Creacion de un proyecto Front end usando la api REST productos – con REACT (vs Angular)

Ya cree el cliente (front end) para la utilizar la API REST (back end) productos utilizando Angular, ahora quisiera hacerlo utilizando REACT.

Ambos parten con la creación de un marco de proyecto inicial sobre el cual luego avanzamos escribiendo código, agregando componentes etc. Angulas usa Typescript mientras que REACT usa Javascript

Angular mediante el comando

**ng new carpeta\nombre\_proyecto --routing** (lo ultimo agrega un servicio para enrutamiento)

REACT mediante el comando

**npx create-react-app nombre\_proyecto**

si quisiera crearlo para usarlo con typescript en vez de Javascript debiera hacerlo de la siguiente manera

|  |  |
| --- | --- |
|  |  |

**npx create-react-app nombre\_proyecto --typescript**

Las aplicación es desarrolladas en Rectt por default levanta el servicio cliente en el puerto 3000, para evitar conflictos con el servidor del front-end que también levanta en el puerto 3000 debo cambiarlo en alguno de los dos. En el caso de querer cambiarlo aquí, una de las posibles formas es generar un archivo en la carpeta raíz de nombre .env y agregarle

**PORT=8001**

Se puede hacer también modificando el package.json.

Antes de empezar a escribir código se debieran de determinar los componente que tendría el proyecto y en angular se crean mediante el comando

**ng generate component carpeta\_componente/nombre\_componete**

En el caso de la aplicación de productos se individualizaron tres componentes

* una barra de navegación (Navbar)
* una lista de productos para listar los mismos (product-list)
* un formulario de productos para editarlos (crear y modificar) (product-form)

Sera necesario la creación de un servicio que es donde se vinculara con los métodos http de la API REST del back-end

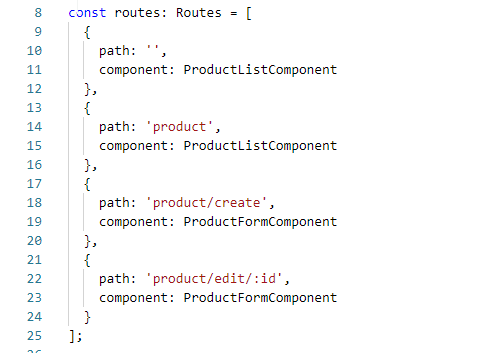
ng generate service carpeta\_servicio/nombre\_servicio

También será necesario la creación de una interfaz producto para manejar la lo que sería ORM o ODM(según sea el tipo de base de datos que utilicemos) esta interfaz de generará a mano en constará de una carpeta\_interfaces y un archivo productos.ts para la interfaz.

Cuando creamos el proyecto en Angular le indicamos mediante **–routing** que iba a ser un proyecto con enrutamiento por esto nos creo dentro de la carpeta /src un archivo para el módulo de enrutamiento **app\_routing.module.ts**. En el vamos a definir todas las rutas que nuestra aplicación posea, en nuestro caso serán cuatro

* la ruta raíz (´´) 🡪 apunta al componente ProductList
* ruta ‘product’ 🡪 apunta al componente ProductList
* ruta ‘product/create’ 🡪 apunta al componente ProductForm
* ruta ‘product/edit/:id 🡪 apunta al componente ProductForm

se encaminaran desde los ítems de la barra de navegación y llamaran a los diferentes métodos http del la API (get, post, delete, put)



Cuando creo los componentes en Angular no tendrán nada más que algún texto que indique el componente, esto es el archivo .html de cada componente solo habrá algo que del tipo

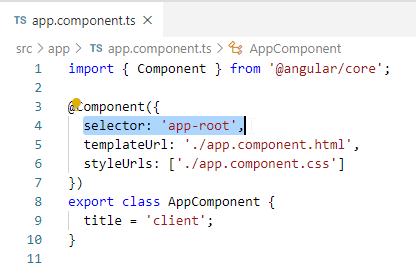
**<h1>product-list works!!!</h1>**

## allí deberemos reemplazar por lo que corresponda al componente que será muy parecido al HTML pero es algo propio de AngularJS que se ve muy parecido al HTML (así como en REACT lo es el JSX)

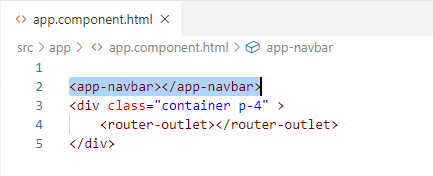
En Angular el archivo inicial de la aplicación es el es index.html allí está la etiqueta principal

**<app\_root></app\_root>**

**app\_root** es el selector que corresponde la **app.component** que es el componente principal de la aplicación y se encuentra definido en la carpeta **/app** en el archivo **app.component.ts**



dentro del **app.component.html** estarán los demás componentes

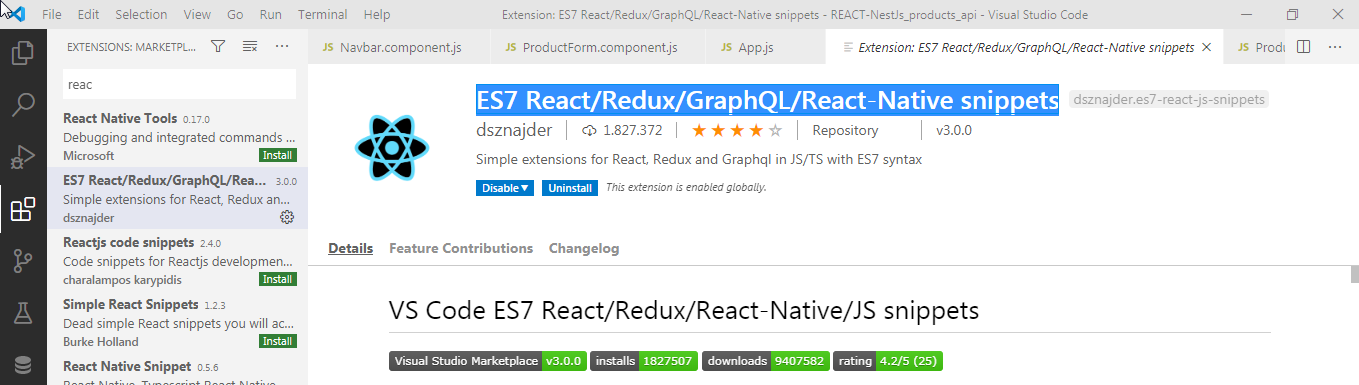


Allí tenemos el componente barra de navegación **<app\_navbar>** y el **<router-outlet>**

**router-outlet** es una directiva que está disponible en el paquete **@angular/router** y es utilizada por el enrutador para marcar en qué lugar de una plantilla se debe insertar un componente coincidente. Gracias a la **router-outlet**, la aplicación tendrá múltiples vistas/páginas y la plantilla de la aplicación actúa como un shell de su aplicación.

## Plugin que me ayuda con React a la hora de escribir texto

Dentro de NodeJS puedo agregar el plugin **ES7 React/Redux/GraphQL/React-Native snippets** utilizando shorcuts me facilita a la hora de escribir texto para muchas cosas de React, como por ejemplo el código base que lleva una clase para un componente, con solo escribir **rcc** (react class component).

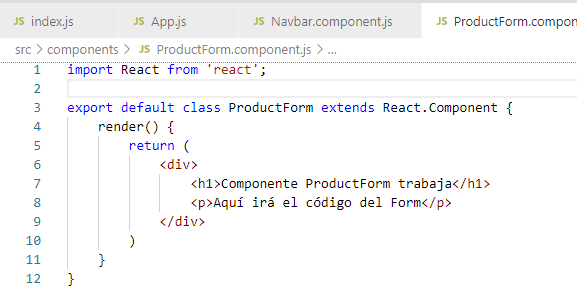


## Crear un componente en React

<https://www.youtube.com/watch?v=0LwhhotHylI>

<https://www.youtube.com/watch?v=5XmBxtOywUk>

Para seguir una convención creamos una carpeta **components** y allí colocamos todo lo referente a componentes que será por el momento un archivo .js por cada componente. En cada uno de ellos escribiremos la clase que lo describe y que debe extender d **React.Components** y se exporta. cada clase tendrá un método **render()** que será el encargado de dibujar el componente en pantalla.

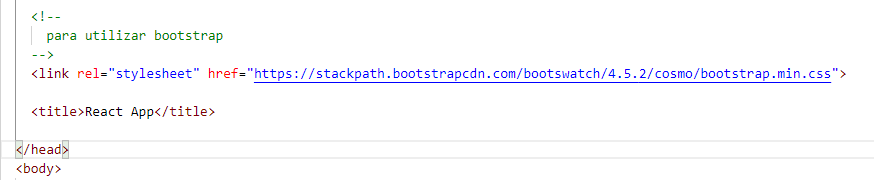


## Agregado de una barra de Navegacion para un componente Navbar

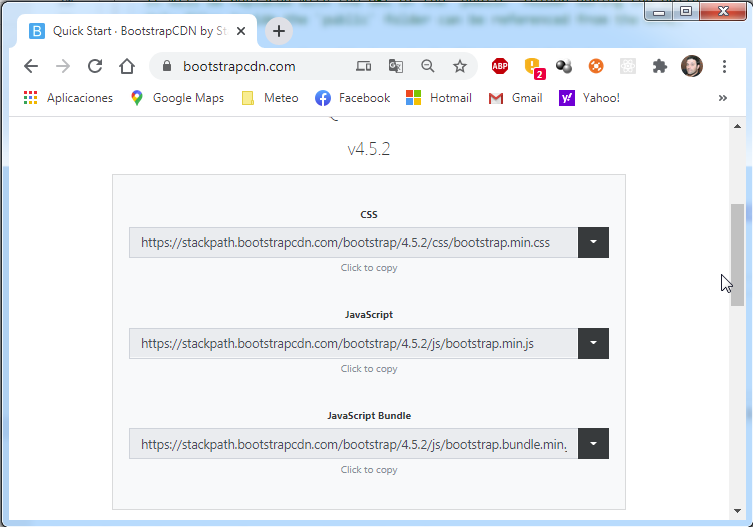
<https://www.youtube.com/watch?v=X3BAQQpKOp0>

<https://www.youtube.com/watch?v=fiQh6xUDBBQ>

para usar una barra de navegación y no crearla desde cero vamos a apelar a **Bootstrap** y para utilizar bootstrap en nuestra aplicación debemos hacer una de dos cosas, o importamos la librería **bootstrap** desde **npm** y la incorporamos como modulo a nuestra aplicación o como hacemos en un HTML en el **<head>** agregamos una línea indicando la URL de la librería de **bootstrap**.



el archivo .html que debemos tocar es el **public/index.html** y debemos agregarle la línea que nos indique <https://www.bootstrapcdn.com/>

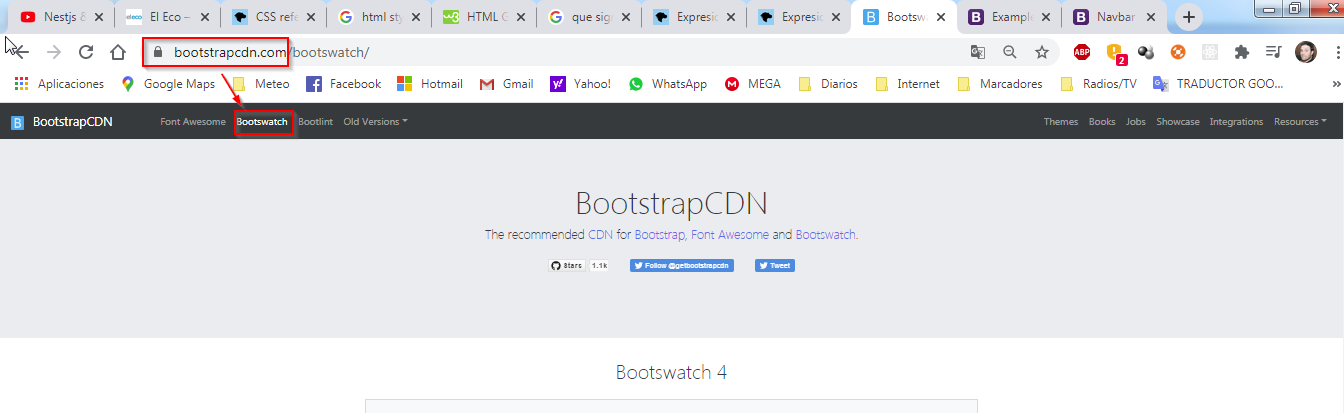


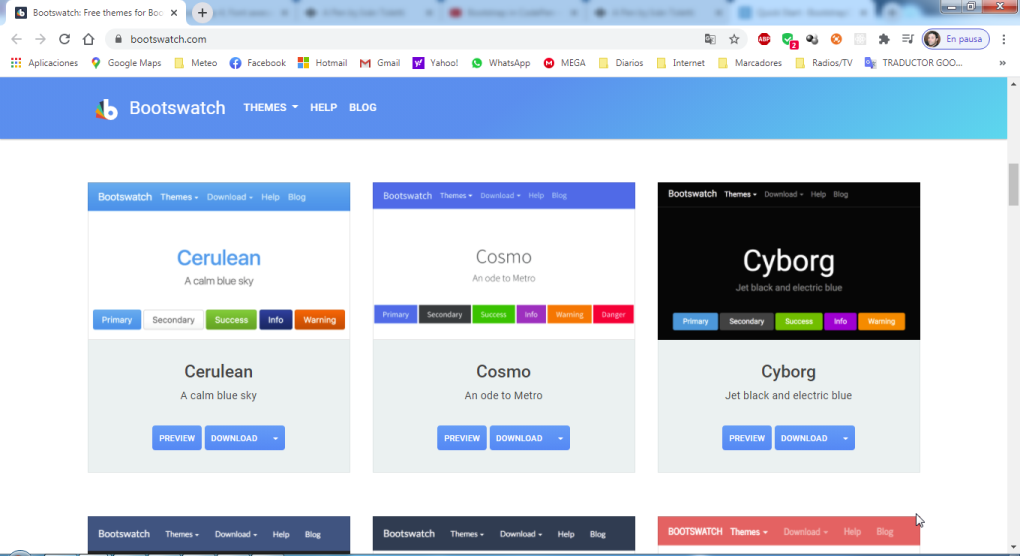
Si solo vamos a utilizar los estilos copiamos la que indica .css si necesitamos la de JavaScript sumamos esa otra también.

Otra cosa que también conviene tener instalado en el Editor si vamos a utilizar Bootstrap es un buen Snippet que nos ayude a autocompletar el código, se pueden encontrar varios dentro del buscador de extensiones de VSCode… obviamente hay plugins para cada editor

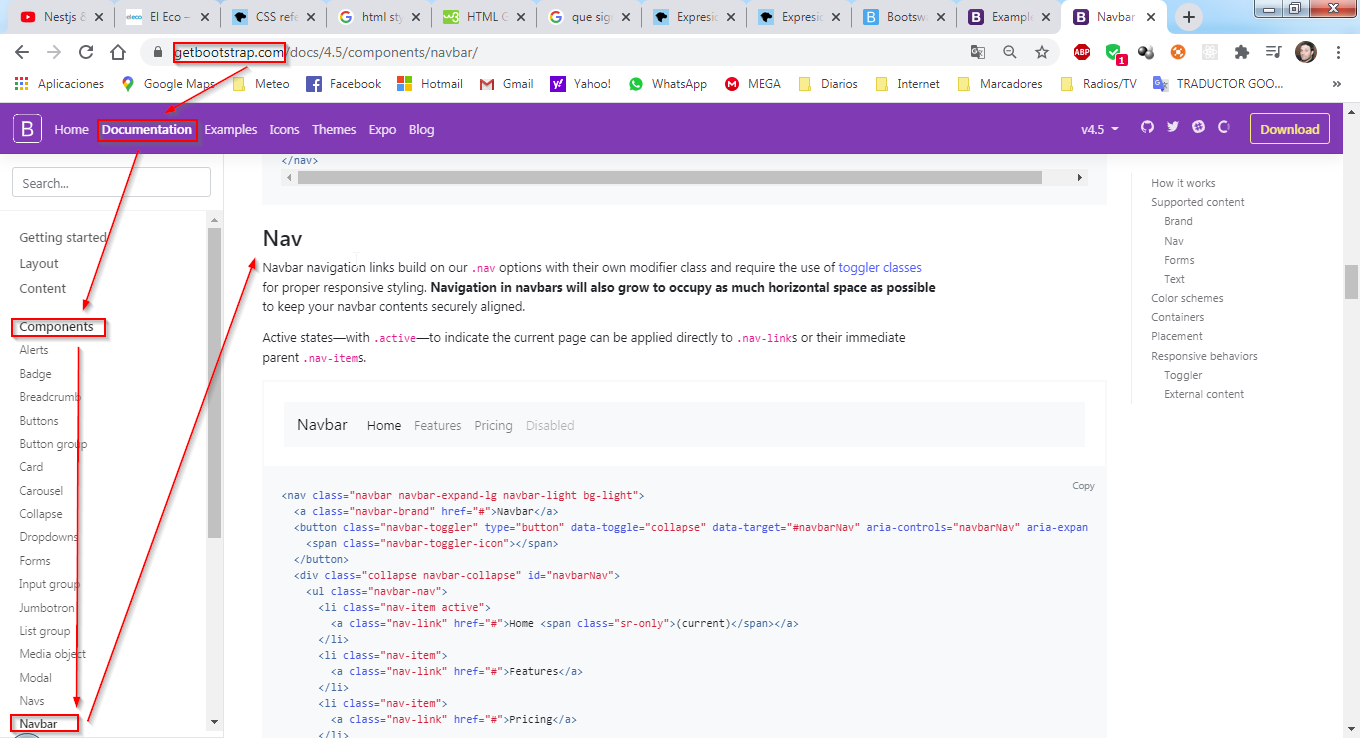


Pero en vez de tomar las URL’s de allí, lo hacemos desde **BootstrapCDN**|**Bootswatch** que nos permiten elegir ya estilos con diferentes formatos y colores.Elegimos por ejemplo Cosmo





y luego desde getbootstraps|documentación|Navbars selecciono **Nav** para copiar el código de una barra de navegación.



Hay algunas cosas que hay que reemplazar ya que no estamos tratando con html sino con JSX que no es lo mismo

* reemplazar class= por className=
* reemplazar href="#" por href="/"
* reemplazar la etiqueta **<a>** por **<Link>** que la obtenemos de importar de **react-router-dom** el modulo Link que no es más que una etiqueta **<a>** lo único que cambia en **<Link>** respecto de **<a>** es que en vez de **href** iria un **to**

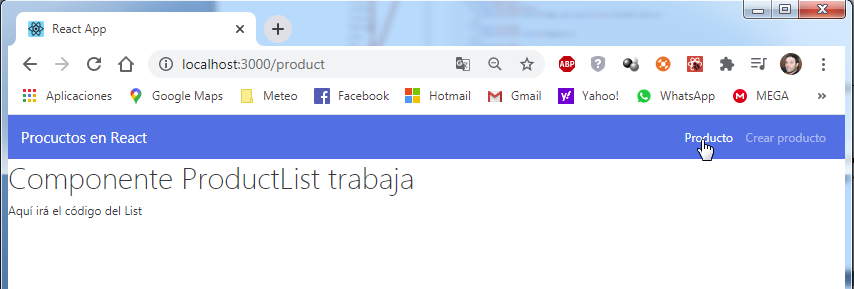
<a className="navbar-brand" href="/">React</a>

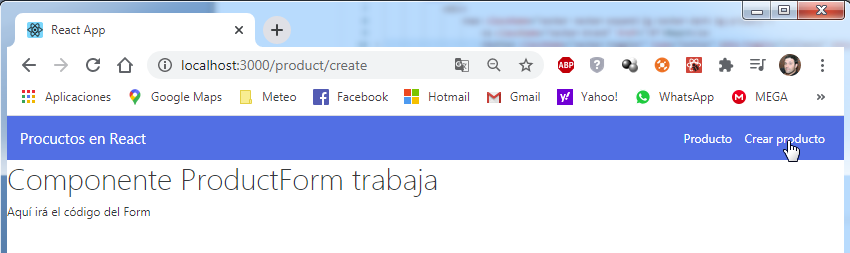
<Link className="navbar-brand" to="/">React</Link>

lo que se quiere lograr al trabajar con react y **react-router** es que en vez de usar el **href=** usar el enrutador de React lo hacemos con el **to=** indicando la ruta (en otra parte del código estará la relación ruta-componente). Esto nos dará el efecto de la navegación entre páginas aunque la aplicación es una aplicación de página única.

## 

Tener en cuenta que se mencionan las rutas raíz “**/**”, ”**/producto**” y “**/producto/create**”, “**/producto/create/:id**” aun no mencionamos mucho acerca del enrutamiento que será lo próximo. Cuando estén definidas las rutas en **app.js** los links de la barra de navegación cargaran los componentes indicados como la lista o el formulario de productos según sea el caso

****

****

## Enrutamiento

Luego de haber agregado la barra de navegación a la aplicación **app.js** podría pensar en agregar los demás componentes que definimos para nuestra aplicación. Pero lo que vamos a intentar hacer es que cada componente de nuestra aplicación se vea como una página dentro de nuestra aplicación de única página, es decir, que cada uno se corresponda a una URL o ruta. Por ejemplo **ProductList**, con la ruta raíz **‘\’**, o con la ruta **\producto** y el componente **ProductForm** con ruta **\producto\create**, en el caso de crear un nuevo producto o a la ruta **\producto\create\:id** en el caso de una actualización. Esto se hace a través de un enrutador y para el caso de React vamos a tener que una librería react – router y para ello vamos a tener que incorporar el módulo **react-router-dom**

npm install react-router-dom --save

luego importamos en el app.js dos componentes

**BrowserRouter**: nos permite crear o definir rutas para nuestra aplicación

**Route**: nos permite especificar dichas rutas

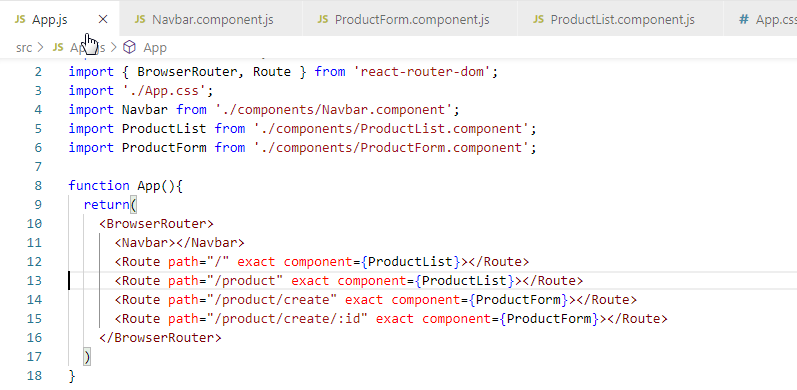
Lo que tenemos que hacer en principio es

* reemplazar en app.js la etiqueta **<div></div>** por **<BrowserRouter></BrouserRouter>**, el contenedor ahora va a pasar a ser **BrowserRouter** en vez del **div**
* la barra de navegación la dejo como esta
* definir mediante **<Route></Route>** las diferentes rutas de la aplicación, en este caso se deben especificar tres propiedades

**path=** a la que le correspondera la ruta o endpoint

**exact** que le indica que el nombre de la ruta debe ser exacto, pues por ejemplo **/product/create** además machea con las anteriores por lo cual aparecerían tres componentes… para que eso no ocurra hay que indicarle que el macheo de ruta sea exacto

**component=** el nombre del componente



# Agregando código los componentes

<https://www.youtube.com/watch?v=-LB0N_EO7X0>

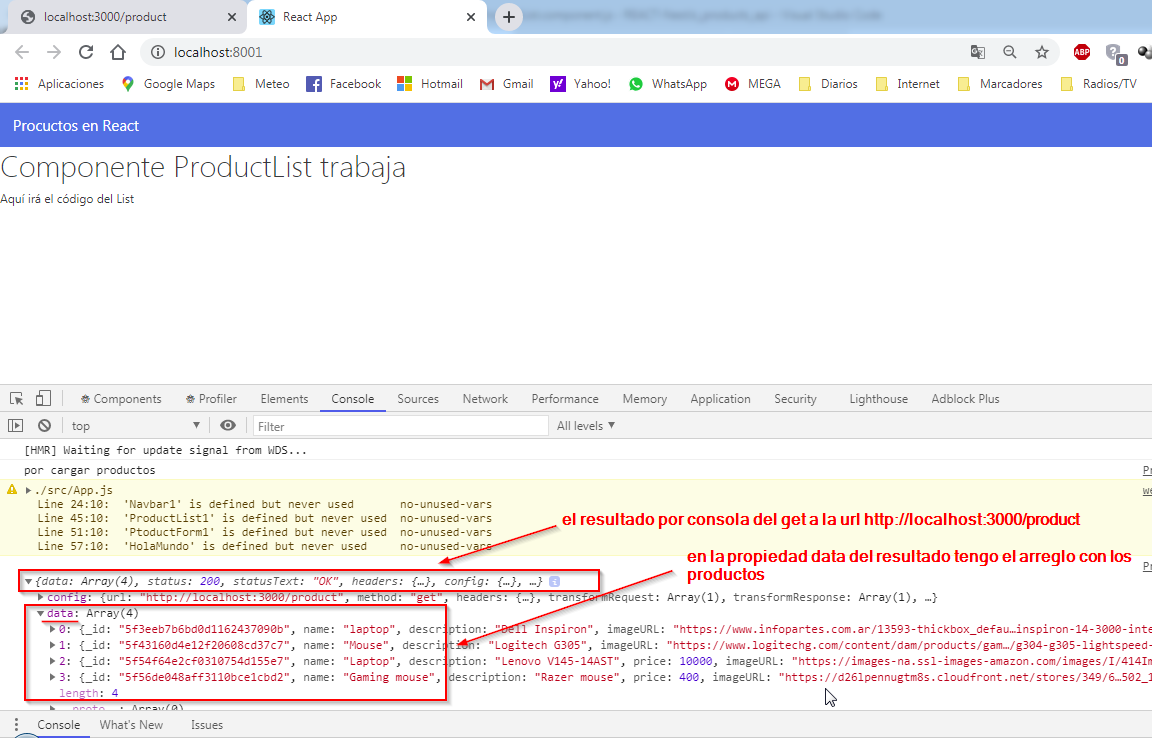
<https://www.youtube.com/watch?v=Hn7mStlvWFE>

## El componente ProductList

en el caso de los componentes como Form y List vamos a necesitar acceder a los datos que nos provee nuestro servidor back-end para mostrarlos en el componente. La clase componente provee un método **componentDidMount()** que permite ejecutar algunas funciones o código una vez que el componente ha sido montado, en nuestro caso lo vamos a utilizar para pedir los datos la back-end y mostrarlo por pantalla para hacer peticiones http utilizaríamos fetch() pero lo lógico es que en React deseemos utilizar alguna biblioteca http que tenga la funcionalidad ya escrita, ese el caso de **axios**.

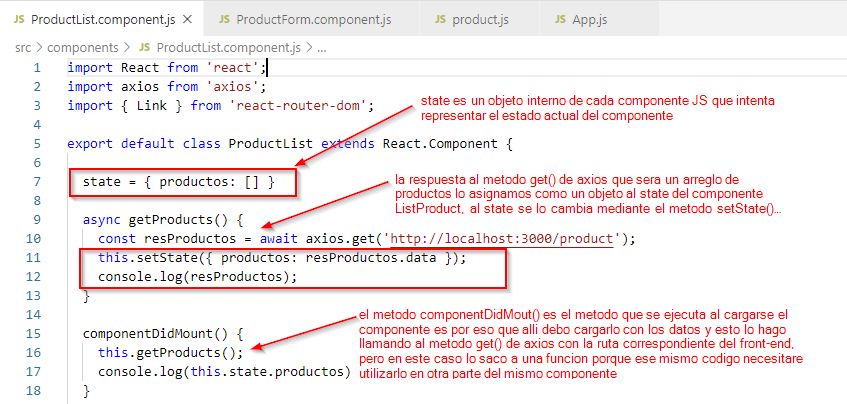


Por ahora lo que necesitamos ver, aunque no tengamos el componente desarrollado , es que al menos el método **render()** dibuja un titulo un párrafo, y que si miramos de de la consola en las herramientas para el desarrollador del navegador podemos ver el arreglo de productos con el cual nos responde el fron-end



Como vemos en el método **componentDidMount()**, llamamos al método **get()** de **axios** pasándole como parámetro la URL de la ruta front- end que requerimos y este nos retorna en una **Promise** todos los productos y el resultado se lo asignamos a una variable productos. Usamos **async await** para la funcion porque se trata de un pedido que hacemos hacia afuera de la aplicación y esperamos la respuesta en una promesa.

Por otro lado, para poder tener llegada a los productos obtenidos mediante el **get()** desde cualquier otro método del componente, lo lógico no sería tener el resultado en una variable del método, sino mantenerla en el estado **state** del componente. En JS el **state** es un objeto interno del componente que intenta representar el estado actual del componente). Entonces en la clase **ListProduct** al **state** le asignamos un objeto que contiene un arreglo productos de productos inicialmente vacio

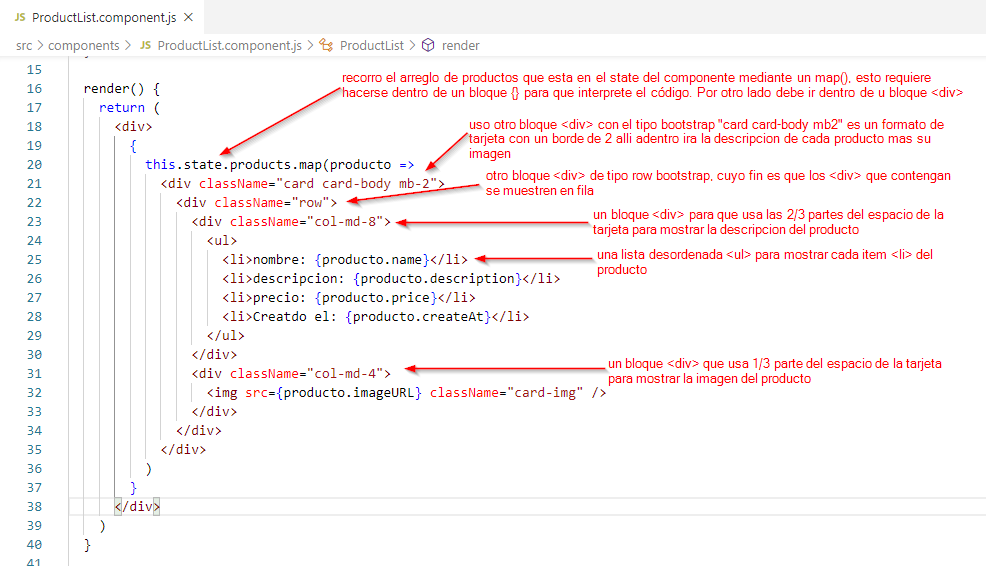


ahora agregamos código al método **render()** del componente que nos dibuja en pantalla el listado de los productos. El código es **JSX** con los estilos que podemos tomar de **bootstrap** para darle una mejor apariencia al los componentes de nuestro componente.

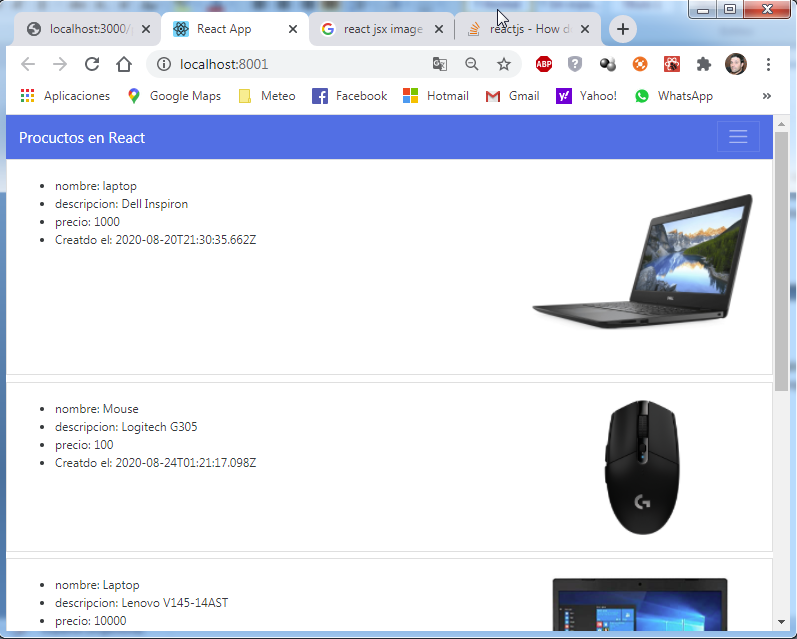
En este caso veamos las particularidades que se tuvieron en cuenta:

Recorrer el arreglo de productos que está en el state del componente. Esto se hace mediante el método **map()** de los arreglos, es necesario ponerlo dentro de un bloque **{}** para que pueda interpretarse el código.

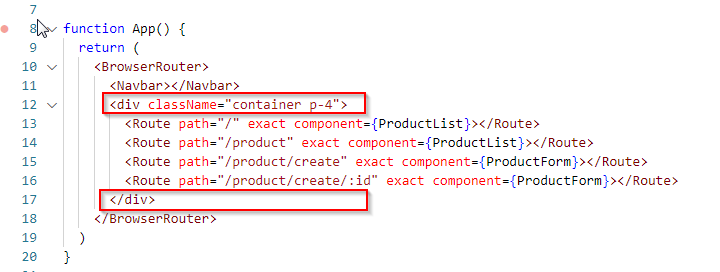
Dentro del render() debe ir al menos un elemento de HTML, entonces englobo dentro de un <div>.



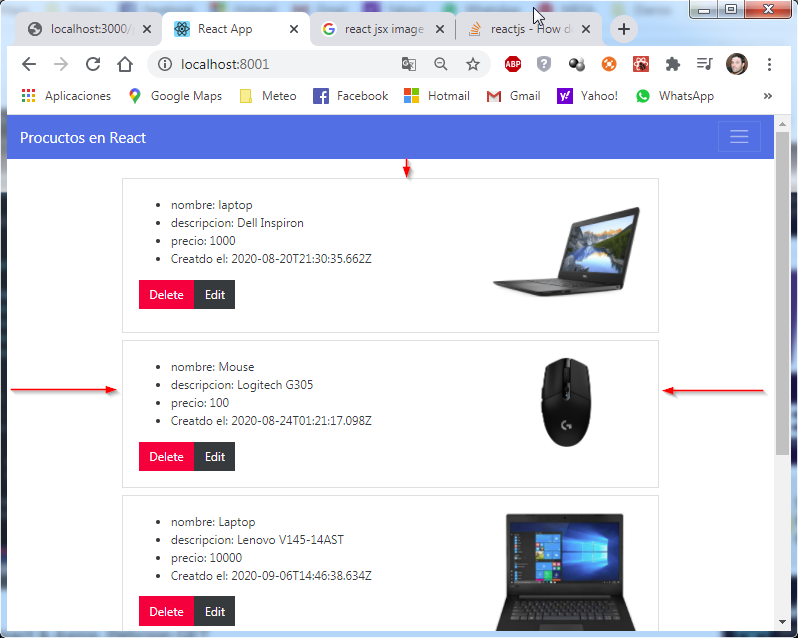
Por cada elemento voy a utilizar un bloque **<div>** con el estilo tarjeta que me brinda **bootstap** y se le da mendiante la propiedad **className**. Allí adentro irán dos bloques **<div>** más, uno que contendrá los datos del producto y otro la imagen del mismo, le damos el estilo mediante **bootstrap** necesario para que uno ocupe 2/3 de la pantalla y el otro 1/3 de la tarjeta. A su vez estos dos últimos para que aparezcan alineados horizontalmente es necesario ponerlos a su vez dentro de otro bloque **<div>** estilizandolo mediante la propiedad bootstap **className= row**.



En pantalla se puede ver que está todo muy cerca de los bordes de la pagina, agregando un bloque **<div>** con la propiedad **calssName=”container p-2”** en el archivo app.js encerrando las rutas



Se logra una separación q le otorga un mejor aspecto. Lo que busco mostrar es como se puede retocar desde diferentes lugares para lograr una vista más agradable del sitio.



En un próximo paso agregaremos botones para eliminar o editar un elemento de la lista.

## Manejo de eventos en React con JSX

Profundizar mejor en el sitio de React

En JSX los nombres de los eventos van con camelcase (a diferencia de HTML que va con minúsculas) y el nombre del manejador no va como string sino en un bloque {}, se llama con this pues el manejador es miembro de la clase y tampoco lleva paréntesis

**HTML** 🡪 <button onclick="activateLasers()">

**JSX** 🡪 <button onclick={this.activateLasers}>

Al implementar un manejador de eventos es habitual que se quiera acceder a las propiedades o métodos del propio componente. Por ejemplo, en el método toggleSlider() queremos cambiar el **state** del componente y por tanto necesitamos manipularlo (al **state**) con **this.setState()**.

Sin embargo aquí nos encontramos con una dificultad que viene del propio lenguaje Javascript no de React específicamente. En las funciones no es posible acceder a **this** como referencia al objeto sobre el que se invoca el método. Es por ello que necesitamos **bindear** el contexto.

Existen varias formas de hacer esto

* desde el constructor del componente

constructor(props) {

super(props);

this.state = {

showSlider: false

}

this.toggleSlider = this.toggleSlider.bind(this);

}

En este constructor estamos haciendo diversas tareas, la primera es la inicialización del estado, pero lo que nos interesa a nosotros es la parte donde se hace

this.toggleSlider = this.toggleSlider.bind(this);

gracias a la llamada a .bind(this) sobre una función o método, le estamos indicando a Javascript qué es lo que queremos que valga "this" dentro del método. Esto es algo que seguramente, si tienes experiencia en Javascript, ya habrás hecho en muchas otras ocasiones.

* También podemos hacer este bindeo de this (como contexto para en handler) a la hora de declarar el evento, en el template. El código nos quedaría más o menos así.

<button onClick={this.toggleSlider.bind(this)}>Mostrar / Ocultar Slider</button>

* También podríamos obtener el mismo resultado usando una arrow function, que es capaz de mantener el contexto de this, con un código como este:

<button onClick={(e) => this.toggleSlider(e)}>Mostrar / Ocultar Slider</button>

Aunque estas opciones nos ensucien algo el template, tienen la ventaja de ahorrarnos algo de ceremonia en el constructor del componente. Sin embargo, en términos de rendimiento resultan una solución peor. El problema de estas alternativas es que, cada vez que se renderiza el componente, se le pasa una nueva función como manejador de evento, donde se tiene que bindear el contexto nuevamente. Incluso, si la función se está pasando hacia componentes hijos, forzaría un redibujado adicional del componente hijo, lo que posiblemente nos lleve a una disminución del rendimiento que podrá notarse en aplicaciones grandes.

**Pasar parámetros a el manejador de eventos**

Dentro de un bucle es muy común querer pasar un parámetro extra a un manejador de eventos. Por ejemplo, si id es el ID de una fila, cualquiera de los códigos a continuación podría funcionar:

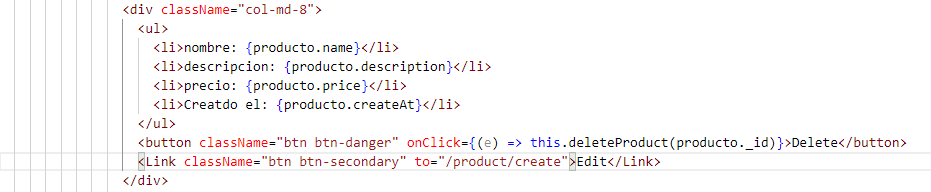
<button onClick={(e) => this.deleteRow(id, e)}>Delete Row</button>

<button onClick={this.deleteRow.bind(this, id)}>Delete Row</button>

Las dos líneas anteriores son equivalentes, y utilizan **funcion flecha** y **Function.prototype.bind** respectivamente.

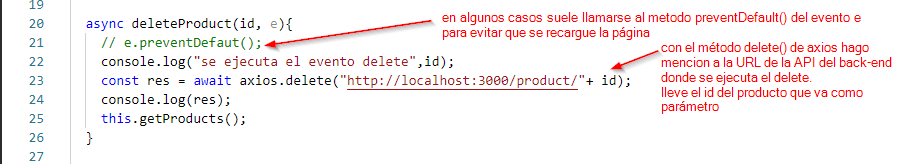
En ambos casos, el argumento **e** que representa el evento de React va a ser pasado como un segundo argumento después del ID. Con una función flecha, tenemos que pasarlo explícitamente, pero con bind cualquier argumento adicional es pasado automáticamente

Entonces para el botón Delete y el Edit en el listado voy a utilizar algo así

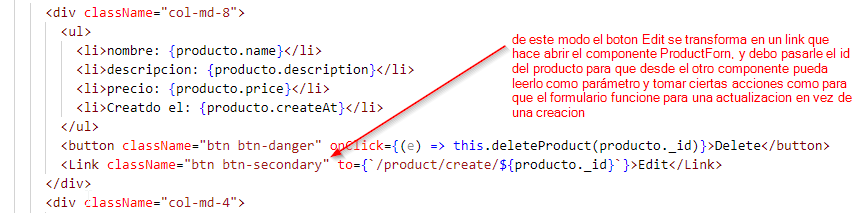


Para el **Delete** un **<button>** con estilo del **bootstrap** **className=“btn btn-danger”** que le dará la apariencia que muestra y por otro lado indico que escuche el evento **onClick** y en ese caso llama a la funcion **deleteProduct()** con el \_id de producto seleccionado para identificarlo.

Defino en la misma clase la función que maneja el evento **deleteProduct(\_id, e)**



Mientras que para el boton **Edit**, más que un <button> uso un **<Link>,** esta etiqueta pertence a la librería *react-router-dom*, por lo cual primero debo importarlo desde dicha librería para usar el **<Link>** en la clase. A este componente **<Link>** le doy el aspecto de botón con **bootstrap** al indicarle **className=”btn btn-secondary”** y por ultimo en **to=** le indico la ruta correspondiente, eso hará que levante el **ProductForm**, pues la ruta se corresponde a ese componente

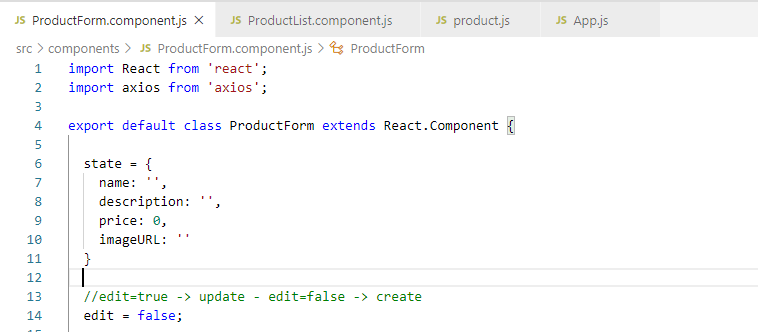


debo sumarle a la ruta el id del producto, luego veremos cómo desde el componente **ProductForm** utilizamos éste al leerlo como parámetro para determinar el funcionamiento del botón Edit de aquel componente. Según la ruta venga con el parámetro id o no (/product/create o /product/crete/id) sabrá si se trata de una actualización de producto o de la creación de uno nuevo.

## El componente ProductForm

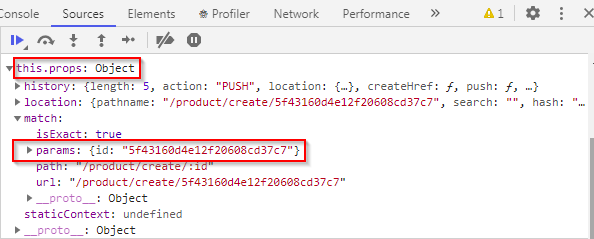
Para este caso se va a proceder de una manera similar vamos a definir el **state** para este componente pero esta vez el objeto será con ítems del producto que necesita mostrar el componente, es decir, del producto su nombre la descripción el precio y la URL de la imagen, el id como la fecha de creación que también son parte del producto como en este componente no son visibles no será necesario tenerlos en el **state**.

Ademas del estado voy a definir una propiedad **edit** para el componente que utilizare para saber cuando el componente se está utilizando para crear un nuevo producto o cuando se lo uza para actualizar un producto existente. Inicialmente lo seteo a **false** lo que indica que el componente es para crear un producto. Esta propiedad en algún momento se seteara a **true** cuando al componente se lo llame desde el componente **ListProduct** desde el botón **Editar** de algún producto.



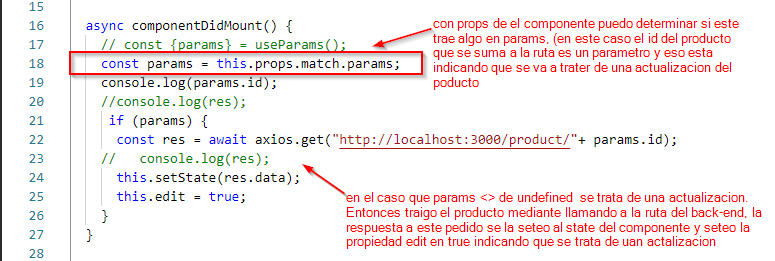
en método componentDidMount tenemos un punto importante (y casi mágico pues me costó encontrar la vuelta de cómo hacerlo) en este punto se usa la técnica de **props** que permite compartir codigo entre componente en React

Para verlo un poco mejor extraje esta imagen desde la consolas de herramientas para el desarrollador en el navegador mientras debugeaba y allí puede verse el objeto **props**, en particulardentro de **match** y a su vez en **params** puede verse el objeto {id: 5f43160d4e12f20608cd37c7}

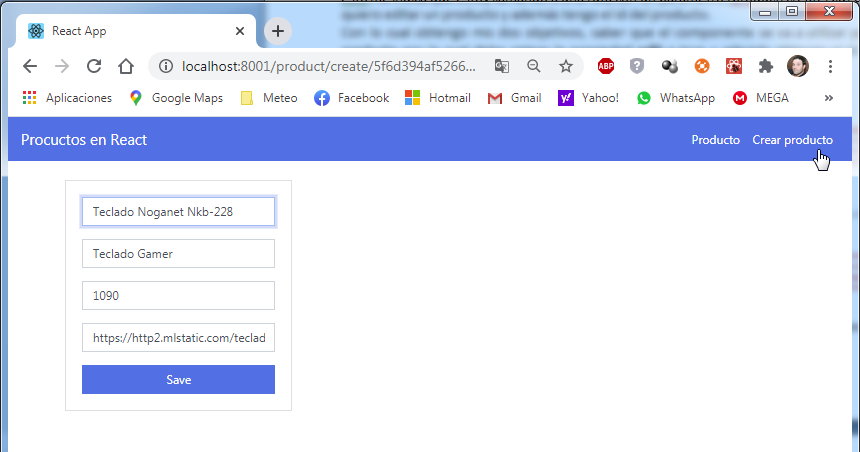


esto me indica que estoy llamando a una ruta con un parámetro (product/create/:id) con lo cual se trata de que quiero editar un producto y además tengo el id del producto.

Con lo cual obtengo mis dos objetivos, saber que el componente se va a utilizar para la actualización de un producto por lo cual debo setear la propiedad **edit** a true y además obtengo el id del producto para poder solicitarlo al back.-end mediante la API http correspondiente

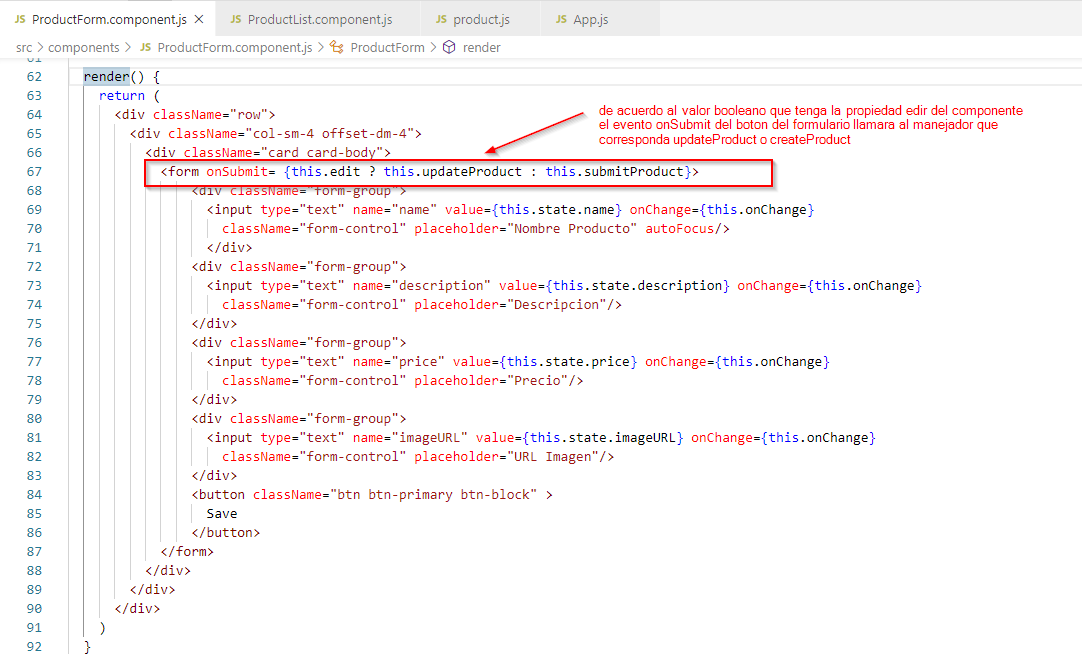


solicito mediante axios.get() el producto al back-end indicándole la URL, y el resultado lo seteo al estado del componente, de esta manera cuando el mismo se renderize mostrara los valores de dicho producto y el componente sabrá que se trata de una actualización .



Ahora, ya que vimos la apariencia del componente **ProductForm** en la pantalla veamos el método **render()** que lo dibuja.

En si consta de un **<form>** y una serie de bloques **<div>** cada uno con su estilo, dado por la propiedad **className** de **bootstap** seleccionada para darle la fisonomía que se le quiera dar. Algunos bloques **<div>** lo acomodan dentro de la pagina y otros acomodan los bloques **<input>** y el **<button>** que van dentro del formulario.



quizá las propiedades para darle los estilos haya que buscarlas bien en **boostrap**, lo interesante aquí es el evento **onSubmit** en ese caso de acuerdo al valor que posea la propiedad **edit** del componente llamara la menejador **this.updateProduct()** o al **this.createProduct()**, ese valor ya vimos arranca por default en **false** y al montar el componente puede cambiar a **true**.

Solo resta ver los manejadores para el evento onSubmit.

Por razones de cómo esta definido el componente (en este caso mediante una clase, pero podría haberlo estado por medio de una funcion ya que JS me permite definirlo de ambas maneras), las funciones las voy a definir de modo flecha es decir

submitProduct = async (e) => {}

en vez de

submitProduct (e) {}

por una cuestión de que reconozcan dentro del código las referencias a las propiedades o métodos que pueda hacer del mismo componente mediante **this**, sino tengo que hacer un **bind** de las funciones dentro de un **constructor()** de la clase del componente. Eso esta explicado un poco más arriba en manejo de eventos dentro de componentes.

En si los manejadores lo que hacen es llamar a la API http correspondiente del back-end mediante axios.post o **axios.put()**, obviamente en el caso de una actualización primero deberá llamar a **axios.get()** para traer los valores actuales del producto y luego mediante el **put()** actualizarlos. en el caso del create los datos estarán con su valor inicial vacio y cuando los haya cargado habrá que llamar al **post()**. En ambos caso nos manejamos con async-await ya que se solicitan afuera y retornan en una **promise**.

Al principio de ambos métodos hay que llamar al método **e.preventDefault()** (método del evento que viaja como e) pues en ambos casos cuando guardo el formulario hace una recarga de la pagina, lo que produce un pestaneo y no es necesario.

Si luego de guardar puedo requerir limpiar los **<imput>** o sencillamente podría saltar al componente **ProductList**





Con esto quedaría funcionando el fron-end que utiliza el back-end de productos pero esta vez desarrollado con React

Una de las cosas que puedo pensar es en que desde el componente **productForm**, una vez que guardo los cambios sea de una actualización como el alta de un nuevo producto, quiera pasar al componente **productList**. Desde el manejador del evento del botón save **updateProdut()** o **submitProduct()** tengo que utilizar esta sentencia

this.props.history.push('/product');

con ella logro saltar al componente de listado de productos desde código JS (no JSX). Es un tanto rara la forma de hacerlo, pero es lo que encontré de como otros hacían esto desde el código de un manejador.

Generalmente a los componentes llego a través de las rutas que definí con **Route** en el app.js. Desde la barra de navegación (componente Navbar), o desde los JSX que generan los métodos **render()** de otros componentes utilizando el **<Link to=></Link>**

En el componente Navbar por ejemplo

<Link className="nav-link" to="/product">Producto</Link>

<Link className="nav-link" to="/product/create">Crear producto</Link>

En el componente ProductList

<Link className="btn btn-secondary" to={`/product/create/${producto.\_id}`}>Edit</Link>

pero estas formas siempre son mediante el JSX. Pero cuando quiero hacerlo desde código JS, por ejemplo desde el código de un manejador de eventos, la única forma que conseguí hacerlo es así