5. React

5.1. Introducción y objetivos

A la hora de desarrollar un *front-end* es muy común utilizar *frameworks* o librerías que faciliten el trabajo del programador en algunos aspectos concretos.

Lo mismo sucede con JavaScript y a su alrededor encontramos dos *frameworks* y una librería que podemos usar para ayudarnos en el desarrollo de una solución *front-end*. Los dos *frameworks* son Angular y Vue.js y la librería es React.

En este tema, nos centraremos en conocer más profundamente la librería React y qué podemos conseguir con ella. La decisión de estudiar React frente a los *frameworks* es la enorme diferencia que encontramos en su curva de aprendizaje (la cantidad de esfuerzo necesario para llegar a sentirse cómodo empezando sin ningún tipo de conocimiento).

Para conocer qué es React y aprender a desarrollar soluciones *front-end* con esta librería, exploraremos los siguientes aspectos:

* JSX.
* Programación orientada a componentes.
* *Hooks*.
* Enrutamiento y navegación.

Todo el código que veremos en este tema ha sido escrito con Visual Studio Code y es recomendable instalar las siguientes extensiones:

* [ES7+ React/Redux/React-Native snippets](https://marketplace.visualstudio.com/items?itemName=dsznajder.es7-react-js-snippets).
* [Simple React Snippets](https://marketplace.visualstudio.com/items?itemName=burkeholland.simple-react-snippets).

<https://github.com/>

No olvides revisar el código de apoyo que usaremos para explicar algunos puntos de este tema en las clases virtuales:

* [Front-End de una pizzería online](https://github.com/UnirCs/front-end-pizza).

5.2. Qué es React

React es una librería desarrollada para el lenguaje JavaScript por Jordan Walke, un ingeniero de software de Facebook. Esta librería fue usada en primer lugar por los desarrolladores de Facebook e Instagram y pasó a ser de código abierto en 2013.

En 2015, se anunció React Native, una versión que permitió (y permite) el desarrollo nativo en Android y iOS. Posteriormente, en 2017, Facebook publicó una nueva versión de la librería que incluía un nuevo algoritmo central llamado React Fiber. Esto hizo que aquella versión se convirtiese en la base de cualquier mejora que ha habido luego. Ese mismo año, el 26 de septiembre, se liberó la versión 16.0 de React al público. Dos años después se lanzó la versión 16.8 que incluyó los famosos *hooks*, los cuales cambiaron totalmente la forma de trabajar con la librería.

Estamos, por tanto, ante una tecnología reciente, ampliamente utilizada y en constante evolución.

Esta librería presenta varias características principales que debemos tener en cuenta:

A blue and white text on a blue background

Description automatically generated

Tabla 1. Características principales de React. Fuente: elaboración propia.

Creación de una aplicación con React

Crear una aplicación con React es realmente sencillo. Para trabajar con esta librería utilizaremos nuestra instalación de Node.js y el gestor de paquetes NPM. Lo primero que debemos hacer es crear un esqueleto sobre el que trabajar posteriormente. Para ello ejecutaremos el siguiente comando en la terminal de Node.js:

npx create-react-app unir-react

cd unir.react

Si obtenemos algún tipo de error, será conveniente revisar que los *scripts* de React incluidos con la instalación de NPM estén actualizados. Esto podemos realizarlo mediante el siguiente comando:

npm install react-scripts@latest

En el caso de que tuviésemos instalada una versión más antigua de Create React App, será necesario desinstalarla mediante uno de estos dos comandos:

npm uninstall -g create-react-app

yarn global remove create-react-app

Tras ejecutar el primer comando ( npx ) el gestor de paquetes NPM comenzará a crear un proyecto «vacío» en el directorio actual de trabajo. Una vez terminado, entraremos a ese directorio y ejecutaremos la aplicación mediante la instrucción *start*.

Si nos situamos dentro del directorio que se ha creado, veremos nuevas carpetas y archivos:

A screenshot of a computer

Description automatically generated

Figura 2. Ficheros generados por el comando Create React App. Fuente: elaboración propia.

Cada uno de esos archivos/directorios tiene un propósito concreto.

A blue and white table with black text

Description automatically generated

Tabla 2. Archivos y su función. Fuente: elaboración propia.

A continuación, vamos a ejecutar la aplicación con el siguiente comando:

npm start

Veremos que automáticamente aparece una nueva pestaña en nuestro navegador con este aspecto:

A blue and white logo

Description automatically generated

Figura 3. Pantalla de inicio de la aplicación con React. Fuente: elaboración propia.

Haremos caso al mensaje que se nos muestra por pantalla y modificaremos ligeramente el archivo App.js que hay dentro de la carpeta  src . Concretamente modificaremos la línea 10 desde Visual Studio Code y guardaremos los cambios. Podemos observar que el contenido de la página ha cambiado sin necesidad de recargar la misma:

A blue and black logo

Description automatically generated

Figura 4. Pantalla de inicio de la aplicación de React tras modificar App.js. Fuente: elaboración propia.

Como habrás podido intuir, React es la librería perfecta para ayudarnos a construir aplicaciones de una sola página (*Single Page Applications o SPAs*).

5.3. JSX y renderizado

JSX

JSX es una extensión de la sintaxis de JavaScript muy utilizada a la hora de trabajar con React. Desde el equipo de React, recomiendan usar esta sintaxis para indicar cómo debe ser la interfaz de usuario.

Un ejemplo de una declaración con JSX es la siguiente (son muy fáciles de reconocer):

const element = <h1>Hola Mundo!</h1>;

No se trata de un *string,* ni de un fragmento HTML. Esa sentencia produce lo que se denomina elementoen React. Si bien no es necesario utilizar JSX, veremos que es muy útil para trabajar con los componentes que se explicarán. Dentro de una sentencia escrita con JSX podemos utilizar llaves ( { } ) para insertar cualquier tipo de expresión en JavaScript. Observemos un ejemplo:

function pintarUsuario(usuario){

return usuario.nombre + ‘ – ‘ + usuario.edad + ‘años’;

}

const jesus = {

nombre: ‘Jesus’,

edad: ‘17’

};

const element = <h1>Hello, {pintarUsuario(jesus)}</h1>

El DOM virtual de React, que es el encargado de modificar finalmente el documento que ve el navegador con los cambios que introducimos mediante JSX, escapa los valores insertados en JSX antes del proceso de renderizado por lo que, en cierto modo, esta funcionalidad previene ataques de tipo XSS (*cross-site-scripting*).

JSX puede utilizarse para renderizar grandes bloques con múltiples etiquetas de HTML. No obstante, siempre se recomienda que los elementos que se creen con JSX sean reusables y que no contengan una estructura innecesariamente compleja. Un ejemplo:

const bloque = (

<div>

<h1>Titulo de Nivel 1</h1>

<h2>Titulo de Nivel 2</h2>

<ul>

<li><h3>Titulo de Nivel 3</h3></li>

<li><h3>Titulo de Nivel 3</h3></li>

<ul>

<img src=‘https://www.unir.net/wp-content/uploads/2021/10/unir-logo-v.jpg’alt=‘unir-img’/>

</div>

);

Renderizado

Hasta ahora nos hemos limitado a descargar código de ejemplo y ejecutarlo para ver qué se acaba mostrando en nuestro navegador, también hemos visto las bases de JSX. Pero ¿cómo se las ingenia React para mostrar en la página todo aquello que escribimos?

El renderizado es, sin duda, una de las partes más importantes cuando trabajamos con React y, en el caso de los proyectos en los que trabajaremos, ocurre gracias al código que podemos encontrar dentro del archivo index.js:

ReactDOM.render(

<React.StrictMode>

<App |>

<| React.StrictMode>,

Document.getElementByld(‘root’)

);

Hacemos uso del DOM virtual de React, ReactDOM, para invocar el método  render , dentro del cual incluimos el componente principal de la aplicación y todos los nodos raíz que manejará la aplicación de React.

Las aplicaciones que están construidas únicamente con React suelen tener un solo nodo raíz, ya que no suele ser necesario disponer de varios nodos de este tipo. Sin embargo, si se estuviese tratando de incorporar React en una aplicación ya desarrollada con otras tecnologías, es posible indicar varios nodos raíz como argumentos a este método  render .

El resultado será que el DOM virtual de React estará atento a todos los cambios que se produzcan dentro de los nodos que se han especificado. Por tanto, si especificamos el nodo cuyo  id  es  root (podemos ver el elemento HTML con este id en el archivo index.html) y no tenemos ningún otro nodo raíz, estaremos indicando que queremos que el DOM de React escuche cambios en cualquier parte de la aplicación.

Gracias a los componentes y al estado de estos, React no necesita recargar la página por completo para mostrar los cambios, sino que actualiza (re-renderiza) únicamente aquellas partes (componentes) que se han visto afectadas por una modificación.

Si quieres saber más sobre JSX, te recomendamos que accedas a la documentación oficial de React disponible en el recurso de A fondo, *JSX en profundidad,* donde podrás ver más en detalle el proceso de traspilación que hay detrás de esta maravillosa sintaxis.

5.4. Programación orientada a componentes

¿Qué son los componentes?

Es imposible trabajar con React y no trabajar orientado a componentes. Los componentes son uno de los elementos clave de esta librería, ya que toda ella está construida sobre este concepto.

Los componentes en React permiten separar la interfaz de usuario en partes totalmente independientes unas de otras con el objetivo de ser reutilizables, llegando a desarrollarse, mantenerse y evolucionar de forma totalmente aislada.

Observemos con atención la siguiente imagen de la conocida web de gifs, [Giphy](https://giphy.com/):

A screenshot of a video game

Description automatically generated

Figura 5. Página principal de Giphy. Fuente: página de Giphy.

¿Has visto algún patrón que se repita en esta imagen? Sí, efectivamente. Hay una barra de búsqueda que, como seguramente podrás intuir, se utilizará en todas las pantallas que podamos encontrarnos en esta página web. También, vemos un menú en la parte superior que estará presente en todas las pantallas y, dentro de este menú, vemos varios elementos de este que también siguen una estilización similar.

A su vez, podemos ver, en la parte derecha, dos botones para crear y subir gifs que, aunque, obviamente, tratan diferentes funcionalidades, lucen exactamente idénticos, salvo por el texto que contienen. Lo mismo podemos decir del botón de *login* situado en el extremo derecho.

A screenshot of a video game

Description automatically generated

Figura 6. Componentes dentro de la página principal de Giphy. Fuente: adaptado de página de Giphy.

Mientras trabajemos con React pensaremos siempre en la idea de componente como un trozo de interfaz totalmente independiente que puede ser reutilizado en diferentes partes de la aplicación. Por tanto, en las aplicaciones que desarrollemos, tendría sentido tener estos componentes (entre otros):

* *Header.*
* *Footer.*
* Menús superiores o laterales.
* Entradas de menú.
* Botones.

De esta forma, React nos permite separar la parte visual y la parte lógica de la aplicación. En el caso de los botones, por ejemplo, es muy común que todos los botones de una aplicación sigan el mismo estilo (aunque realicen diferentes funciones). Gracias a esta metodología es posible separar el código encargado de la interfaz y el encargado de la lógica propia de cada botón.

Accede, si lo deseas, al vídeo *Aprendiendo a identificar componentes de React* para ver más ejemplos de componentes que podemos encontrar en conocidas webs:

Todos los componentes que se verán en este tema son denominados componentes funcionales o *functional components*. Tradicionalmente, los componentes se han creado mediante clases que se extendían de una superclase de React llamada *component*. No obstante, esta forma de trabajar fue la que se adoptó en un primer momento para hacerlo más atractivo a todas aquellas personas que tenían conocimientos de programación orientada a objetos.

Sin embargo, debido a la introducción reciente de los *hooks*, surgió una nueva forma de declarar componentes de una forma más funcional y, por lo tanto, menos farragosa.

Creación de componentes

A continuación, se muestra un ejemplo de un componente funcional de React que sirve para modelar el *header* (muy sencillo y sin demasiado estilo) de una aplicación web:

import React from ‘react’;

import ‘. ./styles/header.css’;

const fechaHoraActual = () =>{

letcurrentdate = new Date();

letdatetime = currentdate.getDate() + “/”

+ (currentdate.getMonth() + 1) + “/”

+ currentdate.getFullYear() + “-”

+ currentdate.getHours() + “:”

+ currentdate.getMinutes() + “:”

+ currentdate.getSeconds();

return datetime;

}

export constHeader = () =>{

return <div className= “unir-header”>

<p><b>UNIR - Header</b></p>

<b>{fechaHoraActual()}</b>

</div>

};

A continuación, pasaremos a analizar las diferentes partes de ese código.

A blue and white text with white text

Description automatically generated

Tabla 3. Diferentes partes del código del *header.* Fuente: elaboración propia.

A continuación, vamos a crear otro componente funcional que servirá como contenedor general para todos los demás componentes que vayamos realizando. Se llamará  MainApp .

import React from ‘react’;

import { Header } from ‘./components/Header’;

export constMainApp = () =>{

return (

<Header|>

)

};

En este caso, el componente funcional que usaremos como contenedor es mucho más sencillo que el que hemos declarado previamente. Como ves, importamos React de la misma forma que ya lo hacíamos, pero ahora incluimos también el componente que hemos creado (e incluiremos todos aquellos que vayamos creando).

En cuanto al código de la función componente, vemos que simplemente devuelve un bloque con el componente que queremos que se pinte.

Hasta este punto tenemos dos nuevos componentes funcionales de React,  MainApp  y  Header , que está incluido dentro del primero. Pero no se están pintando en ningún lado, ya que no le hemos dicho a React que renderice estos componentes. Para hacerlo, modificaremos el archivo index.js, incluyendo nuestro nuevo componente principal  MainApp  en lugar del que viene por defecto en la instalación.

import React from ‘react’;

import ReactDOM from 'react-dom’;

import ‘./index.css’;

import { MainApp } from ‘./MainApp’;

ReactDOM.render(

<React.StrictMode>

<MainApp|>

<|React.StrictMode>

Document.getElementByld(‘root’)

);

De esta forma, le indicamos a React que renderice nuestros nuevos componentes y esté atento a cambios en ellos. El resultado de salvar los archivos (sin ninguna necesidad de recargar la página manualmente) podemos verlo a través del navegador:

A blue background with white text

Description automatically generated

Figura 7. Header construido a partir de los componentes funcionales. Fuente: elaboración propia.

El estilo que hemos utilizado (muy sencillo y rústico para poder ayudarnos a distinguir las diferentes partes de la aplicación en el futuro) es el siguiente:

.unir-header {

background-color: cornflowerblue;

}

.unir-header p {

font-family: fantasy;

font.size: large;

}

*Props*

Los componentes que hemos visto hasta ahora no hacían uso de ningún tipo de argumento para devolver el trozo de interfaz que lo compone a cada uno. No obstante, ya que estamos hablando de componentes funcionales, es decir, expresados como funciones, tiene sentido pensar que sea posible hacer uso de algún tipo de argumento que sea suministrado al componente en su creación. Y así es.

Todo componente funcional puede tener un argumento llamado *props* a partir del cual se puede acceder a todas aquellas propiedades que se hayan indicado a la hora de crear el mismo.

A continuación, veremos cómo cambia el componente  Header  que ya teníamos si usamos las  props que se reciben como argumento:

export constHeader = (props) =>{

return <div className=“unir-header”>

<p><b>UNIR – Header {props.numero}</b></p>

<b>{fechaHoraActualArrow()}</b>

</div>

};

El cambio es tan simple como incluir el argumento en la declaración de la función y acceder a aquellas propiedades que nos interesen. En este caso,  número . No obstante, sería posible refactorizar este código para no tener que estar escribiendo  props  cada vez que queramos referirnos a una de estas propiedades usando la desestructuración:

export constHeader = ({numero}) =>{

return <div className=“unir-header”>

<p><b>UNIR – Header {numero}</b></p>

<b>{fechaHoraActualArrow()}</b>

</div>

};

Ahora que nuestro  Header  es capaz de manejar argumentos de entrada, solo resta modificar también el componente que se encarga de crearlo,  MainApp . Este ahora le dará valor al parámetro número a la hora de crear el *header*. Para ilustrar la diferencia, crearemos dos componentes  Header :

export constMainApp = () =>{

return (

<Headernumero=“1” />

<Headernumero=“2” />

)

};

Habrás notado que, en esta ocasión, no hay ninguna etiqueta en el bloque que agrupa a los dos componentes  Header . Esto es lo que se conoce como *fragment* y lo que se consigue con ellos es que el código HTML resultante no esté dentro de un nuevo  div , sino que se sitúe justo debajo del nodo superior. Lo que se ha escrito es la denominada forma corta.

Existe una forma larga de declarar un *fragment* y es utilizando la etiqueta <React.Fragment>. Recordemosque todo el código HTML resultante se está incluyendo dentro del elemento  root . Por tanto, al renderizar este componente, el resultado sería:

A screen shot of a computer

Description automatically generated

Figura 8. Consecuencia de usar *fragments*. Fuente: elaboración propia.

En el navegador, ahora veríamos dos  Headers  diferentes y cada uno de ellos muestra el valor de la propiedad  número  que recibió en su creación:

A close-up of a blue box

Description automatically generated

Figura 9. Headers construidos usando *props*. Fuente: elaboración propia.

### **5.5. Hooks**

En los últimos años, los desarrolladores detrás de React han descubierto que el paradigma orientado a clases que se usaba en un principio es demasiado farragoso, no favorece la reutilización y, por tanto, constituye un impedimento para el aprendizaje de React.

Aunque React aún soporta la creación de componentes como clases (además de como funciones), los esfuerzos de los desarrolladores pasan por adoptar un modelo más cercano a la **programación funcional.** Es por eso por lo que hemos aprendido a declarar componentes como funciones y el siguiente paso dentro de ese aprendizaje es entender y utilizar los hooks. La traducción al español de hook sería gancho y eso es exactamente lo que hacen estos amigos.

Los hooks permiten «enganchar» los componentes funcionales que desarrollamos con otras características y funcionalidades propias de React (o incluso desarrolladas por nosotros mismos).

React trae en su versión 16.8.0 varios hooks incluidos que, como veremos más adelante, nos ayudarán enormemente a desarrollar nuestras aplicaciones. No obstante, nosotros también podemos construir **nuestros propios** ***hooks*.** En este tema, veremos tres de los hooks incluidos en la distribución (muy usados):  useState ,  useEffect  y  useLocation .

Si lo deseas, puedes obtener más información sobre otros hooks que se incluyen en la distribución de React accediendo a su documentación en el recurso de A fondo, Hooks incluidos en React.

Además, veremos cómo definir hooks personalizados que nos ayuden a reutilizar código y mejorar la legibilidad y presentación del código de nuestras aplicaciones.

### **Manejando el estado con useState**

El hook de estado nos permite definir por cuáles variables está constituido el estado de un componente funcional.

Si cualquiera de las variables que definen el estado del componente **cambia** (y, por tanto, cambia el estado del componente), React modificará su DOM virtual y provocará un renderizado de ese componente únicamente (y no de toda la página).

Para usar el hook de estado debemos hacer uso de la función de React  useState . A continuación, verás el componente  Header  con el que hemos trabajado previamente, pero ligeramente modificado para usar este hook:

import React from ‘react’;

import { useState } from ‘react’;

import ‘. ./styles/header.css’;

export constHeader = ({numero}) =>{

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

console.log(“Renderizado…” + contador);

return <div className=“unir-header”>

<p><b>UNIR – Header {numero}</b></p>

<b>{contador}</b>

<br></br>

<button onClick={() => setContador(

contador + 1)

}>Modificar estado</button>

</div>;

};

Como puedes observar, la función  useState , cuyo argumento es el estado inicial (en este caso, un cero, pues es un simple contador), devuelve un array donde el primer elemento es la variable de estado, contador , y el segundo es una función a través de la cual podremos modificar dicha variable, setContador .

La variable de estado **solo debe ser modificada** a través de la función  setter  que ha devuelto useState  y de ninguna otra forma. Aunque no es obligatorio, para mantener una buena legibilidad, se recomienda que el nombre de la función  setter  esté relacionado con el nombre que le hemos dado a la variable de estado.

En este ejemplo, el estado del componente funcional es un número, pero podría ser cualquier tipo de dato, incluyendo también objetos. Y, además, el estado de un componente **no tiene por qué restringirse a una única variable de estado.** Es decir, podemos usar  useState  tantas veces como queramos dentro de un único componente, pero debemos ser conscientes de que, si el estado cambia, inevitablemente React volverá a renderizar ese componente.

A continuación, vamos a comprobar si efectivamente React cambia únicamente los elementos que debe o si recarga toda la página. Para ello ejecutamos la aplicación de React con la que hemos venido trabajando y veremos algo similar a esto:

A screenshot of a computer

Description automatically generated

Figura 10. Prueba del hook de estado en su estado inicial. Fuente: elaboración propia.

Como podemos observar, tenemos dos componentes  Header  dibujados en la pantalla y cada uno de ellos tiene su propia variable de estado llamada **contador**. Por otro lado, en la consola, vemos que se han impreso dos líneas que indican que el componente se ha renderizado con el valor 0 en su variable de estado. Son dos debido a que tenemos dos componentes  Header .

A continuación, haremos clic sobre el botón **Modificar estado** del componente número 1 y veremos qué ocurre:

A screenshot of a computer

Description automatically generated

Figura 11. Prueba de hook de estado al modificar el estado de un componente. Fuente: elaboración propia.

Como habrás podido observar, **la página no se ha recargado** (esto es algo que tendrás que comprobar haciendo tú mismo el experimento), pero sí que se ha modificado lo que vemos en pantalla. Esto es debido a que React ha detectado un cambio en la variable de estado a través de la función  setContador  y, en ese momento, ha renderizado de nuevo el componente, como podemos comprobar mirando la consola.

Podemos seguir haciendo clic en ambos botones y veremos cómo los cambios en uno de los componentes no afectan al resto de la aplicación, ya que solo se redibuja aquel componente que ha sufrido cambios.

Es muy importante tener en cuenta que la **función** setContador  que utilizamos para modificar el valor de nuestra variable de estado **es** **asíncrona.** Podríamos pensar que se asemeja a otros setters de lenguajes como Java donde, tras ejecutar la operación, el valor de la variable ya se ha modificado. En este caso no es así**. Esta función es una simple petición de actualización que se le hace al** ***core*** **de React,** por lo que no sería seguro tratar de trabajar con el valor de la variable de estado inmediatamente después de haberla modificado a través de esta función, ya que el valor que recibiremos será el que había previamente.

### **Añadiendo efectos con useEffect**

El hook  useEffect  sirve para añadir efectos secundarios en un componente funcional. Más concretamente, ese efecto se llevará a cabo después del correcto renderizado del componente.

Aquí puedes ver el componente  Header  modificado para usar este efecto:

import React, { useEffect } from ‘react’;

import { useState } from ‘react’;

import ‘. ./styles/header.css’;

export constHeader = ({numero}) =>{

const[contadorUno, setContadorUno] = useState(0);

const[contadorDos, setContadorDos] = useState(0);

useEffect(() =>{

console.log(“Efecto de montaje: Solo se ejecuta una vez”)

}, []);

useEfect(() =>{

console.log(“Efecto con cualquier dependencia”)

}, []);

useEfect(() =>{

console.log(“Efecto ligado a contadorUno ha cambiado”)

}, [contadorUno]);

useEfect(() =>{

console.log(“Efecto ligado a contadorDos ha cambiado”)

}, [contadorDos]);

return <div className=“unir-header”>

<p><b>UNIR – Header {numero}</b></p>

<b>{contadorUno}</b>

<br></br>

<button onClick={() =>setContadorUno(contadorUno + 1)}>Modificar contador 1</button>

<br></br>

<b>{contadorDos}</b>

<br></br>

<button onClick={() =>setContadorDos(contadorDos + 1)}>Modificar contador 2</button></div>;};

La función  useEffect  recibe dos argumentos, el primero de ellos es la función (efecto) a ejecutar cada vez que el componente se renderice y el segundo, opcional, es un array de dependencias. **Cualquier efecto se ejecutará al menos una vez,** que es cuando se renderiza el componente por primera vez. A partir de ahí, en función de las dependencias de cada efecto, se ejecutará en unas situaciones u otras.

En el código anterior puedes ver los distintos casos:

* **Sin** ***array*** **de dependencias.** Quiere decir que el efecto se debe ejecutar, además de la primera vez que se renderiza el componente, cada vez que se vuelva a renderizar. Es decir, se ejecutará cuando cualquier variable de estado cambie.
* **Con** ***array*** **de dependencias vacío.** Se conoce como efecto de montaje. Se ejecutará una única vez, tras el primer renderizado del componente.
* **Con variables de estado en el** ***array*** **de dependencias.** Quiere decir que el efecto se debe ejecutar, además de la primera vez que se renderiza el componente, cada vez que se produzca un cambio en la (o las) variable de estado indicada.

Como puedes ver, también hemos añadido un segundo contador dentro del  Header  para poder comprobar si este hook funciona o no. Al igual que en el caso anterior, si ejecutamos la aplicación, veremos algo similar a esto:

A screenshot of a computer program

Description automatically generated

Figura 12. Prueba de  useEffect  en estado inicial. Fuente: elaboración propia.

Si prestamos atención a los mensajes que se muestran en la consola, veremos que, de momento, **todo va según lo esperado** (hay que tener en cuenta que veremos mensajes duplicados inicialmente porque tenemos dos componentes  Header ):

* Los efectos de montaje se han ejecutado una única vez por cada componente.
* Los demás efectos se han ejecutado al menos una vez.

De ahora en adelante, sin importar qué hagamos, los efectos de montaje no deberían volver a ejecutarse nunca. Sin embargo, si tendremos que ver **dos mensajes cada vez que modifiquemos un contador:** uno debido al efecto asociado a la variable de estado del contador y otro debido al efecto que se ejecutaba independientemente de la variable de estado que se modificase.

Veamos si efectivamente esto ocurre. A continuación, haremos clic en el botón para modificar el contador 2 del componente 2:

A screenshot of a computer program

Description automatically generated

Figura 13. Prueba de  useEffect  en estado final. Fuente: elaboración propia.

Si nos fijamos en las dos últimas líneas de la consola, veremos que se han ejecutado los efectos deseados.

El hook de efecto es **muy utilizado habitualmente para el acceso a datos**, ya sea a través de las API (Application Programming Interface) o de llamadas a bases de datos, con el objetivo de actualizar cierta información en base a las acciones del usuario, aunque este es solo uno de sus posibles usos.

### **Customhooks**

Es muy habitual encontrarnos en situaciones en las que tenemos que utilizar el mismo bloque de código para realizar **una acción concreta en muchos sitios.** En otros lenguajes de programación, pensaríamos en utilizar funciones que pudiesen ser accesibles en todas aquellas partes de la aplicación que las necesitasen. En React ocurre lo mismo.

Los custom hooks son, simple y llanamente, funciones que se ejecutan cada vez que un componente funcional se renderiza.

Estas funciones pueden ser de muchos tipos y muy variadas, ya que en cada aplicación se podrían construir muchos y diferentes custom hooks. Como norma general y buena práctica, **todos los** ***custom hooks*** **que creemos** deben empezar por la palabra ***use*.** A continuación, puedes ver un ejemplo donde hemos creado un custom hook que se encarga de hacer peticiones GET (mediante la API Fetch) y devolver su resultado:

import { useEffect, useState } from “react”;

export constuseFetch = (url) =>{

console.log(‘ejecutando function useFetch’);

const[fetchResponse, setFetchResponse] = useState(‘…’);

useEffect(() =>{

constfetchRequest = async () =>{

console.log(“Realizando peticion…”);

letres = await fetch(url);

letdata = await res.json();

setFetchResponse(data.value);

};

fetchRequest();

-; [url])

return { fetchResponse }

}

Como puedes ver, un custom hook se define exactamente igual que un componente funcional (ambos son funciones), salvo por que en este caso no devuelve ningún bloque escrito con JSX. Este custom hookdevuelve datos o funciones **en función de la utilidad** que queramos que tenga el hook.

Dado que es una función, este hook se ejecutará cada vez que se produzca un renderizado en el componente donde lo incluyamos. Para comprobar esto, se escribirá por consola una línea cada vez que se trate de utilizar el hook.

Una cosa curiosa que podemos observar es que, dentro de este hook, estamos llamando a otro, concretamente a  useEffect . En efecto, un custom hook **puede utilizar internamente otros** ***hooks*.**Siempre y cuando se sigan las reglas de los hooks:

* **No llamar a** ***hooks*** **dentro de bucles, sentencias condicionales o funciones recursivas.** En lugar de eso:
  + Llamar a hooks en componentes de nivel superior.

* **No llamar a** ***hooks*** **desde funciones de JavaScript regulares.** En lugar de eso:
  + Llamar a hooks desde componentes funcionales.
  + Llamar a hooks desde custom hooks.

Si quieres profundizar sobre las reglas de los hooks, te recomendamos que eches un vistazo a la documentación oficial que se encuentra en el recurso de A fondo, Reglas de los hooks.

Dentro de nuestro hook personalizado estamos usando el hook de estado para guardar el resultado de la llamada a Fetch. Este componente podría tener también otras variables de estado que podrían indicar, por ejemplo, si la llamada se está realizando todavía o, en caso de que algo no haya ido bien, el error devuelto. Hacemos uso del hook de efecto para que cuando se renderice el componente que utiliza este custom hook se realice automáticamente la petición  GET .

Como puedes observar, la dependencia de este efecto es la **URL recibida contra la que hacer la petición.**Por tanto, aunque este custom hook sea una función y se ejecute con cada renderizado del componente que lo utilice, el hook de efecto solo se ejecutará si la URL cambia. A continuación, vamos a modificar ligeramente nuestro componente  Header  para que haga uso de este hook que acabamos de crear:

import React from ‘react’;

import { useState } from ‘react’;

import { useFetch } from ‘. ./hooks/useFetch’;

import ‘. ./styles/header.css’;

export constHeader = (numero) =>{

const[contadorUno, setContadorUno] = useState(0);

const{fetchResponse} = useFestch(“https://api.chucknorris.io/jokes/random”);

console.log(‘montando header…’);

return<div className=“unir-header”>

<p><b>UNIR – Header {numero}</b></p>

<b>{fetchResponse}</b>

<br></br>

<button onClick={() =>setContadorUni(contadorUno + 1)}>Pedir chiste</button>

</div>;

};

Nótese que, además, se ha modificado el archivo MainApp.js para que solo contenga un componente Header . Para hacer referencia al custom hook no hay más que importarlo y declararlo de la misma forma que lo hacemos con los demás hooks. En **cada renderizado de este componente,** se ejecutará la función useFetch y el argumento será siempre el mismo. En este ejemplo, se ha utilizado una API de chistes aleatorios de Chuck Norris.

Si te das cuenta, indicamos en el efecto de nuestro custom hook que la dependencia sería la URL que se recibiese. Por tanto, si a priori la URL no cambia nunca, el efecto solo debería ejecutarse una vez, independientemente del número de veces que pulsemos en el botón de «Pedir chiste», el cual modifica el valor de la variable de estado contador, lo que fuerza un nuevo renderizado.

**Ejecuta la aplicación de React.** Deberías ver algo similar a esto (ten en cuenta que, probablemente, el chiste que tú veas será distinto al que se muestre en la siguiente imagen):

A blue rectangle with black text

Description automatically generated

Figura 14. Aplicación de React con el chiste devuelto por el custom hook. Fuente: elaboración propia.

Si te das cuenta (esto no lo refleja la imagen), nada más cargar la página han aparecido tres puntos ( … ) encima del botón para pedir chistes. Sin embargo, un instante después, esos puntos han desaparecido y ha aparecido un chiste. ¿Por qué será? Observemos atentamente los mensajes que se han escrito en la consola:

A screenshot of a computer

Description automatically generated

Figura 15. Mensajes escritos en la consola tras la ejecución del custom hook. Fuente: elaboración propia.

De acuerdo con lo que hemos hablado, ¿tienen sentido estos mensajes? ¿Deberían ser más? ¿Deberían ser menos? Discutámoslo siguiendo el orden de ejecución:

A white and blue text on a white background

Description automatically generated

Tabla 4. Orden de ejecución. Fuente: elaboración propia.

Si utilizamos las herramientas de desarrollador de React para ver el estado del componente  Header , veremos que, efectivamente, concuerda con lo que acabamos de explicar:

A screenshot of a computer

Description automatically generated

Figura 16. Props y estado del componente funcional. Fuente: elaboración propia.

Como podemos observar, el contador sigue valiendo 0, pero la variable de estado  fetchResponse  sí tiene el texto del chiste que vemos por pantalla.

### **5.6. React Router**

Hasta ahora hemos trabajado en una aplicación con React que solo tenía una página. Pero sabemos que, en la vida real, las aplicaciones web usan el concepto de navegación a través de una ruta para acceder a una u otra característica. ¿Es posible conseguir esto mismo utilizando React? La respuesta es sí, gracias a React Router.

React Router es un conjunto de componentes de navegación listos para ser utilizados tanto en aplicaciones web como en móviles con React Native. La librería permite implementar enrutamiento dinámico en la aplicación.

Una de las **principales características de Router** **(versión 6)** es que las rutas estarán asociadas (renderizarán) a componentes funcionales de React que ya hayamos definido.

Esta librería nos aporta **nuevos componentes** que tendremos que utilizar:

* BrowserRouter **.** Es el componente funcional que englobará toda la aplicación para permitir el enrutamiento dinámico. En otras palabras, todos los componentes funcionales que utilicemos tienen que estar dentro de la declaración del  BrowserRouter  si queremos que se beneficien del enrutamiento. Además, esto es lo que da a la aplicación acceso a la API historial de HTML5 para realizar operaciones como  pushState ,  replaceState  o  popState  para poder navegar.
* Routes **.** Este componente englobará varios componentes  Route  y será el encargado de renderizar aquel que haya sido elegido en función del path que se haya especificado.
* Route **.** Este componente define una ruta posible en nuestra aplicación. Definir una ruta implica indicar qué componente funcional se va a renderizar si la ruta finalmente se va a tomar. Podemos hacer uso de varias propiedades a la hora de definir una ruta.
  + path **.** La ruta donde se renderizará el componente.
  + exact ***.*** El componente se renderizará si la ruta coincide exactamente con el path que hemos indicado.
  + caseSensitive **.** Se tendrán en cuenta mayúsculas y minúsculas.
  + element **.** Indicamos cuál es el componente que se debe renderizar en la ruta.

Para trabajar con esta librería, la instalaremos haciendo uso de  npm .

npm i react-router

Observa ahora un ejemplo de Router para la aplicación web que hemos venido construyendo hasta ahora:

import React from ‘react’;

import { Route, BrowserRouter, Routes } from “react-router-dom”;

import { Component1 } from ‘../components/Component1’;

import { Component2 } from ‘../components/Component2’;

import { DefaultComponent } from ‘../components/DefaultComponent’;

import { Home } from ‘../components/Home’;

export constAppRouter = () =>{

return (

<BrowserRouter>

<div>

<Routes>

<Route exact path=“/” element={ <Home|> } />

<Route exact path=“/c1/” element={ <Component1|> } />

<Route exact path=“/C2/” element={ <Component2|> } />

<Route exact path=“\*” element={ <DesfaultComponent|> } />

<|Routes>

<div>

<|BrowserRouter>

)

};

Como puedes ver, hemos creado un **nuevo componente funcional** llamado  AppRouter  que, internamente, hace uso de la librería React Router. El componente  BrowserRouter  encapsula todo el código de la aplicación para habilitar la navegación por la misma.

Cada vez que se necesita renderizar un componente de forma condicional basándonos en la ruta en la que estamos, hacemos uso del componente  Routes  y dentro de él, especificamos todas aquellas rutas que sean necesarias.

A continuación, vamos a repasar los conceptos sobre las **propiedades** que puede tener el componente Route  que antes comentábamos.

* La **primera ruta** que nos encontramos se ejecutará cuando nos estemos en el punto de entrada de la aplicación. Si tenemos desplegada nuestra aplicación en local, el host será localhost:3000 (siempre que no se haya cambiado la configuración del puerto). Por tanto, el componente Home se renderizará únicamente cuando naveguemos a las rutas localhost:3000 o localhost:3000/ (con o sin slash final).
* La **segunda ruta** hace uso de  exact , por lo que su componente se renderizará únicamente cuando naveguemos a la ruta definida en  path . En este caso, es localhost:3000/c1/ (con o sin slash final).
* La **tercera ruta** no utiliza  exact , por lo que podremos ver el componente asociado renderizado si navegamos a localhost:3000/C2/, pero también si navegamos a rutas con más niveles de profundidad, por ejemplo, localhost:3000/C2/algo o localhost:3000/C2/algo/algoMas. Lo que siempre deberá ocurrir es que la «C» sea mayúscula, ya que hemos usado caseSensitive.
* La **última ruta** sería el equivalente a un default en una estructura switch tradicional. Solo se renderizará el componente de esta ruta si ninguna de las anteriores ha sido elegida.

React evalúa las rutas de arriba hacia abajo, es decir, en orden de declaración.

Ya tenemos nuestro componente funcional haciendo uso de React Router, pero ¿cómo enganchamos ese componente a lo que ya tenemos construido? Bien, para esto modificaremos ligeramente el archivo MainApp.js:

import React from ‘react’;

import { Header } from ‘./components/Header’;

import { AppRouter } from ‘./router/AppRouter’;

export constMainApp = () =>{

return(

<Headernumero=“1”/>

<AppRouter|>

)

};

Ahora, MainApp incluye una referencia a  AppRouter  justo después de la declaración de nuestro primer componente,  Header . Pero ¿por qué incluimos el  Header  a este nivel? Declarando el  Header  antes de la declaración de nuestro  router  personalizado, aseguramos que, independientemente del componente que renderice  AppRouter , siempre habrá un  Header .

Esta estrategia es perfecta para componentes que deben estar en **cualquier pantalla de una aplicación** (headers ,  footers ,  sidebars , etc.). Dentro de  AppRouter  estamos haciendo referencia a nuevos componentes funcionales cuyo código es el siguiente (hemos obviado las declaraciones de  imports ):

export constHome = () =>{

return <div>

<h1>¡Home!</h1>

</div>;

};

export constDefaultComponent = 0 =>{

return <div>

<h1>Pagina no encontrada!!</h1>

</div>;

};

export constComponent1 = () =>{

return <div>

<h1>Componente 1!</h1>

</div>;

};

export constComponent2 = () =>{

return <div>

<h1>Componente 2!</h1>

</div>;

};

Incluye todos estos componentes en tu proyecto local, arranca la aplicación de React y prueba cada una de las rutas que hemos incluido.

Y ¿qué hay de rutas del estilo  /questions/1/ , donde parte de la ruta será siempre dinámica y necesitamos esa última parte del path para realizar llamadas a una API, acceder a una base de datos, etc.?

Bien, existe una **sintaxis especial** para eso. Si escribimos  /questions/:questionId,/  estaremos indicando que debe haber algún tipo de valor en  :questionId . Para acceder al valor de  questionId puedes usar el hook  useLocation  y luego, acceder al atributo  pathname , aunque ya no es la forma más limpia de hacer esto. Un ejemplo:

constlocation = useLocation();

constquestionld = location.pathname.split(‘/’)[location.pathname.split(‘/’).length -1];

Sin embargo, React Router nos ofrece otra forma de conseguir esto de una manera mucho más sencilla con el uso de un custom hook de esta librería,  useParams . Usaríamos este hook dentro del componente renderizado por la ruta. Ejemplo:

const{ questionld } = useParams();

Accede si lo deseas al vídeo Rutas con React para ver en vivo todos los experimentos que hemos realizado hasta ahora.

Aunque con estos conceptos es más que suficiente para construir el enrutamiento de una aplicación web sencilla, te recomendamos que eches un vistazo al apartado de A fondo, Documentación oficial de React Router,en busca de más ejemplos u otras funcionalidades.

### **5.7. Despliegue de un front-end con React**

Hasta ahora, hemos visto como **crear un** ***front-end*** con React, para incluir en esta ciertas características como enrutadores o hooks personalizados. Nuestro proyecto de software, seguramente, funcione bien en nuestra máquina local, pero ha llegado el momento de dar el **«salto» a la realidad** y desplegar nuestro código en un entorno «productivo», lo que habilita a nuestro proyecto a ser consumido por cualquier persona en cualquier parte del mundo.

El despliegue del software en una plataforma productiva es el **último paso** en el ciclo de vida de desarrollo de este (sin contar actualizaciones, parcheos…), ya que pasará de ser un proyecto local a una **realidad tangible** que el resto del mundo podrá ver. Es habitual hacer uso de **plataformas en la nube públicas,**como Amazon Web Services o Azure, para llevar a cabo estos despliegues. La desventaja es que es necesario registrarse con una tarjeta de crédito, aunque no lleguemos a incurrir en ningún gasto, siempre tendremos desconfianza de que nos cobren algo extra.

Po lo tanto, vamos a utilizar una aplicación web, llamada Vercel, que nos permitirá desplegar nuestros front-end de forma totalmente gratuita y sin necesidad de introducir ningún método de pago.

Vercel nació como una startup argentina, en el año 2015, de la mano de su creador Guillermo Rauch. Si bien desde sus inicios tuvo una gran aceptación, durante la pandemia de la COVID-19 alcanzó una popularidad mayor. Gracias a esta herramienta, es posible **desplegar** en apenas unos minutos grandes proyectos de front-end.

A través del siguiente enlace puedes obtener más información acerca de los orígenes y evolución de esta empresa:

<https://www.iproup.com/startups/24543-vercel-que-hace-este-unicornio-argentino>

Lo primero que tenemos que hacer para desplegar nuestro software es ir a la web de Vercel.

A screenshot of a website

Description automatically generated

Figura 17. Sitio web de Vercel. Fuente: Vercel.

Haremos click en «Sign Up» y elegiremos crear una cuenta de tipo Hobby:

A screenshot of a computer

Description automatically generated

Figura 18. Creación de cuenta en Vercel. Fuente: Vercel.

Una vez creada nuestra cuenta (podemos hacerlo a través de un usuario y contraseña o enlazando nuestra cuenta de Git), descargaremos e instalaremos el CLI (command line interface) de Vercel. Esto nos permitirá trabajar con Vercel directamente desde la terminal de nuestras máquinas.

Podrás instalar la versión acorde a tu sistema operativo desde el siguiente enlace:

<https://vercel.com/docs/cli>

Una vez instalado el CLI, lo siguiente será **disponer de un proyecto** front-end de prueba realizado con React para poder desplegarlo. Para ello, crearemos uno de cero al utilizar el comando  create-react-app, que ya conocemos. Si lo prefieres, puedes usar un **proyecto de React** que tengas creado de antes; sin embargo, asegúrate de que en local se **despliegue correctamente** y que no tenga errores.

[~~ JLumos >> ~/code/js-generic/react-example] $ npx create-reacte-app unir-test

Creating a new React app in /Users/jlumos/code/js-generic/react-example/unir-test.

Installing packages. This might take a couple of minutes.

Intalling react, react.dom, and react, scripts with cra-template…

Added 1420 packages in 1m

231 packages are looking for funding

Run ‘npm fund’ for details

Initialized a git repository.

Installing template dependencies using npm…

Una vez se creado el proyecto, navegaremos a su carpeta raíz en donde ejecutaremos el comando  vercel; si es la primera vez que lo ejecutamos en la sesión, se nos pedirá una **autenticación**. Luego, elegiremos la opción acorde al **registro** que realizamos (cuenta de Git, email…), introduciremos nuestros datos y continuaremos.

Una vez lo hecho ese paso, Vercel detectará que estamos intentando desplegar un proyecto de React y nos preguntará si queremos configurarlo y desplegarlo (esto solo ocurre la primera vez que queremos desplegar un proyecto):

[~~ JLumos >> ~/code/js-generic/react-example/unir-test] $ versel

Versel CLI 28.17.0

> > No existing credentials found. Please log in:

? Log in to Vercel github

> Success! Github authentication complete for jesus.perezmelero@unir.net

? Set up and deploy “~/code/js-generic/react-example/unir-test”? [Y/n]

Le indicaremos que sí queremos continuar y procederemos con el despliegue:

[? Set up and deploy “~/code/js-generic/react-example/unir-test”? [Y/n] y

? Which scope do you want to depaly to? unirdwfs

[? Link to existing project? [Y/n] n

[? What’s your project’s name? unir-test

[? In which directory is your code located?

Local settings detected in vercel.json:

Auto-detected Project Settings (Create React App):

- Build Command: react-scripts build

- Development Command: react-scripts start

- Install Command: ‘yarn install’, ‘pnpm install’, or ‘npm install’

- Output Directory: build

[? Want to modify these settings? [y/N] n

Linked to unirdwfs/unir-test (created .vercel and added it to .gitignore)

Inspect: https://vercel.com/unirdwfs/unir-test/3Zbnkkt31WBdeC4qETwrWbNZUZct [1s]

Production: https://unir-test.vercel.app [28s]

Deployed to production. Run ‘vercel –-prod’ to overwrite later (https://vercel.link/2F).

To change the domain or build command, go to https://vercel.com/unirdwfs/unir-test/settings

Como se puede observar, todas las preguntas se han respondido con las **opciones por defecto** que nos da Vercel. Entonces, una vez finalizado el despliegue, veremos la **URL de producción** de nuestro front-end (en el ejemplo sería https://unir-test.vercel.app), que podremos utilizar desde este momento para acceder a este, ya que es público. Además, si vamos a la web de Vercel veremos esta aplicación dentro de nuestros proyectos:

A screenshot of a computer

Description automatically generated

Figura 19. Proyectos en Vercel. Fuente: Vercel.

Si tratamos de acceder a través de la URL que se nos ha proporcionado, veremos nuestra aplicación desplegada:

A screen shot of a computer

Description automatically generated

Figura 20. Front-end desplegado. Fuente: Versel.

¿Qué ocurre si queremos hacer un cambio? Si quiero modificar el texto que hay debajo del logo de React, ¿cómo podría redesplegar el front-end? Pues, de una forma muy sencilla; no hay más que ejecutar de nuevo el comando  vercel  dentro de la raíz de nuestro proyecto y se volverá a desplegar nuestro front-end. Sin embargo, habrá una diferencia con el escenario anterior, como vemos a continuación:

[~~ JLumos >> ~/code/js-generic/react-example/unir-test] $ versel

Versel CLI 28.17.0

Inspect: https://vercel.com/unirdwfs/unir-test/8oNmBHcFLs5qHMwvBQMzjQNYfREW [3s]

Preview: https://unir-test.vercel.app [19s]

To deploy to production (unir-test.vercel.app), run ‘vercel –-prod’

[~~ JLumos >> ~/code/js-generic/react-example/unir-test] $

En este caso, se nos han devuelto dos URL: una es la misma que ya teníamos, la de producción, donde nada ha cambiado (podemos ver que todo sigue igual); y apareció otra llamada preview, que es ligeramente distinta a la URL de producción. Si entramos en ella, veremos que se reflejan los cambios que hemos hecho:

A screen shot of a computer

Description automatically generated

Figura 21. Preview en Vercel. Fuente: Vercel.

Por lo tanto, siempre que queramos redesplegar algo que ya está desplegado o en producción, usaremos la URL de preview para poder evaluar los cambios que hemos realizado.

Una vez confirmada la versión prewiew como correcta, solo queda pasar los cambios de la preview a la producción. Esto lo conseguimos ejecutando el **comando** vercel –prod , por lo que el software, automáticamente, **despliega** aquello que se encuentra en la versión preview y lo **copia** en la versión productiva.

[~~ JLumos >> ~/code/js-generic/react-example/unir-test] $ versel --prod

Versel CLI 28.17.0

Inspect: https://vercel.com/unirdwfs/unir-test/48Wt2C96oiCXjbxwQYreVqGgNQyV [1s]

Production: https://unir-test.vercel.app [22s]

Adicionalmente, si entramos a la web de Vercel, podremos ver algunos **detalles del proyecto** que hemos construido, como su URL de producción o las previews que tiene habilitadas:

A screenshot of a computer

Description automatically generated

Figura 22. Detalles de un proyecto en Vercel. Fuente: Vercel.

Dentro del apartado «Deployments» podremos ver un **detalle** de todos los despliegues (productivos y previews) de esta aplicación. También, podremos **eliminar** una preview o un despliegue productivo.

A screenshot of a computer

Description automatically generated

Figura 23. Destrucción de un despliegue. Fuente: Vercel.

Esto es algo que se recomienda enormemente, ya que de esta forma no consumimos los recursos de la plataforma en algo que no va a usarse.

Tratamos de actuar siempre de una manera ética y responsable.

Si lo deseas, también puedes aprender a desplegar tu front-end en Vercel siguiendo las instrucciones que verás en el vídeo Despliegue de front-ends en entornos reales.

Por último, no olvides de acceder al repositorio de la asignatura para practicar todo lo que has aprendido en este tema.

<https://github.com/UnirCs>