

**React**

**Índice**

Introducción de React en JavaScript …………………………… **3**

Componente …………………………………………………………………. 5

Props y State ………………………………………………………………… 6

Renderizado condicional ………………………………………………… 8

Proyecto en React …………………………………………………… **9**

Componente ……………………………………………………… 13

Props ………………………………………………………….…… 16

Children ………………………………………………………………… 18

Estilos ………………………………………………………..……. 19

Íconos …………………………………………………………..…. 22

Eventos …………………………………………..…………..…… 23

Formularios ……………………………………………………………. 24

Arrays ………………………………………………….………….. 25

JSON ……………………………………………………………… 26

Hooks ………………………………………………..……………. 27

State ……………………………………………………………………. 27

Effect …………………………………………………………………… 28

Context ………………………………………………………………… 29

Ref y LayoutEffect ……………………………………………………. 32

Memo ………………………………………………………………….. 33

Callback ……………………………………………………………….. 34

Reducer ……………………………………………………………….. 35

Otros Hooks ………………………..…………………………………. 36

React Router (navegación entre páginas SPA) ……………………… 37

createBrowserRouter ………………………………………………… 38

Routes …………………………………………………………………. 40

React Redux (gestor de estados globales) …………………………. 42

React Query (gestor de peticiones externas) ………………………. 48

Build ………………………………………………………………. 54

GitHub pages ……………………………………………………. 55

Modularización …………………….…………………………….. 56

React Native ………………………….…….…….…………………… **58**

Widgets …………………………………………..…………..…… 60

Explorar galería ………………………………………………….. 63

Share ………………………………………………………………

Splash Screen & Icon ……………………………………………

Deploy ……………………………………………………………..

**Introducción de React en JavaScript**

React permite la construccion de interfaces de usuario tanto para aplicaciones web como apps nativas.

Es un framework que cambia el tipo imperativo de JS por un tipo declarativo.

Está basado en componentes.

Para comenzar a usarlo en nuestra página web, es necesario importarlo en nuestro <head> con los siguientes scripts:

React Core:

<script *src*="https://unpkg.com/react@18/umd/react.development.js"></script>

React para Páginas Web:

<script *src*="https://unpkg.com/react-dom@18/umd/react-dom.development.js"></script>

JavaScript XML (**JSX**) para pruebas de funcionamiento (sólo sirve para testing porque enlentece la página). Para que funcione, debes usar el atributo en el script donde se encuentre el código a probar:

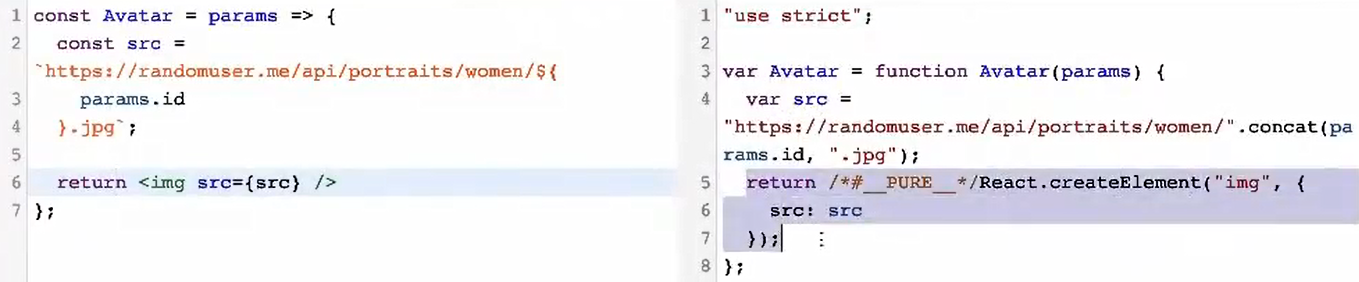
<script *src*="https://unpkg.com/babel-standalone@6/babel.min.js"></script>



Para que este último script funcione, se debe usar el atributo

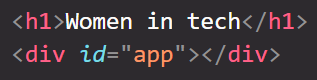
en el archivo de JS de nuestro código para transformarlo en código válido de React / JavaScript.

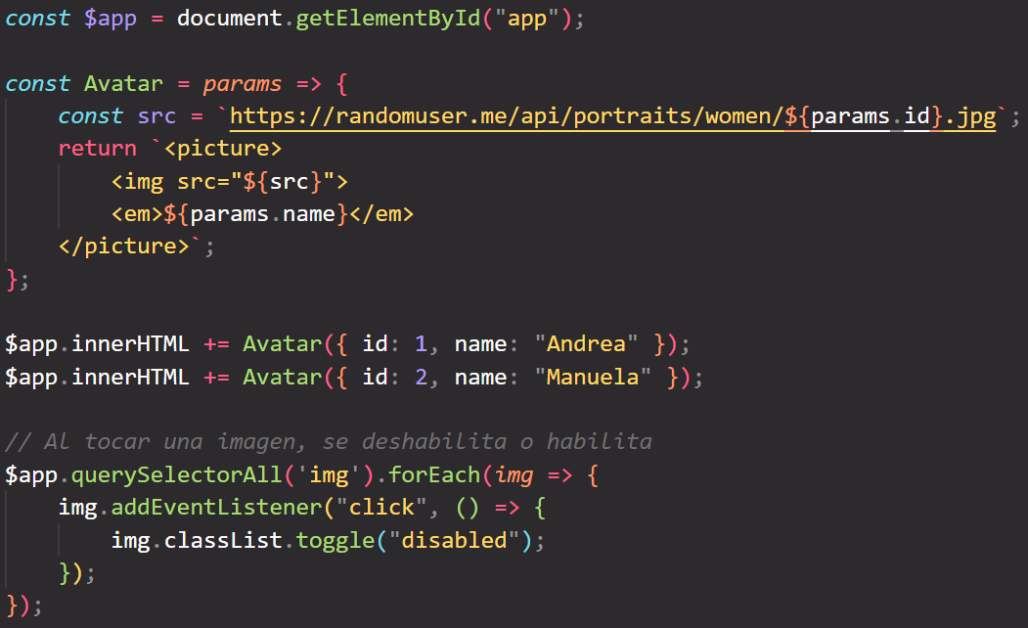
Se pueden hacer pruebas de cómo transforma código aquí: <https://babeljs.io/repl>



ReactJS

JSX

Sin React, procedíamos de la siguiente manera:





Esto implica crear las imágenes una a una, y luego recorrerlas a todas para aplicar, en cada una, el evento que ocurrirá al clickear sobre ellas para cambiarles la clase “disabled”. Esta forma de trabajar se denomina “imperativa” ya que ordenas a cada elemento tras ser creado una accion / evento.

Componente

React cambia esta forma de trabajo de JavaScript por una más declarativa y, junto con el uso de JSX, más amigable a la vista del desarrollador.

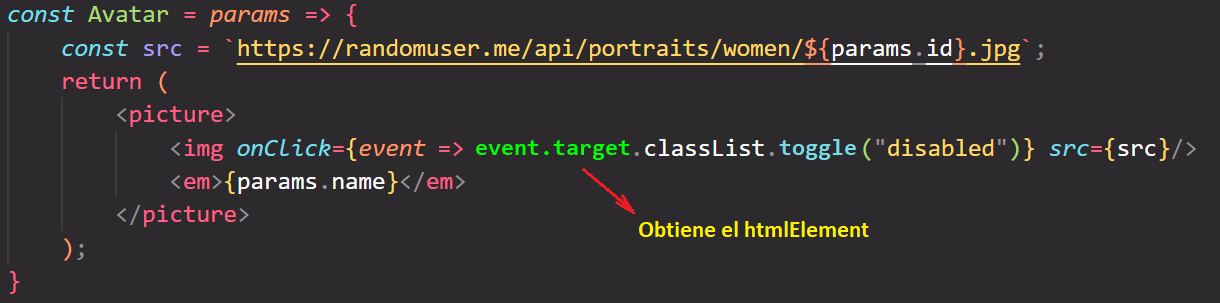
Podemos crear componentes (porciones de interfaz) de la siguiente manera



// Componente

Y, para que tengan funcionalidad, se declara el método onClick al momento de crearlos (*aclaración*: no es lo mismo que **onclick** de **HTML**, la cual es una mala práctica)

En este caso, se puede utilizar así:



Sin embargo, esta forma sigue siendo imperativa, ya que se le está “ordenando” una acción a partir de un evento.

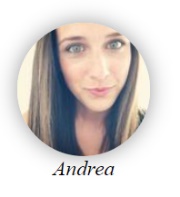
Props y State

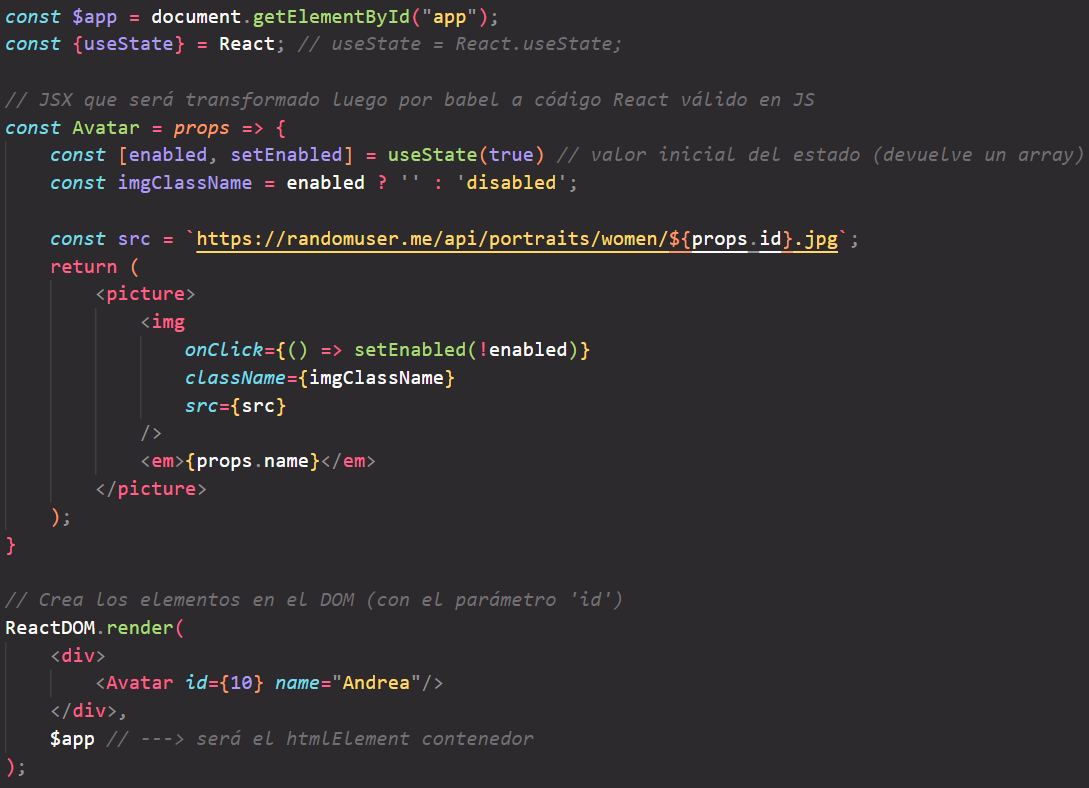
Las **props** son las propiedades de un componente, en general son los parámetros que recibe un componente al momento de ser creado.

Define también comportamientos y otros atributos de un componente.



El **state** es el estado en que se encuentra un componente, por ejemplo, si un componente se encuentra activo o inactivo.

En este caso:

Cada vez que que el **prop** o el **state** de un componente cambia, React vuelve a renderizar el componente para que se visualice actualizado, haciendo las modificaciones mínimas para evitar usar recursos excesivos para volver a cargar todo.

El atributo **class** de **HTML** se llama **className** en **JSX** ya que ”class” en **JavaScript** está siendo usado para otra función (creación de clases / objetos).

* Desestructuración de props

Se pueden desestructurar parámetros (**props**) al crear un componente.





De esta manera, se usarán solo las propiedades de ***id*** y ***name*** en la creación del componente.

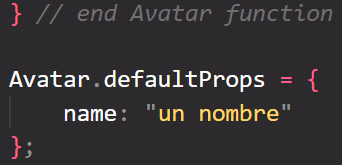
Además, ya no será necesario usar la palabra *props* para acceder a la propiedad.



* Default props

Teniendo las **props** desestructuradas, se le pueden asignar valores por defecto.



Otra forma más antigua de crear un Default prop es mediante la declaración de los valores por defecto por fuera de la función.

Renderizado condicional

Se trata de renderizar una cosa u otra según las **props** o **state** que se tengan declaradas.

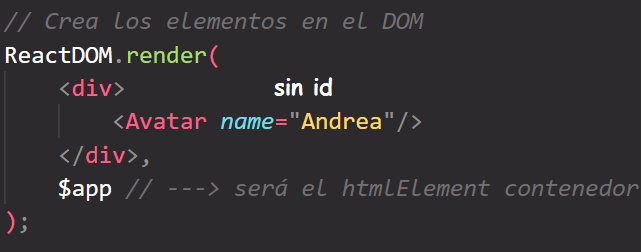
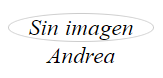
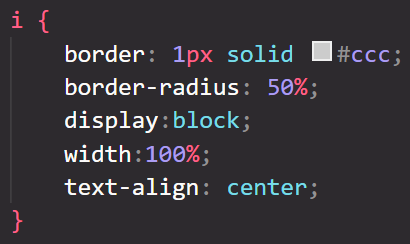
Siguiendo el ejemplo anterior, si se creara un Avatar sin nombre, tenemos definida una prop por defecto. Sin embargo, si se creara un Avatar sin id, se crearía un componente extraño.

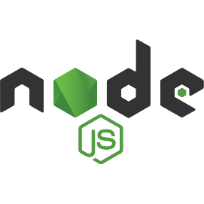
Para evitar que esto suceda, se pueden añadir condicionales como el siguiente:



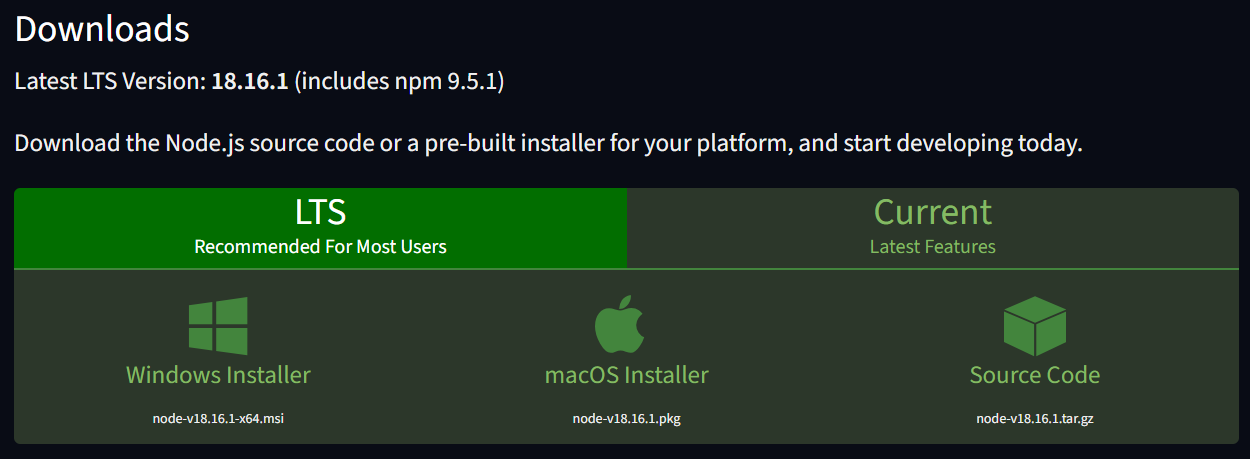
Si el estado está activo, se mostrará el nombre. Si no, “desactivada”.

Se ejecutará esto si la *id* **no** está definida



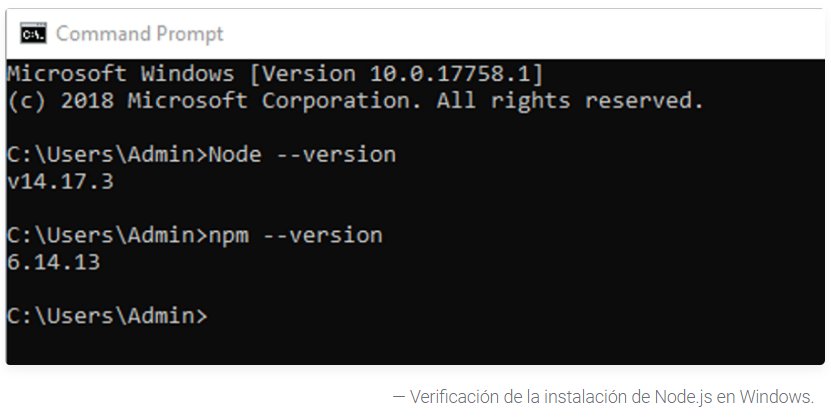
**Proyecto en React**

Para iniciar un nuevo proyecto de React, es necesario **NodeJS** <https://nodejs.org/en/download>



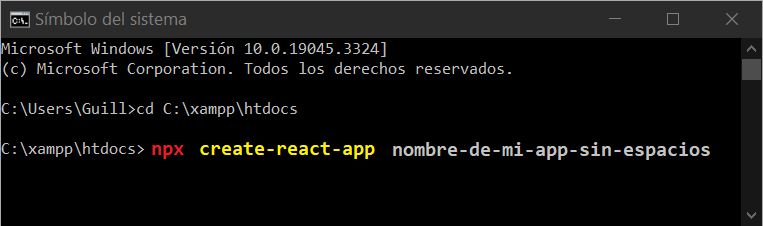
La versión LTS es la más estable, mientras que Current es la última sin soporte actualizado.

Una vez descargado e instalado, comprobar que la instalación fue exitosa abriendo la consola del sistema y verificar la versión instalada.



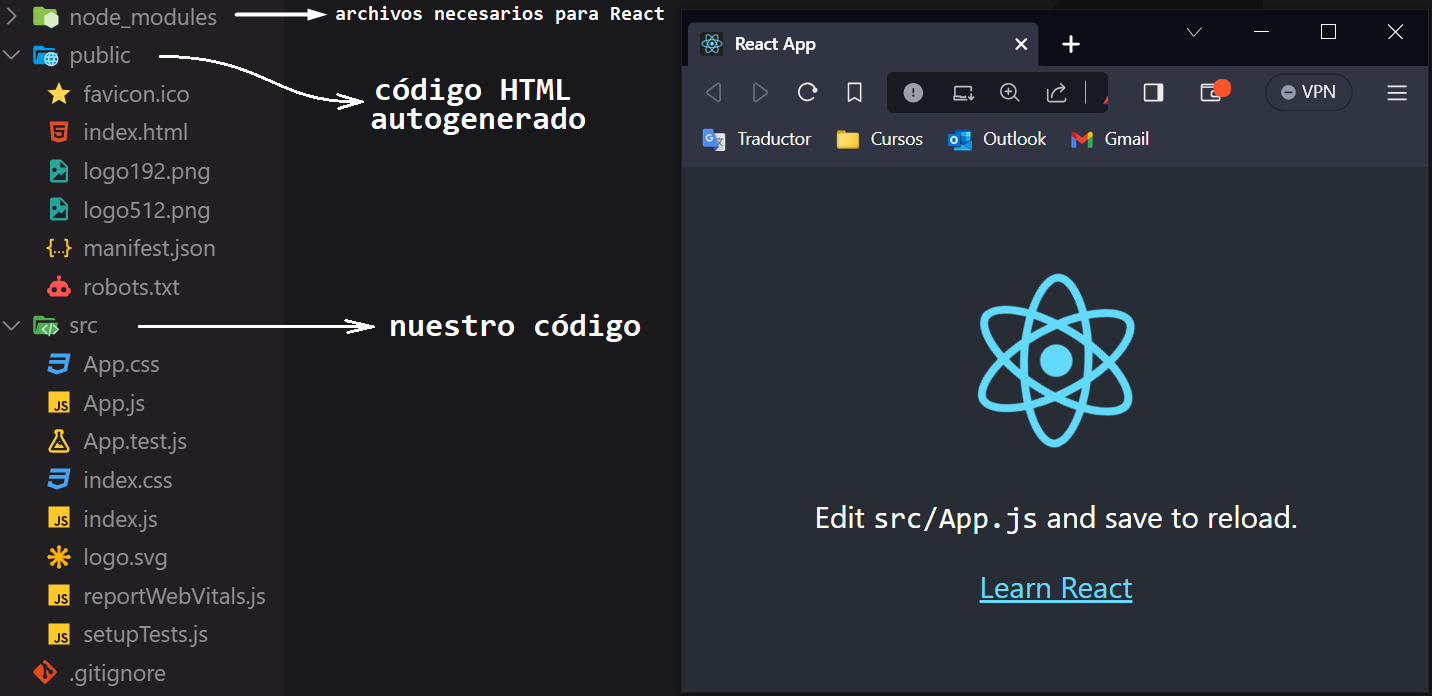
Teniendo instalado **NodeJS**, se puede iniciar un nuevo proyecto desde la terminal, estando ubicados en el directorio donde se quiere crear el repositorio.

Es necesario tener conexión a internet para descargar todos los paquetes necesarios.

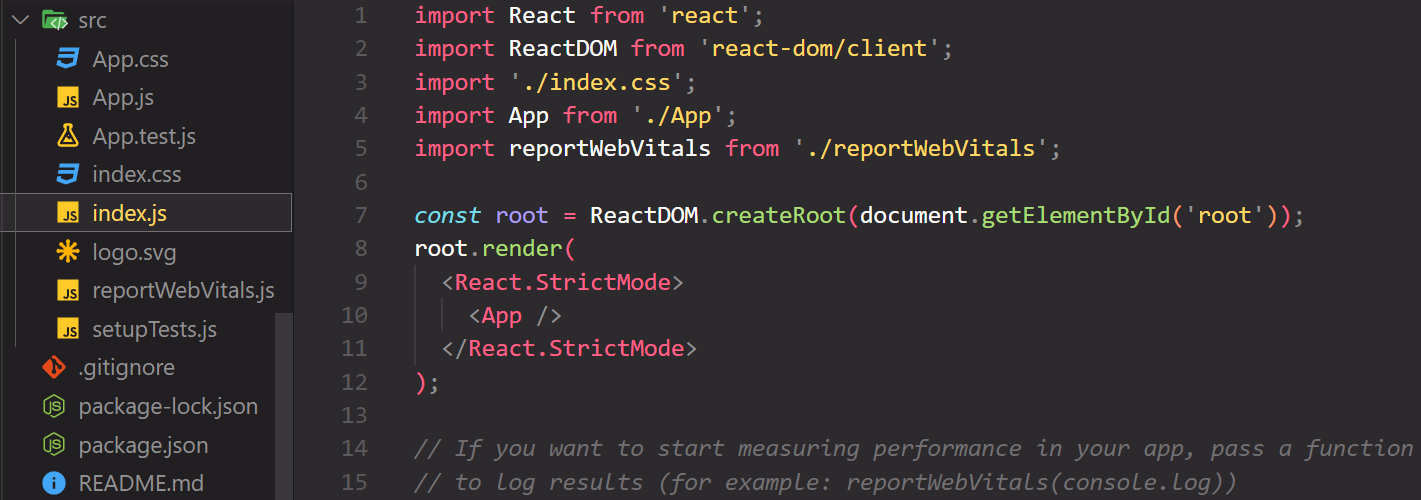


Creado el repositorio, se puede ingresar a la carpeta creada desde la consola para tener disponibles los siguientes comandos:

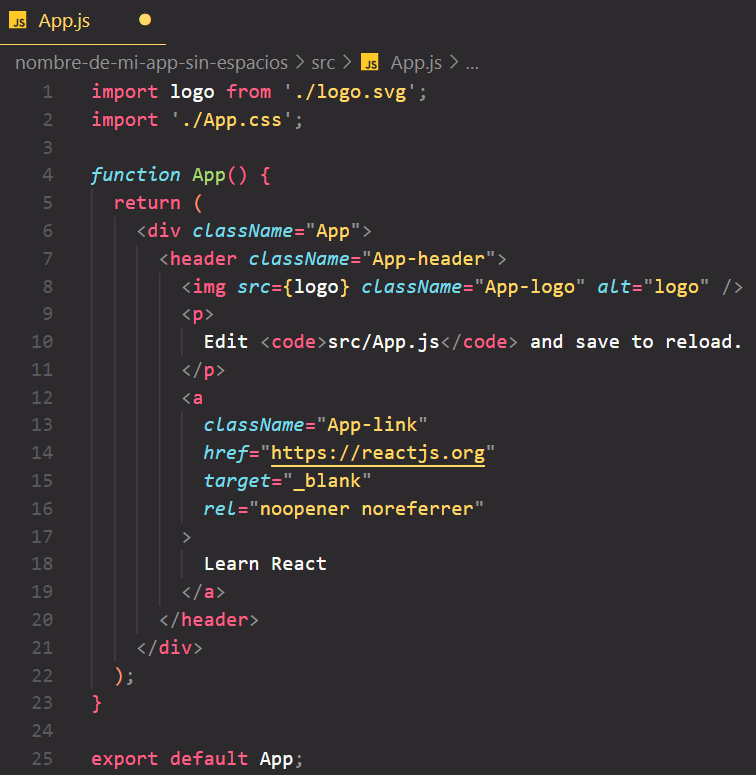
* npm start Inicia el servidor local con el proyecto (Ctrl + C para cerrarlo)
* npm run build Compila el proyecto en la carpeta “build” (para publicarse)
* npm test Inicia el modo test
* npm run ejects Compila el proyecto y elimina este directorio (no hay vuelta atrás)

Además, la carpeta creada con el repositorio contendrá varias carpetas con distintos archivos:

El archivo **index.js** es donde comenzará la ejecución del código. Por defecto, React importa dentro de **index.js** el módulo **App.js**, y es allí dónde comenzaremos a escribir nuestro código.

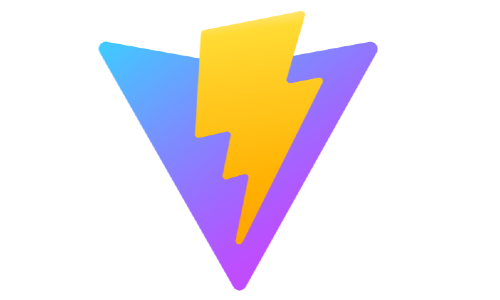


Por defecto, React crea el módulo **App.js** de la siguiente forma:



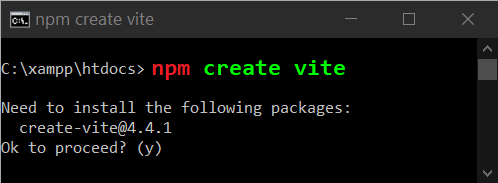
Indica que se exportará todo el módulo

Es una buena práctica que los módulos a utilizar de React tengan extensión JSX.

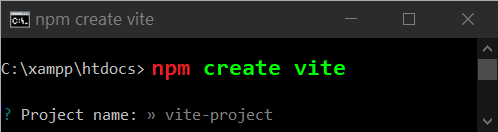
ViteJS

Permite crear proyectos de una forma más eficiente que el modo por defecto que brinda react-create-app.

Teniendo **NodeJS** ya instalado, se debe instalar desde la consola, ubicados en el directorio donde se creará el proyecto.

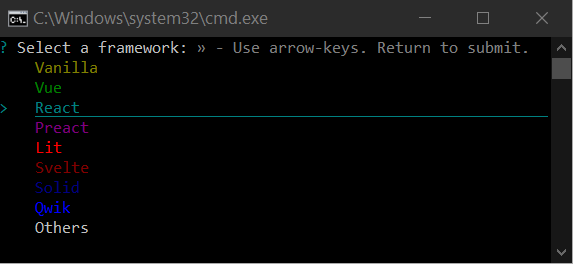
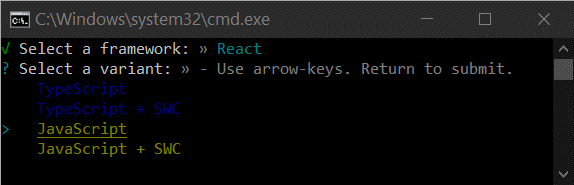


Al intentar crear un nuevo proyecto, la consola misma te dirá que debes instalar el package necesario, para lo cual accedes y se procede a la instalación.



Finalizada la instalación, se pedirá que especifiques el nombre del proyecto, el cual no debe contener ningún espacio.

Luego de esto, se preguntará cuál es el framework con el que se desea trabajar, en este caso será react, seguido de si se quiere usar JavaScript o TypeScript.

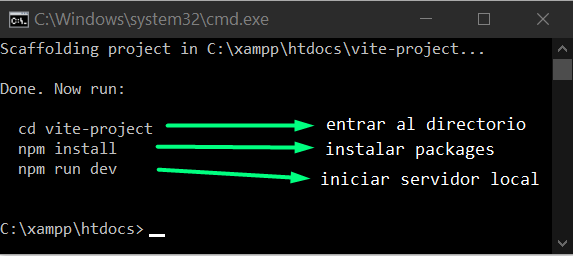


Finalmente se creará el directorio del proyecto.



aquí se ubicarán los archivos de imagen, etc.

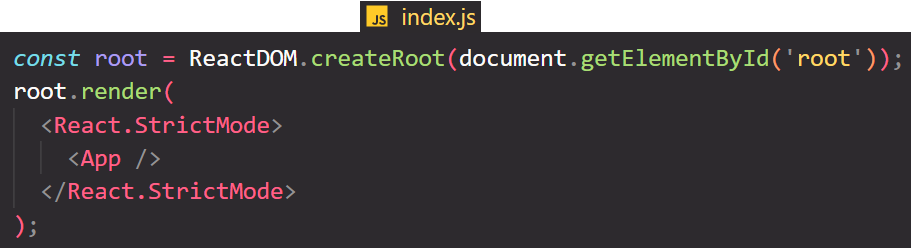
equivalente a **index.js**



En consola se describen los siguientes pasos para finalizar la creación del proyecto.

**Componente**

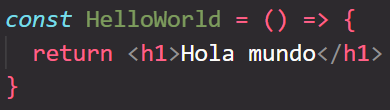
React intenta crear una página web a partir de partes de interfaz. Un componente es una porción de interfaz que puede replicarse, cada una con sus propiedades (**props**) y estados (**state**), con su respectivo código HTML, CSS y JavaScript.

Componente root

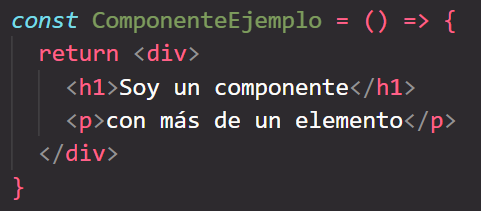
El componente root es similar al **<body>** de **HTML**, es decir, es el componente sobre el cual se crean todos los componentes que compondrán la página.

Creación de un componente

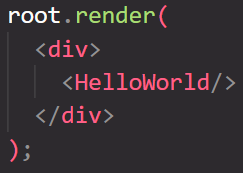
Los componentes siempre son una función que retornan porciones de interfaz, y dicha función siempre está escrita **con la primera letra en mayúscula**.



Estos componentes no necesariamente deben tener un solo htmlElement

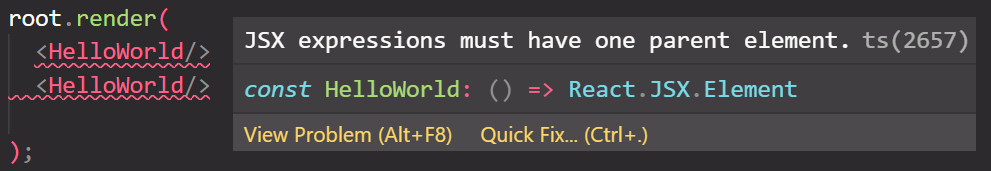
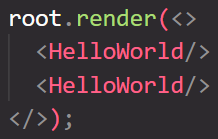


Renderización de componentes

Para renderizar el componente creado, tantas veces como se desee, se utiliza el método render del componente root creado.

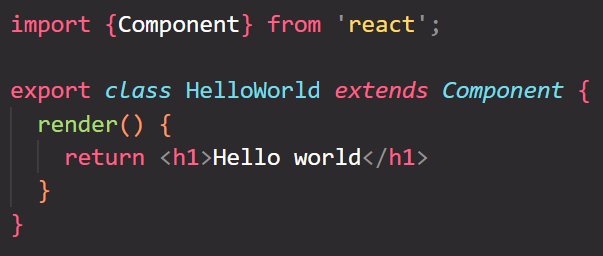
Self Closing Tag

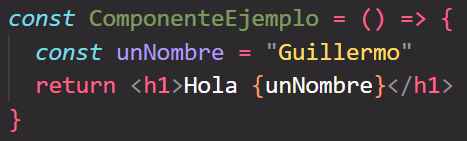
Hay que tener en cuenta que los componentes deben estar contenidos en un htmlContainer (como un div u otro). De lo contrario, no puede renderizarse.



Para evitar esto, podemos utilizar una etiqueta especial de JSX llamada Fragment que actúe como contenedor, la cual debe expresarse como **<> … </>**.

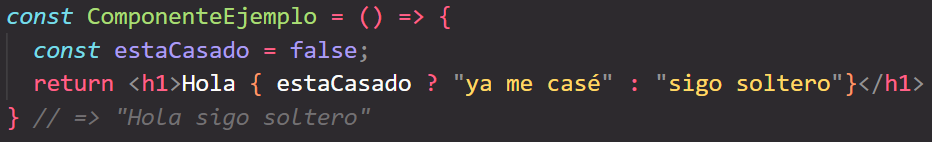
Clases de componentes

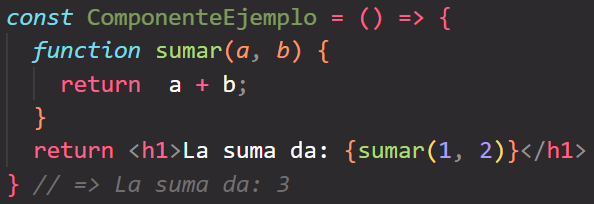
Otra forma de crear un componente es con una class, aunque no es la forma recomendada de hacerlo.

Interpretación de código en componentes

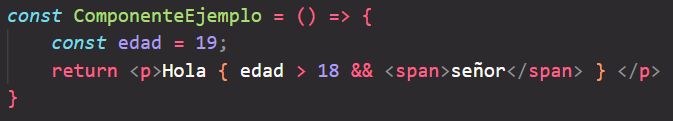
Para añadir variables / constantes a un componente se utilizan las llaves, las cuales permiten la interpretación de código.

También permite la implementación de condicionales con operadores ternarios



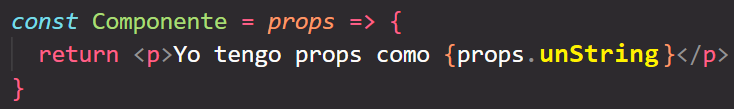
y el uso de funciones

Si sólo se desea mostrar en caso de que la condición sea verdadera (o falsa):

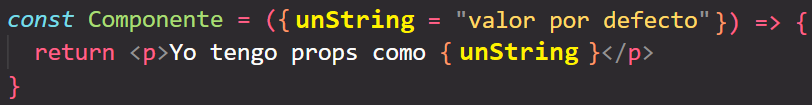


**Props**

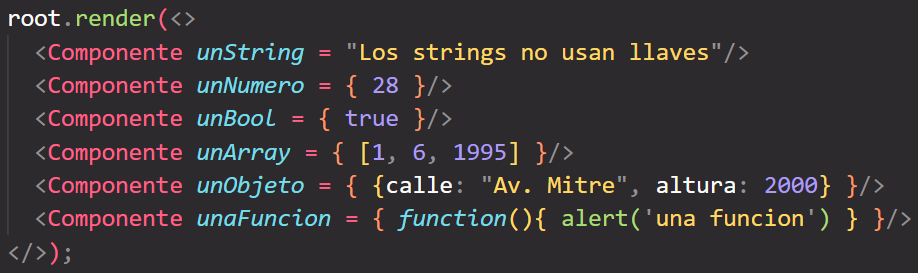
Las **props** son las propiedades (o parámetros) que recibe un componente, los cuales se pasan como un objeto por parámetro a la función que actúa como creador del componente.



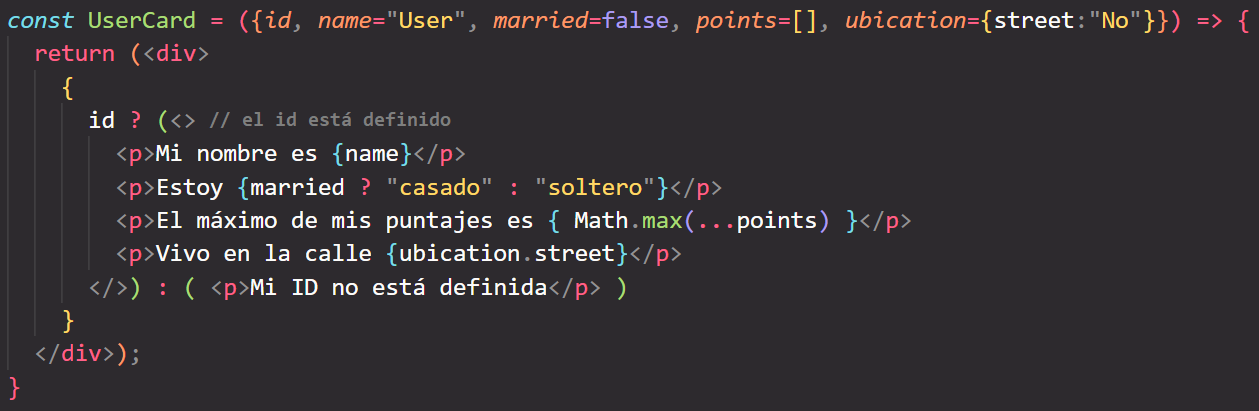
O bien, se puede desestructurar para manejar directamente las props esperables.

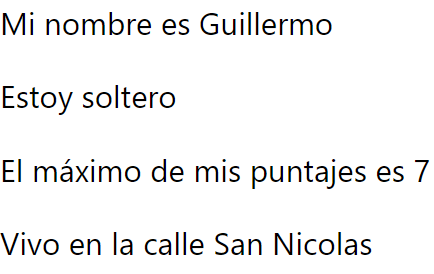


Para que un componente reciba dichas **props**, al momento de renderizar se definen como si fueran atributos de HTML.



Así se pueden crear componentes más complejos tales como





Prop types

Existe una librería de NPM que permite establecer qué tipos de datos puede recibir un componente (como ayuda al desarrollador).

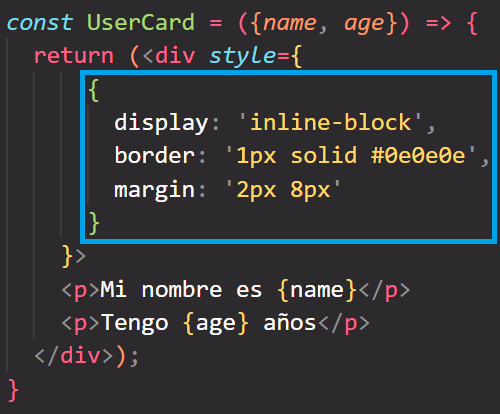
1:29 FAZT

Children

Es una prop especial que permite acceder a los elementos hijos de un componente.

**Estilos**

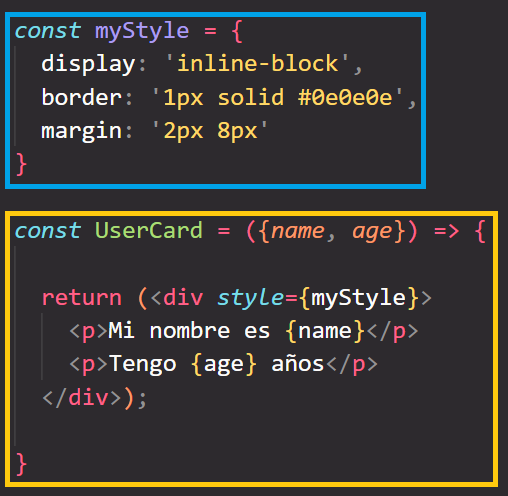
Hay varias formas de añadir estilos a componentes.

Estilos en línea

Como en HTML, se escribe directamente en la etiqueta.

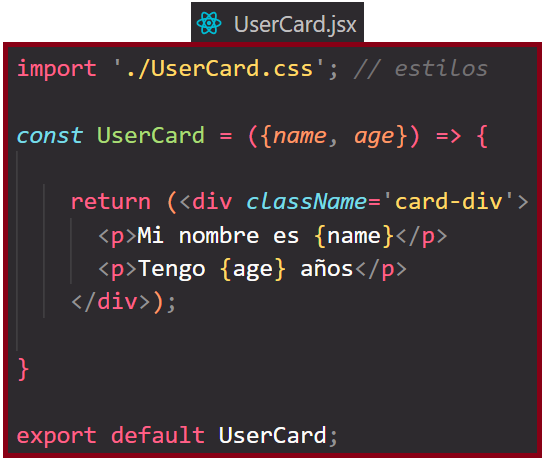
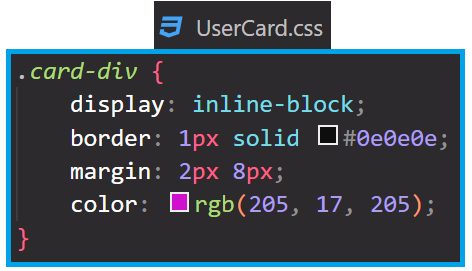
Para que JSX lo reconozca como un estilo en línea, debe escribirse entre llaves y con formato de objeto.

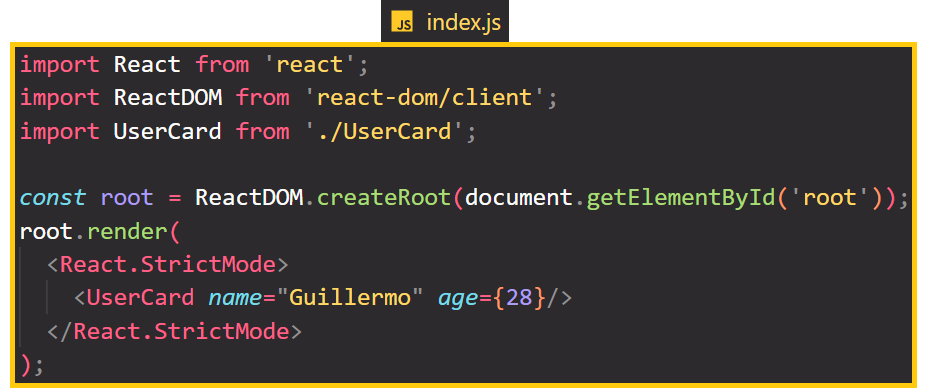
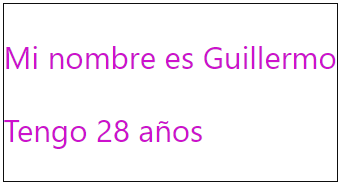
Estos estilos están siendo trabajados como código Javascript (un objeto json), por lo que puede guardarse en una variable separada para que no ocupe el mismo espacio en la etiqueta.



Estilo de un archivo externo

Se utiliza un archivo **CSS** separado del **JSX** donde se ubica nuestro componente.

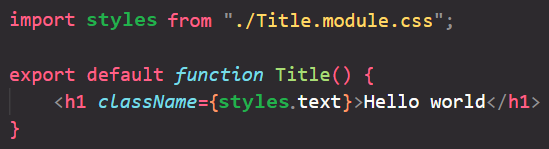




El atributo **class** de **HTML** se llama **className** en **JSX** ya que ”class” en **JavaScript** está siendo usado para otra función (creación de clases / objetos).

Modularización

Se trata de crear archivos CSS con extensión module para poder importar los estilos como un objeto.

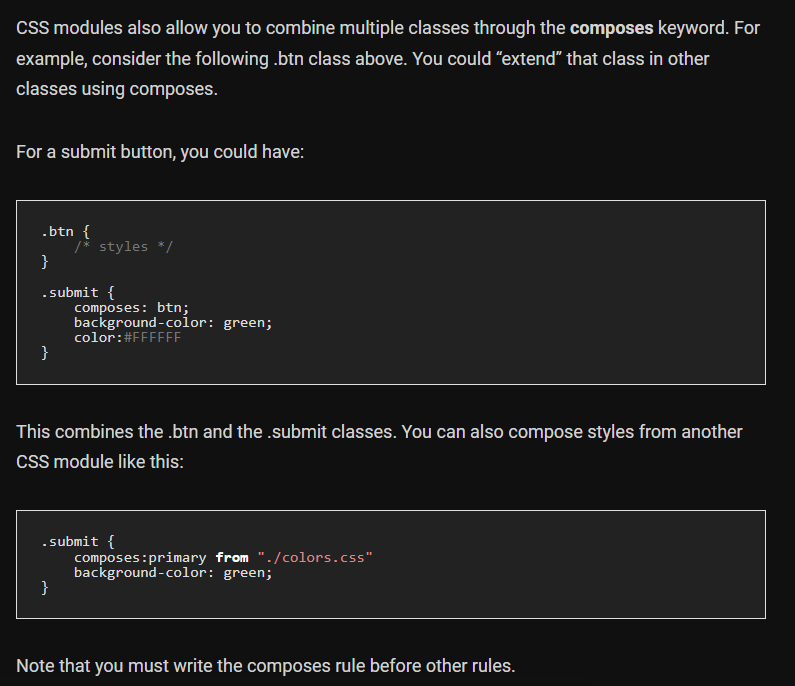


Se pueden concatenar className utilizando template strings



Composición

Utilizando el sistema de módulos, se puede hacer una composición de estilos mediante la palabra reservada **composes**.



**Íconos**

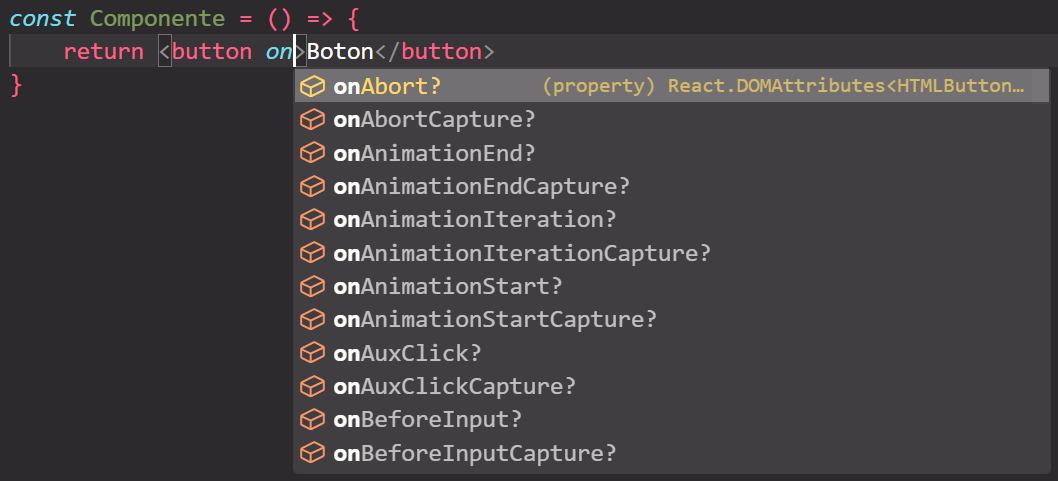
Existen librerías con Íconos especialmente creados para React.

2:17

**Event handler**

Son los eventos relacionados a un componente, como es addEventListener en **JS**.

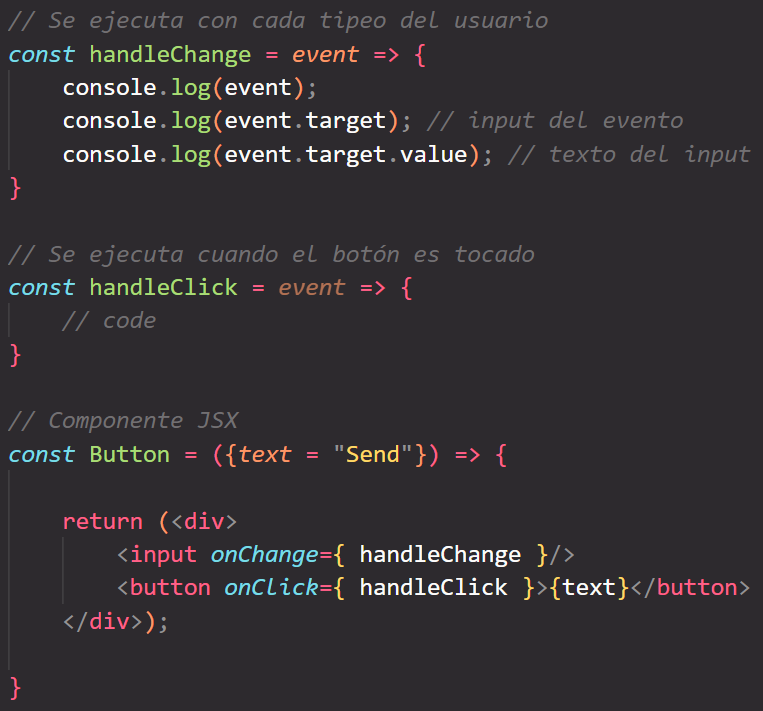
Existen diversos eventos que pueden aplicarse, todos tienen prefijo **on** y van variando según el htmlElement al que se le quiera aplicar:

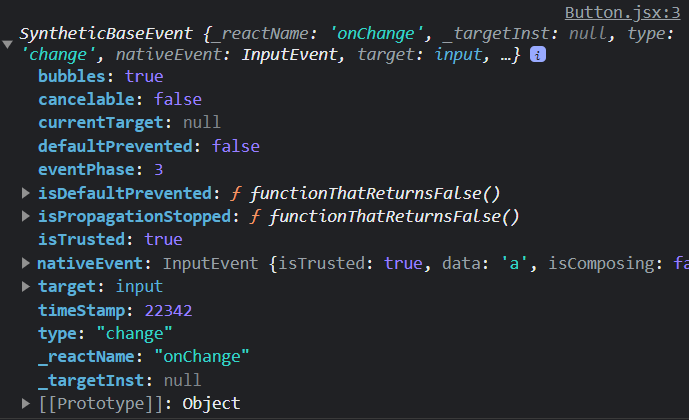


Lista de eventos disponibles

*nota*: no es lo mismo que eventos en línea de HTML, los cuales son una mala práctica

Cada evento tiene un parámetro por defecto, el cual devuelve el mismo evento con información del htmlElement que lo disparó.



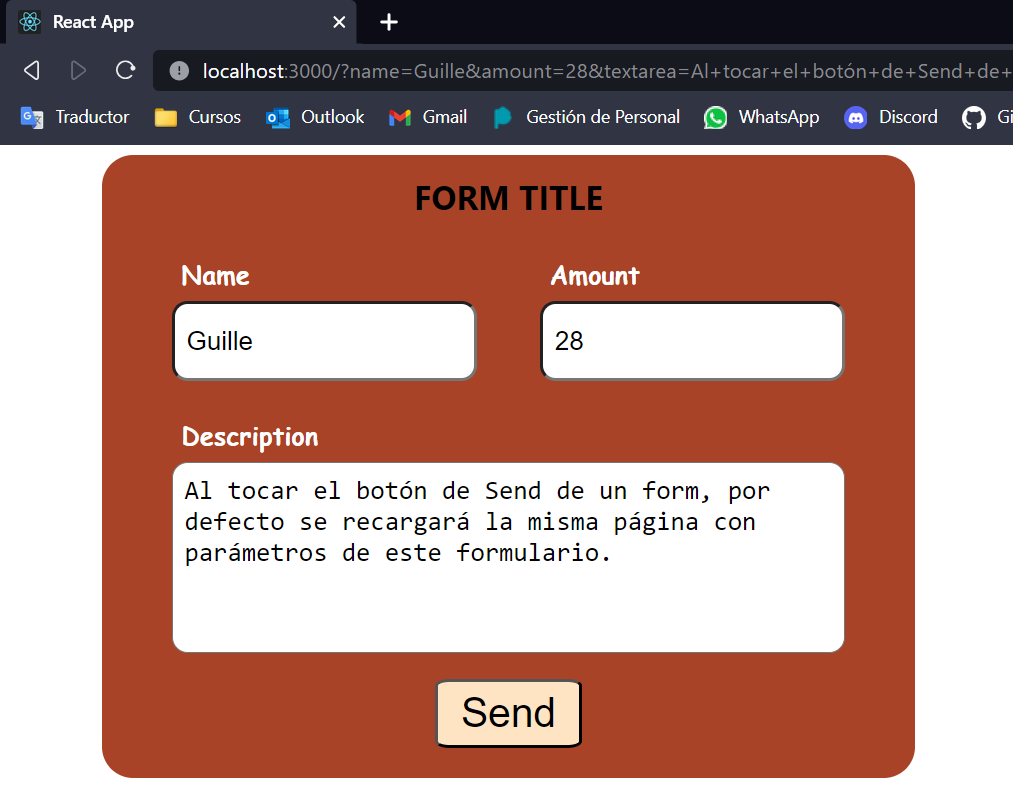


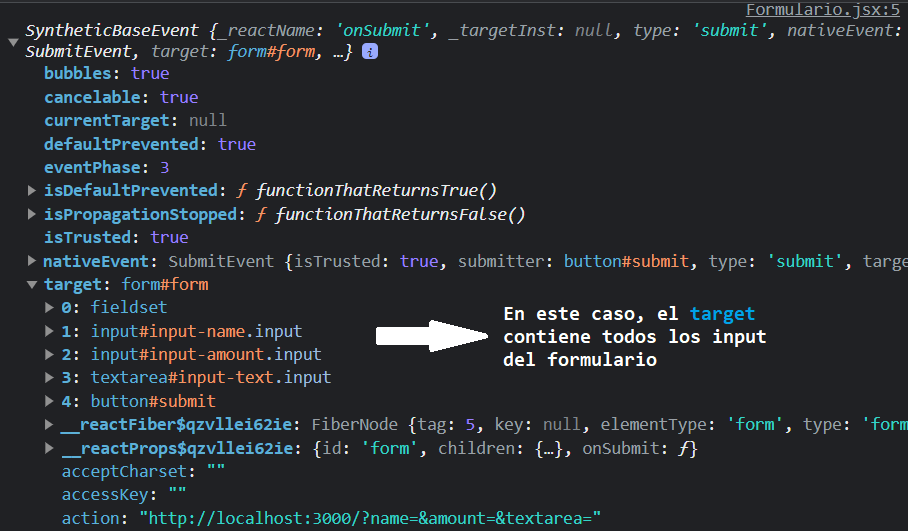


Si se quieren pasar parámetros adicionales a la función, se utiliza una función flecha.

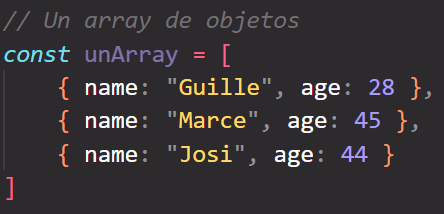
Formularios

Al igual que en HTML, los formularios están diseñados para enviar datos al servidor / backend, por lo que por defecto se intentará hacer esta acción y, al no encontrar, se recargará la misma página enviando por parámetros los contenidos de los input del form.



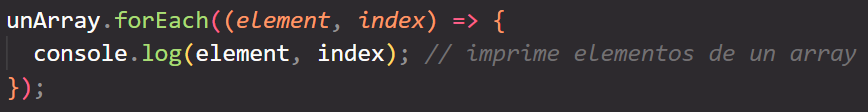
Este comportamiento por defecto se puede quitar haciendo un manejo del evento.



**Arrays**

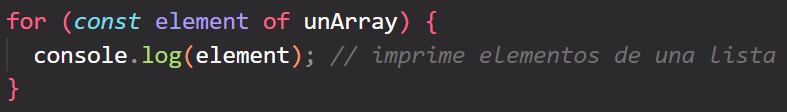
Los arrays son listas de cualquier tipo de datos, desde variables primitivas hasta objetos más complejos.

En **JavaScript** se utilizan bucles para recorrerlos, como los siguientes:



usado para

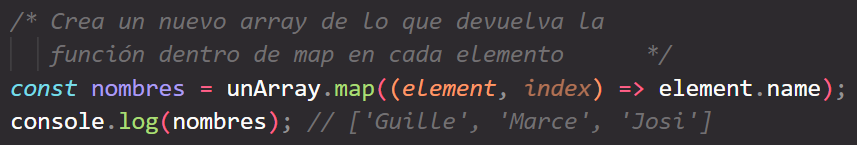




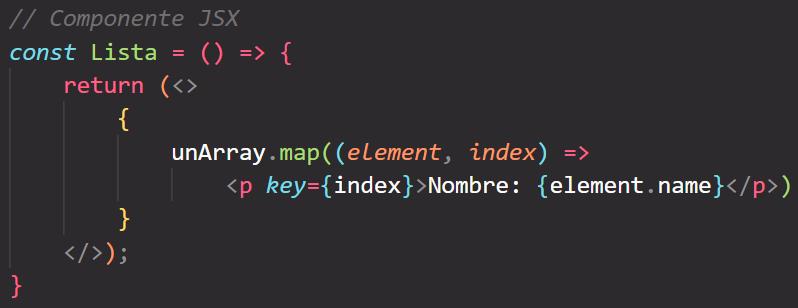
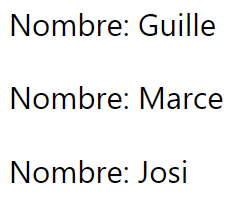
usado para







**React** permite aprovechar los recorridos que ya existen para arrays y utilizarlos dentro de los componentes:

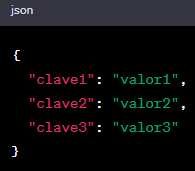


El atributo key no es necesario realmente, pero si no se usa, se generará un warning en la consola. Esto no sucede al compilar la página para producción por lo que puede ser ignorado.

**JSON**

En React se pueden crear archivos JSON e importarlos directamente en el código sin necesidad de acceder a ellos mediante fetch o axios.

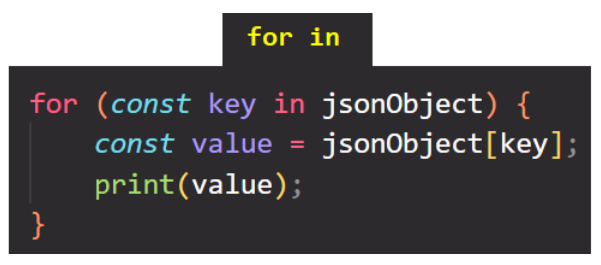


Además, se puede importar sólo una clave específica del JSON en caso de no necesitar el archivo completo.

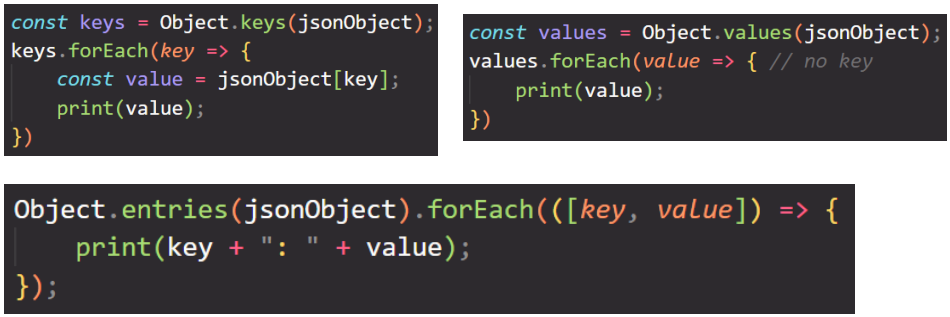


Iteración

A diferencia de un Array, los objetos JSON no pueden recorrerse de cualquier manera, sino que sólo puede hacerse con un **for**-**in**:



O bien, tratándolos como objetos



**Hooks**

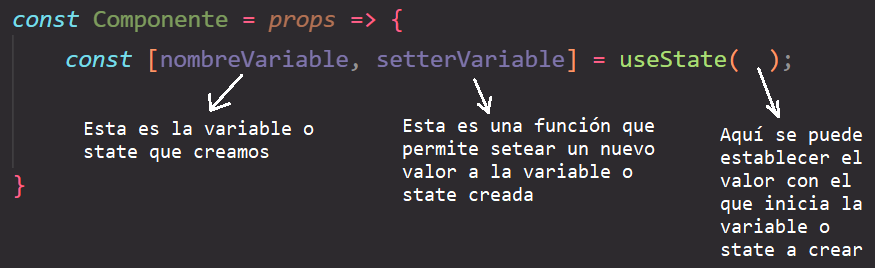
Son funciones que permiten guardar datos, cambiarlos y usar **states**.



* useState

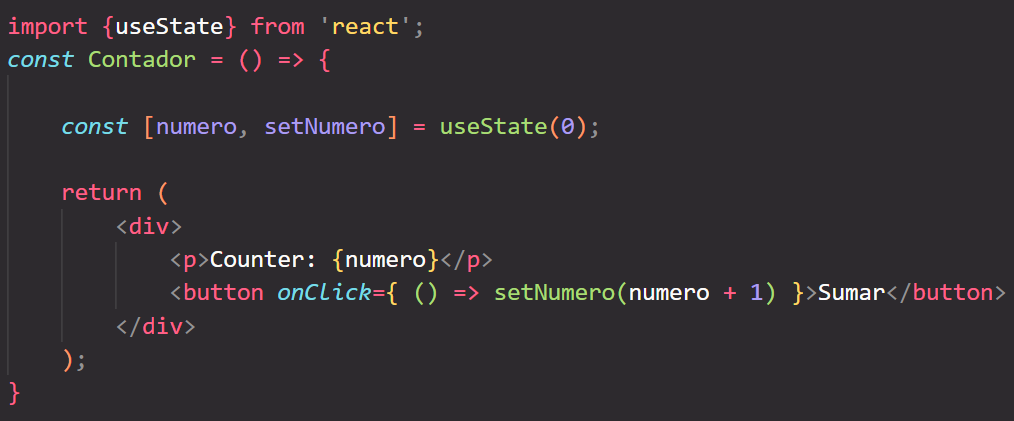
Se utiliza para guardar variables / estados de un componente.

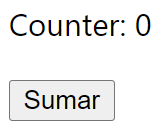
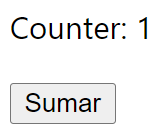
Para comenzar a usarlo, es necesario importarlo y luego crear el objeto dentro del componente.



De esta manera, ya se pueden usar variables / estados dentro de un componente.

Por ejemplo:





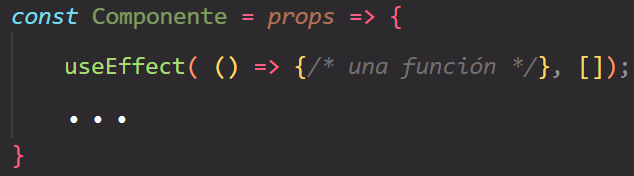
Luego del click

* useEffect

Se utiliza cuando se realizan cambios en un componente de forma asíncrona. Puede ser de mucha utilidad para solicitudes de red.

Existen varias formas de usarse, dependiendo los parámetros que le pasemos a la función dentro del componente.

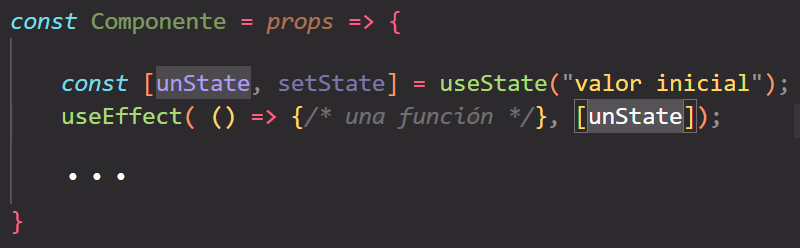
* Se ejecuta sólo cuando el componente es creado



* Se ejecuta siempre que haya un cambio en el componente



* Se ejecuta siempre que haya un cambio en un **state** específico del componente



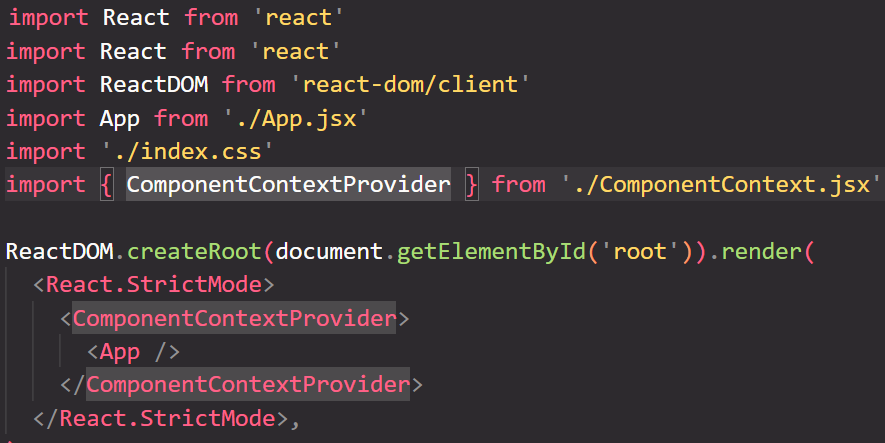
* useContext

Permite la creación de un componente que englobe al resto y que, por lo tanto, contenga la información de todos los componentes hijos (*children*).

Para hacerlo, es necesario crear dicho componente global de la siguiente forma:



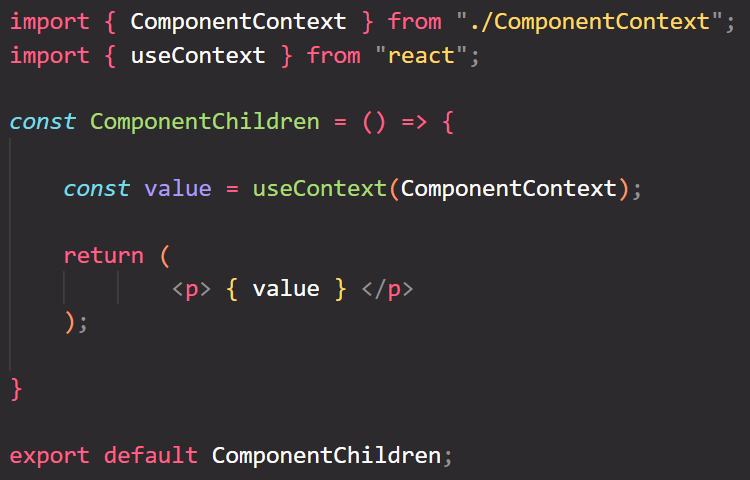
Luego, se debe incorporar el componente global en el renderizado de nuestro **root component**.



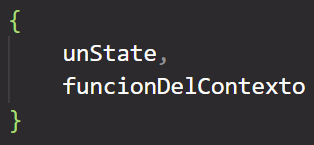
Para poder asignar valores al componente global (**context**) creado, es necesario crear dicha funcionalidad de la siguiente manera:



Así, el componente global contiene una variable a la cual todos sus componentes hijos pueden acceder:

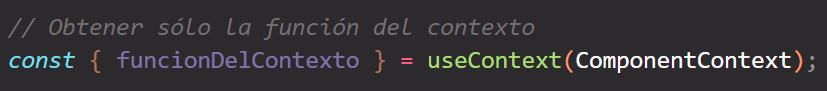


Dentro del value de Provider del componente global (**context**) se pueden enviar desde pequeñas variables (como un **state**) hasta funciones, y si se desean enviar más de una, se las puede englobar en un **objeto**.



En value hay un **objeto**

Una forma de resumir el objeto con clave-valor de mismo nombre es escribirlo directamente así:

Para obtener sólo uno de los elementos pasados en el value del contexto desde un componente children, se utilizan las llaves con el nombre del elemento.

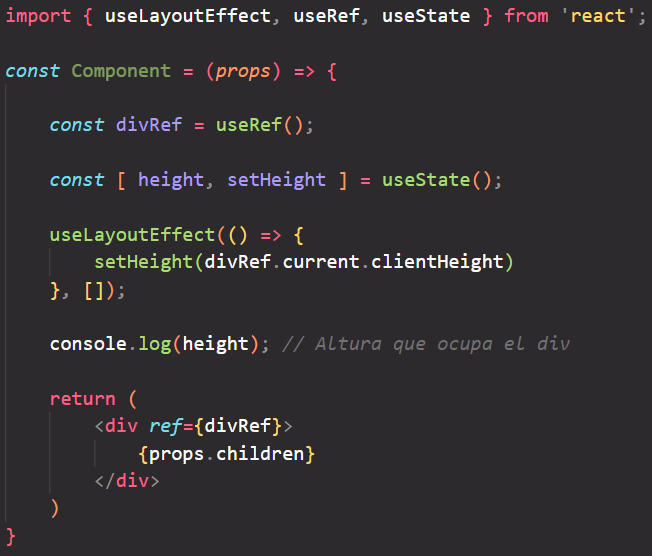
* useRef

Su uso es similar al de un **querySelector**. Permite obtener un elemento específico del Componente.

* useLayoutEffect

Funciona exactamente igual que useEffect, excepto que este hook se ejecuta de forma síncrona.

Puede ser útil para medir elementos del DOM.



* useMemo
* useCallback
* useReducer

Otros Hook

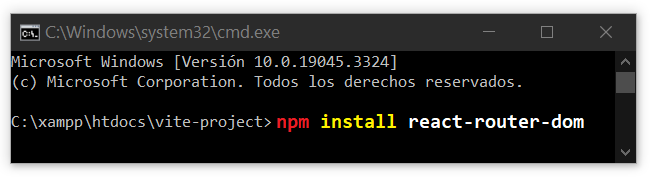
* useImperativeHandle
* useDebugValue
* useTransition
* useId
* useSyncExternalStore
* useInsertionEffect

**React Router**

Permite la navegación entre páginas en React con un formato de SPA (una sola página que actualiza su contenido simulando el paso de una página a otra pero manteniendo las funciones de navegación normales como las flechas adelante y atrás).

Instalación

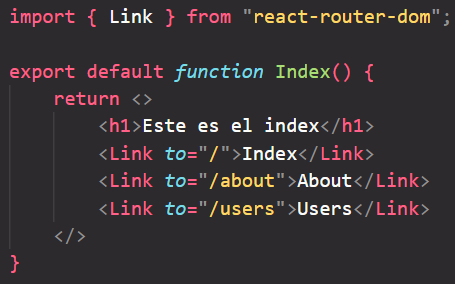
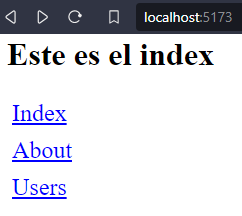
Para usarse, se requiere instalar la librería **react-router-dom** de NPM.



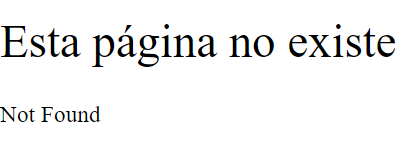
Routes

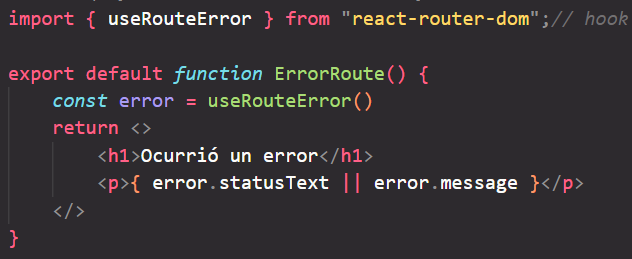
Se crea un directorio (**routes**) que contenga todos los componentes que se renderizarán en cada ruta de la página web.

* Index (*layout*): componente principal que se renderizará al iniciar la página web y, mediante **Link** se pueden acceder a cada ruta.



* Ruta de error: componente que se renderizará cuando ocurra un error.





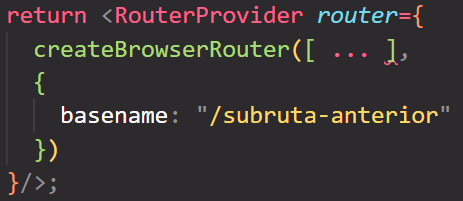
Configuración de las rutas con createBrowserRouter

En el archivo **App.js** se configuran todas las rutas del proyecto.

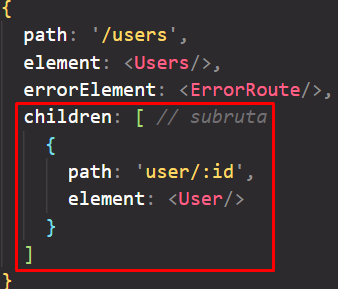


* **Basename**

Si, por algún motivo, se requiere que todas las rutas tengan una subruta previa desde la raíz, se puede usar **basename** como parte de las opciones disponibles.



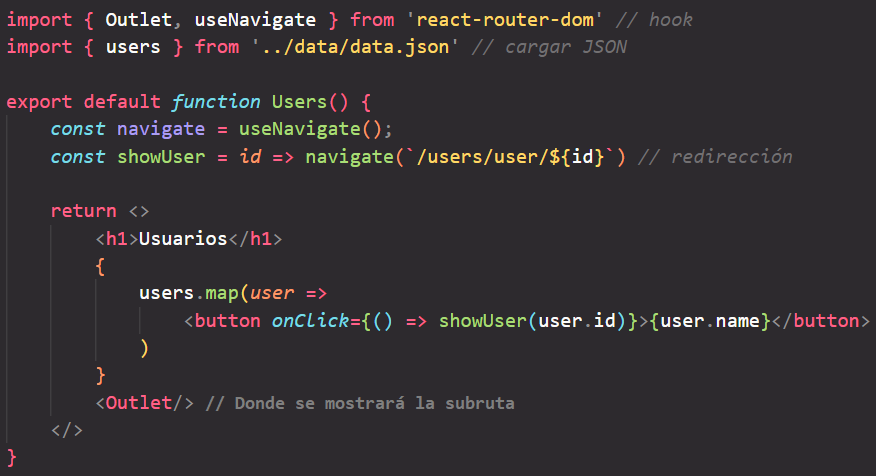
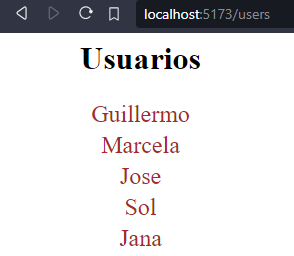
* **Children Routes**

Se pueden establecer sub rutas dentro de una página, con la posibilidad de crear una ruta dinámica, es decir, que podrá cargar una cosa u otra según un parámetro (definido luego de dos puntos).



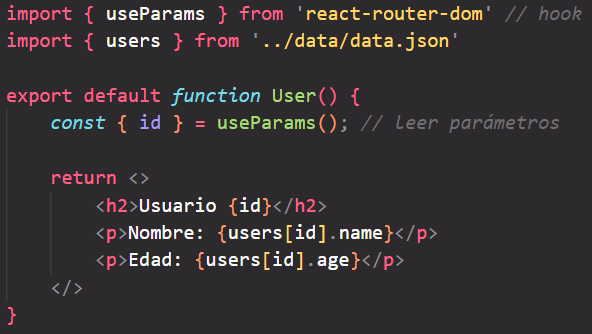
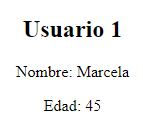
Un ejemplo de uso es el de mostrar elementos que fueron cargados de una fuente externa, como un JSON.



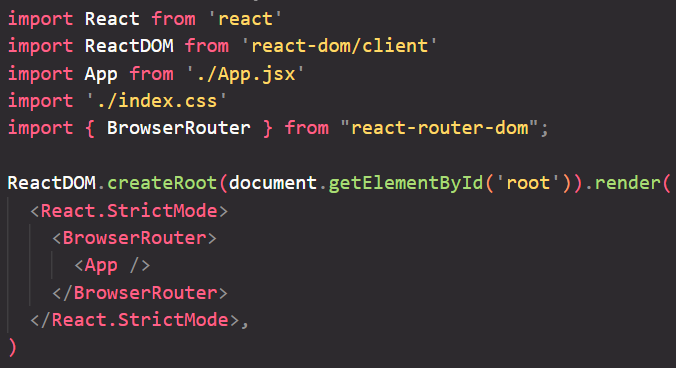


en donde está la etiqueta **Outlet** es donde se cargará el contenido de la subruta

Permite acceder a la subruta sin usar Link



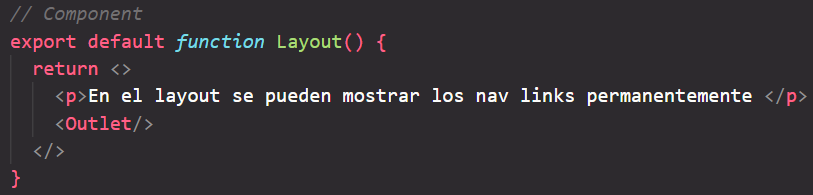
Configuración de las rutas con la etiqueta Routes

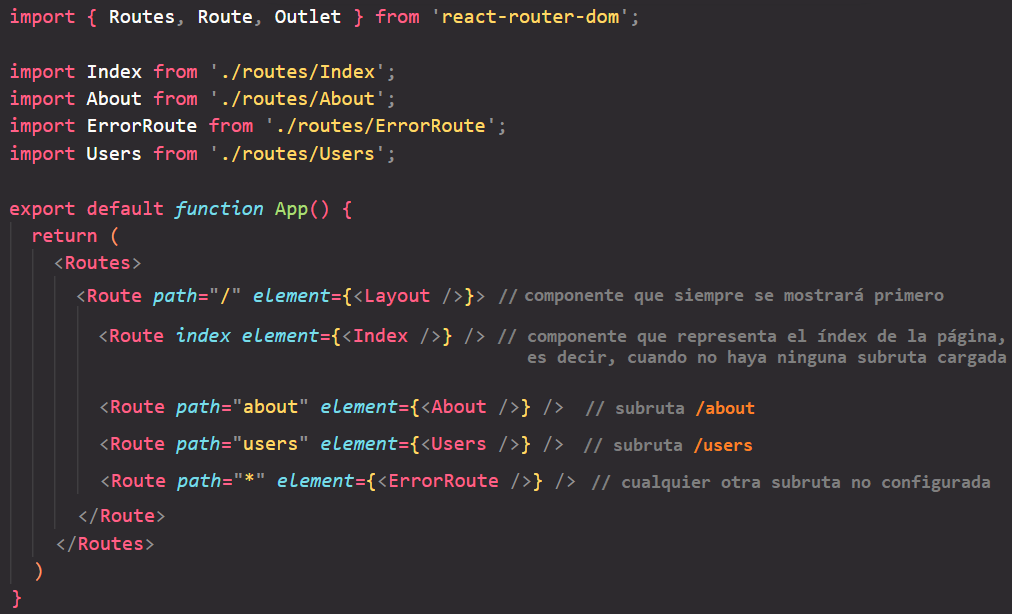


En el archivo **index.jsx** (donde se renderiza el root de la página web) se importa el componente **BrowserRouter**.



Se crea el archivo **Layout.jsx** que representará el header / nav de la página web. Debe contener el elemento **Outlet** que es donde se cargará el resto de la página.



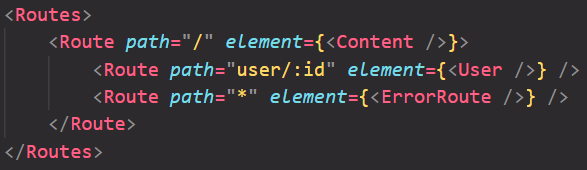
En el archivo **app.jsx** se importan las primeras subrutas del index, es decir, las que corresponden a la ruta **/** y sus primeras subrutas.

* **Children Routes**

Se pueden establecer subrutas directamente dentro del componente que representa una ruta específica.



De esta manera, el contenido de la subruta se mostrará como una página completamente nueva.

Si desea que la subruta sea un componente completamente que se renderiza justo debajo del contenido principal, se cambia la estructura de **Routes** de la siguiente forma:

**React Redux**

Proporciona un flujo de datos unidireccional facilitando la gestión del estado global, lo cual evita la necesidad de pasar props a través de múltiples componentes.

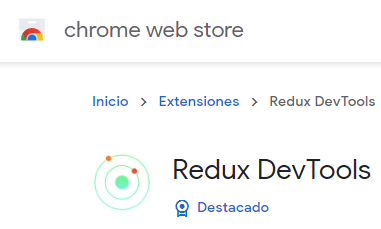
Es similar al useContext, sin embargo, redux permite más funcionalidades.

Instalación

Para usarse, se requiere instalar la librería **react-redux** de NPM.



Es importante instalar la extensión de navegador de Redux para poder visualizar el comportamiento de los stores de redux.



Actions

Objetos literales que están compuestos por dos props.

* Type nombre con el que se identifica la acción.
* payLoad información que se quiere enviar al estado para que se actualice.

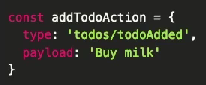
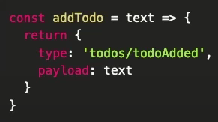
Esto será procesado por el **reducer**

Ejemplo de un contador

* Type: “Incrementar”
* payLoad: +1

Action creators

Funciones que retornan los Actions. Permiten recibir el payLoad de forma dinámica. Muchas veces, se definen los Actions dentro del mismo Action creator.



Reducer

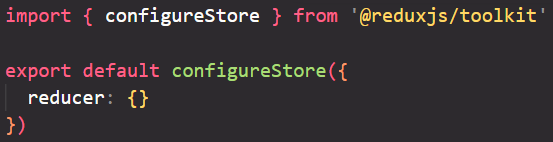
Es una función que toma dos argumentos y se encarga de actualizar el estado.

* Estado previo
* Action

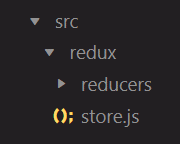
Store

Donde se almacenan los estados globales de la aplicación.

Cuando la página se recarga, se reinician todos los datos del Store.

Configuración inicial

<https://redux-toolkit.js.org/tutorials/quick-start>

Dentro de la carpeta donde se ubicará nuestro redux, se crea un archivo store.js

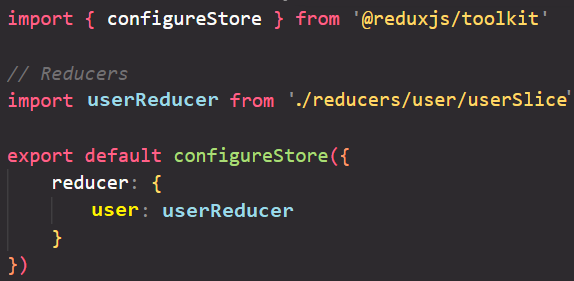


Luego, en el index.js (o main.js) se importa el Store y el Provider de nuestro Redux para poder englobar toda la aplicación.

Finalmente, se pueden comenzar a crear los reducers, y “estados globales”

Action creators

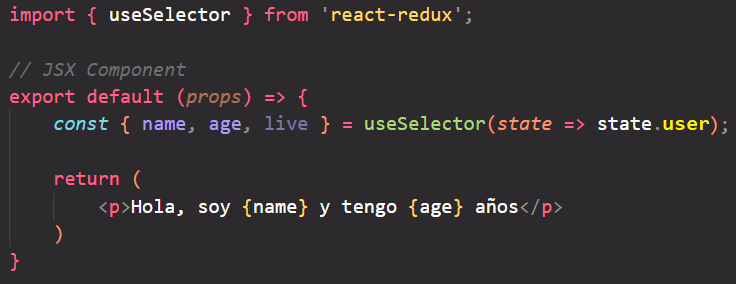
Tras crear un reducer (“estado global”), se lo debe guardar en el Store creado.



nombre del “estado global”

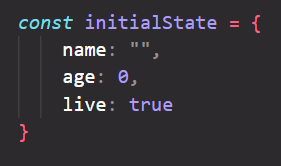
useSelector

Este Hook proporcionado por react-redux permite leer los datos del Store en un componente de React.

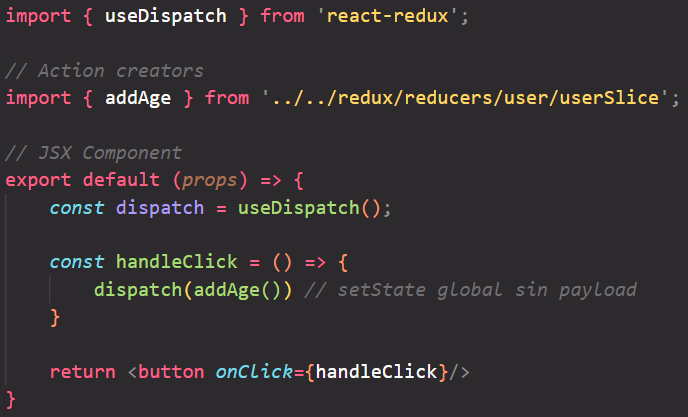


useDispatch

Este Hook proporcionado por react-redux permite actualizar datos del Store desde un componente de React.



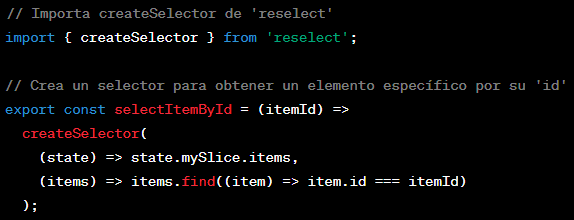
payload

Para ejecutar una función que no requiere parámetros (payload), podemos usar el siguiente ejemplo:

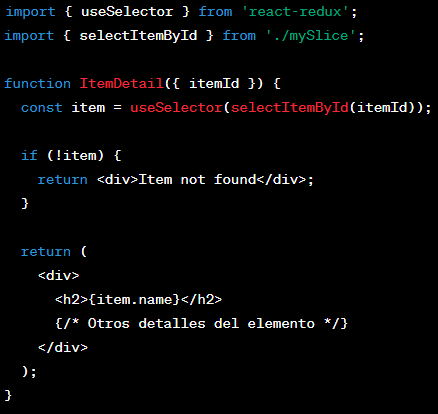
Selector

Cuando se tiene un Slice con un array en el initialState, se pueden crear selectores para obtener sólo un elemento de dicho array.

Para ello, se crea una función que reciba el ID como parámetro y acceda state.

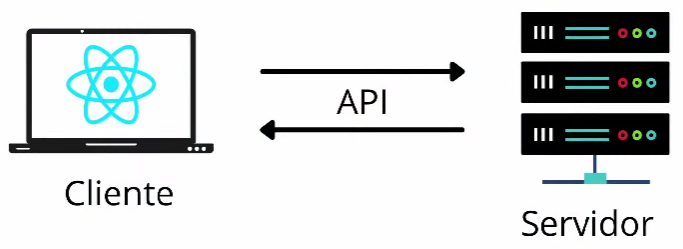


Con la función definida, se la puede trabajar desde el componente deseado.



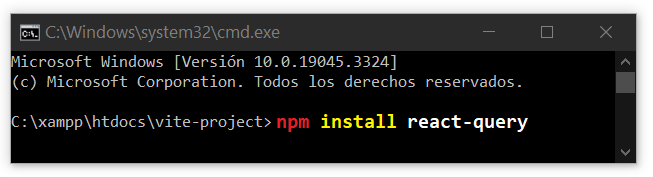
**React Query**

Es una biblioteca que simplifica la gestión de datos obtenidas desde fuentes externas como APIs. Administra una caché automática para agilizar solicitudes repetidas.



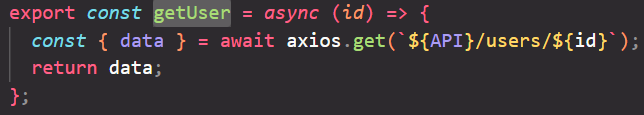
Instalación

Para usarse, se requiere instalar la librería **react-query** de NPM.

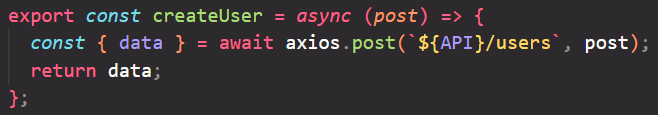


Axios

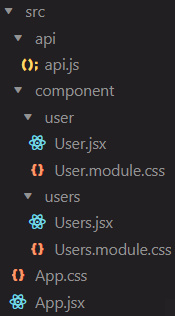
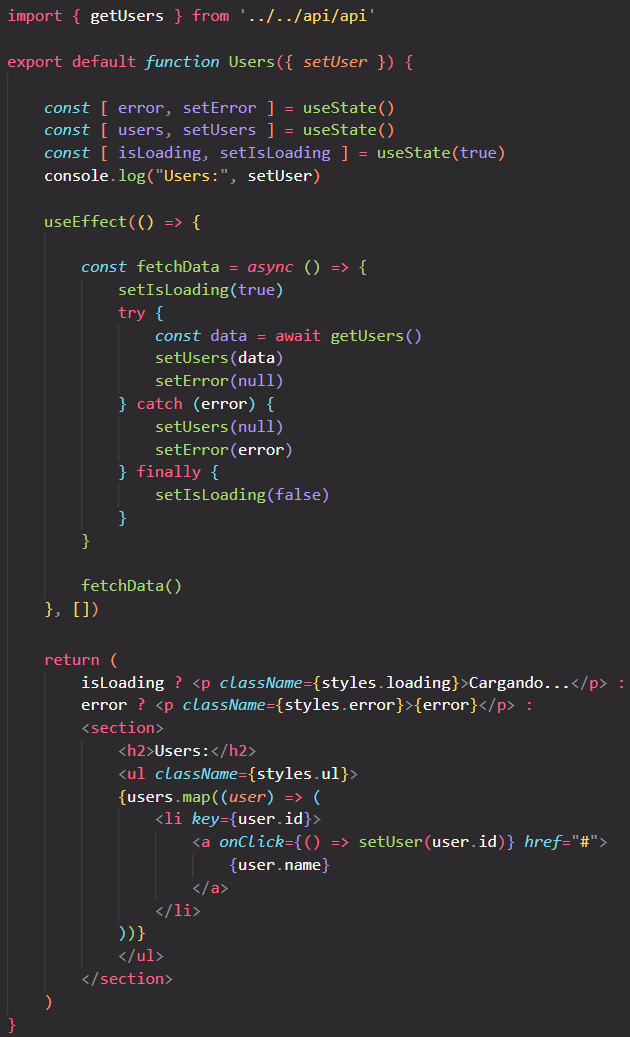
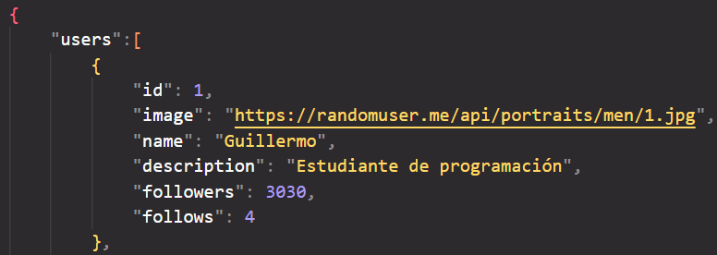
En general, para solicitar datos a una API, se utiliza **fetch** o **axios** para obtener información del objeto en formato JSON.

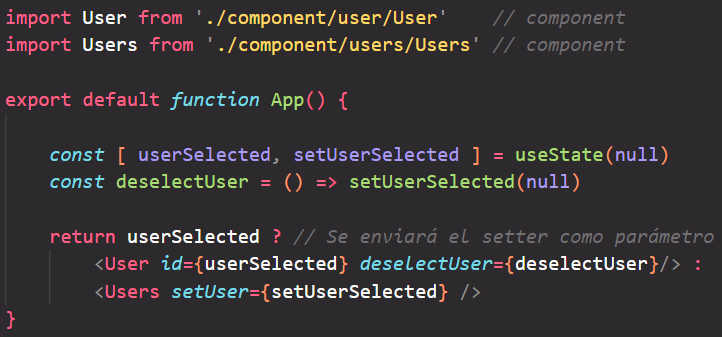


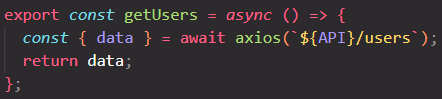
Y para cargar datos nuevos con la API, se usa el método **post**.



Ejemplo de uso sin React Query

Como ejemplo, se usará una “base de datos” simple basada en JSON-Server.





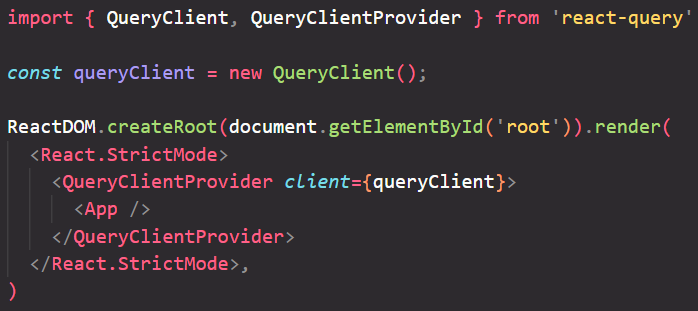
Teniendo la base, se prepara la carga de datos:

se puede modularizar





Configuración de React Query



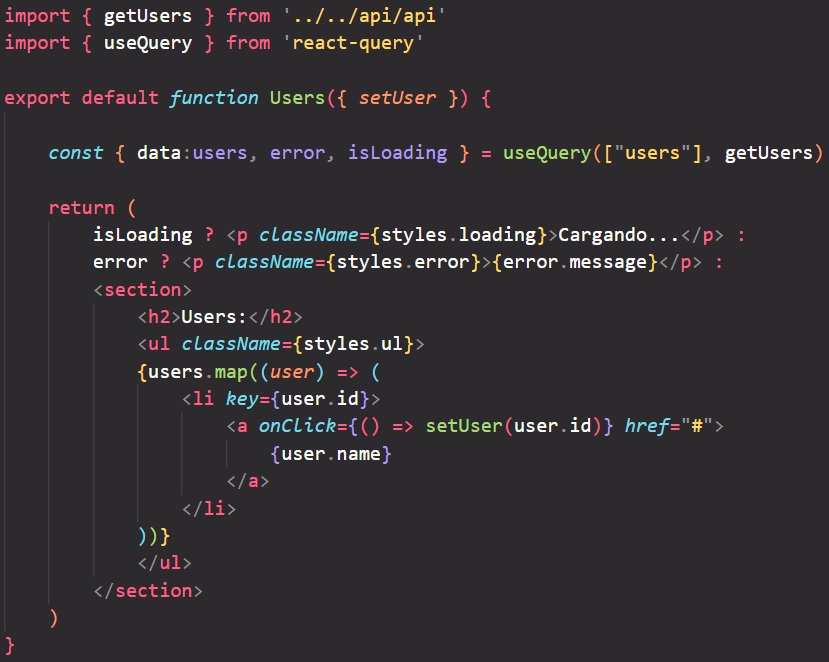
Ahora se puede usar en nuestros componentes por medio de **useQuery** para hacer una petición a la API.



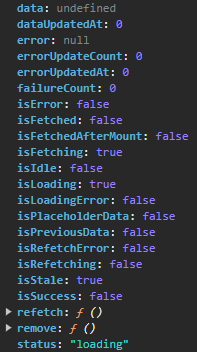
clave que identifica de forma única a la consulta a realizar

promise que devuelva los datos o un error en caso de problemas (axios, fetch, etc)

Así, se simplifica el código utilizado:



Se realizarán 4 intentos antes de lanzar el resultado “error”

Al usar useQuery, éste devuelve un objeto con los siguientes datos:

Refetch

Al usar **useQuery**, los datos obtenidos se guardan en caché y, la próxima vez que se ejecute la misma query (con el mismo identificador ‘*users*’), se mostrarán los datos de la caché.

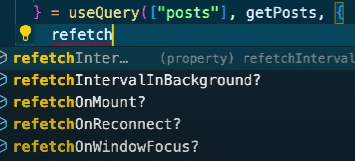
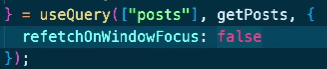
Al mismo tiempo se realizará una revalidación. Es decir, se hará una nueva petición a la API para actualizar los datos mostrados desde caché en tiempo real.

Se sabe si se está realizando esta operación con **isFetching**.



Esto se realiza tanto cuando se sale y vuelve a poner el foco en la página o pestaña web, y cuando se renderiza de nuevo el componente.

Esto se puede configurar con un tercer parámetro en el **useQuery**.



Se pueden desactivar

Además, se puede configurar para que se actualicen los datos cada cierto intervalo de tiempo sin necesidad de que el usuario realice ninguna acción.

ms



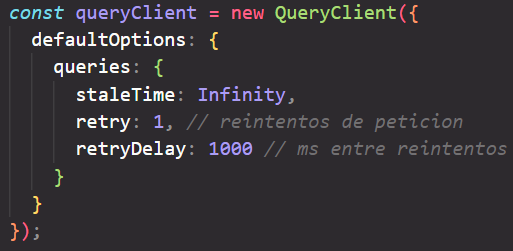
Otra configuración posible es la de establecer el tiempo en que los datos se considerarán como actualizados (0 por defecto).

ms

Con se define que no se actualicen los datos.

La información en caché se borrará por defecto luego de 5 minutos de dejar de mostrarse el componente donde fue cargado.

ms

Configuraciones globales

Las configuraciones de Refetch se pueden realizar directamente en la creación del QueryClient en el index de la app.

Idle

Se pueden deshabilitar las peticiones automáticas que realiza useQuery al cargarse el componente y hacerlo “manualmente”.

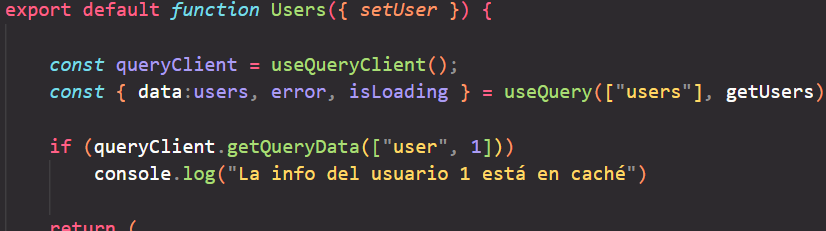


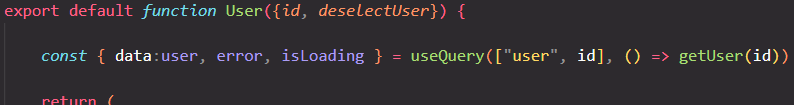


La función “refetch” es la que reanuda las peticiones a la API.

useQueryClient

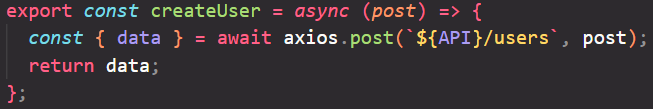
Permite acceder a los datos que se encuentran almacenados en caché.



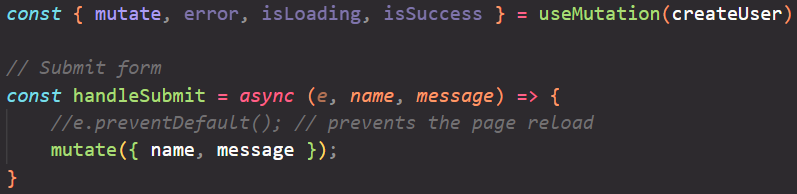


En este caso, al cargar los datos del user 1, dichos datos se almacenarán en caché. Al volver al componente Users, detectará que ya existen datos del user 1 en caché.

useMutation



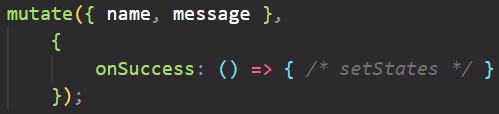
Permite la manipulación de los datos almacenados en caché, útil para realizar un **post** a la base de datos con nuevos datos.



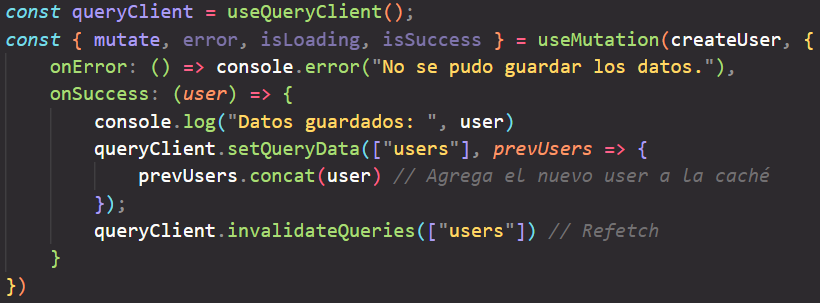
En este caso, el parámetro **post** es el objeto:

Además, es posible ejecutar callbacks en caso de que sucedan ciertos eventos.



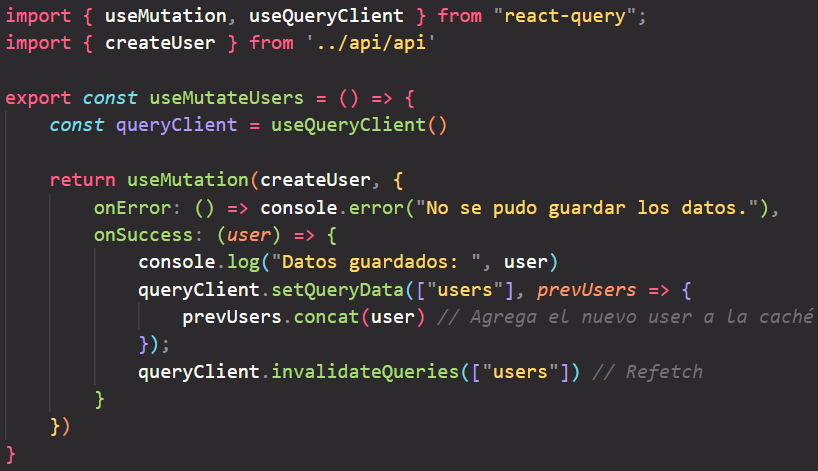
Estos callbacks también pueden definirse en la función **mutate** como segundo parámetro. Es útil para cambiar estados locales.

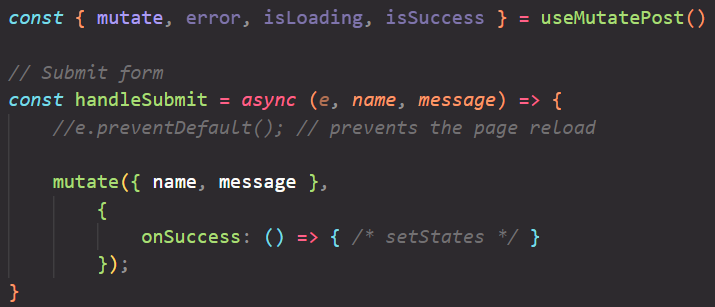
Para resetear todos los estados del useMutation, es posible utilizar la función **reset**.

Se pueden añadir datos en la caché con su id específica para agilizar el proceso de refetch para el usuario.

Hook personalizado

La declaración del useMutation es bastante extensa en el código y puede modularizarse creando un nuevo hook.





**Build**

Todo el código creado en React se hace en un entorno de trabajo, pero este no es el que debe publicarse en un servidor.

Para generar el código que puede publicarse en un servidor se debe ejecutar el comando **npm run build**.

Carpeta PUBLIC

Dentro del proyecto de React existe una carpeta llamada public.

Esta carpeta se “construirá” tal cual como está en el index del proyecto.

Es útil especialmente para guardar archivos de imagen que pueden accederse desde el código y que este pueda variar. Así como archivos de código y otros.

XAMPP

Por lo general, en React se utiliza Express de NodeJS para construir el servidor, pero si se quiere usar XAMPP, se debe publicar el contenido de **dist** (donde se “construye” el proyecto) en la carpeta **htdocs**.

Además, si se usa **react-router**, es necesario configurarlo para que pueda reconocer las rutas alternativas. Se debe crear un archivo en **htdocs** llamado **.htaccess** con el siguiente código de PERL:

RewriteEngine On

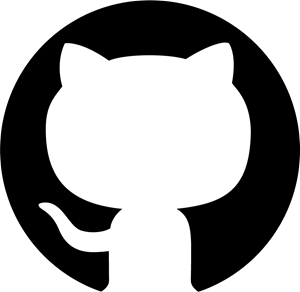
RewriteBase /

RewriteRule ^index\.html$ - [L]

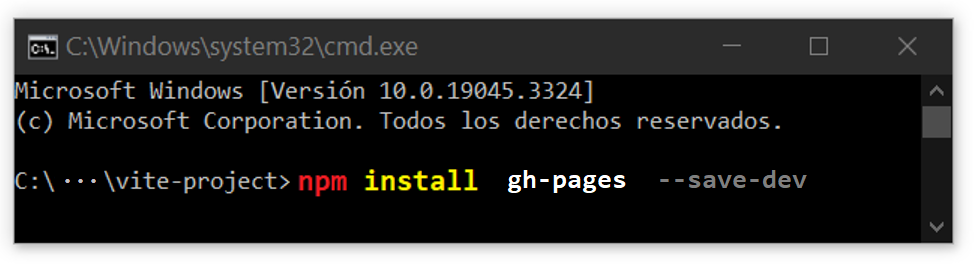
RewriteCond %{REQUEST\_FILENAME} !-f

RewriteCond %{REQUEST\_FILENAME} !-d

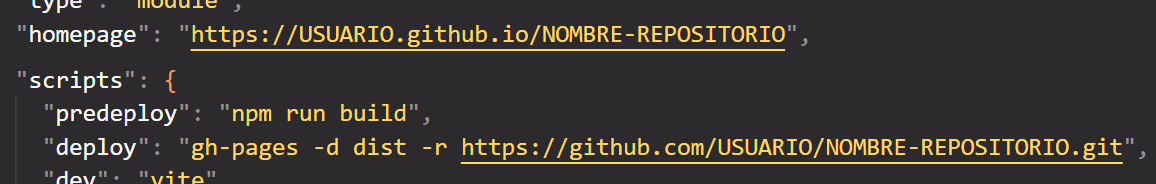
RewriteRule . /index.html [L]

**GitHub Pages**

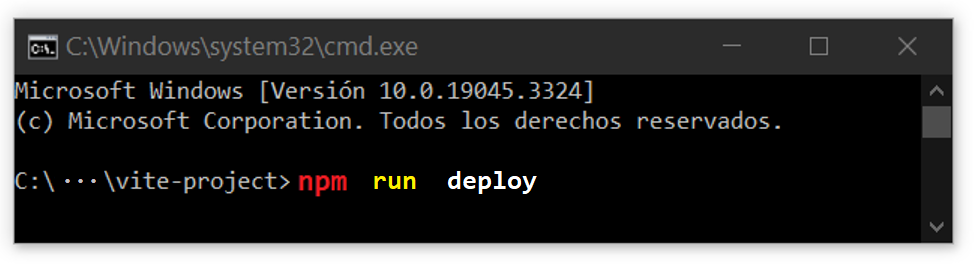
Se puede publicar un proyecto construído con React en GitHub, desde un repositorio. Para esto, es requerido instalar un package.



Luego, será necesario modificar el archivo **package.json** del proyecto, agregando las siguientes líneas:



Finalmente, se procede a ejecutar el comando requerido para publicar la página.



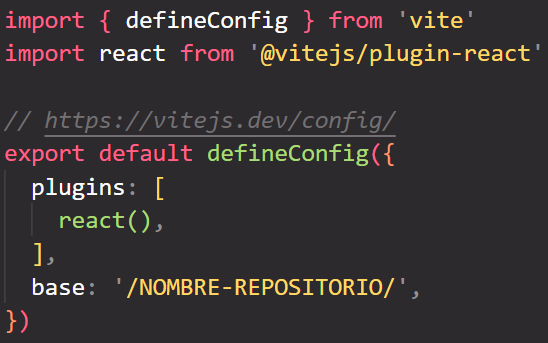
Así ya podrás acceder a la página con la URL

https://USUARIO.github.io/NOMBRE-REPOSITORIO

Si se está usando **react-routes** con varias páginas, puede ser necesario crear un archivo html en el root del proyecto llamado **404.html** que redirija al usuario al **home** de la app. Esto es porque, al recargar la página estando en una subruta, Github Pages no reconocerá la ruta como válida y ejecutará el error 404.

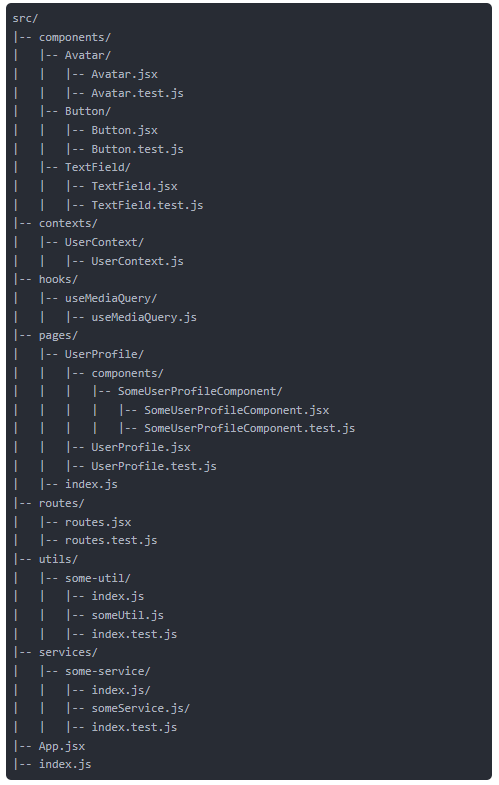


En caso de estar usando Vite, será necesario configurarlo adecuadamente.



**Modularización**

Un proyecto se debe modularizar para un mejor entendimiento de cualquier programador que visualice el código.

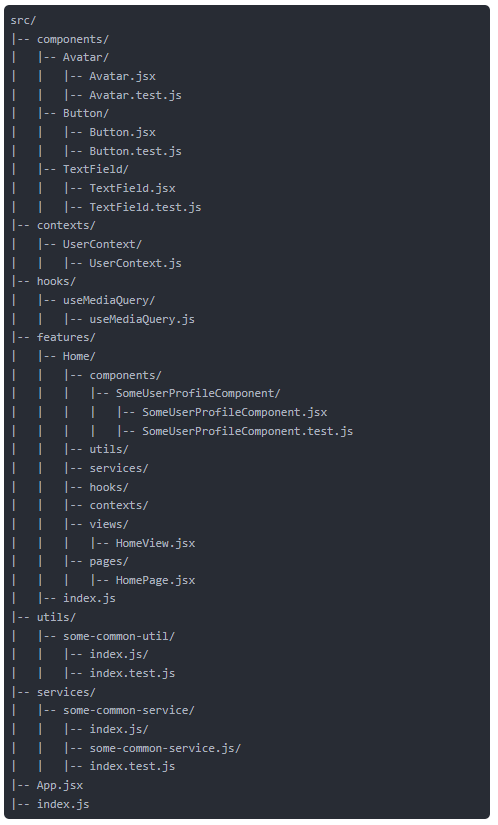


Estructura por tipo de fichero

Diferenciación entre componentes compartidos y páginas. Generalmente utilizada en proyectos pequeños.

Estructura por funcionalidad

La idea principal es que cada módulo que definimos tenga todo el código relacionado con este y solo se importe código del módulo en sí. No se puede compartir código de módulos distintos.



**React Native**

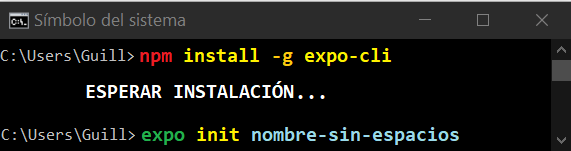
Framewoork que permite la creación de aplicaciones android, ios y desktop utilizando código JavaScript y React, el cual realiza un “bridge” con el código nativo de los respectivos sistemas operativos para desplegar la aplicación.

Para comenzar a programar en React Native es necesario tener **NodeJS**.

Expo

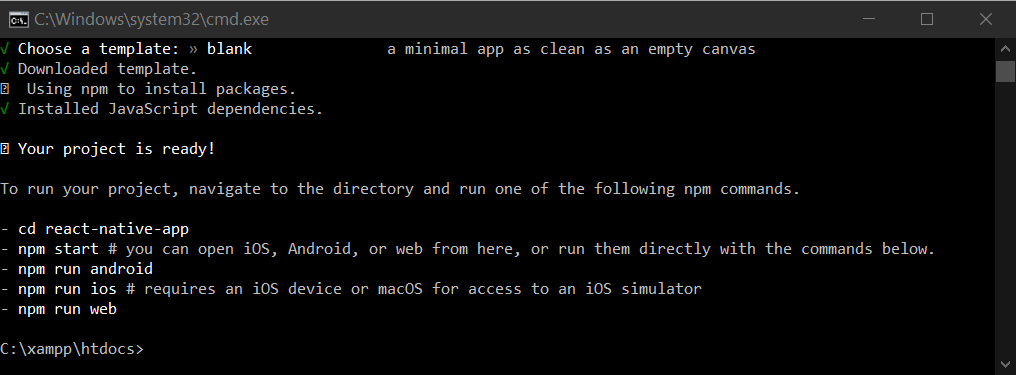
Permite la visualización de un dispositivo virtual para poder ejecutar el programa de React Native de forma sencilla.

Para instalarlo, es necesario abrir la consola y ejecutar los siguientes comandos:



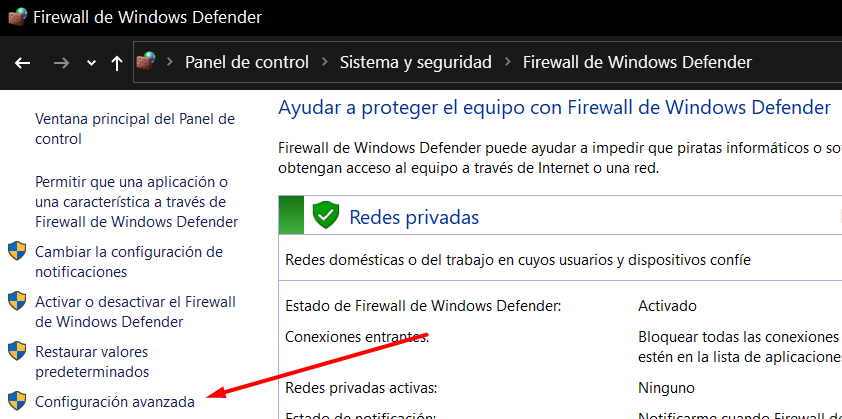
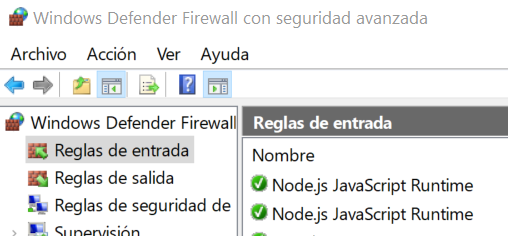
Una vez finalizado el proceso, se preguntará por el tipo de proyecto que se desea crear. En este caso, crearemos un proyecto en blanco para comenzar desde el inicio.

Finalmente se creará el directorio con el nombre del proyecto indicado.



Usando el comando **npm start**, se iniciará el localhost para la app.

Puede ser necesario permitir el acceso en windows defender



Una vez iniciado, descargar la APP de Expo en el dispositivo móvil y escanear el código QR que aparecerá en la consola.



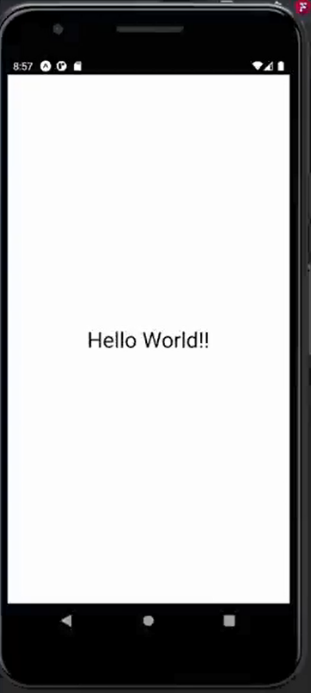
De esta forma, se accede a la app del proyecto con el dispositivo móvil.

**Widgets**

En el desarrollo para dispositivos móviles, varias etiquetas que normalmente son usadas en desarrollo web son reemplazadas por otras.

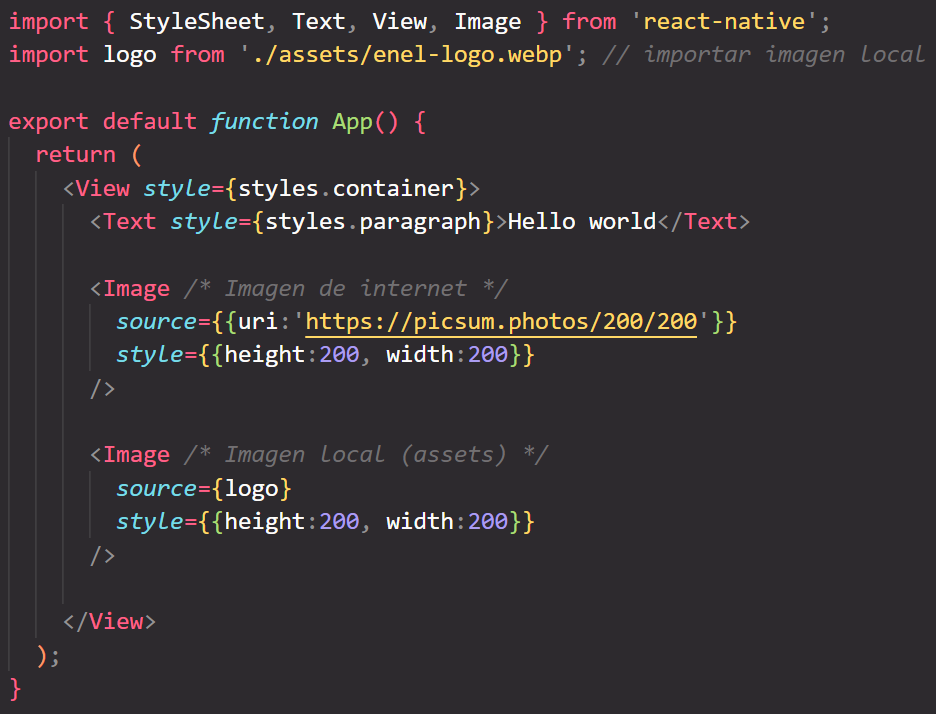
* **<View> es el equivalente a <div>**
* **<Text> es el equivalente a <p>**
* **<Separator> es el equivalente a <hr>**
* **<TextInput> es el equivalente a <input>**
* **<Button> es el equivalente a <button>**
* **<Image> es el equivalente a <img>**
* **<ScrollView> crea un bloque con barra de scroll**
* **<Switch> es el equivalente a un radio button**

Estas etiquetas deben importarse desde **react-native**.

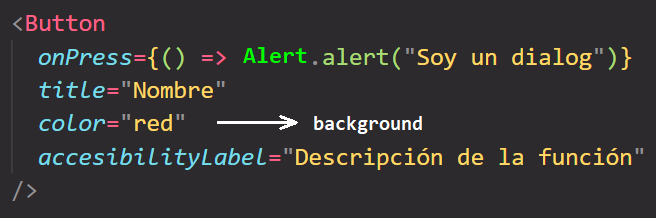
Además, los estilos ya no pueden estar en archivos CSS, sino que pasan a estar escritos en objetos **StyleSheet**.

Imágenes

Permite agregar imágenes locales (almacenadas en **assets**) o de internet.

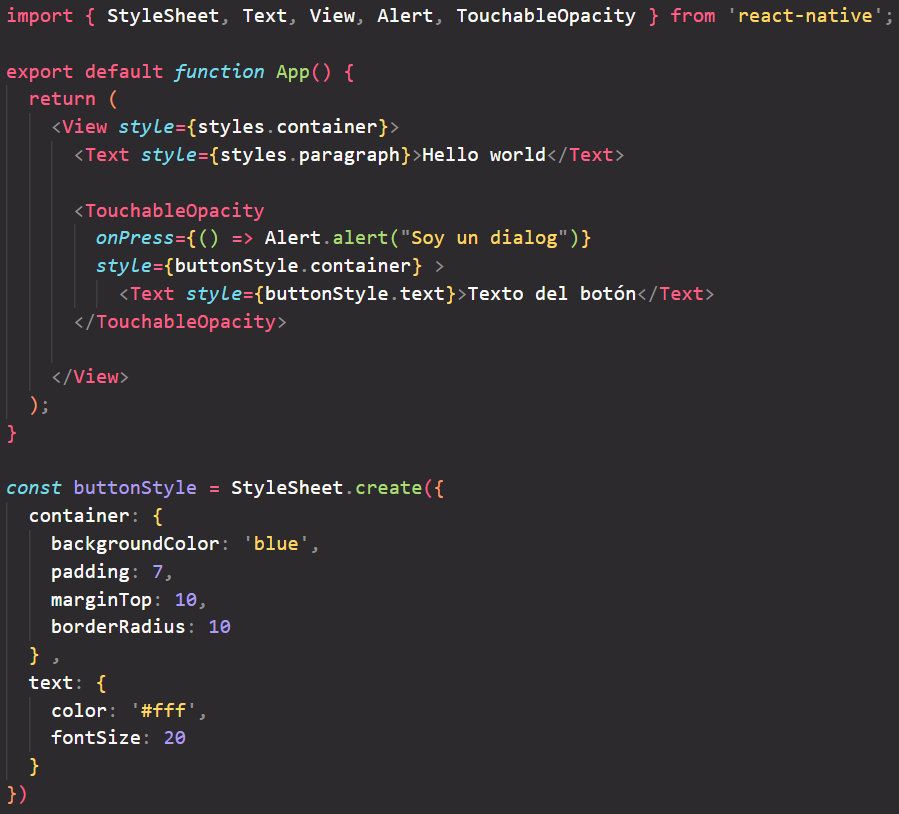


Puede usarse StyleSheet para los estilos.

Botones

Existen diversos tipos de botones.

* <button/> poco personalizable
* <TouchableOpacity> se comporta como un View / Div que ejecuta una función al ser clickeado y posee capacidad de recibir estilos.



**Explorar galería**

React-Native no posee por defecto la posibilidad de explorar la galería de un dispositivo móvil.

Es necesario instalar una librería de terceros para hacerlo.

Desde Expo, se puede instalar por consola:

VER 0:57

**Share**

**Splash Screen & Icon**

1:31

Los íconos deben estar en formato png.

**SharedPreferences**

Permite el almacenado de pequeña información (como configuraciones) en la memoria del dispositivo para que pueda ser utilizada al iniciar la app.

**npm install @react-native-async-storage/async-storage**

Luego, en código ya se podrá utilizar.

import AsyncStorage from '@react-native-async-storage/async-storage';

export default *class* SharedPreferences {

*static* create = (*key*, *value*) => new Promise(*async*(*response*, *reject*) => {

        try { await AsyncStorage.setItem(key, value); response(true); }

        catch(err) { reject(err); }

    });

*static* get = (*key*, *defaultValue*=undefined) => new Promise(*async*(*response*, *reject*) => {

        try {

*const* value = await AsyncStorage.getItem(key);

            if (value !== null) response(value);

            else response(defaultValue ? defaultValue : null);

        } catch(err) { reject(err) }

    });

*static* remove = (*key*) => new Promise(*async*(*response*, *reject*) => {

        try { await AsyncStorage.removeItem(key); response(true); }

        catch(err) { reject(false); }

    });

}

**Deploy**

Compilación el código para ser usado tanto en Web, como en Android y IOs.

Con Expo, se puede hacer con el siguiente comando en consola:

Será necesario tener una cuenta en la página de Expo para esta función.