S - 5. DOM y componentes dinámicos accesibles

UNIVERSIDAD TECNOLÓGICA DE TEHUACÁN

NOMBRE DEL ALUMNO:

- Marco Antonio Aguilar Castillo
- Cleber Antonio Bolaños Moreno
- Osbaldo Alvarez Martinez
- Kevin Ricardo Simon Alfaro
- Samuel Jonathan Trujillo Bolaños

MATERIA:

- Desarrollo Web Profesional

PROFESOR:

- José Miguel Carrera Pacheco

GRADO Y GRUPO:

8° “B”

Integrante	Rol	Tarea Realizada	Objetivo
Osbaldo (TL)	Tech Lead	Boilerplate Next.js, Configuración de jsx-a11y y README de estándares.	Gestión de Foco (Teoría) y Liderazgo.
Marco	Backend	Documento sobre Virtual DOM vs Real DOM y lógica de useState.	Investigación y Manipulación del DOM.
Kevin	DevOps	Dockerfile y GitHub Actions (Pipeline de CI/CD).	CI/CD Pasando.
Cleber	Frontend	Componentes dinámicos y Tests unitarios con Jest/RTL.	Comp. Dinámicos y Tests Frontend.
Samuel	QA	Reporte Lighthouse y video de navegación por teclado.	Evidencia de Foco Correcto.

Reconciliación en React con Virtual DOM y Gestión de Foco en Next.js

Reconciliación y Virtual DOM en React

¿Qué es el Virtual DOM y por qué React lo usa?

El Virtual DOM es una representación ligera del DOM real, mantenida en memoria. Básicamente, es un árbol de objetos JavaScript que refleja la estructura de la interfaz de usuario, pero sin la carga de los nodos reales del navegador. Cuando el estado o las props de un componente cambian, React ejecuta de nuevo la función render() del componente para producir un nuevo árbol de elementos (Virtual DOM). Luego compara este árbol virtual recién generado con el árbol virtual anterior para determinar qué ha cambiado. A este proceso de comparación y cálculo de diferencias se le llama reconciliación. La reconciliación sincroniza los cambios del DOM virtual con el DOM real del navegador. Gracias a este enfoque, React puede calcular todos los cambios en memoria y aplicarlos en lote al DOM real, en lugar de actualizar el DOM directamente ante cada pequeña modificación. Esto mejora significativamente el rendimiento, ya que las manipulaciones directas del DOM son costosas en comparación con operaciones en memoria.

Virtual DOM vs DOM real: Trabajar con el DOM real del navegador es relativamente lento porque cada cambio puede desencadenar recalculos de estilo, reflujo de diseño y repintado visual. En cambio, operar sobre el Virtual DOM (objetos JS) es muy rápido. React aprovecha esto haciendo todos los cálculos de actualización en el Virtual DOM primero, evitando interacciones directas con el DOM real hasta saber exactamente qué debe cambiar. En otras palabras, React calcula el mínimo conjunto de modificaciones necesarias y luego las aplica al DOM real, reduciendo al mínimo las manipulaciones directas. Así logra interfaces más eficientes y rápidas, incluso a medida que la aplicación crece en complejidad.

Proceso de diffing: cómo React decide qué actualizar

Actualizar eficientemente significa cambiar solo lo necesario. React implementa un algoritmo de diferencias (diffing) optimizado (de complejidad aproximada O(n)) basado en dos suposiciones principales:

1. Elementos de distinto tipo producen árboles totalmente diferentes. Si el tipo de nodo ha cambiado, React no intenta una actualización granular: descarta ese subárbol viejo y monta uno nuevo desde cero. Por ejemplo, si un componente retorna en un render y en el siguiente, React tirará abajo el nodo y todos sus hijos, y creará en su lugar el nuevo con sus hijos. Toda la porción anterior del DOM se desmonta (ejecutando métodos de ciclo de vida de desmontaje si aplican) y se pierde su estado, mientras que la nueva se monta limpia.

2. Elementos del mismo tipo pueden reaprovecharse. Si el elemento virtual nuevo tiene el mismo tipo que el anterior, React asume que puede actualizarlo en lugar de reemplazarlo por completo. En ese caso, se conserva el nodo DOM subyacente y solo se ajustan los atributos o propiedades que hayan cambiado. Por ejemplo, supongamos que antes teníamos:

Y tras un update, el componente retorna:

Ambos son un ; React comparará sus atributos. Detectará que solo cambió la propiedad className de "before" a "after", mientras que title sigue igual. Entonces no creará un nodo nuevo, sino que mantendrá el mismo en el DOM y simplemente modificará su atributo className. El resultado es que la actualización en el DOM real se reduce a cambiar esa clase, en lugar de reconstruir todo el elemento. Del mismo modo, si actualizamos estilos, React cambia solo las propiedades de estilo que difieren. Tras actualizar los atributos de un nodo existente, React procede recursivamente a conciliar sus hijos.

1. Componentes del mismo tipo conservan su instancia. Cuando un elemento JSX representa el mismo componente (por ejemplo <MiComponente /> antes y después), React reutiliza la instancia previa del componente en lugar de crear una nueva. Esto significa que el estado interno del componente se preserva entre renders. React actualizará las props en la instancia existente y volverá a llamar a su método render() (o a la función de componente) para obtener un nuevo árbol de elementos hijos. Luego reconcilia la salida anterior y la nueva de ese componente usando el mismo algoritmo de diferencias. Al mantener la misma instancia, se evitan desmontajes innecesarios y se mejora la continuidad (por ejemplo, el estado no se reinicia al re-renderizar un componente con los mismos tipos).

2. Comparación de listas y prop key. Cuando hay listas de elementos hijo, React por defecto difiere posición por posición. Si agregamos un elemento nuevo al final de una lista, React lo detecta fácilmente: los primeros elementos coinciden uno a uno y el nuevo se inserta al final sin rehacer los anteriores. Sin embargo, insertar o quitar elementos al inicio o en medio de una lista puede ser problemático sin una pista adicional. Por ejemplo, cambiar una lista de ['A','B'] a ['X','A','B'] ingenuamente haría que React viera diferencias en todos los índices: podría desmontar A y B y volverlos a montar en otras posiciones. Para evitar este desperdicio, React usa la propiedad especial key en los elementos de lista. Las keys permiten identificar de forma única a cada elemento independientemente de su posición. Con keys, React empareja elementos antiguos y nuevos por key, detectando qué elementos son los mismos movidos de lugar, cuáles son nuevos y cuáles desaparecieron. En el ejemplo anterior, si A y B mantienen

sus keys, React reconocerá que siguen presentes y solo X es nuevo al inicio, por lo que insertará X sin tocar los nodos de A y B. El uso correcto de keys hace la reconciliación de listas mucho más eficiente y predecible.

Aplicando los cambios al DOM real: Una vez identificadas las diferencias mediante el proceso de diffing, React genera la lista mínima de operaciones necesarias para actualizar el DOM real (proceso de patching). Estas incluyen crear, eliminar o cambiar nodos, y ajustar sus atributos o contenido textual. React entonces aplica estas operaciones de manera agrupada al DOM real, típicamente en el paso de “commit” de su renderizado. Gracias a este enfoque, solo se modifican en el DOM las partes que realmente cambiaron, evitando tocar nodos que permanecen iguales. En consecuencia, la interfaz se actualiza con el mínimo trabajo necesario, lo que mejora el rendimiento y evita relayouts costosos de más. En resumen, React utiliza el Virtual DOM y la reconciliación para difundir eficientemente los cambios de estado a la UI, garantizando que cada renderizado calcule las diferencias y solo toque el DOM real donde sea imprescindible.

```
PS C:\Users\clebe\OneDrive\Imágenes\pacre\conecta-tehuacan> cd apps/frontend
PS C:\Users\clebe\OneDrive\Imágenes\pacre\conecta-tehuacan\apps\frontend> npm test

> frontend@0.1.0 test
> jest

 PASS  src/hooks/__tests__/useFocusManagement.test.js
 PASS  src/app/(auth)/login/__tests__/page.test.js

Test Suites: 2 passed, 2 total
Tests:       24 passed, 24 total
Snapshots:   0 total
Time:        1.973 s, estimated 2 s
```

```
PS C:\Users\clebe\OneDrive\Imágenes\pacre\conecta-tehuacan\apps\frontend> npm run test:watch

> frontend@0.1.0 test:watch
> jest --watch
 PASS  src/app/(auth)/login/__tests__/page.test.js
  Login Component - Focus Management
    √ debe renderizar los botones de Candidatos y Dashboard (25 ms)
    √ el botón de Candidatos debe tener foco automáticamente al cargar (9 ms)
    √ al hacer clic en el botón Candidatos, el foco debe permanecer en él (37 ms)
    √ al presionar Tab, el foco debe moverse del botón Candidatos al Dashboard (6 ms)
    √ al presionar Shift+Tab, el foco debe retroceder (16 ms)
    √ al presionar ArrowDown, el foco debe moverse al siguiente botón (9 ms)
    √ al presionar ArrowUp, el foco debe moverse al botón anterior (7 ms)
    √ al presionar Enter en el botón Candidatos, debe llamar a switchToCandidateView (7 ms)
    √ al presionar Enter en el botón Dashboard, debe llamar a switchToRecruiterView (7 ms)
    √ la navegación debe ser cíclica (Tab en Dashboard vuelve a Candidatos) (14 ms)
    √ los botones deben tener atributos aria-label para accesibilidad (2 ms)
    √ el contenedor debe capturar los eventos de teclado (5 ms)

Test Suites: 1 passed, 1 total
Tests:       12 passed, 12 total
Snapshots:   0 total
Time:        1.666 s, estimated 2 s
Ran all test suites related to changed files.
```