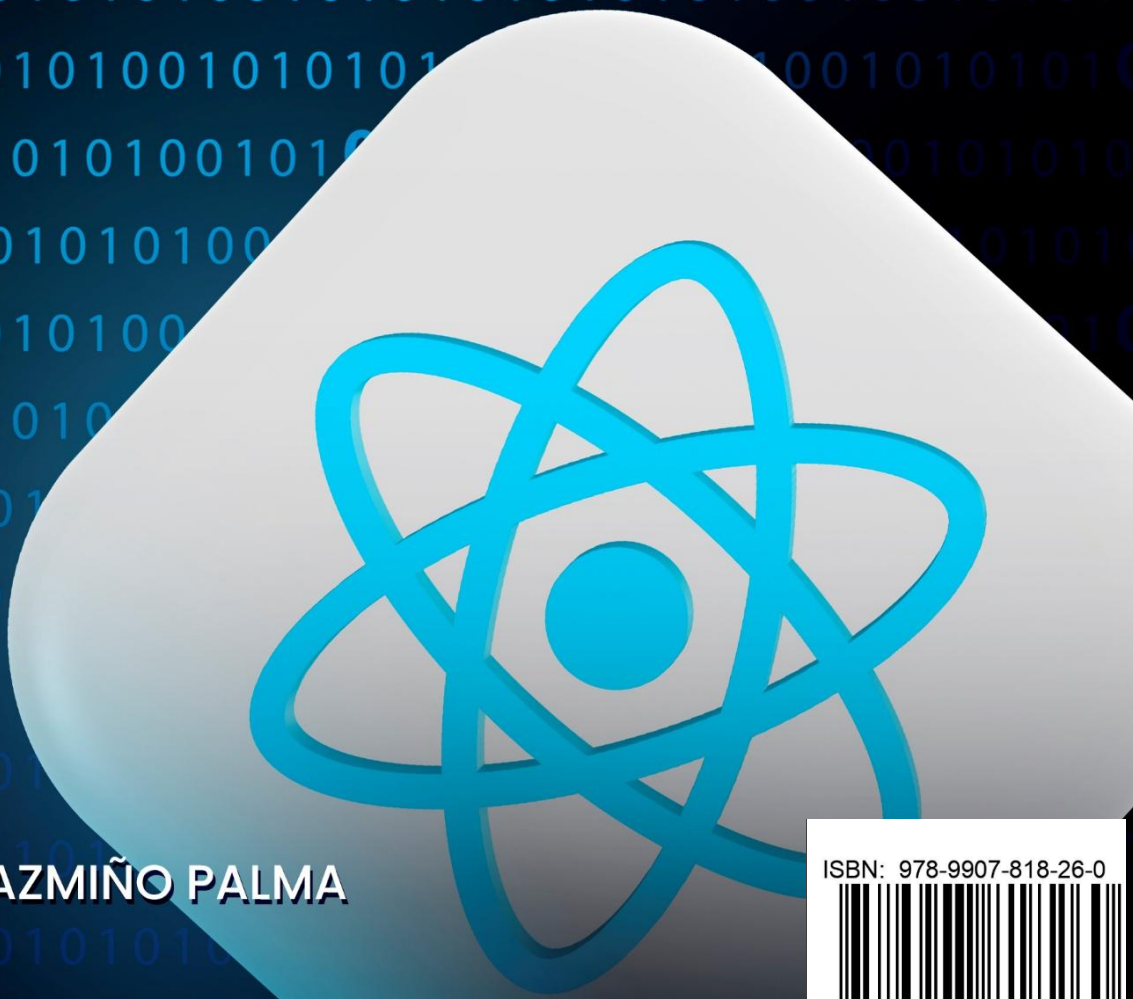
React en PALABRAS SIMPLES

DESARROLLO DE APLICACIONES



FERNANDO PAZMIÑO PALMA

ISBN: 978-9907-818-26-0

React en palabras simples

Desarrollo de Aplicaciones

Autor:

Edgar Fernando Pazmiño Palma

Orcid: https://orcid.org/0009-0005-8752-4556

Correo: edgar.pazmino@upec.edu.ec

Filial: Universidad Politécnica Estatal del Carchi

Editorial “ACACFESA SAS”

La presente obra fue revisada por 2 pares académicos externos ciegos conforme al proceso editorial de ACACFESA SAS.

Los rigurosos procedimientos editoriales de ACACFESA SAS garantizan la selección de manuscritos por sus aportes significativos al conocimiento y cualidades científicas.

Editorial: ACACFESA SAS.

Sello editorial: 978-9907-818-26

Teléfono: (+593) 998955446

Web: https://acacfesa.com/editorial/index.php/1

ISBN: 978-9907-818-26-0

Doi: https://doi.org/10.70577/hzjesj63/ACACFESA.EDITORIAL

Año: Junio 2026

Aviso Legal

El contenido de esta obra, que incluye ilustraciones, textos, tablas, gráficos, cuadros y referencias bibliográficas, es responsabilidad única del autor(a) o autores. Lo que se ha dicho, los criterios y los datos no reflejan necesariamente la posición institucional ni el pensamiento de la Editorial ACACFESA SAS.

Constancia de Arbitraje

La Editorial ACACFESA SAS, certifica que este libro es el resultado de un estudio llevado a cabo por los autores, el cual fue evaluado por jurados expertos bajo el sistema de pares externos, tanto en contenido como en forma. Asimismo, se llevó a cabo un análisis del método de investigación, paradigma y enfoque; a partir de la matriz epistémica adoptada por los autores, utilizando las normas APA, Séptima Edición y el proceso de antiplagio en línea turnitinedu para asegurar que la obra fuera científica.

María José Mendoza Salazar

Nombre del revisor/a 1

Novillo Merino Antonio Alejandro

Nombre del revisor/a 2

PROLOGO

En el vertiginoso ecosistema del desarrollo web contemporáneo, la capacidad de construir interfaces de usuario ágiles, modulares y altamente interactivas se ha consolidado como una competencia técnica fundamental. Dentro de este panorama, React.js se erige como la librería líder indiscutible para la arquitectura del front-end. En este contexto de innovación constante, la obra de Fernando Pazmiño Palma, React en palabras simples: Desarrollo de Aplicaciones, surge como una propuesta pedagógica excepcional, diseñada para transformar la complejidad técnica en un aprendizaje fluido y significativamente práctico.

Con una estructura didáctica impecable, el autor logra un equilibrio perfecto entre la rigurosidad conceptual y la aplicación en el mundo real. A lo largo de sus páginas, el lector es guiado de manera progresiva a través de los fundamentos de la herramienta, desmitificando desde el renderizado condicional hasta la gestión avanzada del estado global mediante tecnologías como Zustand y Redux. El verdadero elemento diferenciador de este volumen es su enfoque metodológico basado en proyectos reales; el texto no se limita a la abstracción teórica, sino que dota al estudiante de las capacidades necesarias para codificar desde una tienda en línea funcional hasta sistemas interactivos con Firebase y Drag & Drop. Pazmiño Palma fomenta así una autonomía técnica profunda y promueve el pensamiento crítico en la resolución de problemas lógicos de software.

Este libro es una contribución académica de gran valor para la formación de las nuevas cohortes de profesionales de la informática. Es una lectura obligatoria tanto para estudiantes de ingeniería como para programadores autodidactas que aspiran a dominar el diseño de aplicaciones modernas con criterios de optimización y escalabilidad. Una obra imprescindible que demuestra que la programación avanzada puede explicarse, en efecto, con palabras simples.

ÍNDICE DE CONTENIDO

ÍNDICE DE FIGURAS

Capítulo 1.
Fundamentos de React

Introducción a React.js

React. js es una de las bibliotecas más relevantes y utilizadas en el desarrollo de interfaces web modernas, gracias a su capacidad para crear aplicaciones dinámicas, escalables y eficaces. Creada por Meta Platforms, esta tecnología se centra en la construcción de interfaces de usuario interactivas mediante una arquitectura basada en componentes reutilizables, lo que facilita reducir el tiempo de desarrollo y mejora la organización del código. Su implementación ha transformado significativamente la manera en que se diseñan las aplicaciones web, haciendo más sencillo construir plataformas rápidas y más fáciles de mantener que se pueden adaptar a diversos entornos digitales.

La importancia de React. js radica en su enfoque declarativo y en el uso del Virtual DOM, un sistema que permite actualizar únicamente los elementos necesarios dentro de una interfaz, lo que aumenta el rendimiento de la aplicación. Este modelo operativo reduce el consumo innecesario de recursos y mejora la experiencia del usuario al ofrecer respuestas más fluidas e interacciones más ágiles. Además, React. js fomenta el modularidad del software, permitiendo que cada componente se desarrolle, pruebe y mantenga de manera independiente dentro de un sistema más complejo.

En el contexto actual de transformación digital, React. js se ha convertido en una herramienta clave para desarrollar aplicaciones web de alta demanda, especialmente en plataformas que necesitan procesamiento en tiempo real, navegación dinámica y compatibilidad entre diferentes sistemas. Su combinación con otras tecnologías y bibliotecas modernas permite crear robustos ecosistemas para aplicaciones empresariales, educativas, comerciales y de servicios digitales. Adicionalmente, su extensa comunidad de desarrolladores y la continua evolución tecnológica han consolidado a React. js como un estándar en el desarrollo frontend contemporáneo.

Breve historia y evolución de React.js

React fue creado por Facebook en 2011 como una alternativa para mejorar la administración de UIs en aplicaciones web muy complejas. En 2013 ocurrió su publicación, con la llegada del Virtual DOM como forma de lograr mejor rendimiento al modificar los componentes de la interfaz del navegador (Banks & Porcello, 2020).

Desde el punto de vista técnico, el Virtual DOM es una representación virtual del Document Object Model (DOM) que facilita la modificación de un UI en base a identificar cuáles son los elementos que se han modificado, con el objetivo de evitar cambios innecesarios en el DOM real. Un ejemplo claro de cómo se logra un mejor rendimiento y resultados dentro del campo del desarrollo web reactivo en sistemas que toman en cuenta un UI con información en constante actualización (Banks & Porcello, 2020).

React presentó un modelo de desarrollo de componentes reutilizables, convirtiéndose en una alternativa que se apoya del modularidad, el mantenimiento y un modelo ordenado del código. De acuerdo a estos autores, es un modelo que permite crear UIs muy complejas, utilizando componentes aislados que son capaces de llevarse a cabo en arquitecturas flexibles y escalables (Stefanov, 2016).

A través de los años, React ha experimentado diversas mejoras, manteniendo su énfasis en el modularidad y la reutilización de sus componentes. Este enfoque ha adquirido gran aceptación, consolidándose como una herramienta fundamental en el desarrollo de páginas web.

Impacto en el Desarrollo Web Moderno

La manera en la que React facilita la reutilización de sus componentes y mejora las actualizaciones de las interfaces de usuario mediante el uso del Virtual DOM ha producido un cambio notable en el desarrollo web moderno. Su estructura permite que React maneje de forma más efectiva los cambios que se reflejan en las interfaces, lo que disminuye significativamente la cantidad de interacciones que se llevan a cabo en el DOM real. Esto contribuye a que las aplicaciones sean más fluidas y responsivas (Banks & Porcello, 2020)

Desde una perspectiva técnica y metodológica, React ha fomentado un modelo para el desarrollo de aplicaciones web centrado en el modularidad y la reutilización de sus componentes, aspectos esenciales para crear aplicaciones web actuales. En donde el modelo también apoya la escalabilidad de ciertos proyectos, facilita el mantenimiento del código y promueve el trabajo en equipo gracias a estructuras organizativas que se pueden reutilizar (Gómez, 2024).

La relación de React con el desarrollo web actual se evidencia en la creciente demanda de aplicaciones que sean interactivas y que puedan gestionar grandes cantidades de datos en tiempo real. Según Banks y Porcello, (2020), indica que el enfoque en componentes y el

renderizado eficiente han dado origen a nuevas tácticas en el desarrollo frontend que enfatizan el rendimiento, la satisfacción del usuario y el desarrollo en múltiples plataformas.

De ahí se puede sostener que React no representa únicamente una herramienta tecnológica para la creación de interfaces, sino que implica un giro conceptual dentro del desarrollo moderno web, dado que introdujo arquitecturas más flexibles, interactivas y orientadas a la creación de sistemas digitales escalables.

Conceptos como la reutilización de componentes y la unidireccionalidad de datos han transformado la forma en que se desarrollan aplicaciones web. El modularidad inherente ha facilitado la gestión y mantenimiento del código, acelerando el desarrollo de aplicaciones escalables.

Comparativa con Otros Frameworks y Bibliotecas

Al compararlo con otras soluciones como Angular o Vue. js, React realmente se destaca por su orientación a la fabricación de interfaces de usuario y su habilidad para gestionar datos de forma reactiva. Mientras que Angular presenta una metodología más estructurada y un amplio conjunto de funcionalidades, React se especializa en componentes reutilizables, otorgando mayor flexibilidad a los desarrolladores.

Aunque Vue. js comparte ciertas características con React, generalmente es más fácil de aprender y adopta un enfoque más gradual. En definitiva, la selección entre estas herramientas depende de las preferencias individuales, los requisitos del proyecto y las capacidades del equipo. React resalta por su rendimiento, su sólido ecosistema de herramientas y la amplia comunidad que lo apoya, lo que lo convierte en una opción popular para diversos tipos de proyectos.

Principios Fundamentales de React

React se basa en principios esenciales que le otorgan singularidad en el ámbito del desarrollo web, proporcionando eficiencia y escalabilidad a las aplicaciones.

- **La "Virtual DOM" y su Impacto en la Eficiencia de React**

La "Virtual DOM" es un concepto fundamental en React. En vez de alterar directamente el DOM, React opera con una representación virtual de este. Esta versión más ligera permite a React realizar comparaciones eficaces entre la versión actual y la previa del DOM virtual, identificando solo los cambios necesarios que deben aplicarse al DOM real. Este enfoque

disminuye las manipulaciones costosas del DOM y mejora el rendimiento general de la aplicación.

- **Declaratividad versus Enfoques Imperativos**

React se caracteriza por su método declarativo, en contraste con los enfoques imperativos convencionales. En la programación declarativa, los programadores se enfocan en describir las acciones que debe realizar la interfaz de usuario de acuerdo con el estado de la aplicación. Por el contrario, en la programación imperativa, se especifica el procedimiento para alcanzar un resultado particular (Gómez, 2024).

La implementación de un enfoque declarativo en React permite a los programadores concentrarse en la lógica de la aplicación en lugar de en la manipulación directa del Document Object Model. Este método produce un código más claro, ordenado y fácil de mantener, lo que ayuda a comprender y gestionar la aplicación a lo largo del tiempo.

- **Unidireccionalidad de los Datos (One-Way Data Binding)**

En React, la unidireccionalidad de los datos significa que el flujo de información sigue una única dirección, generalmente desde los componentes padres hacia los hijos. Esto implica que cualquier modificación en el estado de un componente se difunde hacia abajo en la jerarquía de componentes, actualizando las vistas en consecuencia.

En donde el método facilita la administración del estado de la aplicación, proporcionando previsibilidad y control sobre el flujo de datos. Cada componente se vuelve independiente y más sencillo de depurar, ya que su estado y funcionamiento están claramente definidos.

En el estudio conlleva a tener el principio que son la base fundamental de la eficacia y la eficiencia de React en la creación de aplicaciones web contemporáneas. Entender y aplicar estos conceptos es vital para aprovechar al máximo el potencial de React en el desarrollo de aplicaciones sólidas y de alto rendimiento.

- **Ventajas de utilizar React en el desarrollo web**

React, como una biblioteca de JavaScript, presenta una serie de ventajas clave que favorecen su popularidad y efectividad en la creación de aplicaciones web modernas, generando beneficios significativos tanto para los programadores como para los proyectos en general.

Ventajas Clave de React en el Desarrollo Web

React ha ganado relevancia en el ámbito del desarrollo web por varias razones que realmente lo distinguen:

Eficiencia Mejorada: Gracias a su "DOM virtual", React permite realizar actualizaciones de manera rápida y eficaz, lo que disminuye la carga en el DOM real y optimiza el rendimiento global de la aplicación.

Reutilización de Componentes: Su enfoque centrado en componentes simplifica la creación de elementos modulares que pueden ser reutilizados. Esta característica es fundamental para gestionar el código de forma efectiva y agilizar el proceso de desarrollo, permitiendo crear elementos de interfaz que se pueden utilizar en diferentes partes de la aplicación.

Versatilidad y Compatibilidad: React se acopla con facilidad a otras herramientas y frameworks, brindando a los desarrolladores la opción de incorporar tecnologías adicionales según las necesidades de su proyecto.

- **Reutilización de Componentes y su Impacto en la Productividad del Desarrollador**

La posibilidad de reutilizar componentes en React es extremadamente valiosa. Al desarrollar bloques modulares y auto conclusivos, los desarrolladores pueden crear una biblioteca de componentes que se emplean en múltiples áreas de la aplicación. Esta reutilización no solo mejora la efectividad y la rapidez del desarrollo, sino que también minimiza la redundancia y simplifica el mantenimiento del código.

En donde las metodologías promueven un desarrollo más dinámico al centrarse en las funciones específicas de cada componente, lo que produce aplicaciones más ordenadas y sencillas de mantener durante el ciclo de vida del proyecto.

- **Comunidad, Soporte y Ecosistema de React**

El respaldo de una comunidad activa y comprometida representa un valioso recurso para cualquier tecnología. En el caso de React, su extensa comunidad proporciona una infinidad de recursos, tutoriales, documentación y ejemplos, facilitando el aprendizaje y la resolución de problemas.

Además, esta comunidad dinámica impulsa un ecosistema diverso que enriquece la experiencia de desarrollo. Los desarrolladores pueden acceder a una amplia gama de librerías

y soluciones adicionales que complementan las capacidades de React, adaptándose a las necesidades específicas de cada proyecto.

En conjunto, estas ventajas distinguen a React como una opción atractiva para el desarrollo web, ofreciendo soluciones eficientes, escalables y favoreciendo la productividad de los desarrolladores en la creación de aplicaciones robustas y de alto rendimiento.

Herramientas y Preparación para el Desarrollo en React

Antes de sumergirse en el desarrollo con React, es esencial prepararse con las herramientas adecuadas y comprender los conocimientos necesarios. Esta sección explora la configuración inicial del entorno de desarrollo, los requisitos previos y la instalación de las herramientas esenciales para trabajar con React.

- **Configuración del Entorno de Desarrollo**

El entorno de desarrollo es esencial para trabajar con React de forma eficiente. Se recomienda elegir un editor de texto robusto, como Visual Studio Code o Atom, que incluya extensiones específicas para JavaScript y React. En donde el tipo de editores facilita tanto la escritura de código como la depuración de aplicaciones React. Además, es beneficioso conocer herramientas de desarrollo web, como Chrome DevTools, que permiten analizar, corregir errores y mejorar el rendimiento de las aplicaciones en línea.

- **Conocimientos Previos Recomendados**

Antes de adentrarse en React, resulta ventajoso tener una base básica sobre las tecnologías web clave. Es importante entender HTML para la estructura del contenido, CSS para el estilo y la presentación, y JavaScript para la lógica y la interactividad, ya que esto es fundamental para desarrollar efectivamente con React.

- **Instalación de React y Herramientas Adicionales**

El paso siguiente y fundamental es instalar React y las herramientas adicionales. Utilizar herramientas como `create-react-app` simplifica el proceso de configuración inicial de un proyecto React, proporcionando una estructura establecida y configuraciones eficaces. Es vital conocer y emplear npm (Node Package Manager) para controlar las dependencias y bibliotecas del proyecto. Con npm, puedes instalar, actualizar y gestionar las diversas bibliotecas que son necesarias para el desarrollo en React (Sastre, 2026).

En el preparado con estas herramientas y conocimientos te proporcionará una base sólida para comenzar a trabajar con React. Con una configuración adecuada y un buen

entendimiento de las tecnologías implicadas, estarás mejor capacitado para desarrollar aplicaciones web dinámicas y eficientes utilizando React.

Instalación en macOS:

- Node.js y npm: Se descarga el instalador de Node.js para macOS desde el sitio web oficial de Node.js y sigue las instrucciones para instalarlo. npm se instala automáticamente con Node.js.
- Terminal: Se sugiere abrir la aplicación Terminal, que es la interfaz de línea de comandos en macOS.
- Instalación de create-react-app: Se ejecuta el siguiente comando en la Terminal: npx create-react-app nombre-de-tu-proyecto.
- Reemplaza "nombre-de-tu-proyecto" por el nombre que desees para tu proyecto.

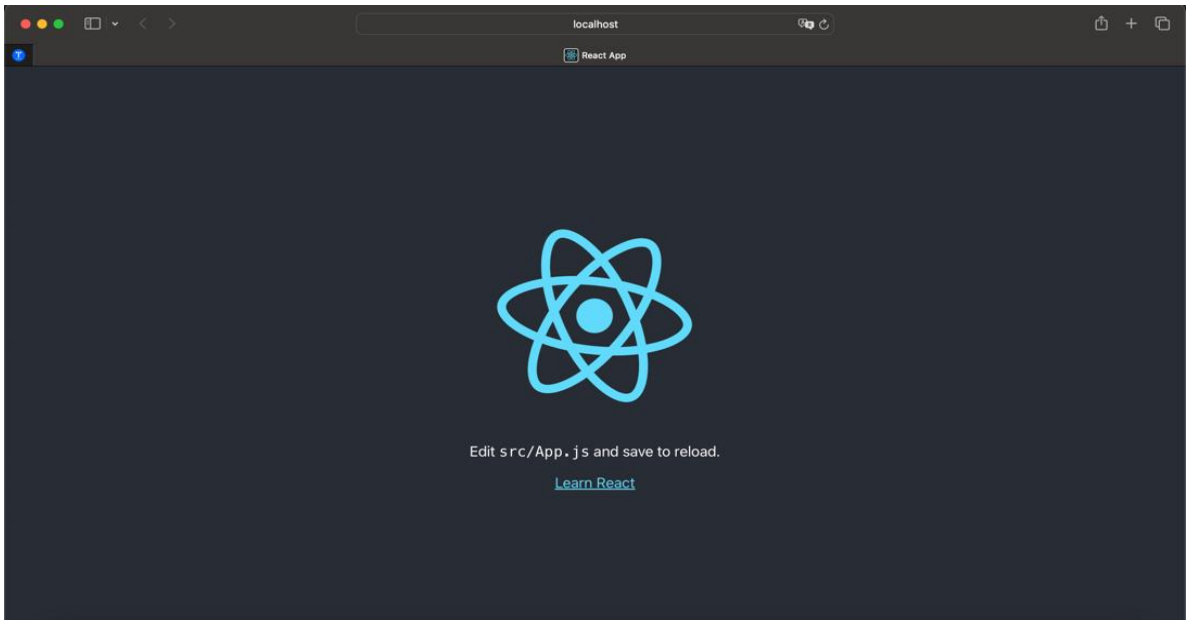
Ejecución del Proyecto React

- Se navega al directorio del proyecto: cd nombre-de-tu-proyecto
- Se inicia el servidor de desarrollo con el comando: npm start

Esto abrirá la nueva aplicación React en el navegador predeterminado.

Figura 1

Aplicación de React



Fuente: Elaboración propia

Extensiones del Editor de Código: Para utilizar Visual Studio Code u otro editor, se puede instalar extensiones como "Reactjs code snippets" para mejorar la experiencia de desarrollo.

Instalación en Windows

- Node.js y npm: Se descarga el instalador de Node.js para Windows desde el sitio web oficial de Node.js y se sigue las instrucciones para instalarlo. npm se instala automáticamente con Node.js.
- Command Prompt o PowerShell: Se sugiere abrir Command Prompt o PowerShell, que son interfaces de línea de comandos en Windows.
    - Instalación de create-react-app: Se ejecuta el siguiente comando en Command
    - Prompt o PowerShell: npx create-react-app nombre-de-tu-proyecto

    Se reemplaza "nombre-de-tu-proyecto" por el nombre que desees para tu proyecto.

    - Ejecución del Proyecto React:
    - Se navega al directorio del proyecto:
    - cd nombre-de-tu-proyecto

    Se inicia el servidor de desarrollo con el comando: npm start

    Esto abrirá la nueva aplicación React en el navegador predeterminado.

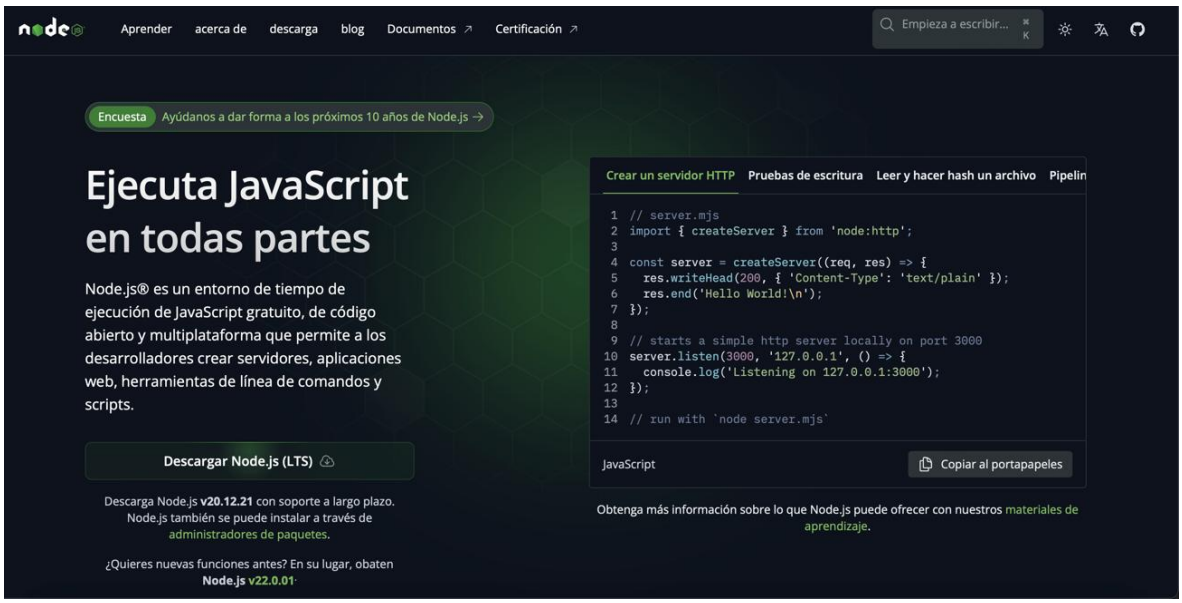
    Creación de un Proyecto Básico de React

Instalar nodejs

En la simplificación del desarrollo del proyecto y la administración de paquetes, utilizaremos npm. Es fundamental que tengas instalado Node. js en tu computadora para poder usarlo adecuadamente.

Figura 2

Instalar nodejs



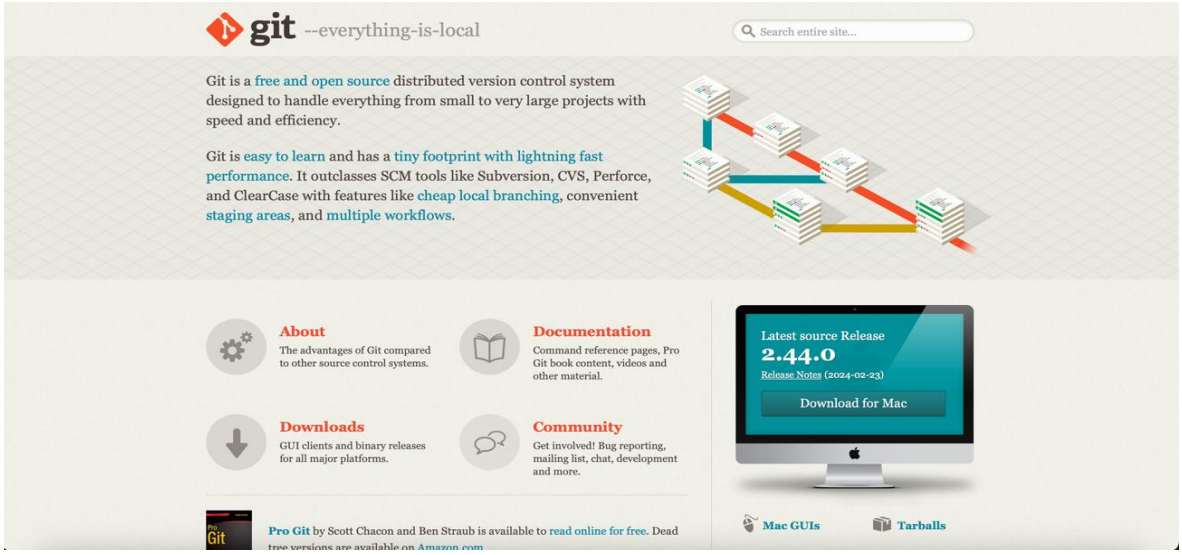
Fuente: Elaboración propia
Fuente: https://nodejs.org/en/

Instalar git

Si bien este procedimiento no se considera obligatorio, se sugiere encarecidamente su realización. Dado que se anticipa la ejecución de comandos desde la terminal y la probable utilización de git durante el desarrollo, se recomienda instalar esta herramienta.

Figura 3

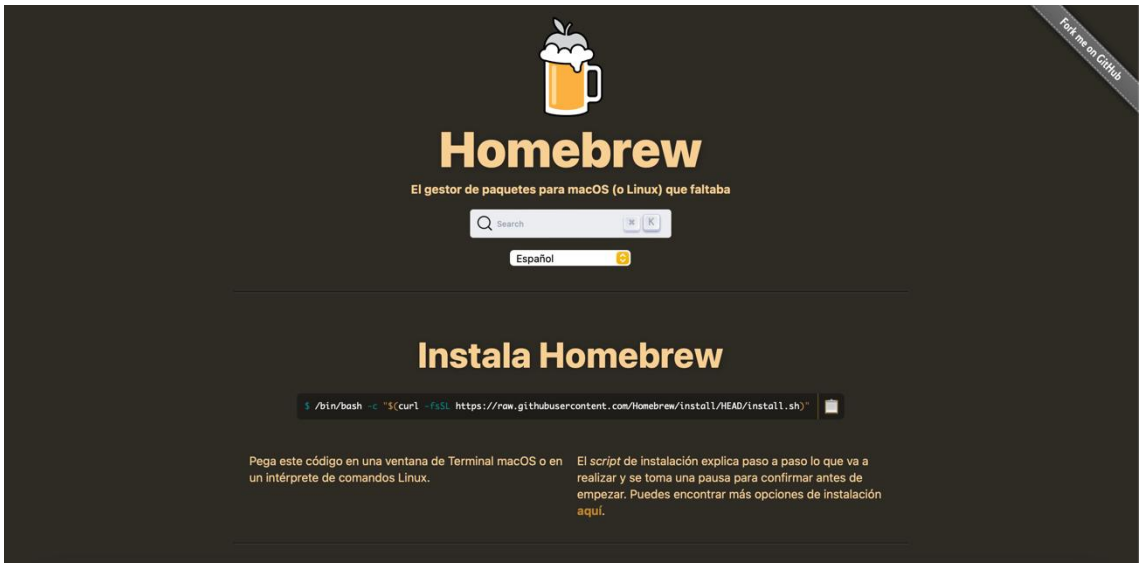
Instalar git 1



Fuente: Elaboración propia
Fuente: https://brew.sh

Figura 4

Instalar git 2



Fuente: Elaboración propia
Fuente: https://brew.sh

Ejemplo de Creación del Proyecto:

- Crear una carpeta

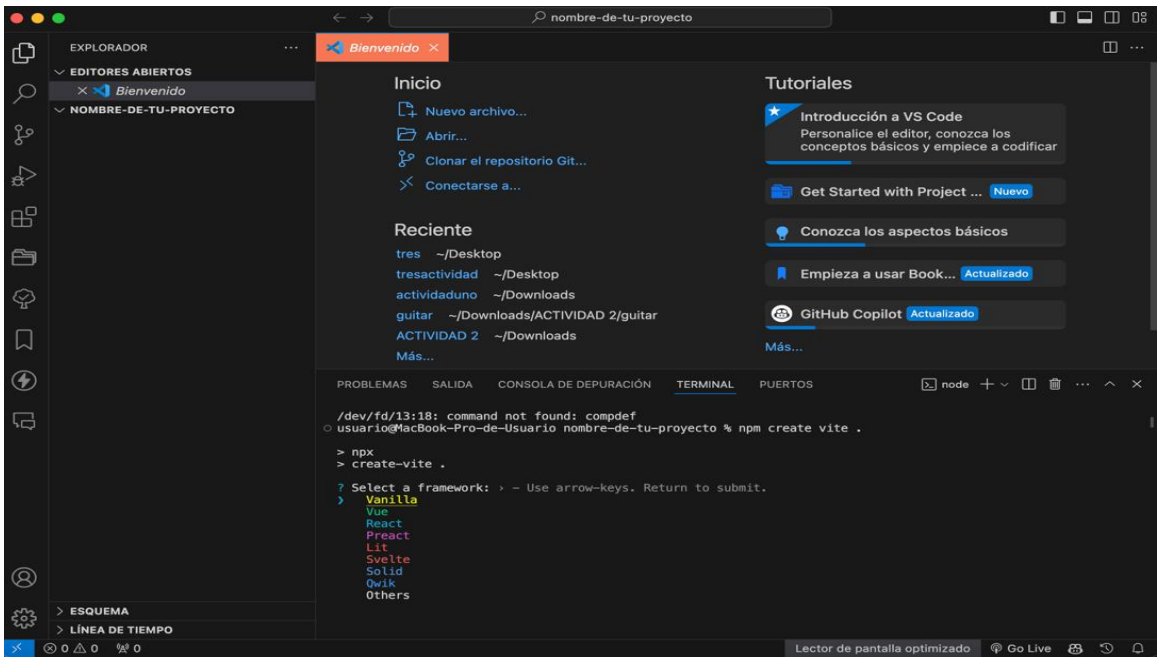
Se sugiere crear una carpeta en el escritorio o en la ubicación preferida del usuario, la cual deberá denominarse "nombre-de-tu-proyecto", sustituyendo esta etiqueta por el nombre deseado para el proyecto, el cual debe estar escrito en minúsculas.

- Crear el proyecto

Se procede a ejecutar el siguiente comando dentro de la carpeta en la que se desea crear el proyecto: npm create vite

Figura 5

Aplicación npm create vite

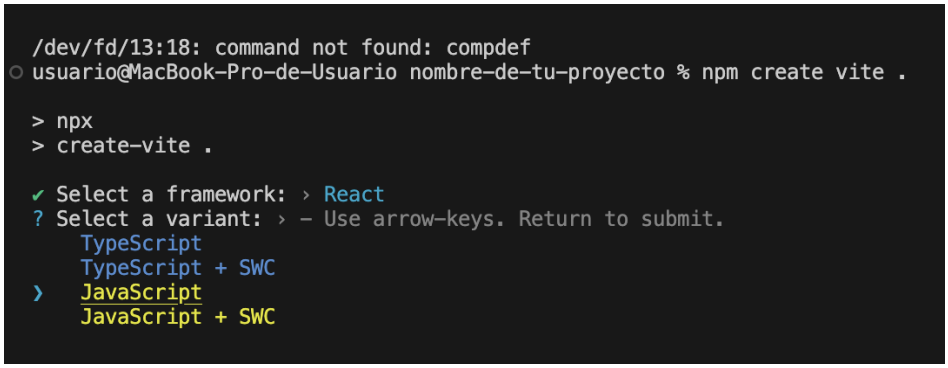


Fuente: Elaboración propia
Fuente: https://brew.sh

Se elige React como la tecnología y posteriormente se selecciona JavaScript como el lenguaje de programación.

Figura 6

Lenguaje de programación npm create vite



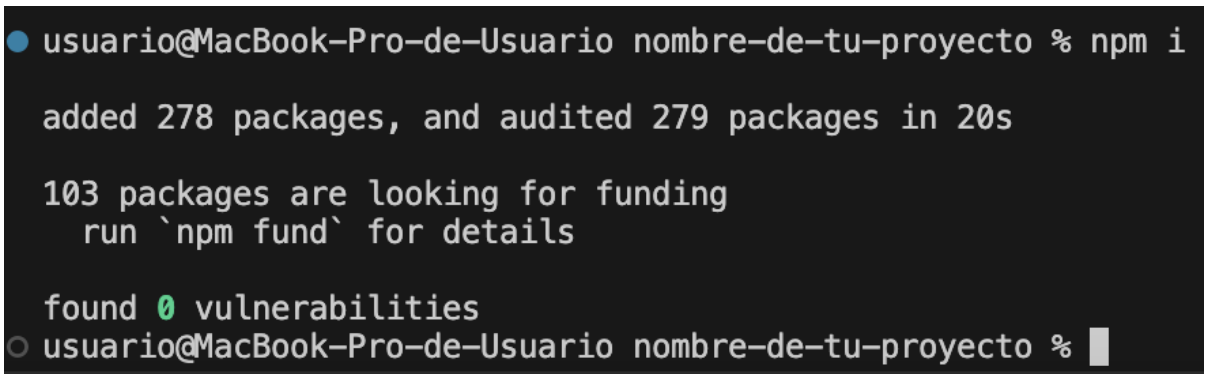
Fuente: Elaboración propia
Fuente: https://brew.sh

- npm i

En trabajos que utilizan Node. js, para agregar las bibliotecas requeridas, tienes la opción de emplear la abreviatura "i", que representa "install". Por lo tanto, "npm i" y "npm install" realizan la misma tarea.

Figura 7

Lenguaje de programación npm i



Fuente: Elaboración propia
Fuente: https://brew.sh

Estructura de un Proyecto React: Se observará que se han generado diversos archivos y directorios. Es crucial comprender la estructura del proyecto. A continuación, se proporciona información sobre cada componente:

package.json:

- Contiene el nombre y la versión del proyecto.
- Incluye los scripts asociados al proyecto.
- Permite gestionar las dependencias. Es posible agregar más dependencias instalando los paquetes mediante el comando: npm i nombre del paquete

Al ejecutar el comando: npm i

Al ejecutar el comando, se procederá a instalar todos los módulos referenciados en el archivo package.json.

Node_modules: en donde el un directorio donde se guardan todas las dependencias del proyecto, las cuales están listadas en el archivo package.json.

Gitignore: Este archivo indica qué archivos y carpetas deben ser ignorados por el sistema de control de versiones git. Por ejemplo, la carpeta node_modules no se debería subir a ningún repositorio, ya que ocupa mucho espacio y las dependencias pueden instalarse automáticamente a partir de la información en package.json usando el comando npm i.

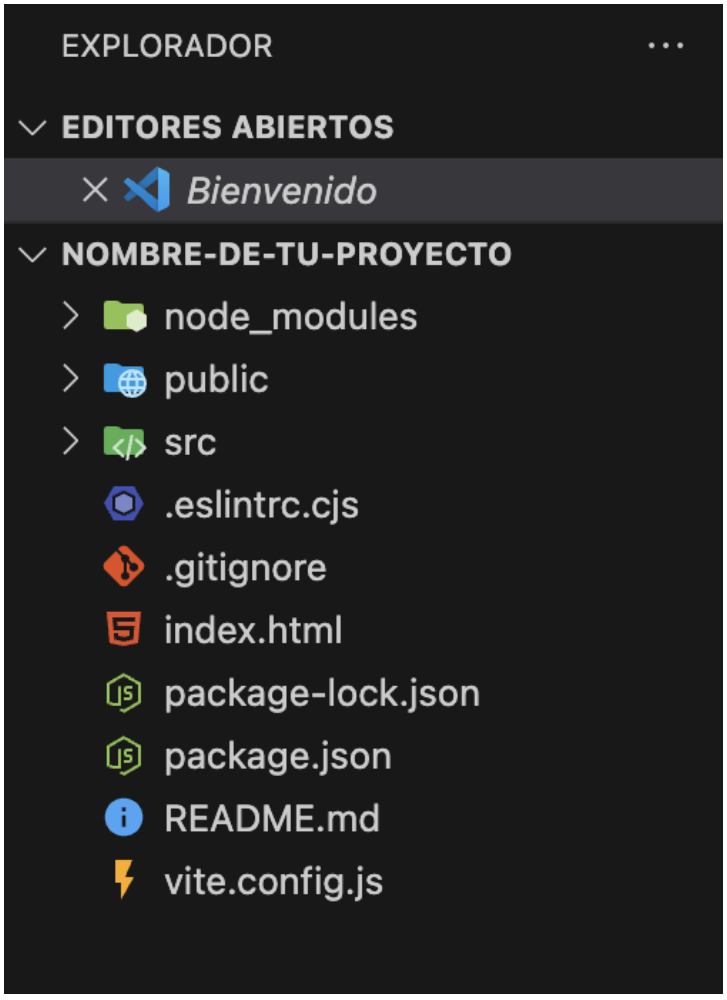
Public: En esta carpeta se pueden guardar los recursos estáticos del proyecto, como videos, audios, imágenes, etc. Todo lo que esté en esta carpeta se transferirá a la carpeta build durante la compilación de la aplicación sin ningún tipo de procesamiento extra. Los recursos

en esta carpeta se pueden acceder mediante código JSX, utilizando, por ejemplo, process.env para obtener la ruta de una imagen.

SRC: albergará el código fuente que se desarrollará durante el proyecto. Esta es la ubicación donde se encuentra la esencia del desarrollo.

Figura 8

Aplicación de SRC



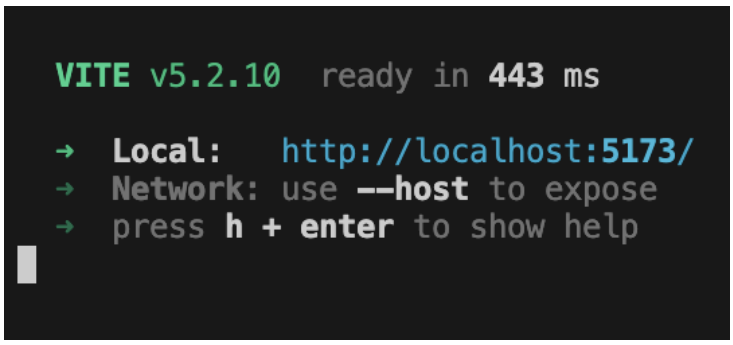
Fuente: Elaboración propia
Fuente: https://brew.sh

Arrancar el proyecto y visualizarlo en el navegador

Para iniciar el proyecto y visualizarlo en el navegador, se puede utilizar el siguiente comando en la terminal: npm run dev

Figura 9

Aplicación de npr run dey

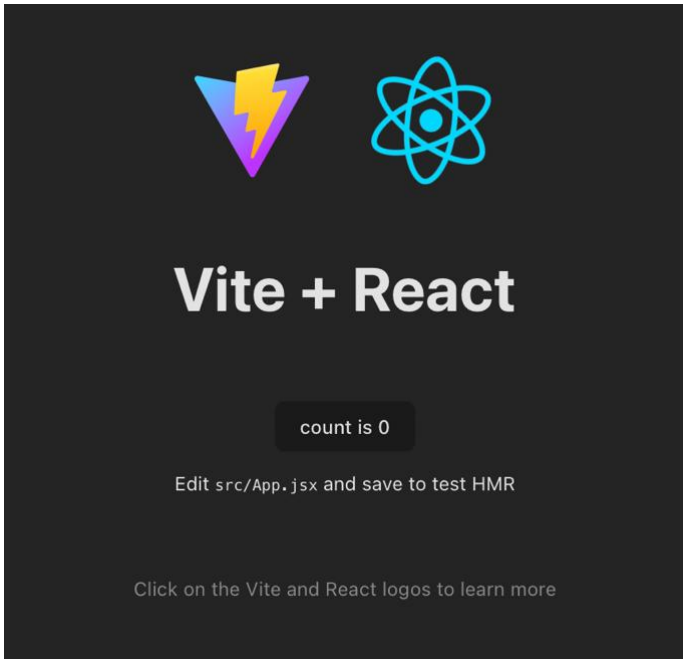


Fuente: Elaboración propia
Fuente: https://brew.sh

Al acceder a **http://localhost:5173/,** se podrá visualizar la aplicación en el navegador. Este enlace representa la dirección local donde se está ejecutando el servidor de desarrollo y donde se podrá interactuar con la aplicación en tiempo real.

Figura 10

Aplicación de localhost



Fuente: Elaboración propia
Fuente: https://brew.sh

Configuración de Visual Studio Code para Desarrollar en React

Trabajar con React en Visual Studio Code se ve optimizado con ciertas extensiones y ajustes en la configuración del editor. Aquí se detallan extensiones clave y ajustes

recomendados para mejorar tu flujo de trabajo al desarrollar en React. Por ende, en las extensiones esenciales para React en Visual Studio Code:

- **ESLint:** Se facilita el cumplimiento de las reglas definidas por ESLint en tu proyecto, mostrando advertencias y errores directamente en el editor.
- **Prettier - Code formatter:** Se garantiza la uniformidad del código al formatearlo automáticamente, siguiendo un estilo predefinido.
- **Babel JavaScript:** Se brinda soporte para resaltado de sintaxis y snippets para JavaScript moderno, incluyendo JSX, facilitando la escritura de código.
- **Auto Close Tag:** Cierre automático de etiquetas HTML, una característica práctica al trabajar con JSX y componentes en React.

Configuración Personalizada:

- Se aconseja abrir Visual Studio Code.
- Dirígete a "File" (Archivo) "Preferences" (Preferencias) "Settings" (Configuración).
- Haz clic en "Open Settings (JSON)" (Abrir Configuración en formato JSON).

Ejemplo de Configuración (settings. json) para React en Visual Studio Code:

Incorpora estas configuraciones en tu `settings. json` para ajustar tu entorno de desarrollo para React.

Figura 11

Lenguaje de programación

```
{
  "editor.formatOnSave": true,
  "javascript.format.enable": false,
  "eslint.alwaysShowStatus": true,
  "prettier.disableLanguages": ["javascript"],
  "prettier.jsxSingleQuote": true,
  "prettier.useTabs": false,
  "emmet.includeLanguages": {
    "javascript": "javascriptreact"
  }
}
```

Fuente: Elaboración propia
Fuente: https://brew.sh

- **`"editor.formatOnSave"`:** true: Activa el formateo automático al guardar el archivo.
- **`"javascript.format.enable"`:** false: Deshabilita el formateo por defecto para JavaScript.

- **`"eslint.alwaysShowStatus"`:** true: Muestra los problemas de ESLint en la barra de estado.
- **`"prettier.disableLanguages"`:** ["javascript"]: Desactiva Prettier para JavaScript ya que ESLint manejará el formateo.
- Ajustes adicionales modifican el comportamiento de Prettier y Emmet para archivos JavaScript y JSX.

Adaptar estas configuraciones a las preferencias específicas o a las necesidades del proyecto puede mejorar significativamente el flujo de trabajo al desarrollar aplicaciones en React dentro de Visual Studio Code.

Profundización en el Código JSX en Proyectos de React

El JSX (JavaScript XML) es una de las bases esenciales en el desarrollo de aplicaciones con React, ya que facilita la unión de la visualización, la funcionalidad y la manipulación mediante un modelo de programación declarativa. Aunque frecuentemente se discute sobre el JSX en un contexto técnico, dado que es una extensión sintáctica muy parecida a HTML, su importancia trasciende los aspectos técnicos; simboliza un enfoque metódico para crear interfaces digitales contemporáneas.

Desde el punto de vista de la ingeniería de software, el JSX está relacionado con enfoques de desarrollo que buscan crear elementos de interfaz que se pueden reutilizar. Esto permite que los componentes, tanto visuales como funcionales, se agrupen en unidades independientes, lo que fomenta el modularidad, la escalabilidad y simplifica el mantenimiento del código en aplicaciones más complejas. De acuerdo con Stoyan Stefanov (2016), el modelo de desarrollo declarativo contribuye a disminuir la complejidad de la manipulación de interfaces en comparación con el método imperativo, lo que facilita la actualización continua de sistemas interactivos.

Además, el JSX desempeña un papel clave en el diseño multimedia y en la experiencia del usuario, ya que mejora nuestra manera de pensar sobre interfaces que reaccionan a cambios de estado e interacciones. Esta propiedad es especialmente relevante en las aplicaciones multimedia actuales, donde la actualización de contenido y la superposición de diversos elementos visuales (Baddeley, Vargha-Khadem y Mishkin, 2014) requieren arquitecturas efectivas para su representación y renderizado. Así, Alex Banks y Eve Porcello (2020) argumentan que JSX mejora la conexión entre la estructura visual y el comportamiento

funcional, lo que permite la creación de sistemas digitales más dinámicos y sencillos de mantener.

En términos cognitivos, el modelo declarativo implementado con JSX influye en la forma en la que los desarrolladores comprenden y organizan las interfaces digitales.A diferencia de los enfoques clásicos, los cuales se apoyan en la aplicación de manipulaciones directas de la Document Object Model, el lenguaje JSX, por el contrario, favorece construcciones de representación estructurada de la interfaz gráficos en voluntades y componentes. Esta forma de estructuración contribuye a disminuir la carga cognitiva durante la/proceso de construcción de las interfaces gráficas, dada la existencia de relaciones más claras entre la lógica de interacción de las personas que utilizan la aplicación y la representación visual de la misma.

Por otro lado, el uso del lenguaje JSX se vincula a tendencias actuales de desarrollo del frontend centradas en la interoperabilidad, la reutilización y la productividad. La combinación entre JavaScript y estructuras declarativas apoyan los ciclos de trabajo colaborativos en proyectos multimedia complejos donde diseñadores, programadores y expertos en experiencia de usuario participan simultánea y recíprocamente a una construcción de interfaces gráficas digitales. Como consecuencia de esto, JSX no debe concebirse como una simple herramienta sintáctica sino como parte de un ecosistema metodológico que transforma los procesos de diseño e implementación de aplicaciones web interactivas.

Finalmente, desde un punto de vista de rendimiento y optimización, el lenguaje JSX también articula mecanismos de renderizados efectivos como él o el Virtual DOM, permitiendo minimizar el número de actualizaciones innecesarias de la interfaz real del navegador. La relación entre representación declarativa y optimización computacional es particularmente importante en sistemas multimedia con una elevada demanda gráfica e interactiva en la que la fluidez y la capacidad de respuesta son factores críticos a tener en cuenta para la experiencia de usuario.

Dentro del entorno académico, el análisis de JSX también es susceptible de ser explorado a partir de enfoques concernientes a la arquitectura de software, la interacción persona-ordenador y al propio diseño de sistemas digitales. El hecho de que JSX integre los modernos frameworks también demuestra la propia evolución de los paradigmas de desarrollo web hacia metodologías más adaptativas, modulares y de interacción constante. Por lo tanto, el estudio de JSX es científico en la medida en que se pone de manifiesto cómo determinados

sistemas tecnológicos son capaces de cambiar los procesos de producción, la representación visual y la experiencia digital del desarrollo multimedia contemporáneo.

Características Destacadas del Código JSX en React

Sintaxis Similar a HTML: El código JSX se asemeja a HTML, lo que facilita la creación de componentes y la representación de elementos visuales. Por ejemplo, para renderizar un elemento `**<div>**`, simplemente escribimos: const elemento = Hola, mundo! ;

Inserción de Expresiones JavaScript: JSX permite la inclusión de expresiones JavaScript dentro de llaves `**{}**`. Esto posibilita la introducción de lógica dinámica en la interfaz de usuario. Por ejemplo:

- const nombre = "Usuario";
- const saludo = <h1>Hola, {nombre}</h1>;

Uso de Componentes: JSX facilita la incorporación de componentes React en tu código. Puedes utilizar otros componentes dentro de tu código JSX, lo que promueve el modularidad y la reutilización de código. Por ejemplo:

Figura 12

Lenguaje de programación JSX

```
const Titulo = (props) => {
  return <h1>{props.texto}</h1>;
};

const App = () => {
  return (
    <div>
      <Titulo texto="¡Bienvenido!" />
    </div>
  );
};
```

Fuente: Elaboración propia
Fuente: https://brew.sh

- Conversión de JSX: Es esencial recordar que los navegadores no procesan JSX de forma directa. En cambio, este tipo de código necesita ser transformado a JavaScript convencional. Para ello, se utilizan herramientas como Babel, que realizan la

conversión del código JSX en invocaciones a `React. createElement()`. Estas invocaciones terminan convirtiéndose en objetos que describen cómo se verá la interfaz de usuario.

- Ventajas de JSX: JSX incrementa la claridad del código, ya que integra de manera armónica HTML y JavaScript. Esto simplifica la gestión y comprensión de la estructura de la interfaz de usuario, resultando en un desarrollo más eficaz.
- Además, JSX es esencial para desarrollar interfaces de usuario dinámicas e interactivas en React, dado que posibilita la manipulación de elementos y la gestión de eventos de manera intuitiva.
- El código JSX es uno de los aspectos que diferencian a React y contribuyen a su popularidad en el desarrollo de aplicaciones web. Gracias a su habilidad para combinar la simplicidad de HTML con la versatilidad de JavaScript, JSX se convierte en un recurso valioso para el diseño de interfaces de usuario modernas y efectivas.

JSX como extensión declarativa que utiliza JavaScript

JSX, que significa JavaScript XML, es una extensión de JavaScript que se utiliza principalmente en React para construir interfaces de usuario mediante una representación declarativa de los componentes visuales. Su estructura es similar a una etiqueta HTML, lo que facilita la visualización y comprensión del diseño en el código. Sin embargo, es crucial no confundir JSX con el lenguaje HTML en sí; más bien, actúa como un formato intermedio que se transforma en instrucciones que el motor de JavaScript puede leer (Banks y Porcello, 2020).

En donde un enfoque metodológico, JSX facilita un desarrollo que produce componentes reutilizables, combinando la apariencia visual con la lógica operativa y el comportamiento interactivo. Este método mejora la capacidad de mantenimiento y escalabilidad de aplicaciones web complejas, especialmente en contextos multimedia que requieren actualizaciones de contenido dinámico y una interacción continua.

Además, JSX permite la incorporación de expresiones de JavaScript dentro de estructuras declarativas utilizando llaves {}.Es particularmente ventajoso para crear contenido dinámico, gestionar estados y realizar interacciones en tiempo real con datos desde el navegador. Por lo tanto, JSX no solo hace que la apariencia visual de la interfaz de una aplicación sea más simple, sino que también fortalece la relación entre el diseño interactivo y la programación funcional.

Uso de Babel en la transformación de JSX

A pesar de que JSX facilita la creación de interfaces de forma declarativa, los navegadores no pueden procesar esta forma de notación de forma directa, lo que implica que no son capaces de interpretarla. Por esa razón, es necesario que el código JSX pase por un proceso de transpiración que lo transforme en JavaScript estándar, el cual es apto para el entorno web. En el ámbito de la programación, Babel es una herramienta fundamental que modifica el código al convertir los elementos JSX en llamadas equivalentes de JavaScript, similares a las realizadas con la función React. Créate Element.

Este procedimiento permite que los sistemas de interfaces diseñados con la sintaxis JSX sean analizados e interpretados de manera adecuada por el navegador. Desde una perspectiva técnica, Babel también ayuda a garantizar que diferentes versiones de JavaScript y una variedad de entornos de ejecución sean compatibles, lo que favorece la interoperabilidad y la estabilidad de las aplicaciones web contemporáneas. Por lo tanto, Babel se considera un elemento clave en los procesos de desarrollo frontend que utilizan React y en aquellas arquitecturas que se fundamentan en un modelo de componentes dinámicos.

El concepto de Hooks en React

Los Hooks son funciones que se implementaron en React para manejar el estado, el ciclo de vida y las lógicas funcionales dentro de los componentes funcionales. Antes de su aparición, estas capacidades eran principalmente atribuidas a los componentes basados en clases. No obstante, los Hooks han facilitado la estructura del código, favoreciendo desarrollos más modulares y utilizables. De acuerdo con Huet, (2020), indica que la introducción de los Hooks sigue un enfoque que busca proporcionar una forma de aprovechar las características avanzadas de React sin depender de una estructura de clase tradicional. Esto, a su vez, mejora la disposición lógica de las aplicaciones y establece un método para compartir funcionalidades entre componentes (Gomila, 2026).

Desde una perspectiva metodológica, los Hooks respaldan los principios de separación de responsabilidades, mantenibilidad y reutilización lógica en cualquier lógica de interacción. También contribuyen a disminuir la complejidad estructural de las aplicaciones web interactivas y de los sistemas multimedia que funcionan con múltiples estados dinámicos.

Funcionamiento del Hook useState

Uno de los Hooks más utilizados en React es useState. Permite gestionar estados internos dentro de los componentes funcionales. El Hook manageState da lugar a una doublet

que incluye un valor de estado y una función de actualización del valor de ese estado, en interacciones que realiza el usuario.

Desde una perspectiva funcional, useState permite la actualización dinámica de la interfaz en respuesta a eventos, cambios de datos o acciones del propio usuario. Cuando se produce un cambio en el estado, React lanza un nuevo proceso de renderizado del componente afectado. En ese momento, se vuelven a renderizar solo aquellos elementos que lo requieran y que estén en el Virtual DOM.

Tal es el caso que en aplicaciones multimedia y en interfaces interactivas, el uso del useState es particularmente importante para gestionar elementos dinámicos como los formularios; los controles visuales; la navegación interactiva; la reproducción multimedia y los cambios de contenido en función del contexto. Por tanto, el Hook es una herramienta fundamental en el desarrollo de experiencias digitales reactivas y adaptativas del ecosistema React.

Uso del Hook `useState` en React

El hook `**useState**` es uno de los hooks fundamentales en React y se emplea para añadir estado a componentes funcionales. Permite a los componentes funcionales tener su propio estado interno, lo que posibilita el seguimiento de cambios y la actualización dinámica de la interfaz de usuario.

Características Clave del Hook `useState`

- Inicialización del Estado: `useState` se utiliza para declarar una variable de estado y establecer su valor inicial. Por ejemplo:

Figura 13

Lenguaje de programación useState

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

const EjemploUseState = () => {
  const [contador, setContador] = useState(0);

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

Fuente: Elaboración propia
Fuente: https://brew.sh

- Actualización del Estado: `**useState**` proporciona una función (`setContador` en el ejemplo anterior) que se utiliza para actualizar el estado. Cuando se llama a esta función con un nuevo valor, React se encarga de re-renderizar el componente y reflejar el cambio en la interfaz.
- Estado con Objetos o Arrays: El estado no está limitado a valores simples (números o strings). Puede ser un objeto, un array, o cualquier otro tipo de dato. Por ejemplo:

  const [usuario, setUsuario] = useState({ nombre: 'Ejemplo', edad: 25 });

- Funciones como Argumentos: En lugar de pasar un valor directamente a la función que actualiza el estado, `**useState**` también permite el uso de una función para calcular el nuevo estado basado en el estado previo.

  const [contador, setContador] = useState(0);

  // Incrementar el contador utilizando una función
  setContador((prevContador) => prevContador + 1);

- El hook `**useState**` es esencial para dotar de dinamismo a los componentes funcionales en React, permitiendo la gestión de su estado interno. Esto posibilita la creación de interfaces de usuario interactivas y reactivas.
- El hook `useState` es esencial en React, permitiendo a los componentes funcionales mantener y gestionar su propio estado interno. Esto les otorga la capacidad de ser

reactivos, actualizar la interfaz de usuario y manejar cambios dinámicos. Aquí se presentan seis ejemplos que ilustran distintos usos del hook `useState` en React.

Estado Simple - Contador

El primer ejemplo muestra cómo useState permite la gestión de un estado simple, en este caso, un contador. Al utilizar `useState`, se inicializa el estado contador con un valor inicial de 0. Al hacer clic en el botón "Incrementar", la función `setContador` actualiza el estado contador, y el componente se re-renderiza mostrando el nuevo valor.

Figura 14

Lenguaje de programación useState

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

const Contador = () => {
  const [contador, setContador] = useState(0);

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

Fuente: Elaboración propia
Fuente: https://brew.sh

Estado con Objetos

En este caso, `**useState**` se utiliza para gestionar un estado más complejo, un objeto usuario con propiedades como `nombre` y `edad`. Al presionar el botón "Actualizar Datos", la función `setUsuario` modifica el estado del usuario a un nuevo objeto con valores diferentes, lo que causa la actualización del componente con los nuevos datos del usuario.

Figura 15

Lenguaje de programación useState

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

const DatosUsuario = () => {
  const [usuario, setUsuario] = useState({ nombre: 'Ejemplo', edad: 25 });

  return (
    <div>
      <p>Nombre: {usuario.nombre}</p>
      <p>Edad: {usuario.edad}</p>
      <button onClick={() => setUsuario({ nombre: 'Nuevo', edad: 30 })}>
        Actualizar Datos
      </button>
    </div>
  );
};
```

Fuente: Elaboración propia
Fuente: https://brew.sh

- Actualización del Estado con Funciones:

Em donde el ejemplo utiliza una función como argumento en `setValor`. El estado actual (`prevValor`) se utiliza para calcular y actualizar un nuevo estado. Al hacer clic en el botón "Incrementar", se incrementa el valor del estado de manera reactiva.

Figura 16

Lenguaje de programación en los estados con funciones

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

const EstadoFuncion = () => {
  const [valor, setValor] = useState(0);

  return (
    <div>
      <p>Valor: {valor}</p>
      <button onClick={() => setValor((prevValor) => prevValor + 1)}>
        Incrementar
      </button>
    </div>
  );
};
```

Fuente: Elaboración propia
Fuente: https://brew.sh

Uso de `useState` para una Lista

Aquí, `useState` se emplea para manejar una lista de elementos. La función `agregarItem` actualiza la lista al añadir un nuevo elemento. Cada vez que se presiona el botón "Agregar Elemento", se agrega un nuevo elemento a la lista y se re-renderiza el componente mostrando la lista actualizada.

Figura 17

Lenguaje de programación useState agregar elementos

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

const ListaItems = () => {
  const [items, setItems] = useState([]);

  const agregarItem = () => {
    setItems([...items, `Elemento ${items.length + 1}`]);
  };

  return (
    <div>
      <button onClick={agregarItem}>Agregar Elemento</button>
      <ul>
        {items.map((item, index) => (
          <li key={index}>{item}</li>
        ))}
      </ul>
    </div>
  );
};
```

Fuente: Elaboración propia
Fuente: https://brew.sh

Uso de `useState` para un Array de Objetos

En este caso, `**useState**` controla un array de objetos (tareas) representando tareas con sus estados de completado. Se muestra una lista de tareas con su estado de completado. El componente se renderiza mostrando las tareas y su estado, ya sea "Completada" o "Pendiente".

Figura 18

Lenguaje de programación useState para un array

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

const Tareas = () => {
  const [tareas, setTareas] = useState([
    { id: 1, nombre: 'Tarea 1', completada: false },
    { id: 2, nombre: 'Tarea 2', completada: true },
  ]);

  return (
    <div>
      <ul>
        {tareas.map((tarea) => (
          <li key={tarea.id}>
            {tarea.nombre} - {tarea.completada ? 'Completada' : 'Pendiente'}
          </li>
        ))}
      </ul>
    </div>
  );
};
```

Fuente: Elaboración propia
Fuente: https://brew.sh

Uso de `useState` para el Control de Formularios

En este ejemplo, `**useState**` es utilizado para controlar el valor de un campo de texto de un formulario. Cada vez que se escribe en el input, la función `**handleChange**` actualiza el estado `**texto**`. Al presionar el botón "Enviar", se muestra en la consola el texto ingresado, demostrando cómo `**useState**` permite el manejo dinámico de formularios.

Figura 19

Lenguaje de programación useState para el control de formularios

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

const Formulario = () => {
  const [texto, setTexto] = useState('');

  const handleChange = (event) => {
    setTexto(event.target.value);
  };

  const handleSubmit = (event) => {
    event.preventDefault();
    console.log('Texto enviado:', texto);
  };

  return (
    <form onSubmit={handleSubmit}>
      <input type="text" value={texto} onChange={handleChange} />
      <button type="submit">Enviar</button>
    </form>
  );
};
```

Fuente: Elaboración propia
Fuente: https://brew.sh

Estos ejemplos ilustran cómo `**useState**` es empleado para manejar distintos tipos de estados en componentes funcionales en React, mostrando su versatilidad y utilidad en el desarrollo de aplicaciones interactivas y dinámicas.

Características de funcionamiento del Hook useState

El Hook useState se considera uno de los mecanismos más importantes dentro de la librería para controlar el estado dentro de los componentes funcionales. Es gracias a este mecanismo que las arquitecturas tradicionales de los componentes despojados de estado basados en clases muy típicos, comienzan a dar paso a las arquitecturas en el modo que describen las estructuras de estilo modular, declarativo, reutilizable y apto para arquitecturas de desarrollo más eficientes asistiendo así a aplicaciones web contemporáneas.

Desde una perspectiva de ingeniería de software, el useState gestiona la forma de información dinámica, como es la del comportamiento de la interfaz del usuario, reflejando así

en la actualización reactiva de componentes que significa la capacidad de React para poder reaccionar, en función de distintas circunstancias o eventos, cambios de datos o interacciones del usuario. A decir de Alex Banks y Eve Porcello (2020), este modo intensifica la construcción de interfaces interactivas a través de un flujo de actualización de estados y renderizando dinámicamente los mismos.

Inicialización del estado

Uno de los aspectos más relevantes del useState consiste en laeпутат declaración de variables de estado en la misma declaración que el valor inicial que les serviría a los mismos. Es gracias a este mecanismo que los componentes funcionales podrán almacenar información en la misma declaración que la inicial de la interacción en la que nos encontramos trabajando.

Desde una perspectiva técnica, la inicialización del estado establece la base de valores sobre la cual se ejecutarán posteriores actualizaciones y renderizado de la interfaz de manera dinámica. Este mecanismo resulta especialmente interesante para las aplicaciones multimedia que requieren la representación de datos de una forma dinámica, trabajos con paneles de información dinámica, formularios o sistemas de navegación adaptable.

Actualización reactiva del estado

El Hook useState permite las funciones de actualización del mismo, pues React proporciona aplicar funciones específicas que modifiquen el estado interno del componente. Cuando el estado varía, React ejecutará nuevamente todo el proceso de renderizado a través del Virtual DOM, refrescando solamente los elementos asociados al mismo dentro de la interfaz.

El razonamiento reactivo del rendimiento de las aplicaciones interactivas se basa precisamente en disminuir la manipulación del DOM real, siguiendo lo argumentado por Stoyan Stefanov (2016), donde los modelos reactivos contribuyen a mejorar eficiencia, mantenibilidad y capacidad de respuesta dentro de sistemas digitales complicados.

La gestión del estado en estados complejos mediante objetos y arrays

La definición del uso de useState no queda restringida a los datos de "tipos simples" -números o strings, por ejemplo- y es capaz de gestionar estructuras de datos complejas incluyendo el uso de objetos y de arrays que se utilizan para representar información dinámica dentro de aplicaciones multimedia y sistemas interactivos.

Desde la óptica metodológica, las características de useState propician gestionar de forma estructurada contenidos asociados a perfiles de usuarios, listas dinámicas, catálogos multimedia, sistemas de tareas y formularios complejos, por ejemplo. Como consecuencia, useState permite gestión de los procesos de organización y de actualización de datos de forma eficiente en interfaces reactivas.

Gestión del estado sobre el estado anterior

Otra de las propiedades relevantes de useState viene aparejada al uso de funciones como argumento para calcular el nuevo estado a partir de la información del estado anterior. Este tipo de mecanismo es especialmente ventajoso en situaciones donde es posible iniciar al mismo tiempo o de forma asincrónica un conjunto de actualizaciones.

En términos de procesamiento, este mecanismo contribuye a mantener la coherencia lógica y la estabilidad de los procesos de actualización en tiempo, evitando así inconsistencias que pueden resultar de los cambios simultáneos que se llevan a cabo en los datos del componente.

useState y el diseño de interfaces interactivas

El uso de useState cobra una especial relevancia dentro del diseño multimedia actual porque permite gestionar procesos de interacción y diseño en tiempo real. Aplicaciones que incluyen formularios inteligentes, sistemas de reproducción de multimedia, visualizaciones de datos, simulaciones interactivas y plataformas de colaboración, se fundamentan en mecanismos eficientes para la gestión de estado que garanticen continuidad y fluidez durante la interacción.

En este sentido, el Hook useState no debe considerarse únicamente como un elemento técnico de programación, sino un elemento estructural dentro de los actuales modelos de experiencia de usuario y de diseño interactivo. Su uso promueve la construcción de interfaces adaptativas que pueden reaccionar de manera inmediata a las acciones del usuario conforme a los principios de usabilidad, accesibilidad y feedback dinámico.

Aplicaciones metodológicas de useState en React

Los distintos usos de useState ponen de manifiesto su versatilidad dentro del desarrollo frontend actual. Desde la gestión de contadores hasta los formularios, listas dinámicas, estructuras de objetos y sistemas de interacción, sugieren que este Hook opera como un mecanismo central a la hora de gestionar estados en arquitecturas que se fundamentan en componentes.

Desde un punto de vista académico a la par que práctico, el estudio de useState conduce a comprender cómo cambian las tecnologías de renderizado reactivo los paradigmas de diseño e implementación de interfaces digitales. Así mismo, pone de relieve la interrelación existente entre arquitectura de software, interacción humano-computador y experiencia de usuario en el desarrollo multimedia actual.

Componentes en React

Los componentes constituyen la unidad estructural más básica en React, ya que permiten construir interfaces de usuario a través de arquitecturas modulares, reutilizables y escalables. Desde el punto de vista de la ingeniería de software, podemos definir un componente como una unidad funcional independiente que encapsula la estructura visual, lógica del comportamiento y mecanismos de interacción dentro de una entrada digital (Banks & Porcello, 2020).

El modelo basado en componentes representa un giro radical en el desarrollo web actual, pues forman parte de arquitecturas modulares a capacidad de reutilización y mantenibilidad del código, y tienden a sustituir los enfoques más tradicionales y monolíticos. Stoyan Stefanov (2016) expone que la fragmentación de las interfaces en componentes independientes contribuye a mejorar la organización estructural de aplicaciones complejas, al tiempo que facilita la escalabilidad, actualización y trabajo en grupo.

La perspectiva metodológica, los componentes permiten construir sistemas digitales de forma ordenada mediante estructuras jerárquicas e interdependientes. Cada componente puede funcionar de manera independiente y, al mismo tiempo, mantiene idoneidad para interactuar con otros componentes del sistema, potenciando así procesos de integración dinámica en el seno de las aplicaciones multimedia y las interfaces de interacción.

En consideración parece especialmente pertinente en un contexto como el del diseño multimedia actual, donde las interfaces requieren contar con contenidos constantemente actualizados, necesitan de adaptabilidad multiplataforma e inmediatez en respuesta a las acciones de los usuarios.

React presenta, fundamentalmente, dos modelos de componentes: componentes funcionales y componentes basados en clases. Los componentes funcionales corresponden a estructuras declarativas basadas en funciones JavaScript capaces de manejar interfaces y gestionar estados mediante Hooks como useState. En contraposición, los componentes de clase

son aquellos que se atacan mediante sintaxis de programación orientada a objetos junto con mecanismos tradicionales del ciclo de vida.

En el desarrollo frontend actual, los componentes funcionales han cobrado mayor importancia a partir de su simplicidad estructural, eficiencia y la posibilidad de integrarse con Hooks. Según Facebook (2019), este modelo favorece paradigmas de desarrollo más flexibles y evita la complejidad típica de las arquitecturas basadas en clases tradicionales.

Además, la utilización de componentes dentro de React se encuentra estrechamente relacionada con principios de reutilización y consistencia visual dentro del diseño de interfaces digitales. Los componentes permiten construir sistemas de diseño ordenados mediante elementos reutilizables como botones, formularios, menús, módulos interactivos, garantizando la consistencia funcional y visual a lo largo de la aplicación.

Desde la óptica científica y académica, el análisis de los componentes en React no puede quedar limitado a su mera descripción técnica. Este aspecto se debe considerar en el contexto de modelos arquitectónicos contemporáneos en arquitectura de software, interacción humano-computadora y experiencia de usuario. Por tanto, los componentes se deben ver como una estrategia de programación, pero también como un paradigma metodológico que cambia la manera en la que se diseñan, estructuran y mantienen las aplicaciones web modernas.

- Componentes Funcionales: Los componentes funcionales son funciones de JavaScript que devuelven elementos JSX. Se han vuelto prominentes debido a la introducción de los hooks en React, lo que les permite manejar el estado y efectos secundarios. Aquí hay un ejemplo simple:

Figura 20

Lenguaje de programación useState del complemento de funcionales

```
import React from 'react';

const Saludo = () => {
  return <h1>Hola, Mundo!</h1>;
};
```

Fuente: Elaboración propia
Fuente: https://brew.sh

- Componentes de Clase: Los componentes de clase son clases de JavaScript que extienden React.Component y han sido la forma tradicional de crear componentes en React.

Figura 21

Lenguaje de programación useState de la react componentes

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

class SaludoClase extends Component {
  render() {
    return <h1>Hola desde un Componente de Clase</h1>;
  }
}
```

Fuente: Elaboración propia
Fuente: https://brew.sh

Ventajas de los Componentes en React

- Reutilización de Código: Los componentes permiten encapsular funcionalidades, lo que facilita su reutilización en diferentes partes de la aplicación.
- Mantenimiento Simplificado: Dividir la interfaz de usuario y la lógica en componentes más pequeños facilita el mantenimiento y la comprensión del código.
- Desarrollo Eficiente: Los equipos pueden trabajar en diferentes componentes simultáneamente, agilizando el proceso de construcción de aplicaciones.
- Ciclo de Vida de los Componentes en React: Los componentes de clase tienen un ciclo de vida con métodos como `componentDidMount`, `componentDidUpdate` y `componentWillUnmount`, que permiten realizar acciones en diferentes etapas del ciclo de vida del componente.

Ejemplos de Componentes en React

- Componente de Lista en donde el componente toma una lista de elementos y los muestra como una lista no ordenada. Utiliza la función `map` para recorrer la lista (`elementos`) y crea un elemento de lista (`<li>`) por cada elemento. La clave (`key`) es necesaria para identificar cada elemento de la lista y optimizar su rendimiento.

Figura 22

Lenguaje de programación useState de los coponentes en React

```
import React from 'react';

const ListaComponent = ({ elementos }) => {
  return (
    <ul>
      {elementos.map((elemento, index) => (
        <li key={index}>{elemento}</li>
      ))}
    </ul>
  );
};
```

Fuente: Elaboración propia
Fuente: https://brew.sh

- Componente de Detalle: El componente de detalle muestra información detallada sobre un objeto específico. Recibe un objeto como prop (`**objeto**`) y muestra su contenido, como el título y la descripción. Este tipo de componente se usa comúnmente para visualizar detalles específicos en una interfaz de usuario.

Figura 23

Lenguaje de programación useState del componente de detalles

```
import React from 'react';

const DetalleComponent = ({ objeto }) => {
  return (
    <div>
      <h2>{objeto.titulo}</h2>
      <p>{objeto.descripcion}</p>
      {/* Otros detalles */}
    </div>
  );
};
```

Fuente: Elaboración propia
Fuente: https://brew.sh

- Formulario de Usuario: Este componente maneja un formulario para la entrada de datos de un usuario. Utiliza el hook `useState` para mantener el estado del usuario a medida que se ingresan los datos. La función `handleChange` actualiza el estado del usuario a

medida que se escriben los datos en los campos del formulario, y `handleSubmit` se activa cuando se envía el formulario.

Figura 24

Lenguaje de programación handleSubmit

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

const FormularioUsuario = () => {
  const [usuario, setUsuario] = useState({ nombre: '', email: '' });

  const handleChange = (event) => {
    setUsuario({ ...usuario, [event.target.name]: event.target.value });
  };

  const handleSubmit = (event) => {
    event.preventDefault();
    console.log('Datos de usuario:', usuario);
    // Aquí iría la lógica para enviar los datos a la API, por ejemplo
  };

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="text"
        name="nombre"
        value={usuario.nombre}
        onChange={handleChange}
        placeholder="Nombre"
      />
      <input
        type="email"
        name="email"
        value={usuario.email}
        onChange={handleChange}
        placeholder="Email"
      />
      <button type="submit">Enviar</button>
    </form>
  );
};
```

Fuente: Elaboración propia
Fuente: https://brew.sh

- Componente de Menú: Es un componente simple que representa un menú de navegación. Proporciona una lista de opciones de navegación, cada una vinculada a una ruta específica en la aplicación. Este tipo de componente es fundamental para la navegación entre secciones o páginas de una aplicación.

Figura 25

Lenguaje de programación useState componente de menú

```
import React from 'react';

const MenuComponent = () => {
  return (
    <nav>
      <ul>
        <li><a href="/">Inicio</a></li>
        <li><a href="/productos">Productos</a></li>
        <li><a href="/contacto">Contacto</a></li>
      </ul>
    </nav>
  );
};
```

Fuente: Elaboración propia
Fuente: https://brew.sh

- Componente de Encabezado El componente de encabezado muestra el título principal de la página, como el nombre de la aplicación, y puede incluir una descripción breve. Este componente es común en muchas páginas web y proporciona información importante y relevante para el usuario.

Figura 26

Lenguaje de programación useState componente de encabezado

```
import React from 'react';

const EncabezadoComponent = () => {
  return (
    <header>
      <h1>Nombre de la Aplicación</h1>
      <p>Bienvenido a nuestra aplicación</p>
    </header>
  );
};
```

Fuente: Elaboración propia
Fuente: https://brew.sh

- Componente de Pie de Página El componente de pie de página muestra información adicional, como los detalles de contacto, enlaces importantes, como la política de privacidad o los términos y condiciones. Es un componente crucial para proporcionar información adicional y enlaces de interés para el usuario.

Figura 27

Lenguaje de programación useState componentes de pie de pagina

```
import React from 'react';

const PiePaginaComponent = () => {
  return (
    <footer>
      <p>Información de contacto</p>
      <a href="/politica-de-privacidad">Política de privacidad</a>
    </footer>
  );
};
```

Fuente: Elaboración propia
Fuente: https://brew.sh

Estos ejemplos demuestran cómo los componentes en React permiten la construcción de aplicaciones modulares y reactivas, facilitando un desarrollo más eficiente y mantenible. La elección entre componentes funcionales y de clase dependerá de los requisitos del proyecto y las preferencias del desarrollador.

Uso de Props en React

Las props, abreviatura de propiedades, son un mecanismo esencial en React que permite la comunicación y la personalización de componentes. Estas propiedades son utilizadas para enviar datos desde un componente padre a sus componentes hijos. La información pasada a través de las props es de solo lectura, lo que significa que los componentes hijos no pueden modificar directamente las props que reciben.

Funcionamiento de las Props en React:

- Pasando Props: Las props se pasan como atributos a los componentes cuando se instancian. Por ejemplo:

<Componente nombre="Ejemplo" edad={25} />

- Recibiendo Props: En el componente receptor, las props se acceden a través del parámetro del componente. Por ejemplo:

```
const Componente = (props) => {
return <h1>Hola, {props.nombre}</h1>;
```

Para componentes de clase:

```
class Componente extends React.Component {
render() {
return <h1>Hola, {this.props.nombre}</h1>;
}
}
```

Tipos de Props: Las props pueden ser de cualquier tipo de dato, incluyendo strings, números, booleanos, funciones, objetos y arrays. Incluso es posible pasar componentes como props.

Personalización de Componentes con Props:

Las propiedades son una herramienta fundamental para adaptar los componentes, lo que los convierte en más útiles y reutilizables. Por ejemplo, piensa en un componente de botón que puede recibir propiedades para establecer su color, el texto que presenta o la función que se ejecuta al hacer clic (Enríquez, 2020).

Ejemplos de Uso de Props

- Componente Botón: En donde el ejemplo ilustra un componente de botón que emplea propiedades para determinar su apariencia y funcionamiento. Las propiedades incluyen `color`, `texto` y `onClick`. Gracias a estas propiedades, es posible ajustar el color de fondo del botón, el texto que se muestra y la acción que se realizará al hacer clic en él. Estas características son fundamentales para que el componente sea adaptable y se pueda utilizar en diversas circunstancias.

Figura 28

Aplicación de React uso de props

```
const Boton = ({ color, texto, onClick }) => {
  return (
    <button style={{ backgroundColor: color }} onClick={onClick}>
      {texto}
    </button>
  );
};

// Uso del componente Boton
<Boton color="blue" texto="Click aquí" onClick={() => console.log('Botón cl
```

Fuente: Elaboración propia
Fuente: https://brew.sh

- Componente de Tarjeta: El componente de Tarjeta muestra información estructurada, como un título, descripción e imagen. Las props `título`, `descripción` e `imagen` permiten personalizar el contenido de la tarjeta. Este tipo de componente es útil para mostrar contenido organizado y esencial en una interfaz de usuario.

Figura 29

Lenguaje de programación de React uso de props

```
const Tarjeta = ({ titulo, descripcion, imagen }) => {
  return (
    <div>
      <h2>{titulo}</h2>
      <p>{descripcion}</p>
      <img src={imagen} alt="Imagen de tarjeta" />
    </div>
  );
};

// Uso del componente Tarjeta
<Tarjeta
  titulo="Título de la tarjeta"
  descripcion="Descripción de la tarjeta"
  imagen="url_de_la_imagen.jpg"
/>
```

Fuente: Elaboración propia
Fuente: https://brew.sh

- Componente de Lista: El componente Lista muestra elementos en una lista no ordenada. Recibe una prop elementos que es un array, y mapea cada elemento del array a un elemento de lista (`**<li>**`). Esta flexibilidad en la gestión de props permite mostrar listas dinámicas de elementos.

Figura 30

Lenguaje de programación de React uso de props 2

```
const Lista = ({ elementos }) => {
  return (
    <ul>
      {elementos.map((elemento, index) => (
        <li key={index}>{elemento}</li>
      ))}
    </ul>
  );
};

// Uso del componente Lista
<Lista elementos={['Elemento 1', 'Elemento 2', 'Elemento 3']} />
```

Fuente: Elaboración propia
Fuente: https://brew.sh

Los ejemplos muestran cómo las props permiten la configuración dinámica y la personalización de componentes en React, lo que facilita la reutilización de componentes en diferentes partes de una aplicación y su adaptación a diferentes necesidades. La flexibilidad proporcionada por las props es fundamental en el diseño de componentes reutilizables y escalables.

Las Props como forma de comunicación entre componentes en React

Las Props constituyen un mecanismo de comunicación en React (properties) ya que permiten gestionar el traspaso de la información entre componentes de forma estructurada y controlada. Desde un enfoque de arquitectura del software, también representan los propios parámetros de configuración que permiten "hacer interactuar" los componentes padres e hijos de una aplicación web bajo la base de los modelos de programación declarativa y modular.

Según Alex Banks y Eve Porcello (2020), las Props redundan en una mayor reutilización y adaptabilidad de las componentes asociadas con la modificación de comportamientos y contenidos por la no modificación de la configuración interna de cada

componente dado. Este hecho contribuye a arquitecturas más robustas y sostenibles del desarrollo frontend actual.

Desde un enfoque metodológico, podemos establecer las Props como una manera de conseguir separar la presentación en sí de los datos de cada componente. Los agentes que reciben la información ejecutan las funciones de props (propiedades) para representar contenido dinámico, realizar acciones o alteraciones de comportamiento interactivos sin necesidad de realizar la gestión de la propia lógica global de la aplicación, de modo que se obtiene como resultado una mejor organización estructural de la funcionalidad en entornos digitales complejos.

Una dirección y control de datos

Uno de los principios más esenciales asociados con las props es el flujo de datos unidireccional. React entiende que la información se transmite de los componentes superiores hacia los hijos a partir de las props (propiedades) que no pueden ser leídas, lo que previene la ejecución de modificaciones directas en la información original y, a su vez, la facilita en materia de seguimiento y control de la información en la aplicación.

Desde el enfoque de ingeniería de software, esta estrategia ayuda a disminuir la posible aparición de errores provocados por una manipulación descontrolada de los estados compartidos, aumentado a la vez la estabilidad y la previsibilidad del sistema; además, el flujo unidireccional de los datos ayuda a depurar y mantener aplicaciones de medios interactivos que incorporan múltiples niveles visuales y actualizaciones dinámicas.

Flexibilidad estructural de las props

Las props pueden contener distintos tipos de datos. Se pueden englobar cadenas de texto, valores numéricos, estructuras booleanas, funciones, objetos, listas e incluso otros componentes. Esta flexibilidad permite construir interfaces especificadas y muy ajustadas a los requerimientos de los entornos digitales actuales.

Desde la independencia funcional, esta flexibilidad es especialmente importante en sistemas multimedia donde los propios componentes están en una constante adaptación a los contenidos, a los contextos de uso y a las necesidades de interacción. Por tanto, las props favorecen la creación de componentes reutilizables como aquellos que pueden ser configurados para adaptarse al entorno externo y a las necesidades de representación visual y comportamiento.

Props y reutilización de componentes

El uso de las props está en sintonía con principios de reutilización y modularidad en el ámbito del frontend. Componentes como botones, tarjetas, formularios o listas pueden alterar su aspecto, contenido y funcionalidad conforme a diferentes configuraciones de las props, evitando duplicaciones de código y, además, aumentando la eficiencia de la estructura del sistema.

Por ejemplo, un componente de botón puede reutilizarse en diferentes contextos variando propiedades relacionadas con colores, textos o acciones o interacciones. De la misma manera, componentes de tarjetas o de listas construyen contenidos dinámicos de múltiples maneras a partir de estructuras configurables dependiendo de unas fuentes de datos externas.

Desde una perspectiva académica, estos ejemplos no deben ser entendidos como unos simples procedimientos técnicos a seguir, sino como la propia evidencia de lo que React es capaz de implementar haciendo uso de principios contemporáneos de arquitectura modular, de diseño reutilizable y de interactividad dentro del desarrollo web contemporáneo.

Interpretación metodológica de las props en el diseño multimedia

En el ámbito del diseño multimedia, las props cobran especial significado porque permiten edificar interfaces adaptativas con capacidad para representar información variada mediante estructuras visuales homogéneas. Tal facultad propicia la posibilidad de construir sistemas escalables pensados para la experiencia del usuario, la accesibilidad y la actualización dinámica del contenido.

Asimismo, la utilización de props hace fortalecer los procesos de trabajo colaborativo en proyectos interdisciplinares, ya que diseñadores y desarrolladores pueden reutilizar componentes ya definidos y escritos sin verse obligados a cambiar la lógica estructural de la misma aplicación. De ahí que los tiempos de desarrollo se vean optimizados y que se garantice la coherencia visual y funcional de las diferentes secciones en el sistema digital.

Desde un enfoque científico, el estudio de las props en React deja ver la evolución de los modelos de desarrollo web hacia arquitecturas basadas en componentes reutilizables, en flujos controlados de la información o en sistemas declarativos de forma que hubiese eficiencia, mantenibilidad e interacción dinámica.

Capítulo 2.
Conceptos Avanzados

Virtual DOM

El Virtual DOM (VDOM) en React es un elemento fundamental que contribuye a optimizar el rendimiento y la eficacia al modificar las interfaces de usuario en aplicaciones web. Funciona como una versión ligera y virtual del DOM auténtico, proporcionando una estructura jerárquica que representa los componentes y su estado en la memoria.

Funcionamiento del Virtual DOM

- Representación en Memoria: El VDOM es una réplica del DOM real que se almacena en la memoria del navegador. Esta representación adopta la forma de un árbol que muestra la jerarquía de los elementos de la interfaz de usuario.
- Comparación y Detección de Cambios: Cuando un componente en React cambia su estado, primero se actualiza en el VDOM. React compara este VDOM con su versión previa para encontrar las diferencias, un procedimiento llamado reconciliación.
- Actualización Eficiente: Después de que se identifican las diferencias entre la versión actual y la anterior del VDOM, React decide los cambios mínimos que debe aplicar al DOM real para reflejar esas alteraciones. Esto permite que solo se actualicen las partes del DOM que han cambiado, evitando nuevas renderizaciones completas.

Ventajas del Virtual DOM

- Optimización del Rendimiento: Al disminuir las manipulaciones directas en el DOM real, el uso del VDOM en React facilita actualizaciones más eficientes, mejorando así el rendimiento total de la aplicación.
- Menos Operaciones en el DOM Real: La actualización específica del DOM real significa realizar menos operaciones de escritura, lo que disminuye los costos de rendimiento asociados con la manipulación directa del DOM.
- Facilita el Desarrollo: Proporciona a los programadores una representación en memoria más accesible para trabajar, lo que agiliza la actualización y gestión de la interfaz de usuario.

Uso del VDOM en React y Fiber Tree

El VDOM juega un papel crucial en React al optimizar las modificaciones del DOM. Por su parte, el algoritmo Fiber, también conocido como Fiber Tree, es una nueva

implementación interna del reconciliador de React que prioriza y organiza las tareas de renderizado y actualización de manera más eficiente, lo cual permite interrupciones y facilita el manejo de eventos del usuario o pausas durante el proceso de renderizado.

React une el VDOM con Fiber para obtener actualizaciones rápidas y mejoradas en la interfaz de usuario. Mientras que el VDOM funciona como una representación abstracta almacenada en la memoria, Fiber organiza y prioriza las tareas de actualización, haciendo el proceso más ágil y fluido. Esta técnica de combinar un Virtual DOM con Fiber Tree en React es fundamental para la eficiencia y reactividad que caracterizan a esta biblioteca. Estas herramientas son vitales para desarrollar interfaces web dinámicas y responsivas, optimizando tanto el rendimiento como la experiencia del usuario.

Renderizado condicional

El renderizado condicional en React es un método esencial para presentar diferentes elementos o componentes en función de ciertas condiciones lógicas. En donde se facilita la creación de interfaces dinámicas que se ajustan a variables, estados o propiedades concretas (Olawanle, 2025).

Métodos de Renderizado Condicional en React

- Operador Ternario: Se utiliza una expresión condicional para decidir qué componente renderizar.

  {condición? < Componente Verdadero /> : <ComponenteFalso / > }

- Declaraciones if: Pueden usarse dentro de la lógica del renderizado, aunque no directamente en el retorno.

Figura 31

Aplicación de React en la lógica del renderizado

```
render() {
  let componente;

  if (condicion) {
    componente = <ComponenteVerdadero />;
  } else {
    componente = <ComponenteFalso />;
  }

  return <div>{componente}</div>;
```

Fuente: Elaboración propia
Fuente: https://brew.sh

Uso de Estado para Renderizado Condicional El renderizado condicional en React se combina comúnmente con el estado para actualizar dinámicamente la interfaz de usuario en respuesta a cambios en los datos o interacciones del usuario.

Ejemplos de Renderizado Condicional

- Para mostrar un mensaje cuando una condición es verdadera, se utiliza un componente llamado `Mensaje`. Si la variable `mostrar Mensaje` es `true`, entonces el componente `Mensaje` se mostrará; de lo contrario, no aparecerá nada.

```
{mostrarMensaje ? <Mensaje /> : null}
```

- Renderizar un Componente si una Variable es Igual a un Valor Específico: Aquí, el `Componente` se muestra solamente si la `variable` variable es exactamente igual a `'valor'`. La evaluación de la expresión lógica cortocircuita la renderización del componente si la condición no se cumple.

```
{variable === 'valor' && <Componente />}
```

- Mostrar Contenido si una Lista no Está Vacía: Este ejemplo muestra el componente `Contenido Lista` únicamente si la lista `mi Lista` tiene elementos (es decir, si su longitud es mayor que cero). Si la lista está vacía, el componente no se renderizará.

```
{miLista.length > 0 && <ContenidoLista />}
```

- Alternar el Renderizado con un Botón: Al presionar el botón, se alterna el estado de `mostrar`. Si es `true`, el componente `Componente` se muestra; de lo contrario, si es `false`, el componente se oculta.

```
<button onClick={() => setMostrar(!mostrar)}>Alternar</button>
{mostrar && <Componente />}
```

- Renderizar Componentes en Base a Condiciones de Iteración: Este ejemplo muestra diferentes componentes (`Componente A` o `Componente B`) en función de una condición determinada (`condición`) para cada elemento de la lista `elementos`. Dependiendo de la condición, se renderizará un componente u otro.

```
{elementos.map((elemento) => (
  {condicion ? <ComponenteA /> : <ComponenteB />}
))}
```

- Manejar Estado y Mostrar un Componente Dinámicamente: Aquí, el estado `mostrar Componente` se alterna al presionar el botón. Si el estado es `true`, el componente se mostrará; de lo contrario, si es `false`, se ocultará.

```
const [mostrarComponente, setMostrarComponente] = useState(false);
<button onClick={() => setMostrarComponente(!mostrarComponente)}>
  Mostrar / Ocultar Componente
</button>
{mostrarComponente && <Componente />}
```

Cargar una imagen

Cargar imágenes en una aplicación de React es una tarea común y sencilla, brindando múltiples métodos para incorporar imágenes desde archivos locales o URLs externas. Se puede lograr mediante el uso de elementos `<img>` o a través de la importación directa en los componentes.

Cargar una Imagen desde un Archivo Local

- Utilizando el Elemento <img>: La carga de una imagen desde un archivo local es simple; basta con utilizar el elemento `<img>` en el componente y definir la ruta del archivo en el atributo `src`.

Figura 32

Aplicación de React en la cargar una imagen desde un archivo local

```
import React from 'react';
import MiImagen from './mi-imagen.jpg';

const App = () => {
  return (
    <div>
      <img src={MiImagen} alt="Mi Imagen" />
    </div>
  );
};

export default App;
```

Fuente: Elaboración propia
Fuente: https://brew.sh

- Método `import` (ideal para imágenes pequeñas): Otra opción es importar directamente la imagen, especialmente recomendado para imágenes pequeñas ya que se integran en el código compilado.

Figura 33

Lenguaje de programación de React import

```
import React from 'react';
import MiImagen from './mi-imagen.jpg';

const App = () => {
  return (
    <div>
      <img src={MiImagen} alt="Mi Imagen" />
    </div>
  );
};

export default App;
```

Fuente: Elaboración propia
Fuente: https://brew.sh

Cargar una Imagen desde una URL Externa: Integrar una imagen desde una URL externa se realiza de manera similar a la carga desde un archivo local. Se usa la URL directa de la imagen en el atributo `**src**`.

Figura 34

Lenguaje de programación de React SRC

```
import React from 'react';

const App = () => {
  return (
    <div>
      <img src="https://ruta-de-la-imagen.com/imagen.jpg" alt="Imagen Exter
    </div>
  );
};

export default App;
```

Fuente: Elaboración propia
Fuente: https://brew.sh

Consideraciones Importantes

- **Optimización de Imágenes:** Es crucial considerar el tamaño y la calidad de las imágenes para mejorar el rendimiento de la aplicación.
- **Manejo de Rendimiento:** La elección del método para cargar imágenes depende de los requisitos del proyecto y las necesidades específicas de rendimiento.

Uso del Hook `useRef` en React:

El hook `useRef` de React es una herramienta muy práctica que permite hacer referencia a elementos del DOM o almacenar valores que pueden cambiar dentro de componentes funcionales. Esta capacidad es especialmente útil para acceder a elementos del DOM, conservar datos que no necesitan causar re-renderizaciones y retener variables durante toda la vida del componente.

A continuación, presento seis ejemplos que ilustran cómo utilizar `useRef` en diversas circunstancias dentro de un componente funcional en React.

Referencia a un Elemento del DOM

En el estudio de caso, se emplea `useRef` para conseguir acceso a un elemento del DOM, en particular un campo de texto. El hook `useRef` establece una referencia (`inputRef`) al elemento correspondiente. Después, se utiliza `useEffect` con un arreglo de dependencias vacío, lo que indica que esta función se ejecutará únicamente una vez, al momento en que el componente se renderiza por primera vez. Dentro del efecto, se invoca a `inputRef. current. focus()` para que el campo de texto se enfoque automáticamente. Asimismo, el botón registra

y muestra en la consola el valor introducido en el campo de texto mediante `inputRef. current. value`.

Figura 35

Lenguaje de programación de React Elemento del DOM

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

const ElementoInput = () => {
  const inputRef = useRef(null);

  useEffect(() => {
    inputRef.current.focus(); // Enfocar el input al renderizar
  }, []);

  return (
    <div>
      <input ref={inputRef} type="text" />
      <button onClick={() => console.log(inputRef.current.value)}>
        Obtener Valor
      </button>
    </div>
  );
};
```

Fuente: Elaboración propia
Fuente: https://brew.sh

- **Manipulación de Estilos en Elementos del DOM**

Aquí se muestra cómo `useRef` puede utilizarse para modificar propiedades de estilo de un elemento. El `buttonRef` es la referencia al botón. Al hacer clic en el botón, se activa la función `cambiarColor`, la cual actualiza el color de fondo del botón utilizando `buttonRef.current.style.backgroundColor = 'red'`.

Figura 36

Lenguaje de programación de estilos elemento del DOM

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

const CambioEstilo = () => {
  const buttonRef = useRef(null);

  const cambiarColor = () => {
    buttonRef.current.style.backgroundColor = 'red';
  };

  return (
    <div>
      <button ref={buttonRef} onClick={cambiarColor}>
        Cambiar Estilo
      </button>
    </div>
  );
};
```

Fuente: Elaboración propia
Fuente: https://brew.sh

- **Almacenamiento de Datos Mutables**

En este caso, se usa `useRef` para almacenar un dato mutable que persiste entre re-renderizaciones. El efecto `useEffect` incrementa el valor almacenado en `dataRef.current` en una unidad al renderizar el componente.

Figura 37

Lenguaje de programación de React Elemento del DOM

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

const DatosMutables = () => {
  const dataRef = useRef(0);

  useEffect(() => {
    dataRef.current = dataRef.current + 1;
  }, []);

  return (
    <div>
      <p>Datos: {dataRef.current}</p>
      <button onClick={() => console.log(dataRef.current)}>
        Mostrar Datos
      </button>
    </div>
  );
};
```

Fuente: Elaboración propia
Fuente: https://brew.sh

- **Control de Foco en Múltiples Elementos**

El ejemplo demuestra cómo `useRef` puede ser usado para manejar múltiples inputs y cambiar el foco entre ellos. Cada `inputRef` es una referencia a un input. Al hacer clic en el botón, se activa la función `enfocarSiguienteInput`, que utiliza `inputRefs.current[index].current.focus()` para cambiar el foco al input siguiente.

Figura 38

Lenguaje de programación de foco de múltiples elementos

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

const VariosInputs = () => {
  const inputRefs = useRef([React.createRef(), React.createRef()]);

  const enfocarSiguienteInput = (index) => {
    inputRefs.current[index].current.focus();
  };

  return (
    <div>
      <input ref={inputRefs.current[0]} />
      <input ref={inputRefs.current[1]} />
      <button onClick={() => enfocarSiguienteInput(1)}>
        Enfocar Siguiente Input
      </button>
    </div>
  );
};
```

Fuente: Elaboración propia
Fuente: https://brew.sh

- **Almacenamiento de Referencias a Elementos Externos**

En este ejemplo se almacena una referencia a un elemento `<video>` usando `useRef`. Al hacer clic en el botón "Reproducir", la función `reproducirVideo` utiliza `videoRef.current.play()` para iniciar la reproducción del video.

Figura 39.

Lenguaje de programación de almacenamiento de referencia useRef

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

const VideoPlayer = () => {
  const videoRef = useRef(null);

  const reproducirVideo = () => {
    videoRef.current.play();
  };

  return (
    <div>
      <video ref={videoRef} src="video.mp4" />
      <button onClick={reproducirVideo}>Reproducir</button>
    </div>
  );
};
```

Fuente: Elaboración propia
Fuente: https://brew.sh

- **Almacenamiento de Callbacks de Eventos**

Aquí, `useRef` se emplea para guardar un callback como una referencia mutable. Al hacer clic en los botones, uno ejecuta el callback almacenado en `callbackRef.current`, mientras que el otro actualiza este callback con una nueva función.

Figura 40

Lenguaje de programación de callbackRef.current

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

const EventCallback = () => {
  const callbackRef = useRef(() => console.log('Callback por defecto'));

  const cambiarCallback = () => {
    callbackRef.current = () => console.log('Nuevo Callback');
  };

  return (
    <div>
      <button onClick={() => callbackRef.current()}>
        Ejecutar Callback
      </button>
      <button onClick={cambiarCallback}>
        Cambiar Callback
      </button>
    </div>
  );
};
```

Fuente: Elaboración propia
Fuente: https://brew.sh

Cada ejemplo ilustra una aplicación particular del hook useRef, ya sea para acceder y manipular elementos del DOM, mantener datos mutables, controlar el foco entre elementos o almacenar referencias a eventos y callbacks. Estos ejemplos detallados ayudan a comprender mejor cómo useRef puede ser utilizado en diversas situaciones dentro de componentes funcionales en React.

Estilos en React

El manejo de estilos en React es clave para que los componentes y aplicaciones luzcan bien. Hay varias maneras de aplicar estilos, desde métodos sencillos como los estilos en línea hasta el uso de bibliotecas y frameworks más sofisticados. A continuación, te presento seis enfoques comunes para gestionar estilos en React:

- **Estilos en línea con objetos JavaScript**

La aplicación de estilos en línea utilizando objetos JavaScript es una forma directa y simple de establecer estilos en un componente. Se definen los estilos como un objeto y se asignan al componente usando el atributo **`style`.**

Figura 41

Lenguaje de programación de JavaScript

```
import React from 'react';

const miEstilo = {
  color: 'blue',
  fontSize: '16px',
  // Otros estilos CSS
};

const App = () => {
  return (
    <div style={miEstilo}>
      <h1>Estilos en React</h1>
      <p>Ejemplo de estilos en línea.</p>
    </div>
  );
};

export default App;
```

Fuente: Elaboración propia
Fuente: https://brew.sh

- **Estilos con CSS en Módulos**

Los módulos de estilos CSS permiten definir estilos en archivos separados y luego aplicarlos a componentes específicos mediante el uso de clases.

Figura 42

Lenguaje de programación con CSS en Módulos

```
/* Estilos.module.css */
.miClase {
  color: red;
  font-size: 18px;
  /* Otras reglas de estilo */
}
```

```
import React from 'react';
import estilos from './Estilos.module.css';

const App = () => {
  return (
    <div className={estilos.miClase}>
      <h1>Estilos con CSS en módulos</h1>
      <p>Utilizando estilos modulares para componentes.</p>
    </div>
  );
};

export default App;
```

Fuente: Elaboración propia
Fuente: https://brew.sh

- **Uso de Frameworks de Estilos**

Integrar frameworks como Bootstrap, Material-UI o Tailwind CSS proporciona componentes con estilos predefinidos, permitiendo una rápida implementación y consistencia visual.

- **Librerías de Estilización en JS**

Bibliotecas como styled-components o Emotion permiten escribir estilos directamente dentro de archivos de componentes mediante JavaScript o TypeScript.

Figura 43

Lenguaje de programación con TypeScript.

```
import styled from 'styled-components';

const MiComponente = styled.div`
  color: green;
  font-size: 20px;
  /* Otras reglas de estilo */
`;
```

Fuente: Elaboración propia
Fuente: https://brew.sh

- **Uso de Preprocesadores CSS**

El empleo de preprocesadores CSS como Sass o Less permite la manipulación de variables, la creación de estructuras anidadas y otras funcionalidades que facilitan la elaboración y organización de estilos.

- **Estilos Globales**

En la implementación de estilos de manera global, se pueden utilizar archivos CSS que afecten a todo el proyecto o la propiedad créate Global Style de ciertas bibliotecas de estilización en JavaScript.

En donde las opciones ofrecen versatilidad y diferentes métodos para aplicar estilos a aplicaciones en React, permitiendo que los desarrolladores seleccionen la opción más adecuada según las necesidades del proyecto y sus preferencias individuales.

Gestión de estilos en React y su relación con la arquitectura de interfaces digitales

La administración de estilos en React es crucial para el diseño de interfaces digitales, ya que influye en la apariencia visual, la experiencia del usuario y la funcionalidad coherente de las aplicaciones web. Desde un enfoque metodológico, la estilización abarca más que el aspecto visual; desempeña un papel importante en áreas como la accesibilidad, la jerarquía visual, el mantenimiento y la capacidad de adaptación en diversas plataformas (Banks y Porcello, 2020).

Así, las distintas opciones de estilización que React proporciona están en sintonía con necesidades específicas relacionadas con la organización del trabajo, la escalabilidad y el modularidad en los proyectos de frontend actuales. Por lo tanto, la elección de un sistema de estilos se basa en tanto criterios técnicos como en las exigencias vinculadas a la experiencia del usuario, la estructura del sistema y la colaboración entre profesionales de diferentes disciplinas (Stefanov, 2016).

- **Estilos en línea y control dinámico de la interfaz**

En relación con los estilos en línea y el control dinámico de la interfaz, el uso de estilos en línea mediante objetos de JavaScript permite incorporar propiedades visuales a elementos específicos. Esto favorece un control dinámico de la interfaz, ya que los estilos se pueden ajustar de acuerdo con el estado de los componentes o la interacción del usuario.

La visión funcional indica que los estilos en línea son bastante apropiados para interfaces que necesitan un elevado grado de interacción, así como en sistemas multimedia

donde los elementos visuales deben ser modificados de forma constante y en tiempo real. No obstante, diversas investigaciones sobre la mantenibilidad del software han señalado que un uso excesivo de estilos incrustados puede resultar en una reducción de la reutilización y un aumento de la complejidad del código en aplicaciones de gran tamaño (Fain, 2018).

- **CSS Modules y encapsulamiento visual.**

Los CSS Modules son un método que se enfoca en proteger los estilos dentro de componentes, evitando así problemas que pueden causar el uso de clases globales y promoviendo un modularidad visual en los sistemas.

Desde la óptica de la arquitectura frontend, los CSS Modules refuerzan la separación de componentes y ayudan a gestionar aplicaciones complejas de manera más efectiva. Además, este método favorece la colaboración entre diseñadores y desarrolladores, ya que organiza los estilos de manera estructurada y coherente en proyectos multimedia que requieren escalabilidad (Banks y Porcello, 2020).

Frameworks de estilos y consistencia visual

El uso de frameworks como Bootstrap, Material UI o Tailwind CSS permite beneficiarse de componentes y sistemas visuales ya existentes, lo que mejora el proceso de desarrollo en el frontend. Desde un punto de vista metodológico, estos frameworks garantizan una cohesión visual, accesibilidad y adaptabilidad responsive (en diferentes aspectos) en las interfaces digitales.

Según Luke Wroblewski (2011), sistemas visuales consistentes facilitan la navegación y disminuyen la carga cognitiva. Sin embargo, varios autores advierten que depender en exceso de estilos preconfigurados puede restringir la personalización visual y uniformar la identidad gráfica de los sistemas digitales (Tidwell, Brewer y Valencia, 2020).

Librerías de estilización en JavaScript

Las bibliotecas como styled-components y Emotion permiten incorporar modelos de estilización basadas en objetos JavaScript, por lo que se puede definir la estilización directamente dentro de los componentes.

En el estudio se permite reforzar la integración entre la lógica funcional y la representativa, favoreciendo arquitecturas más dinámicas y reutilizables. Desde una perspectiva de desarrollo multimedia, estas herramientas contribuyen a la construcción de

interfaces de usuario adaptativas y componentes visuales altamente configurables en función de sus estados, propiedades y comportamientos interactivos (Purdy, 2020).

Uso de preprocesadores CSS

Existen tecnologías como Sass o Less que permiten extender las funcionalidades básicas del CSS tradicional gracias a la posibilidad de usar variables, de realizar anidamientos o implementar estructuras reutilizables.

Los preprocesadores resultan de ayuda a la ingeniaría de software para mejorar la organización estructural y el escalado de hojas de estilos complejas. Por otro lado, permiten también consolidar la instauración de sistemas visuales más homogéneos y hacen más sencillo el mantenimiento de multiplataforma multimedia, donde se tienen que tener en cuenta tantos componentes, el estilo de los mismos, sus variaciones (Keith, 2017).

Estilos globales y coherencia sistémica

Los estilos globales persiguen un tratamiento importante para la instauración de la identidad de la localización visual y de la coherencia sistémica dentro de aplicaciones digitales concretas. Su uso permite instaurar tipografías, colores, espaciados, o comportamientos visuales comunes a la mayor parte de los componentes de la aplicación.

stilísticas globales, desde el punto de vista del diseño multimedia, favorecen la construcción de sistemas visuales integrados y reforzan la coherencia perceptiva que se produce durante la interacción del usuario. Tal y como indica Jenifer Tidwell et al. (2020), la coherencia visual es un elemento relevante que habilita el reconocimiento, la navegación y la previsión de interfaces complejas.

Interpretación metodológica de los sistemas de estilos en React

Los diferentes enfoques o técnicas para estilizar en React no tienen que ser consideradas como alternativas técnicas de implementación de una característica visual. Cada enfoque tiene que ser considerado como la representación de un modelo concreto de organización del software, administración de interfaces o de la construcción de experiencias digitales.

Desde el punto de vista académico, el análisis de estilos de React permite observar la correlación entre la arquitectura frontend, la experiencia de usuario y el diseño visual interactivo. El mismo también permite notar que los sistemas de desarrollo web contemporáneos incluyen también principios como el del modularidad, la reutilización o la

adaptabilidad en entornos multimedia definidos por la continua interacción del usuario (Banks & Porcello, 2020; Stefanov, 2016).

Styled Components

Styled Components es una destacada biblioteca en el entorno de React que revoluciona la gestión de estilos al permitir la creación de componentes con estilos directamente integrados en JavaScript. Esta metodología brinda una forma eficiente y flexible de manejar la presentación visual de la interfaz de usuario.

Características Clave de Styled Components:

- Estilos como Componentes: En lugar de definir estilos en archivos CSS separados, Styled Components utiliza componentes de JavaScript para describir estilos. Esto ofrece una integración directa entre el diseño y la lógica de los componentes.
- Alcance Local de Estilos: Cada componente estilizado con Styled Components genera estilos únicos que se aplican solo al componente específico, evitando conflictos o efectos de cascada global.
- Personalización Dinámica: Los estilos pueden ajustarse en función de las props, el estado o temas, lo que permite crear componentes dinámicos y reutilizables.

Ejemplos de Uso de Styled Components:

- Estilizando un Componente Button:

En este ejemplo, se define un componente `Button` utilizando Styled Components. Los estilos del botón se establecen directamente en el template de la plantilla de la función `styled`.

Se define el color de fondo, el color del texto, el relleno, el borde, el radio de borde y el comportamiento de `hover`.

El `&:hover` se utiliza para modificar el fondo del botón cuando el cursor se sitúa sobre él.

Figura 44

Lenguaje de programación con Styled Components

```
import styled from 'styled-components';

const Button = styled.button`
  background-color: #3498db;
  color: white;
  padding: 8px 16px;
  border: none;
  border-radius: 4px;
  cursor: pointer;

  &:hover {
    background-color: #2980b9;
  }
`;
```

Fuente: Elaboración propia
Fuente: https://brew.sh

Creación de un Componente de Encabezado: Aquí se crea un componente `Header` usando Styled Components, que representa un encabezado (`<h1>`). Los estilos definidos incluyen un tamaño de fuente de 24 píxeles y un color de texto #333.

Figura 45

Lenguaje de programación con Styled Components header

```
const Header = styled.h1`
  font-size: 24px;
  color: #333;
`;
```

Fuente: Elaboración propia
Fuente: https://brew.sh

Aplicando Estilos a un Div: Se crea el componente `Container` utilizando Styled Components para representar un contenedor `<div>`. Los estilos aplicados incluyen un color de fondo #f5f5f5, un relleno de 20 píxeles y un radio de borde de 4 píxeles.

Figura 46

Lenguaje de programación con utilizando Styled

```
const Container = styled.div`
  background-color: #f5f5f5;
  padding: 20px;
  border-radius: 4px;
`;
```

Fuente: Elaboración propia
Fuente: https://brew.sh

Personalización Dinámica de un Componente: Aquí, el componente `Card` permite una personalización dinámica basada en las props. Si la prop `primary` es `true`, el fondo del componente será azul con texto blanco; de lo contrario, el fondo será blanco con texto negro.

Figura 47

Lenguaje de programación con Componentes Card

```
const Card = styled.div`
  background-color: ${props => props.primary ? '#3498db' : '#ffffff'};
  color: ${props => props.primary ? '#ffffff' : '#333'};
`;
```

Fuente: Elaboración propia
Fuente: https://brew.sh

Estilizando un Componente Link: El componente Link define estilos para un enlace (`<a>`) con un color de texto específico y sin decoración por defecto. Al pasar el cursor sobre el enlace, se subraya el texto (`text-decoration: underline`) gracias a `:hover`.

Figura 48

Lenguaje de programación con Hover

```
const Link = styled.a`
  color: #3498db;
  text-decoration: none;

  &:hover {
    text-decoration: underline;
  }
`;
```

Fuente: Elaboración propia
Fuente: https://brew.sh

Aplicando Estilos Globales: Se crea un componente `GlobalStyle` que permite aplicar estilos a nivel global. En este caso, se establece la fuente para el cuerpo del documento y el color de fondo para toda la página.

Figura 49

Lenguaje de programación con GlobalStyle

```
import { createGlobalStyle } from 'styled-components';

export const GlobalStyle = createGlobalStyle`
  body {
    font-family: 'Arial', sans-serif;
    background-color: #f5f5f5;
  }
`;
```

Fuente: Elaboración propia
Fuente: https://brew.sh

Valores por defecto: Los valores por defecto en React son fundamentales para garantizar la coherencia y funcionalidad de los componentes, permitiendo que estos operen sin problemas incluso cuando no reciben ciertos valores o si los valores proporcionados son nulos o indefinidos.

Valores por Defecto en Componentes Funcionales: En componentes funcionales de React, asignar valores por defecto a las props es una práctica común. Por ejemplo:

Figura 50

Lenguaje de programación con React, asignar valores por defecto a las props

```
import React from 'react';

const MiComponente = ({ nombre, edad = 18 }) => {
  return (
    <div>
      <p>Nombre: {nombre}</p>
      <p>Edad: {edad}</p>
    </div>
  );
};

export default MiComponente;
```

Fuente: Elaboración propia
Fuente: https://brew.sh

En el ejemplo, la prop `edad` se establece en 18 si no se proporciona al componente `MiComponente`. Valores por Defecto en Clases de Componentes: En componentes de clase, se pueden asignar valores por defecto utilizando el método `constructor` o la propiedad `defaultProps`.

Figura 51

Lenguaje de programación con defaultProps

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

class MiComponente extends Component {
  constructor(props) {
    super(props);
    this.state = {
      nombre: props.nombre || 'Usuario',
      edad: props.edad || 18
    };
  }

  render() {
    return (
      <div>
        <p>Nombre: {this.state.nombre}</p>
        <p>Edad: {this.state.edad}</p>
      </div>
    );
  }
}

MiComponente.defaultProps = {
  nombre: 'Usuario',
  edad: 18
};

export default MiComponente;
```

Fuente: Elaboración propia
Fuente: https://brew.sh

Aquí, se asignan valores por defecto para nombre y edad en caso de que no se proporcionen.

Ventajas del uso de Valores por Defecto:

- **Robustez:** Evitan errores cuando faltan valores esperados.
- **Flexibilidad:** Permite un comportamiento predecible en ausencia de ciertos datos.

La utilización de valores por defecto en React es esencial para asegurar la consistencia y el correcto funcionamiento de los componentes, proporcionando una experiencia más estable y predecible para los usuarios.

Capítulo 3.
Funcionalidades y Aplicaciones Prácticas

React Router

R React Router es una poderosa herramienta dentro del entorno de React, diseñada para manejar la navegación en aplicaciones del tipo Single Page Application (SPA). Su objetivo principal es facilitar las transiciones entre diversas vistas y componentes, evitando la necesidad de cargar de nuevo toda la aplicación y ofreciendo así interacciones más fluidas y continuas (Banks & Porcello, 2020)

Desde una perspectiva técnica, React Router adopta un método de enrutamiento que se basa en el historial de navegación, lo que permite asociar rutas específicas a componentes particulares dentro de la aplicación. En donde los resulta en una organización eficaz de sistemas frontend más complicados, así como una gestión efectiva de la interacción entre las distintas partes de la interfaz de usuario (Sanchez, 2025).

En el ámbito del desarrollo web hoy en día, la importancia de React Router se incrementa a medida que emergen más aplicaciones interactivas que se enfocan en la experiencia del usuario y en la actualización dinámica de los contenidos. Según Stoyan Stefanov (2016), los métodos de navegación que se centran en los componentes y el renderizado dinámico son aspectos esenciales para optimizar el rendimiento y la continuidad visual de las aplicaciones web contemporáneas.

Además, React Router está profundamente relacionado con el respeto a ciertos principios de usabilidad y organización de la información. Esta navegación sin interrupciones entre diferentes vistas contribuye a crear modelos mentales claros para el usuario, al tiempo que se evita la frustración causada por las recargas completas de la página. De este modo, este enfoque mejora la rapidez, continuidad y estabilidad en las interacciones digitales.

Desde una perspectiva metodológica, React Router también facilita el manejo de aplicaciones que están modularizadas y que cuentan con rutas organizadas en una jerarquía. Esto hace que sea más sencillo crear sistemas escalables donde distintos componentes y vistas se integren de manera sencilla en arquitecturas frontend complejas. Por ende, React Router no es simplemente una herramienta técnica para la navegación, sino que se convierte en un cimiento esencial para la experiencia del usuario, la organización de contenidos y el diseño interactivo de las aplicaciones web actuales.

Principales Conceptos de React Router

Declaratividad en el Enrutamiento:

React Router hace uso de una estructura declarativa para definir las rutas y las correspondencias con los componentes que se deben renderizar.

- Componentes Esenciales:
- BrowserRouter: Componente envolvente que proporciona funcionalidad de enrutamiento.
- Route: Define una ruta y especifica el componente a renderizar para esa ruta.
- Link/NavLink: Componentes que facilitan la navegación entre diferentes rutas.
- Manejo de Rutas Dinámicas:

Es posible trabajar con rutas dinámicas que permiten recibir parámetros en la URL para reflejar información específica.

- Control de Navegación:
- Switch: Se encarga de renderizar solo la primera ruta que coincide con la URL actual.
- Redirect: Redirige a una ruta diferente dependiendo de condiciones específicas.

Implementación Básica

En el ejemplo ilustra cómo utilizar React Router para el enrutamiento en una aplicación de React:

Figura 52

Lenguaje de programación con React Router

```
import { BrowserRouter as Router, Route, Switch, Link } from 'react
import Home from './components/Home';
import About from './components/About';
import Contact from './components/Contact';

const App = () => {
  return (
    <Router>
      <div>
        <nav>
          <ul>
            <li>
              <Link to="/">Home</Link>
            </li>
            <li>
              <Link to="/about">About</Link>
            </li>
            <li>
              <Link to="/contact">Contact</Link>
            </li>
          </ul>
        </nav>

        <Switch>
          <Route path="/about">
            <About />
          </Route>
          <Route path="/contact">
            <Contact />
          </Route>
          <Route path="/">
            <Home />
          </Route>
        </Switch>
      </div>
    </Router>
  );
};

export default App;
```

Fuente: Elaboración propia
Fuente: https://brew.sh

En el estudio básico presenta un uso elemental de React Router en el cual se definen rutas y se asocian a componentes específicos. Al hacer clic en los enlaces de navegación, se renderizan los componentes correspondientes a esas rutas, creando así una transición fluida entre las diferentes secciones de la aplicación.

Recogida de parámetros

La obtención de parámetros utilizando React Router es una tarea fundamental para el enrutamiento adaptable en sitios web. Esta función facilita el envío de datos específicos a través del enlace, lo que simplifica la exhibición de información particular y ajusta la experiencia del usuario.

Recogida de Parámetros con React Router:

- Definición de Rutas con Parámetros: Al definir rutas con React Router, se pueden utilizar parámetros en la URL indicándolos con: seguido del nombre del parámetro en la ruta.

  Ejemplo

  ```
  <Route path="/article/:id" component={ArticleDetails} />
  ```

  En este caso, `**:id**` es un parámetro de ruta que puede tomar distintos valores, como `**/article/123**`, donde `**123**` es el valor del parámetro id.

- Acceso a los Parámetros: El hook useParams de React Router se utiliza para acceder a los parámetros pasados a una ruta dinámica.

Figura 53

Utilización de Parámetros

```
import { useParams } from 'react-router-dom';

const ArticleDetails = () => {
  const { id } = useParams();

  return (
    <div>
      <h2>Detalles del Artículo {id}</h2>
      {/* Mostrar detalles del artículo basado en 'id' */}
    </div>
  );
};
```

Fuente: Elaboración propia
Fuente: https://brew.sh

El objeto retornado por useParams captura los parámetros pasados en la URL. En este caso, `id` representa el valor del parámetro`:id` definido en la ruta. Ejemplos de Uso:

- E-commerce: Visualizar detalles específicos de un producto al acceder a la URL de un artículo.
- Redes Sociales: Ver la página de perfil de un usuario al hacer clic en su identificador único en la URL.
- Blogs: Acceder a una publicación específica al navegar a su URL única.

La capacidad de recoger y utilizar parámetros en las rutas es esencial para construir aplicaciones dinámicas, permitiendo la personalización de las vistas y facilitando la visualización de datos específicos basados en valores proporcionados en la URL.

Redirección desde React

La redirección en una aplicación de React es una característica esencial que permite dirigir dinámicamente a los usuarios a diferentes rutas o páginas en respuesta a ciertas condiciones, eventos o acciones específicas. Esta funcionalidad, comúnmente utilizada con React Router, mejora la experiencia del usuario al proporcionar un flujo de navegación fluido y adaptativo.

Redirección con React: Uso de `<Redirect>` con React Router:

El componente `<Redirect>` de React Router permite redirigir a los usuarios a una ruta específica.

Figura 54

Lenguaje de programación con Redirect

```
import { Redirect } from 'react-router-dom';

const PrivateRoute = ({ isAuthenticated }) => {
  if (!isAuthenticated) {
    return <Redirect to="/login" />;
  }

  return (
    // Contenido protegido
    <div>
      {/* ... */}
    </div>
  );
};
```

Fuente: Elaboración propia
Fuente: https://brew.sh

- Empleo del Hook `useHistory`: El hook `useHistory` facilita la redirección programática en respuesta a eventos o acciones específicas.

Figura 55

Lenguaje de programación con el useHistor

```
import { useHistory } from 'react-router-dom';

const Component = () => {
  const history = useHistory();

  const redirectToAboutPage = () => {
    history.push('/about');
  };

  return (
    <button onClick={redirectToAboutPage}>Ir a la página Acerca de</button>
  );
};
```

Fuente: Elaboración propia
Fuente: https://brew.sh

Situaciones de Uso:

- **Autenticación:** Redirigir a la página de inicio de sesión si el usuario no está autenticado.
- **Navegación Post-Formulario:** Redireccionar después de enviar un formulario, por ejemplo, a una página de confirmación.
- **Manejo de Errores:** Redirigir a una página de error personalizada tras un fallo en la aplicación.

La capacidad de redirección en una aplicación de React es crucial para proporcionar una navegación coherente y receptiva a lo largo de la experiencia del usuario. Esta funcionalidad permite responder dinámicamente a diferentes situaciones, mejorando así la usabilidad y el flujo dentro de la aplicación.

Configurar un Layout Copy

En el desarrollo de aplicaciones realizadas mediante este marco de programación, la preparación de layouts que puedan ser reutilizados en distintas pantallas, configurando adecuadamente los elementos visuales y funcionales de la interfaz de usuario, es una de las estrategias a seguir. Desde un punto de vista de arquitectura del frontend, el layout viene a

definirse como una forma estructural de organizar componentes de navegación, contenido, distribución en el espacio y otros elementos interactivos que están presentes en una aplicación digital (Banks & Porcello, 2020).

El término "Layout Copy" no representa un término técnico utilizado y reconocido ampliamente a nivel académico, ni a nivel de desarrollo de software.Por esta razón, resulta más adecuado emplear expresiones tales como "layouts reutilizables", "estructuras de interfaz reutilizables" o "arquitectura de layouts", porque describen de una forma más precisa la acción organizativa y estructural que pueden tener los layouts en aplicaciones web modernas.

En cuanto a su mirada metodológica, los layouts reutilizables también refuerzan la imagen de la consistencia visual, la estabilidad de la navegación y la coherencia funcional a lo largo de diversas vistas de una aplicación. De acuerdo con Jenifer Tidwell, Charles Brewer y Aynne Valencia (2020), los patrones consistentes de organización visual no sólo ayudan a enmarcar la interfaz que construyen, sino que también la convierten en algo más comprensible y aportan una menor carga cognitiva en el transcurso de las interacciones.

En aplicaciones React, los layouts suelen elaborarse a partir de componentes reutilizables que son capaces de encapsular las estructuras comunes para la navegación; aquellos encabezados, barras laterales, áreas de contenido y pie de páginas. En este sentido, la utilización de layouts reutilizables refuerzan principios de modularidad y reutilización y permite compartir configuraciones visuales y funcionales en diferentes vistas sin replicar código innecesariamente.

En otro sentido, la conformación de layouts es un caso en el que se une a su vez principios de la experiencia de usuario y de la arquitectura de la información. Una adecuada organización de los elementos visuales facilita la orientación espacial del usuario, incrementa la accesibilidad y beneficia procesos de navegación en sistemas digitales complejos (Garrett, 2011).

Desde el contexto del diseño multimedia y del desarrollo frontend actual, los layouts reutilizables quedan también perfectamente enlazados con la optimización de los procesos colaborativos entre diseñadores y desarrolladores. La definición de estructuras base de layouts compartidos crea sistemas visuales escalables y metódicamente coherentes, garantizando, en consecuencia, una coherencia perceptiva en diferentes dispositivos y resoluciones de pantalla.

Proceso de Configuración de un Layout Copy en React:

- **Consonantización**

Dividir la interfaz de usuario en componentes lógicos para facilitar su gestión y reutilización.

Figura 56

Lenguaje de programación de Layout Copy en React

```
// App.js
import Header from './components/Header';
import MainContent from './components/MainContent';
import Footer from './components/Footer';

const App = () => {
  return (
    <div>
      <Header />
      <MainContent />
      <Footer />
    </div>
  );
};
export default App;
```

Fuente: Elaboración propia
Fuente: https://brew.sh

- **Estilización con CSS o Librerías de Estilos**

Utilizar hojas de estilo o librerías como Styled Components para aplicar reglas de diseño que mantengan la coherencia visual en todos los componentes.

Figura 57

Lenguaje de programación de Styled Components

```
// Header.js
import styled from 'styled-components';

const Header = styled.header`
  /* Estilos para el header */
`;

// ... Código adicional para el componente Header
```

Fuente: Elaboración propia
Fuente: https://brew.sh

- Jerarquía de Componentes: Organizar los componentes de manera jerárquica para facilitar su reutilización y mantenimiento.
- Gestión de Estados y Datos: Integrar React Hooks o herramientas como Redux para manejar los estados y la lógica de la interfaz de usuario.

Situaciones de Uso en Aplicaciones React:

- **Diseño Web y Desarrollo de Aplicaciones:** Configurar la estructura de un sitio web o aplicación web, manteniendo la coherencia visual y funcional.
- **Aplicaciones Móviles:** Utilizar la misma estrategia para mantener la uniformidad en el diseño y la usabilidad.

La configuración de un Layout Copy en React busca crear una experiencia de usuario fluida y consistente al estructurar los elementos visuales y funcionales de la interfaz. Este enfoque mejora la mantenibilidad y escalabilidad de la aplicación React.

Enrutar a un enrutamiento

El enrutamiento en aplicaciones React es esencial para la navegación entre diversas secciones y páginas de la interfaz de usuario. Con React Router, se puede implementar un sistema de enrutamiento que permite a los usuarios desplazarse por distintas partes de la aplicación de manera dinámica y fluida.

Enrutar con React Router

Componentes y Rutas

En React Router, se establecen rutas asociadas a componentes específicos que se renderizan cuando la URL coincide con la ruta.

Figura 58

Ejemplo de definición de ruta

```
import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';
import Home from './components/Home';
import About from './components/About';
import Contact from './components/Contact';

const App = () => {
  return (
    <Router>
      <Switch>
        <Route path="/" exact component={Home} />
        <Route path="/about" component={About} />
        <Route path="/contact" component={Contact} />
      </Switch>
    </Router>
  );
};
```

Fuente: Elaboración propia
Fuente: https://brew.sh

- **Navegación entre Rutas:**

Para permitir que los usuarios se desplacen entre diferentes secciones, se utilizan componentes como `**<Link>**` para establecer enlaces a las rutas definidas.

Figura 59

Ejemplo de navegación entre rutas

```
import { Link } from 'react-router-dom';

const Navigation = () => {
  return (
    <div>
      <ul>
        <li>
          <Link to="/">Home</Link>
        </li>
        <li>
          <Link to="/about">About</Link>
        </li>
        <li>
          <Link to="/contact">Contact</Link>
        </li>
      </ul>
    </div>
  );
};
```

Fuente: Elaboración propia
Fuente: https://brew.sh

Casos de Uso en Aplicaciones React

- Navegación entre Secciones: Permitir a los usuarios desplazarse entre diferentes páginas o componentes de la aplicación.
- Gestión de Contenido Dinámico: Mostrar contenido relevante basado en la URL

La implementación de enrutamiento con React Router proporciona una manera eficaz de organizar y presentar la interfaz de usuario, ofreciendo una experiencia de navegación coherente y agradable a los usuarios. Esta funcionalidad es esencial para aplicaciones web complejas que requieren una navegación fluida y dinámica.

UseEffect

Definición

Al useEffect es un hook esencial en React, destinado a manejar efectos secundarios en los componentes funcionales. Estos efectos pueden abarcar una amplia gama de acciones, como la manipulación del DOM, operaciones asíncronas, suscripciones a eventos o incluso la limpieza de recursos (Viera, 2024).

Uso de useEffect en React

Gestión de Actualizaciones Basadas en Dependencias: `useEffect` permite ejecutar código en respuesta a cambios específicos, asegurando que el componente se actualice cuando sea necesario.

Figura 60

Ejemplo de actualización basada en dependencia

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

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

  useEffect(() => {
    document.title = `Clicked ${count} times`;
  }, [count]);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>Click me</button>
    </div>
```

Fuente: Elaboración propia
Fuente: https://brew.sh

- **Manejo de Recursos Asíncronos**

El `useEffect` es idóneo para operaciones asíncronas, como solicitudes a una API o una base de datos.

Figura 61

Ejemplo de solicitud de datos

```
useEffect(() => {
  const fetchData = async () => {
    const result = await axios.get('https://api.example.com/data');
    setData(result.data);
  };

  fetchData();
}, []);
```

Fuente: Elaboración propia
Fuente: https://brew.sh

- **Limpieza de Efectos**

El retorno de una función desde useEffect permite realizar tareas de limpieza, como liberar recursos.

Figura 62

Ejemplo de limpieza

```
useEffect(() => {
  // Lógica de inicialización

  return () => {
    // Limpieza: liberación de recursos
  };
}, []);
```

Fuente: Elaboración propia
Fuente: https://brew.sh

Casos de Uso Comunes:

- Gestión de Suscripciones y Eventos: Iniciar o finalizar suscripciones a eventos.
- Solicitudes de Datos: Obtener información de servicios externos.
- Integración con APIs o Bases de Datos: Manejar la lógica de datos.

El `useEffect` es un mecanismo versátil que facilita la gestión de tareas asíncronas, la manipulación del DOM y otros efectos secundarios en componentes funcionales de React. Su flexibilidad permite el control preciso de la lógica de la aplicación, garantizando un flujo de trabajo más eficiente y organizado.

El Hook useEffect y el ciclo de vida de React

El hook useEffect es fundamental en React para controlar los efectos secundarios dentro de los componentes funcionales. En términos sencillos, un efecto secundario hace referencia a cualquier actividad que implique algo más que simplemente mostrar el componente, como podría ser realizar una petición HTTP, establecer un temporizador, modificar el DOM, suscribirse a algo o utilizar un servicio externo (Banks y Porcello, 2020).

La incorporación de useEffect marca un cambio importante en los componentes funcionales, ya que permite manejar comportamientos que previamente estaban asociados al ciclo de vida de los componentes de clase, pero lo hace de una forma más clara y concisa. Según Facebook (2019), este hook ayuda a organizar de manera lógica los procesos asincrónicos y las interacciones externas en las aplicaciones reactivas modernas.

Relación entre useEffect y el ciclo de vida del componente

Desde una concepción significativa, el useEffect actúa como un puente que enlaza las modificaciones en el estado y el renderizado con la ejecución de los procesos secundarios de manera controlada. Su funcionamiento depende de las dependencias definidas en el interior del Hook para determinar cuándo se tiene que ejecutar el efecto que se relaciona con la actualización del componente.

Este modelo se vincula de forma directa con el ciclo de vida de los componentes funcionales. Cuando un componente se ejecuta por primera vez en su renderizado, el useEffect podría llegar a ejecutar procesos equivalentes a la fase de montaje. De igual forma, cuando determinadas dependencias cambian, el Hook vuelve a ejecutar la lógica referida en el uso anterior, tendiendo así a ejecutar procesos que simulan aquella fase de actualización. Por último, a través de las funciones de limpieza, el useEffect permite gestionar las tareas que simulan la fase de desmontaje del componente.

Desde una concepción metodológica, esta capacidad refuerza la organización del flujo lógico de aplicaciones interactivas y permitir la gestión eficiente de recursos en sistemas multimedia de tipo dinámico.

Gestionar actualizaciones a partir de dependencias

Uno de los usos más interesantes que tiene el useEffect hace referencia a la ejecución controlada de lógica en función de determinadas modificaciones del estado o de las propiedades del mismo componente. Este mecanismo permite mejorar el comportamiento de la interfaz al no realizar ejecuciones disfuncionales, ya que se mejora la computación.

El uso de las dependencias en aplicaciones multimedia o en interfaces interactivas permite coordinar la actualización de datos, los cambios visuales y los procesos de interacción reactiva en tiempo real. Por ello, el useEffect facilita la mantenencia de la coherencia funcional entre la representación visual y el estado interno de la aplicación.

Manejo de operaciones asíncronas

El Hook useEffect adquiere un papel importante en el tratamiento de operaciones asíncronas, como llamadas a APIs, apertura de conexiones con bases de datos o recuperación de información desde ubicaciones remotas. Este tipo de operaciones presentan un papel relevante dentro de las aplicaciones web modernas, que son por definición aplicaciones que permiten la modificación dinámica del contenido y la sincronización constante de datos externos.

Desde el punto de vista de la experiencia del usuario, la apropiada implementación de operaciones asíncronas favorecerá la continuidad de la percepción y la capacidad de respuesta del sistema. En este sentido, y siguiendo las ideas de Jakob Nielsen (1994), los tiempos de respuesta y el trato adecuado de la información son aspectos clave para propiciar la percepción de fluidez y estabilidad en el marco de las interfaces digitales.Funciones de limpieza y control de recursos

Otra característica esencial de useEffect corresponde a la posibilidad de devolver funciones de limpieza dedicadas a liberar recursos o a cancelar procesos cuyos inicios hayan sido previamente ejecutados. Este mecanismo resulta ser de especial significancia en operaciones que se asocian a temporizadores, suscripciones a eventos, conexiones persistentes, etc.

En una presencia computacional, ya que permiten controlar la aparición de fugas de memoria, de procesos en redundancia y de comportamientos inesperados fruto de múltiples ejecuciones que puedan darse simultáneamente, las funciones de limpieza son las responsables de propiciar estabilidad y eficiencia operativa dentro de aplicaciones React especialmente complejas.

Interpretación metodológica de useEffect en aplicaciones funcionales

Los ejemplos que tienen que ver con la actualización basada en dependencias, con las peticiones de datos o con la limpieza de recursos no deben ser considerados solamente como procedimientos técnicos, sino que cada uno de ellos pone de manifiesto la forma en la que useEffect articula cómo puede establecerse la interrelación entre el estado, el renderizado y el comportamiento dinámico dentro de aplicaciones funcionales.

Desde un enfoque académico, useEffect refleja un mecanismo de control del flujo interactivo que permite controlar procesos de actualización, de sincronización y de manejo de recursos en sistemas digitales contemporáneos; y su mismo uso ilustra la forma en la que React hace suya una serie de principios emanados de la programación reactiva, de la modularidad o de la gestión cuidadosa de las interfaces dinámicas en el desarrollo frontend contemporáneo.

Es por esto que el uso de useEffect es de interés no solamente dentro del terreno técnico, sino también en lo que a arquitectura de software, interacción humano-computador y experiencia de usuario se refiere dentro de las aplicaciones multimedia interactivas.

Fetch API y Axios

La Fetch API y Axios son dos herramientas esenciales para realizar solicitudes HTTP en aplicaciones React, siendo fundamentales para interactuar con servidores y obtener o enviar datos desde y hacia diferentes fuentes.

- **Fetch API:**

La Fetch API es una característica nativa de JavaScript que permite realizar peticiones HTTP asincrónicas. Proporciona una interfaz moderna para solicitar recursos a través de la red, utilizando el método fetch.

Figura 63

Ejemplo de solicitud GET con Fetch API

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

Fuente: Elaboración propia
Fuente: https://brew.sh

Axios

Axios, por otro lado, es una librería externa ampliamente utilizada para manejar solicitudes HTTP en navegadores y entornos Node.js. Ofrece una interfaz simple y funcionalidades avanzadas.

Figura 64

Ejemplo de solicitud GET con Axios

```
import axios from 'axios';

axios.get('https://api.example.com/data')
  .then(response => {
    console.log(response.data);
  })
  .catch(error => {
    console.error('Error:', error);
  });
```

Fuente: Elaboración propia
Fuente: https://brew.sh

Diferencias y Ventajas

- Intaxis y Usabilidad: La Fetch API utiliza el método fetch de forma nativa, mientras que Axios ofrece una sintaxis más clara y fácil de usar.
- Compatibilidad: La Fetch API está disponible en la mayoría de los navegadores modernos, mientras que Axios brinda soporte en una amplia gama de entornos, incluyendo versiones más antiguas de navegadores.
- Funcionalidades Adicionales: Axios proporciona características avanzadas, como la cancelación de solicitudes y la conversión automática de datos de respuesta, lo que simplifica la gestión de errores y el manejo de datos.

Casos de Uso

- Fetch API: Es útil para proyectos sencillos o en escenarios donde se requiere una solución nativa y moderna.
- Axios: Perfecto para aplicaciones más complejas que necesitan manejar solicitudes HTTP con funcionalidades avanzadas y amplia compatibilidad.

En resumen, la elección entre la Fetch API y Axios depende de las necesidades específicas del proyecto. Mientras que la Fetch API ofrece una solución nativa y moderna, Axios brinda características avanzadas y una sintaxis más clara, facilitando la gestión de solicitudes HTTP en aplicaciones React.

Fetch API y Axios en la gestión de solicitudes HTTP en React

Gestión de las solicitudes HTTPHay que tener en consideración que la gestión de las solicitudes HTTP es un aspecto fundamental en el desarrollo de aplicaciones que funcionan bajo React; esto se debe a que permite la comunicación entre la interfaz de usuario y servicios externos, servidores u otras bases de datos. En aplicaciones web de última generación, la recuperación, el envío y la actualización de la información dependen de la implementación de mecanismos asincrónicos que permitan que se gestionen flujos de datos dinámicos en tiempo real.

Debido a ello, la Fetch API y Axios son dos de las herramientas más usadas que se utilizan para gestionar las solicitudes HTTP en aplicaciones de frontend. Ambas tecnologías permiten la implementación de procesos de comunicación entre el cliente y el servidor, pero existen algunas características que establecen las diferencias entre una y la otra acerca de la arquitectura, de la funcionalidad y de los modelos de gestión de datos.Fetch API como mecanismo nativo de comunicación asíncrona

La Fetch API corresponde a una interfaz nativa de JavaScript orientada a gestionar solicitudes HTTP de forma asíncrona por medio del método fetch. La inclusión de esta funcionalidad en navegadores modernos supone todo un avance respecto a los mecanismos tradicionales, como el XMLHttpRequest, dado que incorpora un modelo basado en promesas más estructurado y flexible (Flanagan, 2020).

Desde un punto de vista estrictamente técnico, Fetch API permite realizar la recuperación a distancia de recursos y gestionar respuestas de forma asíncrona sin bloquear el hilo principal en el que se ejecuta la aplicación; esta característica es especialmente importante en aplicaciones multimedia e interactivas que requieran de una actualización constante de contenidos sin recargar por completo la página.

Del mismo modo, el uso de Fetch API fomenta los modelos de desarrollo centrados en la interoperabilidad y la estandarización, puesto que es una solución integrada al propio entorno JavaScript moderno. No obstante, algunos estudios consideran que la gestión manual de errores y las transformaciones de respuesta pueden llegar a aumentar la complejidad en aplicaciones a gran escala (Banks y Porcello, 2020).

Axios y la abstracción avanzada de solicitudes HTTP

Axios corresponde a una librería externa que tiene por objetivo simplificar la gestión de las solicitudes HTTP, tanto en navegadores como en entornos Node.js. A diferencia de Fetch API, Axios incorpora funcionalidades avanzadas orientadas a facilitar la gestión de respuestas, la gestión de errores y la transformación automática de datos.

Desde el punto de vista metodológico, Axios disminuye la complejidad en la operación mediante una sintaxis estructurada y por mecanismos predefinidos de configuración; entre estas funcionalidades se encuentran los interceptores de las peticiones, la cancelación de las mismas, la gestión automática de respuestas en formato JSON o el control centralizado de errores. Así pues, estas características favorecen la construcción de aplicaciones escalables en la construcción de sistemas frontend donde la interacción con los datos es muy alta.

Para Alex Banks y Eve Porcello (2020), librerías como Axios favorecen la optimización de la organización lógica y la mantenibilidad de aplicaciones React que requieren de comunicarse constantemente con servicios externos y arquitecturas distribuidas.

Comparación entre Fetch API y Axios

Las diferencias permitidas entre Fetch API y Axios no sólo hay que considerarlas de una forma sintáctica o como un distinto nivel de facilidad de uso, sino que también hay que tenerlas en cuenta en términos de arquitectura y mantenibilidad del sistema.

Desde el punto de vista de la compatibilidad, Fetch API depende en lo principal de la funcionalidad nativa del navegador, mientras que Axios asegura una solución de integración más amplia en todo tipo de entornos y versiones. A su vez, Axios incluye mecanismos de transformación y de gestión de errores automáticos que simplifican el trabajo de desarrollo en aplicaciones más complejas.

Fetch API se aproxima más a una solución ligera y alineada con estándares nativos del JavaScript moderno, algo que puede ser apropiado para aquellos proyectos que no son muy complejos estructuralmente o de disminuida dependencia.Vinculación con aplicaciones multimedia e interfaces dinámicas

En el marco del estado actual del diseño multimedia y el desarrollo frontend, tanto Fetch API como Axios tienen una importancia fundamental para los procesos de actualización dinámica de contenidos, sincronización de datos y diseño de experiencias interactivas. Aplicaciones tales como plataformas de enseñanza y aprendizaje, sistemas multimedia, paneles

interactivos y herramientas de trabajo colaborativo se basan en un uso de la comunicación de tipo asíncrono muy eficiente en términos de garantizar la continuidad de la funcionalidad y una buena capacidad de respuesta.

Desde la óptica del usuario del sistema, la correcta gestión de las solicitudes HTTP afecta de forma directa a los tiempos de respuesta, la percepción de fluidez y la estabilidad de las interacciones. De acuerdo con Jakob Nielsen (1994), la velocidad y continuidad de la interacción son factores determinantes para la percepción de calidad en las interfaces digitales.

Interpretación metodológica de la utilización de Fetch API y Axios

Los ejemplos propuestos acerca de las solicitudes GET, de la gestión de datos y del consumo de APIs no han de ser interpretados como procedimientos operativos, sino como formas de comprensión de los mecanismos de arquitectura frontend contemporánea y programación reactiva.

De este modo, el análisis de Fetch API y Axios permite comprender cómo las aplicaciones React dan sentido a los flujos de comunicación distribuida, la actualización dinámica de contenidos y la gestión eficaz de datos en los entornos digitales interactivos y multimedia.

Storage con Zustand

La combinación del almacenamiento con Zustand hace posible que los datos del estado se conserven en el almacenamiento del navegador, como `localStorage` o `sessionStorage`. Esta función resulta muy valiosa para mantener el estado de la aplicación durante las diferentes sesiones del usuario.

Procedimiento para la Persistencia de Datos con Zustand

- Zustand presenta una función denominada `persist` que facilita la conservación de datos en el almacenamiento del navegador. A continuación, se muestra un ejemplo de su uso:

- Configuración del Store en Zustand:

Figura 65

Supongamos que tienes un store básico en Zustand para manejar el estado del usuario

```
import create from 'zustand';

const useStore = create(set => ({
  user: null,
  setUser: newUser => set({ user: newUser }),
}));
```

Fuente: Elaboración propia
Fuente: https://brew.sh

- Habilitar el Almacenamiento con Zustand:

Para permitir la persistencia de datos, se usa la función `persist` proporcionada por Zustand. Esta función necesita algunos parámetros:

Figura 66

Almacenamiento con Zustand

```
import { persist } from 'zustand/middleware';

const useStore = create(persist(
  set => ({
    user: null,
    setUser: newUser => set({ user: newUser }),
  }),
  {
    name: 'myPersistedStore', // Nombre del store
    getStorage: () => sessionStorage, // Tipo de almacenamiento: sessionStorage o localStorage
  }
));
```

Fuente: Elaboración propia
Fuente: https://brew.sh

Ventajas de la Persistencia con Zustand:

- Persistencia de Datos: Esta función permite que la aplicación conserve su estado incluso después de que el usuario cierre y vuelva a abrir el navegador.
- Facilidad de Uso: Facilita la configuración para guardar y recuperar información del almacenamiento del navegador.

Consideraciones Importantes:

- Seguridad de los Datos: La información sensible debe ser tratada con cuidado cuando se almacena en el navegador.

- Capacidad de Almacenamiento: Grandes volúmenes de datos pueden influir en el rendimiento de la aplicación.

Implementación Eficiente:

Asegúrate de importar la función `persist` desde `zustand/middleware` para incorporar la persistencia de datos en tu aplicación. Además, revisa la documentación oficial de Zustand para explorar opciones avanzadas y obtener más información sobre la persistencia de datos.

Gestión de estado y persistencia de datos con Zustand

Zustand se puede entender como una biblioteca para el manejo del estado de aplicaciones React. La biblioteca se basa en principios de simplicidad, modularidad y bajo nivel de abstracción y orientaciones explícitas al estado global a través de un enfoque ligero que delega las operaciones de administración con flexibilidad.

Al contrario que otras bibliotecas de gestión de estado como Redux, Zustand elimina la complejidad estructural que comporta una gran cantidad de configuraciones, reducers obligatorios, y patrones altamente ceremoniales. Según Poimandres (2023), Zustand se construyó para proporcionar un modelo minimalista de administración de estado que ofrecía un modelo de gestión de estado basado en Hooks con stores individuales, promoviendo así un rendimiento y mantenibilidad en aplicaciones React modernas.Desde una óptica metodológica, Zustand se alinea con la necesidad de gestionar estados compartidos en aplicaciones frente a un frontend caracterizado por interactividad, actualizaciones constantes de datos y sincronización de varios componentes, favoreciendo arquitecturas más simples y escalables en el marco del desarrollo multimedia contemporáneo.

Persistencia de datos mediante almacenamiento del navegador

Una de las cuestiones más importantes de Zustand hace referencia a su capacidad para integrar mecanismos de persistencia de estados mediante almacenamiento local del navegador, esto es, localStorage y sessionStorage, lo cual permite obtener información incluso después del cierre o de la recarga de la aplicación.

La persistencia de datos se implementa mediante el middleware persist, que forma parte del módulo zustand/middleware, permitiendo obtener automáticamente el estado global y posteriormente recuperar este mismo estado de cara a futuras sesiones que realice el usuario.

Desde un punto de vista funcional, esta capacidad resulta especialmente clara en el ámbito de las aplicaciones multimedia e interfaces interactivas en las que se requieran

conservar preferencias del usuario, configuraciones visuales, sesiones activas o datos temporales asociados a procesos de interacción.

Relación entre persistencia y experiencia de usuario

La persistencia del estado afecta directamente a la continuidad de la experiencia dentro de aplicaciones digitales. Según Jakob Nielsen (1994), los sistemas interactivos han de mostrarse coherentes y necesarios para la interacción a efecto de reducir la carga cognitiva y por tanto mejorar la percepción de estabilidad.

En este marco, Zustand puede facilitar mejores experiencias al mantener los estados entre sesiones y evitar reconfiguraciones por parte del usuario; la persistencia deviene, así, un medio para mejorar usabilidad, accesibilidad y percepción de personalización en las aplicaciones web contemporáneas.

Características internas de Zustand

Desde el punto de vista técnico, Zustand también puede considerarse como un modelo de stores globales independientes, donde estos stores son administrados mediante Hooks, evitando así la compleja necesidad de proveedores (Provider) o extensas jerarquías de componentes, lo cual puede resultar en un acoplamiento entre componentes y un modularidad arquitectónico.

De igual forma, Zustand utiliza un modelo de actualizaciones de estado mediante suscripciones selectivas, de manera que solo se rendericen de nuevo los componentes que se ven afectados. Este tipo de optimización, tal y como comentan Alex Banks y Eve Porcello (2020), resulta especialmente interesante en aplicaciones React que buscan un alto rendimiento y/o que están siempre interactivas.

Ventajas comparativas respecto a otras soluciones de gestión de estado

En cuanto a bibliotecas de gestión de estado más tradicionales, Zustand también tiene varias ventajas, en términos de simplicidad de implementación, reducción de código repetido o facilidad para integrarse con Hooks de React.Si bien Redux sigue arquitecturas más pesadas y centralizadas, Zustand se apoya en un enfoque puramente minimalista en clave de flexibilidad y velocidad de desarrollo. Esta diferencia de enfoque hace que Zustand sea especialmente apto para proyectos multimedia, aplicaciones interactivas y sistemas frontend con necesidades de actualización muy dinámicas.

De igual forma, Zustand tiene también una curva de aprendizaje más plana y una complejidad de configuración menor que otras soluciones de gestión global de estado. Como resultado, favorece los procesos de desarrollo colaborativo y la mantenibilidad de aplicaciones de tamaño medio y grande.+

Consideraciones técnicas y de seguridad

No obstante, los beneficios funcionales que se obtienen, la persistencia de datos en local-storage exige criterios muy específicos en relación con la seguridad y la performance. La información sensible no ha de escribirse sin mecanismos adecuados de seguridad, puesto que los datos almacenados en el navegador están absolutamente disponibles en el contexto del cliente.

Del mismo modo, el almacenamiento excesivo de información genera problemas de performance y carga de la aplicación. Por esto diferentes autores (Flanagan, 2020) ya advierten en favor de contener el alcance de la persistencia a los estados caducados, así como organizar adecuadamente los procesos de serialización y de lectura de los datos.

Interpretación metodológica de Zustand en aplicaciones React

Los ejemplos de persistencia de estado y configuración de stores no se deben entender sólo como procesos técnicos discretos. Desde cierta perspectiva académica, Zustand sería un cambio dentro de los modelos actuales de gestión del estado orientados a una simplicidad arquitectónica, al modularidad y a una optimización de la experiencia del usuario.

En consecuencia, el análisis de Zustand ha de permitir comprender cómo las aplicaciones React introducen mecanismos de persistencia, de sincronización y de gestión global de la información en las aplicaciones digitales de naturaleza dinámica e interactiva.

Redux en React

Redux es una librería ampliamente utilizada para gestionar el estado global en aplicaciones JavaScript, especialmente en aquellas construidas con React. Proporciona un contenedor predecible del estado de la aplicación, facilitando el desarrollo de aplicaciones predecibles, escalables y fáciles de probar.

Conceptos Clave en Redux:

- Store: El store es un contenedor que almacena el estado global de la aplicación en un solo objeto.
- Acciones: Las acciones son objetos que describen los cambios que se desean realizar en el estado. Estas acciones son despachadas hacia los reducers.
- Reducers: Los reducers son funciones puras que toman el estado actual y una acción, y devuelven un nuevo estado basado en la acción recibida.
- Inmutabilidad: Redux enfatiza la inmutabilidad del estado. En lugar de modificar el estado directamente, se generan nuevos estados en respuesta a las acciones.

Utilización de Redux en Aplicaciones React:

- Estado Global: Redux es eficaz para aplicaciones con múltiples componentes que comparten el mismo estado.
- Depuración Sencilla: Ofrece un flujo de datos unidireccional, facilitando la identificación y corrección de errores.
- Historial de Acciones: Redux proporciona un historial claro de todas las acciones realizadas, lo que permite retroceder y avanzar a través de ellas.

Implementación Práctica:

La integración de Redux en React se realiza generalmente con la librería `react-redux`, la cual ofrece herramientas como `Provider`, `connect` y hooks personalizados como `useSelector` y `useDispatch` para acceder al estado global y despachar acciones.

Figura 67

Configuración del Store de Redux

```
import { createStore } from 'redux';

const counterReducer = (state = { count: 0 }, action) => {
  if (action.type === 'INCREMENT') {
    return { count: state.count + 1 };
  }
  return state;
};

const store = createStore(counterReducer);
```

Fuente: Elaboración propia
Fuente: https://brew.sh

Figura 68

Uso en Componentes React

```
import { useSelector, useDispatch } from 'react-redux';

function CounterComponent() {
  const count = useSelector(state => state.count);
  const dispatch = useDispatch();

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => dispatch({ type: 'INCREMENT' })}>Incrementa
    </div>
  );
}
```

Fuente: Elaboración propia
Fuente: https://brew.sh

En los estudios se ilustran la configuración básica de Redux, cómo se define un reducer para cambiar el estado y cómo se conectan los componentes para acceder al estado global y despachar acciones.

Redux es una herramienta poderosa que ofrece control sobre el estado global de la aplicación. Sin embargo, su complejidad puede ser excesiva para aplicaciones más simples. Es importante evaluar las necesidades específicas de la aplicación antes de decidir su implementación.

Redux y la arquitectura de la gestión del estado en las aplicaciones React

Redux forma parte de las arquitecturas más destacadas en el desarrollo de aplicaciones modernas en JavaScript, y en particular en aquellas que utilizan React. Su principal objetivo es centralizar el estado de la aplicación y gestionarlo mediante un modelo predecible que utiliza el flujo de datos unidireccional como base.

Redux fue creado en el año 2015 de la mano de Dan Abramov y Andrew Clark, y con inspiración en principios de arquitecturas funcionales como Flux y Elm. Según palabras de Abramov (2018), Redux fue creado para abordar problemas vinculados a la sincronización de estados complejos, la trazabilidad de los cambios y la mantenibilidad de aplicaciones frontend escalables.

Desde la perspectiva metodológica, Redux no tiene una concepción estrictamente como biblioteca de almacenamiento de datos, sino como una arquitectura de gestión del estado diseñada para que garantice predictibilidad, inmutabilidad y control estructurado de los cambios en aplicaciones interactivas.Principios estructurales de Redux

Redux basa su funcionamiento en tres principios arquitectónicos: la existencia de un solo lugar de verdad (un solo lugar de verdad, en inglés), la inmutabilidad de los estados y la actualización mediante funciones puras. Gracias a estos principios se pueden construir aplicaciones en las que los estados pueden ser investigados, reproducidos y depurados de un modo controlado.

El principio de "una sóla fuente de verdad" es el que establece que todo el estado global de la aplicación permanece almacenado en un único store centralizado. Desde la ingeniería de software, esta estrategia hace que la consistencia estructural sea sencilla y, además, la sincronización entre múltiples terminales de la interfaz sea bastante fácil.

Por otra parte, Redux plantea un modelo de inmutabilidad en el que los estados no se modifican directamente. En vez de modificar los datos anteriores, cada cambio da lugar a un nuevo estado.De acuerdo a Eric Elliott (2017), este patrón mejora previsibilidad, hace más fácil la depuración y minimiza errores debidos a mutaciones no deseadas de datos. En resumen, Redux llega a implementar reducers como funciones puras que son las encargadas de calcular un nuevo estado con base en el estado anterior y en una acción dada. Este modelo propicia la lógica modular y hace que la separación de responsabilidades sea incluso más evidente en aplicaciones complicadas.

Store, acciones y reducers como arquitectura funcional

El store es el centro de fidelidad de Redux, y funciona como un contenedor global del estado de la aplicación. Desde el punto de vista de la arquitectura, el store permite coordinar la información compartida entre distintos componentes, y no hace falta utilizar largas cadenas de props o estados locales fragmentados.

Las acciones (actions) son objetos descriptivos que representan eventos o intenciones de cambio dentro del sistema, y su trabajo es comunicar qué transformación se tiene que realizar en el estado global. En este sentido, se mejora la trazabilidad y la transparencia de los flujos de datos.

Los reducers son, por su parte, lógica funcional orientada a producir nuevos estados sin modificar directamente el estado anterior. Este modelo se compensa con principios de programación funcional, y permite obtener comportamiento determinista dentro de la aplicación.

Desde una perspectiva particular de la metodología, la interacción que se va produciendo entre el store, las acciones y los reducers es un modelo de flujo de datos en una única dirección que permite tener un control estructurado y la operativa coherente de sistemas frontend grandes y complejos.

Redux y la importancia de gestionar aplicaciones escalables

Redux juega un papel importante en todos los proyectos React que tengan varios componentes con dependencias entre sí, que sincronicen estados y que tengan una actualización dinámica de la información. En este sentido, Alex Banks y Eve Porcello (2020) sugieren que Redux es muy idóneo cuando en las distintas zonas que componen la interfaz se accede de manera simultánea a la misma información.

Siguiendo el enfoque del diseño multimedia y del desarrollo frontend actuales, Redux ayuda a soportar la administración de datos para la autenticación, la navegación, el contenido multimedia, los formularios complejos y los sistemas colaborativos para ayudar a garantizar la estabilidad y la mantenibilidad de aplicaciones que contienen interacciones para una gran escala.

Flujo unidireccional y depuración del sistema

Un aspecto de Redux que también se debe mencionar es el del flujo unidireccional. En este apartado, las acciones son enviadas hacia el store, son procesadas por los reducers y son mostradas en la interfaz a partir del renderizado reactivo.

Desde el plano computacional, este hecho podría sugerir un sistema que para la trazabilidad y la depuración del sistema mismo es más favorable. Redux incluye herramientas que permiten registrar el historial de las acciones y los cambios en el estado y reproducir las secuencias de interacción como si de un video se tratara, en donde por lo tanto se pueden registrar los errores de forma más ajustada.

En palabras de Martin Fowler (2018), los sistemas que contienen arquitecturas de flujo de datos explícitos favorecen la mantenibilidad y la disminución de la complejidad de los sistemas distribuidos y de las aplicaciones altamente interactivas.

React-Redux y la relación con componentes funcionales

La relación de Redux dentro de una aplicación React suele ser mediante una biblioteca que se llama React Redux, la cual ofrece elementos como Provider, useSelector y useDispatch para poder llamar al estado global y enviar las acciones desde componentes funcionales.

Desde la realidad técnica, React-Redux permite optimizar la relación entre la interfaz y el store global mediante suscripciones selectivas y un renderizado correcto de los componentes, con la cual cosa se optimiza, así, el rendimiento y escalable de aplicaciones frontend actuales.

Límites y valoración crítica de Redux

Siendo un patrón arquitectónico está claro que Redux no está exento de complicaciones vinculadas a la complejidad estructural, ni tampoco en el volumen de código necesario para conseguir determinadas implementaciones sencillas. Contracorriente, varios autores describen que la adopción de Redux para desarrollo en aplicaciones con un bajo volumen de funcionalidades o sistemas con un pequeño nivel de complejidad de estado, no es más que una opción superflua.

Por lo tanto, la decisión de adoptar Redux tiene que ser capaz de responder a las direcciones vinculadas a la escalabilidad y la necesidad de un modelo de sincronización global, así como de una complejidad funcional del propio proyecto. Recientemente, opciones como Zustand, Context API o Recoil constituyen alternativas quizás más livianas para determinadas circunstancias de la fase de desarrollo frontend.

En un marco académico, el análisis de Redux permite poner de manifiesto de qué forma principios de programación como son la programación funcional, la inmutabilidad de las estructuras de datos y el flujo controlado de los datos, tienen una influencia percibible en arquitecturas modernas de aplicaciones web interactivas y sistemas multimedia complejos.

Capítulo 4.
Desarrollo de Aplicaciones Específicas

El diseño de aplicaciones específicas en React es una de las áreas más trascendentes en el desarrollo frontend contemporáneo, debido a que conjuga principios de la arquitectura de software, la experiencia de usuario y el diseño interactivo en ambientes digitales complejos. Durante la actual evolución del desarrollo web, las aplicaciones desarrolladas con React tienen

por características la alta capacidad de reutilización de componentes, la actualización dinámica de las interfaces o la posibilidad de escalar la arquitectura, que son imprescindibles a la hora de desarrollar sistemas digitales orientados a procesos de interacciones continuas y de procesos de información en tiempo real (Banks y Porcello, 2020).

Desde la perspectiva metodológica, el diseño de aplicaciones específicas requiere articular fundamentos técnicos con criterios relacionados con aspectos relacionados con la usabilidad, la accesibilidad y la experiencia de usuario. En el estudio de Norman (2013) las interfaces digitales deben construirse en función de principios de claridad, retroalimentación y coherencia de la interacción, a fin de promover procesos de navegación y comprensión por parte del usuario.

En este contexto, las aplicaciones de comercio electrónico representan uno de los escenarios más representativos para la aplicación de las modernas tecnologías frontend. El comercio electrónico ha transformado los tradicionales modelos de interacción comercial mediante plataformas digitales que han sido diseñadas en función de la navegación interactiva, la gestión dinámica de productos, los sistemas de pago o la personalización de la experiencia de compra. Según la Organisation for Economic Co-operation and Development (OECD, 2019), el crecimiento de las plataformas digitales de comercio electrónico se relaciona de manera directa con la expansión de las tecnologías web orientadas a la interacción y la arquitectura centrada en la experiencia de usuario.

Así pues, sería conveniente reformular el título del contenido del epígrafe como:

Diseño e implementación de una aplicación de comercio electrónico con React

La denominación presenta un enfoque mucho más técnico y académico en el sentido en que delimita bien el objeto de estudio e integra el desarrollo aplicado con unos fundamentos metodológicos y tecnológicos en concreto.

Desde la aproximación de la ingeniería del software, el diseño de una aplicación de comercio electrónico con React consiste en la estructuración de componentes reutilizables, manejo de estados globales, navegación dinámica y optimizar el flujo de interacción entre el usuario y el sistema, incluyendo la inclusión de principios de UX/UI desde el manejo de la accesibilidad, el flujo de navegación y la capacidad de operar a escala en el entorno de la plataforma digital.

Siguiendo a Jesse James Garrett (2011), el diseño de la experiencia de usuario en las aplicaciones digitales ha de considerar, de forma conjunta, la arquitectura de la información, el

diseño visual, la interacción y la funcionalidad, con el objetivo de asegurar la coherencia estructural y la facilidad de uso. En las plataformas de comercio electrónico, todos estos factores adquieren especial relevancia porque son los que determinan, de manera directa y significativa, la confianza, satisfacción y conducta de compra del usuario.

De igual modo, la construcción de la arquitectura que sustenta un componente React favorece la creación de interfaces modulares y escalables, porque permite la separación de funcionalidades relacionadas con catálogos de productos, la autenticación, la gestión del carrito de compras y el procesamiento de pedidos. De acuerdo a Stoyan Stefanov (2016), los modelos de desarrollo de una aplicación frontend compleja basados en componentes aumentan la mantenibilidad, la reutilización y la escalabilidad de las mismas.

Por otra parte, el uso de herramientas complementarias como React Router, Redux o Zustand permite la gestión de la navegación y la persistencia de los datos, así como la sincronización de los estados globales entre las aplicaciones de comercio electrónico que se caracterizan por ser altamente interactivas y de contenidos continuamente cambiantes.

De una forma general, el desarrollo de aplicaciones específicas mediante React no se reduce a los procedimientos técnicos de su implementación, sino que debe integrar una serie de fundamentos derivados de los principios de ingeniería de software, de interacción humano-computador, de experiencia del usuario o de arquitectura frontend contemporánea. La propia integración metodológica que permite fortalecer el rigor científico de este capítulo también permite establecer el contexto tecnológico de desarrollo en relación con problemas reales vinculados con el diseño y funcionamiento de plataformas digitales interactivas.

Aplicación tienda online – por pasos

Paso 1: Configuración del Proyecto

- **Inicializa el Proyecto:** Utilizamos create-react-app para configurar un nuevo proyecto.

```
npx create-react-app mi-tienda-online
```

- **Estructura del Proyecto:** Organiza la estructura de archivos y carpetas. Define una estructura clara para los componentes, estilos, imágenes, y archivos de datos (si es necesario).

Paso 2: Diseño y Componentes

- Creación de la Interfaz de Usuario: Crea un diseño llamativo para tu tienda en línea. Establece las diversas páginas, los elementos y la forma en que se desarrollará la aplicación.
- Elementos de React: Divide tu diseño en partes de React. Por ejemplo: `Encabezado`, `Pie de página`, `ListaDeProductos`, `DetallesDelProducto`, `Carrito`, y así sucesivamente.

Figura 69

Lenguaje de programación de los Componentes React

```
// App.js
import React from 'react';
import Header from './components/Header';
import ProductList from './components/ProductList';
import Footer from './components/Footer';

function App() {
  return (
    <div className="App">
      <Header />
      <ProductList />
      <Footer />
    </div>
  );
}

export default App;
```

Fuente: Elaboración propia
Fuente: https://brew.sh

Paso 3: Datos y Estado

- Datos de Productos: Define la estructura de los datos de tus productos. Puedes simularlos localmente en un archivo JSON o mediante una API externa.
- Estado Global: Considera el uso de Redux o Context API para manejar el estado global de la aplicación, especialmente para el carrito de compras y la autenticación.

Figura 70

Aplicación Redux o Context API

```
// productsData.js
const products = [
  { id: 1, name: 'Producto 1', price: 10 },
  { id: 2, name: 'Producto 2', price: 15 },
  { id: 3, name: 'Producto 3', price: 20 },
  // ...
];

export default products;
```

Fuente: Elaboración propia
Fuente: https://brew.sh

Paso 4: Desarrollo de Funcionalidades

- Página de Listado de Productos: Crea una página que muestre los productos disponibles con detalles como nombre, precio, y botón de "Agregar al Carrito".

Figura 71

Aplicación Redux o Context API 1

```
// ProductList.js
import React from 'react';
import products from '../data/productsData';

const ProductList = () => {
  return (
    <div>
      <h2>Productos</h2>
      {products.map(product => (
        <div key={product.id}>
          <p>{product.name} - ${product.price}</p>
          <button>Agregar al Carrito</button>
        </div>
      ))}
    </div>
  );
};

export default ProductList;
```

Fuente: Elaboración propia
Fuente: https://brew.sh

- Detalles del Producto: Implementa una página para mostrar detalles completos de un producto al hacer clic en él.

Figura 72

Aplicación Redux o Context API 2

```
// ProductDetail.js
import React from 'react';
import { useParams } from 'react-router-dom';
import products from '../data/productsData';

const ProductDetail = () => {
  const { productId } = useParams();
  const product = products.find(product => product.id === parseInt(produ

  return (
    <div>
      <h2>Detalle del Producto</h2>
      <p>{product.name}</p>
      <p>Precio: ${product.price}</p>
      <button>Agregar al Carrito</button>
    </div>
  );
};

export default ProductDetail;
```

Fuente: Elaboración propia
Fuente: https://brew.sh

- Carrito de Compras: Desarrolla la funcionalidad para agregar/quitar productos, ajustar cantidades y calcular el total.

Figura 73

Aplicación Redux o Context API 3.

```
// Componente Cart para visualizar el carrito de compras
import React from 'react';

const Cart = ({ cart, removeFromCart }) => {
  return (
    <div>
      <h2>Carrito de Compras</h2>
      {cart.map(item => (
        <div key={item.id}>
          <p>{item.name}</p>
          <p>Cantidad: {item.quantity}</p>
          <button onClick={() => removeFromCart(item)}>Quitar del Carrit
        </div>
      ))}
    </div>
  );
};

export default Cart;
```

Fuente: Elaboración propia
Fuente: https://brew.sh

Paso 5: Checkout y Finalización de Compra

- Página de Checkout: Diseña y desarrolla una página de checkout con opciones para agregar la dirección de envío y detalles de pago.
- Proceso de Pago: Integra un servicio de pago, como Stripe o PayPal, para permitir a los usuarios finalizar su compra.

Figura 74

Aplicación de React

```
// Checkout.js
import React from 'react';

const Checkout = () => {
  return (
    <div>
      <h2>Checkout</h2>
      <form>
        <input type="text" placeholder="Dirección de Envío" />
        <input type="text" placeholder="Detalles de Pago" />
        <button>Finalizar Compra</button>
      </form>
    </div>
  );
};

export default Checkout;
```

Fuente: Elaboración propia
Fuente: https://brew.sh

Paso 6: Pruebas y Optimización

- Pruebas: Realiza pruebas exhaustivas para asegurarte de que todas las funcionalidades funcionen correctamente.
- Optimización de Rendimiento: Asegúrate de optimizar tu aplicación para un rendimiento óptimo, minimizando el tiempo de carga y mejorando la experiencia del usuario.

Paso 7: Despliegue

En la iniciar tu tienda en línea utilizando React, aquí tienes algunos pasos fundamentales:

- Implementación en Producción: Selecciona un servicio de alojamiento como Netlify, Vercel o Firebase para poner en marcha tu tienda.
- Dominio Propio: Considera la posibilidad de adquirir un dominio para tu negocio y conectarlo con el sitio que has configurado.

En donde los pasos te proporcionan una sólida base para empezar a construir tu tienda en línea. Recuerda que el proceso de desarrollo incluye diversas consideraciones, como la protección de los pagos, la experiencia del usuario y la administración eficiente de productos y pedidos.

Context API

Esta es una función de React que simplifica la gestión y el intercambio de información entre los distintos componentes de tu aplicación. Su propósito principal es evitar el "prop drilling", que es el proceso de transferir propiedades manualmente a través de varios niveles dentro de la jerarquía de componentes, incluso cuando algunos de estos componentes no utilizan esos datos de forma directa (Banks y Porcello, 2020).

Desde un enfoque arquitectónico, la API de Contexto posibilita la centralización de determinados estados o configuraciones globales para múltiples componentes de la aplicación sin necesidad de intermediarios. Esto favorece el modularidad, el mantenimiento y la coherencia funcional en aplicaciones frontend más elaboradas. De acuerdo a Facebook (2019), Context fue creado como una herramienta que facilita el intercambio de datos considerados globales en la aplicación, tales como configuraciones de interfaz, autenticación de usuarios, preferencias visuales o información sobre el estado de sesión. De esta manera, la API de

Contexto proporciona un mecanismo integrado para resolver situaciones donde varios componentes requieren acceder a un mismo conjunto de datos de manera concurrente.

Desde una perspectiva metodológica, la Context API se relaciona al respecto directamente con aquellos principios de arquitectura de software muy presentes en teoría como la separación de las responsabilidades y la gestión correcta de los estados que se comparten. Por lo que su empleo puede contribuir a la reducción del acoplamiento innecesario entre los componentes y a una mejor organización estructural de aplicaciones escritas en React.

Además, la Context API también cobra mucha importancia en el ámbito del diseño multimedia y el desarrollo frontend moderno debido a la creciente complicación que presentan las interfaces digitales interactivas.

Las aplicaciones de comercio electrónico, las plataformas educativas, los sistemas multimedia, los entornos colaborativos requieren mecanismos para compartir información de una forma adecuada que permita la sincronización de los datos a través de las partes del sistema. La coherencia y la continuidad de la interacción son aspectos que determinan la experiencia que tienen los usuarios en sistemas digitales como el que se aborda en este trabajo (Norman, 2013). La Context API juega un papel importante al facilitar la continuidad funcional, ya que permite que diferentes partes del sistema accedan a la información compartida de forma concurrente, pero sin que haya duplicidad de la estructura de la aplicación.

Desde el punto de vista técnico, la Context API se basa en tres elementos fundamentales: el contexto (Context), el proveedor (Provider), y el consumidor (Consumer) o Hooks asociados como el useContext. El proveedor permite la entrega de información compartida y los componentes consumidores acceden de forma dinámica a la información dentro de la jerarquía de componentes. Sin embargo, diversos autores advierten de que la Context API se debe utilizar con cuidado, ya que un uso indiscriminado podría dar lugar a renderizados innecesarios y situaciones de performance negativa en aplicaciones de gran escala. Por ello, tecnologías como Redux o Zustand continúan utilizándose en situaciones que requieren una gestión más segmentada y avanzada del estado global (Stefanov, 2016).

Por tanto, el análisis de la Context API no debe limitarse a una descripción de alcance funcional, sino que debe considerarse dentro del marco más amplio de las arquitecturas de gestión del estado de las aplicaciones escritas con React. Desde el punto de vista académico, esta funcionalidad es un mecanismo orientado a mejorar la comunicación estructural de las

aplicaciones, la mantenibilidad y la eficiencia interactiva en sistemas del frontend contemporáneo.

¿ Cómo funciona la API de Context en React?

- **Creación de Context:**

Se inicia con la creación de un contexto utilizando `**React.createContext()**`. Por ejemplo:

```
const ThemeContext = React.createContext('light');
```

- **Proveedor (Provider):**

El componente `**Provider**` es utilizado para envolver los componentes que necesitan acceder a los datos del contexto. Se define alrededor de la jerarquía de componentes en un nivel superior.

```
<ThemeContext.Provider value="dark">
    {/* Componentes que necesitan acceder al contexto */}
</ThemeContext.Provider>
```

- **Consumidor (Consumer):**

Los componentes que desean consumir los datos del contexto utilizan el componente `**Consumer**` o el hook `**useContext**`.

```
<ThemeContext.Consumer>
    {value => /* Usa el valor del contexto */}
</ThemeContext.Consumer>
```

El desarrollo con el hook`useContext`:

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

Ventajas de Context API en React:

- Acceso a Datos Globales: Permite el acceso a datos compartidos por múltiples componentes sin la necesidad de pasar props manualmente a través de diferentes niveles de la jerarquía.

- Mantenimiento Sencillo: Simplifica el mantenimiento y la actualización de datos, ya que los cambios pueden propagarse fácilmente a través del contexto.

Uso Común de Context API:

- Tema de la Aplicación: Para definir un tema global que afecte diferentes partes de la aplicación, por ejemplo, el modo claro y oscuro.
- Autenticación del Usuario: Almacenar y compartir el estado de autenticación en toda la aplicación.
- Internacionalización: Mantener información sobre el idioma seleccionado y aplicar traducciones.

Figura 75

Ejemplo de Implementación:

```
const ThemeContext = React.createContext('light');

function App() {
  return (
    <ThemeContext.Provider value="dark">
      <Toolbar />
    </ThemeContext.Provider>
  );
}

function Toolbar() {
  return (
    <div>
      <ThemedButton />
    </div>
  );
}

function ThemedButton() {
  const theme = useContext(ThemeContext);
  return <button className={theme}>Switch Theme</button>;
}
```

Fuente: Elaboración propia
Fuente: https://brew.sh

En el estudio sencillo demuestra cómo utilizar la API de Context en React para compartir y acceder a datos globales dentro de una aplicación. Es fundamental comprender cómo se propagan y utilizan los datos a través de esta API para una gestión eficiente del estado en React.

Firebase

Firebase es la plataforma puesta en marcha por Google, para facilitar realizar aplicaciones web y móviles de la forma integrada, mediante servicios como autenticación, bases de datos, almacenamiento, análisis y computación en la nube. En el desarrollo frontend actual, Firebase ha cobrado notoriedad por su capacidad para simplificar tareas de backend y sincronización de datos en tiempo real en aplicaciones que son interactivas.

La versión 9 de Firebase fue determinada por los cambios de forma estructurales producidos en lo que hace a la arquitectura modular, optimización del rendimiento, pero también reducción del tamaño de los paquetes de las aplicaciones de JavaScript modernas.

n la documentación oficial de Firebase (Google, 2023) se argumenta que su actualización o nueva versión se enmarca en el en el objetivo de mejorar la eficiencia operativa, así como su compatibilidad con arquitecturas modernas basadas en módulos ES6 y herramientas de empaquetado durante el frontend.Desde un enfoque técnico, uno de los aspectos que ha cambiado en Firebase v9 es la transición de un modelo basado en espacios de nombres (namespace-based API) a uno modular en base a importaciones específicas. Dicho modelo permite la importación, dentro de la aplicación, de aquellas funciones que le dan soporte, eliminando dependencias innecesarias y optimizando el tamaño del paquete finalmente distribuido por el navegador.

Vista desde la ingeniería de software, está relacionada con principios de modularidad y optimización de recursos computacionales. Siguiendo a Martin Fowler (2018), las arquitecturas modulares favorecen el mantenimiento, la escalabilidad y la eficiencia de los sistemas digitales complejos, ya que permiten desacoplar funcionalidades y reducen la sobrecarga estructural.

En el caso de aplicaciones desarrolladas con React, la integración de Firebase v9 es especialmente interesante para implementar servicios de autenticación de usuarios, almacenamiento en la nube, bases de datos en tiempo real y sincronización de datos. Permiten promover el desarrollo de plataformas interactivas orientadas a la experiencia del usuario, comercio electrónico y plataformas colaborativas, así como aplicaciones de multimedia.

Al mismo tiempo, la arquitectura modular de Firebase v9 coopera perfectamente con los modelos de desarrollo basados en componentes que se utilizan en React: la separación funcional de los servicios contribuye a que la integración entre frontend y backend esté más estructurada para gestionar procesos de autenticación, persistencia de datos y comunicación en tiempo real.

Desde la perspectiva metodológica, la utilización de Firebase v9 también se puede justificar mediante criterios de optimización del rendimiento web ya que la reducción del tamaño del paquete y la carga bajo demanda contribuyen a mejorar los tiempos de respuesta y la eficiencia de las aplicaciones que implementan un frontend moderno. Tal y como señala Jakob Nielsen (1994) la velocidad de respuesta es un criterio importante para determinar la percepción de calidad de la interacción entre la interfaz digital y el usuario.

A su vez, la integración de Firebase en aplicaciones React debe exponer una visión más allá de la aplicación del sistema en sí. Desde un enfoque académico Firebase es una solución que se orienta a integrar servicios distribuidos y computación en la nube y la sincronización de estados en ecosistemas digitales interactivos.

Por consiguiente, el estudio de Firebase v9 permite acercarse a los principios de integración de servicios escalables y de los modelos de comunicación bajo el enfoque del tiempo real que se implementan en las arquitecturas modernas de los frontend donde se va pequeña funcionalidad y eficiencia de uso de las aplicaciones web.

Importación de Módulos Firebase

La versión 9 de Firebase adopta una nueva forma de importar módulos, facilitando el control sobre qué partes de Firebase se utilizan en la aplicación.

Figura 76

Aplicación de Firebase

```
import { initializeApp } from 'firebase/app';
import { getFirestore, collection, getDocs } from 'firebase/firestore';
// Otros módulos requeridos
```

Fuente: Elaboración propia
Fuente: https://brew.sh

Inicialización de la Aplicación

El primer paso es inicializar la aplicación de Firebase. Esto se logra con el método `initializeApp`, proporcionando la configuración de Firebase, como la API Key y el Auth Domain.

Figura 77

Aplicación de React

```
const firebaseConfig = {
  apiKey: 'tu_api_key',
  authDomain: 'tu_auth_domain',
  // Otros detalles de configuración
};

const app = initializeApp(firebaseConfig);
```

Fuente: Elaboración propia
Fuente: https://brew.sh

Acceso a los Servicios de Firebase

Una vez inicializada la app, se puede acceder a los servicios de Firebase, como Firestore, Auth, Storage, etc. usando métodos específicos importados.

Figura 78

Ejemplo de Acceso a Firestore

```
const db = getFirestore(app);
const querySnapshot = await getDocs(collection(db, 'nombre_coleccion'));
querySnapshot.forEach((doc) => {
  console.log(doc.id, ' => ', doc.data());
});
```

Fuente: Elaboración propia
Fuente: https://brew.sh

Gestión de Usuarios

Con la finalidad de que la autenticación de usuarios, se utilizan métodos disponibles en Firebase, como createUserWithEmailAndPassword, signInWithEmailAndPassword, signOut, etc.

Ventajas de Firebase v9

- Gestión del tamaño del paquete: Facilita importar únicamente los módulos que realmente necesitas, lo que contribuye a disminuir el tamaño del paquete.
- Sintaxis más reciente: Utiliza una sintaxis moderna que resulta más accesible para trabajar con los módulos de Firebase.
- Eficiencia y mejora: Proporciona un control superior sobre los módulos utilizados, lo que optimiza la eficacia del código y el rendimiento de la aplicación.

Se debe incorporar Firebase v9 en React requiere entender la nueva forma de importar y cómo manejar los distintos módulos. Este método permite un manejo más específico sobre qué componentes de Firebase se integran en la aplicación, resultando en un rendimiento y eficacia superiores.

Reordenar elementos con Drag & Drop

Para implementar la funcionalidad de reordenar elementos con arrastrar y soltar (Drag & Drop) en React, se puede emplear bibliotecas como `react-beautiful-dnd` o hacerlo directamente con la API de arrastrar y soltar de HTML5. A continuación, se proporciona una visión general utilizando la biblioteca `react-beautiful-dnd`:

Implementación con `react-beautiful-dnd`

- **Instalación de la biblioteca:**

Instala **`react-beautiful-dnd`** a través de npm o yarn:

```
npm install react-beautiful-dnd
```

- **Uso básico en un componente:**

Importa los componentes necesarios y configura tu lista y sus elementos para ser reordenados.

Figura 79

Aplicación de react-beautiful-dnd`

```
import { DragDropContext, Droppable, Draggable } from 'react-beautiful-dnd';

// ... (Componente principal)

return (
  <DragDropContext onDragEnd= { handleDragEnd } >
  <Droppable droppableId="droppable" >
    {(provided) => (
      <ul { ...provided.droppableProps } ref = { provided.innerRef } >
      {
        items.map((item, index) => (
          <Draggable key= { item.id } draggableId = { item.id } index = { index } >
          {(provided) =>(
            <li
                  ref= { provided.innerRef }
                  { ...provided.draggableProps }
                  { ...provided.dragHandleProps }
            >
            { item.content }
            < /li>
        )
      }
        < /Draggable>
        ))}
{ provided.placeholder }
</ul>
    )}
</Droppable>
  < /DragDropContext>
);
```

Fuente: Elaboración propia
Fuente: https://brew.sh

Donde `handleDragEnd` sería la función que maneja el evento de finalización del arrastre.

- Manejo de eventos:

Define la función `handleDragEnd` que actualice el estado con la nueva posición de los elementos después del arrastre.

Figura 80

Aplicación de handleDragEnd`

```
const handleDragEnd = (result) => {
  if (!result.destination) {
    return;
  }
  const reorderedItems = reorder(
    items,
    result.source.index,
    result.destination.index
  );
  setItems(reorderedItems);
};
```

Fuente: Elaboración propia
Fuente: https://brew.sh

En función `handleDragEnd` reordena los elementos dentro del estado items según la posición final después del arrastre.

- **Estilizado personalizado:**

Puedes aplicar estilos personalizados para visualizar el arrastre de elementos a través de las props proporcionadas por `react-beautiful-dnd`.

Esto es un resumen básico de cómo implementar el reordenamiento de elementos con `react-beautiful-dnd` en React. Recuerda adaptar el código a tu aplicación y manipular el estado según sea necesario.

La implementación utiliza la biblioteca `react-beautiful-dnd`, es ideal para situaciones donde se necesita una solución sencilla y de fácil implementación para arrastrar y soltar elementos en la aplicación React.

Aplicación actividad física y seguimiento deportivo

Paso 1: Configuración del Proyecto

Primero, crea un nuevo proyecto de React utilizando create-react-app.

Figura 81

Aplicación de React utilizando create-react-app.

```
npx create-react-app app-deporte
cd app-deporte
npm start
```

Fuente: Elaboración propia
Fuente: https://brew.sh

Paso 2: Estructura de la Aplicación

En la organización de debe generar la estructuración de los archivos y carpetas, por ejemplo:

- Creación de la carpeta 'components' para almacenar los componentes de React.
- La creación de los archivos para diferentes partes de la aplicación: contador, temporizador, entre otros.

Paso 3: Componente de Contador

Crea un componente simple de contador que aumente el número cada vez que se hace clic en un botón.

Figura 82

Aplicación de React utilizando create-react-app componente

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

const Contador = () => {
  const [contador, setContador] = useState(0);

  const incrementarContador = () => {
    setContador(contador + 1);
  };

  return (
    <div>
      <h2>Contador de Ejercicios</h2>
      <p>Número de ejercicios: {contador}</p>
      <button onClick={incrementarContador}>Agregar Ejercicio</button>
    </div>
  );
};

export default Contador;
```

Fuente: Elaboración propia
Fuente: https://brew.sh

Paso 4: Componente de Temporizador

Crea un temporizador para contar el tiempo de ejercicio.

Figura 83

Aplicación de React utilizando create-react-app temporizador

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

const Temporizador = () => {
  const [segundos, setSegundos] = useState(0);

  useEffect(() => {
    const id = setInterval(() => {
      setSegundos(segundos => segundos + 1);
    }, 1000);
    return () => clearInterval(id);
  }, []);

  return (
    <div>
      <h2>Temporizador</h2>
      <p>Tiempo transcurrido: {segundos} segundos</p>
    </div>
  );
};

export default Temporizador;
```

Fuente: Elaboración propia
Fuente: https://brew.sh

Paso 5: Integración de Componentes

Integra los componentes del contador y el temporizador en tu aplicación principal.

Figura 84

Aplicación de React utilizando create-react-app temporizador

```
import React from 'react';
import Contador from './components/Contador';
import Temporizador from './components/Temporizador';

const App = () => {
  return (
    <div>
      <h1>¡Aplicación para Hacer Deporte!</h1>
      <Contador />
      <Temporizador />
    </div>
  );
};

export default App;
```

Fuente: Elaboración propia
Fuente: https://brew.sh

En los pasos son una base para crear una aplicación simple de hacer deporte en React. Se puede agregar más funcionalidades, como diferentes tipos de ejercicios, pausar/reiniciar el temporizador, o guardar el progreso del usuario.

Capítulo 5.

Aspectos Avanzados y Especiales

Aplicación REST con React

Paso 1: Configuración Inicial del Proyecto

- Inicialización del Proyecto React:

Creamos un nuevo proyecto de React. En el terminal se ejecuta los siguientes comandos:

```
npx create-react-app star-wars-characters
cd star-wars-characters
```

- Instalación de Axios:

Para gestionar las solicitudes a la API, instalaremos Axios, una librería popular para realizar peticiones HTTP.

```
npm install axios
```

Paso 2: Configuración y Consumo de la API SWAPI

- Configuración de Axios:

Figura 85

Crea n archivo para configurar Axios y las peticiones a la API.

```
// axiosConfig.js
import axios from 'axios';

const instance = axios.create({
  baseURL: 'https://swapi.dev/api/',
  headers: {
    'Content-Type': 'application/json',
  }
});

export default instance;
```

Fuente: Elaboración propia
Fuente: https://brew.sh

Paso 3: Creación de Componentes

- Componente para Mostrar los Personajes:

Desarrollemos un componente que solicite y visualice la información de los personajes de Star Wars.

Figura 86

Crea n archivo para configurar Axios y las peticiones Star Wars.

```
// CharacterList.js
import React, { useState, useEffect } from 'react';
import axios from './axiosConfig';

const CharacterList = () => {
  const [characters, setCharacters] = useState([]);

  useEffect(() => {
    axios.get('people/')
      .then(response => {
        setCharacters(response.data.results);
      })
      .catch(error => {
        console.error('Error:', error);
      });
  }, []);

  return (
    <div>
      <h1>Personajes de Star Wars:</h1>
      <ul>
        {characters.map(character => (
          <li key={character.url}>{character.name}</li>
        ))}
      </ul>
    </div>
  );
};

export default CharacterList;
```

Fuente: Elaboración propia
Fuente: https://brew.sh

Paso 4: Integración en la Aplicación Principal

- Integración del Componente en la Aplicación Principal:

En el archivo principal de la aplicación (App.js), importa y utiliza el componente que muestra la lista de personajes.

Figura 87

Crea n archivo para configurar Axios y las peticiones Star Wars.APP js

```
// App.js
import React from 'react';
import './App.css';
import CharacterList from './CharacterList';

function App() {
  return (
    <div className="App">
      <CharacterList />
    </div>
  );
}

export default App;
```

Fuente: Elaboración propia
Fuente: https://brew.sh

Paso 5: Ejecución y Visualización

- Ejecución de la Aplicación:

Para visualizar la aplicación y la lista de personajes de Star Wars, ejecuta:

```
npm start
```

Estos pasos permitirán mostrar la lista de personajes de Star Wars utilizando la API SWAPI en tu aplicación React. Logrando personalizar y expandir este código para mostrar más detalles sobre cada personaje o agregar funcionalidades adicionales según las necesidades del proyecto.

Rutas privadas

La implementación de rutas privadas es un mecanismo clave en el contexto del desarrollo de aplicaciones con React, dado que permite restringir el acceso a algunas partes del sistema de acuerdo al nivel de autenticación o autorización del usuario. Desde la óptica de la

seguridad informática, dicho modelo permite proteger recursos, información y funciones sensibles en las aplicaciones web interactivas; el concepto de rutas privadas está134 directamente relacionado con los principios de control de acceso y autenticación: la autenticación es el proceso por el cual el sistema verifica con un usuario determinado la identidad de un usuario, mientras que la autorización indica los recursos o funciones a los cuales puede acceder el usuario autenticado (Stallings & Brown, 2018). En las aplicaciones frontend modernas, las rutas privadas tienden a funcionar como mecanismos de validación que condicionan la navegación en la interfaz digital.Este modelo permite que ciertas vistas o componentes solo estén disponibles para el conjunto de usuarios que han sido autenticados, lo que se traduce en protección estructural y control de la interacción del interior del sistema.

Desde la perspectiva técnica, la aproximación de la implementación de rutas privadas suele implementarse por medio de herramientas como React Router, que incorpora los procesos de verificación respecto de tokens de autenticación, estados globales o servicios de autenticación externos. En este sentido, Firebase o sistemas basados en JSON Web Tokens (JWT) se suelen implementar para gestionar sesiones y permisos de acceso.

En palabras de Martin Fowler (2018), las arquitecturas de las aplicaciones web del presente requieren mecanismos de control de acceso integrados en las aplicaciones que aseguren una clara separación de funcionalidades públicas y privadas de un modo que implique organización del código modular, seguridad y mantenibilidad de las mismas.Del mismo modo, las rutas privadas también están vinculadas a la gestión estatal global de la aplicación en React. Para ello existen herramientas como Redux, Zustand o Context API, que nos permiten guardar datos de autenticación y compartirlos de forma dinámica en los distintos componentes de la aplicación.

Desde un punto de vista metodológico, la implementación de rutas privadas también afecta la experiencia de usuario y la continuidad en la navegación. Según Don Norman (2013), los sistemas digitales deben proporcionar una buena retroalimentación y una buena conducta de los mismos frente a los procesos de interacción y validación. En definitiva, los mecanismos de autenticación deben estar implícitos y ser comprensibles dentro del total de la experiencia del usuario.

En otro orden de cosas, la implementación de rutas privadas también implica tener en cuenta aspectos relacionados con la persistencia de la sesión, la protección de datos o el manejo de estados asíncronos. El proceso de validación de la autenticación debe ser óptimo, de manera

que no se produzcan accesos no autorizados y el mecanismo de la aplicación continúe operando.

Desde un enfoque académico, el análisis de rutas privadas no puede estar restringido a los procedimientos técnicos de implementación de esta. Todo lo contrario, es una parte estructural de la seguridad informática, la arquitectura frontend o la experiencia de usuario dentro del caso del desarrollo de aplicaciones web contemporáneas. Por ello, las rutas privadas son unos mecanismos básicos que nos contribuyen a garantizar la integridad funcional, la protección de los recursos y la organización funcional de los sistemas digitales interactivos.

Paso 1: Configuración de la Autenticación

Implementación de la Autenticación: Selecciona y configura un sistema de autenticación para tu aplicación. Puedes emplear una solución de autenticación preexistente, como Firebase, Auth0, o implementar tu propio sistema de autenticación utilizando JWT (JSON Web Tokens).

Paso 2: Implementación de Rutas Privadas

Desarrollo de Componentes para Rutas Privadas: Crea un componente que administre las rutas privadas, verificando si un usuario está autenticado antes de permitir el acceso a ciertas páginas.

Figura 88

Ejemplo de componente `PrivateRoute`

```
import React from 'react';
import { Route, Redirect } from 'react-router-dom';

const PrivateRoute = ({ component: Component, isAuthenticated, ...rest }
  <Route
    {...rest}
    render={props =>
      isAuthenticated ? (
        <Component {...props} />
      ) : (
        <Redirect to="/login" />
      )
    }
  />
);

export default PrivateRoute;
```

Fuente: Elaboración propia
Fuente: https://brew.sh

Paso 3: Implementar Rutas Protegidas

Configuración de Rutas Privadas: En el archivo principal de enrutamiento, integra las rutas protegidas y las rutas públicas.

Figura 89

Ejemplo de configuración en `App.js`:

```
import React, { useState } from 'react';
import { BrowserRouter as Router, Route, Switch } from 'react-router-dom'
import PrivateRoute from './PrivateRoute';
import Home from './Home';
import Login from './Login';
import Dashboard from './Dashboard';

function App() {
  const [isAuthenticated, setIsAuthenticated] = useState(false);

  return (
    <Router>
      <Switch>
        <Route exact path="/" component={Home} />
        <Route
          path="/login"
          render={props => (
            <Login {...props} setIsAuthenticated={setIsAuthenticated} />
          )}
        />
        <PrivateRoute
          path="/dashboard"
          component={Dashboard}
          isAuthenticated={isAuthenticated}
        />
      </Switch>
    </Router>
  );
}

export default App;
```

Fuente: Elaboración propia
Fuente: https://brew.sh

Paso 4: Componentes Relacionados

Asegúrate de que los elementos de registro (`Login`) manejen de manera adecuada la verificación de usuarios, y que las secciones protegidas (`Dashboard`) solo sean accesibles si el usuario ha realizado el acceso.

En donde el método fundamental para implementar rutas privadas puede ser modificado y ampliado para incluir funcionalidades adicionales o sistemas de verificación más sofisticados, dependiendo de las necesidades específicas de la aplicación.

Custom Hooks

Los Hooks Personalizados en React son una estrategia que permite la reutilización eficiente de la lógica de estado, efectos y otros patrones entre diferentes componentes. A continuación, te presento una guía completa sobre cómo crearlos y utilizarlos

Los Hooks Personalizados son funciones en React que inician con “use” y pueden incorporar cualquier lógica de estado o efectos que desees reutilizar en múltiples componentes. Esto contribuye a encapsular la lógica y a compartir estados complejos, lo que a su vez mejora la claridad y la organización del código.

Creación de un Custom Hook

La forma básica de un Hook Personalizado es bastante sencilla: puedes emplear Hooks ya existentes como `useState` o `useEffect` y luego retornar lo que necesites utilizar en tus componentes. A continuación, te muestro un ejemplo:

Figura 90

Aplicación de Custom Hook

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

function useCustomHook(initialValue) {
  const [value, setValue] = useState(initialValue);

  useEffect(() => {
    // Lógica de efectos
  }, [value]); // Dependencias para el efecto

  function updateValue(newValue) {
    setValue(newValue);
  }

  return { value, updateValue };
}
```

Fuente: Elaboración propia
Fuente: https://brew.sh

Uso del Custom Hook en un Componente

Una vez creado el Custom Hook, puedes usarlo en cualquier componente que necesite la lógica encapsulada. Aquí un ejemplo de cómo se usaría:

Figura 91

Aplicación Custom Hook en un componente

```
import React from 'react';
import useCustomHook from './useCustomHook';

function ComponentUsingCustomHook() {
  const { value, updateValue } = useCustomHook(0);

  return (
    <div>
      <p>Valor: {value}</p>
      <button onClick={() => updateValue(value + 1)}>Incrementar</button
    </div>
  );
}
```

Fuente: Elaboración propia
Fuente: https://brew.sh

Ventajas de los Custom Hooks

- Reutilización de lógica: Facilita el intercambio y la reutilización de la lógica entre distintos elementos.
- Reducción de la complejidad: Permite que la lógica compleja sea más fácil de entender al descomponerla en partes más simples.
- Incremento del modularidad: Ofrece una forma más sencilla de dividir la lógica de distintos componentes.

Los Custom Hooks ofrecen una solución efectiva para compartir lógica entre componentes en React, lo que mejora el modularidad, la reutilización y la claridad del código.

Malentendidos Los errores frecuentes en el ámbito de React suelen reflejar nociones erróneas o confusiones respecto a la biblioteca. A continuación, se presentan algunas de estas nociones junto con sus aclaraciones para evitar malentendidos durante el uso de React:

- React Solo Es Adecuado Para Proyectos GrandesRealidad: Aunque React es ideal para desarrollos grandes gracias a su capacidad de escalar y manejar el estado de manera efectiva, también se puede aplicar en proyectos más pequeños. Su versatilidad y modularidad permiten su uso en diversos entornos.
- React Se Utiliza Exclusivamente Para Interfaces De UsuarioRealidad: Aunque React está enfocado en el diseño y la construcción de interfaces, también se utiliza para crear aplicaciones completas. Combinado con otras herramientas y bibliotecas, React puede gestionar la lógica de la aplicación, así como conectarse con bases de datos, APIs y el backend.
- React Es Un Framework CompletoRealidad: React es una biblioteca especializada en la creación de interfaces de usuario y no un framework que lo abarque todo. Generalmente, se asocia con otras bibliotecas como Redux (para el manejo del estado) o React Router (para el enrutamiento) para desarrollar aplicaciones más completas.
- React Soluciona Automáticamente Todos Los Problemas De RendimientoRealidad: Aunque React es efectivo en términos de rendimiento, no resuelve por sí mismo todos los problemas relacionados con la eficiencia. Es necesario contar con un código bien estructurado y optimizado, así como seguir buenas prácticas de desarrollo, para garantizar un rendimiento óptimo.
- JSX Es La Única Forma De Crear Componentes En ReactRealidad: A pesar de que JSX es el método más común para crear componentes en React, no es el único disponible.

También se pueden desarrollar componentes utilizando JavaScript estándar, lo que brinda mayor flexibilidad en su diseño.

React es Difícil de Aprender:

Una de las valoraciones que existe en el ámbito de la programación frontend actual es pensar que el aprendizaje de React es un proceso difícil. No obstante, múltiples trabajos e investigaciones que tienen que ver con la experiencia en el desarrollo y la adopción de tecnologías indican que dicha percepción depende del nivel de entendimiento que previamente tenga cada uno en JavaScript, programación basada en componentes y arquitectura frontend.

Desde la perspectiva de la metodología, React plantea conceptos de renderizado declarativo, reutilización de componentes y gestión de estado, conceptos que pueden suponer un cambio a nivel conceptual desde una manera tradicional de programación que hace uso de la manipulación directa del DOM. Al mismo tiempo, los componentes reutilizables, el funcionamiento declarativo y una estructura modular facilitan mantener, organizar y escalar aplicaciones web modernas (Banks & Porcello, 2020).

Según Stoyan Stefanov (2016), uno de los motivos que ha favorecido la adopción de React es la claridad estructural que presenta su modelo basado en componentes reutilizables, ya que facilita la separación de responsabilidades y la reutilización de código en proyectos de frontend que son complejos. React también cuenta con una extensa comunidad de documentación técnica y bibliotecas complementarias que ayudan a los desarrolladores en el proceso de aprendizaje. Por otro lado, la cantidad de recursos educativos, tutoriales y documentación oficial de React permiten que se reduzcan las barreras de la entrada y que se puedan entrenar procesos de aprendizaje progresivo a los desarrolladores en fases de iniciación y en niveles intermedios.

Desde una mirada académica, no es correcto tildar a React de ser una tecnología extremadamente difícil de aprender, sino más bien que su adopción significa pasar por un aprendizaje progresivo en relación a principios en cuanto a programación reactiva, manejo de estados y arquitectura basada en componentes. Una vez dominados estos principios, React tiene ventajas importantes en torno a escalabilidad, reutilización y eficiencia en el desarrollo.

Por otro lado, el aprendizaje de React debe contextualizarse a la luz de la evolución del desarrollo web contemporáneo. Martin Fowler (2018) menciona que las arquitecturas modernas del frontend, son arquitecturas que demandan modelos más dinámicos y más modulares por la complejidad que presenta el desarrollo de las interfaces digitales. En ese

sentido, el uso de React aparece como una tecnología que responde a las nuevas exigencias en la construcción y el mantenimiento de aplicaciones interactivas, es decir, permitir que la creación de pantalla pueda ser ágil, rápida y simplificada para todos aquellos que quieran crear sus propias aplicaciones web interactivas.

Por consiguiente, la supuesta dificultad de React, no debe concebirse como una característica de la tecnología sino como una consecuencia de la adaptación a los nuevos modelos de desarrollo frontend. A partir de una mirada científica y técnica, React es una herramienta que proporciona una forma de estructurar el desarrollo basado en el enorme conjunto de documentación que ofrece y que tiene como resultado la construcción de aplicaciones web, escalables, adaptables y eficientes para el medio digital.

Entender estas falacias en torno a React es fundamental para maximizar su utilidad y construir aplicaciones más eficientes y adaptables.

Estructura de Carpetas

La carpeta `trivial-app` contiene la estructura de archivos necesaria para la aplicación, con una carpeta `src` que aloja todos los archivos relacionados con la lógica y los componentes de React. Dentro de `src`, la carpeta components almacena los archivos de componentes React como `Pregunta.js` y `Opciones.js`. La carpeta data contiene un archivo JSON llamado `falacias.json`, que tiene datos sobre las falacias que se mostrarán en la aplicación.

Figura 92

Aplicación de React estructura de carpetas

```
trivial-app/
├── src/
|   ├── components/
|   |   ├── Pregunta.js
|   |   ├── Opciones.js
|   |   └── Resultados.js
|   ├── data/
|   |   └── falacias.json
|   └── App.js
└── package.json
```

Fuente: Elaboración propia
Fuente: https://brew.sh

Falacias Data (falacias.json):

El archivo `falacias.json` es un array que contiene objetos con información sobre las falacias. Cada objeto incluye un ID, nombre, descripción y tipo de falacia. Se utilizará para mostrar la información de las falacias en la aplicación.

Figura 93

Aplicación de Falacias Data

```
[
  {
    "id": 1,
    "nombre": "Falacia de falso dilema",
    "descripcion": "Forzar a alguien a elegir entre dos opciones cuando
    "tipo": "Falacia lógica"
  },
  // Otras falacias...
]
```

Fuente: Elaboración propia
Fuente: https://brew.sh

Componente Pregunta (Pregunta.js):

El componente `Pregunta` está encargado de mostrar una pregunta con una falacia seleccionada al azar del archivo `falacias.json`. Utiliza el estado de React para almacenar la falacia seleccionada y las opciones relacionadas. Utiliza `useEffect` para seleccionar

aleatoriamente una falacia y sus opciones. Luego, muestra la pregunta con el nombre de la falacia y las opciones mediante el componente `Opciones`.

Figura 94

Aplicación de React

```
import React, { useState, useEffect } from 'react';
import Opciones from './Opciones';
import falaciasData from '../data/falacias.json';

const Pregunta = () => {
  const [falacia, setFalacia] = useState({});
  const [respuestas, setRespuestas] = useState([]);
  // Lógica para seleccionar aleatoriamente una falacia y sus opciones
  useEffect(() => {
    const randomFalacia = falaciasData[Math.floor(Math.random() * falaciasData.length)];
    setFalacia(randomFalacia);
    const allFalaciasExceptSelected = falaciasData.filter(f => f.id !== randomFalacia.id);
    const randomRespuestas = [...Array(3)].map(i => {
      const randomIndex = Math.floor(Math.random() * allFalaciasExceptSelected.length);
      return allFalaciasExceptSelected[randomIndex];
    });
    setRespuestas(randomRespuestas);
  }, []);

  return (
    <div>
      <h1>¿Qué tipo de falacia es: {falacia.nombre}?</h1>
      <Opciones respuestas={respuestas} falaciaCorrecta={falacia.tipo} />
    </div>
  );
};

export default Pregunta;
```

Fuente: Elaboración propia
Fuente: https://brew.sh

Componente Opciones (Opciones.js):

El componente `Opciones` recibe las respuestas (falacias diferentes a la correcta) y la falacia correcta. En este ejemplo, muestra botones con los tipos de las falacias como opciones para que el usuario seleccione la respuesta correcta. Sin embargo, en este punto, la lógica para determinar la respuesta correcta aún no está implementada.

Figura 95

Aplicación de React

```
import React from 'react';

const Opciones = ({ respuestas, falaciaCorrecta }) => {
  return (
    <div>
      {respuestas.map(respuesta => (
        <button key={respuesta.id}>
          {respuesta.tipo}
        </button>
      ))}
    </div>
  );
};

export default Opciones;
```

Fuente: Elaboración propia
Fuente: https://brew.sh

En el conjunto conlleva a ser un ejemplo inicial y simplificado. Podrías seguir expandiéndolo implementando lógica para determinar las respuestas correctas, llevar un puntaje, mostrar retroalimentación al usuario y agregar funcionalidades adicionales para hacerlo más interactivo y educativo.

Creación de juego de mesa sensillo "Tres en línea"

La elaboración del juego "Tres en línea" con React resulta ser un ejercicio aplicado en el que se puede analizar los objetivos de la separación de componentes, el desarrollo de la lógica de programación y el control de estados de la interfaz interactiva. Este caso no es una explicación estrictamente instructiva, sino que permite ver cómo una aplicación relativamente sencilla se puede llevar a cabo solamente mediante componentes reutilizables, un manejo dinámico de los datos y la programación lógica de los eventos.

Desde una óptica de la ingeniería del software, el juego funciona organizado mediante componentes funcionales que desempeñan roles explícitos en la interfaz. El componente Square sirve para representar cada una de las casillas del tablero y se alimenta de información mediante props como son el valor representado o una función asociada con el evento de la selección.Esta misma estructura hace gala del hecho de que el principio establecido en React,

la reutilización de sus componentes; de esta forma, una misma unidad de visualización se puede utilizar tantas veces como sea necesario en el sistema.

El componente Board tiene como principal responsabilidad la lógica central de dicho juego, y a través del estado representa las nueve casillas del tablero mediante un array, a la vez que determina de quién es el turno “X” u “O”. La función handleClick actualiza el estado en función de la casilla elegida cuando esta permanece vacía y no hay un ganador. Así, la interacción del usuario con la interfaz del juego se enlaza con la lógica interna de la aplicación.

La función calculateWinner se ha implementado en forma de un algoritmo que evalúa las posibles combinaciones que dan como resultado una victoria. Dicha función compara las posiciones del tablero con las líneas de victoria definidas previamente e establece si hay un resultado final. Desde un punto de vista lógico, se puede separar la lógica del juego de la visualización gráfica, lo cual ayuda igualmente a que el código esté claramente estructurado y que el sistema pueda ser mantenido.

El componente App, por su parte, hace de contenedor de la aplicación y de estructura ordenadora mediante la presentación global del juego. El tablero y la interfaz visual que es visible para el usuario son integrados aquí en este nivel. Al observar la relación entre App, Board y Square se puede apreciar cómo React define las responsabilidades asignadas a los componentes de forma jerárquica, manteniendo el flujo de información y la actualización reactiva del objeto.

Desde un punto de vista académico, este ejemplo muestra los principios del modularidad, la reutilización y la gestión del estado para la construcción de aplicaciones interactivas en una forma organizada. Asimismo, muestra cómo se tiene que tomar en consideración la lógica de la aplicación, la visualización gráfica y el control de los eventos, y cómo es importante en el desarrollo de las interfaces digitales que son dinámicas cuando se utilizan tecnologías como React (Banks & Porcello, 2020; Stefanov, 2016).

Figura 96

Estructura de Carpetas

```
juego-mesa/
├── src/
|   ├── components/
|   |   ├── Square.js
|   |   └── Board.js
|   ├── App.js
|   └── index.js
└── package.json
```

Fuente: Elaboración propia
Fuente: https://brew.sh

Componente Square (Casilla):

El componente Square representa una casilla del juego. Recibe dos props:

- `value`: representa el contenido de la casilla ("X", "O" o `null`).
- `onClick`: es una función que maneja el evento de clic en la casilla.

El componente `Square` es un botón interactivo que muestra el valor recibido en la prop `value`. Al hacer clic en una casilla, ejecuta la función `onClick`.

Figura 97

Aplicación de los componente Square

```
import React from 'react';

const Square = ({ value, onClick }) => {
  return (
    <button className="square" onClick={onClick}>
      {value}
    </button>
  );
};

export default Square;
```

Fuente: Elaboración propia
Fuente: https://brew.sh

Función calculateWinner

Es una función que verifica si hay un ganador basado en las posiciones de las casillas. Compara las líneas posibles del juego para determinar si hay un ganador, retornando “X”, “O” o `null`.

Componente Board - Tablero:

El componente Board representa el tablero del juego. Aquí se realiza la mayor parte de la lógica del juego. Este componente usa el estado de React para mantener el estado actual del tablero y el turno del jugador.

- Utiliza un array de 9 elementos (squares) para representar las casillas del tablero. El estado inicial se establece con Array(9).fill(null).
- La función handleClick maneja el evento de clic en una casilla y actualiza el estado squares. Verifica si hay un ganador o si la casilla ya está marcada.
- renderSquare es una función auxiliar que renderiza un componente Square para cada casilla, pasándole el valor correspondiente y la función handleClick.
- Calcula y muestra el estado del juego, ya sea el mensaje “Next player: X” o “O” para indicar el próximo movimiento o “Winner: X” o “O” si hay un ganador.

Figura 98

Aplicación de calculate de Winner

```
import React, { useState } from 'react';
import Square from './Square';

const Board = () => {
    const [squares, setSquares] = useState(Array(9).fill(null));
    const [xIsNext, setXIsNext] = useState(true);

    const handleClick = index => {
        const newSquares = [...squares];
        if (calculateWinner(newSquares) || newSquares[index]) {
            return;
        }
        newSquares[index] = xIsNext ? 'X' : 'O';
        setSquares(newSquares);
        setXIsNext(!xIsNext);
    };

    const renderSquare = index => {
        return (
            <Square
        value= { squares[index]}
        onClick = {() => handleClick(index)}
/>
    );
  };

const winner = calculateWinner(squares);
const status = winner ? `Winner: ${winner}` : `Next player: ${xIsNext ? 'X' : 'O'}`;

return (
    <div>
    <div className= "status" > { status } < /div>
    < div className = "board-row" >
        { renderSquare(0) }
{ renderSquare(1) }
{ renderSquare(2) }
</div>
    < div className = "board-row" >
        { renderSquare(3) }
{ renderSquare(4) }
{ renderSquare(5) }
</div>
    < div className = "board-row" >
        { renderSquare(6) }
{ renderSquare(7) }
{ renderSquare(8) }
</div>
    < /div>
  );
};

function calculateWinner(squares) {
    const lines = [
        [0, 1, 2],
        [3, 4, 5],
        [6, 7, 8],
        [0, 3, 6],
        [1, 4, 7],
        [2, 5, 8],
        [0, 4, 8],
        [2, 4, 6],
    ];
    for (let i = 0; i < lines.length; i++) {
        const [a, b, c] = lines[i];
        if (squares[a] && squares[a] === squares[b] && squares[a] === squares[c]) {
            return squares[a];
        }
    }
    return null;
}

export default Board;
```

Fuente: Elaboración propia
Fuente: https://brew.sh

Componente App

El componente `App` es la raíz de la aplicación. Aquí se colocan los elementos principales del juego, como el título y el componente `Board`.

El componente **`Board`** es la interfaz principal del juego, mostrando las casillas y el estado del juego.

Capítulo 6.
Técnicas y Estrategias Avanzadas

Compilar (build o publicar) una aplicación de React

Compilar una aplicación escrita en React supone una fase relevante del ciclo de vida del software, ya que se ocupa de la conversión del código fuente, usado en el desarrollo, a archivos optimizados para la ejecución en un entorno de producción. Llevar a cabo un compilado no significa únicamente "publicar" la aplicación, sino que encierra tareas de optimización, empaquetado, minificación y alistamiento de recursos para optimizar la performance, estabilidad y compatibilidad, en definitiva; el proceso de build convierte componentes, estilos, librerías, dependencias, archivos estáticos, etc. en una versión final que es capaz de interpretarse eficientemente en el navegador, siendo herramientas como Webpack, Vite o Create React App las que automatizan el proceso de transpilación, agrupamiento de módulos y minificación de los archivos generados (Banks & Porcello, 2020).

En la ingeniería de software, la compilación es la tarea que conecta el desarrollo con la puesta en funcionamiento. En esta tarea se detectan errores y se optimizan recursos, además de que se hace la preparación del propio sistema para responder a condiciones de uso reales. Así lo destaca Sommerville (2016), cuando señala la necesidad de realizar procesos de validación y de preparación técnica en la entrega del software para asegurar su funcionamiento, la mantenibilidad y la calidad del software que se obtiene.

Por su parte, la publicación de aplicaciones React también debe particularizarse desde la óptica del rendimiento y la experiencia del usuario. Una aplicación mal optimizada, tal y como señala, puede dar lugar a tiempos de carga excesivamente largos, un elevado consumo de recursos o problemas de navegación. Por este motivo el build debe plantearse en términos de tamaño del paquete, carga diferida de componentes, optimización de imágenes, gestión del enrutamiento o interactividad en múltiples dispositivos.

Sin embargo, desde una perspectiva crítica, el hecho de compilar no garantiza la calidad de por sí. La versión de producción debe ser objeto de pruebas funcionales, pruebas de rendimiento o de revisión de accesibilidad antes de ser definitivamente puesta en circulación. Por lo que la compilación debe interpretarse como un proceso técnico y metódico para asegurar que la aplicación mantenga la coherencia funcional, la eficiencia en su funcionamiento y la experiencia del usuario en el propio entorno final.

En parte, pues, el proceso de build en React pone de manifiesto el vínculo que existe entre arquitectura frontend, optimización técnica y calidad de interacción en el contexto actual de desarrollo web. Por lo que el acto de publicar una aplicación no puede entenderse solo como el hecho de hacerla accesible públicamente, sino que debe implicar la garantía de que su funcionamiento responda a características como ser la estabilidad, escalabilidad, seguridad y usabilidad.Aquí se presenta un enfoque detallado sobre cómo realizar este proceso:

Paso 1: Preparación del Proyecto

Si estás usando una aplicación React creada con `**create-react-app**` o una configuración similar, el proceso de compilación es simple y directo. Para llevar a cabo el build, utiliza el siguiente comando:

```
npm run build
```

Paso 2: Verificación de Scripts

Asegúrate de que tu archivo `**package.json**` contenga la sección de scripts con la configuración adecuada para el proceso de build:

Figura 99

Aplicación verificación de Scripts

```
"scripts": {
  "start": "react-scripts start",
  "build": "react-scripts build",
  "test": "react-scripts test",
  "eject": "react-scripts eject"
}
```

Fuente: Elaboración propia
Fuente: https://brew.sh

Paso 3: Ejecución del Build

Al ejecutar el comando `**npm run build**`, la aplicación se compila y se optimiza para la producción.

Paso 4: Resultados del Build

Una vez completada la compilación, la carpeta **`build`** contiene los archivos optimizados listos para ser desplegados.

Paso 5: Despliegue en Producción

Sube la carpeta **`build`** a tu servidor de alojamiento web o plataforma preferida.

Asegúrate de ajustar las rutas y conexiones a recursos estáticos o API para el entorno de producción.

Estos pasos cubren el proceso esencial para compilar y desplegar una aplicación de React en producción. Las configuraciones de código más específicas dependerán de la estructura y necesidades de tu aplicación.

DeepFetch haciendo peticiones a una API en React

El término "DeepFetch" no es una convención específica, pero en el contexto de React, se puede interpretar como la realización de múltiples peticiones a una API con el fin de obtener información relacionada o anidada, lo que podría denominarse como "navegación profunda" en el flujo de datos.

Estrategias para Navegación Profunda en React

- Peticiones Encadenadas:

En la implicación se realizar solicitudes secuenciales a la API, utilizando los resultados de una solicitud como entrada para la siguiente.

Figura 100

Ejemplo de encadenamiento de solicitudes con Promesas

```
fetch('https://api.example.com/users')
  .then(response => response.json())
  .then(users => {
    const userId = users[0].id;
    return fetch(`https://api.example.com/posts?userId=${userId}`);
  })
  .then(response => response.json())
  .then(posts => console.log(posts))
  .catch(error => console.error('Error:', error));
```

Fuente: Elaboración propia
Fuente: https://brew.sh

Utilizando Async/Await:

El uso de Async/Await simplifica el manejo de peticiones anidadas, haciendo que el código sea más legible.

Figura 101

Ejemplo de peticiones anidadas con Async/Await:

```
const fetchData = async () => {
  try {
    const usersResponse = await fetch('https://api.example.com/users');
    const users = await usersResponse.json();
    const userId = users[0].id;

    const postsResponse = await fetch(`https://api.example.com/posts?userId
    const posts = await postsResponse.json();
    console.log(posts);
  } catch (error) {
    console.error('Error:', error);
  }
};

fetchData();
```

Fuente: Elaboración propia
Fuente: https://brew.sh

Casos de Uso

- Exploración Profunda de Datos: Se utiliza para obtener información más detallada de la API, siguiendo relaciones entre diferentes recursos.
- Recuperación de Datos Relacionados: Permite obtener información adicional basada en una solicitud inicial.

El proceso de "DeepFetch" con React involucra la realización de múltiples peticiones a una API para recuperar datos relacionados o anidados. Esto es útil cuando se necesita obtener información dependiente o relacionada, mejorando la comprensión y el contexto de los datos para su uso en aplicaciones React.

Aplicación textos teatro

Para construir una aplicación de textos teatrales en React, es esencial comprender cómo estructurar los componentes para representar escenas, personajes y diálogos. Estos elementos formarán la base de una experiencia teatral inmersiva y visualmente atractiva dentro de la aplicación.

Estructura de la Aplicación:

- Componentes de Escena:

Cada escena se representará como un componente independiente. Incluirá descripciones y posiblemente diálogos, acciones o narraciones relevantes.

- Componentes de Personajes:

Cada personaje será un componente individual. Podrá contener diálogos específicos del personaje y atributos visuales únicos.

- Componentes de Diálogo:

Utilizar componentes para mostrar interacciones entre personajes. Representarán los diálogos en un formato legible y atractivo.

- Componentes de Actos o Escenas:

Organizar los diálogos y elementos visuales en actos o escenas. Permitirán una mejor estructuración y presentación de la obra.

Técnicas a Considerar:

- Estilización y Diseño:

Emplear CSS o Styled Components para dar formato a los diálogos y elementos visuales.

Crear una presentación que simule la estética teatral.

- Gestión del Estado:

Utilizar el estado de React para manejar el avance o retroceso entre escenas o diálogos.

- Interactividad:

Agregar botones o elementos interactivos para el control de la narrativa.

Animaciones (Opcional):

Incorporar animaciones para realzar los cambios de escena o la presentación de diálogos.

A continuación, un ejemplo básico con un componente de escena que incluye descripciones y diálogos de personajes:

Figura 102

Utilizar el estado de React

```
import React from 'react';

const Scene = ({ description, characters }) => {
  return (
    <div>
      <p>{description}</p>
      {characters.map(character => (
        <div key={character.name}>
          <h3>{character.name}</h3>
          <p>{character.dialogue}</p>
        </div>
      ))}
    </div>
  );
};

const TheaterApp = () => {
  const sceneData = {
    description: 'En el jardín...',
    characters: [
      { name: 'Romeo', dialogue: '¿Dónde estás, Julieta?' },
      { name: 'Julieta', dialogue: 'Estoy aquí, Romeo.' }
    ]
  };

  return (
    <div>
      <h2>Acto 1 - Escena 1</h2>
      <Scene {...sceneData} />
    </div>
  );
};

export default TheaterApp;
```

Fuente: Elaboración propia
Fuente: https://brew.sh

Trivial con ReactJS

Desarrollar un juego de Trivial con ReactJS implica la creación de una aplicación interactiva de preguntas y respuestas que desafía a los usuarios a responder preguntas sobre diversos temas. Este tipo de juego generalmente ofrece múltiples opciones de respuesta, control de puntajes, temporizadores y proporciona retroalimentación instantánea sobre las respuestas.

Estructura del Juego Trivial en ReactJS

- Componentes Clave:

Componente de Pregunta: Se encarga de mostrar la pregunta actual y las opciones de respuesta.
Componente de Puntaje: Muestra el puntaje actual y el progreso del juego.

- Manejo de las Preguntas y Respuestas:

Almacena las preguntas, opciones y respuestas correctas en una estructura de datos (por ejemplo, un array de objetos). Gestiona el cambio de preguntas y opciones después de responder o al avanzar a la siguiente pregunta.

- Lógica del Juego:

Controla el estado del juego, como el puntaje y el número de preguntas respondidas correctamente. Implementa un temporizador para responder cada pregunta, si se requiere.

- Retroalimentación y Comentarios:

Ofrece retroalimentación instantánea sobre la respuesta dada por el usuario (correcta/incorrecta). Muestra el puntaje total al finalizar el juego.

Implementación Básica:

Presento una estructura inicial para un juego trivial en ReactJS, utilizando estados y lógica simple:

Figura 103

Aplicación de ReactJS,

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

const TrivialGame = () => {
  const questions = [
    {
      question: '¿Cuál es la capital de Francia?',
      options: ['Madrid', 'París', 'Londres', 'Berlín'],
      correctAnswer: 'París'
    },
    // Otras preguntas se pueden añadir aquí...
  ];

  const [currentQuestion, setCurrentQuestion] = useState(0);
  const [score, setScore] = useState(0);

  const handleAnswer = (selectedAnswer) => {
    if (selectedAnswer === questions[currentQuestion].correctAnswer)
      setScore(score + 1);
    }
    // Lógica para avanzar a la siguiente pregunta
    setCurrentQuestion(currentQuestion + 1);
  };

  return (
    <div>
      <h1>¡Trivial!</h1>
      <h2>Pregunta {currentQuestion + 1}:</h2>
      <p>{questions[currentQuestion].question}</p>
      <ul>
        {questions[currentQuestion].options.map((option, index) => (
          <li key={index} onClick={() => handleAnswer(option)}>{opti
        ))}
      </ul>
      <p>Puntaje: {score}</p>
    </div>
  );
};

export default TrivialGame;
```

Fuente: Elaboración propia
Fuente: https://brew.sh

En el ejemplo inicial brinda una estructura básica para un juego trivial en ReactJS, con preguntas predefinidas, opciones de respuesta, seguimiento del puntaje y una lógica elemental para avanzar entre preguntas. Para un juego más completo, se necesitaría una interfaz más sofisticada, lógica adicional y funcionalidades avanzadas.

Bibliografía

Banks, A. A., & Porcello, E. E. (2020). Learning React: Modern patterns for developing React apps (2nd ed.). O'Reilly Media. Obtenido de https://ebooks.karbust.me/Technology/Learning%20React%20-%20Modern%20Patterns%20for%20Developing%20React%20Apps%20-%20Alex%20Banks,%20Eve%20Porcello%20-%20O'Reilly%20Media%20(2020).pdf

Enríquez, S. M. (2020). Características de las herramientas multimedia para el desarrollo de Presentaciones Interactivas. *Licenciada en Ciencias de la Educación Mención Secretariado , 5*(1), 123-155. https://doi.org/10.5281/zenodo.4452944

Eric Elliott, E. (2017). *Programming JavaScript applications.* O'Reilly Media. Obtenido de https://pepa.holla.cz/wp-content/uploads/2016/08/Programming-JavaScript-Applications.pdf

Flanagan, D. D. (2020). *JavaScript: The definitive guide.* O'Reilly Media 7th ed. Obtenido de https://forum.freemdict.com/uploads/short-url/lI79nDbeiEq7d3M6TUUfZGEP6zc.pdf&ved=2ahUKEwivv7ruudyUAxXNpLAFHWThJBEQFnoECCMQAQ&usg=AOvVaw1kQS_Bc608nYExhGs0pbzQ

Fowler, M. M. (2018). *Refactoring: Improving the design of existing code.* Addison-Wesley (2nd ed.). Obtenido de https://github.com/pratik0197/books/blob/master/software-dev/Refactoring%20%20Improving%20the%20Design%20of%20Existing%20Code.pdf

Gómez, S. L. (15 de Marzo de 2024). *Importancia del desarrollo modular en aplicaciones React*. Obtenido de https://keepcoding.io/blog/desarrollo-modular-en-aplicaciones-react/

Gomila, J. G. (07 de 03 de 2026). *Cómo usar React Hooks como un profesional: patrones, errores comunes y buenas prácticas*. Obtenido de https://cursos.frogamesformacion.com/pages/blog/26-066-react-hooks?srsltid=AfmBOopLhKgVv0PakUzIAepkfOqJrUYE_Tp80CAKkTcHCQYtovkNPCsR

Hostinger. (2013). *Qué es React.* Recuperado de Hostinger. Obtenido de https://react--pdf-org.translate.goog/?_x_tr_sl=en&_x_tr_tl=es&_x_tr_hl=es&_x_tr_pto=tc&_x_tr_hist=true

Huet, P. (14 de Octubre de 2020). *React Hooks: Qué son y qué problemas solucionan*. Obtenido de https://openwebinars.net/blog/react-hooks-que-son-y-que-problemas-solucionan/

Nielsen, J. J. (1994). *Usability engineering.* Morgan Kaufmann. Obtenido de https://www.govinfo.gov/content/pkg/GOVPUB-C13-c6d53b6e12963a6af03c8b21bce1a8c1/pdf/GOVPUB-C13-c6d53b6e12963a6af03c8b21bce1a8c1.pdf

Norman, D. D. (2023). *The design of everyday things.* Revised and expanded ed., Basic Books. Obtenido de https://vpb.smallyu.net/%5BType%5D%20books/The%20Design%20of%20Everyday%20Things%20-%20Don%20Norman.pdf

Olawanle, J. (17 de Enero de 2025). *Domina el Renderizado Condicional en React: Una Inmersión Profunda* . Obtenido de https://kinsta.com/es/blog/react-renderizado-condicional/

Pluralsight. (2024). *Understanding links in ReactJS.* Recuperado de Pluralsight. Obtenido de https://www.filestack.com/wp-content/uploads/2024/10/React-Ebook-Filestack.pdf

React. (2024). *UNIT IV ADVANCED CLIENT SIDE PROGRAMMING.* Recuperado de ReactDOM Components Link. Obtenido de https://petengg.ac.in/department/MCA/notes/Regulation%202021/Sem%202/MC4201%20FSWD/UNIT%204%20FS%20Notes.pdf

Sanchez, A. (01 de Octubre de 2025). *React Router: Navegación y Enrutamiento Esencial*. Obtenido de https://4geeks.com/es/lesson/rutas-dinamicas-y-parametros-de-ruta

Sastre, A. (04 de 05 de 2026). *Instalar React y crear nuevo proyecto*. Obtenido de CertiDevs: https://certidevs.com/tutorial-react-instalacion-creacion-nuevo-proyecto

Sommerville, I. I. (2016). *Software engineering.* Addison Wesley - Pearson (10th ed.). Obtenido de https://engineering.futureuniversity.com/BOOKS%20FOR%20IT/Software-Engineering-9th-Edition-by-Ian-Sommerville.pdf

Viera, U. (04 de Septiembre de 2024). *Domina el Hook useEffect en React*. Obtenido de Sistemas y Fullstack Developer autodidacta: https://urianviera.com/reactjs/domina-el-hook-useeffect