Componentes en React 18.2

Abstracción de elementos del DOM

En HTML los elementos son limitados. Y si queremos hacer elementos compuestos definidos por nosotros para crear elementos más grandes y poder hacer aplicaciones con ellos? La "abstracción de procedimientos" se traduce aquí a la "abstracción de elementos".

Incrustar el resultado de funciones

Podemos hacer funciones nuevas que devuelvan trozos más grandes de UI:

```
const Hola = () => <div>Hola majete!</div>;
```

Si tenemos la función `Hola`, para usarla podemos

```
<div>{Hola()}</div>
```

En realidad `Hola` es un **componente React**. Un componente en React es simplemente una función. Pero se suele llamar así:

```
<div>
  <Hola />
</div>
```

El hecho de usar `Hola` como un tipo de elemento nuevo en el JSX es una simple traducción de la llamada a la función.

Convención Importante: Los componentes del DOM normal conservan su nombre en minúsculas (`div`, `span`, `h1`, `p`, `input`, ...). Los componentes React **empiezan todos en mayúsculas** (`Hola`, `Header`, `HomePage`, `UserList`, etc.). El hecho de empezar en mayúsculas se usa en JSX para discernir elementos del DOM clásicos o bien componentes hechos por el usuario.

Parámetros ("props" o propiedades)

Como a todas las funciones, cómo pasamos parámetros? Una función que actúa como componente `React` recibe **un solo parámetro `props`**, por definición:

```
const Hola = (props) => <div>Hola majete!</div>;
```

El parámetro props contiene como campos todos los atributos del elemento Virtual DOM:

```
<div>
  <Hola nombre="Pepito" efusivo={true} />
</div>
```

hace que `props` sea `{ nombre: "Pepito" }`, así que podemos cambiar el componente hola así:

```
const Hola = (props) => (
  <div>
    Hola {props.nombre}
    {props.efusivo && "!"}
  </div>
);
```

Es una traducción de la llamada:

Función normal	`Hola({ nombre: "Pepito", efusivo: true })`
JSX	`<Hola nombre="Pepito" efusivo={true} />`

Destructuring de props

Dado el *destructuring* en Javascript, en realidad podemos escribir el componente más claramente:

```
const Hola = ({ nombre, efusivo }) => (
  <div>
    Hola {nombre}
    {efusivo && "!"}
  </div>
);
```

Usando tipos Typescript para los props sería:

```
interface HolaProps {
  nombre: string;
  efusivo: string;
}
const Hola = ({ nombre, efusivo }: HolaProps) => (
  <div>
    Hola {nombre}
    {efusivo && "!"}
  </div>
);
```

`children`

Cómo hacemos componentes que puedan tener otros dentro? Usando una propiedad estándar que es `children`.

Dado un componente utilizado así:

```
<Box>
  <p>I am inside the box!</p>
</Box>
```

La forma de obtener el "contenido" de `Box`, es decir, los elementos contenidos dentro de éste es:

```
const Box = ({ children }) => (
  <div className="bg-red-500 border">{children}</div>
);
```

Es decir, que en el código de más arriba, `children` es `<p>I am inside the box!</p>`.

Combinación de Componentes

Usando `React`, al tener componentes podemos cumplir con el principio DRY. Partiendo de los siguientes datos:

```
const usuarios = [
  { id: 1, name: "James", age: 13 },
  { id: 2, name: "Marcia", age: 12 },
  { id: 3, name: "Paul", age: 14 },
  { id: 4, name: "Henrich", age: 10 },
];
```

En vez de escribir una página entera así:

```
<html>
  <head></head>
  <body>
    <header>
      <h1>Lista de Usuarios</h1>
    </header>
    <ul>
      {usuarios.map((user) => (
        <li key={user.id}>
          {user.name} ({user.age})
        </li>
      ))}
    </ul>
  </body>
</html>
```

La organizaremos en partes y subpartes, cada una con su componente:

```
const Header = () => (
  <header>
    <h1>Lista de Usuarios</h1>
  </header>
);

const UserList = ({ users }) => (
```

```
  <ul>
    {users.map((user) => (
      <li key={user.id}>
        {user.name} ({user.age})
      </li>
    ))}
  </ul>
);

const Page = () => (
  <html>
    <head></head>
    <body>
      <Header />
      <UserList users={usuarios} />
    </body>
  </html>
);
```

Componentes como piezas de la aplicación

Dada cualquier aplicación, no es difícil explorar su interfaz gráfica y deducir qué partes son componentes usando dos pistas:

- **Partes autocontenidas de la pantalla** que se corresponden con un concepto entero (y por tanto cuyos datos provienen de un tipo de objeto muy claro). Por ejemplo, un menú, un pie de página, una figura o un *card* que representa un elemento en una lista.
- **Repetición** de esas partes. Aun cuando esto no es un requerimento, los componentes más útiles suelen utilizarse varias veces y eso es lo que permite

El interfaz es una función de los datos

React proporciona ayuda para construir todo el interfaz de una aplicación como una función pura de los datos de entrada. En general, los datos puros de una aplicación se denominan el **modelo**, así que la fórmula sería:

```
UI = f(model)
```

La interpretación que se puede hacer es que lo que se ve en la pantalla son los datos representados de una forma más fácil de digerir para los usuarios, pero son una "versión más concreta" de éstos. La versión más abstracta son los datos puros, que no tienen nada más.

React Server Components

Los componentes `React` pueden obtener datos desde los props, pero en última instancia habrá siempre un sistema de almacenamiento: ficheros, una base de datos, o una API (que en realidad tiene ficheros o una base de datos detrás).

Cómo podemos rellenar nuestros componentes `React` con esos datos?

Ejecutar componentes en el servidor (NextJS 13+)

En un proyecto NextJS 13+ (casi ningún otro framework tiene esto aún!), los componentes `React` son, por defecto, de servidor (React Server Components, o RSC). Es decir, que su código se ejecuta en el servidor y éste traduce los elementos creados a HTML.

Eso permite que los componentes sean asíncronos y podamos leer desde fuentes diversas:

1. Texto cargado de ficheros:

```
import { readFile } from "fs/promises";

export default async function Page() {
  const fileContents = await readFile("./document.txt");
  const text = fileContents.toString();

  return (
    <main>
      <h1>Some text</h1>
      <p>{text}</p>
    </main>
  );
}
```

2. Lista de usuarios (aleatorios, de prueba) de una API:

```
export default async function Page() {
  const response = await fetch("https://randomuser.me/api/?results=20");
  const { results: userList } = await response.json();

  return (
    <main>
      <h1>User List</h1>
      <ul>
        {userList.map(({ email, name }) => (
          <li key={email}>
            {name.first} {name.last}
          </li>
        ))}
```

```
        </ul>
      </main>
    );
  }
```

Por defecto NextJS 13+ intentará, al máximo posible, renderizarlo todo en el servidor, porque ahorra tiempo de cálculo y el envío del código Javascript neceseario para hacer el renderizado.