TUTORIAL ESCRITO
TUS PRIMEROS PASOS EN REACT

By Adrián Berenguer Agulló, Rosa María Rodríguez Lledó, Javier Rodríguez Juan, Ilya Slyusarchuck

ÍNDICE DE CONTENIDOS

Introducción

A continuación realizaremos un tutorial escrito para que aprendas los conceptos básicos para poder desarrollar tus interfaces usando uno de los frameworks de desarrollo web más de moda, React.

React es una librería de JavaScript para construir interfaces de usuario desarrollada por Meta en 2011, aunque no fue hasta 2013 cuando se volvió código abierto. El hecho de que sea una librería, hace que necesite de otras librerías o frameworks para el desarrollo de una aplicación web completa. Esto lo diferencia de frameworks como Angular, el cuál proporciona todas las herramientas para el desarrollo completo de una aplicación.

Para comenzar revisaremos los conceptos teóricos que deben saberse para comenzar a desarrollar en React y más adelante pasaremos a ver todos estos conceptos usándolos en un proyecto *"demo"*, para poder ver todos estos de una forma más clara. Finalmente se mencionan todas las fuentes que nos han ayudado a componer este documento.

Marco teórico

React es un lenguaje **declarativo**, es decir, el desarrollador escribe el código indicando lo qué quiere que React haga y, éste se encarga de renderizar y actualizar los componentes necesarios. Esta característica hace que el código sea más fácil de depurar, por tanto, más rápido a la hora de mostrar la vista. Este tipo de lenguajes son cada vez más usados y combinados con lenguajes imperativos, aquellos en los que se especifica las operaciones a realizar para llegar al resultado.

Dentro de este framework destaca el uso de un '**DOM virtual'**. El DOM es la estructura de objetos que genera el navegador para cada uno de los elementos de la interfaz de usuario. React crea una representación en memoria sincronizada a este DOM, al que llama 'DOM virtual'. Es un DOM poco pesado y con pocos recursos, de manera que cuando algo cambia en el estado de la interfaz, React actualiza el 'DOM virtual' y, éste se encarga de actualizar sólo las partes que han cambiado en el DOM real. El proceso es más rápido que actualizar el DOM al completo, por lo que esto supone una ventaja de React sobre otros frameworks. Además, es más rápido debido a que el código que se escribe es JSX que, con sintaxis HTML, compila un objeto JavaScript que se "*mapeara*" a un elemento del DOM.

Por otro lado, React usa una **estructura en componentes**. Estos componentes, son clases JavaScript independientes y reusables, que se van uniendo para formar interfaces de usuario complejas. Este es un aspecto que tiene en común con Angular, en el que también se parte de un componente raíz y, en base a este se va uniendo el resto y formando la aplicación. El estado es único para cada componente, mientras que las propiedades, las heredan los hijos de cada componente.

Para terminar , a continuación mostramos una tabla comparativa entre React y su gran competidor, Vue. Entre las características a destacar, el binding de datos de 2 sentidos de

Vue en comparación al unidireccional de React. La mayoría de webs de React, lo cuál deja ver la preferencia al uso de los desarrolladores generalmente. Finalmente, observamos varias aplicaciones desarrolladas con los respectivos lenguajes.

CARACTERÍSTICAS	VUE	REACT
Librería		✓
Binding de datos	2 SENTIDOS	1 SENTIDO
Webs activas	2 MILLONES	11 MILLONES
Librería oficial para aplicaciones móviles	✓	✓
Basado en componentes	✓	✓
Aplicaciones conocidas	NETFLIX Alibaba.com	

Cómo crear un proyecto y su estructura

Para poder crear nuestro primer proyecto con React tenemos que asegurarnos de tener instalado el gestor de paquetes de Javascript, o mejor conocido como npm. Si no estamos seguros si lo tenemos o no instalado, tan solo escribiremos ‘npm -v’ en nuestra terminal. Si este nos arroja un resultado, sabremos que npm esta instalado en nuestro equipo. En el caso de que nos lanzase un error, deberíamos entrar en la pagina web de node.js e instalar la ultima version LTS disponible (LTS = Long-Term Support o Soporte de larga duración, básicamente que es la ultima version estable).

Una vez tengamos instalado npm, tendremos que instalar un paquete que nos permitira crear la estructura basica de un proyecto con React. Para ello tendremos que ejecutar el comando ‘npm install create-react-app’. Tras unos segundos ya tendremos el paquete instalado y listo para poder ejecutarse.

En este caso vamos a utilizar una herramienta llamada ‘npx’ que viene incluida con versiones de npm superiores a la 5.2. La diferencia mas importante entre ‘npm’ y ‘npx’ es que ‘npm’ crea enlaces a los paquetes instalados en cada proyecto (carpeta node_modules), ademas de que si queremos ejecutar algun paquete deberemos incluirlo en nuestro archivo package.json. Este archivo incluye todos los paquetes y librerias usadas en nuestro proyecto, y a la hora de compartir nuestro proyecto es muy importante, ya que sin el, nuestro proyecto no sabra que paquetes tiene que instalar a la hora de ejecutar el proyecto. Por otro lado, mientras que ‘npm’ es un manejador de paquetes, ‘npx’ es

simplemente un ‘ejecutador’ de paquetes. Con ‘ejecutador’ quiero decir que podemos especificar un paquete y sin necesidad de incluirlo en nuestro package.json podremos ejecutarlo. Se suele utilizar ’npx’ para ejecutar paquetes que son de un solo uso en la vida de un proyecto como para crear el esqueleto de un proyecto que utiliza algun tipo de framework, o tambien se suelen utilizar para inicializar servidores entre otras cosas.

Con el comando ‘npx create-react-app tutorial’ estaremos creando un proyecto React con el nombre ‘tutorial’. Este proceso puede tardar de 2 a 3 minutos aproximadamente. Tambien podemos añadir plantillas a nuestro proyecto de React con la etiqueta —template a la hora de crear el proyecto (por si queremos utilizar TypeScript en nuestro proyecto) en vez de añadirlas posteriormente, aunque en este tutorial solamente veremos como crear un proyecto con React lo mas simple posible.

Una vez haya finalizado el proceso de creacion, podremos entrar a la carpeta del proyecto con ‘cd tutorial’. De primeras veremos varios archivos y carpetas que pueden parecernos extrañas si es la primera vez que usamos un framework, pero realmente la mayoria de frameworks comparten muchos de estos ficheros por lo que si nos familiarizamos con un framework, trabajar con otro nos sera considerablemente mas facil.

Empezando de abajo a arriba podemos ver un archivo con el nombre de README.md, este archivo contiene informacion basica acerca de React, como ejecutar el proyecto, como ejecutar pruebas unitarias entre otras cosas. Un poco mas arriba podemos encontrar posiblemente uno de los archivos mas importantes de nuestro proyecto, y es el archivo package.json. Este archivo contiene informacion sobre los paquetes que tiene el proyecto instalados, scripts, el nombre del proyecto, e incluso los navegadores donde puede ejecutarse la aplicacion. Para instalar los paquetes tendriamos que ejecutar ‘npm install’

No podemos confundir el archivo package.json y el archivo package-lock.json. Este segundo archivo puede ser diferente para cada desarrollador, y contiene informacion mas precisa de los paquetes que se han instalado y de como se han resuelto las dependencias. Este archivo no es necesario subirlo a nuestro sistema de control de versiones ya que se genera automaticamente cuando ejecutamos ‘npm install’

El archivo .gitignore contiene todos los archivos y carpetas que no queremos que se incluyan en nuestro sistema de control de versiones. Entre estos archivos encontraremos la carpeta node_modules y archivos de entorno local entre otras cosas.

La carpeta src seguramente sera la carpeta donde mas tiempo pasaremos como desarrolladores de React. Aqui se encuentran los componentes de React, las hojas de estilos, los tests y otros archivos JavaScript que nos seran necesarios. Uno de estos archivos es el index.js, este archivo importa index.html de la carpeta public y renderiza el componente App dentro del elemento div con id root. Esto lo hace automaticamente React cada vez que construimos el proyecto por lo que no sera necesario modificar estos archivos.

La carpeta public contiene todos los elementos estaticos de nuesto proyecto como imagenes, videos o ficheros. Este tambien es el hogar del archivo index.html del que hemos hablado anteriormente, que contiene la plantilla necesaria para renderizar nuestro proyecto React.

Para ejecutar el proyecto usamos npm run start, para ejecutar los tests usamos npm run test y para compilar todo el proyecto y empaquetarlo usamos npm run build

En el caso de que empaquetemos el proyecto, se nos creara una carpeta llamada build que contiene los elementos de la carpeta public, y unos archivos con extension javascript y css que contienen todos los datos de nuestro proyecto pero empaquetandolo en un solo archivo

La Reactividad

Para comenzar a explicar este apartado, vamos a definir brevemente unos conceptos básicos.

- **Estado**: son los datos de tu aplicación en un momento particular. Es por eso que el estado tiene una duración determinada y cambiará cuando cambie algún dato.
- **Interfaz basada en el estado**: aquella que usa los datos de la aplicación en todo momento para renderizar sus elementos visuales. Por ejemplo, una web en la que estando logueado o no, se vea una página u otra.
- **Reactividad**: en un ámbito general, podemos dar la siguiente definición: habilidad de una variable para actualizarse automáticamente cuando otra variable que le hace referencia es cambiada, incluso después de haberse declarado. Un sencillo ejemplo es el que dio Rich Harris en la charla Rethinking reactivity, que es el de las casillas en excel. Una casilla se actualiza automáticamente cuando una casilla de la que depende cambia sus datos.

Pero más concretamente en el mundo de la programación web, la **definición** que buscamos es la siguiente: propiedad de una aplicación que permite que la interfaz de usuario se modifique automáticamente cuando se modifica el estado. Permite mantener el estado (Modelo) y el DOM (la vista) sincronizado.

¿Es Javascript reactivo?

```
<script>
    var contador = 0;
    var main = document.getElementById("button");
    var contadorSpan = document.getElementById("contador");
    main.addEventListener("click", function(){
        contador++;
        console.log(contador)
    });
</script>
```

No lo es por defecto: cuando el estado cambia, las vistas dependientes no son re-renderizadas automáticamente. Es por eso responsabilidad del programador (o de algún framework) mantener el modelo y las vistas sincronizadas. Uno de esos frameworks es React.

```
<script>
    var contador = 0;
    var main = document.getElementById("button");
    var contadorSpan = document.getElementById("contador");
    main.addEventListener("click", function(){
        contador++;
        console.log(contador)
        contadorSpan.innerHTML = contador;
    });
</script>
```

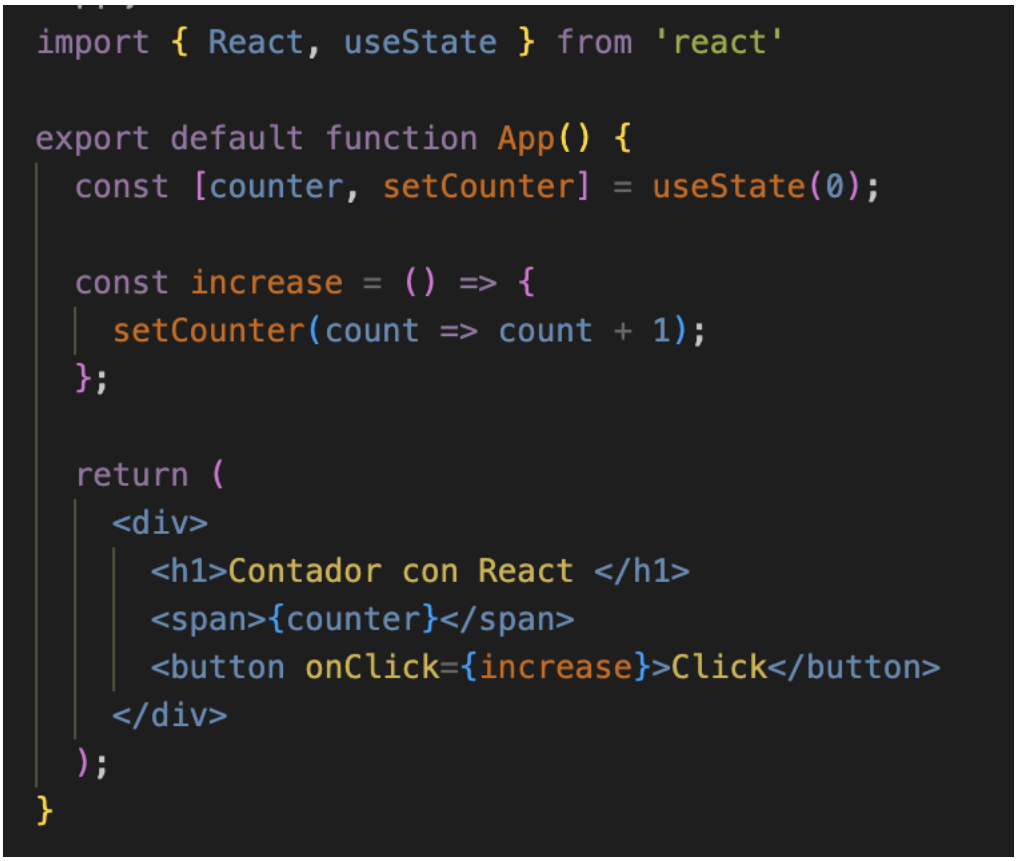
Ejemplo con javascript donde cambiar la variable no actualiza la interfaz y por tanto modelo y vista no están sincronizados.

```
<script>
    var contador = 0;
    var main = document.getElementById("button");
    var contadorSpan = document.getElementById("contador");
    main.addEventListener("click", function(){
        contador++;
        console.log(contador)
        contadorSpan.innerHTML = contador;
    });
</script>
```

Ejemplo donde el programador cambia manualmente el HTML cada vez que se da click.

Ahora imagina este problema: tenemos una aplicación donde la variable contador se utiliza cientos de veces en cientos de lugares diferentes. Si queremos mantener la reactividad con vanilla javascript, tenemos dos opciones: o mantenemos registro de en qué elementos html se ha utilizado la variable contador y los renderizamos uno a uno, o renderizamos la página entera, incluyendo elementos que no han cambiado. Este es el problema que soluciona React.

React consigue la actualización continua del estado con el API *useState*. La función importada de *react* devuelve un array que contiene dos elementos. El primero representa la variable de estado como tal, mientras que el segundo referencia a una función que será invocada para actualizar la variable. En el ejemplo que hemos dado, esa función es setCounter. Eso significa que no deberíamos hacer counter++ o counter—, sino llamar a la función que actualiza al estado. Además, esta solución es más declarativa. A continuación vemos un ejemplo de código de esto, aunque más adelante profundizaremos en el concepto de *useState* para conocer de qué se trata exactamente.



```
import { React, useState } from 'react'

export default function App() {
  const [counter, setCounter] = useState(0);

  const increase = () => {
    setCounter(count => count + 1);
  };

  return (
    <div>
      <h1>Contador con React </h1>
      <span>{counter}</span>
      <button onClick={increase}>Click</button>
    </div>
  );
}
```

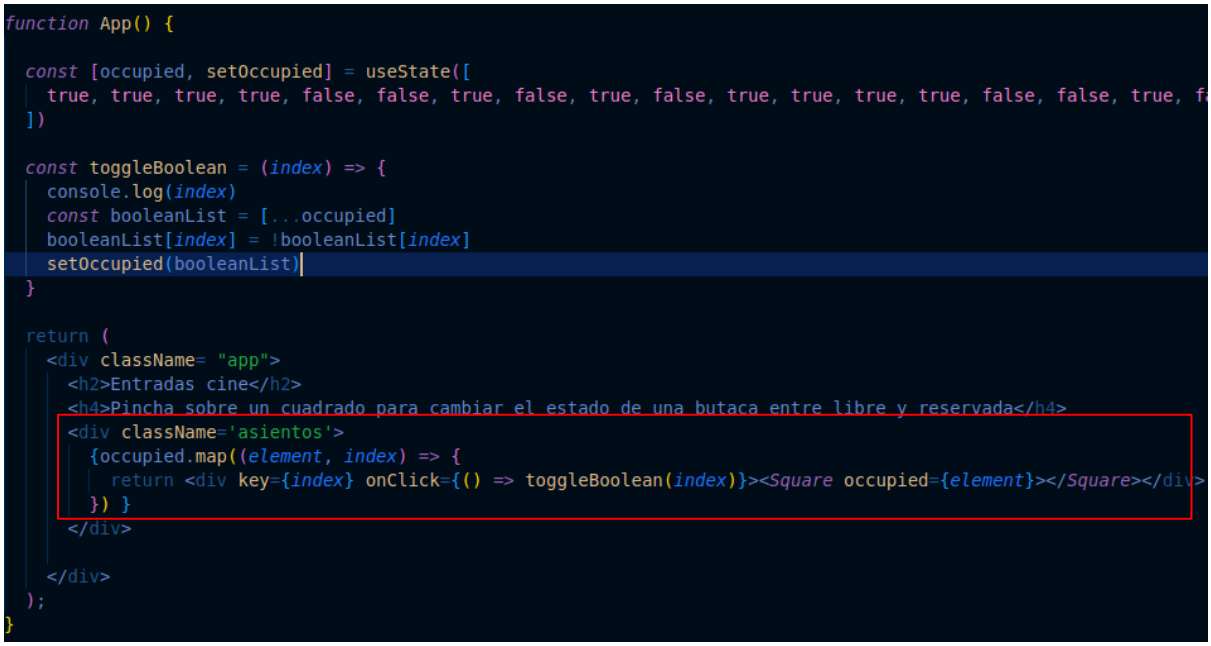
El aspecto declarativo de React

Como hemos comentado en la introducción teórica React es un framework declarativo, en el cuál el programador **define** el estado final del UI que quiere y React se encarga de realizar

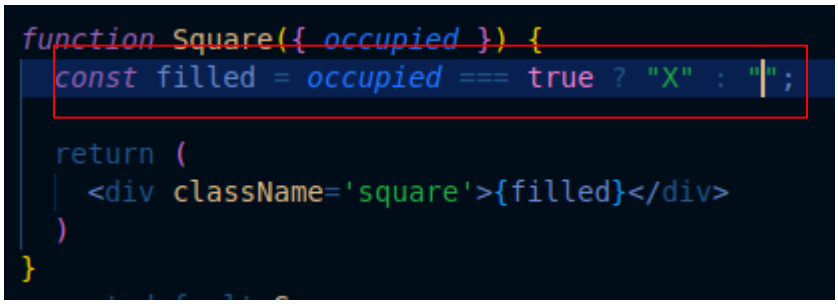
sobre el DOM las mutaciones que necesite para llegar a este. Por esto podemos decir que React es un framework declarativo respecto a la manipulación del DOM.

Además de esto, React también es declarativo respecto al renderizado de sus componentes ya que cuándo definimos funciones, no somos nosotros quienes **invocamos** a estas en el futuro, sino que es el propio framework el que se encarga de invocarlas cuándo es necesario.

Veamos esto con un ejemplo más claro, usando como proyecto de prueba un sistema muy simple de reservas de asientos para el cine.



En la anterior captura podemos ver el componente raíz de nuestra aplicación de ejemplo, aquí se realiza un bucle para crear tantos componentes ***Square*** como asientos se hayan declarado y como se puede observar, se define la función "*toggleBoolean".* Esta función será llamada por React cuándo el usuario pinche sobre un componente hijo *Square*, nosotros nunca la llamaremos directamente.



Para finalizar con este ejemplo aquí tenemos la lógica del componente hijo ***Square***, el cuál también tiene un aspecto declarativo ya que vemos como la programación se orienta a

conseguir la interfaz final que queremos. Esto lo podemos ver en la variable de estado computada llamada "*filled*", la cuál variará siguiendo los cambios de la variable "*occupied*" y que a su vez variará la interfaz de ese componente en la aplicación.

Los componentes

Como en cualquier framework de desarrollo Front-end moderno, en React tenemos la posibilidad de separar las diferentes partes de nuestra interfaz en componentes para tener una estructura más organizada y reutilizable de nuestra aplicación. En react tenemos 2 formas de declarar componentes, ambas igual de válidas.

Componente Clase

De esta manera definimos el componente como si fuera un clase JS.

```
export default class SquareClass extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }

  clicked() {
    this.setState((prevState) => ({
      count: prevState.count + 1
    }))
    console.log(this.state.count)
  }

  render() {
    return (
      <Fragment>
      <div className='square'>{this.props.occupied === true ? 'X' : ''}</div>
      <button onClick={() => this.clicked()}>Pulsa aquí!</button>
      </Fragment>
    )
  }
```

Ahora explicaremos más en detalle la estructura del componente:

- Las **propiedades** serán definidas en un constructor que obligatoriamente siempre llamará al comienzo, al constructor de su clase base.
- En cuánto al **estado**, se inicializará en el constructor de la clase y después podrá variarse usando el método "*setState()*".
  - Si queremos variar el estado dependiendo del estado anterior de esta variable lo recomendable es que este método invoque a una función anónima de flecha en la que se varíe el estado consultando el estado anterior mediante el uso de la variable "*prevState*". Esto evitará comportamientos inesperados en ejecuciones concurrentes.

- Las etiquetas HTML que conforman la **interfaz** las definiremos dentro del método "*render()*".
  - Como podemos observar, envolvemos toda la interfaz dentro de una etiqueta llamada "*Fragment*". Esto lo hacemos porque la sentencia "*return*" solo nos permite devolver un objeto, por lo que si intentamos devolver más de una etiqueta HTML obtendremos un error. Para evitar esto React añadió la etiqueta "*Fragment*" que nos permite envolver toda nuestra interfaz dentro de una etiqueta que después en la renderización del componente no se tendrá en cuenta. Así evitamos el error y mantenemos nuestra interfaz limpia (ya que también evitamos poner etiquetas "*div*" envolventes innecesarias)
- Los **manejadores** los definiremos también dentro de la clase como funciones y accederemos a ellos mediante la referencia "*this*"

Componente Función

De esta forma definiremos los componentes como funciones JS. Esta es la forma más sencilla de definir componentes.

```
function Square({ occupied }) {

  const [counter, setCounter] = useState(0)

  const clicked = () => {
    setCounter((prevState) => (prevState + 1))
    console.log(counter)
  }

  return (
    <Fragment>
      <div className='square'>{occupied === true ? 'X' : ''}</div>
      <button onClick={() => clicked()}>Pulsa aquí!</button>
    </Fragment>
  )
}
export default Square
```

En la captura anterior podemos observar el equivalente del ejemplo mostrado en la sección "Componente clase", ahora definido en forma de componente función. Podemos ver que el componente tiene la siguiente estructura

- Las **propiedades** se definen como argumento de la función que define el componente
- Para declarar el **estado** usamos el "*Hook*" de estado "*useState*". Luego haremos una breve introducción a los "*hooks*", por el momento solo debemos tener en cuenta que el argumento que pasemos a "*useState*" será el valor inicial de nuestra variable de estado y que el método "*set*" que declaramos durante la desestructuración de

"*useState*" será el que usaremos para variar el estado de esta variable (en nuestro caso la variable es "*counter*" y el método para variar el estado "*setCounter*")

- Los **manejadores** los declararemos como funciones de flecha anónimas que asignaremos a variables
- La **interfaz** la incluiremos dentro de la sentencia "*return*" de la función

Ciclo de vida de los componentes

Desde que un componente se inicializa hasta que se destruye, existen ciertos momentos que nos interesa capturar para realizar acciones en estos. A estos momentos los llamamos el ciclo de vida de un componente y en código React los representamos como funciones que podemos declarar con el nombre del momento donde queramos que se ejecute la función y esta se invocará automáticamente cuándo este momento llegue.

A continuación nombraremos y explicaremos todos estos momentos, ahora sí, cabe destacar que podremos usar los métodos del ciclo de vida si usamos componentes en formato clase. Si estamos usando componentes función tenemos la alternativa de los *hooks*, que pueden realizar una función similar, explicaremos estos al final de esta sección.

Métodos del ciclo de vida

Separaremos el ciclo de vida en 3 fases y de cada fase explicaremos cada uno de los métodos del ciclo de vida que podemos utilizar. Nombraremos los métodos en orden de ejecución.

<u>Fase de montaje:</u>

Cuándo se crea el componente y se añade al DOM. Aquí encontramos los siguientes métodos:

- constructor() → Primer método invocado cuándo se inicializa el componente pero aún no se ha renderizado
- getDerivedStateFromProps() → Método estático usado para modificar el valor del estado a partir del valor de alguna propiedad
- render() → Este es el único método requerido para los componentes de formato clase. Este inserta el HTML en el DOM
- componentDidMount() → Último método llamado en el montaje ejecutado justo después del renderizado. En este método se nos permite añadir efectos colaterales como enviar peticiones HTTP o actualizar el estado. Este método está dedicado a esto ya que el resto de métodos están pensados para ser *puros*, es decir, que no pueden ni cambiar el estado, ni enviar peticiones HTTP ni tener ningún tipo de interacción con el navegador

Fase de actualización:

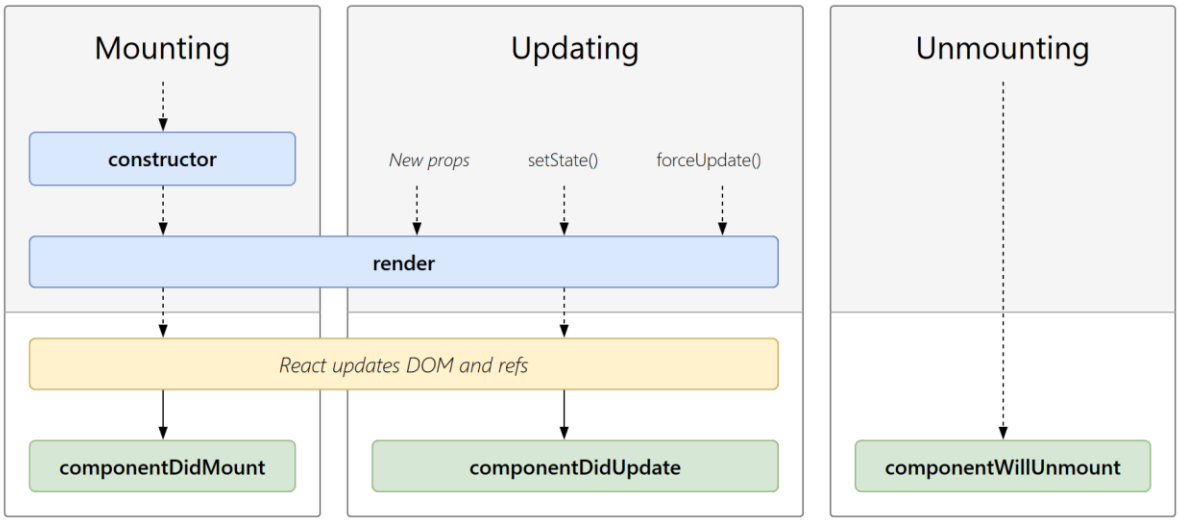
Cuándo el componente se actualiza, esto ocurre cuándo cambian las propiedades o el estado de este.

- getDerivedStateFromProps() → Método llamado justo después de actualizar una propiedad. Este lo usamos si queremos reflejar este cambio en el estado.
- shouldComponentUpdate() → Este permite que le digamos a React cuando no tiene que hacer un re-renderizado. React recomienda no usarlo ya que puede introducir errores y recomienda el uso de los "*PureComponents*"
- getSnapshotBeforeUpdate() → Método que nos da acceso a las propiedades y estado justo antes de actualizarse. Se ejecuta después de render() y antes de componentDidUpdate().
- render() → Realiza la re-renderización del DOM
- componentDidUpdate() → Misma finalidad que en la fase de construcción

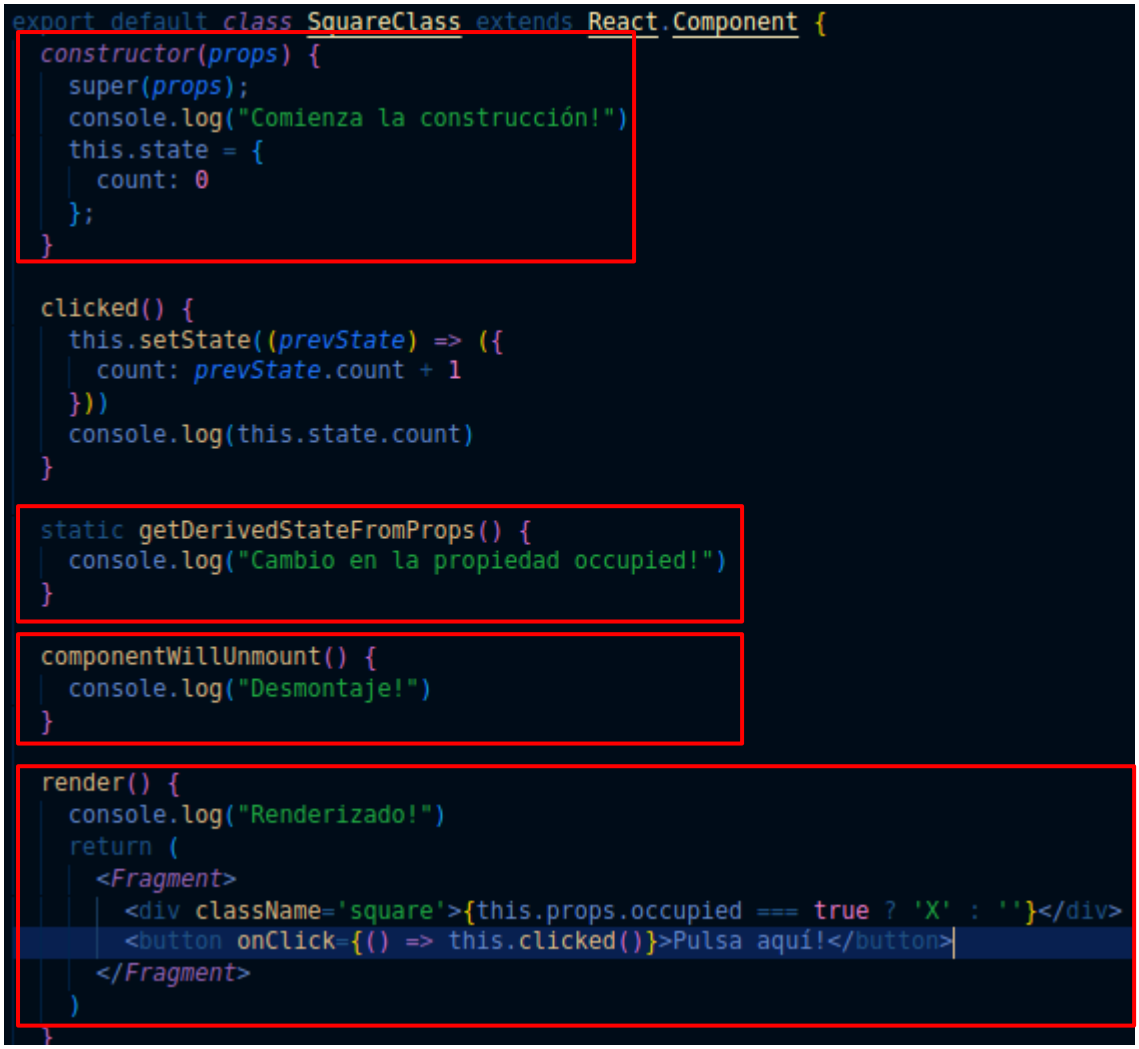
Fase de desmontaje:

Cuándo el componente se elimina del DOM

- componentWillUnmount() → Invocado justo antes de eliminar el componente del DOM. Este se encargará de limpiar el componente (por ejemplo de suscripciones)



A continuación veremos un ejemplo de uso en nuestra aplicación de ejemplo:



Aproximación funcional: Los hooks

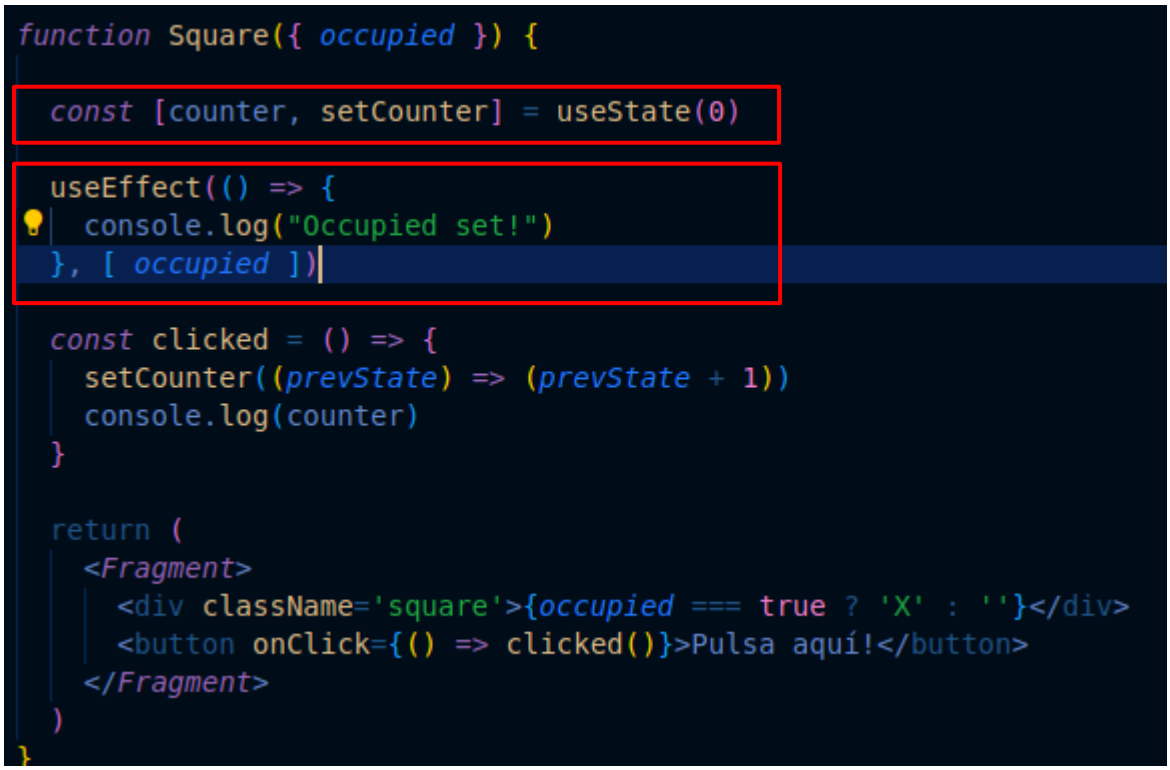
Como hemos visto anteriormente hay una manera funcional de declarar componentes que nos produce escribir código de forma más limpia, sin embargo veíamos que esta aproximación tenía la restricción de ser una aproximación en la cuál no podíamos añadir estados a nuestros componentes. Para eliminar esta restricción llegaron los "*hooks*".

Los "*hooks*" son funciones que proporcionan estado a los componentes funcionales y que dan la posibilidad de que podamos añadir efectos colaterales en estos. Dentro de todos los "*hooks*" que React nos proporciona hay 2 importantes que debemos de saber:

- useState() → Este es el que proporciona que podamos guardar variables de estados en nuestros componentes. Ya hemos hablado más en profundidad de este en secciones anteriores.

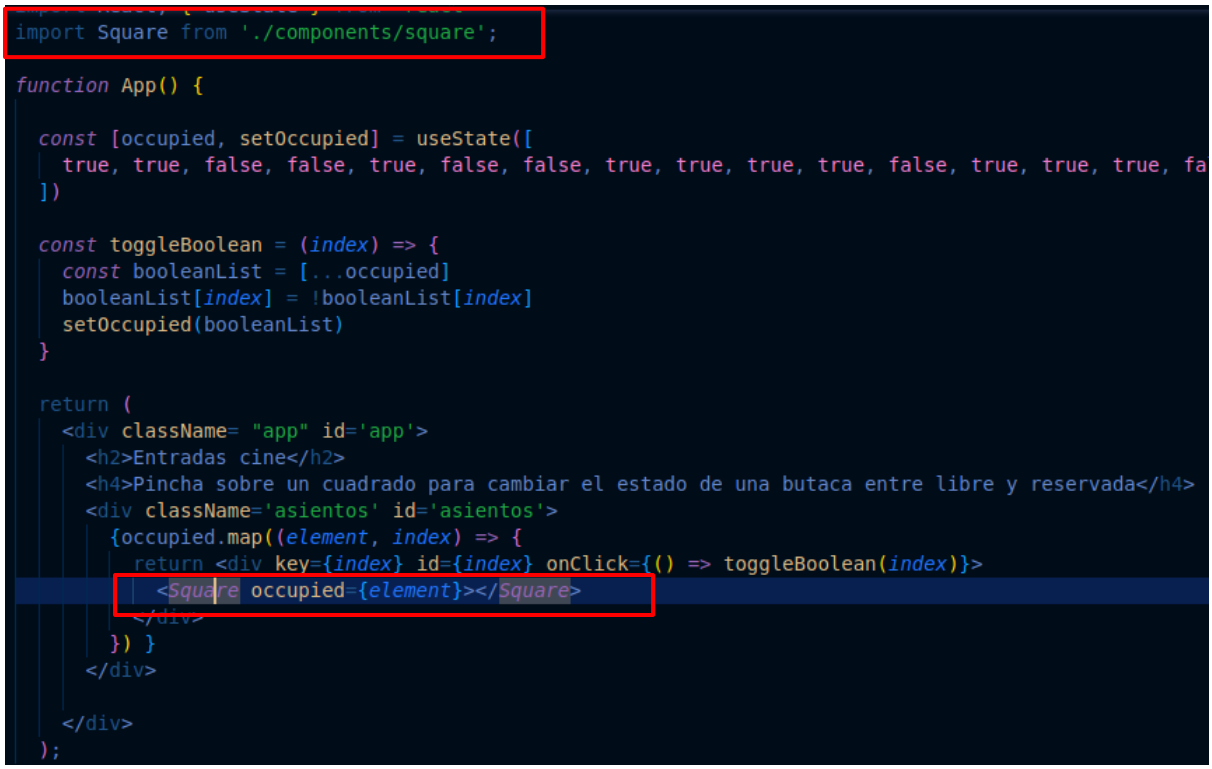
- useEffect() → Este es el que nos permite añadir efectos colaterales al componente, que por defecto debe mantenerse siempre puro. Este acepta una función por parámetro la cuál contiene todos los efectos colaterales que queramos añadir. Cabe

destacar que por defecto este se ejecutará una sola vez después del primer renderizado pero que este *hook* acepta un segundo parámetro que especifica cuándo se debe volver a ejecutar. En este array se definirán las variables de estado o propiedades que si cambian harán que se invoque la función de este *hook*. Si reflexionamos acerca de los métodos de ciclo de vida que hemos visto antes podemos darnos cuenta de que este *hook* tiene una función similar a la de los métodos *componentDidMount()*, *componentDidUpdate()*, y *componentWillUnmount()*.



Interacción entre componentes padre - hijo

React también nos da la posibilidad de anidar unos componentes dentro de otros para crear estructuras complejas sin perder la organización de nuestro código. Para hacer esto, basta con importar el componente hijo que queramos usar y añadirlo al HTML. Cabe destacar que por convenio, React recomienda en las etiquetas HTML para componentes creados por el usuario que la primera letra sea mayúscula. Veamos esto con un ejemplo:



Las propiedades que recibirá el componente hijo serán los atributos que marquemos dentro de la etiqueta HTML de apertura del componente. En este caso desde el componente *Square* podremos acceder a *props.occupied* para recoger el valor que en el componente *App* estamos declarando.

Link a vídeo de Youtube

Adicionalmente a este tutorial también se ha realizado un vídeo tutorial que contiene todo el material expuesto en este documento pero de una forma más práctica, exponiendo todos los conceptos sobre un proyecto real. A continuación el link al vídeo:

https://youtu.be/93xTJCVnQbs

Link a proyecto completo

En el siguiente link puedes encontrar un enlace de Google Drive que contiene el proyecto al completo. En este archivo podrás encontrar todo el proyecto demo usado para las explicaciones prácticas así como las diapositivas usadas en el vídeo y las tomas del vídeo separadas por contenidos. Es recomendable leer el README contenido en el archivo del enlace para ver todos los contenidos disponibles en este.

https://drive.google.com/file/d/1l2c_gMaFm2TvSs-5Vs4SIlo4IMpzKJxe/view?usp=sharing

Referencias bibliográficas

A continuación la lista de fuentes de donde se ha extraído y recopilado toda la información expuesta en este documento.

- https://medium.com/trabe/why-is-react-declarative-a-story-about-function-components-aaae83198f79
- https://alexsidorenko.com/blog/react-is-declarative-what-does-it-mean/
- https://www.geeksforgeeks.org/reactjs-state-react/
- https://reactjs.org/docs/components-and-props.html
- https://www.youtube.com/watch?v=EMk6nom1aS4
- https://retool.com/blog/the-react-lifecycle-methods-and-hooks-explained/
- https://jonmircha.com/reactividad-javascript
- https://www.bigthinkcode.com/insights/reactivity-in-view-libraries
- https://www.freecodecamp.org/news/reactjs-basics-dom-components-declarative-views/
- https://owlcation.com/stem/reactCounter
- https://dev.to/theaswathprabhu/what-is-reactivity-116f
- https://reactjs.org/docs/create-a-new-react-app.html
- https://www.freecodecamp.org/news/reactjs-basics-dom-components-declarative-views/
- https://www.freecodecamp.org/news/how-to-build-a-react-project-with-create-react-app-in-10-steps/
- https://es.reactjs.org/
- https://tech.tribalyte.eu/blog-que-es-react