Módulo 3. Introducción a ReactJS

Introducción

1. La relación con Java (backend)

2. Hook de efecto {#effect-hook}

3. Otros hooks

4. Library Hooks

Referencias

Introducción

ReactJS es una biblioteca de JavaScript para construir interfaces de usuario interactivas y reutilizables mediante componentes, mientras que Java es un lenguaje de programación orientado principalmente al desarrollo *backend*. La conexión entre ambos se establece al utilizar React para el *frontend* web y un *backend* en Java (por ejemplo, con Spring) para la lógica de negocio y la gestión de la base de datos. De este modo, se crean aplicaciones *full-stack*. React se enfoca en la capa de vista, emplea JSX para combinar HTML y JavaScript, y utiliza un DOM virtual para optimizar el rendimiento.

A continuación, se resumen las principales características de ReactJS:

- Es una biblioteca de JavaScript basada en componentes para construir interfaces de

usuario. La interfaz se estructura a partir de piezas independientes y reutilizables —como bloques de Lego—, cada una con su propia lógica y apariencia.

- Fue creado por Facebook (Meta) en 2013.
- Permite desarrollar aplicaciones modulares y reutilizables gracias a su enfoque basado en componentes.
- Utiliza un DOM virtual que facilita una actualización eficiente de la interfaz.
- Adopta un enfoque declarativo: se describe cómo debe verse la interfaz según un estado de datos determinado, y React se encarga de actualizarla de forma eficiente.
- Emplea JSX, una extensión de JavaScript que permite escribir sintaxis similar a HTML dentro del código, lo que facilita la creación de componentes.
- El DOM virtual actúa como una copia en memoria del DOM real y se usa para calcular los cambios y actualizar solo lo necesario, lo cual mejora el rendimiento.

Este enfoque resulta adecuado para aplicaciones web dinámicas, de una sola página (*single page applications*) y móviles, mediante React Native.

CONTINUAR

1. La relación con Java (backend)

React maneja la experiencia del usuario —la «vista» o «V» en un patrón MVC—, mientras que Java se encarga de la lógica del servidor, la base de datos y las API que consume React. Esta separación de responsabilidades permite una estructura clara y mantenible en el desarrollo de aplicaciones.

Un desarrollador puede usar Java para construir un servidor robusto y React para diseñar una interfaz de usuario moderna y dinámica que se comunica con ese servidor mediante API. Por ejemplo, en una aplicación de comercio electrónico, Java y Spring pueden encargarse del manejo de pedidos en el *backend*, mientras que React muestra los productos y el carrito de compras en el *frontend*.

React es una tecnología orientada al navegador que, aunque escrita en JavaScript, se integra eficazmente con *backends* en Java para crear aplicaciones completas y con buen

rendimiento, separando de forma clara la presentación de la lógica del negocio.

VIRTUAL DOM EN JAVA | BENEFICIOS CLAVE

El árbol DOM (*Document Object Model*) en Java es una API que permite representar y manipular documentos estructurados —como HTML o XML— como un árbol jerárquico de nodos (elementos, texto, atributos). Esto permite que los programas escritos en Java puedan leer, modificar o crear el contenido y la estructura del documento, accediendo a cada etiqueta como un objeto en memoria.

A partir de este modelo, algunas bibliotecas de JavaScript, como React, implementan el *virtual DOM* (VDOM), una representación en memoria del DOM real (es decir, del árbol de elementos HTML de una página web). Esta técnica permite actualizar la interfaz de usuario de forma eficiente, ya que React compara el estado anterior y el nuevo del *virtual DOM* para calcular los cambios mínimos necesarios y aplicarlos luego al DOM real. De este modo, se evita redibujar toda la página y se mejora el rendimiento, especialmente en aplicaciones dinámicas y complejas.

Aunque el término *virtual DOM* está muy asociado a JavaScript y React, el concepto de mantener una representación ligera en memoria para optimizar los procesos de renderizado puede aplicarse en otros contextos. Sin embargo, el *virtual DOM* como tal es una implementación específica de bibliotecas de *frontend*.

¿Cómo funciona?

El proceso de actualización mediante *virtual DOM* se desarrolla en tres etapas principales:

- **Cambio de estado y generación del *virtual DOM***. Cuando cambia el estado de la aplicación —por ejemplo, al editar un dato en una tabla—, se genera en memoria un nuevo *virtual DOM*.
- **Comparación o *diffing***: el *framework* (como React) compara este nuevo *virtual DOM* con el anterior para identificar exactamente qué elementos han cambiado.
- **Reconciliación o actualización**: solo las partes modificadas se actualizan en el DOM real del navegador, en lugar de reconstruir todo el árbol, lo que permite un rendimiento mucho mayor.

VIRTUAL DOM EN JAVA

BENEFICIOS CLAVE

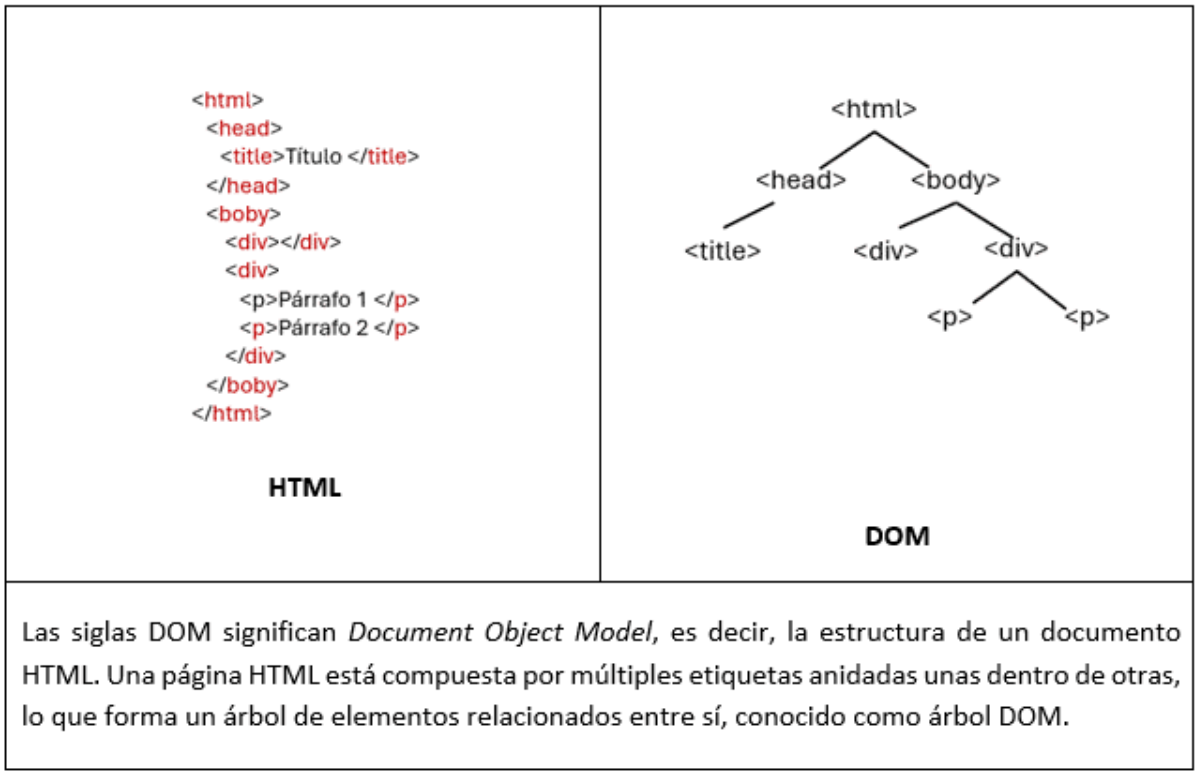
El uso del *virtual DOM* presenta varias ventajas en el desarrollo de interfaces dinámicas. Entre ellas, se destacan las siguientes:

- Mejora del rendimiento, ya que permite realizar actualizaciones mínimas y eficientes, lo que resulta útil en interfaces con numerosos cambios.
- Enfoque declarativo, que facilita describir el estado deseado de la interfaz sin necesidad de manipular directamente el DOM; el *framework* se encarga de aplicar los cambios necesarios.

- Mayor nivel de abstracción, al ocultar la complejidad de la manipulación directa del DOM y simplificar el desarrollo de componentes.

A continuación, se presenta un ejemplo visual que permite comprender la diferencia entre el código HTML y su representación en forma de árbol DOM:

Tabla 1. Relación entre el código HTML y el árbol DOM



Fuente: elaboración propia.

A continuación, se presenta de forma gráfica el concepto de *virtual DOM*. Para ilustrarlo, se parte de una tabla HTML inicial (tabla 2), en la que un usuario modifica el valor de una celda. El resultado de ese cambio se observa en la tabla 3.

Tabla 2. Estado inicial de la tabla en la interfaz

Dato 1	Dato 2	Dato 3	**Editar**
Dato 4	Dato 5	Dato 6	**Editar**
Dato 7	Dato 8	Dato 9	**Editar**

Fuente: Álvarez Caules, 2016, https://goo.su/bYvgUh

Tabla 3. Estado actualizado tras modificar una celda

Dato 1	Dato 2	Dato 3	**Editar**
Dato 4	Dato 5	Dato 6	**Editar**
Dato 7	Dato 8	**Dato 10**	**Editar**

Fuente: Álvarez Caules, 2016, https://goo.su/bYvgUh

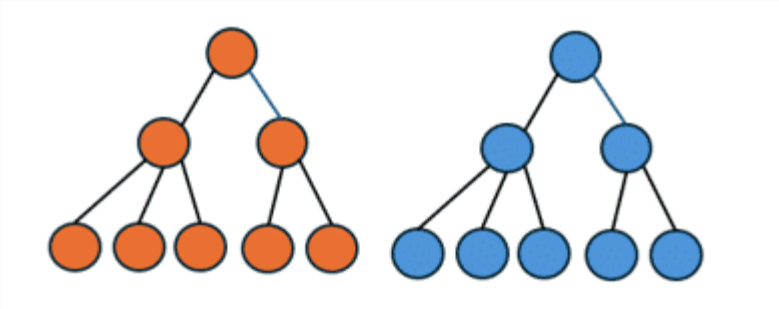
Como se observa en las tablas anteriores, una pequeña modificación en el contenido de una celda puede gestionarse de distintas formas según se utilice o no un *virtual DOM*. A continuación, se explica cómo funciona cada enfoque:

- **Sin *virtual DOM***. Cuando se desarrolla el código fuente de esta tabla, normalmente se parte de un *array* de objetos. Ante cada modificación de datos, el navegador redibuja toda la tabla, ya que vuelve a renderizar por completo el contenido.

- **Con *virtual DOM***: cambiar los nodos del árbol DOM tiene un coste, que aumenta a medida que se modifican más elementos. En estructuras simples, esto puede pasar desapercibido; sin embargo, en aplicaciones más grandes y complejas, el impacto en el

rendimiento se vuelve significativo. En este ejemplo, la tabla representa una sección del árbol DOM que puede actualizarse de manera eficiente mediante el *virtual DOM*, aplicando solo los cambios necesarios.

Figura 1. Comparación visual entre DOM (izquierda) y *virtual DOM* (derecha) ante una actualización

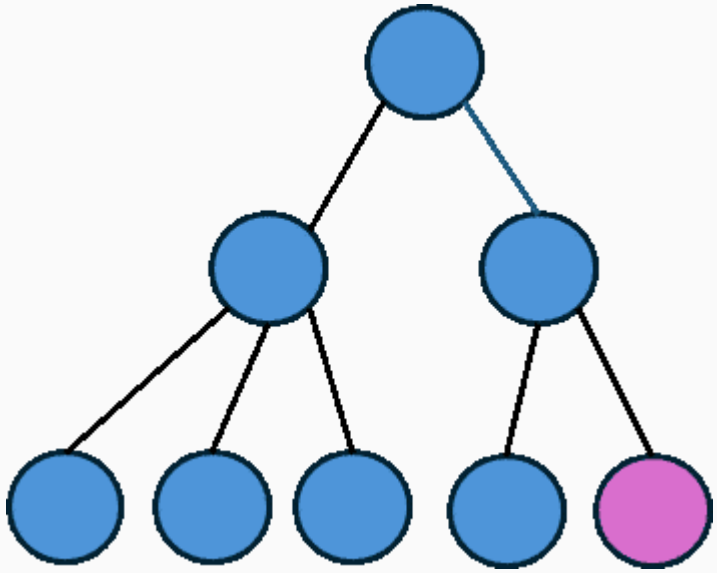


Fuente: Álvarez Caules, 2016, https://goo.su/bYvgUh

Una vez que se ha generado el *virtual DOM*, el *framework* se encarga de supervisar los cambios que se producen sobre esa estructura. De este modo, puede detectar las diferencias entre un estado anterior (A) y uno nuevo (B) cuando se actualiza la información.

A partir de estos cambios mínimos, la biblioteca o el *framework* actualiza únicamente las partes necesarias del árbol DOM real, sin necesidad de redibujarlo por completo.

Figura 2. Actualización parcial del árbol mediante *virtual DOM*



Fuente: Álvarez Caules, 2016, https://goo.su/bYvgUh

Con este tipo de técnicas, se reducen al mínimo los procesos de renderizado en el navegador, lo que contribuye a mejorar

significativamente el rendimiento. El *virtual DOM* es un patrón utilizado para actualizar interfaces web de forma rápida y eficiente. Es ampliamente adoptado en tecnologías como React, y se basa en mantener una copia ligera del DOM en memoria para optimizar los cambios sobre el DOM real.

COMPONENTES DE REACT	PROPS

Los componentes de React son bloques reutilizables que permiten construir interfaces de usuario (*UI*) dividiendo aplicaciones complejas en partes manejables. Cada componente encapsula su propia lógica y apariencia, y puede representar elementos como botones, tarjetas u otros elementos visuales.

Se definen principalmente como funciones de JavaScript —conocidas como componentes funcionales—, aunque también pueden encontrarse componentes de clase, utilizados en versiones anteriores. Los componentes utilizan *«props»* —mecanismo para pasar datos de un componente padre a un componente hijo— para recibir datos desde otros componentes, y emplean la sintaxis JSX (*JavaScript XML*) para combinar JavaScript con HTML dentro de una misma estructura y así describir la interfaz de usuario.

Tipos principales

A continuación, se presentan los principales tipos de componentes utilizados en React:

- **Componentes funcionales**. Son la forma moderna y más utilizada. Se definen como funciones de JavaScript que reciben *props* y devuelven elementos de React. Gracias al uso de *hooks*, como `useState` y `useEffect`, también pueden gestionar estado y efectos secundarios.
- **Componentes de clase**. Basados en clases de ES6, permiten manejar estado y ciclo de vida, pero su uso ha disminuido en favor de los componentes funcionales, que resultan más simples y concisos con los *hooks*.

Características clave

Entre las principales características de los componentes en React, se destacan las siguientes:

- **Reutilizables**. Se crean una vez y pueden utilizarse en distintas partes de la aplicación.
- **Jerárquicos**: pueden anidarse entre sí, formando una estructura en árbol. Por ejemplo, un componente `App` puede contener múltiples componentes `Button`.
- **Uso de JSX**: permite escribir HTML dentro de JavaScript de forma intuitiva. Los nombres de los componentes comienzan con mayúscula para diferenciarlos de las etiquetas HTML, como en `<MiBoton />` frente a `<div>`.
- **Props**: mecanismo para pasar datos de un componente padre a uno hijo. Son inmutables (de solo lectura) y pueden ser de

cualquier tipo admitido por JavaScript, como objetos, arreglos o funciones.

- **Estado**: tanto los componentes funcionales (mediante *hooks*) como los de clase pueden gestionar su propio estado interno, lo que permite que la interfaz se actualice dinámicamente según los cambios en ese estado.

¿Cómo funcionan?

A continuación, se describe cómo funciona un componente en React desde su definición hasta su uso en la interfaz:

1. **Definición**. Se escribe una función (o una clase) que retorna un elemento JSX y se exporta para su reutilización.
2. **Uso**: el componente se importa en otro archivo y se utiliza como si fuera una etiqueta HTML. Por ejemplo, `<MiBoton nombre="Usuario" />`.

Renderizado: React interpreta el JSX y lo representa en el navegador, actualizando solo las partes necesarias de la interfaz cuando cambian los datos o el estado.

COMPONENTES DE REACT	PROPS

En React, las *props* (abreviatura de *properties*) son el mecanismo mediante el cual un componente padre pasa datos a un componente hijo. Funcionan como parámetros que permiten configurar y reutilizar componentes, de manera similar a los atributos HTML, pero con la posibilidad de transmitir

cualquier tipo de dato de JavaScript, como cadenas de texto, números, objetos o funciones.

Las *props* son de solo lectura desde el punto de vista del componente hijo y permiten un flujo de datos unidireccional, es decir, de arriba hacia abajo dentro de la jerarquía de componentes. A continuación, se muestra un ejemplo relacionado con el uso del estado en componentes funcionales.

1. **Uso de *props* en un componente**

En el archivo `main.jsx`, se utiliza el componente `PrimerComponente` y se le envía una *prop* llamada `Nombre` con el valor `"Juan"`. En `primerComponente.jsx`, esa *prop* se recibe como argumento y se accede a su valor mediante `props.Nombre`.

Figura 3. Versión inicial del archivo donde se llama al componente con la *prop* `Nombre`. Archivo: `main.jsx`

```
import { StrictMode } from 'react'
import { createRoot } from 'react-dom/client'
import PrimerComponente from './assets/PrimerComponente'

createRoot(document.getElementById('root')).render(
  <StrictMode>

    <PrimerComponente Nombre= "Juan" />

  </StrictMode>,
)
```

Fuente: elaboración propia

Figura 4. Uso de la prop dentro del componente PrimerComponente. Archivo: primerComponente.jsx

```
import React from 'react'

const PrimerComponente = (props) => {

  return (

    <>

    <h1>Hola {props.Nombre} </h1>

    </>

  )
}

export default PrimerComponente
```

Fuente: elaboración propia

Nota: los círculos señalan el código que se modifica para mostrar el valor recibido por props.

2. **Envío de múltiples props y uso de desestructuración**

Ahora se envían dos props al componente: Nombre y edad. En lugar de acceder a ellas mediante props.Nombre y props.edad, se utiliza desestructuración directamente en los parámetros del componente, lo que simplifica el código.

Figura 5. Desestructuración de props y visualización del flujo de datos entre componentes. Archivos mostrados: main.jsx y primerComponente.jsx

Fuente: elaboración propia

Nota: las flechas rojas indican cómo se envían las props desde main.jsx y cómo se utilizan dentro de primerComponente.jsx.

3. **Resultado final en el navegador**

Como resultado del uso de *props* y su renderizado dinámico, se muestra el siguiente mensaje en pantalla:

Figura 6. Resultado de la ejecución del código mostrado

Fuente: elaboración propia

<u>Reutilización de componentes en React</u>

Para mostrar cómo funciona la reutilización del código, se modifica el ejemplo anterior. Ahora se renderizan varias instancias del mismo componente PrimerComponente, pero con diferentes *props*. Cada una genera un botón con un texto distinto según el valor de la prop Operacion.

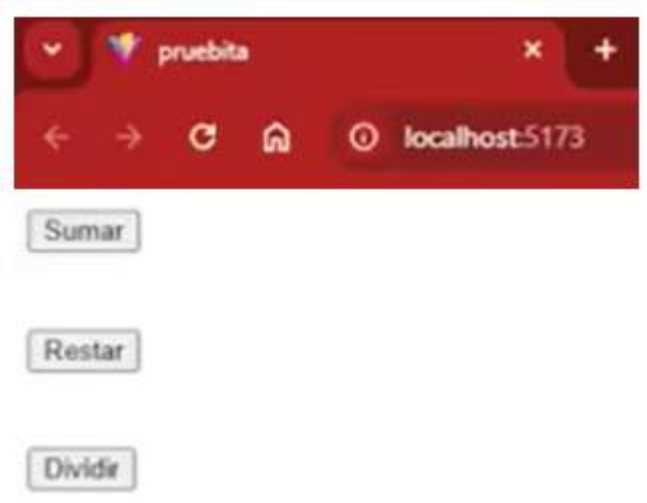
Figura 7. Reutilización del componente con distintas props y visualización del flujo de datos. Archivos mostrados: main.jsx y primerComponente.jsx

Fuente: elaboración propia

Nota: las flechas rojas muestran cómo cada valor de la prop `Operacion` se envía desde `main.jsx` y se utiliza dentro del componente para generar botones con texto personalizado.

A continuación, se muestra el resultado generado a partir del código anterior: tres botones renderizados dinámicamente con las operaciones indicadas.

Figura 8. Resultado de la ejecución en el navegador



Fuente: elaboración propia.

Componente —

Un componente en React es una unidad independiente y reutilizable de la interfaz de usuario (*UI*) que encapsula su propia lógica y diseño, combinando HTML, CSS y JavaScript en un único bloque de código. Este enfoque permite construir aplicaciones complejas de forma modular y escalable.

Los componentes pueden definirse como funciones —componentes funcionales— o como clases —componentes de clase—, y aceptan entradas llamadas *props*, que permiten personalizar su comportamiento y su apariencia.

JSX —

JSX (*JavaScript XML*) es una extensión de sintaxis para JavaScript que permite escribir código similar a HTML dentro de archivos JavaScript. Esto permite combinar la lógica de renderizado y la estructura de la interfaz de usuario en un mismo lugar, lo que vuelve más intuitiva la creación de componentes en React.

Aunque su sintaxis se asemeja a HTML, JSX se transforma en llamadas a `React.createElement()` para que el navegador pueda interpretarlo correctamente.

Hooks —

Un *hook* en React es una función especial que permite acceder al estado y al ciclo de vida desde componentes funcionales, sin necesidad de utilizar clases. Con los *hooks* es posible incorporar funcionalidades como el manejo de estado (`useState`), los efectos secundarios (`useEffect`) o el uso de contexto, lo que permite escribir código más limpio y reutilizable. Los *hooks* fueron incorporados a partir de React 16.8 y representan una forma moderna y eficiente de gestionar la lógica de los componentes.

A partir de estas capacidades, los *hooks* hacen posible incorporar distintas funcionalidades en los componentes funcionales, entre las que se destacan las siguientes:

- Acceder a funcionalidades de React, como el estado, desde componentes funcionales, algo que anteriormente solo era posible en componentes de clase.
- Reutilizar lógica con estado entre diferentes componentes mediante la creación de *hooks* personalizados.
- Separar la lógica según su finalidad —por ejemplo, estado o efectos— en lugar de hacerlo por métodos del ciclo de vida, lo que vuelve el código más claro y legible.

React incorpora varios *hooks* que cubren necesidades comunes en el desarrollo de componentes:

- `useState`, para añadir estado a un componente funcional.

- `useEffect`, para manejar efectos secundarios, como llamadas a API o suscripciones.
- `useContext`, para acceder al contexto de React sin necesidad de pasar *props* manualmente.

Su uso exige cumplir ciertas reglas básicas:

- Deben llamarse siempre en el nivel superior de una función de componente o de otro *hook*, nunca dentro de bucles, condicionales ni funciones anidadas.
- Solo pueden emplearse dentro de componentes funcionales o de otros *hooks* personalizados.

Los *hooks* son el enfoque moderno y recomendado para gestionar estado y lógica en componentes funcionales. Su uso permite construir código más modular, legible y fácil de mantener.

A continuación, se desarrollan algunos *hooks* fundamentales con ejemplos de uso. Parte del material ha sido tomado de las referencias indicadas para que los estudiantes puedan profundizar en los conceptos.

CONTINUAR

2. Hook de efecto {#effect-hook}

El Hook de efecto, useEffect, agrega la capacidad de realizar efectos secundarios desde un El *hook* de efecto, *useEffect*, permite realizar efectos secundarios desde un componente de función. Cumple el mismo propósito que componentDidMount, componentDidUpdate y componentWillUnmount en las clases de React, pero todo se encuentra unificado en una sola API.

Los *hooks* son una incorporación de React 16.8. Permiten utilizar estado y otras características de React sin escribir una

clase. Por ejemplo, este componente establece el título del documento después de que React actualiza el DOM:

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

function Example() {

 const [count, setCount] = useState(0) ;

      // Similar a componentDidMount y
componentDidUpdate:

 useEffect(() => {

   // Actualiza el título del documento usando la Browser
API

  document.title = `Ud.hizo click ${count} veces;`

 });

 return (

  <div>

   <p>Ud.hizo click {count} veces</p>
```

```
      <button onClick={() => setCount(count + 1)}>

        Haga click aquí

      </button>

    </div>

  );

}
```

Cuando se invoca useEffect, se indica a React que ejecute la función de efecto después de aplicar los cambios en el DOM. Los efectos se declaran dentro del componente para que puedan acceder a sus *props* y al estado. Por defecto, React ejecuta los efectos después de cada renderizado, incluido el primero.

Los efectos también pueden especificar, de forma opcional, cómo limpiarse después de ejecutarse, mediante el retorno de una función. A continuación, se muestra un ejemplo en el que un componente se suscribe al estado en línea de un contacto, y cancela la suscripción al desmontarse:

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

function FriendStatus(props) {

 const [isOnline, setIsOnline] = useState(null);

 function handleStatusChange(status) {

  setIsOnline(status.isOnline);

 }

 useEffect(() => {

                ChatAPI.subscribeToFriendStatus(props.friend.id,
handleStatusChange);

  return () => {

      ChatAPI.unsubscribeFromFriendStatus(props.friend.id,
handleStatusChange);

  };

 });
```

```
  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}
```

En este caso, React cancelará la suscripción a la API de chat cuando el componente se desmonte, así como antes de volver a ejecutar el efecto tras un nuevo renderizado.

Reglas de hooks {#rules-of-hooks}

Los *hooks* son funciones de JavaScript, pero imponen dos reglas adicionales:

- Llamarlos solo en el nivel superior. Es decir, no se deben invocar dentro de bucles, condiciones o funciones anidadas.
- Llamarlos únicamente desde componentes de función de React. Es decir, no se deben usar en funciones regulares de

JavaScript. (También pueden llamarse desde *hooks* personalizados).

Construcción de hooks personalizados

En algunas ocasiones, es necesario reutilizar cierta lógica de estado entre componentes. Tradicionalmente, existían dos soluciones comunes para este problema: los componentes de orden superior y las *render props*. Los *hooks* personalizados permiten resolverlo sin agregar más componentes al árbol. A continuación, se presenta un ejemplo:

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

function useFriendStatus(props) {
 const [isOnline, setIsOnline] = useState(null);

 function handleStatusChange(status) {
  setIsOnline(status.isOnline);
 }

 useEffect(() => {
                                ChatAPI.subscribeToFriendStatus(friend.id,
handleStatusChange);
  return () => {
                              ChatAPI.unsubscribeFromFriendStatus(friend.id,
handleStatusChange);
  };
 });

 return isOnline;
}
```

Este *hook* toma friendID como argumento y retorna si el contacto está en línea o no. Ahora puede utilizarse desde diferentes componentes:

```
function FriendStatus(props) {
 const isOnline = useFriendStatus(props.friend.id);

 if (isOnline === null) {
  return 'Loading...';
 }
 return isOnline ? 'Online' : 'Offline';
}
```

```
function FriendStatus(props) {
 const isOnline = useFriendStatus(props.friend.id);

  return (
   <li style={{ color: isOnline ? 'green' : 'black' }}>
    {props.friend.name}
   </li>
 );
}
```

El estado de cada componente es completamente independiente. Los *hooks* permiten reutilizar la lógica de estado, pero no el estado en sí. De hecho, cada invocación de un *hook* mantiene su propio estado — por lo tanto, es posible usar el mismo *hook* personalizado varias veces dentro de un mismo componente.

Los *hooks* personalizados son más una convención que una funcionalidad en sí misma. Si el nombre de una función comienza con «use» y utiliza otros *hooks*, se considera un *hook* personalizado.

La convención de nomenclatura `useSomething` permite que herramientas como el *linter* detecten errores en el uso de *hooks*.

Es posible escribir *hooks* personalizados para una gran variedad de casos de uso, como el manejo de formularios, animaciones, suscripciones declarativas, temporizadores y muchos otros que aún no se han considerado.

CONTINUAR

3. Otros hooks

Existen algunos *hooks* incorporados, de uso menos frecuente, que pueden resultar útiles en determinados casos. Por ejemplo, `useContext` permite suscribirse al contexto de React sin necesidad de generar anidamiento:

```
function Example() {

  const locale = useContext(LocaleContext); const theme = useContext(ThemeContext); // ...

}
```

Por su parte, `useReducer` permite gestionar el estado local de componentes complejos mediante un *reducer*:

```
function Todos() {

 const [todos, dispatch] = useReducer(todosReducer);
```

// ...

Referencia de la API de los hooks

Dependiendo de la necesidad, se pueden utilizar distintos *hooks*. A continuación, se presentan los principales.

HOOKS BÁSICOS | HOOKS ADICIONALES

- **useStat**

El *hook* useState permite agregar estado local a un componente de función. Su sintaxis es la siguiente:

```
const [state, setState] = useState(initialState);
```

Devuelve un valor con estado y una función para actualizarlo. Durante el renderizado inicial, el estado (state) es igual al valor proporcionado como primer argumento (initialState). La función setState se utiliza para actualizar el estado: acepta un nuevo valor y coloca en la cola una nueva renderización del componente:

```
setState(newState);
```

En los renderizados siguientes, el primer valor devuelto por useState será siempre el estado más reciente, después de aplicar las actualizaciones.

Nota: React garantiza que la identidad de la función `setState` es estable y no cambiará en los renderizados posteriores. Por eso, es seguro omitirla de la lista de dependencias en `useEffect` o `useCallback`.

- **`useEffect`**

El *hook* `useEffect` permite ejecutar código imperativo con efectos secundarios después de que un componente se ha renderizado. Su sintaxis básica es la siguiente:

```
useEffect(didUpdate);
```

Las mutaciones, suscripciones, temporizadores, registros y otros efectos secundarios **no deben colocarse dentro del cuerpo principal del componente** (fase de renderizado de React). Si se hace, pueden aparecer errores confusos o inconsistencias en la interfaz de usuario.

En su lugar, se debe utilizar `useEffect`. La función pasada a este *hook* se ejecutará después de que el renderizado se haya confirmado en pantalla. Se puede pensar en los efectos como una vía de escape que React ofrece para salir del mundo puramente funcional y acceder al mundo imperativo.

Por defecto, los efectos se ejecutan después de cada renderizado completado, aunque es posible configurarlos para que se ejecuten únicamente cuando cambian ciertos valores.

- **Limpiando un efecto**

A menudo, los efectos crean recursos que deben limpiarse antes de que el componente se elimine de la pantalla, como una suscripción

o un ID de temporizador. Para manejar esto, la función pasada a useEffect puede devolver otra función que realice la limpieza. Por ejemplo, para crear una suscripción:

```
useEffect(() => {
 const subscription = props.source.subscribe();
 return () => {
   // Limpiar la suscripción
   subscription.unsubscribe();
 };
});
```

La función de limpieza se ejecuta antes de que el componente se elimine de la interfaz de usuario, evitando pérdidas de memoria. Además, si un componente se renderiza varias veces (como suele ocurrir en React), el efecto anterior se limpia antes de ejecutar el siguiente.

En el ejemplo anterior, esto significa que se crea una nueva suscripción en cada actualización. Para evitar que el efecto se ejecute en cada renderizado, se debe configurar su dependencia, como se explica en la siguiente sección.

- **useContext**

El *hook* useContext permite acceder al valor de un contexto de React desde un componente de función. Su sintaxis básica es la siguiente:

```
const value = useContext(MyContext);
```

Recibe un objeto de contexto (el valor devuelto por React.createContext) y devuelve el valor actual de ese contexto. El valor actual se determina por la propiedad value del <MyContext.Provider> más cercano en el árbol que engloba al componente que hace la llamada.

Cuando el `<MyContext.Provider>` más cercano se actualiza, el *hook* provoca una nueva renderización del componente con el valor más reciente del contexto. Incluso si un ancestro utiliza `React.memo` o `shouldComponentUpdate`, la renderización se ejecuta a partir del componente que llama a `useContext`.

El argumento que se pasa a `useContext` debe ser **el objeto de contexto**, no el consumidor ni el proveedor:

- Correcto: `useContext(MyContext)`
- Incorrecto: `useContext(MyContext.Consumer)`
- Incorrecto: `useContext(MyContext.Provider)`

Un componente que llama a `useContext` siempre se volverá a renderizar cuando cambie el valor del contexto. Si la renderización del componente es costosa, se puede optimizar usando memorización.

HOOKS BÁSICOS | **HOOKS ADICIONALES**

A continuación, se mencionan algunos *hooks* adicionales de React.

- **`useReducer`**

El *hook* `useReducer` es una alternativa a `useState` que permite manejar estados más complejos en un componente de función. Su sintaxis básica es la siguiente:

```
const [state, dispatch] = useReducer(reducer, initialArg, init);
```

Este *hook* recibe un *reducer* de tipo `(state, action) => newState` y devuelve el estado actual junto con un método `dispatch` para enviar acciones.

`useReducer` suele ser preferible a `useState` cuando se tiene una lógica compleja que involucra múltiples subvalores o cuando el próximo estado depende del anterior. Además, permite optimizar el rendimiento en componentes que generan actualizaciones profundas.

- **`useCallback`**

El *hook* `useCallback` permite memorizar funciones para que su referencia se mantenga estable entre renderizados. Su sintaxis básica es la siguiente:

```
const memoizedCallback = useCallback(
 () => {
  doSomething(a, b);
 },
 [a, b],
);
```

Recibe un *callback* en línea y un arreglo de dependencias. `useCallback` devuelve una versión memorizada del *callback* que solo cambia si alguna de las dependencias ha cambiado.
Esto es útil cuando se pasan *callbacks* a componentes hijos optimizados que dependen de la igualdad de referencia para evitar renderizados innecesarios (por ejemplo, cuando se usa `shouldComponentUpdate`).

- **`useMemo`**

El *hook* useMemo permite memorizar un valor calculado para evitar recomputaciones innecesarias en cada renderizado. Su sintaxis básica es la siguiente:

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```

Recibe **una función de «crear»** que devuelve el valor a memorizar y un arreglo de dependencias. useMemo solo volverá a calcular el valor memorizado cuando alguna de las dependencias haya cambiado. Esta optimización ayuda a evitar cálculos costosos en cada renderizado.

Importante: la función pasada a useMemo se ejecuta durante el renderizado. No se deben realizar efectos secundarios allí; los efectos secundarios pertenecen a useEffect.

- **useRef**

El *hook* useRef permite crear un **objeto ref mutable** cuya propiedad .current se inicializa con el valor pasado como argumento (initialValue). El objeto devuelto se mantiene persistente durante toda la vida del componente. Su sintaxis básica es la siguiente:

```
const refContainer = useRef(initialValue);
```

Veamos un ejemplo:

```
function TextInputWithFocusButton() {
 const inputEl = useRef(null);
 const onButtonClick = () => {
  // current apunta al elemento de entrada de texto
montado
  inputEl.current.focus();
 };
 return (
```

```
    <>
      <input ref={inputEl} type="text" />
      <button onClick={onButtonClick}>Focus the input</button>
    </>
  );
}
```

- **useImperativeHandle**

El *hook* useImperativeHandle permite **personalizar el valor de la instancia** que se expone a los componentes padres cuando se usa una ref. Su sintaxis básica es la siguiente:

```
useImperativeHandle(ref, createHandle, [deps])
```

Se debe usar junto con forwardRef. Como regla general, el código imperativo que utiliza refs debe evitarse en la mayoría de los casos, y useImperativeHandle se emplea únicamente cuando se necesita exponer funcionalidades específicas de un componente hijo a su padre.

- **useLayoutEffect**

useLayoutEffect se utiliza de manera muy similar a useEffect, pero se ejecuta de forma **síncrona** después de que se han aplicado todas las mutaciones del DOM. Esto permite leer el diseño del DOM y realizar actualizaciones inmediatamente, antes de que el navegador pinte la pantalla.

Las actualizaciones programadas dentro de useLayoutEffect se aplican de manera inmediata, lo que puede bloquear el renderizado

visual si se abusa de ellas. Por eso, se recomienda usar useEffect siempre que sea posible, para evitar problemas de rendimiento y parpadeos en la interfaz.

- **useDebugValue**

El *hook* useDebugValue puede usarse para mostrar una etiqueta en React DevTools cuando se trabaja con *hooks* personalizados. Su sintaxis básica es la siguiente:

```
useDebugValue(value)
```

Por ejemplo, considerando un *hook* personalizado useFriendStatus:

```
function useFriendStatus(friendID) {
 const [isOnline, setIsOnline] = useState(null);

 // ...

// Mostrar una etiqueta en DevTools junto a este Hook
// por ejemplo: "FriendStatus: Online"
useDebugValue(isOnline ? 'Online' : 'Offline');
 return isOnline;
}
```

- **Aplazar el formato de los valores de depuración**

En algunos casos, formatear un valor para mostrar puede ser costoso y, además, innecesario si el *hook* no se inspecciona. Por esta razón, useDebugValue acepta una función de formato como **segundo parámetro opcional**. Esta función solo se ejecuta si se inspeccionan los *hooks*, recibe el valor de depuración como parámetro y debe devolver un valor formateado para mostrar.

Por ejemplo, un *hook* personalizado que devuelve un objeto `Date` podría evitar llamar a `toDateString` innecesariamente usando un formateador:

```
useDebugValue(date, date => date.toDateString());
```

- **`useDeferredValue`**

El *hook* `useDeferredValue` permite posponer actualizaciones de un valor frente a otras actualizaciones más urgentes. Su sintaxis básica es la siguiente:

```
const deferredValue = useDeferredValue(value);
```

Recibe un valor y devuelve una nueva copia del valor que se aplicará de manera diferida frente a actualizaciones urgentes. Por ejemplo, si el renderizado actual se debe a una actualización urgente, como la entrada del usuario, React devolverá primero el valor anterior y luego actualizará al nuevo valor una vez finalizado el renderizado urgente.

Este *hook* es similar a técnicas de optimización como la eliminación de rebotes (*debounce*) o la limitación de velocidad (*throttle*), pero con la ventaja de que React procesa la actualización tan pronto como termina otro trabajo, en lugar de esperar un tiempo arbitrario. Además, al igual que `startTransition`, los valores diferidos pueden suspenderse sin causar una recuperación inesperada del contenido existente.

- **Memorización de componentes secundarios diferidos**

`useDeferredValue` solo difiere el valor que se le pasa. Si se desea evitar que un componente secundario se vuelva a renderizar

durante una actualización urgente, también debe memorizarse usando React.memo o React.useMemo.

- **useTransition**

El *hook* useTransition permite manejar actualizaciones como transiciones, diferenciando entre actualizaciones urgentes y menos prioritarias. Su sintaxis básica es la siguiente:

```
const [isPending, startTransition] = useTransition();
```

Devuelve un **estado** que indica si la transición está pendiente (isPending) y una **función** para iniciar la transición (startTransition). startTransition permite marcar las actualizaciones que se ejecutan en la devolución de llamada proporcionada como transiciones:

```
startTransition(() => {
 setCount(count + 1);
});
```

isPending indica cuándo una transición está activa, lo que permite mostrar un estado de carga o pendiente mientras se realiza la actualización.

- **useId**

El *hook* useId se utiliza para generar **identificadores únicos** que son estables tanto en el servidor como en el cliente, evitando desajustes durante la hidratación. Su sintaxis básica es la siguiente:

```
const id = useId();
```

useId no debe utilizarse para generar claves en listas. Las claves deben derivarse de los datos.

A conitnuación, veamos un ejemplo:

```
function Checkbox() {
 const id = useId();
 return (
   <>
     <label htmlFor={id}> ¿Te gusta React?  </label>
     <input id={id} type="checkbox" name="react"/>
   </>
 );
};
```

CONTINUAR

4. Library Hooks

Los siguientes *hooks* se proporcionan para que los autores de bibliotecas integren su código profundamente en el modelo de React. Normalmente, **no se usan en aplicaciones**, sino en bibliotecas que requieren compatibilidad con renderizado concurrente y otras características avanzadas.

- **`useSyncExternalStore`**

El *hook* `useSyncExternalStore` se recomienda para leer y suscribirse a **fuentes de datos externas**, siendo compatible con renderizado concurrente, hidratación selectiva y segmentación de tiempo. Su sintaxis básica es la siguiente:

```
const state = useSyncExternalStore(subscribe, getSnapshot[, getServerSnapshot]);
```

Este *hook* recibe tres argumentos:

- `subscribe`. Función para registrar una devolución de llamada que se ejecuta cada vez que cambia la fuente de datos.
- `getSnapshot`: función que devuelve el valor actual de la fuente de datos.
- `getServerSnapshot` (opcional): función que devuelve la instantánea usada durante el renderizado en el servidor.

A continuación, se presenta un ejemplo básico:

```
Const state = useSyncExternalStore(store.subscribe,
store.getSnapshot);
```

Este ejemplo se suscribe a todos los cambios de la tienda y mantiene el estado actualizado en el componente.

- **`useInsertionEffect`**

When we show up to the present moment with all of our senses, we invite the world to fill us with joEl *hook* `useInsertionEffect` tiene una firma idéntica a `useEffect`, pero se ejecuta de forma síncrona antes de todas las mutaciones del DOM. Su uso principal es inyectar estilos en el DOM antes de leer el diseño con `useLayoutEffect`.

```
useInsertionEffect(didUpdate);
```

Importante: este *hook* tiene un alcance muy limitado, ya que no tiene acceso a `refs` ni puede programar actualizaciones.y. The pains of the past are behind us. The future has yet to unfold. But the now is full of beauty simply waiting for our attention.

Hooks de React para consumir una API en Java (Spring Boot)

Este ejemplo muestra cómo usar `useState` para manejar estado y `useEffect` para efectos secundarios, en este caso un *fetch* a una API de Java al montar el componente. Resulta útil si el *frontend* está en ReactJS y el *backend* en Java (Spring Boot).

Pasos iniciales

1. Crear el proyecto React

```
npx create-react-app frontend
cd frontend
Crear endpoint en Java (Spring Boot)
@RestController
@RequestMapping("/api")
public class HelloController {
  @GetMapping("/hello")
  public Map<String, String> hello() {
    return Map.of("message", "Hola desde Java!");
  }
}
```

2. Habilitar CORS (para permitir peticiones desde el puerto 3000):

```
@Bean
public WebMvcConfigurer corsConfigurer() {
  return new WebMvcConfigurer() {
    @Override
    public void addCorsMappings(CorsRegistry registry) {
      registry.addMapping("/**").allowedOrigins("http://localhost:3000");
    }
  };
```

```
}
```

3. Usar hooks en React para consumir la API

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

function App() {
 const [message, setMessage] = useState("");

 useEffect(() => {
   fetch("http://localhost:8080/api/hello")
     .then(res => res.json())
     .then(data => setMessage(data.message))
     .catch(console.error);
 }, []); // se ejecuta una vez al montar el componente

 return <h1>{message || "Cargando..."}</h1>;
}

export default App;
```

CONTINUAR

Referencias

Álvarez Caules, C. (2016). ¿Qué es el Virtual DOM y como funciona? *Arquitectura Java*. https://www.arquitecturajava.com/que-es-el-virtual-dom-y-como-funciona/

Referencias bibliográficas de consulta

Deitel, H. y Deitel, P. (2013). *Cómo programar en Java*. Pearson

Joyanes Aguilar, L. (2014). *Fundamentos de programación*. McGraw-Hill

Moreno, P. J. (2015). *Programación orientada a objetos*. RA-MA Editorial

Espark Biz, (s.f.). *Frontend de React con backend de Java: una combinación ganadora para aplicaciones escalables.*

https://www.esparkinfo.com/software-development/technologies/reactjs/react-with-java

Legacy, (s.f.). *Referencia de la API de los hooks*. https://es.legacy.reactjs.org/docs/hooks-reference.html

CONTINUAR