«Talento Tech»

Desarrollo Web 3

Clase 02

Buenos Aires Ciudad

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Clase N° 2 | Estilos y Componentes en React

Temario:

- Integración CSS Bootstrap/Bootswatch - React.
- Componentes y Props en React.
- Estado y ciclo de vida de los componentes.

Integración CSS y Bootswatch - React

La integración de estilos CSS en una aplicación **React** es un proceso sencillo. Puedes seguir estos pasos básicos para agregar estilos CSS a tu componente React:

Creación del Archivo CSS

Crea un archivo CSS para tus estilos. Por ejemplo, puedes llamarlo **styles.css.**

Definición de Estilos

En el archivo **styles.css**, define los estilos que deseas aplicar. Por ejemplo:

```css
/* styles.css */
.container {
  max-width: 800px;
  margin: 0 auto;
  padding: 20px;
}

.title {
  font-size: 24px;
  color: #333;
}
```

Importación en el Componente React

En tu componente **React**, importa el archivo CSS que acabas de crear. Puedes hacerlo en la parte superior del archivo del componente **“.jsx”**

```jsx
// TuComponente.jsx
import React from 'react';
import './styles.css'; // Importa tu archivo CSS

const TuComponente = () => {
  return (
    <div className="container">
      <h1 className="title">Mi Aplicación React</h1>
      {/* Resto del contenido */}
    </div>
  );
}

export default TuComponente;
```

Aplicación de Clases CSS

Utiliza las clases CSS que definiste en tu archivo **styles.css** dentro de tu componente **JSX.** Con estos pasos, has integrado estilos CSS en tu componente **React.** Es importante destacar que esta es solo una forma básica de gestionar estilos. En proyectos más grandes, podrías considerar enfoques como **CSS Modules**, **Styled Components**, o incluso el uso de preprocesadores como **Sass** para una organización más avanzada y mantenible de tus estilos.

Bootswatch e instalación

Bootswatch es una colección de temas visuales prediseñados para el framework de diseño web Bootstrap. **Bootstrap** es una herramienta que facilita el desarrollo de sitios web y aplicaciones móviles al proporcionar estilos CSS y componentes predefinidos. **Bootswatch** aprovecha este marco y ofrece una variedad de estilos y apariencias alternativas, permitiendo a los desarrolladores cambiar fácilmente el aspecto de sus proyectos sin tener que escribir estilos personalizados.

Instalación de BootsWatch

Si deseas integrar **Bootswatch** en tu aplicación **React** para estilizarla de manera rápida y atractiva, puedes seguir estos pasos. Bootswatch es una colección de temas para Bootstrap, un marco de diseño de código abierto para el desarrollo web.

1. Instala **Bootstrap** en tu proyecto. Abre la terminal y ejecuta:

```
npm install bootstrap
```

2. Instala **Bootswatch** como una dependencia. Puedes elegir un tema específico según tus preferencias. Por ejemplo, aquí usaremos el tema **"Superhero"**:

```
npm install bootswatch@latest
```

Importación de Estilos en tu Aplicación

En tu componente **React** o en tu archivo de entrada principal (por ejemplo, index.js), importa los estilos de **Bootstrap** y el tema de **Bootswatch** que elegiste:

```
// index.jsx o tu archivo principal
import 'bootstrap/dist/css/bootstrap.min.css';
```

```
import 'bootswatch/dist/superhero/bootstrap.min.css';
```

Uso en Componentes

Ahora puedes usar las clases de **Bootstrap** en tus componentes **React.**
Por ejemplo:

```
// TuComponente.jsx
import React from 'react';

const TuComponente = () => {
  return (
    <div className="container">
      <h1 className="display-4">Mi Aplicación con Bootswatch</h1>
      <p className="lead">Atractiva y fácil de personalizar.</p>
      {/* Resto del contenido */}
    </div>
  );
}

export default TuComponente;
```

Con estos pasos, has integrado **Bootstrap** y **Bootswatch** en tu aplicación **React**.🎊

Props y comunicación entre componentes

En React, los términos **"Props"** (propiedades) y **"Componente"** son fundamentales para entender cómo se estructuran y comunican los elementos en una aplicación. Aquí hay explicaciones breves para ambos conceptos:

Props (Propiedades)

En **React,** el objeto **props** (abreviatura de "propiedades") es un objeto que contiene información que se pasa a un componente. Cuando creas un componente en **React** y lo utilizas en otro lugar de tu aplicación, puedes enviarle datos a través de las **props**.
El objeto **props** es un argumento que se pasa automáticamente a la función del componente y contiene todas las propiedades que se le han asignado al componente al ser utilizado. Estas propiedades pueden ser cualquier tipo de dato, como cadenas de texto, números, booleanos, funciones, objetos, etc.

Características:

- **Inmutables:** Las props son inmutables, lo que significa que no se deben modificar directamente dentro del componente hijo.

- **Datos de Entrada:** Representan datos de entrada para un componente y son configurables desde el componente padre.

Ejemplo:

```
// Componente Padre
import React from 'react';
import HijoComponente from './HijoComponente';

const App = () => {
  const mensaje = "Hola desde el padre";

  return (
    <div>
      <HijoComponente mensajeProp={mensaje} />
    </div>
  );
}

// Componente Hijo
import React from 'react';

const HijoComponente = (props) => {
  return (
    <div>
      <p>{props.mensajeProp}</p>
    </div>
  );
}

export default HijoComponente;
```

En este ejemplo, mensaje es una **prop** que se pasa desde el componente padre (**App**) al componente hijo (**HijoComponente**) y se muestra en el párrafo.

Destructuring en Props:

El **destructuring** es una característica de **JavaScript** que permite extraer valores de objetos o arreglos de una manera más concisa. En el contexto de **React**, puedes aplicar destructuring para extraer valores específicos de las props de manera clara.

Ejemplo con **Destructuring** en **Props**:

```
// Componente Hijo
import React from 'react';

const HijoComponente = ({ mensajeProp }) => {
  return (
    <div>
      <p>{mensajeProp}</p>
    </div>
  );
}

export default HijoComponente;
```

En este ejemplo, en lugar de acceder a **props.mensajeProp**, hemos utilizado destructuring directamente en los argumentos de la función para extraer la propiedad específica **mensajeProp**. Esto hace que el código sea más limpio y fácil de leer.

Destructuring en Componentes Funcionales:

```
// Componente Funcional
import React from 'react';

const MiComponente = ({ titulo, contenido }) => {
  return (
    <div>
      <h1>{titulo}</h1>
      <p>{contenido}</p>
    </div>
  );
}

export default MiComponente;
```

Aquí, el componente funcional **MiComponente** utiliza destructuring en los argumentos para extraer las props título y contenido. Esto simplifica la forma en que accedemos a estas **props** dentro del componente.

Componente

Un **componente** en React es una unidad autónoma y reutilizable de interfaz de usuario. Puede ser una función o una clase que acepta ciertas entradas (**props**) y devuelve elementos **React** que describen cómo debería verse la interfaz de usuario.

Características:

- **Reutilizable:** Los componentes se pueden reutilizar en diferentes partes de la aplicación.
- **Estructura Lógica y de Interfaz:** Pueden tener lógica interna y definir la estructura de la interfaz de usuario.

Ejemplo:

```
// Componente Funcional
import React from 'react';

const MiComponente = (props) => {
  return (
    <div>
      <h1>{props.titulo}</h1>
      <p>{props.contenido}</p>
    </div>
  );
}

// Uso del Componente
import React from 'react';
import MiComponente from './MiComponente';

const App = () => {
  return (
    <div>
      <MiComponente titulo="Bienvenido" contenido="Esto es un ejemplo de componente en React." />
    </div>
  );
}

export default App;
```

En este ejemplo, **MiComponente** es un componente funcional que recibe props título desde el componente padre (App). El componente padre utiliza **MiComponente** y le pasa valores específicos para las **props**.

Estado y ciclo de vida de los componentes

En **React,** los **"estados de vida"** se refieren a las distintas etapas que atraviesa un componente desde que se crea hasta que se elimina. Aquí tienes una explicación simplificada sin mencionar clases:

- **Montaje**

- **Constructor (opcional):**

Es un lugar para preparar el componente antes de mostrarlo.
Se utiliza para inicializar variables.

- **Render:**

Muestra la interfaz de usuario del componente.

- **ComponentDidMount:**

Se ejecuta después de que el componente se muestra en la pantalla.
Se utiliza para realizar acciones de inicio, como cargar datos desde un servidor.

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

const MontajeComponente = () => {
  // Estado para almacenar datos
  const [datos, setDatos] = useState(null);

  // Efecto que se ejecuta después del montaje
  useEffect(() => {
    // Simulando una llamada a una API
    fetch('https://api.example.com/data')
      .then(response => response.json())
      .then(data => setDatos(data));

    // Cleanup: Este return se ejecuta al desmontar el componente
    return () => {
      console.log('Componente desmontado');
    };
  }, []); // El segundo parámetro [] asegura que solo se ejecute una vez al montar

  return (
    <div>
      <h1>Componente Montaje</h1>
      {datos && <p>Datos cargados: {datos}</p>}
    </div>
  );
};
```

```
export default MontajeComponente;
```

- **Actualización**

- **Render:**

Vuelve a mostrar la interfaz de usuario si hay cambios.

- **ComponentDidUpdate:**

Se ejecuta después de que el componente se actualiza en la pantalla.
Se utiliza para realizar acciones después de un cambio, como actualizar el estado.

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

const ActualizacionComponente = () => {
  // Estado para almacenar un contador
  const [contador, setContador] = useState(0);

  // Efecto que se ejecuta después de cada actualización
  useEffect(() => {
    console.log('Componente actualizado:', contador);

    // Cleanup: Este return se ejecuta antes de la siguiente actualización
    return () => {
      console.log('Limpiando antes de la próxima actualización');
    };
  }, [contador]); // El segundo parámetro [contador] hace que se ejecute cuando el
contador cambia

  const handleIncrement = () => {
    setContador(prevContador => prevContador + 1);
  };

  return (
    <div>
      <h1>Componente Actualización</h1>
      <p>Contador: {contador}</p>
      <button onClick={handleIncrement}>Incrementar</button>
    </div>
  );
};
export default ActualizacionComponente;
```

- **Desmontaje**

- **componentWillUnmount:**

Se ejecuta justo antes de que el componente desaparezca. Se utiliza para limpiar recursos o detener procesos.

```jsx
import React, { useEffect } from 'react';

const DesmontajeComponente = () => {
  // Efecto que se ejecuta después del montaje
  useEffect(() => {
    // Cleanup: Este return se ejecuta al desmontar el componente
    return () => {
      console.log('Componente Desmontaje');
    };
  }, []); // El segundo parámetro [] asegura que solo se ejecute una vez al montar

  return (
    <div>
      <h1>Componente Desmontaje</h1>
      <p>Este componente será desmontado pronto.</p>
    </div>
  );
};

export default DesmontajeComponente;
```

Desafío #1

El objetivo de esta actividad es crear una **aplicación web simple** para mostrar una **lista de tareas** utilizando React como biblioteca de interfaz de usuario, Vite como entorno de desarrollo, y Bootstrap con Bootswatch para estilos predefinidos.
En este primer desafío haremos una **landing page** para continuar en el Desafío #2

Configuración Inicial:

1) Crea una nueva aplicación Vite con el template de React.

npm create vite@latest tarea-app --template react
cd tarea-app
npm install

2) Instala Bootstrap y Bootswatch como dependencias.

Estilos con Bootstrap y Bootswatch:

```
npm install bootstrap bootswatch
```

Configuración de Estilos en Vite

3) Actualiza la importación de estilos en `src/main.jsx` para incluir Bootstrap y Bootswatch.

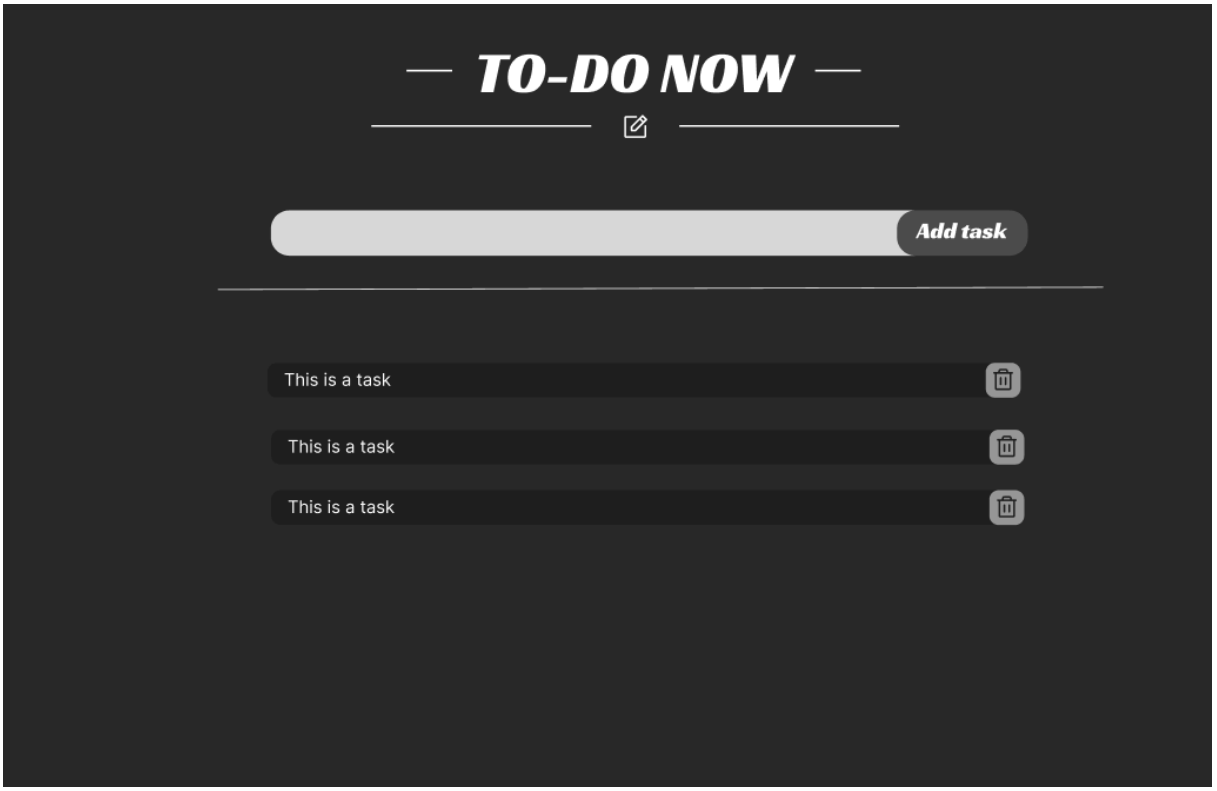
```
// src/main.jsx
import 'bootstrap/dist/css/bootstrap.min.css';
import 'bootswatch/dist/superhero/bootstrap.min.css';
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);
```

El **objetivo final** del desafío será crear un gestor de tareas. Te dejamos unos links útiles para que puedas trabajar con BootsWatch, y así, podrás ir creando los diseños antes de implementar los siguientes pasos en el Desafío #2.

Sitio oficial de BootsWatch:
https://bootswatch.com/

Imagen de ejemplo del desafío #1:




Buenos Aires
aprende
Buenos
Aires
Ciudad