En esta parte vamos a olvidarnos de la parte del servidor y base de datos y vamos a centrarnos más en la parte del HTML, CSS y JS en el navegador.

Crear aplicación de React

Para crear una aplicación React usaremos el comando **npx**. Es un comando parecido a **npm** pero **npx** no guarda nada en nuestro ordenador: descarga los paquetes necesarios, los ejecuta y los borra.

Podríamos descargar e instalar el paquete **create-react-app** en nuestro ordenador y cada vez que queramos crear una nueva aplicación lo ejecutamos. El problema con hacerlo de este modo es que todas las aplicaciones que creemos de este modo tendrán la versión de React que estuviera disponible en el momento de descargarlo: si hay actualizaciones o correcciones, tendríamos que descargar e instalar de nuevo el paquete. Usando **npx** nos evitamos esos pasos: **npx** descarga automáticamente la última versión disponible.

Creamos una carpeta vacía y la abrimos en Visual Studio Code. Abrimos la terminal y escribimos **npx create-react-app**. Después de **npx create-react-app** ponemos la ubicación donde queremos guardar los archivos de la aplicación, en este caso hemos puesto un punto "**.**" para indicarle que nos los guarde en el directorio en el que nos encontramos actualmente.

Importante que la carpeta donde estamos guardando los archivos NO tenga espacios ni mayúsculas o caracteres raros, ya que nos dará error.

Esperamos a que descargue todos los ficheros (puede tardar varios minutos).

Contenido de una aplicación React

En la carpeta **src** tenemos el código de nuestra aplicación. El archivo **App.js** es un archivo JavaScript pero si lo miramos veremos que tenemos código HTML en él, escrito sin comillas.

Al principio del archivo vemos que hay unas líneas que dicen '*import* React *from* 'react''. Esta es la manera que tenemos de importar módulos de Node en JavaScript moderno. Es lo mismo que escribir **const React = require('react');** pero con más funcionalidades.

Este módulo **React** que hemos importado es lo que nos permite utilizar **JSX**: es la sintaxis que usaremos para escribir HTML en un archivo JS. De este modo, podemos poner etiquetas HTML sin usar comillas.

En el archivo index.html tenemos un elemento con id 'root'. En el archivo index.js importamos App del archivo App.js tenemos una función **ReactDOM.render(<App /**
>,document.getElementById('root')) a la que le decimos que nos pinte lo que tenemos en <App> en el elemento con id 'root' del index.html. En el archivo **App.js** tenemos el código que vamos a querer modificar para nuestra aplicación y exportamos el código que hemos escrito mediante **export default** App.

Realmente con React no vamos a hacer nada que no podamos hacer usando HTML y JS en el navegador, pero sí que nos facilitará mucho la tarea. Para ver las similitudes, podríamos crear un archivo HTML con lo siguiente:

```
<div id=”contenido”></div>

<script>
function crearElemento(tipo, contenido){
let cadena = `<${tipo}>${contenido}</${tipo}>`;
return cadena;
}
let html = crearElemento(‘p’,’hola mundo’);
document.getElementById(‘contenido’).innerHTML =html;

</script>
```

Tenemos una función que recibe un tipo y el contenido. Esta función crea una cadena de texto en la que ponemos etiquetas HTML del **tipo** que llega como parámetro y su contenido será el **contenido** que le llega como parámetro. Este sería el comportamiento de React.

App.js es un archivo JavaScript en el que podemos escribir código JavaScript normalmente. Si queremos agregar código JavaScript (el valor de una variable, el valor de un elemento dentro de un array o un objeto, el resultado de una operación…) al código HTML lo haremos usando **{ }:**

```
function App(){
let persona = {nombre: “Peio”, edad: 33};
return (
<div>
<p>esto es el título</p>
</div>
<div>
<ul>
                <li>{persona.nombre}</li>
        <li>uno</li>
        <li>dos</li>
        <li>tres</li>
</ul>
</div>
)
```

En el primer elemento de la lista se mostrará el valor que tenemos en la propiedad “**nombre**” del objeto “**persona**”.

PÁGINA SENCILLA EN REACT

Cuando escribimos código HTML en React, todas los elementos que añadimos tienen que estar envueltos en una sola etiqueta. El siguiente código nos daría error:

```
App.js
function App() {
  return (
```

```
<header>
    <h1>La Pata</h1>
</header>
<main>
    <p>
      Lorem ipsum dolor sit amet consectetur, adipisicing elit. Aliquid est saepe quam, at doloribus iusto temporibus fugit veniam maxime ipsam dolorem nobis aut soluta nesciunt explicado inventore harum reprehenderit eum!
    </p>
</main>
<footer>
    <p>$copy; 2020</p>
</footer>
}
```

Los elementos header, main y footer están al mismo nivel, cosa que no nos permite React. Tendremos que poner una etiqueta HTML que envuelva a todos ellos. Podríamos usar un para ello, pero eso se vería reflejado en nuestra página (acabaríamos con muchos div anidados y puede que no queramos que quede así). Para ello, podemos envolver esos elementos en etiquetas especiales de JSX: <React.Fragment></React.Fragment> o <></>. Son etiquetas que usamos para envolver elementos JSX pero luego no se traducen en etiquetas HTML en nuestra página final.

App.js
```
function App() {
 return (

<>
 <header>
    <h1>La Pata</h1>
 </header>
 <main>
    <p>
      Lorem ipsum dolor sit amet consectetur, adipisicing elit. Aliquid est saepe quam, at doloribus iusto temporibus fugit veniam maxime ipsam dolorem nobis aut soluta nesciunt explicabo inventore harum reprehenderit eum!
    </p>
 </main>
 <footer>
    <p>$copy; 2020</p>
 </footer>
</>
}
```

React nos permite crear nuestra aplicación o página por fragmentos o secciones. En vez de tener todo el código de nuestra página dentro de la función App(), podemos crear diferentes funciones por cada una de las secciones de nuestra página.

Por ejemplo, podemos tener una función Cabecera() que devuelva código JSX:

App.js
```
function Cabecera() {
```

```
  return (
    <header>
      <h1>La Pata</h1>
    </header>
  )
}
```

Y en la función App, si queremos indicar que queremos usar la cabecera lo escribiremos de la siguiente manera:

App.js

```
function App() {
  return (
    <>
  <Cabecera />
      <main>
        <p>
          Lorem ipsum dolor sit amet consectetur, adipisicing elit. Aliquid est saepe quam, at doloribus iusto temporibus fugit veniam maxime ipsam dolorem nobis aut soluta nesciunt explicabo inventore harum reprehenderit eum!
      </p>
      </main>
      <footer>
        <p>$copy; 2020</p>
      </footer>
    </>
  );
}
```

En el lugar donde queremos que se muestre la cabecera, escribimos <Cabecera />. Importante escribir el nombre de la función en mayúscula para que JSX lo interprete correctamente.
Del mismo modo que hemos creado la función Cabecera dentro del archivo App.js, podemos crear esa función en un archivo aparte. Creamos un archivo llamado **Cabecera.js.** En ese archivo importamos React (*import **React** from 'react'*), copiamos la función Cabecera y la exportamos.

Cabecera.js

```
import React from 'react';

function Cabecera() {
  return (
    <header>
      <h1>La Pata</h1>
    </header>
  )
}

export default Cabecera;
```

Ahora, en el archivo **App.js** tendremos que importar la función Cabecera del archivo Cabecera.js

App.js

```
import Cabecera from './Cabecera';
```

Ya podremos usar la función Cabecera en el archivo App.js.

Propiedades

Ya que los componentes son **funciones**, podemos recibir datos en ellos cuando los llamamos. Pero llamamos a los componentes en el JSX:

<Cabecera />

Para pasarle **datos** a la función, podemos utilizar la misma sintaxis que usamos para dar **atributos** a las etiquetas HTML:

```
<Cabecera titulo="Hola mundo" />
```

Cuando React ve que en el JSX hemos pasado datos a un componente, crea un objeto y dentro de ese objeto añade los nombres de los atributos como propiedades (por esto cuando **recibimos** el objeto en la función del componente lo solemos llamar **props**):

```
function Cabecera(props) {
    return (
        <header>
            <h1>{props.titulo}</h1>
        </header>
    );
}
```

Tenemos que hacer que coincida el nombre del **atributo** del componente con el nombre de la propiedad del objeto **props** que sacamos (en este caso titulo), para obtener el valor recibido (en este caso "Hola mundo").

Si quisiéramos pasar más de un valor, tendríamos más de un atributo y podríamos obtenerlos de dentro de props:

<Cabecera titulo="Hola mundo" subtitulo="Un ejemplo de propiedades" />

y dentro de la función Cabecera podemos utilizar **props.titulo** y **props.subtitulo** para obtener los valores.

Siempre que añadimos valores en los atributos, si los ponemos entre comillas (como "Hola mundo") llegarán como Strings dentro del objeto props. Si queremos enviarlos como otro tipo de valores (números, arrays, objetos, …) tenemos que pasarlo al atributo como datos de JavaScript y por lo tanto lo tenemos que envolver entre llaves:

```
<Persona nombre="Iker" edad={40} />
```

Array map

map es una función de array que ejecutamos sobre cada uno de los elementos de ese array,

modificando el valor de cada uno de esos elementos. La sintaxis es la siguiente:

```
nombreArray.map function(elemento){
  return nuevoValor;
};
```

La variable “elemento” que le pasamos es cada uno de los elementos del array original. Dentro de la función, ejecutamos el código que queremos y lo que ponemos en el **return** es el nuevo valor que tomará el elemento del array.

En React, si quisiéramos mostrar todos los elementos de un array dentro de etiquetas HTML, podríamos hacerlo de la siguiente manera:

```
let nombres = ['Iker','Jon','Elena'];
nombres.map function(nombre){
return <h3>{nombre}</h3>
};
```

Una vez tenemos este resultado lo podemos guardar en una variable:

```
let datos = nombres.map(function(nombre) {
        return <h3>{nombre}</h3>;
});
```

y después podemos usar lo que tenemos en la variable datos en la parte de JSX:

```
return (
        <div>
                { datos }
        </div>
);
```

De este modo, podremos convertir arrays de objetos de JavaScript al formato JSX de manera rápida y mostrarlo en la página.