# 4 Diferentes formas de definir un componente

<https://platzi.com/clases/5481-react-typescript/57807-diferentes-formas-de-definir-un-componente/>

**Diferentes formas de definir un componente**

La forma en la que se trabajan los componentes en TypeScript con React es con la extensión .tsx. (archivos de TypeScript que incluyen JSX).

Hay distintas formas de definir un componente en React con TypeScript, veamos las 4 formas más comunes de hacer esto:

\*\*// Forma implícito (JSX:Element)\*\*

**const** MyComponent = () => {

**return** (

<div>

<h1>Hola!</h1>

</div>

)

}

\*\*// Forma explícita "JSX.Element"\*\*

**const** MyComponent = (): JSX.Element => {

**return** (

<div>

<h1>Hola!</h1>

</div>

)

}

\*\*// Usando tipos de React "FunctionComponent"\*\*

**import** **type** { FunctionComponente } **from** "react"

\*\*\*\*

**const** MyComponent: FunctionComponent = () => {

**return** (

<div>

<h1>Hola!</h1>

</div>

)

}

\*\*// Usando tipos de React "FC"\*\*

**import** **type** { FC } **from** "react"

\*\*\*\*

**const** MyComponent: FC = () => {

**return** (

<div>

<h1>Hola!</h1>

</div>

)

}

Hoy en día, el estándar que se manera comúnmente es la segunda manera Forma explícita "JSX.Element", este es el formato de que debemos utilizar.

Todo lo que escribamos de javascript en react es asi mismo Typescript ya que este es un superset de javascript, quiere decir que ts esta por encima de donde esta js, ya que por debajo hace inferencias

Las ultimas dos con import type ya no se utilizan, solo se recomienda JSX.element

# 5 El objeto props y children

## 🆚 Interface vs Type

👀 Ojito que también podemos definir con una interface el tipado del objeto props

**interface** Props {

image: string;

}

**export** **const** RandomFox = ({ image }: Props): JSX.Element => {

**return** <img className="rounded" width={320} height="auto" src={image} />;

};

En caso de que quieras saber más acerca del uso debido de cada forma te dejo este artículo que me gusto mucho acerca de cuando usar interfaces y cuando definir tipos de objetos en TypeScript

👉 [TypeScript React props: interfaces vs type aliases](https://www.benmvp.com/blog/typescript-react-props-interfaces-type-aliases/)

🤔 Como opinión personal me gusta usar más interfaces, pero me gustaría saber cuál es tu opinión?

## Mis Apuntes

Para poder usar las props con TSX primero tenemos que tipar el objeto de props. Lo hacemos asignándole un tipo por cada prop que tengamos, y para hacerlo mas mantenible, separamos esos tipos en un objeto aparte (ya sea con type o interface).

// RandomFox.tsx

**type** Props = {

image: string;

alt: string;

// another prop

};

**export** **const** RandomFox = ({ image, alt }: Props): JSX.Element => {

// Lo que hicimos fue desestructurar el objeto props para

// quedarnos solo con lo que necesitamos (image y alt)

**return** (

<img width={320} height='auto' src={image} alt={alt} className='rounded' />

);

};

**Nota:** En dado caso de que queramos añadir una nueva prop, hay que definirle su tipo o sino TS nos da un error

// index.tsx

**import** { NextPage } **from** 'next';

**import** Head **from** 'next/head'

**import** { RandomFox } **from** '@/components/RandomFox';

**const** random = (): number => Math.floor(Math.random() \* 123) + 1;

**const** Home: NextPage = () => {

**return** (

<>

<Head>

<title>React con TypeScript</title>

<meta name="description" content="Generated by me" />

<meta name="viewport" content="width=device-width, initial-scale=1" />

<link rel="icon" href="/favicon.ico" />

</Head>

<main>

<RandomFox alt='zorrito' image={`https://randomfox.ca/images/${random()}.jpg`} />

{/\* Si tenemos que mandar otra prop hay que definirla en nuestro tipo Props \*/}

</main>

<footer>

El mejor footer de la historia

</footer>

</>

)

}

**export** **default** Home;

# 7 State con otros tipos

<https://platzi.com/clases/5481-react-typescript/57810-state-con-otros-tipos/>

**STATE CON OTROS TIPOS**  
Generalmente como respuesta de un api no es un Array, si no mas bien son objetos JSON.  
.  
Vamos a simular que recibimos un objeto con un id y un url con la imagen zorros. Para generar una id única vamos a utilizar el siguiente código.

// generate simple unique id

const generateId = **()**: string => {

return (

Math.random**()**.**to**String(36).substring(2, 15) +

Math.random**()**.**to**String(36).substring(2, 15)

);

};

Ahora debemos cambiar el generator de useState de tipo Array al tipo ImageItems( un Array de objetos con un id: string y ulr:string)

type ImageItems =Array<{id:string, url:string}>

const Home: NextPage = () => {

**const** [images, setImages] = useState<ImageItems>([

{id:generateId(), url:`https://randomfox.ca/images/${randomNumber()}.jpg` },

{id:generateId(), url:`https://randomfox.ca/images/${randomNumber()}.jpg` },

{id:generateId(), url:`https://randomfox.ca/images/${randomNumber()}.jpg` },

{id:generateId(), url:`https://randomfox.ca/images/${randomNumber()}.jpg` },

]);

**return** (

También podríamos generar el tipo ImageItem (sin el plural), que seria la mínima representación del objeto que queremos tipar. un objeto singular con id: string y url: string

type ImageItem = {id:string, url:string}

const Home: NextPage = () => {

**const** [images, setImages] = useState<Array<ImageItem>>([

{id:generateId(), url:`https://randomfox.ca/images/${randomNumber()}.jpg` },

{id:generateId(), url:`https://randomfox.ca/images/${randomNumber()}.jpg` },

{id:generateId(), url:`https://randomfox.ca/images/${randomNumber()}.jpg` },

{id:generateId(), url:`https://randomfox.ca/images/${randomNumber()}.jpg` },

]);

**return** (

Hacerlo de esta manera nos da la ventaja al momento de utilizar el metodo map. TypeScript sabrá que la unidad es de tipo ImageItem.  
.  
OJO: Recuerda que ahora nuestro estado este compuesto de 2 valores. Entonces debes declarar el valor de image.url y el image. id para los pasar las props al componente RamdomFox o desestructurar los valores para obtener la url y el id

<main >

{images.map(({id, url}) => (

<div key={id} className="p-4">

<RandomFox image={url} alt={`fox`}/>

</div>

))}

</main>

# 8 Tipos para eventos y callbacks de escuchadores

<https://platzi.com/clases/5481-react-typescript/57811-tipos-para-eventos-y-callbacks-de-escuchadores/>

#### Forma no profesional.

Usar any y un tipo void.

**const** evento = (event: { preventDefault: () => void, target: any } ) => {

event.preventDefault();

**const** target = event.target;

}

#### Forma profesional.

Para añadir tipado de forma mas profesional es recomendable **leer la definición** de la creación de tipos del fabricante de la tecnología, en este caso React. ¿Cómo hacer eso? Pues bien, se pone el mouse encima (hover) sobre el evento, a continuación aprietas la tecla ctrl y clic o tambien dando clic derecho y luego en ir a deficinión; con base en ello se puede leer la documentación al respecto para aprender a tipar.  
.  
Un **ejemplo** de un botón onClick para generar una acción sería la siguiente donde el tipado va asignado a la constante y entre el simbolo **(<>)** va el elemento HTML al que se le escucha el evento.

const evento: MouseEventHandler<HTMLButtonElement> = () => {

// code

}

## ****Tipos para eventos y callbacks de escuchadores****

A medida que desarrollamos aplicaciones en TypeScript, una de las tareas más importantes es asegurarnos de que estamos trabajando con tipos de datos precisos y seguros. Esto es particularmente importante cuando trabajamos con eventos y callbacks de escuchadores en React, ya que necesitamos especificar el tipo de evento que se está produciendo.

Afortunadamente, muchas librerías y frameworks modernos ya proporcionan tipos de datos precisos y completos para eventos. Esto significa que podemos confiar en que las funciones que estamos utilizando son seguras y precisas.

Para asegurarnos de que estamos utilizando los tipos de datos correctos para eventos, necesitamos asegurarnos de que nuestra configuración de TypeScript incluya las librerías que estamos utilizando. Podemos hacer esto especificando las librerías que estamos usando en nuestro archivo **tsconfig.json**.

{

"compilerOptions": {

...

\*\*"lib": ["DOM", "DOM.Iterable", "ESNext"]\*\*, // especificamos las librerías que usamos

...

},

...

}

Una vez que hemos configurado nuestro proyecto correctamente, podemos inspeccionar los tipos de eventos utilizando el editor de código que estemos utilizando. Por ejemplo, si queremos saber el tipo de datos que se produce en un evento **onClick** de un botón, podemos simplemente hacer hover sobre el método **onClick** y buscar su definición.

El editor de código nos mostrará la definición del método **onClick**, junto con información sobre su tipo de datos:

En este caso, el tipo de datos del evento es **React.MouseEvent<HTMLButtonElement>**. Esto significa que podemos utilizar este tipo de datos para especificar el tipo de datos que se produce en nuestro callback de escuchador de eventos.

Por ejemplo, podemos definir una función **addNewFox** que se llama cuando se hace clic en un botón. Para especificar el tipo de datos del evento que se produce, podemos incluir **React.MouseEvent<HTMLButtonElement>** como argumento de nuestra función:

**function** **App**(): **JSX**.**Element** {

**const** addNewFox = (\*\*event: React.MouseEvent<HTMLButtonElement>\*\*): void => {

event.preventDefault()

**const** target = event.target

// ...

}

**return** (

// JSX del componente App...

)

}

En este ejemplo, la función **addNewFox** toma un argumento de tipo **React.MouseEvent<HTMLButtonElement>**. Esto nos permite acceder a las propiedades del evento de manera segura y precisa, sin tener que utilizar tipos de datos genéricos como **any**.

También, podemos hacer la importación de los tipos de manera directa:

// importación del tipo desde React

**import** **type** { MouseEvent } **from** "react"

**function** **App**(): **JSX**.**Element** {

// Evitamos escribir de donde viene "React"

**const** addNewFox = (event: MouseEvent<HTMLButtonElement>): void => {

event.preventDefault()

**const** target = event.target

// ...

}

**return** (

// JSX del componente App...

)

}

Esto dependerá bastante de la librería que estemos usando junto con React.

cheatsheet de tipos con ts <https://react-typescript-cheatsheet.netlify.app/docs/basic/getting-started/class_components>

# 11 Componentes que extienden elementos DOM

Yo utilice interface y herede de ImgHTMLAttributes<HTMLImageElement>

**interface** Props **extends** ImgHTMLAttributes<HTMLImageElement> {

}

con esto ya funcionaría pero tendriamos que validar que src tenga algun valor cada vez que queramos utilizarlo utlizarlo por que en la imagenes src: string | undefined, pero esto lo solucione definiendo src solamente como string

**interface** Props **extends** ImgHTMLAttributes<HTMLImageElement> {

src: string

}

## ****Componentes que extienden elementos DOM****

Nuestros componentes evolucionan, por lo que es importante crear componentes escalables que puedan tener funcionalidades más complejas en el futuro… Vamos a ver como podemos hacer que nuestro componente que hemos trabajado sea un componente más global, escalable, y reutilizable.

De **RandomFox.tsx** a **LazyImage.tsx**

**import** { useRef, useEffect, useState } **from** "react"

**type** Props = { image: string }

**export** **default** **function** **LazyImage**( { image }: Props ):**JSX**.**Element** {

**const** node = useRef<HTMLImageElement> (**null**)

**const** [src, setSrc] = useState<string>("data:image/svg+xml;base64,PHN2ZyB3aWR0aD0iMzIwIiBoZWlnaHQ9IjMyMCIgeG1sbnM9Imh0dHA6Ly93d3cudzMub3JnLzIwMDAvc3ZnIiB2ZXJzaW9uPSIxLjEiLz4=")

useEffect(() => {

**const** observer = **new** IntersectionObserver(entries => {

entries.forEach(entry => {

**if** (entry.isIntersecting) {

setSrc(image)

}

})

})

**if** (node.current) {

observer.observe(node.current)

}

**return** () => {

observer.disconnect()

}

},[image])

**return** (

<img ref={node} src={src} className="w-80 h-auto rounded-lg bg-gray-300" />

)

}

Listo, ahora supongamos que una de nuestras compañeras desarrolladoras necesita añadir el evento onCLick={} en nuestro componente, veamos como hacer esto:  
**App.tsx**

**function** **App**():**JSX**.**Element** {

...

**return** (

<main className="App">

...

{

foxes.map((fox, index) => (

<div key={index} className="p-4">

{/\* Esto nos debería dar un error \*/}

<LazyImage image={fox} \*\*onClick={() => console.log('Hola')}\*\* />

</div>

))

}

</main>

)

}

**LazyImage.tsx**

// actualizamos los tipos añadiendo el onClick

**type** Props = { image: string, onClick: () => void }

**export** **default** **function** **LazyImage**( { image, onClick }: Props ):**JSX**.**Element** {

...

**return** (

<img ref={node} src={src} className="w-80 h-auto rounded-lg bg-gray-300" \*\*onClick={onClick}\*\* />

)

}

Si probamos el componente, veremos que funciona, pero, ¿acaso esto escala? Recordemos que como desarrolladores tenemos que hacer componentes globales, escalables, y reutilizables, por ello, podemos cambiar esto para en caso de que alguna de nuestras compañeras quiera añadir funcionalidad extra, esta sea sencilla de implementar. Veamos como:

**LazyImage.tsx**

/\* Importamos el tipo de dato que nos aparece cuando hacemos hover cualquiera

de los eventos de image (como aprendimos antes) \*/

**import** **type** { ImgHTMLAttributes } **from** "react"

// Creamos dos nuevos tipos de datos

**type** LazyImageProps = { image: string }

**type** ImageNativeTypes = ImgHTMLAttributes<HTMLImageElement>

// Los sumamos, haciendo que estos sean un solo tipo de dato

**type** Props = LazyImageProps & ImageNativeTypes

// recibimos todas las propiedades que soportamos de img con el spread operator

**export** **default** **function** **LazyImage**( { image, ...imgProps}: \*\*Props\*\* ):**JSX**.**Element** {

**return** (

{/\* todas las propiedades que recibe un img soportan automática e implícitamente \*/}

<img ... {...imgProps} />

)

}

Ya de esta forma estamos recibiendo todos los parámetros que puede recibir un <img /> en React (title, onClick, className, etc). Y como queremos que nuestro componente sea reutilizable, los estilos y personalización del componente deberían ir en el padre. Vamos a hacer más cambios importantes:

**LazyImage.tsx**

**import** { useRef, useEffect, useState } **from** "react"

**import** **type** { ImgHTMLAttributes } **from** "react"

// actualizamos image para que ahora sea src para no causar confusión

\*\***type** LazyImageProps = { src: string }\*\*

**type** ImageNativeTypes = ImgHTMLAttributes<HTMLImageElement>

**type** Props = LazyImageProps & ImageNativeTypes

**export** **default** **function** **LazyImage**( { \*\*src\*\*, ...imgProps }: Props ):**JSX**.**Element** {

**const** node = useRef<HTMLImageElement> (**null**)

// a src lo cambiaremos a currentSrc paraevitar sobreescritura

**const** \*\*[currentSrc, setCurrentSrc]\*\* = useState<string>("data:image/svg+xml;base64,PHN2ZyB3aWR0aD0iMzIwIiBoZWlnaHQ9IjMyMCIgeG1sbnM9Imh0dHA6Ly93d3cudzMub3JnLzIwMDAvc3ZnIiB2ZXJzaW9uPSIxLjEiLz4=")

// corregimos los datos a recibir

useEffect(() => {

**const** observer = **new** IntersectionObserver(entries => {

entries.forEach(entry => {

**if** (entry.isIntersecting) {

\*\*setCurrentSrc(src)\*\*

}

})

})

**if** (node.current) {

observer.observe(node.current)

}

**return** () => {

observer.disconnect()

}

},[\*\*src\*\*])

**return** (

\*\*<img ref={node} src={currentSrc} {...imgProps} />\*\*

)

}

**App.tsx**

**import** { useState } **from** "react"

**import** LazyImage **from** "./components/LazyImage"

**import** **type** { MouseEvent } **from** "react"

**const** random = (num: number):number => {

**return** Math.floor(Math.random() \* num) + 1

}

**function** **App**():**JSX**.**Element** {

**const** [foxes, setFoxes] = useState<string[]> ([])

**const** addNewFox = (event: MouseEvent<HTMLButtonElement>):void => {

event.preventDefault()

**const** newImage: string = `https://randomfox.ca/images/${random(123)}.jpg`

setFoxes([

...foxes,

newImage

])

}

**return** (

<main className="App">

<h1 className="text-3xl font-bold text-indigo-600 p-6">Random Fox</h1>

<button onClick={addNewFox} className="">Añadir zorro</button>

{

foxes.map((fox, index) => (

<div key={index} className="p-4">

{/\* Utilizamos el atributo src y añadimos la personalización en el componente padre \*/}

<LazyImage \*\*scr={fox}\*\* className="w-80 h-auto rounded-lg bg-gray-300" onClick={() => console.log('Hola')} />

</div>

))

}

</main>

)

}

**export** **default** App

Hemos logrado crear un componente global, escalable, y reutilizable. El cual utiliza las convenciones más convenientes para que nuestro equipo de trabajo tenga una buena experiencia de desarrollo al momento de utilizar nuestro código, haciéndolo fácil de entender y fácil de mantener a lo largo del tiempo.

# 12 Reto: sigamos extendiendo el DOM

<https://platzi.com/clases/5481-react-typescript/57815-reto-sigamos-extiendo-el-dom/>

Estuvo genial el reto, aprendí mucho de useEffect leyendo la [documentación oficial](https://beta.reactjs.org/learn/removing-effect-dependencies) de cómo remover las dependencias de useEffect buscando una solución al bug del segundo reto.  
.  
En resumen, aprendí que el linter siempre estará pendiente de que agreguemos en el array de dependencias de useEffect todos los valores reactivos que usemos dentro del mismo, como props o estados. Como onLazyLoad es una función dentro de los props del componente LazyImage, React lo trata como un valor reactivo. Sin embargo, no queremos agregar esta función en las dependencias de useEffect, porque lo único que nos interesa de onLazyLoad es obtener sus datos, más no ejecutar el callback de useEffect cada vez que onLazyLoad cambie.  
.  
La mejor solución en la documentación es utilizar el hook (en fase experimental a la fecha de escribir este comentario 15/03/23) useEffectEvent. Este hook le indica a useEffect que vamos a utilizar un valor reactivo dentro del mismo, pero no queremos que ‘reaccione’ a sus cambios, sino que solamente lo use para lectura. Este hook al estar en fase experimental, preferí no usarlo pero está bastante interesante para cuando salga.  
.  
Al final, opté por la opción que React no recomienda, que es desactivar el linter. Como en todo, cada cosa a su momento, y en este caso desactivar el linter es la opción que mejor se adapta a mi código porque solamente quiero leer lo que me arroja onLazyLoad, más no reaccionar a sus cambios. Es de esos momentos cuando la solución menos recomendada resulta ser la más óptima para un caso específica, o por lo menos la más sencilla según yo

**interface** LazyImageProps **extends** ImgHTMLAttributes<HTMLImageElement> {

src: string

alt: string,

// especificamos que onLazyLoad es un valor opcional de tipo void que aceptará un argumento de tipo HTMLImageElement

onLazyLoad?: (node: HTMLImageElement) => void

}

**export** **default** **function** **LazyImage**({src, alt, onLazyLoad, ...imgOptionalAttrs}: LazyImageProps) {

**const** node = useRef<HTMLImageElement>(**null**)

**const** [currentSrc, setCurrentSrc] = useState(defaultImg)

useEffect(() => {

**const** intersectionObserver = **new** IntersectionObserver(entries => {

entries.forEach(entry => {

**if** (entry.isIntersecting) {

setCurrentSrc(src)

// ya que onLazyLoad es un valor opcional, validamos que sea verdadero y lo ejecutamos pasando el target de la entrada como argumento el cual contiene todos los datos del elemento img

onLazyLoad && onLazyLoad(entry.target **as** HTMLImageElement)

// le digo al intersectionObserver que deje de observar a img luego de entrar al viewport y ejecutar el código de arriba

intersectionObserver.unobserve(entry.target)

}

})

})

node.current && intersectionObserver.observe(node.current)

**return** () => { intersectionObserver.disconnect() }

// DANGER ZONE: le digo al linter que ignore la línea de abajo

// eslint-disable-next-line react-hooks/exhaustive-deps

}, [src])

**return** (

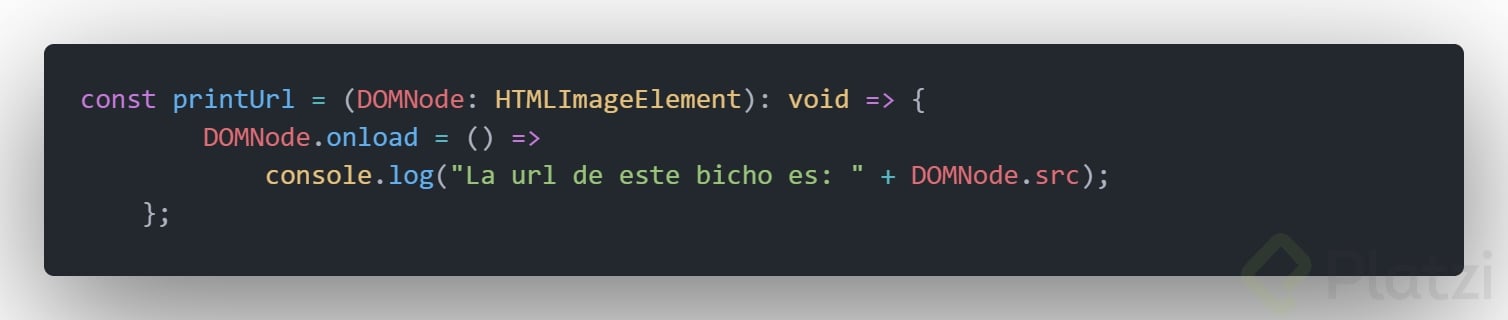
<img ref={node} src={currentSrc} alt={alt} {...imgOptionalAttrs} />

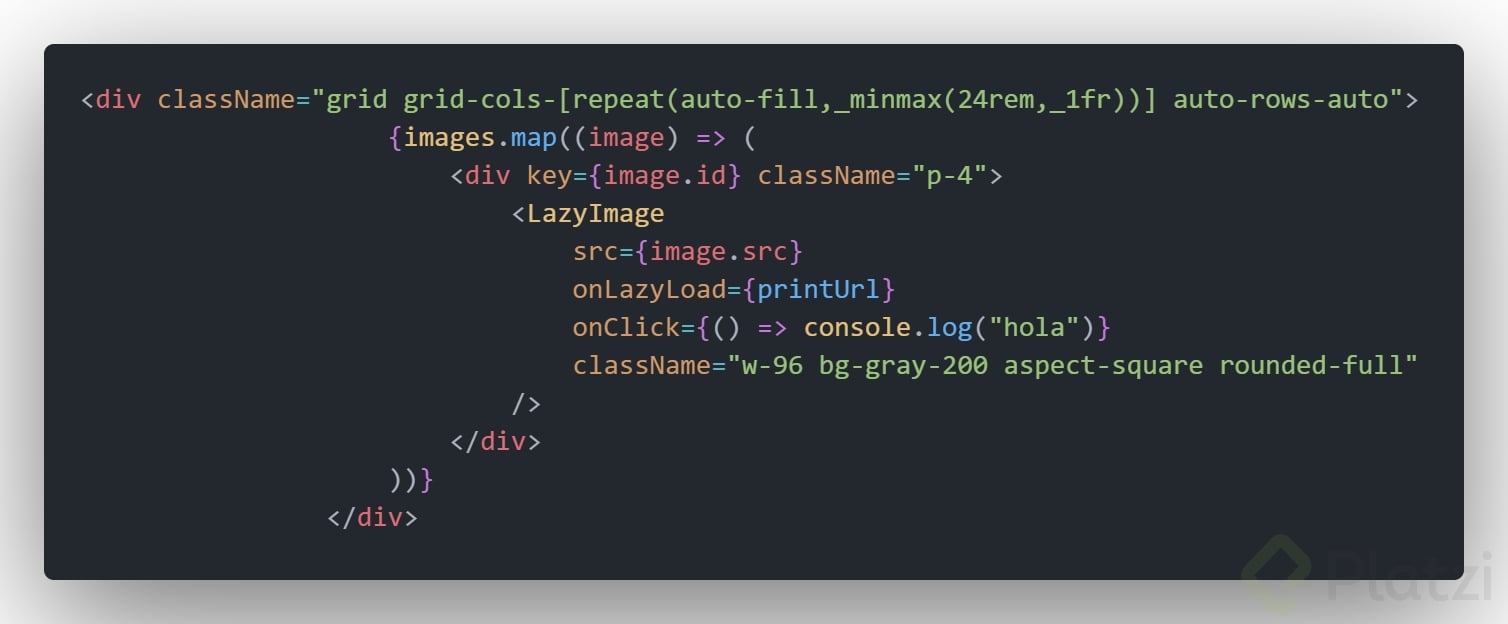
)

}

## Siento que mi solución fue “inocente”, pero aquí va:

Paso 1:  
…  
Le añadí el evento onload al nodo DOM enviado por parámetro.  
😁. Ese es el único paso.  
El código está medio ilegible… pero si a alguien le interesa leer:





# 13. Creando tipos propios para la aplicación

https://platzi.com/clases/5481-react-typescript/57816-creando-tipos-propios-para-la-aplicacion/

Tipos Globals:

* Idealmente corresponden a entidades del contexto de la aplicación, ex: Usuario, Producto
* No Abusar de ellos. Podrían crecer sin control
* Empieza con tipos locales y expórtalos

#### Creando tipos propios para la aplicación

Es bueno pero también malo tener tipos globables. Analiza bien tu necesidad de negocio para saber qué sí debería ser global. **No abuses de los tipos globales**.

# 14Trabajando con librerías no-tipadas

### Resumen

Instala esta librería creada y mantenida por la comunidad si quieres tener unos tipos super completos de lodash, y así te ahorrar lo que hicimos en esta clase.

npm install @types/lodash

Sin embargo si queremos utilizar otra librería y queremos los tipos, podemos probar suerte escribiendo este comando que dejo a continuación

npm install @types/nombe **de** **la** librería **que** quieres usar

Tal y como lo dice el ejemplo de arriba, podemos poner el nombre de la librería y rezar a diosito para que haya un repositorio de la comunidad con dichos tipos y así no tenerlos que hacer nosotros. Si no es así el caso, tenemos de la opción de hacerlo nosotros mismo.

Responder

**Juan Gómez**

[Hace 3 meses](https://platzi.com/comentario/4633400/)

2

Solo seria agregar el flag -D, para que sea incluida en el build final

Tener en cuenta que existen repositorios para este tipos de librerías que no son tipados, y han creado su forma de instalación de forma tipeada, tal como lo es

**Npm install @types/lodash**