1. Introducción REACT

React es una biblioteca de código abierto de JavaScript para crear interfaces de usuario de front-end. A diferencia de otras bibliotecas de JavaScript que proporcionan un marco de trabajo de la aplicación completo, React se centra únicamente en la creación de vistas de aplicación mediante unidades encapsuladas llamadas componentes que mantienen el estado y generan los elementos de la interfaz de usuario. Puede colocar un componente individual en una página web o anidar jerarquías de componentes para crear una interfaz de usuario compleja.

Normalmente, los componentes de React se escriben en JavaScript y JSX (JavaScript XML), que es una extensión de JavaScript que se parece mucho a HTML, pero tiene algunas características de sintaxis que facilitan la realización de tareas comunes, como el registro de controladores de eventos para los elementos de la interfaz de usuario. Un componente de React implementa el método render, que devuelve el código JSX que representa la interfaz de usuario del componente. En una aplicación web, el código JSX devuelto por el componente se traduce en HTML compatible con el explorador y representado en el explorador.

2. Aplicaciones REACT

- Aplicaciones web básicas: sitios web con interacción del usuario
- Aplicaciones de página única (SPA): son sitios web que interactúan con el usuario reescribiendo dinámicamente la página web actual con nuevos datos de un servidor, en lugar del funcionamiento predeterminado del explorador de cargar páginas nuevas enteras.
- Aplicaciones de escritorio nativas: React Native permite crear aplicaciones de escritorio nativas con JavaScript que se ejecutan en varios tipos de equipos de escritorio, portátiles, tabletas, Xbox y dispositivos de realidad mixta.
- Aplicaciones móviles nativas: React Native es una manera multiplataforma de crear aplicaciones Android e iOS con JavaScript que se representan en código de interfaz de usuario de la plataforma nativa.

3. Herramientas REACT

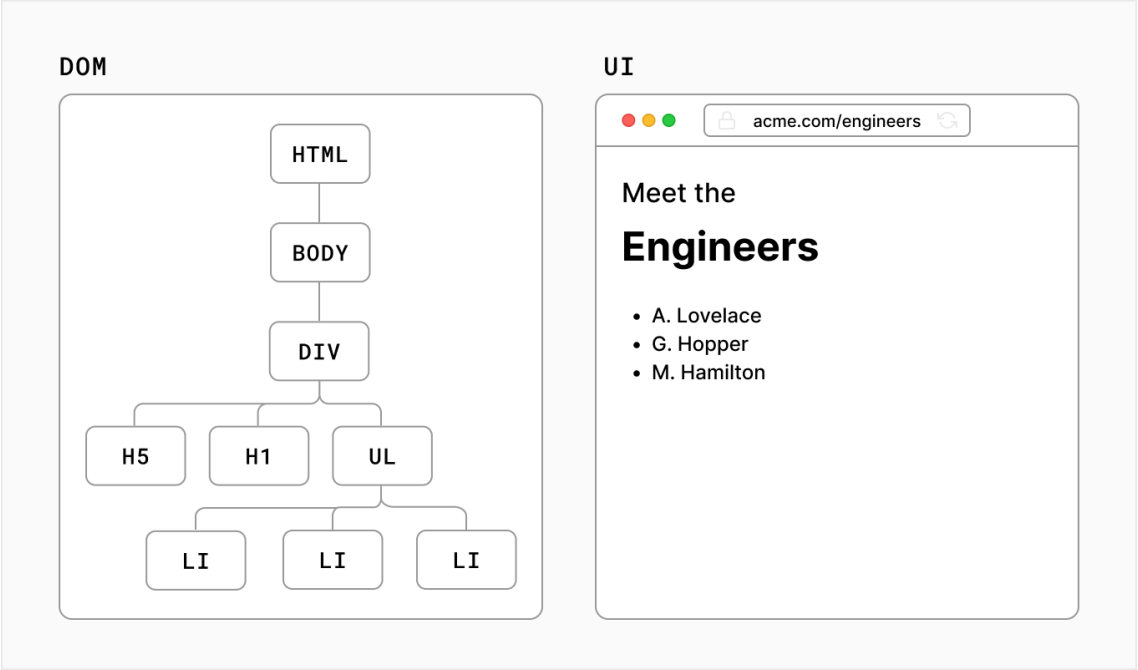
Aunque escribir un componente de React simple en un editor de texto sin formato es una buena introducción a React, el código generado de esta manera es masivo, difícil de mantener e implementar, y lento. Hay algunas tareas comunes que las aplicaciones de producción deberán realizar. Estas tareas se controlan mediante otros marcos de JavaScript que la aplicación toma como una dependencia. Entre las tareas, se incluyen las siguientes:

- Compilación: JSX es el lenguaje que se usa normalmente para las aplicaciones de React, pero los exploradores no pueden procesar esta sintaxis directamente. En su lugar, se debe compilar el código en la sintaxis estándar de JavaScript y se debe personalizar para distintos exploradores. Babel es un ejemplo de compilador que admite JSX.
- Empaquetado: dado que el rendimiento es fundamental para las aplicaciones web modernas, es importante que el código JavaScript de una aplicación incluya solo el código necesario para la aplicación y se combine en el menor número posible de archivos. Un empaquetador, como webpack, realiza esta tarea de forma automática.
- Administración de paquetes: los administradores de paquetes facilitan la inclusión de la funcionalidad de los paquetes de terceros en la aplicación, incluida su actualización y la administración de dependencias. Entre los administradores de paquetes más usados se encuentran Yarn y npm.

Juntos, el conjunto de marcos que le ayudan a crear, compilar e implementar la aplicación se llama cadena de herramientas. Una cadena de herramientas sencilla con la que empezar a trabajar es create-react-app, que genera automáticamente una aplicación de página única sencilla. La única configuración necesaria para usar create-react-app es Node.js.

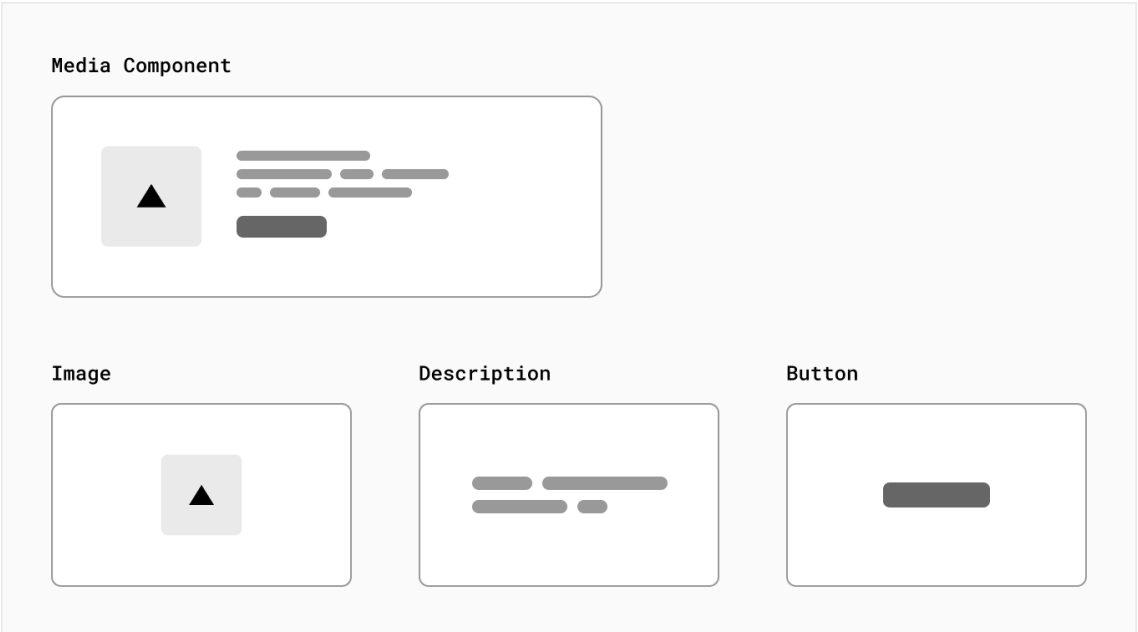
4. DOM virtual

DOM significa Document Object Model y representa la interfaz de usuario de la aplicación. Cada vez que cambia el estado de la interfaz de usuario de la aplicación, se actualiza el DOM para representar el cambio. Cuando el DOM se actualiza con frecuencia, el rendimiento se vuelve lento. Un DOM virtual es solo una representación visual del DOM, por lo que cuando cambia el estado de la aplicación, se actualiza el DOM virtual en lugar del DOM real, lo que reduce el costo de rendimiento. Es una representación de un objeto DOM, como una copia ligera.



5. Componentes

Los componentes son uno de los conceptos esenciales de React. Constituyen los cimientos sobre los que construyes interfaces de usuario.



En la Web, HTML nos permite crear documentos estructurados con su conjunto integrado de etiquetas como <h1> y <li>:

```
<article>
  <h1>My First Component</h1>
  <ol>
    <li>Components: UI Building Blocks</li>
    <li>Defining a Component</li>
    <li>Using a Component</li>
  </ol>
</article>
```

Este HTML representa un artículo <article>, su encabezado <h1>, y una tabla de contenidos (abreviada) representada como una lista ordenada <ol>. Un HTML como este, combinado con CSS para los estilos y JavaScript para la interactividad, están detrás de cada barra lateral, avatar, modal, menú desplegable y cualquier otra pieza de UI que ves en la web.

React te permite combinar tu HTML, CSS y JavaScript en «componentes» personalizados, elementos reutilizables de UI para tu aplicación. El código de la tabla de contenidos que viste arriba pudo haberse transformado en un componente <TableOfContents /> que podrías renderizar en cada página. Por detrás, seguiría utilizando las mismas etiquetas HTML como <article>, <h1>, etc.

De la misma forma que con las etiquetas HTML, puedes componer, ordenar y anidar componentes para diseñar páginas completas.

En la medida en que un proyecto crece, notarás que muchos de los diseños se pueden componer mediante la reutilización de componentes que ya escribiste, acelerando el desarrollo. La tabla de contenido de arriba podría añadirse a cualquier pantalla con <TableOfContents />.

5.1. Definir un componente

Tradicionalmente, como viste en el curso anterior de Introducción al desarrollo, cuando se creaban páginas web, los desarrolladores usaban lenguaje de HTML para describir el contenido y luego añadían interacciones agregando un poco de JavaScript. Esto funcionaba perfectamente cuando las interacciones eran algo deseable, pero no imprescindible en la web. Ahora es algo que se espera de muchos sitios y de todas las aplicaciones. React pone la interactividad primero usando aún la misma tecnología: un

componente de React es una función de JavaScript a la que puedes agregar HTML. Ejemplo:

```
export default function Profile() {
  return (
    <img
      src="https://i.imgur.com/MK3eW3Am.jpg"
      alt="Katherine Johnson"
    />
  )
}
```

1. Exporta el componente: El prefijo export default es parte de la sintaxis estándar de Javascript (no es específico de React). Te permite marcar la función principal en un archivo para que luego puedas importarlas en otros archivos.
2. Define la función: Con function Profile() { } defines una función con el nombre Profile. Los componentes de React son funciones regulares de JavaScript, pero sus nombres deben comenzar con letra mayúscula o no funcionarán.
3. Añadir marcado
   El componente retorna una etiqueta <img /> con atributos src y alt. <img /> se escribe como en HTML, pero en realidad es JavaScript por detrás. Esta sintaxis se llama JSX, y te permite incorporar marcado dentro de JavaScript.

   Las sentencias return se pueden escribir todo en una línea, como en este componente:

   ```
   return <img src="https://i.imgur.com/MK3eW3As.jpg" alt="Katherine Johnson" />;
   ```

   Pero si tu marcado no está todo en la misma línea que la palabra clave return, debes ponerlo dentro de paréntesis como en este ejemplo:

   ```
   return (
     <div>
       <img src="https://i.imgur.com/MK3eW3As.jpg" alt="Katherine Johnson" />
     </div>
   ```

```
);
```

Sin paréntesis, todo el código que está en las líneas posteriores al return serán ignoradas

5.2. Usar un componente

Ahora que has definido tu componente Profile, puedes anidarlo dentro de otros componentes. Por ejemplo, puedes exportar un componente Gallery que utilice múltiples componentes Profile:

```
function Profile() {
  return (
    <img
      src="https://i.imgur.com/MK3eW3As.jpg"
      alt="Katherine Johnson"
    />
  );
}

export default function Gallery() {
  return (
    <section>
      <h1>Amazing scientists</h1>
      <Profile />
      <Profile />
      <Profile />
    </section>
  );
}
```

Nota la diferencia de mayúsculas y minúsculas:

- <section> está en minúsculas, por lo que React sabe que nos referimos a una etiqueta HTML.
- <Profile /> comienza con una P mayúscula, por lo que React sabe que queremos usar nuestro componente llamado Profile.

Y Profile contiene aún más HTML: <img />. Al final lo que el navegador ve es esto:

```
<section>
  <h1>Amazing scientists</h1>
```

```
  <img            src="https://i.imgur.com/MK3eW3As.jpg"
alt="Katherine Johnson" />
  <img            src="https://i.imgur.com/MK3eW3As.jpg"
alt="Katherine Johnson" />
  <img            src="https://i.imgur.com/MK3eW3As.jpg"
alt="Katherine Johnson" />
</section>
```

5.3. Anidar y organizar componentes

Los componentes son funciones regulares de JavaScript, por lo que puedes tener múltiples componentes en el mismo archivo. Esto es conveniente cuando los componentes son relativamente pequeños o están estrechamente relacionados entre sí. Si este archivo se torna abarrotado, siempre puedes mover Profile a un archivo separado.

Dado que los componentes Profile se renderizan dentro de Gallery, incluso varias veces, podemos decir que Gallery es un componente padre, que renderiza cada Profile como un «hijo». Este es la parte mágica de React: puedes definir un componente una vez, y luego usarlo en muchos lugares y tantas veces como quieras.

Los componentes pueden renderizar otros componentes, pero nunca debes anidar sus definiciones:

```
export default function Gallery() {
  // ⚫ ¡Nunca defines un componente dentro de otro
componente!
  function Profile() {
    // ...
  }
  // ...
}
```

El fragmento de código de arriba es muy lento y causa errores. En su lugar, define cada componente en el primer nivel:

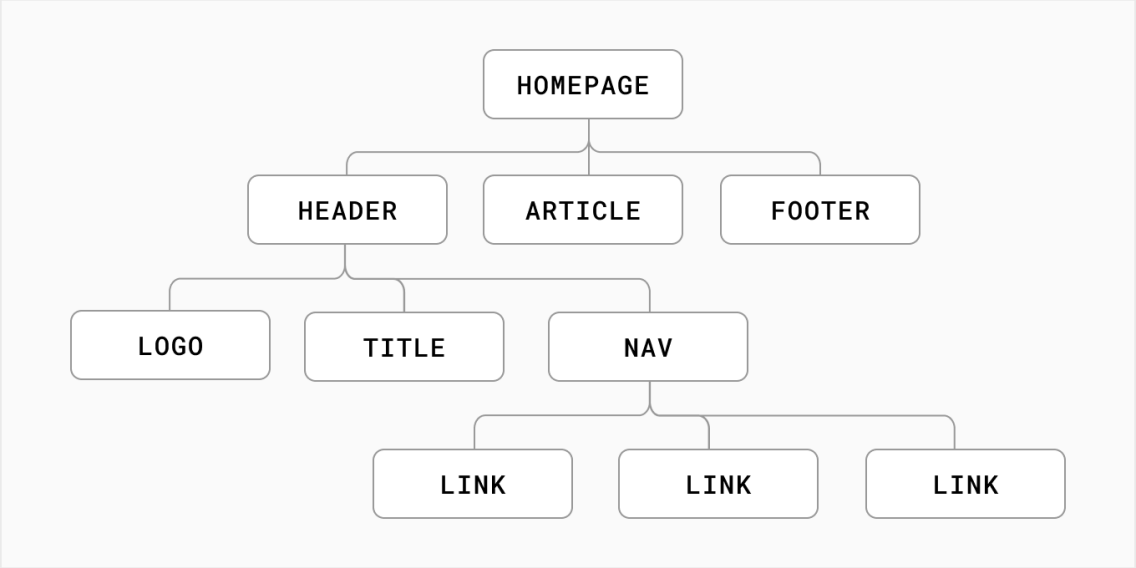
```
export default function Gallery() {
  // ...
}

// ✅ Declara los componentes en el primer nivel
function Profile() {
  // ...
```

}

Cuando un componente hijo necesita datos de su padre, se utilizan props que veremos más adelante, en lugar de anidar las definiciones.

Se pueden formar "arboles de componentes". Por ejemplo, su componente de página de inicio de nivel superior podría contener un encabezado, un artículo y un componente de pie de página. Y cada uno de esos componentes podría tener a su vez sus propios componentes secundarios y así sucesivamente. Por ejemplo, el componente de encabezado podría contener un componente de logotipo, título y navegación.

Este formato modular le permite reutilizar componentes en diferentes lugares dentro de su aplicación.



5.4. Importar y exportar componentes

La magia de los componentes reside en su reusabilidad: puedes crear componentes que se componen a su vez de otros componentes. Pero mientras anidas más y más componentes, a menudo tiene sentido comenzar a separarlos en diferentes archivos. Esto permite que tus archivos se mantengan fáciles de localizar y puedas reutilizar componentes en más lugares.

5.4.1. El archivo de componente raíz

En Tu primer componente, hiciste un componente Profile y un componente Gallery que lo renderiza:

```
function Profile() {
  return (
    <img
      src="https://i.imgur.com/MK3eW3As.jpg"
      alt="Katherine Johnson"
    />
  );
}

export default function Gallery() {
  return (
    <section>
      <h1>Amazing scientists</h1>
      <Profile />
      <Profile />
      <Profile />
    </section>
  );
}
```

Estos componentes están guardados en un archivo de componente raíz, llamadoApp.js. No obstante, en dependencia de tu configuración, tu componente raíz podría estar en otro archivo. Si utilizas un framework con enrutamiento basado en archivos, como Next.js, tu componente raíz será diferente para cada página.

5.4.2. Exportar e importar un componente

¿Y si quisieras cambiar la pantalla de inicio en el futuro y poner allí una lista de libros científicos? ¿O ubicar todos los perfiles en otro lugar? Tiene sentido mover Gallery y Profile fuera del componente raíz. Esto los haría más modulares y reutilizables en otros archivos. Puedes mover un componente en tres pasos:

1. Creá un nuevo archivo JS para poner los componentes dentro.
2. Exportá tu componente de función desde ese archivo (ya sea usando exports por defecto o con nombre).
3. Importalo en el archivo en el que usarás el componente (usando la técnica correspondiente de importar exports por defecto o con nombre).

Aquí tanto Profile y Gallery se han movido fuera de App.js en un nuevo archivo llamado Gallery.js. Ahora puedes cambiar App.js para importar Gallery desde Gallery.js:

```
import Gallery from './Gallery.js';

export default function App() {
  return (
    <Gallery />
  );
}
```

En este ejemplo está descompuesto en dos archivos:

Gallery.js:

- Define el componente Profile que se usa solo dentro del mismo archivo y no se exporta.
- Define el componente Gallery como un export por defecto.

App.js:

- Importa Gallery como un import por defecto desde Gallery.js.
- Exporta el componente raíz App como un export por defecto.

Puede que te encuentres archivos que omiten la extensión de archivo .js de esta forma:

```
import Gallery from './Gallery';
```

Tanto './Gallery.js' como './Gallery' funcionarán con React, aunque la primera forma es más cercana a cómo lo hacen los módulos nativos de ES.

5.4.3. Exportar e importar múltiples componentes del mismo archivo

¿Y si quisieras mostrar solo un Profile en lugar de toda la galería? Puedes exportar el componente Profile también. Pero Gallery.js ya tiene un export por defecto, y no puedes tener dos exports por defecto. Podrías crear un nuevo archivo con un export por defecto, o podrías añadir un export con nombre para Profile. Un archivo solo puede contener un export por defecto, pero puede tener múltiples exports con nombre

Para reducir la potencial confusión entre exports por defecto y con nombre, algunos equipos escogen utilizar solo un estilo (por defecto o con nombre), o evitan mezclarlos en un mismo archivo. Es una cuestión de preferencias.

Primero, exportá Profile desde Gallery.js usando un export con nombre (sin la palabra clave default):

```
export function Profile() {
  // ...
}
```

Luego, importá Profile desde Gallery.js hacia App.js usando un export con nombre (con llaves):

```
import { Profile } from './Gallery.js';
```

Por último, renderizá <Profile /> en el componente App:

```
export default function App() {
    return <Profile />;
}
```

Ahora Gallery.js contiene dos exports: un export por defecto Gallery, y un export con nombre Profile. App.js importa ambos. Intenta editar <Profile /> cambiándolo a <Gallery /> y viceversa en este ejemplo:

App.js

```
import Gallery from './Gallery.js';
import { Profile } from './Gallery.js';

export default function App() {
  return (
    <Profile />
  );
}
```

Gallery.js

```
export function Profile() {
```

```
  return (
    <img
      src="https://i.imgur.com/QIrZWGIs.jpg"
      alt="Alan L. Hart"
    />
  );
}

export default function Gallery() {
  return (
    <section>
      <h1>Amazing scientists</h1>
      <Profile />
      <Profile />
      <Profile />
    </section>
  );
}
```

Ahora estás usando a una mezcla de exports por defecto y con nombre:

Gallery.js:

- Exporta el componente Profile como un export con nombre llamado Profile.
- Exporta el componente Gallery como un export por defecto.

App.js:

- Importa Profile como un import con nombre llamado Profile desde Gallery.js.
- Importa Gallery como un import por defecto desde Gallery.js.
- Exporta el componente raíz App como un export por defecto.

6. PROPS

En React, "prop" (abreviatura de "propiedad") se refiere a un mecanismo que permite pasar datos de un componente principal a un componente secundario como si fueran argumentos de una función.

Cuando se crea un componente en React, se pueden definir props en la declaración del componente. Estas props son variables que se pueden utilizar en el componente y que pueden ser personalizadas cada vez que se instancia el componente. Por ejemplo, si se tiene un componente de botón, se podrían definir las props "text" (para el texto del botón) y "onClick" (para la función que se ejecutará cuando se haga clic en el botón).

Una vez que se han definido las props, se pueden pasar como argumentos al componente secundario a través de la sintaxis de atributos HTML. El componente secundario puede acceder a las props pasadas mediante el objeto "props" y utilizarlas como cualquier otra variable.

El uso de props es una de las características principales de React que lo hace muy útil para construir aplicaciones modulares y reutilizables. Al permitir la personalización de los componentes secundarios a través de las props, se puede crear una jerarquía de componentes que se ajuste a cualquier necesidad específica de la aplicación.

6.1. Pasando una cadena de texto como prop:

En este ejemplo, el componente de encabezado recibe una prop llamada "title" que se utiliza para mostrar el título del sitio web. Al crear una instancia del componente, se le pasa el valor "Mi sitio web" como prop.

```
// Componente de encabezado
function Header(props) {
  return (
    <h1>{props.title}</h1>
  );
}
// Uso del componente
<Header title="Mi sitio web" />
```

6.2. Pasando una función como prop:

En este ejemplo, el componente de botón recibe dos props: "text" y "onClick". "text" se utiliza para mostrar el texto en el botón y "onClick" es una función que se ejecutará cuando se haga clic en el botón. Al crear una instancia del componente, se le pasa una función que imprime un mensaje en la consola.

```
// Componente de botón
function Button(props) {
  return (
    <button onClick={props.onClick}>
      {props.text}
    </button>
  );
}
// Uso del componente
<Button
  text="Haz clic aquí"
  onClick={() => console.log("Botón presionado")}
/>
```

6.3. Pasando una lista como prop:

En este ejemplo, el componente de lista recibe una prop llamada "items" que es una lista de objetos con una propiedad "id" y "text". El componente itera sobre la lista y crea elementos de lista con el texto de cada objeto. Al crear una instancia del componente, se le pasa la lista "items" como prop.

```
// Componente de lista
function List(props) {
  return (
    <ul>
      {props.items.map(item => (
        <li key={item.id}>{item.text}</li>
      ))}
```

```
    </ul>
  );
}

// Uso del componente
const items = [
  { id: 1, text: "Item 1" },
  { id: 2, text: "Item 2" },
  { id: 3, text: "Item 3" }
];
<List items={items} />
```

Estos son solo algunos ejemplos básicos de uso de props en React. Las props pueden contener cualquier tipo de dato que se pueda pasar como argumento en JavaScript, lo que las hace muy versátiles y útiles para personalizar y reutilizar componentes en una aplicación de React.

7. ESTADO Y HOOKS

El estado en React se refiere a los datos que cambian en la aplicación en tiempo de ejecución. Por ejemplo, un contador que se actualiza cada vez que se hace clic en un botón o un formulario que cambia su estado cada vez que se ingresa un valor en un campo.

En React, el estado se almacena en una variable llamada "state". Cada vez que se actualiza el estado, React vuelve a renderizar el componente correspondiente para reflejar el nuevo estado en la interfaz de usuario.

Para manipular el estado, React proporciona una función llamada "setState". Esta función se utiliza para actualizar el valor del estado y activar una nueva renderización del componente.

Los Hooks de React son funciones especiales que permiten a los componentes de React tener estado y otras características sin necesidad de utilizar clases de Javascript. Con los Hooks, los componentes de React pueden usar funciones como state, efectos, context y otros.

7.1. useState

El Hook useState es utilizado para agregar un estado a los componentes de React. Permite que un componente tenga su propio estado local, el cual se puede modificar a lo largo del ciclo de vida del componente.

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


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


  return (
    <div>
      <p>Contador: {count}</p>
      <button onClick={() => setCount(count + 1)}>Incrementar</button>
    </div>
  );
}
```

En este ejemplo, el Hook useState se utiliza para agregar un estado llamado "count" al componente "Counter". El estado se inicializa con un valor de 0 y se actualiza cuando se hace clic en el botón "Incrementar".

7.2. useEffect

El Hook useEffect es utilizado para realizar efectos secundarios en componentes de React. Por ejemplo, se puede utilizar para hacer una llamada a una API o para actualizar el título de la página cuando se cambia el estado.

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


function TitleUpdater() {
  const [title, setTitle] = useState('');


  useEffect(() => {
```

```
    document.title = title;
  }, [title]);

  return (
    <div>
      <input
        type="text"
        value={title}
        onChange={(e) => setTitle(e.target.value)}
      />
    </div>
  );
}
```

En este ejemplo, el Hook useEffect se utiliza para actualizar el título de la página cada vez que el estado "title" cambia. El efecto se define dentro de una función que se ejecuta cuando se monta el componente y se limpia cuando se desmonta.

7.3. useContext

El Hook useContext se utiliza para acceder al contexto en los componentes de React. El contexto es una forma de compartir datos entre componentes sin necesidad de pasar props manualmente a través de cada nivel de la jerarquía de componentes.

En este ejemplo, se define un contexto llamado "MyContext" con un valor predeterminado de "default". Luego, se crea un componente llamado "ComponentOne" que accede al valor del contexto utilizando el Hook useContext. Finalmente, se crea un componente llamado "ComponentTwo" que proporciona un valor de contexto de "Hola Mundo" a través del proveedor MyContext.Provider. Cuando se renderiza el componente "ComponentOne", accede al valor del contexto que se proporcionó desde el componente "ComponentTwo".

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

```
const MyContext = React.createContext('default');

function ComponentOne() {
  const value = useContext(MyContext);

  return <p>{value}</p>;
}

function ComponentTwo() {
  return (
    <MyContext.Provider value="Hola Mundo">
      <ComponentOne />
    </MyContext.Provider>
  );
}
```