«Talento Tech»

Desarrollo Web 3

Clase 03

Buenos Aires Ciudad

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Clase N° 3 | Hooks y Renderizado

Temario:

- React hooks: useState y useEffect
- Renderizado Condicional
- Objeto JSON y Consumo de APIs

React hooks: useState y useEffect

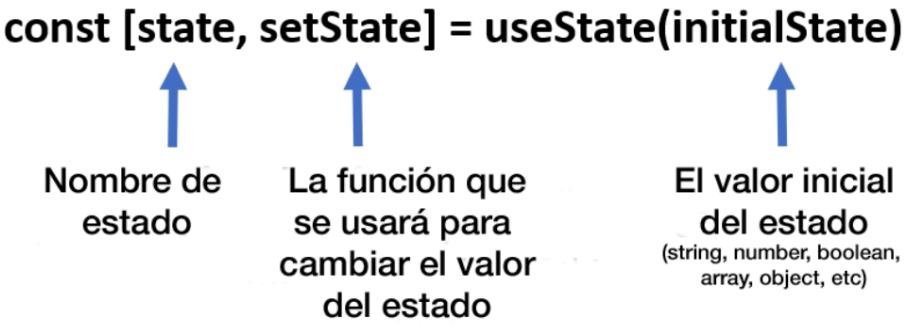
¿Qué son?

Los **hooks** en React son funciones especiales que te permiten utilizar el estado y otras características de React en componentes funcionales. Hasta la introducción de los hooks en React 16.8, las características como el estado y los ciclos de vida de los componentes solo estaban disponibles en componentes de clase. Los **hooks** ofrecen una manera más simple y concisa de trabajar con el estado y otras funcionalidades en componentes funcionales.

Los **hooks** más comunes son:

- **useState**

Permite añadir estado a componentes funcionales.

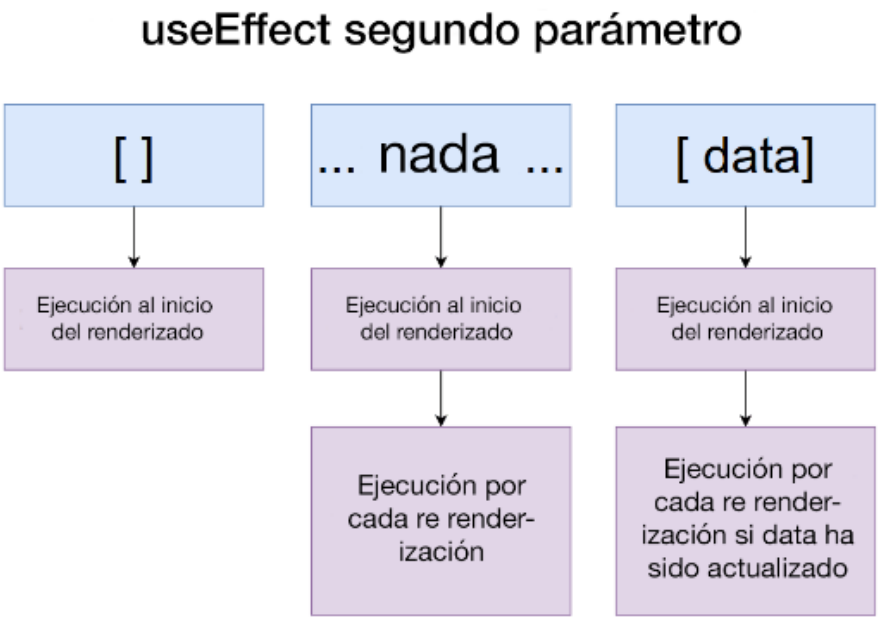



```
const [estado, setEstado] = useState(valorInicial);
```

- **useEffect**

Permite realizar efectos secundarios en componentes funcionales. Se utiliza para tareas como llamadas a APIs, suscripciones a eventos y limpieza de recursos.

Buenos Aires aprende
Agencia de Habilidades para el Futuro



El **useEffect** es otro de los hooks más usados de React. Se puede usar este hook para disparar funciones que queremos ejecutar cuando el render esté completo.

Los efectos se declaran dentro del componente para que tengan acceso a sus props y estado. De forma predeterminada, React ejecuta los efectos después de cada renderizado — incluyendo el primer renderizado.

```
useEffect(() => {
  // Código de efecto secundario
  // ...

  // Cleanup (opcional)
  return () => {
    // Código de limpieza
    // ...
  };
}, [dependencias]);
```

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Pero, ¿y qué son las dependencias?

Dependencias

En el contexto de **useEffect** y otros hooks de React, las **"dependencias"** se refieren a un array de variables que el hook debe observar. Cuando estas variables cambian, el efecto secundario dentro del **useEffect** se vuelve a ejecutar.

Aquí un ejemplo:

```
useEffect(() => {
  // Código de efecto secundario

  return () => {
    // Cleanup (opcional)
  };
}, [dependencia1, dependencia2]);
```

En este ejemplo:

- dependencia1 y dependencia2 son las dependencias.
- Si cualquiera de las dependencias cambia entre renderizaciones, el código dentro de useEffect se ejecutará nuevamente.
- Si no se proporcionan dependencias (como en useEffect(() => { /* ... */ })), el código dentro de useEffect se ejecutará en cada renderizado.

Usar dependencias es importante para evitar efectos secundarios innecesarios y optimizar el rendimiento. Si el efecto secundario depende de valores que pueden cambiar entre renderizaciones, es recomendable incluir esas variables como dependencias para que el efecto se ejecute en el momento adecuado.

Veámoslo con mejor detalle:

useState:

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

const MiComponente = () => {
  // Declaración de estado
  const [contador, setContador] = useState(0);

  // Uso del estado en el componente
  return (
    <div>
      <p>Contador: {contador}</p>
      <button onClick={() => setContador(contador + 1)}>Incrementar</button>
    </div>
  );
};
```

- **contador** es el estado, y **setContador** es la función que se utiliza para actualizar el estado.
- **useState(0)** inicializa el estado con el valor **“0”**.

- **useEffect:** Lo veremos junto al consumo de APIs!!!

Renderizado condicional

El renderizado condicional en React se utiliza para mostrar u ocultar componentes o elementos en función de ciertas condiciones. Puede ser útil cuando deseas que parte de tu interfaz de usuario aparezca solo bajo ciertas circunstancias. Hay varias formas de implementar renderizado condicional en React. Aquí hay ejemplos de algunos enfoques comunes:

Buenos Aires aprende
Agencia de Habilidades para el Futuro

1) Renderizado basado en Condiciones If-Else

```
import React from 'react';

const MiComponente = ({ condicion }) => {
  if (condicion) {
    return <p>Contenido cuando la condición es verdadera</p>;
  } else {
    return <p>Contenido cuando la condición es falsa</p>;
  }
}

export default MiComponente;
```

En este ejemplo, el componente **MiComponente** renderiza un elemento diferente dependiendo de la condición proporcionada como **prop**.

2) Renderizado basado en Operador Ternario

```
import React from 'react';

const MiComponente = ({ condicion }) => {
  return (
    <div>
      {condicion ? (
        <p>Contenido cuando la condición es verdadera</p>
      ) : (
        <p>Contenido cuando la condición es falsa</p>
      )}
    </div>
  );
}

export default MiComponente;
```

Aquí, se utiliza el operador ternario para evaluar la condición y renderizar diferentes elementos en función de si la condición es verdadera o falsa.

En **React**, puedes usar el operador condicional ternario junto con el operador lógico "**&&**" para renderizar elementos condicionalmente.

Aquí tienes un ejemplo simple:

Supongamos que tienes una variable llamada **"isLoggedIn"** que indica si un usuario está o no autenticado. Puedes usar el operador condicional ternario junto con "**&&**" para renderizar diferentes elementos según el estado de autenticación. En este ejemplo, se renderizará un mensaje diferente según si el usuario está o no autenticado:

Import React from 'react';

```
const App = () => {
  const isLoggedIn = true; // Cambia esto según el estado de autenticación de tu aplicación

  return (
    <div>
      {isLoggedIn && <p>¡Bienvenido! Estás autenticado.</p>}
    </div>
  );
};

export default App;
```

En este ejemplo, la expresión **{isLoggedIn && (¡Bienvenido! Estás autenticado.)}** renderizará el elemento solo si isLoggedIn es true. Si isLoggedIn es false, se omitirá la parte derecha del operador "&&" y no se renderizará nada. Este enfoque es útil para renderizar condicionalmente componentes en función de una condición.

Estos son solo algunos ejemplos, y la elección del enfoque dependerá de la complejidad de tu lógica condicional y de tus preferencias personales. Cualquiera de estos métodos puede ser aplicado tanto en componentes funcionales como en componentes de clase en **React.**

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Objeto JSON y APIs

Objeto JSON

JSON (JavaScript Object Notation) es un formato ligero de intercambio de datos.
Representa datos como pares clave-valor, similar a los objetos en JavaScript.
Es fácil de leer y escribir para humanos, y fácil de analizar y generar para máquinas.

API (Interfaz de Programación de Aplicaciones)

Una **API** es un conjunto de reglas que permite que un software se comunique con otro.
Proporciona métodos y estructuras de datos para acceder y manipular funcionalidades de un software o servicio.
En el contexto web, las APIs a menudo entregan datos en formato JSON.

Consumir una API en una React App

- **Fetch API:**

Usa el método **fetch** nativo de JavaScript o la librería **Axios** para hacer solicitudes HTTP a la API.

Por ejemplo, con fetch:

```
fetch('https://api.example.com/data')
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error('Error:', error));
```

Para el consumo de una API estaremos usando los hooks vistos. Para entender un poco mejor el rol del **useEffect** te dejamos el siguiente ejemplo:

Buenos Aires aprende
Agencia de Habilidades para el Futuro

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

const App = () => {
  const [userData, setUserData] = useState(null);

  useEffect(() => {
    // Replace this URL with the actual API endpoint
    const apiUrl = 'https://jsonplaceholder.typicode.com/users/1';

    // Hacer la solicitud a la API
    fetch(apiUrl)
      .then(response => response.json())
      .then(data => setPosts(data))
      .catch(error => setError(error.message))
      .finally(() => setLoading(false));
  }, []);


  return (
    <div>
      <h1>User Information</h1>
      {userData ? (
        <div>
          <p>Name: {userData.name}</p>
          <p>Email: {userData.email}</p>
          <p>Phone: {userData.phone}</p>
          {/* Add other properties as needed */}
        </div>
      ) : (
        <p>Loading...</p>
      )}
    </div>
  );
};

export default App;
```

- **useEffect** recibe dos argumentos: una función con los efectos secundarios y un array de dependencias.
- **La función dentro de useEffect** se ejecuta después del renderizado inicial y después de cada actualización si las dependencias han cambiado.
- En este ejemplo, el array de dependencias está vacío ([]), por lo que useEffect solo se ejecuta una vez después del montaje del componente.

Estos son dos de los **hooks** más fundamentales en **React** que permiten a los componentes funcionales manejar estado y efectos secundarios de manera similar a cómo lo hacen los componentes de clase.

Estos son ejemplos básicos de cómo consumir una API en una aplicación React. Ten en cuenta que en una aplicación real, es posible que desees manejar más casos, como manejar estados de carga, errores y actualizaciones de datos de manera más avanzada.


Buenos Aires
aprende
Buenos
Aires
Ciudad