Componentes Interactivos

Pero hay componentes que no se pueden renderizar en el servidor. Un contador con un botón, una descripción que puede colapsar, un menú, etc. Todos estos componentes tienen *estado*, que puede cambiar en el cliente según el usuario interaccione con ellos.

Componentes de Cliente

Los componentes de cliente tienen estado, que vive en la memoria del navegador, y utilizan un ciclo típico en interfaces gráficos: 1) el usuario provoca un evento, 2) la función encargada de gestionar el evento cambia el estado (o "modelo"), 3) se repinta la pantalla utilizando el estado.

Los componentes de React hacen fácil la programación del punto 3), porque evitan tener que mutar el DOM, ya que generan una descripción de "qué debería haber en la pantalla" (especificación declarativa) y luego React se encarga de hacer los mínimos cambios para satisfacer esa descripción. El JSX, de hecho, genera objetos del Virtual DOM, que son los que se usarán luego para cambiar el DOM real.

Para marcar un componente como cliente, hay que poner, arriba de todo del fichero del componente la directiva:

```
"use client";
```

Esto le indica a NextJS que el componente se renderizará en el cliente, y por tanto hay que preparar el código para que se ejecute allí (enviarlo desde el servidor y ejecutarlo en el cliente). Los componentes de tipo cliente serán algo más lentos en reaccionar una vez cargada la página, pero ese es el precio de la interactividad. En realidad los componentes de cliente se ejecutan dos veces, una en el servidor para poder enviar HTML, y luego en el cliente otra vez para reconstruir el estado (esto se llama "hidratación").

Un componente cliente no puede cargar datos directamente, como hemos hecho en los componentes de servidor, así que no puede ser una función asíncrona. La carga de datos de un componente de tipo cliente se hará en el navegador, así que los métodos son distintos.

Hooks

Los hooks son funciones que React ofrece para proporcionar utilidades a los componentes de cliente y que éstos sigan siendo funciones (hace tiempo eran clases mucho más complejas). Hay hooks para: estado, efectos secundarios en funciones, valores de contexto, referencias al DOM, memoización, etc.

`useState` Para que un componente tenga estado, usaremos `useState`. Este *hook* reserva espacio para un valor Javascript que React guarda y gestiona por nosotros y que el componente puede obtener cuando se ejecuta.

Normalmente un componente se ejecutará cada vez que haya que "re-pintar" la pantalla, así que puede ser que se ejecute muchas veces. Cada vez que se ejecuta, el estado será el mismo, porque React lo guarda y lo proporciona como respuesta a la llamada a `useState`.

`useState` devuelve dos cosas: el estado en sí (un valor javascript), y una función que permite cambiar el estado. La función también tiene el propósito de avisar a React, ya que un cambio de estado debe producir un repintado del componente. La función de cambio de estado puede recibir un valor directo, pero cuando hay que basar el valor nuevo en el anterior, es importante pasar una función que calculará ese cambio, ya que muchos cambios de estado se pueden secuenciar de una manera inesperada y los cálculos pueden salir mal.

Ejemplos:

1. Un componente con estado `boolean` que pone sí o no y cambia al clicar:

```
"use client";

export const YesNo = () => {
  const [yes, setYes] = useState<boolean>(false);
  return (
    <div onClick={() => setYes((prev) => !prev)}>
      {yes ? "yes" : "no"}
    </div>
  );
};
```

2. Un contador, que se incrementa al apretarlo:

```
"use client";

export const Counter = () => {
  const [count, setCount] = useState<number>(0);
  return (
    <button onClick={() => setCount((x) => x + 1)}>
      You have clicked me {count} times
    </button>
  );
};
```

La función `setYes` o `setCount`, que cambian el estado, pueden recibir un valor directo o bien una función que será la que efectúe el cambio, quizás usando el valor anterior. En ambos casos esto es relevante porque `setYes` pasa de `true` a `false` y viceversa y `setCount` incrementa el contador en 1.

`useEffect` `useEffect` se utiliza para producir "efectos secundarios" al pintado del componente. Los típicos casos son:

1. Cargar datos la primera vez que el componente se muestra.
2. Suscribir-se (y más adelante cancelar la suscripción) a algun tipo de evento para mostrar datos en tiempo real.
3. Recalcular o recargar datos en función de cambios en los `props`.

`useEffect` recibe 2 parámetros:

- Una **función** `func` a ejecutar (una clausura del entorno del componente) que es la que tiene "efectos secundarios" (que rompe la pureza del comopnente).
- Un **array** con aquellas variables que, al cambiar, deberían provocar una nueva ejecución de `func`. Este array tiene 2 casos especiales:
  - `[]`: No hay dependencias, `func` solo se ejecutará la primera vez que el componente se ejecute.
  - `undefined` (no hay 2o parámetro): `func` se ejecuta cada vez que el componente se ejecute (*mucho cuidado con este caso*, puede provocar bucles infinitos!).

Ejemplos:

1. Cargar datos cuando el componente se muestra por primera vez:

```
"use client";

export const UserList = () => {
  const [users, setUsers] = useState<any[] | null>(null);

  useEffect(() => {
    fetch("https://randomuser.me/api/?results=20")
      .then((response) => response.json())
      .then(({ results: users }) => setUsers(users));
  }, []); // <-- Array vacío, cargamos datos una vez.

  if (users === null) {
    return <div>Loading...</div>;
  }
  return (
    <ul>
      {users.map((user) => (
        <li key={user.email}>
          {user.name.first} {user.name.last}
        </li>
      ))}
    </ul>
  );
};
```

2. Lista de datos donde la búsqueda viene como un `prop`:

```
"use client";

export const Search = ({
  search,
}: {
  search: string;
}) => {
  const [results, setResults] = useState<any[] | null>(
    null
  );

  useEffect(() => {
    fetch(
      `https://api.punkapi.com/v2/beers?beer_name=${search}`
    )
      .then((response) => response.json())
      .then((results) => setResults(results));
    setResults(null); // Limpiar resultados anteriores
  }, [search]);

  if (results === null) {
    return <div>Loading...</div>;
  }
  return (
    <div>
      {results.map((result) => (
        <div key={result.id}>{result.name}</div>
      ))}
    </div>
  );
};
```

El formulario de búsqueda estaría en un componente más arriba en el árbol de componentes, y a éste se le pasaría el texto a buscar. Cuando el texto cambia, el componente se ejecuta nuevamente con el prop `search` cambiado y entonces `useEffect` relanza la consulta porque `search` aparece como dependencia en el array. Esto mostrará al cabo de un rato resultados distintos. El `setResults(null)` dentro de `useEffect` resetea los resultados en el mismo momento en que se sabe que `search` es otro.

3. Un reloj que se crea un timer (y lo elimina) para ir mostrando la hora actual:

```
"use client";

const pad = (n: number) => String(n).padStart(2, "0");
```

```
export const Clock = () => {
  const [time, setTime] = useState<Date>(new Date());

  useEffect(() => {
    const id = setInterval(() => {
      setTime(new Date());
    }, 1000);
    return () => clearInterval(id);
  }, []);

  const h = time.getHours();
  const m = time.getMinutes();
  const s = time.getSeconds();
  return (
    <div>
      {pad(h)}:{pad(m)}:{pad(s)}
    </div>
  );
};
```

La función `func` que se pasa a `useEffect` devuelve otra función que es la que "deshace" las cosas que ha hecho `func`, en este caso colocar un "interval", que produce un aviso cada segundo. Si no se hace eso, el evento del "interval" seguirá ocurriendo aún cuando quizás el componente no esté visible (que genera un error de ejecución).

`useContext`

Este hook permite instalar un valor en un elemento del árbol de componentes que será visible a todos los hijos del árbol. Esto es útil cuando hay partes del estado/modelo que afectan muchos componentes dispersos (por ejemplo el modo "dark", o el usuario de la aplicación). Esto también permite no tener que ir pasando props hacia los componentes hijos y evitar algo que se denomina el "prop-drilling".

Para usar `useContext` hay que crear primero un contexto con `createContext`:

```
// UserContext.tsx
"use client";

import { createContext } from "react";

type UserContextType = {
  user?: string,
};

export const UserContext =
```

```
  createContext < UserContextType > {};

export function UserProvider({
  children,
}: {
  children: React.ReactNode,
}) {
  return (
    <UserContext.Provider value={{ user: "pauek" }}>
      {children}
    </UserContext.Provider>
  );
}
```

El componente `UserContext` no es esctrictamente necesario pero ayuda a reducir un poco el código necesario para instalar un contexto.

El `UserProvider` se puede instalar en el `Layout`:

```
<html lang="en">
  <body className={inter.className}>
    <UserProvider>{children}</UserProvider>
  </body>
</html>
```

Y luego en cualquier componente de esa página (y que por tanto está debajo del `UserProvider` en el árbol de componentes) se usa así:

```
// CurrentUser.tsx
"use client";

import { UserContext } from "@/lib/user-context";
import { useContext } from "react";

export default function CurrentUser() {
  const { user } = useContext(UserContext);
  return <p>El usuario es {user}</p>;
}
```

Tanto el `UserProvider` como el `CurrentUser` son componentes de cliente, pero el resto pueden ser componentes de servidor y Next ya se encarga de renderizar cada componente donde toca.