**Cómo aprender React.js (clase 1)**

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/)

<https://es.reactjs.org/>

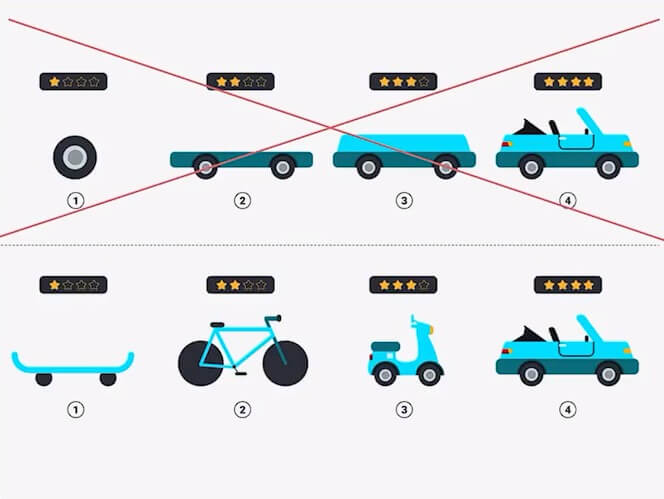
# Cuándo usar React.js (clase 2)

Para saber cuándo usar React es necesario **conocer el tipo de proyecto** que estás desarrollando. Conociendo los alcances actuales y los futuros, podrás tener un panorama completo de las necesidades que vas a cubrir y cuáles son las funciones básicas que debe tener para ser un **MVP**.

React es muy útil cuando queremos construir **rápidamente** la primera versión funcional de una aplicación web sin sacrificar su **escalabilidad**. Si no necesitamos escalarla después, no es necesario utilizar React, podemos utilizar JavaScript vanilla.

## ¿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.  


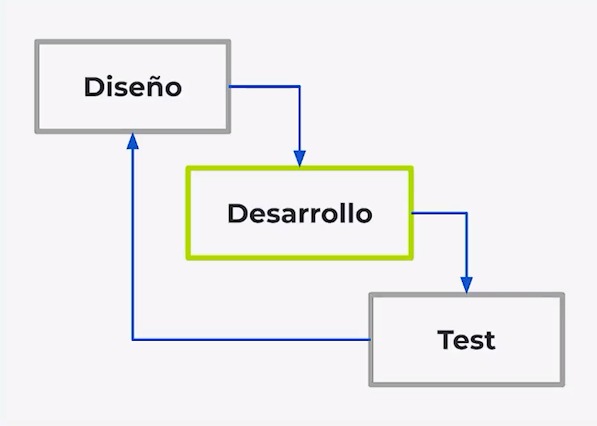
## Flujos de trabajo para aplicaciones web

Existen dos flujos de trabajo para crear aplicaciones web profesionalmente.

### 1. Modelo en cascada

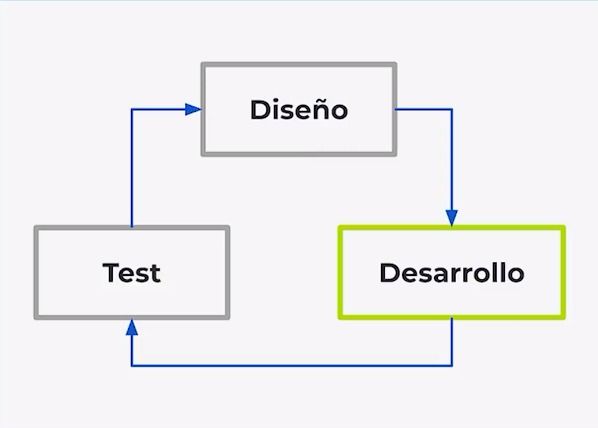
Cada equipo tiene un tiempo para hacer todo su respectivo trabajo para construir una aplicación.

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.  


## Análisis: componentes y comportamientos

**Componentes**: son la forma de estructurar las piezas de nuestra página web para hacerlas escalables, nos ahorran tiempo y esfuerzo. Son abstracciones de los elementos de nuestra página web para ser reusados las veces que necesitemos.

Los componentes pueden trabajar de manera independiente a los demás, pero con React todos los componentes tienen una conexión con el resto de componentes, para que en conjunto puedan reaccionar al comportamiento de los usuarios.

**Comportamiento:** cómo actúan nuestros componentes con las interacciones de los usuarios.

**Cambios en React 18: ReactDOM.createRoot (clase 3)**

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 (clase 4)

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 metadatos de nuestro proyecto, también tendremos dos carpetas principales:

**public/:** Es la carpeta pública en donde tendremos nuestro archivo HTML y algunos assets

**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:

<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).

# JSX: componentes vs. elementos (y props vs. atributos) (clase 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" />

const Componente = (props) => {

// Recibimos las propiedades

return(

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

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

# {props.saludo}

# {/\* props.saludo = 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>

)

}

<https://es.reactjs.org/blog/2020/09/22/introducing-the-new-jsx-transform.html>

# Componentes de TODO Machine (clase 6)

Ya que conocemos [qué es **JSX**](https://platzi.com/clases/2444-react/40304-jsx-componentes-vs-elementos-y-props-vs-atributos/) y cómo funciona, cómo pasar **propiedades** a los componentes, podemos comenzar con la creación de nuestra aplicación.

## 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** (

);

}

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 una 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 (

...

);

}

// Desestructurándolo desde React

import { Fragment } from "react

function App() {

return (

...

);

}

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 completado 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(props){

    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 { CreateTodoButton } from "./CreateTodoButton.js"

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

**return** (

);

}

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.

**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** (

{todos.map(todo => (

))}

);

}

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**}

];

  <TodoList>

        {todos.map((item, index)=>(

        <TodoItem key={index} {...item}/>

        ))}

   </TodoList>

# CSS en React (clase 7)

Existen varias [formas de agregar estilos en React](https://platzi.com/blog/react-css/):

* Hoja de estilos externa: podemos crear nuestro archivo CSS con todos nuestros estilos y después importarlo.
* Estilos en línea: así como en HTML, también podemos agregar estilos en línea, pero con una diferente sintaxis

## Estilos en línea

Para estos estilos necesitamos agregarle las llaves principales a la propiedad {}, para poder pasarle un objeto de JavaScript con nuestros estilos, también cambia un poco la sintaxis de **CSS**, aquí se utiliza el **camelCase** para el nombre de las propiedades de CSS, y para los valores necesitamos escribirlos entre comillas.

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

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

**return**(

<h2 style={{

color: 'red',

backgroundColor: 'yellow'

}}

>

Has complentado 2 de 3 ToDos

h2>

)

}

**export** {TodoCounter};

Los estilos en línea en React son mala práctica por las mismas razones que lo son en HTML.

## Hoja de estilos externa

También podemos importar un archivo CSS directamente en React:

.TodoCounter {

font-size: 24px;

text-align: center;

}

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

**import** './todoCounter.css'

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

**return**(

<h2 className="TodoCounter">

Has complentado 2 de 3 ToDos

h2>

)

}

**export** {TodoCounter};

## Reto

Te animamos a que crees tu propio diseño, que le agregues tu identidad y tus propios estilos.  
¡Esperamos ver tu aplicación pronto en los comentarios!

## Estilos y componentes

Ahora crearemos los estilos de cada uno de nuestros componentes, fíjate bien en la lógica de nuestro TodoItem, estaremos utilizando JavaScript para alternar nuestros estilos, dependiendo si el TODO está completado o no.

### TodoCounter

.TodoCounter {

font-size: 24px;

text-align: center;

margin: 0;

padding: 48px;

}

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

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

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

**return** (

<h2 className="TodoCounter">Has completado 2 de 3 TODOsh2>

);

}

**export** { TodoCounter };

### TodoSearch

.TodoSearch {

background: #F9FBFC;

border-radius: 2px;

border: 2px solid #202329;

margin: 0 24px;

height: 64px;

width: calc(100% - 62px);

font-size: 24px;

text-align: center;

font-family: 'Montserrat';

font-weight: 400;

color: #1E1E1F;

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

}

.TodoSearch::placeholder {

color: #A5A5A5;

font-family: 'Montserrat';

font-weight: 400;

}

.TodoSearch:focus {

outline-color: #61DAFA;

}

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

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

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

**return** (

<input className="TodoSearch" placeholder="Cebolla" />

);

}

export { TodoSearch };

### TodoList

**ul** {

margin: 0;

padding: 0 0 56px 0;

list-style: none;

}

import React from 'react';

import './TodoList.css';

function TodoList(props) {

  return (

    <section>

      <ul>

        {props.children}

      </ul>

    </section>

  );

}

export { TodoList };

### TodoItem

.TodoItem {

background-color: #FAFAFA;

position: relative;

display: flex;

justify-content: center;

align-items: center;

margin-top: 24px;

box-shadow: 0px 5px 50px rgba(32, 35, 41, 0.15);

}

.TodoItem-p {

margin: 24px 0 24px 24px;

width: calc(100% - 120px);

font-size: 18px;

line-height: 24px;

font-weight: 400;

}

.TodoItem-p--complete {

text-decoration: line-through;

}

.Icon {

cursor: pointer;

display: flex;

justify-content: center;

align-items: center;

height: 48px;

width: 48px;

font-size: 24px;

font-weight: bold;

/\* background-color: #CCC; \*/

}

.Icon-check {

position: absolute;

left: 12px;

}

.Icon-check--active {

color: #4caf50;

}

.Icon-delete {

position: absolute;

top: -24px;

right: 0;

}

.Icon-delete:hover {

color: red;

}

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

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

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

**return** (

<li className="TodoItem">

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

√

span>

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

{props.text}

p>

<span className="Icon Icon-delete">

X

span>

li>

);

}

**export** { TodoItem };

### CreateTodoButton

.CreateTodoButton {

background-color: #61DAFA;

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

border: none;

border-radius: 50%;

cursor: pointer;

font-size: 50px;

position: fixed;

bottom: 24px;

right: 24px;

font-weight: bold;

color: #FAFAFA;

display: flex;

justify-content: center;

align-items: center;

height: 64px;

width: 64px;

transform: rotate(0);

transition: .3s ease;

}

.CreateTodoButton:hover {

transform: rotate(224deg);

}

import React from 'react';

import './CreateTodoButton.css';

function CreateTodoButton(props) {

  return (

    <button className="CreateTodoButton"/>

  );

}

export { CreateTodoButton };

## Agregar fuentes

Dentro de la carpeta public/ tenemos nuestro archivo HTML, en donde podemos poner un  con alguna fuente de google fonts.

<html lang="en">

<head>

<meta charset="utf-8" />

<link rel="icon" href="%PUBLIC\_URL%/favicon.ico" />

<meta name="viewport" content="width=device-width, initial-scale=1" />

<meta name="theme-color" content="#000000" />

<meta

name="description"

content="Web site created using create-react-app"

/>

<link rel="apple-touch-icon" href="%PUBLIC\_URL%/logo192.png" />

<link rel="manifest" href="%PUBLIC\_URL%/manifest.json" />

<link href="https://fonts.googleapis.com/css2?family=Montserrat:wght@400;700&display=swap" rel="stylesheet" />

<title>React Apptitle>

head>

<body>

<noscript>You need to enable JavaScript to run this app.noscript>

<div id="root">div>

body>

html>

Una vez importamos nuestra fuente ya la podemos llamar en nuestros estilos.

Index.css

**body** {

margin: 0;

padding: 0;

background: #F2F4F5;

color: #1E1E1F;

font-family: 'Montserrat', Arial, Helvetica, sans-serif;

}

**#root** {

margin: 0 24px;

position: relative;

min-height: 100vh;

}

Ahora ya podemos importar nuestros estilos del index.css, al ser estilos globales es mejor importarlos en el index.js de nuestra aplicación.

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

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

**import** App **from** './App';

**import** './index.css';

ReactDOM.render(

<App />,

document.getElementById('root')

);

## Props para TodoItem

Ya que creamos los estilos para los TODOs completados y no completados, y también en el componente creamos la lógica para alternar los estilos dependiendo si está completado o no, solo nos hace falta pasarle la propiedad completed desde nuestro App.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';

// import './App.css';

**const** todos = [

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

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

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

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

];

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

**return** (

{todos.map(todo => (

))}

);

}

export default App;

## Cambio dinámico de clases

Una de las grandes ventajas de React, es que podemos utilizar los lógicos && y || para jugar con nuestras clases muy fácilmente, y dar estilos dependiendo de la clase que esté activa, esto lo aplicamos cuando está completado o no nuestro TODO.

**Manejo de eventos (clase 8)**

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}

/>

);

}

<https://legacy.reactjs.org/docs/handling-events.html>

**Manejo del estado (clase 9)**

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

Para manejar el estado depende de 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.

# Contando y buscando TODOs (clase 10)

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 a partir 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** (

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

{searchedTodos.map(todo => (

))}

);

}

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} TODOsh2>

);

}

**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** (

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

{searchedTodos.map(todo => (

))}

);

}

export default App;

# Completando y eliminando TODOs (clase 11)

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 };

# Organización de archivos y carpetas (clase 12)

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** (

{searchedTodos.map(todo => (

completeTodo(todo.text)}

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

/>

))}

);

}

export { AppUI };

**Persistencia de datos con Local Storage (clase 13)**

Las aplicaciones web tienen tanto front-end como back-end, en **front-end** se encarga de la parte visual e interactuar con los usuarios, así como de conectarse con el **back-end**, en donde se maneja la autenticación, el almacenamiento de datos en bases de datos, esta es una manera muy utilizada para la persistencia de datos.

También es posible la persistencia de datos sin necesidad del back-end, utilizando la **API de almacenamiento web**, el **localStorage**, que nos permite almacenar datos localmente en el navegador, que persistirán incluso si el usuario recarga la página o cierra el navegador.

Además, existe otra forma de almacenar datos, aunque no es persistente, se llama **sessionStorage**, se utiliza exactamente igual que **localStorage**, la diferencia es que los datos en localStorage son persistentes.

**Local Storage**

Nos permite guardar datos persistentes en el navegador del usuario, que podremos acceder, modificar y hasta eliminar, para esto **localStorage** tiene varios métodos.

* Guardar datos: setItem(nombre, dato)
* Acceder a datos: getItem(nombre)
* Borrar un dato: removeItem(nombre)
* Eliminar todos los datos: clear(nombre)

Es muy importante saber que **localStorage** solamente puede guardar texto, no objetos, arreglos, números, **solo strings** para esto podemos utilizar unos métodos de **JSON**:

* Convertir a texto: JSON.stringify()
* Convertir a JavaScript: JSON.parse()

**Local Storage en ToDo App**

Para crear la lógica de nuestro almacenamiento local, antes de acceder a nuestro ítem, debemos tener en cuenta que nuestro usuario puede ser nuevo y no tener ningún *TODO* creado, en este caso necesitaríamos crear un arreglo vacío, si el usuario ya tiene TODOs creados, deberíamos obtener sus TODOs del **localStorage**.

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

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

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

// Traemos nuestros TODOs almacenados

**const** localStorageTodos = localStorage.getItem('TODOS\_V1');

**let** parsedTodos;

**if** (!localStorageTodos) {

// Si el usuario es nuevo no existe un item en localStorage, por lo tanto guardamos uno con un array vacío

localStorage.setItem('TODOS\_V1', JSON.stringify([]));

parsedTodos = [];

} **else** {

// Si existen TODOs en el localStorage los regresamos como nuestros todos

parsedTodos = JSON.parse(localStorageTodos);

}

// Guardamos nuestros TODOs del localStorage en nuestro estado

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

**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;

Todavía no creamos la lógica para agregar nuevos TODOs, pero si probamos añadiendo TODOs de prueba en nuestro localStorage, ya debería funcionar todo bien, o al menos a simple vista. Cuando interactuamos con nuestra aplicación, completando o eliminando TODOs todavía no se ve reflejado en nuestro localStorage.

Para lograr esto, crearemos una función puente, entre nuestra función que actualiza nuestro estado para actualizar nuestro localStorage.

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

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

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

**const** localStorageTodos = localStorage.getItem('TODOS\_V1');

**let** parsedTodos;

**if** (!localStorageTodos) {

localStorage.setItem('TODOS\_V1', JSON.stringify([]));

parsedTodos = [];

} **else** {

parsedTodos = JSON.parse(localStorageTodos);

}

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

**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);

});

}

// Creamos la función en la que actualizaremos nuestro localStorage

**const** saveTodos = (newTodos) => {

// Convertimos a string nuestros TODOs

**const** stringifiedTodos = JSON.stringify(newTodos);

// Los guardamos en el localStorage

localStorage.setItem('TODOS\_V1', stringifiedTodos);

// Actualizamos nuestro estado

setTodos(newTodos);

};

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

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

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

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

// Cada que el usuario interactúe con nuestra aplicación se guardarán los TODOs con nuestra nueva función

saveTodos(newTodos);

};

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

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

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

newTodos.splice(todoIndex, 1);

// Cada que el usuario interactúe con nuestra aplicación se guardarán los TODOs con nuestra nueva función

saveTodos(newTodos);

};

**return** (

<AppUI

totalTodos={totalTodos}

completedTodos={completedTodos}

searchValue={searchValue}

setSearchValue={setSearchValue}

searchedTodos={searchedTodos}

completeTodo={completeTodo}

deleteTodo={deleteTodo}

/>

);

}

export default App;

No fue tan complicado, ¿Verdad?  
Pero ahora tenemos demasiado código en nuestro archivo App.js eso no es bueno, debemos de intentar separar la lógica para que sea más fácil de leer y esté mejor organizado.

**Custom Hook para Local Storage (clase 14)**

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!

# Manejo de efectos (clase 15)

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 \*/}

);

}

**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** (

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

{error &&

Desespérate, hubo un error...

}

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

{loading &&

Estamos cargando, no desesperes...

}

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

{(!loading && !searchedTodos.length) &&

¡Crea tu primer TODO!

}

{searchedTodos.map(todo => (

completeTodo(todo.text)}

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

/>

))}

);

}

export { AppUI };