

Sesión # 6 Componente Práctico

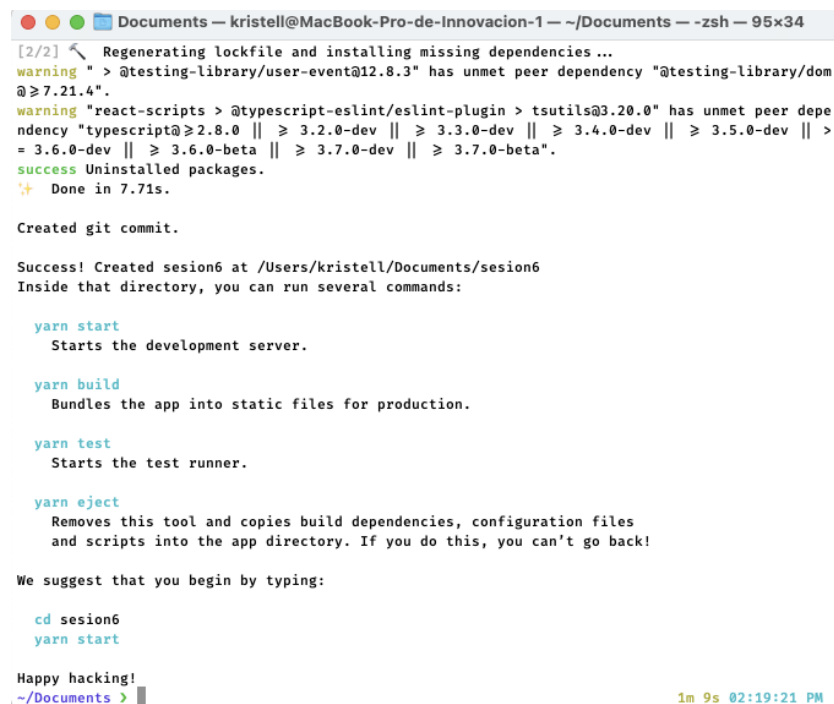
Desarrollo de Front-End web con React

Como ya conoces, React es una biblioteca Javascript para crear interfaces de usuario. Sigue los siguientes pasos para crear nuestra primera aplicación web con esta herramienta. ¡Vamos a construir una app para crear listas de pendientes!

1. Crea la configuración necesaria para un proyecto con React

- Asegúrate de tener una versión reciente de [Node.js](#) instalada. Para validar si la tienes instalada, ejecuta desde consola el comando: `node -v`
- Sigue las [instrucciones de instalación de Create React App](#) para hacer un nuevo proyecto.

- Desde la terminal ejecuta:
`npx create-react-app sesion6`
- Una vez cargue, deberás obtener un resultado así:



```
[2/2] ⚡ Regenerating lockfile and installing missing dependencies ...
warning " > @testing-library/user-event@12.8.3" has unmet peer dependency "@testing-library/dom
@ > 7.21.4".
warning "react-scripts > @typescript-eslint/eslint-plugin > tsutils@3.20.0" has unmet peer depe
ndency "typescript@>=2.8.0 || >= 3.2.0-dev || >= 3.3.0-dev || >= 3.4.0-dev || >= 3.5.0-dev || >
= 3.6.0-dev || >= 3.6.0-beta || >= 3.7.0-dev || >= 3.7.0-beta".
success Uninstalled packages.
Done in 7.71s.

Created git commit.

Success! Created sesion6 at /Users/kristell/Documents/sesion6
Inside that directory, you can run several commands:

  yarn start
    Starts the development server.

  yarn build
    Bundles the app into static files for production.

  yarn test
    Starts the test runner.

  yarn eject
    Removes this tool and copies build dependencies, configuration files
    and scripts into the app directory. If you do this, you can't go back!

We suggest that you begin by typing:

  cd sesion6
  yarn start

Happy hacking!
~/Documents > 1m 9s 02:19:21 PM
```

- Accede a la carpeta del proyecto desde tu editor de texto (VScode)
 - Elimina todos los archivos que se encuentran dentro la carpeta `src/` del nuevo proyecto. No la carpeta.
- Agrega un archivo llamado `index.css` en la carpeta `src/` con [este código CSS](#).

- d. Agrega un archivo llamado index.js en la carpeta src/ con [este código JS](#).
- e. Agrega un archivo llamado App.jsx (jsx no es más que JavaScript con una estructura para la representación de componentes) en la carpeta src/ con el siguiente código:

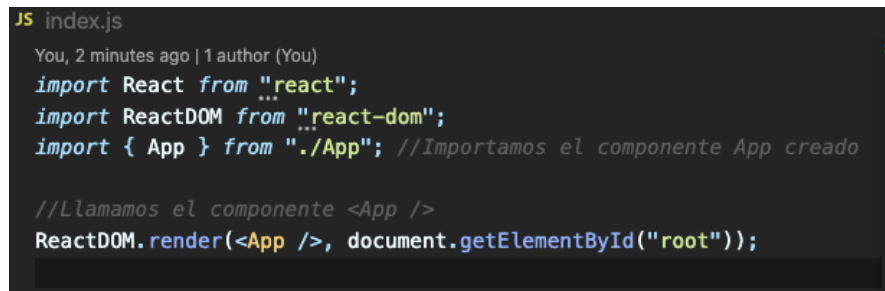
```
import React from "react";

export function App() {

  return <div>Hola Mundo</div>;

}
```

- f. El archivo creado en el paso anterior corresponde a un componente, hagamos el llamado de él dentro de index.js. Para ello, importa el archivo y añadelo en el render. Deberás obtener algo así:



```
JS index.js
You, 2 minutes ago | 1 author (You)
import React from "react";
import ReactDOM from "react-dom";
import { App } from "../App"; //Importamos el componente App creado

//Llamamos el componente <App />
ReactDOM.render(<App />, document.getElementById("root"));
```

- g. Accede desde consola a la carpeta del proyecto y ejecuta npm start
- h. Abre <http://localhost:3000> en el navegador, deberás ver el mensaje 'Hola Mundo'

2. Fundamentos de react

- a. Este código inicial es la base de lo que estás construyendo Ya cuentas con los estilos de CSS así que solo necesitas enfocarte en aprender React y programar nuestra app para listar tareas.
- b. Dentro de la carpeta src/ crea un nuevo folder al que llamaremos 'components' y dentro crea dos nuevos archivos TodoList.jsx y TodoItem.jsx

- c. Dentro del archivo `TodoList`, añade las siguientes líneas:

```
components >  TodoList.jsx > ...  
You, seconds ago | 1 author (You)  
import React from "react";  
  
export function TodoList() {  
  return (  
    <div>  
  
    </div>  
  );  
}
```

- d. Inspeccionando el código del archivo `TodoList.jsx`, notarás que tenemos una función que retorna un `div`. Vamos a reemplazar este elemento `div` por `` porque trabajaremos con una lista y recibiremos como parámetros de la función un arreglo llamado 'listas' que luego vamos a recorrer y devolver ítems de la lista. Replica el siguiente código:

```
components >  TodoList.jsx > ...  
You, seconds ago | 1 author (You)  
import React from "react";  
  
export function TodoList({ listas }) {  
  return (  
    <ul>  
      {listas.map((lista) => (  
        <li></li>  
      ))}  
    </ul>  
  );  
}
```

- e. Vamos a pasar algo de datos de nuestro componente `App` a nuestro componente `TodoList` y para ello usaremos 'Props'. En el archivo `App.jsx` importa el componente `TodoList` y cambia el código para pasar una prop llamada `listas` al `TodoList`:

```
App.jsx > ...
You, seconds ago | 1 author (You)
import React from "react";
import { TodoList } from "../components/TodoList";

export function App() {
  return <TodoList listas=[] />;
}
```

Por el momento no se visualizará nada porque el arreglo se encuentra vacío, para añadir un elemento crea un ítem dentro de la lista desde TodoList.jsx

```
components > TodoList.jsx > ...
You, seconds ago | 1 author (You)
import React from "react";

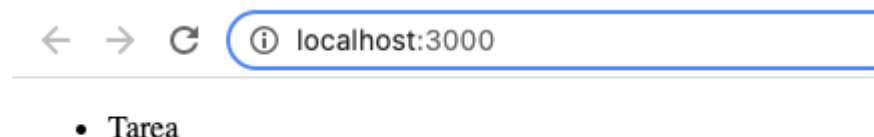
export function TodoList({ listas }) {
  return (
    <ul>
      {listas.map((lista) => (
        <li>Tarea</li>
      ))}
    </ul>
  );
}
```

Y luego añade un elemento añadelo en el arreglo enviado desde App.jsx

```
App.jsx > ...
You, a minute ago | 1 author (You)
import React from "react";
import { TodoList } from "../components/TodoList";

export function App() {
  return <TodoList listas=[{ id: 1, task: "Tarea 1", completed: false }] />;
}
```

- f. Guarda tus cambios y observa ahora en el localhost:3000 el ítem creado



¡Felicidades! Acabas de “pasar una prop” de un componente padre App a un componente hijo TodoItem. Pasando props es cómo la información fluye en apps de React, de padres a hijos.

g. Ahora, vamos a utilizar estados para permitir el re-renderizado de la información en caso de que existan cambios, es decir se añadan nuevas tareas/items a la lista. Para ello,

- En el archivo App.jsx, importa el hook useState que permitirá que los componentes tengan estados.
- Replica el siguiente código

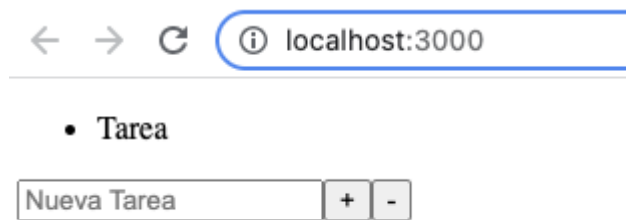
```
App.jsx > ...  
  
import React, { useState } from "react";  
import { TodoList } from "../components/TodoList";  
  
export function App() {  
  // listas: estado  
  // setListas: modificador del estado  
  const [listas, setListas] = useState([  
    { id: 1, task: "Tarea ", completed: false },  
  ]);  
  return <TodoList listas={listas} />;  
}
```

h. Ahora, para añadir nuevas tareas a nuestro listado requerimos contar con un elemento input y un botón para insertarlas. Modifica tu código del archivo App.jsx de la siguiente manera.

```
import React, { Fragment, useState } from "react";  
import { TodoList } from "../components/TodoList";  
  
export function App() {  
  // listas: estado  
  // setListas: modificador del estado  
  const [listas, setListas] = useState([  
    { id: 1, task: "Tarea ", completed: false },  
  ]);  
  return (  
    // Fragment se utiliza como padre para englobar varios elementos.  
    <Fragment>  
      <TodoList listas={listas} />  
      <input type="text" placeholder="Nueva Tarea" />  
      <button>+</button>  
      <button>-</button>  
    </Fragment>  
  );  
}
```

Como puedes notar, añadimos los elementos input y button dentro de un padre Fragment y utilizamos el signo '+' para indicar la acción de Añadir y el signo menos '-' para la acción de Eliminar.

- i. Guarda los cambios y verifica los resultados en localhost:3000



- j. Ahora, accede al archivo TodoItem.jsx y crea aquí el componente hijo que será llamado desde TodoList. TodoItem contendrá una función que recibe un objeto tarea y creará un ítem de la lista con dicha información. Replica el siguiente código en el archivo TodoItem.jsx

```
import React from "react";

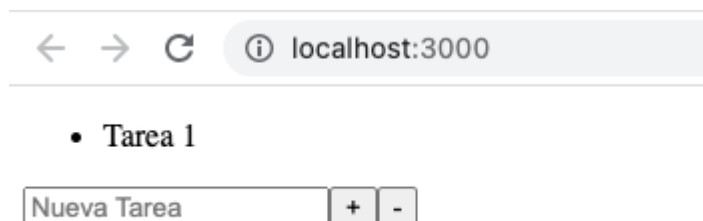
export function TodoItem({ lista }) {
  const { id, task, completed } = lista;
  return <li>{task}</li>;
}
```

- k. Llamamos el componente TodoItem por cada elemento de nuestro TodoList, para ello importa el componente recientemente creado y renderízalo

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

export function TodoList({ listas }) {
  return (
    <ul>
      {listas.map((lista) => (
        <TodoItem key={lista.id} lista={lista} />
      ))}
    </ul>
  );
}
```

- l. Guarda tus cambios y valida el resultado. Como puedes notar, ahora obtienes directamente la información del objeto creado en App.jsx



m. Ahora, si deseas añadir nuevas tareas, es necesario crear un evento en App.jsx que escuche la data al momento de hacer click en el botón de añadir y ejecute una función a la que llamaremos handleTaskAdd.

- Añade una referencia para obtener la data del input
- Implementa el evento onClick en el button de añadir que haga llamado a la función handleTaskAdd

```
return (  
  // Fragment se utiliza como padre para englobar varios elementos.  
  <Fragment>  
    <TodoList listas={listas} />  
    <input ref={taskRef} type="text" placeholder="Nueva Tarea" />  
    <button onClick={handleTaskAdd}>+</button>  
    <button>-</button>  
  </Fragment>  
);
```

- Apaga la consola donde está corriendo la app (donde habilitaste npm start) e instala la librería uuid para la generación aleatoria de ids, esta la usaremos para la creación automática de ids de nuestras tareas. Para ello, ejecuta en terminal el comando: yarn add uuid

```
> yarn add uuid  
yarn add v1.22.13  
[1/4] Resolving packages...  
[2/4] Fetching packages...  
warning Pattern ["uuid@8.3.2"] is trying to unpack in the same destination "/Users/kristell/Library/Caches/Yarn/v6/npm-uuid-8.3.2-80d5b5ced271bb9af6c445f21a1a04c606cefb2-integrity/node_modules/uuid" as pattern ["uuid@8.3.0"]. This could result i  
n non-deterministic behavior, skipping.  
[3/4] Linking dependencies...  
warning " > @testing-library/user-event@12.8.3" has unmet peer dependency "@testing-library/dom@>=7.21.4".  
warning "react-scripts > @typescript-eslint/eslint-plugin > tsutils@3.20.0" has unmet peer dependency "typescript@>=2.8.0 ||  
>= 3.2.0-dev || >= 3.3.0-dev || >= 3.4.0-dev || >= 3.5.0-dev || >= 3.6.0-dev || >= 3.6.0-beta || >= 3.7.0-dev || >= 3.7.0-b  
eta".  
[4/4] Building fresh packages...  
success Saved lockfile.  
success Saved 1 new dependency.  
info Direct dependencies  
└─ uuid@8.3.2  
info All dependencies  
└─ uuid@8.3.2  
Done in 33.04s.
```

- Ejecuta nuevamente npm start.
- Realiza las importaciones necesarias en App.jsx

```
import React, { Fragment, useState, useRef } from  
"react";  
import { v4 as uuidv4 } from "uuid";  
import { TodoList } from "../components/TodoList";
```

- Crea el método handleTaskAdd

```
//Referencia para obtener a la data ingresada y usarla en el handle
const taskRef = useRef();

//Método para añadir tareas
const handleTaskAdd = () => {
  //arrow function
  const task = taskRef.current.value;
  // En caso de que la data sea vacia no realizamos nada
  if (task === "") return;

  //En caso de recibir información, creamos un nuevo
  //elemento y hacemos cambios sobre el estado
  setListas((prevTasks) => {
    return [...prevTasks, { id: uuidv4(), task, completed: false }];
  });
  taskRef.current.value = null; //Limpia el input cuando se añade
};
```

- n. Valida los cambios realizados y comprueba que ya será posible crear nuevas tareas al ingresar información en el input y presionar el botón añadir (+)

← → ↻ ⓘ localhost:3000

- Tarea 1
- Hola
- Prueba

[Archivo App.jsx hasta el momento](#)

- o. Siguiendo con nuestro proyecto, ahora vamos a añadir un input de tipo checkbox en cada tarea de nuestra lista (TodoItem) para marcar su estado es decir, si ya fue completada o no.

```
import React from "react";

export function TodoItem({ lista }) {
  const { id, task, completed } = lista;
  return (
    <li>
      <input type="checkbox">{task}</input>
    </li>
  );
}
```


3. Añade la propiedad `toggleTask` en el listado de tareas que se llama desde `App.jsx` y crea una nueva función `toggleTask` que maneje este elemento (toggle/checkbox). Tu archivo `App.jsx` quedará así hasta el momento:

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

import { v4 as uuidv4 } from "uuid";

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

export function App() {

  const [listas, setListas] = useState([

    { id: 1, task: "Tarea 1", completed: false },

  ]);

  const taskRef = useRef();

  const toggleTask = (id) => {

    //copia de las tareas

    const newTasks = [...listas];

    //encontrar tarea seleccionada, según su id

    const task = newTasks.find((task) => task.id === id);

    task.completed = !task.completed; // si es true se convierte en
false, si es false se convierte en true

    setListas(newTasks); //actualizamos el listado de tareas

  };

  //Método para añadir tareas

  const handleTaskAdd = () => {
```

```

const task = taskRef.current.value;

if (task === "") return;

setListas((prevTasks) => {
  return [...prevTasks, { id: uuidv4(), task, completed: false }];
});

taskRef.current.value = null; //Limpia el input cuando se añade
};

return (
  <Fragment>
    <TodoList listas={listas} toggleTask={toggleTask} />
    <input ref={taskRef} type="text" placeholder="Nueva Tarea" />
    <button onClick={handleTaskAdd}>+</button>
    <button>-</button>
  </Fragment>
);
}

```

4. Modifica los archivos TodoList y TodoItem para añadir el nuevo parámetro creado de toggleTask

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

export function TodoList({ listas, toggleTask }) {
  return (
    <ul>
      {listas.map((lista) => (
        <TodoItem key={lista.id} lista={lista} toggleTask={toggleTask} />
      ))}
    </ul>
  );
}
```

```
import React from "react";

export function TodoItem({ lista, toggleTask }) {
  const { id, task, completed } = lista;

  const handleTaskClick = () => {
    toggleTask(id);
  };

  return (
    <li>
      <input type="checkbox" checked={completed} onChange={handleTaskClick} />
      {task}
    </li>
  );
}
```

Como puedes notar se pasan las propiedades de padre a hijos (del componente de más arriba al más de abajo), además se evidencia la creación de una nueva función `handleTaskClick` en el archivo `TodoItem.jsx` la cual nos permitirá llamar la función `toggleTask` creada en `App.jsx` y hacer su llamado posteriormente en el método `onChange`.

- Ahora, nos hace falta crear la función para eliminar tareas de nuestro listado. Para ello en el archivo `App.jsx` crea una llamada `handleClearAll` y crea un evento en el botón de eliminar para hacer su llamado.

```
//Método para eliminar tareas
const handleClearAll = () => {
  //Hacemos una copia de las tareas creadas y
  // filtramos por aquellas que han sido seleccionadas
  const newTasks = listas.filter((task) => !task.completed);
  // Utilizamos setListas para setear los elementos,
  setListas(newTasks);
};
```

```
return (
  // Fragment se utiliza como padre para englobar
  <Fragment>
    <TodoList listas={listas} toggleTask={toggleTask}>
      <input ref={taskRef} type="text" placeholder="Nueva Tarea" />
      <button onClick={handleTaskAdd}>+</button>
      <button onClick={handleClearAll}>-</button>
    </Fragment>
  </div>
);
```

6. Como puedes ver, ya es posible remover tareas de nuestra lista. Pero podemos notar que al refrescar el sitio se borran las tareas existentes, para ello usemos un hook muy útil que nos permitirá almacenar los datos en nuestro local storage. Importa from react el hook useEffect y haz uso de él para almacenar en local las tareas creadas en la lista y para visualizar aquellas que ya se encuentran creadas al recargar el sitio.

```
//Para escuchar y guardar las nuevas tareas creadas
useEffect(() => {
  localStorage.setItem("listApp.lists", JSON.stringify(listas));
}, [listas]);

//Para visualizar aquellas tareas que ya se encuentren creadas
useEffect(() => {
  //Obtener tareas guardadas
  const storedTasks = JSON.parse(localStorage.getItem("listApp.lists"));
  //Validar que existan,
  if (storedTasks) {
    setListas(storedTasks);
  }
}, []);
```

7. Valida el resultado final

←
→
↻
📄 localhost:3000

- ☐ Tarea 1
- ☐ Hacer ejercicio
- ☐ Estudiar
- ☐ Enviar correo mintic

-
- ☐ Tarea 1
 - ☐ Hacer ejercicio
 - ☒ Estudiar
 - ☐ Enviar correo mintic

Te quedan 3 tareas por terminar

Solución completa:

<https://github.com/Misionic-Ciclo-4A/sesion6-solucion>