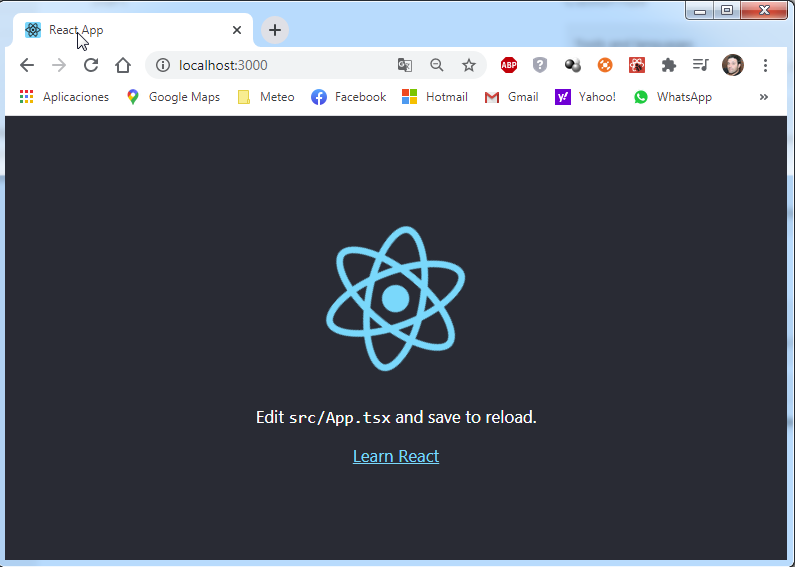
Crear un front-end con REACT y typescript para usar la API Rest de productos

Para crear un proy ecto React que use TS debere de ejecutar el siguiente comando

npx create-react-app REACT-TS\_NestJs\_products\_api --template typescript

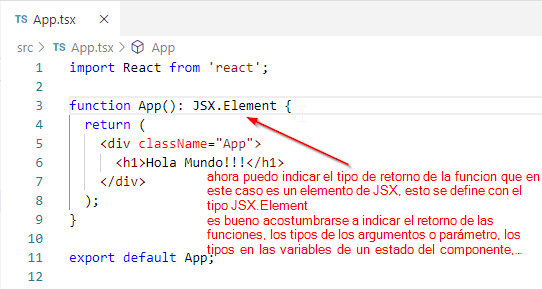
lo que si, la creación en un pricipio me da error y pide que el nombre debe ser en minúscula



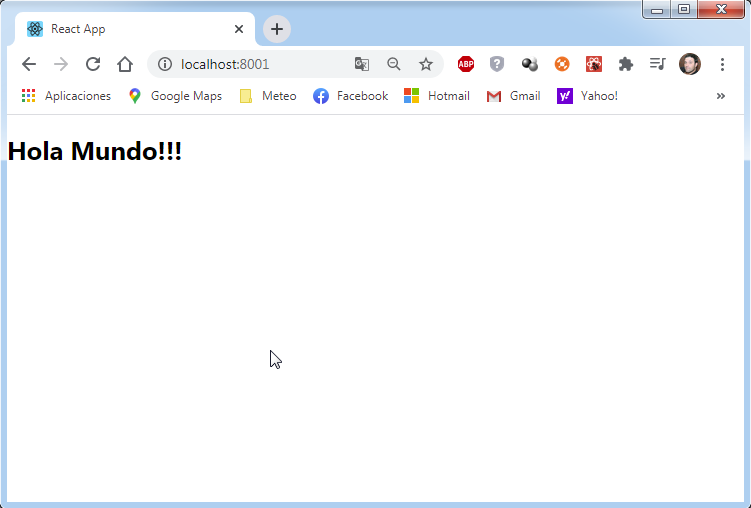
Si observamos en la carpeta que nos crea, agrega un archivo typescript.json que contiene la configuración básica de Typescript que en principio no es necesario tocar, pero quizá en algún momento se requiera modificar o agregar algo. También notaremos un cambio en las extensiones de los archivos ya no serán .js sino .tsx, asi como los .html (jsx) ahora serán .tsx. Prácticamente el contenido será el mismo aunque tenderemos la posibilidad de declarar tipos y algunas otras características que nos da TS.

De la misma manera a como lo hice en el proyecto de React con JS para cambiar el puerto donde levanta el servicio agrego el archivo **.env** en el directorio raíz del proyecto indicando PORT=8001.

Puedo editar el archivo App.tsx y allí eliminar el código JSX que renderiza el componente App para poner en principio un JSX con un Hola Mundo



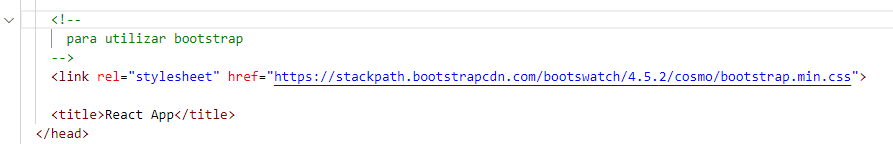
allí se puede definir ahora el tipo de retorno de la función que en este caso es un elemento JSX (JSX.Element). Ahora es bueno acostumbrarse a declarar los tipos de los retornos de las funciones, el tipo de los argumentos o parámetros asi como el tipo de las variables de un estado del componente, los tipos en la declaración de una interfaz, etc.



npm install react-router-dom --save

npm install @types/react-router-dom --save

Agregamos tambien Bootstrap, no vamos a incorporar la libreria de Bootstrap a traves de npm, sino que lo haremos agregando la URL de los estilo en el bloque <head> de Public/main.html



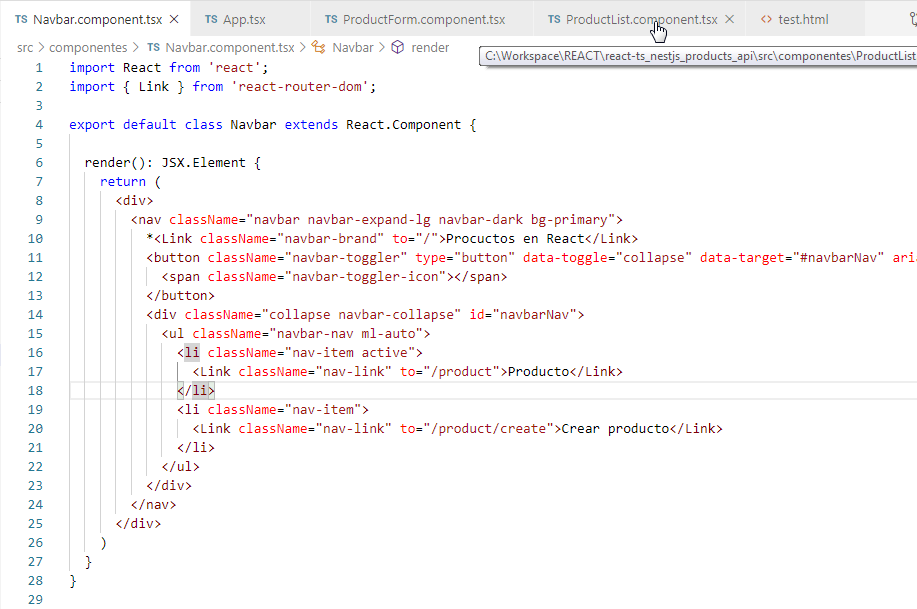
con esto podremos darle estilo a los JSX de los componentes.

En el archivo app.tsx vamos a reemplazar el bloque <div> por un <BrouserRouter> de manera que podamos definir ya las rutas y agregar una barra de navegación, en si no hay muchos cambios en este archivo respcto al del mismo proyecto pero con JavaScrip

## 

## Agregando la barra de navegación

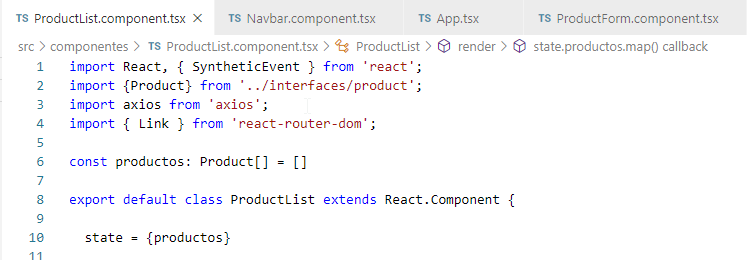
Como hicimos en el otro ejemplo tomamos el código para la Navbar que nos brinda **Bootswatch** para el estilo **Cosmo** que elegimos y lo colocamos dentro del método **render()** del componente **Navbar** que creamos en el archivo **/componentes/Navbar.componente.tsx**



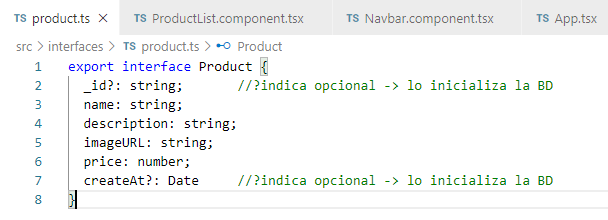
Tampoco hay cambios relevantes en el código de la barra de navegación respecto a la del mismo proyecto pero con JavaScrip. Solo que aquí el método **render()** puede retornar un tipo **JSX.Element**. Cabe aclarar que el método render() solo puede tener un elemento, en este caso un bloque **<div>** luego dentro de este sí hay un anidamiento.

## El componente ProductList

Los cambios en este componente están referidos en lo que a definición de tipo en el **state** y en los parámetros de las llamadas a los manejadores de eventos



En el manejador del evento **onClick** del botón **Delete** por ejemplo se agrega el tipo del argumento **e** de la función flecha, que a su vez llama al método **deleteProduct(producto\_id, e)**, en este caso **producto.\_id** está definido en la interface **producto** como **opcional** y de tipo **string** y

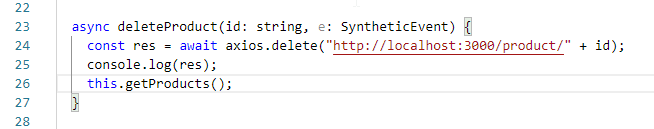


En la definición de la funcion **deleteProduct(id: string, e: SyntheticEvent)** el parámetro id se define como **string**, pero éste, por su definición en la interfaz **producto** como **opcional** (**?**), puede ser **undefined** y esto da un error al momento de transcompilar, para salvar el error en la llamada a la función se le debe colocar el operador **non-null assertion** (**!**) sobre el final **this.deleteProduct(producto.\_id!, e)**

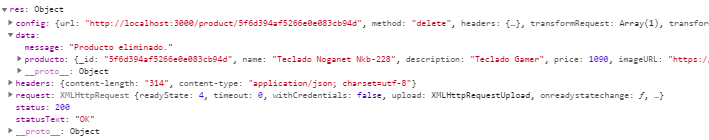


La funcion **deleteProduct()** entonces recibe el id del producto y el objeto **SyntheticEvent** (que en este caso no tiene sentido pero si lo tendrá (en las altas y las actualizaciones ya que se utilizará uno de sus métodos para evitar la recarga de la pagina).

Lama al método delete() de **axios** con el URL de la API del back-end que efectua el delete y luego llama al metodo **getProducts()** del componente que lo que hace es refrescar la lista en la pagina, que ya no mostrará el producto eliminado.



Nos manejamos con la variable res para recibir el resultado de la **Promise** que retorna el **axios.delete()**, en dicho objeto puede verse entre otras, la propiedad data que tiene el mensaje “Producto eliminado” y el producto, cosa que había enviado el back-end.



**Nota**: A los manejadores de eventos se les pasarán instancias de **SyntheticEvent**, **e** un contenedor ‘agnóstico’ al navegador alrededor del evento nativo del navegador. Tiene la misma interfaz que el evento nativo del navegador, incluyendo stopPropagation() y preventDefault(), excepto que los eventos funcionan de manera idéntica en todos los navegadores.

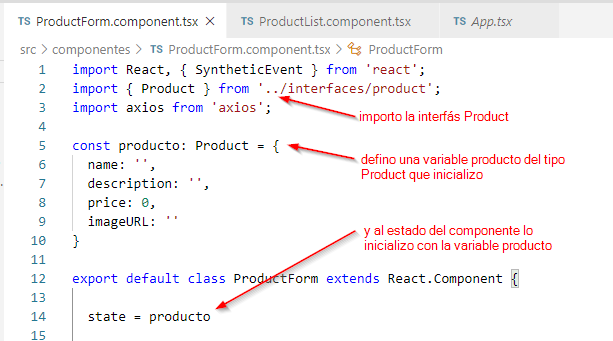
En caso de necesitar el evento del navegador subyacente por alguna razón, simplemente use el atributo nativeEvent para obtenerlo. Los eventos sintéticos son diferentes y no tienen una correspondencia directa con los eventos nativos del navegador.



## El componente ProductForm

Los cambios en este componente están referidos en lo que a definición de tipo en el **state** y en los parámetros de las llamadas a los manejadores de eventos.

Para el **state** ahora que puedo utilizar tipos de datos uso la interfaz **Product**



Otra cosa que utilizaba en el desarrollo anterior para el componente **ProductForm** era la lectura de **params** en las **props** del componente en el método **componentDidMount()**, esto lo hacía, para poder determinar si el componente se usa para una actualización o para el alta de un nuevo producto, ya que, aunque la vista del componente es la misma, el comportamiento no es el mismo.

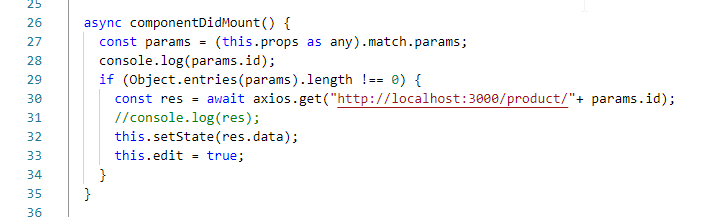
Ahora esa línea donde se leían **params** da error

const params = this.props.match.params;

y hay que hacer algun cambio

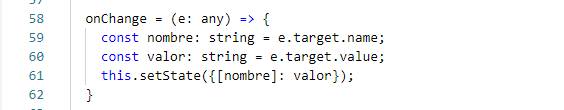
const params = (this.props as any).match.params;

y tambien hay que hacer un cambio en la verificación del contenido de **params**, se ve que con el uso de TypeScript ya no retorna **undefined** como antes cuando no había parámetros, ahora retorna un objeto **{}** y el control de esto se hace verificando que el contenido del objeto sea de 0 elememtos.



Entonces luego de renderizár el componente se ejecuta el **método componentDidMount()** y de acuerdo a si la ruta en que se llamo al componente trae el parámetro id de producto, pide al back-end información del producto con dicho id de producto y sete el estado con los datos de la respuesta del **axios.get()** y configura el boolean **edit** a true para que el botón **Save** del formulario responda llamando al manejador de eventos correspondiente.

El manejador del evento **onChange()** que carga en el **state** lo que el usuario va cargando en los cuadros de edición del formulario **<input>** también cambia respecto a como lo hacía en el desarrollo anterior con JS



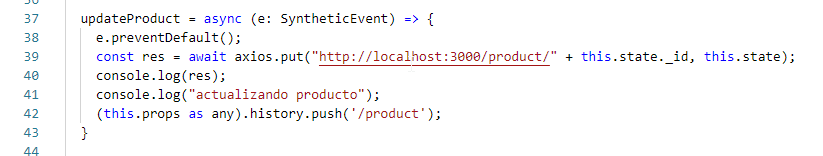
de acuerdo al **<input>** del formulario que corresponda tomo los valores de las propiedades **name** y del **value** y lo seteeo en el **state** el valor que corresponda

el método **render()** tiene el mismo aspecto que en el desarrollo con JS, solo que muestro ahora puedo indicar el tipo de lo que el médodo retorna que es un objeto del tipo JSX.Element.

el evento **onSubmit()** del formulario al igual que en el desarrollo con JS evalúa el valor de de la propiedad **edit** del componente si es true llamara al manejador de eventos correspondiente a las actualizaciones **this.updateProduct** y si es false llamará al evento de el alta de producto **this.submitProduct**



los manejadores de eventos se van a manejar de la misma manera, utilizando funciones flecha por el tema de no tener que vindearlas a cada una en el constructor de la clase para que reconozcan los elementos referenciados con **this** en su codigo. Utilizarán el evento **e** como parámetro para llamar al método preventDefault() que impide que la pagina se recargue evitando el pestañeó. Y sobre el final agregue una sentencia que lo que hace es que cuando termine enrute al componente **productList**



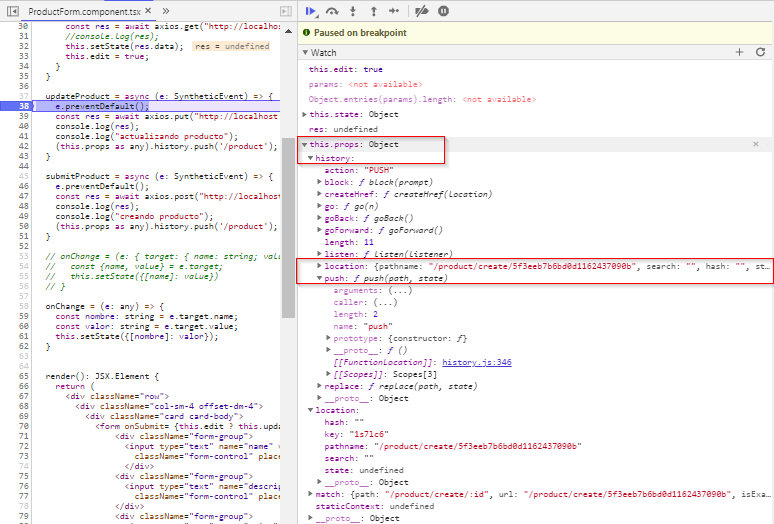


es la forma que encontré para realizar desde el código un ruteo a otro componente, no se si existe alguna otra.

Si observo con el debug **this.props.history** al inicio de la función para **location** el valor **path** esta con valor de la ruta que corresponde al componente **productForm** (/product/create/:id) y luego de que ejecuto la sentencia

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

el contenido de **path** en **location** para **this.props.history** cambia a la ruta del componente **productList** (/product)



En el siguiente paso se ve el cambio en la ruta

