React - R0

Lucio Aguilar

E.E.S.T.Nº5 “Amancio Williams”

Proyecto, Diseño e Implementación de Sistemas Computacionales

Martin Estanga

28 de Octubre de 2024

Titulo nivel 1

Actividad:

Deberán tener y cumplir con los siguientes requerimientos para la siguiente clase

Instalar Node.js

Instalar VSCode con extención react VSC "React snippets"

Buscar diferencias entre react-html, react-css, etc, y su estructura

Y armar un listado de buenas normas de programación en react

Respuesta:

Informe sobre Diferencias y Estructura en React

En React, el flujo principal de trabajo está compuesto por componentes, los cuales se dividen generalmente en tres partes principales: **HTML (JSX)**, **CSS** y **JS (JavaScript)**. Veamos cómo se organizan estos elementos y qué prácticas son recomendadas.

1. **React HTML (JSX)**:
   - **Función**: La estructura HTML de los componentes en React se construye utilizando **JSX** (JavaScript XML). JSX permite escribir HTML en JavaScript, manteniendo todo el componente en un solo archivo y permitiendo la inyección dinámica de datos.
   - **Estructura y Uso**:

Figura 1:

Estructura y uso en JS

```javascript
const MiComponente = () => {
  return (
    <div className="mi-clase">
      <h1>Título del Componente</h1>
      <p>Texto dentro del componente.</p>
    </div>
  );
};
```

Nota: JSX no es HTML puro; es una sintaxis que React transpila a JavaScript, permitiendo lógica y condiciones en el marcado. Además, JSX usa className en lugar de class para evitar conflictos con palabras reservadas de JavaScript.

2. **React CSS**:

Función: React permite aplicar estilos CSS de varias formas: **CSS globales**, **CSS Modules**, y **Styled Components** (CSS-in-JS).

Estructura y Uso:

- **CSS Global**: Se pueden importar archivos .css globales directamente al proyecto, aunque puede afectar la modularidad.

Ejemplo: import './styles.css';

- **CSS Modules**: Utilizan nombres únicos para evitar colisiones entre clases y pueden ser importados específicamente para un componente.

Ejemplos:

```
import styles from './MiComponente.module.css';
<div className={styles.miClase}>Contenido</div>
```

- **Styled Components**: Librería que permite escribir estilos dentro del componente utilizando JavaScript, lo que facilita el uso de lógica condicional en los estilos.

Ejemplo:

```
import styled from 'styled-components';
const MiDiv = styled.div`
  background: blue;
  color: white;
`;
```

3. **React JS (Lógica y Comportamiento)**:

- **Función**: Aquí es donde reside la lógica del componente. Esto incluye los hooks, los estados, y las funciones para manejar eventos y realizar cálculos dentro del componente.
- **Estructura y Uso**:
    - Los hooks como useState o useEffect permiten gestionar el estado y efectos secundarios en componentes funcionales.

Figura 2

Ejemplo de React JS

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

const MiComponente = () => {
  const [contador, setContador] = useState(0);

  useEffect(() => {
    console.log("Componente montado");
  }, []);

  return <button onClick={() => setContador(contador + 1)}>{contador}</button>;
};
```

Nota: Los hooks como useState o useEffect permiten gestionar el estado y efectos secundarios en componentes funcionales.

Buenas Prácticas de Programación en React

1. Dividir el Código en Componentes Reutilizables:
    - Cada componente debe cumplir una función específica. Esto mejora la reutilización del código y facilita el mantenimiento.

2. Usar Hooks en Componentes Funcionales:
    - Prioriza el uso de hooks en lugar de clases, ya que permiten una gestión más simple del estado y efectos secundarios.
3. Nombrar Componentes y Archivos de Forma Clara y Consistente:
    - Usa PascalCase para los componentes (MiComponente) y nombres claros para archivos y clases CSS (MiComponente.module.css).
4. Evitar el Estado Global cuando sea Innecesario:
    - Usa el contexto global solo cuando sea imprescindible, para no aumentar la complejidad. Para componentes independientes, usa useState o useReducer en lugar de Context API o Redux.
5. Gestionar los Efectos Secundarios con Cuidado:
    - Utiliza useEffect adecuadamente y especifica dependencias para evitar renders innecesarios y errores.
6. Mantener Separado el Estilo de la Lógica:
    - Centraliza la lógica en un lugar dentro del componente y el estilo en otro, ya sea usando CSS Modules o Styled Components.
7. Documentar y Comentar el Código:
    - Explica las funciones y estados complejos para que otros desarrolladores puedan entender rápidamente el propósito del componente.
8. Optimizar el Rendimiento:
    - Usa React.memo y el hook useCallback para optimizar el rendimiento en componentes que reciben props o funciones intensivas.
9. Utilizar PropTypes o TypeScript:
    - Define el tipo de datos de las props usando PropTypes o TypeScript para asegurar que los componentes reciban los valores esperados.

10. Evitar la Lógica Compleja en el Render:
    - Divide la lógica compleja en funciones o en hooks personalizados en lugar de ponerla directamente en el bloque de retorno JSX.

Conclusión:

Mantener una estructura clara y seguir buenas prácticas en React mejora la eficiencia, organización y escalabilidad de tu aplicación. Cada archivo y componente debería tener una responsabilidad específica, y los estilos y la lógica deben estar bien organizados para facilitar su reutilización y mantenimiento.