

Curso de React

Diego Saavedra, Paulina Collaguazo

Jul 10, 2024

Table of contents

| | | |
|----------|--|-----------|
| 1 | Bienvenido | 4 |
| 1.1 | ¿De qué trata este curso? | 4 |
| 1.2 | ¿Para quién es este curso? | 4 |
| 1.3 | ¿Cómo contribuir? | 5 |
| I | Unidad 1: Introducción a React | 6 |
| 2 | Introducción a React. | 7 |
| 2.1 | Instalación de React | 7 |
| 3 | Hola Mundo en React | 12 |
| 3.1 | Crear un proyecto Vite | 12 |
| 3.2 | Crear un componente | 13 |
| 3.3 | Retos | 14 |
| 3.4 | Conclusiones | 16 |
| 4 | Estructura de carpetas en React | 17 |
| 4.1 | Estructura de carpetas | 17 |
| 4.1.1 | node_modules | 18 |
| 4.1.2 | public | 18 |
| 4.1.3 | src | 18 |
| 4.1.4 | assets | 19 |
| 4.1.5 | App.css | 19 |
| 4.1.6 | App.jsx | 19 |
| 4.1.7 | index.css | 19 |
| 4.1.8 | main.jsx | 19 |
| 4.1.9 | .eslintrc.cjs | 19 |
| 4.1.10 | .gitignore | 19 |
| 4.1.11 | index.html | 19 |
| 4.1.12 | package-lock.json | 20 |
| 4.1.13 | package.json | 20 |
| 4.1.14 | README.md | 20 |
| 4.1.15 | vite.config.js | 20 |
| 4.2 | Conclusión | 20 |
| 5 | ¿Qué es JSX? | 21 |
| 6 | ¿Por qué JSX? | 22 |
| 7 | Ejemplos: | 23 |

| | | |
|-----------|---|-----------|
| 8 | ¿Cuál es la diferencia con retornar algo en JSX que en JS? | 28 |
| 9 | Reto | 31 |
| 10 | Conclusión | 32 |
| 11 | Componentes en React | 33 |
| 11.1 | Crear un componente | 33 |
| 12 | Props en React. | 35 |
| 13 | Reutilización de los componentes | 37 |
| 13.1 | Reutilizar un componente | 37 |
| 13.2 | PropTypes | 38 |
| 14 | Desestructuración de props | 40 |
| 15 | CSS en React | 42 |
| 15.1 | Crear un componente Card | 42 |
| 16 | Reto | 45 |
| 17 | Hooks en React: useState y useEffect | 48 |
| 17.1 | useState | 48 |
| 17.2 | useEffect | 52 |
| 17.3 | Reto | 55 |
| 17.4 | Conclusiones | 56 |
| 17.5 | Resumen | 56 |
| II | Unidad 2: Material UI | 57 |
| 18 | Material UI en React | 58 |
| 18.1 | Google Material Design | 58 |
| 18.2 | Instalación de Material UI | 59 |
| 18.3 | Uso de Material UI | 59 |

1 Bienvenido

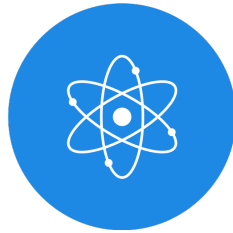


Figure 1.1: React

¡Bienvenido al Curso de React!

En este curso, exploraremos todo, desde los fundamentos hasta las aplicaciones prácticas.

1.1 ¿De qué trata este curso?

Este curso es una introducción a React, una biblioteca de JavaScript para construir interfaces de usuario interactivas y reutilizables.

React es una de las bibliotecas de JavaScript más populares y ampliamente utilizadas en la actualidad. Fue desarrollado por Facebook y lanzado en 2013. Desde entonces, ha ganado una gran popularidad y se ha convertido en una de las herramientas más utilizadas por los desarrolladores web para crear aplicaciones modernas y escalables.

En este curso, aprenderá los conceptos básicos de React, incluidos los componentes, el estado, las propiedades, el ciclo de vida y el enrutamiento. También aprenderá a crear aplicaciones web modernas y escalables utilizando React y otras tecnologías relacionadas, como Redux, React Router y Axios.

1.2 ¿Para quién es este curso?

Este curso es para cualquier persona interesada en aprender React, incluidos:

- Estudiantes que deseen aprender un nuevo framework de JavaScript.
- Desarrolladores web que deseen mejorar sus habilidades y conocimientos en React.
- Profesionales de TI que deseen aprender una nueva tecnología para mejorar su carrera.
- Cualquier persona interesada en aprender a crear aplicaciones web modernas y escalables.

1.3 ¿Cómo contribuir?

Valoramos su contribución a este curso. Si encuentra algún error, desea sugerir mejoras o agregar contenido adicional, me encantaría saber de usted.

Puede contribuir a través del repositorio en línea, donde puede compartir sus comentarios y sugerencias.

Juntos, podemos mejorar continuamente este recurso educativo para beneficiar a la comunidad de estudiantes y entusiastas de la programación.

Este ebook ha sido creado con el objetivo de proporcionar acceso gratuito y universal al conocimiento.

Estará disponible en línea para cualquier persona, sin importar su ubicación o circunstancias, para acceder y aprender a su propio ritmo.

Puede descargarlo en formato PDF, Epub o verlo en línea en cualquier momento y lugar.

¡Gracias por su interés en este curso y espero que disfrute aprendiendo React!

Part I

Unidad 1: Introducción a React

2 Introducción a React.

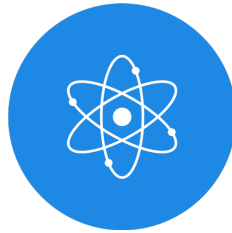


Figure 2.1: React

React es una librería de JavaScript para construir interfaces de usuario. Fue desarrollada por Facebook y es utilizada por muchas empresas, como Instagram, Airbnb, Netflix, WhatsApp, entre otras.

React es una librería declarativa, lo que significa que se enfoca en describir cómo debería ser la interfaz de usuario en lugar de cómo debería funcionar. Esto permite que el código sea más predecible y fácil de depurar.

React utiliza un lenguaje de marcado llamado **JSX**, que es una extensión de JavaScript. JSX permite escribir **HTML** dentro de **JavaScript**, lo que facilita la creación de componentes reutilizables.

En este curso aprenderás los conceptos básicos de React, como **componentes**, **props**, **state**, **eventos** y **ciclos de vida**. También aprenderás a crear aplicaciones utilizando React.

2.1 Instalación de React

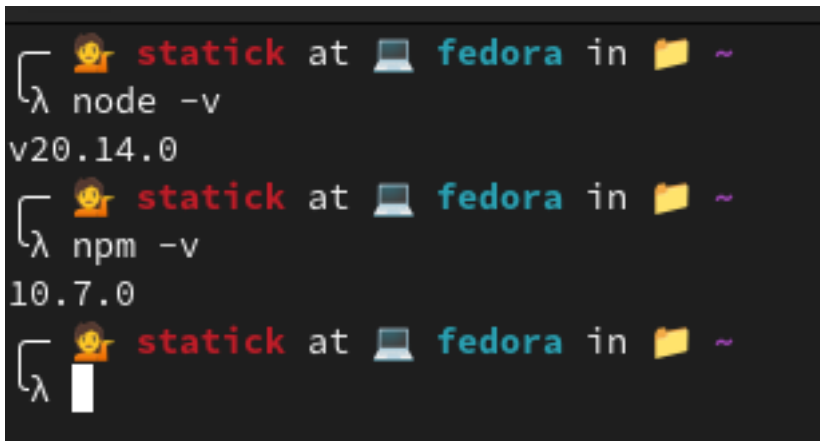
Para comenzar a trabajar con React, necesitas instalar Node.js y npm (Node Package Manager). Puedes descargar Node.js desde su sitio web oficial: <https://nodejs.org/>

Existen gestores de versiones que permiten tener más de una versión de Node.js instalada en tu computadora, como nvm (Node Version Manager) para Linux y macOS, y nvm-windows para Windows. Puedes encontrar más información en la documentación oficial de Node.js: <https://nodejs.org/en/download/package-manager/>

Se recomienda utilizar fnm (Fast Node Manager) que es un gestor de versiones de Node.js más rápido que nvm.

Una vez que hayas instalado Node.js, puedes verificar si npm está instalado ejecutando el siguiente comando en la terminal:

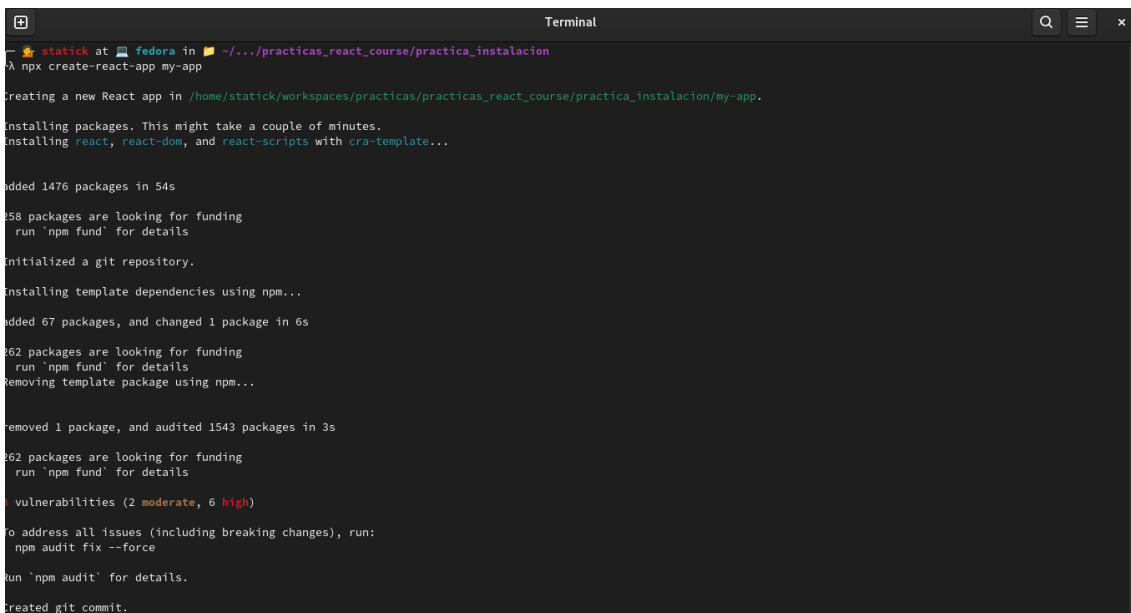
```
npm -v
```



```
statick at fedora in ~
λ node -v
v20.14.0
statick at fedora in ~
λ npm -v
10.7.0
statick at fedora in ~
```

Una vez que tengas npm instalado, puedes instalar React utilizando el siguiente comando:

```
npx create-react-app my-app
```



```
Terminal
statick at fedora in ~/.../practicas_react_course/practica_instalacion
λ npx create-react-app my-app
creating a new React app in /home/statick/workspaces/practicas/practicas_react_course/practica_instalacion/my-app.
Installing packages. This might take a couple of minutes.
Installing react, react-dom, and react-scripts with cra-template...

added 1476 packages in 54s
258 packages are looking for funding
  run `npm fund` for details

initialized a git repository.

Installing template dependencies using npm...
added 67 packages, and changed 1 package in 6s
262 packages are looking for funding
  run `npm fund` for details
Removing template package using npm...

removed 1 package, and audited 1543 packages in 3s
262 packages are looking for funding
  run `npm fund` for details

vulnerabilities (2 moderate, 6 high)

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

run `npm audit` for details.

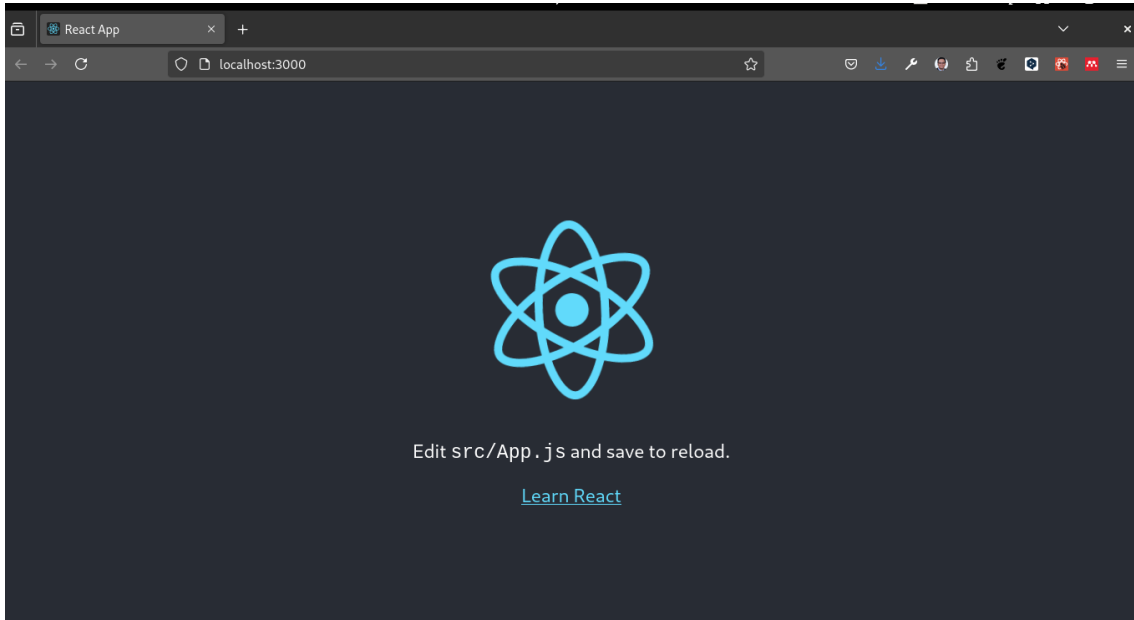
Created git commit.
```

Una vez instalado React, puedes utilizar los siguientes comandos:

- **npm start:** Inicia la aplicación en modo de desarrollo.
- **npm run build:** Compila la aplicación para producción.
- **npm test:** Ejecuta las pruebas de la aplicación.
- **npm run eject:** Expone las configuraciones de la aplicación.

Si todo salio bien puedes ejecutar el servidor de desarrollo y probar tu aplicación mediante el navegador con el siguiente comando:


```
cd my-app  
npm start
```



💡 Tip

La forma actual que se recomienda para utilizar react es a través de Vite, que es un bundler mucho más rápido que create-react-app. Para instalar Vite, ejecuta el siguiente comando:

```
npm create vite@latest
```

Ya sea que crees tu aplicación con **create-react-app** o con **Vite**, se recomienda agregar un `.` al final del comando para que se cree la aplicación en la carpeta actual. Por ejemplo:

```
npx create-react-app .
```

o

```
npm create vite@latest .
```

De esta forma, se creará la aplicación en la carpeta actual y no en una carpeta con el nombre de la aplicación.

Al instalar la aplicación con Vite tendrás unas preguntas para elegir con que lenguaje de programación quieres trabajar, elige TypeScript.

Dentro de las opciones disponibles tendras:

- **Vanilla:** JavaScript puro.
- **Vue:** la librería de JavaScript Vue.js.

- **React:** la librería de JavaScript React.js.
- **Preact:** Preact.js es una versión más ligera de React.js.
- **Lit:** Lit es una librería de JavaScript para crear aplicaciones web.
- **Svelte:** Svelte es un framework de JavaScript para construir aplicaciones web.
- **Quick:** Quick es un framework de JavaScript para construir aplicaciones web.
- **Others:** Otros lenguajes de programación.

En esta curso utilizaremos la opción de React.

```

statick at fedora in ~/.../practica_instalacion/instalacion_vite
λ npm create vite@latest .
  npx
  create-vite .

Select a framework: > - Use arrow-keys. Return to submit.
  Vanilla
  Vue
  React
  Preact
  Lit
  Svelte
  Solid
  Qwik
  Others

```

Una vez seleccionada la opción de react tenemos algunas opciones a elegir:

- **TypeScript:** TypeScript es un lenguaje de programación de código abierto desarrollado por Microsoft.
- **TypeScript + swc:** swc es un compilador de JavaScript y TypeScript escrito en Rust.
- **JavaScript:** JavaScript es un lenguaje de programación de código abierto.
- **JavaScript + swc:** swc es un compilador de JavaScript y TypeScript escrito en Rust.
- **Remix:** Remix es un framework de JavaScript para construir aplicaciones web.

En este curso para iniciar se recomienda utilizar la opción de JavaScript, sin embargo posteriormente se trabajará con TypeScript.

```

statick at fedora in ~/.../practica_instalacion/instalacion_vite
λ npm create vite@latest .
  npx
  create-vite .

Select a framework: > React
Select a variant: > JavaScript

scaffolding project in /home/statick/workspaces/practicas/practicas_react_course/practica_instalacion/instalacion_vite...

done. Now run:

  npm install
  npm run dev

statick at fedora in ~/.../practica_instalacion/instalacion_vite

```

Para probar la aplicación es necesario realizar la instalacion de las dependencias de la aplicación, para ello ejecuta el siguiente comando:

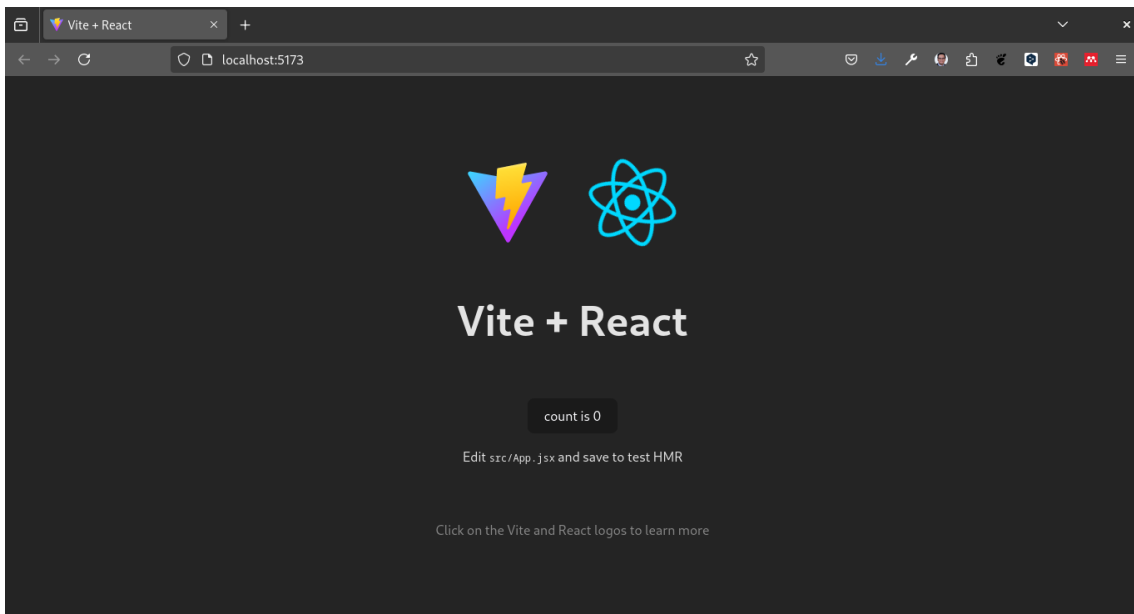
```
npm install
```

 Tip

Una opción de abreviar el comando anterior es utilizando el comando **npm i** que es equivalente a **npm install**.

Con ello es posible correr el servidor de desarrollo y probar la aplicación en el navegador con el siguiente comando:

```
npm run dev
```



De esta forma se podrá visualizar la aplicación en el navegador.

3.2 Crear un componente

Un componente en React es una pieza de código que se encarga de renderizar una parte de la interfaz de usuario, para crear un componente en React, vamos a crear un archivo llamado **HelloWorld.jsx** en la carpeta **src/components** y vamos a agregar el siguiente código:

```
const HelloWorld = () => {
  return (
    <h1>Hola Mundo</h1>
  );
};

export default HelloWorld;
```

Tip

Los componentes de react se pueden exportar de 2 formas diferentes, la primera es la que acabamos de ver, donde se exporta el componente de forma individual y la segunda es exportar el componente de forma anónima, para ello se puede hacer de la siguiente forma:

```
export default function HelloWorld() {
  return <h1>Hello World!</h1>;
}
```

Ambas formas son válidas y se pueden utilizar indistintamente.

Ahora que ya tenemos nuestro componente, vamos a importarlo en el archivo **App.jsx** que se encuentra en la carpeta **src** y vamos a agregar el siguiente código:

```
import HelloWorld from './components/HelloWorld';

function App() {
  return (
    <div>
      <HelloWorld />
    </div>
  );
}

export default App;
```

Tip

Recuerda que el componente que acabamos de importar será renderizado en el archivo **main.js** que se encuentra en la carpeta **src**, podrias modificarlo un poco antes de

correr el servidor y probar si todo funciona correctamente.

```
import React from 'react'  
import ReactDOM from 'react-dom/client'  
import App from './App.jsx'  
  
ReactDOM.createRoot(document.getElementById('root')).render(  
  <React.StrictMode>  
    <App />  
  </React.StrictMode>,  
)
```

Con esto tenemos todo listo, para probar nuestra aplicación vamos a ejecutar el siguiente comando:

```
npm run dev
```

Este comando nos va a iniciar un servidor de desarrollo y nos va a abrir una ventana en el navegador con nuestra aplicación.



Hello World!

Con esto ya tenemos nuestra aplicación funcionando correctamente, en la siguiente sección vamos a entender como está organizada la estructura de un proyecto en React.

3.3 Retos

1. Modifica el componente **HelloWorld.jsx** para que muestre un mensaje diferente como tu nombre.

Ver respuesta

```
const HelloWorld = () => {  
  return (  
    <h1>Hola, mi nombre es Diego Saavedra</h1>  
  );  
};  
  
export default HelloWorld;
```



Hola, mi nombre es Diego Saavedra

2. Crea un nuevo componente llamado **HelloWorld2.jsx** y modifica el componente **App.jsx** para que renderice el nuevo componente.

Ver respuesta

```
const HelloWorld2 = () => {  
  return (  
    <h1>Hola, mi nombre es Diego Saavedra</h1>  
  );  
};  
  
export default HelloWorld2;
```

```
import './App.css'  
  
function App() {  
  
}  
  
return (  

```

```
<div>
  <MostrarNombre />
  <MostrarNombreJs />
</div>
);

const MostrarNombre = () => {
  return <h1>Diego Saavedra</h1>
}

const MostrarNombreJs = () => {
  return 'Diego Saavedra'
}

export default MostrarNombre;
```



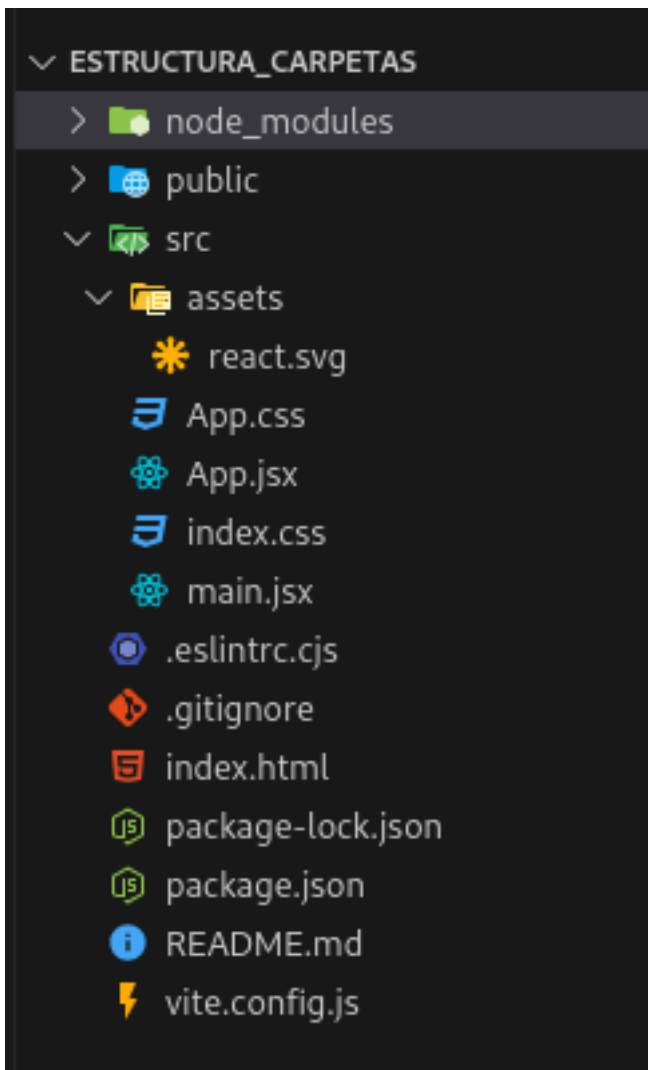
Hola, mi nombre es Diego Saavedra

Hola, mi nombre es Diego Saavedra

3.4 Conclusiones

En esta sección aprendimos a crear un proyecto Vite con React y a crear nuestro primer componente en React, en la siguiente sección vamos a entender como está organizada la estructura de un proyecto en React.

4 Estructura de carpetas en React



En esta sección vamos a analizar la estructura de carpetas que se recomienda para un proyecto de React.

4.1 Estructura de carpetas

La estructura de carpetas que se recomienda para un proyecto de React es la siguiente:

```
node_modules
public/
  vite.svg
src/
  assets/
    react.svg
  App.css
  App.jsx
  index.css
  main.jsx
.eslintrc.cjs
.gitignore
index.html
package-lock.json
package.json
README.md
vite.config.js
.vscode/
  settings.json
```

Tip

En una instalación nueva de Vite, no se encuentra el directorio **src/components** pero se sugiere crearlo manualmente para organizar los componentes de la aplicación.

Analizaremos cada una de las carpetas y archivos que se encuentran en la estructura de carpetas.

4.1.1 **node_modules**

La carpeta **node_modules** es la carpeta donde se instalan todas las dependencias del proyecto. No se debe modificar esta carpeta.

4.1.2 **public**

La carpeta **public** es la carpeta donde se encuentran los archivos estáticos del proyecto. En esta carpeta se encuentra el archivo **index.html** que es el archivo principal de la aplicación.

4.1.3 **src**

La carpeta **src** es la carpeta donde se encuentran los archivos de código fuente del proyecto. En esta carpeta se encuentran los archivos **App.jsx** y **main.jsx** que son los archivos principales de la aplicación.

4.1.4 **assets**

La carpeta **assets** es la carpeta donde se encuentran los archivos de imágenes, fuentes, estilos y otros archivos estáticos del proyecto.

4.1.5 **App.css**

El archivo **App.css** es el archivo de estilos de la aplicación. Este archivo se utiliza para definir los estilos CSS específicos del componente principal de la aplicación en React.

El archivo **App.css** es el archivo de estilos de la aplicación.

4.1.6 **App.jsx**

El archivo **App.jsx** es el archivo principal de la aplicación. En este archivo se define el componente principal de la aplicación.

4.1.7 **index.css**

El archivo **index.css** es el archivo de estilos de la página principal de la aplicación.

4.1.8 **main.jsx**

El archivo **main.jsx** es el archivo principal de la aplicación. En este archivo se importa el componente principal de la aplicación y se renderiza en el archivo **index.html**.

4.1.9 **.eslintrc.cjs**

El archivo **.eslintrc.cjs** es el archivo de configuración de ESLint. En este archivo se definen las reglas de linting para el proyecto.

4.1.10 **.gitignore**

El archivo **.gitignore** es el archivo de configuración de Git. En este archivo se definen los archivos y carpetas que se deben ignorar en el control de versiones.

4.1.11 **index.html**

El archivo **index.html** es el archivo principal de la aplicación. En este archivo se define la estructura HTML de la página principal de la aplicación.

4.1.12 **package-lock.json**

El archivo **package-lock.json** es el archivo de bloqueo de dependencias de Node.js. En este archivo se guardan las versiones exactas de las dependencias instaladas en el proyecto.

4.1.13 **package.json**

El archivo **package.json** es el archivo de configuración de Node.js. En este archivo se definen las dependencias, scripts y metadatos del proyecto.

4.1.14 **README.md**

El archivo **README.md** es el archivo de documentación del proyecto. En este archivo se describe el proyecto, las dependencias, los scripts y otros detalles del proyecto.

4.1.15 **vite.config.js**

El archivo **vite.config.js** es el archivo de configuración de Vite. En este archivo se definen las configuraciones de Vite para el proyecto.

Con este análisis de la estructura de carpetas de un proyecto de React, ya tienes una idea de cómo organizar tus archivos y carpetas en tu proyecto de React.

4.2 **Conclusión**

En esta sección hemos analizado la estructura de carpetas que se recomienda para un proyecto de React. Hemos analizado cada una de las carpetas y archivos que se encuentran en la estructura de carpetas y hemos explicado su propósito.

5 ¿Qué es JSX?

JSX es una extensión de la sintaxis de JavaScript. Es una forma de escribir HTML en JavaScript. JSX puede recordarte a un lenguaje de plantillas, pero viene con todo el poder de JavaScript.

JSX produce elementos de React. Puedes guardar elementos de React en variables, devolverlos en funciones y pasarlos como argumentos.

JSX es una extensión de JavaScript creada por Facebook para el uso con la biblioteca React. No es necesario usar JSX con React, pero es muy recomendable.

JSX hace que tu código sea más legible y más fácil de escribir.

6 ¿Por qué JSX?

Sin JSX, escribir React sería extremadamente tedioso. Por ejemplo, aquí hay un código que crea un elemento sin JSX:

```
const element = React.createElement(  
  'h1',  
  {className: 'greeting'},  
  'Hello, world!'  
);
```

Este código es difícil de leer, escribir y mantener. Con JSX, puedes escribir el mismo código de esta manera:

```
const element = <h1 className="greeting">Hello, world!</h1>;
```

7 Ejemplos:

Estos ejemplos los vamos a trabajar en el archivo **App.jsx** de un proyecto en React.

```
import './App.css'

export default function App() {

  return (
    <>
      <h1>Diego Saavedra</h1>
    </>
  )
}
```

En el ejemplo anterior estamos creando un **componente funcional** que retorna un **h1** con el nombre de **Diego Saavedra**.



Diego Saavedra

Ahora con JSX podemos hacer uso de código javascript dentro de nuestro componente.

```
import './App.css'

export default function App() {

  const nombre = 'Diego Saavedra'
```

```
return (  
  <>  
    <h1>{nombre}</h1>  
  </>  
)  
}
```

En el ejemplo anterior estamos creando una constante **nombre** que contiene el valor de **Diego Saavedra** y lo estamos mostrando en el **h1**. Y el resultado sigue siendo el mismo.



Diego Saavedra

También podemos hacer uso de **JSX** para almacenar código html dentro de una variable.

```
import './App.css'  
  
export default function App() {  
  const nombre = <h1>Diego Saavedra</h1>  
  
  return (  
    <>  
      {nombre}  
    </>  
  )  
}
```

En el ejemplo anterior estamos almacenando el código html en la variable **nombre** y lo estamos mostrando en el componente. Y seguimos obteniendo el mismo resultado.



Diego Saavedra

Con esto podemos comprobar que podemos almacenar partes de código html en variables y mostrarlas en nuestros componentes. Y también podemos hacer uso de código javascript dentro de nuestros componentes.

Ahora compliquemos un poco el código anterior agregando más información.

```
import './App.css'

export default function App() {

  const nombre = 'Diego Saavedra'
  const edad = 36
  const ciudad = 'Quito'

  return (
    <>
    <h1>{nombre}</h1>
    <p>Edad: {edad}</p>
    <p>Ciudad: {ciudad}</p>
    </>
  )
}
```

En el ejemplo anterior estamos creando tres constantes **nombre**, **edad** y **ciudad** y las estamos mostrando en el componente. Y el resultado sigue siendo el mismo.



Diego Saavedra

Edad: 36

Ciudad: Quito

Podemos almacenar toda la información dentro de otra variable que almacene todo el código html.

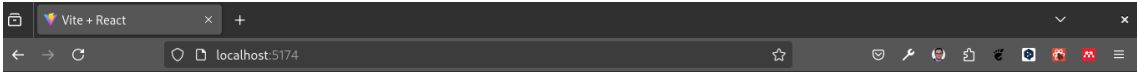
```
import './App.css'

export default function App() {

  const info = (
    <>
    <h1>Diego Saavedra</h1>
    <p>Edad: 36</p>
    <p>Ciudad: Quito</p>
    </>
  )

  return (
    <>
    {info}
    </>
  )
}
```

En el ejemplo anterior estamos almacenando todo el código html en la variable **info** y lo estamos mostrando en el componente. Y el resultado sigue siendo el mismo.



Diego Saavedra

Edad: 36

Ciudad: Quito

8 ¿Cuál es la diferencia con retornar algo en JSX que en JS?

Para ello vamos a utilizar un ejemplo.

```
export default function App() {  
  
  const MostrarNombre = () => {  
    return <h1>Diego Saavedra</h1>  
  }  
  
  const MostrarNombreJS = () => {  
    return 'Diego Saavedra'  
  }  
}
```

En el código anterior estamos creando dos funciones **MostrarNombre** y **MostrarNombreJS**. La primera retorna un **h1** con el nombre de **Diego Saavedra** y la segunda retorna el nombre de **Diego Saavedra**.

Ahora vamos a mostrar el resultado de ambas funciones en el componente.

```
import './App.css'  
  
export default function App() {  
  
  const MostrarNombre = () => {  
    return <h1>Diego Saavedra</h1>  
  }  
  
  const MostrarNombreJS = () => {  
    return 'Diego Saavedra'  
  }  
  
  return (  
    <>  
    <MostrarNombre />  
    <MostrarNombreJS />  
    </>  
  )  
}
```

En el ejemplo anterior estamos mostrando el resultado de ambas funciones en el componente. Y el resultado es el siguiente.



¿Qué paso?

Aparentemente no se ve nada, analicemos que está pasando.

Los componentes en React deben comenzar con una letra mayúscula, si no lo hacemos React no lo va a reconocer como un componente y no lo va a mostrar. Pero este no es el caso, ya que estamos retornando un **h1** en la función **MostrarNombre**. Entonces, ¿Qué está pasando?

La respuesta es que la función **MostrarNombre** está retornando un **h1** pero no lo está mostrando en el componente. Para que se muestre el **h1** debemos hacer uso de **JSX**.

```
import './App.css'

export default function App() {

  const MostrarNombre = () => {
    return <h1>Diego Saavedra</h1>
  }

  const MostrarNombreJS = () => {
    return 'Diego Saavedra'
  }

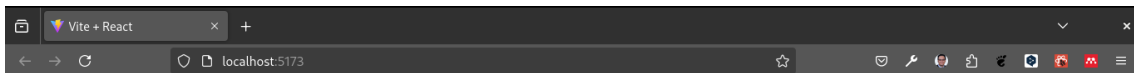
  return (
    <>
      {MostrarNombre}
      {MostrarNombreJS}
    </>
  )
}
```

```
)  
}
```

Sin embargo tampoco vamos a ver nada, esto sucede porque la función **MostrarNombreJS** está retornando un string y no un elemento de React. Para que se muestre el string debemos hacer uso de **JSX**. Olvidemonos de la función **MostrarNombreJS** y hagamos uso de **JSX** en la función **MostrarNombre**.

```
import './App.css'  
  
export default function App() {  
  
  const MostrarNombre = () => {  
    return <h1>Diego Saavedra</h1>  
  }  
  
  return (  
    <>  
      <div className='App'>  
        <MostrarNombre />  
      </div>  
    </>  
  )  
}
```

Como podemos observar en el ejemplo anterior estamos haciendo uso de **JSX** en la función **MostrarNombre** y estamos mostrando el resultado en el componente. Y el resultado es el siguiente.



Diego Saavedra

9 Reto

En el archivo **App.jsx** vamos a crear un componente que muestre la siguiente información:

- Nombre: Su nombre
- Edad: Su edad
- Ciudad: Su ciudad

Ver código

```
import './App.css'

export default function App() {

  const nombre = 'Diego Saavedra'
  const edad = 36
  const ciudad = 'Quito'

  return (
    <>
      <h1>Nombre: {nombre}</h1>
      <p>Edad: {edad}</p>
      <p>Ciudad: {ciudad}</p>
    </>
  )
}
```

10 Conclusión

JSX es una extensión de la sintaxis de JavaScript. Es una forma de escribir HTML en JavaScript. JSX puede recordarte a un lenguaje de plantillas, pero viene con todo el poder de JavaScript.

Los componentes se escriben con mayúsculas para que React los reconozca como componentes. Y para mostrar el resultado de una función en un componente debemos hacer uso de **JSX**.

En el siguiente capítulo vamos a analizar a fondo los Componentes en React y el uso de **Props**.

11 Componentes en React

En este capítulo vamos a ver cómo se pueden crear componentes en React. Los componentes son la base de React, y son la forma en la que se estructura una aplicación en React. Un componente es una pieza de código que se puede reutilizar en diferentes partes de la aplicación, y que se puede componer con otros componentes para crear interfaces de usuario más complejas.

Tip

Antes de iniciar es necesario llevar buenas prácticas de programación, sobretodo un orden en el código que vamos desarrollando, quizá al inicio sea útil solo crear nuestros componentes en el archivo `src/App.jsx` pero a medida que la aplicación crece es necesario separar los componentes en archivos separados. Es por ello que se hace indispensable la creación manual del directorio `src/components` y dentro de este directorio crear los archivos de los componentes que vamos a desarrollar.

11.1 Crear un componente

Para esta unidad vamos a crear un componente llamado **Usuario** que simplemente va a mostrar un mensaje en la pantalla. Para ello vamos a crear un archivo llamado `Usuario.jsx` dentro del directorio `src/components`.

```
export default function Usuario() {
  return <div>
    <h1>Nombre: Diego</h1>
    <p>Edad: 25</p>
    <p>Nacionalidad: Ecuatoriana</p>
  </div>
}
```

En este código estamos creando un componente llamado **Usuario** que simplemente muestra un mensaje en la pantalla. Para poder utilizar este componente en nuestra aplicación, necesitamos importarlo en el archivo `src/App.jsx`.

```
import './App.css'
import { MiComponente } from './components/MiComponente'
function App() {

  return (
```

```
<>
  <div>
    <MiComponente />
  </div>
</>
)
}

export default App
```

En este código estamos importando el componente **Usuario** en el archivo **src/App.jsx** y luego lo estamos utilizando dentro del componente **App**. Si ahora ejecutamos la aplicación, deberíamos ver el mensaje que muestra el componente **Usuario** en la pantalla.



Nombre: Diego

Edad: 25

Nacionalidad: Ecuatoriana

Los componentes son piezas de código que se pueden reutilizar en diferentes partes de la aplicación, y que se pueden componer con otros componentes para crear interfaces de usuario más complejas. En React, los componentes se pueden crear de dos maneras: como funciones o como clases. En este capítulo vamos a ver cómo se pueden crear componentes como funciones.

12 Props en React.

Ahora vamos a ver cómo se pueden pasar propiedades a los componentes en React. Las propiedades son una forma de pasar información de un componente a otro, y se utilizan para personalizar el comportamiento y la apariencia de los componentes.

💡 Tip

Se pueden pasar props a los componentes desde la aplicación principal, sin embargo los componentes no pueden pasar props desde los componentes hijos a los padres. Es decir no podemos pasar props desde los componentes a la aplicación principal.

Para pasar props a los componentes utilizamos una sintaxis similar a la de los atributos de HTML. Por ejemplo, si queremos pasar una prop llamada **nombre** al componente **Usuario**, podemos hacerlo de la siguiente manera:

```
export const MiComponente = (props) => {
  return <div>
    <h1>Nombre: {props.nombre}</h1>
    <p>Edad: {props.edad}</p>
    <p>Nacionalidad: {props.nacionalidad}</p>
  </div>
}
```

En este código estamos pasando una prop llamada **nombre** al componente **Usuario**. Para poder utilizar esta prop en el componente, la recibimos como argumento en la función del componente y la utilizamos dentro del componente.

```
import './App.css'
import { MiComponente } from './components/MiComponente'
function App() {

  return (
    <>
      <div>
        <MiComponente nombre="Diego" edad={36} nacionalidad="Ecuatoriana" />
      </div>
    </>
  )
}

export default App
```

En este código estamos pasando la prop **nombre**, **edad** y **nacionalidad** al componente **MiComponente** con el valor **Diego**. Si ahora ejecutamos la aplicación, deberíamos ver el mensaje que muestra el componente **MiComponente** en la pantalla.



Nombre: Diego

Edad: 36

Nacionalidad: Ecuatoriana

Si quiero pasar un valor entero a un componente, debo hacerlo de la siguiente manera:

```
<MiComponente edad={36}/>
```

Si quiero pasarle un valor booleano a un componente, debo hacerlo de la siguiente manera:

```
<MiComponente esMayorDeEdad={true}/>
```

Si quiero pasar un array a un componente, debo hacerlo de la siguiente manera:

```
<MiComponente colores={['rojo', 'verde', 'azul']}/>
```

Si quiero pasar un objeto a un componente, debo hacerlo de la siguiente manera:

```
<MiComponente persona={{nombre: 'Diego', edad: 36}}/>
```

13 Reutilización de los componentes

En React, los componentes se pueden reutilizar en diferentes partes de la aplicación, y se pueden componer con otros componentes para crear interfaces de usuario más complejas. En este capítulo vamos a ver cómo se pueden reutilizar los componentes en React.

13.1 Reutilizar un componente

Para reutilizar un componente en React, simplemente tenemos que importarlo en el archivo donde queremos utilizarlo y luego utilizarlo en el archivo. Por ejemplo, si queremos reutilizar el componente **Usuario** en el archivo `src/App.jsx`, simplemente tenemos que importarlo y luego utilizarlo en el archivo.

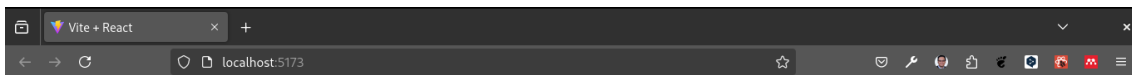
```
import './App.css'
import { MiComponente } from './components/MiComponente'

function App() {

  return (
    <>
      <div>
        <MiComponente nombre="Diego" edad={36} nacionalidad="Ecuatoriana" />
        <MiComponente nombre="Maria" edad={25} nacionalidad="Colombiana" />
        <MiComponente nombre="Pedro" edad={30} nacionalidad="Mexicana" />
      </div>
    </>
  )
}

export default App
```

En este código estamos reutilizando el componente **Usuario** en el archivo `src/App.jsx` y lo estamos utilizando tres veces con diferentes propiedades. Si ahora ejecutamos la aplicación, deberíamos ver el mensaje que muestra el componente **Usuario** en la pantalla.



Nombre: Diego

Edad: 36

Nacionalidad: Ecuatoriana

Nombre: Maria

Edad: 25

Nacionalidad: Colombiana

Nombre: Pedro

Edad: 30

Nacionalidad: Mexicana

13.2 PropTypes

En React, los PropTypes son una forma de validar las propiedades que se pasan a los componentes. Los PropTypes son una forma de documentar los componentes y de asegurarse de que se están pasando las propiedades correctas a los componentes.

Adecuando el componente **MiComponente** con PropTypes, el código quedaría de la siguiente manera:

```
import PropTypes from 'prop-types';

export const MiComponente = (props) => {
  return <div>
    <h1>Nombre: {props.nombre}</h1>
    <p>Edad: {props.edad}</p>
    <p>Nacionalidad: {props.nacionalidad}</p>
  </div>
}

MiComponente.propTypes = {
  nombre: PropTypes.string.isRequired,
  edad: PropTypes.number.isRequired,
  nacionalidad: PropTypes.string.isRequired
};
```

En este código estamos utilizando los PropTypes para validar las propiedades que se pasan al componente **Usuario**. En este caso, estamos validando que las propiedades **nombre** y **nacionalidad** sean de tipo **string** y **edad** sea de tipo **number**, y que todas las propiedades sean requeridas.

Si ahora ejecutamos la aplicación, deberíamos ver un mensaje de error en la consola que nos indica que falta una propiedad requerida.



Nombre: Diego

Edad: 36

Nacionalidad: Ecuatoriana

Nombre: Maria

Edad: 25

Nacionalidad: Colombiana

Nombre: Pedro

Edad: 30

Nacionalidad: Mexicana

De esta forma aseguramos que los componentes se están utilizando de la forma correcta y que se están pasando las propiedades correctas a los componentes. En el caso de que alguna propiedad no sea requerida, simplemente no se le pone el **.isRequired**.

14 Desestructuración de props

En React, la desestructuración de props es una forma de extraer propiedades de un objeto y asignarlas a variables independientes. La desestructuración de props es una forma de simplificar el código y de hacerlo más legible.

Por ejemplo, si queremos extraer las propiedades **nombre**, **edad** y **nacionalidad** del objeto **props** y asignarlas a variables independientes, podemos hacerlo de la siguiente manera:

```
import PropTypes from 'prop-types';

export const MiComponente = (props) => {
  const { nombre, edad, nacionalidad } = props;

  return <div>
    <h1>Nombre: {nombre}</h1>
    <p>Edad: {edad}</p>
    <p>Nacionalidad: {nacionalidad}</p>
  </div>
}

MiComponente.propTypes = {
  nombre: PropTypes.string.isRequired,
  edad: PropTypes.number.isRequired,
  nacionalidad: PropTypes.string.isRequired
};
```

En este código estamos utilizando la desestructuración de props para extraer las propiedades **nombre**, **edad** y **nacionalidad** del objeto **props** y asignarlas a variables independientes. De esta forma, podemos utilizar las variables **nombre**, **edad** y **nacionalidad** en lugar de **props.nombre**, **props.edad** y **props.nacionalidad**.



Nombre: Diego

Edad: 36

Nacionalidad: Ecuatoriana

Nombre: Maria

Edad: 25

Nacionalidad: Colombiana

Nombre: Pedro

Edad: 30

Nacionalidad: Mexicana

15 CSS en React

Ahora vamos a crear otro componente para entender como utilizar css en React, para ello vamos a crear una carpeta para MiComponente en `src/components` y dentro de esta carpeta vamos a mover el archivo `MiComponente.jsx` y vamos a crear otra carpeta llamada `Card` y dentro de esta carpeta vamos a crear un archivo llamado `Card.jsx`.

15.1 Crear un componente Card

Para esta unidad vamos a crear un componente llamado `Card` que simplemente va a mostrar un mensaje en la pantalla. Para ello vamos a crear un archivo llamado `Card.jsx` dentro del directorio `src/components`.

```
export default function Card() {
  return <div className="card">
    <h1>Card Title</h1>
    <p>Card Description</p>
  </div>
}
```

En este código estamos creando un componente llamado `Card` que simplemente muestra un mensaje en la pantalla. Para poder utilizar este componente en nuestra aplicación, necesitamos importarlo en el archivo `src/App.jsx`.

```
import Card from './components/Card/Card'
import { MiComponente } from './components/MiCوپonente/MiComponente'

function App() {

  return (
    <>
      <div>
        <MiComponente nombre="Diego" edad={36} nacionalidad="Ecuatoriana" />
        <MiComponente nombre="Maria" edad={25} nacionalidad="Colombiana" />
        <MiComponente nombre="Pedro" edad={30} nacionalidad="Mexicana" />
        <Card />
      </div>
    </>
  )
}
```

```
export default App
```

En este código estamos importando el componente **Card** en el archivo **src/App.jsx** y luego lo estamos utilizando dentro del componente **App**. Si ahora ejecutamos la aplicación, deberíamos ver el mensaje que muestra el componente **Card** en la pantalla.



Nombre: Diego

Edad: 36

Nacionalidad: Ecuatoriana

Nombre: Maria

Edad: 25

Nacionalidad: Colombiana

Nombre: Pedro

Edad: 30

Nacionalidad: Mexicana

Card Title

Card Description

Ahora vamos a crear un archivo de estilos para el componente **Card**. Para ello vamos a crear un archivo llamado **Card.css** dentro del directorio **src/components/Card** y vamos a agregar los siguientes estilos al archivo.

```
.card {
  background-color: #000000;
  color: #fff;
  border: 1px solid #ddd;
  border-radius: 5px;
  padding: 20px;
  margin: 20px;
}
```

En este código estamos creando un archivo de estilos para el componente **Card** que define los estilos del componente. Para poder utilizar estos estilos en el componente **Card**, necesitamos importar el archivo de estilos en el archivo **Card.jsx**.

```
import "../Card.css";

export default function Card() {
  return <div className="card">
    <h1>Card Title</h1>
    <p>Card Description</p>
  </div>
}
```

```
</div>  
}
```

En este código estamos importando el archivo de estilos en el archivo **Card.jsx** y luego estamos utilizando la clase **card** en el componente **Card** para aplicar los estilos al componente. Si ahora ejecutamos la aplicación, deberíamos ver los estilos aplicados al componente **Card** en la pantalla.



Nombre: Diego

Edad: 36

Nacionalidad: Ecuatoriana

Nombre: Maria

Edad: 25

Nacionalidad: Colombiana

Nombre: Pedro

Edad: 30

Nacionalidad: Mexicana

Card Title

Card Description

16 Reto

1. Crear un componente llamado **Header** que muestre un título y una descripción.
2. Crear un componente llamado **Footer** que muestre un mensaje de derechos de autor.
3. Crear un componente llamado **Button** que muestre un botón con un texto.
4. Crear un componente llamado **Input** que muestre un campo de texto.
5. Crear un componente llamado **Card** que muestre un título y una descripción.

Solución

```
// Header.jsx

export default function Header() {
  return <div>
    <h1>Header</h1>
    <p>Header Description</p>
  </div>
}
```

```
// Footer.jsx

export default function Footer() {
  return <div>
    <p>© 2025 Todos los derechos reservados.</p>
  </div>
}
```

```
// Button.jsx

export default function Button() {
  return <button>Click me</button>
}
```

```
// Input.jsx

export default function Input() {
  return <input type="text" placeholder="Enter your name" />
}
```

```
// Card.jsx

import "./Card.css";
```

```
export default function Card() {
  return <div className="card">
    <h1>Card Title</h1>
    <p>Card Description</p>
  </div>
}
```

```
// App.jsx

import Header from './components/Header/Header'
import Footer from './components/Footer/Footer'
import Button from './components/Button/Button'
import Input from './components/Input/Input'
import Card from './components/Card/Card'

function App() {

  return (
    <>
      <div>
        <Header />
        <Button />
        <Input />
        <Card />
        <Footer />
      </div>
    </>
  )
}

export default App
```



Header

Header Description

[Click me](#)

Card Title

Card Description

© 2025 Todos los derechos reservados.

17 Hooks en React: useState y useEffect

Los hooks en React son funciones que permiten a los componentes funcionales tener estado y efectos secundarios. En este ejercicio vamos a trabajar con dos de los hooks más utilizados: `useState` y `useEffect`.

17.1 useState

El hook `useState` permite a los componentes funcionales tener estado. Para utilizarlo, se debe importar desde la librería de React y llamarlo dentro del componente. `useState` recibe un argumento que es el valor inicial del estado y devuelve un array con dos elementos: el valor actual del estado y una función para actualizarlo.

Para entender esta sección vamos a crear un componente llamado `Counter` que muestre un contador y un botón para incrementar el contador. El componente debe tener un estado inicial de 0 y la función para actualizar el estado debe incrementar el valor actual en 1.

```
import { useState } from "react"

const Counter = () => {

  const [number, setNumber] = useState(0)
  const aumentar = () => {
    setNumber(number + 1)
  }

  return (
    <div>
      <h1>{number}</h1>
      <button onClick={aumentar}>Aumentar</button>
    </div>
  )
}

export default Counter
```

En el código anterior se importa `useState` desde la librería de React y se llama dentro del componente `Counter` con un estado inicial de 0. La función `aumentar` incrementa el valor actual del estado en 1 cada vez que se presiona el botón. Esta es la forma de trabajar con estados en componentes funcionales. Es un poco diferente al uso del DOM en JavaScript puro, pero es muy sencillo y eficiente.

Ahora es necesario importar el componente **Counter** en el componente principal de la aplicación y renderizarlo.

```
import Counter from "../components/Counter";

function App() {

  return (
    <>
      <Counter/>
    </>
  )
}

export default App
```

En el código anterior se importa el componente **Counter** y se renderiza dentro del componente principal de la aplicación. Al ejecutar la aplicación, se debe mostrar un contador con un botón para incrementar el valor.

Ahora vamos a aumentar 2 funcionalidades al componente **Counter**. La primera es un botón para disminuir el contador y la segunda es un botón para resetear el contador. Para esto, se deben crear dos funciones que actualicen el estado de forma adecuada.

```
import { useState } from "react"

const Counter = () => {

  const [number, setNumber] = useState(0)
  const aumentar = () => {
    setNumber(number + 1)
  }

  const disminuir = () => {
    if (number > 0) {
      setNumber(number - 1)
    }
  }

  const reset = () => {
    setNumber(0)
  }

  return (
    <div>
      <h1>{number}</h1>
      <button onClick={aumentar}>Aumentar</button>
      <button onClick={disminuir}>Disminuir</button>
    </div>
  )
}
```

```

        <button onClick={reset}>Reset</button>
      </div>
    )
  }
}

export default Counter

```

En el código anterior se crean dos funciones **disminuir** y **reset** que disminuyen el contador en 1 y resetean el contador a 0 respectivamente. La función **disminuir** solo disminuye el contador si el valor actual es mayor que 0. De esta forma, se evita que el contador sea un número negativo.



Para entender mejor `useState`, vamos a crear un componente llamado **Text** que permita mostrar y ocultar un texto. El componente debe tener un estado inicial de **true** y la función para actualizar el estado debe cambiar el valor actual a **false**.

```

import { useState } from "react";

const Text = () => {

  const [text, setText] = useState(true)

  function handleText() {
    setText(!text)
  }

  return (
    <div>
      <h1>{text ? 'Hola' : 'Adios'}</h1>
      <button onClick={handleText}>Cambiar</button>
    </div>
  )
}

```

```

    </div>
  )
}

export default Text

```

En el código anterior se crea un componente **Text** que muestra un texto y un botón para cambiar el texto. El texto inicial es **Hola** y al presionar el botón, el texto cambia a **Adios**. Esto se logra con el uso de **useState** y una función que cambia el valor actual del estado.

Ahora es necesario importar el componente **Text** en el componente principal de la aplicación y renderizarlo.

```

import Counter from "../components/Counter";
import Text from "../components/Text";

function App() {

  return (
    <>
      <Counter/>
      <Text/>
    </>
  )
}

export default App

```

En el código anterior se importa el componente **Text** y se renderiza dentro del componente principal de la aplicación. Al ejecutar la aplicación, se debe mostrar un texto y un botón para cambiar el texto.



0

Aumentar | Disminuir | Reset

Hola

Cambiar



0

Aumentar Disminuir Reset

Adios

Cambiar

17.2 useEffect

El hook **useEffect** permite a los componentes funcionales tener efectos secundarios. Para utilizarlo, se debe importar desde la librería de React y llamarlo dentro del componente. **useEffect** recibe dos argumentos: una función que realiza el efecto secundario y un array de dependencias. El efecto secundario se ejecuta cada vez que el componente se renderiza y las dependencias cambian.

Para entender esta sección vamos a crear un componente llamado **Fetch** que obtenga datos de una API y los muestre en pantalla. El componente debe tener un estado inicial de un array vacío y la función para actualizar el estado debe obtener los datos de la API.

En este caso vamos a utilizar la api <https://dog.ceo/dog-api/documentation/random>. Esta API nos permite obtener una imagen aleatoria de un perro. Para obtener los datos de la API, se debe utilizar la función **fetch** de JavaScript y la función **json** para convertir los datos en un objeto JSON.

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

const Fetch = () => {

  const [dog, setDog] = useState([])

  useEffect(() => {
    fetch('https://dog.ceo/api/breeds/image/random')
      .then(response => response.json())
      .then(data => setDog(data.message))
  }, [])

  return (
```

```

    <div>
      <img
        style={
          {
            width: '300px',
            height: '300px',
            borderRadius: '10px'
          }
        }
        src={dog}
        alt="dog"/>
    </div>
  )
}

export default Fetch

```

En el código anterior se importa **useState** y **useEffect** desde la librería de React y se llama dentro del componente **Fetch**. **useEffect** se utiliza para obtener los datos de la API y actualizar el estado con los datos obtenidos. La función **fetch** obtiene los datos de la API y la función **json** convierte los datos en un objeto JSON. El estado **dog** se actualiza con la imagen obtenida de la API.

Ahora es necesario importar el componente **Fetch** en el componente principal de la aplicación y renderizarlo.

```

import Fetch from "./components/Fetch";

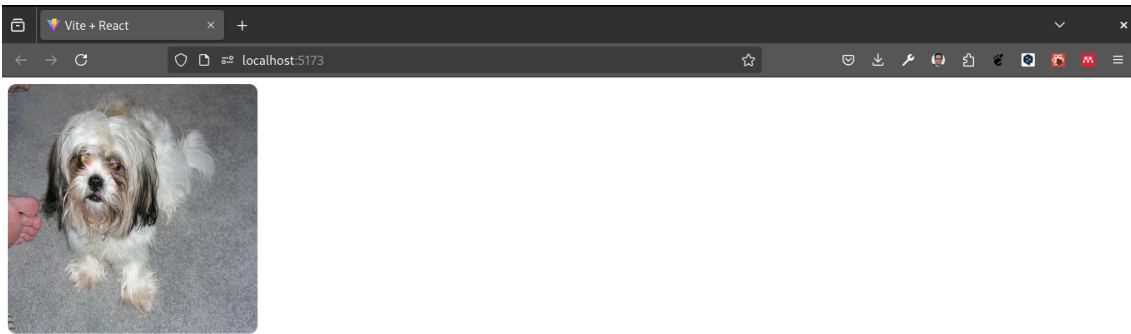
function App() {

  return (
    <>
      <Fetch/>
    </>
  )
}

export default App

```

En el código anterior se importa el componente **Fetch** y se renderiza dentro del componente principal de la aplicación. Al ejecutar la aplicación, se debe mostrar una imagen de un perro obtenida de la API.



Para entender mejor **useEffect**, vamos a crear un componente llamado **Clock** que muestre la hora actual y se actualice cada segundo. El componente debe tener un estado inicial de la hora actual y la función para actualizar el estado debe obtener la hora actual.

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

const Clock = () => {

  const [time, setTime] = useState(new Date().toLocaleTimeString())

  useEffect(() => {
    const interval = setInterval(() => {
      setTime(new Date().toLocaleTimeString())
    }, 1000)

    return () => clearInterval(interval)
  }, [])

  return (
    <div>
      <h1>{time}</h1>
    </div>
  )
}

export default Clock
```

En el código anterior se importa **useState** y **useEffect** desde la librería de React y se llama dentro del componente **Clock**. **useEffect** se utiliza para obtener la hora actual y actualizar el estado con la hora obtenida. La función **setInterval** se utiliza para actualizar

la hora cada segundo y la función **clearInterval** se utiliza para detener el intervalo cuando el componente se desmonta.

Ahora es necesario importar el componente **Clock** en el componente principal de la aplicación y renderizarlo.

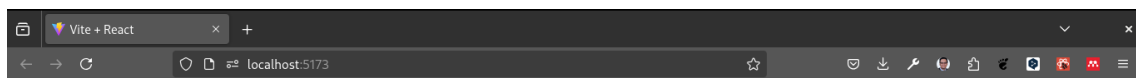
```
import Clock from "../components/Clock";

function App() {

  return (
    <>
      <Clock/>
    </>
  )
}

export default App
```

En el código anterior se importa el componente **Clock** y se renderiza dentro del componente principal de la aplicación. Al ejecutar la aplicación, se debe mostrar la hora actual y actualizarse cada segundo.



03:39:18

17.3 Reto

1. Crear un componente llamado **Form** que tenga un input y un botón. El input debe permitir al usuario ingresar un nombre y el botón debe mostrar un mensaje de bienvenida con el nombre ingresado. El componente debe tener un estado inicial de un string vacío y la función para actualizar el estado debe obtener el valor del input.

2. Crear un componente llamado **List** que muestre una lista de elementos. El componente debe tener un estado inicial de un array vacío y la función para actualizar el estado debe agregar un elemento a la lista. El componente debe tener un input y un botón para agregar elementos a la lista.
3. Crear un componente llamado **Fetch2** que obtenga datos de una API y los muestre en pantalla. El componente debe tener un estado inicial de un objeto vacío y la función para actualizar el estado debe obtener los datos de la API. El componente debe tener un input y un botón para obtener los datos de la API. Puede utilizar esta lista de APIs Disponibles. <https://github.com/public-apis/public-apis?tab=readme-ov-file>
4. Crear un componente llamado **Clock2** que muestre la hora actual y se actualice cada segundo. El componente debe tener un estado inicial de la hora actual y la función para actualizar el estado debe obtener la hora actual. El componente debe tener un botón para detener la actualización de la hora.

17.4 Conclusiones

Los hooks en React son funciones que permiten a los componentes funcionales tener estado y efectos secundarios. **useState** se utiliza para tener estado en componentes funcionales y **useEffect** se utiliza para tener efectos secundarios en componentes funcionales. Estos hooks son muy útiles para trabajar con componentes funcionales en React y permiten tener un código más limpio y eficiente.

En este ejercicio aprendimos a utilizar los hooks **useState** y **useEffect** en React. Creamos varios componentes funcionales que utilizan estos hooks para tener estado y efectos secundarios. También aprendimos a trabajar con APIs y a actualizar la hora actual cada segundo. Los hooks en React son una forma muy eficiente de trabajar con componentes funcionales y permiten tener un código más limpio y fácil de mantener.

17.5 Resumen

- **useState** permite a los componentes funcionales tener estado.
- **useEffect** permite a los componentes funcionales tener efectos secundarios.
- **useState** y **useEffect** son dos de los hooks más utilizados en React.
- Los hooks en React permiten tener un código más limpio y eficiente.

Part II

Unidad 2: Material UI

18 Material UI en React

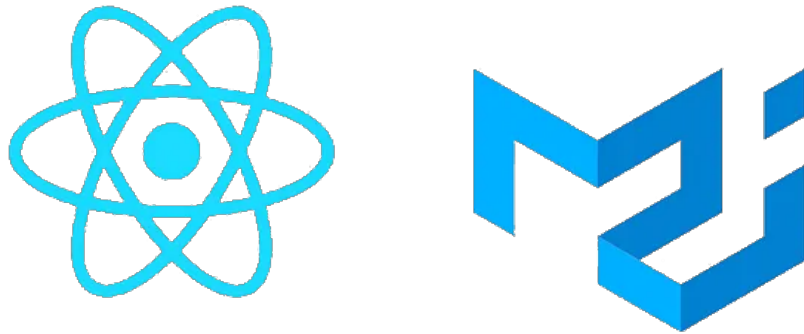


Figure 18.1: Material UI

Material UI es una librería de componentes de React que implementa el diseño de Google Material Design. En este tutorial aprenderemos a instalar Material UI en un proyecto de React y a utilizar algunos de sus componentes.

El objetivo de este tema es:

- Aprender a instalar y utilizar Material UI, una librería de componentes de React que implementa el diseño de Google Material Design, enfocándose en crear interfaces que simulan objetos físicos mediante sombras, bordes redondeados y animaciones, siguiendo los principios fundamentales de Material Design: materialidad y movimiento.

18.1 Google Material Design

Material Design es un sistema de diseño creado por Google que se basa en la idea de que los elementos de la interfaz deben parecerse a objetos físicos del mundo real. Los elementos de Material Design tienen sombras, bordes redondeados y animaciones que los hacen parecer más reales.

Material Design se basa en tres principios fundamentales:

1. **Material:** Los elementos de la interfaz deben parecerse a objetos físicos del mundo real.
2. **Movimiento:** Los elementos de la interfaz deben tener animaciones y transiciones suaves.
3. **Diseño adaptativo:** Los elementos de la interfaz deben adaptarse a diferentes tamaños de pantalla y dispositivos.

Material Design se basa en una paleta de colores, tipografías y elementos de diseño que se pueden utilizar para crear interfaces coherentes y atractivas.

18.2 Instalación de Material UI

Para instalar Material UI en un proyecto de React, primero debemos crear un nuevo proyecto de React utilizando Create React App. Luego, podemos instalar Material UI utilizando npm o yarn.

Para crear un nuevo proyecto de React, ejecutamos el siguiente comando en la terminal:

```
npm create vite@latest .
```

Luego, instalamos Material UI utilizando npm o yarn:

```
npm install @mui/material @emotion/react @emotion/styled
```

Una vez que hemos instalado Material UI, podemos importar los componentes de Material UI en nuestros archivos de React y utilizarlos en nuestra aplicación.

18.3 Uso de Material UI



Para utilizar Material UI en un proyecto de React, primero debemos importar los componentes de Material UI que queremos utilizar en nuestros archivos de React. Por ejemplo, para importar un botón de Material UI, podemos hacer lo siguiente:

```
import Button from '@mui/material/Button';

export default function ButtonUsage() {
  return <Button variant="contained">Hello world</Button>;
}
```

En este ejemplo, importamos el componente **Button** de Material UI y lo utilizamos en nuestra aplicación. Podemos personalizar el botón utilizando las propiedades del componente, como **variant**, **color**, **size**, etc.

Material UI proporciona una amplia variedad de componentes que podemos utilizar en nuestras aplicaciones, como botones, barras de navegación, tarjetas, formularios, etc. Podemos consultar la documentación de Material UI para obtener más información sobre los componentes disponibles y cómo utilizarlos.

Ahora vamos a crear el siguiente proyecto:

1. **Configuración Inicial del Proyecto:** Iniciar un proyecto React con Vite, seguido de la instalación y configuración de Material UI y TailwindCSS para preparar el entorno de diseño.

Para realizar este proyecto, seguiremos los siguientes pasos:

1.1. **Instalación de Material UI y TailwindCSS:** Crear un proyecto React con Vite y configurar Material UI y TailwindCSS para la creación de interfaces atractivas y coherentes.

Ahora vamos a crear un proyecto React con Vite

```
npm create vite@latest .
```

Instalamos Material UI

```
npm install @mui/material @emotion/react @emotion/styled
```

Instalamos TailwindCSS e inicializamos la configuración

```
npm install -D tailwindcss@latest postcss@latest autoprefixer@latest
npx tailwindcss init -p
```

1.2. **Creación de Componentes Básicos:** Implementar componentes básicos como botones y tarjetas utilizando Material UI, y aplicar estilos personalizados con TailwindCSS para mejorar la apariencia de la interfaz.

En el archivo **App.jsx**, podemos crear los siguientes componentes básicos utilizando **Material UI**:

```

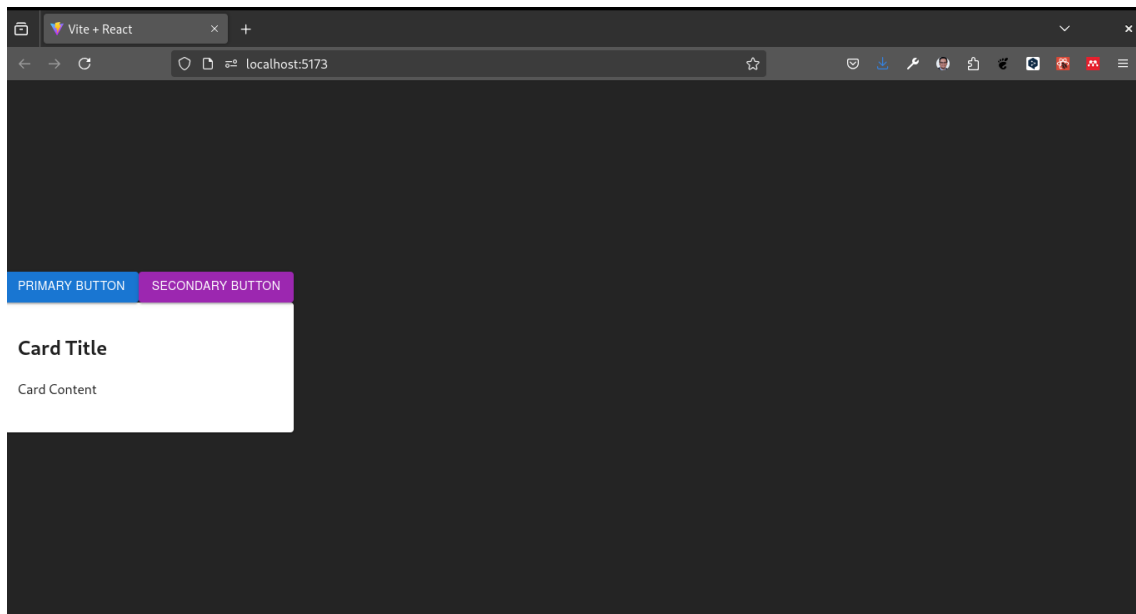
import Button from '@mui/material/Button';
import Card from '@mui/material/Card';
import CardContent from '@mui/material/CardContent';

function App() {
  return (
    <div className="p-4 space-y-4">
      <Button variant="contained" color="primary">
        Primary Button
      </Button>
      <Button variant="contained" color="secondary">
        Secondary Button
      </Button>
      <Card>
        <CardContent>
          <h2 className="text-xl font-bold">Card Title</h2>
          <p className="text-gray-500">Card Content</p>
        </CardContent>
      </Card>
    </div>
  );
}

export default App;

```

En el código anterior creamos un botón primario y un botón secundario utilizando el componente **Button** de Material UI. También creamos una tarjeta con un título y contenido utilizando los componentes **Card** y **CardContent** de Material UI.



1.3. Personalización de Estilos con TailwindCSS: Aplicar estilos personalizados con

TailwindCSS para mejorar la apariencia de los componentes de Material UI y crear una interfaz visualmente atractiva y coherente.

En el archivo **index.css**, podemos personalizar los estilos de los componentes de Material UI utilizando TailwindCSS:

```
@import 'tailwindcss/base';
@import 'tailwindcss/components';
@import 'tailwindcss/utilities';

@layer components {
  .MuiButton-root {
    @apply px-4 py-2 rounded-md shadow-md;
  }
  .MuiButton-containedPrimary {
    @apply bg-blue-500 text-white;
  }
  .MuiButton-containedSecondary {
    @apply bg-red-500 text-white;
  }
  .MuiCard-root {
    @apply shadow-lg rounded-lg;
  }
  .MuiCardContent-root {
    @apply p-4;
  }
  .MuiCardContent-root h2 {
    @apply text-xl font-bold;
  }
  .MuiCardContent-root p {
    @apply text-gray-500;
  }
}
```

En el código anterior aplicamos estilos personalizados a los componentes de Material UI utilizando TailwindCSS. Utilizamos las clases de TailwindCSS para añadir estilos como **padding**, **margins**, **bordes redondeados**, **sombras**, **colores**, etc.

2. **Desarrollo de Componentes Esenciales:** Implementar componentes fundamentales como el Header, Footer, y ProductCard, aprovechando las capacidades de Material UI para la navegación y presentación de productos.

Creamos el archivo Header.jsx para el componente de encabezado, Footer.jsx para el componente de pie de página, y ProductCard.jsx para el componente de tarjeta de producto:

```
import AppBar from '@mui/material/AppBar';

function Header() {
  return (
```

```

    <AppBar position="static">
      <div className="container mx-auto p-4">
        <h1 className="text-2xl font-bold text-white">E-Commerce App</h1>
      </div>
    </AppBar>
  );
}

export default Header;

```

Ahora creamos el archivo Footer.jsx para el componente de pie de página:

```

import AppBar from '@mui/material/AppBar';

function Footer() {
  return (
    <AppBar position="static">
      <div className="container mx-auto p-4">
        <p className="text-sm text-white">© 2022 E-Commerce App. All rights reserved.</p>
      </div>
    </AppBar>
  );
}

export default Footer;

```

Finalmente creamos el archivo ProductCard.jsx para el componente de tarjeta de producto:

```

import Card from '@mui/material/Card';
import CardContent from '@mui/material/CardContent';
import CardMedia from '@mui/material/CardMedia';
import Button from '@mui/material/Button';

function ProductCard({ product }) {
  return (
    <Card>
      <CardMedia
        component="img"
        height="200"
        image={product.image}
        alt={product.title}
      />
      <CardContent>
        <h2 className="text-xl font-bold">{product.title}</h2>
        <p className="text-gray-500">{product.description}</p>
        <Button variant="contained" color="primary">

```

```

        Add to Cart
      </Button>
    </CardContent>
  </Card>
);
}

export default ProductCard;

```

3. Implementación de Navegación y Rutas: Configurar React Router para la navegación entre páginas y rutas de la aplicación, y utilizar los componentes de Material UI para mejorar la experiencia del usuario.

Par ello es necesario instalar React Router:

```
npm install react-router-dom
```

En el archivo App.jsx, podemos configurar React Router para la navegación entre la página principal y la página de detalles del producto:

```

import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';
import Header from './components/Header';
import Footer from './components/Footer';
import ProductCard from './components/ProductCard';

function App() {
  return (
    <Router>
      <Header />
      <div className="container mx-auto p-4">
        <Switch>
          <Route path="/" exact>
            <ProductCard product={{ title: 'Product 1', description: 'Description 1', ima
            <ProductCard product={{ title: 'Product 2', description: 'Description 2', ima
            <ProductCard product={{ title: 'Product 3', description: 'Description 3', ima
          </Route>
          <Route path="/product/:id">
            <ProductCard product={{ title: 'Product 1', description: 'Description 1', ima
          </Route>
        </Switch>
      </div>
      <Footer />
    </Router>
  );
}

export default App;

```


En el código anterior configuramos React Router para mostrar la página principal con una lista de productos y la página de detalles del producto con un producto específico. Utilizamos el componente **ProductCard** para mostrar los productos en ambas páginas.

4. **Optimización de la Interfaz de Usuario:** Mejorar la interfaz de usuario con estilos personalizados y globales, y aplicar las mejores prácticas de diseño para crear una experiencia de usuario atractiva y coherente. En el archivo `index.css`, podemos aplicar estilos personalizados y globales para mejorar la apariencia de la interfaz de usuario:

```
@tailwind base;
@tailwind components;
@tailwind utilities;

@layer components {
  .MuiAppBar-root {
    @apply bg-blue-500;
  }
  .MuiAppBar-root .container {
    @apply flex justify-between items-center;
  }
  .MuiAppBar-root h1 {
    @apply text-white;
  }
  .MuiAppBar-root p {
    @apply text-white;
  }
  .MuiCard-root {
    @apply shadow-lg rounded-lg;
  }
  .MuiCardMedia-root {
    @apply rounded-t-lg;
  }
  .MuiCardContent-root {
    @apply p-4;
  }
  .MuiButton-root {
    @apply px-4 py-2 rounded-md shadow-md;
  }
  .MuiButton-containedPrimary {
    @apply bg-blue-500 text-white;
  }
}
```

En el código anterior aplicamos estilos personalizados y globales para mejorar la apariencia de los componentes de Material UI y crear una interfaz de usuario atractiva y coherente.

5. **Configuración del Entorno de Desarrollo y Control de Versiones:** Establecer un flujo de trabajo eficiente con Git y GitHub para el control de versiones, y optimizar

Visual Studio Code con extensiones relevantes para React, además de configurar TailwindCSS.

```
git init
git add .
git commit -m "Initial commit"
git branch -M main
git remote add origin <repository-url>
git push -u origin main
```

Con los comandos anteriores inicializamos un repositorio Git, añadimos los archivos al área de preparación, realizamos el primer commit, configuramos la rama principal, añadimos el repositorio remoto, y subimos los cambios al repositorio remoto.

4. **Diseño y Navegación:** Crear plantillas para las páginas principales y de detalles de productos, facilitando la navegación con React Router, y aplicar estilos personalizados y globales para cohesión visual y mejora de la interfaz de usuario.

En el archivo Home.jsx, podemos crear una plantilla para la página principal con una lista de productos:

```
import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';
import Header from './components/Header';
import Footer from './components/Footer';
import ProductCard from './components/ProductCard';
import Home from './pages/Home';
import Product from './pages/Product';

function App() {
  return (
    <Router>
      <Header />
      <div className="container mx-auto p-4">
        <Switch>
          <Route path="/" exact component={Home} />
          <Route path="/product/:id" component={Product} />
        </Switch>
      </div>
      <Footer />
    </Router>
  );
}

export default App;
```

En el código anterior utilizamos las plantillas **Home** y **Product** para las páginas principal y de detalles del producto, respectivamente. Utilizamos el componente **ProductCard** para mostrar los productos en la página principal.

En el archivo Home.jsx, podemos crear la plantilla para la página principal con una lista de productos:

```
import ProductCard from '../components/ProductCard';

function Home() {
  return (
    <div className="grid grid-cols-1 md:grid-cols-3 gap-4">
      <ProductCard product={{ title: 'Product 1', description: 'Description 1', image: 'h
      <ProductCard product={{ title: 'Product 2', description: 'Description 2', image: 'h
      <ProductCard product={{ title: 'Product 3', description: 'Description 3', image: 'h
    </div>
  );
}

export default Home;
```

En el código anterior creamos una lista de productos utilizando el componente **ProductCard** en la página principal. Utilizamos la clase **grid** de TailwindCSS para mostrar los productos en una cuadrícula de 3 columnas en pantallas medianas y grandes.

En el archivo Product.jsx, podemos crear la plantilla para la página de detalles del producto con un producto específico:

```
import { useParams } from 'react-router-dom';
import ProductCard from '../components/ProductCard';

function Product() {
  const { id } = useParams();

  return (
    <ProductCard product={{ title: `Product ${id}`, description: `Description ${id}`, ima
  );
}

export default Product;
```

En el código anterior utilizamos el componente **ProductCard** para mostrar un producto específico en la página de detalles del producto. Utilizamos el **useParams** hook de React Router para obtener el ID del producto de la URL.

5. **Implementación de Funcionalidades Avanzadas:** Agregar funcionalidades avanzadas como filtros de productos, carrito de compras, y autenticación de usuarios, utilizando Material UI y React para mejorar la experiencia del usuario.

En el archivo App.jsx, podemos agregar un filtro de productos y un carrito de compras utilizando Material UI y React:

```

import { useState } from 'react';
import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';
import Header from './components/Header';
import Footer from './components/Footer';
import ProductCard from './components/ProductCard';
import Home from './pages/Home';
import Product from './pages/Product';

function App() {
  const [cart, setCart] = useState([]);

  const addToCart = (product) => {
    setCart([...cart, product]);
  };

  return (
    <Router>
      <Header />
      <div className="container mx-auto p-4">
        <Switch>
          <Route path="/" exact component={Home} />
          <Route path="/product/:id" component={Product} />
        </Switch>
      </div>
      <Footer />
    </Router>
  );
}

export default App;

```

En el código anterior utilizamos el estado local **cart** y la función **addToCart** para gestionar el carrito de compras. Utilizamos el componente **ProductCard** para mostrar los productos en la página principal y la página de detalles del producto.

En el archivo ProductCard.jsx, podemos agregar un botón para añadir productos al carrito de compras:

```

import { useState } from 'react';
import Card from '@mui/material/Card';
import CardContent from '@mui/material/CardContent';
import CardMedia from '@mui/material/CardMedia';
import Button from '@mui/material/Button';

function ProductCard({ product, addToCart }) {
  const [isAdded, setIsAdded] = useState(false);

  const handleAddToCart = () => {

```

```

    addToCart(product);
    setIsAdded(true);
  };

  return (
    <Card>
      <CardMedia
        component="img"
        height="200"
        image={product.image}
        alt={product.title}
      />
      <CardContent>
        <h2 className="text-xl font-bold">{product.title}</h2>
        <p className="text-gray-500">{product.description}</p>
        <Button variant="contained" color="primary" onClick={handleAddToCart} disabled={i
          {isAdded ? 'Added to Cart' : 'Add to Cart'}
        </Button>
      </CardContent>
    </Card>
  );
}

export default ProductCard;

```

En el código anterior utilizamos el estado local **isAdded** para controlar si un producto ha sido añadido al carrito de compras. Utilizamos la función **handleAddToCart** para añadir productos al carrito y actualizar el estado de **isAdded**.