar.

2. Agregamos un listener sobre la etiqueta `<form>` en el evento `onSubmit()` y un listener en el input en el evento onChange.

3. Al gatillarse el envío del formulario, se llamará a la función `enviarFormulario` la cual primero evitará el comportamiento por defecto del formulario cuando este se envíe con `e.preventDefault()` y luego, mostrará en la consola del navegador un mensaje que dice Enviando formulario.

4. Al gatillarse una modificación del input, se llamará a la función `capturarInput` quien mostrará en la consola del navegador el valor actual del input.

Continuemos con nuevos pasos en el ejercicio.

- ***Paso 5:*** Agregamos el estado a nuestro componente `Tarea.jsx` con el hook `useState()`. Este se llamará `nombreTarea` y el setter `setNombreTarea`.
  En nuestro input, en vez de enviar información a la consola, cambiaremos el estado inicial por los datos que ingrese el usuario utilizando el `setNombreTarea()`.
  Veamos cómo queda el código:

```
// components/Tareas.jsx

import { useState } from "react"

const Tareas = () => {
    const [nombreTarea, setNombreTarea] = useState("")
    const listaTareas = ["Tarea 1", "Tarea 2", "Tarea 3"]

    //Función al enviar el formulario
    const enviarFormulario = (e) => {
        e.preventDefault()
        console.log('Enviando formulario')
    }

    //Función al escribir sobre el input del formulario
    const capturaInput = (e) => {
        setNombreTarea(e.target.value)
    }

    return (
      <>
        <div>
          <form onSubmit={enviarFormulario}>
            <input name="nombreTarea" onChange={capturaInput}
value={nombreTarea} />
            <button> Agregar Tarea </button>
          </form>

          <ul>
            {listaTareas.map(tarea => <li key={tarea}> {tarea} </li> )}
          </ul>
        </div>
      </>
    )
  }
```

```
export default Tareas
```

### Entendamos el código

1. Al input agregamos el listener onChange que llame a la función `capturaInput` y agregamos el value `nombreTarea` para asignar el valor inicial y mapear el valor del elemento al estado.
   Cuando el input cambie, se llamará a la función `capturaInput`, dentro de esta actualizaremos el estado `nombreTarea`
   - Prueba sacar `value={nombreTarea}` recarga la página. ¿Qué sucede? Luego vuelve a agregarlo

2. Comprobemos en react developer tools cómo se modifica el estado, veamos las siguientes imágenes.

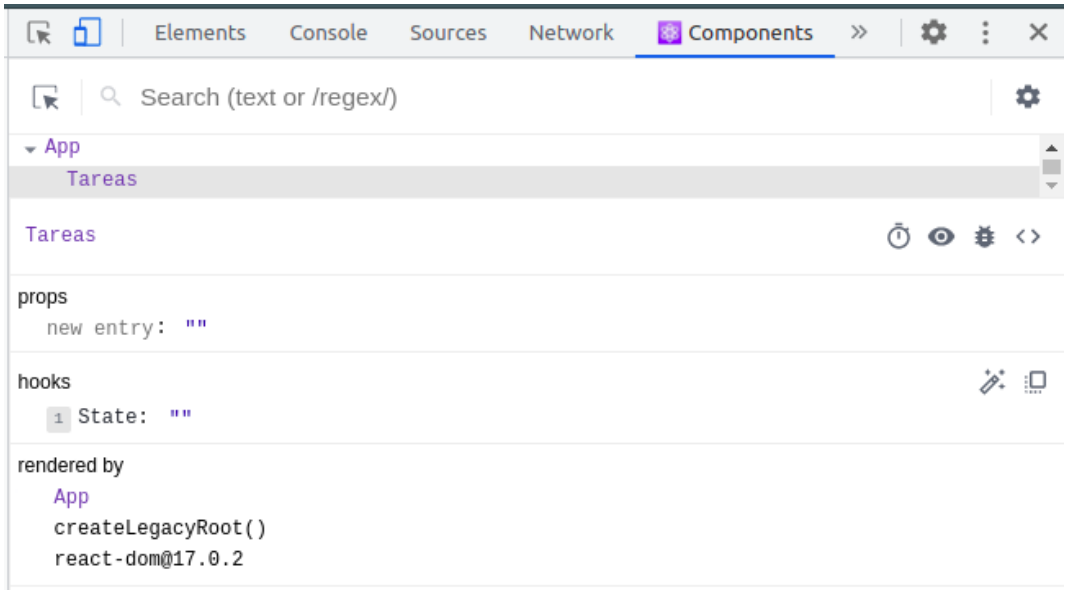


Imagen 3. Estado inicial aplicación
Fuente: Desafío Latam

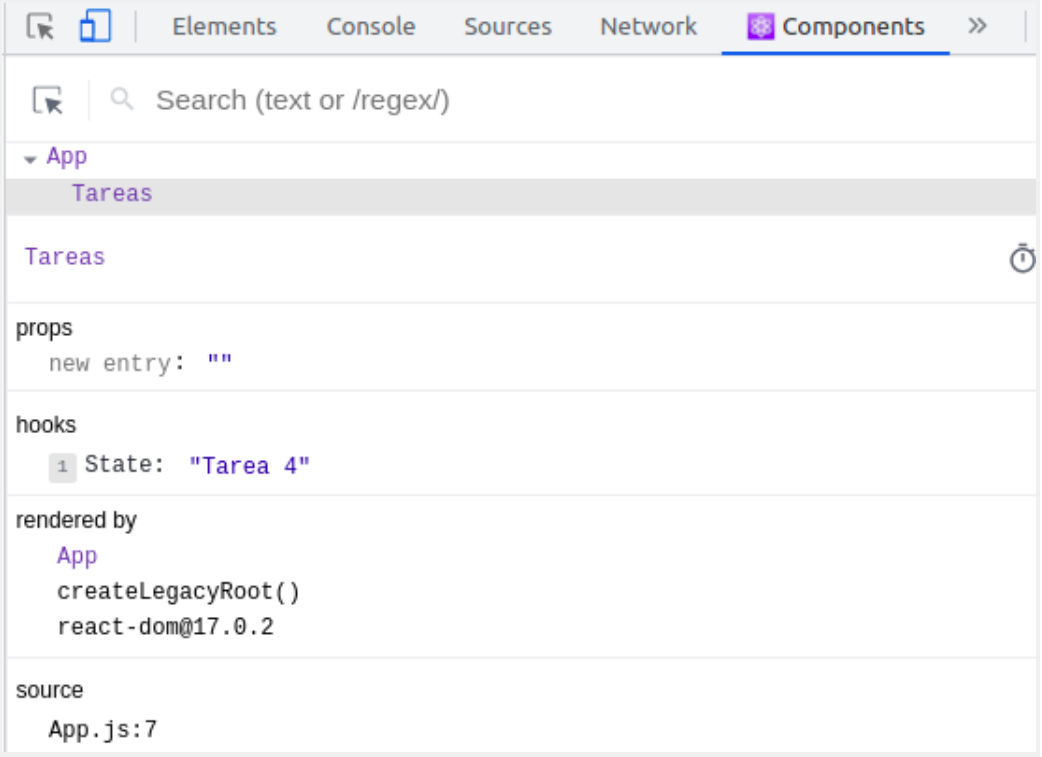


Imagen 4. Estado modificado aplicación
Fuente: Desafío Latam

- ***Paso 6:*** Modificamos la asignación de `listaTareas` agregando un nuevo estado para almacenarlas de forma que cuando incorporemos una nueva la lista se actualice automáticamente.

```
// components/Tareas.jsx

  const [listaTareas, setListaTareas] = useState(["Tarea 1", "Tarea 2", "Tarea 3"])
```

- ***Paso 7:*** Modificamos el método `enviarFormulario` para poder agregar una tarea

```
// components/Tareas.jsx

import { useState } from "react"

const Tareas = () => {
    const [nombreTarea, setNombreTarea] = useState("")
    const [listaTareas, setListaTareas] = useState(["Tarea 1", "Tarea 2", "Tarea 3"])
```

```
    // Función al enviar el formulario
    const enviarFormulario = (e) => {
        e.preventDefault()
        setListaTareas([...listaTareas, nombreTarea]) // Agregamos la
tarea
        setNombreTarea("")  // Vaciamos el formulario

    }

    //Función al escribir sobre el input del formulario
    const capturaInput = (e) => {
        setNombreTarea(e.target.value)
    }

    return (
      <>
        <div>
          <form onSubmit={enviarFormulario}>
            <input name="nombreTarea" onChange={capturaInput} />
            <button> Agregar Tarea </button>
          </form>

          <ul>
            {listaTareas.map(tarea => <li key={tarea}> {tarea} </li> )}
          </ul>
        </div>
      </>
    )
  }

  export default Tareas
```

Al recargar la página deberíamos poder agregar tareas en la lista.

Revisemos el código:

1. La forma de agregar elementos puede parecer extraña, probemos lo siguiente en la consola:

   ```
   arr = [1,2,3,4]
   nuevo_arr = [...arr, 5]
   console.log(nuevo_arr)
   ```

   La razón por la que estamos agregando los elementos de esta forma la explicaremos un poco más adelante.

2. Adicional a lo anterior, el único otro cambio es que borramos el nombre de la tarea del input después de ingresar una tarea.
   Podemos borrar esta línea de código `setNombreTarea("")` recargar la página y agregar una tarea. ¿Qué cambió?

Finalmente, observemos los cambios con nuestra extensión React Developer Tools. En la cual ahora tendremos un estado inicial con un arreglo de tareas y al enviar el formulario se mostrarán las tareas iniciales más la nueva incorporada.


Imagen 5. Estado inicial como arreglo
Fuente: Desafío Latam

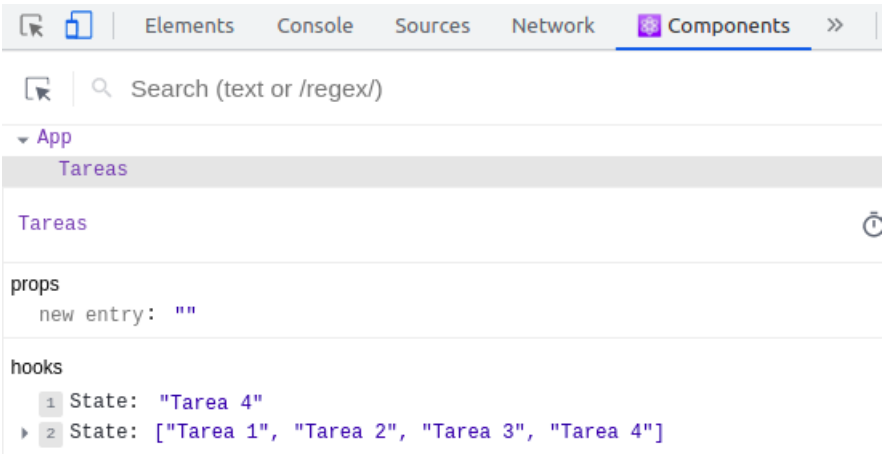


Imagen 6. Estado modificado en arreglo
Fuente: Desafío Latam

¡Felicitaciones! Hasta aquí has creado un pequeño ejercicio simulando una aplicación de tareas. Si lo ves bien, hiciste una pequeña aplicación que tiene las funcionalidades de Crear y Listar elementos, los cuales serían las letras C (*create*) y R (*read*) de un CRUD. Más adelante, en el resto de las unidades veremos con más profundidad los CRUDs en React.

Otro acercamiento para resolver el ejercicio es sin el formulario, y solo con un botón.

## ¿Por qué agregamos los datos al arreglo utilizando el operador spread?

Como hemos visto hasta ahora, utilizando la extensión de chrome cada estado se guarda, de esta forma tenemos un historial de los estados y cada cambio que se haya hecho. Por lo que cuando agregamos estados, tenemos que tener cuidado de no cambiar los anteriores.

Para entender por qué un estado podría cambiar sigamos la siguiente idea.
Supongamos que partimos con el valor de un estado en 1, luego después de un evento, -como un clic - cambia el valor a 2, y luego a 3, etc. Aunque el valor cambie, sabemos que en el estado anterior tenía otro valor

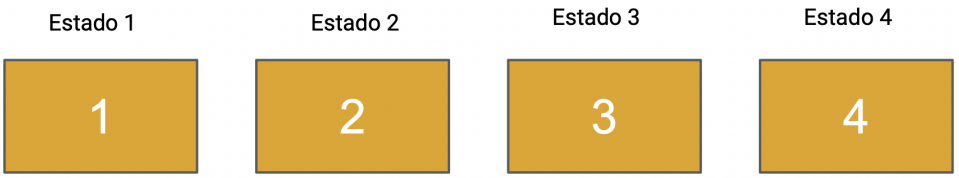


Imagen 7. Cambiando el estado
Fuente: Desafío Latam

Guardando números o strings no tendremos problemas, pero con los arreglos y objetos es un poco más complejo, si no tenemos cuidado al asignar un estado nuevo podríamos modificar el estado anterior.

Si creamos un nuevo arreglo a partir de hacer un push en un arreglo existente, tanto el existente como el anterior cambiarán.

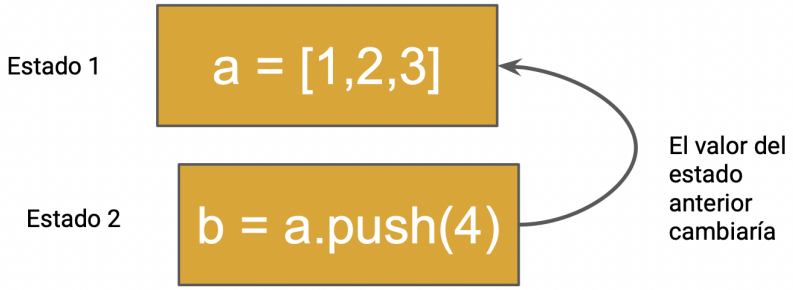


Imagen 8. Cambiando el estado anterior por error
Fuente: Desafío Latam

Una forma de evitar esto es creando un arreglo nuevo, y esto lo logramos con el operador spread.

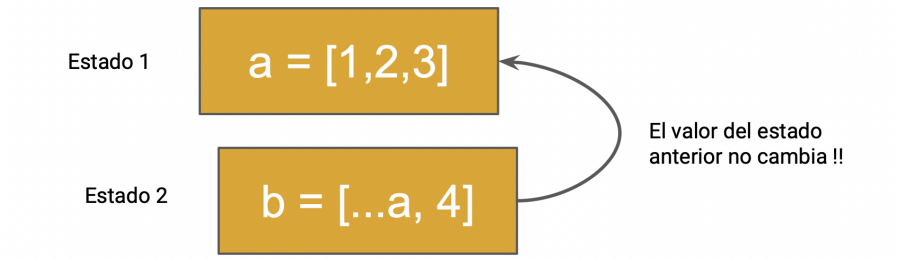


Imagen 9. Creando un nuevo arreglo
Fuente: Desafío Latam

Es por esto que utilizaremos operaciones spread o copiaremos un arreglo o un objeto antes de cambiar un estado.

## Completando tareas

A continuación, veamos qué ocurre cuando se necesita la funcionalidad de poder eliminar aquellas tareas que hayan sido resueltas.

- ***Paso 8:*** Cambiaremos la estructura de datos donde teníamos las tareas por un arreglo de objetos. Este arreglo de objetos tendrá una nueva propiedad que llamaremos `completada` e iniciará con el valor de `false`.
- Nuestra constante se llamará `tareasIniciales`.
- Estos datos los almacenaremos en la carpeta src con el nombre `TareasIniciales.js`.

```
// src/TareasIniciales.js

export const tareasIniciales = [
    {
      nombre: "Tarea 1",
      completada: false,
    },
    {
      nombre: "Tarea 2",
      completada: false,
    },
    {
      nombre: "Tarea 3",
      completada: true,
    }
]
```

- ***Paso 9:*** Importamos nuestras `TareasIniciales.js` en el componente `Tareas.jsx` y lo utilizamos para definir el estado inicial de las tareas. Adicionalmente, como ahora estamos trabajando con objetos donde mostramos el nombre, ahora lo haremos como tarea.nombre, donde el código de nuestro componente quedará de la siguiente manera:

```
// components/Tareas.jsx

import { useState } from "react"
import { tareasIniciales } from "../TareasIniciales"

const Tareas = () => {

    const [nombreTarea, setNombreTarea] = useState("")
    const [listaTareas, setListaTareas] = useState(tareasIniciales)

    // Función al enviar el formulario
    const enviarFormulario = (e) => {
        e.preventDefault()
        setListaTareas([...listaTareas, { nombre: nombreTarea, completada:
false }])

    }

    //Función al escribir sobre el input del formulario
    const capturaInput = (e) => {
        setNombreTarea(e.target.value)
    }

    return (
      <>
        <div>
          <form onSubmit={enviarFormulario}>
            <input name="nombreTarea" onChange={capturaInput} />
            <button> Agregar Tarea </button>
          </form>

          <ul>
            {listaTareas.map(tarea => <li key={tarea.nombre}>
{tarea.nombre}</li> )}
          </ul>
        </div>
      </>
    )
```

```
  }

  export default Tareas
```

Al agregar el nombre, tenemos que marcar las tareas iniciales como no completadas, de forma que los datos que ingresemos sean consistentes con los iniciales.

- ***Paso 10:*** Mostrando tachado las tareas completadas

En la página web marcaremos como tachadas las tareas que ya están completadas, dentro de nuestra lista inicial hay una que cumple esa condición.


Imagen 10. Tareas completadas tachadas.
Fuente: Desafío Latam

Para eso, dentro de cada una de las tareas utilizaremos un render condicional, si cumple la condición de completada agregaremos la propiedad text-decoration: line-through

```
// components/Tareas.jsx

import { useState } from "react"
import { tareasIniciales } from "../TareasIniciales"

const Tareas = () =>
{
   const [nombreTarea, setNombreTarea] = useState("")
   const [listaTareas, setListaTareas] = useState(tareasIniciales)

   // Función al enviar el formulario
   const enviarFormulario = (e) => {
       e.preventDefault()
       setListaTareas([...listaTareas, { nombre: nombreTarea, completada:
false }])
   }

   //Función al escribir sobre el input del formulario
   const capturaInput = (e) => {
       setNombreTarea(e.target.value)
```

```
  }

  return (
    <>
      <div>
        <form onSubmit={enviarFormulario}>
          <input name="nombreTarea" onChange={capturaInput} />
          <button> Agregar Tarea </button>
        </form>

        <ul>
          {listaTareas.map(tarea => (
            <li
              key={tarea.nombre}
              onClick={() => console.log(tarea)}
              style={tarea.completada === true ? { textDecoration:
'line-through' } : {}}>
              {tarea.nombre}
            </li>))}
        </ul>
      </div>
    </>
  )
}
 export default Tareas
```

Si hacemos clic sobre la tarea 1 obtendremos en el log de la consola: {nombre: 'Tarea 1', completada: false}

- ***Paso 11:*** Marcando una tarea como completada

El mismo setter que utilizamos para agregar tareas lo podemos utilizar para completarlas o incluso borrarlas. Para marcar las tareas como completadas agregaremos un botón.


Imagen 11. Botón para tareas completadas.
Fuente: Desafío Latam

Al hacer clic en este botón cambiaremos el estado de la tarea, pero como ahora queremos cambiar solo una tarea, tenemos identificar cuál vamos a completar de la lista.

Para lograr esto, buscaremos el índice de la tarea en el arreglo a partir del nombre y con ese índice, modificaremos su estado.

También tenemos que recordar que no debemos modificar el estado anterior, así que haremos una copia antes de modificarlo. Entonces:

1. Agregamos el botón en el evento `onClick` llamaremos a la función `completarTarea`
2. En la función `completarTarea` copiamos la lista de tareas, buscamos la tarea dentro de la copia, la modificamos y le pasamos al setter para actualizarla.

```
// components/Tareas.jsx
import { useState } from "react"
import { tareasIniciales } from "../tareasIniciales"

const Tareas = () =>
{
   const [nombreTarea, setNombreTarea] = useState("")
   const [listaTareas, setListaTareas] = useState(tareasIniciales)

   // Función al enviar el formulario
   const enviarFormulario = (e) => {
       e.preventDefault()
       setListaTareas([...listaTareas, { nombre: nombreTarea, completada:
false }])
   }

   //Función al escribir sobre el input del formulario
   const capturaInput = (e) => {
       setNombreTarea(e.target.value)
   }
   const completarTarea = (tarea) => {
     const nuevasTareas = [...listaTareas] // Copiamos las tareas anteriores
     const index = nuevasTareas.findIndex(el => el.nombre === tarea.nombre)
// Buscamos la tarea a completar en la lista
     nuevasTareas[index].completada = true
     setListaTareas(nuevasTareas)
   }

   return (
     <>
       <div>
         <form onSubmit={enviarFormulario}>
```

```
          <input name="nombreTarea" onChange={capturaInput} />
          <button> Agregar Tarea </button>
        </form>

        <ul>
           {listaTareas.map(tarea => <li
            key={tarea.nombre}
            style={tarea.completada === true ? { textDecoration:
'line-through' } : {}}>
            {tarea.nombre}
            {tarea.completada === false ? <button onClick={() =>
completarTarea(tarea)}> Completar </button> : ''}
            </li>
          )}
        </ul>
      </div>
    </>
  )
}
 export default Tareas
```

### Entendamos el código

La función `completarTarea` se ejecuta cuando se hace clic en el botón “Completar” a través del evento onClick.
Dentro de esta función tenemos lo siguiente:

- Realizamos una copia del estado anterior de la lista de tareas con el operador de propagación de JavaScript.
  `const nuevasTareas = [...listaTareas]`

- Buscamos el índice de la tarea con el método findIndex y seleccionamos aquel elemento cuyo índice coincida con la propiedad nombre de la tarea.
  `const index = nuevasTareas.findIndex(el => el.nombre === tarea.nombre)`

- Marcamos la tarea como completada `nuevasTareas[index].completada = true`

- Actualizamos el estado pasando a la nueva lista `setListaTareas(nuevasTareas)`

## Eliminando tareas

La última funcionalidad que implementaremos será la de eliminar tareas. Para esto agregaremos un botón al lado de cada tarea, que al hacerle clic nos permitirá borrar la tarea.

- ***Paso 12:*** Esn el `<li>` que muestra las tareas, agregaremos a través del evento `onClick()` una función `eliminarTarea` para eliminar tareas del listado mostrado en el navegador.

```
      <ul>
        {listaTareas.map(tarea => <li
          key={tarea.nombre}
          style={tarea.completada === true ? { textDecoration:
'line-through' } : {}}>
          {tarea.nombre}
          {tarea.completada === false ? <button onClick={() =>
completarTarea(tarea)}> Completar </button> : ''}
          <button onClick={() => eliminarTarea(tarea)}> Borrar
</button>
          </li>)}
      </ul>
```

Nuestro código dejará de funcionar hasta que implementemos el método `eliminarTarea`

- ***Paso 13:*** Para eliminar tareas ocuparemos el método filter, el cual crea un nuevo arreglo con los elementos que cumplen una condición. De esta forma, si queremos mostrar todas las tareas excepto una, filtramos por esa tarea en específico. Este nuevo arreglo se lo pasaremos a `setListaTareas()`

```
// components/Tareas.jsx

import { useState } from "react"
import { tareasIniciales } from "../TareasIniciales"

const Tareas = () =>
{
   const [nombreTarea, setNombreTarea] = useState("")
   const [listaTareas, setListaTareas] = useState(tareasIniciales)

   // Función al enviar el formulario
   const enviarFormulario = (e) => {
       e.preventDefault()
```

```
        setListaTareas([...listaTareas, { nombre: nombreTarea,
completada: false }])
    }

    //Función al escribir sobre el input del formulario
    const capturaInput = (e) => {
        setNombreTarea(e.target.value)
    }
    const completarTarea = (tarea) => {
      const nuevasTareas = [...listaTareas] // Copiamos las tareas
anteriores
      const index = nuevasTareas.findIndex(el => el.nombre ===
tarea.nombre) // Buscamos la tarea a modificar en la lista
      nuevasTareas[index].completada = true
      setListaTareas(nuevasTareas)
    }

    const eliminarTarea = (tarea) => {
       const listaFiltrada = listaTareas.filter(el => el.nombre !==
tarea.nombre)
       setListaTareas(listaFiltrada)
    }

    return (
      <>
        <div>
          <form onSubmit={enviarFormulario}>
            <input name="nombreTarea" onChange={capturaInput} />
            <button> Agregar Tarea </button>
          </form>

          <ul>
            {listaTareas.map(tarea => <li
              key={tarea.nombre}
              style={tarea.completada === true ? { textDecoration:
'line-through' } : {}}>
              {tarea.nombre}
              {tarea.completada === false ? <button onClick={() =>
completarTarea(tarea)}> Completar </button> : ''}
              <button onClick={() => eliminarTarea(tarea)}> Borrar
</button>
              </li>)}
          </ul>
        </div>
```

```
    </>
  )
}
 export default Tareas
```

**Nota:** Podrás acceder al repositorio de este ejercicio en el LMS con el nombre **Repositorio Ejercicio - Lista de Tareas**.

Te invitamos a terminar el CRUD de este ejercicio creando la función para poder editar una tarea.

¡Felicitaciones! Llegaste al final de la aplicación lista de tareas donde creaste paso a paso una mini aplicación web donde se pueden insertar, actualizar y borrar datos. Esta es la base de una gran cantidad de aplicaciones que se pueden construir en React.