

Curso de React.js

Apuntes

- React is a javascript library created by facebook
- Is a tool for building UI components
- React create virtual dom in memory.
- React not manipulating the browsers'DOM
- Need node.js installed
- ES6 (2015) = Class, Arrow function, destructuring, module, spread operator, method array(map)
- var = function scope , respecta block scope (not block scope)
- [].map() : return a new [] , in react it used for list
- Destructuring Array: extract only what is needed
 - const [car, truck, suv] = vehicles.
- Destructuring Objects :
 - const {} = object
- Spread Operator : copy all or part of [] -----> const myobject = {...object}
- Ternary Operator : condition ? :
- https://www.w3schools.com/react/react_render.asp

1. ¿Qué necesitas para aprender React.js?

- Programación Básica
- HTML y CSS
- JavaScript
- Git y GitHub
- NPM

Iniciar proyecto

```
git clone git@github.com:platzi/curso-react-intro.git
cd curso-react-intro
npm i
```

Si te aparece algo como esto, tranqui, según leí, esto no nos dará problemas, pero ya veremos que pasa...

```
└ npm i
npm WARN deprecated w3c-hr-time@1.0.2: Use your platform's native
performance.now() and performance.timeOrigin.
npm WARN deprecated svgo@1.3.2: This SVGO version is no longer sup-
ported. Upgrade to v2.x.x.
npm WARN deprecated stable@0.1.8: Modern JS already guarantees Arr
ay#sort() is a stable sort, so this library is deprecated. See the
compatibility table on MDN: https://developer.mozilla.org/en-US/d
```

```
ocs/Web/JavaScript/Reference/Global_Objects/Array/sort#browser_compatibility
```

```
added 1393 packages, and audited 1394 packages in 26s
```

```
212 packages are looking for funding  
  run `npm fund` for details
```

```
12 vulnerabilities (3 moderate, 9 high)
```

```
To address issues that do not require attention, run:  
  npm audit fix
```

```
To address all issues (including breaking changes), run:  
  npm audit fix --force
```

```
Run `npm audit` for details.
```

Veamos como arranca nuestro proyecto:

```
npm start
```

- Repo: [Curso react intro](#)
- ReactWiki by: [midudev](#)
- Proyecto Task tune
- Repo Task tune

2. ¿Qué es un componente?

En React, un componente es una pieza fundamental para construir interfaces de usuario reutilizables y modulares. Puedes considerar un componente como un bloque de construcción que encapsula la lógica y la representación visual de una parte específica de la interfaz de usuario.

En React, los componentes se crean utilizando JavaScript y se escriben como clases o funciones. Las clases se definen extendiendo la clase base [React.Component](#), mientras que las funciones son componentes de función. Con la introducción de los Hooks en React, las funciones también pueden tener estado y realizar acciones adicionales como componentes de clase.

Un componente en React puede aceptar entradas llamadas "props" (abreviatura de propiedades) y devuelve elementos React que describen lo que se debe mostrar en la interfaz de usuario. Las props son objetos que contienen datos y se pasan al componente desde su parent. Estos datos pueden ser cualquier cosa, desde cadenas de texto hasta funciones o incluso componentes completos.

Aquí hay un ejemplo sencillo de un componente de función en React que muestra un saludo personalizado:

```

import React from "react";

function Saludo(props) {
    return <h1>Hola, {props.nombre}!</h1>;
}

export default Saludo;

```

Este componente de función llamado `Saludo` acepta una prop llamada `nombre` y muestra un saludo personalizado. Puede ser utilizado en otros componentes de la siguiente manera:

```

import React from "react";
import Saludo from "./Saludo";

function App() {
    return (
        <div>
            <Saludo nombre="Juan" />
            <Saludo nombre="María" />
        </div>
    );
}

export default App;

```

En este ejemplo, `Saludo` se utiliza como un componente dentro de `App` y se le pasan diferentes valores para la prop `nombre`. Como resultado, se mostrarán dos saludos personalizados en la interfaz de usuario.

Principales formas de crear componentes

1. Componentes de clase: Antes de la introducción de los Hooks en React, los componentes de clase eran la forma principal de crear componentes. Los componentes de clase son clases de JavaScript que extienden la clase base `React.Component`. Dentro de la clase, se define el método `render()` que devuelve los elementos React que describen la interfaz de usuario. Aquí tienes un ejemplo de un componente de clase:

```

import React from "react";

class MiComponente extends React.Component {
    render() {
        return <h1>Hola desde un componente de clase!</h1>;
    }
}

export default MiComponente;

```

2. Componentes de función: Con la introducción de los Hooks en React, los componentes de función se volvieron más populares. Los componentes de función son simplemente funciones de JavaScript que devuelven elementos React. Pueden aceptar props y también pueden tener estado y realizar acciones adicionales utilizando Hooks. Aquí tienes un ejemplo de un componente de función:

```
import React from "react";

function MiComponente() {
    return <h1>Hola desde un componente de función!</h1>;
}

export default MiComponente;
```

Ambas formas de crear componentes son válidas en React, pero los componentes de función con Hooks se han vuelto más comunes debido a su simplicidad y capacidad de reutilización. Sin embargo, las clases todavía se utilizan en casos específicos, especialmente en proyectos heredados o en bibliotecas y frameworks que aún no han adoptado completamente los Hooks.

Los componentes en React ofrecen una forma poderosa y flexible de construir interfaces de usuario reutilizables y mantener un código ordenado y fácil de mantener. Puedes combinar y anidar componentes para construir aplicaciones más complejas y escalables.

JavaScript XML (JSX)

JSX (JavaScript XML) es una extensión de sintaxis utilizada en React y otras bibliotecas similares para construir interfaces de usuario. JSX combina JavaScript y XML (lenguaje de marcado extensible) para definir la estructura y apariencia de los componentes de React.

En lugar de utilizar métodos tradicionales para crear elementos de interfaz de usuario, JSX permite escribir código similar a HTML dentro de archivos JavaScript. Esto facilita la creación de componentes reutilizables y mejora la legibilidad del código.

JSX (JavaScript XML) se utiliza en React para escribir las estructuras y el contenido de los componentes de la interfaz de usuario de manera declarativa.

Cuando se escribe código JSX, se puede pensar en él como una mezcla de JavaScript y código HTML/XML. Permite combinar la lógica de JavaScript con la representación visual de la interfaz de usuario en un solo lugar.

Aquí hay un ejemplo sencillo de cómo se utiliza JSX en un componente de función en React:

```
import React from "react";

function MiComponente() {
    return (
        <div>
            {" "}
        </div>
    );
}
```

```

    <h1>Hola, mundo!</h1>
    <p>Este es un ejemplo de JSX en React.</p>
  </div>
);
}

export default MiComponente;

```

En este ejemplo, el código dentro de las etiquetas `<div>` es JSX. Puedes ver que se mezclan elementos HTML (`<h1>`, `<p>`) con código JavaScript (`{}`) para agregar lógica o referenciar variables dentro del JSX.

JSX se transpila a JavaScript puro utilizando herramientas como Babel durante el proceso de construcción de la aplicación. Esto significa que aunque estés escribiendo código JSX y este se vea similar a HTML, en última instancia se traduce a JavaScript puro que puede ser interpretado por el navegador o por el entorno de ejecución de JavaScript.

En resumen, JSX se utiliza en React para definir la estructura de la interfaz de usuario de manera declarativa y combinar el código JavaScript con la representación visual. Facilita la creación de componentes reutilizables y el desarrollo de interfaces de usuario dinámicas y interactivas.

Nomenclatura PascalCase

Cuando se utiliza JSX (una extensión de sintaxis de JavaScript utilizada en React), los nombres de componentes deben comenzar con una letra mayúscula para que React pueda diferenciar entre componentes y elementos JSX.

```

import React from "react";

// Componente con nombre en PascalCase
class MyCustomComponent extends React.Component {
  render() {
    return <div>Hello, World!</div>;
  }
}

// Elemento JSX con nombre en minúsculas
const myHtmlElement = <div>Hello, World!</div>;

// Uso de componentes en React
const App = () => {
  return (
    <div>
      <MyCustomComponent />
      {myHtmlElement}
    </div>
  );
};

```

Carpetas y Archivos de la clase

Archivo `package.json`

El archivo `package.json` es un archivo de configuración utilizado en proyectos de Node.js y en particular en proyectos de React. Contiene información sobre el proyecto y sus dependencias, así como scripts personalizados y configuraciones adicionales.

Explicación breve del contenido del archivo `package.json`:

- `"name"`: Es el nombre del proyecto, en este caso, "platzi-intro-react-base".
- `"version"`: Es la versión actual del proyecto, en este caso, "0.1.0".
- `"dependencies"`: Es un objeto que enumera las dependencias del proyecto junto con sus versiones. En este caso, las dependencias son "react", "react-dom", "react-scripts" y "web-vitals".
- `"scripts"`: Es un objeto que define comandos de script que puedes ejecutar en el proyecto. En este caso, los scripts son "start", "build" y "eject", que están asociados a los comandos proporcionados por "react-scripts".
- `"eslintConfig"`: Es un objeto que contiene la configuración de ESLint, una herramienta de linting de JavaScript. En este caso, se extiende la configuración "react-app" predefinida.
- `"browserslist"`: Es un objeto que define la lista de navegadores a los que se dirige el proyecto en diferentes entornos (producción y desarrollo), lo cual es útil para la compatibilidad con los navegadores.

El archivo `package.json` es importante porque permite gestionar las dependencias del proyecto, ejecutar scripts personalizados y proporciona información esencial sobre el proyecto en general.

Dependencia

Una dependencia es un elemento externo utilizado por un proyecto para funcionar correctamente. Puede ser una biblioteca, un framework, un módulo o cualquier otro componente que el proyecto necesita para cumplir con sus requisitos y funcionalidades.

Las dependencias se utilizan para aprovechar el trabajo previo realizado por otros desarrolladores y para acceder a funcionalidades específicas sin tener que implementarlas desde cero. Al utilizar dependencias, los desarrolladores pueden ahorrar tiempo y esfuerzo, ya que no necesitan reinventar la rueda y pueden construir sobre componentes ya existentes y bien probados.

En el caso de aplicaciones en React, como en el ejemplo del archivo `package.json`, las dependencias especificadas indican las bibliotecas y módulos que el proyecto necesita para ejecutarse correctamente. Por ejemplo, "react" y "react-dom" son dependencias fundamentales para construir aplicaciones en React, mientras que "react-scripts" es una herramienta que simplifica el proceso de desarrollo y construcción de la aplicación.

Las dependencias se gestionan utilizando un gestor de paquetes, como npm (Node Package Manager) en el caso de proyectos de Node.js. El gestor de paquetes se encarga de descargar, instalar y mantener actualizadas las dependencias del proyecto, asegurando que todas las piezas encajen correctamente.

ESLint

ESLint es una herramienta de linting de código estática para JavaScript. Su objetivo principal es identificar y reportar patrones problemáticos o errores en el código JavaScript, ayudando a mantener un código limpio, legible y libre de errores.

Estas son algunas de las principales razones por las que se utiliza ESLint en proyectos de JavaScript:

1. **Detección temprana de errores y problemas de código:** ESLint analiza el código y señala posibles errores, inconsistencias y malas prácticas. Esto permite identificar problemas en etapas tempranas del desarrollo, evitando que se propaguen y se conviertan en problemas más graves o difíciles de solucionar.
2. **Consistencia y buenas prácticas de codificación:** ESLint ayuda a aplicar y hacer cumplir un conjunto de reglas y convenciones de codificación en todo el proyecto. Esto asegura que todos los desarrolladores sigan un estilo de codificación coherente, facilitando la lectura y colaboración en el código.
3. **Configuración personalizada:** ESLint permite personalizar las reglas y configuraciones según las necesidades del proyecto y del equipo. Se pueden habilitar o deshabilitar reglas específicas, ajustar el nivel de severidad de los errores, e incluso crear reglas personalizadas para adaptarse a los estándares y requisitos del proyecto.
4. **Integración en el flujo de desarrollo:** ESLint se integra con el flujo de trabajo de desarrollo, ya sea a través de la línea de comandos o mediante la integración con editores de código y entornos de desarrollo integrados (IDE). Esto permite que las verificaciones de ESLint se realicen automáticamente durante la escritura del código, proporcionando retroalimentación instantánea al desarrollador.
5. **Compatibilidad con proyectos y frameworks populares:** ESLint es ampliamente compatible con una variedad de proyectos y frameworks de JavaScript, incluyendo React, Vue.js, Angular, Node.js, entre otros. Esto permite aplicar reglas y configuraciones específicas para cada uno de estos proyectos y frameworks.

En resumen, ESLint se utiliza para mejorar la calidad y legibilidad del código JavaScript, ayudando a encontrar y corregir errores, aplicar convenciones de codificación y mantener un estilo de codificación consistente en todo el proyecto.

Carpeta **public** y **src**

En proyectos de React, como en muchos otros frameworks y bibliotecas de desarrollo web, las carpetas "public" y "src" desempeñan roles importantes en la estructura y organización del proyecto.

Explicación del uso de cada una de estas carpetas:

1. Carpeta "public":

La carpeta "public" contiene los archivos estáticos que se sirven directamente al navegador sin procesamiento por parte de React u otras herramientas de compilación. Algunos de los archivos comunes que se encuentran en esta carpeta son el archivo HTML principal, imágenes, fuentes, archivos de manifest para aplicaciones web progresivas, entre otros. Estos archivos están

disponibles públicamente y son accesibles desde la raíz del dominio o subdirectorio donde se encuentra alojada la aplicación.

2. Carpeta "src":

La carpeta "src" es el corazón del proyecto de React y contiene la mayor parte del código fuente de la aplicación. Aquí se encuentran los componentes de React, estilos CSS, archivos JavaScript, imágenes y otros recursos utilizados en la construcción de la interfaz de usuario. La estructura interna de la carpeta "src" puede variar según las preferencias y la arquitectura del proyecto, pero generalmente se organiza en subcarpetas temáticas o basadas en características, como "components" (componentes reutilizables), "pages" (páginas de la aplicación), "styles" (estilos CSS), entre otros.

Es importante destacar que la carpeta "src" es el punto de entrada para el proceso de compilación y construcción del proyecto. Las herramientas de construcción, como Webpack o Babel, toman el código fuente de la carpeta "src" y lo transforman en un bundle (paquete) optimizado y listo para ser servido en el navegador. Es en esta carpeta donde se realiza el desarrollo activo de la aplicación, escribiendo código, creando componentes y definiendo la lógica de la interfaz de usuario.

En resumen, la carpeta "public" contiene los archivos estáticos que se sirven directamente al navegador, mientras que la carpeta "src" alberga el código fuente de la aplicación de React, incluyendo componentes, estilos y otros recursos. Ambas carpetas tienen roles distintos pero complementarios en la estructura y construcción del proyecto de React.

Archivo robots.txt

El archivo **robots.txt** es un archivo de texto utilizado para comunicarse con los robots de los motores de búsqueda, como los rastreadores de los buscadores web. Su objetivo principal es indicar a estos robots qué partes del sitio web deben ser rastreadas o no.

Cuando un robot de búsqueda visita un sitio web, lo primero que hace es buscar y leer el archivo **robots.txt** en la raíz del dominio. Este archivo proporciona directivas específicas para los rastreadores, indicándoles qué páginas o secciones del sitio deben ser accedidas y cuáles deben ser ignoradas.

El formato básico de un archivo **robots.txt** es el siguiente:

```
User-agent: [nombre_del_robot]
Disallow: [ruta_del_directorio_o_archivo_a_excluir]
Allow: [ruta_del_directorio_o_archivo_a_permitir]

User-agent: *
Disallow: [ruta_del_directorio_o_archivo_a_excluir]
Allow: [ruta_del_directorio_o_archivo_a_permitir]
```

[Más info](#)

Código de la clase

Habiendo echado un vistazo a las carpetas y archivos más relevantes del proyecto, ahora toca centrarnos en el archivo `index.html` dentro de carpeta `public` también en los archivos `index.js` y `App.js` de la carpeta `src`.

`src > App.js`

```
import logo from "./platzi.webp";
import "./App.css";

function App() {
    return (
        <div className="App">
            <TodoItem /> {/* ✓ */} 
            <TodoItem /> {/* ✗ */} 
            <TodoItem /> {/* ✗ */} 
            <TodoItem /> {/* ✗ */} 
            <header className="App-header">
                <img src={logo} className="App-logo" alt="logo" />
                <p>
                    Edita el archivo <code>src/App.js</code> y guarda para
                    recargar.
                </p>
                <a
                    className="App-link"
                    href="https://platzi.com/reactjs"
                    target="_blank"
                    rel="noopener noreferrer"
                >
                    Learn React
                </a>
            </header>
        </div>
    );
}

function TodoItem() {
    return (
        <li>
            <span>✓</span>
            <p>Don't cry</p>
            <span>✗</span>
        </li>
    );
}

export default App;
```

Este código es un ejemplo de un componente de React llamado `App`.

Explicación paso a paso del código:

1. `import logo from './platzi.webp';`: Esta línea de código importa el archivo `platzi.webp` y lo asigna a la variable `logo` pude haber sido cualquier nombre, en este caso se eligió el nombre de logo para hacer referencia a esa imagen. Esto se hace utilizando la sintaxis de importación de JavaScript. El archivo `platzi.webp` se importa para ser utilizado posteriormente en el componente `App`.
2. `import './App.css';`: Esta línea de código importa el archivo `App.css`, que contiene estilos CSS específicos para el componente `App`. Los estilos importados se aplicarán a los elementos dentro del componente `App`.
3. La función `App()`: Esta función es el componente principal llamado `App`. Es una función de React que devuelve un elemento JSX que define la estructura y la apariencia del componente.
4. El contenido dentro de `<div className="App">...</div>`: Este es el contenido principal del componente `App`. Contiene varios elementos JSX que representan una lista de elementos `TodoItem`, un encabezado con una imagen, un párrafo y un enlace.
5. `<TodoItem/>`: Esta línea de código renderiza el componente `TodoItem`. El componente `TodoItem` es una función separada que devuelve un elemento JSX que representa un elemento de una lista de tareas pendientes.
6. ``: Esta línea de código muestra una imagen en el componente `App`. Utiliza la variable `logo` importada anteriormente como la fuente (`src`) de la imagen.
7. `<p>...</p>`: Este es un elemento de párrafo que muestra un texto.
8. `<a ...>...`: Este es un elemento de enlace que muestra un texto y tiene un atributo `href` que apunta a un sitio web externo.
9. `function TodoItem() { ... }`: Esta es una función separada que define el componente `TodoItem`. Es un componente simple que devuelve un elemento JSX que representa un elemento de una lista de tareas pendientes.
10. `export default App;`: Esta línea de código exporta el componente `App` para que pueda ser utilizado en otros archivos de la aplicación.

Si nos fijamos también estamos usando clases como la siguiente `<div className="App">`, pero se utiliza `className` en lugar de `class` debido a que estamos escribiendo código en JSX, que es una extensión de sintaxis de JavaScript utilizada en React.

En JavaScript, `class` es una palabra clave reservada para definir clases. Como JSX es una mezcla de JavaScript y XML, se utiliza la convención de usar `className` en lugar de `class` para asignar una clase CSS a elementos JSX. Esto se hace para evitar conflictos con la palabra clave `class` de JavaScript.

Es importante tener en cuenta que en el DOM resultante, el atributo `className` se renderizará como `class`. Esto significa que en el navegador, el elemento `<div className="App">` se representará como `<div class="App">`, y se aplicarán los estilos CSS correspondientes definidos en el archivo `App.css`.

src > index.js

```
import React from "react";
import ReactDOM from "react-dom/client";
import "./index.css";
import App from "./App";

const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(<App />);
```

Este código importa las bibliotecas necesarias y renderiza el componente principal de la aplicación en el elemento con el ID "root".

Explicación paso a paso:

1. `import React from 'react';`: Esta línea de código importa la biblioteca principal de React, que es necesaria para utilizar los componentes y la funcionalidad de React.
2. `import ReactDOM from 'react-dom/client';`: Esta línea de código importa la biblioteca `ReactDOM` que proporciona métodos para interactuar con el DOM (Document Object Model) y renderizar los componentes de React en el navegador. En este caso, estás importando una versión específica de `ReactDOM` llamada `react-dom/client`.
3. `import './index.css';`: Esta línea de código importa el archivo CSS llamado `index.css`. Este archivo contiene estilos CSS específicos para la aplicación.
4. `import App from './App';`: Esta línea de código importa el componente `App` desde el archivo `App.js` o `App.jsx` en el mismo directorio. El componente `App` es el componente principal de la aplicación.
5. `const root = ReactDOM.createRoot(document.getElementById('root'));`: Esta línea de código utiliza el método `createRoot` de `ReactDOM` para crear un "root" (raíz) de la aplicación. El elemento HTML con el ID "root" se pasa como argumento a `createRoot`. Esto establece el punto de entrada para el renderizado de la aplicación.
6. `root.render(<App />);`: Esta línea de código llama al método `render` del objeto `root` creado anteriormente. El método `render` toma como argumento el componente `App` encapsulado en JSX (`<App />`) y se encarga de renderizar ese componente en el elemento raíz de la aplicación.

En resumen, el código importa las bibliotecas necesarias, establece el punto de entrada para el renderizado de la aplicación en el elemento con el ID "root", y finalmente renderiza el componente `App` en ese punto de entrada. Esto inicia la ejecución de la aplicación y muestra el contenido del componente `App` en el navegador.

public > index.html

```
<!DOCTYPE html>
<html lang="en">
```

```

<head>
  <meta charset="utf-8" />
  <!-- Define la codificación de caracteres del documento -->
  <link rel="icon" href="%PUBLIC_URL%/favicon.ico" />
  <!-- Enlace al ícono del sitio web -->
  <meta name="viewport" content="width=device-width, initial-scale=1" />
  <!-- Configuración de la vista en dispositivos móviles -->
  <meta name="theme-color" content="#97ca3f" />
  <!-- Define el color de tema de la aplicación -->
  <meta
    name="description"
    content="Web site created using create-react-app"
  />
  <!-- Descripción del sitio web -->
  <link rel="apple-touch-icon" href="%PUBLIC_URL%/react192.png" />
  <!-- Enlace al ícono de la aplicación en dispositivos Apple -->
  <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
  <!-- Enlace al archivo de manifiesto de la aplicación web -->
  <title>React App</title>
  <!-- Título de la página web -->
</head>
<body>
  <noscript>You need to enable JavaScript to run this app.</noscript>
  <!-- Mensaje para navegadores sin JavaScript habilitado -->
  <div id="root"></div>
  <!-- Punto de montaje para la aplicación de React -->
</body>
</html>

```

En resumen, tenemos 3 archivos que se van a enlazar e interactuar entre sí, ver la estructura actual.

```

.
├── README.md
├── node_modules
├── package-lock.json
└── package.json
└── public
    ├── favicon.ico
    ├── index.html
    ├── manifest.json
    ├── react192.png
    ├── react512.png
    └── robots.txt
└── src
    ├── App.css
    ├── App.js
    ├── index.css
    ├── index.js
    └── platzι.webp

```

The diagram illustrates the structure of a React application across three files:

- index.html**: Contains the main HTML structure with a `<div id="root">` element.
- index.js**: A script file where a `ReactDOM.createRoot` instance is created for the `'root'` element, and the `App` component is rendered into it.
- App.js**: A component file containing the `App()` function, which returns the `App` component. It also includes a `TodoItem` component and an export statement for the `App` component.

Annotations explain the process:

- 1. Creamos un <div> al que le insertaremos componentes por medio de su id.**
- Importamos...** (points to the `App` import in index.js)
- Conectamos con el archivo HTML** (points to the `root.render` call in index.js)
- 3. Hacemos uso del archivo App.js e insertamos el componente App() al archivo index.html en la etiqueta <div>**
- JSX** (points to the JSX code in App.js)
- Componente** (points to the `App` component definition in App.js)
- Insertamos** (points to the `root.render` call in index.js)
- 2. Creamos componentes usando JSX.**
- Componente** (points to the `TodoItem` component definition in App.js)
- Exportamos...** (points to the `export default App;` statement in App.js)

En este ejemplo insertamos un componente dentro de otro componente para luego insertar el componente "Padre" en el HTML.

Web scraping

Web scraping es el proceso de extraer información o datos de sitios web de manera automatizada. Consiste en utilizar un programa o un conjunto de herramientas para analizar y recopilar datos de las páginas web, generalmente a través del análisis del código HTML de esas páginas.

El objetivo del web scraping es obtener datos estructurados y relevantes de las páginas web para su posterior uso o análisis. Puede implicar la extracción de texto, imágenes, enlaces, tablas u otros elementos específicos de una página web.

El proceso de web scraping suele seguir estos pasos:

1. Obtener la URL del sitio web desde el cual se desea extraer los datos.
2. Descargar el código fuente HTML de la página web.
3. Analizar el código HTML para identificar los elementos de interés, como etiquetas, clases o identificadores específicos.
4. Extraer los datos deseados utilizando técnicas como la búsqueda, filtrado o extracción basada en patrones.
5. Almacenar los datos extraídos en un formato estructurado, como un archivo CSV, JSON o una base de datos.

El web scraping puede tener diversas aplicaciones, como la recopilación de información para análisis de mercado, seguimiento de precios, comparación de productos, monitoreo de noticias, obtención de datos para investigación o generación de conjuntos de datos para entrenar modelos de aprendizaje automático, entre otros.

Es importante tener en cuenta que al realizar web scraping, es fundamental respetar los términos de servicio y las políticas de privacidad de los sitios web. Algunos sitios pueden tener restricciones o prohibiciones sobre la extracción automatizada de datos, por lo que es recomendable revisar y cumplir con las políticas de cada sitio antes de realizar web scraping.

Documentación oficial de React

- Documentación React
- Documentación React Español

Complementos:

- Curso de React 2023 por midudev
- Aprende React Desde Cero - Curso de React Con Proyectos

Dato útil

Autocompletado de elementos JSX

Para poder tener autocompletado de elementos JSX en React debes agregar lo siguiente dentro de las configuraciones de Visual Studio Code: (**Ctrl + ,**) y luego en el ícono de la parte superior derecha ↗

```
"emmet.includeLanguages": {
  "javascript": "javascriptreact"
}
```

3. Componentes de TODO Machine

Código de la clase

Vamos a crear diferentes componentes, cada uno con su respectivo archivo, de tal manera que se vea tal cual la siguiente estructura:

```
└ tree -L 2
.
├── README.md
├── node_modules
├── package-lock.json
├── package.json
└── public
    ├── favicon.ico
    ├── index.html
    ├── manifest.json
    ├── react192.png
    ├── react512.png
    └── robots.txt
└── src
    ├── App.css
    ├── App.js
    ├── TodoButton.js
    ├── TodoCounter.js
    ├── TodoItem.js
    ├── TodoList.js
    ├── TodoSearch.js
    └── index.css
```

```
└── index.js  
└── platzi.webp
```

src > TodoCounter.js

```
function TodoCounter() {  
    return <h1>Has completado 3 de 5 ToDos</h1>;  
}  
  
export { TodoCounter };
```

src > TodoSearch.js

```
function TodoSearch() {  
    return <input placeholder="Lorem lorem lorem" />;  
}  
  
export { TodoSearch };
```

src > TodoList.js

```
function TodoList(props) {  
    return <ul>{props.children}</ul>;  
}  
  
export { TodoList };
```

src > TodoItem.js

```
function TodoItem() {  
    return (  
        <li>  
            <span>✓</span>  
            <p>Don't cry</p>  
            <span>✗</span>  
        </li>  
    );  
}  
  
export { TodoItem };
```

src > TodoButton.js

```

function TodoButton() {
    return <button>Heart</button>;
}

export { TodoButton };

```

Una vez tenemos creado los componentes, ahora si podemos importarlos en nuestro archivo [App.js](#).

```

import { TodoCounter } from "./TodoCounter";
import { TodoSearch } from "./TodoSearch";
import { TodoList } from "./TodoList";
import { TodoItem } from "./TodoItem";
import { TodoButton } from "./TodoButton";

import "./App.css";

function App() {
    return (
        <div className="App">
            <TodoCounter />
            <TodoSearch />

            <TodoList>
                <TodoItem />
                <TodoItem />
                <TodoItem />
            </TodoList>

            <TodoButton />
        </div>
    );
}

export default App;

```

❖ Dato: Si queremos importar de manera rápida un componente previamente creado, podemos presionar [Ctrl + Barra Espaciadora](#) sobre la invocación del componente, en este caso podría ser `<TodoButton>` y luego damos enter.

Props

En React, los props (abreviatura de "propiedades") son uno de los conceptos fundamentales para pasar datos y configuraciones entre componentes. Los props son utilizados para transmitir información desde un componente padre a un componente hijo.

Cuando creas un componente en React, puedes pasarte datos utilizando atributos similares a los atributos HTML. Estos datos se denominan props y se pasan como argumentos al componente en su declaración.

Los props son objetos que contienen pares clave-valor, donde la clave es el nombre del prop y el valor es el dato que se está pasando.

Aquí tienes un ejemplo básico para ilustrar cómo se utilizan los props:

```
// ComponentePadre.js
import React from "react";
import ComponenteHijo from "./ComponenteHijo";

const ComponentePadre = () => {
  const nombre = "Juan";
  const edad = 25;

  return (
    <div>
      <ComponenteHijo nombre={nombre} edad={edad} />
    </div>
  );
};

export default ComponentePadre;
```

```
// ComponenteHijo.js
import React from "react";

const ComponenteHijo = (props) => {
  return (
    <div>
      <h2>Nombre: {props.nombre}</h2>
      <p>Edad: {props.edad}</p>
    </div>
  );
};

export default ComponenteHijo;
```

En el ejemplo anterior, el componente `ComponentePadre` pasa los props `nombre` y `edad` al componente `ComponenteHijo`. El componente hijo recibe los props como argumento en su función y puede acceder a ellos utilizando la sintaxis `props.propName`. En este caso, se muestra el nombre y la edad recibidos en el componente hijo.

Los props son de solo lectura, lo que significa que no se deben modificar dentro del componente hijo. Si necesitas modificar datos dentro de un componente, puedes utilizar el estado (state). Los props se utilizan principalmente para transmitir datos estáticos o configuraciones entre componentes.

Además de los datos primitivos como cadenas de texto o números, también puedes pasar funciones como props para permitir la comunicación entre componentes y manejar eventos o acciones en el componente

padre.

En resumen, los props en React son utilizados para pasar datos y configuraciones entre componentes. Permiten la comunicación unidireccional desde un componente padre a un componente hijo. Los props son de solo lectura y se accede a ellos dentro del componente hijo a través del objeto `props`.

Exportar e Importar Componentes

En React, existen diferentes formas de exportar e importar componentes. Las dos formas mencionadas en clase son `export default App;` y `export { App };`, son dos enfoques distintos para exportar un componente desde un archivo.

1. `export default App;`: Esta sintaxis se utiliza para exportar un solo valor por defecto desde un archivo. Es comúnmente utilizado para exportar el componente principal de una aplicación React. Solo se puede tener un valor `default` por archivo. Al importar el componente en otro archivo, no es necesario usar llaves de desestructuración y se puede elegir cualquier nombre para el componente importado.

Ejemplo de exportación:

```
// App.js
import React from "react";

const App = () => {
  // ...
};

export default App;
```

Ejemplo de importación:

```
// OtroArchivo.js
import MiComponente from "./App"; // No se usan llaves de desestructuración

// ...
```

2. `export { App };`: Esta sintaxis se utiliza para exportar uno o varios valores específicos desde un archivo. Puedes exportar múltiples componentes o variables utilizando esta sintaxis. Al importar los valores en otro archivo, debes usar llaves de desestructuración y respetar el nombre exacto utilizado en la exportación.

Ejemplo de exportación:

```
// Componentes.js
import React from "react";
```

```

const Componente1 = () => {
    // ...
};

const Componente2 = () => {
    // ...
};

export { Componente1, Componente2 };

```

Ejemplo de importación:

```

// OtroArchivo.js
import { Componente1, Componente2 } from "./Componentes"; // Se usan llaves de
desestructuración

// ...

```

En resumen, `export default` se utiliza para exportar un solo valor por defecto, mientras que `export { }` se utiliza para exportar múltiples valores específicos. La elección de cuál usar depende de la estructura y necesidades de tu aplicación.

- [Extensión: ES7+ React/Redux/React-Native snippets rfce](#)

4. ¿Cómo se comunican los componentes? Props y atributos

Desestructuración

La desestructuración es una característica de JavaScript que también se puede utilizar en React como una alternativa para acceder a los props de manera más concisa y directa. En lugar de acceder a los props a través del objeto `props.propName`, puedes extraer los props específicos que necesitas y utilizarlos directamente en tu componente.

Aquí tienes un ejemplo que muestra cómo se utiliza la desestructuración con los props:

```

// ComponenteHijo.js
import React from "react";

const ComponenteHijo = ({ nombre, edad }) => {
    return (
        <div>
            <h2>Nombre: {nombre}</h2>
            <p>Edad: {edad}</p>
        </div>
    );
};

export default ComponenteHijo;

```

En este ejemplo, en lugar de utilizar `props.nombre` y `props.edad`, hemos desestructurado los props en los parámetros de la función del componente hijo: `({ nombre, edad })`. Esto significa que solo estamos extrayendo los valores de `nombre` y `edad` del objeto `props`, lo que nos permite utilizar directamente esas variables en el componente sin necesidad de acceder a través de `props`.

Es importante destacar que la desestructuración solo extrae los props necesarios del objeto `props`. Si hay otros props que no se han desestructurado, seguirán estando disponibles en el objeto `props`.

En resumen, la desestructuración es una característica de JavaScript que se puede utilizar en React para acceder a los props de manera más concisa y directa. Permite extraer los props necesarios y utilizarlos como variables individuales en lugar de acceder a través del objeto `props`. Esto simplifica la sintaxis y mejora la legibilidad del código.

Propiedad Children

La prop `children` en React es una prop especial que permite pasar contenido entre las etiquetas de apertura y cierre de un componente. Esta prop se utiliza para transmitir elementos hijos directos a un componente y proporciona una forma flexible de componer componentes y anidar contenido dentro de ellos.

Cuando utilizas la prop `children`, puedes incluir cualquier tipo de contenido dentro del componente, ya sean elementos de React, texto, números u otros componentes. Puedes pensar en `children` como el espacio reservado para el contenido que se encuentra entre las etiquetas de apertura y cierre de un componente.

Aquí tienes un ejemplo para ilustrar cómo se utiliza la prop `children`:

```
// ComponentePadre.js
import React from "react";

const ComponentePadre = () => {
  return (
    <div>
      <h1>Título del componente padre</h1>
      <ComponenteHijo>
        <p>Este es un párrafo dentro del componente hijo.</p>
        <button>Haz clic</button>
      </ComponenteHijo>
    </div>
  );
};

export default ComponentePadre;
```

```
// ComponenteHijo.js
import React from "react";
```

```

const ComponenteHijo = ({ children }) => {
  return (
    <div>
      <h2>Componente Hijo</h2>
      {children}
    </div>
  );
};

export default ComponenteHijo;

```

En este ejemplo, el componente `ComponentePadre` pasa contenido entre las etiquetas de apertura y cierre del componente `ComponenteHijo`. El contenido incluye un párrafo y un botón. En el componente hijo, utilizamos la prop `children` para mostrar el contenido pasado.

La prop `children` puede utilizarse de diversas formas en el componente receptor. Puede ser renderizada directamente utilizando `{children}` como en el ejemplo anterior, o puedes manipularla, recorrerla o aplicarle lógica según tus necesidades.

Es importante mencionar que el componente receptor puede tener otros props además de `children`. Puedes combinar la prop `children` con otros props para crear componentes más flexibles y reutilizables.

La prop `children` permite componer componentes de manera dinámica y anidar contenido de una forma intuitiva. Es útil cuando deseas que un componente contenga contenido variable o cuando necesitas crear componentes reutilizables que pueden envolver otros elementos o componentes.

En resumen, la prop `children` en React permite pasar contenido entre las etiquetas de apertura y cierre de un componente. Es una forma de componer componentes y anidar contenido dentro de ellos. Puedes utilizar cualquier tipo de contenido, como elementos de React, texto o componentes, y acceder a él utilizando la prop `children` en el componente receptor.

`<React.Fragment>` o `<> </>`

En React, `<React.Fragment>` o `<> </>` (también conocido como JSX Fragment) es una característica que te permite agrupar múltiples elementos hijos sin necesidad de agregar un elemento contenedor adicional como un `div`.

Cuando trabajas con JSX en React, generalmente se espera que devuelvas un solo elemento JSX en el método `render()` de un componente. Sin embargo, puede haber situaciones en las que deseas renderizar varios elementos adyacentes sin envolverlos en un elemento contenedor adicional. Aquí es donde `<React.Fragment>` o `<> </>` resultan útiles.

Aquí tienes un ejemplo para ilustrar cómo se utiliza `<React.Fragment>`:

```

import React from "react";

const ComponentePadre = () => {
  return (

```

```

        <React.Fragment>
          <h1>Título del componente</h1>
          <p>Este es un párrafo dentro del componente.</p>
          <button>Haz clic</button>
        </React.Fragment>
      );
    };

export default ComponentePadre;

```

En este ejemplo, hemos utilizado `<React.Fragment>` para envolver múltiples elementos adyacentes: un encabezado (`<h1>`), un párrafo (`<p>`) y un botón (`<button>`). `<React.Fragment>` no genera un elemento adicional en el DOM, sino que solo actúa como un contenedor imaginario para agrupar los elementos.

Una forma más corta y concisa de utilizar fragmentos es utilizando la sintaxis `<> </>` (también conocida como fragment shorthand o fragmento abreviado) en lugar de `<React.Fragment>`:

```

import React from "react";

const ComponentePadre = () => {
  return (
    <>
      <h1>Título del componente</h1>
      <p>Este es un párrafo dentro del componente.</p>
      <button>Haz clic</button>
    </>
  );
};

export default ComponentePadre;

```

En este caso, hemos utilizado `<> </>` en lugar de `<React.Fragment>`, lo cual es una forma más compacta y legible de lograr el mismo resultado.

Al utilizar `<React.Fragment>` o `<> </>`, puedes agrupar elementos sin crear nodos adicionales en el DOM. Esto puede ser útil cuando necesitas renderizar una lista de elementos sin agregar un contenedor adicional o cuando deseas evitar estilos o efectos no deseados que podrían ser aplicados por el elemento contenedor.

En resumen, `<React.Fragment>` o `<> </>` en React son utilizados para agrupar múltiples elementos hijos sin necesidad de agregar un elemento contenedor adicional. Proporcionan una forma conveniente de renderizar elementos adyacentes y evitar nodos adicionales en el DOM.

Renderizar elementos a través de un Array

En React, puedes renderizar elementos a través de un array utilizando el método `map()`. El método `map()` itera sobre cada elemento de un array y devuelve un nuevo array con los elementos modificados según la lógica que definas. Puedes utilizar este nuevo array para renderizar elementos en tu componente.

Aquí tienes un ejemplo para ilustrar cómo renderizar elementos a través de un array en React:

```
import React from "react";

const ComponentePadre = () => {
  const elementos = ["Elemento 1", "Elemento 2", "Elemento 3"];

  return (
    <div>
      {elementos.map((elemento, index) => (
        <p key={index}>{elemento}</p>
      ))}
    </div>
  );
};

export default ComponentePadre;
```

En este ejemplo, tenemos un componente `ComponentePadre` que contiene un array llamado `elementos`. Utilizamos el método `map()` en `elementos` para iterar sobre cada elemento y generar un nuevo array de elementos `<p>`.

Dentro del método `map()`, utilizamos una función de flecha para definir la lógica de renderizado de cada elemento. En este caso, estamos generando un `<p>` para cada elemento en el array `elementos`. La propiedad `key` se establece en el índice del elemento para proporcionar una identificación única a cada elemento.

Finalmente, el nuevo array de elementos generados se renderiza dentro del componente `ComponentePadre`.

Ten en cuenta que cuando utilizas el método `map()` para renderizar elementos a partir de un array, es importante proporcionar una `key` única para cada elemento. La `key` ayuda a React a realizar una actualización eficiente de los elementos cuando cambian.

Código de la clase

Vamos a recorrer un array para mostrar por default al usuario algunas tareas:

`src > App.js`

```
import { TodoCounter } from "./TodoCounter";
import { TodoSearch } from "./TodoSearch";
import { TodoList } from "./TodoList";
import { TodoItem } from "./TodoItem";
import { TodoButton } from "./TodoButton";

import "./App.css";
import React from "react";
```

```

const defaultTodos = [
  { text: "Lorem lorem", completed: false },
  { text: "Don't cry", completed: false },
  { text: "Lorem lorem", completed: false },
  { text: "Don't cry", completed: false },
  { text: "Lorem lorem", completed: false },
];

function App() {
  return (
    <React.Fragment>
      <TodoCounter completed={16} total={25} />
      <TodoSearch />

      <TodoList>
        {defaultTodos.map((todo) => (
          <TodoItem
            key={todo.text}
            text={todo.text}
            completed={todo.completed}
          />
        ))}
      </TodoList>

      <TodoButton />
    </React.Fragment>
  );
}

export default App;

```

src > TodoCounter.js

```

function TodoCounter({ completed, total }) {
  return (
    <h1>
      Has completado {completed} de {total} ToDos
    </h1>
  );
}

export { TodoCounter };

```

src > TodoList.js

```

function TodoList({ children }) {
  return <ul>{children}</ul>;

```

```
}

export { TodoList };
```

src > TodoItem.js

```
function TodoItem(props) {
  return (
    <li>
      <span>✓</span>
      <p>{props.text}</p>
      <span>✗</span>
    </li>
  );
}

export { TodoItem };
```

[Dominando las keys en React.js: aprende cómo implementarlas](#)

5. Estilos CSS en React

En React, puedes agregar estilos a tus componentes de varias formas. Aquí te explicaré algunas opciones comunes:

1. CSS Externo: Puedes usar archivos CSS externos de la misma manera que lo harías en una página web regular. Simplemente importa el archivo CSS en el componente donde deseas aplicar los estilos. Por ejemplo:

```
import React from "react";
import "./styles.css";

const MiComponente = () => {
  return <div className="mi-estilo">Contenido del componente</div>;
};

export default MiComponente;
```

En el archivo CSS `styles.css`, puedes definir la clase `mi-estilo` con los estilos deseados:

```
.mi-estilo {
  color: blue;
  font-weight: bold;
}
```

2. Estilos en línea: Puedes aplicar estilos en línea directamente a elementos JSX utilizando el atributo **style**. Define un objeto JavaScript con las propiedades de estilo y asignalo al atributo **style** del elemento. Por ejemplo:

```
import React from "react";

const MiComponente = () => {
    const estilo = {
        color: "blue",
        fontWeight: "bold",
    };

    return <div style={estilo}>Contenido del componente</div>;
};

export default MiComponente;
```

3. Módulos de Estilo: React también admite módulos de estilo, que te permiten definir estilos específicos para cada componente. Los módulos de estilo generan nombres de clase únicos y los asignan automáticamente a los elementos JSX en tu componente. Para usar módulos de estilo, debes renombrar tu archivo de estilo con una extensión **.module.css**. Por ejemplo, **styles.module.css**.

```
import React from "react";
import styles from "./styles.module.css";

const MiComponente = () => {
    return <div className={styles.miEstilo}>Contenido del componente</div>;
};

export default MiComponente;
```

En el archivo **styles.module.css**, define las clases de estilo como propiedades del objeto **styles**:

```
.miEstilo {
    color: blue;
    font-weight: bold;
}
```

Estas son solo algunas de las formas comunes de agregar estilos en React. Puedes elegir la opción que mejor se adapte a tus necesidades.

Clases condicionales basadas en propiedades

En React, las clases CSS que se generan dinámicamente utilizando expresiones lógicas, se conocen comúnmente como "clases condicionales" o "clases condicionales basadas en propiedades".

En el siguiente ejemplo:

```
<p className={`${props.completed && "p--completed"}`}>...</p>
```

Se utiliza una expresión lógica para condicionar la aplicación de una clase CSS. Si `props.completed` es verdadero, se agrega la clase "`p--completed`"; de lo contrario, no se agrega la clase.

El uso de las llaves `{}` y la sintaxis `${}` dentro de la cadena de clase permite evaluar la expresión lógica y generar una cadena de clase condicionalmente.

Es importante tener en cuenta que esto es una técnica común en React para condicionar la aplicación de clases CSS en función de las propiedades o el estado de los componentes. Puedes utilizar esta técnica en combinación con cualquier librería de estilos en React, como CSS Modules, Styled Components o cualquier otro enfoque que prefieras para manejar los estilos en tu aplicación React.

Otro ejemplo:

Supongamos que tenemos un componente llamado `TaskItem` que representa un elemento de una lista de tareas, y queremos aplicar una clase condicionalmente según si la tarea está completada o no:

```
import React from "react";

const TaskItem = (props) => {
  return (
    <div className={`${task-item ${props.completed ? "task-completed" : ""}}`}>
      {props.taskName}
    </div>
  );
};

export default TaskItem;
```

En este ejemplo, el componente `TaskItem` recibe una propiedad `completed` que indica si la tarea está completada. Utilizamos la expresión lógica `props.completed ? 'task-completed' : ''` para generar de manera condicional la cadena de clase CSS.

Si `props.completed` es verdadero, se agrega la clase "`task-completed`" a la clase principal "`task-item`". De lo contrario, no se agrega ninguna clase adicional.

Luego, en el renderizado del componente, utilizamos la clase resultante en el elemento `<div>` que representa el elemento de la tarea.

Código de la clase

Para realizar esta parte y tener un poco de orden cree la siguiente estructura:

```
|- package-lock.json  
|- package.json  
|- public  
|   |- index.html  
|   |- manifest.json  
|   |- robots.txt  
|- src  
  |- App.js  
  |- components   
    |- TodoButton.js  
    |- TodoCounter.js  
    |- TodoItem.js  
    |- TodoList.js  
    |- TodoSearch.js  
  |- css   
    |- TodoButton.css  
    |- TodoCounter.css  
    |- TodoItem.css  
    |- TodoList.css  
    |- TodoSearch.css  
    |- index.css  
  |- index.js  
  |- svg   
    |- add-purple.svg  
    |- check-completed.svg  
    |- check.svg  
    |- delete-hover.svg  
    |- delete.svg  
    |- search.svg
```

src > App.js

```
import React from "react";  
import { TodoCounter } from "./components/TodoCounter";  
import { TodoSearch } from "./components/TodoSearch";  
import { TodoList } from "./components/TodoList";  
import { TodoItem } from "./components/TodoItem";  
import { TodoButton } from "./components/TodoButton";  
  
const defaultTodos = [  
  { text: "Lorem lorem", completed: true },   
  { text: "Don't cry", completed: false },  
  { text: "Lorem ipsum", completed: false },  
  { text: "Take care", completed: false },  
  { text: "Loremlorem", completed: false },  
];
```

```

function App() {
  return (
    <>
      <TodoCounter completed={16} total={25} />
      <TodoSearch />

      <TodoList>
        {defaultTodos.map((todo) => (
          <TodoItem
            key={todo.text}
            text={todo.text}
            completed={todo.completed} ✎ 00 ⚡
          />
        ))}
      </TodoList>

      <TodoButton />
    </>
  );
}

export default App;

```

En el `index.html` agregamos las fuentes de nuestra preferencia.

`src > css > index.css`

```

* {
  margin: 0;
  padding: 0;
  box-sizing: border-box;
}

body {
  background-color: #090b10;
  padding: 4rem 1.5rem 2rem;
  font-family: "Montserrat", Arial, Helvetica, sans-serif;
  min-height: 100vh;
}

```

Componente TodoCounter

`src > components > TodoCounter.js`

```

import "../css/TodoCounter.css";

function TodoCounter({ completed, total }) {
  return (
    <h1>

```

```

        Has completado <span className="completed">{completed}</span> de"
    "}
        <span className="total">{total}</span> ToDos
    </h1>
);
}

export { TodoCounter };

```

src > css > TodoCounter.css

```

h1 {
    width: auto;
    height: 15vh;
    font-size: 24px;
    text-align: center;
    margin: 0 auto;
    /* display: flex;
    justify-content: center;
    align-items: center; */
    color: #cbd5e1;
    padding: 0 2rem;
}

span {
    color: #4f46e5;
}

```

Componente TodoSearch

src > components > TodoSearch.js

```

import "../css/TodoSearch.css";

function TodoSearch() {
    return <input className="search" placeholder="Search..." />;
}

export { TodoSearch };

```

src > css > TodoSearch.css

```

input {
    margin: 1.5rem auto 2rem;
    display: flex;
    width: 15rem;
}

```

```

height: 2rem;
border-radius: 10px;
padding: 1rem;

background-image: url("../svg/search.svg");
background-repeat: no-repeat;
background-position: 202px center;

border: 1px solid #4f46e5;
box-shadow: -5px 5px 5px -5px #4f46e5;
/* background: rgba(255, 255, 255, 0.1);
background: hsla(0,0%,100%,.5); */
background-color: rgba(0.035, 0.043, 0.063, 0.1);
color: #cbd5e1;
}

input::placeholder {
    color: #cbd5e1;
}

```

Componente TodoList

src > components > TodoList.js

```

import "../css/TodoList.css";

function TodoList({ children }) {
    return <ul>{children}</ul>;
}

export { TodoList };

```

src > css > TodoList.css

```

ul {
    display: grid;
    grid-template-columns: 1fr;
    gap: 1rem;
    justify-items: center;
}

```

Componente TodoItem

src > components > TodoItem.js

```

import "../css/TodoItem.css";

function TodoItem(props) {
  return (
    <li>
      <span className={`check ${props.completed && "check--active"}`}>
        <img alt="checkmark icon" data-bbox="231 169 301 186" />
      <p className={`${$props.completed && "p--completed"}`}>
        {props.text}
      </p>
      <span className={`delete`}></span> <img alt="trash icon" data-bbox="581 248 651 265" />
    </li>
  );
}

export { TodoItem };

```

Si dentro del array `defaultTodos` creado en el componente padre `App.js` existe alguna tarea en `true` esta cumplirá con los parámetros definidos dentro de la `className` del elemento `p` definida en el componente `TodoItems.js` y dará el estilo `text-decoration: line-through;` que tachará nuestra tarea como realizada.

`src > css > TodoItem.css`

```

li {
  list-style: none;
  background-color: #cbd5e1;
  width: 15rem;
  height: 3rem;
  border-radius: 5px;
  display: flex;
  justify-content: space-between;
  align-items: center;
  padding: 0 1rem 0 0.5rem;

  position: relative;
}

.check {
  background-image: url("../svg/check.svg");
  background-repeat: no-repeat;
  background-position: center;
  background-size: contain;
  width: 20px;
  height: 20px;
  cursor: pointer;
}

```

```

.check--active {
    background-image: url("../svg/check-completed.svg");
}

.delete {
    background-image: url("../svg/delete.svg");
    background-repeat: no-repeat;
    background-position: center;
    background-size: contain;
    width: 15px;
    height: 15px;
    cursor: pointer;

    position: absolute;
    right: 0;
    top: 0;
}

.delete:hover {
    background-image: url("../svg/delete-hover.svg");
}

p {
    width: 85%;
    height: auto;
    margin: 0 5px 0;
    color: #090b10;
}

.p--completed {
    text-decoration: line-through;
}

```

Componente TodoButton

src > components > TodoButton.js

```

import "../css/TodoButton.css";

function TodoButton() {
    return <button className="add"></button>;
}

export { TodoButton };

```

src > css > TodoButton.css

```

.add {
    border: none;
    background-color: #090b10;
    border-radius: 50%;
    width: 3rem;
    height: 3rem;

    background-image: url("../svg/add-purple.svg");
    background-size: contain;
    background-repeat: no-repeat;
    background-position: center;

    position: fixed;
    bottom: 1rem;
    right: 1rem;
    cursor: pointer;

    transition: transform 0.3s ease;
}

.add:hover {
    transform: rotate(90deg);
}

```

- Código de mi proyecto
- Diseño en Figma
- Diseño Prototipo
- Guía BEM
- Proyecto 01
- Repo P01
- Proyecto 02
- Repo 01
- Repo 02

6. Eventos en React: onClick, onChange

En React, los eventos son acciones o interacciones que ocurren en los elementos de la interfaz de usuario, como hacer clic en un botón, mover el mouse sobre un elemento, ingresar texto en un campo de entrada, etc. Los eventos en React se manejan utilizando funciones llamadas "manejadores de eventos" que se ejecutan cuando ocurre el evento.

Aquí hay algunos conceptos clave relacionados con los eventos en React:

1. Sintaxis de manejo de eventos: En React, los manejadores de eventos se pasan como funciones a los elementos JSX utilizando una sintaxis especial. Por ejemplo, para manejar el evento de clic en un botón, puedes hacer algo como esto:

```
<button onClick={handleClick}>Haz clic aquí</button>
```

En este ejemplo, `handleClick` es la función que se ejecutará cuando se haga clic en el botón.

2. Eventos sintéticos: React utiliza eventos sintéticos que son una envoltura cruzada de los eventos nativos del navegador. Estos eventos sintéticos tienen la misma interfaz que los eventos nativos del navegador, pero se comportan de manera consistente en todos los navegadores. Por lo tanto, no necesitas preocuparte por la compatibilidad del navegador al manejar eventos en React.
3. Pasar argumentos a los manejadores de eventos: Si necesitas pasar argumentos adicionales a un manejador de eventos, puedes hacerlo utilizando una función de flecha o una función anónima en el lugar donde se define el manejador de eventos. Por ejemplo:

```
<button onClick={() => handleClick(arg1, arg2)}>Haz clic aquí</button>
```

Aquí, `arg1` y `arg2` son los argumentos que se pasan al manejador de eventos `handleClick`.

4. Prevención de comportamiento predeterminado: En algunos casos, es posible que desees evitar el comportamiento predeterminado de un evento, como evitar que un formulario se envíe o evitar que un enlace cambie de página. En React, puedes llamar al método `preventDefault()` en el evento pasado al manejador de eventos para evitar el comportamiento predeterminado. Por ejemplo:

```
function handleSubmit(event) {  
  event.preventDefault();  
  // Resto del código de manejo del formulario  
}  
  
<form onSubmit={handleSubmit}>  
  /* Campos de formulario */  
  <button type="submit">Enviar</button>  
</form>;
```

En este ejemplo, al llamar a `event.preventDefault()`, evitamos que el formulario se envíe y la página se recargue.

Estos son solo algunos aspectos básicos de los eventos en React. React proporciona una amplia gama de eventos que se pueden utilizar para interactuar con los elementos de la interfaz de usuario y controlar el flujo de la aplicación. Puedes consultar la documentación oficial de React para obtener más información sobre los eventos y cómo utilizarlos en tu aplicación.

`onClick` y `onChange`

En React, `onClick` y `onChange` son dos de los eventos más comunes utilizados para manejar interacciones y cambios en los elementos de la interfaz de usuario.

1. **onClick**: El evento **onClick** se dispara cuando se hace clic en un elemento, como un botón, un enlace o una imagen. Puedes asignar una función al evento **onClick** para ejecutar acciones específicas cuando se produce el clic. Por ejemplo:

```
<button onClick={handleClick}>Haz clic aquí</button>
```

En este caso, cuando el botón se hace clic, se ejecutará la función **handleClick**.

2. **onChange**: El evento **onChange** se utiliza principalmente para elementos de entrada, como campos de texto o selectores. Se dispara cuando el valor del elemento cambia, generalmente cuando el usuario ingresa texto o selecciona una opción diferente. Puedes asignar una función al evento **onChange** para manejar los cambios y actualizar el estado de la aplicación. Por ejemplo:

```
<input type="text" onChange={handleChange} />
```

Aquí, **handleChange** es la función que se ejecutará cuando el valor del campo de texto cambie.

Cuando se utiliza **console.log(event)** para registrar la información proporcionada por un evento, se mostrará en la consola un objeto **Event** que contiene varios datos relevantes. Algunos de los datos más comunes y útiles que se pueden encontrar en el objeto **event** son:

- **event.target**: Hace referencia al elemento del DOM en el que ocurrió el evento. Puede ser útil para identificar el elemento específico que desencadenó el evento.
- **event.currentTarget**: Es similar a **event.target**, pero hace referencia al elemento en el que se definió el manejador de eventos. En la mayoría de los casos, **event.currentTarget** y **event.target** serán iguales, pero pueden diferir en situaciones donde se usan eventos delegados.
- **event.preventDefault()**: Una función que se puede llamar para evitar el comportamiento predeterminado del evento, como evitar que un enlace cambie de página o que un formulario se envíe.
- **event.stopPropagation()**: Una función que se puede llamar para detener la propagación del evento a elementos superiores. Esto evita que el evento se propague a través de la jerarquía de elementos en el DOM.
- **event.keyCode** o **event.key**: Estos campos contienen información sobre la tecla que se presionó en eventos de teclado, lo cual puede ser útil para realizar acciones específicas en respuesta a una tecla en particular.

Estos son solo algunos ejemplos de la información relevante que se puede encontrar en el objeto **event**. La disponibilidad y los detalles específicos pueden variar según el tipo de evento y el contexto en el que se utiliza.

Código de la clase

src > components > TodoButton.js

```

import "../css/TodoButton.css";

function TodoButton() {
  return (
    <button
      className="add"
      onClick={(event) => { 🗑️
        console.log(event);
        console.log(event.target);
        //<button class="add"></button>
      }}
    ></button>
  );
}

export { TodoButton };

```

src > components > TodoSearch.js

```

import "../css/TodoSearch.css";

function TodoSearch() {
  return (
    <input
      className="search"
      placeholder="Search..."
      onChange={(event) => { 🗑️
        console.log("search");
        console.log(event);
        console.log(event.target);
        console.log(event.target.value);
        // h hi
      }}
    />
  );
}

export { TodoSearch };

```

❖ Dato: Eliminamos algunas cosas del archivo `manifest.json`, ya que la consola mostraba errores por usar un icono de React, el cual ya había eliminado previamente.

```

"icons": [
{
  "src": "favicon.ico",
  "sizes": "64x64 32x32 24x24 16x16",
  "type": "image/x-icon"
}
]

```

```

},
{
  "src": "react192.png",
  "type": "image/png",
  "sizes": "192x192"
},
{
  "src": "react512.png",
  "type": "image/png",
  "sizes": "512x512"
}
],

```

7. ¿Qué es el estado?

En React, los estados son objetos que contienen datos y representan la información que puede cambiar durante el ciclo de vida de un componente. Los estados son utilizados para almacenar y controlar la información dinámica en una aplicación React. Cuando un estado cambia, React se encarga de actualizar automáticamente la interfaz de usuario para reflejar ese cambio.

Aquí hay algunos conceptos clave sobre los estados en React:

1. Declaración del estado: En un componente de React, puedes declarar un estado utilizando el hook `useState`. El hook `useState` retorna un array con dos elementos: el valor actual del estado y una función que se utiliza para actualizar ese estado. Por ejemplo:

```

import React, { useState } from "react";

function MyComponent() {
  const [count, setCount] = useState(0);

  // Resto del código del componente
}

```

En este ejemplo, `count` es el estado y `setCount` es la función que se utiliza para actualizar el estado.

2. Actualización del estado: Para actualizar el estado, debes llamar a la función que se utiliza para actualizarlo, en este caso `setCount`. Puedes llamar a esta función en respuesta a eventos o en cualquier otro lugar donde deseas actualizar el estado. Por ejemplo:

```

function handleButtonClick() {
  setCount(count + 1);
}

```

En este caso, `setCount` se llama con el nuevo valor del estado, en este caso, `count + 1`. Al llamar a `setCount`, React se encargará de actualizar el estado y volverá a renderizar el componente con el nuevo

valor.

3. Renderizado condicional: Puedes utilizar el estado para controlar el renderizado condicional de elementos en tu interfaz de usuario. Por ejemplo, puedes mostrar u ocultar un elemento basado en el valor de un estado. Aquí hay un ejemplo:

```
function MyComponent() {  
  const [isVisible, setIsVisible] = useState(true);  
  
  return (  
    <div>  
      {isVisible && (  
        <p>Este elemento se muestra si isVisible es verdadero</p>  
      )}  
      <button onClick={() => setIsVisible(!isVisible)}>  
        Mostrar/Ocultar  
      </button>  
    </div>  
  );  
}
```

En este ejemplo, el elemento `<p>` se muestra si el estado `isVisible` es `true`. Al hacer clic en el botón, se llama a la función `setIsVisible` para cambiar el valor del estado entre `true` y `false`, lo que provoca la aparición o desaparición del elemento `<p>` en la interfaz de usuario.

4. Uso de múltiples estados: Puedes utilizar múltiples estados en un componente para manejar diferentes datos o características. Simplemente declara diferentes estados utilizando `useState` según tus necesidades. Por ejemplo:

```
function MyComponent() {  
  const [name, setName] = useState("");  
  const [age, setAge] = useState(0);  
  
  // Resto del código del componente  
}
```

En este caso, `name` y `age` son dos estados diferentes que se pueden actualizar y utilizar por separado.

Los estados en React son fundamentales para crear componentes interactivos y dinámicos. Al utilizar estados, puedes controlar y actualizar los datos en tu aplicación de manera eficiente, lo que permite que la interfaz de usuario reaccione de manera automática y refleje los cambios en los datos.

Aquí tienes un ejemplo básico de cómo utilizar estados en React para crear un contador simple:

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

```

function Counter() {
  const [count, setCount] = useState(0);

  const increment = () => {
    setCount(count + 1);
  };

  const decrement = () => {
    setCount(count - 1);
  };

  return (
    <div>
      <h1>Contador: {count}</h1>
      <button onClick={increment}>Incrementar</button>
      <button onClick={decrement}>Decrementar</button>
    </div>
  );
}

export default Counter;

```

En este ejemplo, utilizamos el hook `useState` para declarar un estado llamado `count` con un valor inicial de `0`. Luego, definimos dos funciones `increment` y `decrement` que actualizan el estado `count` aumentándolo o disminuyéndolo en 1 respectivamente.

Dentro del componente, mostramos el valor actual del contador utilizando la variable `count` entre llaves `{count}`. Al hacer clic en los botones "Incrementar" o "Decrementar", se llama a las funciones correspondientes y se actualiza el estado `count`, lo que provoca que React vuelva a renderizar el componente con el nuevo valor del contador.

Atributos `value` y `placeholder`

La diferencia principal entre los atributos `value` y `placeholder` en un elemento `input` en HTML es la siguiente:

- **`value`**: El atributo `value` especifica el valor inicial o el valor actual de un campo de entrada. Muestra un texto predeterminado dentro del campo de entrada cuando se carga la página. El usuario puede editar o modificar este valor antes de enviarlo. Si el usuario no modifica el valor, se enviará el valor predeterminado establecido en el atributo `value`. Ejemplo: `<input type="text" value="Ejemplo de valor predeterminado">`.
- **`placeholder`**: El atributo `placeholder` se utiliza para proporcionar una sugerencia o una pista sobre el formato o el tipo de datos que se espera en el campo de entrada. Es un texto de marcador de posición que se muestra en el campo de entrada antes de que el usuario escriba algo. No se envía junto con el formulario al enviarlo, y desaparece tan pronto como el usuario comienza a escribir en el campo. El objetivo principal del atributo `placeholder` es brindar orientación al usuario sobre qué tipo de información debe ingresar en el campo. Ejemplo: `<input type="text" placeholder="Ingrese su nombre">`.

Código de la clase

src > components > TodoSearch.js

```
import React from "react"; ↪@@
import "../css/TodoSearch.css";

function TodoSearch() {
  const [searchValue, setSearchValue] = React.useState(""); ↪@@

  console.log('Users search ToDos from ' + searchValue);

  return (
    <input
      placeholder="Search..."
      className="search"
      value={searchValue} ↪@@
      onChange={(event) => {
        setSearchValue(event.target.value); ↪@@
      }}
    />
  );
}

export { TodoSearch };
```

[Curso de React.js: Manejo Profesional del Estado](#)

8. Contando TODOs

En React, la comunicación entre un componente hijo y un componente padre se puede lograr mediante el uso de estados y funciones de devolución de llamada (callbacks).

Aquí tienes los pasos básicos para lograr la comunicación entre un componente hijo y un componente padre:

1. En el componente padre, define un estado y una función de devolución de llamada que se pasará al componente hijo como una prop.

```
import React, { useState } from "react";
import ChildComponent from "./ChildComponent";

function ParentComponent() {
  const [childData, setChildData] = useState("");

  const handleChildData = (data) => {
    setChildData(data);
  };
}
```

```

    return (
      <div>
        <ChildComponent onChildData={handleChildData} />
        <p>Data from child: {childData}</p>
      </div>
    );
}

export default ParentComponent;

```

2. En el componente hijo, define una función que se activará cuando ocurra algún evento o acción en el componente hijo. Luego, llama a la función de devolución de llamada pasada desde el componente padre, pasando los datos relevantes como argumento.

```

import React from "react";

function ChildComponent({ onChildData }) {
  const handleClick = () => {
    const data = "Hello from child!";
    onChildData(data);
  };

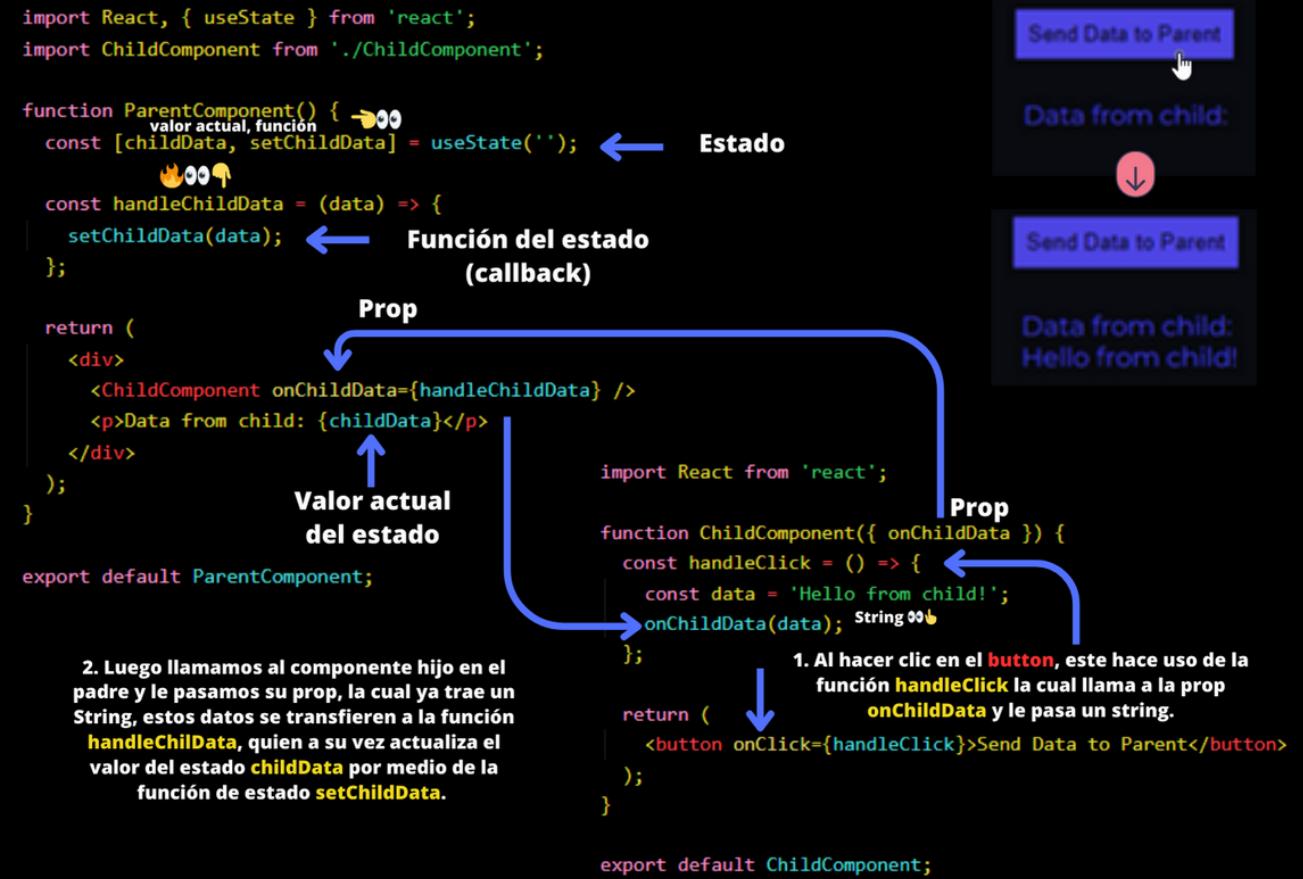
  return <button onClick={handleClick}>Send Data to Parent</button>;
}

export default ChildComponent;

```

En este ejemplo, el componente hijo (`ChildComponent`) tiene un botón que, cuando se hace clic, activa la función `handleClick`. Dentro de esta función, se crea una variable `data` que contiene los datos que deseas enviar al componente padre. Luego, se llama a la función de devolución de llamada `onChildData` pasada como prop desde el componente padre, pasando `data` como argumento.

En el componente padre (`ParentComponent`), la función `handleChildData` se ejecuta cuando se activa la función de devolución de llamada desde el componente hijo. Dentro de esta función, se actualiza el estado `childData` con los datos recibidos. El valor actualizado del estado se muestra en el componente padre.



De esta manera, los datos se comunican desde el componente hijo hasta el componente padre mediante el uso de estados y funciones de devolución de llamada.

Estados derivados

En React, los estados derivados se refieren a los estados que se calculan o derivan a partir de otros estados existentes. Estos estados derivados no se almacenan como datos en sí mismos, sino que se calculan dinámicamente según los cambios en los estados base.

Un ejemplo común de un estado derivado en React es calcular el total de una lista de elementos.

Supongamos que tienes una lista de números y quieres calcular la suma de todos ellos. Puedes almacenar la lista de números en un estado base y luego calcular el total como un estado derivado.

Aquí tienes un ejemplo de cómo podrías implementar esto en un componente de React:

```

import React, { useState } from "react";

const NumberList = () => {
  const [numbers, setNumbers] = useState([1, 2, 3, 4, 5]);

  // Cálculo del estado derivado
  const total = numbers.reduce(
    (accumulator, current) => accumulator + current,
    0
  );
}

```

```

    return (
      <div>
        <ul>
          {numbers.map((number, index) => (
            <li key={index}>{number}</li>
          )))
        </ul>
        <p>Total: {total}</p>
      </div>
    );
  };

export default NumberList;

```

En este ejemplo, el estado base es `numbers`, que es un arreglo de números. El estado derivado `total` se calcula utilizando el método `reduce()` en el arreglo `numbers`. Cada vez que se actualizan los números, el total se recalcula automáticamente.

Los estados derivados son útiles cuando necesitas realizar cálculos o transformaciones basadas en los estados existentes. Ayudan a mantener la lógica de tu componente más modular y fácil de entender, ya que puedes calcular valores derivados según sea necesario en lugar de almacenarlos explícitamente como estados separados.

Operador `!!`

El operador `!!` en JavaScript se utiliza para convertir un valor en su equivalente booleano. Esencialmente, se utiliza para obtener el valor booleano de una expresión, independientemente del tipo de datos original.

Cuando se aplica el operador `!!` a un valor, se realiza una conversión implícita a `boolean` siguiendo las reglas de conversión del lenguaje. El resultado final será `true` si el valor original se considera "verdadero" o `false` si se considera "falso".

La conversión a booleano sigue las siguientes reglas:

- Valores considerados "falsos": `false`, `0`, `""` (cadena vacía), `null`, `undefined`, `NaN`.
- Valores considerados "verdaderos": cualquier valor que no sea "falso".

Aquí hay algunos ejemplos para ilustrar el uso del operador `!!`:

```

console.log (!!0); // false
console.log (!!1); // true
console.log (!!"Hello"); // true
console.log (!! ""); // false
console.log (!!null); // false
console.log (!!undefined); // false
console.log (!!NaN); // false

```

En relación con los tipos de datos booleanos en JavaScript, el operador `!!` se utiliza a menudo para garantizar que una variable o una expresión se evalúe como un valor booleano explícito. Esto puede ser útil en situaciones donde necesitas asegurarte de que un valor se interprete como `true` o `false`, independientemente de su tipo original.

Por ejemplo, supongamos que tienes una variable `x` que puede tener diferentes tipos de datos y quieres asegurarte de que se evalúe como un booleano. Puedes usar `!!` para lograrlo:

```
var x = "Hello";
var booleanValue = !!x; // true

console.log(booleanValue); // true
```

En este caso, `!!x` convierte el valor de `x` en un booleano explícito, y el resultado es `true`, ya que cualquier valor que no sea "falso" se considera "verdadero" en JavaScript.

Constructor `Boolean()`

El constructor `Boolean()` puede ser utilizado para crear un objeto booleano a partir de un valor. Si pasas un valor falsy, como `0`, `null`, `undefined`, `NaN` o una cadena vacía, el objeto booleano será falso; de lo contrario, será true. Puedes usar el operador de coerción para obtener el valor primitivo booleano. Por ejemplo:

```
var dato = 0;
var booleano = Boolean(dato); // Equivalente booleano usando el constructor
Boolean()
console.log(booleano); // false

booleano = new Boolean(dato).valueOf(); // Obtener el valor primitivo booleano
del objeto Boolean
console.log(booleano); // false
```

Es importante tener en cuenta que el uso del operador `"!!"`, es común en JavaScript debido a su concisión y claridad. Sin embargo, la alternativa mencionada proporciona una opción adicional para obtener el equivalente booleano de un dato.

Métodos `filter` y `find`

El método `filter` se utiliza para filtrar elementos de una matriz (array) según un criterio específico y crear una nueva matriz con los elementos que cumplen con ese criterio. Toma una función de devolución de llamada (callback) como argumento, que se ejecuta para cada elemento de la matriz y devuelve `true` si se debe incluir en la nueva matriz filtrada, o `false` si no se debe incluir.

Aquí tienes un ejemplo de cómo usar `filter`:

```

const numbers = [1, 2, 3, 4, 5, 6];

const evenNumbers = numbers.filter(function (number) {
    return number % 2 === 0;
});

console.log(evenNumbers); // Resultado: [2, 4, 6]

```

En el ejemplo anterior, se define una matriz llamada `numbers` que contiene números del 1 al 6. Luego, se utiliza el método `filter` para crear una nueva matriz llamada `evenNumbers`, que solo contiene los números pares de la matriz original.

El método `find`, por otro lado, se utiliza para encontrar el primer elemento que cumple con un criterio específico en una matriz. Al igual que `filter`, también toma una función de devolución de llamada como argumento. Esta función se ejecuta para cada elemento de la matriz y devuelve `true` si se encuentra el elemento deseado, o `false` si no se encuentra. El método `find` devuelve el primer elemento que cumple con el criterio o `undefined` si no se encuentra ningún elemento.

Aquí tienes un ejemplo de cómo usar `find`:

```

const fruits = ["apple", "banana", "orange", "mango"];

const foundFruit = fruits.find(function (fruit) {
    return fruit === "orange";
});

console.log(foundFruit); // Resultado: 'orange'

```

En el ejemplo anterior, se define una matriz llamada `fruits` que contiene diferentes frutas. Luego, se utiliza el método `find` para encontrar la primera fruta que sea igual a '`orange`'. El método devuelve '`orange`', que es el primer elemento que cumple con el criterio.

Código de la clase

`src > App.js`

```

import React from "react";
import { TodoCounter } from "./components/TodoCounter";
import { TodoSearch } from "./components/TodoSearch";
import { TodoList } from "./components/TodoList";
import { TodoItem } from "./components/TodoItem";
import { TodoButton } from "./components/TodoButton";

const defaultTodos = [
    { text: "Lorem lorem", completed: true },
    { text: "Don't cry", completed: false },

```

```

    { text: "Lorem ipsum", completed: false },
    { text: "Take care", completed: false },
    { text: "Loremlorem", completed: true },
];

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;

    console.log("Users search Todos from " + searchValue);

    return (
        <>
            <TodoCounter completed={completedTodos} total={totalTodos} />
            <TodoSearch
                searchValue={searchValue}
                setSearchValue={setSearchValue}
            />

            <TodoList>
                {defaultTodos.map((todo) => (
                    <TodoItem
                        key={todo.text}
                        text={todo.text}
                        completed={todo.completed}
                    />
                ))}
            </TodoList>

            <TodoButton />
        </>
    );
}

export default App;

```

src > components > TodoSearch.js

```

import React from "react";
import "../css/TodoSearch.css";

function TodoSearch({ searchValue, setSearchValue }) {
    return (
        <input
            placeholder="Search..."
            className="search"
            value={searchValue}
            onChange={(event) => {

```

```

        setSearchValue(event.target.value);
    }
  />
);
}

export { TodoSearch };

```

9. Buscando TODOs

Método `includes()`

En JavaScript o React, `include` es un método que se utiliza para verificar si un elemento específico está presente en un arreglo o cadena de texto. Este método devuelve un valor booleano (`true` o `false`) según si el elemento se encuentra o no en la colección.

El método `include` se utiliza principalmente para realizar comprobaciones de pertenencia en arrays y cadenas de texto. A continuación, te mostraré ejemplos de cómo se puede utilizar en ambos casos:

- Con arrays:

```

const numbers = [1, 2, 3, 4, 5];

console.log(numbers.includes(3)); // true
console.log(numbers.includes(6)); // false

```

En este ejemplo, el método `includes` se utiliza para verificar si el número 3 está presente en el array `numbers`. Como el número 3 está en el array, el método devuelve `true`. Luego, se verifica si el número 6 está presente en el array, pero como no existe, el método devuelve `false`.

- Con cadenas de texto:

```

const message = "Hello, world!";

console.log(message.includes("Hello")); // true
console.log(message.includes("foo")); // false

```

Aquí, se utiliza `includes` para verificar si la cadena de texto "Hello" está presente en el mensaje. Como la cadena "Hello" se encuentra en el mensaje, el método devuelve `true`. Sin embargo, al verificar la presencia de la cadena "foo", que no está en el mensaje, el método devuelve `false`.

- Con cadenas de texto vacías:

```

const vacio = ""; // Recuerda esto, es muy importante

```

```
console.log(vacio.includes("")); // true
```

El método `includes` es útil para realizar acciones condicionales basadas en la presencia o ausencia de un elemento en un arreglo o cadena de texto. Puedes utilizarlo para realizar búsquedas simples y determinar si un valor específico se encuentra en una colección de datos.

Métodos `toLowerCase()` y `toLocaleLowerCase()`

En JavaScript y React, tanto `toLowerCase()` como `toLocaleLowerCase()` son métodos que se utilizan para convertir una cadena de texto a minúsculas. Sin embargo, existen diferencias sutiles entre ellos.

El método `toLowerCase()` convierte todos los caracteres de una cadena de texto a minúsculas según las reglas de conversión de minúsculas de Unicode. A continuación, te muestro un ejemplo de cómo se puede utilizar:

```
const message = "Hello, World!";  
  
console.log(message.toLowerCase()); // "hello, world!"
```

En este ejemplo, el método `toLowerCase()` se aplica a la cadena de texto "Hello, World!" y devuelve la cadena en minúsculas "hello, world!".

Por otro lado, el método `toLocaleLowerCase()` también convierte una cadena de texto a minúsculas, pero lo hace utilizando las reglas de conversión de minúsculas específicas del idioma actual. Esto significa que el resultado puede variar según el idioma del entorno en el que se esté ejecutando el código. Aquí tienes un ejemplo:

```
const message = "Île-de-France";  
  
console.log(message.toLocaleLowerCase()); // "île-de-france" (dependiendo del  
idioma del entorno)
```

En este ejemplo, el método `toLocaleLowerCase()` se aplica a la cadena de texto "Île-de-France" y devuelve la cadena en minúsculas "île-de-france" utilizando las reglas de conversión de minúsculas específicas del idioma actual.

En resumen, `toLowerCase()` convierte una cadena de texto a minúsculas utilizando las reglas de conversión de Unicode, mientras que `toLocaleLowerCase()` lo hace utilizando las reglas de conversión de minúsculas específicas del idioma actual. La elección entre uno u otro depende del contexto y de las necesidades específicas de tu aplicación.

Código de la clase

src > App.js

```

import React from "react";
import { TodoCounter } from "./components/TodoCounter";
import { TodoSearch } from "./components/TodoSearch";
import { TodoList } from "./components/TodoList";
import { TodoItem } from "./components/TodoItem";
import { TodoButton } from "./components/TodoButton";

const defaultTodos = [
  { text: "Lorem lorem", completed: true },
  { text: "Don't cry", completed: false },
  { text: "Lorem ipsum", completed: false },
  { text: "Take care", completed: false },
  { text: "Loremlorem", completed: true },
];
;

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;

  const searchedTodos = todos.filter((todo) => {
    const todoText = todo.text.toLowerCase();
    const searchText = searchValue.toLowerCase();
    return todoText.includes(searchText); // "" vacío ⏪ ⓘ
  });

  console.log("Users search ToDos from " + searchValue);

  return (
    <>
      <TodoCounter completed={completedTodos} total={totalTodos} />
      <TodoSearch searchValue={searchValue} setSearchValue={setSearchValue} />

      <TodoList>
        {searchedTodos.map((todo) => ( ⏪ ⓘ
          <TodoItem
            key={todo.text}
            text={todo.text}
            completed={todo.completed}
          />
        ))}
      </TodoList>

      <TodoButton />
    </>
  );
}

export default App;

```

Dato importante

💡 ¿Por qué si `searchValue` está vacío "", devuelve todos los valores del array cuando se filtra?

```
const searchedTodos = todos.filter((todo) => {
  const todoText = todo.text.toLowerCase();
  const searchText = searchValue.toLowerCase();
  return todoText.includes(searchText);
});
```

Lo primero, estamos aplicando el método `includes()` de strings, es decir:

```
const todoText = todo.text.toLowerCase(); // string
const searchText = searchValue.toLowerCase(); // string
return todoText.includes(searchText); // booleano
// "texto".include("") true
```

Si aplicamos un `includes()` cuyo valor es vacío, él va a devolver un `TRUE`, por ejemplo:

```
console.log({
  letra: "A".includes(""), //True
  vacio: "".includes(""), //True
  nombre: "Ale".includes(""), //True
  nombreCompleto: "Ale Roses".includes(""), //True
  numero: "3".includes(""), //True
});
```

◆ Como resultado, cada elemento `todo` recorrido va a ser `True` y por ende el `filter()` aplicado va a devolver cada elemento del array.

[Explicación relevante sobre `include\(\)`](#)

Normalizar strings

En un ámbito profesional (dependiendo cada caso de uso), para un campo de buscar, podemos normalizar ambos strings (ToDos ingresados previamente y ToDo de búsqueda), ignorando mayúsculas, ignorando acentos, quitando espacios, en cualquier posición del string.

Método:

```
const normalizeString = (string) => {
  string = string || "";
  string = string.toLowerCase();
  // remover acentos
  string = string.normalize("NFD").replace(/[\u0300-\u036f]/g, ""); ⌂ 00
```

```
//Regex
    string = string.trim();
    return string;
};
```

Uso:

```
const filteredTodos = todos.filter((todo) => {
  let { text: normalizedTodo } = todo;
  normalizedTodo = normalizeString(normalizedTodo);
  let normalizedSearch = normalizeString(searchValue);

  return normalizedTodo.includes(normalizedSearch);
});
```

10. Completando y eliminando TODOs

Operador de propagación

El operador `[...array]` en JavaScript se conoce como el operador de propagación (spread operator). Se utiliza para descomponer o "desempaquetar" un elemento iterable, como un array o un objeto iterable, en elementos individuales. Esto permite copiar los elementos de un array u objeto iterable en otro array o en los argumentos de una función de una manera más conveniente.

Aquí tienes algunos ejemplos para comprender mejor cómo se usa el operador de propagación:

1. Copiar un array:

```
const originalArray = [1, 2, 3];
const newArray = [...originalArray];

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

En este ejemplo, el operador de propagación `[...originalArray]` descompone el array `originalArray` en elementos individuales y crea un nuevo array `newArray` que contiene los mismos elementos. Esto se conoce como una copia superficial (shallow copy) del array.

2. Unir arrays:

```
const array1 = [1, 2, 3];
const array2 = [4, 5, 6];
const mergedArray = [...array1, ...array2];

console.log(mergedArray); // Resultado: [1, 2, 3, 4, 5, 6]
```

En este ejemplo, el operador de propagación se utiliza para combinar los elementos de `array1` y `array2` en un solo array `mergedArray`. Al descomponer ambos arrays con `...array1` y `...array2`, se obtienen los elementos individuales y se combinan en un nuevo array.

3. Pasar argumentos a una función:

```
function sum(a, b, c) {  
    return a + b + c;  
}  
  
const numbers = [1, 2, 3];  
const result = sum(...numbers);  
  
console.log(result); // Resultado: 6
```

En este ejemplo, el operador de propagación se utiliza para pasar los elementos del array `numbers` como argumentos a la función `sum`. Al descomponer el array con `...numbers`, los elementos individuales se pasan como argumentos a la función, lo que permite realizar operaciones con ellos.

El operador de propagación `[...algo]` es una forma útil de trabajar con arrays y objetos iterables en JavaScript, ya sea para copiar, combinar o pasar elementos como argumentos. Te permite trabajar con los elementos individuales de manera más flexible y concisa.

Método `findIndex`

El método `findIndex` en JavaScript se utiliza para encontrar el índice del primer elemento en un array que cumple con un criterio determinado. Devuelve el índice del elemento encontrado, o -1 si no se encuentra ningún elemento que cumpla con el criterio.

La sintaxis general del método `findIndex` es la siguiente:

```
array.findIndex(callback( element[, index[, array]] )[, thisArg])
```

- `callback`: Una función de devolución de llamada que se ejecuta para cada elemento del array. Recibe hasta tres argumentos opcionales:
 - `element`: El elemento actual que se está procesando en el array.
 - `index` (opcional): El índice del elemento actual en el array.
 - `array` (opcional): El array en el que se está llamando a `findIndex`.
- `thisArg` (opcional): Un valor que se utiliza como `this` cuando se ejecuta la función de devolución de llamada.

Aquí tienes un ejemplo para comprender cómo se utiliza `findIndex`:

```
const numbers = [1, 2, 3, 4, 5];
```

```

const evenIndex = numbers.findIndex(function (number, index) {
    return number % 2 === 0 && index % 2 === 0;
});

console.log(evenIndex); // Resultado: -1 sin coincidencias

```

En este ejemplo, `findIndex` se utiliza para buscar el índice del primer número par que también tiene un índice par en el array `numbers`. La función de devolución de llamada comprueba si el número es par (`number % 2 === 0`) y si el índice es par (`index % 2 === 0`). En este caso, no hay coincidencias, por lo tanto, `evenIndex` se establece en -1.

Aquí tienes otro ejemplo que utiliza `findIndex`:

```

const numeros = [10, 20, 30, 40, 50];

const indice = numeros.findIndex((elemento) => elemento > 30);

console.log(indice); // Devuelve indice 3

```

En este ejemplo, el arreglo `numeros` contiene una serie de números. Utilizamos `findIndex` para encontrar el índice del primer elemento que sea mayor a 30. La función de callback `elemento > 30` devuelve `true` para el elemento `40`, y `findIndex` devuelve el índice correspondiente, que es 3.

Es importante tener en cuenta que `findIndex` finaliza tan pronto como encuentra un elemento que cumple con el criterio y devuelve su índice correspondiente. Si no se encuentra ningún elemento que cumpla con el criterio, se devuelve -1.

Método `splice()`

El método `splice` en JavaScript se utiliza para modificar el contenido de un array al eliminar, reemplazar o agregar elementos en posiciones específicas. Puede realizar cambios en el lugar (es decir, modificar el array original) y también devuelve un nuevo array que contiene los elementos eliminados.

La sintaxis general del método `splice` es la siguiente:

```
array.splice(start, deleteCount, item1, item2, ...);
```

- `start`: Un índice entero que especifica la posición en la que se inicia la modificación del array. Si es un número negativo, se cuenta desde el final del array. Si es mayor que la longitud del array, `splice` actuará al final del array.
- `deleteCount` (opcional): Un entero que indica el número de elementos que se deben eliminar a partir de la posición `start`. Si se omite o es 0, no se eliminarán elementos.
- `item1, item2, ...` (opcional): Elementos que se agregarán al array a partir de la posición `start`.

A continuación, te mostraré algunos ejemplos para comprender cómo se usa el método `splice`:

1. Eliminar elementos de un array:

```
const numbers = [1, 2, 3, 4, 5];
numbers.splice(2, 2);

console.log(numbers); // Resultado: [1, 2, 5]
```

En este ejemplo, `splice(2, 2)` elimina 2 elementos a partir del índice 2 en el array `numbers`. Como resultado, los elementos 3 y 4 son eliminados, y el array se modifica para contener `[1, 2, 5]`.

2. Reemplazar elementos en un array:

```
const fruits = ["apple", "banana", "orange", "mango"];
fruits.splice(1, 2, "grape", "kiwi");

console.log(fruits); // Resultado: ['apple', 'grape', 'kiwi', 'mango']
```

En este ejemplo, `splice(1, 2, 'grape', 'kiwi')` reemplaza 2 elementos a partir del índice 1 en `fruits` con los elementos `'grape'` y `'kiwi'`. Como resultado, los elementos `'banana'` y `'orange'` son reemplazados, y el array se modifica para contener `['apple', 'grape', 'kiwi', 'mango']`.

3. Agregar elementos a un array:

```
const colors = ["red", "blue", "green"];
colors.splice(1, 0, "yellow", "purple");

console.log(colors); // Resultado: ['red', 'yellow', 'purple', 'blue', 'green']
```

En este ejemplo, `splice(1, 0, 'yellow', 'purple')` agrega los elementos `'yellow'` y `'purple'` en el índice 1 de `colors`. Como `deleteCount` es 0, no se eliminan elementos. Los nuevos elementos se insertan en la posición especificada, y el array se modifica para contener `['red', 'yellow', 'purple', 'blue', 'green']`.

El método `splice` es una forma poderosa de modificar arrays en JavaScript al eliminar, reemplazar o agregar elementos en posiciones específicas. Te permite realizar cambios en el lugar y obtener los elementos eliminados en caso de necesitarlos.

El operador `delete`

El operador `delete` se utiliza para eliminar una propiedad de un objeto o un elemento de un arreglo.

La sintaxis general del operador `delete` es la siguiente:

```
delete objeto.propiedad; // Elimina una propiedad de un objeto
```

o

```
delete arreglo[indice]; // Elimina un elemento de un arreglo
```

Aquí tienes algunos ejemplos de cómo se utiliza el operador `delete`:

```
const persona = {
  nombre: "Juan",
  edad: 30,
  ciudad: "Madrid",
};

delete persona.edad; // Elimina la propiedad "edad" del objeto "persona"

console.log(persona); // Muestra: { nombre: "Juan", ciudad: "Madrid" }

const numeros = [10, 20, 30, 40, 50];

delete numeros[2]; // Elimina el elemento en el índice 2 del arreglo "numeros"

console.log(numeros); // Muestra: [10, 20, undefined, 40, 50]
```

Es importante tener en cuenta que el operador `delete` solo puede eliminar propiedades de objetos que sean configurables. Al intentar eliminar una propiedad no configurable o una variable declarada con `var`, `let` o `const`, el operador `delete` no tendrá ningún efecto y devolverá `false`. Además, el operador `delete` no puede eliminar variables o funciones declaradas con `var`, `let` o `const`.

Código de la clase

src > App.js

```
import React from "react";
import { TodoCounter } from "./components/TodoCounter";
import { TodoSearch } from "./components/TodoSearch";
import { TodoList } from "./components/TodoList";
import { TodoItem } from "./components/TodoItem";
import { TodoButton } from "./components/TodoButton";

const defaultTodos = [
  { text: "Lorem lorem", completed: true },
  { text: "Don't cry", completed: false },
  { text: "Lorem ipsum", completed: false },
  { text: "Take care", completed: false },
  { text: "Loremlorem", completed: true },
];
```

```

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;

  const searchedTodos = todos.filter((todo) => {
    const todoText = todo.text.toLowerCase();
    const searchText = searchValue.toLowerCase();
    return todoText.includes(searchText);
  });

  const completeTodo = (text) => {
    const newTodos = [...todos]; // nuevo array
    const todoIndex = newTodos.findIndex((todo) => todo.text == text);

    // newTodos[todoIndex].completed = true;
    // true = false / false = true
    newTodos[todoIndex].completed = !newTodos[todoIndex].completed;
    setTodos(newTodos); // Actualiza
  };

  const deleteTodo = (text) => {
    const newTodos = [...todos];
    const todoIndex = newTodos.findIndex((todo) => todo.text == text);

    newTodos.splice(todoIndex, 1);
    setTodos(newTodos); // Actualiza
  };

  return (
    <>
      <TodoCounter completed={completedTodos} total={totalTodos} />
      <TodoSearch searchValue={searchValue} setSearchValue={setSearchValue} />

      <TodoList>
        {searchedTodos.map((todo) => (
          <TodoItem
            key={todo.text}
            text={todo.text}
            completed={todo.completed}
            // Alimenta la función completeTodo con el texto
            onComplete={() => completeTodo(todo.text)}
            onDelete={() => deleteTodo(todo.text)}
          />
        )));
      </TodoList>

      <TodoButton />
    </>
  );
}

```

```
}

export default App;
```

src > components > TodoItem.js

```
import "../css/TodoItem.css";

function TodoItem(props) {
  return (
    <li>
      <span
        className={`check ${props.completed && "check--active"}`}
        onClick={props.onComplete}
      ></span>
      <p className={`${props.completed && "p--completed"}`}>
        {props.text}
      </p>
      <span className="delete" onClick={propsonDelete}></span>
    </li>
  );
}

export { TodoItem };
```

src > components > TodoItem.js

```
import "../css/TodoCounter.css";

function TodoCounter({ completed, total }) {
  return total == completed ? ( ✅)
    <h1 className="total">Completaste todos los ToDos</h1>
  ) : ( 🚫
    <h1>
      Has completado <span className="completed">{completed}</span> de{" "}
      <span className="total">{total}</span> ToDos
    </h1>
  );
}

export { TodoCounter };
```

11. Iconos en React: librerías y SVG

React Icons es una biblioteca de componentes de iconos populares que se pueden utilizar en aplicaciones React. Proporciona una forma conveniente de agregar iconos a tus componentes sin necesidad de descargar imágenes o utilizar fuentes de iconos externas.

Para comenzar a utilizar React Icons, primero debes instalarlo en tu proyecto. Puedes hacerlo a través de npm o yarn ejecutando el siguiente comando en la línea de comandos:

```
npm install react-icons
```

Una vez que hayas instalado React Icons, puedes importar los iconos individuales que deseas utilizar en tus componentes. Por ejemplo, si quieres utilizar el ícono de corazón de la biblioteca de FontAwesome, puedes importarlo de la siguiente manera:

```
import { FaHeart } from "react-icons/fa";

const MiComponente = () => {
  return (
    <div>
      <h1>Mi componente con ícono</h1>
      <FaHeart />
    </div>
  );
}
```

En este ejemplo, importamos el componente de ícono `FaHeart` de la biblioteca de FontAwesome. Luego, simplemente colocamos el componente `<FaHeart />` en el lugar donde deseamos que aparezca el ícono.

React Icons ofrece una amplia variedad de bibliotecas de íconos populares, como FontAwesome, Material Design Icons, Ionicons y muchas más. Puedes explorar la documentación de React Icons para obtener una lista completa de las bibliotecas de íconos compatibles y los íconos disponibles.

Recuerda que para utilizar los íconos, debes asegurarte de tener instalada la biblioteca de íconos correspondiente junto con React Icons. Además, puedes personalizar los íconos utilizando las propiedades y estilos de React según tus necesidades específicas.

React Icons

Código de la clase

src > components > CompleteIcon.js

```
import React from "react";
import { TodoIcon } from "./TodoIcon";

function CompleteIcon() {
  return <TodoIcon type="check" color="gray" />;
}

export { CompleteIcon };
```

src > components > DeleteIcon.js

```
import React from "react";
import { TodoIcon } from "./TodoIcon";

function DeleteIcon() {
    return <TodoIcon type="delete" color="red" />;
}

export { DeleteIcon };
```

src > components > TodoItem.js

```
import { CompleteIcon } from "./CompleteIcon";
import { DeleteIcon } from "./DeleteIcon";
import "../css/TodoItem.css";

function TodoItem(props) {
    return (
        <li>
            <CompleteIcon />
            <p className={`${props.completed && "p--completed"}`}>
                {props.text}
            </p>
            <DeleteIcon />
        </li>
    );
}

export { TodoItem };
```

src > components > TodoIcon.js

```
import { ReactComponent as CheckSvg } from "../svg/check.svg";
import { ReactComponent as DeleteSvg } from "../svg/delete.svg";

const iconTypes = {
    check: <CheckSvg />,
    delete: <DeleteSvg />,
};

function TodoIcon({ type }) {
    return <span className={`${type} check--active`}>{iconTypes[type]}</span>;
/*
    <span
        className={`check ${props.completed && "check--active"}`}
        onClick={props.onComplete}

```

```

        ></span>
      <span className="delete" onClick={props.onDelete}>
        </span>
      */
    }

export { TodoIcon };

```

src > css > TodoItem.css

```

li {
  list-style: none;
  background-color: #cbd5e1;
  width: 15rem;
  height: 3rem;
  border-radius: 5px;
  display: flex;
  justify-content: space-between;
  align-items: center;
  padding: 0 1rem 0 0.5rem;

  position: relative;
}

.check {
  /* background-image: url("../svg/check.svg");
  background-repeat: no-repeat;
  background-position: center;
  background-size: contain;
  width: 20px;
  height: 20px; */
  cursor: pointer;
}

.check--active {
  /* background-image: url("../svg/check-completed.svg"); */
}

.delete {
  /* background-image: url("../svg/delete.svg");
  background-repeat: no-repeat;
  background-position: center;
  background-size: contain;
  width: 15px;
  height: 15px; */
  cursor: pointer;

  position: absolute;
  right: 0;
  top: 0;
}

```

```

.delete:hover {
  /* background-image: url("../svg/delete-hover.svg"); */
}

p {
  width: 85%;
  height: auto;
  margin: 0 5px 0;
  color: #090b10;
}

.p--completed {
  text-decoration: line-through;
}

```

12. Iconos con colores dinámicos

Prop drilling

El prop drilling, también conocido como prop passing, es un patrón común en React donde los datos se pasan desde un componente principal a través de varios niveles de componentes secundarios para llegar al componente que realmente necesita esos datos. Esto ocurre cuando un componente necesita acceder a ciertos datos o funciones que se encuentran en un componente superior en la jerarquía de componentes.

En React, los datos se pasan de arriba hacia abajo a través de las props, que son propiedades inmutables. Sin embargo, cuando se tienen muchos niveles de componentes anidados y se necesita acceder a los datos en componentes más profundos, es necesario pasar las props a través de cada nivel de la jerarquía de componentes, incluso si los componentes intermedios no los utilizan directamente. Esto puede llevar a un código más complicado y difícil de mantener.

El prop drilling se considera una solución "inelegante" debido a que puede hacer que el código sea más difícil de leer y mantener. Además, si en algún momento se necesita agregar un nuevo componente en medio de la cadena de componentes, se tendría que actualizar la forma en que se pasan las props a través de todos los niveles anteriores.

Para evitar el prop drilling, se pueden utilizar otras técnicas como el uso de Context API de React o la implementación de un estado global con bibliotecas como Redux o MobX. Estas soluciones permiten acceder a los datos desde cualquier componente en la aplicación sin tener que pasar las props a través de cada nivel de componentes.

En resumen, el prop drilling es un patrón en React donde los datos se pasan desde un componente principal a través de varios niveles de componentes secundarios, lo cual puede complicar el código y dificultar el mantenimiento. Se recomienda explorar otras técnicas como el uso de Context API o la implementación de un estado global para evitar el prop drilling en situaciones donde sea necesario compartir datos entre componentes.

Render Props

El patrón Render Props en React es una técnica que permite compartir código y funcionalidad entre componentes utilizando una prop especial llamada "render prop". En esencia, un componente con Render Props acepta una función como prop y utiliza esa función para renderizar su contenido.

El concepto clave del patrón Render Props es que un componente proporciona una función a otro componente a través de una prop, y el componente receptor puede invocar esa función y utilizar el resultado para renderizar su propio contenido. Esto permite la reutilización de lógica y comportamiento entre componentes de manera flexible.

Aquí hay un ejemplo básico para ilustrar cómo funciona el patrón Render Props:

```
// Componente con Render Props
class RenderPropComponent extends React.Component {
  render() {
    // Llama a la función prop y pasa un valor como argumento
    return this.props.render("Hola desde el Render Prop");
  }
}

// Componente que utiliza el Render Prop
class App extends React.Component {
  render() {
    return (
      <div>
        <h1>App</h1>
        <RenderPropComponent render={(message) => <p>{message}</p>} />
      </div>
    );
  }
}
```

En este ejemplo, el componente `RenderPropComponent` acepta una prop llamada `render` que es una función. Luego, invoca esa función pasando un mensaje como argumento. El componente `App` utiliza `RenderPropComponent` y pasa una función como `render` prop que renderiza un elemento `p` con el mensaje recibido.

El patrón Render Props es útil cuando se desea compartir lógica o comportamiento complejo entre componentes sin tener que depender de la herencia de componentes o de bibliotecas externas. Permite una mayor flexibilidad y reutilización de código al permitir que los componentes consumidores controlen cómo se renderiza el contenido proporcionado por el componente con Render Props.

Cabe destacar que el patrón Render Props puede ser combinado con otros patrones y técnicas de React, como el uso de hooks o context, para crear componentes más poderosos y flexibles.

[Render Props - documentación](#)

[create-react-app](#)

La herramienta "create-react-app" es una utilidad de línea de comandos (CLI) que facilita la creación de aplicaciones web de React. Fue desarrollada por Facebook y proporciona una configuración predeterminada y una estructura de directorios predefinida para comenzar a desarrollar rápidamente con React.

Create React App configura automáticamente un entorno de desarrollo moderno con una configuración optimizada para crear aplicaciones React. Maneja tareas como la configuración de Webpack, Babel y otros paquetes necesarios para la compilación y ejecución de la aplicación. También incluye un servidor de desarrollo para facilitar la visualización y la recarga en vivo durante el desarrollo.

Al utilizar la herramienta "create-react-app", puedes generar rápidamente una base sólida para tu aplicación React sin tener que preocuparte por la configuración inicial. Esto te permite concentrarte en escribir código y desarrollar tu aplicación de manera más eficiente.

Para utilizar "create-react-app", necesitarás tener Node.js instalado en tu computadora. Puedes instalarlo utilizando el gestor de paquetes npm (Node Package Manager). Una vez que Node.js esté instalado, puedes utilizar el siguiente comando en tu terminal para crear un nuevo proyecto de React:

```
npx create-react-app nombre-del-proyecto
```

Esto creará un nuevo directorio llamado "nombre-del-proyecto" con la estructura y configuración necesarias para comenzar a desarrollar tu aplicación React. A partir de ahí, puedes utilizar otros comandos proporcionados por "create-react-app" para iniciar el servidor de desarrollo, generar una versión optimizada para producción y realizar otras tareas comunes relacionadas con el desarrollo de React.

En resumen, "create-react-app" es una herramienta que simplifica la configuración inicial y el entorno de desarrollo para crear aplicaciones web con React, permitiéndote comenzar a desarrollar rápidamente sin tener que preocuparte por la configuración complicada.

Empaquetamiento de CRA

El empaquetamiento de CRA (Create React App) se refiere al proceso de generar los archivos finales optimizados y listos para producción de una aplicación creada con la herramienta "create-react-app".

Cuando desarrollas una aplicación de React con CRA, normalmente trabajas con archivos JavaScript y otros recursos (como hojas de estilo, imágenes, fuentes, etc.) de manera modular y separados en diferentes módulos y componentes. Durante el desarrollo, CRA utiliza un servidor de desarrollo interno que se encarga de compilar y empaquetar estos archivos de manera eficiente para que puedan ser ejecutados en el navegador.

Sin embargo, cuando estás listo para llevar tu aplicación a producción, necesitas generar una versión optimizada y lista para ser distribuida en un servidor web. Para eso, CRA proporciona un comando llamado "npm run build" que realiza el empaquetamiento de la aplicación.

Cuando ejecutas "npm run build", CRA realiza una serie de tareas, incluyendo:

1. Combinación y minificación de archivos: Los diferentes archivos JavaScript y recursos de la aplicación se combinan en un solo archivo JavaScript (normalmente llamado "bundle.js"). Además,

se realiza la minificación de este archivo, lo que implica eliminar espacios en blanco, comentarios y reducir el tamaño del código para mejorar la velocidad de carga de la aplicación.

2. Optimización de recursos: Las imágenes, fuentes y otros recursos utilizados en la aplicación se optimizan para reducir su tamaño sin comprometer significativamente su calidad. Esto ayuda a mejorar la velocidad de carga de la aplicación y a reducir el consumo de ancho de banda.
3. Generación de archivos estáticos: CRA genera una serie de archivos estáticos que contienen la aplicación empaquetada y lista para ser servida por un servidor web. Estos archivos incluyen el archivo HTML principal, el archivo JavaScript empaquetado, las hojas de estilo y otros recursos necesarios.

Una vez que el proceso de empaquetamiento se completa, obtendrás una carpeta llamada "build" que contiene todos los archivos necesarios para desplegar tu aplicación en un servidor web. Puedes tomar estos archivos y subirlos a un servidor web estático, un servicio de alojamiento o cualquier otra plataforma de tu elección para que tu aplicación esté disponible en línea.

En resumen, el empaquetamiento de CRA es el proceso de generar los archivos finales optimizados y listos para producción de una aplicación de React creada con "create-react-app". Este proceso combina, minifica y optimiza los archivos de la aplicación, generando una versión lista para ser desplegada en un servidor web.

Entendamos lo que se hizo...

Antes de colocar el código dejaré algunos resúmenes para entender mejor lo que se hizo en clase:

Explicación 01

1. Debemos partir desde `App.js` que es el primer lugar en el cual enviamos una función encapsulada dentro de una `prop` a cada uno de los componentes `TodoItem` que se crean.

```
<TodoList>
  {searchedTodos.map((todo) => (
    <TodoItem
      key={todo.text}
      text={todo.text}
      completed={todo.completed}
      // Pasar una función a un componente sin ejecutarla inmediatamente
      ♦♦ onComplete={() => completeTodo(todo.text)} ✎
      ♦♦ onDelete={() => deleteTodo(todo.text)} ✎
    />
  ))}
</TodoList>
```

Si recordamos la función `completeTodo` compara el texto del ToDo renderizado en pantalla con una nueva lista y si ambos son iguales debe cambiar el `completed` a `true` para luego ser actualizado con la función `setTodos` dentro del estado `React.useState`. Algo similar pasa con la función `deleteTodo` la

única diferencia es que esta compara los textos para saber cuál ToDo debe eliminar e inmediatamente actualizar el estado.

2. Como se aprecia, estamos pasando las funciones mencionadas anteriormente en las props `onComplete` y `onDelete` del componente `TodoItem`. Luego en nuestro componente `TodoItem` reemplazamos los `span` por dos nuevos componentes `<CompleteIcon onComplete={props.onComplete} />` y `<DeleteIcon onDelete={propsonDelete} />` las que recibirán las funciones anteriores en dos `props` que volvemos a llamar `onComplete` y `onDelete`.

```
function TodoItem(props) {
  return (
    <li>
      <CompleteIcon completed={props.completed} onComplete={props.onComplete} />
      <p className={`${props.completed && "p--completed"}`}>{props.text}</p>
      <DeleteIcon onDelete={propsonDelete} />
    </li>
  );
}
```

3. Las recibiremos en nuestros componentes `CompleteIcon` y `DeleteIcon` respectivamente, aquí también recibimos la propiedad `onClick` que viene desde el componente `TodoIcon`.

```
function CompleteIcon({ completed, onComplete }) {
  return (
    <TodoIcon
      type="check"
      color={completed ? "#4CAF50" : "#4F46E5"}
      onClick={onComplete}
    />
  );
}
```

```
function DeleteIcon({ onDelete }) {
  return <TodoIcon type="delete" color="#4F46E5" onClick={onDelete} />;
}
```

```

App.js
src > App.js > default
return (
  <>
    <TodoCounter completed={completedTodos} total={totalTodos} />
    <TodoSearch searchValue={searchValue} setSearchValue={setSearchValue} />

    <TodoList>
      {searchedTodos.map((todo) => (
        <TodoItem
          key={todo.text}
          text={todo.text}
          completed={todo.completed}
          // Pasar una función a un componente sin ejecutarla
          // inmediatamente
          onComplete={() => completeTodo(todo.text)}
          onDelete={() => deleteTodo(todo.text)}
        />
      ))}
    </TodoList>
  ...
)

DeleteIcon.js
src > components > DeleteIcon.js ...
import React from "react";
import { TodoIcon } from "./TodoIcon";

function DeleteIcon({ onDelete }) {
  return <TodoIcon type="delete" color="#4F46E5" onClick={onDelete}>;
}

export { DeleteIcon };

```

```

TodoItem.js
src > components > TodoItem.js > ...
import { CompleteIcon } from "./CompleteIcon";
import { DeleteIcon } from "./DeleteIcon";
import "../css/TodoItem.css";

function TodoItem(props) {
  return (
    <li>
      <CompleteIcon completed={props.completed}>
        <onComplete={props.onComplete}>
          <p className={`${props.completed ? "p-completed" : ""}`}>{props.text}</p>
        </onComplete>
      </CompleteIcon>
      <DeleteIcon onDelete={props.onDelete}>
        <onDelete={props.onDelete}>
      </DeleteIcon>
    </li>
  );
}

export { TodoItem };

CompleteIcon.js
src > components > CompleteIcon.js ...
import React from "react";
import { TodoIcon } from "./TodoIcon";

function CompleteIcon({ completed, onComplete }) {
  return (
    <TodoIcon
      type="check"
      color={completed ? "#4CAF50" : "#4F46E5"}
      onClick={onComplete}
    />
  );
}

```

The diagram illustrates the prop flow between four files: App.js, TodoItem.js, TodoIcon.js, and DeleteIcon.js.
 - In App.js, the 'TodoItem' component is rendered with 'onComplete' and 'onDelete' props. A pink arrow points from the 'TodoItem' component in App.js to its definition in TodoItem.js. Another pink arrow points from the 'DeleteIcon' component in App.js to its definition in DeleteIcon.js.
 - In TodoItem.js, the 'DeleteIcon' component receives the 'onDelete' prop. A blue arrow points from the 'DeleteIcon' component in TodoItem.js back to its definition in DeleteIcon.js.
 - In TodoIcon.js, the 'DeleteIcon' component receives the 'onDelete' prop. A blue arrow points from the 'DeleteIcon' component in TodoIcon.js back to its definition in DeleteIcon.js.
 - In DeleteIcon.js, the 'DeleteIcon' component is defined with an 'onClick' prop. A blue arrow points from the 'DeleteIcon' component in DeleteIcon.js to its definition in TodoIcon.js.

Finalmente, la propiedad `onClick` también la enviamos a nuestro componente `TodoIcon` donde se encuentra el elemento `` y es allí en donde por fin ocurre el evento `onClick`.

```

function TodoIcon({ type, color, onClick }) {
  return (
    /* type puede ser check o delete */
    <span className={`${
      type
    }`} onClick={onClick}>
      {iconTypes[type](color)}
    </span>
  );
}

```

Comentario

Explicación 02

Lógica para renderizar SVG's de forma dinámica

Presentamos la siguiente situación:

Tenemos un componente llamado `TodoItem`, el cual renderiza cada uno de los elementos de nuestro listado de tareas a completar. Cada ítem contiene 3 elementos:

- Un botón de completado que contiene un ícono ✓
- El texto de la tarea
- Un botón para eliminar la tarea que también contiene un ícono ✗

Para insertar esos íconos se pueden usar diferentes métodos como:

- Emojis usando un plugin de Visual Studio Code o usando `Windows + .` para insertarlos directamente en el código.
- Librerías de íconos para React.
- Importando los SVG de forma dinámica como componentes de React. ♦♦

En este caso seguiremos el tercer camino, en primer lugar, vamos a añadir diferentes archivos en nuestra carpeta `src`:

- Los SVG llamados en este caso:
 - `check.svg`
 - `delete.svg`
- Un archivo JS para cada ícono a renderizar:
 - `CompleteIcon.js`
 - `DeleteIcon.js`
- Un archivo para contener la lógica de importación de los SVG's para todos los íconos con su respectivo CSS:
 - `TodoIcon.js`
 - `TodoIcon.css` ♦♦

Esta estructura de carpetas y archivos se vería así:

```
├── package-lock.json
├── package.json
└── public
    ├── index.html
    ├── manifest.json
    └── robots.txt
└── src
    ├── App.js
    └── components
        ├── CompleteIcon.js ↗ ①
        ├── DeleteIcon.js ↗ ①
        ├── TodoButton.js
        ├── TodoCounter.js
        ├── TodoIcon.js ↗ ①
        ├── TodoItem.js
        ├── TodoList.js
        └── TodoSearch.js
    └── css
        ├── TodoButton.css
        ├── TodoCounter.css
        ├── TodoIcon.css ↗ ①
        ├── TodoItem.css
        ├── TodoList.css
        └── TodoSearch.css
    └── index.css
    └── index.js
    └── svg
```

```

├── add.svg
├── check.svg
└── delete.svg
└── search.svg

```

La serie de pasos sería la siguiente:

1. Archivos: `TodoIcon.js` y `TodoIcon.css`.

```

// Importamos los svg
import { ReactComponent as CheckSvg } from "../svg/check.svg";
import { ReactComponent as DeleteSvg } from "../svg/delete.svg";
import "../css/TodoIcon.css"; // Estilos

// Object = { key: value }
const iconTypes = {
  check: (color) => <CheckSvg fill={color} />,
  delete: (color) => (
    <DeleteSvg
      className="icon"
      fill="#4F46E5"
      stroke="#4F46E5"
      strokeWidth="0.1"
    />
  ),
};

function TodoIcon({ type, color, onClick }) {
  return (
    /* type puede ser check o delete */
    <span className={`${type}`} onClick={onClick}>
      {iconTypes[type](color)}
    </span>
  );
}

export { TodoIcon };

```

En este archivo lo primero que necesitamos es importar los íconos SVG como `ReactComponent as <Nombre-para-diferenciar>`. Luego importamos el archivo donde ubicamos los estilos para los íconos. Creamos un objeto llamado `iconTypes` que contendrá un diccionario de íconos, le pasamos una `key` que será `check` o `delete` y como `value` le damos una `arrow function` para enviar la prop `color` al renderizado. Dentro de esta función es donde se hace el llamado a los archivos SVG `<CheckSvg/>` y `<DeleteSvg/>`, se les agrega una clase para los estilos en el CSS y se asigna el color del SVG con la propiedad `fill`.

Creamos un componente para llamar al renderizado de los íconos. Este componente `TodoIcons` recibe 3 `props`:

- **type**: El tipo de ícono que recibirá (`check` o `delete`)
- **color**: El color de relleno del ícono.
- **onClick**: El evento que va a realizar el ícono (botón) al darle clic.

❖ Dato importante: No confundir el `onClick` creado como prop con el `onClick` definido dentro de la etiqueta `span`, este último es un evento y el anterior como ya se dijo es una propiedad.

Luego el retorno de este componente será la plantilla para renderizar cualquier ícono. En este caso en particular los íconos de `check` y `delete` irán en una línea, por lo que invocamos un `span` y ubicamos clases generales que servirán como contenedores de los íconos, para darles la posición, tamaño, disposición, etc. Usamos `template literals` para pasar de forma dinámica el tipo de ícono que vamos a renderizar cada vez que se llame la función, dependiendo del tipo se usará uno u otro estilo del documento CSS.

Dentro del `span` se llama al objeto `{iconTypes}` creado al inicio, luego le pasamos el `[type]` y finalmente accedemos a la `arrow function` pasándole como argumento el `(color)` entre paréntesis porque la función tiene un parámetro.

Toda esta lógica es una especie de componente plantilla para renderizar cualquier ícono. Dentro de este componente se realiza:

- El `import` del SVG
- El renderizado del ícono SVG
- Se llama a los estilos para el ícono y el contenedor del ícono
- La activación del evento `onClick`.

2. `CompleteIcon.js` y `DeleteIcon.js` hijos de `TodoItem` y padres de `TodoIcons`.

```
import React from "react";
import { TodoIcon } from "./TodoIcon";

function CompleteIcon({ completed, onComplete }) {
  return (
    <TodoIcon
      type="check"
      color={completed ? "#4CAF50" : "#4F46E5"}
      onClick={onComplete} // Este onClick es un prop no un event
    />
  );
}

export { CompleteIcon };
```

```
import React from "react";
import { TodoIcon } from "./TodoIcon";

function DeleteIcon({ onDelete }) {
```

```

    return <TodoIcon type="delete" color="#4F46E5" onClick={onDelete} />; //
Este onClick es un prop
}

export { DeleteIcon };

```

Este archivo es más simple y su función es de recibir los props enviados del componente padre. Los props deconstruidos serían: `completed` y `onComplete / onDelete`. El primero sirve para determinar con un condicional ternario si el color a enviar será uno u otro. El segundo es equivalente para el ícono de `check` y `delete` que sería el evento al darle `click`.

Por lo tanto, este archivo sirve para:

- Recibir los props deconstruidos de su componente padre `TodoItem`: `completed` y `onComplete / onDelete`.
- Crear el listado de props a enviar al componente hijo `TodoIcon`, los props son: `type`, `color` y `onClick`.
- Declarar los valores para esos props que se enviarán al componente hijo, es decir, qué tipo de ícono es, el color y la acción que se realizará al darle `click` al ícono.

3. Pasamos al componente `TodoItem`. Este componente es hijo de `App` y padre de `CompleteIcon` y `DeleteIcon`. Como hijo recibe props y como padre envía `completed` y `onComplete / onDelete`.

```

import { CompleteIcon } from "./CompleteIcon";
import { DeleteIcon } from "./DeleteIcon";
import "../css/TodoItem.css";

function TodoItem(props) {
  return (
    <li>
      <CompleteIcon
        completed={props.completed}
        onComplete={props.onComplete}
      />
      <p className={`${props.completed && "p--completed"}`}>
        {props.text}
      </p>
      <DeleteIcon onDelete={props.onDelete} />
    </li>
  );
}

export { TodoItem };

```

Las funciones de este componente serán:

- Llamar a los componentes `CompleteIcon` y `DeleteIcon` para ser renderizados en `TodoItem`.
- Recibir los props `completed` y `onComplete / onDelete` del componente padre.

- Crear el listado de props a enviar a los componentes hijos que necesitan saber los estados de los eventos. Para el ícono de `check` se necesita saber si está completado o no y la acción a realizar cuando esté completado. Para el ícono de `delete` únicamente la acción a realizar al darle `click` al botón, es decir `onDelete`.

Se recibe también como props el texto y se realizan otras funciones para el renderizado de cada ítem, pero esto no es parte de la lógica del renderizado dinámico de los íconos.

Comentario

Código de la clase

`src > components > TodoIcon.js`

```
import { ReactComponent as CheckSvg } from "../svg/check.svg";
import { ReactComponent as DeleteSvg } from "../svg/delete.svg";
import "../css/TodoIcon.css";

const iconTypes = {
    check: (color) => <CheckSvg fill={color} />,
    delete: (color) => (
        <DeleteSvg
            className="icon"
            fill="#4F46E5"
            stroke="#4F46E5"
            strokeWidth="0.1"
        />
    ),
};

function TodoIcon({ type, color, onClick }) {
    return (
        /* type puede ser check o delete */
        <span className={`$ {type}`} onClick={onClick}>
            {iconTypes[type](color)}
        </span>
    );
}

export { TodoIcon };
```

`src > css > TodoIcon.css`

```
.icon:hover {
    fill: #ea0031;
}
```

`src > components > CompleteIcon.js`

```

import React from "react";
import { TodoIcon } from "./TodoIcon";

function CompleteIcon({ completed, onComplete }) {
    return (
        <TodoIcon
            type="check"
            color={completed ? "#4CAF50" : "#4F46E5"}
            onClick={onComplete}
        />
    );
}

export { CompleteIcon };

```

src > components > DeleteIcon.js

```

import React from "react";
import { TodoIcon } from "./TodoIcon";

function DeleteIcon({ onDelete }) {
    return <TodoIcon type="delete" color="#4F46E5" onClick={onDelete} />;
}

export { DeleteIcon };

```

src > components > TodoItem.js

```

import { CompleteIcon } from "./CompleteIcon";
import { DeleteIcon } from "./DeleteIcon";
import "../css/TodoItem.css";

function TodoItem(props) {
    return (
        <li>
            <CompleteIcon
                completed={props.completed}
                onComplete={props.onComplete}
            />
            <p className={`${props.completed && "p--completed"}`}>
                {props.text}
            </p>
            <DeleteIcon onDelete={props.onDelete} />
        </li>
    );
}

```

```
export { TodoItem };
```

src > css > TodoItem.css

```
li {
  list-style: none;
  background-color: #cbd5e1;
  width: 15rem;
  height: 3rem;
  border-radius: 5px;
  display: flex;
  justify-content: space-between;
  align-items: center;
  padding: 0 1rem 0 0.5rem;

  position: relative;
}

.check {
  cursor: pointer;
}

.delete {
  cursor: pointer;

  position: absolute;
  right: 0;
  top: 0;
}

p {
  width: 85%;
  height: auto;
  margin: 0 5px 0;
  color: #090b10;
}

.p--completed {
  text-decoration: line-through;
}
```

13. Local Storage con React.js

El `localStorage` es una característica de JavaScript que permite almacenar datos en el navegador de forma persistente. Puedes utilizar el `localStorage` en una aplicación de React para almacenar y recuperar datos locales en el navegador del usuario.

Aquí hay una explicación básica de cómo funciona y cómo se usa el `localStorage` en React:

1. Importa el `localStorage`: Primero, debes importar el `localStorage` en tu archivo de componente de React. Puedes hacerlo agregando la siguiente línea al principio de tu archivo:

```
import React from "react";
```

2. Guardar datos en el `localStorage`: Para guardar datos en el `localStorage`, puedes utilizar el método `setItem(key, value)`. El "key" es una cadena que identifica el dato que deseas guardar, y el "value" es el dato que deseas almacenar. Por ejemplo, si deseas guardar el nombre del usuario, puedes hacerlo de la siguiente manera:

```
localStorage.setItem("nombre", "John");
```

3. Recuperar datos del `localStorage`: Para recuperar datos del `localStorage`, utiliza el método `getItem(key)`. Proporciona la "key" del dato que deseas recuperar y el método devolverá el valor asociado a esa clave. Por ejemplo, para obtener el nombre del usuario que guardamos anteriormente, puedes usar:

```
const nombre = localStorage.getItem("nombre");
console.log(nombre); // Imprime "John"
```

4. Eliminar datos del `localStorage`: Si deseas eliminar un dato específico del `localStorage`, puedes utilizar el método `removeItem(key)`. Proporciona la "key" del dato que deseas eliminar y se eliminará del almacenamiento local. Por ejemplo:

```
localStorage.removeItem("nombre");
```

5. Limpiar todos los datos del `localStorage`: Si deseas eliminar todos los datos almacenados en el `localStorage`, puedes usar el método `clear()`. Esto eliminará todos los datos guardados en el almacenamiento local. Por ejemplo:

```
localStorage.clear();
```

Es importante tener en cuenta que el `localStorage` tiene capacidad limitada (generalmente alrededor de 5 MB) y los datos almacenados en él están disponibles incluso después de cerrar y volver a abrir el navegador.

Recuerda que el `localStorage` es específico del navegador y solo puede ser accedido por la misma aplicación en el mismo dominio. Si necesitas compartir datos entre diferentes aplicaciones o dominios,

puedes considerar otras opciones como el `sessionStorage` o utilizar una solución de almacenamiento en el lado del servidor.

Guardando `strings`

El objeto `localStorage` en los navegadores web es una función que permite almacenar datos de forma persistente en el navegador. Sin embargo, `localStorage` solo puede almacenar datos en forma de cadenas de texto (strings). Esto significa que, si deseas guardar estructuras complejas de datos, como objetos JavaScript o matrices, deberás convertirlos en cadenas de texto antes de almacenarlos en `localStorage`.

Para guardar una estructura compleja en `localStorage`, puedes utilizar el método `JSON.stringify()` para convertir el objeto en una cadena de texto JSON legible. Luego, puedes almacenar esa cadena de texto en `localStorage`. Por ejemplo:

```
var myObject = { name: "John", age: 30 };
var jsonString = JSON.stringify(myObject);
localStorage.setItem("myData", jsonString);
```

Para recuperar los datos de `localStorage` y convertirlos nuevamente en una estructura compleja, puedes usar el método `JSON.parse()`. Por ejemplo:

```
var storedData = localStorage.getItem("myData");
var parsedObject = JSON.parse(storedData);
console.log(parsedObject.name); // "John"
console.log(parsedObject.age); // 30
```

Código de la clase

Hacemos algunas pruebas en la consola del navegador `Ctrl + Shift + i`:

```
const defaultTodos = [
  { text: "Lorem lorem", completed: true },
  { text: "Don't cry", completed: false },
  { text: "Lorem ipsum", completed: false },
  { text: "Take care", completed: false },
  { text: "Loremlorem", completed: true },
];

let stringTodos = JSON.stringify(defaultTodos);
localStorage.setItem("ToDos_v1", stringTodos);

const localStorageTodos = localStorage.getItem("ToDos_v1");
let parsedItems = JSON.parse(localStorageTodos);

localStorage.removeItem("ToDos_v1");
```

```

localStorage.getItem("ToDos_v1");

// Nuevamente agregamos info para visualizar en la web
const defaultTodos = [
  { text: "Lorem lorem", completed: true },
  { text: "Don't cry", completed: false },
  { text: "Lorem ipsum", completed: false },
  { text: "Take care", completed: false },
  { text: "Loremlorem", completed: true },
];

localStorage.setItem("ToDos_v1", JSON.stringify(defaultTodos));

```

src > App.js

```

import React from "react";
import { TodoCounter } from "./components/TodoCounter";
import { TodoSearch } from "./components/TodoSearch";
import { TodoList } from "./components/TodoList";
import { TodoItem } from "./components/TodoItem";
import { TodoButton } from "./components/TodoButton";

/* const defaultTodos = [
  { text: "Lorem lorem", completed: true },
  { text: "Don't cry", completed: false },
  { text: "Lorem ipsum", completed: false },
  { text: "Take care", completed: false },
  { text: "Loremlorem", completed: true },
]; */

localStorage.setItem("ToDos_v1", JSON.stringify(defaultTodos)); */
// localStorage.removeItem("ToDos_v1");

function App() {
  const localStorageTodos = localStorage.getItem("ToDos_v1"); ⏪⑩

  let parsedTodos; ⏪⑩ ↵
  if (!localStorageTodos) {
    localStorage.setItem("ToDos_v1", JSON.stringify([]));
    parsedTodos = [];
  } else {
    parsedTodos = JSON.parse(localStorageTodos);
  }

  const [todos, setTodos] = React.useState(parsedTodos); ⏪⑩
  const [searchValue, setSearchValue] = React.useState("");

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

  const searchedTodos = todos.filter((todo) => {

```

```

const todoText = todo.text.toLowerCase();
const searchText = searchValue.toLowerCase();
return todoText.includes(searchText);
});

const saveTodos = (newTodos) => { 🗑️
  localStorage.setItem("ToDos_v1", JSON.stringify(newTodos));
  setTodos(newTodos);
};

const completeTodo = (text) => {
  const newTodos = [...todos];
  const todoIndex = newTodos.findIndex((todo) => todo.text === text);

  // newTodos[todoIndex].completed = true;
  // true = false / false = true
  newTodos[todoIndex].completed = !newTodos[todoIndex].completed;
  ♦♦ saveTodos(newTodos); 🗑️
};

const deleteTodo = (text) => {
  const newTodos = [...todos];
  const todoIndex = newTodos.findIndex((todo) => todo.text === text);

  newTodos.splice(todoIndex, 1);
  ♦♦ saveTodos(newTodos); 🗑️
};

return (
  <>
    <TodoCounter completed={completedTodos} total={totalTodos} />
    <TodoSearch searchValue={searchValue} setSearchValue={setSearchValue} />

    <TodoList>
      {searchedTodos.map((todo) => (
        <TodoItem
          key={todo.text}
          text={todo.text}
          completed={todo.completed}
          // Pasar una función a un componente sin ejecutarla inmediatamente
          onComplete={() => completeTodo(todo.text)}
          onDelete={() => deleteTodo(todo.text)}
        />
      )));
    </TodoList>

    <TodoButton />
  </>
);
}

export default App;

```

💡 Evita acceder al `localStorage` dentro del componente

Acceder a los valores del `localStorage` dentro del componente es muy pesado en cuanto al rendimiento, ya que se **ejecuta sincrónicamente en cada re-renderizado del componente**. En su lugar, puedes leerlo utilizando un `callback` que retorne el valor inicial del `useState`, esto permitirá acceder a la información una sola vez al momento que se crea el componente, esto por la definición de `useState`.

```
const [todos, setTodos] = useState(() => {
  const todosFromStorage = window.localStorage.getItem("TODOS_V1");
  if (todosFromStorage) return JSON.parse(todosFromStorage);
  return [];
});
```

14. Custom Hooks

Los Custom Hooks en React son una característica que te permite extraer lógica de componentes funcionales para reutilizarla en diferentes componentes. Los Custom Hooks son funciones JavaScript que siguen ciertas convenciones al nombrar y utilizar los hooks de React existentes, como `useState`, `useEffect`, `useContext`, entre otros.

Al crear un Custom Hook, puedes encapsular una funcionalidad específica y luego utilizarla en múltiples componentes funcionales. Esto promueve la reutilización de código y ayuda a mantener tus componentes más limpios y enfocados en su lógica principal.

Para definir un Custom Hook, simplemente creas una función que utilice uno o más hooks existentes. Por convención, el nombre de un Custom Hook debe comenzar con el prefijo "use". A partir de ahí, puedes escribir la lógica personalizada que deseas encapsular y reutilizar.

Aquí hay un ejemplo básico de un Custom Hook que maneja un contador:

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

function useCounter(initialValue) {
  const [count, setCount] = useState(initialValue);

  const increment = () => {
    setCount(count + 1);
  };

  const decrement = () => {
    setCount(count - 1);
  };
  // También se puede retornar como objeto {}
  return [count, increment, decrement];
}

export default useCounter;
```

En este ejemplo, el Custom Hook `useCounter` utiliza el hook `useState` para manejar el estado del contador. Proporciona una interfaz para obtener el valor del contador, así como para incrementarlo y decrementarlo.

Luego, en cualquier componente funcional, puedes utilizar este Custom Hook para agregar la funcionalidad del contador:

```
import React from "react";
import useCounter from "./useCounter";

function CounterComponent() {
    // También se puede colocar como objeto {} incluso crear dos const y
    // nombrarlas como counterOne y counterTwo esto para acceder con el . (punto) a
    // increment y decrement
    const [count, increment, decrement] = useCounter(0);

    return (
        <div>
            <p>Count: {count}</p>
            <button onClick={increment}>Increment</button>
            <button onClick={decrement}>Decrement</button>
            // {counterOne.increment}
            // {counterTwo.decrement}
        </div>
    );
}

export default CounterComponent;
```

En este ejemplo, `CounterComponent` utiliza el Custom Hook `useCounter` para obtener el estado del contador y las funciones `increment` y `decrement`. Esto te permite reutilizar la lógica del contador en cualquier otro componente funcional sin tener que repetir el código.

💡 La CLAVE de los CUSTOM HOOKS en React

Código de la clase

src > App.js

```
import React from "react";
import { TodoCounter } from "./components/TodoCounter";
import { TodoSearch } from "./components/TodoSearch";
import { TodoList } from "./components/TodoList";
import { TodoItem } from "./components/TodoItem";
import { TodoButton } from "./components/TodoButton";

/* const defaultTodos = [
```

```

    { text: "Lorem lorem", completed: true },
    { text: "Don't cry", completed: false },
    { text: "Lorem ipsus", completed: false },
    { text: "Take care", completed: false },
    { text: "Loremlorem", completed: true },
];

```

```

localStorage.setItem("ToDos_v1", JSON.stringify(defaultTodos)); */
// localStorage.removeItem("ToDos_v1");

```

```

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) => {
    localStorage.setItem(itemName, JSON.stringify(newItem));
    setItem(newItem);
  };

  return [item, saveItem];
}

```

```

function App() {
  const [todos, saveTodos] = useLocalStorage('ToDos_v1', []);
  const [searchValue, setSearchValue] = React.useState("");

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

  const searchedTodos = todos.filter((todo) => {
    const todoText = todo.text.toLowerCase();
    const searchText = searchValue.toLowerCase();
    return todoText.includes(searchText);
  });

  const completeTodo = (text) => {
    const newTodos = [...todos];
    const todoIndex = newTodos.findIndex((todo) => todo.text === text);

    // newTodos[todoIndex].completed = true;
    // true = false / false = true
    newTodos[todoIndex].completed = !newTodos[todoIndex].completed;
    saveTodos(newTodos);
  };
}

```

```

const deleteTodo = (text) => {
  const newTodos = [...todos];
  const todoIndex = newTodos.findIndex((todo) => todo.text == text);

  newTodos.splice(todoIndex, 1);
  saveTodos(newTodos);
};

return (
  <>
    <TodoCounter completed={completedTodos} total={totalTodos} />
    <TodoSearch searchValue={searchValue} setSearchValue={setSearchValue} />

    <TodoList>
      {searchedTodos.map((todo) => (
        <TodoItem
          key={todo.text}
          text={todo.text}
          completed={todo.completed}
          // Pasar una función a un componente sin ejecutarla inmediatamente
          onComplete={() => completeTodo(todo.text)}
          onDelete={() => deleteTodo(todo.text)}
        />
      ))}
    </TodoList>

    <TodoButton />
  </>
);
}

export default App;

```

➊ Evita las dependencias dentro de tus componentes con Custom Hooks

Cuando estás utilizando paquetes dentro de React, por ejemplo, el paquete inventado [QueryPlatzi](#), **evita importarlo en cada componente**, a menos que sea necesario.

```

import { useQueryPlatzi } from 'query-platzi'

function Component () {
  const query = useQueryPlatzi()

  return ...
}

```

En su lugar, crea un custom Hook para abstraer la funcionalidad del paquete.

```
// archivo -> hooks/useQueryApp.js
import { useQueryPlatzi } from "query-platzi";

export function useQueryApp() {
    return useQueryPlatzi;
}
```

Aunque parezca algo insignificante, es realmente poderoso, ya que, si en un futuro necesitas cambiar el paquete **QueryPlatzi** por otro, **solo lo harás en un solo sitio**. De esta forma el componente se mantiene lo más **declarativo** posible.

```
import { useQueryApp } from '@hooks/useQueryApp'

function Component () {
    const query = useQueryApp()
    // Solo realizo una función, utilizar la query

    return ...
}
```

Finalmente, según la documentación de React, si observas un **useEffect** muy usado o con mucha lógica, lo más seguro es que puedas abstraerlo en un custom Hook.

Fuente: [Reutilización de lógica utilizando Hooks personalizados](#)

15. Organización de archivos y carpetas

```

├── package-lock.json
└── package.json
└── public
    ├── index.html
    └── manifest.json
    └── robots.txt
└── src
    ├── App.js
    └── components ↗️ ↘️ ↖️
        ├── CompleteIcon
        │   ├── check.svg
        │   └── index.js
        ├── DeleteIcon
        │   ├── delete.svg
        │   └── index.js
        └── TodoButton
            ├── TodoButton.css
            ├── add.svg
            └── index.js
    └── TodoCounter

```

```

    └── TodoCounter.css
    └── index.js
  └── TodoIcon
    ├── TodoIcon.css
    └── index.js
  └── TodoItem
    ├── TodoItem.css
    └── index.js
  └── TodoList
    ├── TodoList.css
    └── index.js
  └── TodoSearch
    ├── TodoSearch.css
    ├── index.js
    └── search.svg
  └── test.js
  └── css
    ├── index.css
    └── test.css
  └── index.js
  └── svg
    ├── add-pink.svg
    ├── check-completed.svg
    └── delete-hover.svg

```

◆ Para ordenar de manera más rápida puedes hacer esto:

Teniendo un archivo `algo.js` puedes darle a `rename`, le agregas algo así `algo/index.js` y le das enter. De esta manera se crea una carpeta `algo` con un archivo `index.js` dentro.

Crear carpetas automáticamente

Un comando para que se cree la carpeta automáticamente con el nombre y luego lo guarde en la misma

```

#!/bin/bash

# Obtener todos los nombres de archivo con extensión .tsx
files=$(find . -type f -name "*.tsx")

# Recorrer los nombres de archivo
for file in $files; do
  # Obtener el nombre del archivo sin la extensión .tsx
  filename=$(basename "$file" .tsx)

  # Crear la carpeta con el mismo nombre
  mkdir "$filename"

  # Mover el archivo a la carpeta
  mv "$file" "$filename/"

done

```

```
echo "Se han creado las carpetas y se han movido los archivos .tsx."
```

Código de la clase

Debes cambiar todos los `import` que entren en conflicto y colocar las rutas correctas.

16. Feature-First Directories en React

En React, "Feature-First Directories" se refiere a una estructura de organización de archivos y carpetas en un proyecto de React basada en características o funcionalidades. En lugar de organizar los archivos por tipos (componentes, estilos, etc.), se agrupan según las características o características específicas de la aplicación.

En un enfoque de "Feature-First Directories", se crea una carpeta separada para cada característica o módulo de la aplicación. Dentro de cada carpeta, se colocan todos los archivos relacionados con esa característica, como componentes, estilos, pruebas y cualquier otro archivo necesario específicamente para esa característica.

Esta estructura tiene varias ventajas. Primero, facilita la comprensión y navegación del código, ya que los archivos relacionados están agrupados juntos. Además, es más escalable, ya que es más fácil agregar nuevas características o modificar características existentes sin afectar otras partes del proyecto. También promueve una mayor reutilización de componentes, ya que los componentes específicos de una característica están ubicados en la misma carpeta y pueden ser más fácilmente identificados y reutilizados en otros lugares si es necesario.

Aquí hay un ejemplo de cómo podría verse la estructura de directorios en un enfoque de "Feature-First Directories":

```
/src
  /features
    /Home
      /components
        HomePage.js
        HomeHeader.js
        HomeFooter.js
      /styles
        home.css
      /tests
        HomePage.test.js
        index.js
    /Login
      /components
        LoginForm.js
        LoginButton.js
      /styles
        login.css
      /tests
        LoginForm.test.js
```

```

index.js
/shared
  /components
    Header.js
    Footer.js
  /styles
    shared.css
  /tests
    Header.test.js
App.js
index.js

```

En este ejemplo, hay dos características principales: "Home" y "Login". Cada una tiene su propia carpeta que contiene los componentes, estilos y pruebas específicos de esa característica. Además, hay una carpeta "shared" para componentes, estilos y pruebas compartidos que pueden ser utilizados por múltiples características.

La estructura de "Feature-First Directories" no es una convención estricta en React, pero puede ser una forma organizativa útil y efectiva dependiendo del tamaño y complejidad del proyecto. Es importante tener en cuenta que la estructura de directorios puede variar según las preferencias del equipo de desarrollo y las necesidades específicas del proyecto.

Código de la clase

En esta clase también creamos una carpeta para el archivo `App.js` que ahora se llamará `index.js` y además movimos una parte del código a otro archivo llamado `useLocalStorage.js` en esta misma carpeta:

`src > App > index.js`

```

import React from "react";
import { TodoCounter } from "../components/TodoCounter/index";
import { TodoSearch } from "../components/TodoSearch/index";
import { TodoList } from "../components/TodoList/index";
import { TodoItem } from "../components/TodoItem/index";
import { TodoButton } from "../components/TodoButton/index";
import { useLocalStorage } from "./useLocalStorage";

/* const defaultTodos = [
  { text: "Lorem lorem", completed: true },
  { text: "Don't cry", completed: false },
  { text: "Lorem ipsum", completed: false },
  { text: "Take care", completed: false },
  { text: "Loremlorem", completed: true },
]; */

localStorage.setItem("Todos_v1", JSON.stringify(defaultTodos)); */
// localStorage.removeItem("Todos_v1");

```

```

function App() {
  const [todos, saveTodos] = useLocalStorage("ToDos_v1", []);
  const [searchValue, setSearchValue] = React.useState("");

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

  const searchedTodos = todos.filter((todo) => {
    const todoText = todo.text.toLowerCase();
    const searchText = searchValue.toLowerCase();
    return todoText.includes(searchText);
  });

  const completeTodo = (text) => {
    const newTodos = [...todos];
    const todoIndex = newTodos.findIndex((todo) => todo.text === text);

    // newTodos[todoIndex].completed = true;
    // true = false / false = true
    newTodos[todoIndex].completed = !newTodos[todoIndex].completed;
    saveTodos(newTodos);
  };

  const deleteTodo = (text) => {
    const newTodos = [...todos];
    const todoIndex = newTodos.findIndex((todo) => todo.text === text);

    newTodos.splice(todoIndex, 1);
    saveTodos(newTodos);
  };

  return (
    <>
      <TodoCounter completed={completedTodos} total={totalTodos} />
      <TodoSearch
        searchValue={searchValue}
        setSearchValue={setSearchValue}
      />

      <TodoList>
        {searchedTodos.map((todo) => (
          <TodoItem
            key={todo.text}
            text={todo.text}
            completed={todo.completed}
            // Pasar una función a un componente sin ejecutarla
            inmediatamente
            onComplete={() => completeTodo(todo.text)}
            onDelete={() => deleteTodo(todo.text)}
          />
        )));
      </TodoList>
    </>
  );
}

```

```

        <TodoButton />
    </>
);
}

export default App;

```

src > App > useLocalStorage.js

```

import React from "react";

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) => {
        localStorage.setItem(itemName, JSON.stringify(newItem));
        setItem(newItem);
    };

    return [item, saveItem];
}

export { useLocalStorage };

```

[4 estructuras para organizar tu proyecto de React y React Native](#)

17. Tips para naming y abstracción de componentes React

Stateless component y Stateful component

En el contexto de React, los términos "stateless" y "stateful" se utilizan para describir los componentes y su manejo de datos.

1. Stateless component (componente sin estado): También conocido como componente funcional, es un componente de React que no tiene estado interno y no utiliza el concepto de "estado" de React. Se implementa como una función en lugar de una clase y generalmente se utiliza para componentes simples que no requieren mantener o manipular datos.

Un componente sin estado se basa únicamente en las props que recibe como argumento y devuelve elementos de React (generalmente JSX) según esas props. No almacena información adicional ni realiza cambios en su propio estado interno. Estos componentes son más fáciles de entender, probar y mantener debido a su simplicidad y falta de lógica interna compleja.

Ejemplo de un componente sin estado en React:

```
function Greeting(props) {
    return <h1>Hello, {props.name}!</h1>;
}
```

2. Stateful component (componente con estado): También conocido como componente de clase, es un componente de React que mantiene un estado interno y puede realizar cambios en ese estado. Se implementa como una clase que extiende la clase base `React.Component`.

Un componente con estado tiene la capacidad de almacenar y manipular datos a través de su estado interno. Utiliza el método `setState()` para actualizar su estado y, cuando el estado cambia, React se encarga de actualizar automáticamente la interfaz de usuario correspondiente. Estos componentes son útiles para manejar componentes más complejos que requieren interacción y actualización dinámica.

Ejemplo de un componente con estado en React:

```
class Counter extends React.Component {
    constructor(props) {
        super(props);
        this.state = { count: 0 };
    }

    increment() {
        this.setState({ count: this.state.count + 1 });
    }

    render() {
        return (
            <div>
                <p>Count: {this.state.count}</p>
                <button onClick={() => this.increment()}>Increment</button>
            </div>
        );
    }
}
```

Statefull: Componentes que se enfocan en el manejo de estados y la logica de la app (delete, complete, save)

Stateless: Componentes que se encargan de la interfaz (AppUI)

En resumen, los componentes sin estado (stateless) en React son funciones que no tienen un estado interno y se basan únicamente en las props recibidas, mientras que los componentes con estado (stateful) son clases que mantienen un estado interno y pueden realizar cambios en ese estado utilizando `setState()`.

AppUI.js

El archivo `AppUI.js` en React puede ser un componente personalizado que define la interfaz de usuario (UI) principal de una aplicación. No hay ninguna convención específica en React que establezca el nombre `AppUI.js`, por lo que es posible que sea un nombre elegido por los desarrolladores para su componente de interfaz de usuario principal.

En general, el componente `AppUI.js` podría contener la estructura básica de la interfaz de usuario de una aplicación, como la disposición de los componentes, la navegación, los estilos y cualquier otro elemento visual. Puede incluir otros componentes de React y utilizar props y estado para gestionar la interacción y el flujo de datos dentro de la aplicación.

Aquí hay un ejemplo simplificado de cómo podría verse un archivo `AppUI.js`:

```
import React from "react";

const AppUI = () => {
  return (
    <div>
      <header>
        <h1>My App</h1>
      </header>
      <nav>{/* Componente de navegación */}</nav>
      <main>{/* Componentes de contenido */}</main>
      <footer>{/* Componente de pie de página */}</footer>
    </div>
  );
};

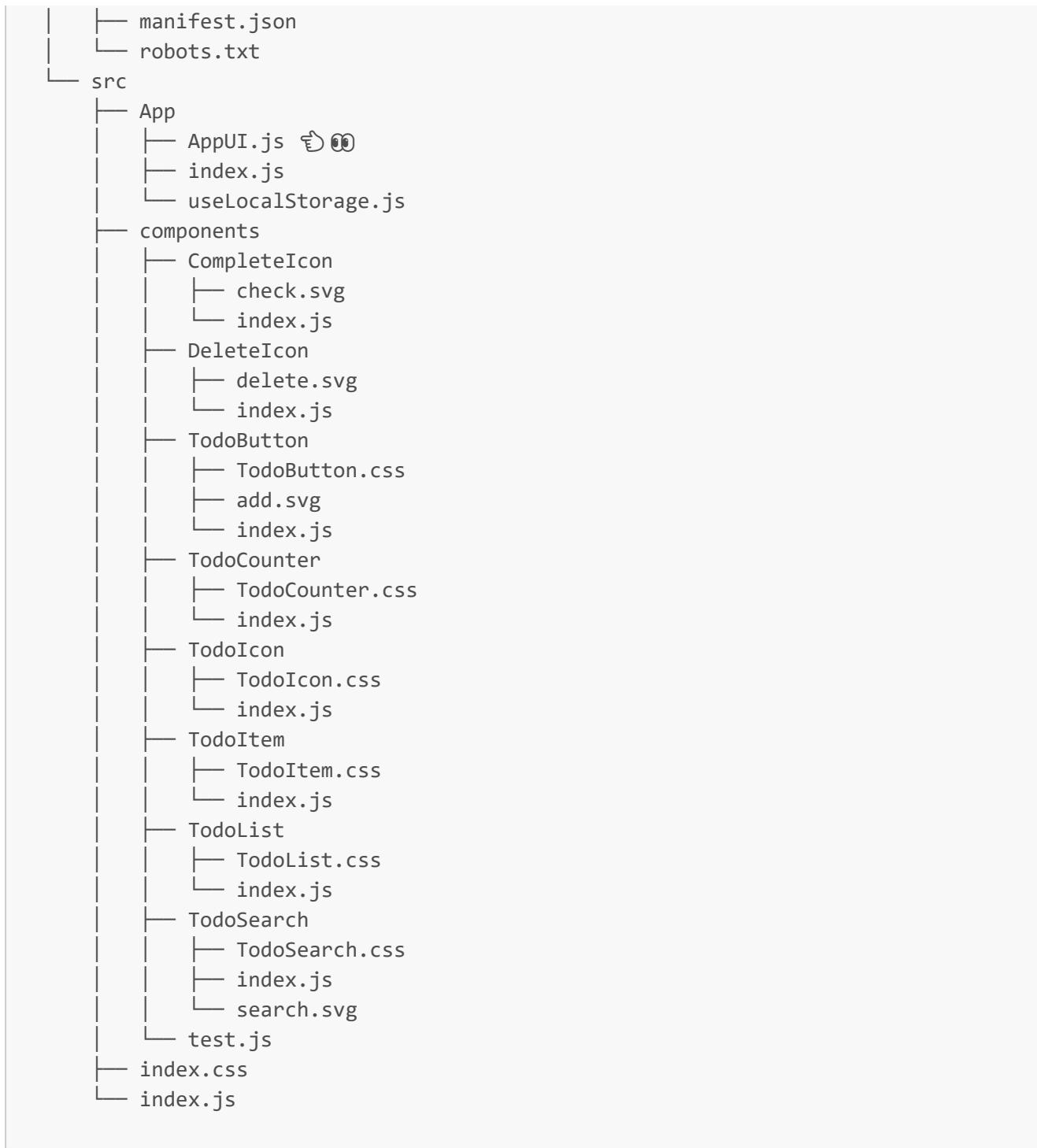
export default AppUI;
```

En este ejemplo, el archivo `AppUI.js` define la estructura básica de la aplicación, incluyendo un encabezado, una navegación, un contenido principal y un pie de página. Cada sección puede contener componentes adicionales que se importan y se utilizan dentro de `AppUI`.

Código de la clase

Creamos un nuevo archivo llamado `AppUI.js` así que terminamos con la siguiente estructura:

```
└── package-lock.json
└── package.json
└── public
    └── index.html
```



src > App > index.js

```

import React from "react";
import { AppUI } from "./AppUI";
import { useLocalStorage } from "./useLocalStorage";

function App() {
  const [todos, saveTodos] = useLocalStorage("ToDos_v1", []);
  const [searchValue, setSearchValue] = React.useState("");

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

```

```

const searchedTodos = todos.filter((todo) => {
  const todoText = todo.text.toLowerCase();
  const searchText = searchValue.toLowerCase();
  return todoText.includes(searchText);
});

const completeTodo = (text) => {
  const newTodos = [...todos];
  const todoIndex = newTodos.findIndex((todo) => todo.text === text);

  // newTodos[todoIndex].completed = true;
  // true = false / false = true
  newTodos[todoIndex].completed = !newTodos[todoIndex].completed;
  saveTodos(newTodos);
};

const deleteTodo = (text) => {
  const newTodos = [...todos];
  const todoIndex = newTodos.findIndex((todo) => todo.text === text);

  newTodos.splice(todoIndex, 1);
  saveTodos(newTodos);
};

return (
  <AppUI
    completedTodos={completedTodos}
    totalTodos={totalTodos}
    searchValue={searchValue}
    setSearchValue={setSearchValue}
    searchedTodos={searchedTodos}
    completeTodo={completeTodo}
    deleteTodo={deleteTodo}
  />
);
}

export default App;

```

src > App > AppUI.js

```

import { TodoCounter } from "../components/TodoCounter/"; //index
import { TodoSearch } from "../components/TodoSearch/index";
import { TodoList } from "../components/TodoList/index";
import { TodoItem } from "../components/TodoItem/index";
import { TodoButton } from "../components/TodoButton/index";

function AppUI({
  completedTodos,
  totalTodos,
  searchValue,

```

```

        setSearchValue,
        searchedTodos,
        completeTodo,
        deleteTodo,
    }) {
    return (
        <>
        <TodoCounter completed={completedTodos} total={totalTodos} />
        <TodoSearch
            searchValue={searchValue}
            setSearchValue={setSearchValue}
        />

        <TodoList>
            {searchedTodos.map((todo) => (
                <TodoItem
                    key={todo.text}
                    text={todo.text}
                    completed={todo.completed}
                    // Pasar una función a un componente sin ejecutarla
inmediatamente
                    onComplete={() => completeTodo(todo.text)}
                    onDelete={() => deleteTodo(todo.text)}
                />
            )));
        </TodoList>

        <TodoButton />
    </>
);
}

export { AppUI };

```

18. ¿Qué son los efectos en React? useEffect()

En React, los efectos son funciones especiales que permiten realizar tareas secundarias (side effects) en componentes funcionales. Los efectos se ejecutan después de que el componente se renderiza en la interfaz de usuario y se utilizan principalmente para manejar operaciones asíncronas, suscripciones a eventos, manipulación del DOM y otras interacciones con el entorno externo.

El hook `useEffect` es el que se utiliza para definir efectos en un componente funcional de React. Se ejecuta después del renderizado inicial del componente y luego en cada actualización del mismo, a menos que se especifique lo contrario.

La sintaxis básica del `useEffect` es la siguiente:

```

import React, { useEffect } from "react";

function MyComponent() {

```

```

useEffect(() => {
    // 1er argumento () => {}
    // Lógica del efecto
    // Se ejecuta después del renderizado inicial y en cada actualización
    // del componente

    // Retorno opcional de una función de limpieza
    return () => {
        // Lógica de limpieza (opcional)
        // Se ejecuta antes de desmontar el componente o antes de la
        // siguiente ejecución del efecto
    };
}, [dependency1, dependency2]); // 2do argumento []
// ...
}

```

El primer argumento de `useEffect` es una función que contiene la lógica del efecto. Esta función se ejecuta después del renderizado inicial y en cada actualización del componente. Puedes realizar cualquier tarea dentro de esta función, como realizar llamadas a API, suscribirte a eventos, actualizar el estado del componente, etc.

El segundo argumento de `useEffect` es una matriz opcional de dependencias. Estas dependencias son variables o propiedades que el efecto debe observar. Si alguna de estas dependencias cambia entre renderizaciones, el efecto se volverá a ejecutar. Si no se proporciona un arreglo de dependencias, el efecto se ejecutará en cada actualización del componente.

Además, la función de retorno opcional dentro del efecto se utiliza para realizar la limpieza de recursos o cancelar tareas cuando el componente se desmonta o antes de que se ejecute el efecto nuevamente.

Ejemplo de uso

Aquí tienes un ejemplo de cómo se puede utilizar el hook `useEffect` en un componente funcional de React:

```

import React, { useState, useEffect } from "react";

function Timer() {
    const [count, setCount] = useState(0);

    useEffect(() => {
        // Función que se ejecuta después del renderizado inicial y en cada
        // actualización del componente
        console.log("Efecto ejecutado");

        // Actualizar el título de la página con el valor actual de count
        document.title = `Contador: ${count}`;

        // Retorno de la función de limpieza
    }, []);
}

```

```

        return () => {
          // Función de limpieza que se ejecuta antes de desmontar el
          componente o antes de la siguiente ejecución del efecto
          console.log("Efecto limpiado");
        };
      }, [count]);

      return (
        <div>
          <p>Contador: {count}</p>
          <button onClick={() => setCount(count + 1)}>Incrementar</button>
        </div>
      );
    }

    export default Timer;

```

En este ejemplo, hemos creado un componente funcional llamado `Timer` que muestra un contador y un botón de incremento. El valor del contador se almacena en el estado utilizando el hook `useState`.

Luego, utilizamos el hook `useEffect` para establecer un efecto que se ejecutará después del renderizado inicial y en cada actualización del componente. En este caso, el efecto actualiza el título de la página con el valor actual de `count` y muestra mensajes en la consola.

También hemos especificado `[count]` como una dependencia del efecto, lo que significa que el efecto se ejecutará solo cuando el valor de `count` cambie.

Finalmente, hemos proporcionado una función de limpieza dentro del efecto que se ejecutará antes de desmontar el componente o antes de la siguiente ejecución del efecto. En este ejemplo, simplemente muestra un mensaje en la consola.

Cuando ejecutes este componente, verás que el título de la página se actualiza con el valor actual del contador y los mensajes de consola se mostrarán cuando el efecto se ejecute o se limpie.

Espero que este ejemplo te ayude a comprender cómo se utiliza el hook `useEffect` en un componente funcional de React.

19. Estados de carga y error

En esta clase aún faltó algo de código, así que lo agregaré completo en la siguiente clase.

20. Actualizando estados desde useEffect

Código de la clase

A causa de un error, eliminamos el `ToDos_v1` del `localStorage` para volverlo a crear de inmediato.

```
localStorage.removeItem("ToDos_v1"); // 1er paso ⏪
```

```

const defaultTodos = [
    // 2do paso ↪ 88 ↪
    { text: "Lorem lorem", completed: true },
    { text: "Don't cry", completed: false },
    { text: "Lorem ipsum", completed: false },
    { text: "Take care", completed: false },
    { text: "Loremlorem", completed: true },
];

localStorage.setItem("ToDos_v1", JSON.stringify(defaultTodos));

```

src > App > index.js

```

import React from "react";
import { AppUI } from "./AppUI";
import { useLocalStorage } from "./useLocalStorage";

function App() {
    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;

    const searchedTodos = todos.filter((todo) => {
        const todoText = todo.text.toLowerCase();
        const searchText = searchValue.toLowerCase();
        return todoText.includes(searchText);
    });

    const completeTodo = (text) => {
        const newTodos = [...todos];
        const todoIndex = newTodos.findIndex((todo) => todo.text === text);

        // newTodos[todoIndex].completed = true;
        // true = false / false = true
        newTodos[todoIndex].completed = !newTodos[todoIndex].completed;
        saveTodos(newTodos);
    };

    const deleteTodo = (text) => {
        const newTodos = [...todos];
        const todoIndex = newTodos.findIndex((todo) => todo.text === text);

        newTodos.splice(todoIndex, 1);
        saveTodos(newTodos);
    };
}

```

```

    };

    return (
      <AppUI
        loading={loading}
        error={error}
        completedTodos={completedTodos}
        totalTodos={totalTodos}
        searchValue={searchValue}
        setSearchValue={setSearchValue}
        searchedTodos={searchedTodos}
        completeTodo={completeTodo}
        deleteTodo={deleteTodo}
      />
    );
}

export default App;

```

src > App > useLocalStorage.js

```

import React from "react";

function useLocalStorage(itemName, initialValue) {
  const [item, setItem] = React.useState(initialValue);
  const [loading, setLoading] = React.useState(true);
  const [error, setError] = React.useState(false);

  React.useEffect(() => {
    setTimeout(() => {
      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);
        }

        setLoading(false);
      } catch (error) {
        setLoading(false);
        setError(true);
      }
    }, 0);
  }, []);
}

export default useLocalStorage;

```

```

        }, 2000);
    }, []));

const saveItem = (newItem) => {
    localStorage.setItem(itemName, JSON.stringify(newItem));
   setItem(newItem);
};

return { item, saveItem, loading, error };
}

export { useLocalStorage };

```

src > App > AppUI.js

```

import { TodoCounter } from "../components/TodoCounter/"; //index
import { TodoSearch } from "../components/TodoSearch/index";
import { TodoList } from "../components/TodoList/index";
import { TodoItem } from "../components/TodoItem/index";
import { TodoButton } from "../components/TodoButton/index";

function AppUI({
    loading,
    error,
    completedTodos,
    totalTodos,
    searchValue,
    setSearchValue,
    searchedTodos,
    completeTodo,
    deleteTodo,
}) {
    return (
        <>
            <TodoCounter completed={completedTodos} total={totalTodos} />
            <TodoSearch
                searchValue={searchValue}
                setSearchValue={setSearchValue}
            />

            <TodoList>
                {loading && <span>Loading...</span>}
                {error && <span>An error occurred!!! 🙄</span>}
                {!loading && searchedTodos.length === 0 && (
                    <span>Create your first ToDo 🐾</span>
                )}

                {searchedTodos.map((todo) => (
                    <TodoItem
                        key={todo.text}
                        text={todo.text}

```

```

        completed={todo.completed}
        // Pasar una función a un componente sin ejecutarla
inmediatamente
        onComplete={() => completeTodo(todo.text)}
        onDelete={() => deleteTodo(todo.text)}
    />
)}
```

</TodoList>

</>

);

export { AppUI };

- [Introduction to Backend Development by Bryan Garay](#)
- [Curso de React con Vite by Bryan Garay](#)

21. Reto: loading skeletons

Para esta clase creamos algunas carpetas y archivos:

```

├── package-lock.json
├── package.json
└── public
    ├── index.html
    ├── manifest.json
    └── robots.txt
└── src
    ├── App
    │   ├── AppUI.js
    │   ├── index.js
    │   └── useLocalStorage.js
    ├── components
    │   ├── CompleteIcon
    │   │   ├── check.svg
    │   │   └── index.js
    │   ├── DeleteIcon
    │   │   ├── delete.svg
    │   │   └── index.js
    │   ├── TodoButton
    │   │   ├── TodoButton.css
    │   │   ├── add.svg
    │   │   └── index.js
    │   ├── TodoCounter
    │   │   ├── TodoCounter.css
    │   │   └── index.js
    │   └── TodoIcon
    │       └── TodoIcon.css

```

```

    └── index.js
  └── TodoItem
    ├── TodoItem.css
    └── index.js
  └── TodoList
    ├── TodoList.css
    └── index.js
  └── TodoSearch
    ├── TodoSearch.css
    ├── index.js
    └── search.svg
  └── TodosEmpty ✎ 00 ↗
    └── index.js
  └── TodosError ✎ 00
    └── index.js
  └── TodosLoading ✎ 00
    ├── TodosLoading.css
    └── index.js
  └── css
    └── index.css
  └── index.css
  └── index.js
  └── index.js

```

Código de la clase

src > App > AppUI.js

```

import { TodoCounter } from "../components/TodoCounter/"; //index
import { TodoSearch } from "../components/TodoSearch/index";
import { TodoList } from "../components/TodoList/index";
import { TodoItem } from "../components/TodoItem/index";
import { TodoButton } from "../components/TodoButton/index";
import { TodosLoading } from "../components/TodosLoading";
import { TodosError } from "../components/TodosError";
import { TodosEmpty } from "../components/TodosEmpty";

function AppUI({
  loading,
  error,
  completedTodos,
  totalTodos,
  searchValue,
  setSearchValue,
  searchedTodos,
  completeTodo,
  deleteTodo,
}) {
  return (
    <>

```

```

        <TodoCounter completed={completedTodos} total={totalTodos} />
        <TodoSearch
            searchValue={searchValue}
            setSearchValue={setSearchValue}
        />

        <TodoList>
            {loading && <TodosLoading />} ⏳ 00 🚧
            {error && <TodosError />} ⏳ 00 🚧
            {!loading && searchedTodos.length === 0 && <TodosEmpty />}{" "}
            ⏳ 00 🚧
            {searchedTodos.map((todo) => (
                <TodoItem
                    key={todo.text}
                    text={todo.text}
                    completed={todo.completed}
                    // Pasar una función a un componente sin ejecutarla
inmediatamente
                    onComplete={() => completeTodo(todo.text)}
                    onDelete={() => deleteTodo(todo.text)}
                />
            )))
        </TodoList>

        <TodoButton />
    </>
);
}

export { AppUI };

```

src > components > TodosLoading.js

```

import React from "react";
import "./TodosLoading.css";

function TodosLoading({}) {
    return (
        <div className="container">
            <span></span>
            <span></span>
            <span></span>
            <span></span>
        </div>
    );
}

export { TodosLoading };

```

```
.container {  
    position: absolute;  
    top: 60%;  
    left: 50%;  
    border-radius: 50%;  
    height: 96px;  
    width: 96px;  
    animation: rotate_3922 1.2s linear infinite;  
    background-color: #9b59b6;  
    background-image: linear-gradient(#4f46e5, #090b10, #4f46e5);  
}  
  
.container span {  
    position: absolute;  
    border-radius: 50%;  
    height: 100%;  
    width: 100%;  
    background-color: #9b59b6;  
    background-image: linear-gradient(#4f46e5, #090b10, #4f46e5);  
}  
  
.container span:nth-of-type(1) {  
    filter: blur(5px);  
}  
  
.container span:nth-of-type(2) {  
    filter: blur(10px);  
}  
  
.container span:nth-of-type(3) {  
    filter: blur(25px);  
}  
  
.container span:nth-of-type(4) {  
    filter: blur(50px);  
}  
  
.container::after {  
    content: "";  
    position: absolute;  
    top: 10px;  
    left: 10px;  
    right: 10px;  
    bottom: 10px;  
    background-color: #090b10;  
    border: solid 5px #4f46e5;  
    border-radius: 50%;  
}  
  
@keyframes rotate_3922 {
```

```

from {
  transform: translate(-50%, -50%) rotate(0deg);
}

to {
  transform: translate(-50%, -50%) rotate(360deg);
}

```

src > components > TodosEmpty.js

```

import React from "react";

function TodosEmpty({}) {
  return <span>Create your first ToDo...</span>;
}

export { TodosEmpty };

```

src > components > TodosError.js

```

import React from "react";

function TodosError({}) {
  return <span>Error...</span>;
}

export { TodosError };

```

❖ Create, share, and use beautiful custom elements made with CSS or Tailwind

22. ¿Qué es React Context?

En React, el método `createContext` se utiliza para crear un contexto, que es una forma de compartir datos entre componentes en una jerarquía sin tener que pasar explícitamente las props a través de cada nivel. Proporciona una forma eficiente de transmitir datos a través de múltiples componentes sin necesidad de utilizar props intermedias.

La sintaxis básica para utilizar `createContext` es la siguiente:

```
const MiContexto = React.createContext(valorPorDefecto);
```

Aquí, `valorPorDefecto` es el valor inicial que se proporciona al contexto. Este valor se utilizará cuando un componente consumidor no pueda encontrar un proveedor correspondiente.

Para utilizar el contexto creado, hay dos componentes principales involucrados:

1. **Proveedor (Provider)**: El proveedor es un componente de React que envuelve a los componentes que desean consumir el contexto. Proporciona el valor actual del contexto a los componentes descendientes.
2. **Consumidor (Consumer)**: El consumidor es un componente de React que se utiliza para acceder al valor del contexto proporcionado por el proveedor. Puede ser utilizado en cualquier nivel de la jerarquía de componentes descendientes del proveedor.

Aquí tienes un ejemplo de cómo se utiliza `createContext` en React:

```
// Crear el contexto
const MiContexto = React.createContext("valor por defecto");

// Componente proveedor
function ProveedorComponente() {
    const valorContexto = "Valor del contexto proporcionado";

    return (
        <MiContexto.Provider value={valorContexto}>
            <ComponenteHijo />
        </MiContexto.Provider>
    );
}

// Componente consumidor
function ComponenteHijo() {
    return (
        <MiContexto.Consumer>
            {(valor) => <p>El valor del contexto es: {valor}</p>}
        </MiContexto.Consumer>
    );
}

// Componente principal que utiliza el proveedor y el consumidor
function App() {
    return (
        <div>
            <ProveedorComponente />
        </div>
    );
}

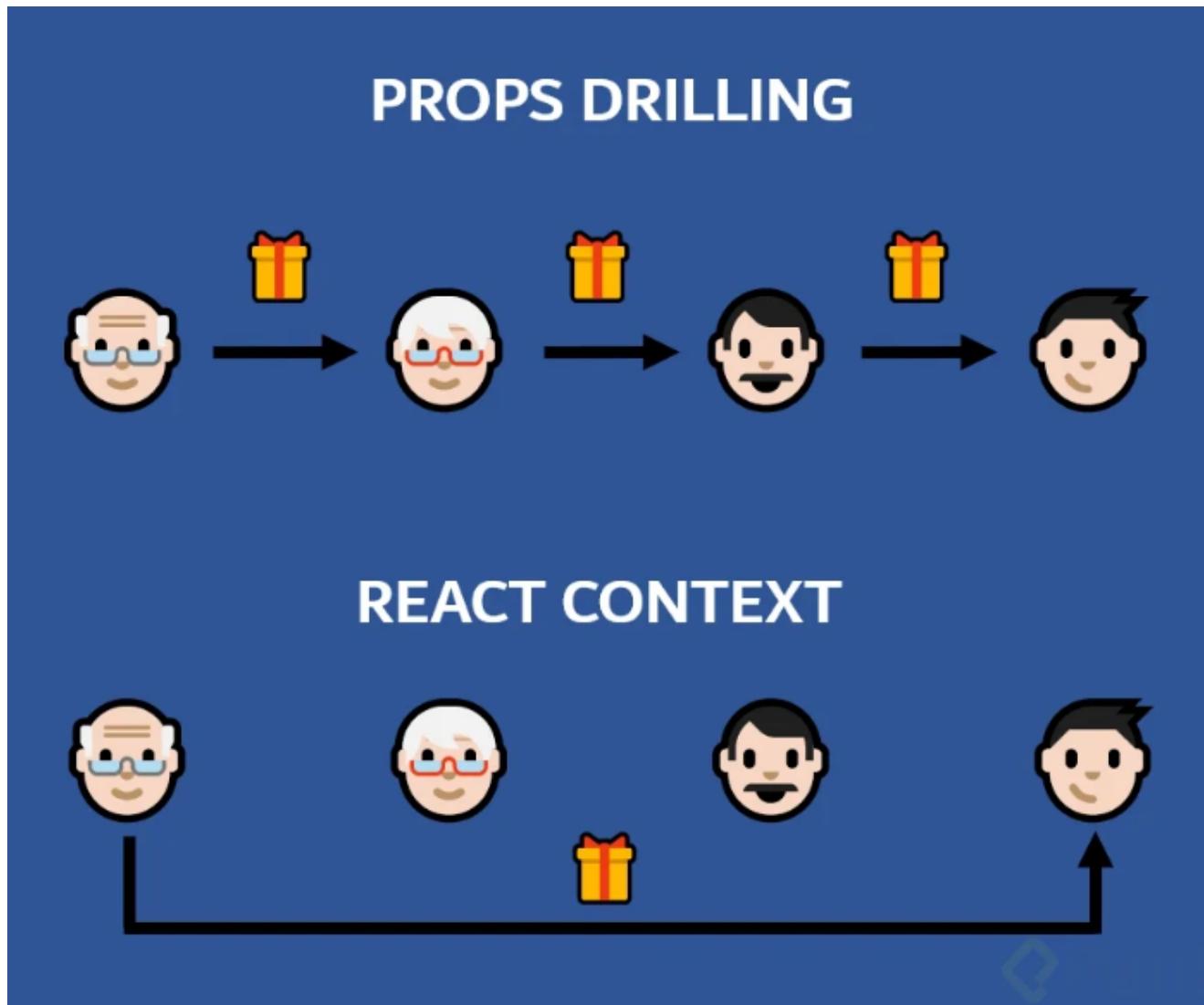
ReactDOM.render(<App />, document.getElementById("root"));
```

En este ejemplo, se crea el contexto `MiContexto` con un valor por defecto de "valor por defecto". Luego, se utiliza el componente `MiContexto.Provider` para envolver al componente `ComponenteHijo` y proporcionarle un valor específico ("Valor del contexto proporcionado"). Dentro del componente

ComponenteHijo, se utiliza el componente `MiContexto.Consumer` para acceder al valor del contexto y renderizarlo.

Es importante tener en cuenta que los componentes consumidores solo pueden acceder al contexto si están dentro del árbol de componentes descendientes del proveedor correspondiente. Si no hay un proveedor en el árbol, se utilizará el valor por defecto proporcionado al crear el contexto.

[[#12. Iconos con colores dinámicos#Render Props]]



Código de la clase

En esta clase el código quedó inconcluso, por lo que se agregará en la siguiente clase. Por el momento dejaré la estructura de los archivos que se agregaron:

```
├── package-lock.json  
├── package.json  
└── public  
    ├── index.html  
    ├── manifest.json  
    └── robots.txt  
└── src
```

```
App
  AppUI.js
  index.js
components
  CompleteIcon
    check.svg
    index.js
  DeleteIcon
    delete.svg
    index.js
  TodoButton
    TodoButton.css
    add.svg
    index.js
  TodoContext ⏺ 00
    index.js
    useLocalStorage.js ✨ Lo movimos a esta carpeta
  TodoCounter
    TodoCounter.css
    index.js
  TodoIcon
    TodoIcon.css
    index.js
  TodoItem
    TodoItem.css
    index.js
  TodoList
    TodoList.css
    index.js
  TodoSearch
    TodoSearch.css
    index.js
    search.svg
  TodosEmpty
    index.js
  TodosError
    index.js
  TodosLoading
    TodosLoading.css
    index.js
css
  index.css
index.css
index.js
```

23. ✨ useContext

En React, el hook `useContext` se utiliza para acceder al valor de un contexto creado con `createContext`. Proporciona una forma sencilla de consumir el contexto en componentes funcionales sin necesidad de utilizar el componente `Context.Consumer`.

La sintaxis básica para utilizar `useContext` es la siguiente:

```
const valorContexto = useContext(MiContexto);
```

Aquí, `MiContexto` es el contexto creado con `createContext`, y `valorContexto` es la variable donde se almacenará el valor actual del contexto.

Para utilizar `useContext`, es necesario que el componente funcional en el que se esté utilizando esté dentro del árbol de componentes descendientes del proveedor correspondiente al contexto. De lo contrario, `useContext` devolverá el valor por defecto proporcionado al crear el contexto.

Aquí tienes un ejemplo de cómo se utiliza `useContext` en React:

```
// Crear el contexto
const MiContexto = React.createContext("valor por defecto");

// Componente proveedor
function ProveedorComponente() {
    const valorContexto = "Valor del contexto proporcionado";

    return (
        <MiContexto.Provider value={valorContexto}>
            <ComponenteHijo />
        </MiContexto.Provider>
    );
}

// Componente consumidor
function ComponenteHijo() {
    const valorContexto = useContext(MiContexto);

    return <p>El valor del contexto es: {valorContexto}</p>;
}

// Componente principal que utiliza el proveedor y el consumidor
function App() {
    return (
        <div>
            <ProveedorComponente />
        </div>
    );
}

ReactDOM.render(<App />, document.getElementById("root"));
```

En este ejemplo, se crea el contexto `MiContexto` con un valor por defecto de "valor por defecto". Luego, se utiliza el componente `MiContexto.Provider` para envolver al componente `ComponenteHijo` y

proporcionarle un valor específico ("Valor del contexto proporcionado"). Dentro del componente **ComponenteHijo**, se utiliza el hook **useContext** para acceder al valor del contexto y renderizarlo.

Es importante tener en cuenta que **useContext** solo puede ser utilizado en componentes funcionales y no en componentes de clase. Además, **useContext** debe ser utilizado dentro del cuerpo de la función del componente funcional, no puede ser utilizado en bloques de código condicionales o loops.

Código de la clase

src > App > index.js

```
import React from "react";
import { AppUI } from "./AppUI";
import { TodoProvider } from "../components/TodoContext";

function App() {
  return (
    <TodoProvider>
      <AppUI />
    </TodoProvider>
  );
}

export default App;
```

src > components > TodoContext > index.js

```
import React from "react";
import { useLocalStorage } from "./useLocalStorage";

const TodoContext = React.createContext();

function TodoProvider({ children }) {
  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;

  const searchedTodos = todos.filter((todo) => {
    const todoText = todo.text.toLowerCase();
    const searchText = searchValue.toLowerCase();
    return todoText.includes(searchText);
  });
}
```

```

const completeTodo = (text) => {
  const newTodos = [...todos];
  const todoIndex = newTodos.findIndex((todo) => todo.text === text);

  // newTodos[todoIndex].completed = true;
  // true = false / false = true
  newTodos[todoIndex].completed = !newTodos[todoIndex].completed;
  saveTodos(newTodos);
};

const deleteTodo = (text) => {
  const newTodos = [...todos];
  const todoIndex = newTodos.findIndex((todo) => todo.text === text);

  newTodos.splice(todoIndex, 1);
  saveTodos(newTodos);
};

return (
  <TodoContext.Provider
    value={{
      loading,
      error,
      completedTodos,
      totalTodos,
      searchValue,
      setSearchValue,
      searchedTodos,
      completeTodo,
      deleteTodo,
    }}
  >
  {children}
  </TodoContext.Provider>
);
}

export { TodoContext, TodoProvider };

```

src > App > AppUI.js

```

import React from "react";
import { TodoCounter } from "../components/TodoCounter/"; //index
import { TodoSearch } from "../components/TodoSearch/index";
import { TodoList } from "../components/TodoList/index";
import { TodoItem } from "../components/TodoItem/index";
import { TodoButton } from "../components/TodoButton/index";
import { TodosLoading } from "../components/TodosLoading";
import { TodosError } from "../components/TodosError";
import { TodosEmpty } from "../components/TodosEmpty";

```

```

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

function AppUI({}) {
    const { loading, error, searchedTodos, completeTodo, deleteTodo } =
        React.useContext(TodoContext);
    return (
        <>
            <TodoCounter />
            <TodoSearch />

            <TodoList>
                {loading && <TodosLoading />}
                {error && <TodosError />}
                {!loading && searchedTodos.length === 0 && <TodosEmpty />}

                {searchedTodos.map((todo) => (
                    <TodoItem
                        key={todo.text}
                        text={todo.text}
                        completed={todo.completed}
                        // Pasar una función a un componente sin ejecutarla
inmediatamente
                        onComplete={() => completeTodo(todo.text)}
                        onDelete={() => deleteTodo(todo.text)}
                    />
                ))}
            </TodoList>

            <TodoButton />
        </>
    );
}

export { AppUI };

```

src > components > TodoContext > useLocalStorage.js

```

import React from "react";

function useLocalStorage(itemName, initialValue) {
    const [item, setItem] = React.useState(initialValue);
    const [loading, setLoading] = React.useState(true);
    const [error, setError] = React.useState(false);

    React.useEffect(() => {
        setTimeout(() => {
            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);
        }

        setLoading(false);
    } catch (error) {
        setLoading(false);
        setError(true);
    }
}, 2000);
}, []);

const saveItem = (newItem) => {
    localStorage.setItem(itemName, JSON.stringify(newItem));
    setItem(newItem);
};

return { item, saveItem, loading, error };
}

export { useLocalStorage };

```

src > components > TodoCounter > index.js

```

import React from "react";
import "../TodoCounter/TodoCounter.css";
import { TodoContext } from "../TodoContext";

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

    return totalTodos == completedTodos ? (
        <h1 className="total">Completaste todos los ToDos</h1>
    ) : (
        <h1>
            Has completado <span className="completed">{completedTodos}</span>
        {" "}
            de <span className="total">{totalTodos}</span> ToDos
        </h1>
    );
}

export { TodoCounter };

```

```
import React from "react";
import "./TodoSearch.css";
import { TodoContext } from "../TodoContext";

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

  return (
    <input
      placeholder="Search..."
      className="search"
      value={searchValue}
      onChange={(event) => {
        setSearchValue(event.target.value);
      }}
    />
  );
}

export { TodoSearch };
```

24. ¿Qué son los React Portals?

En React, los Portals son una característica que permite renderizar un componente en un nodo del DOM que está fuera de la jerarquía de componentes principal. Esto significa que puedes renderizar un componente en un lugar del DOM que no está directamente relacionado con el árbol de componentes en el que se encuentra el componente padre.

Los Portals son útiles en situaciones en las que necesitas renderizar un componente en un nodo específico del DOM, como un contenedor de diálogo, un modal, un portal de notificaciones o cualquier otro elemento que deba mostrarse fuera de su ubicación original en el árbol de componentes.

La sintaxis para utilizar Portals en React es la siguiente:

```
ReactDOM.createPortal(componente, contenedor);
```

Aquí, **componente** es el componente que deseas renderizar, y **contenedor** es un elemento del DOM en el que se desea renderizar el componente.

Aquí tienes un ejemplo de cómo se utiliza un Portal en React:

```
import React from "react";
import ReactDOM from "react-dom";
```

```

// Componente principal
function App() {
  return (
    <div>
      {/* ... */}
      {ReactDOM.createPortal(
        <ModalComponente />,
        document.getElementById("modal-root")
      )}
    </div>
  );
}

// Componente de modal
function ModalComponente() {
  return (
    <div className="modal">
      <h2>Modal</h2>
      {/* ... */}
    </div>
  );
}

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

```

En este ejemplo, el componente `ModalComponente` se está renderizando utilizando un Portal. El componente `ModalComponente` se renderizará en el elemento con el ID `modal-root`, que puede ser cualquier elemento del DOM existente o incluso un elemento creado dinámicamente.

Al utilizar Portals, puedes crear componentes modales, diálogos o cualquier otra interfaz que necesite renderizarse en un lugar específico del DOM sin que su estructura se vea afectada por la jerarquía de componentes circundante.

Código de la clase

`src > App > index.js`

```

import React from "react";
import { AppUI } from "./AppUI";
import { TodoProvider } from "../components/TodoContext";

function App() {
  return (
    <TodoProvider>
      <AppUI />;
    </TodoProvider>
  );
}

```

```
export default App;
```

src > App > AppUI.js

```
import React from "react";
import { TodoCounter } from "../components/TodoCounter/"; //index
import { TodoSearch } from "../components/TodoSearch/index";
import { TodoList } from "../components/TodoList/index";
import { TodoItem } from "../components/TodoItem/index";
import { TodoButton } from "../components/TodoButton/index";
import { TodosLoading } from "../components/TodosLoading";
import { TodosError } from "../components/TodosError";
import { TodosEmpty } from "../components/TodosEmpty";
import { Modal } from "../components/Modal";
import { TodoContext } from "../components/TodoContext";

function AppUI({}) {
  const {
    loading,
    error,
    searchedTodos,
    completeTodo,
    deleteTodo,
    openModal,
    setOpenModal,
  } = React.useContext(TodoContext);
  return (
    <>
      <TodoCounter />
      <TodoSearch />

      <TodoList>
        {loading && <TodosLoading />}
        {error && <TodosError />}
        {!loading && searchedTodos.length === 0 && <TodosEmpty />}

        {searchedTodos.map((todo) => (
          <TodoItem
            key={todo.text}
            text={todo.text}
            completed={todo.completed}
            // Pasar una función a un componente sin ejecutarla
            inmediatamente
            onComplete={() => completeTodo(todo.text)}
            onDelete={() => deleteTodo(todo.text)}
          />
        ))}
      </TodoList>

      <TodoButton />
    </>
  );
}
```

```

        {openModal && <Modal>Add ToDo </Modal>}
    </>
);
}

export { AppUI };

```

src > components > TodoContext > index.js

```

import React from "react";
import { useLocalStorage } from "./useLocalStorage";

const TodoContext = React.createContext();

function TodoProvider({ children }) {
    const {
        item: todos,
        saveItem: saveTodos,
        loading,
        error,
    } = useLocalStorage("ToDos_v1", []);
    const [searchValue, setSearchValue] = React.useState("");
    const [openModal, setOpenModal] = React.useState(true); //👉@@

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

    const searchedTodos = todos.filter((todo) => {
        const todoText = todo.text.toLowerCase();
        const searchText = searchValue.toLowerCase();
        return todoText.includes(searchText);
    });

    const completeTodo = (text) => {
        const newTodos = [...todos];
        const todoIndex = newTodos.findIndex((todo) => todo.text === text);

        // newTodos[todoIndex].completed = true;
        // true = false / false = true
        newTodos[todoIndex].completed = !newTodos[todoIndex].completed;
        saveTodos(newTodos);
    };

    const deleteTodo = (text) => {
        const newTodos = [...todos];
        const todoIndex = newTodos.findIndex((todo) => todo.text === text);

        newTodos.splice(todoIndex, 1);
        saveTodos(newTodos);
    };
}

```

```

        return (
            <TodoContext.Provider
                value={{
                    loading,
                    error,
                    completedTodos,
                    totalTodos,
                    searchValue,
                    setSearchValue,
                    searchedTodos,
                    completeTodo,
                    deleteTodo,
                    openModal,
                    setOpenModal,
                }}
            >
                {children}
            </TodoContext.Provider>
        );
    }

export { TodoContext, TodoProvider };

```

src > components > Modal > index.js

```

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

function Modal({ children }) {
    return ReactDOM.createPortal(
        <div className="Modal">{children}</div>,
        document.getElementById("modal")
    );
}

export { Modal };

```

public > index.html

```

<body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
    <div id="root"></div>
    <div id="modal"></div>
</body>

```

25. Reto: estados para abrir y cerrar un modal

Código de la clase

src > components > TodoContext > index.js

```
import React from "react";
import { useLocalStorage } from "./useLocalStorage";

const TodoContext = React.createContext();

function TodoProvider({ children }) {
  const {
    todos,
    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;

  const searchedTodos = todos.filter((todo) => {
    const todoText = todo.text.toLowerCase();
    const searchText = searchValue.toLowerCase();
    return todoText.includes(searchText);
  });

  const completeTodo = (text) => {
    const newTodos = [...todos];
    const todoIndex = newTodos.findIndex((todo) => todo.text === text);

    // newTodos[todoIndex].completed = true;
    // true = false / false = true
    newTodos[todoIndex].completed = !newTodos[todoIndex].completed;
    saveTodos(newTodos);
  };

  const deleteTodo = (text) => {
    const newTodos = [...todos];
    const todoIndex = newTodos.findIndex((todo) => todo.text === text);

    newTodos.splice(todoIndex, 1);
    saveTodos(newTodos);
  };

  return (
    <TodoContext.Provider
      value={{
        loading,
        error,
```

```

        completedTodos,
        totalTodos,
        searchValue,
        setSearchValue,
        searchedTodos,
        completeTodo,
        deleteTodo,
        openModal,
        setOpenModal,
    )}
>
{children}
</TodoContext.Provider>
);
}

export { TodoContext, TodoProvider };

```

src > App > AppUI.js

```

import React from "react";
import { TodoCounter } from "../components/TodoCounter/"; //index
import { TodoSearch } from "../components/TodoSearch/index";
import { TodoList } from "../components/TodoList/index";
import { TodoItem } from "../components/TodoItem/index";
import { TodoButton } from "../components/TodoButton/index";
import { TodosLoading } from "../components/TodosLoading";
import { TodosError } from "../components/TodosError";
import { TodosEmpty } from "../components/TodosEmpty";
import { Modal } from "../components/Modal";
import { TodoContext } from "../components/TodoContext";

function AppUI({}) {
    const {
        loading,
        error,
        searchedTodos,
        completeTodo,
        deleteTodo,
        openModal,
        setOpenModal,
    } = React.useContext(TodoContext);
    return (
        <>
            <TodoCounter />
            <TodoSearch />
            <TodoList>
                {loading && <TodosLoading />}
                {error && <TodosError />}
                {!loading && searchedTodos.length === 0 && <TodosEmpty />}
        </>
    );
}

```

```

        {searchedTodos.map((todo) => (
          <TodoItem
            key={todo.text}
            text={todo.text}
            completed={todo.completed}
            // Pasar una función a un componente sin ejecutarla
            inmediatamente
            onComplete={() => completeTodo(todo.text)}
            onDelete={() => deleteTodo(todo.text)}
          />
        )));
      </TodoList>
      <TodoButton setOpenModal={setOpenModal} /> 📲
      {openModal && <Modal>Add ToDo 🎨</Modal>}
    </>
  );
}

export { AppUI };

```

src > components > TodoButton > TodoButton.css

```

.add {
  border: none;
  background-color: #090b10;
  border-radius: 50%;
  width: 3rem;
  height: 3rem;

  background-image: url("./add.svg");
  background-size: contain;
  background-repeat: no-repeat;
  background-position: center;

  /* box-shadow: -5px 5px 5px -5px #4f46e5; */

  position: fixed;
  bottom: 1rem;
  right: 1rem;
  cursor: pointer;

  transition: transform 0.3s ease;
  z-index: 1;
}

.add:hover {
  transform: rotate(90deg);
}

```

src > components > TodoButton > index.js

```
import "./TodoButton.css";

function TodoButton({ setOpenModal }) {
    return (
        <button
            className="add"
            onClick={() => {
                setOpenModal((state) => !state);
            }}
        ></button>
    );
}

export { TodoButton };
```

src > components > Modal > index.js

```
import React from "react";
import ReactDOM from "react-dom";
import "./Modal.css";

function Modal({ children }) {
    return ReactDOM.createPortal(
        <div className="ModalBackground">{children}</div>,
        document.getElementById("modal")
    );
}

export { Modal };
```

src > components > Modal > Modal.css

```
.ModalBackground {
    width: 80dvw;
    max-width: 25rem;
    height: 40dvh;
    background-color: rgba(79, 70, 229, 0.99);
    /* border: 1px solid #090b10;
    box-shadow: -5px 5px 5px -5px #090b10; */
    border-radius: 1rem;
    display: flex;
    justify-content: center;
    align-items: center;
    margin: 0 auto;
    color: #090b10;
```

```
    font-weight: 600;

    position: fixed;
    top: 14rem;
    left: 0;
    right: 0;
    /* bottom: 0; */
}
```

26. Maquetando formularios en React

Código de la clase

src > App > AppUI.js

```
import React from "react";
import { TodoCounter } from "../components/TodoCounter/"; //index
import { TodoSearch } from "../components/TodoSearch/index";
import { TodoList } from "../components/TodoList/index";
import { TodoItem } from "../components/TodoItem/index";
import { TodoButton } from "../components/TodoButton/index";
import { TodosLoading } from "../components/TodosLoading";
import { TodosError } from "../components/TodosError";
import { TodosEmpty } from "../components/TodosEmpty";
import { Modal } from "../components/Modal";
import { TodoContext } from "../components/TodoContext";
import { TodoForm } from "../components/TodoForm";

function AppUI({}) {
  const {
    loading,
    error,
    searchedTodos,
    completeTodo,
    deleteTodo,
    openModal,
    setOpenModal,
  } = React.useContext(TodoContext);
  return (
    <>
      <TodoCounter />
      <TodoSearch />

      <TodoList>
        {loading && <TodosLoading />}
        {error && <TodosError />}
        {!loading && searchedTodos.length === 0 && <TodosEmpty />}

        {searchedTodos.map((todo) => (
          <TodoItem
```

```

                key={todo.text}
                text={todo.text}
                completed={todo.completed}
                // Pasar una función a un componente sin ejecutarla
inmediatamente
                onComplete={() => completeTodo(todo.text)}
                onDelete={() => deleteTodo(todo.text)}
            />
        )}
    </TodoList>

    <TodoButton setOpenModal={setOpenModal} />

    {openModal && (
        <Modal>
            <TodoForm />
        </Modal>
    )}
</>
);
}

export { AppUI };

```

src > components > TodoForm > index.js

```

import React from "react";
import "./TodoForm.css";

function TodoForm() {
    return (
        <form
            onSubmit={(event) => {
                event.preventDefault();
            }}
            action=""
        >
            <label htmlFor="">Write a new ToDo</label>
            <textarea
                name=""
                id=""
                cols="30"
                rows="10"
                placeholder="Write something..."
            />
            <div className="buttons">
                <button type="" className="TodoForm cancel">
                    Cancel
                </button>
                <button type="" className="TodoForm save">
                    Save
                </button>
            </div>
        
```

```

        </button>
    </div>
</form>
);
}

export { TodoForm };

```

src > components > TodoForm > TodoForm.css

```

form {
    width: 80%;
    height: 80%;
    display: grid;
    grid-template-rows: 0.5fr 2fr 0.5fr;
    /* justify-items: center; */
}

label {
    text-align: center;
}

textarea {
    font-family: 'Montserrat', Arial, Helvetica, sans-serif;
    width: 100%;
    height: 90%;
    border-radius: 10px;
    padding: 0.5rem;
    background-color: #cbd5e1;
}

.buttons {
    display: flex;
    justify-content: space-between;
}

.TodoForm {
    font-family: 'Montserrat', Arial, Helvetica, sans-serif;
    border: none;
    border-radius: 10px;
    width: 5rem;
    cursor: pointer;
    color: bold;
    font-weight: 600;
}

.cancel {
    background-color: #cbd5e1;
}

.save {

```

```
background-color: #090b10;
color: #cbd5e1;
}
```

27. Crear TODOs: React Context dentro de React Portals

Estado Local

En el contexto de React, un estado local se refiere a la capacidad de un componente de mantener y gestionar su propio conjunto de datos internos. El estado local es una característica importante de React que permite a los componentes almacenar y actualizar información específica de su estado interno sin depender de otros componentes.

Cuando se crea un componente en React, se puede definir un estado local utilizando el método `useState` provisto por la biblioteca. El estado local es una variable que contiene datos específicos del componente y se puede modificar mediante la función proporcionada por `useState`.

Aquí tienes un ejemplo básico de cómo se puede utilizar el estado local en un componente de React:

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

function MiComponente() {
  const [contador, setContador] = useState(0);

  const incrementarContador = () => {
    setContador(contador + 1);
  };

  return (
    <div>
      <p>Contador: {contador}</p>
      <button onClick={incrementarContador}>Incrementar</button>
    </div>
  );
}
```

En este ejemplo, el componente `MiComponente` tiene un estado local llamado `contador` que se inicializa en 0 utilizando `useState`. Luego se define una función `incrementarContador` que actualiza el estado `contador` al incrementarlo en 1 cuando se hace clic en el botón.

Cada vez que el estado local cambia, React se encarga de actualizar automáticamente la interfaz de usuario del componente para reflejar los cambios. Esto permite que los componentes de React sean reactivos y se actualicen dinámicamente en función de su estado interno.

Código de la clase

`src > components > TodoForm > index.js`

```

import React from "react";
import { TodoContext } from "../TodoContext";
import "./TodoForm.css";

function TodoForm() {
    const { addTodo, setOpenModal } = React.useContext(TodoContext);

    const onSubmit = (event) => {
        event.preventDefault();
        addTodo(newTodoValue);
        setOpenModal(false);
    };

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

    const onCancel = () => {
        setOpenModal(false);
    };

    const onChange = (event) => {
        setNewTodoValue(event.target.value);
    };

    return (
        <form onSubmit={onSubmit} action="">
            <label htmlFor="">Write a new ToDo</label>
            <textarea
                name=""
                id=""
                cols="30"
                rows="10"
                placeholder="Write something..."
                value={newTodoValue}
                onChange={onChange}
            />
            <div className="buttons">
                <button
                    type="button"
                    className="TodoForm cancel"
                    onClick={onCancel}
                >
                    Cancel
                </button>
                <button type="submit" className="TodoForm save">
                    Save
                </button>
            </div>
        </form>
    );
}

export { TodoForm };

```

```

import React from "react";
import { useLocalStorage } from "./useLocalStorage";

const TodoContext = React.createContext();

function TodoProvider({ children }) {
  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;

  const searchedTodos = todos.filter((todo) => {
    const todoText = todo.text.toLowerCase();
    const searchText = searchValue.toLowerCase();
    return todoText.includes(searchText);
  });

  const addTodo = (text) => {
    const newTodos = [...todos];

    newTodos.push({
      text,
      completed: false,
    });
    saveTodos(newTodos);
  };

  const completeTodo = (text) => {
    const newTodos = [...todos];
    const todoIndex = newTodos.findIndex((todo) => todo.text === text);

    // newTodos[todoIndex].completed = true;
    // true = false / false = true
    newTodos[todoIndex].completed = !newTodos[todoIndex].completed;
    saveTodos(newTodos);
  };

  const deleteTodo = (text) => {
    const newTodos = [...todos];
    const todoIndex = newTodos.findIndex((todo) => todo.text === text);
  };
}

```

```

        newTodos.splice(todoIndex, 1);
        saveTodos(newTodos);
    };

    return (
        <TodoContext.Provider
            value={{
                loading,
                error,
                completedTodos,
                totalTodos,
                searchValue,
                setSearchValue,
                searchedTodos,
                completeTodo,
                deleteTodo,
                openModal,
                setOpenModal,
                addTodo,
            }}
        >
            {children}
        </TodoContext.Provider>
    );
}

export { TodoContext, TodoProvider };

```

28. Despliegue de TODO Machine en GitHub Pages

Primero instalamos [GitHub Pages](#)

```
npm i --save-dev gh-pages
```

Entramos al archivo [package.json](#) y agregamos:

```
{
  "browserslist": {
    "development": [
      "last 1 chrome version",
      "last 1 firefox version",
      "last 1 safari version"
    ],
    "production": [
      ">0.2%",
      "not dead",
      "not op_mini all"
    ]
  }
}
```

```
},
"dependencies": {
  "react": "^18",
  "react-dom": "^18",
  "react-scripts": "^5.0.1",
  "web-vitals": "^2.1.4"
},
"eslintConfig": {
  "extends": [
    "react-app"
  ]
},
"homepage": ".", ✨
"name": "platzi-intro-react-base",
"scripts": {
  "start": "react-scripts start",
  "build": "react-scripts build",
  "predeploy": "npm run build", ✨
  "deploy": "gh-pages -d build", ✨
  "eject": "react-scripts eject"
},
"version": "0.1.0",
"devDependencies": {
  "gh-pages": "^6.0.0"
}
}
```

Ahora:

```
npm run build // Crea carpeta build
npm run deploy
```

Revisa tu repositorio en GitHub: [Settings > Pages](#) espera unos minutos y listo.



npm i --save-dev gh-pages



```
"scripts": {  
  "start": "react-scripts start",  
  "build": "react-scripts build",  
  "predeploy": "npm run build",  
  "deploy": "gh-pages -d build",  
  "eject": "react-scripts eject"  
},  
"homepage": ".">
```

npm run build

npm run deploy

Documentación

29. Presentación de proyectos para tu portafolio

Puedes cambiar la rama principal de tu repo:

Settings > Branches gh-pages

Añade un README.md

Aquí tienes un ejemplo de cómo podría ser un README.md para tu proyecto de ToDo List en GitHub:

```
# Todo List  
  
![Todo List](screenshot.png)  
  
Este es un proyecto de Todo List simple y fácil de usar. Te permite crear, organizar y realizar un seguimiento de tus tareas pendientes de manera eficiente.  
  
## Características  
  
- Agrega tareas con un título y una descripción.  
- Marca las tareas como completadas.  
- Elimina las tareas que ya no necesitas.  
- Filtra las tareas según su estado (completadas, pendientes).  
- Interfaz intuitiva y fácil de usar.
```

Tecnologías utilizadas

- HTML
- CSS
- JavaScript
- React

Demo

Puedes ver una demostración en vivo del proyecto [aquí](<https://tu-url-de-demo>).

Capturas de pantalla

![Captura de pantalla 1](Screenshot1.png)
![Captura de pantalla 2](Screenshot2.png)

Instalación

1. Clona este repositorio: `git clone https://tu-repositorio.git`
2. Abre el archivo `index.html` en tu navegador web.

Contribuciones

Las contribuciones son bienvenidas. Si tienes alguna idea para mejorar este proyecto, no dudes en abrir un problema o enviar una solicitud de extracción.

Licencia

Este proyecto está bajo la Licencia [MIT](LICENSE).

En este ejemplo, he incluido secciones como Características, Tecnologías utilizadas, Demo, Capturas de pantalla, Instalación, Contribuciones y Licencia. Puedes personalizar el contenido según las características específicas de tu proyecto y agregar cualquier otra sección que creas necesaria.

Recuerda reemplazar las URLs de las capturas de pantalla y la demostración en vivo con las URLs correspondientes a tu proyecto. Además, asegúrate de tener un archivo de captura de pantalla llamado `screenshot.png` (o cambia el nombre en el archivo README.md) en el mismo directorio que el archivo README.md.

- Un interesante proyecto
- Repo
- Otro diseño

Probar hosting: Vercel

🌐 Subir Tu Página Web a Internet ¡GRATIS! con Vercel ★

30. Diferencias entre versiones de React.js

Cambiemos la versión de React en el `package.json` de **18** a **17.0.2**:

```
{  
  "browserslist": {  
    "development": [  
      "last 1 chrome version",  
      "last 1 firefox version",  
      "last 1 safari version"  
    ],  
    "production": [  
      ">0.2%",  
      "not dead",  
      "not op_mini all"  
    ]  
},  
  "dependencies": {  
    "react": "^17.0.2",   
    "react-dom": "^17.0.2",   
    "react-scripts": "^5.0.1",  

```

Eliminemos la carpeta `node_modules` y también el `package-lock.json`:

```
rm -fr node_modules  
rm package-lock.json
```

Instalemos todo pero ahora con la versión especificada en el archivo `package.json`:

```
npm i
```

Corramos el proyecto y veamos lo que está fallando para corregir:

```
npm start
```

En búsqueda de soluciones

Algo que nos puede ayudar a encontrar la solución a los problemas de migrar un proyecto a una versión anterior es buscar:

- ¿Qué diferencias hay entre una versión en específico y la última?
- Change React 18 17.0.2

Info

Código de la clase

```
src > index.js
```

```
import React from "react";
import { render } from "react-dom";
import "./index.css";
import App from "./App";

const root = document.getElementById("root");
render(<App />, root);
```

¿Por qué es importante saber actualizar un proyecto a versiones anteriores y viceversa?

Por las siguientes razones:

1. Mantenimiento de compatibilidad: React es una biblioteca de JavaScript en constante evolución, y periódicamente se lanzan nuevas versiones con mejoras, correcciones de errores y nuevas características. Sin embargo, estas actualizaciones pueden introducir cambios en la API o comportamiento del código existente, lo que podría romper tu proyecto si no se actualiza correctamente. Saber cómo actualizar un proyecto de React a una nueva versión te permite aprovechar las mejoras y mantener tu código compatible con las últimas características y estándares.
2. Estabilidad y rendimiento: Las actualizaciones de React a menudo incluyen mejoras de rendimiento y correcciones de errores. Al actualizar tu proyecto a versiones más recientes, puedes beneficiarte de estas mejoras y asegurarte de que tu aplicación funcione de manera más eficiente y estable.

3. Comunidad y soporte: React cuenta con una gran comunidad de desarrolladores y una amplia gama de bibliotecas y herramientas complementarias. Estas comunidades y bibliotecas tienden a centrarse en las versiones más recientes de React, por lo que estar actualizado te permite aprovechar al máximo los recursos disponibles, obtener soporte y beneficiarte de las últimas soluciones y mejores prácticas.
4. Seguridad: Las versiones más recientes de React suelen incluir parches de seguridad que abordan vulnerabilidades conocidas. Al mantener tu proyecto actualizado, te aseguras de que estás utilizando una versión de React que tiene las correcciones de seguridad más recientes, lo que ayuda a proteger tu aplicación y los datos de los usuarios.

Sin embargo, también es importante saber cómo retroceder a versiones anteriores en algunos casos. Puede haber situaciones donde una actualización de React cause problemas de compatibilidad con ciertas dependencias o bibliotecas utilizadas en tu proyecto. Saber cómo volver a una versión anterior de React te permite solucionar estos problemas y mantener la estabilidad de tu aplicación hasta que los problemas de compatibilidad se resuelvan.

Retroceder a la versión 16 de React

Aquí hay algunos ajustes que podrías considerar al retroceder de la versión 17 a la versión 16 de React:

1. Ajustes en las importaciones: En la versión 17 de React, se introdujo una nueva forma de importar los módulos, utilizando el formato de "namespace import". En la versión 16, puedes volver a utilizar las importaciones normales de ES6. Asegúrate de actualizar las importaciones en tu código, por ejemplo, cambiando `import * as React from 'react'` a `import React from 'react'`.
2. Cambios en el uso de las refs: La forma de acceder a las refs en los componentes cambió entre la versión 16 y la versión 17. En la versión 16, puedes utilizar `ref={(ref) => this.myRef = ref}` para asignar una ref a un elemento. Asegúrate de ajustar cualquier código relacionado con el uso de refs en tu aplicación.
3. Revisar cambios específicos: La versión 17 de React introdujo algunos cambios específicos en el comportamiento y las API de React. Al retroceder a la versión 16, revisa los cambios entre las versiones y ajusta tu código en consecuencia.
4. Realizar pruebas exhaustivas: Después de realizar los ajustes, es importante realizar pruebas exhaustivas en tu aplicación para asegurarte de que todo funcione correctamente. Prueba todas las funcionalidades y verifica que no haya errores o problemas.

Recuerda que es recomendable seguir utilizando la versión más reciente de React, ya que cada versión trae mejoras, correcciones de errores y nuevas características. Retroceder a versiones anteriores puede llevar a problemas de compatibilidad y limitar las capacidades de tu aplicación.

31. ¿Cuándo realmente necesitas React.js?

React.js es una biblioteca de JavaScript que se utiliza ampliamente en el desarrollo de aplicaciones web y móviles. Aquí hay algunas situaciones en las que puede ser especialmente beneficioso utilizar React.js:

1. Aplicaciones de una sola página (Single-Page Applications, SPAs): React.js es ideal para construir SPAs, donde el contenido se carga de forma dinámica en una sola página sin necesidad de recargarla por completo. React permite crear interfaces de usuario interactivas y receptivas, lo que

mejora la experiencia del usuario al proporcionar una navegación fluida y una respuesta rápida a las acciones del usuario.

2. Interfaces de usuario complejas: Si estás construyendo una aplicación con una interfaz de usuario compleja y dinámica, React.js te permite dividir la interfaz en componentes más pequeños y reutilizables. Esto facilita la gestión del estado de la aplicación y mejora la organización y mantenimiento del código.
3. Actualización de partes específicas de la interfaz de usuario: React.js utiliza un enfoque basado en componentes y el concepto de Virtual DOM (DOM virtual). Esto permite actualizar solo las partes específicas de la interfaz de usuario que han cambiado, en lugar de volver a renderizar toda la página. Esto mejora el rendimiento y la eficiencia, especialmente en aplicaciones con un gran número de componentes.
4. Colaboración con otras bibliotecas o frameworks: React.js se puede utilizar junto con otras bibliotecas o frameworks, como Redux para la gestión del estado, React Router para la navegación, o incluso integrarse con frameworks de desarrollo móvil como React Native. Esto brinda flexibilidad y facilidad para aprovechar diferentes herramientas y tecnologías en tu proyecto.
5. Desarrollo de aplicaciones móviles: Si deseas desarrollar aplicaciones móviles nativas para iOS y Android, React Native, basado en React.js, es una opción poderosa. React Native te permite escribir código en JavaScript y compilarlo en código nativo, lo que facilita el desarrollo multiplataforma y el uso compartido de código entre plataformas.

En resumen, React.js es especialmente útil cuando necesitas crear interfaces de usuario dinámicas, interactivas y complejas, así como cuando deseas desarrollar aplicaciones web o móviles eficientes y de alto rendimiento.

Modelo MVP

El Modelo MVP (Modelo-Vista-Presentador) es un patrón de arquitectura de software que se utiliza para diseñar y organizar aplicaciones. Proporciona una separación clara de responsabilidades entre las diferentes capas de la aplicación, lo que facilita el mantenimiento, la reutilización de código y la prueba unitaria.

En el patrón MVP, los componentes principales son:

1. Modelo (Model): Representa los datos y la lógica de negocio de la aplicación. El modelo no se ocupa de la interfaz de usuario, sino que se enfoca en la manipulación y gestión de datos.
2. Vista (View): Es responsable de la presentación de la interfaz de usuario y la interacción con el usuario. La vista se encarga de recibir las interacciones del usuario y mostrar los datos proporcionados por el presentador.
3. Presentador (Presenter): Actúa como intermediario entre el modelo y la vista. El presentador recibe las acciones del usuario desde la vista, realiza las operaciones necesarias en el modelo y actualiza la vista con los datos correspondientes.

Ahora, veamos cómo se puede aplicar el patrón MVP en una aplicación de React:

1. Modelo: En una aplicación de React, el modelo puede consistir en componentes o clases que representan los datos y la lógica de negocio de la aplicación. Estos componentes no deben estar directamente relacionados con la interfaz de usuario, sino que se centran en la gestión de datos y en proporcionar métodos y funciones para manipularlos.
2. Vista: En React, los componentes pueden actuar como la vista en el patrón MVP. Estos componentes son responsables de la presentación de la interfaz de usuario y la interacción con el usuario. Reciben datos del presentador y los muestran en la interfaz de usuario. Además, también pueden enviar eventos o acciones al presentador cuando ocurren interacciones del usuario.
3. Presentador: El presentador en React puede ser un componente o una clase que se encarga de la lógica de presentación y comunicación entre la vista y el modelo. Recibe eventos o acciones de la vista y realiza las operaciones necesarias en el modelo. Luego, actualiza la vista con los datos correspondientes.

La comunicación entre la vista y el presentador en React se puede lograr a través de la propagación de props o mediante el uso de contextos o bibliotecas de administración de estados como Redux o MobX.

Al utilizar el patrón MVP en React, se logra una clara separación de responsabilidades entre la lógica de negocio, la presentación de la interfaz de usuario y la comunicación entre ellos. Esto facilita la comprensión, el mantenimiento y la extensibilidad del código, ya que cada componente tiene una única responsabilidad y se puede probar de manera aislada.

32. React con Create React App

Crearemos un proyecto desde cero, así que debes ubicarte en una carpeta donde puedas empezar este proyecto y abre la CLI:

```
npx create-react-app nombre-del-proyecto  
npx create-react-app cra-test ⏎
```

Ahora entramos a la carpeta creada y la abrimos con Visual Studio Code:

```
ls  
cd cra-test  
ll  
code ./ -r  
npm start
```

Con esto ya está todo listo para empezar a trabajar en tu proyecto.

Otros pasos

Para comenzar un proyecto con React utilizando Create React App, sigue los siguientes pasos:

1. Instala Node.js: Antes de comenzar, asegúrate de tener Node.js instalado en tu sistema. Puedes descargarlo e instalarlo desde el sitio oficial de Node.js (<https://nodejs.org>).

2. Instala Create React App: Una vez que tengas Node.js instalado, abre tu terminal o línea de comandos y ejecuta el siguiente comando para instalar Create React App de forma global:

```
npm install -g create-react-app
```

3. Crea un nuevo proyecto de React: En la ubicación donde deseas crear tu proyecto, ejecuta el siguiente comando para generar una nueva aplicación de React utilizando Create React App:

```
npx create-react-app my-app
```

Reemplaza "my-app" con el nombre que deseas para tu proyecto.

4. Espera a que se complete la instalación: Create React App descargará todas las dependencias necesarias y configurará la estructura básica de tu proyecto. Esto puede llevar unos minutos.

5. Accede al directorio del proyecto: Una vez que la instalación se haya completado, accede al directorio de tu proyecto ejecutando el siguiente comando:

```
cd my-app
```

Reemplaza "my-app" con el nombre de tu proyecto.

6. Inicia el servidor de desarrollo: Ahora, ejecuta el siguiente comando para iniciar el servidor de desarrollo y ver tu aplicación en el navegador:

```
npm start
```

Esto abrirá automáticamente tu aplicación en tu navegador predeterminado en la dirección <http://localhost:3000>.

¡Y eso es todo! Ahora tienes un proyecto de React configurado y listo para comenzar a desarrollar. Puedes editar los archivos en la carpeta `src` para construir tu aplicación de React y ver los cambios en tiempo real en el servidor de desarrollo.

Create React App proporciona una configuración predeterminada y herramientas útiles para el desarrollo de aplicaciones de React. Puedes encontrar más información sobre cómo personalizar y utilizar Create React App en la documentación oficial: <https://create-react-app.dev/docs/getting-started/>

Alternativas a Create React App

Además de Create React App, existen otras herramientas y alternativas populares para iniciar y configurar proyectos de React. Algunas de ellas son:

1. Next.js: Next.js es un framework de React que permite crear aplicaciones web de React con renderizado del lado del servidor (SSR) y generación de sitios estáticos. Proporciona una configuración optimizada y muchas características adicionales, como enruteamiento incorporado, pre-renderizado, manejo de API, entre otros.
2. Gatsby: Gatsby es otro framework popular de React que se utiliza para crear sitios web estáticos y de alto rendimiento. Gatsby combina React con GraphQL y proporciona una amplia gama de características, como optimización de rendimiento, pre-renderizado, enruteamiento y una gran cantidad de complementos para ampliar su funcionalidad.
3. Parcel: Parcel es un empaquetador de módulos web que puede ser utilizado para proyectos de React. Es fácil de usar y no requiere una configuración compleja. Parcel se encarga de la compilación y el empaquetado de los archivos de tu proyecto, lo que simplifica el proceso de configuración inicial.
4. webpack: webpack es una herramienta de construcción muy popular y altamente personalizable. Puede ser configurado para manejar proyectos de React y proporciona una amplia gama de características, como el empaquetado de módulos, la optimización de código, el manejo de assets y el soporte para una variedad de loaders y plugins.

Estas son solo algunas de las alternativas disponibles para iniciar proyectos de React. Cada herramienta tiene sus propias características y enfoques, por lo que es recomendable investigar y evaluar cuál se adapta mejor a las necesidades específicas de tu proyecto.

33. React con Next.js

Crearemos un proyecto desde cero, así que debes ubicarte en una carpeta donde puedas empezar este proyecto y abre la CLI:

```
npx create-next-app@latest next-test
y
No
No
No

cd next-test
code ./ -r
npm run dev
```

Esto te mostrará en consola lo siguiente, solo debes hacer un **Ctrl + Click** sobre el enlace <http://localhost:3000>:

```
> next-test@0.1.0 dev
> next dev
```

```
- ready started server on [::]:3000, url: http://localhost:3000 📲🌐💡
Attention: Next.js now collects completely anonymous telemetry regarding usage.
🕒🕒🕒
This information is used to shape Next.js' roadmap and prioritize features.
You can learn more, including how to opt-out if you'd not like to participate in this anonymous program, by visiting the following URL:
https://nextjs.org/telemetry
```

Otros pasos

Para comenzar un proyecto con React utilizando Next.js, sigue los siguientes pasos:

1. Instala Node.js: Antes de comenzar, asegúrate de tener Node.js instalado en tu sistema. Puedes descargarlo e instalarlo desde el sitio oficial de Node.js (<https://nodejs.org>).
2. Crea un nuevo proyecto de Next.js: Abre tu terminal o línea de comandos y ejecuta el siguiente comando para crear un nuevo proyecto de Next.js:

```
npx create-next-app my-app
```

Reemplaza "my-app" con el nombre que deseas para tu proyecto.

3. Espera a que se complete la instalación: Next.js descargará todas las dependencias necesarias y configurará la estructura básica de tu proyecto. Esto puede llevar unos minutos.
4. Accede al directorio del proyecto: Una vez que la instalación se haya completado, accede al directorio de tu proyecto ejecutando el siguiente comando:

```
cd my-app
```

Reemplaza "my-app" con el nombre de tu proyecto.

5. Inicia el servidor de desarrollo: Ahora, ejecuta el siguiente comando para iniciar el servidor de desarrollo y ver tu aplicación en el navegador:

```
npm run dev
```

Esto abrirá automáticamente tu aplicación en tu navegador predeterminado en la dirección <http://localhost:3000>.

¡Y eso es todo! Ahora tienes un proyecto de React configurado con Next.js y listo para comenzar a desarrollar. Puedes editar los archivos en la carpeta **pages** para construir tus páginas de React y ver los cambios en tiempo real en el servidor de desarrollo.

Next.js proporciona un conjunto de características adicionales, como el renderizado del lado del servidor (SSR), la generación de sitios estáticos, el enrutamiento incorporado, la carga de datos inicial, entre otros. Puedes encontrar más información sobre cómo utilizar y personalizar Next.js en la documentación oficial: <https://nextjs.org/docs/getting-started>

Next.js es especialmente útil para construir aplicaciones web de React con rendimiento optimizado y características avanzadas, como la carga de datos previa a la renderización y la generación de sitios estáticos.

Server-side rendering

Server-side rendering (SSR), también conocido como renderizado del lado del servidor, es un enfoque en el cual la generación inicial de la interfaz de usuario de una aplicación web se realiza en el servidor y se envía al cliente como HTML completo. En lugar de enviar solo el código JavaScript de la aplicación y esperar a que se ejecute en el navegador para generar la interfaz de usuario, SSR permite que el servidor envíe una versión ya generada de la interfaz de usuario al cliente.

En un proceso de SSR típico:

1. El cliente realiza una solicitud al servidor para una página web.
2. El servidor ejecuta la lógica de la aplicación y genera la interfaz de usuario en HTML.
3. El servidor envía el HTML completamente renderizado al cliente.
4. El cliente recibe el HTML y lo muestra en el navegador.
5. El cliente también puede recibir el código JavaScript necesario para la interactividad adicional de la aplicación.

La principal ventaja del SSR es que permite que el cliente vea rápidamente el contenido completo de la página, incluso antes de que se cargue y ejecute el código JavaScript de la aplicación. Esto mejora la velocidad de carga percibida y la accesibilidad, especialmente en conexiones de red más lentas o dispositivos menos potentes.

Además, el SSR también puede mejorar el SEO (Optimización para Motores de Búsqueda) al permitir que los motores de búsqueda indexen y comprendan mejor el contenido de la página en su formato HTML completo.

Sin embargo, el SSR también puede tener algunas limitaciones y desafíos. El renderizado en el servidor puede requerir más recursos computacionales en el servidor, y las interacciones dinámicas en la interfaz de usuario pueden requerir una comunicación adicional entre el cliente y el servidor.

En resumen, el server-side rendering (SSR) es un enfoque en el que la generación inicial de la interfaz de usuario se realiza en el servidor y se envía al cliente como HTML completo, lo que mejora la velocidad de carga percibida y la accesibilidad de la aplicación web.

34. React con Vite

```
npm -v
npm create vite@latest vite-test
  - ✓ Select a framework: » React
```

```
- ✓ Select a variant: » JavaScript

// Usar este comando en caso de tener npm desactualizado
npm create vite@latest vitetest -- --template react

// Luego
cd vite-test
npm i
code ./ -r
npm run dev
```

Otros pasos

Para comenzar un proyecto con React utilizando Vite, sigue los siguientes pasos:

1. Instala Node.js: Antes de comenzar, asegúrate de tener Node.js instalado en tu sistema. Puedes descargarlo e instalarlo desde el sitio oficial de Node.js (<https://nodejs.org>).
2. Crea un nuevo proyecto de React con Vite: Abre tu terminal o línea de comandos y ejecuta el siguiente comando para crear un nuevo proyecto de React con Vite:

```
npx create-vite my-app --template react
```

Reemplaza "my-app" con el nombre que deseas para tu proyecto.

3. Espera a que se complete la instalación: Vite descargará todas las dependencias necesarias y configurará la estructura básica de tu proyecto. Esto puede llevar unos minutos.
4. Accede al directorio del proyecto: Una vez que la instalación se haya completado, accede al directorio de tu proyecto ejecutando el siguiente comando:

```
cd my-app
```

Reemplaza "my-app" con el nombre de tu proyecto.

5. Inicia el servidor de desarrollo: Ahora, ejecuta el siguiente comando para iniciar el servidor de desarrollo y ver tu aplicación en el navegador:

```
npm run dev
```

Esto abrirá automáticamente tu aplicación en tu navegador predeterminado en la dirección <http://localhost:3000>.

¡Y eso es todo! Ahora tienes un proyecto de React configurado con Vite y listo para comenzar a desarrollar. Puedes editar los archivos en la carpeta `src` para construir tu aplicación de React y ver los cambios en tiempo real en el servidor de desarrollo.

Vite es una herramienta de desarrollo rápida que permite un tiempo de compilación instantáneo y una experiencia de desarrollo optimizada. Proporciona un servidor de desarrollo rápido, soporte para módulos ES y una configuración mínima necesaria.

Puedes encontrar más información sobre cómo utilizar y personalizar Vite en la documentación oficial: <https://vitejs.dev/>

Examen ✎

- Haz clic para ver los resultados ⓘ

a

La forma de recibir/escuchar/reaccionar ante las acciones o interacciones de los usuarios en nuestra aplicación.

b ✎

La forma de comunicar componentes entre sí para transportar información.

c

La forma en que React guarda información de nuestro componente para escuchar cuando tenga cambios y disparar un nuevo render.

2. ¿Cuál de las siguientes es una forma o herramienta válida para trabajar proyectos con React.js?

a

Configuración personalizada con Webpack.

b

Create React App

c ✎

Todas las respuestas son correctas.

d

Vite

e

Next.js

3. ¿Cómo podemos enviar información de un componente "abuelo" a un componente "nieto" sin necesidad de pasar las props por el componente "hijo/padre"?

a

Usando React State.

b

Usando React Props.

c ✎

Usando React Context.

d

Usando React Portals.

4. ¿Cómo escuchamos cuando los usuarios envíen un formulario con React?

a

`NombreComponente.addEventListener('submit')`

b

`onsubmit`

c

`submit`

d

`NombreComponente.addEventListener('formsubmit')`

e ✎

`onSubmit`

5. ¿Para qué sirve React Context?

a

Para comunicar componentes entre sí a pesar de tener componentes padres diferentes.

b ✎

Para comunicar componentes sin tener que pasar la información como props por cada componente intermedio.

c

Para teletransportar componentes a un nodo de HTML distinto al nodo donde hace render el resto de la aplicación.

d

Para teletransportar componentes a un documento HTML distinto a donde hace render el resto de la aplicación.

6. ¿Cómo creamos un contexto en React?

a 

`React.createContext`

b

`ReactDOM.createContext`

c

`ReactDOM.context`

d

`React.context`

7. ¿Qué son los eventos en React?

a

La forma de recibir/escuchar/reaccionar ante los renders de nuestros componentes.

b 

La forma de recibir/escuchar/reaccionar ante las acciones o interacciones de los usuarios en nuestra aplicación.

c

La forma de recibir/escuchar/reaccionar ante los cambios en el estado de nuestros componentes.

8. ¿Qué es React.js?

a

Solo una libreria.

b 

React es tanto una librería como una arquitectura.

c

Solo una arquitectura.

9. ¿Qué significa el "ecosistema de React"?

a

Todas las herramientas open-source (únicamente las oficiales) relacionadas con React.

b 

Todas las herramientas open-source (oficiales y no oficiales) relacionadas con React.

c

Algunas de las herramientas internas que usa React para construir su código fuente.

10. ¿Cómo escuchamos el evento de click en un botón con React?

a

`onclick`

b 

`onClick`

c

`click`

d

`NombreComponente.addEventListener('click')`

11. ¿Qué es el estado en React?

a 

La forma en que React guarda información de nuestro componente para escuchar cuando tenga cambios y disparar un nuevo render.

b

La forma de recibir/escuchar/reaccionar ante las acciones o interacciones de los usuarios en nuestra aplicación.

c

La forma de comunicar componentes entre sí para transportar información.

12. ¿Qué es JSX?

a 

Una sintaxis especial de JavaScript para escribir elementos y componentes de React que se siente como HTML.

b

Una versión muy futura de ECMAScript que nos permite escribir React con superpoderes de HTML.

c

Una sintaxis especial de HTML para escribir elementos y componentes de React que se siente como JavaScript.

d

Una versión muy futura de ECMAScript que nos permite escribir React con superpoderes de JavaScript.

13. ¿Podemos crear más de un estado en nuestros componentes de React?

a

Falso

b ✅

Verdadero

14. ¿Para qué sirven los efectos en React?

a

Para guardar información de nuestro componente, escuchar cuando tenga cambios y disparar un nuevo render.

b ✅

Para ejecutar bloques de código en componentes únicamente si se cumplen ciertas condiciones en cada nuevo render.

c

Para ejecutar bloques de código que requieren asincronismo dentro de los componentes de nuestra aplicación.

d

Para ejecutar bloques de código dentro de los componentes únicamente cuando los usuarios realicen cualquier acción o interacción.

15. ¿Cómo escuchamos cuando un usuario escribe en un input o textarea con React?

a

`oninputchange`

b

`onchange`

c

`change`

d

onWrite

e ✎

onChange

16. ¿Cómo creamos un portal en React?

a

React.createPortal

b ✎

ReactDOM.createPortal

17. ¿Cuál es la diferencia entre componentes y elementos en React?

a

Los elementos se crean con clases que extienden de React.Component. Los componentes son funciones que pueden usar React Hooks.

b ✎

Los componentes son grupos de elementos, reciben props y pueden crear estado o efectos. Los elementos reciben atributos o eventos y (casi siempre) se transforman en etiquetas de HTML.

c

Los componentes se crean con clases que extienden de React.Component. Los elementos son funciones que pueden usar React Hooks.

d

Los elementos son grupos de componentes, reciben props y pueden crear estado o efectos. Los componentes reciben atributos o eventos y (casi siempre) se transforman en etiquetas de HTML.

18. ¿Qué propiedad debemos enviarle al Provider de un contexto en React para consumirlo desde su respectivo Consumer?

a

context

b

Todas las propiedades enviadas al componente Provider podrán consumirse desde el componente Consumer.

c ✎

value

d

state

19. ¿Cómo usamos React Context con la sintaxis de React Hooks?

a ✘

`useContext(Contexto)`

b

`useContext(Contexto.Provider)`

c

`useContext("NombreDelContexto")`

d

`useContext(Contexto.Consumer)`

20. ¿Cuál de las siguientes es una forma VÁLIDA de crear un estado en React?

a

```
const { nombreDelEstado, setNombreDelEstado } = React.useState("valor inicial de estado");
```

b ✘

```
const [nombreDelEstado, setNombreDelEstado] = React.useState("valor inicial de estado");
```

c

```
const nombreDelEstado = React.useState("valor inicial de estado");
const setNombreDelEstado = nombreDelEstado.setState();
```

21. ¿Para qué sirven los portales en React?

a

Para teletransportar componentes a un documento HTML distinto a donde hace render el resto de la aplicación

b

Para comunicar componentes sin tener que pasar la información como props por cada componente intermedio.

c ✎

Para teletransportar componentes a un nodo de HTML distinto al nodo donde hace render el resto de la aplicación.

d

Para comunicar componentes entre sí a pesar de tener componentes padres diferentes.

22. ¿Cuál de los siguientes bloques de código ejecuta nuestro efecto únicamente la primera vez que se renderiza nuestro componente?

a

```
React.useEffect(() => { console.log("Efectito"); });
```

b

```
React.useEffect(() => { console.log("Efectito"); }, window);
```

c ✎

```
React.useEffect(() => { console.log("Efectito"); }, []);
```

d

```
React.useEffect(() => { console.log("Efectito"); },
document.addEventListener('load'));
```

23. ¿Por qué debemos compilar nuestro proyecto con React.js antes de subirlo a GitHub Pages?

a

No es obligatorio compilar nuestro proyecto antes de subirlo a GitHub Pages.

b ✎

Porque GitHub Pages solo nos permite desplegar aplicaciones estáticas.

c

Porque GitHub Pages tiene muy poco espacio de almacenamiento.

d

Porque GitHub Pages no soporta interacciones de los usuarios.