React
Guia de atajos

{dev/talles}

Commandos Básicos

Crear un nuevo proyecto de cero.

```
npm create vite
```

Seguir los pasos que se muestran.

Componentes

Estructura de un componentes.

```
export function MyComponent() {
 return <h1>Hola Mundo</h1>;
}
```

Importante:
Es obligatorio que cada componente regrese siempre un **único elemento**, si se necesita retornar más de uno, se debe de envolver en un elemento padre para respetar la regla.

```
export function MyComponent() {
  return (
    <div>
      <h1>Hola Mundo</h1>
      <p>Este es un párrafo</p>
    </div>
  );
}
```

Fragmentos

Es un elemento que no es renderizado en el DOM y sirve para agrupar sin afectar la estructura.

```
import { Fragment } from 'react';

export function MyComponent() {
  return (
    <Fragment>
      <h1>Hola Mundo</h1>
      <p>Este es un párrafo</p>
    </Fragment>
  );
}
```

Pro Tip:
La forma abreviada del fragmento es esta, y no requiere importaciones.

```
export function MyComponent() {
  return (
    <>
      <h1>Hola Mundo</h1>
      <p>Este es un párrafo</p>
    </>
  );
}
```

Uso de componentes dentro de otros.

```
function MyComponent() {
  return (
    <>
      <h1>Hola Mundo</h1>
      <p>Este es un párrafo</p>
    </>
  );
}

function MyApp() {
  return (
    <>
      <MyComponent />
      <MyComponent />
      <MyComponent />
    </>
  );
}
```

Propiedades de un componente (Props)

Son valores o funciones que se envían al componente, deben de ser inmutables y permiten la personalización.

```
interface Props {
  firstName: string;
};

const Greeting = ({ firstName }: Props) => {
  return <h1>Hola, {firstName}!</h1>;
};

// Usando el componente
<Greeting firstName="Fernando" />
```

Tip:
El código anterior está en TypeScript, ya que las interfaces no existen en JavaScript, pero nos ayudan a decir cómo luce el objeto Props.

Tip 2:
Presta atención a la desestructuración de argumentos a la función y nombre de la prop al usar el componente.

Children Prop

Es una forma de mandar elementos hijos a un componente para renderizarlos en otro.

```
type BoxProps = {
  children: React.ReactNode; // 👈 Este es el
contenido que se coloca dentro del componente
};

const Box = ({ children }: BoxProps) => {
  return (
    <div style={{ border: '2px solid blue'}}>
 {/* Renderiza lo que se pase dentro del componente */}
      {children}
    </div>
  );
};

export default Box;
```

Para usar el componente “Box” creado anteriormente:

```
const App = () => {
  return (
    <div>
{/* Pasamos elementos como children dentro
del componente Box */}
      <Box>
        <h2>Hola Mundo</h2>
        <p>Este es un párrafo</p>
      </Box>
    </div>
  );
};
```

Importante:
Los comentarios dentro del JSX, son expresiones de JavaScript con su comentario interno, aquí no funciona el **<!— —>**, porque no es HTML.

Renderización condicional

En React, no hay directivas para renderizar condicionalmente o realizar ciclos, dependen enteramente de expresiones de JavaScript.

```
const isLoggedIn = true;

const LoginMessage = () => {
  return (
    <div>
   {/* Renderizado condicional con ternario */}
      {isLoggedIn ? (
        <h2>Bienvenido de nuevo! 🎉</h2>
      ) : (
        <h2>Por favor, inicie sesión. 🔒</h2>
      )}
    </div>
  );
};
```

Tip:
Hay alternativas para condicionales simples.

```
// Mostrar solo si isLoggedIn es true
{isLoggedIn && <p>Bienvenido de nuevo! 🎉</p>}

// Mostrar si no está logueado
{!isLoggedIn && <p>Por favor, inicie sesión.</p>}
```

Es posible tener retornos prematuros para mantener el código limpio.

```
type Status = 'loading' | 'error' | 'success';

const MultiReturnConditional = () => {
  const status: Status = 'loading';

  // 🔁 Render según el estado
  if (status === 'loading') {
    return <p>Cargando... ⏳</p>;
  }

  if (status === 'error') {
    return <p>Ups! Algo salió mal ❌</p>;
  }

  if (status === 'success') {
    return (
      <div>
        <h2>Datos cargados correctamente ✅</h2>
        <p>Aquí está tu contenido increíble!</p>
      </div>
    );
  }

  // Este return es obligatorio por TypeScript,
aunque nunca se debería alcanzar
  return <p>Status no reconocido</p>;
};
```

Listas de componentes

Podemos barrer listados en formato de arreglo utilizando el método **.map()** de los arreglos y retornar un JSX en la función callback.

```
const UserList = () => {
  // Lista de usuarios
  const users = [
    { id: 1, name: 'Fernando' },
    { id: 2, name: 'Melissa' },
    { id: 3, name: 'Natalia' },
  ];

  return (
    <div>
      <h2>Usuarios</h2>

      <ul>
       {users.map((user) => (
          <li key={user.id}>
{/* Siempre se recomienda usar una key
única, como el ID */}
            👤 {user.name}
          </li>
        ))}
      </ul>
    </div>
  );
};
```

Importante:
Cuando se usa el **.map()**, siempre debes de colocar el atributo key.

Hooks

Son simples funciones que se definen al inicio del functional component y permiten acceso a objetos o funciones dentro del contexto o almacenar estado en el componente.

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

export const MyCounter = () => {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Count: {count}</p>
    </div>
  );
};
```

Listado de los hooks

Un listado de los hooks propios de React

Básicos

Hooks que encontrarás en toda aplicación de React.

Comando	Descripción
useState	Maneja un estado local en el componente.
useEffect	Ejecuta efectos secundarios y limpieza al desmontar el componente.
useContext	Accede al valor alojado en el contexto. (árbol de componentes)

Adicionales

Hooks que ofrecen comportamientos adicionales.

Comando	Descripción
useReducer	Alternativa al useState para lógica compleja.
useRef	Referencias mutables que no causan re-render.
useMemo	Memoriza valores para evitar volverlos a calcular entre re-renders.
useCallback	Memoriza funciones para evitar recreaciones innecesarias.
useLayoutEffect	Similar al useEffect, pero sincronizado justo después del render.
useImperativeHandle	Expone métodos desde un componente con forwardRef

Relacionados al DOM

Hooks orientados a información del DOM

Comando	Descripción
useDeferredValue	Difiere valores para mejorar el rendimiento entre re-renders.
useTransition	Permite renderizar partes del UI en el background, pueden ser vistas como actualizaciones no urgentes.
useInsertionEffect	Se ejecuta antes del render para estilos dinámicos.

Últimos Hooks

Comando	Descripción
useFormStatus	Lee el estado de un <form>, el último posteo.
useActionState	Actualiza el estado basado en el resultado de un posteo de formulario.
useOptimistic	Muestra valores optimistas antes de que una acción sea resuelta.

Importante:
Existen otros hooks para casos específicos, pueden encontrar el listado en React Docs.

Reglas de los Hooks

Estas reglas están dadas por React y se deben de respetar si quieres que tu aplicación funciones como esperas.

Regla	Explicación
Siempre llama a los hooks en el **mismo orden**.	No pongas hooks dentro de condiciones (if), bucles (for, while) o funciones internas.
Solo usa hooks en **componentes funcionales** o custom hooks.	No puedes usar hooks dentro de funciones normales, clases, condicionales globales.
Los hooks deben comenzar con **use**.	Si creas un custom hook, asegúrate de llamarlo useAlgo, por ejemplo: useFetch, useAuth, etc.
Evita efectos secundarios innecesarios **en useEffect.**	No pongas funciones que modifiquen estado sin incluirlas en las dependencias, o se puede crear un bucle infinito.
No modifiques el estado directamente.	Siempre usa setState (como setCount(newValue)) en lugar de modificar el estado actual directamente.
Separa lógica en **hooks personalizados** si se repite.	Si tienes lógica repetida (por ejemplo, un useEffect que hace fetch), considera extraerla en un custom hook para reutilizarla.
Efectos del useEffect **deben de ser atómicos**.	No crees un efecto gigante, sepáralo en efectos pequeños con responsabilidades bien definidas.

Custom Hooks - Hooks personalizados

Es posible separar lógica en hooks personalizados para reutilizarla o bien crear componentes más limpios.

```
// Nombre del archivo: useMessage.ts
import { useState } from 'react';

// Hook que maneja un mensaje y permite
actualizarlo
const useMessage = () => {

  const [message, setMessage] =
useState<string>('Hola Mundo!');

  // Método para cambiar el mensaje
  const updateMessage = (newMessage: string) => {
    setMessage(newMessage);
  };

  return {
    message,
    updateMessage,
  };
};

export default useMessage;
```

Consumo del hook personalizado **useMessage**

```
const MessageComponent = () => {
  const { message, updateMessage } = useMessage();

  return (
    <div>
      <h2>{message}</h2>
      <button onClick={() =>
updateMessage('Saludos a todos! 🚀')}>
        Change message
      </button>
    </div>
  );
};
```

React API - use

"use" no es un hook

Permite leer un valor de un recurso como una promesa o contexto, suspendiendo la creación hasta tener una resolución. Va de la mano del componente **Suspense**.

```
const MyComponent = () => {

  const message = use(wait(2000)); // Suspende el
componente 2s, luego retorna el mensaje

  return <h2>{message}</h2>;
};
```

Se debe de colocar el **<Suspense> component**

```
import { Suspense } from 'react';
import MyComponent from './MyComponent';

export default function App() {
  return (
    <Suspense fallback={<p>Loading...</p>}>
      <MyComponent />
    </Suspense>
  );
```

Componente <Suspense>

Permite desplegar un contenido hasta que sus hijos terminen de cargar.

```
<Suspense fallback={<Loading />}>
  <Albums />
</Suspense>
```

Importante:
Si hay un único suspense, el contenido se mostrará hasta que todos los hijos resuelvan.

Si se desea una aproximación más granular, donde los hijos se muestran tan pronto terminan, se deben de colocar **múltiples suspense**.

```
<Suspense fallback={<BigSpinner />}>
  <Biography />
  <Suspense fallback={<AlbumsGlimmer />}>
    <Panel>
      <Albums />
    </Panel>
  </Suspense>
</Suspense>
```

React compiler

El compilador de React permite optimizaciones de código sin necesidad de memorización manual por parte del desarrollador.

Manejo de eventos

Los eventos de los elementos son los mismos que ya estamos acostumbrados a usar en el DOM tradicional, pero los nombres son con **camelCase**.

```
import { ChangeEvent, FC, MouseEvent, useState } from
'react';

export const EventExample: FC = () => {
  const [nombre, setNombre] = useState<string>('');

  const handleInputChange = (event:
ChangeEvent<HTMLInputElement>): void => {
    setNombre(event.target.value);
  };

  const handleClick = (event:
MouseEvent<HTMLButtonElement>): void => {
    console.log(event);
    alert(`Hola, ${nombre.trim() !== '' ? nombre :
'invitado'}!`);
  };

  return (
    <>
      <h2>👋 Bienvenido</h2>

      <input
        type="text"
        placeholder="Escribe tu nombre"
        value={nombre}
        onChange={handleInputChange}
      />

      <button onClick={handleClick}>Saludar</button>
    </>
  );
};
```

Formularios

React no cuenta con componentes propios o especiales para el manejo de formularios.

```
import { ChangeEvent, FC, FormEvent, useState } from
'react';

export const ContactForm: FC = () => {
  // Estados tipados
  const [nombre, setNombre] = useState<string>('');
  const [email, setEmail] = useState<string>('');
  const [mensaje, setMensaje] = useState<string>('');

  // Manejadores de cambio por input
  const handleNombreChange = (e:
ChangeEvent<HTMLInputElement>): void => {
    setNombre(e.target.value);
  };

  const handleEmailChange = (e:
ChangeEvent<HTMLInputElement>): void => {
    setEmail(e.target.value);
  };

  const handleMensajeChange = (e:
ChangeEvent<HTMLTextAreaElement>): void => {
    setMensaje(e.target.value);
  };

  // Manejador del envío del formulario
  const handleSubmit = (e: FormEvent<HTMLFormElement>):
void => {
    e.preventDefault();

    console.log('📬 Enviando formulario:', {
      nombre,
      email,
      mensaje,
    });

    alert(`Gracias por tu mensaje, ${nombre}!`);
    // Resetear el formulario
    setNombre('');
    setEmail('');
    setMensaje('');
  };

  return (
    <form onSubmit={handleSubmit}>
      <h2>📩 Contáctanos</h2>

      <input
        type="text"
        placeholder="Tu nombre"
        value={nombre}
        onChange={handleNombreChange}
      />

      <input
        type="email"
        placeholder="Tu correo"
        value={email}
        onChange={handleEmailChange}
      />

      <textarea
        placeholder="Escribe tu mensaje"
        value={mensaje}
        onChange={handleMensajeChange}
      />

      <button type="submit">Enviar</button>
    </form>
  );
};
```

Importante:
Hay muchas formas de trabajar estos cambios, React tiene cientos de paquetes para el manejo de formularios con diferentes aproximaciones.

Paquetes de React recomendados

- React Hook Form
- Formik
- React Hot Toast
- TanStack Query
- Shadcn/ui
- React Bits
- Zustand
- React Router
- TanStack Router
- Axios

Estilos y CSS

Inline Styles

```
import { CSSProperties } from 'react';

export const InlineBox = () => {
  const boxStyle: CSSProperties = {
    backgroundColor: 'tomato',
    borderRadius: '8px',
  };

  return <div style={boxStyle}>Caja con
estilo en línea</div>;
};
```

CSS Tradicional

Usar la palabra **“className”** en lugar de class para referirse a clases de CSS

```
import './Box.css';

export const CssBox: React.FC = () => {
  return <div className="box">Caja con clase
CSS</div>;
};
```

CSS Modules

```
import styles from './Box.module.css';

export const CssModuleBox: React.FC = () =>
{
  return <div className={styles.box}>Caja
con CSS Module</div>;
};
```

Estilos condicionales

El estilo condicional se puede aplicar en diferentes niveles de componente.

```
interface Props {
  tipo: 'warning' | 'info';
}

export const DynamicStyle = ({ tipo }: Props) => {
  const estilo: React.CSSProperties = {
    color: tipo === 'warning' ? 'orange' : 'blue',
    padding: '0.5rem',
  };

  return <p style={estilo}>Este es un mensaje de tipo
{tipo}</p>;
};
```

Buenas prácticas

- Componentes pequeños y reutilizables.
- Nombrado en PascalCase.
- Separar lógica del render.
- Extraer handlers (handleClick, handleChange).
- No usar Hooks dentro de condiciones o loops.
- Crear hooks personalizados.

Glosario

Los siguientes son términos importantes que se deben de manejar por todo desarrollador de React.

React
Es una biblioteca de JavaScript para crear interfaces de usuario.

JSX
Es un lenguaje de marcado que permite escribir una sintaxis similar a HTML en JavaScript.

Componentes
Son piezas de código que se pueden reutilizar y que se encargan de una parte de la interfaz de usuario.

Props
Son datos que se pasan a un componente para que pueda usarlos.

Estado
Es un objeto que contiene datos que pueden cambiar con el tiempo.

Hooks
Son funciones que permiten a los componentes usar el estado y otras características de React.

Context
Es un objeto que contiene datos que pueden ser compartidos entre componentes.

Renderizado (Render)
Es el proceso de convertir el código en HTML y mostrarlo en el navegador.

Re-render
Cuando React vuelve a dibujar un componente porque cambió su estado o props.

Inmutabilidad
Es el principio de que los datos no se pueden cambiar, sino que se deben crear nuevos datos.

Virtual DOM
Representación en memoria del DOM real para hacer actualizaciones eficientes.

Event Handler
Función que se ejecuta al ocurrir un evento (como onClick, onMouseOver).

Controlled Component
Elemento de formulario cuyo valor es manejado por el estado de React.

Uncontrolled Component
Elemento de formulario que maneja su propio valor internamente, accesado con ref.

Keys
Identificadores únicos usados en listas para ayudar a React a detectar cambios.

Fragment
Permite agrupar múltiples elementos sin añadir un nodo extra (<> ... </>).

Children
Prop especial que representa los elementos anidados dentro de un componente.

Custom Hook
Función que encapsula lógica con hooks y puede reutilizarse entre componentes.

Context API
Tradicionalmente el nombre que se da al gestor de estado propio en React.

Server Component
Componente que se ejecuta en el servidor (React 18+).

Suspense
Componente que permite renderizar un fallback mientras se carga algo (como lazy).

Lazy loading
Técnica para cargar componentes o recursos solo cuando se necesitan.

Memorización
Optimización para evitar renders innecesarios (React.memo, useMemo).

Prop Drilling
Significa pasar props de un componente a otro, a través de muchos niveles intermedios.

Gestor de estado - State Manager
Es una herramienta o patrón que te ayuda a controlar y organizar los datos de la aplicación, particularmente útil para evitar el Prop drilling.