Cuso REACT.js

Comandos;

Para crear proyecto;

npx create-react-app nombre-del-proyecto o npx create-react-app nombre-del-proyecto --template typescript

para inicializar y abrir en la web;

npm start

Para comenzar tu camino en el mundo de React necesitas tener una **base sólida** en las principales tecnologías del front end: **HTML**, **CSS** y **JavaScript**, te recomendamos seguir la [**Escuela de Desarrollo Web**](https://platzi.com/web/) en orden.

El trabajo de un desarrollador front-end es darle la mejor experiencia posible a nuestros usuarios en:

* Flujo de navegación
* Tiempos de carga
* Accesibilidad
* Optimización
* Usabilidad

**¿Qué es React.js?**

React es una herramienta de JavaScript en el front end que nos ayuda a construir la parte visual de nuestras aplicaciones web, las interfaces con las que van a interactuar nuestros usuarios.

React es una librería progresiva, esto porque puedes hacer desde una página sencilla hasta poder desarrollar aplicaciones gigantes, y llegar a convertirse en un framework en conjunto de muchas herramientas que existen a su alrededor.

**Pero… ¿Por qué aprender React?**

Mientras más crecen nuestras aplicaciones web, se vuelve más complicado nuestro trabajo utilizando solamente HTML, CSS y JavaScript. La magia de React es que nos facilita crear y organizar nuestro código, acelerar el proceso de creación, además de ofrecernos una mejor experiencia como desarrolladores.

Seguramente por esto quieres aprender React:

* Curiosidad sobre por qué React es tan poderoso
* Quieres conseguir un mejor empleo
* Te apasiona el desarrollo web
* Quieres mejorar tu salario

Sea cual sea tu motivación, React te puede ayudar.

**Requisitos para el curso:**

Profundizar y dominar estos temas retornará buenos frutos y te convertirá en un excelente desarrollador.

* [HTML y CSS](https://platzi.com/cursos/html-css/)
* [JavaScript](https://platzi.com/cursos/basico-javascript/)
* [ECMAScript 6+](https://platzi.com/cursos/ecmascript-6/)
* [Clousures y Scope](https://platzi.com/cursos/scope/)
* [POO](https://platzi.com/cursos/javascript-poo/)
* [JavaScrip Engine](https://platzi.com/cursos/javascript-navegador/)
* [Asincronismo](https://platzi.com/cursos/asincronismo-js/)
* [Peticiones a API](https://platzi.com/cursos/api/)
* [Manipulación del DOM](https://platzi.com/cursos/dom/)
* [Frameworks y librerías de JavaScript](https://platzi.com/cursos/frameworks-javascript/)

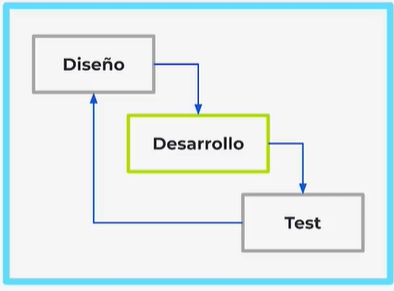
Primer modelo de desarrollo web; cascada

### 1. Modelo en cascada

Cada equipo tiene un tiempo para hacer todo su respectivo trabajo para construir una aplicación.

Por ejemplo: el equipo de diseño diseña la aplicación, el de desarrollo programa todo lo diseñado y finalmente se lanza la app recogiendo el feedback de los usuarios y se vuelve a empezar tomando en cuenta lo dicho por los usuarios.

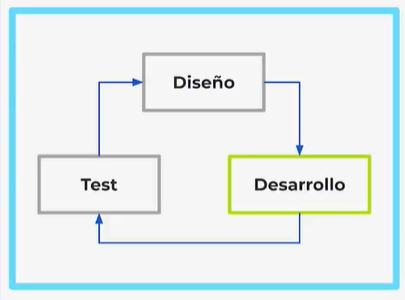
Es problemático porque el esfuerzo puede ser en vano porque no se tiene un feedback hasta que se termina de construir la app.



### 2. Modelo del ciclo MVP iterativo

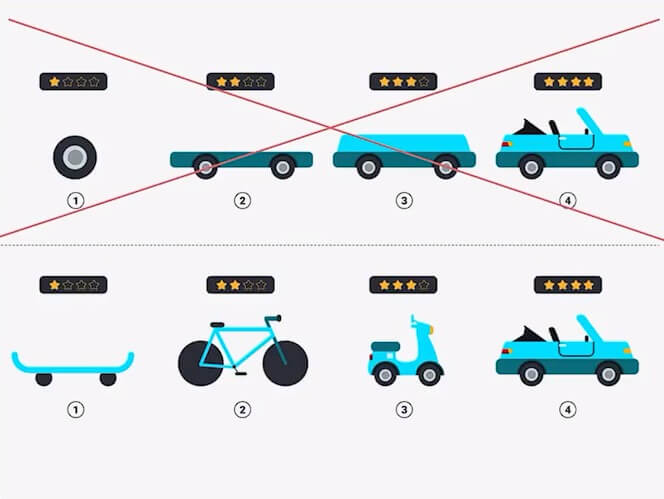
El modelo del ciclo MVP se refiere a que cada equipo se enfocará en ciclos pequeños para construir partes más pequeñas de una app que en conjunto se puedan ir ensamblando para hacer la app completa.

Resuelve el problema del modelo anterior, aquí se recibe el feedback al terminar cada pequeño ciclo, recibiendo así el feedback de manera más inmediata que antes.



## ¿Qué es un MVP?

**MVP (Minimum Viable Product - Producto Mínimo Viable):** se refiere a construir funcionalidades pequeñas que sean completas para que podamos lanzarla y medir su impacto y decidir si seguir ese camino o probar otra cosa.

Hay que escoger el problema más crucial que debe resolver la aplicación.  


React 18 fue publicado el 29 de marzo de 2022. Sus cambios más importantes van a ayudar muchísimo para optimizar el render e hidratación de aplicaciones o componentes de React en el DOM. El más importante, crucial y significativo fue la migración de **ReactDOM.render** a **ReactDOM.createRoot**.

En la próxima clase vamos a usar **Create React App**, una de las herramientas más populares para instalar un entorno de desarrollo con React.js de forma muy rápida. **No hay absolutamente ningún problema si decides usar React 18**. Solo ten en consideración los cambios para hacer el render principal de tu aplicación.

## Migración de ReactDOM.render a ReactDOM.createRoot

Antes de React 18:

**const** root = document.getElementById('root');

ReactDOM.render(e(LikeButton), root);

Desde React 18:

**const** rootElement = document.getElementById('root');

**const** root = ReactDom.createRoot(rootElement);

root.render(e(LikeButton));

Antes de React 18:

ReactDOM.render(

<React.StrictMode>

<App />

</React.StrictMode>,

document.getElementById('root')

);

Desde React 18:

const rootElement = document.getElementById('root');

const root = ReactDom.createRoot(rootElement);

root.render(

<React.StrictMode>

<App />

</React.StrictMode>

);

## Actualizaciones de React

En los siguientes recursos puedes estudiar más a detalle los cambios de React y ReactDOM en su versión 18:

* [React Conf 2021: anuncios, cambios y el futuro de React.js](https://platzi.com/blog/react-conf-2021/)
* [CHANGELOG de React](https://github.com/facebook/react/blob/main/CHANGELOG.md)
* [React 18 is now available on npm!](https://reactjs.org/blog/2022/03/29/react-v18.html)
* [How to Upgrade to React 18](https://reactjs.org/blog/2022/03/08/react-18-upgrade-guide.html)

Además, si quieres conocer un poco más a fondo la filosofía y principios de diseño de React para elegir cómo hacer sus actualizaciones, te recomiendo tomar la siguiente clase:

* [Filosofía y principios de diseño en React](https://platzi.com/clases/2457-react-patrones-render/40852-filosofia-y-principios-de-diseno-en-react/)

⚠️ Si apenas estás empezando tu ruta de aprendizaje con React.js, no tienes que preocuparte por nada de esto. Todas las herramientas se actualizan constantemente, igual que los cursos de Platzi. Esa es la norma y el día a día en el mundo de la tecnología. Más adelante comprenderás la importancia y lo divertido de todas estas actualizaciones.

# Instalación con Create React App

4/23

**RECURSOS**

**MARCADORES**

Una manera muy sencilla de crear un proyecto con React es utilizando **Create React App**, así puedes **iniciar un proyecto** sin preocuparte por la configuración de herramientas como webpack o babel.

💡 Aunque siempre será mejor si nosotros hacemos esta configuración desde cero, ya que tendremos mayor control de todo e incluso nuestra aplicación tendría un mejor rendimiento.

## Maneras de instalar React

Existen varias formas de trabajar con React, cada una tiene sus ventajas y desventajas, siempre que trabajemos con React necesitaremos las dependencias react y react-dom.

### React en JavaScript vanilla

Podemos importar los scripts del código fuente de React, existen las versiones para **desarrollo** que nos facilita debuggear y para **producción** que está optimizada para el producto final.

React con JavaScript vanilla casi no se usa, lo ideal es crear un entorno de desarrollo que facilite nuestro trabajo.

### Configuración manual desde cero

Es la mejor manera si queremos tener control absoluto de nuestro entorno de desarrollo, necesitamos emplear y configurar varias herramientas.

#### Create React App

Esta es la manera más simple y rápida para trabajar con React, solo necesitamos ejecutar el comando: npx create-react-app nombre-del-proyecto o npx create-react-app nombre-del-proyecto --template typescript para typescript y en unos instantes tendremos un entorno de desarrollo totalmente configurado para comenzar a trabajar.

## Entorno de Create React App

Dentro de este entorno tenemos un archivo package.json en el que se encuentran nuestros scripts, dependencias y meta datos de nuestro proyecto, también tendremos dos carpetas principales:

1. **public/:** Es la carpeta pública en donde tendremos nuestro archivo HTML y algunos assets
2. **src/:** Es la carpeta fuente, en donde tendremos todos nuestros archivos de trabajo

Dentro del index.html siempre tendremos un div con un id, como root que será la raíz de nuestro proyecto, y la usaremos para empezar a construir con JavaScript:

<!-- Aquí es en donde todo nuestro código será renderizado. -->

<div id="root"></div>

## ¿Cómo inicializar nuestro servidor?

Para iniciar el entorno de desarrollo podemos ejecutar el comando npm start, con esto tendremos nuestro servidor corriendo en el puerto 3000 y también se refrescará automáticamente con cualquier cambio hecho en el proyecto. (A excepción de los cambios hechos en el archivo index.js).

CLASE VIDEO 5

JSX es una extensión de JavaScript creada por Facebook para usarse con React.js. Nos presenta muchas ventajas el trabajar con elementos y componentes muy similar a la sintaxis de HTML.

La función que JSX tiene es de ser un preprocesador y transformar el código a JavaScript.

💡 JSX es solamente azúcar sintáctica para el método React.createElement(component, props, ...children) de React.

**Nota:** dentro del código encontrarás comentarios que explicarán que es lo que se está añadiendo o algunos consejos.

## ¿Cómo crear un componente?

Existen varias formas de crear un componente en React, por convención siempre los creamos utilizando **PascalCase** (La primera letra de cada palabra en mayúscula y juntas).

### Crear un componente con clases

Este es el modo que se empleaba antes, ahora ya casi nadie la utiliza, pero es bueno saber cómo funciona, por si llegamos a trabajar con proyectos que las usen, con el método render podemos renderizar el JSX que retorna nuestra clase.

**class** **Componente** **extends** **React**.**Component** {

render() {

**return** (

// JSX

)

}

}

Podemos agregar JSX entre los paréntesis del return.

### Crear un componente con funciones

Los componentes funcionales son los más utilizados hoy en día, ya que nos permiten controlar el ciclo de vida mucho más fácil con los [hooks de React](https://platzi.com/cursos/react-hooks/):

**function** **Component**() {

**return** (

// JSX

)

}

// Utilizando arrow function

**const** Component = () => {

**return**(

// JSX

)

}

## Componentes vs. Elementos

Los componentes son invisibles para HTML, pero no para React, de hecho React utiliza los componentes para renderizar, y optimizar los re-renderizados.

### Componente

Un componente es una pieza de código que describe una parte reutilizable de la interfaz, recibe propiedades y retornan elementos, dentro de los componentes podemos utilizar variables de JavaScript con ayuda de las llaves {}.

**const** Component = () => {

**const** titulo = Soy un título;

**return**(

<h1>{titulo}h1>

)

}

### Elemento

Un elemento es lo que devuelve un componente, es una representación de un nodo en el DOM.

<h1>Soy un títuloh1>

## Propiedades vs. Atributos

La diferencia principal es que un atributo no se puede modificar y una propiedad si, ya que los atributos son de HTML y las propiedades son de JavaScript.

### Atributo

Los atributos los pueden tener las etiquetas de HTML.

<h1 class="titulo">Soy un títuloh1>

### Propiedad

Las propiedades las pueden recibir los elementos y componentes en React.

**const** Component = () => {

**return**(

<h1 className="titulo">

Soy un titulo

h1>

)

}

Es importante notar que algunos atributos de HTML se escriben diferente como propiedades, por ejemplo; el atributo class de HTML no se debe utilizar como propiedad de una clase o elemento de React, ya que class es una palabra reservada para crear clases en JavaScript, en su lugar utilizamos className.

## Pasando propiedades a nuestros componentes

Algo mágico de React es que podemos pasarle propiedades a nuestros componentes.

// Le pasamos la propiedad saludo

"Oli" />

// Recibimos las propiedades

const Componente = (props) => {

return(

{/\* ¡Así creamos un comentario en JSX! \*/}

{/\* Accedemos a saludo desde las props \*/}

# {props.saludo}

# {/\* props.salido = Oli \*/}

# 

)

}

## Propiedad children

También podemos utilizar los componentes de React como etiquetas abiertas, para pasarle contenido, elementos o incluso otros componentes, la manera de acceder a ellos es con la propiedad especial children.

# ¡Soy un título anidado!

**const** Componente = (props) => {

**return**(

<div className="titulo">

{props.children}

{/\* props.children = <h1>¡Soy un título anidado!h1> \*/}

div>

)

}

VIDEO CLASE 6

## Análisis de nuestra aplicación

Anteriormente, hemos detectado los componentes y sus comportamientos dentro de nuestro TODO app, necesitamos crear:

* **Counter:** para llevar un conteo de las tareas totales y las completadas.
* **Search**: para filtrar los TODOs de la lista.
* **List**: en donde tendremos cada uno de los TODOs.
* **Item**: será cada uno de los TODOs.
* **Add Todo**: botón para agregar un nuevo TODO.

Adicionalmente, tendremos que crear un modal para ingresar los datos del nuevo TODO (Lo veremos más adelante).

Para empezar a trabajar en el código, primero eliminaremos algunos archivos que no son necesarios para nuestra aplicación, solamente dejaremos dentro de nuestra carpeta src/ los archivos index.js, App.js y los estilos.

## index.js

En la [versión 18 de React](https://platzi.com/clases/2444-react/48389-cambios-en-react-18-reactdomcreateroot/) añadieron cambios importantes, principalmente el método para crear una raíz para renderizar o desmontar:

### Versión menor a React V18

**import** React **from** 'react'

**import** ReactDOM **from** 'react-dom'

**import** App **from** './components/App'

// ReactDOM.render(\_\_qué\_\_, \_\_dónde\_\_);

ReactDOM.render(<App />, document.getElementById('root'))

En la versión menor a la 18 de React, utilizamos el método render de **ReactDOM**, que recibe dos argumentos, el componente que queremos renderizar y el contenedor.

### Versión mayor o igual a React V18

**import** React **from** "react";

**import** ReactDOM **from** "react-dom/client";

**import** App **from** "./App";

**const** root = ReactDOM.createRoot(document.getElementById("root"));

root.render(<App />);

En la versión 18 o superior de React, utilizamos el método createRoot de **ReactDom**, en el que le pasamos el contenedor, guardamos este contenedor en una variable y luego ejecutamos el método render con el componente que queremos renderizar.

## App.js

Dentro de nuestro componente App borraremos todo lo que estaba dentro del componente (No tengas miedo a romper el código), también quitaremos las importaciones de los estilos y de la imagen.

Ahora empezaremos a escribir nuestros componentes, tendremos algo como lo siguiente:

**function** **App**() {

**return** (

<TodoCounter />

<TodoSearch />

<TodoList>

<TodoItem />

</TodoList>

<CreateTodoButton />

);

}

export default App;

Una vez iniciamos el proyecto nos aparecerá un error como el siguiente: SyntaxError: Adjacent JSX elements must be wrapped in an enclosing tag., esto es porque solamente se puede regresar un solo componente al trabajar con JSX, si queremos regresar varios necesitamos encerrarlos en un solo elemento padre.

Para esto existen 2 posibles opciones:

1. Envolver todos nuestros elementos y componentes en un <div></div> u otra etiqueta.
2. Utilizar el componente Fragment, que será invisible al momento de renderizar nuestros elementos en el DOM.

Para no tener problemas con los estilos y no tener que crear etiquetas extras por cada componente, se usa Fragment y para utilizarlo existen 2 formas.

### Maneras de utilizar Fragment en React

import React from "react

function App() {

return (

<React.Fragment>

...

</React.Fragment>

);

}

// Desestructurándolo desde React

import { Fragment } from "react

function App() {

return (

<Fragment>

...

</Fragment>

);

}

O una forma muy común y la más utilizada, envolviendo nuestros elementos dentro de etiquetas vacías, que es lo equivalente a React.Fragment

**function** **App**() {

**return** (

<>

...

</>

);

}

Puedes utilizar la manera que veas más cómoda.

Ahora si vemos nuestro proyecto nos aparecerá otro error, como ya te imaginas es porque no hemos creado nuestros componentes, eso es justo lo que haremos.

Para esto utilizaremos los módulos de JavaScript, existen varias maneras de **exportar** nuestros componentes, podemos exportarlos por defecto, pero no es una buena práctica porque puede causar futuros problemas al **importar** estos componentes.

Para evitar este problema debemos intentar evitar export default, y exportar nuestros componentes por su nombre, por ejemplo: export { Componente };, así lo tendremos que importar exactamente como lo exportamos: import { Componente } from './Componente';

### TodoCounter

**import** React **from** "react";

**function** **TodoCounter**(){

**return**(

<h2> Has complentado 2 de 3 ToDos</h2>

)

}

**export** {TodoCounter};

### TodoSearch

**import** react **from** "react";

**function** **TodoSearch**(){

**return**(

<input placeholder="Cebolla" />

);

}

export {TodoSearch};

### TodoList

**import** react **from** "react";

**function** **TodoList**(props){

**return**(

<section>

<ul>

{props.children}

</ul>

</section>

);

}

**export** { TodoList};

### TodoItem

**import** react **from** "react";

**function** **TodoItem**(props){

**return**(

<li>

<span>C</span>

<p>{props.text}</p>

<span>X</span>

</li>

);

}

**export** { TodoItem };

### CreateTodoButtom

**import** react **from** "react";

**function** **CreateTodoButtom**(){

**return**(

<button>+</button>

);

}

**export** { CreateTodoButtom};

## Importar componentes

Una vez creamos nuestros componentes no se nos olvide importarlos en App.js para ahora si usarlos.

**import** react **from** "react";

**import** { TodoCounter } **from** "./TodoCounter";

**import** { TodoSearch } **from** "./TodoSearch.js";

**import** { TodoList } **from** "./TodoList.js";

**import** { TodoItem } **from** "./TodoItem.js";

**import** { CreateTodoButtom } **from** "./CreateTodoButton.js";

**function** **App**() {

**return** (

<react.Fragment>

<TodoCounter />

<TodoSearch />

<TodoList>

<TodoItem />

</TodoList>

<CreateTodoButtom />

</react.Fragment>

);

}

export default App;

## Lógica para renderizar nuestros TODOs

Ahora crearemos la lógica para renderizar todas las TODOs que creen nuestros usuarios, nuestro componente ya recibe una propiedad llamada texto, entonces creemos un arreglo con algunas tareas dentro de App.js y ahora **mapearemos** cada una de nuestras TODOs con map y por cada TODO regresaremos un componente TodoItem.

**import** react **from** "react";

**import** { TodoCounter } **from** "./TodoCounter";

**import** { TodoSearch } **from** "./TodoSearch.js";

**import** { TodoList } **from** "./TodoList.js";

**import** { TodoItem } **from** "./TodoItem.js";

**import** { CreateTodoButtom } **from** "./CreateTodoButton.js";

**const** todos=[

{text:'Cortar cebolla', completed:**false**},

{text:'Tormar el curso de intro a react', completed:**false**},

{text:'Llorar con la llorona', completed:**false**}

];

**function** **App**() {

**return** (

<react.Fragment>

<TodoCounter />

<TodoSearch />

<TodoList>

{todos.map(todo => (

<TodoItem text={todo.text} />

))}

</TodoList>

<CreateTodoButtom />

</react.Fragment>

);

}

export default App;

¡Nuestra aplicación está tomando forma! Todavía nos faltan algunas cosas, pero… ¿Notaste que en la consola ahora nos aparece este error? Warning: Each children in a list should have a unic "key" prop.

Esto es porque cuando renderizamos varios elementos en una lista debemos que pasarle una propiedad especial a cada item, que es key, esta propiedad ayuda a React para mantener un registro de los elementos que han cambiado, y saber cuál elemento es cuál, también es importante que esta propiedad no se repita en ningún otro item.

Entonces añadamos esta propiedad:

**import** react **from** "react";

**import** { TodoCounter } **from** "./TodoCounter";

**import** { TodoSearch } **from** "./TodoSearch.js";

**import** { TodoList } **from** "./TodoList.js";

**import** { TodoItem } **from** "./TodoItem.js";

**import** { CreateTodoButtom } **from** "./CreateTodoButton.js";

**const** todos = [

{text:'Cortar cebolla', completed:**false**},

{text:'Tormar el curso de intro a react', completed:**false**},

{text:'Llorar con la llorona', completed:**false**}

];

**function** **App**() {

**return** (

<react.Fragment>

<TodoCounter />

<TodoSearch />

<TodoList>

{todos.map(todo => (

<TodoItem key={todo.text} text={todo.text} />

))}

</TodoList>

<CreateTodoButtom />

</react.Fragment>

);

}

export default App;

# Manejo de eventos

8/23

**RECURSOS**

**MARCADORES**

Manejar eventos en React es muy similar a manejar eventos en el DOM, solo necesitamos pasarle una propiedad **on + evento**, por ejemplo: onClick, onChange, onMouseOver, que será igual a una función en la que estará el código que se ejecutará cuando ocurra dicho evento.

A diferencia de los eventos del DOM, para manejar eventos en React tenemos unas pequeñas diferencias en la sintaxis:

1. En React los eventos son nombrados usando **camelCase**.
2. Tenemos que pasar una función, ya sea en línea o almacenada en una variable.
3. No puedes regresar false para prevenir el comportamiento por defecto, debes utilizar preventDefault explícitamente.

**HTML**

<button onclick="click()"></button>

**React**

<button onClick={click}></button>

## Pasando argumentos a escuchadores de eventos

En React no tenemos que ejecutar el código nosotros, React ya maneja esto por debajo, solo es necesario pasar una función, React solito la ejecutará cuando ocurra el evento que estemos escuchando.

Si necesitamos pasar argumentos a nuestras funciones, necesitamos encerrar nuestra función dentro de otra función, esto porque al pasarle argumentos a una función la estamos ejecutando, veamos un ejemplo:

**function** **CreateTodoButton**(props) {

**const** onClickButton = (msg) => {

alert(msg);

};

**return** (

<>

{/\* ✅ \*/}

<button

className="CreateTodoButton"

onClick={() => onClickButton('Aquí se debería abrir el modal')}

>

+

</button>

{/\* ❌ \*/}

<button

className="CreateTodoButton"

onClick={onClickButton('Esta función se ejecuta al inicio, no al presionar el botón'}

>

+

</button>

</>

);

}

✅ Es importante siempre pasar una función.

Dentro de estos eventos también puedes recibir como parámetro la información del evento, en donde puedes encontrar propiedades muy interesantes, como por ejemplo, el valor de algún input, con event.target.value.

**function** **TodoSearch**() {

**const** onSearchValueChange = (event) => {

console.log(event.target.value);

};

**return** (

<input

className="TodoSearch"

placeholder="Cebolla"

onChange={onSearchValueChange}

/>

);

}

Clase 8 manejo de eventos

Manejar eventos en React es muy similar a manejar eventos en el DOM, solo necesitamos pasarle una propiedad **on + evento**, por ejemplo: onClick, onChange, onMouseOver, que será igual a una función en la que estará el código que se ejecutará cuando ocurra dicho evento.

A diferencia de los eventos del DOM, para manejar eventos en React tenemos unas pequeñas diferencias en la sintaxis:

1. En React los eventos son nombrados usando ***camelCase***.
2. Tenemos que pasar una función, ya sea en línea o almacenada en una variable.
3. No puedes regresar false para prevenir el comportamiento por defecto, debes utilizar preventDefault explícitamente.

**HTML**

<button onclick="click()"></button>

**React**

<button onClick={click}></button>

**Pasando argumentos a escuchadores de eventos**

En React no tenemos que ejecutar el código nosotros, React ya maneja esto por debajo, solo es necesario pasar una función, React solito la ejecutará cuando ocurra el evento que estemos escuchando.

Si necesitamos pasar argumentos a nuestras funciones, necesitamos encerrar nuestra función dentro de otra función, esto porque al pasarle argumentos a una función la estamos ejecutando, veamos un ejemplo:

**function** **CreateTodoButton**(props) {

**const** onClickButton = (msg) => {

alert(msg);

};

**return** (

<>

{/\* ✅ \*/}

<button

className="CreateTodoButton"

onClick={() => onClickButton('Aquí se debería abrir el modal')}

>

+

</button>

{/\* ❌ \*/}

<button

className="CreateTodoButton"

onClick={onClickButton('Esta función se ejecuta al inicio, no al presionar el botón'}

>

+

</button>

</>

);

}

✅ Es importante siempre pasar una función.

Dentro de estos eventos también puedes recibir como parámetro la información del evento, en donde puedes encontrar propiedades muy interesantes, como por ejemplo, el valor de algún input, con event.target.value.

**function** **TodoSearch**() {

**const** onSearchValueChange = (event) => {

console.log(event.target.value);

};

**return** (

<input

className="TodoSearch"

placeholder="Cebolla"

onChange={onSearchValueChange}

/>

);

}

Clase video 9

El estado en React nos ayuda a crear **datos mutables** o datos que pueden ser modificados.

Para manejar el estado depende del si nuestro componente es generado con una clase o si es un componente funcional.

* **Clase:** Para manejar el estado dentro de una clase podemos crearlo en el constructor de la clase, y para actualizarlo utilizamos el **setter** setState
* **Función** Si estamos en un componente funcional podemos utilizar el hook de estado, que nos regresará arreglo con un **getter** (que será el valor de nuestro estado) y un **setter** (una función para actualizar el estado).

**¿Qué son los Hooks?**

Los **Hooks** son funciones que te permiten **enganchar** el estado de React y el ciclo de vida desde componentes funcionales, entre muchas otras cosas. Nos permiten usar React sin clases.

**Estado en componentes clase**

Para manejar el estado en componentes clase necesitamos crearlo como una propiedad dentro de nuestro componente clase, usamos this.state, y para actualizar el estado, la clase de React ya tiene un setter: this.setState. (Dentro de este tipo de componentes no se pueden usar los hooks).

**class** **Component** **extends** **React**.**Component** {

**constructor**(){

**this**.state = {

patito: '👍'

}

}

render(){

**return** (

<button onClick={()=>this.setState("Has dado un like")}>{this.state.patito}</button>

)

}

}

**Estado en componentes funcionales**

El manejo del estado en estos componentes es mucho más sencillo, utilizando el **hook de estado**.  
Podemos importar este hook directamente de React o desestructurándolo de React:

**import** React **from** 'react';

**function** **Component**() {

**const** [count, setCount] = React.useState("");

}

**import** { useState } **from** 'react';

**function** **Component**() {

**const** [count, setCount] = useState("");

}

Una vez lo importamos ya podemos usarlo en nuestro componente, este hook nos regresará un array con dos elementos:

1. El valor de nuestro estado **(Getter)**.
2. La función que se ocupa de actualizar nuestro estado **(Setter)**.

También podemos pasarle un valor inicial a nuestro estado dentro de los paréntesis.  
Por ejemplo, en el buscador de nuestra aplicación,

**import** React **from** 'react';

**import** './TodoSearch.css';

**function** **TodoSearch**() {

**const** [searchValue, setSearchValue] = React.useState('');

**const** onSearchValueChange = (event) => {

console.log(event.target.value);

setSearchValue(event.target.value);

};

**return** [

<input

className="TodoSearch"

placeholder="Cebolla"

value={searchValue}

onChange={onSearchValueChange}

/>,

<p>{searchValue}</p>

];

}

export { TodoSearch };

**Tipos de componentes**

**Stateful:** son componentes que tienen declaración de estado en su función.

**Stateless:** son componentes que no tienen ningún tipo de estado declarado en el cuerpo de la función.

Clase video 10, contando y buscando TODOs

El **levantamiento de estado** es una técnica de React que pone el estado en una localización donde se pueda pasar como **props** a los componentes.

Lo ideal es poner el estado en el lugar más cercano a todos los componentes que quieren compartir esa información, así todos nuestros componentes tendrán el mismo estado y cuando este cambie sólo re-renderizará lo necesario.

Esto es justamente lo que haremos ahora para hacer funcionar nuestro contador y nuestro buscador, pero debemos tener mucho **cuidado al manejar re-renderizados**, porque estos pueden causar una mala experiencia de usuario o incluso romper nuestra aplicación.

## Contando TODOs

Dentro de nuestro componente App.js primero necesitamos crear el estado de nuestros TODOs, para apartir de ahí poder saber cuántos TODOs existen y cuántos están completados.

**import** React **from** 'react';

**import** { TodoCounter } **from** './TodoCounter';

**import** { TodoSearch } **from** './TodoSearch';

**import** { TodoList } **from** './TodoList';

**import** { TodoItem } **from** './TodoItem';

**import** { CreateTodoButton } **from** './CreateTodoButton';

// import './App.css';

**const** defaultTodos = [

{ text: 'Cortar cebolla', completed: **true** },

{ text: 'Tomar el cursso de intro a React', completed: **false** },

{ text: 'Llorar con la llorona', completed: **true** },

{ text: 'LALALALAA', completed: **false** },

];

**function** **App**() {

// Estado inicial de nuestros TODOs

**const** [todos, setTodos] = React.useState(defaultTodos);

// Cantidad de TODOs completados

**const** completedTodos = todos.filter(todo => !!todo.completed).length;

// Cantidad total de TODOs

**const** totalTodos = todos.length;

**return** (

<React.Fragment>

{/\* Pasamos el estado a nuestro componente \*/}

<TodoCounter

total={totalTodos}

completed={completedTodos}

/>

<TodoSearch />

<TodoList>

{searchedTodos.map(todo => (

<TodoItem

key={todo.text}

text={todo.text}

completed={todo.completed}

/>

))}

</TodoList>

<CreateTodoButton />

</React.Fragment>

);

}

export default App;

Una vez teniendo estos datos, ya podemos recibir las props en nuestro contador.

**import** React **from** 'react';

**import** './TodoCounter.css';

// Desestructuramos los props que pasamos al componente

**function** **TodoCounter**({ total, completed }) {

**return** (

<h2 className="TodoCounter">Has completado {completed} de {total} TODOs</h2>

);

}

**export** { TodoCounter };

## Buscando TODOs

Para esto haremos algo parecido a lo que hicimos para contar nuestros TODOs, para tener acceso al valor de la búsqueda desde nuestro componente App.js y ahí filtrar nuestros TODOs.

Primero crearemos nuestro estado de búsqueda en App.js, y utilizaremos el método filter de JavaScript para filtrar los TODOs que coincidan con nuestra búsqueda, y también haremos uso de toLowerCase(), para poder filtrar sin importar si las letras son mayúsculas o minúsculas.

**import** React **from** 'react';

**import** { TodoCounter } **from** './TodoCounter';

**import** { TodoSearch } **from** './TodoSearch';

**import** { TodoList } **from** './TodoList';

**import** { TodoItem } **from** './TodoItem';

**import** { CreateTodoButton } **from** './CreateTodoButton';

// import './App.css';

**const** defaultTodos = [

{ text: 'Cortar cebolla', completed: **true** },

{ text: 'Tomar el cursso de intro a React', completed: **false** },

{ text: 'Llorar con la llorona', completed: **true** },

{ text: 'LALALALAA', completed: **false** },

];

**function** **App**() {

**const** [todos, setTodos] = React.useState(defaultTodos);

// El estado de nuestra búsqueda

**const** [searchValue, setSearchValue] = React.useState('');

**const** completedTodos = todos.filter(todo => !!todo.completed).length;

**const** totalTodos = todos.length;

// Creamos una nueva variable en donde guardaremos las coincidencias con la búsqueda

**let** searchedTodos = [];

// Lógica para filtrar

**if** (!searchValue.length >= 1) {

searchedTodos = todos;

} **else** {

searchedTodos = todos.filter(todo => {

**const** todoText = todo.text.toLowerCase();

**const** searchText = searchValue.toLowerCase();

**return** todoText.includes(searchText);

});

}

**return** (

<React.Fragment>

<TodoCounter

total={totalTodos}

completed={completedTodos}

/>

<TodoSearch

searchValue={searchValue}

setSearchValue={setSearchValue}

/>

<TodoList>

{/\* Regresamos solamente los TODOs buscados \*/}

{searchedTodos.map(todo => (

<TodoItem

key={todo.text}

text={todo.text}

completed={todo.completed}

/>

))}

</TodoList>

<CreateTodoButton />

</React.Fragment>

);

}

export default App;

clase video 11; eliminando todos

Para crear las funcionalidades de completar y eliminar TODOs podemos crear una función que reciba el **id** o **texto** de nuestro TODO, para después editarlo o eliminarlo.

## Completando TODOs

Creamos la función completeTodo, que recibirá el texto de nuestro TODO, ubicamos el TODO en nuestro arreglo, cambiamos el valor de la propiedad completed de nuestro TODO y muy importante actualizar nuestro estado, para que React pueda re-renderizar nuestros TODOs con los nuevos datos.

**const** completeTodo = (text) => {

**const** todoIndex = todos.findIndex(todo => todo.text === text);

**const** newTodos = [...todos];

newTodos[todoIndex].completed = **true**;

setTodos(newTodos);

};

## Eliminando TODOs

Podemos hacer algo parecido a la función de completar, pero ahora, en lugar de cambiar si está completada o no, solamente la eliminaremos de nuestros TODOs con el método splice, y también regresaremos un nuevo arreglo con los TODOs actualizados.

**const** deleteTodo = (text) => {

**const** todoIndex = todos.findIndex(todo => todo.text === text);

**const** newTodos = [...todos];

newTodos.splice(todoIndex, 1);

setTodos(newTodos);

};

## App.js

Una vez tenemos creada la lógica para completar y eliminar TODOs, podemos pasar esas funciones a nuestros TodoItem.

**import** React **from** 'react';

**import** { TodoCounter } **from** './TodoCounter';

**import** { TodoSearch } **from** './TodoSearch';

**import** { TodoList } **from** './TodoList';

**import** { TodoItem } **from** './TodoItem';

**import** { CreateTodoButton } **from** './CreateTodoButton';

// import './App.css';

**const** defaultTodos = [

{ text: 'Cortar cebolla', completed: **true** },

{ text: 'Tomar el cursso de intro a React', completed: **false** },

{ text: 'Llorar con la llorona', completed: **true** },

{ text: 'LALALALAA', completed: **false** },

];

**function** **App**() {

**const** [todos, setTodos] = React.useState(defaultTodos);

**const** [searchValue, setSearchValue] = React.useState('');

**const** completedTodos = todos.filter(todo => !!todo.completed).length;

**const** totalTodos = todos.length;

**let** searchedTodos = [];

**if** (!searchValue.length >= 1) {

searchedTodos = todos;

} **else** {

searchedTodos = todos.filter(todo => {

**const** todoText = todo.text.toLowerCase();

**const** searchText = searchValue.toLowerCase();

**return** todoText.includes(searchText);

});

}

**const** completeTodo = (text) => {

**const** todoIndex = todos.findIndex(todo => todo.text === text);

**const** newTodos = [...todos];

newTodos[todoIndex].completed = **true**;

setTodos(newTodos);

};

**const** deleteTodo = (text) => {

**const** todoIndex = todos.findIndex(todo => todo.text === text);

**const** newTodos = [...todos];

newTodos.splice(todoIndex, 1);

setTodos(newTodos);

};

**return** (

{searchedTodos.map(todo => (

completeTodo(todo.text)}

onDelete={() => deleteTodo(todo.text)}

/>

))}

);

}

export default App;

## TodoItem.js

Para que nuestra aplicación funcione también tenemos que recibir las props en nuestros ítems y usarlas.

**import** React **from** 'react';

**import** './TodoItem.css';

**function** **TodoItem**(props) {

**return** (

<li className="TodoItem">

<span

className={`Icon Icon-check ${props.completed && 'Icon-check--active'}`}

onClick={props.onComplete}

>

√

span>

<p className={`TodoItem-p ${props.completed && 'TodoItem-p--complete'}`}>

{props.text}

p>

<span

className="Icon Icon-delete"

onClick={props.onDelete}

>

X

span>

li>

);

}

**export** { TodoItem };

clase video 12; organización de archivos y carpetas

La organización de un proyecto es algo muy importante, nos ayuda a tener un mejor control y orden sobre nuestra aplicación.

## ¿Cuál es la mejor estructura?

**No existe una mejor estructura** de carpetas, existen varias estructuras, y la más utilizada es la **agrupación por tipo de archivo**, puedes usar la que más te guste, la que mejor se adapte a tu proyecto, o ¡incluso crear una propia!

## Agrupación por tipo de archivo

En esta estructura solo agrupamos los **archivos similares**, es la más recomendada para proyectos grandes, también sirve para tener una excelente organización en proyectos pequeños, por ejemplo:

└── src/

├── assets/

│ ├── img/

│ └── foto.jpg

│ ├── fonts/

│ └──ubuntu.woff2

├── components/

│ ├── CreateTodoButton/

│ ├── index.js

│ └── CreateTodoButton.css

│ ├── TodoCounter/

│ ├── TodoItem/

│ ├── TodoList/

│ ├── TodoSearch/

├── context/

│ ├── TodoContext.js

├── hooks/

│ ├── useLocalStorage.js

├── utils/

│ ├── fetch.js

## Creando la estructura de nuestro proyecto

Puedes usar la estructura que más te guste, en nuestra aplicación, ya que son muy pocos componentes, solamente crearemos una carpeta por cada componente.

Una vez tengamos esto listo, ahora podemos importarlo dentro de nuestro archivo App.js, pero si queremos seguir las reglas de stateful y stateless, no tiene mucho sentido, por eso vamos a abstraer la UI de nuestro archivo App.js en otro componente que llamaremos AppUI.js:

### App/index.js

**import** React **from** 'react';

**import** { AppUI } **from** './AppUI';

**const** defaultTodos = [

{ text: 'Cortar cebolla', completed: **true** },

{ text: 'Tomar el cursso de intro a React', completed: **false** },

{ text: 'Llorar con la llorona', completed: **true** },

{ text: 'LALALALAA', completed: **false** },

];

**function** **App**() {

**const** [todos, setTodos] = React.useState(defaultTodos);

**const** [searchValue, setSearchValue] = React.useState('');

**const** completedTodos = todos.filter(todo => !!todo.completed).length;

**const** totalTodos = todos.length;

**let** searchedTodos = [];

**if** (!searchValue.length >= 1) {

searchedTodos = todos;

} **else** {

searchedTodos = todos.filter(todo => {

**const** todoText = todo.text.toLowerCase();

**const** searchText = searchValue.toLowerCase();

**return** todoText.includes(searchText);

});

}

**const** completeTodo = (text) => {

**const** todoIndex = todos.findIndex(todo => todo.text === text);

**const** newTodos = [...todos];

newTodos[todoIndex].completed = **true**;

setTodos(newTodos);

};

**const** deleteTodo = (text) => {

**const** todoIndex = todos.findIndex(todo => todo.text === text);

**const** newTodos = [...todos];

newTodos.splice(todoIndex, 1);

setTodos(newTodos);

};

**return** (

<AppUI

totalTodos={totalTodos}

completedTodos={completedTodos}

searchValue={searchValue}

setSearchValue={setSearchValue}

searchedTodos={searchedTodos}

completeTodo={completeTodo}

deleteTodo={deleteTodo}

/>

);

}

export default App;

### App/AppUI.js

**import** React **from** 'react';

**import** { TodoCounter } **from** '../TodoCounter';

**import** { TodoSearch } **from** '../TodoSearch';

**import** { TodoList } **from** '../TodoList';

**import** { TodoItem } **from** '../TodoItem';

**import** { CreateTodoButton } **from** '../CreateTodoButton';

**function** **AppUI**({

totalTodos,

completedTodos,

searchValue,

setSearchValue,

searchedTodos,

completeTodo,

deleteTodo,

}) {

**return** (

<React.Fragment>

<TodoCounter

total={totalTodos}

completed={completedTodos}

/>

<TodoSearch

searchValue={searchValue}

setSearchValue={setSearchValue}

/>

<TodoList>

{searchedTodos.map(todo => (

<TodoItem

key={todo.text}

text={todo.text}

completed={todo.completed}

onComplete={() => completeTodo(todo.text)}

onDelete={() => deleteTodo(todo.text)}

/>

))}

</TodoList>

<CreateTodoButton />

</React.Fragment>

);

}

export { AppUI };

# clase video 14 Custom Hook para Local Storage

Algo super interesante de React es que podemos crear **hooks personalizados** para ejecutar procesos para manejar información sin que afecte a otros componentes, lo que haremos será abstraer nuestra lógica de localStorage para manejarla dentro de nuestro propio **hook**.

**Reglas para crear un custom hook:**

1. Nuestro hook personalizado debe empezar por **use**, por ejemplo: *usePatito*, *useTodos* o *useUnicornio*.
2. **No anidar hooks** en loops u otros bloques.
3. Llamar dentro de componentes de React o hooks propios, nunca dentro de funciones normales.

**Creando nuestro Custom Hook**

El objetivo de un custom hook es **reutilizar código**, entonces este hook debería poder funcionar para guardar cualquier tipo de dato en el localStorage.

Primero necesitamos analizar que **parámetros** necesita tener nuestro custom hook:

1. Un nombre para el item en nuestro localStorage.
2. Un estado inicial

También tenemos que regresar algunos datos para que nuestro hook sea funcional:

1. Los datos actuales de nuestro ítem en el localStorage.
2. Una función para actualizar los datos de este ítem.

¡Ahora que sabemos qué tenemos que hacer, podemos empezar a crear nuestro custom hook!

// Recibimos como parámetros el nombre y el estado inicial de nuestro item.

**function** **useLocalStorage**(itemName, initialValue) {

// Guardamos nuestro item en una constante

**const** localStorageItem = localStorage.getItem(itemName);

**let** parsedItem;

// Utilizamos la lógica que teníamos, pero ahora con las variables y parámentros nuevos

**if** (!localStorageItem) {

localStorage.setItem(itemName, JSON.stringify(initialValue));

parsedItem = initialValue;

} **else** {

parsedItem = JSON.parse(localStorageItem);

}

// ¡Podemos utilizar otros hooks!

**const** [item, setItem] = React.useState(parsedItem);

// Actualizamos la función para guardar nuestro item con las nuevas variables y parámetros

**const** saveItem = (newItem) => {

**const** stringifiedItem = JSON.stringify(newItem);

localStorage.setItem(itemName, stringifiedItem);

setItem(newItem);

};

// Regresamos los datos que necesitamos

**return** [

item,

saveItem,

];

}

Ahora que hemos creado nuestro custom hook podemos usarlo las veces que queramos.  
¡Vamos a añadirlo a la lógica de nuestra aplicación!

**import** React **from** 'react';

**import** { AppUI } **from** './AppUI';

// const defaultTodos = [

// { text: 'Cortar cebolla', completed: true },

// { text: 'Tomar el cursso de intro a React', completed: false },

// { text: 'Llorar con la llorona', completed: true },

// { text: 'LALALALAA', completed: false },

// ];

**function** **useLocalStorage**(itemName, initialValue) {

**const** localStorageItem = localStorage.getItem(itemName);

**let** parsedItem;

**if** (!localStorageItem) {

localStorage.setItem(itemName, JSON.stringify(initialValue));

parsedItem = initialValue;

} **else** {

parsedItem = JSON.parse(localStorageItem);

}

**const** [item, setItem] = React.useState(parsedItem);

**const** saveItem = (newItem) => {

**const** stringifiedItem = JSON.stringify(newItem);

localStorage.setItem(itemName, stringifiedItem);

setItem(newItem);

};

**return** [

item,

saveItem,

];

}

**function** **App**() {

// Desestructuramos los datos que retornamos de nuestro custom hook, y le pasamos los argumentos que necesitamos (nombre y estado inicial)

**const** [todos, saveTodos] = useLocalStorage('TODOS\_V1', []);

**const** [searchValue, setSearchValue] = React.useState('');

**const** completedTodos = todos.filter(todo => !!todo.completed).length;

**const** totalTodos = todos.length;

**let** searchedTodos = [];

**if** (!searchValue.length >= 1) {

searchedTodos = todos;

} **else** {

searchedTodos = todos.filter(todo => {

**const** todoText = todo.text.toLowerCase();

**const** searchText = searchValue.toLowerCase();

**return** todoText.includes(searchText);

});

}

**const** completeTodo = (text) => {

**const** todoIndex = todos.findIndex(todo => todo.text === text);

**const** newTodos = [...todos];

newTodos[todoIndex].completed = **true**;

saveTodos(newTodos);

};

**const** deleteTodo = (text) => {

**const** todoIndex = todos.findIndex(todo => todo.text === text);

**const** newTodos = [...todos];

newTodos.splice(todoIndex, 1);

saveTodos(newTodos);

};

**return** (

<AppUI

totalTodos={totalTodos}

completedTodos={completedTodos}

searchValue={searchValue}

setSearchValue={setSearchValue}

searchedTodos={searchedTodos}

completeTodo={completeTodo}

deleteTodo={deleteTodo}

/>,

);

}

export default App;

Ahora nuestro código está mucho mejor organizado, si queremos tener aún más control de nuestro proyecto, incluso podemos crear una carpeta para hooks, y luego poder importarlos a cualquier parte de nuestro proyecto.  
¡Te retamos a que lo hagas!

El shortcut para seleccionar todas las palabras iguales sin seleccionar una por una, es: dar click sobre la palabra (localStorageItem) y luego Ctrl+Shift+L

Clase video 15 MANEJO DE EFECTOS

El **hook de efecto** en react nos permite ejecutar un pedazo de código cada vez que necesitemos, a lo largo de la vida de nuestro componente, también cuando se cumplan ciertas condiciones.

Algo curioso e importante de saber es que el código dentro de nuestro **hook de efecto** no se ejecuta como el resto del código, se ejecutará inicialmente justo antes de hacer el renderizado del HTML que resulte de nuestro código de React.

## Condiciones para nuestro hook de efecto

El hook de React, useEffect, puede recibir dos argumentos:

1. Función que se ejecutará en cada fase del ciclo de vida de nuestro componente.
2. Las condiciones de cuándo debe ejecutarse esta función dentro de un arreglo, cada que se actualice cualquier dato que le pasemos a este arreglo, se volverá a ejecutar nuestra función.

React.useEffect(funcion, [dato1, dato2, datoN])

## Diferentes maneras de actualizar nuestros componentes

Existen tres diferentes maneras para aplicar el hook de efecto, todas funcionan diferente a la hora de re-renderizar nuestros componentes.

1. **Sin pasar un arreglo como segundo argumento: useEffect(funcion)**  
   Cuando no le pasamos un segundo argumento con las condiciones de cuándo se debe re-ejecutar nuestra función, React tomará como condiciones que se debe ejecutar nuestra función todas las veces que ocurra un re-renderizado, y también cada vez que se actualice alguna **prop** (Si es que el componente recibe alguna).
2. **Pasando un arreglo vacío: useEffect(funcion, [])**  
   Cuando pasamos un arreglo vacío, le estás diciendo a React que no hay ninguna condición para volver a ejecutar el código de nuestra función, entonces solamente se ejecutará en el renderizado inicial.
3. **Pasando un arreglo con datos: useEffect(funcion, [val1, val2, valN])**  
   Cuando especificas las condiciones dentro del arreglo del segundo parámetro, le estás diciendo a React que ejecute nuestra función en el renderizado inicial y también cuando algún dato del arreglo cambie.

## Simulando una petición a una API

Dentro de una aplicación web, al trabajar con APIs, existen muchos factores para determinar cuánto tardará en cargar nuestra aplicación, como la velocidad de nuestro internet, la distancia del servidor, etc.

Al trabajar con APIs también debemos tener en cuenta que puede tardar en cargar mucho nuestra aplicación, o incluso puede ocurrir algún error, todo esto lo debemos de manejar para mantener a nuestro usuario informado.

El hook de efecto nos permite saber cuando ya renderizó nuestra aplicación, así podemos mostrar un mensaje de cargando o alguna animación en lo que se completa la petición, también con JavaScript podemos manejar los errores con try y catch, y haciendo uso del hook de estado podemos guardar si está cargando o hubo algún error.

### App/index.js

**import** React **from** 'react';

**import** { AppUI } **from** './AppUI';

**function** **useLocalStorage**(itemName, initialValue) {

// Creamos el estado inicial para nuestros errores y carga

**const** [error, setError] = React.useState(**false**);

**const** [loading, setLoading] = React.useState(**true**);

**const** [item, setItem] = React.useState(initialValue);

React.useEffect(() => {

// Simulamos un segundo de delay de carga

setTimeout(() => {

// Manejamos la tarea dentro de un try/catch por si ocurre algún error

**try** {

**const** localStorageItem = localStorage.getItem(itemName);

**let** parsedItem;

**if** (!localStorageItem) {

localStorage.setItem(itemName, JSON.stringify(initialValue));

parsedItem = initialValue;

} **else** {

parsedItem = JSON.parse(localStorageItem);

}

setItem(parsedItem);

} **catch**(error) {

// En caso de un error lo guardamos en el estado

setError(error);

} **finally** {

// También podemos utilizar la última parte del try/cath (finally) para terminar la carga

setLoading(**false**);

}

}, 1000);

});

**const** saveItem = (newItem) => {

// Manejamos la tarea dentro de un try/catch por si ocurre algún error

**try** {

**const** stringifiedItem = JSON.stringify(newItem);

localStorage.setItem(itemName, stringifiedItem);

setItem(newItem);

} **catch**(error) {

// En caso de algún error lo guardamos en el estado

setError(error);

}

};

// Para tener un mejor control de los datos retornados, podemos regresarlos dentro de un objeto

**return** {

item,

saveItem,

loading,

error,

};

}

**function** **App**() {

// Desestructuramos los nuevos datos de nustro custom hook

**const** {

item: todos,

saveItem: saveTodos,

loading,

error,

} = useLocalStorage('TODOS\_V1', []);

**const** [searchValue, setSearchValue] = React.useState('');

**const** completedTodos = todos.filter(todo => !!todo.completed).length;

**const** totalTodos = todos.length;

**let** searchedTodos = [];

**if** (!searchValue.length >= 1) {

searchedTodos = todos;

} **else** {

searchedTodos = todos.filter(todo => {

**const** todoText = todo.text.toLowerCase();

**const** searchText = searchValue.toLowerCase();

**return** todoText.includes(searchText);

});

}

**const** completeTodo = (text) => {

**const** todoIndex = todos.findIndex(todo => todo.text === text);

**const** newTodos = [...todos];

newTodos[todoIndex].completed = **true**;

saveTodos(newTodos);

};

**const** deleteTodo = (text) => {

**const** todoIndex = todos.findIndex(todo => todo.text === text);

**const** newTodos = [...todos];

newTodos.splice(todoIndex, 1);

saveTodos(newTodos);

};

**return** (

{/\* Pasamos los valores de loading y error \*/}

<AppUI

loading={loading}

error={error}

totalTodos={totalTodos}

completedTodos={completedTodos}

searchValue={searchValue}

setSearchValue={setSearchValue}

searchedTodos={searchedTodos}

completeTodo={completeTodo}

deleteTodo={deleteTodo}

/>

);

}

**export** **default** App;

Una vez sabemos exactamente cuándo una aplicación está cargando y cuándo ha ocurrido algún error, podemos usar esta información para mostrar algo al usuario.

### App/AppUI.js

**import** React **from** 'react';

**import** { TodoCounter } **from** '../TodoCounter';

**import** { TodoSearch } **from** '../TodoSearch';

**import** { TodoList } **from** '../TodoList';

**import** { TodoItem } **from** '../TodoItem';

**import** { CreateTodoButton } **from** '../CreateTodoButton';

// Desescructuramos las nuesvas props

**function** **AppUI**({

loading,

error,

totalTodos,

completedTodos,

searchValue,

setSearchValue,

searchedTodos,

completeTodo,

deleteTodo,

}) {

**return** (

<React.Fragment>

<TodoCounter

total={totalTodos}

completed={completedTodos}

/>

<TodoSearch

searchValue={searchValue}

setSearchValue={setSearchValue}

/>

<TodoList>

// Mostramos un mensaje en caso de que ocurra algún error

{error && <p>Desespérate, hubo un error...</p>}

// Mostramos un mensaje de cargando, cuando la aplicación está cargando lo sdatos

{loading && <p>Estamos cargando, no desesperes...</p>}

// Si terminó de cargar y no existen TODOs, se muestra un mensaje para crear el primer TODO

{(!loading && !searchedTodos.length) && <p>¡Crea tu primer TODO!</p>}

{searchedTodos.map(todo => (

<TodoItem

key={todo.text}

text={todo.text}

completed={todo.completed}

onComplete={() => completeTodo(todo.text)}

onDelete={() => deleteTodo(todo.text)}

/>

))}

</TodoList>

<CreateTodoButton />

</React.Fragment>

);

}

export { AppUI };

# clase video 15, React Context: estado compartido

Es una forma de **tener acceso a datos** a través del árbol de componentes **sin tener que pasar props manualmente** en cada nivel, usando datos globales.

Para esto tenemos un proveedor que envolverá a todos los componentes que serán los consumidores de nuestro contexto.

**Fases:**

1. **Crear** el contexto de nuestra aplicación
2. **Proveer** nuestro contexto con los datos que queremos globales.
3. **Consumir** los datos desde cualquier parte de nuestra aplicación.

## Creando el contexto de nuestra aplicación

Primero necesitamos crear nuestra carpeta TodoContext, aunque también en muchos proyectos crean una carpeta context aparte, en donde tendrán todo el contexto de su aplicación.

Es importante crear el contexto con createContext, ya que este nos regresará dos componentes: proveedor y consumidor.

### TodoContext/index.js

**import** React **from** 'react';

**import** { useLocalStorage } **from** './useLocalStorage';

// Al crear el contexto también podemos pasarle un valor inicial entre los paréntesis

**const** TodoContext = React.createContext();

**function** **TodoProvider**(props) {

// Nos traemos todo el estado y las funciones de nuestra aplicación que queremos globales

**const** {

item: todos,

saveItem: saveTodos,

loading,

error,

} = useLocalStorage('TODOS\_V1', []);

**const** [searchValue, setSearchValue] = React.useState('');

**const** completedTodos = todos.filter(todo => !!todo.completed).length;

**const** totalTodos = todos.length;

**let** searchedTodos = [];

**if** (!searchValue.length >= 1) {

searchedTodos = todos;

} **else** {

searchedTodos = todos.filter(todo => {

**const** todoText = todo.text.toLowerCase();

**const** searchText = searchValue.toLowerCase();

**return** todoText.includes(searchText);

});

}

**const** completeTodo = (text) => {

**const** todoIndex = todos.findIndex(todo => todo.text === text);

**const** newTodos = [...todos];

newTodos[todoIndex].completed = **true**;

saveTodos(newTodos);

};

**const** deleteTodo = (text) => {

**const** todoIndex = todos.findIndex(todo => todo.text === text);

**const** newTodos = [...todos];

newTodos.splice(todoIndex, 1);

saveTodos(newTodos);

};

// Retornamos nuestro proveedor con nuestro contexto en la etiqueta value, que recibirá a toda nuestra aplicación, por eso necesitamos la prop children

**return** (

<TodoContext.Provider value={{

loading,

error,

totalTodos,

completedTodos,

searchValue,

setSearchValue,

searchedTodos,

completeTodo,

deleteTodo,

}}>

{props.children}

</TodoContext.Provider>

);

}

// Exportamos nuestro proveedor y nuestro contexto, en el context también esta el consumer, para acceder a nuestro contexto

export { TodoContext, TodoProvider };

Los componentes que consumen el contexto deben estar envueltos en la etiqueta proveedora.

### App/index.js

Ahora importamos nuestro proveedor para envolver nuestra aplicación:

**import** React **from** 'react';

**import** { TodoProvider } **from** '../TodoContext';

**import** { AppUI } **from** './AppUI';

**function** **App**() {

**return** (

<TodoProvider>

<AppUI />

</TodoProvider>

);

}

export default App;

## Consumir el contexto desde otro componente

Una vez ya tenemos nuestro proveedor envolviendo toda nuestra aplicación, ya podemos acceder a los datos desde cualquier componente hijo y olvidarnos de pasar props componente por componente.

### AppUI.js

**import** React **from** 'react';

// Importamos nuestro contexto

**import** { TodoContext } **from** '../TodoContext';

**import** { TodoCounter } **from** '../TodoCounter';

**import** { TodoSearch } **from** '../TodoSearch';

**import** { TodoList } **from** '../TodoList';

**import** { TodoItem } **from** '../TodoItem';

**import** { CreateTodoButton } **from** '../CreateTodoButton';

**function** **AppUI**() {

**return** (

<React.Fragment>

<TodoCounter />

<TodoSearch />

{/\* Podemos acceder a nuestro contexto con el consumer \*/}

<TodoContext.Consumer>

{({

error,

loading,

searchedTodos,

completeTodo,

deleteTodo,

}) => (

<TodoList>

{error && <p>Desespérate, hubo un error...</p>}

{loading && <p>Estamos cargando, no desesperes...</p>}

{(!loading && !searchedTodos.length) && <p>¡Crea tu primer TODO!</p>}

{searchedTodos.map(todo => (

<TodoItem

key={todo.text}

text={todo.text}

completed={todo.completed}

onComplete={() => completeTodo(todo.text)}

onDelete={() => deleteTodo(todo.text)}

/>

))}

</TodoList>

)}

</TodoContext.Consumer>

<CreateTodoButton />

</React.Fragment>

);

}

export { AppUI };

Con esto evitas pasar props a todos los componentes. Puedes tener muchos componentes que consuman un solo contexto y también varios contextos.

Si el valor del contexto cambia, todos los componentes suscritos se re-renderizan y actualizarán su estado.

Todavía nos falta utilizar nuestro **consumer** en todos los demás componentes, pero la verdad es que esta no es la mejor forma de consumir nuestro contexto.

# useContext

17/23

**RECURSOS**

**MARCADORES**

El **hook de contexto** nos ayuda a acceder a datos globales de nuestro contexto, desde cualquier componente hijo, sin tener que pasar estos datos por **props** componente por componente.

Tiene la misma funcionalidad que el consumer de nuestro contexto, pero **useContext** también tiene una manera más sencilla de utilizar y una sintaxis mucho más clara.

## Context.Consumer a useContext

**import** React **from** 'react';

// También es importante importar nuestro contexto

**import** { TodoContext } **from** '../TodoContext';

**import** { TodoCounter } **from** '../TodoCounter';

**import** { TodoSearch } **from** '../TodoSearch';

**import** { TodoList } **from** '../TodoList';

**import** { TodoItem } **from** '../TodoItem';

**import** { CreateTodoButton } **from** '../CreateTodoButton';

**function** **AppUI**() {

// Desesctructuramos los valores de nuestro contexto

**const** {

error,

loading,

searchedTodos,

completeTodo,

deleteTodo,

} = React.useContext(TodoContext);

**return** (

<React.Fragment>

<TodoCounter />

<TodoSearch />

<TodoList>

{error && <p>Desespérate, hubo un error...</p>}

{loading && <p>Estamos cargando, no desesperes...</p>}

{(!loading && !searchedTodos.length) && <p>¡Crea tu primer TODO!</p>}

{searchedTodos.map(todo => (

<TodoItem

key={todo.text}

text={todo.text}

completed={todo.completed}

onComplete={() => completeTodo(todo.text)}

onDelete={() => deleteTodo(todo.text)}

/>

))}

</TodoList>

<CreateTodoButton />

</React.Fragment>

);

}

export { AppUI };

Ahora solo nos queda utilizar este hook para acceder a nuestro contexto desde los componentes faltantes.

### TodoCounter.js

**import** React **from** 'react';

**import** { TodoContext } **from** '../TodoContext';

**import** './TodoCounter.css';

**function** **TodoCounter**() {

**const** { totalTodos, completedTodos } = React.useContext(TodoContext);

**return** (

<h2 className="TodoCounter">Has completado {completedTodos} de {totalTodos} TODOs</h2>

);

}

**export** { TodoCounter };

### TodoSearch.js

**import** React **from** 'react';

**import** { TodoContext } **from** '../TodoContext';

**import** './TodoSearch.css';

**function** **TodoSearch**() {

**const** { searchValue, setSearchValue } = React.useContext(TodoContext);

**const** onSearchValueChange = (event) => {

console.log(event.target.value);

setSearchValue(event.target.value);

};

**return** (

<input

className="TodoSearch"

placeholder="Cebolla"

value={searchValue}

onChange={onSearchValueChange}

/>

);

}

export { TodoSearch };

## ¿Cuándo se recomienda emplear React Context?

* Estado global
* Tema
* Configuración de la app
* Autenticación de usuario
* Configuración de usuario
* Lenguaje preferido
* Colección de servicios

# Portales: teletransportación de componentes

Los portales nos permiten **teletransportar componentes** a otro nodo de HTML, y seguir comunicándose con otros componentes como si estuviera en el mismo nodo.

Se emplean en ocasiones donde los estilos CSS restringen los elementos. Por ejemplo, problemas de apilamiento z-index y desbordamiento overflow.

**¿Para qué podemos usarlos?**

* Modales
* Tooltips
* Menús flotantes
* Widgets

// Es el mismo import para las versiones mayores o iguales a ReactV18

**import** ReactDOM **from** "react-dom";

ReactDOM.createPortal(componente, contenedor)

## Creando nuestro modal

Lo primero que necesitamos es crear otro div dentro del body en nuestro HTML.

<div id="modal"></div>

Una vez tenemos nuestro div exclusivo para nuestro modal ya podemos trabajar con React:

### Modal/index.js

Crearemos un componente Modal:

**import** React **from** 'react';

// Necesitamos ReactDOM para renderizar nuestro modal en el DOM

**import** ReactDOM **from** 'react-dom';

**import** './Modal.css';

**function** **Modal**({ children }) {

// ReactDom tiene este método para crear portales

**return** ReactDOM.createPortal(

<div className="ModalBackground">

{children}

</div>,

document.getElementById('modal')

);

}

**export** { Modal };

### Modal.css

También agreguemos un poco de CSS para que se vea como un modal.

.ModalBackground {

background: rgba(32,35,41,.8);

position: fixed;

top: -10px;

left: -10px;

right: -10px;

bottom: -10px;

display: flex;

justify-content: center;

align-items: center;

color: white;

}

## AppUI.js

Lo interesante es que podemos pasarle otros componentes y hasta tener acceso al contexto de nuestra aplicación al importar nuestro modal dentro de la aplicación.

**import** React **from** 'react';

**import** { TodoContext } **from** '../TodoContext';

**import** { TodoCounter } **from** '../TodoCounter';

**import** { TodoSearch } **from** '../TodoSearch';

**import** { TodoList } **from** '../TodoList';

**import** { TodoItem } **from** '../TodoItem';

**import** { CreateTodoButton } **from** '../CreateTodoButton';

**import** { Modal } **from** '../Modal';

**function** **AppUI**() {

**const** {

error,

loading,

searchedTodos,

completeTodo,

deleteTodo,

} = React.useContext(TodoContext);

**return** (

<React.Fragment>

<TodoCounter />

<TodoSearch />

<TodoList>

{error && <p>Desespérate, hubo un error...</p>}

{loading && <p>Estamos cargando, no desesperes...</p>}

{(!loading && !searchedTodos.length) && <p>¡Crea tu primer TODO!</p>}

{searchedTodos.map(todo => (

<TodoItem

key={todo.text}

text={todo.text}

completed={todo.completed}

onComplete={() => completeTodo(todo.text)}

onDelete={() => deleteTodo(todo.text)}

/>

))}

</TodoList>

<Modal>

<p>{searchedTodos[0]?.text}</p>

</Modal>

<CreateTodoButton />

</React.Fragment>

);

}

export { AppUI };

Si hacemos esto, ya tendremos nuestro modal en nuestra aplicación, pero hay un pequeño detalle, el modal aparece sobre toda nuestra aplicación, necesitamos crear un estado para decirle si estará abierto o cerrado, y también para poder cerrarlo cuando sea necesario.

Dentro de nuestro contexto crearemos un estado extra que será para nuestro modal:

### TodoContext/index.js

**const** [openModal, setOpenModal] = React.useState(**false**);

...

// Y lo exportamos

**return** (

<TodoContext.Provider value={{

....

openModal

setOpenModal,

}}>

{props.children}

</TodoContext.Provider>

);

}

### Cambios en AppUI.js

¡Ahora podemos abrir y cerrar nuestro modal desde cualquier parte de nuestra aplicación!  
Por ejemplo, en nuestro componente CreateTodoButton, primero necesitamos crear la lógica para mostrar el Modal solo cuando openModal, sea true en nuestro componente.

**import** React **from** 'react';

**import** { TodoContext } **from** '../TodoContext';

**import** { TodoCounter } **from** '../TodoCounter';

**import** { TodoSearch } **from** '../TodoSearch';

**import** { TodoList } **from** '../TodoList';

**import** { TodoItem } **from** '../TodoItem';

**import** { CreateTodoButton } **from** '../CreateTodoButton';

**import** { Modal } **from** '../Modal';

**function** **AppUI**() {

**const** {

error,

loading,

searchedTodos,

completeTodo,

deleteTodo,

openModal,

setOpenModal,

} = React.useContext(TodoContext);

**return** (

<React.Fragment>

<TodoCounter />

<TodoSearch />

<TodoList>

{error && <p>Desespérate, hubo un error...</p>}

{loading && <p>Estamos cargando, no desesperes...</p>}

{(!loading && !searchedTodos.length) && <p>¡Crea tu primer TODO!</p>}

{searchedTodos.map(todo => (

<TodoItem

key={todo.text}

text={todo.text}

completed={todo.completed}

onComplete={() => completeTodo(todo.text)}

onDelete={() => deleteTodo(todo.text)}

/>

))}

</TodoList>

{!!openModal && (

<Modal>

<p>{searchedTodos[0]?.text}</p>

</Modal>

)}

<CreateTodoButton

setOpenModal={setOpenModal}

/>

</React.Fragment>

);

}

export { AppUI };

Y ahora que pasamos nuestro contexto por props, ya podemos cambiar este estado para abrir nuestro modal.

**import** React **from** 'react';

**import** './CreateTodoButton.css';

**function** **CreateTodoButton**(props) {

**const** onClickButton = () => {

props.setOpenModal(**true**);

};

**return** (

<button

className="CreateTodoButton"

onClick={onClickButton}

>

+

</button>

);

}

**export** { CreateTodoButton };

Pero, antes de avanzar, todavía hay un pequeño problema, no podemos cerrar nuestro modal, esto queda como reto para ti, deja correr tu creatividad.

Puedes crear un botón de cerrar nuestro modal, o que al hacer clic en el background del modal se cierre, también puedes hacer un toggle para el botón de crear TODO, o hasta añadir un botón de cancelar en el modal para cerrarlo. ¡Esperamos ver tu resultado en los comentarios!

**clase video 19; Formulario para crear TODOs**

Algo muy importante al crear formularios es tener en cuenta que React funciona un poco diferente al HTML, ya que en HTML conservan naturalmente algún estado interno.

En React nosotros podemos mutar el estado de nuestros componentes con el hook de estado, un **componente controlado** es simplemente un componente en el que sus valores son controlados por React.

## Creando el formulario para crear un nuevo TODO

En nuestro archivo AppUi.js importaremos y añadiremos un componente TodoForm:

**import** React **from** "react";

**import** { TodoContext } **from** "../TodoContext";

**import** { TodoCounter } **from** "../TodoCounter";

**import** { TodoSearch } **from** "../TodoSearch";

**import** { TodoList } **from** "../TodoList";

**import** { TodoItem } **from** "../TodoItem";

**import** { TodoForm } **from** "../TodoForm";

**import** { CreateTodoButton } **from** "../CreateTodoButton";

**import** { Modal } **from** "../Modal";

**function** **AppUI**() {

**const** { error, loading, searchedTodos, completeTodo, deleteTodo, openModal, setOpenModal } =

React.useContext(TodoContext);

**return** (

<React.Fragment>

<TodoCounter />

<TodoSearch />

<TodoList>

{error && <p>Desespérate, hubo un error...</p>}

{loading && <p>Estamos cargando, no desesperes...</p>}

{!loading && !searchedTodos.length && <p>¡Crea tu primer TODO!</p>}

{searchedTodos.map((todo) => (

<TodoItem

key={todo.text}

text={todo.text}

completed={todo.completed}

onComplete={() => completeTodo(todo.text)}

onDelete={() => deleteTodo(todo.text)}

/>

))}

</TodoList>

{!!openModal && (

<Modal>

<TodoForm />

</Modal>

)}

<CreateTodoButton setOpenModal={setOpenModal} />

</React.Fragment>

);

}

export { AppUI };

En un momento crearemos este componente, primero necesitamos añadir una función para añadir nuestro nuevo TODO, dentro de nuestro contexto para utilizarla en nuestro formulario.

### TodoContext

**import** React **from** 'react';

**import** { useLocalStorage } **from** './useLocalStorage';

**const** TodoContext = React.createContext();

**function** **TodoProvider**(props) {

**const** {

item: todos,

saveItem: saveTodos,

loading,

error,

} = useLocalStorage('TODOS\_V1', []);

**const** [searchValue, setSearchValue] = React.useState('');

**const** [openModal, setOpenModal] = React.useState(**false**);

**const** completedTodos = todos.filter(todo => !!todo.completed).length;

**const** totalTodos = todos.length;

**let** searchedTodos = [];

**if** (!searchValue.length >= 1) {

searchedTodos = todos;

} **else** {

searchedTodos = todos.filter(todo => {

**const** todoText = todo.text.toLowerCase();

**const** searchText = searchValue.toLowerCase();

**return** todoText.includes(searchText);

});

}

// Función para añadir un nuevo TODO

**const** addTodo = (text) => {

**const** newTodos = [...todos];

newTodos.push({

completed: **false**,

text,

});

saveTodos(newTodos);

};

**const** completeTodo = (text) => {

**const** todoIndex = todos.findIndex(todo => todo.text === text);

**const** newTodos = [...todos];

newTodos[todoIndex].completed = **true**;

saveTodos(newTodos);

};

**const** deleteTodo = (text) => {

**const** todoIndex = todos.findIndex(todo => todo.text === text);

**const** newTodos = [...todos];

newTodos.splice(todoIndex, 1);

saveTodos(newTodos);

};

**return** (

<TodoContext.Provider value={{

loading,

error,

totalTodos,

completedTodos,

searchValue,

setSearchValue,

searchedTodos,

addTodo,

completeTodo,

deleteTodo,

openModal,

setOpenModal,

}}>

{props.children}

</TodoContext.Provider>

);

}

export { TodoContext, TodoProvider };

### TodoForm.js

Ahora que ya tenemos prácticamente todo, solo queda utilizar la función addTodo para añadir nuestro TODO desde nuestro modal.

**import** React **from** 'react';

**import** { TodoContext } **from** '../TodoContext';

**import** './TodoForm.css';

**function** **TodoForm**() {

// Creamos un estado para nuestro nuevo TODO

**const** [newTodoValue, setNewTodoValue] = React.useState('');

// Desestructuramos las funciones que necesitamos para añadir un TODO y cerrar nuestro modal

**const** {

addTodo,

setOpenModal,

} = React.useContext(TodoContext);

// Creamos una función para actualizar el estado de nuestro nuevo TODO

**const** onChange = (event) => {

setNewTodoValue(event.target.value);

};

// Función para cerrar el modal

**const** onCancel = () => {

setOpenModal(**false**);

};

// Función para agregar nuestro nuevo TODO

**const** onSubmit = (event) => {

// prevent default para evitar recargar la página

event.preventDefault();

// Utilizamos nuestra función para añadir nuestro TODO

addTodo(newTodoValue);

// Cerramos nustro modal

setOpenModal(**false**);

// También estaría bien resetear nuestro formulario

setNewTodoValue('')

};

**return** (

<form onSubmit={onSubmit}>

<label>Escribe tu nuevo TODO</label>

<textarea

value={newTodoValue}

onChange={onChange}

placeholder="Cortar la cebolla para el almuerzo"

/>

<div className="TodoForm-buttonContainer">

<button

type="button"

className="TodoForm-button TodoForm-button--cancel"

onClick={onCancel}

>

Cancelar

</button>

<button

type="submit"

className="TodoForm-button TodoForm-button--add"

>

Añadir

</button>

</div>

</form>

);

}

export { TodoForm };

También le podemos añadir unos estilos a nuestro formulario:

### TodoForm.css

**form** {

width: 90%;

max-width: 300px;

background-color: #fff;

padding: 33px 40px;

display: flex;

justify-content: center;

align-items: center;

flex-direction: column;

}

**label** {

text-align: center;

font-weight: bold;

font-size: 20px;

color: #1E1E1F;

margin-bottom: 26px;

}

**textarea** {

background-color: #F9FBFC;

border: 2px solid #202329;

border-radius: 2px;

box-shadow: 0px 5px 50px rgba(32, 35, 41, 0.25);

color: #1E1E1F;

font-size: 20px;

text-align: center;

padding: 12px;

height: 96px;

width: calc(100% - 25px);

}

**textarea**::placeholder {

color: #A5A5A5;

font-family: 'Montserrat';

font-weight: 400;

}

**textarea**:focus {

outline-color: #61DAFA;

}

.TodoForm-buttonContainer {

margin-top: 14px;

display: flex;

justify-content: space-between;

align-items: center;

width: 100%;

}

.TodoForm-button {

cursor: pointer;

display: inline-block;

font-size: 20px;

color: #202329;

font-weight: 400;

width: 120px;

height: 48px;

border-radius: 2px;

border: none;

font-family: 'Montserrat';

}

.TodoForm-button--add {

background: #61DAFA;

box-shadow: 0px 5px 25px rgba(97, 218, 250, 0.5);

}

.TodoForm-button--cancel {

background: transparent;

}

Si tienes un estilo parecido al resultado final, te animamos a que le des tu personalidad a tu aplicación, no solamente de estilos, también agrega nuevas funcionalidades, rompe el código, mejóralo, la práctica es algo de lo más importante para aprender.  
¡Esperamos ver tus resultados!

# Deploy con GitHub Pages

SUBIR A UN SERVIDO DE INTERNET

Existen varias maneras para hacer deploy de nuestras aplicaciones, una de las más populares es **GitHub Pages**.

GitHub Pages es un servicio de GitHub que nos permite **alojar nuestros proyectos** y mostrarlos en vivo en una página web estática sin necesidad de pagar por hosting o tener conocimientos en servidores, aunque tiene algunas limitantes.

Para inicializar nuestro proyecto con **Create React App**, utilizamos el comando npm start, lo que hace este comando, es que iniciar un servidor de node.js para poder compilar nuestro código de React a JavaScript, también para poder estar al pendiente por cambios y refrescar automáticamente nuestro proyecto al hacer algún cambio, entre muchas otras cosas.

La principal desventaja de **GitHub Pages**, es que no podemos correr un servidor de node.js, solamente soporta archivos estáticos:

* HTML
* CSS
* JavaScript
* Imágenes

Para hacer el deploy de nuestra aplicación, necesitaremos hacer lo siguiente:

1. Instalar la dependencia gh-pages con npm.
2. Modificar un poco el package.json
3. Y utilizaremos otros comandos de Create React App para construir una versión de producción de nuestro proyecto.

## Instalar gh-pages

Para instalar esta dependencia en nuestro proyecto, necesitamos ejecutar este comando en la terminal, dentro de la **carpeta root** en la que está nuestro proyecto: npm install --save-dev gh-pages o npm i -D gh-pages que es lo mismo.

⚠️ Si aún no tienes tu proyecto en un repositorio de GitHub, es el momento perfecto para subirlo, con todos tus cambios.

## Modificar nuestro package.json

Primero necesitamos agregar la regla como la siguiente:

"homepage":"https://tu-usuario-de-github.github.io/nombre-del-repositorio"

Para que funcione necesitas poner los datos necesarios en la URL.  
Y ahora agregamos unos scrips para de gh-pages el deploy, estos scripts prepararán nuestra aplicación para el deploy.  
Dentro de npm podemos utilizar los prefijos: pre y post para decir que ejecute cualquier cosa que queramos antes o después de ejecutar el script que necesitamos.

Nuestro package.json quedaría como el siguiente:

{

"homepage":"https://brandonargel.github.io/todo-app"

"name": "curso-intro-react",

"version": "0.1.0",

"private": **true**,

"dependencies": {

"@testing-library/jest-dom": "^5.11.4",

"@testing-library/react": "^11.1.0",

"@testing-library/user-event": "^12.1.10",

"react": "^17.0.2",

"react-dom": "^17.0.2",

"react-scripts": "4.0.3",

"web-vitals": "^1.0.1"

},

"scripts": {

"start": "react-scripts start",

"build": "react-scripts build",

"test": "react-scripts test",

"eject": "react-scripts eject",

"predeploy": "npm run build",

"deploy": "gh-pages -d build"

},

"eslintConfig": {

"extends": [

"react-app",

"react-app/jest"

]

},

"browserslist": {

"production": [

">0.2%",

"not dead",

"not op\_mini all"

],

"development": [

"last 1 chrome version",

"last 1 firefox version",

"last 1 safari version"

]

},

"devDependencies": {

"gh-pages": "^3.2.3"

},

}

## Deploy de nuestra aplicación

Ahora solo necesitamos ejecutar nuestros scripts: npm run deploy.

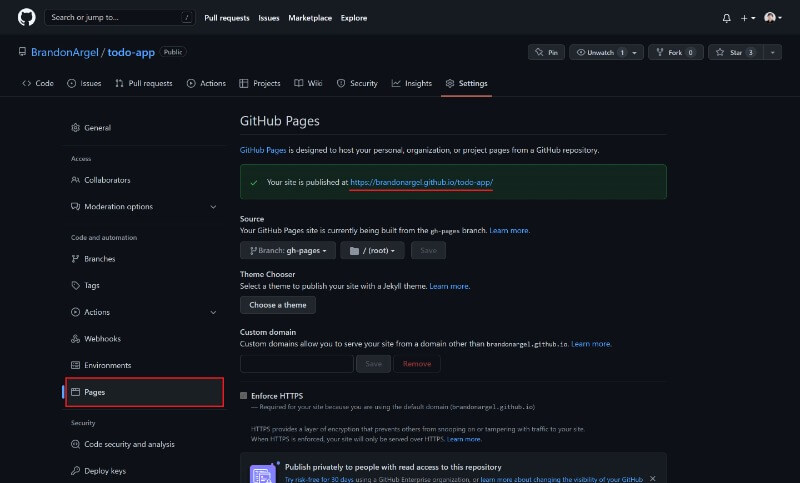
¡Al hacer esto, si no nos aparece ningún error, quiere decir nuestra página ya está en vivo!

Si te aparece algún error, no dudes en dejar tu pregunta para que podamos ayudarte, también puedes revisar los pasos anteriores o revisar esta [Guía para subir tus proyectos de React con gh-pages](https://platzi.com/tutoriales/1548-react/4065-guia-para-usar-github-pages-en-tus-proyectos-de-reactjs/){target="\_blank"}.

## ¿Cómo puedo ver mi aplicación en la web?

Para poder ver tu aplicación ya en la web puedes entrar a la URL que escribiste en el homepage de tu package.json,

Otra forma es entrar al repositorio de tu proyecto en **GitHub -> Ajustes**, y en el menú de la izquierda, en **Code and automation** te aparecerá una sección de **Pages**, ahí estará un link para ver tu proyecto.



Contribución creada por: Brandon Argel.

# Toma el Curso de React.js: Patrones de Render y Composición

¡Felicitaciones! 🥳  
Estás a punto de terminar el [Curso de Introducción a React.js](https://platzi.com/cursos/react/), a lo largo de este curso construimos una pequeña aplicación, pero muy completa y también aprendiste las **funcionalidades fundamentales** de React.js, como:

* Cómo crear componentes
* Manejar eventos
* Manejar el estado
* Manejar efectos
* Comunicar componentes con props
* Comunicar componentes con el contexto de React
* Crear custom hooks
* Persistir nuestro estado con localStorage
* Simular una petición a una API, para así manejar el estado de carga y también manejar errores en su caso
* Crear portales para teletransportar componentes

Pero espera, este es solamente el inicio, aunque aprendiste muchísimas cosas, también queda demasiado por aprender, el siguiente curso es el [Curso de React.js: Patrones de Render y Composición](https://platzi.com/cursos/react-patrones-render/), es la continuación para mejorar las técnicas que utilizamos en nuestra aplicación, en ese curso aprenderás muchas más herramientas de React, así como técnicas para evitar re-renders innecesarios, ¡Entre otras muchas cosas!

Contribución creada por: Brandon Argel.

#### **Lecturas recomendadas**

[Curso de React.js: Patrones de Render - Platzi](https://platzi.com/clases/react-patrones-render/" \t "_blank)

[https://platzi.com/clases/react-patrones-render/](https://platzi.com/clases/react-patrones-render/" \t "_blank)

COMPILAR

Prueba la nueva tecnología PowerShell multiplataforma https://aka.ms/pscore6

PS C:\Users\E490\Documents\proyectos\int\_react> npm install --save-dev gh-pages

npm WARN config global `--global`, `--local` are deprecated. Use `--location=global` instead.

added 16 packages, and audited 1463 packages in 10s

206 packages are looking for funding

run `npm fund` for details

6 high severity vulnerabilities

To address all issues (including breaking changes), run:

Run `npm audit` for details.

PS C:\Users\E490\Documents\proyectos\int\_react> npm run deploy

npm WARN config global `--global`, `--local` are deprecated. Use `--location=global` instead.

npm ERR! code EJSONPARSE

npm ERR! path C:\Users\E490\Documents\proyectos\int\_react/package.json

npm ERR! JSON.parse Unexpected string in JSON at position 491 while parsing '{

npm ERR! JSON.parse "name": "int\_react",

npm ERR! JSON.parse "version": "0'

npm ERR! JSON.parse Failed to parse JSON data.

npm ERR! JSON.parse Note: package.json must be actual JSON, not just JavaScript.

npm ERR! A complete log of this run can be found in:

npm ERR! C:\Users\E490\AppData\Local\npm-cache\\_logs\2022-08-25T17\_49\_45\_225Z-debug-0.log

PS C:\Users\E490\Documents\proyectos\int\_react> npm run deploy

npm WARN config global `--global`, `--local` are deprecated. Use `--location=global` instead.

> curso-intro-react@0.1.0 predeploy

> npm run build

npm WARN config global `--global`, `--local` are deprecated. Use `--location=global` instead.

> curso-intro-react@0.1.0 build

> react-scripts build

Creating an optimized production build...

Compiled successfully.

File sizes after gzip:

48.31 kB build\static\js\main.8f1554b2.js

1.22 kB build\static\css\main.3a4635a9.css

The project was built assuming it is hosted at /Intro\_React/.

You can control this with the homepage field in your package.json.

The build folder is ready to be deployed.

Find out more about deployment here:

https://cra.link/deployment

> curso-intro-react@0.1.0 deploy

> gh-pages -d build

Published

PS C:\Users\E490\Documents\proyectos\int\_react> git add -A

The file will have its original line endings in your working directory

warning: LF will be replaced by CRLF in package-lock.json.

The file will have its original line endings in your working directory

warning: LF will be replaced by CRLF in package.json.

The file will have its original line endings in your working directory

PS C:\Users\E490\Documents\proyectos\int\_react> git commit -m "deploy"

[master 161caa7] deploy

3 files changed, 346 insertions(+), 10 deletions(-)

PS C:\Users\E490\Documents\proyectos\int\_react> git push

fatal: The current branch master has no upstream branch.

To push the current branch and set the remote as upstream, use

git push --set-upstream origin master

PS C:\Users\E490\Documents\proyectos\int\_react> git push origin master

Enumerating objects: 9, done.

Counting objects: 100% (9/9), done.

Delta compression using up to 8 threads

Compressing objects: 100% (5/5), done.

Writing objects: 100% (5/5), 3.21 KiB | 193.00 KiB/s, done.

Total 5 (delta 4), reused 0 (delta 0), pack-reused 0

remote: Resolving deltas: 100% (4/4), completed with 4 local objects.

To https://github.com/elifred2022/Intro\_React.git

78ba671..161caa7 master -> master

PS C:\Users\E490\Documents\proyectos\int\_react>

\* Historial restaurado

Windows PowerShell

Copyright (C) Microsoft Corporation. Todos los derechos reservados.

Prueba la nueva tecnología PowerShell multiplataforma https://aka.ms/pscore6

PS C:\Users\E490\Documents\proyectos\int\_react>

